TypeScript简明指北

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

```
npm i -g typescript
# 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

hello world

helloWorld.ts

```
function helloWorld(name:string){
  return `${name} say Hello World`
}

const tomName:string = 'tom'

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。

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

13 helloWorld(tomAge)
              ~~~~~~

Found 1 error.
```

编译完成生成 helloWorld.js 文件

```
function helloWorld(name) {
    return name + " say Hello World";
}
var tomName = 'tom';
var tomAge = 20;
helloWorld(tomName);
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

```
/* 字符串 */
// 普通字符串
let firstName: string = 'TypeScript';
let lastName: string = 'Language';
// 模板字符串
let fullName: string = `${firstName} ${lastName}`;

/* 布尔值 */
let isGood: boolean = true;

/* 数字 */
let myAge: number = 10;

/* 数组 */
// 使用元素类型+[]的方式定义
let list: number[] = [1, 2, 3, 4];
```

```ts
// 使用数组范型的方式定义
let list2: Array<string> = ['foo', 'bar'];

/* 元组Tuple */
// "元组类型"是已知道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。
let list3: [string, number, boolean] = ['foo', 20, true];

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

/* Any */
// 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）。
let notSure: any = 4;
notSure = 'foo';
notSure = false;
notSure = [1, 2, 3];
// 当不知道数组中元素类型时，可以使用any类型
let list4: any[] = [1, true, 'free', {}, undefined, null];

/* Void */
// 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
function noReturn(): void {
  console.log('This is a function no return value!');
}

/* Null 和 Undefined */
// 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。
let u: undefined = undefined;
let n: null = null;

/* Never类型 */
// never类型表示的是那些永不存在的值的类型
function error(message: string): never {
  throw new Error(message);
}

// 返回never的函数必须存在无法达到的终点
function infiniteLoop(): never {
  while (true) {}
}

// Object类型
const obj: object = {};
const obj3: object = Object.create({});

```

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

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right
}

enum Color {
  RED,
  GREEN,
  BLUE
}

enum DirectionString {
  Up = 'UP',
  Down = 'DOWN',
  Left = 'LEFT',
  Right = 'RIGHT'
}

```

将会编译成：

```
var Direction;
(function(Direction) {
  Direction[(Direction['Up'] = 1)] = 'Up';
  Direction[(Direction['Down'] = 2)] = 'Down';
  Direction[(Direction['Left'] = 3)] = 'Left';
  Direction[(Direction['Right'] = 4)] = 'Right';
})(Direction || (Direction = {}));

var Color;
(function(Color) {
  Color[(Color['RED'] = 0)] = 'RED';
  Color[(Color['GREEN'] = 1)] = 'GREEN';
  Color[(Color['BLUE'] = 2)] = 'BLUE';
})(Color || (Color = {}));

var DirectionString;
(function(DirectionString) {
  DirectionString['Up'] = 'UP';
  DirectionString['Down'] = 'DOWN';
  DirectionString['Left'] = 'LEFT';
  DirectionString['Right'] = 'RIGHT';
})(DirectionString || (DirectionString = {}));

```

可以看到，并没有"黑魔法"。

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

```
// 命名函数
function add(x, y) {
    return x + y;
}

// 匿名函数
let myAdd = function(x, y) { return x + y; };
```

使用Typescript分别创建这两种类型的函数

```
// 命名函数
// 完整的函数声明
function add(x:number, y:number):number {
    return x + y;
}
// 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。
function add(x:number, y:number) {
    return x + y;
}

// 匿名函数
// 只指定函数类型
let myAdd = function(x: number, y: number): number { return x + y; };
// 完整的函数类型
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。

```
// 不使用接口
const foo(fooObj:{name:'foo',age?:number}){
    console.log(fooObj.name) // 'foo'
}
const myObj = {name:'my',age:20}
foo(myObj)

// 使用接口
interface Foo{
    name:string;
    age?:number;
}
const foo(fooObj:Foo){
    console.log(fooObj.name) // 'foo'
}
```

一个更复杂的例子

```
interface FooObj {
  name: string;
  age?: number;
}

interface Foo {
  (fooObj: FooObj): string;
  method: (bar: string) => string;
}

const foo = (fooObj: FooObj) => {
  console.log(fooObj.name); // 'foo'
};
foo.method = (bar: string) => {
  return bar;
};

```

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。

```
function foo<T>(arg: T): T {
    return arg;
}

const stringFoo = foo<string>("foo1")
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。

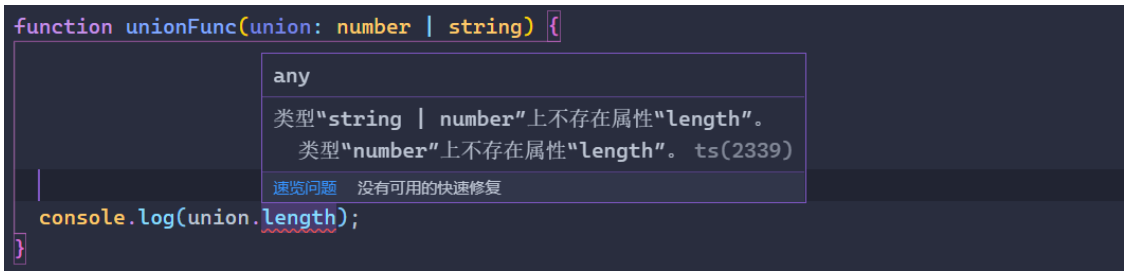
  ```
  interface Type1 {
    name: string;
    age: number;
  }

  interface Type2 {
    name: string;
    method: (name: string) => number;
  }

  const type1: Type1 = {
    name: 'type1',
    age: 20
  };

  const type2: Type2 = {
    method(name) {
  ```

```
    return type1.age;
  }
};

const mixedType: Type1 & Type2 = { ...type1, ...type2 };

```

- **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。



  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：

```
npm i  react react-dom
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

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

- 安装@babel/preset-env 编译ES最新的语法

```
npm i -D @babel/preset-env
```

- 安装@babel/preset-react 用于编译React

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

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

```
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

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

配置babel

在根目录下新建并配置 `.babelrc` 文件

```
{
  "presets": [
    "@babel/preset-react",
    "@babel/preset-typescript",
    "@babel/preset-env"
  ],
  "plugins": [
        "@babel/proposal-class-properties",
    ]
}
```

配置Webpack

新建并配置 `webpack.config.js` 文件

```
const path = require('path');
module.exports = {
  mode: 'development',
  entry: path.join(__dirname, 'src', 'index'),
  watch: true,
  output: {
    path: path.join(__dirname, 'build'),
    filename: 'bundle.js',
  },
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        include: [path.resolve(__dirname, 'src')],
        exclude: [path.resolve(__dirname, 'node_modules')],
        loader: 'babel-loader',
      },
    ],
  },
};
```

配置package.json script

```
{
    "scripts": {
    "start": "webpack --config webpack.config.js"
  },
}
```

函数式组件

ButtonFC.tsx

```tsx
import React from 'react';
import PropTypes from 'prop-types';

interface ButtonProps extends React.HTMLAttributes<HTMLButtonElement> {
  name: string;
}

const Button: React.FC<ButtonProps> = props => {
  const { children, className, ...otherProps } = props;
  const buttonClassName = className;
  return (
    <button className={buttonClassName} {...otherProps}>
      {children}
    </button>
  );
};

Button.propTypes = {
  name: PropTypes.string.isRequired
};

export default Button;

```

类组件

```tsx
import React from 'react';
import PropTypes from 'prop-types';

interface ButtonCountProps extends React.HTMLAttributes<HTMLButtonElement>
{
  name?: string;
}

interface ButtonCountState {
  count: number;
}

class ButtonCount extends React.Component<ButtonCountProps,
ButtonCountState> {
  static propTypes = {
    name: PropTypes.string
  };
  state = {
    count: 0
```

```
  };

  handleClick = () => {
    this.setState(state => ({ count: state.count + 1 }));
  };

  render() {
    return (
      <button onClick={this.handleClick}>Click {this.state.count}
times</button>
    );
  }
}

export default ButtonCount;

```

入口文件

App.ts

```
import React from 'react';

import ButtonFC from './ButtonFC';
import ButtonClass from './ButtonClass';

const App: React.FC = () => (
  <div>
    <div>
      <ButtonFC name="ButtonFC">ButtonFC</ButtonFC>
    </div>
    <div>
      <ButtonClass>ButtonClass</ButtonClass>
    </div>
  </div>
);

export default App;

```

index.tsx

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root'));
```

编译打包

```
npm run start
```

预览

ButtonFC

Click 4 times