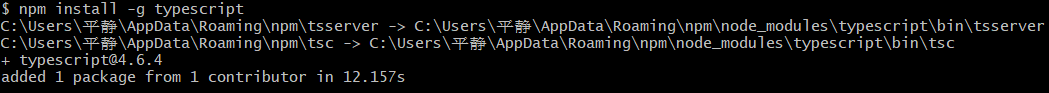
# 1.简介和环境搭建

简介：ts是js的超集

搭建：npm install -g typescript



安装成功：查看tsc -v 查看版本



使用:新建demo1.ts(注意文件后缀)

function hello() {

    let str: String = 'hello world!'

    console.log(str)

}

hello()

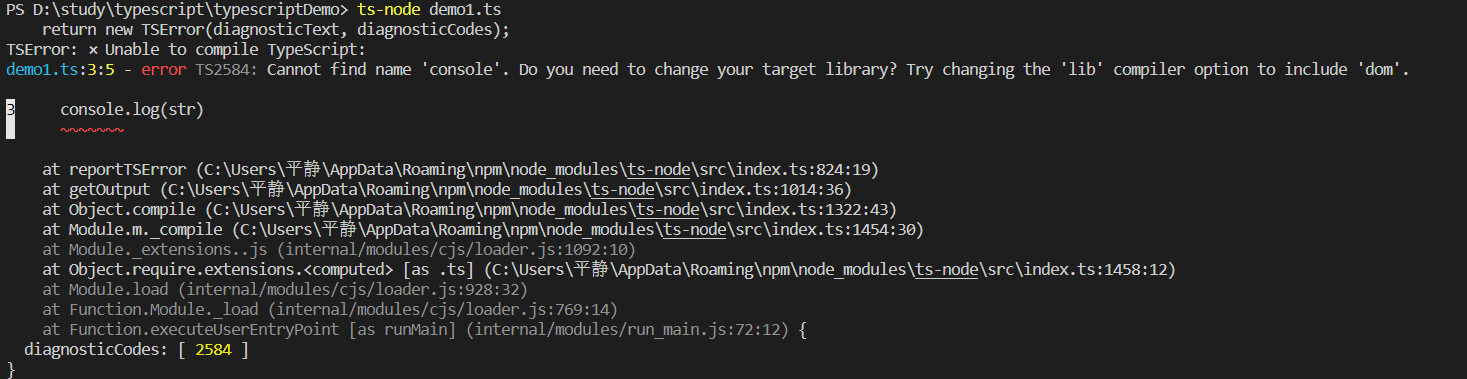
运行：

方式一：转换为js：tsc demo1.ts，会多出来js文件，然后运行node .\demo1.js

方式二：ts-node直接运行，这个比较慢，是因为中间做转换

npm install -g ts-node

备注：运行报错，是因为ts-node版本太高，换个低版本的就行



使⽤ npm uni -g ts-node 卸载掉ts-node；再使⽤ npm i -g ts-node@8.5.4



## 2.静态类型

1）定义变量时定义变量的类型：一方面维护更方便，另一方面编辑器能联想对应的方法更友好。

let age: number = 18

console.log(age.toFixed(2))

interface Person {

    name: String,

    age: number,

    sex: string

}

const wenda: Person = {

    name:'wenda',

    age:30,

    sex:'女'

}

console.log(wenda)

基础静态类型

string number boolean null undefined void symbol

const str: string = 'wenda'

const num: number = 18

const isFlag: boolean = true

// x可能是数字或null或undefined

let x: number | null | undefined;

x = 1;  // 运行正确

x = undefined;    // 运行正确

x = null;    // 运行正确

// void 用于方法无返回值

function hello(): void {

    console.log('hello')

}

对象静态类型

普通对象，数组，类，函数

// 基础对象

const Dog: {

    name: string,

    age: number

} = {

    name: 'test',

    age: 1

}

// 数组

const arr: string[] = ['a', 'b', 'c']

// 类

class Person { }

const wenda: Person = new Person()

// 函数

const say: () => string = () => { return '1' }

# 4.类型注解和类型推断

// 类型注解

let a: number

a = 123

// 类型推断

let b = 123

# 5.函数参数和返回值的注解

// 函数参数1.基础数据类型

function func1(a: number) {

    return a

}

// 函数参数2.对象

function func2({ a }: { a: number }) {

    return a

}

// 函数传参3.对象数组

function func3(a: { a: number; b: string; }[]): { a: number, b: string }[] {

    return a

}

// 函数返回值1.基础数据类型

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

    return a + b

}

// 函数返回值2.无返回值

function sayHello(): void {

    console.log(1);

}

// 函数返回值3.never 函数永远执行不完

function neverFunc1(): never {

    while (true) {

        console.log(1)

    }

    console.log(2)

}

function neverFunc2(): never {

    throw new Error()

    console.log(2)

}

# 6.ts中数组的注解

let arr1: number[] = [1, 2, 3]

let arr2: string[] = ['1', '2']

let arr3: undefined[] = [undefined, undefined]

let arr4: {

    name: string;

    value: number;

}[] = [{ name: 'a', value: 1 }]

type ObjType = { name: string, value: number }

let arr5: ObjType[] = [{ name: 'a', value: 1 }]

class ObjClass { name: string; value: number; }

let arr6: ObjClass[] = [{ name: 'a', value: 1 }]

# 7.ts中的元组

// 数组 存在问题，程序中使用index获取对应的名称，职业和年龄，数组中的值稍微调换一下顺序就错了，但编码无任何错误提示

let arr: (string | number)[] = ['wenda', 'front-end-develop', 30]

// 元组 更严谨，但使用场景不多

let yuanzu: [string, string, number] = ['wenda', 'front-end-develop', 30]

// CSV 会用到元组，数组中每个元素为元组

let csv:[string, string, number][] = [

    ['wenda', 'front-end-develop', 30],

    ['wenda2', 'front-end-develop', 20]

]

# 8.interface

重复的类型注解，可以定义成统一的接口

interface User {

    name: string;

    job: string;

    age: number;

    aihao?: string

}

// function getInfo(obj: { name: string, job: string, age: number, aihao?: string }): void {

function getInfo(obj: User): void {

    console.log(`姓名：${obj.name}`);

    console.log(`职业：${obj.job}`);

    console.log(`年龄：${obj.age}`);

    obj.aihao && console.log(`爱好：${obj.aihao}`);

}

// function test(obj: { name: string, job: string, age: number, aihao?: string }): void {

function test(obj: User): void {

    if (obj.age < 30) {

        console.log('通过')

        return

    }

    console.log('未通过')

}

接口和type的区别，接口不能是基础数据类型，type可以

aihao?: string; // 不必须的属性

[propname: string]: any; //任意属性

接口对类的约束: implements

interface User {

    name: string;

    job: string;

    age: number;

    aihao?: string; // 不必须的属性

    [propname: string]: any; //任意属性

    say():string; //方法

}

class Vip implements User {

    name = 'wenda';

    job = '程序员';

    age = 30;

    say(){

        return '说话'

    }

}

接口继承：extends

interface User {

    name: string;

    job: string;

    age: number;

    aihao?: string; // 不必须的属性

    [propname: string]: any; //任意属性

    say():string; //方法

}

interface SupperUser extends User {

    write(): string

}

# 9类

## 类的继承，重写，supper

// 类

class Parent {

    name='名称';

    sayHello(){

        return '说话'

    }

}

// 子类

class Child extends Parent {

    // 重写

    sayHello(){

        // return '子类说话'

        // super关键字

        return '子类' + super.sayHello()

    }

    witerInfo(){

        return '子类写字'

    }

}

const p1 = new Parent()

console.log(p1.sayHello())

const c1 = new Child()

console.log(c1.sayHello())

console.log(c1.witerInfo())

## public private protected

public 公有：内部外部均可使用

pravite 私有：只有自己可用

protected 保护：自己和子类可用

class ParentObj {

    // public 公有：内部外部均可使用

    public name:string

    // pravite 私有：只有自己可用

    private age:number

    // protected 保护：自己和子类可用

    protected job:string

    constructor(name:string,age:number,job:string){

        this.name = name;

        this.age = age;

        this.job = job;

    }

    sayName():void{

        console.log(this.name,'你好！')

    }

    sayAge():void{

        console.log(this.name, this.age,'岁了！')

    }

    sayJob():void{

        console.log(this.name, '工作：', this.job)

    }

}

const pobj1 = new ParentObj('Wenda', 30, '搬砖')

pobj1.sayName()

console.log(pobj1.name) // 只有public可用

class ChildObj extends ParentObj{

    writeName():void{

        console.log('写', this.name,'你好！')

    }

    writeJob():void{

        console.log(this.name, '目前的工作：', this.job)

    }

}

const cobj1 = new ChildObj('Tuanzi', 2 , '长大')

cobj1.sayName()

cobj1.writeName()

cobj1.writeJob()

## 构造函数 constructor, super

class Person {

    // constructor 构造函数，未写默认构造函数为空constructor() { }

    constructor(public name: string) { }

}

const person = new Person('Wenda')

console.log(person.name)

class Teacher extends Person {

    constructor(name: string, public age: number) {

        // 只要写构造函数，就一定要写super,为空也要写

        super(name)

    }

}

const teacher = new Teacher('King', 30)

get set static readonly

get set 可以针对private的属性进行封装属性值的获取和设置

static标记的属性和方法不需要实例化，通过类名访问

readably 只读属性

// get set 可以针对private的属性进行封装属性值的获取和设置

// static标记的属性和方法不需要实例化，通过类名可以访问

class PersonTest{

    constructor(public name:string,private \_age:number){}

    get age():number{

        return this.\_age -10 // 获取就能自定义化处理

    }

    set age(age:number){

        this.\_age = age + 10 // 设置就能自定义化处理

    }

}

const person1 = new PersonTest('Wenda',30)

console.log(person1.name)

console.log(person1.age)

person1.age = 20;

console.log(person1.age)

class People{

    constructor(public name:string){}

    sayName():void{

      console.log(this.name)

    }

    // static 所以直接访问

    static sayHello():void{

        console.log('你好')

    }

}

const people1 = new People('Wenda')

people1.sayName()

People.sayHello();

## 抽象类 abstract

抽象类中必有抽象方法，继承抽象类的子类必要实现父类的抽象方法

abstract class Test1{

    abstract sayHello()

}

class Test1Child extends Test1{

    sayHello() {

        console.log('子类1打招呼')

    }

}

class Test2Child extends Test1{

    sayHello() {

        console.log('子类2打招呼')

    }

}

# 10.tsconfig.json

tsconfig.json

1.typeScript的编译配置文件：tsc --init命令生成的,

1）此时直接使用tsc demo1.ts,生成了demo1.js ，但是编译配置文件并没有起作用，直接使用tsc 才可以，

2）ts-node此时也是使用的编译配置文件

2. removeComments 编译的时候忽略注释

3.include exclude files

// "include": ["demo1.ts"], // 编译这些文件

  // "files": ["demo1.ts"], // 编译这些文件 只能是文件，不能是文件夹，不常用

  "exclude": ["demo1.ts"], // 编译忽略这些文件

4.strict开启后下面紧跟部分自动开启，表示严格遵循ts规范，只有关闭时下面的配置项才能配置，使用时建议开启，更规范，避免问题

// 1.noImplicitAny 注释any一定要写

function test(param: any){

   console.log(param)

}

//2.strictNullChecks string不能初始化为null

let str:string = ''

5.rootDir:要编译的根目录；outDir:编译后存放的目录

"rootDirs": ["./src"],

"outDir": "./build",

6. sourceMap编译时生成map文件，ts文件和编译后生成的js文件的映射关系

7. noUnusedLocals 若有未使用的局部变量则抛错。

noUnusedParameters 若有未使用的参数则抛错

# 11.联合类型和类型保护

所谓联合类型，可以认为一个变量可能有两种或两种以上的类型

Type1|Type2|Type3

类型保护：

1）断言 as

2）in

3)typeof

4)instanceof

# 12.枚举类型

用于定义数值集合

enum Color { Red, Green, Blue }

let color: Color = Color.Red

console.log(color)

console.log(Color[0]) // 反查

{ '0': 'Red', '1': 'Green', '2': 'Blue', Red: 0, Green: 1, Blue: 2 } // 实际打印结果，故访问如此

# 13.泛型

## 函数中泛型的使用

1）命名：T可以起任意名称，具体传值类型由调用的地方指定

2）使用：<>,且可以使用多个泛型

3）泛型的类型推断（不推荐）

4）泛型中数组的使用 T[] Array<T>

// 泛型

// T可以起任意名称，具体传值类型由调用的地方指定

function add<T>(a: T, b: T) {

    return `${a} ${b}`

}

// 可以使用多个泛型

function add2<T, P>(a: T, b: P) {

    return `${a} ${b}`

}

console.log(add2<string, number>('1', 2))

// 泛型的类型推断（不推荐）

console.log(add2('1', 2))

// 泛型中数组的使用 T[] Array<T>

function join<T>(params:T[]){

    return params.join(';')

}

function join2<T>(params:Array<T>){

    return params.join(';')

}

console.log(join<string>(['1', '2']))

console.log(join2<string>(['1', '2']))

## 类中泛型的使用

1）类中泛型的基本使用<T>

2）类中泛型的约束<T extends string | number>

3) 类中泛型的继承，<T extends Person>Person是一个接口

// 1）类中泛型的基本使用<T>

class Test18\_2<T> {

    constructor(private days: T[]) { }

    selectOneDay(day: number): T {

        return this.days[day]

    }

}

let test18\_2 = new Test18\_2<string>(['5.1', '5.2', '5.3'])

console.log(test18\_2.selectOneDay(1))

// 2）类中泛型的约束<T extends string | number>

class Test18\_3<T extends string|number> {

    constructor(private days: T[]) { }

    selectOneDay(day: number): T {

        return this.days[day]

    }

}

let test18\_3 = new Test18\_3<string>(['5.1', '5.2', '5.3'])

let test18\_32 = new Test18\_3<number>([123, 234, 345])

console.log(test18\_3.selectOneDay(1))

console.log(test18\_32.selectOneDay(1))

// 3) 类中泛型的继承，<T extends Person>Person是一个接口

interface Person18 {

    name: string;

}

class Test18\_4<T extends Person18> {

    constructor(private days: T[]) { }

    selectOneDay(day: number): T {

        return this.days[day]

    }

}

let test18\_4 = new Test18\_4([{ name: '5.1' }, { name: '5.2' }, { name: '5.3' }])

console.log(test18\_4.selectOneDay(1))

# 命名空间

1）用途：防止变量污染

2）使用：namespace Home{}

需要暴漏出去的使用export,其他的都是私有的

namespace Home{

class Header{

    constructor(){

        let ele = document.createElement('div')

        ele.innerText = '我是页面头部'

        document.body.appendChild(ele)

    }

}

class Content{

    constructor(){

        let ele = document.createElement('div')

        ele.innerText = '我是页面内容'

        document.body.appendChild(ele)

    }

}

class Footer{

    constructor(){

        let ele = document.createElement('div')

        ele.innerText = '我是页面底部'

        document.body.appendChild(ele)

    }

}

export class Page{

    constructor(){

        new Header()

        new Content()

        new Footer()

    }

}

}

用命名空间实现组件化

将组件部分提出去放在components.ts中

namespace Components {

    export class Header {

        constructor() {

            let ele = document.createElement('div')

            ele.innerText = '我是页面头部'

            document.body.appendChild(ele)

        }

    }

    export class Content {

        constructor() {

            let ele = document.createElement('div')

            ele.innerText = '我是页面内容'

            document.body.appendChild(ele)

        }

    }

    export class Footer {

        constructor() {

            let ele = document.createElement('div')

            ele.innerText = '我是页面底部'

            document.body.appendChild(ele)

        }

    }

}

添加引入

<!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">

    <script src="./build/components.js"></script>

    <script src="./build/page.js"></script>

    <title>Document</title>

</head>

<body>

    <script>new Home.Page()</script>

</body>

</html>

多文件编译成一个文件，tsconfig中的配置项修改为

"outFile": "./build/page.js",

"module": "amd",

子命名空间：命名空间的嵌套

namespace Components {

    export namespace ChildComponent{

        export class Test {

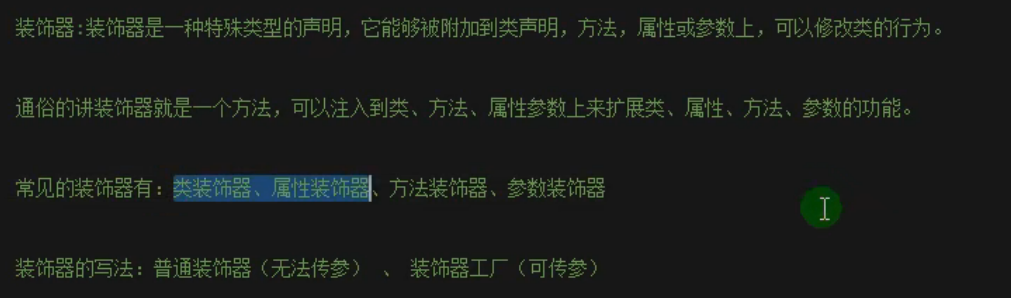
        }

}

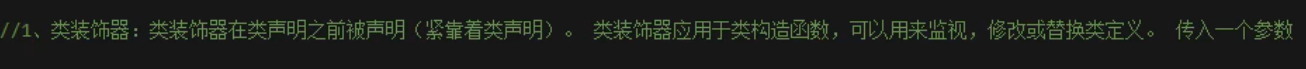
}

new Components.ChildComponent.Test()

# 装饰器



1）类装饰器



装饰器是个方法，可以被附加到类的声明，方法，属性或参数上，可以修改，扩展类的行为

类型：类~，属性~，方法~，参数~

写法：普通装饰器（无传参），装饰器工厂（有传参）

// 无参数的装饰器 params传入的是调用的当前类

function logClass1(params:any){

    console.log(params)

    // 装饰器扩展类的属性

    params.prototype.url = 'https://a.b.com'

    // 装饰器扩展类的方法

    params.prototype.runlog = function(){

        console.log('记录日志')

    }

}

@logClass1

class HttpClient{

    constructor(){

    }

    getData(){

    }

}

let httpClient:any = new HttpClient()

console.log(httpClient.url)

httpClient.runlog()

// 有参数的装饰器,闭包中传入的是类，装饰器函数传入的是装饰器调用的直接传参

function logClass2(params:string){

    return function(target:any){

        console.log(params)

        console.log(target)

        // 扩展类的属性

        target.prototype.url = params

    }

}

@logClass2('http://wenda.com')

class HttpClient2{

    constructor(){}

    getData(){

    }

}

let httpClient2:any = new HttpClient2()

console.log(httpClient2.url)

// 装饰器中构造函数的重载

function logClass3(target:any){

   return class extends target{

       url:string = 'http://logclass3'

       getData(){

          console.log(`装饰器：${this.url}`)

       }

   }

}

@logClass3

class HttpClient3{

    public url:string|undefined

    constructor(){

        this.url = 'http://class3'

    }

    getData(){

        console.log(this.url)

    }

}

let httpClient3:any = new HttpClient3()

httpClient3.getData()

2）属性装饰器

运行时调用

传参有两个，一个是类的原型对象，一个是属性名称

// 属性装饰器

function logProperty(params:any){

    console.log(params)

    // target是类的原型对象 attr是属性名

    return function(target:any,attr:any){

        console.log(target)

        console.log(attr)

        target[attr] = params

    }

}

// 无参数的装饰器 params传入的是调用的当前类

function logClass4(params:any){

    console.log(params)

    // 装饰器扩展类的属性

    params.prototype.url = 'https://a.b.com'

    // 装饰器扩展类的方法

    params.prototype.runlog = function(){

        console.log('记录日志')

    }

}

@logClass4

class HttpClient4{

    @logProperty('1')

    public data:any

    constructor(){

    }

    getData(){

        console.log(`属性装饰器给的默认值：${this.data}`)

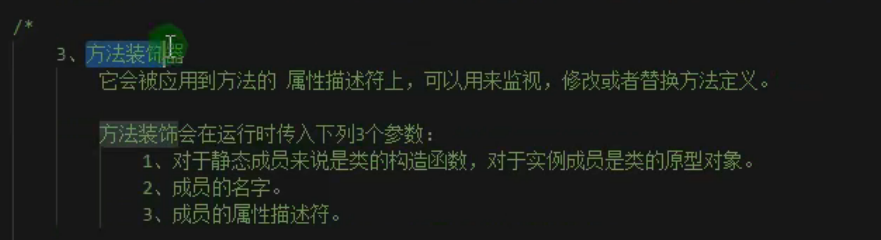
    }

}

let httpClient4:any = new HttpClient4()

httpClient4.getData()

3）方法装饰器



4）方法参数装饰器

传参有三，1.原型对象，2.方法名，3.参数索引

5）装饰器的执行顺序

属性>方法>方法参数>类

同级别的有多个则从后往前







