基于WebGL的三维评审前端组件化规范与实现

|  |  |
| --- | --- |
| 部 门： | 项目中心研发部 |
| 姓 名： | 周文 |
| 晋升岗位： | 交互设计师C |
| 晋升时间： | 2018年12月 |
| 论文类型： | [ ]方案研究 [√]技术研究 |

**ABSTRACT :** Web pages are more and more like desktop programs, requiring a team division of labor, schedule management, unit testing, etc. Developers have to use software engineering methods to manage the business logic of web pages. Javascript modular programming has become an urgent need. Ideally, developers only need to implement the core business logic, others can load modules that others have already written, and modular development comes into being. The so-called modularity refers to dividing the system code into a series of single-responsible and replaceable modules. Modular development refers to how to develop new modules and reuse existing modules to realize the functions of the applications. The code is easy to reuse, greatly improving development efficiency, and also facilitating post-maintenance.

**Key Words:** Javascript, modularity, code reuse, development efficiency, easy maintenance

**摘要：**网页越来越像桌面程序，需要一个团队分工协作、进度管理、单元测试等等，开发者不得不使用软件工程的方法，管理网页的业务逻辑。Javascript模块化编程，已经成为一个迫切的需求。理想情况下，开发者只需要实现核心的业务逻辑，其他都可以加载别人已经写好的模块，模块化开发应运而生。所谓的模块化就是指把系统代码分为一系列职责单一且可替换的模块，模块化开发是指如何开发新的模块和复用已有的模块来实现应用的功能。代码方便重用，很大的提升开发效率，同时也方便后期维护。

**关键字：**Javascript、模块化、代码重用、开发效率、方便维护

# 0、引言

在ajax还未提出之前，js还只是一种“玩具语言”，由Brendan Eich花了不到十天时间发明，用来在网页上进行表单校验、实现简单的动画效果等等，你可以回想一下那个网页上到处有公告块飘来飘去的时代。这个时候并没有前端工程师，服务端工程师只需在页面上随便写写js就能搞定需求。2006年，ajax的概念被提出，前端拥有了主动向服务端发送请求并操作返回数据的能力，随着Google将此概念的发扬光大，传统的网页慢慢的向“富客户端”发展。前端的业务逻辑越来越多，代码也越来越多，于是一些问题就暴漏了出来：全局变量的灾难、函数命名冲突、依赖关系不好管理等等。直到2009年，nodejs横空出世，开创了一个新纪元，人们可以用js来编写服务端的代码了。如果说浏览器端的js即便没有模块化也可以忍的话，那服务端是万万不能的。于是大牛云集的CommonJs社区发力，制定了Modules/1.0规范，此规范一出，立刻产生了良好的效果，由于其简单而直接，在nodejs中，这种模块化方案立刻被推广开了。但是Modules/1.0规范源于服务端，无法直接用于浏览器端，所以后面便产生了AMD/RequireJs和后起之秀CMD/SeaJs。

本文将针对AMD/RequireJs展开研究与分析，基于require.js制定相应的模块化规范，通过规范将基于WebGL的三维评审项目进行前端组件化，提升开发和维护的效率。

# 模块化规范介绍

## CommonJs

CommonJS是nodejs也就是服务器端广泛使用的模块化机制。 该规范的主要内容是，模块必须通过module.exports 导出对外的变量或接口，通过 require() 来导入其他模块的输出到当前模块作用域中。根据这个规范，每个文件就是一个模块，有自己的作用域，文件中的变量、函数、类等都是对其他文件不可见的。如果想在多个文件分享变量，必须定义为global对象的属性（不推荐）。

1. 定义模块

在每个模块内部，module变量代表当前模块。它的exports属性是对外的接口，将模块的接口暴露出去。其他文件加载该模块，实际上就是读取module.exports变量。

|  |
| --- |
| var x = 5;  var addX = function (value) {  return value + x;  };  module.exports.x = x;  module.exports.addX = addX; |

1. 加载模块

require方法用于加载模块，后缀名默认为.js。

|  |
| --- |
| var app = require('./app.js'); |

模块加载的顺序，按照其在代码中出现的顺序,根据参数的不同格式，require命令去不同路径寻找模块文件。

如果参数字符串以“/”开头，则表示加载的是一个位于绝对路径的模块文件。

如果参数字符串以“./”开头，则表示加载的是一个位于相对路径的模块文件.

如果参数字符串不以“./“或”/“开头，则表示加载的是一个默认提供的核心模块（node核心模块，或者通过全局安装或局部安装在node\_modules目录中的模块）

1. 入口文件

一般都会有一个主文件（入口文件），在index.html中加载这个入口文件，然后在这个入口文件中加载其他文件。commonJS用同步的方式加载模块，在服务端，模块文件都存在本地磁盘，读取非常快，所以这样做不会有问题。但是在浏览器端，限于网络原因，更合理的方案是使用异步加载。

## CMD/SeaJs

CMD 即Common Module Definition通用模块定义，CMD规范是国内发展出来的，就像AMD有个requireJS，CMD有个浏览器的实现SeaJS，SeaJS要解决的问题和requireJS一样，只不过在模块定义方式和模块加载（可以说运行、解析）时机上有所不同。Sea.js 推崇一个模块一个文件，遵循统一的写法。

1. 定义模块

在 CMD 规范中，一个模块就是一个文件，代码的书写格式如下:

|  |
| --- |
| define(function(require, exports, module) {  // 模块代码  }); |

require是可以把其他模块导入进来的一个参数;而exports是可以把模块内的一些属性和方法导出的;module 是一个对象，上面存储了与当前模块相关联的一些属性和方法。

1. 加载模块

AMD是依赖关系前置,在定义模块的时候就要声明其依赖的模块;CMD则是按需加载依赖就近,只有在用到某个模块的时候再去require。

|  |
| --- |
| define(function (require, exports, module) {  var moduleA = require('add.js')  console.log(moduleA.add(10,20))  }) |

1. 入口文件

requireJS通过data-main来设置入口，而seaJS则通过sea.use()来设置。sea.js 在下载完成后，会自动加载入口模块。

|  |
| --- |
| <script src="sea.js"></script>  <script>  seajs.use('js/main');  </script> |

Sea.js  配置（多放在入口文件中）

|  |
| --- |
| seajs.config({  // 别名配置  alias: {  'es5-safe': 'gallery/es5-safe/0.9.3/es5-safe',  'json': 'gallery/json/1.0.2/json',  'jquery': 'jquery/jquery/1.10.1/jquery'  },  // 路径配置  paths: {  'gallery': 'https://a.alipayobjects.com/gallery'  },  // Sea.js 的基础路径  base: 'http://example.com/path/to/base/',  }); |

## AMD/RequireJs

AMD是Asynchronous Module Definition的缩写，即异步模块加载机制，从它的规范描述页面看，AMD很短也很简单，但它却完整描述了模块的定义，依赖关系，引用关系以及加载机制。对于 JSer 来说，异步是再也熟悉不过的词了，所有的模块将被异步加载，模块加载不影响后面语句运行。所有依赖某些模块的句均放置在回调函数中。

1. 定义模块

define方法用于定义模块，RequireJS要求每个模块放在一个单独的文件里。按照是否依赖其他模块，可以分成两种情况讨论。第一种情况是定义独立模块，即所定义的模块不依赖其他模块；第二种情况是定义非独立模块，即所定义的模块依赖于其他模块。

1. 独立模块

如果被定义的模块是一个独立模块，不需要依赖任何其他模块，可以直接用define方法生成。

|  |
| --- |
| define({  method1: function() {},  method2: function() {},  }); |

上面代码生成了一个拥有method1、method2两个方法的模块。

1. 非独立模块

如果被定义的模块需要依赖其他模块，则define方法必须采用下面的格式。

|  |
| --- |
| define(['module1', 'module2'], function(m1, m2) {  ...  }); |

define方法的第一个参数是一个数组，它的成员是当前模块所依赖的模块。比如，[‘module1’, ‘module2’]表示我们定义的这个新模块依赖于module1模块和module2模块，只有先加载这两个模块，新模块才能正常运行。一般情况下，module1模块和module2模块指的是，当前目录下的module1.js文件和module2.js文件，等同于写成[’./module1’, ‘./module2’]。

1. 加载模块

require方法用于调用模块。它的参数与define方法类似。

|  |
| --- |
| require(['foo', 'bar'], function ( foo, bar ) {  foo.doSomething();  }); |

上面方法表示加载foo和bar两个模块，当这两个模块都加载成功后，执行一个回调函数，该回调函数就用来完成具体的任务。

1. 入口文件

require.js在加载的时候会检查data-main属性，当requireJS自身加载执行后，就会再次异步加载data-main属性指向的main.js。这个main.js是当前网页所有逻辑的入口，理想情况下，整个网页只需要这一个script标记，利用requireJS加载依赖的其它文件。

注意在main.js中所设置的脚本是异步加载的。所以如果在页面中配置了其它JS加载，则不能保证它们所依赖的JS已经加载成功。在RequireJS内部，会使用head.appendChild()将每一个模块依赖加载为一个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。

## 总结

CommonJs用在服务器端，AMD和CMD用在浏览器环境。

AMD 是 RequireJS 在推广过程中对模块定义的规范化产出。CMD 是 SeaJS 在推广过程中对模块定义的规范化产出。

AMD:提前执行（异步加载：依赖先执行）+延迟执行。CMD:延迟执行（运行到需加载，根据顺序执行）。

AMD和CMD最大的区别是对依赖模块的执行时机处理不同，注意不是加载的时机或者方式不同。很多人说requireJS是异步加载模块，SeaJS是同步加载模块，这么理解实际上是不准确的，其实加载模块都是异步的，只不过AMD依赖前置，js可以方便知道依赖模块是谁，立即加载，而CMD就近依赖，需要使用把模块变为字符串解析一遍才知道依赖了那些模块，这也是很多人诟病CMD的一点，牺牲性能来带来开发的便利性，实际上解析模块用的时间短到可以忽略。

同样都是异步加载模块，AMD在加载模块完成后就会执行该模块，所有模块都加载执行完后会进入require的回调函数，执行主逻辑，这样的效果就是依赖模块的执行顺序和书写顺序不一定一致，看网络速度，哪个先下载下来，哪个先执行，但是主逻辑一定在所有依赖加载完成后才执行。CMD加载完某个依赖模块后并不执行，只是下载而已，在所有依赖模块加载完成后进入主逻辑，遇到require语句的时候才执行对应的模块，这样模块的执行顺序和书写顺序是完全一致的。这也是很多人说AMD用户体验好，因为没有延迟，依赖模块提前执行了，CMD性能好，因为只有用户需要的时候才执行的原因。

# RequireJs使用详解

## 2.1、require.js引入

RequireJS的基本思想是，通过define方法，将代码定义为模块；通过require方法，实现代码的模块加载。

首先，将require.js嵌入网页，然后就能在网页中进行模块化编程了。

|  |
| --- |
| <script data-main="scripts/main" src="scripts/require.js"></script> |

上面代码的data-main属性不可省略，用于指定主代码所在的脚本文件，在上例中为scripts子目录下的main.js文件。用户自定义的代码就放在这个main.js文件中。

## 2.2、封装定义控件模块

define方法用于定义模块，RequireJS要求每个模块放在一个单独的文件里。按照是否依赖其他模块，可以分成两种情况讨论。第一种情况是定义独立模块，即所定义的模块不依赖其他模块；第二种情况是定义非独立模块，即所定义的模块依赖于其他模块。

1. AMD实例

下面代码定义了一个alpha模块，并且依赖于内置的require，exports模块，以及外部的beta模块。可以看到，第三个参数是回调函数，可以直接使用依赖的模块，他们按依赖声明顺序作为参数提供给回调函数。

　　这里的require函数让你能够随时去依赖一个模块，即取得模块的引用，从而即使模块没有作为参数定义，也能够被使用；exports是定义的alpha 模块的实体，在其上定义的任何属性和方法也就是alpha模块的属性和方法。通过exports.verb = ...就是为alpha模块定义了一个verb方法。例子中是简单调用了模块beta的verb方法。

|  |
| --- |
| define("alpha", ["require", "exports", "beta"], function (require, exports, beta) {    　　exports.verb = function() {  　　 return beta.verb();  　　}  }); |

1. 匿名模块

define方法允许你省略第一个参数，这样就定义了一个匿名模块，这时候模块文件的文件名就是模块标识。如果这个模块文件放在a.js中，那么a就是模块名。可以在依赖项中用"a"来依赖于这个匿名模块。这带来一个好处，就是模块是高度可重用的。你拿来一个匿名模块，随便放在一个位置就可以使用它，模块名就是它的文件路径。这也很好的符合了DRY（Don't Repeat Yourself）原则。

下面的代码就定义了一个依赖于alpha模块的匿名模块：

|  |
| --- |
| define(["alpha"], function (alpha){  　　return {  　　 verb: function(){  　　 return alpha.verb()+2;  　　 }  　　};  }); |

1. 独立模块

如果被定义的模块是一个独立模块，不需要依赖任何其他模块，可以直接用define方法生成。

|  |
| --- |
| define({  method1: function() {},  method2: function() {},  }); |

上面代码生成了一个拥有method1、method2两个方法的模块。

另一种等价的写法是，把对象写成一个函数，该函数的返回值就是输出的模块。

|  |
| --- |
| define(function () {  return {  method1: function() {},  method2: function() {},  };  }); |

后一种写法的自由度更高一点，可以在函数体内写一些模块初始化代码。

值得指出的是，define定义的模块可以返回任何值，不限于对象。

1. 非独立模块

如果被定义的模块需要依赖其他模块，则define方法必须采用下面的格式。

|  |
| --- |
| define(['module1', 'module2'], function(m1, m2) {  ...  }); |

define方法的第一个参数是一个数组，它的成员是当前模块所依赖的模块。比如，['module1', 'module2']表示我们定义的这个新模块依赖于module1模块和module2模块，只有先加载这两个模块，新模块才能正常运行。一般情况下，module1模块和module2模块指的是，当前目录下的module1.js文件和module2.js文件，等同于写成['./module1', './module2']。

define方法的第二个参数是一个函数，当前面数组的所有成员加载成功后，它将被调用。它的参数与数组的成员一一对应，比如function(m1, m2)就表示，这个函数的第一个参数m1对应module1模块，第二个参数m2对应module2模块。这个函数必须返回一个对象，供其他模块调用。

|  |
| --- |
| define(['module1', 'module2'], function(m1, m2) {  return {  method: function() {  m1.methodA();  m2.methodB();  }  };  }); |

上面代码表示新模块返回一个对象，该对象的method方法就是外部调用的接口，menthod方法内部调用了m1模块的methodA方法和m2模块的methodB方法。

需要注意的是，回调函数必须返回一个对象，这个对象就是你定义的模块。如果依赖的模块很多，参数与模块一一对应的写法非常麻烦。

|  |
| --- |
| define(  [ 'dep1', 'dep2', 'dep3', 'dep4', 'dep5', 'dep6', 'dep7', 'dep8'],  function(dep1, dep2, dep3, dep4, dep5, dep6, dep7, dep8){  ...  }  ); |

为了避免像上面代码那样繁琐的写法，RequireJS提供一种更简单的写法。

|  |
| --- |
| define(  function (require) {  var dep1 = require('dep1'),  dep2 = require('dep2'),  dep3 = require('dep3'),  dep4 = require('dep4'),  dep5 = require('dep5'),  dep6 = require('dep6'),  dep7 = require('dep7'),  dep8 = require('dep8');  ...  }  }); |

1. 文件目录结构规范

整个预览服务前端资源目录结构参考下图：

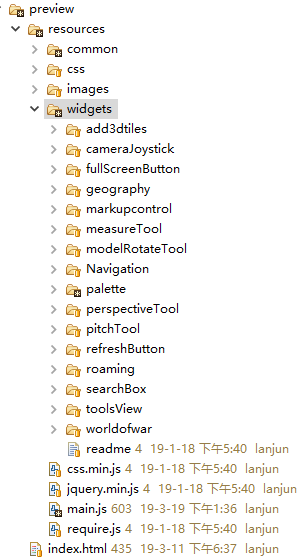


图1前端资源目录结构

index.html页面是整个预览服务的主页面，resources是项目的资源文件目录，通用js插件放common文件夹，框架样式和通用的图片分别放在css和images文件夹。widgets文件夹存放各个模块化组件，每个组件涉及的html、css、图片和js都存放在该组件对应文件夹里面。css.min.js是require需要用到的css加载插件，jquery.min.js是jquery依赖，main.js是js入口文件，require.js是模块化必须的依赖文件。



图2 index.html页面

该页面只引用页面框架布局相关js和css文件，一个div容器和require引入的script标签，页面简单明了。

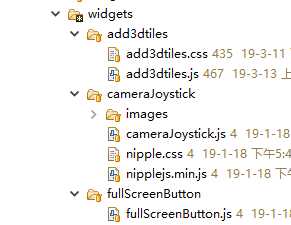


图3模块化组件目录

各个组件相对应的css、js和图片都在各自的文件夹，各自互不影响，css的依赖会在main.js里面进行声明。

## 2.3、调用模块

require方法用于调用模块。它的参数与define方法类似。

|  |
| --- |
| require(['foo','bar'], function ( foo, bar ) {  foo.doSomething();  }); |

上面方法表示加载foo和bar两个模块，当这两个模块都加载成功后，执行一个回调函数。该回调函数就用来完成具体的任务。require方法的第一个参数，是一个表示依赖关系的数组。

require方法也可以用在define方法内部。

|  |
| --- |
| define(function (require) {  var otherModule = require('otherModule');  }); |

require方法允许添加第三个参数，即错误处理的回调函数。

|  |
| --- |
| require(  [ "backbone" ],  function ( Backbone ) {  return Backbone.View.extend({ /\* ... \*/ });  },  function (err) {  // ...  }  ); |

require方法的第三个参数，即处理错误的回调函数，接受一个error对象作为参数。

require对象还允许指定一个全局性的Error事件的监听函数。所有没有被上面的方法捕获的错误，都会被触发这个监听函数。

|  |
| --- |
| requirejs.onError = function (err) {  // ...  }; |

小结：define和require这两个定义模块、调用模块的方法，合称为AMD模式。它的模块定义的方法非常清晰，不会污染全局环境，能够清楚地显示依赖关系。

AMD模式可以用于浏览器环境，并且允许非同步加载模块，也可以根据需要动态加载模块。

## 2.4、配置require.js：config方法

require方法本身也是一个对象，它带有一个config方法，用来配置require.js运行参数。config方法接受一个对象作为参数。

|  |
| --- |
| require.config({  paths: {  jquery: [ '//cdnjs.cloudflare.com/ajax/libs/jquery/2.0.0/jquery.min.js',  'lib/jquery'  ]  }  }); |

config方法的参数对象有以下主要成员：

1. paths

paths参数指定各个模块的位置。这个位置可以是同一个服务器上的相对位置，也可以是外部网址。可以为每个模块定义多个位置，如果第一个位置加载失败，则加载第二个位置，上面的示例就表示如果CDN加载失败，则加载服务器上的备用脚本。需要注意的是，指定本地文件路径时，可以省略文件最后的js后缀名。

|  |
| --- |
| paths : {  jquery : 'jquery.min',  util : 'common/util',  Cesium :'common/Cesium/Cesium',  DrawUtil :'common/DrawUtil/DrawUtil'  } |

1. baseUrl

baseUrl参数指定本地模块位置的基准目录，即本地模块的路径是相对于哪个目录的。该属性通常由require.js加载时的data-main属性指定。

1. map

map告诉RequireJS在任何模块之前，都先载入这个css模块。

1. shim

有些库不是AMD兼容的，这时就需要指定shim属性的值。shim可以理解成“垫片”，用来帮助require.js加载非AMD规范的库。

|  |
| --- |
| require.config({  paths: {  "backbone": "vendor/backbone",  "underscore": "vendor/underscore"  },  shim: {  "backbone": {  deps: [ "underscore" ],  exports: "Backbone"  },  "underscore": {  exports: "\_"  }  }  }); |

上面代码中的backbone和underscore就是非AMD规范的库。shim指定它们的依赖关系（backbone依赖于underscore），以及输出符号（backbone为“Backbone”，underscore为“\_”）。

# 后台管理系统

## 3.1、整体功能界面

三维评审前端组件化后台管理系统主要包括列表展现、查询、新增、编辑、删除和预览功能。列表展示控件名称、创建时间、更新时间、控件描述和操作列。查询框可以对列表进行查询筛选，新增按钮弹出新增表单，可以上传相关控件附件，操作列可以对单条数据进行编辑和删除。预览按钮点击会根据勾选的控件进行预览，主界面如下：

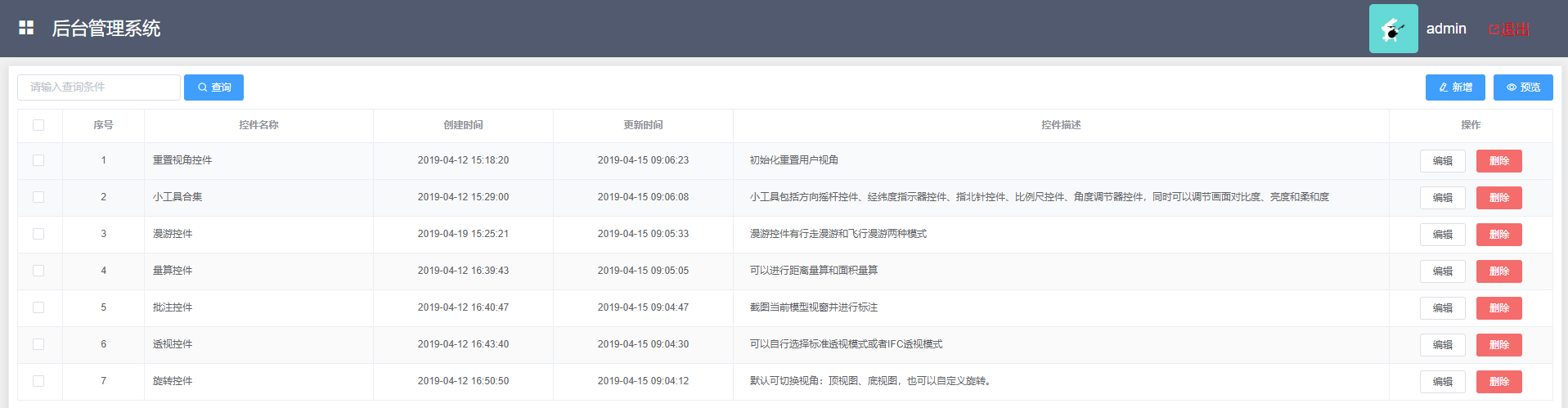


图4 后台管理系统界面



图5新增表单



图6编辑表单

## 3.2、后台解析原理

用户在后台管理系统上传该控件的压缩包，根据用户勾选的id列表，获取对应的压缩包文件(每个压缩包文件名称与js对象名称对应)，解压压缩包，将解压出来的压缩包数据拷贝到widgets文件夹下面，根据压缩包的文件名称(与js的对象名称保持一致)，生成模板数据，通过freemaker模板引擎生成main.js，替换预览服务中的原main.js，即可让脚本立即生效，从而实现预览效果。

其中动态生成main.js代码如下：

动态生成path

|  |
| --- |
| paths : {  <#list refs?keys as key>  ${key}: '${refs["${key}"]}',  </#list>    }, |

动态引用组件

|  |
| --- |
| require( ["Cesium","util","FullScreenUtil","imageterrainProvide","add3dtiles","worldofwar","fullScreenButton","searchBox","palette",  <#list jsonObjs as jsonObj>  "${jsonObj}",  </#list>  ],  function(Cesium,util,FullScreenUtil,imageterrainProvide,add3dtiles,worldofwar,fullScreenButton,searchBox, palette,  <#list jsonObjs as jsonObj>  ${jsonObj},  </#list>  ){} |

动态调用组件

|  |
| --- |
| <#list inits as init>  ${init};  </#list> |

# 4、总结

CommonJS规范主要用于服务端编程，加载模块是同步的，这并不适合在浏览器环境，因为同步意味着阻塞加载，浏览器资源是异步加载的，因此有了AMD CMD解决方案。

AMD规范在浏览器环境中异步加载模块，而且可以并行加载多个模块。RequireJS 采取的是在源码中预留接口的形式，源码中留有为插件而写的代码。

CMD规范与AMD规范很相似，都用于浏览器编程，依赖就近，延迟执行，可以很容易在Node.js中运行。不过，依赖SPM 打包，模块的加载逻辑偏重。

ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。但是因为采用了最新的ES2015标准，在浏览器兼容方面存在很大问题，Edge 15可以支持 96% 的 ES6 新特性。Edge 14 可以支持 93% 的 ES6 新特性。（IE7~11 基本不支持 ES6）win10之后的浏览器是edge。之前的是IE。但是这是一个很好的规范，可以拿来借鉴。

前端技术的发展是飞快的，模块化也是必须的，除了解决命名冲突和依赖管理，使用模块化开发还可以带来很多好处：

模块的版本管理。通过别名等配置，配合构建工具，可以比较轻松地实现模块的版本管理。

提高可维护性。模块化可以让每个文件的职责单一，非常有利于代码的维护。

前端性能优化。RequireJS 和 SeaJS 都是通过异步加载模块，这对页面性能非常有益。Sea.js 还提供了 combo、flush等插件，配合服务端，可以很好地对页面性能进行调优。

跨环境共享模块。CMD 模块定义规范与 Node.js 的模块规范非常相近。

# 5、参考文献

[1] 赵建保 JavaScript前端开发模块化教程 2019