# TS简介

## TypeScript是什么？

·以JavaScript为基础构建的语言。

·JavaScript的超集。

·可以在任何支持JS的平台中执行，但不能被JS解析器直接执行。

·TS扩展了JS，并添加了类型。

## TS增加了什么？

·类型

·支持ES新特性

·添加ES不具备的新特性

·丰富的配置选项

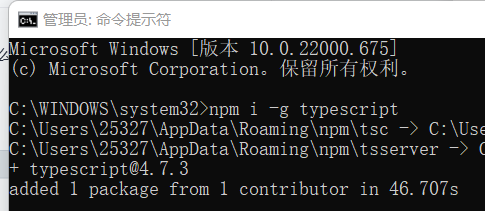
·强大的开发工具

# 开发环境搭建

## 安装nodejs

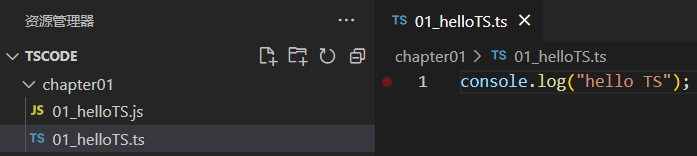
## 全局安装TS

·npm i -g typescript

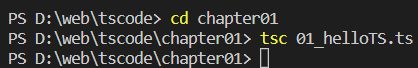


## 使用TS

·编写TS代码



·编译



# TS的类型声明

## 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编译器会自动检查值是否符合类型声明，符合则赋值，否则报错

·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;

- function fn(参数:类型,参数:类型):类型{

}

·02\_basis.ts代码

// 声明一个变量a，同时指定类型为number

let a: number;

// a的类型设置为number，在以后使用过程中a的值只能是number类型

a = 10;

// a = "hello";//报错，会提示错误，但也可以编译成功

let b: string;

b = "hello";

// b = 13;// 报错

// 声明完变量直接进行赋值

let c: boolean = true;

// let d: boolean = 123;//报错

// 当对变量的声明和赋值是同时进行时，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是同时进行时，可以省略类型声明

let d = true;

// 指定形参类型

function sum(a: number,b: number){

    return a + b;

}

sum(123,456);

// sum(123,"456");// 检查类型，报错

// sum(123,456，789);// 检查个数，报错

// 还可以指定返回值类型

function add(a: number,b: number): number{

    return a + b;

    // return "dasd";//报错

}

### 自动类型判断

·TS拥有自动类型判断机制

·当对变量的声明和赋值是同时进行时，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是同时进行时，可以省略类型声明

# TS中的类型

## 类型



·03\_types.ts

// 也可以直接使用字面量进行类型声明

let a:10;

a = 10;

// a = 11; // 报错,a只能是10

// 可以使用|来连接多个类型（联合类型）

// b只能是male或female

let b:"male" | "female";

// c只能是boolean或string类型

let c: boolean | string;

// 声明任意类型any(显示any)

// 使用TS时，不建议使用any类型

// 声明变量如果不指定类型，默认为any类型（隐式any）

let d:any;

d = 10;

d = "hello";

d = true;

// unknown 声明未知类型

let e: unknown;

e = 10;

e = "hello";

e = true;

// d的类型是any，它可以赋值给任意变量

// 这也是不用any的原因

let s:string;

s = d;// 不报错

// s = e;// 报错

// unknown 实际上就是一个类型安全的any

// unknown类型的变量，不能直接复制给其他变量

// e的类型与s的类型相同，不报错

if(typeof e === "string"){

    s = e;

}

// 或者使用类型断言，可以用来告诉编译器变量的实际类型

// 告诉编译器e就是字符串

s = e as string;

// 或者，与上一句意思一样

s = <string>e;

// void 空

// 返回值为空

function fn() : void{

    return undefined;// 可以不写也不会报错

}

// never 永远不会返回结果

// 没有返回值

// 有种函数不需要返回结果，是用来报错的

function fu1() : never{

    throw new Error("报错了！");

}

// object表示js对象，一般不会使用

let f:object;

f = {};

f = function (){};

// g是对象，且有类型为string的name属性

let g : {name: string};

// g = {};//报错

// g = {name:1};//报错

g = {name:"hello"};//不报错

// 在属性名后面加上?表示可选值

// age可有可无

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

h = {name:"hello"};//不报错

// [propName:string]:any表示任意个数任意类型的属性

let i : {name:string,[propName:string]:any}

i = {name:"sdsa",age:18,sex:"dasdas"};//写多少个属性都行

// [propName:string]:string表示任意个数string类型的属性

let j : {name:string,[propName:string]:string}

j = {name:"sdsa",age:"18",sex:"dasdas"};//写多少个属性都行

/\*

    设置函数结构的类型声明

    语法：(形参:类型,形参:类型)=>返回值

\*/

// k是函数，有两个number类型的参数，返回number类型

let k : (a:number,b:number)=>number;

// 不报错

k = function (n1,n2){

    return n1 + n2

}

// 报错

// k = function (n1,n2,n3){

//     return n1 + n2;

// }

// 报错

// k = function (n1:string,n2){

//     return n1 + n2

// }

/\*

    数组的类型声明

    类型[]

    Array<类型>

\*/

// 声明全是string类型的数组

// string[] 便是字符串数组

let l:string[];

l = ["a","b","c"];//不报错

// l = [1,2,3];//报错

// 或者

let n:Array<string>;

n = ["a","b","c"];//不报错

// n = [1,2,3];//报错

/\*

    元组，固定长度的数组

    [类型,类型]

\*/

// 声明两个值为string类型的数组

let m:[string,string];

m = ["a","b"];//不报错

// m = ["a","b","c"];//报错

/\*

    enum 枚举

\*/

// 定义枚举

enum Gender{

    Male,

    female

}

let o:{name:string,gender:Gender};

o = {

    name:"tom",

    gender:Gender.Male

}

// 判断

console.log(o.gender === Gender.Male);

// &表示同时

let p:{name:string} & {age:number};

p = {

    name:"tom",

    age:18

}//不报错

// p = {

//     name:"tom",

//     age:"18"

// }//报错

// 类型的别名

// 设置1 | 2 | 3 | 4 | 5类型的别名为myType

type myType = 1 | 2 | 3 | 4 | 5;

let q:myType;

let r:myType;

# TS编译选项

## 自动编译文件

·编译文件时，使用-2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。

- tsc xxx.ts -w

## 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命令的前提是，要先在项目根目录下创建一个ts的配置文件tsconfig.json

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

### 配置选项

·include

- 定义希望被编译文件所在的目录

- 默认值：[“\*\*/\*”]

- 示例：

{

    "include":[

        "./src/\*\*/\*"

    ],

}

- 两个\*表示任意目录，一个\*表示任意文件

- 上述示例中，src目录下的文件都会被编译

·exclude

- 定义需要排除在外的目录

- 默认值：[“node\_modules”,”bower\_components”,”jspm\_packages”]

- 示例：

{

    "exclude": [

        "./src/hello/\*\*/\*"

    ]

}

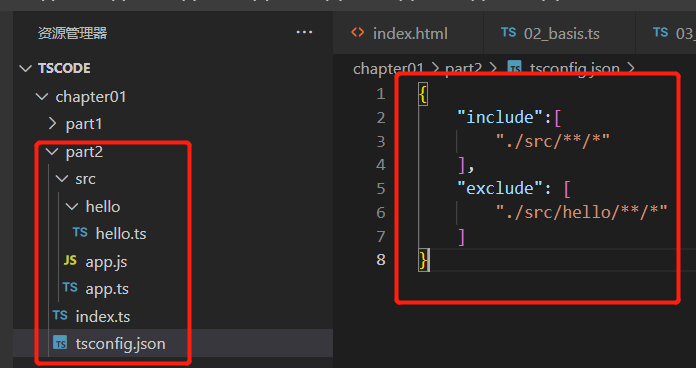
- 上述示例中，src下hello目录下的文件都不会被编译

·extends

- 定义被继承的配置文件

·files

- 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



·compilerOptions

- 编译选项是配置文件中非常重要也比较复杂的配置选项

- 包含多个子选项，用来完成对编译的配置

{

    "include": [

        "./src/\*\*/\*"

    ],

    "compilerOptions": {

        // 用来指定ts被编译为的es的版本

        //'es3', 'es5', 'es6', 'es2015', 'es2016', 'es2017', 'es2018', 'es2019', 'es2020', 'es2021', 'es2022', 'esnext'

        "target": "ES6",

        // 用来指定要使用的模块化的规范

        // 'none', 'commonjs', 'amd', 'system', 'umd', 'es6', 'es2015', 'es2020', 'es2022', 'esnext', 'node16', 'nodenext'

        "module": "commonjs",

        // 用来指定项目中要使用的库

        // 一般不用写

        // 'es5', 'es6', 'es2015', 'es7', 'es2016', 'es2017', 'es2018', 'es2019', 'es2020', 'es2021', 'es2022', 'esnext',

        // 'dom', 'dom.iterable', 'webworker', 'webworker.importscripts', 'webworker.iterable', 'scripthost', 'es2015.core',

        // 'es2015.collection', 'es2015.generator', 'es2015.iterable', 'es2015.promise', 'es2015.proxy', 'es2015.reflect',

        // 'es2015.symbol', 'es2015.symbol.wellknown', 'es2016.array.include', 'es2017.object', 'es2017.sharedmemory',

        // 'es2017.string', 'es2017.intl', 'es2017.typedarrays', 'es2018.asyncgenerator', 'es2018.asynciterable',

        // 'es2018.intl', 'es2018.promise', 'es2018.regexp', 'es2019.array', 'es2019.object', 'es2019.string',

        // 'es2019.symbol', 'es2020.bigint', 'es2020.date', 'es2020.promise', 'es2020.sharedmemory', 'es2020.string',

        // 'es2020.symbol.wellknown', 'es2020.intl', 'es2020.number', 'es2021.promise', 'es2021.string', 'es2021.weakref',

        // 'es2021.intl', 'es2022.array', 'es2022.error', 'es2022.intl', 'es2022.object', 'es2022.string', 'esnext.array',

        // 'esnext.symbol', 'esnext.asynciterable', 'esnext.intl', 'esnext.bigint', 'esnext.string', 'esnext.promise',

        // 'esnext.weakref'

        // "lib": [

        //     "es6",

        //     "dom"

        // ]

        // 用于指定编译后文件的输出目录

        "outDir": "./dist",

        // 将代码合并为一个文件输出

        // 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"

        // 是否对js文件进行编译，默认是false不编译

        "allowJs": true,

        // 是否检查js代码是否符合ts语法规范，默认是false

        "checkJs": true,

        // 是否移除注释

        "removeComments": true,

        // 不生成编译后的文件，只想检查语法的时候使用

        // "noEmit": true,

        // 当有错误的时候，不生成编译文件

        "noEmitOnError": true,

        // 设置编译后的文件是否使用严格模式，默认false

        // 引入的模块会自动进入严格模式，不需要设置

        "alwaysStrict": true,

        // 不允许使用隐式any语法

        "noImplicitAny": true,

        // 不允许不明确类型的this

        "noImplicitThis": true,

        // 严格检查空值

        "strictNullChecks": true,

        // 所有严格检查的总开关,开发时一般都打开

        "strict": false

    }

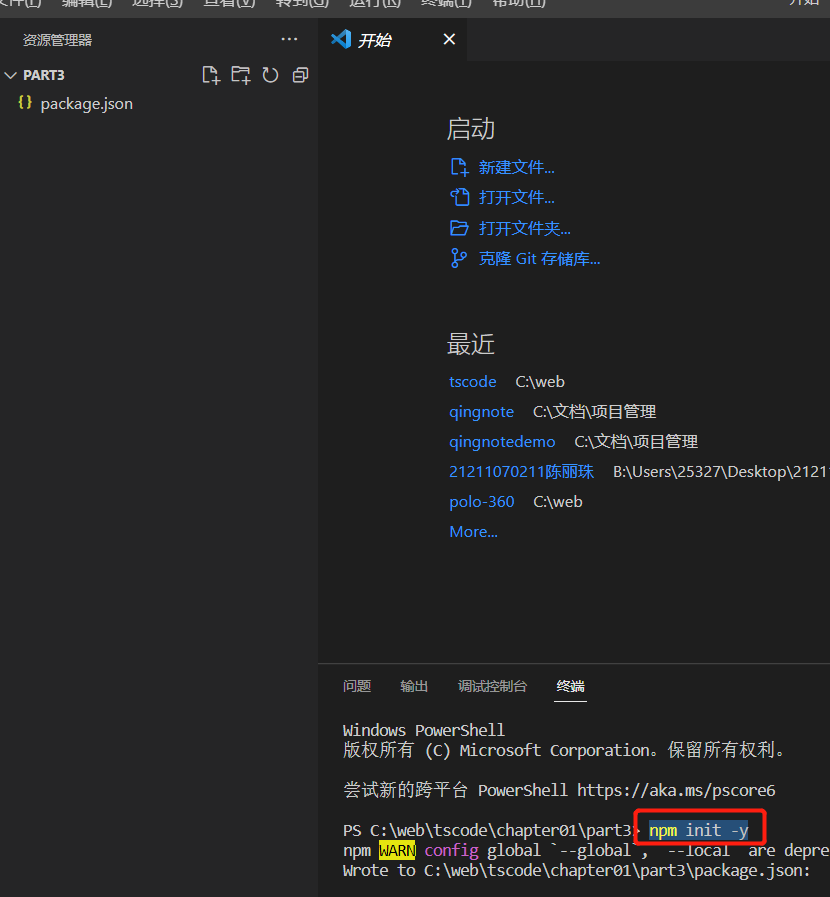
}

# webpack与ts

## 使用webpack打包ts代码

·生成package.json文件

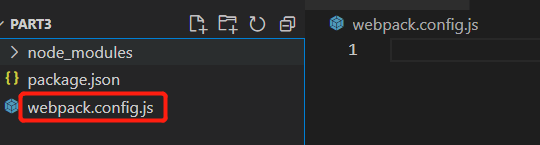
- npm init -y



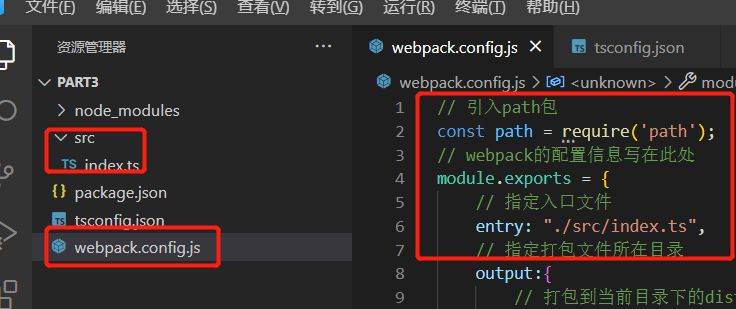
·安装依赖

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

·新建webpack配置文件



·编写webpack配置文件



// 引入path包

const path = require('path');

// webpack的配置信息写在此处

module.exports = {

    // 指定入口文件

    entry: "./src/index.ts",

    // 指定打包文件所在目录

    output:{

        // 打包到当前目录下的dist目录

        path: path.resolve(\_\_dirname,'dist'),

        // 打包后的文件名

        filename: "bundle.js"

    },

    // 指定webpack打包时要使用的模块

    module: {

        // 指定要加载的规则

        rules: [

            {

                // 指定规则生效的文件，匹配以ts结尾的文件

                test: /\.ts$/,

                // 要使用的loader

                use: 'ts-loader',

                // 要排除的文件，排除node-modules目录下的文件

                exclude: /node-modules/

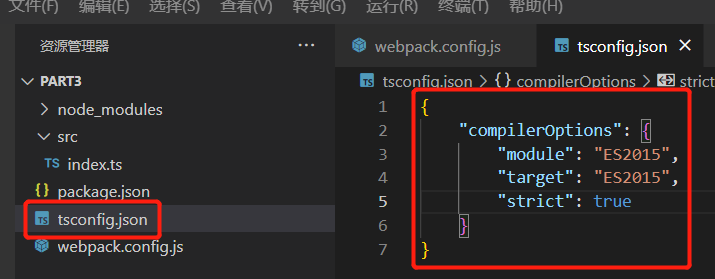
            }

        ]

    }

};

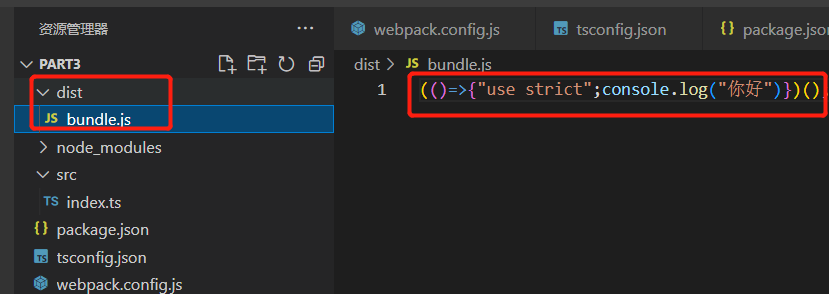
·新建tsconfig.json文件并编写



·修改package.json文件，通过build命令直接执行webpack



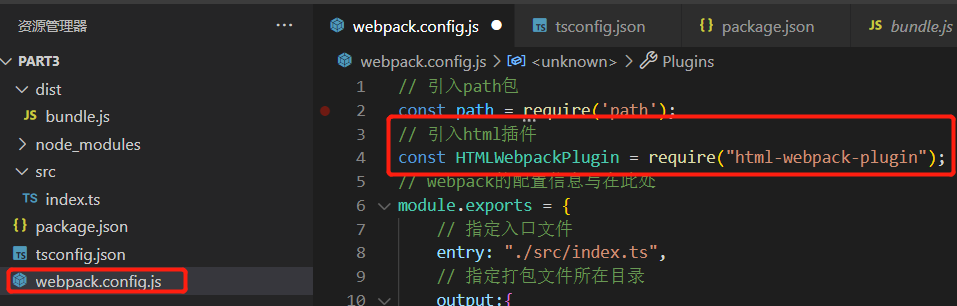
·使用npm run build开始打包



## 自动生成html文件

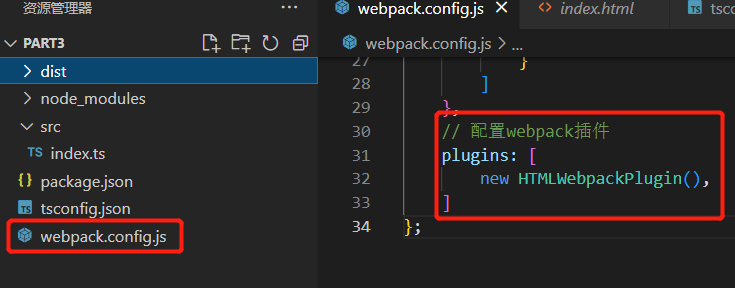
·安装html-webpack-plugin

- cnpm i -D html-webpack-plugin



// 引入html插件

const HTMLWebpackPlugin = require("html-webpack-plugin");

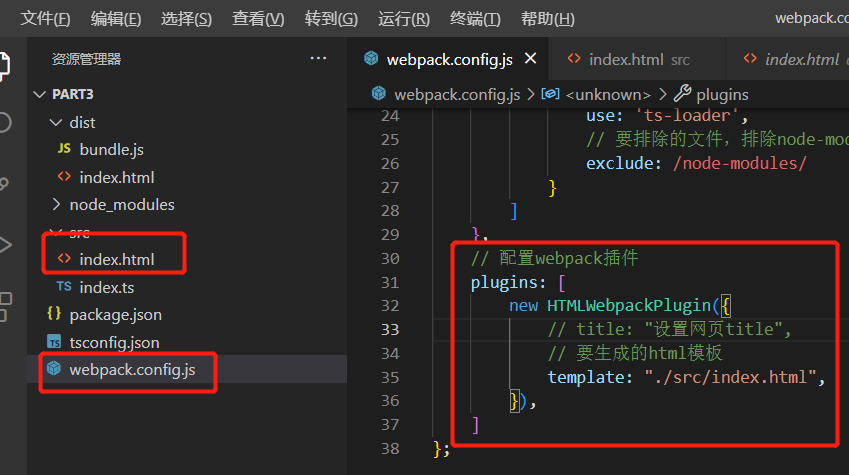


    // 配置webpack插件

    Plugins: [

        new HTMLWebpackPlugin(),

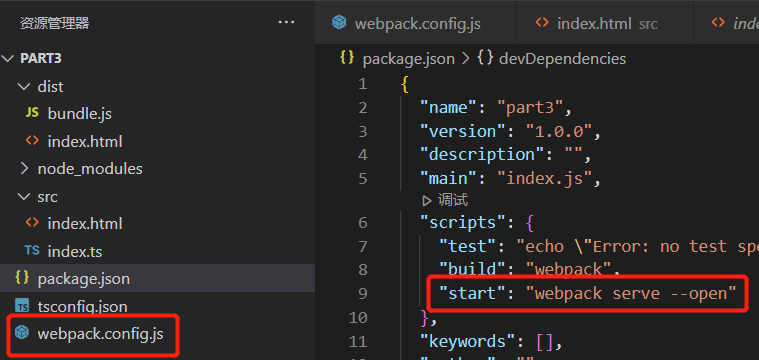
    ]



## 实时更新

·安装webpack-dev-server

- cnpm i -D webpack-dev-server

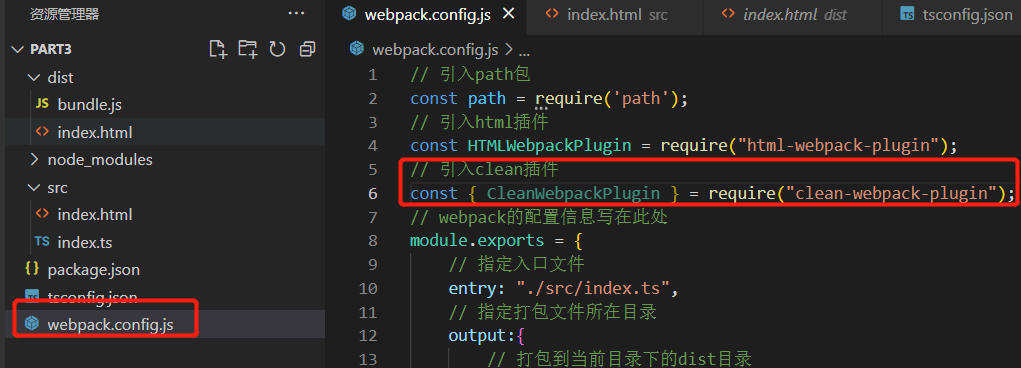


·使用npm start命令启动

## 实时打包

·安装clean-webpack-plugin

- cnpm i -D clean-webpack-plugin



## Babel代码转换

·安装@babel/core @babel/preset-env babel-loader core-js

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

- @babel/core

-- babel核心工具

- @babel/preset-env

-- 预设环境，使代码兼容不同浏览器

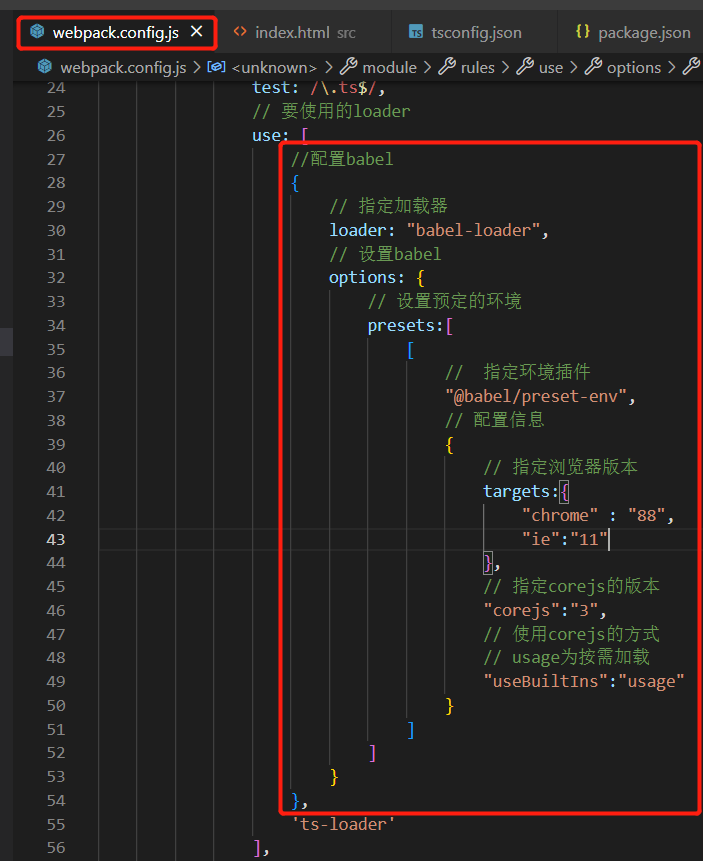
- babel-loader

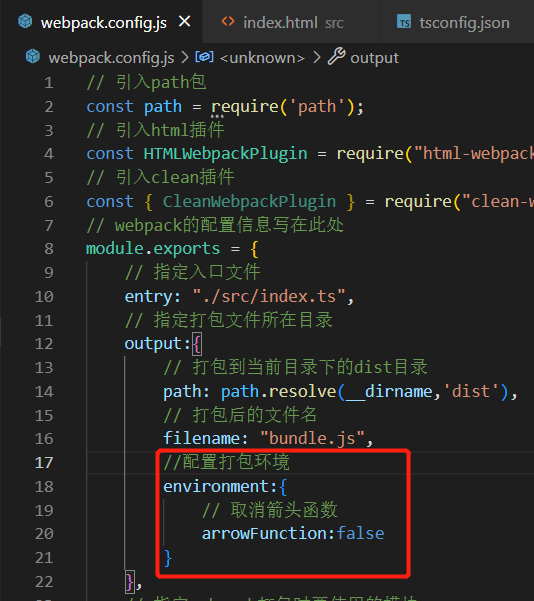
-- 将babel和webpack结合的工具

- core-js

-- 模拟js运行环境的工具

·修改webpack.config.js





## 本节webpack.config.js代码

// 引入path包

const path = require('path');

// 引入html插件

const HTMLWebpackPlugin = require("html-webpack-plugin");

// 引入clean插件

const { CleanWebpackPlugin } = require("clean-webpack-plugin");

// webpack的配置信息写在此处

module.exports = {

    // 指定入口文件

    entry: "./src/index.ts",

    // 指定打包文件所在目录

    output:{

        // 打包到当前目录下的dist目录

        path: path.resolve(\_\_dirname,'dist'),

        // 打包后的文件名

        filename: "bundle.js",

        //配置打包环境

        environment:{

            // 取消箭头函数

            arrowFunction:false

        }

    },

    // 指定webpack打包时要使用的模块

    module: {

        // 指定要加载的规则

        rules: [

            {

                // 指定规则生效的文件，匹配以ts结尾的文件

                test: /\.ts$/,

                // 要使用的loader

                use: [

                    //配置babel

                    {

                        // 指定加载器

                        loader: "babel-loader",

                        // 设置babel

                        options: {

                            // 设置预定的环境

                            presets:[

                                [

                                    //  指定环境插件

                                    "@babel/preset-env",

                                    // 配置信息

                                    {

                                        // 指定浏览器版本

                                        targets:{

                                            "chrome" : "88",

                                            "ie":"11"

                                        },

                                        // 指定corejs的版本

                                        "corejs":"3",

                                        // 使用corejs的方式

                                        // usage为按需加载

                                        "useBuiltIns":"usage"

                                    }

                                ]

                            ]

                        }

                    },

                    'ts-loader'

                ],

                // 要排除的文件，排除node-modules目录下的文件

                exclude: /node-modules/

            }

        ]

    },

    // 配置webpack插件

    plugins: [

        new HTMLWebpackPlugin({

            // title: "设置网页title",

            // 要生成的html模板

            template: "./src/index.html",

        }),

    ]

};

# 面向对象

## 简介

·程序中所有操作都需要通过对象来完成

·对象

- 对现实事物的抽象

- 组成

-- 数据（属性）

-- 功能（方法）

## 类

·类就是对象的模型

- 如通过Person创建人的对象

·要有对象先有类

·使用

- class关键字创建类

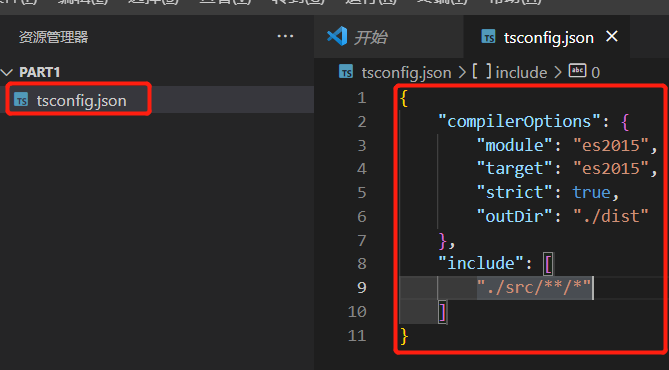
- 实例属性

- 类属性（静态属性）

- 只读属性

- 实例方法

- 类方法（静态方法）



{

    "compilerOptions": {

        "module": "es2015",

        "target": "es2015",

        "strict": true,

        "outDir": "./dist"

    },

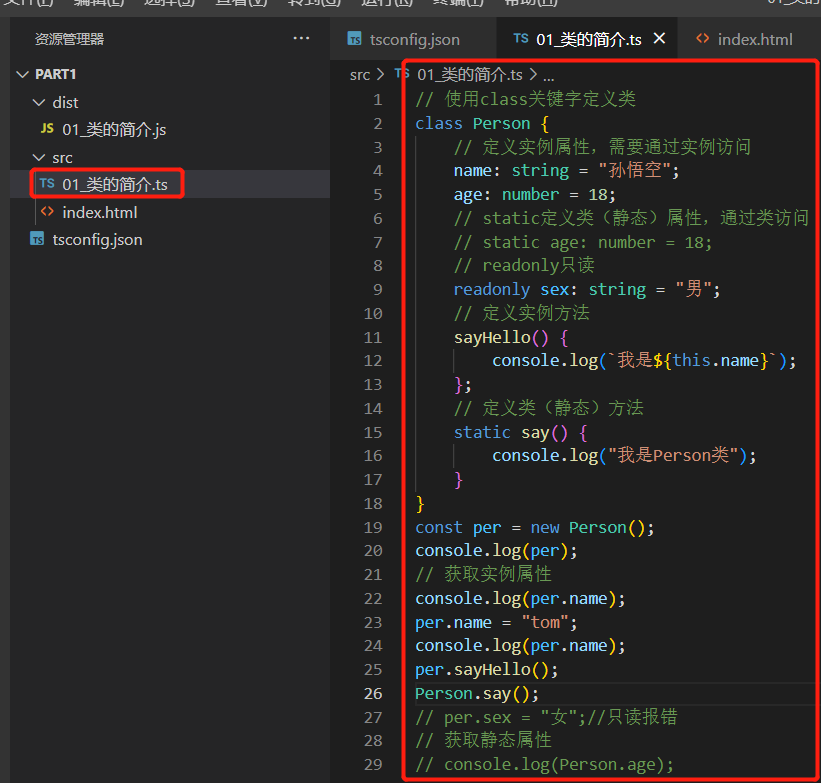
    "include": [

        "./src/\*\*/\*"

    ]

}





// 使用class关键字定义类

class Person {

    // 定义实例属性，需要通过实例访问

    name: string = "孙悟空";

    age: number = 18;

    // static定义类（静态）属性，通过类访问

    // static age: number = 18;

    // readonly只读

    readonly sex: string = "男";

    // 定义实例方法

    sayHello() {

        console.log(`我是${this.name}`);

    };

    // 定义类（静态）方法

    static say() {

        console.log("我是Person类");

    }

}

const per = new Person();

console.log(per);

// 获取实例属性

console.log(per.name);

per.name = "tom";

console.log(per.name);

per.sayHello();

Person.say();

// per.sex = "女";//只读报错

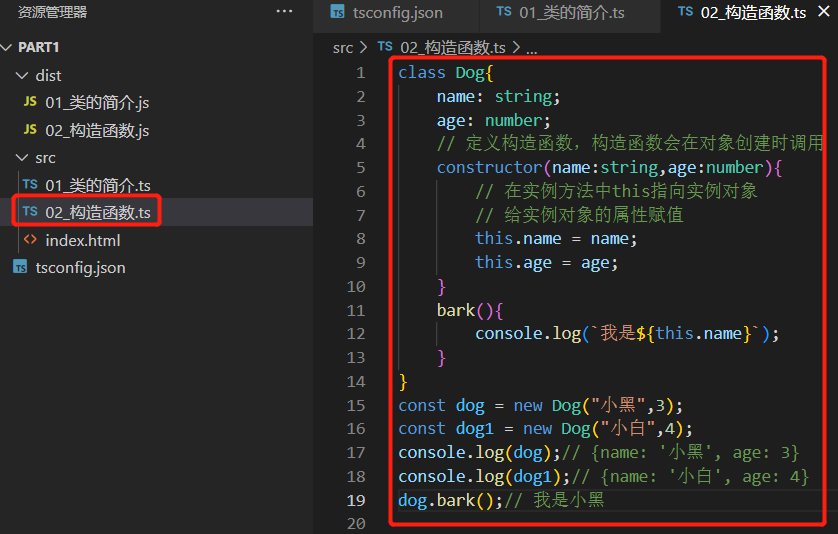
// 获取静态属性

// console.log(Person.age);

## 构造函数

·constructor定义构造函数，构造函数会在对象创建时调用。

·在实例方法中this指向实例对象。



class Dog{

    name: string;

    age: number;

    // 定义构造函数，构造函数会在对象创建时调用

    constructor(name:string,age:number){

        // 在实例方法中this指向实例对象

        // 给实例对象的属性赋值

        this.name = name;

        this.age = age;

    }

    bark(){

        console.log(`我是${this.name}`);

    }

}

const dog = new Dog("小黑",3);

const dog1 = new Dog("小白",4);

console.log(dog);// {name: '小黑', age: 3}

console.log(dog1);// {name: '小白', age: 4}

dog.bark();// 我是小黑

## 继承

·在开发中不要改别人的类！！！

·OCP开闭原则

- 对扩展开发，对修改关闭

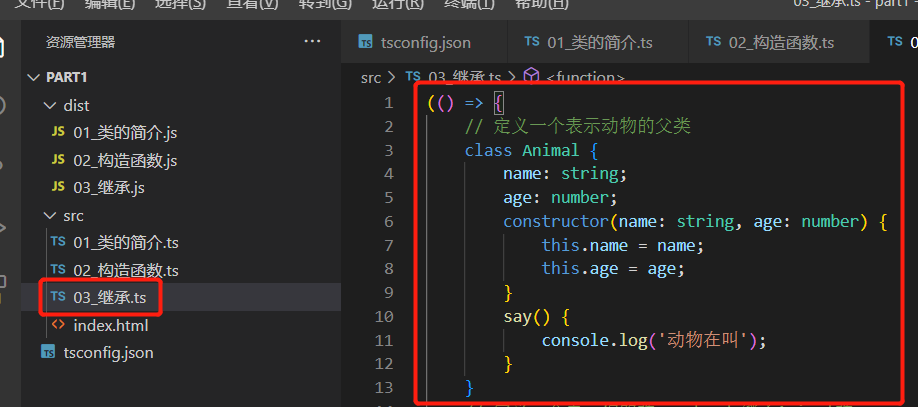
·使用extends关键字继承父类

·继承可以将多个类中共有的代码写在父类

·继承后子类会有父类中的方法和属性

·方法重写

- 在子类中添加与父类相同的方法，覆盖父类方法



(() => {

    // 定义一个表示动物的父类

    class Animal {

        name: string;

        age: number;

        constructor(name: string, age: number) {

            this.name = name;

            this.age = age;

        }

        say() {

            console.log('动物在叫');

        }

    }

    // 定义一个表示狗的类，extends继承Animal类

    // 继承可以将多个类中共有的代码写在父类

    // 继承后子类会有父类中的方法和属性

    class Dog extends Animal {

        // 添加方法

        run() {

            console.log(`${this.name}在跑~`);

        }

        // 如果在子类中添加与父类相同的方法，则会覆盖父类方法

        // 重写父类方法

        say() {

            console.log("汪汪汪！");

        }

    }

    // 定义一个表示狗的类，extends继承Animal类

    // 继承后子类会有父类中的方法和属性

    class Cat extends Animal {

        // 添加方法

        run() {

            console.log(`${this.name}在跑~`);

        }

        // 如果在子类中添加与父类相同的方法，则会覆盖父类方法

        // 重写父类方法

        say() {

            console.log("喵喵喵~");

        }

    }

    const dog = new Dog("旺财", 5);

    const cat = new Cat("臭臭", 3);

    console.log(dog);

    dog.say();

    dog.run();

    console.log(cat);

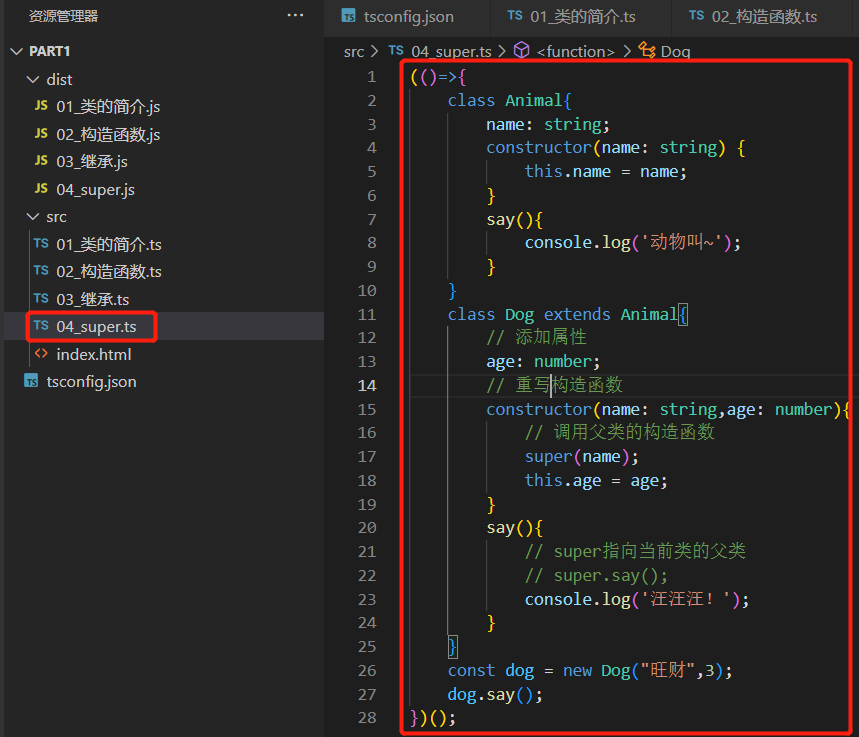
    cat.say();

})();

## super关键字

·super指向当前类的父类

·在子类中重写构造函数时，使用super()调用父类构造函数



(()=>{

    class Animal{

        name: string;

        constructor(name: string) {

            this.name = name;

        }

        say(){

            console.log('动物叫~');

        }

    }

    class Dog extends Animal{

        // 添加属性

        age: number;

        // 重写构造函数

        constructor(name: string,age: number){

            // 调用父类的构造函数

            super(name);

            this.age = age;

        }

        say(){

            // super指向当前类的父类

            // super.say();

            console.log('汪汪汪！');

        }

    }

    const dog = new Dog("旺财",3);

    dog.say();

})();

## 抽象类

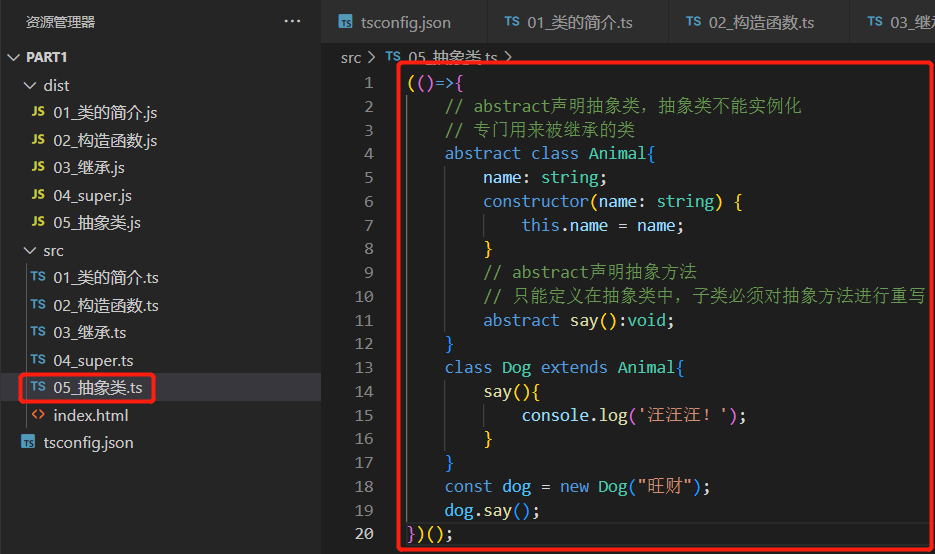
·不能实例化，专门用来被继承的类

·abstract声明抽象类

·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声明抽象方法



(()=>{

    // abstract声明抽象类，抽象类不能实例化

    // 专门用来被继承的类

    abstract class Animal{

        name: string;

        constructor(name: string) {

            this.name = name;

        }

        // abstract声明抽象方法

        // 只能定义在抽象类中，子类必须对抽象方法进行重写

        abstract say():void;

    }

    class Dog extends Animal{

        say(){

            console.log('汪汪汪！');

        }

    }

    const dog = new Dog("旺财");

    dog.say();

})();

## 接口

·目的

- 为了定义规范

·作用

- 定义类或对象的结构

- 定义一个类中包含哪些属性和方法

- 也可以当成类型别名使用，与type类似

·接口可重复声明，实现的类或对象要同时满足同名接口

·接口中所有属性都不能赋值

·接口只定义类的结构，不考虑实际值

·接口中所有的方法都是抽象方法

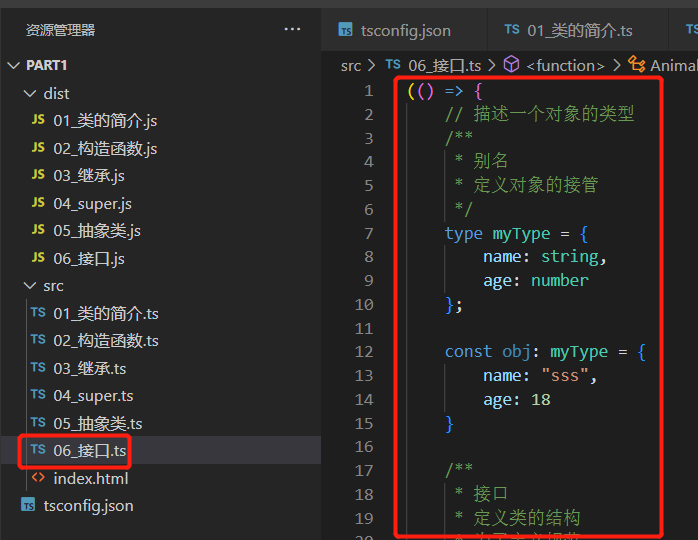
·与抽象类的区别

- 抽象类可以有抽象方法和普通方法

- 接口只能有抽象方法

- 用抽象类时使用extends关键字

- 用接口时使用implements关键字



(() => {

    // 描述一个对象的类型

    /\*\*

     \* 别名

     \* 定义对象的接管

     \*/

    type myType = {

        name: string,

        age: number

    };

    const obj: myType = {

        name: "sss",

        age: 18

    }

    /\*\*

     \* 接口

     \* 定义类的结构

     \* 为了定义规范

     \* 定义一个类中包含哪些属性和方法

     \* 也可以当成类型别名使用，与type类似

     \* 可重复声明

     \*/

    // 当成类型别名使用

    interface myInterface {

        name: string;

        age: number;

    }

    interface myInterface {

        gender: string;

    }

    const obj1: myInterface = {

        name: "sss",

        age: 18,

        gender: "男"

    }

    // 定义类的结构

    // 接口中所有属性都不能赋值

    // 只定义类的结构，不考虑实际值

    // 所有的方法都是抽象方法

    interface IAnimal{

        name: string;

        say():void;

    }

    // 定义类时，可以实现接口

    class Animal implements IAnimal{

        name: string;

        constructor(name:string){

            this.name = name;

        }

        say() {

            console.log('动物叫~');

        }

    }

})();

## 属性的封装

·属性公有化的问题

- 属性可以直接修改会导致对象的数据变得非常不安全

·TS可以在属性前添加属性的读写权限

- public（公有属性）

-- 可以在任意位置读写 (默认值)

- private（私有属性）

-- 只能在类内部读写

-- 通过在类中添加get和set方法使该属性可在外部读写

- protected（保护属性）

-- 只能在当前类或当前类的后代类中读写

·设置get和set的方式

        get name(){

            return this.\_name

        }

        set name(name:string){

            this.\_name = name;

        }

·快速生成get和set的方式

- 输入prop回车

- 输入变量名回车

- 按tap键

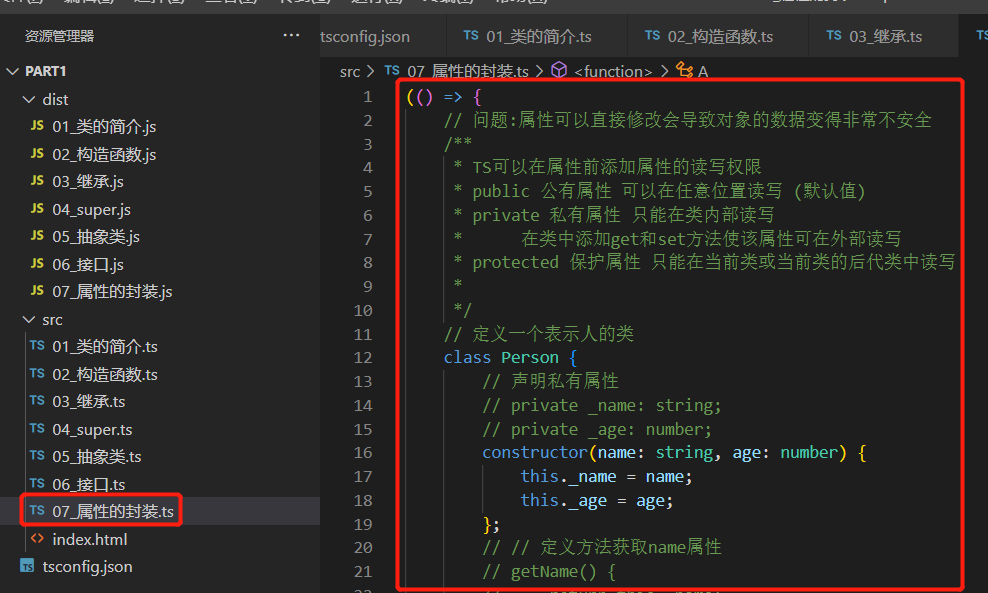
- 输入数据类型

·可以将属性声明在构造函数中

        // 可以将属性声明在构造函数中

        constructor(public name:string){

        }



(() => {

    // 问题:属性可以直接修改会导致对象的数据变得非常不安全

    /\*\*

     \* TS可以在属性前添加属性的读写权限

     \* public 公有属性 可以在任意位置读写 (默认值)

     \* private 私有属性 只能在类内部读写

     \*      在类中添加get和set方法使该属性可在外部读写

     \* protected 保护属性 只能在当前类或当前类的后代类中读写

     \*

     \*/

    // 定义一个表示人的类

    class Person {

        // 声明私有属性

        // private \_name: string;

        // private \_age: number;

        constructor(name: string, age: number) {

            this.\_name = name;

            this.\_age = age;

        };

        // // 定义方法获取name属性

        // getName() {

        //     return this.\_name;

        // }

        // // 定义方法修改name属性

        // setName(name: string) {

        //     this.\_name = name;

        // }

        // // 定义方法获取age属性

        // getAge() {

        //     return this.\_age;

        // }

        // // 定义方法修改age属性

        // setAge(age: number) {

        //     // 判断年龄是否合法

        //     if(age >= 0 && age <= 150){

        //         this.\_age = age;

        //     }else{

        //         throw new Error("年龄不合法");

        //     }

        // }

        // TS中设置get和set的方法

        // get name(){

        //     return this.\_name

        // }

        // set name(name:string){

        //     this.\_name = name;

        // }

        /\*\*

         \* 快速生成get和set的方法:

         \* 输入prop回车

         \* 输入变量名回车

         \* 按tap键

         \* 输入数据类型

         \*/

        private \_name: string;

        public get name(): string {

            return this.\_name;

        }

        public set name(v: string) {

            this.\_name = v;

        }

        private \_age: number;

        public get age(): number {

            return this.\_age;

        }

        public set age(v: number) {

            this.\_age = v;

        }

    }

    const per = new Person("孙悟空", 18);

    // per.setAge(-33);

    // console.log(per);

    // per.name会直接调用get或set方法

    per.name = "猪八戒";

    console.log(per.name);

    class A{

        // 可以将属性声明在构造函数中

        constructor(public name:string){

        }

    }

})()

## 泛型

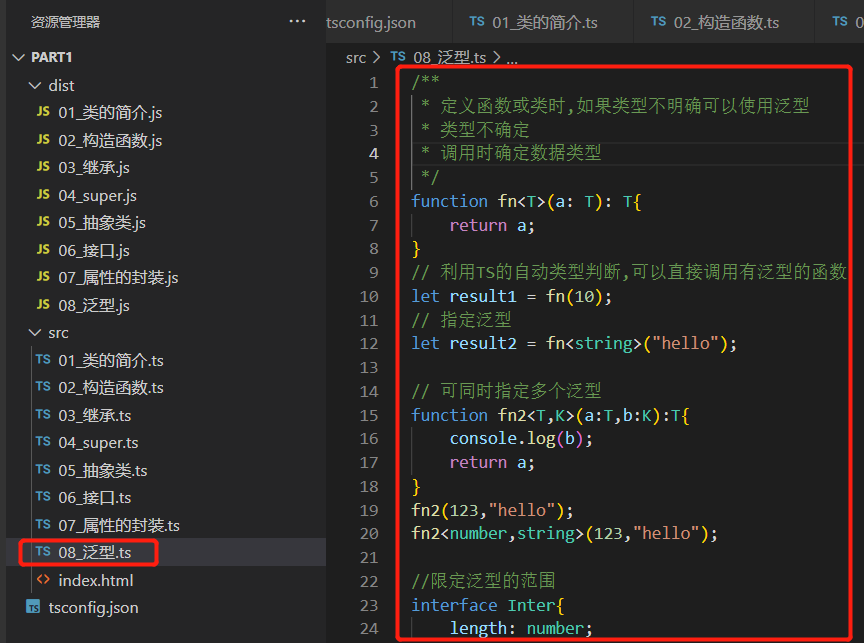
·定义函数或类时,如果类型不明确可以使用泛型

·调用时确定数据类型

·可同时指定多个泛型

·可限定泛型的范围

·类中也可以用泛型



/\*\*

 \* 定义函数或类时,如果类型不明确可以使用泛型

 \* 类型不确定

 \* 调用时确定数据类型

 \*/

function fn<T>(a: T): T{

    return a;

}

// 利用TS的自动类型判断,可以直接调用有泛型的函数

let result1 = fn(10);

// 指定泛型

let result2 = fn<string>("hello");

// 可同时指定多个泛型

function fn2<T,K>(a:T,b:K):T{

    console.log(b);

    return a;

}

fn2(123,"hello");

fn2<number,string>(123,"hello");

//限定泛型的范围

interface Inter{

    length: number;

}

// 指定泛型T必须实现Inter接口

// 以下的a必须有length属性

function fn3<T extends Inter>(a: T): number{

    return a.length;

}

fn3("hello");

// fn3(123);// 报错.无length属性

fn3({name:"hello",length:2});

// 类中也可以用泛型

class MyClass<T>{

    constructor(public name:T){

        this.name = name;

    }

}

const mc = new MyClass<string>("孙悟空");