**前端工程引入Typescript规范化编码**

**目录**

[一、 背景 2](#_Toc75511828)

[二、 Webpack解决的问题 2](#_Toc75511829)

[三、 使用webpack模块化打包 2](#_Toc75511830)

[四、 Webpack使用loader实现特殊资源加载 2](#_Toc75511831)

[五、 Webpack使用plugin扩展功能 2](#_Toc75511832)

[六、 Webpack的整体架构 3](#_Toc75511833)

[七、 Webpack的工作流程 3](#_Toc75511834)

[八、 Webpack的核心概念 3](#_Toc75511835)

摘要

关键词

正文

# Typescript是什么

拥有类型系统的JavaScript超集，可以编译成纯JavaScript，(为JavaScript提供静态类型检查)

# Typescript的特点

### 类型检查

TypeScript会在编译代码时，进行严格的静态类型检查。意味着可以在编码阶段发现存在的隐患，而不用把他们带到线上去

### 语言扩展

TypeScript会包括 来自ES6 和未来提案中的特性，比如异步操作和装饰器，也会从其他语言借鉴特性，比如接口和抽象类

### 工具属性

TypeScript可以编译成 JavaScript，在任何浏览器，操作系统上运行。无需任何运行时的额外开销

# 基础类型

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

## 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。

## 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。

## 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

## 枚举

enum类型是对JavaScript标准数据类型的一个补充。

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;

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];

console.log(colorName);  // 显示'Green'因为上面代码里它的值是2

## 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

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：

## 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：

## 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，当它们被永不为真的类型保护所约束时。

## Object

object表示非原始类型，也就是除number，string，boolean，symbol，null或undefined之外的类型。

使用object类型，就可以更好的表示像Object.create这样的API。

# 接口

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。

function printLabel(labelledObj: { label: string }) {

    console.log(labelledObj.label);

  }

  let myObj = { size: 10, label: "Size 10 Object" };

  printLabel(myObj);

//   改写后

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，那么它就是被允许的。

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。

## 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:

## 额外的属性检查

## 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。

## 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。

## 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。

### 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：

## 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。

# Symbols

自ECMAScript 2015起，symbol成为了一种新的原生类型，就像number和string一样。

symbol类型的值是通过Symbol构造函数创建的。

Symbols是不可改变且唯一的。

像字符串一样，symbols也可以被用做对象属性的键。

let sym2 = Symbol("key");

let sym3 = Symbol("key");

sym2 === sym3; // false, symbols是唯一的

let sym = Symbol();

let obj = {

    [sym]: "value"

};

console.log(obj[sym]); // "value"

## **Symbol.hasInstance**

方法，会被instanceof运算符调用。构造器对象用来识别一个对象是否是其实例。

## **Symbol.isConcatSpreadable**

布尔值，表示当在一个对象上调用Array.prototype.concat时，这个对象的数组元素是否可展开。

## **Symbol.iterator**

方法，被for-of语句调用。返回对象的默认迭代器。

## **Symbol.match**

方法，被String.prototype.match调用。正则表达式用来匹配字符串。

## **Symbol.replace**

方法，被String.prototype.replace调用。正则表达式用来替换字符串中匹配的子串。

## **Symbol.search**

方法，被String.prototype.search调用。正则表达式返回被匹配部分在字符串中的索引。

## **Symbol.species**

函数值，为一个构造函数。用来创建派生对象。

## **Symbol.split**

方法，被String.prototype.split调用。正则表达式来用分割字符串。

## **Symbol.toPrimitive**

方法，被ToPrimitive抽象操作调用。把对象转换为相应的原始值。

## **Symbol.toStringTag**

方法，被内置方法Object.prototype.toString调用。返回创建对象时默认的字符串描述。

## **Symbol.unscopables**

对象，它自己拥有的属性会被with作用域排除在外。

总结