

TypeScript简明指北

TypeScript简明指北

TL;DR

起步

安装

初始化

hello world

编译

类型

基础类型

基础声明一览

枚举

函数

接口

泛型

高级类型

Typescript搭配React

安装依赖:

配置babel

配置Webpack

配置package.json script

函数式组件

类组件

入口文件

编译打包

预览

TL;DR

TypeScript 是 JavaScript 的类型的超集，它可以编译成纯 JavaScript。编译出来的 JavaScript 可以运行在任何浏览器上。

优点：

- 代码可读性和可维护性
- 超好的兼容性，从文件类型、变量类型、编译、第三方库等各个方面都兼容现有JS
- TypeScript 是开源的

缺点：

- 需要学习成本，对于没接触过后端语言的前端工程师可能成本更高
- 前期会增加开发成本
- 构建成本
- 现有库的声明文件缺失

起步

安装

```
1 npm i -g typescript
2 # yarn global add typescript
```

初始化

使用 `tsc --init` 可以在当前目录下快速生成 `tsconfig.json` 文件

```
{
  "compilerOptions": {
    "target": "es5",                          /* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2015',
                                                'ES2016', 'ES2017', 'ES2018', 'ES2019' or 'ESNEXT'. */
    "module": "commonjs",                     /* Specify module code generation: 'none', 'commonjs', 'amd', 'system',
                                                'umd', 'es2015', or 'ESNext'. */
    "strict": true,                           /* Enable all strict type-checking options. */
    "esModuleInterop": true                   /* Enables emit interoperability between CommonJS and ES Modules via
                                                creation of namespace objects for all imports. Implies 'allowSyntheticDefaultImports'. */
  }
}
```

hello world

helloWorld.ts

```
1 function helloWorld(name:string){
2   return `${name} say Hello World`
3 }
4
5 const tomName:string = 'tom'
6
7 helloWorld(tomName)
```

编译

使用 `tsc helloWorld.ts` 将ts文件编译为js文件，可以使用glob模式匹配。

编译过程中如果遇到类型错误会提示，但是仍然能够编译成功。

```
1 function helloWorld(name:string){
2   return `${name} say Hello World`
3 }
4
5 const tomName:string = 'tom'
6 const tomAge = 20;
7
8 helloWorld(tomName)
9
10
11
12
13 helloWorld(tomAge)
```

```
const tomAge: 20
```

类型"20"的参数不能赋给类型"string"的参数。 ts(2345)

[速览问题](#) 没有可用的快速修复

```
$ tsc helloWorld.ts
helloWorld.ts:13:12 - error TS2345: Argument of type '20' is not assignable to parameter of type 'string'.

13 helloWorld(tomAge)
    ~~~~~
    not good not good not good

Found 1 error.
```

编译完成生成 helloWorld.js 文件

```
1 function helloworld(name) {
2     return name + " say Hello world";
3 }
4 var tomName = 'tom';
5 var tomAge = 20;
6 helloworld(tomName);
7 helloworld(tomAge);
```

类型

基础类型

Javascript中的基础类型有：

- 值类型（基本类型）
 - 字符串 (String)
 - 数字(Number)
 - 布尔(Boolean)
 - 空 (Null)
 - 未定义 (Undefined)
 - Symbol
- 引用数据类型
 - 对象(Object)
 - 数组(Array)
 - 函数(Function)

Typescript支持Javascript的所有基础类型，并且新增了 **枚举类型** 供我们使用，后文将单独进行介绍。

基础声明一览

```
1  /* 字符串 */
2  // 普通字符串
3  let firstName: string = 'TypeScript';
4  let lastName: string = 'Language';
5  // 模板字符串
6  let fullName: string = `${firstName} ${lastName}`;
7
8  /* 布尔值 */
9  let isGood: boolean = true;
10
11 /* 数字 */
12 let myAge: number = 10;
13
14 /* 数组 */
15 // 使用元素类型+[]的方式定义
16 let list: number[] = [1, 2, 3, 4];
```

```

17 // 使用数组范型的方式定义
18 let list2: Array<string> = ['foo', 'bar'];
19
20 /* 元组Tuple */
21 // "元组类型"是已知元素数量和类型的数组的语义类型，在Typescript中并没有定义这一种类型。
22 let list3: [string, number, boolean] = ['foo', 20, true];
23
24 /* 枚举 */
25 enum color {
26     Red,
27     Green,
28     Blue
29 }
30 let c: color = color.Green;
31
32 /* Any */
33 // Any类型是一种动态类型，即可以为任意类型，适用于在编程阶段无法确定的类型（不要滥用Any类型）。
34 let notSure: any = 4;
35 notSure = 'foo';
36 notSure = false;
37 notSure = [1, 2, 3];
38 // 当不知道数组中元素类型时，可以使用any类型
39 let list4: any[] = [1, true, 'free', {}, undefined, null];
40
41 /* void */
42 // void类型代表没有任何类型，void类型只能赋予undefined和null，一般用于没有返回值的函数的返回值类型
43 function noReturn(): void {
44     console.log('This is a function no return value!');
45 }
46
47 /* Null 和 Undefined */
48 // null 和 undefined 各自对应 null 和 undefined 类型，这两个类型是所有其他类型的子类型。
49 let u: undefined = undefined;
50 let n: null = null;
51
52 /* Never类型 */
53 // never类型表示的是那些永不存在的值的类型
54 function error(message: string): never {
55     throw new Error(message);
56 }
57
58 // 返回never的函数必须存在无法达到的终点
59 function infiniteLoop(): never {
60     while (true) {}
61 }
62
63 // Object类型
64 const obj: object = {};
65 const obj3: object = Object.create({});
66

```

枚举

使用枚举我们可以定义一些带名字的常量。

枚举的特性：

- 支持名称和值的相互映射
- 自增长特性

```
1  enum Direction {
2      Up = 1,
3      Down,
4      Left,
5      Right
6  }
7
8  enum Color {
9      RED,
10     GREEN,
11     BLUE
12 }
13
14 enum DirectionString {
15     Up = 'UP',
16     Down = 'DOWN',
17     Left = 'LEFT',
18     Right = 'RIGHT'
19 }
20
```

将会编译成：

```
1  var Direction;
2  (function(Direction) {
3      Direction[Direction['Up'] = 1] = 'Up';
4      Direction[Direction['Down'] = 2] = 'Down';
5      Direction[Direction['Left'] = 3] = 'Left';
6      Direction[Direction['Right'] = 4] = 'Right';
7  })(Direction || (Direction = {}));
8
9  var Color;
10 (function(Color) {
11     Color[Color['RED'] = 0] = 'RED';
12     Color[Color['GREEN'] = 1] = 'GREEN';
13     Color[Color['BLUE'] = 2] = 'BLUE';
14 })(Color || (Color = {}));
15
16 var DirectionString;
17 (function(DirectionString) {
18     DirectionString['Up'] = 'UP';
19     DirectionString['Down'] = 'DOWN';
20     DirectionString['Left'] = 'LEFT';
21     DirectionString['Right'] = 'RIGHT';
22 })(DirectionString || (DirectionString = {}));
23
```

可以看到，并没有“黑魔法”。

```
1 console.log(Direction['Down']) // 2
2 console.log(Direction[2]) // 'Down'
```

函数

Javascript中的函数有两种类型：命名函数、匿名函数

```
1 // 命名函数
2 function add(x, y) {
3     return x + y;
4 }
5
6 // 匿名函数
7 let myAdd = function(x, y) { return x + y; };
```

使用Typescript分别创建这两种类型的函数

```
1 // 命名函数
2 // 完整的函数声明
3 function add(x:number, y:number):number {
4     return x + y;
5 }
6 // 一般我们可以省略返回值声明，TS会自动推断出返回值类型。
7 function add(x:number, y:number) {
8     return x + y;
9 }
10
11 // 匿名函数
12 // 只指定函数类型
13 let myAdd = function(x: number, y: number): number { return x + y; };
14 // 完整的函数类型
15 let myAdd:(x:number,y:number)=>number = function(x: number, y: number):
    number { return x + y; };
```

接口

接口用于定义结构和类型。

```
1 // 不使用接口
2 const foo(fooObj:{name:'foo',age?:number}){
3     console.log(fooObj.name) // 'foo'
4 }
5 const myObj = {name:'my',age:20}
6 foo(myObj)
7
8 // 使用接口
9 interface Foo{
10     name:string;
11     age?:number;
12 }
13 const foo(fooObj:Foo){
14     console.log(fooObj.name) // 'foo'
15 }
```

一个更复杂的例子

```
1 interface FooObj {
2   name: string;
3   age?: number;
4 }
5
6 interface Foo {
7   (fooObj: FooObj): string;
8   method: (bar: string) => string;
9 }
10
11 const foo = (fooObj: FooObj) => {
12   console.log(fooObj.name); // 'foo'
13 };
14 foo.method = (bar: string) => {
15   return bar;
16 };
17
```

泛型

泛型 用来创建可重用的组件，一个组件可以支持多种类型的数据。

```
1 function foo<T>(arg: T): T {
2   return arg;
3 }
4
5 const stringFoo = foo<string>("foo1")
6 const numberFoo = foo<number>(1)
```

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。相当于对所有类型取合集。

```
1 interface Type1 {
2   name: string;
3   age: number;
4 }
5
6 interface Type2 {
7   name: string;
8   method: (name: string) => number;
9 }
10
11 const type1: Type1 = {
12   name: 'type1',
13   age: 20
14 };
15
16 const type2: Type2 = {
17   method(name) {
```

```

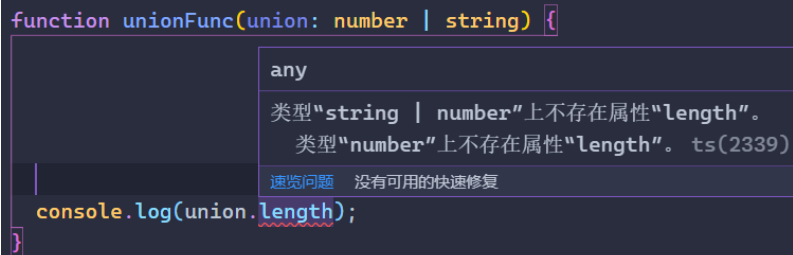
18     return type1.age;
19   }
20 };
21
22 const mixedType: Type1 & Type2 = { ...type1, ...type2 };
23

```

• 联合类型 (Union Types)

联合类型表示一个值可以是几种类型之一。写法是用竖线 (`|`) 分隔每个类型，因此 `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。



```

function unionFunc(union: number | string) {
  console.log(union.length);
}

```

any
类型"string | number"上不存在属性"length"。
类型"number"上不存在属性"length"。 ts(2339)

[速览问题](#) 没有可用的快速修复

上面的代码中参数 `union` 的类型是 `number | string` 的联合类型，因此直接访问 `length` 属性将会报错，因为 `number` 类型上没有 `length` 属性。

Typescript搭配React

Typescript和React可以很好的一起工作，一般我们使用React离不开Webpack。因此在这里介绍下如何构建Typescript+React+Webpack技术栈。

安装依赖：

```

1 npm i react react-dom
2 npm i -D webpack-cli webpack @types/react @types/react-dom

```

在Webpack中解析Typescript需要使用loader，目前主要有以下几种解决方案：

- awesome-typescript-loader (没用过。。。)
- ts-loader (官方)
- babel-loader (babel 7.0版本之后)

很多情况下我们使用了ts-loader之后仍然需要使用babel-loader转换一些ts-loader无法转换的语法特性，因此为什么不直接使用babel-loader来转换ts文件呢。babel从7.0版本后开始支持编译Typescript。在这里主要介绍一下使用babel-loader编译Typescript和React。

- 安装babel-core和babel-loader

```

1 npm i -D @babel/core babel-loader@latest

```

- 安装@babel/preset-env 编译ES最新的语法

```

1 npm i -D @babel/preset-env

```

- 安装@babel/preset-react 用于编译React


```
1 | npm i -D @babel/preset-react
```

- 安装 @babel/preset-typescript 用于编译Typescript

```
1 | npm i -D @babel/preset-typescript
```

- 安装 @babel/plugin-proposal-class-properties 插件支持类的静态属性语法

```
1 | npm i -D @babel/plugin-proposal-class-properties
```

配置babel

在根目录下新建并配置 .babelrc 文件

```
1 | {
2 |   "presets": [
3 |     "@babel/preset-react",
4 |     "@babel/preset-typescript",
5 |     "@babel/preset-env"
6 |   ],
7 |   "plugins": [
8 |     "@babel/proposal-class-properties",
9 |   ]
10 | }
```

配置Webpack

新建并配置 webpack.config.js 文件

```
1 | const path = require('path');
2 | module.exports = {
3 |   mode: 'development',
4 |   entry: path.join(__dirname, 'src', 'index'),
5 |   watch: true,
6 |   output: {
7 |     path: path.join(__dirname, 'build'),
8 |     filename: 'bundle.js',
9 |   },
10 |   module: {
11 |     rules: [
12 |       {
13 |         test: /\.tsx?$/,
14 |         include: [path.resolve(__dirname, 'src')],
15 |         exclude: [path.resolve(__dirname, 'node_modules')],
16 |         loader: 'babel-loader',
17 |       },
18 |     ],
19 |   },
20 | };
```

配置package.json script

```

1  {
2    "scripts": {
3      "start": "webpack --config webpack.config.js"
4    },
5  }

```

函数式组件

ButtonFC.tsx

```

1  import React from 'react';
2  import PropTypes from 'prop-types';
3
4  interface ButtonProps extends React.HTMLAttributes<HTMLButtonElement> {
5    name: string;
6  }
7
8  const Button: React.FC<ButtonProps> = props => {
9    const { children, className, ...otherProps } = props;
10   const buttonClassName = className;
11   return (
12     <button className={buttonClassName} {...otherProps}>
13       {children}
14     </button>
15   );
16 };
17
18 Button.propTypes = {
19   name: PropTypes.string.isRequired
20 };
21
22 export default Button;
23

```

类组件

```

1  import React from 'react';
2  import PropTypes from 'prop-types';
3
4  interface ButtonCountProps extends React.HTMLAttributes<HTMLButtonElement> {
5    name?: string;
6  }
7
8  interface ButtonCountState {
9    count: number;
10 }
11
12 class ButtonCount extends React.Component<ButtonCountProps,
13   ButtonCountState> {
14   static propTypes = {
15     name: PropTypes.string
16   };
17   state = {
18     count: 0
19   };
20 }
21

```

```

18   };
19
20   handleClick = () => {
21     this.setState(state => ({ count: state.count + 1 }));
22   };
23
24   render() {
25     return (
26       <button onClick={this.handleClick}>Click {this.state.count}
times</button>
27     );
28   }
29 }
30
31 export default ButtonCount;
32

```

入口文件

App.ts

```

1  import React from 'react';
2
3  import ButtonFC from './ButtonFC';
4  import ButtonClass from './ButtonClass';
5
6  const App: React.FC = () => (
7    <div>
8      <div>
9        <ButtonFC name="ButtonFC">ButtonFC</ButtonFC>
10      </div>
11      <div>
12        <ButtonClass>ButtonClass</ButtonClass>
13      </div>
14    </div>
15  );
16
17  export default App;
18

```

index.tsx

```

1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import App from './App';
4
5  ReactDOM.render(<App />, document.getElementById('root'));

```

编译打包

```
1 | npm run start
```

预览

ButtonFC

Click 4 times