# TypeScript Notebook

Pionpill [1]

本文档为作者学习 TypeScript 的笔记。

2022 年 12 月 13 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

**前言：**

本位介绍 TypeScript 相关语法。TypeScript 是 JavaScript 的超集，是有类型的 JavaScript，他比 JavaScript 更加严格，有更多的 OOP 理念，适用于大型项目，对后端程序员更加友好。

TypeScript 是 JavaScript 的超集。在实际应用中先用 TypeScript 语法编写脚本，再使用 `tsc` 命令将 ts 文件编译成 js 文件，在运行。因此，JavaScript 能用的语法，TypeScript 绝大部分都能用，即使报错不建议用，也可以强制编译。

本文默认读者会 JavaScript，因此相关内容不再说明。主要参考资料为微软官方的 TypeScript HandBook。

环境如下:

- OS: Window11
- TypeScript: 3

2022 年 12 月 13 日

# 目录

# 第一部分

# ECMAScript

# I 基础语法

## 1 基础类型

TypeScript 支持 JavaScript 的几乎所有数据类型，同时还提供了枚举类型。

### 1.1 固有数据类型

#### 布尔值

布尔值仅有 `true/false` 值。

```
let isDone: boolean = false;
```

#### 数字

和 JavaScript 相同。

```
let decLiteral: number = 6;
let hexLiteral: number = 0xf00d;
let binaryLiteral: number = 0b1010;
let octalLiteral: number = 0o744;
```

#### 字符串

和 JavaScript 相同，支持模组字符串。

```
let name: string = "bob";
let you: string = "tom"
let sentence: string = "Hello ${ you }, my name is ${ name }."
```

#### 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。

当指定了`--strictNullChecks`标记(也建议指定),`null`和 `undefined`只能赋值给 `void`和它们各自。

### 数组

TypeScript 定义了两种操作数组元素的方式。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，第二种方式是使用数组泛型，Array< 元素类型 >。

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

### Object

`object`表示非原始类型，是 JavaScript 中最重要的类型，JavaScript 中所有的对象都源自`Object`。

## 1.2 新增数据类型

### 元组

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。

```
let x: [string, number];
x = ['hello', 10];
x = [10, 'hello'];
```

### 枚举

`enum`是对 JavaScript 数据类型的一个补充。默认情况下枚举元素从下标 0 开始，也可以手动指定下标。

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

### Any

`any`表示任意类型，是不确定类型时`Object`的替代方案，因为`any`封装了属于他自己的方法。

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

### Void

void 像 any 的反义词，表示没有任意类型。由于 TypeScript 是强类型语言，所以需要引入 void 作为某些函数的返回值。

```
function warnUser(): void {
    console.log("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;
```

### Never

never 类型表示的是那些用不存在的值的类型。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，当它们被永不为真的类型保护所约束时。

```
// 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) {
    }
}
```

## 1.3 变量声明

TypeScript 是强类型 (静态类型) 语言，它的变量声明机制和 Python 的 type hint 类似，函数声明后面也可以加 : 表示返回值类型:

```
let [变量名] : [类型] = 值;
let num: number = 6;
```

可以在类型中增加 | 表示或:

```
let list: [number | boolean];
```

如果没有显示指出类型，TypeScript 编译器会自动推断。

### 类型断言

类型断言是指程序员知道某个数据类型是什么，因而显示地在数据中写入对应的类型。这类似于 Java 中的泛型。

在 TypeScript 中有两种类型断言语法，一种是泛型式地尖括号，一种是 as 关键字。两种写法等价:

```
let someValue: any = "this is a string";
// 尖括号形式
let strLength: number = (<string>someValue).length;
// as 关键字
let strLength: number = (someValue as string).length;
```

如果在 TypeScript 中使用 JSX，只能使用 as 语法。

# 2 函数

我们知道，JavaScript 对待函数是十分宽裕的，可以有任意个参数，对参数只做匹配不检查。但 TypeScript 不同，TypeScript 要求函数参数必须有类型，且标注返回值类型，即使没有显示指定类型，编译器也会自动推导类型。

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

TypeScript 里的每个函数参数都是必须的。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 中每个函数参数都是可选的，不传的时候就是 `undefined`。在 TypeScript 中可选参数需要使用 `?` 关键字, 同时可选参数必须跟在必选参数后。

```
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
```

当然 TypeScript 也支持默认参数值，用法和 JavaScript 一致。

TypeScript 拾取剩余参数的方式和 JavaScript 一致，只不过只要指定类型为 `string[]`。

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

### this

JavaScript 里，`this` 的值在函数 (非箭头函数) 被调用的时候才会指定，从而确定上下文。在 TypeScript 中，我们可以通过将函数的第一个参数设置为 `this` 用于表示返回的 `this` 指代函数本身。

```
function f(this: void) {
```

```
    // make sure `this` is unusable in this standalone function
}
```

### 函数重载

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。由于 TypeScript 是静态语言，也就失去了动态处理函数参数的能力，但同时，TypeScript 允许函数重载。

# II 面向对象

## 3 接口

### 3.1 对象接口

由于 JavaScript 没有类这个概念，只有对象与原型，因此 TypeScript 的接口也是基于对象。

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

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

类型检查其会查看 `printLabel` 的调用，只有参数是存在一个名为 `label` 的 `string` 对象时，才能通过。

这样写比较混乱，实参类型太复杂了，可以更面向对象一点:

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

可以看出，可传统的编程语言相比，我们并没有继承这个 `LabelledValue` 接口，该接口存在的意义仅仅是做检查，检查是否满足 `interface` 定义的东西，而不是实现接口。

#### 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。可以通过在形参后加 `?` 表示可选:

```
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"});
```

### 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;
}
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!
```

如果像赋值到普通数组，可以这样写:

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

`readonly` 与 `const` 类似。一般 `readonly` 用作对象的属性，`const` 则作为一般变量。

### 额外的属性检查

在 `TypeScript` 中，**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。

```
// 基于前面的 SquareConfig
// 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;
}
```

还有最后一种跳过这些检查的方式，将这个对象赋值给一个另一个变量。因为只有字面量才会做格外的属性检查。

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

最佳的实践是，在传入对象字面量时保证只有接口中规定的属性，否则使用引用传参。

## 3.2 接口拓展

TypeScript 中的接口除了规范对象外，还可以规范函数，类... 毕竟它们本质上都是对象。

### 函数接口

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;
}
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。

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

### 可索引类型

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];
```

TypeScript 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。

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!
```

## 3.3 类接口

### 类接口定义与是实现

TypeScript 的类接口和 Java 类似，仅检查公有部分，需要实现:

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

### 接口继承

接口是可以相互继承的，并且和 Java 一样支持多继承:

```
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;
```

TypeScript 允许接口继承类，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。

# 4 类

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。

## 4.1 类基础

在 TypeScript 里，我们可以使用常用的面向对象模式。基于类的程序设计中一种最基本的模式是允许使用继承来扩展现有的类

```
class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}${distanceInMeters}m.`);
    }
}
class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');
    }
}
const dog = new Dog();
dog.bark();
dog.move(10);
```

在功能上，TypeScript 的类与 Java 类似。

### 成员作用域

是的，TypeScript 支持类成员作用域: `public`, `protected`, `private`。默认为 `public`, 且这三种作用域功能和 Java 中的类似。

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。

```
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; // 错误: Animal 与 Employee 不兼容.
```

### 参数属性

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。

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

仅在构造函数里使用 `readonly 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。

### 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。

```
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，但是这也可能会带来麻烦。

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);
}
```

只带有 get 不带有 set 的存取器自动被推断为 readonly。

## 静态属性

TypeScript 的类支持静态属性，只需要通过类名访问即可:

```
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...');
    }
}
```

## 4.2 类的本质

当一个 TypeScript 类被编译成 JavaScript 后，他会变成什么样子?

```
// TypeScript
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());

// JavaScript
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。

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。