TypeScript

微软公司， 设计的 TypeScript

TS是什么？

出现的目的： 解决 JS 出现的一些问题， TS 是以 JS 为 基础构建的语言，是一个JS的超集

TS 完全支持，兼容JS， 给动态的JS变量类型， 设置为静态的变量类型。 主要是加上了 Type

TS特点

- TypeScript 扩展了 JS， 并添加了类型
- 可以在任何支持JS的平台中执行
- TS不能被JS解析器直接执行， 需要将 TS 编译为 JS文件

增加了类型

变量

ts 的变量

1. Ts 的变量声明

```
// 1. Ts 的变量声明

let a:number;

// a 的类型设置为了 Number， 在以后的使用过程中  a 的值只能是 number 类型
a = 10;

a = 33;

// a = 'hello';    // 这里直接报错了，不能赋值为 字符串 ，默认情况下，ts 会报错了，依旧会被编译为js文件
```

2. 可以使用 | 来 连接多个类型 （联合类型）

```
// let c: 'male' | 'female'
let c: boolean | string;  //  c  可以是 boolean 类型，  也可以是 string 类型
c = true
c = 'true'
```

3. *any* 类型

```
// 一个变量设置类型为 any 后，相当于对该变量关闭了Ts的类型检测

// let d:any;  // any 可以是任以类型  - 显示类型
// 同样 直接声明 d，没有确定类型，  它的类型也是 any
let d;  // 默认类型为 any  - 声明变量，不指定类型，  ts 解析器 自动判断类型为 any  (隐式类型)
d = 12;
d = 'abc';
d = false;
d = [1, 2, 3];

// any 的缺点，  会污染其他类型变量
// 例如
let s: string;

s = d;  // 不会报错，  s 赋值为 d ;  d 此时是一个 [] ,  而 s 则为 string
// s 出现变量被污染  ->  any 的缺点
```

4. *unknown* 表示位置的值

```
// 4.  unknown 表示位置的值
let e: unknown;
e = 10;
e = 'hello';

// s = e;  // 这里会报错，  e 的类型不确定 s -> String,   e -> unknown

// unknown 实际上就是一个类型 安全的 any
// unknown 类型的变量，  不能直接赋值给其他变量  -> 需要类型判断

if (typeof e === 'string') {
  s = e;  // 这样就不会报错
}
```

5. 类型断言 ： 直接告诉编译器 *xxx* 类型就是 *xx* 类型 实际类型

```
// 5.
// 类型断言  ：直接告诉编译器  xxx 类型就是 xx 类型  实际类型

s = e as string;   // x as 类型
// 第二种写法
s = <string>e;
/**
 * 语法
 * 1.  xx as <类型>
 * 2.  <类型>xx
 */
```

6. *void* 用来表示没有返回值

```
// 以函数为例
function fn(): void {
  // 1. 不写return
  // 2. return;    return 为空值
  // 3. return undefined | null 。 都表示为空
  // 4. return 123 // 会报错
}
```

7. *never* 表示永远不会返回结果，

```
// 7. never 表示永远不会返回结果，
function fn2(): never {
  throw new Error('报错了')
}
```

引用类型的变量

1. *object*

```
// 1.  Ts 的 object


let a:object; //   object 表示一个对象

a = {};
a = function () {};  // function 也是一个对象

// a:object 不是很实用， 因为JS中 object 的类型数据实在太多了
```

使用 `type` 自定义

```
// 描述一个对象类型
type myType = {   // 自定义的类型
  // 对下面的obj 类型 做限制
  name: string,
  age: number,
}

// 使用 type <Name> {}  设置对象的类型是常见的


// 声明对象
const obj:myType = {  // 使用自定义的类型
  name: 'Yellowsea',
  age: 1
}
console.log(obj)
```

2. {} 用来指定对象中可以包含哪些属性

```
// 2. {} 用来指定对象中可以包含哪些属性
// 语法: {属性名: 属性值}
// 在属性后面加上 ? 表示属性是可选的

let b: {name: string, age?: number};

// b = {name: "Yellowsea"};
// b = {name: "123", age: 123}; // 报错，必须按照 b 定义的方式，多一个少一个都不行

// 在 加上 age? 后
b = {name: "Yellowsea", age:123};
b = {name: "Yellowsea"};  // 都是可以的


// 定义任意多个类型的属性
// [propName: string]: any   表示任意类型的属性 propName: 属性名-> string 属性值 :any

let c: {name: string, [propName: string]: any}; // 常用的语法
c = {name: "Yellowsea", age:123, sex: '男'}  // 这样写多个属性


// 2. 设置函数结构的类型声明
// 语法: (形参:类型, 形参:类型...)=> 返回值

// 设置 d 为一个函数，a,b 都是参数， 返回值是 num 类型
let d:(a:number, b:number) => number;  // 用来定义函数的结构 ->  常用的语法

d = function (n1, n2) { // n1, n2 对应 a,b 。 多一个少一个都不行
  return n1 + n2
}
```

3. *array* 数组

```
// 3. array 数组
// Ts定义 数组的语法:
/**
 *  - 类型[]
 *  - Array<类型>
 */

// string[] 表示字符串数组
let e:string[];

e = ['a','b', 'c'];  // good
// e = ['1', '2', 3];  // bad


// number[] 表示数值数组
let f:number[];

// 或者
```

```ts
let g: Array<number>;  // 也表示存储Number 类型的数组
g = [1,2,3];


```

4. 元组，元组就是固定长度的数组

```ts
/**
 * 4. 元组，元组就是固定长度的数组
 * 定义元组： let h: [string, string];
 *  - 语法：  [类型， 类型， 类型]
 */

// 定义元组，
let h: [string, string];  // 固定两个长度的 string 类型

h = ["hello", "abc"];   // good
// h = [1, "1"]   // bad

```

5. *enum* 枚举

```ts
/**
 * 5. enum 枚举
 *  - Ts 语法特有的
 */

// 普通定义数据时 设置性别为 0 | 1
// let i: {name: string, gender: 0 | 1};

// i = {
//   name: 'Yellowsea',
//   gender: 0
// }

// 但是这样定义会出现问题，当用户使用时，不知道 0 | 1 是什么、


// 使用枚举
enum Gender { // 定义枚举
  // 不需要知道枚举的内容是什么
  Male,
  Female
}

// 使用枚举
let i: {name: string, gender: Gender};  // gender 使用 枚举， Gender

i = {
  name: 'Yellowsea',
  // gender: Gender.Male
  gender: Gender.Female,  // 这是好的定义
}

console.log(i.gender == Gender.Male);
```

6. 补充内容

```
// 6. 补充内容

// |  或
// let j: string | number;   //  j 的类型是 string 或  number 类型
//  j = "hello"  | j = 123
// & 与

// let j: {name: string} | {age: number}

let j: {name: string} & {age: number}

j = {name: 'yellowsea', age: 123};
```

tsconfig配置选项

使用 `tsc --init` 进行 初始化， 会自动创建 `tsconfig.json` 文件

include

`include` 编译指定目录下的需要编译的文件

```
{
  /**
  include
  配置ts编译器，编译指定目录下的需要编译的文件
  写在 tsconfig.json 的最外侧
    路径:  ** 表示任意目录
           * 表示任意文件
  */
  // 编译指定目录下的需要编译的文件
  "include": [
    "./src/**/*"
  ],
}
```

exclude

`exclude` : 需要排除编译的文件

```
{
  /**
      exclude : 需要排除编译的文件
        // 它是可选的， 具有默认值：  ["node_modules", "bower_components", "jspm_packages"]
    */
  "exclude": [
    // 不让 hello 文件夹下的文件被编译
    "./src/hello/*"
  ],
}
```

extends

`extends` 继承 配置

```
{
    /*
   extends 继承   配置。
    - 比如你还有另一个 xxx.json 的配置文件，可以通过 extends 引入进来
    - 相当于  合并两个文件的配置

    - 这里不演示

    - 语法： "extends": "./configs/base"
  */

  "extends": "./configs/base.json",
}
```

file

`file` 表示需要编译的单个文件， 需要列举出来

```
{
    /*
    flies :   文件。 跟include 类似，
      - 表示需要编译的单个文件， 需要列举出来
      语法：
        files: [
          "./fileName.ts",
          ...
        ]

    // 在 base.json 中 配置 files
  */
}
```

compilerOptions

"`compilerOptions`" 编译器的选项， 在 *tsconfig.json* 中最重要的配置

学习配置*ts* ， 也就是 学习配置 *compilerOptions* , 学会它的 子选项的配置

> 这里的配置项基本上都有自己的默认值，如果需要配置某个配置， 设置一个错误的值，编译后出错，可以看到所有的选项

1. target

```
//  ts 编译器的选项
"compilerOptions": {
  //1. target 用来指定 ts 文件被编译为 ES 几 的版本
  // 默认TS会转为 ES3 版本， 兼容行好
  // target: 的属性值 :
  // 'es3', 'es5', 'es6', 'es2015', 'es2016', 'es2017', 'es2018', 'es2019', 'es2020', 'es2021', 'es2022', 'esnext'.
  "target": "es6",
}
```

2. module

```
"compilerOptions": {
  //2. module 指定要使用的模块化的规范
  // module的属性值： 'none', 'commonjs', 'amd', 'system', 'umd', 'es6', 'es2015', 'es2020', 'es2022', 'esnext', 'node12', 'nodenext'
  "module": "commonjs",
}
```

3. *lib*

```
"compilerOptions": {
  //3. lib 用来指定项目中要用到的库 - 浏览器运行环境
  // 一般不去动它，也不用去配置，如果写了， 什么都不配置，表示没有使用任何库， 然后写TS代码时候，没有任何补全
  // 它的属性值很多，在写 lib: xx 让它报错，然后可以查看它的属性值
  // "lib": ["DOM", "ES2015"]
}
```

4. outDir

```
"compilerOptions": {
  //4. outDir 用来指定编译后文件所在的目录
  // 表示 编译TS文件后 生成 JS文件存放的目录,  将源码 和 生成的目录分开
  "outDir": "./dist",
}
```

5. *outFile*

```
"compilerOptions": {
  // 5. outFile  将代码合并为一个文件
  // 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
  // "outFile": "./dist/app.js",  // 使用时 module 需要改为 system
  // 用的不多，一把结合打包工具使用
}
```

6. *allowJs*

```
"compilerOptions": {
  // 6.  allowJs  是否 对JS文件进行编译,  默认 false
  //  改为true 后，同样编译 src 目录下的  js 文件
  // "allowJs": false,
  "allowJs": true,   // 一般用在项目中，不得不使用的第三方库,  js 文件的编译
}
```

7. *checkJs*

```
"compilerOptions": {

  // 7.  checkJs 是否检查js代码是否符合语法规范,  默认值为 false
  // "checkJs": false,   // 不检查js的语法
  "checkJs": true,  // 和 allowJs 具有冲突，一般使用它们之间的其中一个
}
```

8. *removeComments*

```
"compilerOptions": {
  // 8. removeComments : 在编译ts中,  是否移除注释 , 默认为false
  "removeComments": true,
}
```

9. *noEmit*

```
"compilerOptions": {
  // 9. noEmit 不生成编译后的文件, 默认为false
  // "noEmit": false
  // "noEmit": true,  // 不常用
}
```

10. *noEmitOnError*

```
"compilerOptions": {
  // 10. noEmitOnError :  当有错误时候 不生成  编译后的文件,  默认为 false
  "noEmitOnError": true,
}
```

11. *strict*

TS 的语法检查 选项

```
"compilerOptions": {
  // 接下来就是 TS 的语法检查 选项
  // strict : Ts 语法检查的总开关,  默认为 false,
  "strict": true, //  项目中都会打开，这样代码出错的比较少
}
```

`alwaysStrict`

```
"compilerOptions": {
  // 接下来就是 TS 的语法检查 选项
  // strict : Ts 语法检查的总开关,  默认为 false,
  "strict": true, //  项目中都会打开，这样代码出错的比较少


  // 11. alwaysStrict :  用来设置编译后的文件是否使用 严格模式,  默认为 false
  "alwaysStrict": true,  // 编译生成的 JS文件,  会加上 严格模式  : "use strict";

}
```

`noImplicitAny`

```
"compilerOptions": {
  // 接下来就是 TS 的语法检查 选项
  // strict : Ts 语法检查的总开关， 默认为 false,
  "strict": true, //  项目中都会打开，这样代码出错的比较少



  // 12. noImplicitAny : 是否允许隐式的 any 类型 ,  默认是false
  "noImplicitAny": true,
}
```

`noImplicitThis`

```
"compilerOptions": {
  // 接下来就是 TS 的语法检查 选项
  // strict : Ts 语法检查的总开关， 默认为 false,
  "strict": true, //  项目中都会打开，这样代码出错的比较少


  // 13. noImplicitThis :  是否允许 不明确类型的 this , 默认值为 false;
  "noImplicitThis": true,
}
```

`strictNullChecks`

```
"compilerOptions": {
  // 接下来就是 TS 的语法检查 选项
  // strict : Ts 语法检查的总开关， 默认为 false,
  "strict": true, //  项目中都会打开，这样代码出错的比较少

  // 14. strictNullChecks : 严格检查空值
  "strictNullChecks": true,
}
```

webpack + TS

项目初始化

```
mkdir webpack-ts &&  cd webpack-ts

npm init -y

tsc --init

mkdir src && cd src && touch index.ts

touch webpack.config.js
```

```
// 安装的依赖
npm install -D \
  ts-loader \
  typescript \
  webpack \
  webpack-cli \
  webpack-dev-server \
  html-webpack-plugin \
  core-js \
  @babel/core \
  @babel/preset-env \
  babel-loader \
```

源代码

src/index.ts

```
// 省略了 m 的扩展名

import { name } from "./m"

console.log("Hello webpack+Ts");


let div = document.createElement('div');
// const box = document.querySelector('#box');
if (div !== null) {
  div.innerHTML = `<h1>Hello Webpack + Ts </h1>`
}

document.body.append(div);


// 写好的TS代码
function sum (a: number, b: number): number {
  return a + b;
}

console.log(sum(1,2));
console.log(sum(123, 456));
```

```
console.log(sum(123, 777));
console.log(name);


// m.ts

export const name = "yellowsea";

let num = 123;

num = 10;

console.log(num);

```

src/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>webpack + Ts</title>
</head>
<body>
</body>
</html>
```

配置项

package.json

```
{
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack",
    "start": "webpack serve --open"
  },
}
```

tsconfig.json

```
{
  // ts 编译器的配置
  "compilerOptions": {
    //  也可以使用 commonjs
    "module": "ES6",
    "target": "ES6",
    "strict": true,
    "sourceMap": true,  // 编译后的文件出错时， 回溯源代码的所在的位置的映射
  }
}
```

wbepack.config.js

```

const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');


module.exports = {
  // mode: 'development',
  mode: 'production',
  entry: {
    index: './src/index.ts',
  },
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
    clean: true
  },
  //
  module: {
    rules: [
      // 规则
      {
        test: /\.ts$/,
        use: [
          // 配置babel
          {
            loader: 'babel-loader',
            // 配置babel
            options: {
              // 设置预定的环境
              presets: [
                [
                  // 指定的环境
                  "@babel/preset-env",
                  // 配置信息
                  {
                    // 要兼容的浏览器
                    targets: {
                      "chrome": "58",
                      "ie": "11"
                    },
                    // 指定corejs 的版本
                    "corejs": "3",
```

```
                    // 使用 corejs 的方式,  "usage" 表示按需加载
                    "useBuiltIns": "usage"
                  }
                ]
              ]
            }
          },
          'ts-loader'
        ],
        exclude: /node_modules/
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: 'webpack + ts',
      template: './src/index.html'
    })
  ],

  resolve: {
    // 省略引入的扩展名
    extensions: ['.js', '.jsx', '.ts']
  }
}


/**
 * {
 *  loader: 'babel-loader',
 *  presets: [
 *    "@babel/preset-env"
 *  ]
 * }
 */
```

TS 类

TS 的类和JS类基本一样，都是使用 `class` 关键字

类中包含主要两部分

- 属性
- 方法

```
class Person {

```

```
  // 定义属性 - 在 class 内的 叫做实例属性  - 需要通过 实例去访问
  name:string = 'Yellowsea';
  // 加上 readonly 后 只能读， 不能修改
  readonly age:number = 123;


  // 在属性前 使用 static 关键字 可以定义 类属性，  叫做静态属性
  // 静态属性： 定义类时， 不用创建实例，通过类名，能够访问到的属性
  static test:string = '这是static test'



  // 定义方法  --- 实例方法
  sayHello() {
    console.log("Hello")
  }

  // 静态方法
  static sayHello2() {
    console.log("Hello2")
  }
}

const p = new Person();


console.log(p);
console.log(p.age);
console.log(p.name);

// 通过实例修改 实例属性
p.name = 'Hidie'
console.log(p.name);

// p.age = 123123;   // readonly ，修改不了


// console.log(Person.age);  // 访问不到 age

console.log(Person.test);  // 能够访问到test

Person.test = "testtest";  // 可以修改， 静态属性加上 readonly 后也修改不了
console.log(Person.test);



// 通过实例调用 方法
p.sayHello();

// 加上 static 方法名 ，静态方法

Person.sayHello2();
```

类的构造函数

```

class Dog {
  // TS 需要在类中 提前定义 name age 并赋予类型
  name: string;
  age: number;
  // 构造函数   constructor
  // 构造函数 是在new XXX 的时候， 传递参数，在构造函数中能够获取得到 参数
  constructor(name:string, age:number) {
    // 在 new XXX 的 时候执行 ，  此时的 this 就表示当前的的 实例
    // console.log("constructor执行了", this)

    this.name = name;
    this.age = age;

    // console.log(this)
  }
  bark() {
    // 这里的 this 表示 当前调用方法的对象
    console.log("bark", this)
  }
}


const dog = new Dog("小黑", 123);
const dog2 = new Dog("小白", 123);

dog.bark();
dog2.bark();

```

继承

```

  // 将所有的 变量 写在 这个作用域里
  // class Dog {
    // name: string;
    // age: number;

    //  constructor(name:string, age:number) {
    //    this.name = name;
    //    this.age = age;
    //  }

    //  sayHello () {
    //    console.log('wangwanga')
    //  }
  // }

  // class Cat  {
    // name: string;
    // age: number;

    //  constructor(name:string, age:number) {
    //    this.name = name;
    //    this.age = age;
```

```
    //  }

    //  sayHello () {
    //    console.log('wangwanga')
    //  }
  // }
```

```
// 因为在学习ts过程中， ts 会检查当前目录的所有变量， 当变量名相同 就会报错

// 使用 立即执行函数  解决

(function () {
  class Animal {
    name: string;
    age: number;

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

     sayHello (call:string) {
       console.log(call)
     }
  }

  // 使用 extend 关键字 继承 父类的 方法 和 属性
  class Dog extends Animal {

    // Dog 类 本身自己的方法
    run () {
      console.log(this.name, "在跑````")
    }
  }

  class Cat extends Animal {
    // 修改父类的 sayHello 方法
    sayHello() {
      console.log(this.name, "miaomiaoamia asdsadasd")
    }
  }

  const dog = new Dog("旺财", 123);
  dog.sayHello("wangwangawang");
  dog.run();

  const cat = new Cat('咪咪', 123123);
  cat.sayHello();

  //
})()
```

super

```
// super 关键字
(function () {
  class Animal {
    name: string;
    constructor(name:string) {
      this.name = name;
    }
    sayHello () {
      console.log('sayHello');
    }
  }

  class Dog extends Animal {

    // 2. 在子类中使用 构造函数
    age: number;
    constructor(name:string, age:number) {

      // 在最前面 调用父类的  constructor 方法,
      // 直接 调用 super() -  必须手动调用
      super(name)   // 这里就是在调用 父类的构造函数

      this.age = age;  // 报错


      // 因为，继承时，相同的方法会发生重写， 子类写了 constructor 构造函数，相当于重写了 父类的构造函数

      // 解决， 在使用子类的 构造函数时， 必须先进行 对父类的构造函数的调用 , 执行 super()

    }



    // 1.
    sayHello () { // 表示重写

      // 然后使用 super 关键字
      super.sayHello();
      // 这里的super 表示当前的父类， 能够拿到父类的的属性|方法

      // super , 一般用在 子类 继承 父类时， 需要使用 父类的 方法 或 属性时， 进行调用

      // 例如 调用 查看父类的 name
      // console.log(super.name);  // undefined
      // console.log(super);   // super 后面必须要跟 父类的 某一个属性 或 方法
    }
  }



  const dog = new Dog("旺财", 123);

  dog.sayHello()
})()
```

抽象类

```
(function () {

  /**
   * 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 用来创建兑现实例化
   *    - 抽象类就是专门用来被继承的类
   */
  abstract class Animal {
    name: string;
    constructor(name:string) {
      this.name = name;
    }

    // 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
    abstract sayHello():void;



    // sayHello () {
    //   console.log(this.name);
    //   console.log('sayHello');
    // }
  }

  class Dog extends Animal {
    //必须要对 sayHello方法进行重写
    sayHello() {
      console.log(this.name);
    }
  }

  class Cat extends Animal {
    sayHello() {
      console.log(this.name);
    }
  }

  const dog = new Dog("🐕");
  dog.sayHello()

  const cat = new Cat("🐱")
  cat.sayHello()

  // 这里直接 实例化  Animal 这里大的总类，
  // const an = new Animal("🐱")  // 使用 abstract class Animal 抽象类后  这里不能够 实例化了
  // an.sayHello()



```

```
  // 以上类方法中的 Animal 类， 如果 Animal 是一个 大型的 类， 并且只能拿来继承

  // 所以  可以把Animal 变为一个 抽象类
})()
```

属性的封装

```

/**
 * 属性的封装
 */


(function() {

  class Person {

    // Ts 独有的 对属性的 修饰方式
    /**
     * 1. public 修饰的属性可以在任意位置 访问 包括子类 (修改)  默认值，
     *
     * 2.  private 私有属性， 私有属性只能在类内部进行访问，（修改）
     *    - 如果徐需要修改，通过在类中添加方法使得私有属性可以被外部访问
     *
     * 3. protected  受包含的属性，只能在当前类 和 当前类的子类 能访问，其他情况下不能访问
     */

    // 默认值 public _name   可以读， 也可以修改
    // _name: string;
    // _age: number;

    private _name: string;
    private _age: number;

    constructor(name: string, age: number) {
      this._name = name;
      this._age = age;
    }

    /**
     * getter 方法用来读取属性
     * setter 方法用来设置属性
     * 这两个方法被称为属性的存储器
     *
     * 一把JS中对 属性的处理基本上是这样处理
     *
    // 读取属性值
    getName() {
      return this._name;
```

```
    }
    // 修改属性值
    setName(value: string) {
      this._name = value;
    }
    // age 也一样
    getAge () {
      return this._age
    }
    setAge (value: number) {
      // 进行判断
      if (value >= 0) {
        this._age = value
      }
    }
     */


    /**
     * 在 TS中 提供了 一种更灵活的方式对 类中的属性 监控
     *
     *
    // get 属性名 () {}   ->
    // 然后实例中 使用  p.属性名 访问， 相当于 调用了TS定义的 get 属性名这个方法
     */

    // 读取属性方法
    get name() {
      console.log("get name () 方法被调用了")
      return this._name

      // 然后在 实例中 使用 p.name  调用 get name () 方法
    }

    // 修改属性方法
    set name (value:string) {
      console.log("set name () 方法执行")
      this._name = value;
    }

    // 同理， age也一样
    get age() {
      return this._age
    }
    set age(value:number) {
      if (value >= 0) {
        this._age = value
      }
    }
  }

  /**
   * 现在属性是在对象中设置，属性可以任意的被修改
   *    - 属性可以任被修改将会导致对象中的数据 变得 非常不安全
   */
  const p = new Person("Yellowsea", 123);
  console.log(p)

```

```ts
//  添加 private 后 实例对象 就不能通过 .属性名 修改属性值了
// p._name = 'Hidie'
// p._age = 456


// 实例对象通过 类中的 方法 修改属性值

// console.log(p.getName())  // 通过get 方法获取属性值
// p.setName('Hidie')   // 通过 set 方法修改 属性值

// console.log(p.getAge())
// // p.setAge(-123)  //经过判断， 修改不了
// p.setAge(456)
// console.log(p)



// 实例使用 TS  的 get set 方法访问属性
console.log(p.name);  //  get name () 方法被调用了 -->   Yellowsea
p.name = 'Hidie';  // set name () 方法执行

// 同理 age
console.log(p.age);
p.age = 456;
console.log(p);



// 2. 这里讲 protected
// protected  受包含的属性，只能在当前类 和 当前类的子类 能访问，其他情况下不能访问

class A {
  // num 属性，是一个 protected 限制的
  protected num: number;
  constructor(num: number) {
    this.num = num;
  }
}

class B extends A {
  // B 继承了A
  show() {
    // 子类B 能够访问到 A 中的 num
    console.log(this.num);
  }
}

const b = new B(123);
b.show()  // 123



// 3. 属性的封装简洁写法
class C {
  // 直接在构造函数中 确定的 属性的类型，不用再写this.xxx  = xxx
  constructor(public num: number, public age: number) {}
}

```

```
  const c = new C(123,456);
  console.log(c)
})()


/**
 * Ts 中提供的 getter setter 方法
 * 使用场景：当对某一个属性值 有比较高的 严格需求时
 */
```

接口

```

(function () {

  // 使用接口来描述对象类型
  /**
   * 接口： 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
   *   - 接口可以重复声明
   */

  // 1. 接口用于 类型声明，  - 使用在对象中， 跟 type Mytype = {} 类似
  // 定义接口
  interface myObject {
    name: string;
    age: number;
  }

  // 接口可以重复声明 ， 相当于合并 两个接口
  interface myObject {
    gender: string;
  }

  // 使用接口
  const obj: myObject = {
    name: "yellowsea",
    age: 1,
    gender: "男"
  }

  console.log(obj)



  /**
   * 2. 接口可以在定义类的时候去限制类的结构
   *    - 接口中的所有属性都不能有实际的值
   *    - 接口只定义对象的结构，而不考虑实际的值
```

```
 */

interface myInter {
  name: string;
  sayHello(): void;  // 返回值为 空

  // 接口中的所有属性都不能有实际的值
  // 接口只定义对象的结构，而不考虑实际的值
  // 接口里的 方法 就是 抽象方法
}

// 定义类时， 可以使用类去实现一个接口 ， 必须使用 implements 指定的 接口
// 这个类就是 满足接口的要求
class MyClass implements myInter {
  name: string;
  constructor (name: string) {
    this.name = name;
  }
  sayHello(){
    console.log("Hello")
  }

}


/**
 * 最后讲讲 接口的作用
 *  - 接口实际就是定义了一个规范， 当类满足了规范，才能在特定的场景使用
 *  - 实际上是对类的限制
 */
})()
```

泛型

```
/**
 *
 * 08 泛型
 */

// 问题:
// function fn (a: number):number {
//   return a;
// }
/**
 * fn 函数，确定了 a 的类型时，它的返回值同样也确定了
 *
 * 当 a 的类型不确定时， 有应该怎么 保证 a 的类型 和 函数的返回值呢
 *
 *   - 使用 any ， 但是使用了 any， 在 TS 中就相当于关闭了 变量的类型检查。
 *
 *   - 使用泛型。 当函数中的类型不确定时， 使用泛型
 */
```

```ts
// 使用泛型
function fn<T>(a: T):T {  // 这里的  T  就是泛型,
  return a;
}

// 调用
let result =  fn(10)  // 泛型 T,  自动检查 参数 10 , 然后确定 T 的类型 -> number
// 使用了 TS 中的 自动判断类型


let result2 = fn<string>('hello')  // 手动指定 泛型 T 的类型为 string


// 泛型的好处,  就是确定了 参数类型 的明确
// 在调用时,  不用担心类型的不明确



// 2. 泛型可以指定 多个

function fn2<K, T>(a: T, b: K):T {
  console.log(b)
  return a
}

// 使用时,最好加上类型,  这样更好的避免出错
fn2<string, number>(123,'Hello')



// 3. 指定接口的泛型

interface Inter {
  length: number
}

// T extends Inter   定义的泛型 T 指定 Inter 接口
// T extends Inter 泛型T  必须实现 Inter 这个类
function fn3<T extends Inter>(a: T):number {
  return a.length
}

fn3('Hello')  // str 中有 length 属性
// fn3(123)  // error ,
// fn3({name:'yellowsea'}) // error
fn3({length: 2})  // 指定length的属性



//  4. 在类中使用 泛型 T
class MyClass<T> {
  // 类中使用 泛型 T
  name:T;
  constructor(name:T) {
    this.name = name;
  }
}

```

```
const myc = new MyClass(123)
const myc1 = new MyClass('123')



/**
 * 总结：
 *
 * 泛型就是 在变量不明确时，使用一个变量，用这个变量来表示 泛型
 *
 */
```