ts学习

# 基本类型

## boolean:false;

布尔值:true 和false;

**var** isFlag:**boolean**=**false**;

## number:0

一个没有限制的数值;长度64位,c#中的long

**var** orderNumber:**number**=100;

## string:""

一个字符串

**var** userName:**string**="龙梅子";

## Array:

**var** list1:Array<**number**> = [1, 2, 3];//泛型数组

**var** list:**number**[] = [1, 2, 3];

## enum 枚举

不定义值的话,默认从0开始;

**enum** Color1 {Red = 1, Green = 2, Blue = 4};

**var** c2: Color1 = Color1.Green;

设定起始值;

**enum** Color2 {Red = 1, Green, Blue};

**var** colorName: **string** = Color2[2];

## any:undefined

## Void:void

和‘any’相对的数据类型则是’Void‘，它代表没有任何数据类型。我们常用的一个方法

没有任何返回值：

//,格式如：function doMain:void{}

## null 和undefined

Undefined类型只有一个值，即undefined。

Null类型也只有一个值，即null。

undefined 表示一个只有声明没有存储任何值的情况;

null 在执行的时候,引用一个不存在的对象

alert(**typeof** undefined); //output "undefined"

alert(**typeof** **null**); //output "object"

ECMAScript认为undefined是从null派生出来的，所以把它们定义为相等的。

alert(**null** == undefined); //output "true"

如何区分两种原始类型

alert(**null** === undefined); //output "false"

alert(**typeof** **null** == **typeof** undefined); //output "false"

# 资源引用

ts类的引用

/\*\*\*\*使用reference 和path 引入jquery.d.ts 文件使用jquery$就不会报错.\*\*/

/// <reference path="../plugins/typescript/typings/jquery.d.ts" />

\*.d.ts 文件类似一种接口,这种接口在编写代码的时候会有提示,可正常使用;在调试或是发布的时候

D3.d.ts 文件，**.d.ts** 文件不会编译成js 文件

# 接口

四种定义格式,三种实现方式

定义格式:

1.纯属性(包含自选属性) 1 2 3

2.拥有函数定义1 2

3.纯匿名函数定义4

4.纯数组5

5.属性+函数1 2

6.纯构造函数定义 6

实现方式

1.implements :

2. var myaquare1: SquareConfig = { color: "123", time: 0 };

3. var myaquare1 = <SquareConfig>{};

var myaquare1 = <SquareConfig>{};

myaquare1.color = "d";

myaquare1.traceHello = function (info: string) { };

4. var cs: IClock1 = Clock1;//创建一个符合接口定义的类,然后类赋值

5. var myArray: StringArray;

myArray = ["Bob", "Fred"];

6. var mySearch : searchFunt;

mySearch = function(source:string,subString:string)

这里更像是一个结构类型

/\*\*--声明一个接口，这个接口不会在js 上面出现，只会在显示一个user 对象在getUserInfo\*/

**interface** IUserInfo{

age : **any**;//定义一个任何变量的age.

userName :**string**;//定义一个username.

}

/\*\*\*\*\*\*\*\*\*获取用户信息\*\*\*\*\*\*\*/

**function** getUserInfo(user : IUserInfo):**string**{

**return** user.age+"======"+user.userName;

}

//接口的实现,传入这个对象才不会编辑的时候有报错提示  
**var** userObj={userName:'周伯通',age:100};

接口在js中暂时不存在

## (定义自选属性结构)自选属性

这里更像是一个结构类型,可以选择使用变量

**interface** SquareConfig {

color?: **string**;//

width?: **number**;

}

### 赋值方式

赋值1

//必须定义的值不能少

var myaquare1: SquareConfig = { color: "123",time:1 };

赋值2

//必须定义的值可以暂时不赋值,默认为类型初始值;

**var** mySquare2 = < SquareConfig > { };

mySquare2. color = "1234";

赋值3

//可以添加类自己的属性和方法

class Mysquare implements SquareConfig {

time = 0;

public traceHello(info: string) {

alert(info);

}

}

## (定义匿名函数)js的search 函数;

//这种匿名函数定义方式不能被类实现,只能赋值函数;

//只要接口包含匿名函数,就不能使用implements 实现

**interface** searchFunt{

//声明一个两个变量..

(source: **string**, subString: **string**): **boolean**;

}

**var** mySearch : searchFunt;//声明一个interface 变量接收

//传入源字符串和一个字符串,在源字符串中搜索子字符串;

mySearch = **function**(source:**string**,subString:**string**){

**var** result = source.search(subString);

**if** (result == -1) {

**return false**;

}

**else** {

**return true**;

}

}

## 定义Array类型

**interface** StringArray {

[index: **number**]: **string**;

//length: number;

}

**var** myArray:StringArray;

myArray = ["Bob", "Fred"];

## (定义class类型)class实现接口

//当接口中定义方法时,需要用class 实现

//只要接口定义了函数,就必须用**implements或**= <SquareConfig>{}形式**实现**

**interface** IClock {

currentTime: Date;

setTime(d: Date);

}

//--实现IClock 接口

**class** Clock **implements** IClock{

currentTime:Date;

**constructor**(h: **number**, m: **number**) { }//--构造函数方法

setTime(d:Date){

**this**.currentTime=d;

}

}

==============================

## 定义构造函数

//定义构造函数,但不能使用implments 实现,不能包含其它方法;不能包含属性;

**interface** IClock1 {

**new** (hour: **number**, minute: **number**);

}

**class** Clock1 {

currentTime: Date;

**constructor**(h: **number**, m: **number**) { }

}

**var** cs: IClock1 = Clock1;

**var** newClock = **new** cs(7, 30);

console.log(newClock);

## 接口继承

**interface** ISquare **extends** IShape,PenStroke {

sideLength: **number**;

}

# 类

## 构造函数

**constructor**(msg : **string**){

**this**.username=msg;//从构造方法传一个用户字符串过去.

}

## super关键字

在类的任何方法中都可使用

代表对父类的引用;

ts中没有重载,只有覆盖,并且不用override关键字;

## 作用域

public protected(1.3版本) private

默认值是public

protected:只能在子类中使用,实例不能使用;

## static关键字

通过类调用

# module块

## 定义

**module** ValidationUtils{

//提供给外部使用的类

**export class** EmailValidator{

}

}

**module** Shapes {

//===========================多边形===========================

**export module** Polygons {

//===========================三角形===========================

**export class** Triangle {

## import、require、export 关键字的使用

三个关键字搭配使用,达到实现module的效果

//--导入--ValidationUtils3.ts 文件---

//将ValidationUtils3'中的类导入到validation module 中;

// validation模块只能在本文件中使用;

**import** validation = **require**('./ValidationUtils3');

export = 对象的使用

**export = TelValidator4;//后面是某类的类名**

**说明**

如果我们在typescript 使用了module 函数，则生成的代码在

浏览器端执行时，需要有一些script loader 的支持。对于浏览器端代码，我们一般生成amd 风

格的代码，所以需要找一个支持amd 的库放在前端,我这里首选的是AMD

## module的别名使用

**import** polygons = Shapes.Polygons;

## module内部模块