

# TypeScript概述

---

## 1.开发环境搭建

### 下载安装node.js

- 命令行/powershell指令
  - 查看node.js版本 `node -v`
  - 全局安装typescript `npm i -g typescript`
  - 查看typescript tsc

### 使用国内镜像

```
npm config set registry https://registry.npmmirror.com
```

### 使用npm全局安装typescript

- 进入命令行
- 输入：

```
npm i -g typescript
```

### 创建一个ts文件

### 使用tsc对ts文件进行编译

- 进入命令行
- 进入ts文件所在目录
- 执行命令：

```
tsc xxx.ts
```

## 2.基本类型

### 类型声明

- 类型声明是TS非常重要的一个特点
- 通过类型声明可以指定TS中变量（参数、形参）的类型
- 指定类型后，当为变量赋值时，TS编译器会自动检查是否符合类型声明，符合则赋值，否则报错
- 简而言之，类型声明给变量设置了类型，使得变量只能存储某种类型的值
- 语法：

```

1 let 变量: 类型;
2
3 let 变量: 类型 = 值;
4
5 function fn(参数1: 类型, 参数2: 类型): 类型 {
6
7 }

```

## 类型推断

- TS拥有自动的类型判断机制
- 当对变量的声明和赋值时同时进行的，TS编译器会自动判断变量的类型
- 所以如果你的变量的声明和赋值时同时进行的，可以省略掉类型声明

## 类型：

类型	例子	描述
number	1, -33, 2.5	任意数字
string	"hello"	任意字符串
boolean	true、false	布尔值 true 或 false
字面量	其本身	限制变量的值就是该字面量的值
any	*	任意类型
unknown	*	类型安全的 any
void	空值 (undefined)	没有值 (或undefined)
never	没有值	不能是任何值
object	{name: '孙悟空' }	任意的 JS 对象
array	[1, 2, 3]	任意的 JS 数组
tuple	[4, 5]	云阿U说，TS新增类型，固定长度数组
enum	enum{A, B}	枚举，TS新增类型

## number

- ```

1 let decimal: number = 6;
2 let hex: number = 0xf00d;
3 let binary number = 1b1010;
4 let octal: number = 0o744;
5 let big: bigint = 100n;

```

## string

```
1 let b: "male" | "female";
2 b = "male";
```

## boolean

- ```
1 let c: boolean | string;
2 c = true;
3 c = 'hello';
```

## any

- ```
1 // any 表示任意类型。一个变量设置了 any 后，相当于对该变量关闭了TS的类型检测
2 // 使用 TS 时，不建议使用 any 类型
3 let d: any; // 显示 any
4 let e; // 隐式 any （声明变量时不指定类型）
```

## unknown

- ```
1 // unknown 表示未知类型的值
2 let f: unknown;
3 f = 15;
4 f = true;
5 f = 'hello';
```

- any类型的变量可以赋值给任意变量；但unknown 类型的变量不能直接赋值给其他变量

- ```
1 let s: string;
2 // d 的类型是any，可赋值给任意变量，不报错
3 s = d;
4 // f 的类型是 unknown，不可以【直接赋值】给任意变量，会报错
5 // s = f; // 报错
```

## 类型检查

- ```
1 // 类型检查后可以赋值
2 if(typeof f == "string") {
3     s = f;
4 }
```

## 类型断言 (Type Assertion)

- 类型断言可以用来手动指定一个值的类型，即允许变量从一种类型更改为另一种类型。

- ```
1 // 类型断言 - 可以用来告诉解析器变量的实际类型
2 s = f as string;
3 // s = <string>f; // 另一种表示方法
4 console.log(s);
```

## void

- void 用来表示空值，以函数为例，就表示没有返回值（或返回undefined）的函数

```

1 function fn2(): void{
2
3 }
4 123

```

## never

- never 表示永远不会返回结果；没有值（比较少用，一般是用来抛出错误）

```

1 function fn3():never {
2     throw new Error('报错了!')
3 }

```

## object

- 用来指定对象中可以包含哪些属性

```

1 // object 表示一个 JS 对象
2 let a: object;
3 a = {};
4 a = function () {
5
6 };
7
8 // {} 用来指定对象中可以包含哪些属性
9 // 语法: {属性名: 属性值, 属性名: 属性值}
10 // 在属性名后加 ? , 表示为可选属性
11 let b: { name: string, age?: number };
12 b = { name: 'coco', age: 18 };
13
14 // 需求: 除 name 外, 对象后可追加多个属性 解决: []
15 // 1. propName 变量名, 可随意 -- [propName: string] 表示任意字符串类型属性名
16 // 2. : any 表示任意类型
17 // 3. [propName: string]: any -- 表示任意类型的属性
18 let c: {name: string, [propName: string]: any};
19 c = {name: 'coco', a: 1, b: 2}

```

## 设置函数结构的类型声明

- ```

1 // 需求: 限制变量的 函数结构: 参数个数, 返回值类型 解决: 类似箭头函数
2 /* 语法:
3     (形参: 类型, 形参: 类型) => 返回值类型 */
4 let d: (a: number, b: number) => number;
5 d = function (n1: number, n2: number) {
6     return n1 + n2;
7 }

```

## array

```

1 // sting[] 表示字符串数组
2 let e: string[];
3 e = ['1', '2', 3]; // 报错: 不能将类型“number”分配给类型“string”
4 // number[] / Array<number> 表示数值数组
5 let f: number[];
6 f = [1, 2, 'hello']; // 报错: 不能将类型“string”分配给类型“number”
7 let g: Array<number>;
8 g = [1, 2, 'hello2']; // 报错: 不能将类型“string”分配给类型“number”

```

## tuple (ts新增类型)

- tuple(元组): 就是固定长度的数组

```

1 /*
2 语法: [类型, 类型, 类型]
3 */
4 let h: [string, number];
5 h = ['1', 2];

```

## enum (ts新增类型)

- 枚举可以把所有可能的值都列举出来

```

1 // 结果是在多个值之间选择时, 适合用枚举 enum
2 // 定义一个枚举类
3 enum Gender {
4     male = 1,
5     female = 0
6 }
7 let i: { name: string, gender: Gender };
8 i = {
9     name: 'coco',
10    gender: Gender.male // 'male'
11 }
12 console.log(i.gender === Gender.male);

```

## 补充 (& 与 类型别名)

```

1 // & 同时满足
2 // 用以连接两个对象, 需同时满足才不报错
3 let j: {name: string} & {age: number};
4 j = {name: 'coco', age: 18};

```

```

1 // 比较麻烦
2 let k: 1 | 2 | 3 | 4 | 5;
3 let m: 1 | 2 | 3 | 4 | 5;
4 // 使用类型别名简化
5 type myType = 1 | 2 | 3 | 4 | 5;
6 let n: myType;
7 n = 6; // 报错: 不能将类型“6”分配给类型“myType”

```

### 3.编译选项

#### 自动编译文件

- 单文件自动编译
  - tsc TS文件 -w
- 多文件自动编译
  - 先在文件夹下新建 tsconfig.json 文件，进行编译配置
  - 然后在命令行执行 tsc -w 可以对**所有ts文件**进行监视，若有修改则会自动重新编译。

**tsconfig.json 是ts编译器的配置文件，ts编译器可以根据它的信息来对代码进行编译**

#### tsconfig.json文件

- ```
1  {
2      /*
3         tsconfig.json 是ts编译器的配置文件，ts编译器可以根据它的信息来对代码进行编译
4      */
5      // 1. "include" 用来指定那些 TS 文件需要被编译
6      // 2. 路径: **表示任意目录，*表示任意文件
7      "include": [
8          "./src/**/*.ts"
9      ],
10
11     // 1. "exclude" 用来表示不需要被编译的文件目录
12     // 2. ["node_modules", "bower_components", "jspm_packages"]
13     "exclude": [
14         "./src/hello/**/*.ts" // 仅 hello 下文件不被编译
15     ],
16
17
18     // "extends" 定义被继承的配置文件
19     "extends": [
20         "./configs/base" // 当前配置文件中会自动包含configs目录下base.json中的所有配置
21     ],
22
23     // 指定被编译文件的列表，只有需要编译的文件少时才会用到
24     "files": [ // 列表中的文件都会被 TS 编译器编译
25         "core.ts",
26         "core2.ts",
27         "core3.ts",
28         "core4.ts",
29         "core5.ts",
30     ],
31
32     // compilerOptions有很多子选项，compilerOptions有很多子选项
33     "compilerOptions": {
34         // target 用来指定 ts 被编译为 ES 的版本
35         // "es3", "es5", "es6", "es2015", "es2016", "es2017", "es2018", "es2019",
36         "es2020", "esnext"
37         "target": "es2016",
38
39         // module 指定要使用的模块化的规范
```

```

39     // "none", "commonjs", "amd", "system", "es6", "es2015", "es2020",
    "esnext"
40     "module": "commonjs",
41
42     // "lib" 用来指定项目使用的库
43     // ES5、ES6、ESNext、DOM、WebWorker.....
44     "lib": [],    // 一般情况下不需要设置（浏览器运行不用管，nodejs运行的再根据需求设置）
45
46     // "outDir" 用来指定编译后文件所在的目录，分离源码和编译后的文件
47     "outDir": "./dist",
48
49     // "outFile" 将代码合并为一个文件
50     // 设置 outFile 后，所有的全局作用域中的代码会合并到同一个文件中
51     "outFile": "./dist/app.js", // 将编译后的文件合并到app.js中
52
53     // 是否对 js 文件进行编译。默认是 false
54     "allowJs": false,
55
56     // 是否检查 js 代码是否符合语法规则，默认是 false
57     "checkJs": false,
58
59     // 是否移除注释
60     "removeComments": true,
61
62     // 不生成编译后的文件
63     "noEmit": false,
64
65     // 报错时不生成编译文件
66     "noEmitOnError": true,
67
68     // 所有严格检查的总开关，包括下面 4 个（如果相同可直接用这个，下面 4 个省略）
69     "strict": true,
70
71     // 用来设置编译后的文件是否使用严格模式。默认时false
72     "alwaysStrict": true,
73
74     // 不允许隐式的any类型
75     "noImplicitAny": true,
76
77     // 不允许不明确类型的this
78     "noImplicitThis": true,
79
80     // 严格检查空值
81     "strictNullChecks": true
82 }
83 }

```

## 4.webpack

一般项目中我们不会直接编译ts代码，而是使用打包工具来进行

**创建项目文件夹，生成package.json，作用：管理项目**

- 初始化项目：

```
npm init -y
```

### 安装使用webpack所用的依赖：（4个包）

- npm
- cnpm（国内镜像，速度较快）

```
cnpm i -D webpack webpack-cli typescript ts-loader
```

- -D：开发依赖，saveDev的简写
- webpack：打包工具的核心代码
- webpack-cli：命令行工具，可使用命令行使用webpack
- typescript：ts核心包
- ts-loader：webpack的加载器，将typescript和webpack整合到一起

### 问题 1

### webpack目录下新建webpack.config.js文件

- ```
1 // 引入一个包(nodejs中一个模块，主要作用时拼接路径)
2 const path = require('path');
3
4 // webpack 中所有的配置信息都应该写在 module.exports 中
5 module.exports = {
6   // 指定入口文件
7   entry: './src/index.ts',
8
9   // 指定打包文件所在目录
10  output: {
11    // 指定打包文件目录
12    path: path.resolve(__dirname, 'dist'),
13    // 打包后文件的名字
14    filename: 'bundle.js'
15  },
16
17  // 指定webpack打包时要使用的模块
18  module: {
19    // 指定要加载的规则
20    rules: [
21      {
22        // test指定的时规则生效的文件
23        test: /\.ts$/,
24        // 要使用的loader
25        use: 'ts-loader',
26        // 指定要排除的文件
27        exclude: /node-modules/
28      }
29    ]
30  },
31  mode: "development"
32 }
33
```



## webpack目录下新建 tsconfig.json 文件

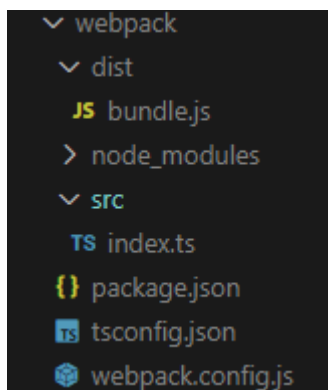
- ```
1 {
2   "compilerOptions": {
3     "module": "ES2015",
4     "target": "ES2015",
5     "strict": true
6   }
7 }
```

## 修改 package.json 文件

- ```
1 {
2   "name": "webpack",
3   "version": "1.0.0",
4   "main": "index.js",
5   "scripts": {
6     "test": "echo \"Error: no test specified\" && exit 1",
7     "build": "webpack"
8   },
9   "keywords": [],
10  "author": "",
11  "license": "ISC",
12  "description": "",
13  "devDependencies": {
14    "ts-loader": "^9.3.0",
15    "typescript": "^4.6.4",
16    "webpack": "^5.72.0",
17    "webpack-cli": "^4.9.2"
18  }
19 }
20
```

- 在package.json中加上build命令 "build": "webpack", 执行 `npm run build` 即可进行编译打包

 `npm run build`



## 常用插件

### 1.html-webpack-plugin

- html插件能帮助我们在打包时自动地生成html文件
  - `cnpm i -D html-webpack-plugin`
  - 下载完成后, package.json 会更新

```
1  "devDependencies": {
2    "html-webpack-plugin": "^5.5.0", // 更新
3    "ts-loader": "^9.3.0",
4    "typescript": "^4.6.4",
5    "webpack": "^5.72.0",
6    "webpack-cli": "^4.9.2"
```

- 在 webpack.config.js 文件中引入 html 插件

```
1  // 引入一个包(nodejs中一个模块, 主要作用时拼接路径)
2  const path = require('path');
3  // 引入 html 插件
4  const HTMLWebpackPlugin = require('html-webpack-plugin');
5
6  *****
7
8    mode: "development",
9
10   // 配置 webpack 插件
11   plugins: [
12     new HTMLWebpackPlugin({
13       // title: "出岫构建",
14       // 设置 html 模版 (在src文件夹下新建 index.html文件, 并设置模版样式)
15       template: "./src/index.html"
16     }),
17   ]
```

在src目录下新建 index.html 模版文件

`npm run build`

## 2.webpack-dev-server

- 该插件能自动响应浏览器更新
- 安装:

`cnpm i -D webpack-dev-server`

- 在package.json中加上start命令:

`"start": "webpack serve --open chrome.exe"`

```
1  "scripts": {
2    "test": "echo \"Error: no test specified\" && exit 1",
3    "build": "webpack",
4    "start": "webpack serve --open --mode production" // 更新
5  }
```

 npm start

### 3.clean-webpack-plugin

- 能在build前清空dist目录所有文件，避免旧文件的遗留
- 目前：在 output 写入 clean: true

```
1 output: {  
2     // 指定打包文件目录  
3     path: path.resolve(__dirname, 'dist'),  
4     // 打包后文件的名称  
5     filename: "bundle.js",  
6     // 在build前清空dist目录所有文件，避免旧文件的遗留  
7     clean: true  
8 },
```

- 安装:

 cnpm i -D clean-webpack-plugin

- 使用:

```
1 // 引入一个包(nodejs中一个模块，主要作用时拼接路径)  
2 const path = require('path');  
3 // 引入 html 插件  
4 const HTMLWebpackPlugin = require('html-webpack-plugin');  
5 // 引入 clean 插件  
6 const { cleanWebpackPlugin } = require('clean-webpack-plugin');  
7  
8 *****  
9  
10 // 配置 webpack 插件  
11 plugins: [  
12     new cleanWebpackPlugin(),  
13     new HTMLWebpackPlugin({  
14         // title: "出岫构建",  
15         // 设置 html 模版 (在src文件夹下新建 index.html文件，并设置模版样式)  
16         template: "./src/index.html"  
17     }),  
18 ]
```

### 4.babel

为了使得代码能兼容不同浏览器，我们需要使用babel工具（与webpack结合一起使用）。

- 安装依赖：（四个）

 cnpm i -D @babel/core @babel/preset-env babel-loader core-js

```
1 | - @babel/core—babel核心的工具
```

- @babel/preset-env—babel的预设环境
- babel-loader—babel与webpack结合的工具

- core-js—模拟js运行环境（使用时可以按需引入）
- 修改webpack配置文件

在loader加载器中加入babel（loader中的执行顺序是从下往上，所以需要将' ts-loader' 放在最后）

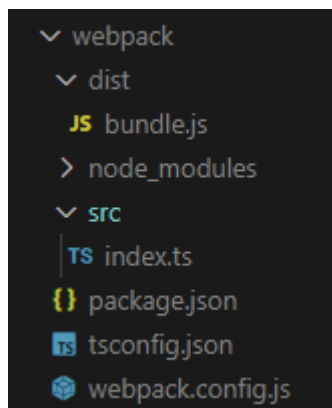
```
1 // 引入一个包(nodejs中一个模块，主要作用时拼接路径)
2 const path = require('path');
3 // 引入 html 插件
4 const HTMLWebpackPlugin = require('html-webpack-plugin');
5 // 引入 clean 插件
6 const {
7   cleanWebpackPlugin
8 } = require('clean-webpack-plugin');
9
10 // webpack 中所有的配置信息都应该写在 module.exports 中
11 module.exports = {
12   // 指定入口文件
13   entry: "./src/index.ts",
14
15   // 指定打包文件所在目录
16   output: {
17     // 指定打包文件目录
18     path: path.resolve(__dirname, 'dist'),
19     // 打包后文件的名字
20     filename: "bundle.js",
21     // 在build前清空dist目录所有文件，避免旧文件的遗留
22     clean: true
23   },
24
25   // 指定webpack打包时要使用的模块
26   module: {
27     // 指定要loader加载的规则
28     rules: [{
29       // test指定的是规则生效的文件
30       test: /\.ts$/, //以ts结尾的文件
31       // 要使用的loader
32       use: [
33         // 配置babel
34         {
35           //指定加载器
36           loader: "babel-loader",
37           // 设置babel
38           options: {
39             //设置预定义的环境
40             presets: [
41               [
42                 //指定环境的插件
43                 "@babel/preset-env",
44                 // 配置信息
45                 {
46                   // 要兼容的目标浏览器及版本
47                   targets: {
48                     "chrome": "58",
```

```

49         "ie": "11"
50     },
51     //指定corejs的版本 (根据package.json中的版本, 只写整数)
52     "corejs": "3",
53     //使用corejs的方式 "usage" 表示按需加载
54     "useBuiltIns": "usage"
55 }
56
57     ]
58 }
59
60 },
61 // 'babel-loader',
62 'ts-loader'
63 ],
64 // 要排除的文件
65 exclude: /node-modules/
66 }
67 },
68 mode: "development",
69 // 配置 webpack 插件
70 plugins: [
71     new HTMLWebpackPlugin({
72         // title: "出岫构建",
73         // 设置 html 模版 (在src文件夹下新建 index.html文件, 并设置模版样式)
74         template: "./src/index.html"
75     }),
76 ]
77 }

```

## 最基本的webpack 配置



## tsconfig.json 配置

```

1  {
2      // "include": [
3      //     "./src/**/*"
4      // ],
5      /* "exclude": [
6
7      ], */

```

```

8     "compilerOptions": {
9         "target": "es6",
10        "module": "es6",
11        "strict": true,
12        // "outDir": "./dist"
13    }
14 }

```

## package.json 配置

```

1  {
2    "name": "webpack",
3    "version": "1.0.0",
4    "main": "index.js",
5    "scripts": {
6      "test": "echo \"Error: no test specified\" && exit 1",
7      "build": "webpack"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC",
12   "description": "",
13   "devDependencies": {
14     "ts-loader": "^9.3.0",
15     "typescript": "^4.6.4",
16     "webpack": "^5.72.0",
17     "webpack-cli": "^4.9.2"
18   }
19 }

```

## webpack.config.js 配置

```

1  // 引入一个包(nodejs中一个模块, 主要作用时拼接路径)
2  const path = require('path');
3
4  // webpack 中所有的配置信息都应该写在 module.exports 中
5  module.exports = {
6    // 指定入口文件
7    entry: "./src/index.ts",
8
9    // 指定打包文件所在目录
10   output: {
11     // 指定打包文件目录
12     path: path.resolve(__dirname, 'dist'),
13     // 打包后文件的名字
14     filename: "bundle.js"
15   },
16
17   // 指定webpack打包时要使用的模块
18   module: {
19     // 指定要加载的规则
20     rules: [

```

```

21     {
22         // test指定的时规则生效的文件
23         test: /\.ts$/,
24         // 要使用的loader
25         use: 'ts-loader',
26         // 指定要排除的文件
27         exclude: /node-modules/
28     }
29 ]
30 },
31 mode: "development"
32 }

```

## 安装插件后的配置

### package.json

```

1  {
2    "name": "webpack",
3    "version": "1.0.0",
4    "main": "index.js",
5    "scripts": {
6      "test": "echo \"Error: no test specified\" && exit 1",
7      "build": "webpack",
8      "start": "webpack serve --open --mode production"
9    },
10   "keywords": [],
11   "author": "",
12   "license": "ISC",
13   "description": "",
14   "devDependencies": {
15     "@babel/core": "^7.17.10",
16     "@babel/preset-env": "^7.17.10",
17     "babel-loader": "^8.2.5",
18     "clean-webpack-plugin": "^4.0.0",
19     "core-js": "^3.22.4",
20     "html-webpack-plugin": "^5.5.0",
21     "ts-loader": "^9.3.0",
22     "typescript": "^4.6.4",
23     "webpack": "^5.72.0",
24     "webpack-cli": "^4.9.2",
25     "webpack-dev-server": "^4.9.0"
26   }
27 }

```

### webpack.config.js

```

1  // 引入一个包(nodejs中一个模块, 主要作用时拼接路径)
2  const path = require('path');
3  // 引入 html 插件
4  const HTMLWebpackPlugin = require('html-webpack-plugin');
5  // 引入 clean 插件

```

```

6  const {
7    cleanwebpackPlugin
8  } = require('clean-webpack-plugin');
9
10 // webpack 中所有的配置信息都应该写在 module.exports 中
11 module.exports = {
12   // 指定入口文件
13   entry: "./src/index.ts",
14
15   // 指定打包文件所在目录
16   output: {
17     // 指定打包文件目录
18     path: path.resolve(__dirname, 'dist'),
19     // 打包后文件的名字
20     filename: "bundle.js",
21     // 在build前清空dist目录所有文件，避免旧文件的遗留
22     clean: true
23   },
24
25   // 指定webpack打包时要使用的模块
26   module: {
27     // 指定要loader加载的规则
28     rules: [{
29       // test指定的是规则生效的文件
30       test: /\.ts$/, //以ts结尾的文件
31       // 要使用的loader
32       use: [
33         // 配置babel
34         {
35           //指定加载器
36           loader: "babel-loader",
37           // 设置babel
38           options: {
39             //设置预定义的环境
40             presets: [
41               [
42                 //指定环境的插件
43                 "@babel/preset-env",
44                 // 配置信息
45                 {
46                   // 要兼容的目标浏览器及版本
47                   targets: {
48                     "chrome": "58",
49                     "ie": "11"
50                   },
51                   //指定corejs的版本（根据package.json中的版本，只写整数）
52                   "corejs": "3",
53                   //使用corejs的方式 "usage" 表示按需加载
54                   "useBuiltIns": "usage"
55                 }
41             ]
56           ]
57         }
58       ]

```



```

59         }
60     },
61     // 'babel-loader',
62     'ts-loader'
63 ],
64 // 要排除的文件
65 exclude: /node-modules/
66 }]
67 },
68 mode: "development",
69 // 配置 webpack 插件
70 plugins: [
71     new HTMLWebpackPlugin({
72         // title: "出岫构建",
73         // 设置 html 模版 (在src文件夹下新建 index.html文件, 并设置模版样式)
74         template: "./src/index.html"
75     }),
76 ]
77 }

```

## tsconfig.json

```

1  {
2      "include": [
3          "./src/**/*"
4      ],
5      "exclude": [
6
7      ],
8      "compilerOptions": {
9          "target": "es6",
10         "module": "es6",
11         "strict": true,
12         "outDir": "./dist",
13         "noEmitOnError": true
14     }
15 }

```

## 5.面向对象

### 1.面向对象

面向对象是程序中一个非常重要的思想，简而言之就是程序之中所有的操作都需要通过对象来完成。对象中有属性和方法。

例子：

```

1  操作浏览器要使用window对象
2  操作网页要使用document对象
3  操作控制台要使用console对象

```

### 2.类

- 类描述了所创建的对象共同的属性和方法。
- TypeScript 支持面向对象的所有特性，比如 类、接口等。

## 1.创建类

```
1 // 使用 class 关键字 定义一个类
2 /* 对象中主要包含两个部分：
3     属性 + 方法 */
4 class Person {
5     // 定义实例属性
6     name: string = "coco",
7     readonly age: number = 18,
8     // 在属性前使用 static 关键字可以定义 类属性（静态属性）
9     // 类属性（静态属性）：不需要创建对象就可使用
10    static eyeColor: string = "blue"
11
12    // 定义实例方法
13    sing() {
14        console.log('hello~~');
15    }
16    // 定义类方法
17    static singlala() {
18        console.log('la~la~la~');
19    }
20 }
21
22 const per1 = new Person();
23 console.log(per1);
24 console.log(per1.age);
25 console.log(Person.age); // 报错。无法访问类的实例属性
26 console.log(Person.eyeColor);
27
28
29 // 实例属性可读写，readonly 开头的属性表示为只读属性，无法修改
30 per1.name = "elice";
31 console.log(per1.name);
32 per1.age = "20"; // 报错： 无法分配到 "age" ， 因为它是只读属性
33 // 静态属性也可加 readonly 变为只读静态属性
34
35 // 调用类的实例方法
36 per1.sing();
37 // 调用类方法
38 Person.singlala();
```

## 2.构造函数

```
1 class Dog {
2     name: string;
3     age: number;
4     // constructor: 构造函数(), 会在对象创建时调用
5     constructor(name: string, age: number) {
6         // 在构造函数中, this 表示当前的实例对象
7         // 可通过 this 向新建对象中添加属性
```

```

8
9      // 向构造函数写入参数可创建不同的对象
10     this.name = name;
11     this.age = age;
12 }
13
14 bark() {
15     console.log(this);
16 }
17 }
18 }
19
20 const dog = new Dog("coco", 2);
21 const dog2 = new Dog("divde", 3);
22
23 console.log(dog);
24 console.log(dog2);
25
26 dog.bark();
27 dog2.bark();

```

### 3.继承

- 不适用类继承所产生问题

```

1  // 避免不同文件变量名冲突，代码写在立即执行函数中，使其作用域不同
2  (function () {
3      // 定义一个 Dog 类
4      class Dog {
5          name: string;
6          age: number;
7
8          constructor(name: string, age: number) {
9              this.name = name;
10             this.age = age;
11         }
12
13         sayHello() {
14             console.log('汪汪汪');
15         }
16     }
17 }
18 const dog = new Dog("旺财", 5);
19 console.log(dog);
20 dog.sayHello();
21
22 // 定义一个 Cat 类
23 class Cat {
24     name: string;
25     age: number;
26
27     constructor(name: string, age: number) {
28         this.name = name;

```

```

29         this.age = age;
30     }
31
32     sayHello() {
33         console.log('喵喵喵');
34     }
35 }
36
37 const cat = new Cat("kitty", 3);
38 console.log(cat);
39 cat.sayHello();
40
41 // 问题：两个不同类有代码重复的部分
42 // 解决：将相同的代码提取出来定义一个大的 Animal 类
43 class Animal {
44     name: string;
45     age: number;
46
47     constructor(name: string, age: number) {
48         this.name = name;
49         this.age = age;
50     }
51
52     sayHello() {
53         console.log('动物在叫');
54     }
55 }
56
57 })();

```

- 子类继承父类的属性和方法
- 子类添加属性+方法
- 子类覆盖掉父类的属性和方法 —— 方法重写

```

1  // 避免不同文件变量名冲突，代码写在立即执行函数中，使其作用域不同
2  (function () {
3      /*
4       *   Dog extends Animal
5       *   - 此时，Animal 被称为 父类， Dog 被称为 子类
6       *   - 使用继承后，子类将继承父类的所有属性和方法
7       *   - 通过继承可以将多个类中共同代码写在一个父类中
8       *   - 这样只需写一次代码可让所有子类同属拥有父类中的属性和方法
9       *
10      *   - 想添加父类没有的属性和方法，可在子类中直接写入属性和方法
11      *   - 子类中添加于父类相同的方法，子类方法会覆盖父类方法
12      */
13
14      // 问题：两个不同类有代码重复的部分
15      // 解决：将相同的代码提取出来定义一个大的 Animal 类
16      class Animal {
17          name: string;
18          age: number;

```

```

19     static gender: number;
20
21     constructor(name: string, age: number, gender: number) {
22         this.name = name;
23         this.age = age;
24         Animal.gender = gender;
25     }
26
27     sayHello() {
28         console.log('动物在叫');
29     }
30 }
31
32 // 定义一个 Dog 类
33 // 继承 Animal 类
34 class Dog extends Animal {
35     // 添加父类中没有的 run() 方法
36     run() {
37         console.log(`${this.name}在跑~`);
38     }
39
40     //
41     sayHello() {
42         console.log(`${this.name}在汪~汪~汪~!`);
43     }
44 }
45 const dog = new Dog("旺财", 5, 1);
46 console.log(dog);
47 dog.sayHello();
48 dog.run();
49
50 // 定义一个 Cat 类
51 // 继承 Animal 类
52 class Cat extends Animal {
53
54 }
55 const cat = new Cat("kitty", 3, 0);
56 console.log(cat);
57 cat.sayHello();
58 })()

```

#### 4.super

- 类方法中，super 表示当前类的父类
- 子类中，可通过super引用父类

```

1 (function () {
2     class Animal {
3         name: string;
4         constructor(name: string) {
5             this.name = name;
6         }
7     }

```

```

8       sayHello() {
9           console.log('动物在说 hello! ~');
10      }
11  }
12
13  class Dog extends Animal {
14      sayHello() {
15          // 在类的方法中, super 就表示当前类的父类
16          super.sayHello();
17      }
18  }
19
20  const dog = new Dog("旺财");
21  dog.sayHello(); // 输出: 动物在说 hello! ~
22
23  })();

```

- 如果在子类中写了构造函数，则在子类构造函数中必须对父类的构造函数进行调用

```

1  (function () {
2      class Animal {
3          name: string;
4          constructor(name: string) {
5              this.name = name;
6          }
7
8          sayHello() {
9              console.log('动物在说 hello! ~');
10         }
11     }
12
13     class Dog extends Animal {
14         /* sayHello() {
15             // 在类的方法中, super 就表示当前类的父类
16             super.sayHello();
17         } */
18         age: number;
19
20         constructor(name: string, age: number) {
21             // 子类中写了构造函数, 必须对父类的构造函数进行调用
22             // 否则会出错
23             // 使用 super(...父类参数名) 调用父类构造函数
24             super(name);
25             this.age = age;
26
27         }
28     }
29
30     const dog = new Dog("旺财", 5);
31     dog.sayHello(); // 输出: 动物在说 hello! ~
32
33 })();

```

## 5.抽象类

- 以abstract 开头的类是抽象类
- 抽象类和其他类区别不大，只是不能用来创建对象
- 抽象类就是专门用来被继承的类
- 抽象类中可以添加抽象方法

```
1 (function () {
2     /*
3     以 abstract 开头的是抽象类
4     抽象类和其他类区别不大，只是不能用来创建对象
5     抽象类是用来创建对象的类
6
7     抽象类中可以添加抽象方法
8     期望： 在父类中自定义方法的解构，具体需求实现由子类完成
9     定义抽象方法：在 方法 前添加 abstract, 没有方法体
10    抽象方法只能定义在抽象类中，子类必须对抽象方法进行重写
11    */
12    abstract class Animal {
13        name: string;
14        constructor(name: string) {
15            this.name = name;
16        }
17
18        abstract sayHello():void;
19    }
20
21    class Dog extends Animal {
22        sayHello(): void {
23            console.log("汪汪汪汪");
24        }
25    }
26
27    class Cat extends Animal { // 报错，子类没重写父类的抽象方法
28    }
29
30
31
32    const dog = new Dog("旺财");
33    dog.sayHello();
34
35    const an = new Animal(); // 报错：无法创建抽象类的实例。
36
37 }());
```

## 6.接口 (ts 新增)

接口用来定义一个类结构, 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。

- 接口可以在定义类的时候去限制类的结构
- 接口中所有的属性都不能有实际的值
- 接口只定义对象的结构，而不考虑实际值

- 在接口中所有的方法都是抽象类

```
1 (function() {
2
3     // 描述一个对象的类型
4     type myType = {
5         name: string;
6         age: number;
7         [porpName: string]: any
8     }
9
10
11     const obj: myType = {
12         name: 'Alice',
13         age: 18
14     }
15
16     // 也可用接口的形式定义一个类结构,用来定义一个类中应包含哪些属性和方法
17     //
18     interface myInterface {
19         name: string;
20         age: number
21     }
22
23     interface myInterface {
24         gender: number
25     }
26
27     const obj2: myInterface = {
28         name: "bobo",
29         age: 20,
30         gender: 0
31     }
32
33     // 类型声明只能存在一个, 接口可以多次声明, 最终效果为合集
34
35     /*
36     接口可以在定义类的时候去限制类的结构,
37     接口的所有属性都不能有实际值
38     接口只定义对象的结构, 而不考虑实际值
39     接口中所有的方法都是抽象方法
40     */
41     interface myInter {
42         name: string;
43         sayHello():void;
44     }
45
46     // 定义类时, 可使用类去实现一个接口
47     // 实现接口就是使类满足接口的要求
48     class myClass implements myInter {
49         name: string;
50
51         constructor(name: string) {
```



```

52         this.name = name
53     }
54
55     sayHello(): void {
56         throw new Error("Method not implemented.");
57     }
58
59 }
60
61 })()

```

接口与抽象类类似，区别在于

- 抽象类可以有抽象方法也可以有普通方法；但接口只能有抽象方法
- 抽象类使用extends继承；接口使用implements实现

## 7.属性的封装

1.TS可以在类中的属性前添加属性的修饰符

- public 修饰的属性可以在任意位置访问（修改） **默认值**
- private 私有属性, 私有属性只能在类内部进行访问（修改）；但可以通过在类中添加方法使得私有属性可以被外部访问
- protected 受保护的属性,只能在当前类和当前类的子类中访问(修改)

2.js和ts封装属性的区别

由于属性是在对象中设置的，属性可以任意的被修改，将会导致对象中的数据变得非常不安全。因此需要对属性进行封装。

- **js封装的属性存取器使用时需要调用相应的getter和setter方法；**
- **而ts封装的属性存取器使用时可直接当作变量来用就行。**

*加getter和setter方法只是为了对属性的值做判断，如果不需做判断则没必要使用。*

问题：某些情况下，出于数据安全考虑，需灵活控制对象访问权

```

1  (function () {
2      // 定义一个表示人的类
3      class Person {
4          // TS可以在属性前添加属性的修饰符
5          /*
6           *   public(默认)   公共属性：修饰的属性可以在任意位置读写
7           *   private       私有属性： 只能在类内部进行修改
8           *       - 通过在类中添加方法使得私有属性可以在外部访问
9           *       - 虽然很麻烦，但属性访问权可以控制，删除方法便不可访问
10          */
11         */
12         private _name: string;
13         private _age: number;
14
15         constructor(name: string, age: number) {
16             this._name = name;

```

```

17         this._age = age;
18     }
19     // 定义方法, 用来获取 name 属性
20     getName() {
21         return this._name;
22     }
23     // 定义方法, 用来设置 name 属性
24     setName(value: string) {
25         this._name = value;
26     }
27     // 定义方法, 用来获取 age 属性
28     getAge() {
29         return this._age;
30     }
31     // 定义方法, 用来修改 age 属性
32     // 出于 数据安全 考虑, 可通过判断后再修改
33     setAge(value: number) {
34         if (value > 0) {
35             this._age = value
36         }
37     }
38
39 }
40
41 const per = new Person("coco", 18);
42 // 现在属性是在对象中设置的, 属性可以任意修改
43 // - 属性可以被任意修改将会导致对象中的数据变得非常不安全
44 console.log(per);
45 per.setName('elice');
46 console.log(per.getName()); // 输出: elice
47 per.setAge(-33);
48 console.log(per.getAge()); // 输出: 18
49 })();

```

改进: 在TS中, 提供一种更加灵活的方式

应用场景: 属性容易被修改错

```

1  (function () {
2      // 定义一个表示人的类
3      class Person {
4          // TS可以在属性前添加属性的修饰符
5          /*
6              *   public(默认)   公共属性: 修饰的属性可以在任意位置读写
7              *   private      私有属性: 只能在类内部进行修改
8              *               - 通过在类中添加方法使得私有属性可以在外部访问
9              *               - 虽然很麻烦, 但属性访问权可以控制, 删除方法便不可访问
10             *   protected   受保护的属性: 只能在当前类和其子类中访问
11             */
12         private _name: string;
13         private _age: number;
14
15         constructor(name: string, age: number) {

```

```

16         this._name = name;
17         this._age = age;
18     }
19     // 可直接通过 per.name 获取 name 原理是调用此方法
20     get name() {
21         console.log('get 被调用了');
22         return this._name
23     }
24
25     set name(value: string) {
26         console.log('set 被调用了');
27         this._name = value;
28     }
29
30     get age() {
31         return this._age;
32     }
33
34     set age(value: number) {
35         if (value > 0) {
36             this._age = value
37         }
38     }
39
40 }
41
42 const per = new Person("coco", 18);
43 // 现在属性是在对象中设置的, 属性可以任意修改
44 // - 属性可以被任意修改将会导致对象中的数据变得非常不安全
45 console.log(per);
46 per.name = 'elice';
47 console.log(per.name);
48 per.age = -33;
49 console.log(per.age); // 输出: 18
50 })();

```

3.在定义类时可以直接将属性定义在构造函数中（简化代码），实际上是语法糖

```

1  /* class Person {
2      name: string;
3      age: number;
4
5      constructor(name: string, age: number) {
6          this.name = name;
7          this.age = age;
8      }
9  } */
10
11 // ↑ 等价 ↓
12
13 class Person {
14     // 可以直接将属性定义在构造函数中
15     constructor(public name: string, public age: number) {

```

```

16
17     }
18 }
19 const per = new Person('coco', 18);
20 console.log(per.name, per.age);

```

## 8.泛型 (ts新增)

泛型就是不确定的类型 **(定义时不确定, 执行了才确定)**。在定义函数或是类时,如果遇到类型不明确就可以使用泛型。

```

1  /* function fn(a: any): any {
2      return a;
3  } */
4
5  // 在定义函数或是类时,如果遇到类型不明确就可以使用泛型。
6  /* 优点
7      1: 不用 any , 避免跳过类型检查
8      2: 体现出参数和返回值一致 */
9
10 function fn<K>(a: K): K {
11     return a;
12 }
13
14 // 可以直接调用具有泛型的函数
15 let result = fn(10); // 不指定泛型, TS 自动对类型进行推断
16 let result2 = fn<string>('hello') // 指定泛型
17
18 // 泛型可以指定多个
19 function fn2<K, T>(a: K, b: T) {
20     console.log(b);
21     return a;
22 }
23
24 let result3 = fn2<number, string>(123, 'hello');
25
26 interface Inter {
27     length: number;
28 }
29 // K extends Inter 表示泛型 K 必须是 Inter实现类 (子类)
30 function fn3<K extends Inter>(a: K) {
31     return a.length;
32 }
33
34 // 类中也可以使用泛型
35 class myClass<K> {
36     name: K;
37     constructor(name: K) {
38         this.name = name;
39     }
40 }
41 const mc = new myClass<string>('coco');

```

## 6.实战项目：贪吃蛇

### 1.项目搭建

- 准备好之前的webpack.config.js、tsconfig.json、package.json、package-lock.json四个文件，然后执行 npm i 安装依赖

- 安装依赖

```
npm i
```

新建src文件夹，新建 index.ts、index.html 文件，运行 npm run build

- 安装其他依赖

```
npm i -D less less-loader css-loader style-loader
```

- 修改webpack配置文件—在rules中添加

```
1  // 引入一个包(nodejs中一个模块，主要作用时拼接路径)
2  const path = require('path');
3  // 引入 html 插件
4  const HTMLWebpackPlugin = require('html-webpack-plugin');
5  // 引入 clean 插件
6  const {
7    cleanWebpackPlugin
8  } = require('clean-webpack-plugin');
9
10 // webpack 中所有的配置信息都应该写在 module.exports 中
11 module.exports = {
12   // 指定入口文件
13   entry: "./src/index.ts",
14
15   // 指定打包文件所在目录
16   output: {
17     // 指定打包文件目录
18     path: path.resolve(__dirname, 'dist'),
19     // 打包后文件的名字
20     filename: "bundle.js",
21     // 在build前清空dist目录所有文件，避免旧文件的遗留
22     clean: true
23   },
24
25   // 指定webpack打包时要使用的模块
26   module: {
27     // 指定要loader加载的规则
28     rules: [{
29       // test指定的是规则生效的文件
30       test: /\.ts$/, //以ts结尾的文件
31       // 要使用的loader
32       use: [
33         // 配置babel
34         {
35           //指定加载器
36           loader: "babel-loader",
```

```

37         // 设置babel
38         options: {
39             //设置预定义的环境
40             presets: [
41                 [
42                     //指定环境的插件
43                     "@babel/preset-env",
44                     // 配置信息
45                     {
46                         // 要兼容的目标浏览器及版本
47                         targets: {
48                             "chrome": "58",
49                             "ie": "11"
50                         },
51                         //指定corejs的版本（根据package.json中的版本,
只写整数)
52                         "corejs": "3",
53                         //使用corejs的方式 "usage" 表示按需加载
54                         "useBuiltIns": "usage"
55                     }
56                 ]
57             ]
58         },
59         // 'babel-loader',
60         'ts-loader'
61     ],
62     // 要排除的文件
63     exclude: /node-modules/,
64     // 设置less文件的处理
65     {
66         test: /\.less$/,
67         use: [
68             "style-loader",
69             "css-loader",
70             "less-loader"
71         ]
72     }
73 ],
74 mode: "development",
75 // 配置 webpack 插件
76 plugins: [
77     new HTMLWebpackPlugin({
78         // title: "出岫构建",
79         // 设置 html 模版 (在src文件夹下新建 index.html文件, 并设置模版样式)
80         template: "./src/index.html"
81     }),
82 ]
83 }

```

- src下新建文件夹 style, style下新建 index.less, 设置简单样式;  
在index.ts中引入样式

```
1 // 引入 css 样式
2 import './style/index.less';
```

查看是否生效

 npm run build

- 安装postcss来处理css的浏览器兼容性问题（类似babel），并在webpack中引入

 npm i -D postcss postcss-loader postcss-preset-env

```
1 // 引入一个包(nodejs中一个模块, 主要作用时拼接路径)
2 const path = require('path');
3 // 引入 html 插件
4 const HTMLWebpackPlugin = require('html-webpack-plugin');
5 // 引入 clean 插件
6 const {
7   cleanwebpackPlugin
8 } = require('clean-webpack-plugin');
9
10 // webpack 中所有的配置信息都应该写在 module.exports 中
11 module.exports = {
12   // 指定入口文件
13   entry: './src/index.ts',
14
15   // 指定打包文件所在目录
16   output: {
17     // 指定打包文件目录
18     path: path.resolve(__dirname, 'dist'),
19     // 打包后文件的名字
20     filename: 'bundle.js',
21     // 在build前清空dist目录所有文件, 避免旧文件的遗留
22     clean: true
23   },
24
25   // 指定webpack打包时要使用的模块
26   module: {
27     // 指定要loader加载的规则
28     rules: [{
29       // test指定的是规则生效的文件
30       test: /\.ts$/, //以ts结尾的文件
31       // 要使用的loader
32       use: [
33         // 配置babel
34         {
35           //指定加载器
36           loader: 'babel-loader',
37           // 设置babel
38           options: {
39             //设置预定义的环境
```

```

40         presets: [
41             [
42                 //指定环境的插件
43                 "@babel/preset-env",
44                 // 配置信息
45                 {
46                     // 要兼容的目标浏览器及版本
47                     targets: {
48                         "chrome": "58",
49                         "ie": "11"
50                     },
51                     //指定corejs的版本（根据package.json中的版本，只写整数）
52                     "corejs": "3",
53                     //使用corejs的方式 "usage" 表示按需加载
54                     "useBuiltIns": "usage"
55                 }
56             ]
57         ]
58     ],
59     },
60     // 'babel-loader',
61     'ts-loader'
62 ],
63 // 要排除的文件
64 exclude: /node-modules/
65 },
66 // 设置less文件的处理
67 {
68     test: /\.less$/,
69     use: [
70         "style-loader",
71         "css-loader",
72         //引入postcss
73         {
74             loader: "postcss-loader",
75             options: {
76                 postcssOptions: {
77                     plugins: [
78                         [
79                             "postcss-preset-env",
80                             {
81                                 browsers: 'last 2 versions'
82                             }
83                         ]
84                     ]
85                 }
86             }
87         },
88         "less-loader"
89     ]
90 }
91

```



```
92     ]
93   },
94   mode: "development",
95   // 配置 webpack 插件
96   plugins: [
97     new HTMLWebpackPlugin({
98       // title: "出岫构建",
99       // 设置 html 模版 (在src文件夹下新建 index.html文件, 并设置模版样式)
100       template: "./src/index.html"
101     }),
102   ]
103 }
```

---

1. 无法将“cnpm”项识别为 cmdlet、函数、脚本文件或可运行程序的名称↩