VSCODE自动编译TS

1. 安装vscode+typescript
2. Vscode -> 终端->配置任务->tsc: 监视（F1->tasks configure -> tsc: 监视） - > 运行任务

类型

- 布尔类型（boolean）

```
let bool:boolean = true
```

- 数字类型（number）

```
let num:number = 1
```

- 字符串类型(string)

```
let str:string= '123'
```

- 数组类型（array）

```
let arr:number[] = [1,2,3]
// 泛型
let arr:Array<number> = [1,2,3]
```

- 元组类型（tuple）

```
// 属于数组的一种 （可以指定数组中的多种类型）
let arr:[string,number,boolean] = ['a',1,true]
```

- 枚举类型（enum）

```
// 定义标识符 flag   1 ture  0 false
let enum flag = {false=0, true=1}
let f:flag = flag.false // 0

let enum color = {red, blue, orange}
// 默认不复赋值,   取索引
console.llog(color.blue) // 1

let enum color = {red, blue=5, orange}
// 默认取上一个 索引
console.llog(color.orange) // 6
```

- 任意类型（any）

```
let num:any;
num = 1
num = '2'
```

- null 和 undefined
  - 其他数据类型(never 类型)的子类型

```
let num:undefined;
console.log(num) // undefined

let num:number;
console.log(num) // 异常

var num:number|undefined
console.log(num) // undefined
num = 3
console.log(num) // 3
```

```
let num:null
num=null
num = 3 // 异常
```

- void类型

```
// 表示没有任何类型，一般用于定义方法没有返回值
function func():void{}
```

- never类型

```
// 是其他类型（包含null和undefined）的子类型，代表从不会出现的值
// 声明的 never 类型只能被never类型赋值
let func:never
func = (()=>{
  throw new Error('error')
})()
```

函数

```
// 函数声明
function add(x:number, y: number):number{
  return x+y
}
// 函数表达式
let func = function(x:number, y: number):number{
    return x+y
}
// 所有定义的参数都必须传
func(1) // 编译错误

// 可选参数 y?: number
// 必须配置到参数列表的最后
let func = function(x:number, y?: number):number{
  if(y) return x+y
   return x
}
func(1) // 正常通过
// 默认参数 y: number=20
let func = function(x:number, y: number=2):number{
    return x+y
}
func(1) // 3

// 剩余参数
let func = function (...y: number[]): number {
// 本案例：需要考虑未传入
    return y.reduce((a, b) => {
        return a + b
    })
```

```
}
func(1,2,3,4) // 10
```

函数的重载

重载是指：同一个函数名，参数不同，这种时候 会出现重载

```
function func(x:number):number;
function func(x:number,y:string):string;
function func(x:any,y?:any):any{
    if(y){
        return x+y
    }
    return x
}
console.log(func(1)); // 1
console.log(func(1,'2')); // 12
console.log(func(1,2)) // 异常
```

类

(es5) 基础类

```
//es5
function Person(){
  this.name = 'hello'
}
let p = new Person()
p.name // hello
```

(es5) 构造函数和原型链

```
// 构造函数 ， 不会被共享
function Person(){
  this.name = 'hello'
  this.say = function(){ // 实例方法
```

```
    console.log(this.name)
  }
}
// 原型链 属性会被多个属性共享
Person.prototype.age = 12
let p = new Person()
p.say()
Person.getInfo = function(){} //  静态方法
Person.getInfo()
```

```
// ts 中类的实现
class Person{
    name:string;
    constructor(name:string){
        this.name = name
    }
    getName():string{
        return this.name
    }
    setName(name:string):Person{
        this.name = name
        return this
    }
}
let p = new Person('张三')
p.getName() // 张三
p.setName('李四')
p.getName()  // 李四
```

ts中类的继承 extends

```
// 继承父类  extends
class Man extends Person{
  // 必须执行构造函数
    constructor(name:string){
        super(name)
    }
}
let m = new Man('男人')
console.log(m.getName());
```

类的修饰符

属性默认不加修饰符，默认是public

- public // 当前类、子类、类外部可访问
- protected // 当前类、子类可访问 `类外部不能访问`
- private // 在当前类里可访问，子类、外部都不可以访问

```
class Person{
   public name:string;
   protected sex:string;
   private age:number
    constructor(name:string,sex:string, age:number){
        this.name = name
        this.sex = sex
        this.age = age
    }
    getName():string{
        return this.name
    }
    getSex():string{
        return this.sex
    }
    getAge():number{
        return this.age
    }
}

let p = new Person('张三','男',20)
console.log(p.getName()); // 张三
console.log(p.getSex()); // 男
console.log(p.getAge()); // 20
// 外部访问
console.log(p.name); // 张三
// console.log(p.age); // 编译不通过
// console.log(p.sex); // 编译不通过

class Man extends Person{
    constructor(name:string,sex:string, age:number){
        super(name,sex,age)
    }
    showName():string{
        return this.name
    }
    // 子类访问受保护
    showSex():string{
    // 受保护，  可以访问
        return this.sex
    }
    // 子类访问私有
    // showAge():number{
    //     // 编译错误
    //     return this.age
    // }
}
```

类的静态方法/属性

```
 class Person{
```

```
    public name:string;
    static sex = 10
    constructor(name:string){
        this.name = name
    }
    // 静态方法 没法直接调用类里的 属性
    // static print(){
    //       //error TS2339
    //     console.log(this.name);
    // }
    static showSex(){
        console.log(Person.sex); // 10
        console.log(this.sex); // 10
    }
}
console.log(Person.sex); // 10
Person.showSex()
```

多态

多态：父类定义一个方法不去实现，让继承他的子类趋势线，每一个子类有不同的表现

多态属于继承，是继承的一种表现

```
class Animal {
    name: string;
    constructor(name: string) {
        this.name = name
    }
    eat() {
        console.log(`动物：${this.name}`);
    }
}

class Dog extends Animal {
    constructor(name: string) {
        super(name)
    }
  // 实现父类方法
    eat() {
        console.log(`${this.name}吃肉`);
    }
}

class Sheep extends Animal {
    constructor(name: string) {
        super(name)
    }
    eat() {
        console.log(`${this.name}吃草`);
    }
```

```
}

let dog = new Dog('小狗狗')
dog.eat()
let sheep = new Sheep('小羊')
sheep.eat()
```

abstract

abstract 抽象类：提供其他类继承的基类，不能实例化；定义抽象类和抽象方法，不包含具体实现并且 `必须在派生类中实现`

抽象方法只能放在抽象类中

```
abstract class Animal {
    name:string;
    constructor(name:string) {
        this.name = name
    }
    abstract eat():void; // 抽象方法必须实现
    run():void{
       console.log('run');
    }
}
// let a = new Animal() // 编译不通过

// 子类必须实现抽象类的所有方法
class Dog extends Animal {
    constructor(name:string) {
        super(name);
    }
    eat(){
        console.log(`${this.name}吃肉`);
    }
}
let dog = new Dog('小狗')
dog.eat()
```

接口

接口的作用：在面向对象的编程中，接口是一种规范的定义，它定义了行为和动作的规范，在程序设计里面，接口起到一种限制和规范的作用。接口定义了某一批类所需要遵守的规范，接口不关心这些类的内部状态数据，也不关心这些类里方法的实现细节，它只规定这批类里必须提供某些方法，提供这些方法的类就可以满足实际需要。 typescrip中的接口类似于java，同时还增加了更灵活的接口类型，包括属性、函数、可索引和类等。

定义标准。

属性类接口

```
interface Label{
    first: string;
    last :string
}
function printLabel (label: Label): void{
    console.log(label.first + ' ' + label.last );
}
// 传入的结构必须满足interface
printLabel({
    first:'姓',
    last:'名'
}) // 姓 名
let l: Label = {
    first: '第一',
    last:'第二'
}
printLabel(l) // 第一 第二
```

可选属性

```
interface Label {
  first: string;
  last?: string; // 属性的可选
}
function printLabel(label: Label): void {
  if (label.last) {
    console.log(label.first + " " + label.last);
  } else {
    console.log(label.first);
  }
}
printLabel({
  first: "姓",
  last: "名",
}); // 姓名
printLabel({
  first: "姓"
}); // 姓
```

函数类型接口

```
interface encrypt{
    (key:string, value:string):string
}
var md5: encrypt = function (key: string, value: string): string{
    return key + value
}
console.log(md5('a', 'b'));  //ab
```

可索引接口

数组或者对象的约束

```
// 数组
interface UserArr{
    [index:number]:string
}
// let arr: UserArr = ['a', 'b']
// let arr: UserArr = ['a', 2] //  编译错误

// 对象
interface UserObj {
    [index: string]: string;
}
let obj: UserObj = {
    name: '你好',
}
```

类类型接口

对类的约束

```
interface Animal{
    name: string,
    eat(str:string):void
}

// 实现这个接口
class Dog implements Animal {
    name: string;
    constructor(name: string) {
```

```
        this.name = name
    }
    // 必须实现 eat
    // 可以不满足 eat 参数
    eat (): void{
        console.log(this.name);
    }
}

let dog = new Dog('小狗')
dog.eat() // 小狗
```

接口扩展

```
interface Animal {
    eat (): void;
}
interface Person extends Animal{
    work():void
}
class Programmer {
    name: string;
    constructor(name:string) {
       this.name = name
    }
    coding (code: string) {
        console.log(`${this.name} == ${code}`);

    }
}
class Man extends Programmer implements Person{
    constructor(name:string) {
      super(name)
    }
    // 必须同时实现 接口定义
    work () {
        console.log('程序员');
    }
    eat () {
        console.log('吃饭');
    }
}
let m = new Man('程序员')
m.coding('java') // 程序员 == java
```

泛型

泛型的定义

泛型：软件工程中，我们不仅要创建一致的定义良好的API，同时也要考虑可重用性。 组件不仅能够支持当前的数据类型，同时也能支持未来的数据类型，这在创建大型系统时为你提供了十分灵活的功能。

在像C#和Java这样的语言中，可以使用泛型来创建可重用的组件，一个组件可以支持多种类型的数据。 这样用户就可以以自己的数据类型来使用组件。

通俗理解：泛型就是解决 类 接口 方法的复用性、以及对不特定数据类型的支持(类型校验)

泛型函数

```
// 泛型支持 不特定的数据类型
// 传入的参数和返回的类型一致
// T 表示泛型 具体什么类型调用这个方法的时候决定
function func<T>(value:T):T {
    return value
}
console.log(func<string>('123')); // '123'
console.log(func<number>(123));
```

泛型类

```
class MinClass<T>{
    list: T[] = []
    add(value: T) {
        this.list.push(value)
    }
    min(): T {
        let minNum = this.list[0]
        for (let index = 1; index < this.list.length; index++) {
            if (this.list[index] < minNum) {
                minNum = this.list[index]
            }
        }
        return minNum
    }
}

let m1 = new MinClass<number>()

m1.add(1)
m1.add(5)
```

```
m1.add(3)
console.log(m1.min());

let m2 = new MinClass<string>()
m2.add('b')
m2.add('a')
console.log(m2.min());
```

```
// 泛型类 约束
class User{
    username:string| undefined;
    pasword:string | undefined;
    constructor(username:string,pasword:string){
        this.username = username
        this.pasword = pasword
    }
}

class MysqlDb<T>{
    add(value:T):boolean{
        console.log(value);
        return true
    }
}

let u = new User('张', '123')
let db = new MysqlDb<User>() // 传入数据 验证类型
db.add(u)
```

泛型接口

```
interface Config{
    <T>(value:T):T
}
var getData:Config = function<T>(value:T):T{
    return value
}
//function getData<T>(value:T):T{
//  return value
//}
getData<string>("a")
getData<number>(1)
```

模块

模块的的概念（官方）：

关于术语的一点说明: 请务必注意一点，TypeScript 1.5里术语名已经发生了变化。"内部模块"现在称做"命名空间"。"外部模块"现在则简称为"模块" 模块在其自身的作用域里执行，而不是在全局作用域里；这意味着定义在一个模块里的变量，函数，类等等在模块外部是不可见的，除非你明确地使用export形式之一导出它们。 相反，如果想使用其它模块导出的变量，函数，类，接口等的时候，你必须要导入它们，可以使用 import形式之一。

模块的概念（自己理解）：

- 我们可以把一些公共的功能单独抽离成一个文件作为一个模块。
- 模块里面的变量 函数 类等默认是私有的，如果我们要在外部访问模块里面的数据（变量、函数、类），我们需要通过export暴露模块里面的数据（变量、函数、类...）。
- 暴露后我们通过 import 引入模块就可以使用模块里面暴露的数据（变量、函数、类...）。

```
// db.ts
let dbUrl = 'xxx'

export  function getUrl(value:string):string{
    return value + dbUrl
}

// index.ts
import { getUrl }  from 'db.ts'
 console.log(getUrl('www.baidu.com'));
```

命名空间

命名空间:

在代码量较大的情况下，为了避免各种变量命名相冲突，可将相似功能的函数、类、接口等放置到命名空间内同Java的包、.Net的命名空间一样，TypeScript的命名空间可以将代码包裹起来，只对外暴露需要在外部访问的对象。命名空间内的对象通过export关键字对外暴露。

命名空间和模块的区别：
命名空间：内部模块，主要用于组织代码，避免命名冲突。
模　块：ts的外部模块的简称，侧重代码的复用，一个模块里可能会有多个命名空间。

```
namespace A{
    export function add(a:number,b:number):number{
        return a+b
    }
}
let a1 = A.add(1,2)
console.log(a1);
```

外部引入

```
// add.ts
export namespace A{
    export function add(a:number,b:number):number{
        console.log('A 空间');
        return a+b
    }
}

export namespace B{
    export function add(a:number,b:number):number{
        console.log('B 空间');
        return a+b
    }
}
// index.ts
import { A, B } from "./add"
A.add(1,2) // A 空间
B.add(3,4) // B 空间
```

装饰器

装饰器:装饰器是一种特殊类型的声明，它能够被附加到类声明，方法，属性或参数上，可以修改类的行为。通俗的讲装饰器就是一个方法，可以注入到类、方法、属性参数上来扩展类、属性、方法、参数的功能。

1. 常见的装饰器有：类装饰器、属性装饰器、方法装饰器、参数装饰器
2. 装饰器的写法：普通装饰器（无法传参）　、装饰器工厂（可传参）
3. 装饰器是过去几年中js最大的成就之一，已是Es7的标准特性之一

普通装饰器

```
//普通装饰器
// params 当前类
function log(params:any){
  console.log(params);   // [Function: htpClient]
  parmas.prototype.url = 'xxx'
  parmas.prototype.run = function(){
    console.log('run');
  }
}
@log
class htpClient{
  constructor(){

  }
}
let http:any = new htpClient()
console.log(http.url); // xxx
http.run() // 'run'
```

装饰器工厂

```
// params 参数
function log(params:any){
  // target 当前类
  return function(target:any){
     console.log(params);   // hello world
     console.log(target);   // [Function: htpClient]
    target.prototype.url = params
  }
}
@log('http://baidu.com')
class htpClient{
  constructor(){}
}
let http:any = new htpClient()
console.log(http.url); // http://baidu.com
```

类装饰器

重载类的的构造函数

- 类装饰器表达式会在运行时当作函数被调用，类的构造函数作为其唯一的参数
- 如果类装饰器返回一个值，它会使用提供的构造函数来替换类的声明。

```
// 装饰器 函数
function log(parmas:any){
  console.log(params); // [Function: HttpClient]
```

```
  return class extends params{
    url:any = '装饰器的URl'
    showUrl(){
      this.url = this.url + '---'
      console.log(this.url);
    }
  }
}
@log
class HttpClient {
  url:string|undefined
  constructor() {
    this.url = '构造函数的URL'
  }
  showUrl():void{
    console.log(this.url);

  }
}
let http = new HttpClient()
http.showUrl() //  装饰器的URl---
```

属性装饰器

属性装饰器表达式会在运行时当作函数被调用，传入下列2个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。

```
// 类装饰器
function log(parmas:any){
   return function(target:any){
     console.log(parmas);    // xxx
     console.log(target);    // [Function: HttpClient]
  }
}
// 属性装饰器
function logProperty(params:any) {
// target 实例成员的 原型对象
// attr 成员名称
  return function(target:any, attr:any){
    console.log(params);    // www.baidu.com
    console.log(target); //HttpClient { showUrl: [Function] }
    console.log(attr);    // url
    target[attr] = params
 }
}
@log('xxx')
class HttpClient {
  @logProperty('www.baidu.com')
  url:string|undefined
```

```
  constructor() {}
  showUrl():void{
    console.log(this.url); // www.baidu.com
  }
}
let http = new HttpClient()
http.showUrl()
```

方法装饰器

动态修改当前方法

```
function log(params:any) {
  console.log(params); // xxxx
  // target 当前实例（静态方法会是当前类）
  // method 当前方法
  // desc 成员属性描述符
  return function (target:any,method:any, desc:any) {
     console.log(target); // HttpClient { getData: [Function] }
     console.log(method); // getData
     console.log(desc); // value 当前方法
    /*
    {
      value: [Function],
      writable: true,
      enumerable: true,
      configurable: true
    }
    */
    let oldMethod = desc.value // 保存当前方法
    // 覆写方法
    desc.value = function (...args: any[]) {
      console.log(args); // ['b']
      oldMethod.apply(this, args)

    }
  }
}
class HttpClient{
  url: any | undefined
  constructor() {

  }
  @log('xxxx')
  getData (str:string) {
    console.log(`getData --- ${str}`); // getData --- ddd
  }
}
let h = new HttpClient()
h.getData('b')
```

方法参数装饰器

```
function log (params: any) {
  console.log(params); // uuid
   // target 当前实例（静态方法会是当前类）
  // method 当前方法名称
  // index 参数索引
  return function (target:any,method:any, index:any) {
    console.log(target); // HttpClient { getData: [Function] }
    console.log(method); // getData
    console.log(index); // 0
  }
}
class HttpClient{
  url: any | undefined
  constructor() {}
  getData (@log('uuid') uuid: any) {
    console.log(uuid);    // 123
  }
}
let h = new HttpClient()
h.getData(123)
```

执行顺序

属性-> 方法-> 方法参数 -> 类

多个从后向前

```
function logClass1(params?:string){
    return function(target:any){
      console.log('类装饰器1')
    }
}

function logClass2(params?:string){
    return function(target:any){
      console.log('类装饰器2')
    }
}

function logAttribute1(params?:string){
    return function(target:any,attrName:any){
      console.log('属性装饰器1')
    }
}
function logAttribute2(params?:string){
    return function(target:any,attrName:any){
      console.log('属性装饰器2')
    }
```

```
}
function logMethod1(params?:string){
    return function(target:any,attrName:any,desc:any){
      console.log('方法装饰器1')
    }
}
function logMethod2(params?:string){
    return function(target:any,attrName:any,desc:any){
      console.log('方法装饰器2')
    }
}
function logParams1(params?:string){
    return function(target:any,attrName:any,desc:any){
      console.log('方法参数装饰器1')
    }
}
function logParams2(params?:string){
    return function(target:any,attrName:any,desc:any){
      console.log('方法参数装饰器2')
    }
}

 // 从后向前
@logClass1()
@logClass2()
class HttpClient{
   // 从后向前
    @logAttribute1()
    @logAttribute2()
    public apiUrl:string | undefined;
    constructor(){
    }
   // 从后向前
    @logMethod1()
    @logMethod2()
    getData(){
        return true;
    }
   // 从后向前
    setData(@logParams1() attr1:any,@logParams2() attr2:any,){}
}
var http:any=new HttpClient();
//属性装饰器2
//属性装饰器1
//方法装饰器2
//方法装饰器1
//方法参数装饰器2
//方法参数装饰器1
//类装饰器2
//类装饰器1
```