TypeScript

**一.介绍**

1.TypeScript是由**微软**开发的开源，跨平台的编程语言，简称TS

2.TypeScript是Javascript的超集，是在JavaScript基础上进行的功能性扩展，语法更严格，更简洁

3.TypeScript无法直接运行，需要转换为JavaScript才能运行

4.TypeScript代码是定义在以.ts结尾的文件中，最终需要编译成为.js结尾的文件

**二.优点**

1.提供类型系统：增强了代码的可读性和可维护性，在编译阶段就能发现大部分错误

2.**支持ES6：ES6规范是客户端脚本语言发展的方向**

3.强大的IDE支持：类型检测，语法提示

**三.配置环境**

TS是基于JS的，所有JS的代码都能在TS文件里写

但需要编译器，所以需要安装全局的编译器

安装node会自带npm

终端输入命令：sudo npm install typescript -g

**四.TS编译成JS**

当前TS文件下终端输入**tsc xxx.ts**

当前目录会自动生成一个同名的JS文件

**自动监视编译**

基于TS项目!

根目录下会存在一个tsconfig.json文件

执行**tsc --init** 让TS项目初始化，就会自动生成tsconfig.json文件

自动监视编译 tsc -w -p tsconfig.json

**变量**

JS中定义变量：var/let/const变量名=变量值;

TS中定义变量：var/let/const变量名:数据类型=变量值;

增加了数据类型的限制

如果直接赋值没有说明变量类型，那么默认加上了第一次赋的值的类型作为该变量的变量类型。

弱类型语言：在定义变量时无需指定数据类型，且变量的类型可以修改，如JS PHP

强类型语言：在定义变量时需要指定数据类型，且变量的类型不能修改，如TS JAVA

**数据类型**

JS中的类型：**string boolean number Array null undefined** Object Function

TS中新增的：**元组 enum枚举 any任何类型 void never**

**let a:string = ‘helllo’;**

**let a:string[ ] = [ ]字符串数组定义**

**let a:Array<number> = [ ]数组泛型定义**

**null和undefined是其他类型的子类型，可以将他们赋值给其他类型（需要关闭严格类型）**

**1**.元组：特殊的数组，一般数组中的元素是动态的，元组可以限制元素数目，每个元素的类型。

**let tuple:[string,number,boolean] = [ ]**表示该数组里只能有3个元素，第一个为字符串类型，第二个为数值型，第三个为布尔型。

**2**.enum枚举，用来限制可取值的范围

enum Season {

spring = ‘春’，

summer = ‘夏’,

autumn = ‘秋’,

winter = ‘冬’

}

let s:Season = Season.summer;

**3**.any任意类型，当暂时不确定变量的类型

**let a:any;**

**4**.void空类型 取值只能为null或undefined，定义函数如果没返回值，声明返回值为void

**function show( ):void{ }**

**5**.never永不为真的类型

函数内部是死循环可以定义函数返回值为never

**6**.object 非原始类型 （**string boolean number symbol null undefined**）除外都是非原始类型

**高级类型Union Types**

联合类型规定某几种类型之一

**let e:string|number = 2;**可以是字符串也可以是数值

**类型的断言**

做一个假设，使编译通过，两种写法

let f:string|number;

如果想获取变量f的长度

那么必须断言

**1**.使用尖括号，语法：<类型>值

if（（<string>f）.length）{

console.log((<string>f).length);

}

**2**.使用as

if（（f as string）.length）{

console.log((f as string).length);

}

**函数**

**TS函数**

**function 函数名(参数名：数据类型，参数名？：数据类型，参数名：数据类型=默认值)：返回值类型 { }**

加？表示可传可不传，而且一定要把这个参数放在最后面，排后面的参数省略，前面就不能

传undefined就是不传，但占实参位子

函数定义类型有

function f1(a,b){return a+b}

var f2=function (a,b) {return a+b}

var f3=(a,b) => a+b;

**剩余参数，可以传入任意个参数**

**function f5(a:string,...b：number[ ]){ }**

**b实际是一个数组**

**定义一个值为函数类型,一个参数是string类型，一个是number，返回值是boolean**

**先定义类型：**

let f8 : (a:string,b:number) => boolean;

**然后赋值：**

f8 = function （a:string,b:number）：boolean{return true }

**对象（类）**

class 类名{

属性

name : string;

age : number;

方法

show(num:number):boolean{ }

构造函数

constructor(name:string){

this.name = name;

}

}

**调用**

let xxx = new 类名( );

let xxx = new 类名(“yyy”);自动调用构造函数

**继承**

class 子类 extends 父类{ }

父类

class Animal {

name:string;

sex:string;

constructor(name:string,sex:string){

this.name = name;

this.sex = sex;

}

cry(){console.log(“///////”)}

}

子类

class Dog extends Animal{

hobby:string;

constructor(name:string,sex:string,hobby:string){

**super(name,sex);**位于this之前

**this.hobby=hobby;**

}

cry(){console.log(“????”)}方法重写

}

var dog = new Dog(“///”,“///”,“///”);

dog.name;

dog.sex;

dog.hobby;

dog.cry();

**修饰符**

用来控制属性和方法的访问范围

1.public（公开）可以在任何地方访问（默认）

2.protected（受保护）可以在当前类和子类中访问，在类的外部无法访问

3.private（私有）只能在当前类中访问

4.static（静态）不属于任何

class Person{

public name : string = “tom”;

protected age : number = 12;

private hobby : string = “game”;

}

**封装（属性私有化）**

**1**.将属性私有化

使用private修饰属性，命名上一般以下划线 \_开头

**2**.提供对外访问的方法，用于对属性进行取值和赋值

取值：get新属性名（）{控制私有化属性的取值}

赋值：set新属性名（新值）{控制私有化属性的赋值}

**3**.访问新属性，实际上就是在对私有属性进行操作

取值：对象名.新属性名

赋值：对象名.新属性名=新值

注：本质上是通过JS中的Object.defineProperty进行数据劫持

class Person{

private \_hobby : string = “game”;

get hobby( ){

return this.\_hobby

}

set hobby(value){

this.\_hobby = value

}

}

let p =new Person( );

p.hobby;

p.hobby = ///;

**抽象类（不能被实体化，只用于被继承）**

使用abstract关键字

1.被abstract修饰的类，就是**抽象类**

定义方法：abstract class 类名{ }

不能使用new去创建对象，只能被继承

含有抽象方法的类必须要是抽象类

2.被abstract修饰的方法，就是**抽象方法**

定义方法：abstract 方法名（）：返回值类型；

抽象方法只有声明，没有具体实现，没有方法体，以分号结尾

抽象类不一定有抽象方法

子类必须重写

**abstract class Pet**{

name:string;

constructor(name:string){

this.name = name;

}

show():void{

consloe.log(/////);

}

**abstract cry( ):string;**抽象方法

}

class Dog extends Pet{

constructor(name:string){

super(name);

}

cry( ){ /////}

}

**多态（表现出多种形态的能力的特征）一种事物多个特性**

将父类的引用指向子类对象

将父级作为方法形参，将子类的对象作为方法实参，从而实现多态

abstract class Person{

name:string;

show():void{

console.log(‘我是一个人’);

}

}

class Teacher extends Person{

school:string;

show():void{

cosole.log(“我叫”+this.name+“我是一个老师”);

}

teach():void{

console.log(“我正在”+this.school+“进行教学”);

}

}

var person:Person = new Teacher( );将父类的引用指向子类的实例

p.name=“tom”;

p.show( );调用子类重写后的方法

不能访问子类特有的，因为类型是父类类型

**1.**使用类型**断言的方式**进行类型的制定

(<Teacher>.p).school = “////”;

(<Teacher>.p).teach( );

**2.**先做类型判断再调用

if(p instanceof Teacher){

p.school = “////”;

p.teach( );

}

**用途:**将父类作为方法的形参，将子类的对象作为方法实参，从而实现多态

function getPerson(p:Person){

p.show( );

if(p instanceof Teacher){

p.teach( );

}

}

let teacher = new Teacher( );

teacher.name = “///”;

teacher.school = “////”;

getPerson(teacher);

**简化属性定义**

**一般来说写法**

class Student{

id:string;

name:string;

costructor(id:string,name:string){

this.id=id;

this.name=name;

}

}

let stu = new Student(////////);

**简写（通过构造函数和属性修饰符）**

class Student{

costructor(public id:string,

public name:string){ }

}

let stu = new Student(////////);

上下相等

**接口**

**定义**

是一种**规范约束**，起到限制和规范的作用，可能强制一个类必须符合某个规范，即实现接口

**定义接口**

interface 接口名{

声明属性（只申明不赋值）;

声明方法（只申明不定义方法）

}

**实现接口**

class 类名 implements 接口名{

必须实现接口中所有的方法和属性

}

interface USB{

weight:number;

power( ):void;

}

class Mouse implements USB{

weight = 100;

power( ):void{

console.log(/////);

}

}

let usb:USB = new Mouse( );

**TS中的接口还可以对函数，对象，数组等进行约束**

使用接口表示函数类型，即通过接口对函数进行约束

interface myFuction{

(a:string):boolean

}

let f1:myFunction;

使用接口表示数组类型，即通过接口对数组进行约束

interface myArray{

[index:number]:string 索引是数值，值为string

}

let a1:myArray;

使用接口表示对象类型，即通过接口对对象进行约束

interface myObject{

name:string;必选属性

age?:number; 可选属性

}

function show (obj:myObject a){///}

**泛型**

**通用的类型**

**泛型类**

class Student<T>{

field:T;

}

let stu =new Student<string>( );一旦确定类型就不能再修改了

stu.field = “///”；

泛型接口

interface USB<T>{

power(args:T):boolean

}

泛型函数

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

return args;

}

**模块**

**导出模块xxx.ts文件**

**1.**在声明是导出

export var a = 5；

export function show( ){////}

**2.**在声明后统一导出

export{a,show}

**3**.默认导出,只能有一次

export default{

a,show

}

**导入模块到yyy.ts文件**

**1**.一般导入

import {a,show} from ‘./xxx’；

**2**.导入默认的导出

import data from ‘./xxx’；

data.a;

data.show;

**命名空间**

**namespace** A{

export let msg:string = “world”;

export class Student{

constructor(

public id:number;

public name:string;

){ }

}

}

A.msg

**装饰器**

随着TS和ES6里引入了类，在一些场景下我们需要额外的特性来支持标注或修改类及其成员。

需要在tsconfig.json文件里修改

“experimentalDecorators”: “true”

class Student{

name:string = “tom”;

show(){

console.log(////);

}

}

let stu = new Student( );

console.log(stu.name);

stu.show( );

动态的改变Student类里面的属性，但不改变其里面的代码

**一.类装饰器**

**1.定义装饰器，其实就是一个函数**

（1）普通装饰器，无法自定义传参

function **decorator1**(target){

console.log(target); 对于类装饰器，接受一个参数target指向类Student，表示构造函数

target.prototype.age = 20; 那么就可以在原型链上装饰他

}

（2）装饰器工厂，可以自定义传参

function **decorator2**(param:string){ 这里是装饰工厂

return function(target){ 这是装饰器

console.log(target);

console.log(param);可以获取到使用装饰器时传递的参数

target.prototype.msg = param;

}

}

**2.使用装饰器**

**@decorator1**

**@decorator2(“hello”)**传给param

class Student{

name:string = “tom”;

show(){

console.log(////);

}

}

let stu = new Student( );

console.log(stu.name);

stu.show( );

**二.属性装饰器**

function **decorator3**(param:number ){

return function(target,property){ 对于属性装饰器，可接受两个参数

console.log(target); 表示当前类的原型对象

console.log(property); 表示当前属性名

console.log(param);

为属性扩展功能

target**[property]** = param;

}

}

class Student{

name:string = “tom”;

**@decorator3(30)**

age:number;

}

**三.方法装饰器**

function **decorator4**(param){

return function(target,methon:string,descriptor){对于方法装饰器，可接受三个参数

console.log(target); 表示当前类的原型对象

console.log(methon);表示当前的方法名

console.log(descriptor);方法的描述符

console.log(descriptor.value)代表方法自身

为方法扩展功能：让方法可以接受并输出

let oldMethod = descriptor.value

descriptor.value = function(...args:any[ ]){重新定义方法

oldMethod.call(this);调用原来的方法，此处的this表示调用方法分对象

console.log(args);

}

}

}

class Student{

name:string = “tom”;

**@decorator4(666)**

print( ){

console.log(/////)

}

}

**描述符里有三个属性（值是布尔值）**

1.writable能不能改值

2.enumerable能不能遍历值

3.configurable能不能删除值

**4.value代表真正的方法**