TypeScript

> 本讲内容：TypeScript、Node.js、Webpack
>
> 参照2020暑培讲义：JS&TS - EESAST Training 2020以及深入理解 TypeScript 以及官方文档（中文）。
>
> 此处仅作简单介绍，更加深入的学习可以参考上述**后两项**。

JavaScript 是一个不具有强类型的动态语言，这赋予了它极大的灵活性，但也带来了开发和生产上可能存在的问题。TypeScript 是 JavaScript 的超集，使得 JavaScript 中的每一个变量和函数都具有和 C 一样的类型定义。

将“Type”放在名字当中，可见TypeScript中“类型”的重要性。

你可以利用 TypeScript 在编译期进行类型检查，提前发现错误。我们在使用ts的时候，最终还是会将其编译为js代码，但是在编译的时候会进行静态类型检查如果发现有错误，编译的时候就会报错。

可以认为TypeScript就是增加了类型检查的JavaScript。

安装TypeScript

有两种主要的方式来获取TypeScript工具：

- 通过npm（Node.js包管理器）
- 安装Visual Studio的TypeScript插件

Visual Studio 2017和Visual Studio 2015 Update 3默认包含了TypeScript。 如果你的Visual Studio还没有安装TypeScript，你可以下载它。

针对使用npm的用户：

```
npm install -g typescript
```

使用`tsc -v`检查是否安装成功。

输入

```
tsc <待编译文件路径>
```

即可运行TypeScript编译器，生成一个编译后的js文件。

类型注解

TypeScript中的类型注解用于对变量添加约束，可以使用`:`来添加类型注解，直接初始化变量相当于隐式添加类型注解。

编译器会在编译TS时检查变量类型是否符合注解。

```
function greeter(person: string) {
    return "Hello, " + person;
}
let user = [0, 1, 2];

greeter(user);
//greeter.ts(7,26): error TS2345: Argument of type 'number[]' is not
assignable to parameter of type 'string'.

```

- TypeScript中的基础类型

```
//布尔值
let isDone: boolean = false;

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

//字符串
let name: string = "bob";
name = "smith";
let sentence: string = `Hello, my name is ${ name }.`

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

//元组,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
// 声明元组
let x: [string, number];
// 正确初始化
x = ['hello', 10]; // OK
// 错误初始化
x = [10, 'hello']; // Error

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

//Void，表示一个函数没有返回值
function warnUser(): void {
    console.log("This is my warning message");
}

//Null 和 Undefined
let u: undefined = undefined;
let n: null = null;
// 注意：他们是所有类型的子类型
// 这样不会报错
let num: number = undefined;
```

Any类型

Any（任意值）用来表示允许赋值为任意类型，并且可以对其访问任何属性，调用任何方法。它主要用于为那些在编程阶段还不清楚类型的变量指定一个类型。同时，如果你在声明变量时没有指定类型和初值，变量就会被自动识别为Any类，Any会在不太合规的TypeScript开发中大量出现。

指定函数类型

函数本身也是"数"的一种，因此我们同样可以对function进行类型注解。

下面给出指定函数类型的例子：

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

//推断
// 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; };

```

ts还可以设定可选参数，在可选参数后面加`?`即可。

```
let myAdd = function(x: number = 1, y?: number): number { ...};
```

interface

TypeScript的核心原则之一是对值所具有的结构进行类型检查。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。简单的说，在 TypeScript 中，我们使用接口（Interfaces）来定义对象的类型。

可以类比C中的结构体，开发者可自行定义结构体作为新的数据类型。

```
interface Person {
    name: string;
    age: number;
}

let tom: Person = {
    name: 'Tom',
    age: 25
};

let jack: Person = {
```

```
    name: 'Jack'
};
// index.ts(6,5): error TS2322: Type '{ name: string; }' is not assignable
to type 'Person'.
//   Property 'age' is missing in type '{ name: string; }'.
```

当然也可以加入可选属性，上面的错误可以这样解决:

```
interface Person {
    name: string;
    age?: number;
}
let jack: Person = {
    name: 'Jack'
};
```

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覆盖编译器的推断，告诉编译器，“相信我，我知道自己在干什么，请不要再发出错误”。

类型断言的两种方式为:

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

let strLength1: number = (<string>someValue).length;

let strLength2: number = (someValue as string).length;

```

使用例子:

```
const foo = {};
foo.bar = 123; // Error: 'bar' 属性不存在于 ‘{}’
foo.bas = 'hello'; // Error: 'bas' 属性不存在于 '{}'

// 类型断言
interface Foo {
  bar: number;
  bas: string;
}

const foo = {} as Foo;
foo.bar = 123;
foo.bas = 'hello';
```

> 很多时候，类型断言并不是很安全，就像上面的代码，如果你没有按约定添加属性，TypeScript编译器并不会对此发出错误警告。

联合类型

联合类型（Union Types）表示取值可以为多种类型中的一种。联合类型使用 `|` 分隔每个类型。

```ts
let myFavoriteNumber: string | number;
myFavoriteNumber = 'seven';
myFavoriteNumber = 7;
```

当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，我们**只能访问此联合类型的所有类型里共有的属性或方法。**

```ts
function getLength(something: string | number): number {
    return something.length;
}

// index.ts(2,22): error TS2339: Property 'length' does not exist on type 'string | number'.
//   Property 'length' does not exist on type 'number'.
```

此时我们可以用到类型断言。

类型别名

使用type我们可以创建类型别名，常用于联合类型。

```ts
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
```

模块化

后续在讲到*CommonJS*时会再提及此模块化的思想。

全局模块

在默认情况下，当你开始在一个新的 TypeScript 文件中写下代码时，它处于全局命名空间中。如在 foo.ts 里的以下代码。

```ts
const foo = 123;
```

如果你在相同的项目里创建了一个新的文件 `bar.ts`，TypeScript 类型系统将会允许你使用变量 `foo`，就好像它在全局可用一样：

```ts
const bar = foo; // allowed
```

毋庸置疑，使用全局变量空间是危险的，因为它会与文件内的代码命名冲突。我们推荐使用下文中将要提到的文件模块。

文件模块

文件模块也被称为外部模块。如果在你的 TypeScript 文件的根级别位置含有 `import` 或者 `export`，那么它会在这个文件中创建一个本地的作用域。因此，我们需要把上文 `foo.ts` 改成如下方式（注意 `export` 用法）：

```ts
export const foo = 123;
```

在全局命名空间里，我们不再有 `foo`，这可以通过创建一个新文件 `bar.ts` 来证明：

```
const bar = foo; // ERROR: "cannot find name 'foo'"
```

如果你想在 `bar.ts` 里使用来自 `foo.ts` 的内容，你必须显式地导入它，更新后的 `bar.ts` 如下所示。

```
import { foo } from './foo';
const bar = foo; // allow
```

在 `bar.ts` 文件里使用 `import` 时，它不仅允许你使用从其他文件导入的内容，还会将此文件 `bar.ts` 标记为一个模块，文件内定义的声明也不会“污染”全局命名空间。

命名导出

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。

可以在声明的时候直接导出

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

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

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

也可以在声明之后的任意位置导出，并且可以重命名

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

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；并且一个文件只能够有一个模块的 `default` 导出。对于 `default` 模块在导入的时候不必加大括号，而且可以直接重命名。

```
//OneTwoThree.ts
export default "123";

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

导入

在导入的时候，可以直接导入，也可以进行重命名

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

import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
```