
TYPESCRIPT

作者：Alvin

开始时间：2022.4.12

结束时间：2022.4.17

起步

Typescript 相比于 Javascript，对 Javascript 的变量进行指定类型，在 Javascript 中一个变量可以是各种类型，在 Typescript 中可以指定类型。他可以运行在各种 Javascript 支持的平台上，例如node，但不能直接用浏览器引入 ts，需要先进行编译将 ts 转换成 js。

首先安装：`npm i typescript -g` 也可以使用 cnpm

测试安装：cmd 中输入 tsc 没有报错 说明安装成功。

使用 tsc 对 ts 文件进行编译，编译后的 js 可引用。

配置编译选项：

`tsc filename -w` 监视文件，当文件发生变化时，会自动编译成 js，不适合日常使用。

可以使用 `tsc` 这个命令直接编译文件夹中所有的 ts 文件，但在那之前需要先写一个 `tsconfig.json` 文件，tsc 会读取此配置文件，根据配置文件进行编译。

`tsc -w` 监视目录下所有文件，文件中需要有 `tsconfig.json` 文件

`tsconfig.json` :

```
{
  // 指定哪些文件需要被编译
  // ** 表示任意目录
  // * 表示任意文件
  "include": [
    "./src/**/*.ts"
  ],
  // 指定哪些文件不希望被编译
  // ./src/hello/**/*.ts 表示hello下的所有目录所有文件都不监视
  "exclude": [
    "./src/hello/**/*.ts"
  ],
  // 指定要编译的文件名，y列出来
  "files": [],
  // most import 编译器选项
  "compilerOptions": {
    // 需要那个 ES 版本
    // 支持es3\es5\es6\es2016\es2017\es2018\es2019\es2020\esnext
    "target": "ES6",
    // 指定模块化解决方案
    // es6和es2015是一样的 import {fn()} from app.js
    // commonjs → require("app.js")
    "module": "es2015",
    // lib 用来指定项目种使用的库，一般不需要设置
    "lib": [],
    // 编译后的文件放在哪里 一般放在 dist 文件夹种
    "outDir": "./dist",
    // 将所有文件都合并成app.js一个文件中
    // 只有 system 和 commonjs 可以合并
    "outFile": "./dist/app.js",
    // 是否对 js 文件进行编译，默认为false
    "allowJs": false,
    // 是否对 js 进行语法规范，默认是false
    "checkJs": true,
    // 编译后是否移除注释 默认为false
    "removeComments": false,
    // 不生成编译后的文件
    // 有时候不想生成编译文件，只想检查一下语法使用
    "noEmit": false,
    // 当 ts 中有错误时，不会通过编译
    "noEmitOnError": false,
```


```
// 严格检查的总开关
"strict":true,
// 设置编译后的 js 文件使用严格模式
"alwaysStrict":false,
// 不允许出现隐式 any 类型
"noImplicitAny":false,
// 不允许不明确类型的 this
"noImplicitThis":false,
// 严格的检查空值
"strictNullChecks":false

}
}
```

变量

定义变量类型

声明一个变量 a，指定他的类型为 number：



```
let a: number;
// ok
a = 10;
// ok
a = 20;
// err 不能将类型“string”分配给类型“number”
a = "nihao";
```

声明一个变量 b，指定他的类型为 string：



```
let b;  
// ok  
b = 'hello';  
// err 不能将类型“number”分配给类型“string”  
b = 123;
```

这里编译完成之后，js 文件中 let 变成了 var，这是因为 ts 会编译各种版本的 es，这里将 ts 转换成 es3 格式的 js，在编译器设置中可调版本。

注：就算报错，ts 也会将文件编译成 js，但会显示错误信息。

声明完变量直接进行赋值：



```
let c: boolean = false;  
// 也可以  
let c;  
c = false;  
// err 如果是变量定义和赋值同时进行的话，会自动进行类型检测，防止错误  
c = 123
```

函数

在 js 中，函数参数是不用定义类型的，虽然这样很方便，但是也会造成一些安全隐患，例如：



```
function sum(a, b){  
    return a + b;  
}  
// 400  
sum(100, 300)  
// "100300"  
sum(100, "300")
```

在 ts 中可以这样写：

```
function sum(a:number, b:number){
    return a + b;
}
// err 类型“string”的参数不能赋给类型“number”的参数
sum(123, "456")
// err 应有 2 个参数, 但获得 3 个。
sum(123, 345, 456)
```

定义返回值的类型：

```
// 返回值类型为 number
function sum(a:number, b:number): number{
    return a + b;
}
// err 不能将类型“string”分配给类型“number”。
function sum(a:number, b:number): number{
    return a + "b";
}
```

其他数据类型

ts 还有很多新增的数据类型，不管是 js 有的还是没有的。

联合类型（字面量）：

使用字面量进行类型声明：

```
// 这样写 b 这个变量只能为 "male" 或者 "female", 取其他值报错
let b: "male" | "female";

let c: boolean | string
```

any（任意类型）：

将变量设置为 any，此变量将可以是任何类型，和 js 一样，相当于关闭了 ts 的类型检测，将 any 传给一个已知类型的变量，此变量也会百变成 any，且不会有报错，所以尽量别用此类型！！！！

```
let d: any;
// 声明变量如果不指定任何类型，则 ts 会自动判断类型为 any
let d;
d = 10;
d = 'hello';
d = true;
// 这里 s 也会被变成 any，也就是 true，且不会有报错
let s: string;
s = d;
```

unknown（未知类型）：

表示未知类型的值，不同于 any，将 unknown 赋值给已知类型将会报错，unknown 不能直接赋值给其他变量。

```
let e: unknown;
e = 10;
e = 'hello';
e = true;
let s: string;
// err 不能将类型“unknown”分配给类型“string”。
s = e;
```

如果想要赋值给其他变量：



```
let e:unknown;
e = 10;
e = 'hello';
e = true;
let s: string;
if(typeof e === 'string'){
    s = e;
}
```

或者使用类型断言，告诉解析器变量的实际类型：



```
s = e as string;
// 或者
s = <string>e;
```

void (空) :

void 表示空，以函数为例，表示没有返回值的函数。



```
function fn():void{
    // err 不能将类型“number”分配给类型“void”。
    return 123;
}
```

never (永远不会返回结果) :



```
function fn():never{
    // err 不能将类型“number”分配给类型“void”。
    throw new Error('报错啦! ');
}
```

object (对象) :

定义一个 js 对象。

```
let a: object;  
a = {}
```

一般我们用来指定对象中包含哪些属性。

语法: {属性名: 属性值, 属性名: 属性值}

```
let b: {name: string, age: number}  
b = {name: '孙悟空', age: 16}  
// err  
b = {name: '孙悟空'}
```

可以在属性名前边加 ? 表示此属性是可选的。

```
let b: {name: string, age?: number}  
// ok  
b = {name: '孙悟空'}  
// err c中只定义了一个 string 类型的 name, 不允许添加其他变量  
let c: {name: string}  
c = {name: '猪八戒', a: 1, b: 2}
```

在对象中加入 [propName: string]: any 表示可以添加其他任意属性, 任意值。

```
let c: {name: string, [propName: string]: any}  
c = {name: '猪八戒', a: 1, b: 2}
```

也可以使用这种方法设置函数的类型声明,

语法: {形参: 类型, 形参: 类型 ... } ⇒ 返回值



```
let d = (a: number, b: number) => number;
```

array (数组) :

在 js 中数组可以储存任何值。



```
let e: string[];  
e = ['a', 'b', 'c'];  
  
let f: number[];  
// 也可以  
let g: Array<number>
```

tuple (元组) :

元组就是固定长度的数组。



```
let h: [string, string];  
// ok  
h = ['hello', 'nihao'];  
// err 只能有两个元素  
h = ['hello', 'hi', 'nihao']
```

enum (枚举) :

将可能的情况都列出来，如果值确定在几个之间，可以使用。

```
enum Gender{
    Male;
    Female;
}
let i:{name: string, gender: Gender}
i = {
    name: '孙悟空',
    gender: Gender.Male;
}
```

类型的别名：

```
type myType = number OR type myType = 1 | 2 | 3 | 4 | 5
```

WEBPACK+TS

一般我们都使用 `webpack + ts` 来打包 `ts` 文件。

安装 `webpack`

首先在项目目录下初始化 `npm`：`npm init -y`

在目录下会生成一个 `package.json`。

安装 `webpack` 依赖包：`cnpm i -D webpack webpack-cli typescript ts-loader,`

安装一个 `html` 插件，自动的生成 `html` 文件：`cnpm i -D html-webpack-plugin,`

安装插件可以转化各种语法：`cnpm i -D @babel/core @babel/preset-env babel-loader core-js`

清除删除 `dist` 文件夹的文件插件：`cnpm i -D clean-webpack-plugin`

安装webpack 服务器: `cnpm i -D webpack-dev-server`

编写一个 webpack 配置文件, 创建一个 `webpack.config.js` 用来配置 webpack

在 `webpack.config.js`:

```
const path = require("path")
const HTML = require('html-webpack-plugin')
const cleanwebpack = require("clean-webpack-plugin")
// webpack 中所有的配置信息都应该写在 module.exports中
module.exports = {
  // 指定入口文件
  entry: "./src/index.ts"
  // 指定打包文件所在目录
  output: {
    // 指定打包文件的目录
    path: path.resolve(__dirname, 'dist'),
    // 打包文件名
    filename: "bundle.js",
    // 告诉 webpack 别用箭头函数了, 以此来兼容 ie11
    enviroment:{
      arrowFunction: false
    }
  },
  // 指定 webpack 打包时要使用的模块
  module: {
    // 指定要加载的规则
    rules: {
      // test 指定的时规则生效的文件
      test: /\.ts$/,
      // 要使用的 loader
      use:[
        // 配置 babel
        {
          // 指定加载器
          loader: 'babel-loader',
          // 设置 babel
          options: {
            // 设置预定义的环境
            presets: [
              // 指定环境的插件
              '@babel/preset-env',
              // 配置信息
              {
                // 要兼容的浏览器版本
                targets: {
```

```

        "chrome": 88
      },
      // 指定corejs版本
      "corejs": 3,
      // 按照 corejs 的方式 "usage" 表示按需加载
      "useBuiltIns": "usage"
    }
  ]
}
},
, 'ts-loader']
//要排除的文件
exclude: /node-modules/
}
},
// 配置 webpack 插件
plugins: [
  // 传入一个自定义的设置
  new HTML({
    title: '这是一个自定义的title',
    // 设置的网页模板, 会自动根据模板生成index.js
    "template": './index.html'
  }),
  new cleanwebpack(),
],
// 用来设置引用模块
resolve: {
  extensions: ['.ts', '.js']
}
}

```

创建 `tsconfig.json` 来配置 ts 选项。

```

{
  "compilerOptions": {
    "module": "ES2015",
    "target": "ES2015",
    "strict": true
  }
}

```

面向对象

创建类

定义属性：

使用 static 可以定义类属性（静态属性）可以直接用对象.属性调用。

readonly 开头可以将属性定义为只读，不允许修改。

定义方法：

在方法前使用 static 可以直接使用对象调用

```
class Person{

    name:string = '孙悟空',
    // 在属性前使用static关键字可以定义类属性（静态属性）
    age:number = 15,
    sayHello() {
        console.log("Hello 大家好! ")
    }
}

const per = new Person();
console.log(per);
console.log(per.name,per.age)
console.log(Person.age)
per.sayHello();
```

构造函数

当创建对象时将会自动执行构造函数，也就是 `constructor` 函数

```
class Dog{
  name: string,
  age: number,
  // 构造函数
  constructor(name: string, age: number){
    this.name = name;
    this.age = age;
  },
  bark() {
    alert('汪汪汪!!! ')
  }
}

const dog = new Dog('旺财', 4);
console.log(dog);
```

继承

使用 `extends` 关键字来继承。

继承后子类可以拥有父类全部的方法和属性，如果希望在子类中添加一些父类没有的方法，直接在子类中添加即可

```
class Animal {
  name: string,
  age: number,
  constructor(name: string, age: number){
    this.name = name;
    this.age = age;
  },
```

```

    sayHello() {
        console.log("动物在叫! ")
    }
}

class Dog extends Animal {

}

class Cat extends Animal {

}

```

`super()` 可以调用父类, `super` 表示当前类的父类



```

class Animal {
    name: string,
    constructor(name: string){
        this.name = name;
    },
    sayHello() {
        console.log("动物在叫! ")
    }
}

class Dog extends Animal {
    age: number;
    // 如果在子类中写了构造函数, 在子类构造函数中必须对父类进行调用
    constructor(name: string, age: number){
        // 调用父类的构造函数
        super(name);
        this.age = age;
    }
}

const dog = new Dog("旺财", 3)

```

抽象类

在上述例子中，Animal 这个类不可避免的会被创建对象，但有时我们并不希望创建对象，我们更希望这个类只对子类进行继承，这时我们可以使用 抽象类。

抽象类和其他类区别不大，只是不能用来创建对象，是专门用来被继承的类。

在 抽象类 中可以添加抽象方法（注意：抽象方法只能定义在抽象类中），抽象方法使用 `abstract` 开头，没有方法体，子类必须对抽象方法进行重写。

```
abstract class Animal {
  name: string,
  constructor(name: string){
    this.name = name;
  },
  sayHello() {
    console.log("动物在叫! ")
  }
  abstract sayHello(): void;
}

class Dog extends Animal {
  sayHello() {
    console.log("汪汪汪! ")
  }
}

const dog = new Dog("旺财")
```

接口

接口就是用来定义一个类结构，以 `interface` 开头。

接口用来定义一个类中应该包含哪些属性和方法，同时接口也可以当成类型声明去使用。

接口可以重复赋值，最后接口的值 = 之前所有的接口加一起。


```

interface myInterface{
    name: string;
    age: number;
}
interface myInterface{
    gender: string;
}
// 相当于
/*
 * interface myInterface {
 *     name: string;
 *     age: number;
 *     gender: string;
 * }
 *
 */
const obj: myInterface {
    name: '孙悟空',
    age: 20,
    gender: '男'
}

```

接口中的所有属性都不能有实际的值，接口值定义对象的结构，而不考虑实际值，在接口中所有的方法都是抽象方法。

实现类时，可以使类去实现一个接口，实现接口就是使类满足接口的要求。

```

interface myInter {
    name: string;
    sayHello(): void;
}
class MyClass implements myInter{
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    sayHello() {
        console.log('大家好~~~');
    }
}

```

接口本质上就是对类的限制。

属性的封装

在一个类中。不想让外界随意修改对象里的属性，可以使用属性修饰符，类似 Java 的 getter 和 setter：

- `public` 修饰的属性可以在任意位置访问（修改）值。
- `private` 修饰的属性只能在当前类内部进行访问。
- `protected` 只能在此类和其子类中访问。

```
class Person{
  private name: string;
  private age: number;
  constructor(name: string, age: number){
    this.name = name;
    this.age = age;
  }
  getName() {
    return this.name;
  }
  setName(name: string) {
    this.name = name;
  }
  getAge() {
    return this.age;
  }
  setAge(age: number) {
    this.age = age;
  }
}

const per = new Person('孙悟空', 10)
// err
per.name = '猪八戒';
// ok
per.setname('猪八戒')
```

TS 中设置 getter 方法的方式：

直接使用 `get name()`，当要读取 `per.name` 时会自动调用 `name()` 这个方法，并执行其中内容。

当设置 `per.name` 时会自动调用 `set name()` 此方法，进行设置

```
class Person{
  private name: string;
  private age: number;
  constructor(name: string, age: number){
    this.name = name;
    this.age = age;
  }
  get name() {
    return this.name;
  }
  set name(name: string) {
    this.name = name;
  }
  get age() {
    return this.age;
  }
  set age(age: number) {
    this.age = age;
  }
}
const per = new Person('孙悟空', 10)
// '孙悟空'
console.log(per.name)
per.name = '猪八戒'
```

创建类的简单写法：

```
class C{
  // 可以直接将属性写在构造函数中
  constructor(public name: string, public age: number){

  }
}
const c = new C('孙悟空', 11);
```

泛型

在定义函数或者类时，如果遇到类型不明确就可以使用泛型。

```
function fn<k>(a: k): k {  
    return a;  
}
```

可以直接调用具有泛型的函数

```
function fn<k>(a: k): k {  
    return a;  
}  
// 这里 k 会自动被赋值为 number  
// 不指定泛型类型，ts 会自动对类型进行推断  
let result = fn(10);  
// 指定泛型 这里的 k 就是 string  
let result1 = fn<string>('hello')
```

也可以指定多个泛型

```
function fn2<T,K>(a: T, b: K): T {  
    console.log(b);  
    return a;  
}  
fn2<number,string>(123, 'hello')
```

泛型可以继承接口，必须实现是接口的实现类

```
interface Inter{
    length: number;
}
// 这里的这个泛型必须为 Inter 的实现类，也就是必须是 number
function fn3<T extends Inter>(a: T): number {
    return a.length;
}
```

也可以在类中使用泛型

贪吃蛇项目

01. 在文件夹下使用 `npm init` 初始化项目 `npm init`
02. 创建 `src` 文件夹，其中包含 `index.html`(这是一个模板) `index.ts` 两个文件，打包后的文件会在 `dist` 文件夹中
03. 配置在 `package.json` 文件中，使用 `npm build` 打包文件，使用 `npm start` 启动服务器

```
// ... 忽略以上代码
"scripts": {
    "build": "webpack",
    // "start": "webpack serve --open chrome.exe"
    "start": "webpack serve"
}
// ... 忽略以下代码
```

04. 首先创建文件 `tsconfig.json`

```
{
  "compilerOptions": {
    // 指定模块化解决方案
    "module": "ES2015",
    // 需要那个 ES 版本
    "target": "ES2015",
    // 严格检查的总开关
    "strict": true,
    // 如果有报错的话就不要通过编译
    "noEmitOnError": true
  }
}
```

05. 创建 `webpack.config.js`

```
const path = require("path")
const HTML = require('html-webpack-plugin')
const cleanwebpack = require("clean-webpack-plugin")
// webpack 中所有的配置信息都应该写在 module.exports 中
module.exports = {
  // 指定入口文件
  entry: "./src/index.ts",
  // 指定打包文件所在目录
  output: {
    // 指定打包文件的目录
    path: path.resolve(__dirname, 'dist'),
    // 打包文件名
    filename: "bundle.js",
    // 告诉 webpack 别用箭头函数了，以此来兼容 ie11
    environment: {
      arrowFunction: false
    }
  },
  // 指定 webpack 打包时要使用的模块
  module: {
    // 指定要加载的规则
    rules: [{
      // test 指定的规则生效的文件
      test: /\.ts$/,
      // 要使用的 loader
      use: [
```

```

    // 配置 babel
    {
      // 指定加载器
      loader: 'babel-loader',
      // 设置 babel
      options: {
        // 设置预定义的环境
        presets: [
          // 指定环境的插件
          '@babel/preset-env',
          // 配置信息
          {
            // 要兼容的浏览器版本
            targets: {
              "chrome": 88
            },
            // 指定corejs版本
            "corejs": 3,
            // 按照 corejs 的方式 "usage" 表示按需加载
            "useBuiltIns": "usage"
          }
        ]
      },
      //要排除的文件
      exclude: /node-modules/
    }
  ],
  // 配置 webpack 插件
  plugins: [
    // 传入一个自定义的设置
    new HTML({
      title: '这是一个自定义的title',
      // 设置的网页模板, 会自动根据模板生成index.js
      "template": './index.html'
    }),
    new cleanwebpack(),
  ],
  // 用来设置引用模块
  resolve: {
    extensions: ['.ts', '.js']
  }
}

```

06. 安装需要的依赖

- 安装 webpack 依赖包: `cnpm i -D webpack webpack-cli typescript ts-loader,`

- 安装一个 html 插件，自动的生成 html 文件: `cnpm i -D html-webpack-plugin`,
- 安装插件可以转化各种语法: `cnpm i -D @babel/core @babel/preset-env babel-loader core-js`
- 清除删除 dist 文件夹的文件插件: `cnpm i -D clean-webpack-plugin`
- 安装webpack 服务器: `cnpm i -D webpack-dev-server`
- 因为此项目要使用 `less` , 需要下载 `less` 依赖: `npm i -D less less-loader css-loader style-loader`
- 为了使 `css` 文件对大多数浏览器生效, 需要安装 `postcss` : `npm i -D postcss postcss-loader postcss-preset-env`

07. 在 `webpack.config.js` 中修改规则:

```
rules:[
  {},
  // ... 忽略以上代码
  // 设置less文件的处理
  {
    test: /\.less$/,
    // 这里的读取顺序是从下往上读取的, 所以先需要的要写在下边
    use: [
      "style-loader",
      "css-loader",
      // 引入 postcss
      {
        loader: "postcss-loader",
        options: {
          postcssOptions: {
            plugins: [
              [
                "postcss-preset-env",
                {
                  browsers: 'last 2 versions'
                }
              ]
            ]
          }
        }
      },
      "less-loader"
    ]
  }
]
```

剩下就是编写代码了，这里就不演示了...（主要是不会）

至此，typescript 暂时完结，撒花🎉。