一、TypeScript简介（Type类型的意思）

在js中没有变量的类型，这样的话，在不同的地方操作的话，比如说，要进行购物车的加减运算，在程序中把变量不小心改成字符串了，这样一减，结果就出错，但是不会报错。

函数参数也没有类型。比如说一个加法的函数，按道理只能传number类型，但是它不管你传什么，都直接传进去。它不会报错。代码一多维护非常麻烦。

微软搞的TypeScript，它为了解决js中的缺点。它并不是去替换js。

1、TypeScript是JavaScript的**超集**。

2、它对JS进行了扩展，向JS中引入了**类型的概念**，并添加了许多新的特性。

3、TS代码需要**通过编译器编译**为JS，然后再交由JS解析器执行。

4、TS**完全兼容JS**，换言之，任何的JS代码都可以直接当成JS使用。

5、相较于JS而言，TS拥有了静态类型，更加严格的语法，更强大的功能；

TS可以在代码执行前就完成代码的检查，减小了运行时异常的出现的几率；

TS代码可以编译为**任意版本的JS代码**，可有效解决不同JS运行环境的兼容问题；

同样的功能，TS的代码量要大于JS，但由于TS的代码结构更加清晰，变量类型更加明确，在后期代码的维护中TS却远远胜于JS。

注意：TS不能直接被js解析器直接执行的，xx.ts不能在浏览器中直接执行，目前不能。

它增加了类型，支持ES的新特性，添加ES不具备的新特性，丰富的配置选项（编译成ES6、ES7、ES2020等等版本的js），强大的开发工具。

二、TypeScript 开发环境搭建

1、下载Node.js

- 64位：https://nodejs.org/dist/v14.15.1/node-v14.15.1-x64.msi

- 32位：https://nodejs.org/dist/v14.15.1/node-v14.15.1-x86.msi

2、安装Node.js

3、使用npm全局安装typescript

（1）进入命令行

（2）输入：npm i -g typescript

4、创建一个ts文件

5、使用tsc对ts文件进行编译

（1）进入命令行

（2）进入ts文件所在目录

（3）执行命令：tsc xxx.ts

三、基本类型

1、类型声明（变量）

- **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编译器会自动检查值是否符合类型声明，符合则赋值，否则报错。

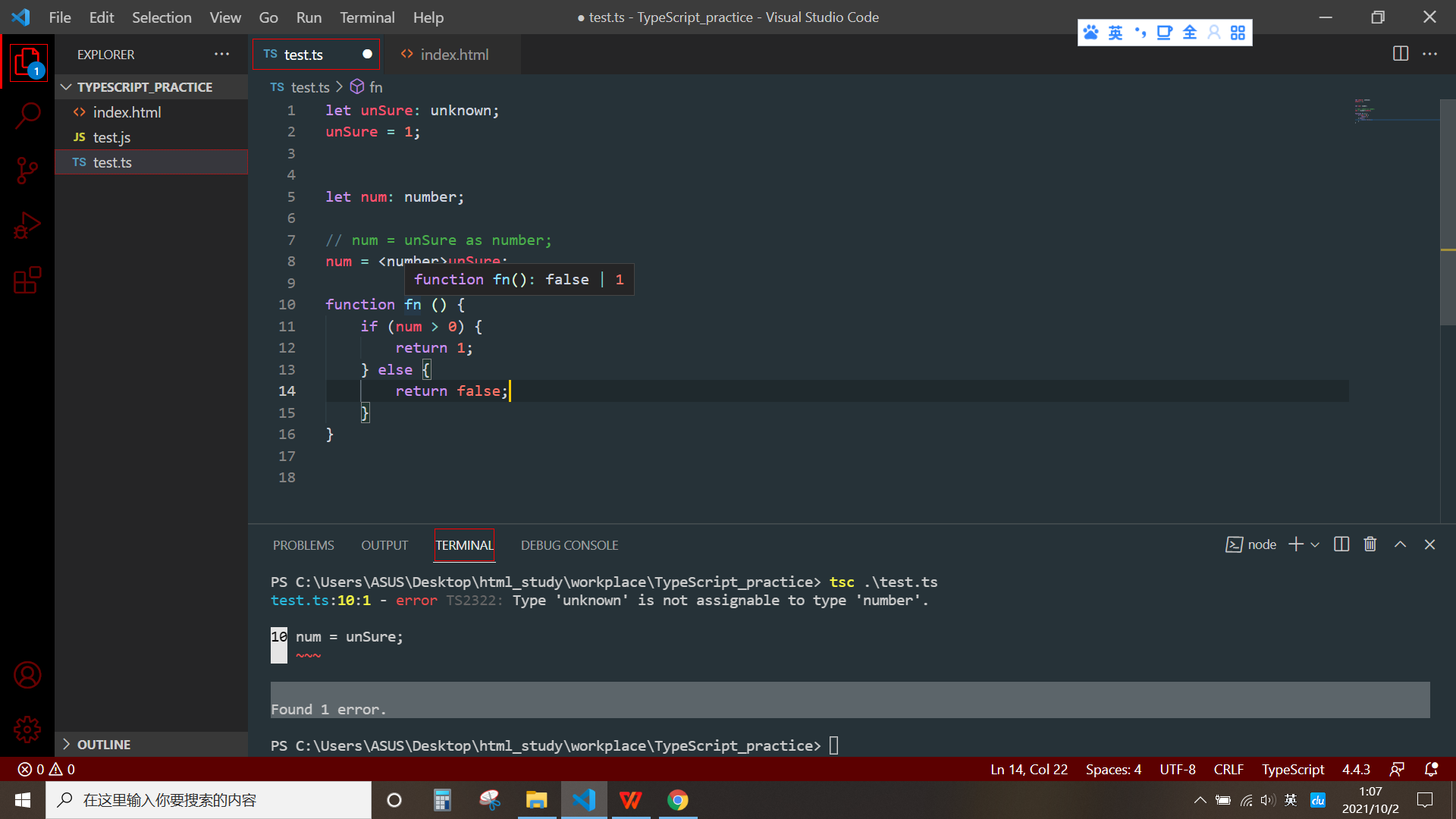
- 简而言之，类型声明给**变量设置了类型**，使得**变量只能存储某种类型的值**

- 语法：

|  |
| --- |
| let 变量: 类型;  let 变量: 类型 = 值; |
| 在vscode中会提示红色波浪线，就是有问题的。  使用tsc也会报错。但是编译能通过。这个后续能配置解决。      let在tsc下编译生成var，还没配置。 |

1. 在函数中运用
2. 参数的类型
3. 函数的返回值

返回值的类型它会根据参数随着变化，在不写情况下，默认值是什么？？？它会根据return的值来确定，没有的话void。



在js中的函数是不考虑参数的类型和个数的

|  |
| --- |
| function fn(参数: 类型, 参数: 类型): **返回值类型**{  ...  } |
| function sum (a, b) {  return a + b;  }  console.log(sum(100, 200));  console.log(sum(100, '200')); |

3、自动类型判断

（1）TS拥有自动的类型判断机制

当对变量的**声明和赋值是同时**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。

|  |
| --- |
| {  let flag:boolean = true;  flag = false;  flag = '123'; // 会报错  } |
| {  let flag = true;  flag = false;  **flag = '123'; // 会报错**  } |

4、类型（都是小写的）

|  |  |  |
| --- | --- | --- |
| 类型 | 例子 | 描述 |
| number | 1, -33, 2.5 | 任意数字 |
| string | 'hi', "hi", hi | 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  let num：10其中的10表示的是num的类型。意味着num赋值10没有问题。  num = 11就错误了，有点类似于常量，一般比较少用。 |
| any | \* | 任意类型，一个变量设置类型为any，就相当于对该变量关闭了TS的类型检测。 |
| unknown | \* | 类型安全的any |
| void | 空值（undefined） | 没有值（或undefined），主要用来**设置函数的返回空值。** |
| never | 没有值 | 不能是任何值。也是用于函数返回值，**表示永远不会返回结果。** |
| object | {name:'孙悟空'} | 任意的JS对象。 |
| array（常用） | [1,2,3] | 任意JS数组。 |
| tuple | [4,5] | 元素，TS新增类型，固定长度数组。 |
| enum | enum{A, B} | 枚举，TS中新增类型。 |

（1）number任意数字

|  |
| --- |
| let decimal: number = 6;  let hex: number = 0xf00d;  let binary: number = 0b1010;  let octal: number = 0o744;  let big: bigint = 100n; |

（2）boolean布尔值true或false

|  |
| --- |
| let isDone: boolean = false; |

（3）string任意字符串

|  |
| --- |
| let color: string = "blue";  color = 'red';  let fullName: string = `Bob Bobbington`;  let age: number = 37;  let sentence: string = `Hello, my name is ${fullName}.  I'll be ${age + 1} years old next month.`; |

1. 字面量，限制变量的值就是该字面量的值

也可以使用字面量去指定变量的类型，通过字面量可以确定变量的取值范围。

竖线表示“或”，可以赋哪些值，可以连接多个类型（联合类型）。

|  |
| --- |
| let color: 'red' | 'blue' | 'black';  let num: 1 | 2 | 3 | 4 | 5; |

（5）any任意类型

使用ts时，不建议使用，因为设置后变成js那种类型，想变就变，爱怎么变怎么变。编译器都不管了。

隐式any：声明变量如果不指定类型，则ts解析器会自动判断变量的类型为any。（避免使用）

|  |
| --- |
| let d: any = 4;  d = 'hello';  d = true; |

它可以赋值给任意的变量。

|  |
| --- |
| let d;  d = 'hello';  let num: number;  num = d; |

（6）unknown类型安全的any

实际就是一个类型安全的any，unknown类型的变量，不能直接赋值给其他变量。（不确定情况下，尽量用unknown）

|  |
| --- |
| let notSure: unknown = 4;  notSure = 'hello'; // 它的类型还是unknown |
| let unSure: unknown;  let num: number;  num = unSure; // 会提示错误 |

报错的处理方式：

|  |
| --- |
| 1. 使用typeof进行判断   let num: number;  if (typeof unSure === 'number') {  num = unSure;  **}**   1. 类型断言   **// num = unSure as number;**  **num = <number>unSure;** |

（7）void没有值（或undefined）

|  |
| --- |
| let unusable: void = undefined; |

（8）never不能是任何值

|  |
| --- |
| function error(message: string): never {  throw new **Error**(message);  }  上边的函数一执行就报错，就没有返回值了。程序中断了，不会有返回值 |

1. object任意的JS对象

在结构中的属性后边加上**问号**，表示该属性是可选的。（属性名可写少）

任意字符串的属性名：任意类型（可以按需修改）

[propertyName: string]: any

|  |
| --- |
| 1、第一种表示方式，它并不是很实用。函数也是对象，这样就没有限制的效果，一般不去使用。  let obj: object = {};  2、第二种方式：{ } 用来指定对象中可以包含哪些属性。  语法：{ 属性名：属性值, 属性名：属性值 }  一般是想去限制它拥有了哪些属性，b变量他是一个对象，同时对象里要有name属性。  这个对象中的属性，必须跟结构中的属性一样，多了也不行，少了也不行。  let obj: {name: string, age?: number, [propertyName: string]: any};  obj = {name: 'lxf', sex: '1'}  [propertyName: string]: any表示任意类型的属性（可以自己去命名的属性）   1. 设置函数结构的类型声明   语法：(形参：类型, 形参：类型) => 返回值  let fn: (a: number, b:number) => number;  fn = function (a: number, b:number) {  return a + b;  } |

1. array任意JS数组（能不用any，尽量不要用any）

数组一般是用来存放相同类型的值，比如都是number、string

方式一：类型【】

number[]、string[]...

方式二：Array<类型>

|  |
| --- |
| let list: number[] = [1, 2, 3];  let list: Array<number> = [1, 2, 3]; |

1. Tuple元组，TS新增类型，元组就是固定长度数组（数量固定的情况下，用元组比数组效率会高）

语法：[类型，类型，类型]

|  |
| --- |
| let x: [string, number];  x = ["hello", 10]; |

1. enum枚举，TS中新增类型

比例说性别，只有两种值，可以用0，1来代替。如果直接使用0，1，很难分辨出来，enum就能让它有语义化，一看就知道意思。有可读性

|  |
| --- |
| enum Color {  Red,  Green,  Blue,  }  let c: Color = Color.Green;  enum Color {  Red = 1,  Green,  Blue,  }  let c: Color = Color.Green;  enum Color {  Red = 1,  Green = 2,  Blue = 4,  }  let c: Color = Color.Green; |

5、类型断言

有些情况下，变量的类型对于我们来说是很明确，但是TS编译器却并不清楚，此时，可以通过**类型断言来告诉编译器变量的类型**，断言有两种形式：

1. 第一种：变量 as 类型

|  |
| --- |
| let someValue: unknown = "this is a string";  let strLength: number = (**someValue as string**).length; |

（2）第二种：<类型>变量

|  |
| --- |
| let someValue: unknown = "this is a string";  let strLength: number = (**<string>**someValue).length; |

&同时满足，对象可以用

（3）感叹号是非null和非undefined的类型断言

官方文档上的一个例子很好的说明了这个问题

|  |
| --- |
| interface Entity {  name: string  }  // Compiled with --strictNullChecks  function validateEntity(e?: Entity) {  // Throw exception if e is null or invalid entity  }  function processEntity(e?: Entity) {  validateEntity(e);  let s = e!.name; // Assert that e is non-null and access name  } |

如果直接使用let s = e.name;，编译器会抛出**e可能不存在的错误**，但是**使用非空断言**，**则表示e肯定是存在的，从而不会产生编译问题**。

文档的相关说明：

https://github.com/Microsoft/TypeScript/wiki/What's-new-in-TypeScript#non-null-assertion-operator

（4）应用场景

一些dom元素的类型，它可能判断会有问题，此时就可以使用断言，在贪吃蛇案例可以看到

body是一个HTMLCollection，它里边的元素类型是Element。实际上它的类型是HTMLElement。Element是接口，HTMLElement是它的子类。给他类型断言，有时候判断不来，就用类型断言，告诉它。

1. 类型的别名

语法：type 名字: 类型的组成

|  |
| --- |
| let num1: 1 | 2 | 3 | 4;  num1 = 2;  let num2: 1 | 2 | 3 | 4; |
| type typeName = 1 | 2 | 3 | 4;  let num1: typeName;  num1 = 2;  let num2: typeName; |

编译配置选项

1、自动编译单个ts文件

编译文件时，使用tsc xxx.ts -w 指令后，TS编译器会自动监视文件的变化，并在文件发生变化时对文件进行重新编译。（不如直接敲来得快，它需要一定的时间编译）

2、自动编译整个项目

如果直接使用tsc指令，则可以自动将当前项目下的**所有ts文件**编译为js文件。

但是能直接使用tsc命令的前提时，要先在项目根目录下创建一个ts的配置文件 tsconfig.json。

（1）tsconfig.json是一个JSON文件，添加配置文件后，只需 tsc 命令即可完成对整个项目的编译。

它是json文件，但是比较特殊，可以写注释。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。

|  |
| --- |
| 在项目目录下输入 tsc -init 自动生成tsconfig.json文件。  PS C:\Users\ASUS\Desktop\html\_study\workplace\TypeScript\_practice> **tsc -init**  message TS6071: Successfully created a tsconfig.json file. |

（2\*）include，用来指定哪些目录下的ts文件需要被编译

默认值：["\*\*/\*"]。

\*表示任意文件，\*\*表示任意的目录。

|  |
| --- |
| **"include":["src/\*\*/\*", "tests/\*\*/\*"]**，上述示例中，所有src目录和tests目录下的文件都会被编译。 |

（3）exclude，不需要被编译的文件目录，一般是不会用到的，它有默认值

默认值：["node\_modules", "bower\_components", "jspm\_packages"]

（4）extends

定义被继承的配置文件

|  |
| --- |
| "extends": "./configs/base"  当前配置文件中会自动包含config目录下base.json中的所有配置信息 |

（5）files，跟include比较类似，但是它是单个文件

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。需要列举每个文件

|  |
| --- |
| "files": [  "core.ts",  "sys.ts",  "types.ts",  "scanner.ts",  "parser.ts",  "utilities.ts",  "binder.ts",  "checker.ts",  "tsc.ts"  ]  列表中的文件都会被TS编译器所编译 |

（6）compilerOptions，编译器选项

编译选项是配置文件中非常重要也比较复杂的配置选项，在compilerOptions中包含多个子选项，用来完成对编译的配置。

注：当你不知道可以有哪些选项值的时候，可以给错误值去编译，编译结果会给予提示

① target，设置ts代码编译的目标ES版本。

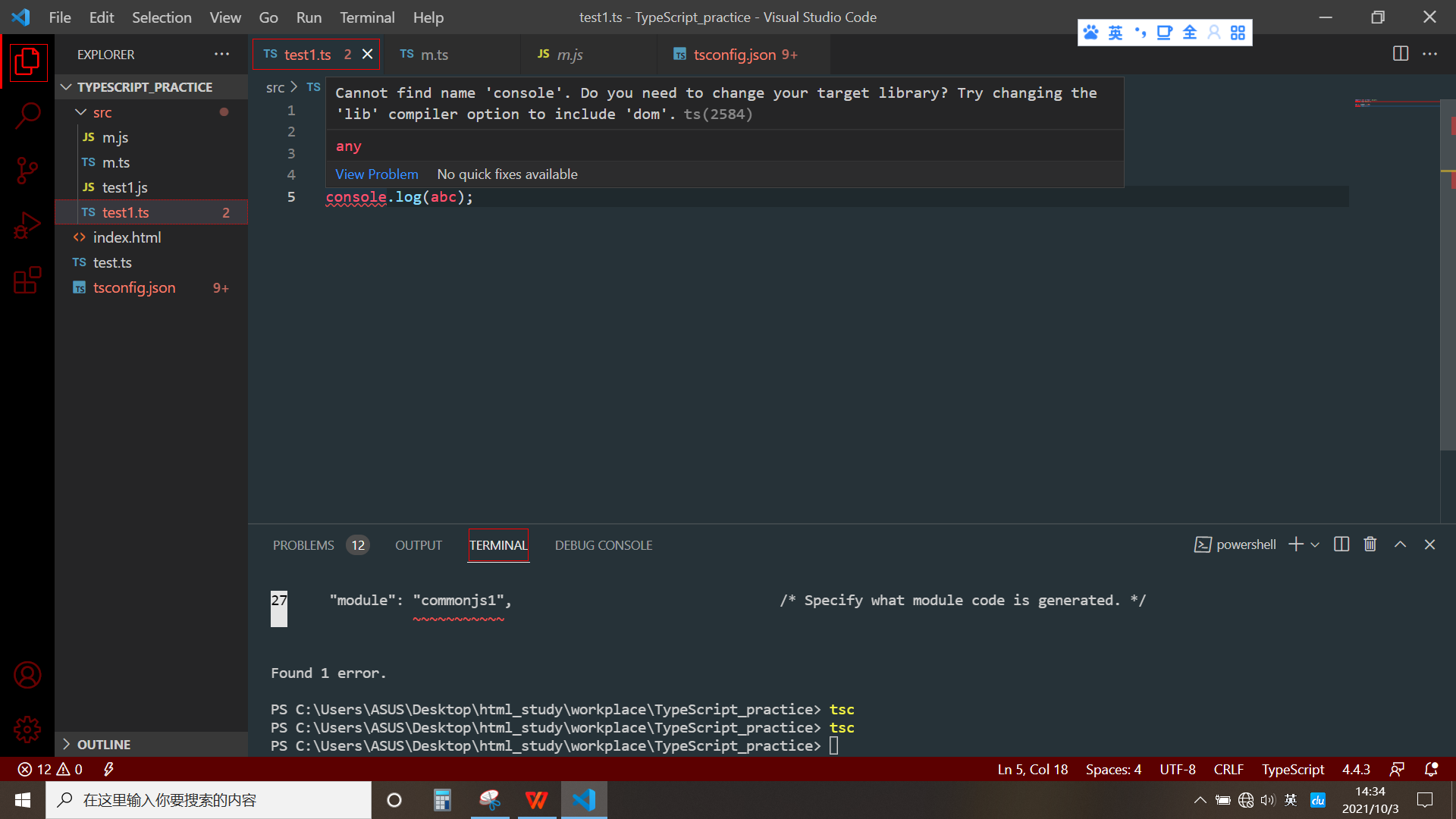
可选值：option must be: 'es3', 'es5', 'es6', 'es2015', 'es2016', 'es2017', 'es2018', 'es2019', 'es2020', 'es2021', 'esnext'.

|  |
| --- |
| "compilerOptions": {  "target": "ES6"  }  如上设置，我们所编写的ts代码将会被编译为ES6版本的js代码 |

② lib，一般情况下，不需要动的，用来指定项目中要使用的库。

可选值：ES5、ES6/ES2015、ES7/ES2016、ES2017、ES2018、ES2019、ES2020、ESNext、DOM、WebWorker、ScriptHost ......

比如说document.getElement的操作，如果 ”lib”:[] 为空的话，那么很多的这个代码无法使用。console就会提示错误。



|  |
| --- |
| "compilerOptions": {  "target": "ES6",  "lib": ["ES6", "DOM"],  "outDir": "dist",  "outFile": "dist/aa.js"  } |

③ module，指定要使用模块化的规范

可选值：option must be: 'none', 'commonjs', 'amd', 'system', 'umd', 'es6', 'es2015', 'es2020', 'esnext'.

|  |
| --- |
| "compilerOptions": {  "module": "CommonJS"  } |

|  |
| --- |
| 采用commonjs规范    采用es6规范、const变成var是被编译成了ES5版本 |

④ outDir，编译后文件的所在目录。

默认情况下，编译后的js文件会和ts文件位于相同的目录，设置outDir后可以改变编译后文件的位置。

注意：使用相对路径的话，它是相对这个tsconfig.json文件的路径而言。

|  |
| --- |
| "compilerOptions": {  "outDir": "dist"  }  设置后编译后的js文件将会生成到dist目录 |

⑤ outFile，（用的不多）将所有的文件编译为一个js文件。

默认会将所有的编写在全局作用域中的代码合并为一个js文件，如果module制定了None、System或AMD则会将模块一起合并到文件之中。

注意：Only 'amd' and 'system' modules are supported alongside --outFile.

"module": "commonjs", /\* Specify what module code is generated. \*/这个模块规范是不支持的

|  |
| --- |
| "compilerOptions": {  "outFile": "dist/app.js"  } |

⑥ rootDir

指定代码的根目录，默认情况下编译后文件的目录结构会以最长的公共目录为根目录，通过rootDir可以手动指定根目录。

|  |
| --- |
| "compilerOptions": {  "rootDir": "./src"  } |

⑦ allow**Js**，是否对js文件编译。默认是false

⑧ check**Js**，是否对js文件进行检查。默认是false

有时候js的语法是不符合的，比如说那个动画el.offsetLeft，它就只有一句话。

|  |
| --- |
| "compilerOptions": {  "allowJs": true,  "checkJs": true  } |

⑨ removeComments，是否删除注释。

默认值：false

⑩ sourceMap，是否生成sourceMap。

默认值：false。

⑪ noEmit，不生成编译后的文件。如果只想检查语法，不想得到转换后js文件，就可以使用

⑫ noEmitOnError，当有错误时，不生成编译后的文件。

|  |
| --- |
| strict  所有严格检查的总开关，默认值为true，设置后相当于开启了所有的严格检查。如果不需要严格检测某一项，需要去对其设置成false。  alwaysStrict  用来设置编译后的文件是否使用严格模式，默认false  noImplicitAny（**implicit隐式的意思**）  不允许隐式的any类型，函数的参数它能检测出来。  **但是 let a; 这个它检测不出来。？？？**  noImplicitThis  不允许不明确类型的this  function test (**this: Window**) {  console.log(this)  }  可以在参数中指定一下this的类型。这样的依据是什么？？？  strictBindCallApply  严格检查bind、call和apply的参数列表  strictFunctionTypes  严格检查函数的类型  strictNullChecks  严格的空值检查，默认是开启的。在获取dom元素的时候，可能获取不到，它有可能是null，编译代码的话是检测不到的。  它会提示波浪线，表示可能有问题，我们就通过对null值进行判断就可以。    if (box !== null) {  box.onclick = function () {  console.log('hello');  }  }  onclick无法写成下边的写法  这个？是什么操作？？？？  box?.addEventListener ('click', function () {  console.log('hello')  })  strictPropertyInitialization  严格检查属性是否初始化  额外检查  noFallthroughCasesInSwitch  检查switch语句包含正确的break  noImplicitReturns  检查函数没有隐式的返回值  noUnusedLocals  检查未使用的局部变量  noUnusedParameters  检查未使用的参数  高级  allowUnreachableCode  检查不可达代码  可选值：  true，忽略不可达代码  false，不可达代码将引起错误  noEmitOnError  有错误的情况下不进行编译  默认值：false |

webpack

通常情况下，实际开发中我们都需要使用构建工具对代码进行打包，TS同样也可以结合构建工具一起使用，下边以webpack为例介绍一下如何结合构建工具使用TS。

1、初始化项目npm init -y

进入项目根目录，执行命令 npm init -y，主要作用：创建package.json文件

2、下载构建工具

npm i -D webpack webpack-cli webpack-dev-server typescript ts-loader clean-webpack-plugin

|  |
| --- |
| 共安装了7个包  webpack  构建工具webpack  webpack-cli  webpack的命令行工具  webpack-dev-server  webpack的开发服务器  typescript  ts编译器  ts-loader  ts加载器，用于在webpack中编译ts文件  html-webpack-plugin  webpack中html插件，用来自动创建html文件  clean-webpack-plugin  webpack中的清除插件，每次构建都会先清除目录，一般是dist目录 |

3、根目录下创建webpack的配置文件webpack.config.js

注意：这里为方便查看，如果要复制使用，需要先进行处理，直接复制是有问题的，一些空格的问题。微软它把一个空格换成两个了。

|  |
| --- |
| // 引入一个包  const path = require('path');  // 引入html插件，自动生成html  const HtmlWebpackPlugin = require('html-webpack-plugin');  const { CleanWebpackPlugin } = require('clean-webpack-plugin');  const { resolve } = require('path');  // webpack中的所有配置信息都要写在module.exports中  module.exports = {      // 指定入口文件      entry: "./src/index.ts",      // 指定打包文件的目录      output: {          // 指定打包文件的目录          path: path.resolve(\_\_dirname, 'dist'),          // 打包后文件的名字          filename: 'bundle.js',          // 在输出bundle.js，它默认在最外层是一个箭头函数的自调用。它在ie是无法运行的。需要配置，因为它不是我们写的代码          // 告诉webpack不使用箭头函数          environment: {              arrowFunction: false,          }      },      // 指定webpack打包时候要使用模块,有很多模块，图片，css等等      module: {          // 指定要加载的规则          rules: [              {                  // test指定的是规则生效的文件                  test: /\.ts$/,                  // 要使用的loader,loader的加载顺序是从后往前                  // use: 'ts-loader',                  use: [                      // 配置babel                      {                          loader: 'babel-loader',                          options: {                              presets: [                                  [                                      // 指定环境的插件                                      "@babel/preset-env",                                      // 该插件的配置对象信息                                      {                                          // 我的代码运行哪个浏览器版本下                                          // 它会先转成ts，ts中配置的是转成ES6，但是这里要兼容ie11，所以bundle.js中const会被转换成var                                          targets: {                                              "chrome": "58",                                              "ie": "11"                                          },                                          // 使用哪个版本的corejs版本                                          "corejs": "3" ,                                          // 使用corejs的方式，usage按需加载，不要引入所有的功能，会影响性能                                          // 比如promise，ie11是没有的，如果不使用corejs，那么它会出错，它会添加很多代码，去实现promise                                          "useBuiltIns": "usage"                                      }                                  ]                              ]                          }                      },                      // 不用配置就写名字就可以                      'ts-loader'                  ],                  // 要排除的文件                  exclude: /node\_modules/              }          ]      },      // 配置webpack插件      plugins: [          new HtmlWebpackPlugin({              // 使用模板来生成html              template: './src/index.html',          }),            new CleanWebpackPlugin()      ],      // 设置引用模块，不然webpack不能识别模块      // 注意，js跟ts缺一不可，暂时它是怎么处理      resolve: {          extensions: ['.js', '.ts']      }  } |

4、根目录下创建tsconfig.json，配置可以根据自己需要

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

5、修改package.json添加如下配置

|  |
| --- |
| {  ...略...  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1",  "build": "webpack",  "start": "webpack serve --open chrome.exe"  },  ...略...  } |

6、在src下创建ts文件，npm run build（在）

Babel

经过一系列的配置，使得TS和webpack已经结合到了一起，除了webpack，开发中还经常需要结合babel来**对代码进行转换**以使其可以**兼容到更多的浏览器**，在上述步骤的基础上，通过以下步骤再将babel引入到项目中。

1、安装依赖包：

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

|  |
| --- |
| 共安装了4个包，分别是：  - @babel/core  babel的核心工具  - @babel/preset-env  babel的预定义环境  - @babel-loader  babel在webpack中的加载器，结合webpack  - core-js  core-js用来使老版本的浏览器支持新版ES语法，按需加载，有的用不到，不用全部加载 |

2、修改webpack.config.js配置文件

|  |
| --- |
| ...略...  module: {  rules: [  {  test: /\.ts$/,  use: [  {  loader: "babel-loader",  options:{  presets: [  [  "@babel/preset-env",  {  "targets":{  "chrome": "58",  "ie": "11"  },  "corejs":"3",  "useBuiltIns": "usage"  }  ]  ]  }  },  {  loader: "ts-loader",  }  ],  exclude: /node\_modules/  }  ]  }  ...略... |

如此一来，使用ts编译后的文件将会再次被babel处理，使得代码可以在大部分浏览器中直接使用，可以在配置选项的targets中指定要兼容的浏览器版本。

面向对象

面向对象是程序中一个非常重要的思想，它被很多同学理解成了一个比较难，比较深奥的问题，其实不然。面向对象很简单，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。

举例来说：

操作浏览器要使用window对象

操作网页要使用document对象

操作控制台要使用console对象

一切操作都要通过对象，也就是所谓的面向对象，那么对象到底是什么呢？这就要先说到程序是什么，计算机程序的本质就是对现实事物的抽象，抽象的反义词是具体，比如：照片是对一个具体的人的抽象，汽车模型是对具体汽车的抽象等等。程序也是对事物的抽象，在程序中我们可以表示一个人、一条狗、一把枪、一颗子弹等等所有的事物。一个事物到了程序中就变成了一个对象。

在程序中所有的对象都被分成了两个部分数据和功能，以人为例，人的姓名、性别、年龄、身高、体重等属于数据，人可以说话、走路、吃饭、睡觉这些属于人的功能。数据在对象中被成为属性，而功能就被称为方法。所以简而言之，在程序中一切皆是对象。

1、类（class）

要想面向对象，操作对象，首先便要拥有对象，那么下一个问题就是如何创建对象。要创建对象，必须要先定义类，所谓的类可以理解为对象的模型，程序中可以根据类创建指定类型的对象，举例来说：可以通过Person类来创建人的对象，通过Dog类创建狗的对象，通过Car类来创建汽车的对象，不同的类可以用来创建不同的对象。

（1）定义类：类中主要是包括了**属性**、**方法。**

首先需要声明属性。

|  |
| --- |
| class 类名 {  // 定义属性，不加static，是实例属性。（需要创建实例，由实例对象访问）  // 在属性前使用static关键字可以定义静态属性。（不要创建实例就可以通过类名进行使用）  属性名 **:** 类型;    // 构造函数会在对象创建的时候调用。  constructor(参数: 类型){  // 在构造函数中，this指向新建的对象  this.属性名 = 参数;  }    方法名(){  // 在实例方法中，this就表示当前实例对象  ....  }  } |

（2）示例：

|  |
| --- |
| class Person{  name: string;  age: number;  constructor(name: string, age: number){  this.name = name;  this.age = age;  }  sayHello(){  console.log(`大家好，我是${this.name}`);  }  } |

（3）使用类：

|  |
| --- |
| const p = new Person('孙悟空', 18);  p.sayHello(); |

二、面向对象的特点

1、封装性以及修饰符

对象实质上就是属性和方法的容器，它的主要作用就是存储属性和方法，这就是所谓的封装

（1）readonly只读属性：如果在声明属性时添加一个readonly，则属性便成了**只读属性，无法修改。**

（2）TS中属性具有三种修饰符：

public（默认值），可以在类、子类和对象中修改

protected ，可以在类、子类中修改

private ，可以在类中修改

（3）示例public：

|  |
| --- |
| class Person{  public name: string; // 写或什么都不写都是public  public age: number;  constructor(name: string, age: number){  this.name = name; // 可以在类中修改  this.age = age;  }  sayHello(){  console.log(`大家好，我是${this.name}`);  }  }  class Employee extends Person{  constructor(name: string, age: number){  super(name, age);  this.name = name; //子类中可以修改  }  }  const p = new Person('孙悟空', 18);  p.name = '猪八戒';// 可以通过对象修改 |

（4）示例protected

|  |
| --- |
| class Person{  protected name: string;  protected age: number;  constructor(name: string, age: number){  this.name = name; // 可以修改  this.age = age;  }  sayHello(){  console.log(`大家好，我是${this.name}`);  }  }  class Employee extends Person{  constructor(name: string, age: number){  super(name, age);  this.name = name; //子类中可以修改  }  }  const p = new Person('孙悟空', 18);  p.name = '猪八戒';// 不能修改 |

（5）示例private

|  |
| --- |
| class Person{  private name: string;  private age: number;  constructor(name: string, age: number){  this.name = name; // 可以修改  this.age = age;  }  sayHello(){  console.log(`大家好，我是${this.name}`);  }  }  class Employee extends Person{  constructor(name: string, age: number){  super(name, age);  this.name = name; //子类中不能修改  }  }  const p = new Person('孙悟空', 18);  p.name = '猪八戒';// 不能修改 |

2、静态属性

静态属性（方法），也称为类属性。使用静态属性无需创建实例，通过类即可直接使用。静态属性（方法）使用static开头。

注意：static readonly是可以写在后边的，readonly是无法写在前边的。

|  |
| --- |
| class Tools{  static PI = 3.1415926;    static sum(num1: number, num2: number){  return num1 + num2  }  }  console.log(Tools.PI);  console.log(Tools.sum(123, 456)); |

3、this

在类中，使用this表示当前对象。

4、继承性

继承时面向对象中的又一个特性。

通过继承可以将其他类中的属性和方法引入到当前类中。

Animal被称为父类，Dog被称为子类。使用继承后，子类将会拥有父类所有的方法和属性（静态方法、属性无法继承，如果有修饰符有的也不能继承）

|  |
| --- |
| class Animal{  name: string;  age: number;  constructor(name: string, age: number){  this.name = name;  this.age = age;  }  }  class Dog **extends** Animal{  bark(){  console.log(`${this.name}在汪汪叫！`);  }  }  const dog = new Dog('旺财', 4);  dog.bark(); |

（1）通过继承可以在不修改类的情况下完成对类的扩展

（2）重写

发生继承时，如果子类中的方法会**替换**掉父类中的同名方法，这就称为方法的重写。（相当于它在原型添加了自己的方法，就不会去找原型链上父类的方法）

|  |
| --- |
| class Animal{  name: string;  age: number;  constructor(name: string, age: number){  this.name = name;  this.age = age;  }  run(){  console.log(`父类中的run方法！`);  }  }  class Dog extends Animal{  bark(){  console.log(`${this.name}在汪汪叫！`);  }  run(){  console.log(`子类中的run方法，会重写父类中的run方法！`);  }  }  const dog = new Dog('旺财', 4);  dog.bark(); |

（3）在子类中可以使用super来完成对父类的引用。

在类的方法中，super就表示当前类的父类。

在子类的构造函数中必须对父类的构造函数进行调用，不去调用，继承会有问题。super()就是表示调用父类的构造函数。

（4）子类的构造函数如果不写是默认调用父类的构造函数吗？？？

|  |
| --- |
| class Person {  name: string;  age: number;  static staticPropertyTest: string;  constructor (name: string, age: number) {  this.name = name;  this.age = age;  }  say () {  console.log('hello');  }  static staticMethodTest () {  console.log('静态方法');  }  }  class Student **extends** Person {  // 学号的属性  num: number;  constructor (name: string, age: number, num: number) {  **super**(name, age);  this.num = num;  }  say () {  console.log('i am a student');  }  showNum () {  console.log(`my num is ${this.num}`)  }  }  var stu = new Student('Tom', 33, 123456);  stu.showNum(); |

5、抽象类（abstract class）

抽象类是**专门**用来**被其他类所继承**的类，它只能被其他类所继承**不能用来创建实例**。它不希望被用来创建对象，它表示的范围比较大。

它可以添加抽象方法。

|  |
| --- |
| abstract class Animal{  // 定义一个抽象方法，抽象方法使用**abstract开头**，没有方法体。  // 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**。  没有函数体的  abstract run(): void;  bark(){  console.log('动物在叫~');  }  }  class Dog extends Animals{  run(){  console.log('狗在跑~');  }  } |

（1）抽象方法

使用abstract开头的方法叫做抽象方法，抽象方法没有方法体只能定义在抽象类中，继承抽象类时抽象方法**必须要实现**。

三、接口（Interface）

接口就是用来定义一个类的结构。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。

1、接口的特点

（1）接口是可以重复声明的，可以添加别的属性

（2）接口中的所有属性都不能有实际的值

（3）在接口中所有的方法都是抽象的方法（没有函数体）

2、作用：定义类时，让类去实现接口

实现接口就是使类满足接口的要求，意思就是要求有name属性，和一个say的方法的实现。

接口就是定义了一个规范，实现了这个接口就是满足了这个规范。

|  |
| --- |
| interface myInterface {  name: string;  say (): void;  }  class TestInterface **implements** myInterface {  name: string;  constructor (name: string) {  this.name = name;  }  say () {  console.log('hello');  }  }  const test = new TestInterface('lxf');  test.say(); |

四、属性修饰符

现在属性是在对象中设置的，属性可以任意的被修改。

属性可以任意被修改将会导致对象中的数据变得非常不安全。

1、修饰符

默认情况下，对象的属性是可以任意的修改的，为了确保数据的安全性，在TS中可以对属性的权限进行设置。

（1）public 修饰的属性可以在**任意位置**访问（修改），子类可以访问，它是默认值

1. private 私有属性，它只能在**类内部**进行访问（修改）。可以通过在类中添加方法使得私有属性可以被外部访问

（3）protected 受保护属性，**只能在当前类和当前类的子类**中进行访问（修改）

2、外部去 读取/设置 private属性（更安全）

直接将其设置为private将导致无法再通过对象修改其中的属性。

**读取属性**的方法叫做getter方法，**设置属性**的方法叫做setter方法（在方法里边就可以对数据的有效性进行判断，比如分数是一般是没有负值的，如果通过属性设置的方式就无法达到判断的效果），这只是它的一种叫法，它命名可以根据需求设置。

好处：它的修改都是由我提供的方法来修改的，控制权在我的手上，我不想可以修改，就不提供这个方法。

|  |
| --- |
| class Person {  private name: string;  private age: number;  constructor (name: string, age: number) {  this.name = name;  this.age = age;  }  // 一般的getter、setter方法，获取，设置的方法  getName () {  return this.name;  }  setAge (age: number) {  if (age >= 1) {  this.age = age;  }  }  }  const tom = new Person('Tom', 22);  console.log(tom.getName());  tom.setAge(33);  console.log(tom); |

|  |
| --- |
| 实际采用ES6的get，set方法？？？  属性的名字跟这里的get、set的名字可能稍微要有点区别，不然会报重复定义错误  class Person {  private **\_name**: string;  private \_age: number;  constructor (name: string, age: number) {  this.\_name = name;  this.\_age = age;  }    // 通过 ts 设置、获取方法  get **name**() {  console.log('我是通过get方法');  return this.\_name;  }  set name(name: string) {  console.log('我是通过set方法');  this.\_name = name;  }  }  const tom = new Person('Tom', 22);  console.log(tom.name);  tom.name = 'Jerry';  console.log(tom); |

（1）编译后的js代码

|  |
| --- |
| class Person {  \_name: string;  constructor (name: string) {  this.\_name = name;  }  get name () {  return this.\_name;  }  set name (name: string) {  this.\_name = name;  }  } |
| "use strict";  var Person = /\*\* @class \*/ (function () {      function Person(name) {          this.\_name = name;      }      Object.defineProperty(Person.prototype, "name", {          get: function () {              return this.\_name;          },          set: function (name) {              this.\_name = name;          },          enumerable: false,          configurable: true      });      return Person;  }()); |

3、类中属性的简写方式（属于语法糖）

可以直接将**属性定义在构造函数**中，在构造函数参数中**添加上属性修饰符**就可以，在函数体内不用this赋值。

|  |
| --- |
| class Person {  name: string;  age: number;  constructor (name: string, age: number) {  this.name = name;  this.age = age;  }  }  let tom = new Person('Tom', 33);  console.log(tom);  class Person1 {  constructor (**public** name: string, **public** age: number) {  // this.name = name;  // this.age = age;  // 这些语句就可以不用写了，按上边的写法，它就相当于这样操作了。  }  }  let jerry = new Person1('Jerry', 44);  console.log(jerry); |
|  |

五、泛型（Generic）

定义一个函数或类时，有些情况下无法确定其中要使用的具体类型（返回值、参数、属性的类型不能确定），此时泛型便能够发挥作用。

|  |
| --- |
| function test(arg: any): any{  return arg;  } |

上例中，test函数有一个参数类型不确定，但是能确定的时其返回值的类型和参数的类型是相同的，由于类型不确定所以参数和返回值均使用了any，但是很明显这样做是不合适的，首先使用any会关闭TS的类型检查，其次这样设置也**不能体现出参数和返回值是相同**的类型。

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。

1、使用泛型：

|  |
| --- |
| function test<T>(arg: T): T{  return arg;  } |

这里的**<T>就是泛型**，T是我们给这个类型起的名字（不一定非叫T），设置泛型后即可在函数中使用T来表示该类型。所以泛型其实很好理解，就表示某个类型。

arg: T这里之所以能使用 T ，是前边 <T> 定义了。在函数执行的时候，才能确定。

2、如何使用上边的函数呢？

（1）方式一（直接使用）：可以直接调用具有泛型的函数。

|  |
| --- |
| test(10)  a的类型是number，它会自动判断 T 为 number，这个泛型 T 就是 number。 |

使用时可以直接传递参数使用，类型会由TS自动推断出来，但有时编译器无法自动推断时还需要使用下面的方式。

（2）建议方式二（指定类型）：能减少出错几率

|  |
| --- |
| test**<number>**(10) |

也可以在函数后手动指定泛型。

3、可以同时指定多个泛型，泛型间使用逗号隔开：

|  |
| --- |
| function test<**T**, **K**>(a: **T**, b: **K**): K{  return b;  }  test<number, string>(10, "hello"); |

使用泛型时，完全可以将泛型当成是一个普通的类去使用。

4、类中同样可以使用泛型：

|  |
| --- |
| class MyClass<T>{  prop: T;  constructor(prop: T){  this.prop = prop;  }  } |

5、泛型的范围进行约束（不是很理解）

|  |
| --- |
| interface MyInter{  length: number;  }  function test<T extends MyInter>(arg: T): number{  return arg.length;  }  test<string>(‘123’)，字符串123，它里边有length属性 |

使用T extends MyInter表示**泛型T必须是MyInter的子类**，不一定非要使用接口类，抽象类同样适用。

六、奇怪的问题

在vscode打开tsc编译之后的js文件，在ts文件中会显示函数重复声明，把tsc编译出来的js文件关闭就不会。它会导致其他奇奇怪怪的问题