

Typescript 中文手册

2017.06

Table of Contents

Typescript 中文手册.....	1
基础类型.....	1
介绍.....	1
布尔值.....	1
数字.....	1
字符串.....	1
数组.....	1
元组 Tuple.....	1
枚举.....	2
Any.....	3
Void.....	3
Null 和 Undefined.....	4
Never.....	4
类型断言.....	5
关于 let.....	6
变量声明.....	6
变量声明.....	6
var 声明.....	6
let 声明.....	10
const 声明.....	15
let vs. const.....	16
解构.....	16
接口.....	21
介绍.....	21
接口初探.....	21
可选属性.....	22
只读属性.....	23
额外的属性检查.....	24
函数类型.....	26
可索引的类型.....	27
类类型.....	28
继承接口.....	31
混合类型.....	32
接口继承类.....	32
类.....	34
介绍.....	34
类.....	34
继承.....	35

公共，私有与受保护的修饰符.....	36
readonly 修饰符.....	39
存取器.....	40
静态属性.....	42
抽象类.....	42
高级技巧.....	44
函数.....	46
介绍.....	46
函数.....	46
函数类型.....	47
可选参数和默认参数.....	49
剩余参数.....	51
this.....	51
重载.....	55
泛型.....	57
介绍.....	57
泛型之 Hello World.....	58
使用泛型变量.....	59
泛型类型.....	60
泛型类.....	62
泛型约束.....	63
枚举.....	65
枚举.....	65
外部枚举.....	67
类型推论.....	68
介绍.....	68
基础.....	68
最佳通用类型.....	68
上下文类型.....	69
类型兼容性.....	70
介绍.....	70
开始.....	71
比较两个函数.....	72
枚举.....	74
类.....	74
泛型.....	75
高级主题.....	76
高级类型.....	77
交叉类型 (Intersection Types)	77

联合类型 (Union Types)	78
类型保护与区分类型 (Type Guards and Differentiating Types)	80
可以为 null 的类型.....	84
类型别名.....	86
字符串字面量类型.....	88
可辨识联合 (Discriminated Unions)	89
多态的 this 类型.....	91
索引类型 (Index types)	93
映射类型.....	94
Symbols.....	97
介绍.....	97
众所周知的 Symbols.....	98
迭代器和生成器.....	100
可迭代性.....	100
模块.....	102
介绍.....	102
导出.....	103
导入.....	104
默认导出.....	105
export = 和 import = require().....	107
生成模块代码.....	108
简单示例.....	109
可选的模块加载和其它高级加载场景.....	111
使用其它的 JavaScript 库.....	112
创建模块结构指导.....	115
命名空间.....	121
介绍.....	121
第一步.....	121
命名空间.....	122
分离到多文件.....	123
别名.....	126
使用其它的 JavaScript 库.....	126
命名空间和模块.....	127
介绍.....	127
使用命名空间.....	128
使用模块.....	128
命名空间和模块的陷阱.....	128
模块解析.....	130
声明合并.....	142

介绍.....	142
基础概念.....	143
合并接口.....	143
合并命名空间.....	145
命名空间与类和函数和枚举类型合并.....	147
非法的合并.....	148
模块扩展.....	148
JSX.....	150
介绍.....	150
基本用法.....	150
as 操作符.....	151
类型检查.....	151
JSX 结果类型.....	156
嵌入的表达式.....	156
React 整合.....	157
装饰器.....	157
介绍.....	157
装饰器.....	158
Mixins.....	169
介绍.....	169
混入示例.....	170
理解这个例子.....	171
三斜线指令.....	173

基础类型

介绍

为了让程序有价值，我们需要能够处理最简单的数据单元：数字，字符串，结构体，布尔值等。TypeScript 支持与 JavaScript 几乎相同的数据类型，此外还提供了实用的枚举类型方便我们使用。

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**（其它语言中也一样）。

```
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.";
```

数组

TypeScript 像 JavaScript 一样可以操作数组元素。有两种方式可以定义数组。第一种，可以在元素类型后面接上 `[]`，表示由此类型元素组成的一个数组：

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

第二种方式是使用数组泛型，`Array<元素类型>`：

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

元组 Tuple

元组类型允许表示一个已知元素数量和类型的数组，各元素的类型不必相同。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
```

联合类型是高级主题，我们会在以后的章节里讨论它。

枚举

enum 类型是对 JavaScript 标准数据类型的一个补充。像 C# 等其它语言一样，使用枚举类型可以为一组数值赋予友好的名字。

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

默认情况下，从 **0** 开始为元素编号。你也可以手动的指定成员的数值。例如，我们将上面的例子改成从 **1** 开始编号：

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
```

或者，全部都采用手动赋值：

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

枚举类型提供的一个便利是你可以由枚举的值得到它的名字。例如，我们知道数值为 2，但是不确定它映射到 **Color** 里的哪个名字，我们可以查找相应的名字：

```
enum Color {Red = 1, Green, Blue}
```



```
let colorName: string = Color[2];

alert(colorName);
```

Any

有时候，我们会想要为那些在编程阶段还不清楚类型的变量指定一个类型。这些值可能来自于动态的内容，比如来自用户输入或第三方代码库。这种情况下，我们不希望类型检查器对这些值进行检查而是直接让它们通过编译阶段的检查。那么我们可以使用 **any** 类型来标记这些变量：

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

在对现有代码进行改写的时候，**any** 类型是十分有用的，它允许你在编译时可选择地包含或移除类型检查。你可能认为 **Object** 有相似的作用，就像它在其它语言中那样。但是 **Object** 类型的变量只是允许你给它赋任意值 - 但是却不能够在它上面调用任意的方法，即便它真的有这些方法：

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.
```

当你只知道一部分数据的类型时，**any** 类型也是有用的。比如，你有一个数组，它包含了不同的类型的数据：

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

Void

某种程度上来说，**void** 类型像是与 **any** 类型相反，它表示没有任何类型。当一个函数没有返回值时，你通常会见到其返回值类型是 **void**：

```
function warnUser(): void {
```

```
    alert("This is my warning message");  
}
```

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

```
let unusable: void = undefined;
```

Null 和 Undefined

TypeScript 里，**undefined** 和 **null** 两者各自有自己的类型分别叫做 **undefined** 和 **null**。和 **void** 相似，它们的本身的类型用处不是很大：

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

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

然而，当你指定了 **--strictNullChecks** 标记，**null** 和 **undefined** 只能赋值给 **void** 和它们各自。这能避免很多常见的问题。也许在某处你想传入一个 **string** 或 **null** 或 **undefined**，你可以使用联合类型 **string | null | undefined**。再次说明，稍后我们会介绍联合类型。

注意：我们鼓励尽可能地使用 **--strictNullChecks**，但在本手册里我们假设这个标记是关闭的。

Never

never 类型表示的是那些永不存在的值的类型。例如，**never** 类型是那些总是会抛出异常或根本就不会有返回值的函数表达式或箭头函数表达式的返回值类型；变量也可能是 **never** 类型，当它们被永不真的类型保护所约束时。

never 类型是任何类型的子类型，也可以赋值给任何类型；然而，没有类型是 **never** 的子类型或可以赋值给 **never** 类型（除了 **never** 本身之外）。即使 **any** 也不可以赋值给 **never**。

下面是一些返回 `never` 类型的函数：

```
// 返回 never 的函数必须存在无法达到的终点
function error(message: string): never {
    throw new Error(message);
}

// 推断的返回值类型为 never
function fail() {
    return error("Something failed");
}

// 返回 never 的函数必须存在无法达到的终点
function infiniteLoop(): never {
    while (true) {
    }
}
```

类型断言

有时候你会遇到这样的情况，你会比 TypeScript 更了解某个值的详细信息。通常这会发生在你清楚地知道一个实体具有比它现有类型更确切的类型。

通过 *类型断言* 这种方式可以告诉编译器，“相信我，我知道自己在干什么”。类型断言好比其它语言里的类型转换，但是不进行特殊的数据检查和解构。它没有运行时的影响，只是在编译阶段起作用。TypeScript 会假设你，程序员，已经进行了必须的检查。

类型断言有两种形式。其一是“尖括号”语法：

```
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
```

另一个为 `as` 语法：

```
let someValue: any = "this is a string";
```

```
let strLength: number = (someValue as string).length;
```

两种形式是等价的。至于使用哪个大多数情况下是凭个人喜好；然而，当你在 TypeScript 里使用 JSX 时，只有 `as` 语法断言是被允许的。

关于 `let`

你可能已经注意到了，我们使用 `let` 关键字来代替大家所熟悉的 JavaScript 关键字 `var`。`let` 关键字是 JavaScript 的一个新概念，TypeScript 实现了它。我们会在以后详细介绍它，很多常见的问题都可以通过使用 `let` 来解决，所以尽可能地使用 `let` 来代替 `var` 吧。

变量声明

变量声明

`let` 和 `const` 是 JavaScript 里相对较新的变量声明方式。像我们之前提到过的，`let` 在很多方面与 `var` 是相似的，但是可以帮助大家避免在 JavaScript 里常见一些问题。`const` 是对 `let` 的一个增强，它能阻止对一个变量再次赋值。

因为 TypeScript 是 JavaScript 的超集，所以它本身就支持 `let` 和 `const`。下面我们会详细说明这些新的声明方式以及为什么推荐使用它们来代替 `var`。

如果你之前使用 JavaScript 时没有特别在意，那么这节内容会唤起你的回忆。如果你已经对 `var` 声明的怪异之处了如指掌，那么你可以轻松地略过这节。

`var` 声明

一直以来我们都是通过 `var` 关键字定义 JavaScript 变量。

```
var a = 10;
```

大家都能理解，这里定义了一个名为 **a** 值为 **10** 的变量。

我们也可以在函数内部定义变量：

```
function f() {  
    var message = "Hello, world!";  
  
    return message;  
}
```

并且我们也可以在其它函数内部访问相同的变量。

```
function f() {  
    var a = 10;  
    return function g() {  
        var b = a + 1;  
        return b;  
    }  
}  
  
var g = f();  
g(); // returns 11;
```

上面的例子里，**g** 可以获取到 **f** 函数里定义的 **a** 变量。每当 **g** 被调用时，它都可以访问到 **f** 里的 **a** 变量。即使当 **g** 在 **f** 已经执行完后才被调用，它仍然可以访问及修改 **a**。

```
function f() {  
    var a = 1;  
  
    a = 2;  
    var b = g();  
    a = 3;  
  
    return b;  
  
    function g() {  
        return a;  
    }  
}
```

```
f(); // returns 2
```

作用域规则

对于熟悉其它语言的人来说，**var** 声明有些奇怪的作用域规则。看下面的例子：

```
function f(shouldInitialize: boolean) {  
    if (shouldInitialize) {  
        var x = 10;  
    }  
  
    return x;  
}  
  
f(true); // returns '10'  
f(false); // returns 'undefined'
```

有些读者可能要多看几遍这个例子。变量 **x** 是定义在 **if** 语句里面，但是我们却可以在语句的外面访问它。这是因为 **var** 声明可以在包含它的函数，模块，命名空间或全局作用域内部任何位置被访问（我们后面会详细介绍），包含它的代码块对此没有什么影响。有些人称此为 **var** 作用域或函数作用域。函数参数也使用函数作用域。

这些作用域规则可能会引发一些错误。其中之一就是，多次声明同一个变量并不会报错：

```
function sumMatrix(matrix: number[][]) {  
    var sum = 0;  
    for (var i = 0; i < matrix.length; i++) {  
        var currentRow = matrix[i];  
        for (var i = 0; i < currentRow.length; i++) {  
            sum += currentRow[i];  
        }  
    }  
  
    return sum;  
}
```

这里很容易看出一些问题，里层的 **for** 循环会覆盖变量 **i**，因为所有 **i** 都引用相同的函数作用域内的变量。有经验的开发者们很清楚，这些问题可能在代码审查时漏掉，引发无穷的麻烦。

变量获取怪异之处

快速的猜一下下面的代码会返回什么：

```
for (var i = 0; i < 10; i++) {  
    setTimeout(function() { console.log(i); }, 100 * i);  
}
```

介绍一下，**setTimeout** 会在若干毫秒的延时后执行一个函数（等待其它代码执行完毕）。

好吧，看一下结果：

```
10  
10  
10  
10  
10  
10  
10  
10  
10  
10  
10
```

很多 JavaScript 程序员对这种行为已经很熟悉了，但如果你很不解，你并不是一个人。大多数人期望输出结果是这样：

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

还记得我们上面讲的变量获取吗？

每当 `g` 被调用时，它都可以访问到 `f` 里的 `a` 变量。

让我们花点时间考虑在这个上下文里的情况。`setTimeout` 在若干毫秒后执行一个函数，并且是在 `for` 循环结束后。`for` 循环结束后，`i` 的值为 `10`。所以当函数被调用的时候，它会打印出 `10`！

一个通常的解决方法是使用立即执行的函数表达式（IIFE）来捕获每次迭代时 `i` 的值：

```
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);  
}
```

这种奇怪的形式我们已经司空见惯了。参数 `i` 会覆盖 `for` 循环里的 `i`，但是因为我们起了同样的名字，所以我们不用怎么改 `for` 循环体里的代码。

let 声明

现在你已经知道了 `var` 存在一些问题，这恰好说明了为什么用 `let` 语句来声明变量。除了名字不同外，`let` 与 `var` 的写法一致。

```
let hello = "Hello!";
```

主要的区别不在语法上，而是语义，我们接下来会深入研究。

块作用域

当用 `let` 声明一个变量，它使用的是 *词法作用域* 或 *块作用域*。不同于使用 `var` 声明的变量那样可以在包含它们的函数外访问，块作用域变量在包含它们的块或 `for` 循环之外是不能访问的。

```
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;
}

```

这里我们定义了 2 个变量 **a** 和 **b**。**a** 的作用域是 **f** 函数体内，而 **b** 的作用域是 **if** 语句块里。

在 **catch** 语句里声明的变量也具有同样的作用域规则。

```

try {
    throw "oh no!";
}
catch (e) {
    console.log("Oh well.");
}

// Error: 'e' doesn't exist here
console.log(e);

```

拥有块级作用域的变量的另一个特点是，它们不能在声明之前读或写。虽然这些变量始终“存在”于它们的作用域里，但在直到声明它的代码之前的区域都属于**暂时性死区**。它只是用来说明我们不能在 **let** 语句之前访问它们，幸运的是 TypeScript 可以告诉我们这些信息。

```

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

```

注意一点，我们仍然可以在一个拥有块作用域变量被声明前获取它。只是我们不能在变量声明前去调用那个函数。如果生成代码目标为 ES2015，现代的运行时会抛出一个错误；然而，现今 TypeScript 是不会报错的。

```

function foo() {
    // okay to capture 'a'
    return a;
}

```

```
// 不能在'a'被声明前调用'foo'
// 运行时应该抛出错误
foo();

let a;
```

关于暂时性死区的更多信息，查看这里 [Mozilla Developer Network](#).

重定义及屏蔽

我们提过使用 **var** 声明时，它不在乎你声明多少次；你只会得到 1 个。

```
function f(x) {
  var x;
  var x;

  if (true) {
    var x;
  }
}
```

在上面的例子里，所有 **x** 的声明实际上都引用一个相同的 **x**，并且这是完全有效的代码。这经常会成为 bug 的来源。好的是，**let** 声明就不会这么宽松了。

```
let x = 10;
let x = 20; // 错误，不能在 1 个作用域里多次声明`x`
```

并不是要求两个均是块级作用域的声明 TypeScript 才会给出一个错误的警告。

```
function f(x) {
  let x = 100; // error: interferes with parameter declaration
}

function g() {
  let x = 100;
  var x = 100; // error: can't have both declarations of 'x'
}
```

并不是说块级作用域变量不能用函数作用域变量来声明。而是块级作用域变量需要在明显不同的块里声明。

```
function f(condition, x) {  
  if (condition) {  
    let x = 100;  
    return x;  
  }  
  
  return x;  
}  
  
f(false, 0); // returns 0  
f(true, 0); // returns 100
```

在一个嵌套作用域里引入一个新名字的行为称做屏蔽。它是一把双刃剑，它可能会不小心地引入新问题，同时也可能会解决一些错误。例如，假设我们现在用 `let` 重写之前的 `sumMatrix` 函数。

```
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;  
}
```

这个版本的循环能得到正确的结果，因为内层循环的 `i` 可以屏蔽掉外层循环的 `i`。

通常来讲应该避免使用屏蔽，因为我们需要写出清晰的代码。同时也有些场景适合利用它，你需要好好打算一下。

块级作用域变量的获取

在我们最初谈及获取用 **var** 声明的变量时，我们简略地探究了一下在获取到了变量之后它的行为是怎样的。直观地讲，每次进入一个作用域时，它创建了一个变量的环境。就算作用域内代码已经执行完毕，这个环境与其捕获的变量依然存在。

```
function theCityThatAlwaysSleeps() {  
  let getCity;  
  
  if (true) {  
    let city = "Seattle";  
    getCity = function() {  
      return city;  
    }  
  }  
  
  return getCity();  
}
```

因为我们已经在 **city** 的环境里获取到了 **city**，所以就算 **if** 语句执行结束后我们仍然可以访问它。

回想一下前面 **setTimeout** 的例子，我们最后需要使用立即执行的函数表达式来获取每次 **for** 循环迭代里的状态。实际上，我们做的是为获取到的变量创建了一个新的变量环境。这样做挺痛苦的，但是幸运的是，你不必在 TypeScript 里这样做了。

当 **let** 声明出现在循环体里时拥有完全不同的行为。不仅是在循环里引入了一个新的变量环境，而是针对每次迭代都会创建这样一个新作用域。这就是我们在使用立即执行的函数表达式时做的事，所以在 **setTimeout** 例子里我们仅使用 **let** 声明就可以了。

```
for (let i = 0; i < 10 ; i++) {  
  setTimeout(function() {console.log(i); }, 100 * i);  
}
```

会输出与预料一致的结果：

```
0  
1
```

```
2
3
4
5
6
7
8
9
```

const 声明

const 声明是声明变量的另一种方式。

```
const numLivesForCat = 9;
```

它们与 **let** 声明相似，但是就像它的名字所表达的，它们被赋值后不能再改变。换句话说，它们拥有与 **let** 相同的作用域规则，但是不能对它们重新赋值。

这很好理解，它们引用的值是 *不可变的*。

```
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--;
```

除非你使用特殊的方法去避免，实际上 `const` 变量的内部状态是可修改的。幸运的是，TypeScript 允许你将对象的成员设置成只读的。[接口](#)一章有详细说明。

let VS. const

现在我们有两种作用域相似的声明方式，我们自然会问到底应该使用哪个。与大多数泛泛的问题一样，答案是：依情况而定。

使用[最小特权原则](#)，所有变量除了你计划去修改的都应该使用 `const`。基本原则就是如果一个变量不需要对它写入，那么其它使用这些代码的人也不能够写入它们，并且要思考为什么会需要对这些变量重新赋值。使用 `const` 也可以让我们更容易的推测数据的流动。

另一方面，用户很喜欢 `let` 的简洁性。这个手册大部分地方都使用了 `let`。

跟据你的自己判断，如果合适的话，与团队成员商议一下。

解构

Another TypeScript 已经可以解析其它 ECMAScript 2015 特性了。完整列表请参见 [the article on the Mozilla Developer Network](#)。本章，我们将给出一个简短的概述。

解构数组

最简单的解构莫过于数组的解构赋值了：

```
let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2
```

这创建了 2 个命名变量 `first` 和 `second`。相当于使用了索引，但更为方便：

```
first = input[0];
second = input[1];
```

解构作用于已声明的变量会更好：

```
// swap variables
[first, second] = [second, first];
```

作用于函数参数：

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

你可以在数组里使用...语法创建剩余变量：

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // outputs 1
console.log(rest); // outputs [ 2, 3, 4 ]
```

当然，由于是 JavaScript，你可以忽略你不关心的尾随元素：

```
let [first] = [1, 2, 3, 4];
console.log(first); // outputs 1
```

或其它元素：

```
let [, second, , fourth] = [1, 2, 3, 4];
```

对象解构

你也可以解构对象：

```
let o = {
  a: "foo",
  b: 12,
  c: "bar"
};
```

```
let { a, b } = o;
```

这通过 `o.a` 和 `o.b` 创建了 `a` 和 `b`。注意，如果你不需要 `c` 你可以忽略它。

就像数组解构，你可以用没有声明的赋值：

```
({ a, b } = { a: "baz", b: 101 });
```

注意，我们需要用括号将它括起来，因为 Javascript 通常会将以 `{` 起始的语句解析为一个块。

你可以在对象里使用 `...` 语法创建剩余变量：

```
let { a, ...passthrough } = o;  
let total = passthrough.b + passthrough.c.length;
```

属性重命名

你也可以给属性以不同的名字：

```
let { a: newName1, b: newName2 } = o;
```

这里的语法开始变得混乱。你可以将 `a: newName1` 读做 "`a` 作为 `newName1`"。方向是从左到右，好像你写成了以下样子：

```
let newName1 = o.a;  
let newName2 = o.b;
```

令人困惑的是，这里的冒号不是指示类型的。如果你想指定它的类型，仍然需要在其后写上完整的模式。

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

默认值

默认值可以让你在属性为 `undefined` 时使用缺省值：

```
function keepWholeObject(wholeObject: { a: string, b?: number }) {  
  let { a, b = 1001 } = wholeObject;
```



```
}
```

现在，即使 **b** 为 `undefined`，`keepWholeObject` 函数的变量 `wholeObject` 的属性 **a** 和 **b** 都会有值。

函数声明

解构也能用于函数声明。看以下简单的情况：

```
type C = { a: string, b?: number }  
function f({ a, b }: C): void {  
    // ...  
}
```

但是，通常情况下更多的是指定默认值，解构默认值有些棘手。首先，你需要在默认值之前设置其格式。

```
function f({ a, b } = { a: "", b: 0 }): void {  
    // ...  
}  
f(); // ok, default to { a: "", b: 0 }
```

上面的代码是一个类型推断的例子，将在本手册后文介绍。

其次，你需要知道在解构属性上给予一个默认或可选的属性用来替换主初始化列表。要知道 **C** 的定义有一个 **b** 可选属性：

```
function f({ a, b = 0 } = { a: "" }): void {  
    // ...  
}  
f({ a: "yes" }); // ok, default b = 0  
f(); // ok, default to {a: ""}, which then defaults b = 0  
f({}); // error, 'a' is required if you supply an argument
```

要小心使用解构。从前面的例子可以看出，就算是最简单的解构表达式也是难以理解的。尤其当存在深层嵌套解构的时候，就算这时没有堆叠在一起的重命名，默认值和类型注解，也是令人难以理解的。解构表达式要尽量保持小而简单。你自己也可以直接使用解构将会生成的赋值表达式。

展开

展开操作符正与解构相反。它允许你将一个数组展开为另一个数组，或将一个对象展开为另一个对象。例如：

```
let first = [1, 2];
let second = [3, 4];
let bothPlus = [0, ...first, ...second, 5];
```

这会令 **bothPlus** 的值为 **[0, 1, 2, 3, 4, 5]**。展开操作创建了 **first** 和 **second** 的一份浅拷贝。它们不会被展开操作所改变。

你还可以展开对象：

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };
let search = { ...defaults, food: "rich" };
```

search 的值为 **{ food: "rich", price: "\$\$", ambiance: "noisy" }**。对象的展开比数组的展开要复杂的多。像数组展开一样，它是从左至右进行处理，但结果仍为对象。这就意味着出现在展开对象后面的属性会覆盖前面的属性。因此，如果我们修改上面的例子，在结尾处进行展开的话：

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };
let search = { food: "rich", ...defaults };
```

那么，**defaults** 里的 **food** 属性会重写 **food: "rich"**，在这里这并不是我们想要的结果。

对象展开还有其它一些意想不到的限制。首先，它只包含自身的可枚举的属性。首先，当你展开一个对象实例时，你会丢失其方法：

```
class C {
  p = 12;
  m() {
  }
}
let c = new C();
let clone = { ...c };
clone.p; // ok
clone.m(); // error!
```

其次，TypeScript 编译器不允许展开泛型函数上的类型参数。这个特性会在 TypeScript 的未来版本中考虑实现。

接口

介绍

TypeScript 的核心原则之一是对值所具有的 *结构* 进行类型检查。它有时被称做“鸭式辨型法”或“结构性子类型化”。在 TypeScript 里，接口的作用就是为这些类型命名和为你的代码或第三方代码定义契约。

接口初探

下面通过一个简单示例来观察接口是如何工作的：

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

类型检查器会查看 `printLabel` 的调用。`printLabel` 有一个参数，并要求这个对象参数有一个名为 `label` 类型为 `string` 的属性。需要注意的是，我们传入的对象参数实际上会包含很多属性，但是编译器只会检查那些必需的属性是否存在，并且其类型是否匹配。然而，有些时候 TypeScript 却并不会这么宽松，我们下面会稍做讲解。

下面我们重写上面的例子，这次使用接口来描述：必须包含一个 `label` 属性且类型为 `string`：

```
interface LabelledValue {  
    label: string;  
}  
  
function printLabel(labelledObj: LabelledValue) {  
    console.log(labelledObj.label);  
}
```

```
}  
  
let myObj = {size: 10, label: "Size 10 Object"};  
printLabel(myObj);
```

LabelledValue 接口就好比一个名字，用来描述上面例子里的要求。它代表了有一个 **label** 属性且类型为 **string** 的对象。需要注意的是，我们在这里并不能像在其它语言里一样，说传给 **printLabel** 的对象实现了这个接口。我们只会去关注值的外形。只要传入的对象满足上面提到的必要条件，那么它就被允许的。

还有一点值得提的是，类型检查器不会去检查属性的顺序，只要相应的属性存在并且类型也是对的就可以。

可选属性

接口里的属性不全都是必需的。有些是只在某些条件下存在，或者根本不存在。可选属性在应用“option bags”模式时很常用，即给函数传入的参数对象中只有部分属性赋值了。

下面是应用了“option bags”的例子：

```
interface SquareConfig {  
  color?: string;  
  width?: number;  
}  
  
function createSquare(config: SquareConfig): {color: string; area: number} {  
  let newSquare = {color: "white", area: 100};  
  if (config.color) {  
    newSquare.color = config.color;  
  }  
  if (config.width) {  
    newSquare.area = config.width * config.width;  
  }  
  return newSquare;  
}
```

```
let mySquare = createSquare({color: "black"});
```

带有可选属性的接口与普通的接口定义差不多，只是在可选属性名字定义的后面加一个`?`符号。

可选属性的好处之一是可以对可能存在的属性进行预定义，好处之二是可以捕获引用了不存在的属性时的错误。比如，我们故意将 `createSquare` 里的 `color` 属性名拼错，就会得到一个错误提示：

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  let newSquare = {color: "white", area: 100};
  if (config.color) {
    // Error: Property 'clor' does not exist on type 'SquareConfig'
    newSquare.color = config.clor;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({color: "black"});
```

只读属性

一些对象属性只能在对象刚刚创建的时候修改其值。你可以在属性名前用 `readonly` 来指定只读属性：

```
interface Point {
  readonly x: number;
  readonly y: number;
}
```

你可以通过赋值一个对象字面量来构造一个 `Point`。赋值后，`x` 和 `y` 再也不能被改变了。

```
let p1: Point = { x: 10, y: 20 };
```

```
p1.x = 5; // error!
```

TypeScript 具有 `ReadonlyArray<T>` 类型，它与 `Array<T>` 相似，只是把所有可变方法去掉了，因此可以确保数组创建后再也不能被修改：

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

上面代码的最后一行，可以看到就算把整个 `ReadonlyArray` 赋值到一个普通数组也是不可以的。但是你可以用类型断言重写：

```
a = ro as number[];
```

readonly VS const

最简单判断该用 `readonly` 还是 `const` 的方法是看要把它做为变量使用还是做为一个属性。做为变量使用的话用 `const`，若做为属性则使用 `readonly`。

额外的属性检查

我们在第一个例子里使用了接口，TypeScript 让我们传入 `{ size: number; label: string; }` 到仅期望得到 `{ label: string; }` 的函数里。我们已经学过了可选属性，并且知道他们在“option bags”模式里很有用。

然而，天真地将这两者结合的话就会像在 JavaScript 里那样搬起石头砸自己的脚。比如，拿 `createSquare` 例子来说：

```
interface SquareConfig {
  color?: string;
  width?: number;
}
```

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

注意传入 `createSquare` 的参数拼写为 `colour` 而不是 `color`。在 JavaScript 里，这会默默地失败。

你可能会争辩这个程序已经正确地类型化了，因为 `width` 属性是兼容的，不存在 `color` 属性，而且额外的 `colour` 属性是无意义的。

然而，TypeScript 会认为这段代码可能存在 bug。对象字面量会被特殊对待而且会经过 *额外属性检查*，当将它们赋值给变量或作为参数传递的时候。如果一个对象字面量存在任何“目标类型”不包含的属性时，你会得到一个错误。

```
// error: 'colour' not expected in type 'SquareConfig'  
let mySquare = createSquare({ colour: "red", width: 100 });
```

绕开这些检查非常简单。最简便的方法是使用类型断言：

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

然而，最佳的方式是能够添加一个字符串索引签名，前提是你能够确定这个对象可能具有某些做为特殊用途使用的额外属性。如果 `SquareConfig` 带有上面定义的地类型的 `color` 和 `width` 属性，并且还会带有任意数量的其它属性，那么我们可以这样定义它：

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

我们稍后会讲到索引签名，但在这我们要表示的是 `SquareConfig` 可以有任意数量的属性，并且只要它们不是 `color` 和 `width`，那么就无所谓它们的类型是什么。

还有最后一种跳过这些检查的方式，这可能会让你感到惊讶，它就是将这个对象赋值给一个另一个变量：因为 `squareOptions` 不会经过额外属性检查，所以编译器不会报错。

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

要留意，在像上面一样的简单代码里，你可能不应该去绕开这些检查。对于包含方法和内部状态的复杂对象字面量来讲，你可能需要使用这些技巧，但是大部额外属性检查错误是真正的 bug。就是说你遇到了额外类型检查出的错误，比如“option bags”，你应该去审查一下你的类型声明。在这里，如果支持传入 `color` 或 `colour` 属性到 `createSquare`，你应该修改 `SquareConfig` 定义来体现出这一点。

函数类型

接口能够描述 JavaScript 中对象拥有的各种各样的外形。除了描述带有属性的普通对象外，接口也可以描述函数类型。

为了使用接口表示函数类型，我们需要给接口定义一个调用签名。它就像是一个只有参数列表和返回值类型的函数定义。参数列表里的每个参数都需要名字和类型。

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

这样定义后，我们可以像使用其它接口一样使用这个函数类型的接口。下例展示了如何创建一个函数类型的变量，并将一个同类型的函数赋值给这个变量。

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

对于函数类型的类型检查来说，函数的参数名不需要与接口里定义的名字相匹配。比如，我们使用下面的代码重写上面的例子：

```
let mySearch: SearchFunc;
mySearch = function(src: string, sub: string): boolean {
  let result = src.search(sub);
  return result > -1;
}
```


函数的参数会逐个进行检查，要求对应位置上的参数类型是兼容的。如果你不想指定类型，TypeScript 的类型系统会推断出参数类型，因为函数直接赋值给了 `SearchFunc` 类型变量。函数的返回值类型是通过其返回值推断出来的（此例是 `false` 和 `true`）。如果让这个函数返回数字或字符串，类型检查器会警告我们函数的返回值类型与 `SearchFunc` 接口中的定义不匹配。

```
let mySearch: SearchFunc;
mySearch = function(src, sub) {
    let result = src.search(sub);
    return result > -1;
}
```

可索引的类型

与使用接口描述函数类型差不多，我们也可以描述那些能够“通过索引得到”的类型，比如 `a[10]` 或 `ageMap["daniel"]`。可索引类型具有一个索引/签名，它描述了对象索引的类型，还有相应的索引返回值类型。让我们看一个例子：

```
interface StringArray {
    [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

上面例子里，我们定义了 `StringArray` 接口，它具有索引签名。这个索引签名表示了当用 `number` 去索引 `StringArray` 时会得到 `string` 类型的返回值。

共有支持两种索引签名：字符串和数字。可以同时使用两种类型的索引，但是数字索引的返回值必须是字符串索引返回值类型的子类型。这是因为当使用 `number` 来索引时，JavaScript 会将它转换成 `string` 然后再去索引对象。也就是说用 `100`（一个 `number`）去索引等同于使用 `"100"`（一个 `string`）去索引，因此两者需要保持一致。

```
class Animal {
    name: string;
}
class Dog extends Animal {
```

```

    breed: string;
}

// Error: indexing with a 'string' will sometimes get you a Dog!
interface NotOkay {
    [x: number]: Animal;
    [x: string]: Dog;
}

```

字符串索引签名能够很好的描述 **dictionary** 模式，并且它们也会确保所有属性与其返回值类型相匹配。因为字符串索引声明了 **obj.property** 和 **obj["property"]** 两种形式都可以。下面的例子里，**name** 的类型与字符串索引类型不匹配，所以类型检查器给出一个错误提示：

```

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

```

最后，你可以将索引签名设置为只读，这样就防止了给索引赋值：

```

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

```

你不能设置 **myArray[2]**，因为索引签名是只读的。

类类型

实现接口

与 C# 或 Java 里接口的基本作用一样，TypeScript 也能够用它来明确的强制一个类去符合某种契约。

```

interface ClockInterface {
    currentTime: Date;
}

```

```

}

class Clock implements ClockInterface {
    currentTime: Date;
    constructor(h: number, m: number) { }
}

```

你也可以在接口中描述一个方法，在类里实现它，如同下面的 `setTime` 方法一样：

```

interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

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

```

接口描述了类的公共部分，而不是公共和私有两部分。它不会帮你检查类是否具有某些私有成员。

类静态部分与实例部分的区别

当你操作类和接口的时候，你要知道类是具有两个类型的：静态部分的类型和实例的类型。你会注意到，当你用构造器签名去定义一个接口并试图定义一个类去实现这个接口时会得到一个错误：

```

interface ClockConstructor {
    new (hour: number, minute: number);
}

class Clock implements ClockConstructor {
    currentTime: Date;
    constructor(h: number, m: number) { }
}

```

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

因此，我们应该直接操作类的静态部分。 看下面的例子，我们定义了两个接口，`ClockConstructor` 为构造函数所用和 `ClockInterface` 为实例方法所用。 为了方便我们定义一个构造函数 `createClock`，它用传入的类型创建实例。

```
interface ClockConstructor {
    new (hour: number, minute: number): ClockInterface;
}
interface ClockInterface {
    tick();
}

function createClock(ctor: ClockConstructor, hour: number, minute: number): ClockInterface {
    return new ctor(hour, minute);
}

class DigitalClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("beep beep");
    }
}

class AnalogClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("tick tock");
    }
}

let digital = createClock(DigitalClock, 12, 17);
let analog = createClock(AnalogClock, 7, 32);
```

因为 `createClock` 的第一个参数是 `ClockConstructor` 类型，在 `createClock(AnalogClock, 7, 32)` 里，会检查 `AnalogClock` 是否符合构造函数签名。

继承接口

和类一样，接口也可以相互继承。这让我们能够从一个接口里复制成员到另一个接口里，可以更灵活地将接口分割到可重用的模块里。

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

一个接口可以继承多个接口，创建出多个接口的合成接口。

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

混合类型

先前我们提过，接口能够描述 JavaScript 里丰富的类型。因为 JavaScript 其动态灵活的特点，有时你会希望一个对象可以同时具有上面提到的多种类型。

一个例子就是，一个对象可以同时做为函数和对象使用，并带有额外的属性。

```
interface Counter {  
  (start: number): string;  
  interval: number;  
  reset(): void;  
}  
  
function getCounter(): Counter {  
  let counter = <Counter>function (start: number) { };  
  counter.interval = 123;  
  counter.reset = function () { };  
  return counter;  
}  
  
let c = getCounter();  
c(10);  
c.reset();  
c.interval = 5.0;
```

在使用 JavaScript 第三方库的时候，你可能需要像上面那样去完整地定义类型。

接口继承类

当接口继承了一个类类型时，它会继承类的成员但不包括其实现。就好像接口声明了所有类中存在的成员，但并没有提供具体实现一样。接口同样会继承到类的 `private` 和 `protected` 成员。这意味着当你创建了一个接口继承了一个拥有私有或受保护的成员的类时，这个接口类型只能被这个类或其子类所实现（implement）。

当你有一个庞大的继承结构时这很有用，但要指出的是你的代码只在子类拥有特定属性时起作用。这个子类除了继承至基类外与基类没有任何关系。例：

```
class Control {
    private state: any;
}

interface SelectableControl extends Control {
    select(): void;
}

class Button extends Control {
    select() { }
}

class TextBox extends Control {
    select() { }
}

class Image {
    select() { }
}

class Location {
    select() { }
}
```

在上面的例子里，**SelectableControl** 包含了 **Control** 的所有成员，包括私有成员 **state**。因为 **state** 是私有成员，所以只能是 **Control** 的子类们才能实现 **SelectableControl** 接口。因为只有 **Control** 的子类才能够拥有一个声明于 **Control** 的私有成员 **state**，这对私有成员的兼容性是必需的。

在 **Control** 类内部，是允许通过 **SelectableControl** 的实例来访问私有成员 **state** 的。实际上，**SelectableControl** 就像 **Control** 一样，并拥有一个 **select** 方法。**Button** 和 **TextBox** 类是 **SelectableControl** 的子类（因为它们都继承自 **Control** 并有 **select** 方法），但 **Image** 和 **Location** 类并不是这样的。

类

介绍

传统的 JavaScript 程序使用函数和基于原型的继承来创建可重用的组件，但对于熟悉使用面向对象方式的程序员来讲就有些棘手，因为他们用的是基于类的继承并且对象是由类构建出来的。从 ECMAScript 2015，也就是 ECMAScript 6 开始，JavaScript 程序员将能够使用基于类的面向对象的方式。使用 TypeScript，我们允许开发者现在就使用这些特性，并且编译后的 JavaScript 可以在所有主流浏览器和平台上运行，而不需要等到下个 JavaScript 版本。

类

下面看一个使用类的例子：

```
class Greeter {  
    greeting: string;  
    constructor(message: string) {  
        this.greeting = message;  
    }  
    greet() {  
        return "Hello, " + this.greeting;  
    }  
}  
  
let greeter = new Greeter("world");
```

如果你使用过 C# 或 Java，你会对这种语法非常熟悉。我们声明一个 **Greeter** 类。这个类有 3 个成员：一个叫做 **greeting** 的属性，一个构造函数和一个 **greet** 方法。

你会注意到，我们在引用任何一个类成员的时候都用了 **this**。它表示我们访问的是类的成员。

最后一行，我们使用 **new** 构造了 **Greeter** 类的一个实例。它会调用之前定义的构造函数，创建一个 **Greeter** 类型的新对象，并执行构造函数初始化它。

继承

在 TypeScript 里，我们可以使用常用的面向对象模式。当然，基于类的程序设计中最基本的模式是允许使用继承来扩展现有的类。

看下面的例子：

```
class Animal {
  name:string;
  constructor(theName: string) { this.name = theName; }
  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Snake extends Animal {
  constructor(name: string) { super(name); }
  move(distanceInMeters = 5) {
    console.log("Slithering...");
    super.move(distanceInMeters);
  }
}

class Horse extends Animal {
  constructor(name: string) { super(name); }
  move(distanceInMeters = 45) {
    console.log("Galloping...");
    super.move(distanceInMeters);
  }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

这个例子展示了 TypeScript 中继承的一些特征，它们与其它语言类似。我们使用 `extends` 关键字来创建子类。你可以看到 `Horse` 和 `Snake` 类是基类 `Animal` 的子类，并且可以访问其属性和方法。

包含构造函数的派生类必须调用 `super()`，它会执行基类的构造方法。

这个例子演示了如何在子类里可以重写父类的方法。`Snake` 类和 `Horse` 类都创建了 `move` 方法，它们重写了从 `Animal` 继承来的 `move` 方法，使得 `move` 方法根据不同的类而具有不同的功能。注意，即使 `tom` 被声明为 `Animal` 类型，但因为它的值是 `Horse`，`tom.move(34)` 会调用 `Horse` 里的重写方法：

```
Slithering...
Sammy the Python moved 5m.
Gallopig...
Tommy the Palomino moved 34m.
```

公共，私有与受保护的修饰符

默认为 `public`

在上面的例子里，我们可以自由的访问程序里定义的成员。如果你对其它语言中的类比较了解，就会注意到我们在之前的代码里并没有使用 `public` 来做修饰；例如，C# 要求必须明确地使用 `public` 指定成员是可见的。在 TypeScript 里，成员都默认为 `public`。

你也可以明确的将一个成员标记成 `public`。我们可以用下面的方式来重写上面的 `Animal` 类：

```
class Animal {
  public name: string;
  public constructor(theName: string) { this.name = theName; }
  public move(distanceInMeters: number) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}
```

理解 `private`

当成员被标记成 `private` 时，它就不能在声明它的类的外部访问。比如：

```
class Animal {  
    private name: string;  
    constructor(theName: string) { this.name = theName; }  
}  
  
new Animal("Cat").name; // Error: 'name' is private;
```

TypeScript 使用的是结构性类型系统。当我们比较两种不同的类型时，并不在乎它们从何处而来，如果所有成员的类型都是兼容的，我们就认为它们的类型是兼容的。

然而，当我们比较带有 `private` 或 `protected` 成员的类型的时候，情况就不同了。如果其中一个类型里包含一个 `private` 成员，那么只有当另外一个类型中也存在这样一个 `private` 成员，并且它们都是来自同一处声明时，我们才认为这两个类型是兼容的。对于 `protected` 成员也使用这个规则。

下面来看一个例子，更好地说明了这一点：

```
class Animal {  
    private name: string;  
    constructor(theName: string) { this.name = theName; }  
}  
  
class Rhino extends Animal {  
    constructor() { super("Rhino"); }  
}  
  
class Employee {  
    private name: string;  
    constructor(theName: string) { this.name = theName; }  
}  
  
let animal = new Animal("Goat");  
let rhino = new Rhino();  
let employee = new Employee("Bob");
```

```
animal = rhino;  
animal = employee; // Error: Animal and Employee are not compatible
```

这个例子中有 **Animal** 和 **Rhino** 两个类，**Rhino** 是 **Animal** 类的子类。还有一个 **Employee** 类，其类型看上去与 **Animal** 是相同的。我们创建了几个这些类的实例，并相互赋值来看看会发生什么。因为 **Animal** 和 **Rhino** 共享了来自 **Animal** 里的私有成员定义 **private name: string**，因此它们是兼容的。然而 **Employee** 却不是这样。当把 **Employee** 赋值给 **Animal** 的时候，得到一个错误，说它们的类型不兼容。尽管 **Employee** 里也有一个私有成员 **name**，但它明显不是 **Animal** 里面定义的那个。

理解 **protected**

protected 修饰符与 **private** 修饰符的行为很相似，但有一点不同，**protected** 成员在派生类中仍然可以访问。例如：

```
class Person {  
    protected name: string;  
    constructor(name: string) { this.name = name; }  
}  
  
class Employee extends Person {  
    private department: string;  
  
    constructor(name: string, department: string) {  
        super(name)  
        this.department = department;  
    }  
  
    public getElevatorPitch() {  
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;  
    }  
}  
  
let howard = new Employee("Howard", "Sales");  
console.log(howard.getElevatorPitch());  
console.log(howard.name); // error
```

注意，我们不能在 **Person** 类外使用 **name**，但是我们仍然可以通过 **Employee** 类的实例方法访问，因为 **Employee** 是由 **Person** 派生而来的。

构造函数也可以被标记成 **protected**。这意味着这个类不能在包含它的类外被实例化，但是能被继承。比如，

```
class Person {
    protected name: string;
    protected constructor(theName: string) { this.name = theName; }
}

// Employee can extend Person
class Employee extends Person {
    private department: string;

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

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John"); // Error: The 'Person' constructor is protected
```

readonly 修饰符

你可以使用 **readonly** 关键字将属性设置为只读的。只读属性必须在声明时或构造函数里被初始化。

```
class Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;
    constructor (theName: string) {
        this.name = theName;
    }
}
```

```
}  
let dad = new Octopus("Man with the 8 strong legs");  
dad.name = "Man with the 3-piece suit"; // error! name is readonly.
```

参数属性

在上面的例子中，我们不得不定义一个受保护的成员 `name` 和一个构造函数参数 `theName` 在 `Person` 类里，并且立刻给 `name` 和 `theName` 赋值。这种情况经常会遇到。参数属性可以方便地让我们在一个地方定义并初始化一个成员。下面的例子是对之前 `Animal` 类的修改版，使用了参数属性：

```
class Animal {  
  constructor(private name: string) { }  
  move(distanceInMeters: number) {  
    console.log(`${this.name} moved ${distanceInMeters}m.`);  
  }  
}
```

注意看我们是如何舍弃了 `theName`，仅在构造函数里使用 `private name: string` 参数来创建和初始化 `name` 成员。我们把声明和赋值合并至一处。

参数属性通过给构造函数参数添加一个访问限定符来声明。使用 `private` 限定一个参数属性会声明并初始化一个私有成员；对于 `public` 和 `protected` 来说也是一样。

存取器

TypeScript 支持通过 `getters/setters` 来截取对对象成员的访问。它能帮助你有效的控制对对象成员的访问。

下面来看如何把一个简单的类改写成使用 `get` 和 `set`。首先，我们从一个没有使用存取器的例子开始。

```
class Employee {  
  fullName: string;  
}  
  
let employee = new Employee();
```

```
employee.fullName = "Bob Smith";
if (employee.fullName) {
    console.log(employee.fullName);
}
```

我们可以随意的设置 `fullName`，这是非常方便的，但是这也会带来麻烦。

下面这个版本里，我们先检查用户密码是否正确，然后再允许其修改员工信息。我们把对 `fullName` 的直接访问改成了可以检查密码的 `set` 方法。我们也加了一个 `get` 方法，让上面的例子仍然可以工作。

```
let passcode = "secret passcode";

class Employee {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }

    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;
        }
        else {
            console.log("Error: Unauthorized update of employee!");
        }
    }
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    alert(employee.fullName);
}
```

我们可以修改一下密码，来验证一下存取器是否是工作的。当密码不对时，会提示我们没有权限去修改员工。

对于存取器有下面几点需要注意的：

首先，存取器要求你将编译器设置为输出 ECMAScript 5 或更高。不支持降级到 ECMAScript 3。其次，只带有 **get** 不带有 **set** 的存取器自动被推断为 **readonly**。这在从代码生成 **.d.ts** 文件时是有帮助的，因为利用这个属性的用户会看到不允许够改变它的值。

静态属性

到目前为止，我们只讨论了类的实例成员，那些仅当类被实例化的时候才会被初始化的属性。我们也可以创建类的静态成员，这些属性存在于类本身上面而不是类的实例上。在这个例子里，我们使用 **static** 定义 **origin**，因为它是所有网格都会用到的属性。每个实例想要访问这个属性的时候，都要在 **origin** 前面加上类名。如同在实例属性上使用 **this.**前缀来访问属性一样，这里我们使用 **Grid.**来访问静态属性。

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

let grid1 = new Grid(1.0); // 1x scale
let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```

抽象类

抽象类做为其它派生类的基类使用。它们一般不会直接被实例化。不同于接口，抽象类可以包含成员的实现细节。**abstract** 关键字是用于定义抽象类和在抽象类内部定义抽象方法。

```
abstract class Animal {
  abstract makeSound(): void;
  move(): void {
    console.log('roaming the earch...');
  }
}
```



```
}  
}
```

抽象类中的抽象方法不包含具体实现并且必须在派生类中实现。抽象方法的语法与接口方法相似。两者都是定义方法签名但不包含方法体。然而，抽象方法必须包含 **abstract** 关键字并且可以包含访问修饰符。

```
abstract class Department {  
  
    constructor(public name: string) {  
    }  
  
    printName(): void {  
        console.log('Department name: ' + this.name);  
    }  
  
    abstract printMeeting(): void; // 必须在派生类中实现  
}  
  
class AccountingDepartment extends Department {  
  
    constructor() {  
        super('Accounting and Auditing'); // constructors in derived classes must call super()  
    }  
  
    printMeeting(): void {  
        console.log('The Accounting Department meets each Monday at 10am.');    }  
  
    generateReports(): void {  
        console.log('Generating accounting reports...');  
    }  
}  
  
let department: Department; // ok to create a reference to an abstract type  
department = new Department(); // error: cannot create an instance of an abstract class  
department = new AccountingDepartment(); // ok to create and assign a non-abstract subclass  
department.printName();  
department.printMeeting();  
department.generateReports(); // error: method doesn't exist on declared abstract type
```

高级技巧

构造函数

当你在 TypeScript 里声明了一个类的时候，实际上同时声明了很多东西。首先就是类的实例的类型。

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}

let greeter: Greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

这里，我们写了 `let greeter: Greeter`，意思是 `Greeter` 类的实例的类型是 `Greeter`。这对于用过其它面向对象语言的程序员来讲已经是老习惯了。

我们也创建了一个叫做构造函数的值。这个函数会在我们使用 `new` 创建类实例的时候被调用。下面我们来看看，上面的代码被编译成 JavaScript 后是什么样子的：

```
let Greeter = (function () {
  function Greeter(message) {
    this.greeting = message;
  }
  Greeter.prototype.greet = function () {
    return "Hello, " + this.greeting;
  };
  return Greeter;
})();

let greeter;
```

```
greeter = new Greeter("world");
console.log(greeter.greet());
```

上面的代码里，**let Greeter** 将被赋值为构造函数。当我们调用 **new** 并执行了这个函数后，便会得到一个类的实例。这个构造函数也包含了类的所有静态属性。换个角度说，我们可以认为类具有**实例部分**与**静态部分**这两个部分。

让我们稍微改写一下这个例子，看看它们之前的区别：

```
class Greeter {
  static standardGreeting = "Hello, there";
  greeting: string;
  greet() {
    if (this.greeting) {
      return "Hello, " + this.greeting;
    }
    else {
      return Greeter.standardGreeting;
    }
  }
}

let greeter1: Greeter;
greeter1 = new Greeter();
console.log(greeter1.greet());

let greeterMaker: typeof Greeter = Greeter;
greeterMaker.standardGreeting = "Hey there!";

let greeter2: Greeter = new greeterMaker();
console.log(greeter2.greet());
```

这个例子里，**greeter1** 与之前看到的一样。我们实例化 **Greeter** 类，并使用这个对象。与我们之前看到的一样。

再之后，我们直接使用类。我们创建了一个叫做 **greeterMaker** 的变量。这个变量保存了这个类或者说保存了类构造函数。然后我们使用 **typeof Greeter**，意思是取 **Greeter** 类的类型，而不是实例的类型。或者更确切的说，"告诉我 **Greeter** 标识符的类型"，也就是构造函数的类型。这个类型包含了类

的所有静态成员和构造函数。之后，就和前面一样，我们在 `greeterMaker` 上使用 `new`，创建 `Greeter` 的实例。

把类当做接口使用

如上一节里所讲的，类定义会创建两个东西：类的实例类型和一个构造函数。因为类可以创建出类型，所以你能够在允许使用接口的地方使用类。

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

interface Point3d extends Point {
  z: number;
}

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

函数

介绍

函数是 JavaScript 应用程序的基础。它帮助你实现抽象层，模拟类，信息隐藏和模块。在 TypeScript 里，虽然已经支持类，命名空间和模块，但函数仍然是主要的定义行为的地方。TypeScript 为 JavaScript 函数添加了额外的功能，让我们可以更容易地使用。

函数

和 JavaScript 一样，TypeScript 函数可以创建有名字的函数和匿名函数。你可以随意选择适合应用程序的方式，不论是定义一系列 API 函数还是只使用一次的函数。

通过下面的例子可以迅速回想起这两种 JavaScript 中的函数：

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

// Anonymous function
let myAdd = function(x, y) { return x + y; };
```

在 JavaScript 里，函数可以使用函数体外部的变量。当函数这么做时，我们说它‘捕获’了这些变量。至于为什么可以这样做以及其中的利弊超出了本文的范围，但是深刻理解这个机制对学习 JavaScript 和 TypeScript 会很有帮助。

```
let z = 100;

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

函数类型

为函数定义类型

让我们为上面那个函数添加类型：

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

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

我们可以给每个参数添加类型之后再为函数本身添加返回值类型。TypeScript 能够根据返回语句自动推断出返回值类型，因此我们通常省略它。

书写完整函数类型

现在我们已经为函数指定了类型，下面让我们写出函数的完整类型。

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

函数类型包含两部分：参数类型和返回值类型。当写出完整函数类型的时候，这两部分都是需要的。我们以参数列表的形式写出参数类型，为每个参数指定一个名字和类型。这个名字只是为了增加可读性。我们也可以这么写：

```
let myAdd: (baseValue:number, increment:number) => number =  
    function(x: number, y: number): number { return x + y; };
```

只要参数类型是匹配的，那么就认为它是有效的函数类型，而不在乎参数名是否正确。

第二部分是返回值类型。对于返回值，我们在函数和返回值类型之前使用(`=>`)符号，使之清晰明了。如之前提到的，返回值类型是函数类型的必要部分，如果函数没有返回任何值，你也必须指定返回值类型为 `void` 而不能留空。

函数的类型只是由参数类型和返回值组成的。函数中使用的捕获变量不会体现在类型里。实际上，这些变量是函数的隐藏状态并不是组成 API 的一部分。

推断类型

尝试这个例子的时候，你会发现如果你在赋值语句的一边指定了类型但是另一边没有类型的话，TypeScript 编译器会自动识别出类型：

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

这叫做“按上下文归类”，是类型推论的一种。它帮助我们更好地为程序指定类型。

可选参数和默认参数

TypeScript 里的每个函数参数都是必须的。这不是指不能传递 **null** 或 **undefined** 作为参数，而是说编译器检查用户是否为每个参数都传入了值。编译器还会假设只有这些参数会被传递进函数。简短地说，传递给一个函数的参数个数必须与函数期望的参数个数一致。

```
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");           // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result3 = buildName("Bob", "Adams");   // ah, just right
```

JavaScript 里，每个参数都是可选的，可传可不传。没传参的时候，它的值就是 **undefined**。在 TypeScript 里我们可以在参数名旁使用 **?** 实现可选参数的功能。比如，我们想让 last name 是可选的：

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}

let result1 = buildName("Bob"); // works correctly now
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result3 = buildName("Bob", "Adams"); // ah, just right
```

可选参数必须跟在必须参数后面。如果上例我们想让 first name 是可选的，那么就必须调整它们的位置，把 first name 放在后面。

在 TypeScript 里，我们也可以为参数提供一个默认值当用户没有传递这个参数或传递的值是 **undefined** 时。它们叫做有默认初始化值的参数。让我们修改上例，把 last name 的默认值设置为 **"Smith"**。

```
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}
```

```
let result1 = buildName("Bob");           // works correctly now, returns "Bob Smith"
let result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith"
let result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result4 = buildName("Bob", "Adams");    // ah, just right
```

在所有必须参数后面的带默认初始化的参数都是可选的，与可选参数一样，在调用函数的时候可以省略。也就是说可选参数与末尾的默认参数共享参数类型。

```
function buildName(firstName: string, lastName?: string) {
    // ...
}
```

和

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
}
```

共享同样的类型(`firstName: string, lastName?: string`) => `string`。默认参数的默认值消失了，只保留了它是一个可选参数的信息。

与普通可选参数不同的是，带默认值的参数不需要放在必须参数的后面。如果带默认值的参数出现在必须参数前面，用户必须明确的传入 `undefined` 值来获得默认值。例如，我们重写最后一个例子，让 `firstName` 是带默认值的参数：

```
function buildName(firstName = "Will", lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");           // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result3 = buildName("Bob", "Adams");    // okay and returns "Bob Adams"
let result4 = buildName(undefined, "Adams"); // okay and returns "Will Adams"
```


剩余参数

必要参数，默认参数和可选参数有个共同点：它们表示某一个参数。有时，你想同时操作多个参数，或者你并不知道会有多少参数传递进来。在 JavaScript 里，你可以使用 **arguments** 来访问所有传入的参数。

在 TypeScript 里，你可以把所有参数收集到一个变量里：

```
function buildName(firstName: string, ...restOfName: string[]) {  
    return firstName + " " + restOfName.join(" ");  
}  
  
let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

剩余参数会被当做个数不限的可选参数。可以一个都没有，同样也可以有任意个。编译器创建参数数组，名字是你在省略号 (...) 后面给定的名字，你可以在函数体内使用这个数组。

这个省略号也会在带有剩余参数的函数类型定义上使用到：

```
function buildName(firstName: string, ...restOfName: string[]) {  
    return firstName + " " + restOfName.join(" ");  
}  
  
let buildNameFun: (fname: string, ...rest: string[]) => string = buildName;
```

this

学习使用 JavaScript 里 **this** 就好比一场成年礼。由于 TypeScript 是 JavaScript 的超集，TypeScript 程序员也需要弄清 **this** 工作机制并且当有 bug 的时候能够找出错误所在。幸运的是，TypeScript 能通知你错误地使用了 **this** 的地方。如果你了解 JavaScript 里的 **this** 是如何工作的，那么首先阅读 Yehuda Katz 写的 [Understanding JavaScript Function Invocation and "this"](#)。Yehuda 的文章详细的阐述了 **this** 的内部工作原理，因此我们这里只做简单介绍。

this 和箭头函数

JavaScript 里，**this** 的值在函数被调用的时候才会指定。这是个既强大又灵活的特点，但是你需要花点时间弄清楚函数调用的上下文是什么。但众所周知，这不是一件很简单的事，尤其是在返回一个函数或将函数当做参数传递的时候。

下面看一个例子：

```
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function() {
    return function() {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

      return {suit: this.suits[pickedSuit], card: pickedCard % 13};
    }
  }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

可以看到 **createCardPicker** 是个函数，并且它又返回了一个函数。如果我们尝试运行这个程序，会发现它并没有弹出对话框而是报错了。因为 **createCardPicker** 返回的函数里的 **this** 被设置成了 **window** 而不是 **deck** 对象。因为我们只是独立的调用了 **cardPicker()**。顶级的非方法式调用会将 **this** 视为 **window**。（注意：在严格模式下，**this** 为 **undefined** 而不是 **window**）。

为了解决这个问题，我们可以在函数被返回时就绑好正确的 **this**。这样的话，无论之后怎么使用它，都会引用绑定的 ‘deck’ 对象。我们需要改变函数表达式来使用 ECMAScript 6 箭头语法。箭头函数能保存函数创建时的 **this** 值，而不是调用时的值：

```
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
```

```

cards: Array(52),
createCardPicker: function() {
    // NOTE: the line below is now an arrow function, allowing us to capture 'this' right
    here
    return () => {
        let pickedCard = Math.floor(Math.random() * 52);
        let pickedSuit = Math.floor(pickedCard / 13);

        return {suit: this.suits[pickedSuit], card: pickedCard % 13};
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);

```

更好事情是，TypeScript 会警告你犯了一个错误，如果你给编译器设置了`--noImplicitThis` 标记。它会指出`this.suits[pickedSuit]`里的`this` 的类型为 `any`。

this 参数

不幸的是，`this.suits[pickedSuit]`的类型依旧为 `any`。这是因为 `this` 来自对象字面量里的函数表达式。修改的方法是，提供一个显式的 `this` 参数。`this` 参数是个假的参数，它出现在参数列表的最前面：

```

function f(this: void) {
    // make sure `this` is unusable in this standalone function
}

```

让我们往例子里添加一些接口，`Card` 和 `Deck`，让类型重用能够变得清晰简单些：

```

interface Card {
    suit: string;
    card: number;
}

interface Deck {
    suits: string[];
}

```

```

    cards: number[];
    createCardPicker(this: Deck): () => Card;
}
let deck: Deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    // NOTE: The function now explicitly specifies that its callee must be of type Deck
    createCardPicker: function(this: Deck) {
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);

```

现在 TypeScript 知道 `createCardPicker` 期望在某个 `Deck` 对象上调用。也就是说 `this` 是 `Deck` 类型的，而非 `any`，因此 `--noImplicitThis` 不会报错了。

`this` 参数在回调函数里

你可以也看到过在回调函数里的 `this` 报错，当你将一个函数传递到某个库函数里稍后会被调用时。因为当回调被调用的时候，它们会被当成一个普通函数调用，`this` 将为 `undefined`。稍做改动，你就可以通过 `this` 参数来避免错误。首先，库函数的作者要指定 `this` 的类型：

```

interface UIElement {
    addClickListener(onclick: (this: void, e: Event) => void): void;
}

```

`this: void` means that `addClickListener` expects `onclick` to be a function that does not require a `this` type. Second, annotate your calling code with `this`:

```

class Handler {

```

```

    info: string;
    onClickBad(this: Handler, e: Event) {
        // oops, used this here. using this callback would crash at runtime
        this.info = e.message;
    }
}
let h = new Handler();
uiElement.addEventListener(h.onClickBad); // error!

```

指定了 `this` 类型后，你显式声明 `onClickBad` 必须在 `Handler` 的实例上调用。然后 TypeScript 会检测到 `addEventListener` 要求函数带有 `this: void`。改变 `this` 类型来修复这个错误：

```

class Handler {
    info: string;
    onClickGood(this: void, e: Event) {
        // can't use this here because it's of type void!
        console.log('clicked!');
    }
}
let h = new Handler();
uiElement.addEventListener(h.onClickGood);

```

因为 `onClickGood` 指定了 `this` 类型为 `void`，因此传递 `addEventListener` 是合法的。当然了，这也意味着不能使用 `this.info`。如果你两者都想要，你不得不使用箭头函数了：

```

class Handler {
    info: string;
    onClickGood = (e: Event) => { this.info = e.message }
}

```

这是可行的因为箭头函数不会捕获 `this`，所以你总是可以把它传给期望 `this: void` 的函数。缺点是每个 `Handler` 对象都会创建一个箭头函数。另一方面，方法只会被创建一次，添加到 `Handler` 的原型链上。它们在不同 `Handler` 对象间是共享的。

重载

JavaScript 本身是个动态语言。JavaScript 里函数根据传入不同的参数而返回不同类型的数据是很常见的。

```

let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit: "hearts", card: 4 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);

```

pickCard 方法根据传入参数的不同会返回两种不同的类型。如果传入的是代表纸牌的对象，函数作用是从中抓一张牌。如果用户想抓牌，我们告诉他抓到了什么牌。但是这怎么在类型系统里表示呢。

方法是为同一个函数提供多个函数类型定义来进行函数重载。编译器会根据这个列表去处理函数的调用。下面我们来重载 **pickCard** 函数。

```

let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
}

```

```

    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit: "hearts", card: 4 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);

```

这样改变后，重载的 `pickCard` 函数在调用的时候会进行正确的类型检查。

为了让编译器能够选择正确的检查类型，它与 JavaScript 里的处理流程相似。它查找重载列表，尝试使用第一个重载定义。如果匹配的话就使用这个。因此，在定义重载的时候，一定要把最精确的定义放在最前面。

注意，`function pickCard(x): any` 并不是重载列表的一部分，因此这里只有两个重载：一个是接收对象另一个接收数字。以其它参数调用 `pickCard` 会产生错误。

泛型

介绍

软件工程中，我们不仅要创建一致的定义良好的 API，同时也要考虑可重用性。组件不仅能够支持当前的数据类型，同时也能支持未来的数据类型，这在创建大型系统时为你提供了十分灵活的功能。

在像 C# 和 Java 这样的语言中，可以使用 `泛型` 来创建可重用的组件，一个组件可以支持多种类型的数据。这样用户就可以以自己的数据类型来使用组件。

泛型之 Hello World

下面来创建第一个使用泛型的例子：`identity` 函数。这个函数会返回任何传入它的值。你可以把这个函数当成是 `echo` 命令。

不用泛型的话，这个函数可能是下面这样：

```
function identity(arg: number): number {  
    return arg;  
}
```

或者，我们使用 `any` 类型来定义函数：

```
function identity(arg: any): any {  
    return arg;  
}
```

虽然使用 `any` 类型后这个函数已经能接收任何类型的 `arg` 参数，但是却丢失了一些信息：传入的类型与返回的类型应该是相同的。如果我们传入一个数字，我们只知道任何类型的值都有可能被返回。

因此，我们需要一种方法使返回值的类型与传入参数的类型是相同的。这里，我们使用了类型变量，它是一种特殊的变量，只用于表示类型而不是值。

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

我们给 `identity` 添加了类型变量 `T`。`T` 帮助我们捕获用户传入的类型（比如：`number`），之后我们就可以使用这个类型。之后我们再次使用了 `T` 当做返回值类型。现在我们可以知道参数类型与返回值类型是相同的了。这允许我们跟踪函数里使用的类型的信息。

我们把这个版本的 `identity` 函数叫做泛型，因为它可以适用于多个类型。不同于使用 `any`，它不会丢失信息，像第一个例子那样保持准确性，传入数值类型并返回数值类型。

我们定义了泛型函数后，可以用两种方法使用。第一种是，传入所有的参数，包含类型参数：


```
let output = identity<string>("myString"); // type of output will be 'string'
```

这里我们明确的指定了 **T** 是 **string** 类型，并做为一个参数传给函数，使用了 **<>** 括起来而不是 **()**。

第二种方法更普遍。利用了 *类型推论*-- 即编译器会根据传入的参数自动地帮助我们确定 **T** 的类型：

```
let output = identity("myString"); // type of output will be 'string'
```

注意我们没必要使用尖括号 (**<>**) 来明确地传入类型；编译器可以查看 **myString** 的值，然后把 **T** 设置为它的类型。类型推论帮助我们保持代码精简和高可读性。如果编译器不能够自动地推断出类型的话，只能像上面那样明确的传入 **T** 的类型，在一些复杂的情况下，这是可能出现的。

使用泛型变量

使用泛型创建像 **identity** 这样的泛型函数时，编译器要求你在函数体必须正确的使用这个通用的类型。换句话说，你必须把这些参数当做是任意或所有类型。

看下之前 **identity** 例子：

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

如果我们想同时打印出 **arg** 的长度。我们很可能会这样做：

```
function loggingIdentity<T>(arg: T): T {  
    console.log(arg.length); // Error: T doesn't have .length  
    return arg;  
}
```

如果这么做，编译器会报错说我们使用了 **arg** 的 **.length** 属性，但是没有地方指明 **arg** 具有这个属性。记住，这些类型变量代表的是任意类型，所以使用这个函数的人可能传入的是个数字，而数字是没有 **.length** 属性的。

现在假设我们想操作 **T** 类型的数组而不直接是 **T**。由于我们操作的是数组，所以 **.length** 属性是应该存在的。我们可以像创建其它数组一样创建这个数组：

```
function loggingIdentity<T>(arg: T[]): T[] {  
    console.log(arg.length); // Array has a .length, so no more error  
    return arg;  
}
```

你可以这样理解 **loggingIdentity** 的类型：泛型函数 **loggingIdentity**，接收类型参数 **T** 和参数 **arg**，它是个元素类型是 **T** 的数组，并返回元素类型是 **T** 的数组。如果我们传入数字数组，将返回一个数字数组，因为此时 **T** 的类型为 **number**。这可以让我们把泛型变量 **T** 当做类型的一部分使用，而不是整个类型，增加了灵活性。

我们也可以这样实现上面的例子：

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {  
    console.log(arg.length); // Array has a .length, so no more error  
    return arg;  
}
```

使用过其它语言的话，你可能对这种语法已经很熟悉了。在下一节，会介绍如何创建自定义泛型像 **Array<T>** 一样。

泛型类型

上一节，我们创建了 **identity** 通用函数，可以适用于不同的类型。在这节，我们研究一下函数本身的类型，以及如何创建泛型接口。

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

```
function identity<T>(arg: T): T {  
    return arg;  
}  
  
let myIdentity: <T>(arg: T) => T = identity;
```

我们也可以使用不同的泛型参数名，只要在数量上和使用方式上能对应上就可以。

```
function identity<T>(arg: T): T {  
    return arg;  
}  
  
let myIdentity: <U>(arg: U) => U = identity;
```

我们还可以使用带有调用签名的对象字面量来定义泛型函数：

```
function identity<T>(arg: T): T {  
    return arg;  
}  
  
let myIdentity: {<T>(arg: T): T} = identity;
```

这引导我们去写第一个泛型接口了。我们把上面例子里的对象字面量拿出来做为一个接口：

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

一个相似的例子，我们可能想把泛型参数当作整个接口的一个参数。这样我们就能清楚的知道使用的具体是哪个泛型类型（比如：**Dictionary<string>**而不只是**Dictionary**）。这样接口里的其它成员也能知道这个参数的类型了。

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

注意，我们的示例做了少许改动。不再描述泛型函数，而是把非泛型函数签名作为泛型类型一部分。当我们使用 `GenericIdentityFn` 的时候，还得传入一个类型参数来指定泛型类型（这里是：`number`），锁定了之后代码里使用的类型。对于描述哪部分类型属于泛型部分来说，理解何时把参数放在调用签名里和何时放在接口上是很有帮助的。

除了泛型接口，我们还可以创建泛型类。注意，无法创建泛型枚举和泛型命名空间。

泛型类

泛型类看上去与泛型接口差不多。泛型类使用 (`<>`) 括起泛型类型，跟在类名后面。

```
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; };
```

`GenericNumber` 类的使用是十分直观的，并且你可能已经注意到了，没有什么去限制它只能使用 `number` 类型。也可以使用字符串或其它更复杂的类型。

```
let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };

alert(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

与接口一样，直接把泛型类型放在类后面，可以帮助我们确认类的所有属性都在使用相同的类型。

我们在[类](#)那节说过，类有两部分：静态部分和实例部分。泛型类指的是实例部分的类型，所以类的静态属性不能使用这个泛型类型。

泛型约束

你应该会记得之前的一个例子，我们有时候想操作某类型的一组值，并且我们知道这组值具有什么样的属性。在 `loggingIdentity` 例子中，我们想访问 `arg` 的 `length` 属性，但是编译器并不能证明每种类型都有 `length` 属性，所以就报错了。

```
function loggingIdentity<T>(arg: T): T {  
    console.log(arg.length); // Error: T doesn't have .length  
    return arg;  
}
```

相比于操作 `any` 所有类型，我们想要限制函数去处理任意带有 `.length` 属性的所有类型。只要传入的类型有这个属性，我们就允许，就是说至少包含这一属性。为此，我们需要列出对于 `T` 的约束要求。

为此，我们定义一个接口来描述约束条件。创建一个包含 `.length` 属性的接口，使用这个接口和 `extends` 关键字还实现约束：

```
interface Lengthwise {  
    length: number;  
}  
  
function loggingIdentity<T extends Lengthwise>(arg: T): T {  
    console.log(arg.length); // Now we know it has a .length property, so no more error  
    return arg;  
}
```

现在这个泛型函数被定义了约束，因此它不再是适用于任意类型：

```
loggingIdentity(3); // Error, number doesn't have a .length property
```

我们需要传入符合约束类型的值，必须包含必须的属性：

```
loggingIdentity({length: 10, value: 3});
```

在泛型约束中使用类型参数

你可以声明一个类型参数，且它被另一个类型参数所约束。比如，现在我们想要用属性名从对象里获取这个属性。并且我们想要确保这个属性存在于对象 **obj** 上，因此我们需要在这两个类型之间使用约束。

```
function getProperty(obj: T, key: K) {  
    return obj[key];  
}  
  
let x = { a: 1, b: 2, c: 3, d: 4 };  
  
getProperty(x, "a"); // okay  
getProperty(x, "m"); // error: Argument of type 'm' isn't assignable to 'a' | 'b' | 'c' | 'd'.
```

在泛型里使用类类型

在 TypeScript 使用泛型创建工厂函数时，需要引用构造函数的类类型。比如，

```
function create<T>(c: {new(): T}): T {  
    return new c();  
}
```

一个更高级的例子，使用原型属性推断并约束构造函数与类实例的关系。

```
class BeeKeeper {  
    hasMask: boolean;  
}  
  
class ZooKeeper {  
    nametag: string;  
}  
  
class Animal {  
    numLegs: number;  
}  
  
class Bee extends Animal {  
    keeper: BeeKeeper;  
}
```

```

class Lion extends Animal {
    keeper: ZooKeeper;
}

function createInstance<A extends Animal>(c: new () => A): A {
    return new c();
}

createInstance(Lion).keeper.nametag; // typechecks!
createInstance(Bee).keeper.hasMask; // typechecks!

```

枚举

枚举

使用枚举我们可以定义一些有名字的数字常量。枚举通过 **enum** 关键字来定义。

```

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

```

一个枚举类型可以包含零个或多个枚举成员。枚举成员具有一个数字值，它可以是 *常数* 或是 *计算得出的值* 当满足如下条件时，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中一个操作对象 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。

```
enum FileAccess {
  // constant members
  None,
  Read  = 1 << 1,
  Write = 1 << 2,
  ReadWrite = Read | Write,
  // computed member
  G = "123".length
}
```

枚举是在运行时真正存在的一个对象。其中一个原因是因为这样就可以从枚举值到枚举名进行反向映射。

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

编译成：

```
var Enum;
(function (Enum) {
  Enum[Enum["A"] = 0] = "A";
})(Enum || (Enum = {}));
var a = Enum.A;
var nameOfA = Enum[Enum.A]; // "A"
```

生成的代码中，枚举类型被编译成一个对象，它包含双向映射（**name->value**）和（**value->name**）。引用枚举成员总会生成一次属性访问并且永远不会内联。在大多数情况下这是很好的并且正确的解决方案。然而有时候需求却比较严格。当访问枚举值时，为了避免生成多余的代码和间接引用，可以使用常数枚举。常数枚举是在 **enum** 关键字前使用 **const** 修饰符。

```
const enum Enum {
  A = 1,
```



```
B = A * 2  
}
```

常数枚举只能使用常数枚举表达式并且不同于常规的枚举的是它们在编译阶段会被删除。常数枚举成员在使用的地方被内联进来。这是因为常数枚举不可能有计算成员。

```
const enum Directions {  
    Up,  
    Down,  
    Left,  
    Right  
}  
  
let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right]
```

生成后的代码为：

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

外部枚举

外部枚举用来描述已经存在的枚举类型的形状。

```
declare enum Enum {  
    A = 1,  
    B,  
    C = 2  
}
```

外部枚举和非外部枚举之间有一个重要的区别，在正常的枚举里，没有初始化方法的成员被当成常数成员。对于非常数的外部枚举而言，没有初始化方法时被当做需要经过计算的。

类型推论

介绍

这节介绍 TypeScript 里的类型推论。即，类型是在哪里如何被推断的。

基础

TypeScript 里，在有些没有明确指出类型的地方，类型推论会帮助提供类型。如下面的例子

```
let x = 3;
```

变量 `x` 的类型被推断为数字。这种推断发生在初始化变量和成员，设置默认参数值和决定函数返回值时。

大多数情况下，类型推论是直截了当地。后面的小节，我们会浏览类型推论时的细微差别。

最佳通用类型

当需要从几个表达式中推断类型时候，会使用这些表达式的类型来推断出一个最合适的通用类型。例如，

```
let x = [0, 1, null];
```

为了推断 `x` 的类型，我们必须考虑所有元素的类型。这里有两种选择：`number` 和 `null`。计算通用类型算法会考虑所有的候选类型，并给出一个兼容所有候选类型的类型。

由于最终的通用类型取自候选类型，有些时候候选类型共享相同的通用类型，但是却没有一个类型能做为所有候选类型的类型。例如：

```
let zoo = [new Rhino(), new Elephant(), new Snake()];
```

这里，我们想让 zoo 被推断为 `Animal[]` 类型，但是这个数组里没有对象是 `Animal` 类型的，因此不能推断出这个结果。为了更正，当候选类型不能使用的时候我们需要明确的指出类型：

```
let zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

如果没有找到最佳通用类型的话，类型推论的结果是空对象类型，`{}`。因为这个类型没有任何成员，所以访问其成员的时候会报错。

上下文类型

TypeScript 类型推论也可能按照相反的方向进行。这被叫做“按上下文归类”。按上下文归类会发生在表达式的类型与所处的位置相关时。比如：

```
window.onmousedown = function(mouseEvent) {  
    console.log(mouseEvent.buton); //<- Error  
};
```

这个例子会得到一个类型错误，TypeScript 类型检查器使用 `Window.onmousedown` 函数的类型来推断右边函数表达式的类型。因此，就能推断出 `mouseEvent` 参数的类型了。如果函数表达式不是在上下文类型的位置，`mouseEvent` 参数的类型需要指定为 `any`，这样也不会报错了。

如果上下文类型表达式包含了明确的类型信息，上下文的类型被忽略。重写上面的例子：

```
window.onmousedown = function(mouseEvent: any) {  
    console.log(mouseEvent.buton); //<- Now, no error is given  
};
```

这个函数表达式有明确的参数类型注解，上下文类型被忽略。这样的话就不报错了，因为这里不会使用到上下文类型。

上下文归类会在很多情况下使用到。通常包含函数的参数，赋值表达式的右边，类型断言，对象成员和数组字面量和返回值语句。上下文类型也会做为最佳通用类型的候选类型。比如：

```
function createZoo(): Animal[] {  
    return [new Rhino(), new Elephant(), new Snake()];  
}
```

这个例子里，最佳通用类型有 4 个候选者：**Animal**，**Rhino**，**Elephant** 和 **Snake**。当然，**Animal** 会被做为最佳通用类型。

类型兼容性

介绍

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。

关于可靠性的注意事项

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。

开始

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) {  
    alert('Hello, ' + n.name);  
}  
greet(y); // OK
```

注意，**y** 有个额外的 **location** 属性，但这不会引发错误。只有目标类型（这里是 **Named**）的成员会被一一检查是否兼容。

这个比较过程是递归进行的，检查每个成员及子成员。

比较两个函数

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。

你可能会疑惑为什么允许忽略参数，像例子 **y = x** 中那样。原因是忽略额外的参数在 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));
```

下面来看看如何处理返回值类型，创建两个仅是返回值类型不同的函数：

```
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 **undefined**。

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。

枚举

枚举类型与数字类型兼容，并且数字类型与枚举类型兼容。不同枚举类型之间是不兼容的。比如，

```
enum Status { Ready, Waiting };  
enum Color { Red, Blue, Green };  
  
let status = Status.Ready;  
status = Color.Green; //error
```

类

类与对象字面量和接口差不多，但有一点不同：类有静态部分和实例部分的类型。比较两个类类型的对象时，只有实例的成员会被比较。静态成员和构造函数不在比较的范围内。


```

class Animal {
    feet: number;
    constructor(name: string, numFeet: number) { }
}

class Size {
    feet: number;
    constructor(numFeet: number) { }
}

let a: Animal;
let s: Size;

a = s; //OK
s = a; //OK

```

类的私有成员

私有成员会影响兼容性判断。当类的实例用来检查兼容时，如果目标类型包含一个私有成员，那么源类型必须包含来自同一个类的这个私有成员。这允许子类赋值给父类，但是不能赋值给其它有同样类型的类。

泛型

因为 TypeScript 是结构性的类型系统，类型参数只影响使用其做为类型一部分的结果类型。比如，

```

interface Empty<T> {
}

let x: Empty<number>;
let y: Empty<string>;

x = y; // okay, y matches structure of x

```

上面代码里，**x** 和 **y** 是兼容的，因为它们的结构使用类型参数时并没有什么不同。把这个例子改变一下，增加一个成员，就能看出是如何工作的了：

```

interface NotEmpty<T> {

```

```

    data: T;
}
let x: NotEmpty<number>;
let y: NotEmpty<string>;

x = y; // error, x and y are not compatible

```

在这里，泛型类型在使用时就好比不是一个泛型类型。

对于没指定泛型类型的泛型参数时，会把所有泛型参数当成 **any** 比较。然后用结果类型进行比较，就像上面第一个例子。

比如，

```

let identity = function<T>(x: T): T {
    // ...
}

let reverse = function<U>(y: U): U {
    // ...
}

identity = reverse; // Okay because (x: any)=>any matches (y: any)=>any

```

高级主题

子类型与赋值

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，有两种类型的兼容性：子类型与赋值。它们的不同点在于，赋值扩展了子类型兼容，允许给 **any** 赋值或从 **any** 取值和允许数字赋值给枚举类型或枚举类型赋值给数字。

语言里的不同地方分别使用了它们之中的机制。实际上，类型兼容性是由赋值兼容性来控制的甚至在 **implements** 和 **extends** 语句里。更多信息，请参阅 [TypeScript 语言规范](#)。

高级类型

交叉类型 (Intersection Types)

交叉类型是将多个类型合并为一个类型。这让我们可以把现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性。例如，`Person & Serializable & Loggable` 同时是 `Person` 和 `Serializable` 和 `Loggable`。就是说这个类型的对象同时拥有了这三种类型的成员。

我们大多是在混入（mixins）或其它不适合典型面向对象模型的地方看到交叉类型的使用。（在 JavaScript 里发生这种情况的场合很多！）下面是如何创建混入的一个简单例子：

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        (<any>result)[id] = (<any>first)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id];
        }
    }
    return result;
}

class Person {
    constructor(public name: string) { }
}

interface Loggable {
    log(): void;
}

class ConsoleLogger implements Loggable {
    log() {
        // ...
    }
}

var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
```

```
jim.log();
```

联合类型 (Union Types)

联合类型与交叉类型很有关联，但是使用上却完全不同。偶尔你会遇到这种情况，一个代码库希望传入 **number** 或 **string** 类型的参数。例如下面的函数：

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: any) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'.`);
}

padLeft("Hello world", 4); // returns "    Hello world"
```

padLeft 存在一个问题，**padding** 参数的类型指定成了 **any**。这就是说我们可以传入一个既不是 **number** 也不是 **string** 类型的参数，但是 TypeScript 却不报错。

```
let indentedString = padLeft("Hello world", true); // 编译阶段通过，运行时报错
```

在传统的面向对象语言里，我们可能会将这两种类型抽象成有层级的类型。这么做显然是非常清晰的，但同时也存在了过度设计。**padLeft** 原始版本的好处之一是允许我们传入原始类型。这样做的话使用起来既简单又方便。如果我们就是想使用已经存在的函数的话，这种新的方式就不适用了。

代替 **any**，我们可以使用联合类型做为 **padding** 的参数：

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
```

```

* If 'padding' is a number, then that number of spaces is added to the left side.
*/
function padLeft(value: string, padding: string | number) {
    // ...
}

let indentedString = padLeft("Hello world", true); // errors during compilation

```

联合类型表示一个值可以是几种类型之一。我们用竖线（|）分隔每个类型，所以 `number | string | boolean` 表示一个值可以是 `number`，`string`，或 `boolean`。

如果一个值是联合类型，我们只能访问此联合类型的所有类型里共有的成员。

```

interface Bird {
    fly();
    layEggs();
}

interface Fish {
    swim();
    layEggs();
}

function getSmallPet(): Fish | Bird {
    // ...
}

let pet = getSmallPet();
pet.layEggs(); // okay
pet.swim();   // errors

```

这里的联合类型可能有点复杂，但是你很容易就习惯了。如果一个值的类型是 `A | B`，我们能够确定的是它包含了 `A` 和 `B` 中共有的成员。这个例子里，`Bird` 具有一个 `fly` 成员。我们不能确定一个 `Bird | Fish` 类型的变量是否有 `fly` 方法。如果变量在运行时是 `Fish` 类型，那么调用 `pet.fly()` 就出错了。

类型保护与区分类型 (Type Guards and Differentiating Types)

联合类型适合于那些值可以为不同类型的情况。但当我们想确切地了解是否为 **Fish** 时怎么办？

JavaScript 里常用来区分 2 个可能值的方法是检查成员是否存在。如之前提及的，我们只能访问联合类型中共同拥有的成员。

```
let pet = getSmallPet();

// 每一个成员访问都会报错
if (pet.swim) {
    pet.swim();
}
else if (pet.fly) {
    pet.fly();
}
```

为了让这段代码工作，我们要使用类型断言：

```
let pet = getSmallPet();

if ((<Fish>pet).swim) {
    (<Fish>pet).swim();
}
else {
    (<Bird>pet).fly();
}
```

用户自定义的类型保护

这里可以注意到我们不得不多次使用类型断言。假若我们一旦检查过类型，就能在之后的每个分支里清楚地知道 **pet** 的类型的話就好了。

TypeScript 里的类型保护机制让它成为了现实。类型保护就是一些表达式，它们会在运行时检查以确保在某个作用域里的类型。要定义一个类型保护，我们只要简单地定义一个函数，它的返回值是一个类型谓词：

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

在这个例子里，`pet is Fish` 就是类型谓词。谓词为 `parameterName is Type` 这种形式，`parameterName` 必须是来自于当前函数签名里的一个参数名。

每当使用一些变量调用 `isFish` 时，TypeScript 会将变量缩减为那个具体的类型，只要这个类型与变量的原始类型是兼容的。

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

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

typeof 类型保护

现在我们回过头来看看怎么使用联合类型书写 `padLeft` 代码。我们可以像下面这样利用类型断言来写：

```
function isNumber(x: any): x is number {  
    return typeof x === "number";  
}  
  
function isString(x: any): x is string {  
    return typeof x === "string";  
}
```

```
function padLeft(value: string, padding: string | number) {
  if (isNumber(padding)) {
    return Array(padding + 1).join(" ") + value;
  }
  if (isString(padding)) {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'.`);
}
```

然而，必须要定义一个函数来判断类型是否是原始类型，这太痛苦了。幸运的是，现在我们可以不必将 `typeof x === "number"` 抽象成一个函数，因为 TypeScript 可以将它识别为一个类型保护。也就是说我们可以直接在代码里检查类型了。

```
function padLeft(value: string, padding: string | number) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'.`);
}
```

这些 *`typeof`* 类型保护只有两种形式能被识别：`typeof v === "typename"` 和 `typeof v !== "typename"`，"typename" 必须是 "number"，"string"，"boolean" 或 "symbol"。但是 TypeScript 并不会阻止你与其它字符串比较，语言不会把那些表达式识别为类型保护。

instanceof 类型保护

如果你已经阅读了 `typeof` 类型保护并且对 JavaScript 里的 `instanceof` 操作符熟悉的话，你可能已经猜到了这节要讲的内容。

`instanceof` 类型保护是通过构造函数来细化类型的一种方式。比如，我们借鉴一下之前字符串填充的例子：

```
interface Padder {
```



```

    getPaddingString(): string
}

class SpaceRepeatingPadder implements Padder {
    constructor(private numSpaces: number) { }
    getPaddingString() {
        return Array(this.numSpaces + 1).join(" ");
    }
}

class StringPadder implements Padder {
    constructor(private value: string) { }
    getPaddingString() {
        return this.value;
    }
}

function getRandomPadder() {
    return Math.random() < 0.5 ?
        new SpaceRepeatingPadder(4) :
        new StringPadder(" ");
}

// 类型为 SpaceRepeatingPadder | StringPadder
let padder: Padder = getRandomPadder();

if (padder instanceof SpaceRepeatingPadder) {
    padder; // 类型细化为'SpaceRepeatingPadder'
}
if (padder instanceof StringPadder) {
    padder; // 类型细化为'StringPadder'
}

```

instanceof 的右侧要求是一个构造函数，TypeScript 将细化为：

1. 此构造函数的 **prototype** 属性的类型，如果它的类型不为 **any** 的话
2. 构造签名所返回的类型的联合

以此顺序。

可以为 null 的类型

TypeScript 具有两种特殊的类型，**null** 和 **undefined**，它们分别具有值 `null` 和 `undefined`。我们在[基础类型](#)一节里已经做过简要说明。默认情况下，类型检查器认为 **null** 与 **undefined** 可以赋值给任何类型。**null** 与 **undefined** 是所有其它类型的一个有效值。这也意味着，你阻止不了将它们赋值给其它类型，就算是你想要阻止这种情况也不行。**null** 的发明者，Tony Hoare，称它为[价值亿万美金的错误](#)。

--strictNullChecks 标记可以解决此错误：当你声明一个变量时，它不会自动地包含 **null** 或 **undefined**。你可以使用联合类型明确的包含它们：

```
let s = "foo";
s = null; // 错误, 'null'不能赋值给'string'
let sn: string | null = "bar";
sn = null; // 可以

sn = undefined; // error, 'undefined'不能赋值给'string | null'
```

注意，按照 JavaScript 的语义，TypeScript 会把 **null** 和 **undefined** 区别对待。**string | null**，**string | undefined** 和 **string | undefined | null** 是不同的类型。

可选参数和可选属性

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

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

可选属性也会有同样的处理：

```
class C {
```

```

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

```

类型保护和类型断言

由于可以为 `null` 的类型是通过联合类型实现，那么你需要使用类型保护来去除 `null`。幸运地是这与在 JavaScript 里写的代码一致：

```

function f(sn: string | null): string {
    if (sn == null) {
        return "default";
    }
    else {
        return sn;
    }
}

```

这里很明显地去除了 `null`，你也可以使用短路运算符：

```

function f(sn: string | null): string {
    return sn || "default";
}

```

如果编译器不能够去除 `null` 或 `undefined`，你可以使用类型断言手动去除。语法是添加 `!` 后缀：
`identifier!` 从 `identifier` 的类型里去除了 `null` 和 `undefined`：

```

function broken(name: string | null): string {
    function postfix(epithet: string) {
        return name.charAt(0) + ' the ' + epithet; // error, 'name' is possibly null
    }
    name = name || "Bob";
    return postfix("great");
}

```

```

}

function fixed(name: string | null): string {
  function postfix(epithet: string) {
    return name!.charAt(0) + '. the ' + epithet; // ok
  }
  name = name || "Bob";
  return postfix("great");
}

```

本例使用了嵌套函数，因为编译器无法去除嵌套函数的 `null`（除非是立即调用的函数表达式）。因为它无法跟踪所有对嵌套函数的调用，尤其是你将内层函数做为外层函数的返回值。如果无法知道函数在哪里被调用，就无法知道调用时 `name` 的类型。

类型别名

类型别名会给一个类型起个新名字。类型别名有时和接口很像，但是可以作用于原始值，联合类型，元组以及其它任何你需要手写的类型。

```

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

```

起别名不会新建一个类型 - 它创建了一个新名字来引用那个类型。给原始类型起别名通常没什么用，尽管可以作为文档的一种形式使用。

同接口一样，类型别名也可以是泛型 - 我们可以添加类型参数并且在别名声明的右侧传入：

```

type Container<T> = { value: T };

```

我们也可以使用类型别名来在属性里引用自己：

```
type Tree<T> = {  
    value: T;  
    left: Tree<T>;  
    right: Tree<T>;  
}
```

与交叉类型一起使用，我们可以创建出一些十分稀奇古怪的类型。

```
type LinkedList<T> = T & { next: LinkedList<T> };  
  
interface Person {  
    name: string;  
}  
  
var people: LinkedList<Person>;  
var s = people.name;  
var s = people.next.name;  
var s = people.next.next.name;  
var s = people.next.next.next.name;
```

然而，类型别名不能出现在声明右侧的任何地方。

```
type Yikes = Array<Yikes>; // error
```

接口 vs. 类型别名

像我们提到的，类型别名可以像接口一样；然而，仍有一些细微差别。

其一，接口创建了一个新的名字，可以在其它任何地方使用。类型别名并不创建新名字—比如，错误信息就不会使用别名。在下面的示例代码里，在编译器中将鼠标悬停在 **interfaced** 上，显示它返回的是 **Interface**，但悬停在 **aliased** 上时，显示的却是对象字面量类型。

```
type Alias = { num: number }  
interface Interface {  
    num: number;  
}  
  
declare function aliased(arg: Alias): Alias;
```

```
declare function interfaced(arg: Interface): Interface;
```

另一个重要区别是类型别名不能被 **extends** 和 **implements**（自己也不能 **extends** 和 **implements** 其它类型）。因为软件中的对象应该对于扩展是开放的，但是对于修改是封闭的，你应该尽量去使用接口代替类型别名。

另一方面，如果你无法通过接口来描述一个类型并且需要使用联合类型或元组类型，这时通常会使用类型别名。

字符串字面量类型

字符串字面量类型允许你指定字符串必须的固定值。在实际应用中，字符串字面量类型可以与联合类型，类型保护和类型别名很好的配合。通过结合使用这些特性，你可以实现类似枚举类型的字符串。

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
  animate(dx: number, dy: number, easing: Easing) {
    if (easing === "ease-in") {
      // ...
    }
    else if (easing === "ease-out") {
    }
    else if (easing === "ease-in-out") {
    }
    else {
      // error! should not pass null or undefined.
    }
  }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // error: "uneasy" is not allowed here
```

你只能从三种允许的字符中选择其一来做为参数传递，传入其它值则会产生错误。

```
Argument of type '"uneasy"' is not assignable to parameter of type '"ease-in" | "ease-out" | "ease-in-out"'
```

字符串字面量类型还可以用于区分函数重载：

```
function createElement(tagName: "img"): HTMLImageElement;
function createElement(tagName: "input"): HTMLInputElement;
// ... more overloads ...
function createElement(tagName: string): Element {
    // ... code goes here ...
}
```

可辨识联合 (Discriminated Unions)

你可以合并字符串字面量类型，联合类型，类型保护和类型别名来创建一个叫做 *可辨识联合* 的高级模式，它也称做 *标签联合* 或 *代数数据类型*。可辨识联合在函数式编程很有用处。一些语言会自动地为你辨识联合；而 TypeScript 则基于已有的 JavaScript 模式。它具有 3 个要素：

1. 具有普通的字符串字面量属性 — *可辨识的特征*。
2. 一个类型别名包含了那些类型的联合 — *联合*。
3. 此属性上的类型保护。

```
interface Square {
    kind: "square";
    size: number;
}
interface Rectangle {
    kind: "rectangle";
    width: number;
    height: number;
}
interface Circle {
    kind: "circle";
    radius: number;
}
```

首先我们声明了将要联合的接口。每个接口都有 `kind` 属性但有不同的字符串字面量类型。`kind` 属性称做可辨识的特征或标签。其它的属性则特定于各个接口。注意，目前各个接口间是没有联系的。下面我们把它们联合到一起：

```
type Shape = Square | Rectangle | Circle;
```

现在我们使用可辨识联合：

```
function area(s: Shape) {
  switch (s.kind) {
    case "square": return s.size * s.size;
    case "rectangle": return s.height * s.width;
    case "circle": return Math.PI * s.radius ** 2;
  }
}
```

完整性检查

当没有涵盖所有可辨识联合的变化时，我们想让编译器可以通知我们。比如，如果我们添加了 `Triangle` 到 `Shape`，我们同时还需要更新 `area`：

```
type Shape = Square | Rectangle | Circle | Triangle;
function area(s: Shape) {
  switch (s.kind) {
    case "square": return s.size * s.size;
    case "rectangle": return s.height * s.width;
    case "circle": return Math.PI * s.radius ** 2;
  }
  // should error here - we didn't handle case "triangle"
}
```

有两种方式可以实现。首先是启用 `--strictNullChecks` 并且指定一个返回值类型：

```
function area(s: Shape): number { // error: returns number | undefined
  switch (s.kind) {
    case "square": return s.size * s.size;
    case "rectangle": return s.height * s.width;
    case "circle": return Math.PI * s.radius ** 2;
  }
}
```



```
}
```

因为 **switch** 没有包涵所有情况，所以 TypeScript 认为这个函数有时候会返回 **undefined**。如果你明确地指定了返回值类型为 **number**，那么你会看到一个错误，因为实际上返回值的类型为 **number | undefined**。然而，这种方法存在些微妙之处且 **--strictNullChecks** 对旧代码支持不好。

第二种方法使用 **never** 类型，编译器用它来进行完整性检查：

```
function assertNever(x: never): never {  
    throw new Error("Unexpected object: " + x);  
}  
  
function area(s: Shape) {  
    switch (s.kind) {  
        case "square": return s.size * s.size;  
        case "rectangle": return s.height * s.width;  
        case "circle": return Math.PI * s.radius ** 2;  
        default: return assertNever(s); // error here if there are missing cases  
    }  
}
```

这里，**assertNever** 检查 **s** 是否为 **never** 类型—即为除去所有可能情况后剩下的类型。如果你忘记了某个 case，那么 **s** 将具有一个真实的类型并且你会得到一个错误。这种方式需要你定义一个额外的函数，但是在你忘记某个 case 的时候也更加明显。

多态的 **this** 类型

多态的 **this** 类型表示的是某个包含类或接口的子类型。这被称做 *F*-bounded 多态性。它能很容易的表现连贯接口间的继承，比如。在计算器的例子里，在每个操作之后都返回 **this** 类型：

```
class BasicCalculator {  
    public constructor(protected value: number = 0) { }  
    public currentValue(): number {  
        return this.value;  
    }  
    public add(operand: number): this {  
        this.value += operand;  
        return this;  
    }  
}
```

```

    }
    public multiply(operand: number): this {
        this.value *= operand;
        return this;
    }
    // ... other operations go here ...
}

let v = new BasicCalculator(2)
    .multiply(5)
    .add(1)
    .currentValue();

```

由于这个类使用了 **this** 类型，你可以继承它，新的类可以直接使用之前的方法，不需要做任何的改变。

```

class ScientificCalculator extends BasicCalculator {
    public constructor(value = 0) {
        super(value);
    }
    public sin() {
        this.value = Math.sin(this.value);
        return this;
    }
    // ... other operations go here ...
}

let v = new ScientificCalculator(2)
    .multiply(5)
    .sin()
    .add(1)
    .currentValue();

```

如果没有 **this** 类型，**ScientificCalculator** 就不能够在继承 **BasicCalculator** 的同时还保持接口的连贯性。**multiply** 将会返回 **BasicCalculator**，它并没有 **sin** 方法。然而，使用 **this** 类型，**multiply** 会返回 **this**，在这里就是 **ScientificCalculator**。

索引类型 (Index types)

使用索引类型，编译器就能够检查使用了动态属性名的代码。例如，一个常见的 JavaScript 模式是从对象中选取属性的子集。

```
function pluck(o, names) {  
    return names.map(n => o[n]);  
}
```

下面是如何在 TypeScript 里使用此函数，通过索引类型查询和索引访问操作符：

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

编译器会检查 `name` 是否真的是 `Person` 的一个属性。本例还引入了几个新的类型操作符。首先是 `keyof T`，索引类型查询操作符。对于任何类型 `T`，`keyof T` 的结果为 `T` 上已知的公共属性名的联合。例如：

```
let personProps: keyof Person; // 'name' | 'age'
```

`keyof Person` 是完全可以与 `'name' | 'age'` 互相替换的。不同的是如果你添加了其它的属性到 `Person`，例如 `address: string`，那么 `keyof Person` 会自动变为 `'name' | 'age' | 'address'`。你可以在像 `pluck` 函数这类上下文里使用 `keyof`，因为在使用之前你并不清楚可能出现的属性名。但编译器会检查你是否传入了正确的属性名给 `pluck`：

```
pluck(person, ['age', 'unknown']); // error, 'unknown' is not in 'name' | 'age'
```

第二个操作符是 **T[K]**，索引访问操作符。在这里，类型语法反映了表达式语法。这意味着 **person['name']** 具有类型 **Person['name']**— 在我们的例子里则为 **string** 类型。然而，就像索引类型查询一样，你可以在普通的上下文里使用 **T[K]**，这正是它的强大所在。你只要确保类型变量 **K extends keyof T** 就可以了。例如下面 **getProperty** 函数的例子：

```
function getProperty<T, K extends keyof T>(o: T, name: K): T[K] {  
    return o[name]; // o[name] is of type T[K]  
}
```

getProperty 里的 **o: T** 和 **name: K**，意味着 **o[name]: T[K]**。当你返回 **T[K]** 的结果，编译器会实例化键的真实类型，因此 **getProperty** 的返回值类型会随着你需要的属性改变。

```
let name: string = getProperty(person, 'name');  
let age: number = getProperty(person, 'age');  
let unknown = getProperty(person, 'unknown'); // error, 'unknown' is not in 'name' | 'age'
```

索引类型和字符串索引签名

keyof 和 **T[K]** 与字符串索引签名进行交互。如果你有一个带有字符串索引签名的类型，那么 **keyof T** 会是 **string**。并且 **T[string]** 为索引签名的类型：

```
interface Map<T> {  
    [key: string]: T;  
}  
let keys: keyof Map<number>; // string  
let value: Map<number>['foo']; // number
```

映射类型

一个常见的任务是将一个已知的类型每个属性都变为可选的：

```
interface PersonPartial {  
    name?: string;  
    age?: number;  
}
```

或者我们想要一个只读版本：

```
interface PersonReadOnly {  
    readonly name: string;  
    readonly age: number;  
}
```

这在 JavaScript 里经常出现，TypeScript 提供了从旧类型中创建新类型的一种方式 — **映射类型**。在映射类型里，新类型以相同的形式去转换旧类型里每个属性。例如，你可以令每个属性成为 **readonly** 类型或可选的。下面是一些例子：

```
type Readonly<T> = {  
    readonly [P in keyof T]: T[P];  
}  
type Partial<T> = {  
    [P in keyof T]?: T[P];  
}
```

像下面这样使用：

```
type PersonPartial = Partial<Person>;  
type ReadonlyPerson = Readonly<Person>;
```

下面来看看最简单的映射类型和它的组成部分：

```
type Keys = 'option1' | 'option2';  
type Flags = { [K in Keys]: boolean };
```

它的语法与索引签名的语法类型，内部使用了 **for .. in**。具有三个部分：

1. 类型变量 **K**，它会依次绑定到每个属性。
2. 字符串字面量联合的 **Keys**，它包含了要迭代的属性名的集合。
3. 属性的结果类型。

在个简单的例子里，**Keys** 是硬编码的的属性名列表并且属性类型永远是 **boolean**，因此这个映射类型等同于：

```
type Flags = {  
    option1: boolean;  
    option2: boolean;  
}
```

在真正的应用里，可能不同于上面的 **Readonly** 或 **Partial**。它们会基于一些已存在的类型，且按照一定的方式转换字段。这就是 **keyof** 和索引访问类型要做的事情：

```
type NullablePerson = { [P in keyof Person]: Person[P] | null }
type PartialPerson = { [P in keyof Person]?: Person[P] }
```

但它更有用的地方是可以有一些通用版本。

```
type Nullable<T> = { [P in keyof T]: T[P] | null }
type Partial<T> = { [P in keyof T]?: T[P] }
```

在这些例子里，属性列表是 **keyof T** 且结果类型是 **T[P]** 的变体。这是使用通用映射类型的一个好模版。因为这类转换是**同态**的，映射只作用于 **T** 的属性而没有其它的。编译器知道在添加任何新属性之前可以拷贝所有存在的属性修饰符。例如，假设 **Person.name** 是只读的，那么 **Partial<Person>.name** 也将是只读的且为可选的。

下面是另一个例子，**T[P]**被包装在 **Proxy<T>** 类里：

```
type Proxy<T> = {
  get(): T;
  set(value: T): void;
}
type Proxify<T> = {
  [P in keyof T]: Proxy<T[P]>;
}
function proxify<T>(o: T): Proxify<T> {
  // ... wrap proxies ...
}
let proxyProps = proxify(props);
```

注意 **Readonly<T>** 和 **Partial<T>** 用处不小，因此它们与 **Pick** 和 **Record** 一同被包含进了 TypeScript 的标准库里：

```
type Pick<T, K extends keyof T> = {
  [P in K]: T[P];
}
type Record<K extends string, T> = {
  [P in K]: T;
}
```

Readonly，**Partial** 和 **Pick** 是同态的，但 **Record** 不是。因为 **Record** 并不需要输入类型来拷贝属性，所以它不属于同态：

```
type ThreeStringProps = Record<'prop1' | 'prop2' | 'prop3', string>
```

非同态类型本质上会创建新的属性，因此它们不会从它处拷贝属性修饰符。

由映射类型进行推断

现在你了解了如何包装一个类型的属性，那么接下来就是如果拆包。其实这也非常容易：

```
function unproxify<T>(t: Proxify<T>): T {
    let result = {} as T;
    for (const k in t) {
        result[k] = t[k].get();
    }
    return result;
}

let originalProps = unproxify(proxyProps);
```

注意这个拆包推断只适用于同态的映射类型。如果映射类型不是同态的，那么需要给拆包函数一个明确的类型参数。

Symbols

介绍

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 {
  [classNamesymbol]() {
    return "C";
  }
}

let c = new C();
let className = c[classNamesymbol]; // "C"
```

众所周知的 Symbols

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。正则表达式用来分割字符串。

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。

迭代器和生成器

可迭代性

当一个对象实现了 `Symbol.iterator` 属性时，我们认为它是可迭代的。一些内置的类型如 `Array`，`Map`，`Set`，`String`，`Int32Array`，`Uint32Array` 等都已经实现了各自的 `Symbol.iterator`。对象上的 `Symbol.iterator` 函数负责返回供迭代的值。

for..of 语句

`for..of` 会遍历可迭代的对象，调用对象上的 `Symbol.iterator` 方法。下面是在数组上使用 `for..of` 的简单例子：

```
let someArray = [1, "string", false];

for (let entry of someArray) {
  console.log(entry); // 1, "string", false
}
```

for..of VS. for..in 语句

for..of 和 **for..in** 均可迭代一个列表；但是用于迭代的值却不同，**for..in** 迭代的是对象的键的列表，而 **for..of** 则迭代对象的键对应的值。

下面的例子展示了两者之间的区别：

```
let list = [4, 5, 6];

for (let i in list) {
    console.log(i); // "0", "1", "2",
}

for (let i of list) {
    console.log(i); // "4", "5", "6"
}
```

另一个区别是 **for..in** 可以操作任何对象；它提供了查看对象属性的一种方法。但是 **for..of** 关注于迭代对象的值。内置对象 **Map** 和 **Set** 已经实现了 **Symbol.iterator** 方法，让我们可以访问它们保存的值。

```
let pets = new Set(["Cat", "Dog", "Hamster"]);
pets["species"] = "mammals";

for (let pet in pets) {
    console.log(pet); // "species"
}

for (let pet of pets) {
    console.log(pet); // "Cat", "Dog", "Hamster"
}
```

代码生成

目标为 ES5 和 ES3

当生成目标为 ES5 或 ES3，迭代器只允许在 **Array** 类型上使用。在非数组值上使用 **for..of** 语句会得到一个错误，就算这些非数组值已经实现了 **Symbol.iterator** 属性。

编译器会生成一个简单的 `for` 循环做为 `for..of` 循环，比如：

```
let numbers = [1, 2, 3];
for (let num of numbers) {
  console.log(num);
}
```

生成的代码为：

```
var numbers = [1, 2, 3];
for (var _i = 0; _i < numbers.length; _i++) {
  var num = numbers[_i];
  console.log(num);
}
```

目标为 *ECMAScript 2015 或更高*

当目标为兼容 ECMAScript 2015 的引擎时，编译器会生成相应引擎的 `for..of` 内置迭代器实现方式。

模块

关于术语的一点说明:请务必注意一点，TypeScript 1.5 里术语名已经发生了变化。“内部模块”现在称做“命名空间”。“外部模块”现在则简称为“模块”，这是为了与 *ECMAScript 2015* 里的术语保持一致，(也就是说 `module X {` 相当于现在推荐的写法 `namespace X {`)。

介绍

从 ECMAScript 2015 开始，JavaScript 引入了模块的概念。TypeScript 也沿用这个概念。

模块在其自身的作用域里执行，而不是在全局作用域里；这意味着定义在一个模块里的变量，函数，类等等在模块外部是不可见的，除非你明确地使用 `export` 形式之一导出它们。相反，如果想使用其它模块导出的变量，函数，类，接口等的时候，你必须要导入它们，可以使用 `import` 形式之一。

模块是自声明的；两个模块之间的关系是通过在文件级别上使用 `imports` 和 `exports` 建立的。

模块使用模块加载器去导入其它的模块。在运行时，模块加载器的作用是在执行此模块代码前去查找并执行这个模块的所有依赖。大家最熟知的 JavaScript 模块加载器是服务于 Node.js 的 [CommonJS](#) 和服务于 Web 应用的 [Require.js](#)。

TypeScript 与 ECMAScript 2015 一样，任何包含顶级 **import** 或者 **export** 的文件都被当成一个模块。

导出

导出声明

任何声明（比如变量，函数，类，类型别名或接口）都能够通过添加 **export** 关键字来导出。

Validation.ts

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

ZipCodeValidator.ts

```
export const numberRegex = /^[0-9]+$/;  
  
export class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && numberRegex.test(s);  
    }  
}
```

导出语句

导出语句很便利，因为我们可能需要对导出的部分重命名，所以上面的例子可以这样改写：

```
class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && numberRegex.test(s);  
    }  
}
```

```
export { ZipCodeValidator };  
export { ZipCodeValidator as mainValidator };
```

重新导出

我们经常会去扩展其它模块，并且只导出那个模块的部分内容。重新导出功能并不会在当前模块导入那个模块或定义一个新的局部变量。

ParseIntBasedZipCodeValidator.ts

```
export class ParseIntBasedZipCodeValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && parseInt(s).toString() === s;  
    }  
}  
  
// 导出原先的验证器但做了重命名  
export { ZipCodeValidator as RegExpBasedZipCodeValidator } from "../ZipCodeValidator";
```

或者一个模块可以包裹多个模块，并把他们导出的内容联合在一起通过语法：**export * from "module"**。

AllValidators.ts

```
export * from "../StringValidator"; // exports interface StringValidator  
export * from "../LettersOnlyValidator"; // exports class LettersOnlyValidator  
export * from "../ZipCodeValidator"; // exports class ZipCodeValidator
```

导入

模块的导入操作与导出一样简单。可以使用以下 **import** 形式之一来导入其它模块中的导出内容。

导入一个模块中的某个导出内容

```
import { ZipCodeValidator } from "../ZipCodeValidator";  
  
let myValidator = new ZipCodeValidator();
```

可以对导入内容重命名

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";  
let myValidator = new ZCV();
```

将整个模块导入到一个变量，并通过它来访问模块的导出部分

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

具有副作用的导入模块

尽管不推荐这么做，一些模块会设置一些全局状态供其它模块使用。这些模块可能没有任何的导出或用户根本就不关注它的导出。使用下面的方法来导入这类模块：

```
import "./my-module.js";
```

默认导出

每个模块都可以有一个 **default** 导出。默认导出使用 **default** 关键字标记；并且一个模块只能够有一个 **default** 导出。需要使用一种特殊的导入形式来导入 **default** 导出。

default 导出十分便利。比如，像 JQuery 这样的类库可能有一个默认导出 **jQuery** 或 **\$**，并且我们基本上也会使用同样的名字 **jQuery** 或 **\$** 导出 JQuery。

JQuery.d.ts

```
declare let $: JQuery;  
export default $;
```

App.ts

```
import $ from "jQuery";  
  
$("button.continue").html( "Next Step..." );
```

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

ZipCodeValidator.ts

```
export default class ZipCodeValidator {
  static numberRegexp = /^[0-9]+$/;
  isAcceptable(s: string) {
    return s.length === 5 && ZipCodeValidator.numberRegexp.test(s);
  }
}
```

Test.ts

```
import validator from "./ZipCodeValidator";

let myValidator = new validator();
```

或者

StaticZipCodeValidator.ts

```
const numberRegexp = /^[0-9]+$/;

export default function (s: string) {
  return s.length === 5 && numberRegexp.test(s);
}
```

Test.ts

```
import validate from "./StaticZipCodeValidator";

let strings = ["Hello", "98052", "101"];

// Use function validate
strings.forEach(s => {
  console.log(`${s} ${validate(s) ? " matches" : " does not match"}`);
});
```

default 导出也可以是一个值

OneTwoThree.ts

```
export default "123";
```

Log.ts

```
import num from "./OneTwoThree";
```



```
console.log(num); // "123"
```

export = 和 import = require()

CommonJS 和 AMD 都有一个 **exports** 对象的概念，它包含了一个模块的所有导出内容。

它们也支持把 **exports** 替换为一个自定义对象。默认导出就好比这样一个功能；然而，它们却并不相互兼容。TypeScript 模块支持 **export =** 语法以支持传统的 CommonJS 和 AMD 的工作流模型。

export = 语法定义一个模块的导出对象。它可以是类，接口，命名空间，函数或枚举。

若要导入一个使用了 **export =** 的模块时，必须使用 TypeScript 提供的特定语法 **import module = require("module")**。

ZipCodeValidator.ts

```
let numberRegexp = /^[0-9]+$/;
class ZipCodeValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
export = ZipCodeValidator;
```

Test.ts

```
import zip = require("./ZipCodeValidator");

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validator = new zip();

// Show whether each string passed each validator
strings.forEach(s => {
  console.log(`"${s}" - ${validator.isAcceptable(s) ? "matches" : "does not match"}`);
});
```

生成模块代码

根据编译时指定的模块目标参数，编译器会生成相应的供 Node.js ([CommonJS](#)), [Require.js](#) ([AMD](#)), [isomorphic](#) ([UMD](#)), [SystemJS](#) 或 [ECMAScript 2015 native modules](#) (ES6) 模块加载系统使用的代码。想要了解生成代码中 **define**, **require** 和 **register** 的意义，请参考相应模块加载器的文档。

下面的例子说明了导入导出语句里使用的名字是怎么转换为相应的模块加载器代码的。

SimpleModule.ts

```
import m = require("mod");
export let t = m.something + 1;
```

AMD / RequireJS SimpleModule.js

```
define(["require", "exports", "./mod"], function (require, exports, mod_1) {
    exports.t = mod_1.something + 1;
});
```

CommonJS / Node SimpleModule.js

```
let mod_1 = require("./mod");
exports.t = mod_1.something + 1;
```

UMD SimpleModule.js

```
(function (factory) {
    if (typeof module === "object" && typeof module.exports === "object") {
        let v = factory(require, exports); if (v !== undefined) module.exports = v;
    }
    else if (typeof define === "function" && define.amd) {
        define(["require", "exports", "./mod"], factory);
    }
})(function (require, exports) {
    let mod_1 = require("./mod");
    exports.t = mod_1.something + 1;
});
```

System SimpleModule.js

```
System.register(["./mod"], function(exports_1) {
    let mod_1;
```

```

    let t;
    return {
      setters:[
        function (mod_1_1) {
          mod_1 = mod_1_1;
        },
        execute: function() {
          exports_1("t", t = mod_1.something + 1);
        }
      ]
    }
  });

```

Native ECMAScript 2015 modules SimpleModule.js

```

import { something } from "./mod";
export let t = something + 1;

```

简单示例

下面我们来整理一下前面的验证器实现，每个模块只有一个命名的导出。

为了编译，我们必需要在命令行上指定一个模块目标。对于 Node.js 来说，使用 `--module commonjs`；对于 Require.js 来说，使用 `--module amd`。比如：

```
tsc --module commonjs Test.ts
```

编译完成后，每个模块会生成一个单独的 `.js` 文件。好比使用了 `reference` 标签，编译器会根据 `import` 语句编译相应的文件。

Validation.ts

```

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

```

LettersOnlyValidator.ts

```

import { StringValidator } from "./Validation";

const lettersRegexp = /^[A-Za-z]+$/;

```

```
export class LettersOnlyValidator implements StringValidator {
  isAcceptable(s: string) {
    return lettersRegexp.test(s);
  }
}
```

ZipCodeValidator.ts

```
import { StringValidator } from "./Validation";

const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
```

Test.ts

```
import { StringValidator } from "./Validation";
import { ZipCodeValidator } from "./ZipCodeValidator";
import { LettersOnlyValidator } from "./LettersOnlyValidator";

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
strings.forEach(s => {
  for (let name in validators) {
    console.log(`"${s}" - ${validators[name].isAcceptable(s) ? "matches" : "does not match"} ${name}`);
  }
});
```

可选的模块加载和其它高级加载场景

有时候，你只想在某种条件下才加载某个模块。在 TypeScript 里，使用下面的方式来实现它和其它的高级加载场景，我们可以直接调用模块加载器并且可以保证类型完全。

编译器会检测是否每个模块都会在生成的 JavaScript 中用到。如果一个模块标识符只在类型注解部分使用，并且完全没有在表达式中使用，就不会生成 `require` 这个模块的代码。省略掉没有用到的引用对性能提升是很有益的，并同时提供了选择性加载模块的能力。

这种模式的核心是 `import id = require("...")` 语句可以让我们访问模块导出的类型。模块加载器会被动调用（通过 `require`），就像下面 `if` 代码块里那样。它利用了省略引用的优化，所以模块只在被需要时加载。为了让这个模块工作，一定要注意 `import` 定义的标识符只能在表示类型处使用（不能在会转换成 JavaScript 的地方）。

为了确保类型安全性，我们可以使用 `typeof` 关键字。`typeof` 关键字，当在表示类型的地方使用时，会得出一个类型值，这里就表示模块的类型。

示例：Node.js 里的动态模块加载

```
declare function require(moduleName: string): any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    let ZipCodeValidator: typeof Zip = require("./ZipCodeValidator");
    let validator = new ZipCodeValidator();
    if (validator.isAcceptable("...")) { /* ... */ }
}
```

示例：require.js 里的动态模块加载

```
declare function require(moduleNames: string[], onLoad: (...args: any[]) => void): void;

import * as Zip from "./ZipCodeValidator";

if (needZipValidation) {
    require(["./ZipCodeValidator"], (ZipCodeValidator: typeof Zip) => {
```

```

    let validator = new ZipCodeValidator.ZipCodeValidator();
    if (validator.isAcceptable("...")) { /* ... */ }
  });
}

```

示例: *System.js* 里的动态模块加载

```

declare const System: any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
  System.import("./ZipCodeValidator").then((ZipCodeValidator: typeof Zip) => {
    var x = new ZipCodeValidator();
    if (x.isAcceptable("...")) { /* ... */ }
  });
}

```

使用其它的 JavaScript 库

要想描述非 TypeScript 编写的类库的类型，我们需要声明类库所暴露出的 API。

我们叫它声明因为它不是“外部程序”的具体实现。它们通常是在 *.d.ts* 文件里定义的。如果你熟悉 C/C++，你可以把它们当做 *.h* 文件。让我们看一些例子。

外部模块

在 Node.js 里大部分工作是通过加载一个或多个模块实现的。我们可以使用顶级的 *export* 声明来为每个模块都定义一个 *.d.ts* 文件，但最好还是写在一个大的 *.d.ts* 文件里。我们使用与构造一个外部命名空间相似的方法，但是这里使用 *module* 关键字并且把名字用引号括起来，方便之后 *import*。例如：

node.d.ts (simplified excerpt)

```

declare module "url" {
  export interface Url {
    protocol?: string;
    hostname?: string;
  }
}

```

```

    pathname?: string;
  }

  export function parse(urlStr: string, parseQueryString?, slashesDenoteHost?): Url;
}

declare module "path" {
  export function normalize(p: string): string;
  export function join(...paths: any[]): string;
  export let sep: string;
}

```

现在我们可以`/// <reference>node.d.ts` 并且使用 `import url = require("url");`或 `import * as URL from "url"`加载模块。

```

/// <reference path="node.d.ts"/>
import * as URL from "url";
let myUrl = URL.parse("http://www.typescriptlang.org");

```

外部模块简写

假如你不想在使用一个新模块之前花时间去编写声明，你可以采用声明的简写形式以便能够快速使用它。

declarations.d.ts

```
declare module "hot-new-module";
```

简写模块里所有导出的类型将是 `any`。

```

import x, {y} from "hot-new-module";
x(y);

```

模块声明通配符

某些模块加载器如 `SystemJS` 和 `AMD` 支持导入非 JavaScript 内容。它们通常会使用一个前缀或后缀来表示特殊的加载语法。模块声明通配符可以用来表示这些情况。

```

declare module ".*!text" {
  const content: string;
  export default content;
}

```

```

}
// Some do it the other way around.
declare module "json!*" {
  const value: any;
  export default value;
}

```

现在你可以就导入匹配`"*!text"`或`"json!*"`的内容了。

```

import fileContent from "./xyz.txt!text";
import data from "json!http://example.com/data.json";
console.log(data, fileContent);

```

UMD 模块

有些模块被设计成兼容多个模块加载器，或者不使用模块加载器（全局变量）。它们以 [UMD](#) 或 [Isomorphic](#) 模块为代表。这些库可以通过导入的形式或全局变量的形式访问。例如：

math-lib.d.ts

```

export const isPrime(x: number): boolean;
export as namespace mathLib;

```

之后，这个库可以在某个模块里通过导入来使用：

```

import { isPrime } from "math-lib";
isPrime(2);
mathLib.isPrime(2); // ERROR: can't use the global definition from inside a module

```

它同样可以通过全局变量的形式使用，但只能在某个脚本里。（脚本是指一个不带有导入或导出的文件。）

```

mathLib.isPrime(2);

```


创建模块结构指导

尽可能地在顶层导出

用户应该更容易地使用你模块导出的内容。嵌套层次过多会变得难以处理，因此仔细考虑一下如何组织你的代码。

从你的模块中导出一个命名空间就是一个增加嵌套的例子。虽然命名空间有时候有它们的用处，在使用模块的时候它们额外地增加了一层。这对用户来说是很不便的并且通常是多余的。

导出类的静态方法也有同样的问题 - 这个类本身就增加了一层嵌套。除非它能方便表述或便于清晰使用，否则请考虑直接导出一个辅助方法。

如果仅导出单个 *class* 或 *function*，使用 *export default*

就像“在顶层上导出”帮助减少用户使用的难度，一个默认的导出也能起到这个效果。如果一个模块就是为了导出特定的内容，那么你应该考虑使用一个默认导出。这会令模块的导入和使用变得些许简单。比如：

MyClass.ts

```
export default class SomeType {  
  constructor() { ... }  
}
```

MyFunc.ts

```
export default function getThing() { return 'thing'; }
```

Consumer.ts

```
import t from "./MyClass";  
import f from "./MyFunc";  
let x = new t();  
console.log(f());
```

对用户来说这是最理想的。他们可以随意命名导入模块的类型（本例为 `t`）并且不需要多余的 `(.)` 来找到相关对象。

如果要导出多个对象，把它们放在顶层里导出

MyThings.ts

```
export class SomeType { /* ... */ }  
export function someFunc() { /* ... */ }
```

相反地，当导入的时候：

明确地列出导入的名字

Consumer.ts

```
import { SomeType, SomeFunc } from "./MyThings";  
let x = new SomeType();  
let y = someFunc();
```

使用命名空间导入模式当你要导出大量内容的时候

MyLargeModule.ts

```
export class Dog { ... }  
export class Cat { ... }  
export class Tree { ... }  
export class Flower { ... }
```

Consumer.ts

```
import * as myLargeModule from "./MyLargeModule.ts";  
let x = new myLargeModule.Dog();
```

使用重新导出进行扩展

你可能经常需要去扩展一个模块的功能。JS 里常用的一个模式是 JQuery 那样去扩展原对象。如我们之前提到的，模块不会像全局命名空间对象那样去合并。推荐的方案是不要去改变原来的对象，而是导出一个新的实体来提供新的功能。

假设 **Calculator.ts** 模块里定义了一个简单的计算器实现。这个模块同样提供了一个辅助函数来测试计算器的功能，通过传入一系列输入的字符串并在最后给出结果。

Calculator.ts

```
export class Calculator {
    private current = 0;
    private memory = 0;
    private operator: string;

    protected processDigit(digit: string, currentValue: number) {
        if (digit >= "0" && digit <= "9") {
            return currentValue * 10 + (digit.charCodeAt(0) - "0".charCodeAt(0));
        }
    }

    protected processOperator(operator: string) {
        if (["+","-", "*", "/"].indexOf(operator) >= 0) {
            return operator;
        }
    }

    protected evaluateOperator(operator: string, left: number, right: number): number {
        switch (this.operator) {
            case "+": return left + right;
            case "-": return left - right;
            case "*": return left * right;
            case "/": return left / right;
        }
    }

    private evaluate() {
        if (this.operator) {
            this.memory = this.evaluateOperator(this.operator, this.memory, this.current);
        }
        else {
            this.memory = this.current;
        }
        this.current = 0;
    }
}
```

```

public handelChar(char: string) {
  if (char === "=") {
    this.evaluate();
    return;
  }
  else {
    let value = this.processDigit(char, this.current);
    if (value !== undefined) {
      this.current = value;
      return;
    }
    else {
      let value = this.processOperator(char);
      if (value !== undefined) {
        this.evaluate();
        this.operator = value;
        return;
      }
    }
  }
  throw new Error(`Unsupported input: '${char}'`);
}

public getResult() {
  return this.memory;
}
}

export function test(c: Calculator, input: string) {
  for (let i = 0; i < input.length; i++) {
    c.handelChar(input[i]);
  }

  console.log(`result of '${input}' is '${c.getResult()}'`);
}

```

这是使用导出的 `test` 函数来测试计算器。

TestCalculator.ts

```
import { Calculator, test } from "./Calculator";

let c = new Calculator();
test(c, "1+2*33/11="); // prints 9
```

现在扩展它，添加支持输入其它进制（十进制以外），让我们来创建 **ProgrammerCalculator.ts**。

ProgrammerCalculator.ts

```
import { Calculator } from "./Calculator";

class ProgrammerCalculator extends Calculator {
    static digits = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E", "F"];

    constructor(public base: number) {
        super();
        if (base <= 0 || base > ProgrammerCalculator.digits.length) {
            throw new Error("base has to be within 0 to 16 inclusive.");
        }
    }

    protected processDigit(digit: string, currentValue: number) {
        if (ProgrammerCalculator.digits.indexOf(digit) >= 0) {
            return currentValue * this.base + ProgrammerCalculator.digits.indexOf(digit);
        }
    }
}

// Export the new extended calculator as Calculator
export { ProgrammerCalculator as Calculator };

// Also, export the helper function
export { test } from "./Calculator";
```

新的 **ProgrammerCalculator** 模块导出的 API 与原先的 **Calculator** 模块很相似，但却没有改变原模块里的对象。下面是测试 **ProgrammerCalculator** 类的代码：

TestProgrammerCalculator.ts

```
import { Calculator, test } from "./ProgrammerCalculator";

let c = new Calculator(2);
test(c, "001+010="); // prints 3
```

模块里不要使用命名空间

当初次进入基于模块的开发模式时，可能总会控制不住要将导出包裹在一个命名空间里。模块具有其自己的作用域，并且只有导出的声明才会在模块外部可见。记住这点，命名空间在使用模块时几乎没什么价值。

在组织方面，命名空间对于在全局作用域内对逻辑上相关的对象和类型进行分组是很便利的。例如，在 C# 里，你会从 `System.Collections` 里找到所有集合的类型。通过将类型有层次地组织在命名空间里，可以方便用户找到与使用那些类型。然而，模块本身已经存在于文件系统之中，这是必须的。我们必须通过路径和文件名找到它们，这已经提供了一种逻辑上的组织形式。我们可以创建 `/collections/generic/` 文件夹，把相应模块放在这里面。

命名空间对解决全局作用域里命名冲突来说是很重要的。比如，你可以有一个 `My.Application.Customer.AddForm` 和 `My.Application.Order.AddForm`-- 两个类型的名字相同，但命名空间不同。然而，这对于模块来说却不是一个问题。在一个模块里，没有理由两个对象拥有同一个名字。从模块的使用角度来说，使用者会挑出他们用来引用模块的名字，所以也没有理由发生重名的情况。

更多关于模块和命名空间的资料查看[命名空间和模块](#)

危险信号

以下均为模块结构上的危险信号。重新检查以确保你没有在对模块使用命名空间：

- 文件的顶层声明是 `export namespace Foo { ... }`（删除 `Foo` 并把所有内容向上层移动一层）
- 文件只有一个 `export class` 或 `export function`（考虑使用 `export default`）
- 多个文件的顶层具有同样的 `export namespace Foo {`（不要以为这些会合并到一个 `Foo` 中！）

命名空间

关于术语的一点说明:请务必注意一点，TypeScript 1.5 里术语名已经发生了变化。

“内部模块”现在称做“命名空间”。“外部模块”现在则简称为“模块”，这是为了与 [ECMAScript 2015](#) 里的术语保持一致，(也就是说 `module X {` 相当于现在推荐的写法 `namespace X {`)。

介绍

这篇文章描述了如何在 TypeScript 里使用命名空间（之前叫做“内部模块”）来组织你的代码。就像我们在术语说明里提到的那样，“内部模块”现在叫做“命名空间”。另外，任何使用 `module` 关键字来声明一个内部模块的地方都应该使用 `namespace` 关键字来替换。这就避免了让新的使用者被相似的名称所迷惑。

第一步

我们先来写一段程序并将在整篇文章中都使用这个例子。我们定义几个简单的字符串验证器，假设你会使用它们来验证表单里的用户输入或验证外部数据。

所有的验证器都放在一个文件里

```
interface StringValidator {
    isAcceptable(s: string): boolean;
}

let lettersRegexp = /^[A-Za-z]+$/;
let numberRegexp = /^[0-9]+$/;

class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}

class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
```

```

        return s.length === 5 && numberRegex.test(s);
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        let isMatch = validators[name].isAcceptable(s);
        console.log(`'${s}' ${isMatch ? "matches" : "does not match"} '${name}'`);
    }
}

```

命名空间

随着更多验证器的加入，我们需要一种手段来组织代码，以便于在记录它们类型的同时还不用担心与其它对象产生命名冲突。因此，我们把验证器包裹到一个命名空间内，而不是把它们放在全局命名空间下。

下面的例子里，把所有与验证器相关的类型都放到一个叫做 **Validation** 的命名空间里。因为我们想让这些接口和类在命名空间之外也是可访问的，所以需要使用 **export**。相反的，变量 **lettersRegex** 和 **numberRegex** 是实现细节，不需要导出，因此它们在命名空间外是不能访问的。在文件末尾的测试代码里，由于是在命名空间之外访问，因此需要限定类型的名称，比如 **Validation.LettersOnlyValidator**。

使用命名空间的验证器

```

namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }
}

```



```

const lettersRegexp = /^[A-Za-z]+$/;
const numberRegexp = /^[0-9]+$/;

export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(`"${s}" - ${validators[name].isAcceptable(s) ? "matches" : "does not match"} ${name}`);
    }
}

```

分离到多文件

当应用变得越来越大时，我们需要将代码分离到不同的文件中以便于维护。

多文件中的命名空间

现在，我们把 **Validation** 命名空间分割成多个文件。尽管是不同的文件，它们仍是同一个命名空间，并且在使用的时候就如同它们在一个文件中定义的一样。因为不同文件之间存在依赖关系，所以我们加入了引用标签来告诉编译器文件之间的关联。我们的测试代码保持不变。

Validation.ts

```
namespace Validation {  
    export interface StringValidator {  
        isAcceptable(s: string): boolean;  
    }  
}
```

LettersOnlyValidator.ts

```
/// <reference path="Validation.ts" />  
namespace Validation {  
    const lettersRegexp = /^[A-Za-z]+$/;  
    export class LettersOnlyValidator implements StringValidator {  
        isAcceptable(s: string) {  
            return lettersRegexp.test(s);  
        }  
    }  
}
```

ZipCodeValidator.ts

```
/// <reference path="Validation.ts" />  
namespace Validation {  
    const numberRegexp = /^[0-9]+$/;  
    export class ZipCodeValidator implements StringValidator {  
        isAcceptable(s: string) {  
            return s.length === 5 && numberRegexp.test(s);  
        }  
    }  
}
```

Test.ts

```
/// <reference path="Validation.ts" />  
/// <reference path="LettersOnlyValidator.ts" />  
/// <reference path="ZipCodeValidator.ts" />
```

```
// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(""" + s + "" " + (validators[name].isAcceptable(s) ? " matches " : " does
not match ") + name);
    }
}
```

当涉及到多文件时，我们必须确保所有编译后的代码都被加载了。我们有两种方式。

第一种方式，把所有的输入文件编译为一个输出文件，需要使用`--outFile` 标记：

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

编译器会根据源码里的引用标签自动地对输出进行排序。你也可以单独地指定每个文件。

```
tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts ZipCodeValidator.ts Test.ts
```

第二种方式，我们可以编译每一个文件（默认方式），那么每个源文件都会对应生成一个 JavaScript 文件。然后，在页面上通过`<script>` 标签把所有生成的 JavaScript 文件按正确的顺序引进来，比如：

MyTestPage.html (excerpt)

```
<script src="Validation.js" type="text/javascript" />
<script src="LettersOnlyValidator.js" type="text/javascript" />
<script src="ZipCodeValidator.js" type="text/javascript" />
<script src="Test.js" type="text/javascript" />
```

别名

另一种简化命名空间操作的方法是使用 `import q = x.y.z` 给常用的对象起一个短的名字。不要与用来加载模块的 `import x = require('name')` 语法弄混了，这里的语法是为指定的符号创建一个别名。你可以用这种方法为任意标识符创建别名，也包括导入的模块中的对象。

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

注意，我们并没有使用 `require` 关键字，而是直接使用导入符号的限定名赋值。这与使用 `var` 相似，但它还适用于类型和导入的具有命名空间含义的符号。重要的是，对于值来讲，`import` 会生成与原始符号不同的引用，所以改变别名的 `var` 值并不会影响原始变量的值。

使用其它的 JavaScript 库

为了描述不是用 TypeScript 编写的类库的类型，我们需要声明类库导出的 API。由于大部分程序库只提供少数的顶级对象，命名空间是用来表示它们的一个好办法。

我们称其为声明是因为它不是外部程序的具体实现。我们通常在 `.d.ts` 里写这些声明。如果你熟悉 C/C++，你可以把它们当做 `.h` 文件。让我们看一些例子。

外部命名空间

流行的程序库 D3 在全局对象 `d3` 里定义它的功能。因为这个库通过一个 `<script>` 标签加载（不是通过模块加载器），它的声明文件使用内部模块来定义它的类型。为了让 TypeScript 编译器识别它的类型，我们使用外部命名空间声明。比如，我们可以像下面这样写：

D3.d.ts (部分摘录)

```
declare namespace D3 {  
  export interface Selectors {  
    select: {  
      (selector: string): Selection;  
      (element: EventTarget): Selection;  
    };  
  }  
  
  export interface Event {  
    x: number;  
    y: number;  
  }  
  
  export interface Base extends Selectors {  
    event: Event;  
  }  
}  
  
declare var d3: D3.Base;
```

命名空间和模块

关于术语的一点说明:请务必注意一点, TypeScript 1.5 里术语名已经发生了变化。“内部模块”现在称做“命名空间”。“外部模块”现在则简称为“模块”, 这是为了与 [ECMAScript 2015](#) 里的术语保持一致, (也就是说 `module X {` 相当于现在推荐的写法 `namespace X {`)。

介绍

这篇文章将概括介绍在 TypeScript 里使用模块与命名空间来组织代码的方法。我们也会谈及命名空间和模块的高级使用场景, 和在使用它们的过程中常见的陷阱。

查看[模块](#)章节了解关于模块的更多信息。查看[命名空间](#)章节了解关于命名空间的更多信息。

使用命名空间

命名空间是位于全局命名空间下的一个普通的带有名字的 JavaScript 对象。这令命名空间十分容易使用。它们可以在多文件中同时使用，并通过 `--outFile` 结合在一起。命名空间是帮你组织 Web 应用不错的方式，你可以把所有依赖都放在 HTML 页面的 `<script>` 标签里。

但就像其它的全局命名空间污染一样，它很难去识别组件之间的依赖关系，尤其是在大型的应用中。

使用模块

像命名空间一样，模块可以包含代码和声明。不同的是模块可以声明它的依赖。

模块会把依赖添加到模块加载器上（例如 CommonJs / Require.js）。对于小型的 JS 应用来说可能没必要，但是对于大型应用，这一点点的花费会带来长久的模块化和可维护性上的便利。模块也提供了更好的代码重用，更强的封闭性以及更好的使用工具进行优化。

对于 Node.js 应用来说，模块是默认并推荐的组织代码的方式。

从 ECMAScript 2015 开始，模块成为了语言内置的部分，应该会被所有正常的解释引擎所支持。因此，对于新项目来说推荐使用模块做为组织代码的方式。

命名空间和模块的陷阱

这部分我们会描述常见的命名空间和模块的使用陷阱和如何去避免它们。

对模块使用 `/// <reference>`

一个常见的错误是使用 `/// <reference>` 引用模块文件，应该使用 `import`。要理解这之间的区别，我们首先应该弄清编译器是如何根据 `import` 路径（例如，`import x from "...";` 或 `import x = require("...")` 里面的 `...`，等等）来定位模块的类型信息的。

编译器首先尝试去查找相应路径下的 `.ts`，`.tsx` 再或者 `.d.ts`。如果这些文件都找不到，编译器会查找外部模块声明。回想一下，它们是在 `.d.ts` 文件里声明的。

- `myModules.d.ts`

```
// In a .d.ts file or .ts file that is not a module:  
declare module "SomeModule" {  
    export function fn(): string;  
}
```

- `myOtherModule.ts`

```
/// <reference path="myModules.d.ts" />  
import * as m from "SomeModule";
```

这里的引用标签指定了外来模块的位置。这就是一些 TypeScript 例子中引用 `node.d.ts` 的方法。

不必要的命名空间

如果你想把命名空间转换为模块，它可能会像下面这个文件一件：

- `shapes.ts`

```
export namespace Shapes {  
    export class Triangle { /* ... */ }  
    export class Square { /* ... */ }  
}
```

顶层的模块 `Shapes` 包裹了 `Triangle` 和 `Square`。对于使用它的人来说这是令人迷惑和讨厌的：

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";  
let t = new shapes.Shapes.Triangle(); // shapes.Shapes?
```

TypeScript 里模块的一个特点是不同的模块永远也不会在相同的作用域内使用相同的名字。因为使用模块的人会为它们命名，所以完全没有必要把导出的符号包裹在一个命名空间里。

再次重申，不应该对模块使用命名空间，使用命名空间是为了提供逻辑分组和避免命名冲突。模块文件本身已经是一个逻辑分组，并且它的名字是由导入这个模块的代码指定，所以没有必要为导出的对象增加额外的模块层。

下面是改进的例子：

- `shapes.ts`

```
export class Triangle { /* ... */ }  
export class Square { /* ... */ }
```

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";  
let t = new shapes.Triangle();
```

模块的取舍

就像每个 JS 文件对应一个模块一样，TypeScript 里模块文件与生成的 JS 文件也是一一对应的。这会产生一种影响，根据你指定的目标模块系统的不同，你可能无法连接多个模块源文件。例如当目标模块系统为 `commonjs` 或 `umd` 时，无法使用 `outFile` 选项，但是在 TypeScript 1.8 以上的版本能够使用 `outFile` 当目标为 `amd` 或 `system`。

模块解析

这节假设你已经了解了模块的一些基本知识 请阅读[模块](#)文档了解更多信息。

模块解析就是指编译器所要依据的一个流程，用它来找出某个导入操作所引用的具体值。假设有一个导入语句 `import { a } from "moduleA";` 为了去检查任何对 `a` 的使用，编译器需要准确的知道它表示什么，并且会需要检查它的定义 `moduleA`。

这时候，编译器会想知道 “`moduleA` 的 shape 是怎样的？” 这听上去很简单，`moduleA` 可能在你写的某个 `.ts/.tsx` 文件里或者在你的代码所依赖的 `.d.ts` 里。

首先，编译器会尝试定位表示导入模块的文件。编译会遵循下列二种策略之一：[Classic](#) 或 [Node](#)。这些策略会告诉编译器到[哪里](#)去查找 `moduleA`。

如果它们失败了并且如果模块名是非相对的（且是在"`moduleA`"的情况下），编译器会尝试定位一个[外部模块声明](#)。我们接下来会讲到非相对导入。

最后，如果编译器还是不能解析这个模块，它会记录一个错误。在这种情况下，错误可能为 `error TS2307: Cannot find module 'moduleA'.`

相对 vs. 非相对模块导入

根据模块引用是相对的还是非相对的，模块导入会以不同的方式解析。

*相对*导入是以`/`，`./`或`../`开头的。下面是一些例子：

- `import Entry from "./components/Entry";`
- `import { DefaultHeaders } from "../constants/http";`
- `import "/mod";`

所有其它形式的导入被当作*非相对*的。下面是一些例子：

- `import * as $ from "jQuery";`
- `import { Component } from "@angular/core";`

相对导入解析时是相对于导入它的文件来的，并且不能解析为一个外部模块声明。你应该为你自己写的模块使用相对导入，这样能确保它们在运行时的相对位置。

非相对模块的导入可以相对于 `baseUrl` 或通过下文会讲到的路径映射来进行解析。它们还可以被解析成[外部模块声明](#)。使用非相对路径来导入你的外部依赖。

模块解析策略

共有两种可用的模块解析策略：[Node](#) 和 [Classic](#)。你可以使用 `--moduleResolution` 标记指定使用哪种模块解析策略。若未指定，那么在使用了 `--module AMD | System | ES2015` 时的默认值为 [Classic](#)，其它情况时则为 [Node](#)。

Classic

这种策略以前是 TypeScript 默认的解析策略。现在，它存在的理由主要是为了向后兼容。

相对导入的模块是相对于导入它的文件进行解析的。因此 `/root/src/folder/A.ts` 文件里的 `import { b } from "./moduleB"` 会使用下面的查找流程：

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`

对于非相对模块的导入，编译器则会从包含导入文件的目录开始依次向上级目录遍历，尝试定位匹配的声明文件。

比如：

有一个对 `moduleB` 的非相对导入 `import { b } from "moduleB"`，它是在 `/root/src/folder/A.ts` 文件里，会以如下的方式来定位 `"moduleB"`：

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`
3. `/root/src/moduleB.ts`
4. `/root/src/moduleB.d.ts`
5. `/root/moduleB.ts`
6. `/root/moduleB.d.ts`
7. `/moduleB.ts`
8. `/moduleB.d.ts`

Node

这个解析策略试图在运行时模仿 [Node.js](#) 模块解析机制。完整的 Node.js 解析算法可以在 [Node.js module documentation](#) 找到。

Node.js 如何解析模块

为了理解 TypeScript 编译依照的解析步骤，先弄明白 Node.js 模块是非常重要的。通常，在 Node.js 里导入是通过 **require** 函数调用进行的。Node.js 会根据 **require** 的是相对路径还是非相对路径做出不同的行为。

相对路径很简单。例如，假设有一个文件路径为 **/root/src/moduleA.js**，包含了一个导入 **var x = require("./moduleB");** Node.js 以下面的顺序解析这个导入：

1. 将 **/root/src/moduleB.js** 视为文件，检查是否存在。
2. 将 **/root/src/moduleB** 视为目录，检查是否它包含 **package.json** 文件并且其指定了一个 **"main"** 模块。在我们的例子里，如果 Node.js 发现文件 **/root/src/moduleB/package.json** 包含了 **{ "main": "lib/mainModule.js" }**，那么 Node.js 会引用 **/root/src/moduleB/lib/mainModule.js**。
3. 将 **/root/src/moduleB** 视为目录，检查它是否包含 **index.js** 文件。这个文件会被隐式地当作那个文件夹下的 **"main"** 模块。

你可以阅读 Node.js 文档了解更多详细信息：[file modules](#) 和 [folder modules](#)。

但是，**非相对模块名**的解析是个完全不同的过程。Node 会在一个特殊的文件夹 **node_modules** 里查找你的模块。**node_modules** 可能与当前文件在同一级目录下，或者在上层目录里。Node 会向上级目录遍历，查找每个 **node_modules** 直到它找到要加载的模块。

还是用上面例子，但假设 **/root/src/moduleA.js** 里使用的是非相对路径导入 **var x = require("moduleB");**。Node 则会以下面的顺序去解析 **moduleB**，直到有一个匹配上。

1. `/root/src/node_modules/moduleB.js`
2. `/root/src/node_modules/moduleB/package.json`(如果指定了"main"属性)
3. `/root/src/node_modules/moduleB/index.js`
4. `/root/node_modules/moduleB.js`
5. `/root/node_modules/moduleB/package.json`(如果指定了"main"属性)
6. `/root/node_modules/moduleB/index.js`
7. `/node_modules/moduleB.js`
8. `/node_modules/moduleB/package.json`(如果指定了"main"属性)
9. `/node_modules/moduleB/index.js`

注意 Node.js 在步骤（4）和（7）会向上跳一级目录。

你可以阅读 Node.js 文档了解更多详细信息：[loading modules from node_modules](#)。

TypeScript 如何解析模块

TypeScript 是模仿 Node.js 运行时的解析策略来在编译阶段定位模块定义文件。因此，TypeScript 在 Node 解析逻辑基础上增加了 TypeScript 源文件的扩展名（`.ts`，`.tsx` 和 `.d.ts`）。同时，TypeScript 在 `package.json` 里使用字段 `"typings"` 来表示类似 `"main"` 的意义 - 编译器会使用它来找到要使用的 `"main"` 定义文件。

比如，有一个导入语句 `import { b } from './moduleB'` 在 `/root/src/moduleA.ts` 里，会以下的流程来定位 `./moduleB`：

1. `/root/src/moduleB.ts`
2. `/root/src/moduleB.tsx`
3. `/root/src/moduleB.d.ts`
4. `/root/src/moduleB/package.json`(如果指定了 `"typings"` 属性)
5. `/root/src/moduleB/index.ts`
6. `/root/src/moduleB/index.tsx`
7. `/root/src/moduleB/index.d.ts`

回想一下 Node.js 先查找 `moduleB.js` 文件，然后是合适的 `package.json`，再之后是 `index.js`。

类似地，非相对的导入会遵循 Node.js 的解析逻辑，首先查找文件，然后是合适的文件夹。因此 `/root/src/moduleA.ts` 文件里的 `import { b } from "moduleB"` 会以下面的查找顺序解析：

1. `/root/src/node_modules/moduleB.ts`
2. `/root/src/node_modules/moduleB.tsx`
3. `/root/src/node_modules/moduleB.d.ts`
4. `/root/src/node_modules/moduleB/package.json`(如果指定了 `"typings"` 属性)
5. `/root/src/node_modules/moduleB/index.ts`
6. `/root/src/node_modules/moduleB/index.tsx`
7. `/root/src/node_modules/moduleB/index.d.ts`
8. `/root/node_modules/moduleB.ts`
9. `/root/node_modules/moduleB.tsx`
10. `/root/node_modules/moduleB.d.ts`
11. `/root/node_modules/moduleB/package.json`(如果指定了 `"typings"` 属性)
12. `/root/node_modules/moduleB/index.ts`
13. `/root/node_modules/moduleB/index.tsx`
14. `/root/node_modules/moduleB/index.d.ts`
15. `/node_modules/moduleB.ts`
16. `/node_modules/moduleB.tsx`
17. `/node_modules/moduleB.d.ts`
18. `/node_modules/moduleB/package.json`(如果指定了 `"typings"` 属性)
19. `/node_modules/moduleB/index.ts`
20. `/node_modules/moduleB/index.tsx`
21. `/node_modules/moduleB/index.d.ts`

不要被这里步骤的数量吓到 - TypeScript 只是在步骤（8）和（15）向上跳了两次目录。这并不比 Node.js 里的流程复杂。

附加的模块解析标记

有时工程源码结构与输出结构不同。通常是要经过一系统的构建步骤最后生成输出。它们包括将 **.ts** 编译成 **.js**，将不同位置的依赖拷贝至一个输出位置。最终结果就是运行时的模块名与包含它们声明的源文件里的模块名不同。或者最终输出文件里的模块路径与编译时的源文件路径不同了。

TypeScript 编译器有一些额外的标记用来通知编译器在源码编译成最终输出的过程中都发生了哪个转换。

有一点要特别注意的是编译器不会进行这些转换操作；它只是利用这些信息来指导模块的导入。

Base URL

在利用 AMD 模块加载器的应用里使用 **baseUrl** 是常见做法，它要求在运行时模块都被放到了一个文件夹里。这些模块的源码可以在不同的目录下，但是构建脚本会将它们集中到一起。

设置 **baseUrl** 来告诉编译器到哪里去查找模块。所有非相对模块导入都会被当做相对于 **baseUrl**。

baseUrl 的值由以下两者之一决定：

- 命令行中 **baseUrl** 的值（如果给定的路径是相对的，那么将相对于当前路径进行计算）
- ‘tsconfig.json’里的 **baseUrl** 属性（如果给定的路径是相对的，那么将相对于 ‘tsconfig.json’ 路径进行计算）

注意相对模块的导入不会被设置的 **baseUrl** 所影响，因为它们总是相对于导入它们的文件。

阅读更多关于 **baseUrl** 的信息 [RequireJS](#) 和 [SystemJS](#)。

路径映射

有时模块不是直接放在 `baseUrl` 下面。比如，充分 `jquery` 模块地导入，在运行时可能被解释为 `"node_modules/jquery/dist/jquery.slim.min.js"`。加载器使用映射配置来将模块名映射到运行时的文件，查看 [RequireJs documentation](#) 和 [SystemJS documentation](#)。

TypeScript 编译器通过使用 `tsconfig.json` 文件里的 `"paths"` 来支持这样的声明映射。下面是一个如何指定 `jquery` 的 `"paths"` 的例子。

```
{
  "compilerOptions": {
    "baseUrl": ".", // This must be specified if "paths" is.
    "paths": {
      "jquery": ["node_modules/jquery/dist/jquery"] // 此处映射是相对于"baseUrl"
    }
  }
}
```

请注意 `"paths"` 是相对于 `"baseUrl"` 进行解析。如果 `"baseUrl"` 被设置成了除 `"."` 外的其它值，比如 `tsconfig.json` 所在的目录，那么映射必须要做相应的改变。如果你在上例中设置了 `"baseUrl": "./src"`，那么 `jquery` 应该映射到 `"../node_modules/jquery/dist/jquery"`。

通过 `"paths"` 我们还可以指定复杂的映射，包括指定多个回退位置。假设在一个工程配置里，有一些模块位于一处，而其它的则在另个的位置。构建过程会将它们集中至一处。工程结构可能如下：

```
projectRoot
├─ folder1
│   ├─ file1.ts (imports 'folder1/file2' and 'folder2/file3')
│   └─ file2.ts
├─ generated
│   ├─ folder1
│   └─ folder2
│       └─ file3.ts
└─ tsconfig.json
```

相应的 `tsconfig.json` 文件如下：

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "*": [
        "*",
        "generated/*"
      ]
    }
  }
}
```

它告诉编译器所有匹配 "*"（所有的值）模式的模块导入会在以下两个位置查找：

1. "*": 表示名字不发生改变，所以映射为
`<moduleName>=><baseUrl>/<moduleName>`
2. "generated/*"表示模块名添加了 “generated”前缀，所以映射为
`<moduleName>=><baseUrl>/generated/<moduleName>`

按照这个逻辑，编译器将会如下尝试解析这两个导入：

- 导入'folder1/file2'
 1. 匹配'*'模式且通配符捕获到整个名字。
 2. 尝试列表里的第一个替换：'*' -> `folder1/file2`。
 3. 替换结果为非相对名 - 与 `baseUrl` 合并 -> `projectRoot/folder1/file2.ts`。
 4. 文件存在。完成。
- 导入'folder2/file3'
 1. 匹配'*'模式且通配符捕获到整个名字。
 2. 尝试列表里的第一个替换：'*' -> `folder2/file3`。
 3. 替换结果为非相对名 - 与 `baseUrl` 合并 -> `projectRoot/folder2/file3.ts`。
 4. 文件不存在，跳到第二个替换。
 5. 第二个替换：'generated/*' -> `generated/folder2/file3`。
 6. 替换结果为非相对名 - 与 `baseUrl` 合并 -
`>projectRoot/generated/folder2/file3.ts`。
 7. 文件存在。完成。

利用 `rootDirs` 指定虚拟目录

有时多个目录下的工程源文件在编译时会进行合并放在某个输出目录下。这可以看做一些源目录创建了一个“虚拟”目录。

利用 `rootDirs`，可以告诉编译器生成这个虚拟目录的 `roots`；因此编译器可以在“虚拟”目录下解析相对模块导入，就好像它们被合并在了一起一样。

比如，有下面的工程结构：

```
src
├── views
│   ├── view1.ts (imports './template1')
│   └── view2.ts
└── generated
    ├── templates
    │   └── views
    │       └── template1.ts (imports './view2')
```

`src/views` 里的文件是用于控制 UI 的用户代码。`generated/templates` 是 UI 模版，在构建时通过模版生成器自动生成。构建中的一步会将 `src/views` 和 `generated/templates/views` 的输出拷贝到同一个目录下。在运行时，视图可以假设它的模版与它同在一个目录下，因此可以使用相对导入 `"/template"`。

可以使用 `"rootDirs"` 来告诉编译器。`"rootDirs"` 指定了一个 `roots` 列表，列表里的内容会在运行时被合并。因此，针对这个例子，`tsconfig.json` 如下：

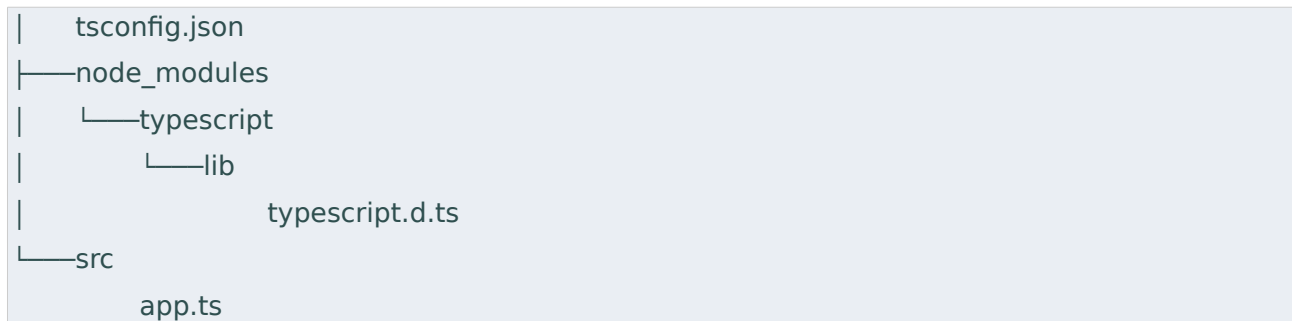
```
{
  "compilerOptions": {
    "rootDirs": [
      "src/views",
      "generated/templates/views"
    ]
  }
}
```

第当编译器在 `rootDirs` 的子目录下找到一个相对模块导入，它会尝试从 `rootDirs` 里导入。

跟踪模块解析

如之前讨论，编译器在解析模块时可能访问当前文件夹外的文件。这会导致很难诊断模块为什么没有被解析，或解析到了错误的位置。通过 `--traceResolution` 启用编译器的模块解析跟踪，它会告诉我们在模块解析过程中发生了什么。

假设我们有一个使用了 `typescript` 模块的简单应用。`app.ts` 里有一个这样的导入 `import * as ts from "typescript"`。



使用 `--traceResolution` 调用编译器。

```
tsc --traceResolution
```

输出结果如下：

```
===== Resolving module 'typescript' from 'src/app.ts'. =====
Module resolution kind is not specified, using 'NodeJs'.
Loading module 'typescript' from 'node_modules' folder.
File 'src/node_modules/typescript.ts' does not exist.
File 'src/node_modules/typescript.tsx' does not exist.
File 'src/node_modules/typescript.d.ts' does not exist.
File 'src/node_modules/typescript/package.json' does not exist.
File 'node_modules/typescript.ts' does not exist.
File 'node_modules/typescript.tsx' does not exist.
File 'node_modules/typescript.d.ts' does not exist.
Found 'package.json' at 'node_modules/typescript/package.json'.
'package.json' has 'typings' field './lib/typescript.d.ts' that references
'node_modules/typescript/lib/typescript.d.ts'.
```

```
File 'node_modules/typescript/lib/typescript.d.ts' exist - use it as a module resolution result.
===== Module name 'typescript' was successfully resolved to
'node_modules/typescript/lib/typescript.d.ts'. =====
```

需要留意的地方

- 导入的名字及位置

```
===== Resolving module 'typescript' from 'src/app.ts'. =====
```

- 编译器使用的策略

```
Module resolution kind is not specified, using 'NodeJs'.
```

- 从 npm 加载 typings

```
'package.json' has 'typings' field './lib/typescript.d.ts' that references
'node_modules/typescript/lib/typescript.d.ts'.
```

- 最终结果

```
===== Module name 'typescript' was successfully resolved to
'node_modules/typescript/lib/typescript.d.ts'. =====
```

使用 `--noResolve`

正常来讲编译器会在开始编译之前解析模块导入。每当它成功地解析了对一个文件 `import`，这个文件会被加到一个文件列表里，以供编译器稍后处理。

`--noResolve` 编译选项告诉编译器不要添加任何不是在命令行上传入的文件到编译列表。编译器仍然会尝试解析模块，但是只要没有指定这个文件，那么它就不会被包含在内。

比如

`app.ts`

```
import * as A from "moduleA" // OK, moduleA passed on the command-line
import * as B from "moduleB" // Error TS2307: Cannot find module 'moduleB'.
tsc app.ts moduleA.ts --noResolve
```

使用 `--noResolve` 编译 `app.ts`：

- 可能正确找到 `moduleA`，因为它在命令行上指定了。
- 找不到 `moduleB`，因为没有在命令行上传递。

常见问题

为什么在 `exclude` 列表里的模块还会被编译器使用

`tsconfig.json` 将文件夹转变一个“工程” 如果不指定任何“`exclude`”或“`files`”，文件夹里的所有文件包括 `tsconfig.json` 和所有的子目录都会在编译列表里。如果你想利用“`exclude`”排除某些文件，甚至你想指定所有要编译的文件列表，请使用“`files`”。

有些是被 `tsconfig.json` 自动加入的。它不会涉及到上面讨论的模块解析。如果编译器识别出一个文件是模块导入目标，它就会加到编译列表里，不管它是否被排除了。

因此，要从编译列表中排除一个文件，你需要在排除它的同时，还要排除所有对它进行 `import` 或使用了 `/// <reference path="..." />` 指令的文件。

声明合并

介绍

TypeScript 中有些独特的概念可以在类型层面上描述 JavaScript 对象的模型。这其中尤其独特的一个例子是“声明合并”的概念。理解了这个概念，将有助于操作现有的 JavaScript 代码。同时，也会有助于理解更多高级抽象的概念。

对本文件来讲，“声明合并”是指编译器将针对同一个名字的两个独立声明合并为单一声明。合并后的声明同时拥有原先两个声明的特性。任何数量的声明都可被合并；不局限于两个声明。

基础概念

TypeScript 中的声明会创建以下三种实体之一：命名空间，类型或值。创建命名空间的声明会新建一个命名空间，它包含了用 (.) 符号来访问时使用的名字。创建类型的声明是：用声明的模型创建一个类型并绑定到给定的名字上。最后，创建值的声明会创建在 JavaScript 输出中看到的值。

Declaration Type	Namespace	Type	Value
Namespace	X		X
Class		X	X
Enum		X	X
Interface		X	
Type Alias		X	
Function			X
Variable			X

理解每个声明创建了什么，有助于理解当声明合并时有哪些东西被合并了。

合并接口

最简单也最常见的声明合并类型是接口合并。从根本上说，合并的机制是把双方的成员放到一个同名的接口里。

```
interface Box {  
    height: number;  
    width: number;  
}  
  
interface Box {  
    scale: number;  
}  
  
let box: Box = {height: 5, width: 6, scale: 10};
```

接口的非函数的成员必须是唯一的。如果两个接口中同时声明了同名的非函数成员编译器则会报错。

对于函数成员，每个同名函数声明都会被当成这个函数的一个重载。同时需要注意，当接口 **A** 与后来的接口 **A** 合并时，后面的接口具有更高的优先级。

如下例所示：

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

这三个接口合并成一个声明：

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

注意每组接口里的声明顺序保持不变，但各组接口之间的顺序是后来的接口重载出现在靠前位置。

这个规则有一个例外是当出现特殊的函数签名时。如果签名里有一个参数的类型是单一的字符串字面量（比如，不是字符串字面量的联合类型），那么它将会被提升到重载列表的最顶端。

比如，下面的接口会合并到一起：

```
interface Document {  
    createElement(tagName: any): Element;  
}  
interface Document {
```

```

    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
}
interface Document {
    createElement(tagName: string): HTMLElement;
    createElement(tagName: "canvas"): HTMLCanvasElement;
}

```

合并后的 **Document** 将会像下面这样：

```

interface Document {
    createElement(tagName: "canvas"): HTMLCanvasElement;
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
    createElement(tagName: string): HTMLElement;
    createElement(tagName: any): Element;
}

```

合并命名空间

与接口相似，同名的命名空间也会合并其成员。命名空间会创建出命名空间和值，我们需要知道这两者都是怎么合并的。

对于命名空间的合并，模块导出的同名接口进行合并，构成单一命名空间内含合并后的接口。

对于命名空间里值的合并，如果当前已经存在给定名字的命名空间，那么后来的命名空间的导出成员会被加到已经存在的那个模块里。

Animals 声明合并示例：

```

namespace Animals {
    export class Zebra { }
}

namespace Animals {
    export interface Legged { numberOfLegs: number; }
}

```

```
export class Dog { }  
}
```

等同于：

```
namespace Animals {  
  export interface Legged { numberOfLegs: number; }  
  
  export class Zebra { }  
  export class Dog { }  
}
```

除了这些合并外，你还需要了解非导出成员是如何处理的。非导出成员仅在其原有的（合并前的）命名空间内可见。这就是说合并之后，从其它命名空间合并进来的成员无法访问非导出成员。

下例提供了更清晰的说明：

```
namespace Animal {  
  let haveMuscles = true;  
  
  export function animalsHaveMuscles() {  
    return haveMuscles;  
  }  
}  
  
namespace Animal {  
  export function doAnimalsHaveMuscles() {  
    return haveMuscles; // <-- error, haveMuscles is not visible here  
  }  
}
```

因为 `haveMuscles` 并没有导出，只有 `animalsHaveMuscles` 函数共享了原始未合并的命名空间可以访问这个变量。`doAnimalsHaveMuscles` 函数虽是合并命名空间的一部分，但是访问不了未导出的成员。

命名空间与类和函数和枚举类型合并

命名空间可以与其它类型的声明进行合并。只要命名空间的定义符合将要合并类型的定义。合并结果包含两者的声明类型。TypeScript 使用这个功能去实现一些 JavaScript 里的设计模式。

合并命名空间和类

这让我们可以表示内部类。

```
class Album {  
    label: Album.AlbumLabel;  
}  
  
namespace Album {  
    export class AlbumLabel { }  
}
```

合并规则与上面[合并命名空间](#)小节里讲的规则一致，我们必须导出 **AlbumLabel** 类，好让合并的类能访问。合并结果是一个类并带有一个内部类。你也可以使用命名空间为类增加一些静态属性。

除了内部类的模式，你在 JavaScript 里，创建一个函数稍后扩展它增加一些属性也是很常见的。TypeScript 使用声明合并来达到这个目的并保证类型安全。

```
function buildLabel(name: string): string {  
    return buildLabel.prefix + name + buildLabel.suffix;  
}  
  
namespace buildLabel {  
    export let suffix = "";  
    export let prefix = "Hello, ";  
}  
  
alert(buildLabel("Sam Smith"));
```

相似的，命名空间可以用来扩展枚举型：

```
enum Color {
```

```

    red = 1,
    green = 2,
    blue = 4
}

namespace Color {
    export function mixColor(colorName: string) {
        if (colorName == "yellow") {
            return Color.red + Color.green;
        }
        else if (colorName == "white") {
            return Color.red + Color.green + Color.blue;
        }
        else if (colorName == "magenta") {
            return Color.red + Color.blue;
        }
        else if (colorName == "cyan") {
            return Color.green + Color.blue;
        }
    }
}

```

非法的合并

TypeScript 并非允许所有的合并。目前，类不能与其它类或变量合并。想要了解如何模仿类的合并，请参考 [TypeScript 的混入](#)。

模块扩展

虽然 JavaScript 不支持合并，但你可以为导入的对象打补丁以更新它们。让我们考察一下这个玩具性的示例：

```

// observable.js
export class Observable<T> {
    // ... implementation left as an exercise for the reader ...
}

```

```
// map.js
import { Observable } from "./observable";
Observable.prototype.map = function (f) {
  // ... another exercise for the reader
}
```

它也可以很好地工作在 TypeScript 中，但编译器对 `Observable.prototype.map` 一无所知。你可以使用扩展模块来将它告诉编译器：

```
// observable.ts stays the same
// map.ts
import { Observable } from "./observable";
declare module "./observable" {
  interface Observable<T> {
    map<U>(f: (x: T) => U): Observable<U>;
  }
}
Observable.prototype.map = function (f) {
  // ... another exercise for the reader
}

// consumer.ts
import { Observable } from "./observable";
import "./map";
let o: Observable<number>;
o.map(x => x.toFixed());
```

模块名的解析和用 `import/export` 解析模块标识符的方式是一致的。更多信息请参考 [Modules](#)。当这些声明在扩展中合并时，就好像在原始位置被声明了一样。但是，你不能在扩展中声明新的顶级声明——仅可以扩展模块中已经存在的声明。

全局扩展

你也可以在模块内部添加声明到全局作用域中。

```
// observable.ts
export class Observable<T> {
```

```

    // ... still no implementation ...
}

declare global {
    interface Array<T> {
        toObservable(): Observable<T>;
    }
}

Array.prototype.toObservable = function () {
    // ...
}

```

全局扩展与模块扩展的行为和限制是相同的。

JSX

介绍

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，但是也被其它应用所使用。TypeScript 支持内嵌，类型检查和将 JSX 直接编译为 JavaScript。

基本用法

想要使用 JSX 必须做两件事：

1. 给文件一个 **.tsx** 扩展名
2. 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
<code>preserve</code>	<code><div /></code>	<code><div /></code>	<code>.jsx</code>
<code>react</code>	<code><div /></code>	<code>React.createElement("div")</code>	<code>.js</code>
<code>react-native</code>	<code><div /></code>	<code><div /></code>	<code>.js</code>

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。

注意：`React` 标识符是写死的硬编码，所以你必须保证 `React`（大写的 R）是可用的。

as 操作符

回想一下怎么写类型断言：

```
var foo = <foo>bar;
```

这里我们断言 `bar` 变量是 `foo` 类型的。因为 TypeScript 也使用尖括号来表示类型断言，JSX 的语法带来了解析的困难。因此，TypeScript 在 `.tsx` 文件里禁用了使用尖括号的类型断言。

为了弥补 `.tsx` 里的这个功能，新加入了一个类型断言符号：`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，并且与其它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：

1. 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)`）。

2. 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。

注意：你也可以在 `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
```

可以限制基于值的元素的类型。然而，为了这么做我们需要引入两个新的术语：*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，那么它的类类型就是这个类。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，实例类型就确定了，为类类型调用签名的返回值与构造签名的联合类型。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 }
```

元素的实例类型很有趣，因为它必须赋值给 `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。

```
declare namespace JSX {
  interface ElementClass {
    render: any;
  }
}

class MyComponent {
  render() {}
}

function MyFactoryFunction() {
  return { render: () => {} }
}

<MyComponent />; // 正确
<MyFactoryFunction />; // 正确

class NotAValidComponent {}
function NotAValidFactoryFunction() {
  return {};
}

<NotAValidComponent />; // 错误
<NotAValidFactoryFunction />; // 错误
```

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。

```
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" />
```

元素属性类型用于在 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
```

注意：如果一个属性名不是个合法的 JS 标识符（像 `data-*` 属性），并且它没出现在元素属性类型里时不会当做一个错误。

延展操作符也可以使用：

```
var props = { requiredProp: 'bar' };  
<foo {...props} />; // 正确  
  
var badProps = {};  
<foo {...badProps} />; // 错误
```

JSX 结果类型

默认地 JSX 表达式结果类型为 `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>
```

React 整合

要想一起使用 JSX 和 React，你应该使用 [React 类型定义](#)。这些类型声明定义了 **JSX** 合适命名空间来使用 React。

```
/// <reference path="react.d.ts" />

interface Props {
  foo: string;
}

class MyComponent extends React.Component<Props, {}> {
  render() {
    return <span>{this.props.foo}</span>
  }
}

<MyComponent foo="bar" />; // 正确
<MyComponent foo={0} />; // 错误
```

装饰器

介绍

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。Javascript 里的装饰器目前处在[建议征集的第二阶段](#)，但在 TypeScript 里已做为一项实验性特性予以支持。

注意 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

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" ...
}
```

注意 后面类装饰器小节里有一个更加详细的例子。

装饰器工厂

如果我们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"...
  }
}
```

注意 下面[方法装饰器](#)小节里有一个更加详细的例子。

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 $(f \circ g)(x)$ 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。

如果我们使用[装饰器工厂](#)的话，可以通过下面的例子来观察它们求值的顺序：

```
function f() {
  console.log("f(): evaluated");
  return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
    console.log("f(): called");
  }
}

function g() {
  console.log("g(): evaluated");
  return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
    console.log("g(): called");
  }
}
```

```
class C {  
    @f()  
    @g()  
    method() {}  
}
```

在控制台里会打印出如下结果：

```
f(): evaluated  
g(): evaluated  
g(): called  
f(): called
```

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：

1. 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。
2. 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。
3. 参数装饰器应用到构造函数。
4. 类装饰器应用到类。

类装饰器

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。

注意 如果你要返回一个新的构造函数，你必须注意处理好原来的原型链。在运行时的装饰器调用逻辑中不会为你做这些。

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。(注：参见 [Object.seal](#))

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：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 成员的属性描述符。

注意 如果代码输出目标版本小于 ES5，属性描述符将会是 `undefined`。

如果方法装饰器返回一个值，它会被用作方法的属性描述符。

注意 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：

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

我们可以用下面的函数声明来定义 `@enumerable` 装饰器：

```
function enumerable(value: boolean) {  
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
```



```
        descriptor.enumerable = value;
    };
}
```

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

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。

注意 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：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 成员的属性描述符。

注意 如果代码输出目标版本小于 ES5，`Property Descriptor` 将会是`undefined`。

如果访问器装饰器返回一个值，它会被用作方法的属性描述符。

注意 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：

```
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; }
}
```

我们可以通过如下函数声明来定义@configurable 装饰器：

```
function configurable(value: boolean) {
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        descriptor.configurable = value;
    };
}
```

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：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。

注意 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。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。

如果属性装饰器返回一个值，它会被用作方法的属性描述符。

注意 如果代码输出目标版本小于 ES5，返回值会被忽略。

如果访问符装饰器返回一个值，它会被用作方法的属性描述符。

我们可以用它来记录这个属性的元数据，如下例所示：

```
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);
  }
}
```

然后定义 `@format` 装饰器和 `getFormat` 函数：

```
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);
}
```

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。

注意 这个例子需要使用 `reflect-metadata` 库。查看[元数据](#)了解 `reflect-metadata` 库更详细的信息。

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：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 参数在函数参数列表中的索引。

注意 参数装饰器只能用来监视一个方法的参数是否被传入。

参数装饰器的返回值会被忽略。

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);
    }
}
```

@required 装饰器添加了元数据实体把参数标记为必需的。**@validate** 装饰器把 **greet** 方法包裹在一个函数里在调用原先的函数前验证函数参数。

注意 这个例子使用了 **reflect-metadata** 库。查看[元数据](#)了解 **reflect-metadata** 库的更多信息。

元数据

一些例子使用了 **reflect-metadata** 库来支持实验性的 **metadata API**。这个库还不是 ECMAScript (JavaScript) 标准的一部分。然而，当装饰器被 ECMAScript 官方标准采纳后，这些扩展也将被推荐给 ECMAScript 以采纳。

你可以通过 npm 安装这个库：

```
npm i reflect-metadata --save
```

TypeScript 支持为带有装饰器的声明生成元数据。你需要在命令行或 **tsconfig.json** 里启用 **emitDecoratorMetadata** 编译器选项。

Command Line:

```
tsc --target ES5 --experimentalDecorators --emitDecoratorMetadata
```

tsconfig.json:

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

当启用后，只要 **reflect-metadata** 库被引入了，设计阶段添加的类型信息可以在运行时使用。

如下例所示：

```
import "reflect-metadata";

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

class Line {
  private _p0: Point;
  private _p1: Point;

  @validate
  set p0(value: Point) { this._p0 = value; }
  get p0() { return this._p0; }

  @validate
  set p1(value: Point) { this._p1 = value; }
  get p1() { return this._p1; }
}

function validate<T>(target: any, propertyKey: string, descriptor:
TypedPropertyDescriptor<T>) {
```

```

let set = descriptor.set;
descriptor.set = function (value: T) {
    let type = Reflect.getMetadata("design:type", target, propertyKey);
    if (!(value instanceof type)) {
        throw new TypeError("Invalid type.");
    }
    set(value);
}
}

```

TypeScript 编译器可以通过 `@Reflect.metadata` 装饰器注入设计阶段的类型信息。你可以认为它相当于下面的 TypeScript:

```

class Line {
    private _p0: Point;
    private _p1: Point;

    @validate
    @Reflect.metadata("design:type", Point)
    set p0(value: Point) { this._p0 = value; }
    get p0() { return this._p0; }

    @validate
    @Reflect.metadata("design:type", Point)
    set p1(value: Point) { this._p1 = value; }
    get p1() { return this._p1; }
}

```

注意 装饰器元数据是个实验性的特性并且可能在以后的版本中发生破坏性的改变 (breaking changes)。

Mixins

介绍

除了传统的面向对象继承方式，还流行一种通过可重用组件创建类的方式，就是联合另一个简单类的代码。你可能在 Scala 等语言里对 mixins 及其特性已经很熟悉了，但它在 JavaScript 中也是很流行的。

混入示例

下面的代码演示了如何在 TypeScript 里使用混入。后面我们还会解释这段代码是怎么工作的。

```
// Disposable Mixin
class Disposable {
  isDisposed: boolean;
  dispose() {
    this.isDisposed = true;
  }
}

// Activatable Mixin
class Activatable {
  isActive: boolean;
  activate() {
    this.isActive = true;
  }
  deactivate() {
    this.isActive = false;
  }
}

class SmartObject implements Disposable, Activatable {
  constructor() {
    setInterval(() => console.log(this.isActive + " : " + this.isDisposed), 500);
  }

  interact() {
    this.activate();
  }

  // Disposable
  isDisposed: boolean = false;
  dispose: () => void;
  // Activatable
  isActive: boolean = false;
  activate: () => void;
```



```

    deactivate: () => void;
}
applyMixins(SmartObject, [Disposable, Activatable]);

let smartObj = new SmartObject();
setTimeout(() => smartObj.interact(), 1000);

//////////
// In your runtime library somewhere
//////////

function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        });
    });
}

```

理解这个例子

代码里首先定义了两个类，它们将做为 **mixins**。可以看到每个类都只定义了一个特定的行为或功能。稍后我们使用它们来创建一个新类，同时具有这两种功能。

```

// Disposable Mixin
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }
}

// Activatable Mixin
class Activatable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
}

```

```
    deactivate() {  
        this.isActive = false;  
    }  
}
```

下面创建一个类，结合了这两个 mixins。下面来看一下具体是怎么操作的：

```
class SmartObject implements Disposable, Activatable {
```

首先应该注意到的是，没使用 **extends** 而是使用 **implements**。把类当成了接口，仅使用 Disposable 和 Activatable 的类型而非其实现。这意味着我们需要在类里面实现接口。但是这是我们在用 mixin 时想避免的。

我们可以这么做来达到目的，为将要 mixin 进来的属性方法创建出占位属性。这告诉编译器这些成员在运行时是可用的。这样就能使用 mixin 带来的便利，虽说需要提前定义一些占位属性。

```
// Disposable  
isDisposed: boolean = false;  
dispose: () => void;  
// Activatable  
isActive: boolean = false;  
activate: () => void;  
deactivate: () => void;
```

最后，把 mixins 混入定义的类，完成全部实现部分。

```
applyMixins(SmartObject, [Disposable, Activatable]);
```

最后，创建这个帮助函数，帮我们做混入操作。它会遍历 mixins 上的所有属性，并复制到目标上去，把之前的占位属性替换成真正的实现代码。

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {  
    baseCtors.forEach(baseCtor => {  
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {  
            derivedCtor.prototype[name] = baseCtor.prototype[name];  
        })  
    });  
}
```

三斜线指令

三斜线指令是包含单个 XML 标签的单行注释。注释的内容会做为编译器指令使用。

三斜线指令 仅可放在包含它的文件的最顶端。一个三斜线指令的前面只能出现单行或多行注释，这包括其它的三斜线指令。如果它们出现在一个语句或声明之后，那么它们会被当做普通的单行注释，并且不具有特殊的涵义。

```
/// <reference path="..." />
```

/// <reference path="..." /> 指令是三斜线指令中最常见的一种。它用于声明文件间的依赖。

三斜线引用告诉编译器在编译过程中要引入的额外的文件。

当使用 `--out` 或 `--outFile` 时，它也可以做为调整输出内容顺序的一种方法。文件在输出文件内容中的位置与经过预处理后的输入顺序一致。

预处理输入文件

编译器会对输入文件进行预处理来解析所有三斜线引用指令。在这个过程中，额外的文件会加到编译过程中。

这个过程会以一些根文件开始；它们是在命令行中指定的文件或是在 `tsconfig.json` 中的 `"files"` 列表里的文件。这些根文件按指定的顺序进行预处理。在一个文件被加入列表前，它包含的所有三斜线引用都要被处理，还有它们包含的目标。三斜线引用以它们在文件里出现的顺序，使用深度优先的方式解析。

一个三斜线引用路径是相对于包含它的文件的，如果不是根文件。

错误

引用不存在的文件会报错。一个文件用三斜线指令引用自己会报错。

使用`--noResolve`

如果指定了`--noResolve`编译选项，三斜线引用会被忽略；它们不会增加新文件，也不会改变给定文件的顺序。

`/// <reference types="..." />`

与`/// <reference path="..." />`指令相似，这个指令是用来声明依赖的；一个`/// <reference path="..." />`指令声明了对`@types`包的一个依赖。

在声明文件里包含`/// <reference types="node" />`，表明这个文件使用了`@types/node/index.d.ts`里面声明的名字；并且，这个包要在编译阶段与声明文件一起被包含进来。

解析`@types`包的名字的过程与解析`import`语句里模块名的过程类似。所以可以简单的把三斜线类型引用指令想像成针对包的`import`声明。

仅当在你需要写一个`d.ts`文件时才使用这个指令。

对于那些在编译阶段生成的声明文件，编译器会自动地添加`/// <reference types="..." />`；当且仅当结果文件中使用了引用的`@types`包里的声明时才会在生成的声明文件里添加`/// <reference types="..." />`语句。

若要在`.ts`文件里声明一个对`@types`包的依赖，使用`--types`命令行选项或在`tsconfig.json`里指定。查看[在tsconfig.json里使用@types, typeRoots和types](#)了解详情。

```
/// <reference no-default-lib="true"/>
```

这个指令把一个文件标记成默认库。你会在 `lib.d.ts` 文件和它不同的变体的顶端看到这个注释。

这个指令告诉编译器在编译过程中不要包含这个默认库（比如，`lib.d.ts`）。这与在命令行上使用 `--noLib` 相似。

还要注意，当传递了 `--skipDefaultLibCheck` 时，编译器只会忽略检查带有 `/// <reference no-default-lib="true"/>` 的文件。

```
/// <amd-module />
```

默认情况下生成的 AMD 模块都是匿名的。但是，当一些工具需要处理生成的模块时会产生问题，比如 `r.js`。

`amd-module` 指令允许给编译器传入一个可选的模块名：

`amdModule.ts`

```
///<amd-module name='NamedModule'/>
export class C {
}
```

这会将 `NamedModule` 传入到 `AMDdefine` 函数里：

`amdModule.js`

```
define("NamedModule", ["require", "exports"], function (require, exports) {
    var C = (function () {
        function C() {
        }
        return C;
    })();
    exports.C = C;
});
```

/// <amd-dependency />

注意：这个指令被废弃了。使用 `import "moduleName";` 语句代替。

`/// <amd-dependency path="x" />` 告诉编译器有一个非 TypeScript 模块依赖需要被注入，做为目标模块 `require` 调用的一部分。

`amd-dependency` 指令也可以带一个可选的 `name` 属性；它允许我们为 `amd-dependency` 传入一个可选名字：

```
/// <amd-dependency path="legacy/moduleA" name="moduleA"/>
declare var moduleA:MyType
moduleA.callStuff()
```

生成的 JavaScript 代码：

```
define(["require", "exports", "legacy/moduleA"], function (require, exports, moduleA) {
    moduleA.callStuff()
});
```