TypeScript

JavaScript Development Guide

中文译本

Nicholas Brown

TypeScript

JavaScript Development Guide

By Nicholas Brown

免责声明

虽然尝试了一切办法来验证本书中提供的信息，但作者对于其中可能包含的错误，遗漏或相反的解释不承担任何责任。本书中提供的信息仅用于教育和娱乐目的。读者对自己的行为负责，作者对使用此信息造成的任何真实或错觉，责任或损害不承担任何责任。

翻译免责声明

本书的翻译未经作者本人授权，翻译结果仅用于学习交流目的，不得用于任何商业目的。以任何形式传播、复制该翻译结果导致的后果，本人不承担任何责任。

DLR

目录

介绍

对于大型应用——特别是 web 应用——的开发一直有很大的需求，背后的原因是应用中需要实现的功能不断的增加，这就要求我们学习如何开发大型应用。这只能通过支持开发大型应用的编程语言来完成。TypeScript 就是其中一种这样的语言，它支持创建大型的应用。这本书是您学习如何用 TypeScript 编程的绝佳的方式。尽情享受阅读吧！

第 1 章．TypeScript 简介

TypeScript 是微软开发和维护的一种编程语言，它是一个开源的语言。该语言是 JavaScript 的超集，因为它为其添加了一些额外的功能，例如支持面向对象编程和可选静态类型。该语言是为了创建大型应用程序而开发的，它总是编译成 JavaScript。你可以使用 TypeScript 来为客户端和服务器端开发 JavaScript 应用程序。由于我们已经说过 TypeScript 是 javaScript 的超集，所以用 JavaScript 编写的任何程序也可以被认为是一个 TypeScript 程序。TypeScript 编译器也是用 TypeScript 编写的。

TypeScript 的目的是为开发人员提供一种开发大型应用的简单机制。它包含类，继承，接口，模块等概念，这使得开发人员创建大型应用程序变得容易。

第 2 章．TypeScript 安装

有两种方法可以在我们的系统中安装 TypeScript。这些包括以下：
1.通过使用 npm（节点包管理器）工具安装。
2.通过为 TypeScript 安装 Visual studio 插件。

对于 Windows 用户，您可以使用第二个选项，因为它可能更方便。您只需安装 Visual Studio，然后下载并安装用于在 TypeScript 中编程必要的插件。与使用其他代码编辑器相比，这个选项更加优越。

对于使用 Windows 以外平台的用户，只需确保您系统具有文本编辑器，浏览器和 TypeScript 的 npm 软件包，就能够使用 TypeScript 编程了。

可以执行以下步骤来完成安装：

1.首先安装 Node Package Manager（npm）。只需打开终端，执行下面给出的命令：

```
$ curl http://npmjs.org/install.sh | sh
$ npm —version
1.1.70
```

2.在下一步中，安装 TypeScript npm 软件包，这个需要在命令行中进行全局安装。 以下命令可用于执行这个操作：

```
$ npm install -g typescript
$ npm view typescript version
npm http GET https://registry.npmjs.org/typescript
npm http 304 https://registry.npmjs.org/typescript
0.8.1-1
```

确保您使用最新版本的浏览器，比如 Chrome 浏览器。 任何文本编辑都是可以的，比如 Sublime 文本编辑器。

HelloWorld 实例

现在我们已经设置好了编程的环境，我们可以开始编写程序了。 我们将从创建 Hello world 示例开始。请记住，TypeScript 是 ES5 的超集，并且比 ES6 具有更多的功能。TypeScript 最后也被编译成 JavaScript。

我们要创建我们的 index.html 文件，然后在里面引用外部文件。以下是文件“index.html”的代码：

```
<!doctype html>
<html lang=“en”>
<head>
<meta charset=“UTF-8”>
<title>This is TypeScript</title>
</head>
<body>
<script src=“hello.ts”></script>
```

```
</body>
</html>
```

如上面的代码所示，我们已在 index.html 文件中的引用代码文件“hello.ts”。这已经在脚本源中的“<script src =”hello.ts“> </ script>行中完成。件名中的扩展名“.ts”表示它是一个 TypeScript 文件。

现在，创建一个新文件并给它命名为“hello.ts”。然后将以下代码行添加到文件中：

```
alert(‘hello world, this is TypeScript!’);
```

之后，只需启动命令行，然后导航到具有文件“hello.ts”的文件夹。通过执行以下命令编译文件：

```
tsc hello.ts
```

TypeScript 中的 tsc 命令表示 TypeScript 编译器，它将生成一个名称为“hello.js”的新文件。从文件扩展名可以看出，我们将把 TypeScript 转换为 JavaScript 文件，这在 TypeScript 编程中是经常需要做的事情。

第 3 章．类型注释

此功能在 TypeScript 中是可选的，它允许我们在我们的 TypeScript 程序中表示和实现 indentation[1]。我们通过创建一个用来计算矩形面积的程序来证明这一点。 只需打开您的文本编辑器并创建一个名为“area.ts.”的新文件。将以下代码添加到文件中：

```
function area(shape: string, width: number, height: number) {
var area = width * height;
return “The shape is ” + shape + ” having an area of ” + area + ” cm squared.”;
} d
ocument.body.innerHTML = area(“rectangle”, 20, 15);
```

接下来，打开文件“index.html”，将引入的脚本源从“type.ts”更改为“area.ts”。打开终端，然后执行以下命令，以编译整个代码：

```
tsc area.ts
```

页面将显示与矩形区域相关的信息。 想必你肯定注意到，类型注释被传递给函数的参数，它们用于指示该函数支持的参数种类。在上面的例子中，参数“shape”是一个字符串，而参数 length 和 width 是数字。

您必须知道在编译期间会强制执行类型注释和其他 TypeScript 特性。如果将其他类型的值传递给这些参数，编译器将显示编译时错误。当涉及到大型应用程序时，这是一个很好的主意。

我们需要在我们的案例中展示如何检查错误。考虑下面给出的代码：

```
function area(shape: string, width: number, height: number) {
var area = width * height;
return “The shape is ” + shape + ” having an area of ” + area + ” cm squared.”;
}
document.body.innerHTML = area(“rectangle”, “width”, 15); // wrong type of width
```

在上面的例子中，宽度作为字符串被传入，这是错误的。 编译器期望宽度应该是一个数字。在编译上面给出的程序之后，您会得到以下错误：

```
$ tsc type.ts
type.ts(6,26): Supplied parameters do not match any signature of call target
```

这就是你得到的这种错误。

但是，您一定已经意到，虽然由于类型不匹配而发生警告，但这并没有阻止 TypeScript 文件的编译和创建 JavaScript 文件。

如上述示例所示，很清楚，注释用于记录函数或变量旨在执行的功能。 假设你有一个函数，期望只有一个参数应该被传递给它，并且该参数应该是字符串类型。一旦我们传递数组，应该怎么办？ 考虑下面给出的例子：

```
function welcome(name: string) {
```

```
return “Hello, ” + name;
}
var user = [0, 1, 2];
document.body.innerHTML = wlecome(user);
```

当你编译上面这个例子的时候，你将会得到如下错误：

```
welcome.ts(7,26): Supplied parameters do not match any signature of call target
```

同样，您可以尝试从 welcome call 函数中删除所有参数。TypeScript 编译器警告你传递了不必要的参数。

第 4 章．接口

我们使用我们前面的例子并对它进行扩展，以便实现一个将形状描述为具有可选颜色属性的对象的接口。

创建一个新文件，并给它命名为“interface.ts”。打开文件“index.html”，然后修改它，使引入的脚本源为“interface.js”。将以下代码添加到文件“interface.ts”：

```
interface Shape {
name: string;
width: number;
height: number;
color?: string;
}
function area(shape : Shape) {
var area = shape.width * shape.height;
return “The shape is ” + shape.name + ” having an area of ” + area + ” cm
squared”;
}
console.log( area( {name: “rectangle”, width: 20, height: 15} ) );
console.log( area( {name: “square”, width: 20, height: 25, color: “blue”} ) );
```

接口是给予对象类型的名称。 我们可以声明接口，同时在我们的类型注释中使用它们。

当你编译完“interface.js”文件，它不会有报任何错误。为了让它给我们报告一个错误，让我们向文件“interface.js”添加一行新的代码，该文件中的形状没有 name 属性，然后在浏览器的控制台中观察结果。 将下面给出的代码行添加到文件“interface.js”中：

```
console.log( area( {width: 20, height: 15} ) );
```

然后，您可以运行命令“tsc interface.js”，以便编译文件中的代码。一个错误将会输出，但现在不要担心。刷新浏览器，然后观察控制台给您的内容。文本和矩形的面积将在控制台上输出。错误应该如下：

```
interface.ts(26,13): Supplied parameters do not match any signature of call target:
Could not apply type ‘Shape’ to argument 1, which is of type ‘{ width: number;
height: number; }’
```

上面显示的错误是因为传递给函数“area（）”的对象不符合名为“shape”的接口，为了让它符合，你需要传递一个 name 属性。

考虑下面给出的代码，它显示了如何在 TypeScript 中使用接口：

```
interface Person {
    fName: string;
    lName: string;
}
```

```
function welcome(person: Person) {
    return “Hello, ” + person.fName + ” ” + person.lName;
}
var user = { fName: “Jane”, lName: “User” };
document.body.innerHTML = welcome(user);
```

在上面的例子中，我们已经实现了一个用来描述个人姓名的接口。对于 JavaScript 中兼容的两种类型，您必须确保其内部兼容。 因此，我们将能够通过使用接口所需的样子来实现一个接口，我们不需要使用一个明确的“implements”子句。

第 5 章．箭头函数表达式

要完全了解关键字“this”的作用域是非常具有挑战性的。 TypeScript 支持使用箭头函数和表达式，以便使您更容易处理“this”。 这也正好是 ES6 首次引入的新功能。在箭头函数中，关键字“this”的值被保留，我们很容易编写和使用回调。考虑下面给出的示例代码：

```
var objectShape = {
  name: “rectangle”,
  popup: function() {
    console.log(‘The inside popup(): ‘ + this.name);
    setTimeout(function() {
        console.log(‘The inside setTimeout(): ‘ + this.name);
        console.log(“The shape is a ” + this.name + “!”);
    }, 3000);
  }
};
objectShape.popup();
```

“this.name”将为空，这将在浏览器控制台上清楚地描述。

可以使用 TypeScript 箭头函数来解决这个问题。您必须用“（）=>”替换“function（）”。这在下面给出的代码中展示：

```
var objectShape = {
  name: “rectangle”,
  popup: function() {
    console.log(‘The inside popup(): ‘ + this.name);
    setTimeout( () => {
        console.log(‘The inside setTimeout(): ‘ + this.name);
        console.log(“The shape is a ” + this.name + “!”);
    }, 3000);
  }
};
objectShape.popup();
```

然后，您可以查看生成的 JavaScript 文件。 您将观察到编译器已经注入了一个新的变量，即“var _this = this”，这将在回调函数“setTimeout（）”中使用，“name”属性将被引用。

考虑下面给出的纯 JavaScript 类：

```
function Person(age) {
  this.age = age
  this.becomeOld = function() {
    this.age++;
  }
}
```

```
var person = new Person(1);
setTimeout(person.becomeOld,1000);
setTimeout(function() { console.log(person.age); },2000); // the 1 should have been set to 2
```

一旦在浏览器中执行了上述代码，对象“this”将指向“window”，因为它将是用于执行“becomeOld”函数的对象。 这只能通过执行如下所示的箭头函数来解决：

```
function Person(age) {
  this.age = age
  this.becomeOld = () => {
    this.age++;
  }
}
var person = new Person(1);
setTimeout(person.becomeOld,1000);
setTimeout(function() { console.log(person.age); },2000); // 2
```

上述代码背后的原因在于，引用对象“this”将从我们的函数体外面捕获。 上面的代码类似于下面给出的 JavaScript 代码：

```
function Person(age) {
  this.age = age
  var _this = this; // capture the this
  this.becomeOld = function() {
    _this.age++; // using the already captured this
  }
}
var person = new Person(1);
setTimeout(person.becomeOld,1000);
setTimeout(function() { console.log(person.age); },2000); // 2
```

由于我们使用 TypeScript，我们可以通过语法糖来组合类和箭头函数。这在下面给出的示例代码中展示：

```
class Person {
  constructor(public age:number) {}
  becomeOld = () => {
    this.age++;
  }
}
var person = new Person(1);
setTimeout(person.becomeOld,1000);
setTimeout(function() { console.log(person.age); },2000); // 2
```

但是，您需要知道，只有将函数赋给另外一个变量去调用的时候箭头函数才应该使用。如下所示：

```
var becomeOld = person.becomeOld;
// This will be called by someone else later:
becomeOld();
```

如果你直接调用它，你应该这样：

```
person.becomeOld();
```

“this”对象将被设置为正确的上下文。

在大多数情况下，库会使用对象“this”，以便传递当前正在迭代的对象。 如果您需要访问库传递的“this”，或者是周围的变量，也可以使用像在没有箭头函数的情况下使用的临时变量。这在下面给出的代码中显示：

```
let _self = this;
thing.each(function() {
  console.log(_self); // this is your lexically scoped value
  console.log(this); // this is a library passed value
});
```

第 6 章. 类

在 TypeScript 中，有一些类具有公共或私有访问属性。TypeScript 中类的使用方式与 ES6 相同。创建一个新文件，并给它名称“class.ts”。将以下代码添加到类中：

```
class Shape {
    area: number;
    color: string;
    constructor ( name: string, width: number, height: number ) {
        this.area = width * height;
        this.color = “red”;
    };
    showit() {
        return “The object is” + this.color + ” ” + this.name + ” having an area of ” +
        this.area + ” cm squared.”;
    }
}
var square = new Shape(“square”, 20, 20);
console.log( square.showit() );
console.log( ‘Area of Object: ‘ + square.area );
console.log( ‘Name of Object: ‘ + square.name );
console.log( ‘Color of Object: ‘ + square.color );
console.log( ‘Width of Object: ‘ + square.width );
console.log( ‘Height of Object: ‘ + square.height );
```

我们上面的例子中的对象有两个属性，即“area”和“color”。我们还实现了一个构造函数和方法“showit()”。我们的构造函数参数作用域限于该构造函数。因此，浏览器和编译器都会出现错误，如下所示：

```
class.ts(12,42): The ‘name’property does not exist on the value of type ‘Shape’
class.ts(20,40): The ‘name’property does not exist on the value of type ‘Shape’
class.ts(22,41): The ‘width’property does not exist on the value of type ‘Shape’
class.ts(23,42): The ‘height’property does not exist on the value of type ‘Shape’
```

TypeScript 支持“公共”和“私有”访问修饰符。对于公共成员，我们可以从各地访问它们，而对于私有成员，我们只能在类中访问它们，没有可以用于强制隐私的功能，这就是为什么在编译期间被强制执行，并且用于警告开发者他们打算将其作为私有成员。

我们需要向构造函数参数“name”添加公共可访问性修饰符，然后给“color”成员属性添加一个私有可访问性修饰符。一旦将辅助修饰符添加到构造函数的参数中，这些参数将会自动成为具有该辅助功能修饰符的类的成员。考虑下面给出的例子：

```
…
private color: string;
…
constructor ( public name: string, width: number, height: number ) {
…
```

考虑下面给出的 ES 6 类：

```
class Monster {
    constructor(name, initPosition) {
        this.name = name;
        this.initPosition = initPosition;
    }
}
```

在 ES 6 中，上述类在以定义的方式使用时将是可行的。 但是，在 TypeScript 中是不行的，因为您需要在 TypeScript 中定义对象上的某些属性。这在下面给出的代码中显示：

```
interface Point {
    a: number,
    b: number
}
class Monster {
    name: string;
    initPosition: Point
    constructor(name, initPosition) {
        this.name = name;
        this.initPosition = initPosition;
    }
}
var scaring = new Monster(“Alien”, {a: 0, b: 0});
```

但是，有一个速记来完成上面的代码。以下代码会做同样的事情：

```
interface Point {
    a: number,
    b: number
}
class Monster {
    constructor(public name: string, public initPosition: Point) {

    }
}
```

```
var scaring = new Monster(“Alien”, {a: 0, b: 0});
```

您一定已经注意到，构造函数在每个变量名称之前使用关键字“public”。这表明我们需要使参数成为我们对象的公共属性。

继承

在 TypeScript 中，“extends”关键字用于支持单一继承。下面的代码演示了如何做到这一点：

```
class Point3 extends Point {
    z: number;
    constructor(x: number, y: number, z: number) {
        super(x, y);
        this.z = z;
    }
    add(point: Point3) {
        var point2 = super.add(point);
        return new Point3(point2.x, point2D.y, this.z + point.z);
    }
}
```

如果你的父类中有一个构造函数，你将必须从子类的构造函数调用父构造函数。这将确保与设置有关的工作正确完成（意思就是实现继承父亲的成员）。在调用“super”之后，可以添加任何子类构造函数所需的其他东西。

考虑下面给出的第二个例子，它使用“extends”关键字在 TypeScript 中实现继承：

```
class MyReport {
    name: string;
    constructor (name: string) {
        this.name = name;
    }
    print() {
        alert(“My Report: ” + this.name);
    }
}
class CashReport extends Report {
    constructor (name: string) {
        super(name);
    }
    print() {
        alert(“Cash Report: ” + this.name);
    }
```

```
    getLineItems() {
        alert("5 line items");
    }
}
var myreport = new CashReport("Month's Sales");
myreport.print();
myreport.getLineItems();
```

在上面的例子中，我们创建了一个名为“MyReport”的基类，它有一个成员“name”，一个用于接受字符串 name 作为参数的构造函数和一个名为“print（）”的函数。“类”CashReport“使用”extends“关键字继承自”MyReport“类。这意味着子类将可以直接访问已经在基类中定义的”print（）“函数。 类“CashReport”将创建自己的“print（）”方法并覆盖基类中的“print（）”方法。 类“CashReport”也将调用“super（）”获得其在基类构造函数中接收的 name 值。

静态

TypeScript 支持静态属性，这些属性由类的所有实例共享（**译者注：确实是实例中共享，但是实例是不能直接拿到的，都是通过类名拿到的静态属性**）。 放置这些静态属性的最好的位置是在类的定义中，这正是在 TypeScript 中所做的。请考虑以下代码，其中显示了这是如何发生的：

```
class MyClass {
    static instances = 0;
    constructor() {
        MyClass.instances++;
    }
}
var s1 = new MyClass ();
var s2 = new MyClass ();
console.log(MyClass.instances); // 2
```

一个类可以同时具有静态成员和静态函数。

访问修饰符

在 TypeScript 中，支持访问修饰符，它们用于控制变量如何被访问。一个变量可以在类之外直接通过它的实例访问，也可能在子类中访问。以下是访问修改器的三种类型：

1. public: 通过实例访问.
2. private: 类外面不允许访问.

3. protected: 子类中可以访问，不允许通过实例访问.

它们对你的程序的编写可能没有什么大的影响，但是如果你在运行时尝试访问它们，会给你一些错误。 考虑下面给出的示例，其中显示了这一点：

```
class ClassBase {
    public a: number;
    private b: number;
    protected c: number;
}
//EFFECT ON the INSTANCES
var myVar = new ClassBase();
foo.a; // okay
foo.b; // ERROR : private
foo.c; // ERROR : protected
// EFFECT ON THE CHILD CLASSES
class ClassChild extends ClassBase {
    constructor() {
        super();
        this.a; // okay
        this.b; // ERROR: private
        this.c; // okay
    }
}
```

您需要知道，上述修饰符同时适用于成员属性和成员函数。

使用构造函数定义成员

类中的成员可以按下面的方式初始化：

```
class MyClass {
    x: number;
    constructor(x:number) {
        this.x = x;
    }
}
```

这在 TypeScript 中是一个非常常见的场景，你也可以简单的在成员前面加一个访问修饰符前缀，这样它将在类中自动声明，然后从构造函数中拷贝赋值。有了这个，我们以前的例子可以写成如下：

```
class MyClass {
    constructor(public x:number) {
```

```
    }
}
```

成员属性初始化

这是 TypeScript 中支持的一个非常重要的功能。您可以在构造函数之外初始化类中的任何成员，并提供默认值。考虑下面给出的例子：

```
class MyClass {
    members = []; // Initializing directly
    add(x) {
        this.members.push(x);
    }
}
```

剩余参数

这个功能允许在函数中接受多个参数，然后以一个数组的形式获取它们。 这可以如下面给出的示例表示：

```
Function allFunction(first, second, …rest) {
    console.log(rest);
}
allFunction (‘sample’, ‘bar’); // []
allFunction (‘sample’, ‘bar’, ‘bas’, ‘qux’); // [‘bas’,‘qux’]
```

第 7 章. 解构

解构意味着打破结构。 TypeScript 支持以下类型的解构：
1、对象解构
2、数组解构

简单起见，可以将解构视为结构化的反向。下面给出的示例显示了如何在 JavaScript 中完成结构化：

```
var myVar = {
    bar: {
        bas: 789
    }
};
```

有了 JavaScript 对结构化的支持，创建新对象的过程将变得非常容易。有了解构的支持，我们也可以方便地从特定来源获取数据。

对象解构

有了解构，人们可以将以前只能在许多条代码中处理的事情放在一条代码中完成。考虑下面给出的例子：

```
var rectangle = { a: 0, b: 10, width: 20, height: 30 };
// Destructuring the assignment
var {a, b, width, height} = rectangle;
console.log(a, b, width, height);
```

如果我们不使用解构，则 a，b，width 和 height 将从 rectangle 中一个个提取。也可以通过使用解构从数据结构获得深层次的数据。以下代码最好地说明了如何做到这一点：

```
var foo = { bar: { bas: 123 } };
var {bar: {bas}} = foo; // Effectively `var bas = foo.bar.bas;`
```

数组解构

译者注：下面这些与“数组解构”主题不符，可能是原作者的疏忽

有了解构，人们可以将以前只能在许多条代码中处理的事情放在一条代码中

完成。考虑下面给出的例子：

“const”是新增的，现在在 TypeScript 和 ES6 中提供。有了它，变量可以是不可变的。 这在运行时和文档中都是一个很好的目的。为了使用它，你只需要用“const”替换关键字“var”。例子如下：

```
const myConst = 123;
```

与其他编程语言中使用的 const 相比，在 TypeScript 中使用 const 很容易。它有助于我们避免使用魔术文字，这是提高可读性和可维护性的良好做法。考虑下面给出的代码：

```
// Low readability
if (a > 10) {
} //
This one is Better!
const maximumRows = 10;
if (a > maximumRows) {
}
```

声明 const 常量时必须给它初始化，否则将会报错：

```
const myConst // ERROR: const declarations must be initialized
```

此外，将一个新的值赋给它们是不可能的。这是因为在创建常量之后，它们变得不可变，所以如果你尝试将一个新的值分配给它们，你会得到一个编译错误。这在下面给出的代码中展示：

```
const myConst = 123;
myConst = 540; // ERROR: Left-hand side of an assignment expression cannot be a constant
```

如上所示，尝试这样做会导致已经提过的编译错误。

块级作用域

“const”是一个已经被限定的块，就像我们使用“let”一样。以下给出的代码描述了这一点：

```
const myConst = 123;
if (true) {
    const myConst = 500; // Allowed as its a new variable limited to this `if` block
}
```

深度不可变

我们可以使用“const”用于对象字面量，只要你所关心的是保护变量的“引用”。 如下图所示：

```
const myConst = { bar: 123 };
myConst = { bar: 500 }; // ERROR : Left hand side of an assignment expression
cannot be a constant
```

但是，您可以改变对象的子属性。这在下面给出的代码中展示：

```
const myConst = { bar: 123 };
myConst.bar = 500; // This one is Allowed!
console.log(myConst); // { bar: 500 }
```

也就是说引用（指针）不可被修改，但引用的那个地址的数据可以被修改。这就是为什么建议您将 const 用于不可变数据结构和字面量。当您需要维护变量的值而不更改变量时，常量是非常好的。

第 8 章．for...of 遍历

“for ... in”不会遍历数组的元素，这是 JavaScript 初学者遇到的常见问题。“for ... of”是用来遍历传入的对象的键。考虑下面给出的示例，很好地描述了这一点：

```
var myArray = [7, 2, 5];
for (var item in myArray) {
    console.log(item); // 0,1,2
}
```

在上面的例子中，虽然我们希望得到 7,2,5，但实际我们将得到索引 0,1,2。这就是为什么在“TypeScript”中引入了“for ...of”。 考虑下面给出的示例，其将遍历数组的元素并输出预期的结果：

```
var myArray = [7, 2, 5];
for (var item of myArray) {
    console.log(item); // 7,2,5
}
```

以上示例将输出您期望的结果，而不是输出数组索引。

类似地，在 TypeScript 中，通过使用“for ... of”的遍历字符串中的字符并不麻烦，这在下面给出的代码中有所展示：

```
var hi = “are you the one looking for me?”;
for (var char of hi) {
    console.log(char); // are you the one looking for me?
}
```

JS 版本

TypeScript 将始终为 ES6 之前版本目标生成标准“for（var i = 0; i <list.length; i ++）”。 我们上面的例子将产生以下内容：

```
var myArray = [7, 2, 5];
for (var item of myArray) {
    console.log(item);
}
//it will become //
for (var _j = 0; _j < myArray.length; _j++) {
    var item = myArray[_j];
    console.log(item);
```

```
}
```

如上述示例所示，使用“for ...of”将使“意图”更清晰，并且编写的代码量以及创建的变量数也将减少。

限制

对于 ES6 以下的版本，TypeScript 生成的 JS 代码将假定属性“length”存在于我们的对象上，并且该对象可以通过数字进行索引。 这意味着它只支持在传统 JS 引擎的数组和字符串中使用。

如果 TypeScript 注意到您没有使用数组或字符串，那么您将收到错误报告。考虑下面给出的例子：

```
let artParagraphs = document.querySelectorAll(“article > p”);
// Error: Nodelist is not an array type or a string type
for (let par of artParagraphs) {
    par.classList.add(“read”);
}
```

从上述演示中，很明显“for ... of”只能用于数组或字符串对象。

第 9 章．迭代器

迭代器是一种主要用于面向对象编程语言的特性。它是一个对象，用于实现以下类型的接口：

```
interface Iterator<T> {
    next(value?: any): IteratorResult<T>;
    return?(value?: any): IteratorResult<T>;
    throw?(e?: any): IteratorResult<T>;
}
```

该接口允许我们从属于对象的集合或序列中检索特定的值。想象一下，你有一个框架对象的情况，这个框架对象包含我们的框架组件的列表。使用迭代器接口，我们可以从框架对象中检索组件，如下面给出的代码所示：

```
‘use strict’;
class Component {
    constructor (public name: string) {}
}
class Frame implements Iterator<Component> {
    private point = 0;
    constructor(public name: string, public components: Component[]) {}
    public next(): IteratorResult<Component> {
        if (this.point < this.components.length) {
            return {
            done: false,
            value: this.components[this.point++]
            }
        }
        else {
            return {
            done: true
            }
        }
    }
}
let fr = new Frame(“Door”, [new Component(“top”), new Component(“bottom”),
new Component(“left”), new Component(“right”)]);
let iteratorResult_1 = fr.next(); //{ done: false, value: Component { name: ‘top’} }
let iteratorResult_2 = fr.next(); //{ done: false, value: Component { name: ‘bottom’} }
let iteratorResult_3 = fr.next(); //{ done: false, value: Component { name: ‘left’} }
let iteratorResult_4 = fr.next(); //{ done: false, value: Component { name: ‘right’} }
let iteratorResult_5 = fr.next(); //{ done: true }
```

```
//It is possible for us to access a value of iterator result by use of the value property:
let component = iteratorResult_1.value; //Component { name: 'top'}
```

您必须知道迭代器本身不是 TypeScript 中的一个功能，这意味着无需显式实现迭代器和迭代结果,代码就可以工作。然而，为了代码的一致性，对我们来说使用 ES6 迭代器接口特性是有好处的。

考虑下面给出的代码，其中显示了如何做到这一点：

```
class Frame implements Iterable<Component> {
    constructor(public name: string, public components: Component[]) {}
    [Symbol.iterator]() {
        let p = 0;
        let components = this.components;
        return {
            next(): IteratorResult<Component> {
                if (p < components.length) {
                  return {
                      done: false,
                      value: components[p++]
                      }
                } else {
                    return {
                    done: true
                    }
                }
            }
        }
    }
}
let fr = new Frame("Door", [new Component("top"), new Component("bottom"),
    new Component("left"), new Component("right")]);
for (let cmp of frame) {
    console.log(cmp);
}
```

迭代器不能迭代有无限值。 一个很好的例子就是 Finonacci 序列。这在下面给出的代码中演示：

```
class Fibonacci implements IterableIterator<number> {
    protected fbn1 = 0;
    protected fbn2 = 1;
    constructor(protected maxValue?: number) {}
    public next(): IteratorResult<number> {
        var current = this.fbn1;
```

```
            this.fbn1 = this.fbn2;
            this.fbn2 = current + this.fbn1;
            if (this.maxValue && current <= this.maxValue) {
                return {
                    done: false,
                    value: current
                }
            } else {
                return {
                    done: true
                }
            }
            [Symbol.iterator](): IterableIterator<number> {
                return this;
            }
        }
}
let fib = new Fibonacci();
fib.next() //{ done: false, value: 0 }
fib.next() //{ done: false, value: 1 }
fib.next() //{ done: false, value: 1 }
fib.next() //{ done: false, value: 2 }
fib.next() //{ done: false, value: 3 }
fib.next() //{ done: false, value: 5 }
let fibMax50 = new Fibonacci(50);
console.log(Array.from(fibMax50)); // [ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ]
let fibMax21 = new Fibonacci(21);
for(let num of fibMax21) {
    console.log(num); //will print a fibonacci sequence of 0 to 21
}
```

第 10 章．模板字符串

模板字符串使用反引号而不是单引号或双引号。他们主要用在三种场景：

1、多行字符串
2、字符串内插
3、Tagged 模板

多行字符串

有时，可能需要在 JavaScript 字符串中嵌入某些内容。 为了做到这一点，您需要使用正确的转义字符，以便手动将新行放在字符串中。 这在下面给出的代码中展示：

```
var lcs = “You should never give up \
\nAll shall be well with you friend”;
```

在 TypeScript 中，可以使用如下所示的模板字符串来完成：

```
var lcs = ` You should never give up
All shall be well with you friend`;
```

字符串内插

您可能需要从静态字符串或一些变量生成字符串。达到这个目的需要一些模板逻辑，这也正是模板字符串的名称的由来。一个 html 字符串可以如下生成：

```
var lcs = ‘ You should never give up ‘;
var html = ‘<div>’ + lcs + ‘</div>’;
```

如果使用模板字符串的话，你可以像这样写：

```
var lcs = ‘ You should never give up ‘;
var html = `<div>${lcs}</div>`;
```

由于插值内的占位符被视为表达式，因此，和上面类似，您可以执行以下数学表达式：

```
console.log(`1 and 1 will make ${1 + 1}`);
```

tagged 模板

一个函数可以放在一个模板字符串之前，这将为它提供一个预处理模板字符串文字以及占位符表达式然后返回一个结果的机会。你必须知道，所有的静态文字应该作为数组传入第一个参数。

考虑下面的例子：

```
var words = “if you are a man> you wear trousers”;
var html = htmlEscape `<div> I would like to denote that : ${words}</div>`;
// an example of tag function
function htmlEscape(literals, …placeholders) {
    let result = ””;
    // interleave your literals with the placeholders
    for (let j = 0; j < placeholders.length; j++) {
        result += literals[j];
        result += placeholders[j]
        .replace(/&/g, ‘&amp;’)
        .replace(/”/g, ‘&quot;’)
        .replace(/’/g, ‘&#39;’)
        .replace(/</g, ‘&lt;’)
        .replace(/>/g, ‘&gt;’);
    }
    //adding the last literal
    result += literals[literals.length - 1];
    return result;
}
```

第 11 章．展开运算符

该操作符旨在展开数组对象。

Apply

该运算符用于将数组展开到函数参数中。在过去，有人会使用函数“Function.prototype.apply”来实现。这在下面的代码示例中展示：

```
function myFunction(a, b, c) { }
var args = [0, 1, 2];
myFunction.apply(null, args);
```

为了使用展开运算符做到这一点，可以用（...）的参数前缀。如下图所示：

```
function myFunction(a, b, c) { }
var args = [0, 1, 2];
myFunction(…args);
```

Destructuring

你已经见过展开运算符用于解构，下面展示了：

```
var [a, b, …remaining] = [1, 2, 3, 4];
console.log(a, b, remaining); // 1, 2, [3,4]
```

这么做的动机是在数组解构时更方便的捕获数组元素

Array Assignment

扩展运算符可以允许您将已展开的数组放置到另一个数组中。考虑下面给出的示例，这说明了如何做到这一点：

```
var list = [5, 6];
list = […list, 7, 8];
console.log(list); // [5,6,7,8]
```

第 12 章．枚举

枚举为我们提供了一种组织具有相似值的集合的方式。除 JavaScript 之外的其他编程语言为我们提供枚举数据类型。枚举也在 TypeScript 中得到支持。在 TypeScript 中，可以声明如下所示的枚举：

```
enum Card {
    Clubs,
    Spades,
    Diamonds,
    Hearts
}
//A sample usage
var card = Card.Clubs;
// Safety
card = “It is not a member of the card suit”; // Error : string is not assignable to type `CardSuit`
```

枚举及数值

TypeScfript 中的枚举是基于数值的。这表明我们可以在 TypeScript 中为枚举的实例分配数值，这也同样适用于与数值兼容的任何类型。以下给出的代码很好地描述了如何做到这一点：

```
enum Color {
    Red,
    Green,
    Blue
}
var color = Color.Red;
color = 0; // This is the same as Color.Red
```

枚举及字符串

我们想知道 TypeScript 中枚举生成的 JavaScript 长啥样。 考虑下面给出的 TypeScript 示例：

```
enum MyEnum {
    False,
    True,
    Unknown
```

```
}
```

以上 TypeScript 代码将生成以下 JavaScript 代码：

```
var MyEnum;
(function (MyEnum) {
    MyEnum [MyEnum [“False”] = 0] = “False”;
    MyEnum [MyEnum [“True”] = 1] = “True”;
    MyEnum [MyEnum [“Unknown”] = 2] = “Unknown”;
})( MyEnum || (MyEnum = {}));
```

变量“MyEnum”可用于将字符串版本的枚举转换为数值或数值版本的枚举。以下给出的代码描述了这一点：

```
enum MyEnum {
    False,
    True,
    Unknown
}
console.log(MyEnum [0]); // “False”
console.log(MyEnum [“False”]); // 0
console.log(MyEnum [MyEnum.False]); // “False” because ` MyEnum.False == 0`
```

我们也可以更改与枚举相关联的数值。默认设置是枚举为 0，并且每个后续增量以 1 的间隔自动进行。请考虑以下给出的代码，其很好地描述了这一点：

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

与枚举相关联的特定数值可以通过专门分配来自动更改。下面给出的示例显示了如何完成这一操作，我们从 3 开始，然后从 3 开始增加，如下所示：

```
enum Color {
    LightRed = 3, // 3
    DarkGreen, // 4
    LightBlue // 5
}
```

你也必须知道枚举是开放的。考虑下面给出的 JavaScript 代码，它是从枚举生成的：

```
var MyEnum;
(function (MyEnum) {
    MyEnum [MyEnum [“False”] = 0] = “False”;
    MyEnum [MyEnum [“True”] = 1] = “True”;
    MyEnum [MyEnum [“Unknown”] = 2] = “Unknown”;
})( MyEnum || (MyEnum = {}));
```

枚举定义可以分散在多个文件中。考虑下面给出的示例，其中我们将 Color 的定义分散到两个块：

```
enum Color {
    Red,
    Green,
    Blue
}
enum Color {
    LightRed = 3,
    DarkGreen,
    LightBlue
}
```

枚举为标志

标志为我们提供了使用枚举的极好的能力。考虑下面的例子：

```
enum Animal {
    None = 0,
    PossesClaws = 1 << 0,
    Flying = 1 << 1,
    ConsumesFish = 1 << 2,
    Endangered = 1 << 3
}
```

在上述情况下，我们需要通过移动 1 来获得按位分离的运算符。
考虑下面的例子：

```
enum Animal {
    None = 0,
    PossesClaws = 1 << 0,
    Flies = 1 << 1,
}

function printAnimalAbilities(animal) {
```

```
    var animFlags = animal.flags;
    if (animFlags & AnimFlags.PossesClaws) {
        console.log('The animal has claws');
    }
    if(animFlags & AnimFlags.Flies) {
        console.log('The animal can fly');
    }
    if(animFlags == AnimFlags.None) {
        console.log('nothing');
    }
}
var animal = { flags: AnimFlags.None };
printAnimalAbilities(animal); // nothing
animal.flags |= AnimFlags.PossesClaws;
printAnimalAbilities(animal); //The animal has claws
animal.flags &= ~AnimFlags.PossesClaws;
printAnimalAbilities(animal); // nothing
animal.flags |= AnimFlags.PossesClaws | AnimFlags.Flies;
printAnimalAbilities(animal); // The animal has claws and the animal it flies
```

标志可以组合起来，以便在枚举的定义中创建方便的快捷方式。 这在下面给出的代码中展示了：

```
enum Animal {
    None = 0,
    PossesClaws = 1 << 0,
    Flies = 1 << 1,
    EatsFish = 1 << 2,
    Endangered = 1 << 3,
    EndangeredFliesClawedEatFish = PossesClaws | Flies | EatsFish | Endangered,
}
```

常量枚举

假定你有定义为下面的枚举：

```
enum MyEnum {
    False,
    True,
    Unknown
}
var lie = MyEnum.False;
```

为了提高性能，您应当将枚举标记为“常量枚举”。如下面给出的代码所示：

```
const enum MyEnum {
    False,
    True,
    Unknown
}
var lie = MyEnum.False;
```

编译后上面的代码将转换成如下 JavaScript 代码：

```
var lie = 0;
```

编译器所做的是它内联了所有使用的枚举。此外，编译器不会为枚举的定义生成 JavaScript。

第 13 章．let

在 JavaScript 中，变量具有函数级的作用域。对于其他几种编程语言（如 Java 和 C＃），其变量是块级作用域的。

考虑下面给出的 JavaScript 代码：

```
var myVar = 123;
if (true) {
    var myVar = 500;
}
console.log(myVar); // 500
```

虽然你希望上面的代码打印 123，它会打印 500。这是因为我们的变量“myVar”在内部和外部的“if”块中是一样的。这是 JavaScript 中最常见的错误来源之一。在 TypeScript 中，引入了关键字“let”，以使程序员能够定义具有真正块级作用域的变量。这意味着如果使用关键字“let”而不是“var”关键字来定义变量，则在“if”块内部和外部它将被视为不同的。前面的例子可以使用“let”关键字写成如下：

```
let myVar = 123;
if (true) {
  let myVar = 500;
} c
onsole.log(myVar); // 123
```

考虑下面给出的示例，其中显示了如何在循环中使用“let”关键字：

```
var index = 0;
var myArray = [1, 2, 3];
for (let index = 0; index < myArray.length; index++) {
    console.log(myArray[index]);
}
console.log(index); // 0,0,0
```

用“let”关键字替换它可以帮助您避免循环错误。如下图所示：

```
var index = 0;
var myArray = [1, 2, 3];
for (let index = 0; index < myArray.length; index++) {
    console.log(myArray[index]);
}
console.log(index); // 1,2,3
```

如上所示，尽可能使用“let”关键字总是很好，您将避免遇到各种奇怪的错误。

函数也可用于创建新的作用范围。我们将演示如何使用一个函数在 JavaScript

中创建一个新的变量范围。考虑下面给出的代码：

```
 var myVar = 123;
function testFunction() {
    var myVar = 500;
}
testFunction();
console.log(myVar); // 123
```

上面的代码将像你想要的那样运行。 没有它，就很难得到正确的结果。

生成 JS

TypeScript 生成一个 JS 代码时，如果在周围存在类似的名称，则可以简单地重命名 let 变量。考虑下面给出的代码，它显示了使用 let 替换 var 的简单方法。这是代码：

```
if (true) {
    let myVar = 123;
}
//will become //
if (true) {
    var myVar = 123;
}
```

然而，如果变量的名称已经被周围的范围所占据，那么将生成一个新的变量名称，如下面的代码所示：

```
var myVar = '123';
if (true) {
    let myVar = 123;
}
//will become //
var myVar = '123';
if (true) {
    var _myVar = 123; // It has been Renamed
}
```

闭包中的 let

考虑下面简单的 JavaScript 代码：

```
var functions = [];
// creating a bunch of functions
for (var j = 0; j < 3; j++) {
    functions.push(function() {
        console.log(j);
```

```
        })
    }
    //call them
    for (var k = 0; k < 3; k++) {
        functions[k]();
    }
```

在大多数面试问题中，都会询问有关上述 JavaScript 文件中的输出信息。虽然大多数人都认为答案是 0,2,3，但真正的答案是 3,3,3。原因是这三个函数都是用我们外部范围的变量“j”，我们将依次执行它们后变量“j”的值将为 3。它将是第一个循环的终止条件。

这个问题的解决方案是为每个变量创建一个变量，这应变量该专门用于该循环中的迭代。如你所知，变量的新范围可以通过创建一个新的函数然后立即执行。这在下面给出的代码中显示[1]：

```
var functions = [];
// creating a bunch of functions
for (var j = 0; j < 3; j++) {
    (function() {
        var local = j;
        functions.push(function() {
            console.log(local);
        })
    })();
}
//calling them
for (var k = 0; k < 3; k++) {
    functions [k]();
}
```

一旦使用 let 关键字而不是 var 关键字，就会创建一个唯一的为每个循环迭代的变量。

总结

我们为本指南做一个总结。TypeScript 是一种编程语言，它总是编译成 JavaScript。 该语言是开源的，由 Microsoft 开发和维护。它有利于开发大型应用程序，支持静态类型和面向对象编程等功能。这两个功能是适合用于开发大型应用程序的主要功能。

该语言支持类型注释，这对于维护程序中的缩进非常有用。 缩进通常确保程序的可读性很好。

您必须记住，TypeScript 是 JavaScript 的超集，这意味着任何 JavaScript 代码也是一个 TypeScript 代码。 这就是为什么 TypeScript 总是编译成 JavaScript 的原因。为了开始在 TypeScript 中进行编程，您必须首先使环境准备就绪。对于 Windows 用户，您只需要安装 Visual Studio，然后下载并安装必要的插件，您将可以开始进行编程。

如果您正在使用任何其他平台，那么您必须依靠名为“npm”的 Node 包。确保您具有文本编辑器和浏览器。一旦安装了用于 TypeScript 的 npm 软件包，您可以继续开始编写您的 TypeScript 程序。