基于WebGL的三维可视化组件化规范与实现

|  |  |
| --- | --- |
| 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 :** The concept of front-end modularization has a long history. Modular development not only simplifies the structure of code, but also facilitates the combination and decomposition of modules. The more important ones can eliminate the dependency and naming conflicts between various files. It does not interfere, facilitates the debugging and upgrading of individual module functions, and also greatly improves the convenience for later maintenance.

**Key Words:** Modular, Simplified code

**摘要：**前端模块化的概念由来已久，采用模块化开发不仅可以简化代码的结构，方便模块之间的组合和分解，更重要的式可以消除各种文件之间的依赖和命名冲突，多人协作互不干扰，方便单个模块功能调试和升级，同时也给后期的维护大大提升了便利性。

**关键字：**模块化、简化代码

# 0、引言

模块化就是将程序划分成若干个独立的模块，每个模块完成一个特定子功能，每个模块即相对独立，又相互联系，他们共同完成系统指定的各项功能。目前公司基于WebGL的三维可视化项目涉及的前端插件较多，页面复杂度几冗余度比较高，js插件之间的依赖关系不明确，开发和维护的成本比较高。所以我们需要对其进行优化，采用模块化开发规范对功能进行整理分割，按照功能提取其相关的js、css和与该功能相关的文件做成单独的模块，降低各个组件之间的耦合度，最终实现可插拔式的模块。服务器端我们可以采用CommonJs模块化规范，浏览器端则可以采用AMD或者CMD规范。

# 模块化规范介绍

## CommonJs

CommonJS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。

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。

|  |
| --- |
| 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'); |

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和AMD

由于CommonJs是服务器端的规范，浏览器端无法使用，所以浏览器端我们可以采用CMD和AMD规范，下表列出CMD和AMD两个浏览器端规范的异同：

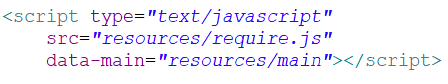
|  |  |  |
| --- | --- | --- |
|  | CMD | AMD |
| 使用范围 | 浏览器端使用 | 浏览器端使用 |
| 实现 | Sea.js | Require.js |
| 加载方式 | 异步加载 | 异步加载 |
| 模块依赖 | 就近依赖，需要把模块变为字符串解析一遍才知道依赖了那些模块 | 依赖前置，js可以方便知道依赖模块是谁，立即加载 |
| 模块执行时机 | 加载完某个依赖模块后并不执行，只是下载而已，在所有依赖模块加载完成后进入主逻辑，遇到require语句的时候才执行对应的模块，这样模块的执行顺序和书写顺序是完全一致的。 | 加载模块完成后就会执行该模块，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 |
| 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  }); | **独立模块：**  define({  method1: function() {},  method2: function() {},  });  **非独立模块：**  define(['module1', 'module2'], function(m1, m2) {  }); |
| 加载模块 | define(function (require, exports, module) {  var moduleA = require('add.js')  console.log(moduleA.add(10,20))  }) | require(['foo', 'bar'], function ( foo, bar ) { foo.doSomething();  }); |
| 入口文件 | 通过sea.use()来设置：  <script src="sea.js"></script>  <script>  seajs.use('js/main');  </script> | 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：  <script data-main="scripts/main" src="scripts/require.js"></script> |

# 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。



## 2.2、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。

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。

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

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。

**下面介绍几种模块定义的方法：**

|  |  |  |
| --- | --- | --- |
| 匿名模块 | 定义 | 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。 |
| 代码 | define(["alpha"], function (alpha){  　　return {  　　 verb: function(){  　　 return alpha.verb()+2;  　　 }  　　};  }); |
| 独立模块 | 定义 | 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。 |
| 代码 | define({  method1: function() {},  method2: function() {},  }); |
| 非独立模块 | 定义 | 如果被定义的模块需要依赖其他模块，则define方法必须加入依赖的模块参数。第一个参数是一个数组，是当前模块所依赖的模块，只有先加载这些模块，新模块才能正常运行。  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 |
| 代码 | define(['module1', 'module2'], function(m1, m2) {  ...  });  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，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');  ...  }  }); |

## 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**

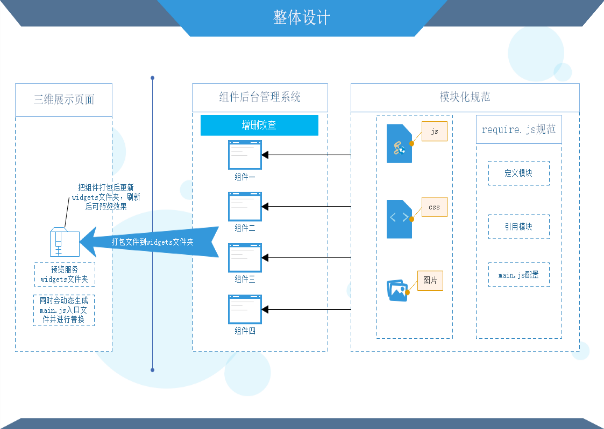
有些库不是AMD兼容的，这时就需要指定shim属性的值。shim可以理解成“垫片”，用来帮助require.js加载非AMD规范的库。

|  |
| --- |
| 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“\_”）。

# 组件化规范与实现

## 3.1、整体设计



整体设计思路：按照require.js模块化定义模块组件，把该组件相关的js、css和图片等资源按文件夹级别分类存放。通过组件后台管理系统进行管理，可以对组件进行增删改查，勾选需要的组件点击预览会将勾选的组件打包至预览服务的widgets文件夹，同时动态改变入口文件main.js，刷新页面即可展现预览效果。

## 3.2、文件目录结构

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：

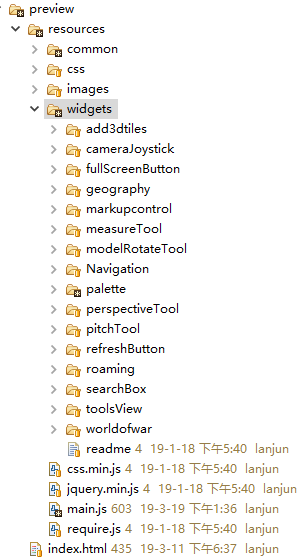


图1前端资源目录结构

|  |  |
| --- | --- |
| **名称** | **说明** |
| index.html | 整个预览服务的主页面 |
| resources | 项目的资源文件目录 |
| common | 框架通用js插件目录 |
| css | 框架通用css目录 |
| images | 框架通用图片目录 |
| widgets | 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子文件夹里面 |
| main.js | require.js的入口文件 |
| css.min.js | require需要用到的css加载插件 |



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。

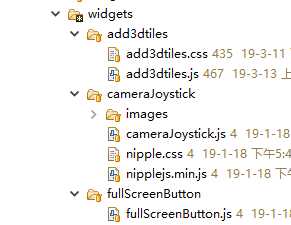


图3模块化组件目录

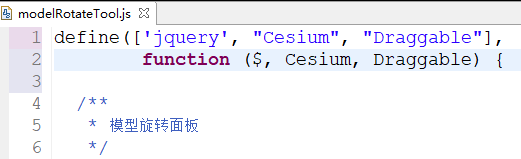
各个组件相对应的css、js和图片都在各自的文件夹，各自互不影响，css的依赖会在main.js里面进行声明。

## 3.3、模块化示例

下面以模型旋转组件为例：

1. **定义模块：**

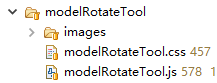
采用匿名模块定义方式，因为该组件需要依赖其他模块，所以是非独立模块，定义方式如下：



数组为依赖的模块，第二个参数是回调函数，可以在回调函数进行业务处理，回调函数的参数和依赖模块保持一至，这样我们就可以在回调中使用这些依赖的模块。最后通过return把要公开的方法return出来，其他模块就可以通过依赖该组件然后调用return出来的方法。

1. **配置模块：**

定义完成的模型旋转组件目录结构如下：

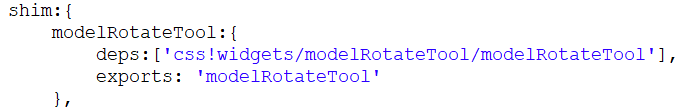


模型旋转组件的js、css和需要的图片都在这个文件夹，需要通过main.js进行配置。

path指定模块对应的文件路径：

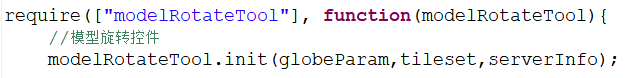


shim制定模块依赖的css文件



1. **调用模块**

现在我们就可以直接通过require的方式进行调用：



## 3.4、已模块化组件列表

基于WebGL的三维评审项目对用到的组件和插件进行了模块化，所有的模块化组件如下图：

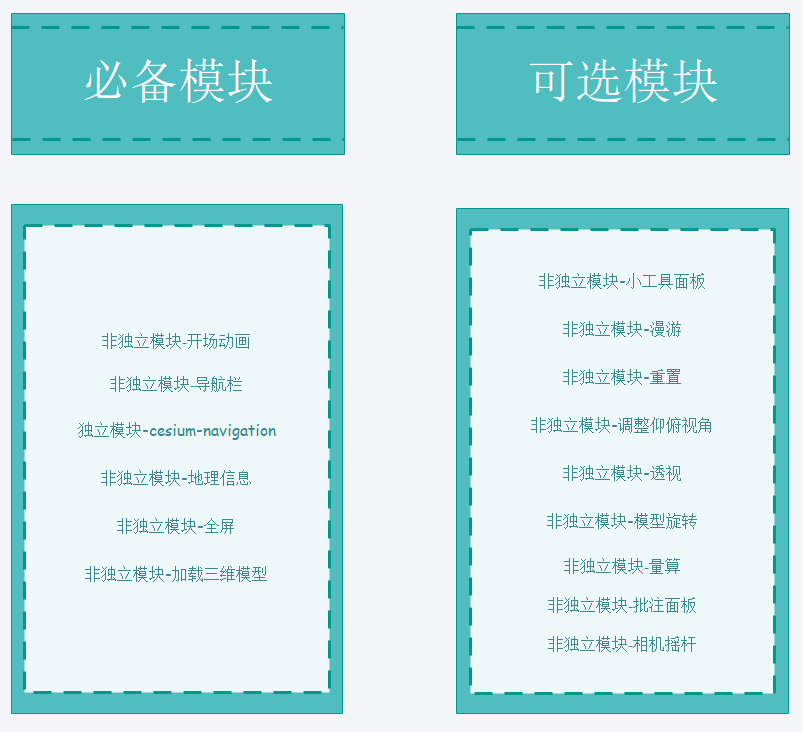


图3模块化组件目录

**必选模块**：Web展现三维模型必须的模块，主要是加载三维模型并显示，必选模块如下：

|  |  |
| --- | --- |
| 开场动画 | C:\Users\zw\Desktop\lwpic\非独立模块-开场动画.jpg |
| 导航栏 | C:\Users\zw\Desktop\lwpic\非独立模块-导航栏.jpg |
| cesium-navigation | C:\Users\zw\Desktop\lwpic\独立模块-cesium-navigation.jpg |
| 地理信息 | C:\Users\zw\Desktop\lwpic\非独立模块-地理信息.jpg |
| 全屏 | C:\Users\zw\Desktop\lwpic\非独立模块-全屏.jpg |
| 加载三维模型 | C:\Users\zw\Desktop\lwpic\非独立模块-加载三维模型.jpg |
|  |  |

**可选模块**：Web展现三维模型非必需模块，根据自身需要的功能选择性加载，可选模块如下：

|  |  |
| --- | --- |
| 小工具面板 | C:\Users\zw\Desktop\lwpic\非独立模块-小工具面板.jpg |
| 漫游 | C:\Users\zw\Desktop\lwpic\非独立模块-漫游.jpg |
| 重置 | C:\Users\zw\Desktop\lwpic\非独立模块-重置.jpg |
| 调整仰俯视角 | C:\Users\zw\Desktop\lwpic\非独立模块-调整仰俯视角.jpg |
| 透视 | C:\Users\zw\Desktop\lwpic\非独立模块-透视.jpg |
| 模型旋转 | C:\Users\zw\Desktop\lwpic\非独立模块-模型旋转.jpg |
| 量算 | C:\Users\zw\Desktop\lwpic\非独立模块-量算.jpg |
| 批注面板 | C:\Users\zw\Desktop\lwpic\非独立模块-批注面板.jpg |
| 相机摇杆 | C:\Users\zw\Desktop\lwpic\非独立模块-相机摇杆.jpg |

# 组件后台管理

## 4.1、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：

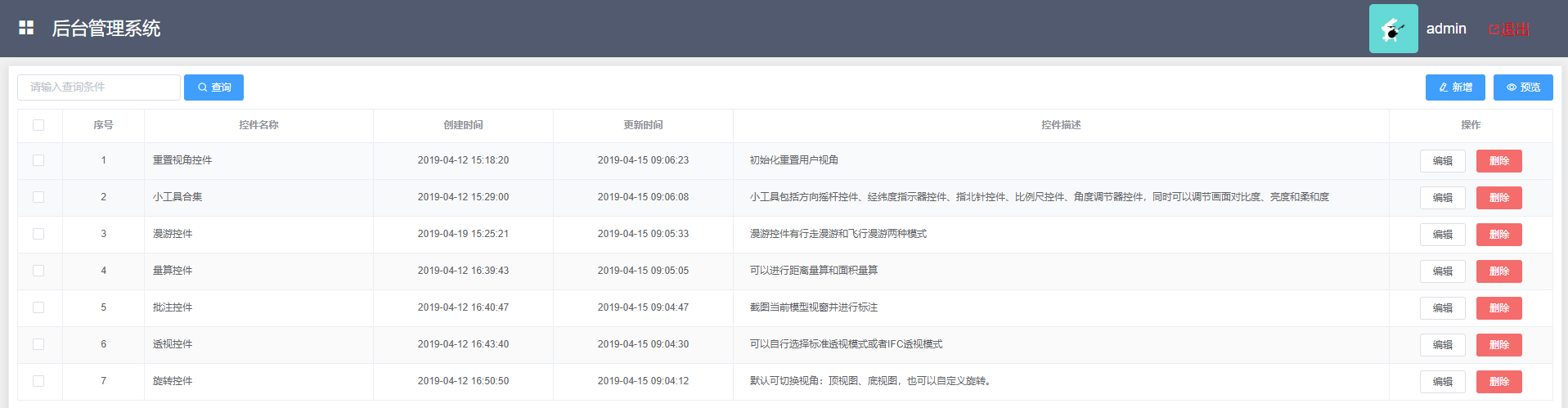


图19 后台管理系统界面



图20新增



图21编辑

## 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。

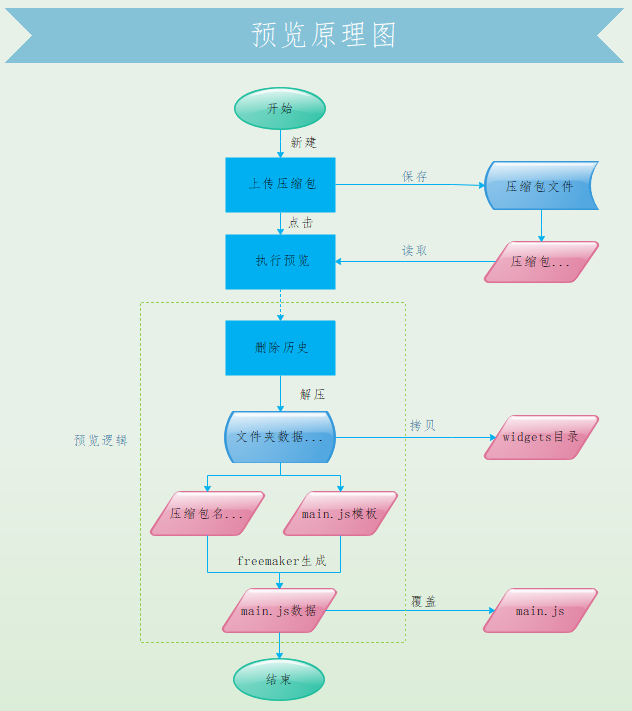


图22预览原理图

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> |

# 5、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。

# 6、参考文献

[1] 赵建保 JavaScript前端开发模块化教程 2019