TypeScript 命名空间和模块

目标

1. 熟悉TypeScript中模块的概念和使用
2. 理解和使用命令空间

大纲

# 模块（module）

## 模块介绍

为了解决“全局作用域”可能存在的“命名冲突隐患”问题，从ECMAScript 2015开始，JavaScript引入了模块的概念。TypeScript也沿用这个概念。

模块以文件为单位，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，类等等在模块外部不可见，除非明确“通知”外部公开了哪些内容。

## 模块规范

### CommonJS规范

以node.js为代表，使用require()同步的方法来加载模块。

因为node.js在服务端环境可以直接同步加载本地目录中的文件模块，而速度就是硬盘读写速度。

require()只适用于系统环境（如本地或者服务端系统），浏览器端由于缺少必要环境条件（不具备module、exports、require、global环境参数）不支持，后期要学习的webpack可用于将CommonJS规范应用到浏览器环境中。

例如（按需加载当前要使用的模块）：

**var *MA*** = ***require***( "MA" );

### AMD规范

AMD是"Asynchronous Module Definition"的缩写，意思就是"异步模块定义"。

它采用异步方式加载模块，模块的加载不影响它后面语句的运行。

所有依赖这个模块的语句，都定义在一个回调函数中，等到加载完成之后，这个回调函数才会运行。

在浏览器环境中，因为require一个服务端模块相当于下载文件的过程，这需要时间，也就是只能异步完成。而依赖模块的代码必须等待模块加载完成，才能开始作用。

AMD模块规范以require.js为代表，成功解决一两大问题：

1. 实现js文件的异步加载，避免网页失去响应；
2. 管理模块之间的依赖性，便于代码的编写和维护。

例如（前置加载依赖的所有模块）：

***require***( [ "MA", "MB", "MC" ], **function**( MA, MB, MC )  
{  
 // todo...  
} );

### CMD规范

CMD同样是异步加载模块，只不过其依赖模块的加载采用“就近原则”，只有使用时才require()，而不像AMD规范那样将所有依赖前置。

CMD模块规范以SeaJS为代表，作者是前淘宝UED，现支付宝前端工程师玉伯。

SeaJS 追求简单、自然的代码书写和组织方式，具有以下核心特性：

1. **简单友好的模块定义规范**：SeaJS 遵循CMD规范，可以像Node.js一般书写模块代码。
2. **自然直观的代码组织方式**：依赖的自动加载、配置的简洁清晰，可以让我们更多地享受编码的乐趣。

### ES6 Module

这是ES6标准中所规定的模块规范，目前在TypeScript中，已完全支持ES6模块。

使用export来公开模块内部的内容，使用import来引用这些模块内容。

注意：TypeScript与ECMAScript 2015一样，任何包含顶级import或者export的文件都被当成一个模块。

## 模块导出

### 导出声明

任何声明（比如变量，函数，类，类型别名或接口）都能够通过添加export关键字来导出。

**export let** x = 100;// 变量  
**export const** PI = ***Math***.**PI**;// 常量  
**export type** Name = **string**;// 类型别名  
**export function** *square*( x ){ **return** x \* x; }// 函数  
**export class** C{}// 类  
**export interface IC**{}// 接口

### 导出语句

以上方案难免冗余，出现了太多次数的export，可以集中进行导出：

**let** x = 100;// 变量  
**const** PI = ***Math***.**PI**;// 常量  
**type** Name = **string**;// 类型别名  
**function** *square*( x ){ **return** x \* x; }// 函数  
**class** C{}// 类  
**interface IC**{}// 接口

**export** { x, PI, Name, *square*, C, **IC** }

### 默认导出

每个文件允许存在一个默认导出的default成员，用于在其它模块中加载时指定默认引用的成员。

**export default class** C{}// 导出默认类模块

## 模块导入

即当我们需要使用通过export导出的模块时，使用import关键字来引用这些模块进行使用。

**import** { x} **from** "./test";

**import** { PI} **from** "./test";  
**import** { x, PI, Name, *square*, C, **IC** } **from** "./test";  
***console***.info( x, PI, *square*, C );

以上导入指定模块中单个或者多个模块内容的方法。

警告：from后面的模块文件路径必须以“./”等方式指定路径，若只指定“模块文件名”则代表从npm包的“node\_modules”目录下加载指定的模块。

### 导入默认模块

对于标记于default的模块内容，按以下方法引用进行导入：

// ----- test.ts -----  
**export default class** C{}// 类  
  
// ----- user.ts -----  
**import** C **from** "./test";  
***console***.info( C );

import之后的C可不必要一定与模块定义时的一致，可以是任何新的名称，因为不管什么名字都会指向模块中默认的内容。

### 导入模块时设置别名

只可以对非默认模块内容，使用关键字as定义新的别名，而后通过别名使用。

// ----- test.ts -----  
**export const** PI = ***Math***.**PI**;// 常量  
  
// ----- user.ts -----  
**import** { PI **as** Pai } **from** "./test";  
***console***.info( **Pai** );

同时还可以将整个模块导入到一个变量，通过该变量来访问导出的模块内容：

**import** \* **as** api **from** "./test";

***console***.info( **api.PI** );

## 模块总结

模块存在的意义在于以文件为单位划分“模块作用域”，以避免冲突问题。

若模块中只存在一个导出内容时，强烈建议修改为default默认导出。

——————下午内容——————

# 命名空间（namespace）

## 命名空间介绍

与模块类似，同样是为了避免命名冲突，将多个成员（函数、变量、类、接口）封装到一个指定的“作用域”中，即“命名空间”。

与模块不同的是，每个模块占据一个文件；而命名空间可以分布到不同的文件中，只要命名空间的名字相同，则所有包含的成员都属于同一个命名空间中。

命名空间使用namespace关键字进行定义：

**namespace Validation**{  
 **export interface Validator** {  
 isAcceptable( s:**string** ):**boolean**;  
 }  
}

命名空间也是个封闭的作用域空间，若想让成员在命名空间之外也是可访问的，需要使用 export来进行标记导出。

## 分离到多个文件

// ----- a.ts -----  
**namespace Validation**{  
 **export interface Validator** {  
 isAcceptable( s:**string** ):**boolean**;  
 }  
}  
  
// ----- b.ts -----  
///<reference path="a.ts"/>  
**namespace Validation**{  
 **const** *lettersRegexp* = /^[A-Za-z]+$/;  
 **export class** LettersOnlyValidator **implements Validator** {  
 isAcceptable( s:**string** )  
 {  
 **return** *lettersRegexp*.test( s );  
 }  
 }  
}  
  
// ----- use.ts -----  
///<reference path="a.ts"/>  
///<reference path="b.ts"/>  
  
// use code...

处于同一个命名空间下的内容可以直接访问，因此在b.ts中实现接口时可以直接写“**Validator**”而不需要加命名空间前缀。

而在use.ts中已经不再处于相同的命名空间作用域下，所以访问其中的成员要加前缀如：

“**Validation.Validator**”

### 分离后的使用方法

第一种办法，可以将所有命名空间的分离文件打包到同一个文件中。

tsc --outFile sample.js a.ts b.ts use.ts

另外，若目标环境是部署到浏览器的HTML文件中，同样可以将每个ts文件单独编译，然后通过<script>标签来进行逐个加载。

<script src="a.js" />  
<script src="b.js" />  
<script src="use.js" />

# 模块和命名空间总结

命名空间无法有效的组织和识别各自之间的依赖关系，尤其是大型应用中。

而模块是Node.js默认推荐的组织代码的方式，可以有效的处理模块之间的依赖关系，具有更强的封闭性，尤其是对“组件化”编程有举足轻重的作用。

对于模块而言，应该使用import进行导入，而不是错误的使用 /// <reference>。

编译器在解析import语句时，总是会优先查找指定路径下的.ts、.tsx，若找不到则会查找外部模块的声明.d.ts。

# 应用模块化流程

## TypeScript编译AMD规范模块

按照ES6 Module规范进行导出（export）和导入（import）模块，然后再使用以下命令行工具进行编译，则自动转换成define()的异步模块JavaScript代码。

tsc Button.ts --module amd --target es5 --lib es2015,dom

--module amd 编译成amd模块规范

--target es5 编译成符合ECMAScript5规范的JavaScript代码

--lib es2015,dom 附加promise，async/await等高级语言特性支持

## 使用require.js加载使用AMD模块

### require.js加载机制

RequireJS使用head.appendChild()将每一个依赖加载为一个script标签。

RequireJS等待所有的依赖加载完毕，计算出模块定义函数正确调用顺序，然后依次调用它们。

### 使用示例（JavaScript）：

***require***( [ "helper/util" ], **function**( util )  
{  
 // util.js 使用 define() 来定义异步 AMD 模块  
 // 加载 helper/util.js 模块完成后，执行该回调方法  
 // 所加载的依赖模块会注入到该函数的 util 参数  
} );

### 配置与使用第三方库

以jQuery为例，其具备相当多的版本如jQuery-v1.x、jQuery-v2.x等等

可以使用RequireJS的config来进行目录指定从而进行加载不同版本的第三方库。

***require***.**config**( {  
 **baseUrl**: 'js/lib',  
 **paths**: {  
 **jquery**: 'jquery-1.9.0'  
 // https://cdn.xx.com/lib/jquery-1.9.0.min.js  
 }  
} );