# Javascript

## Js模块化开发

**一、原始写法**

模块就是实现特定功能的一组方法。

只要把不同的函数（以及记录状态的变量）简单地放在一起，就算是一个模块

function m1(){  
　　　　//...  
　　}

function m2(){  
　　　　//...  
　　}

上面的函数m1()和m2()，组成一个模块。使用的时候，直接调用就行了。

这种做法的缺点很明显："污染"了全局变量，无法保证不与其他模块发生变量名冲突，而且模块成员之间看不出直接关系。

**二、对象写法**

为了解决上面的缺点，可以把模块写成一个对象，所有的模块成员都放到这个对象里面。

var module1 = new Object({

　　　　\_count : 0,

　　　　m1 : function (){  
　　　　　　//...  
　　　　},

　　　　m2 : function (){  
　　　　　　//...  
　　　　}

　　});

上面的函数m1()和m2(），都封装在module1对象里。使用的时候，就是调用这个对象的属性。

module1.m1();

但是，这样的写法会暴露所有模块成员，内部状态可以被外部改写。比如，外部代码可以直接改变内部计数器的值。

module1.\_count = 5;

**三、立即执行函数写法**

使用"[立即执行函数](http://benalman.com/news/2010/11/immediately-invoked-function-expression/)"（Immediately-Invoked Function Expression，IIFE），可以达到不暴露私有成员的目的

var module1 = (function(){

　　　　var \_count = 0;

　　　　var m1 = function(){  
　　　　　　//...  
　　　　};

　　　　var m2 = function(){  
　　　　　　//...  
　　　　};

　　　　return {  
　　　　　　m1 : m1,  
　　　　　　m2 : m2  
　　　　};

　　})();

使用上面的写法，外部代码无法读取内部的\_count变量。

**四、放大模式**

如果一个模块很大，必须分成几个部分，或者一个模块需要继承另一个模块，这时就有必要采用"放大模式"（augmentation）。

var module1 = (function (mod){

　　　　mod.m3 = function () {  
　　　　　　//...  
　　　　};

　　　　return mod;

　　})(module1);

上面的代码为module1模块添加了一个新方法m3()，然后返回新的module1模块。

**五、宽放大模式（Loose augmentation）**

在浏览器环境中，模块的各个部分通常都是从网上获取的，有时无法知道哪个部分会先加载。如果采用上一节的写法，第一个执行的部分有可能加载一个不存在空对象，这时就要采用"宽放大模式"。

var module1 = ( function (mod){

　　　　//...

　　　　return mod;

　　})(window.module1 || {});

与"放大模式"相比，＂宽放大模式＂就是"立即执行函数"的参数可以是空对象。

**六、输入全局变量**

独立性是模块的重要特点，模块内部最好不与程序的其他部分直接交互。

为了在模块内部调用全局变量，必须显式地将其他变量输入模块。

var module1 = (function ($, YAHOO) {

　　　　//...

　　})(jQuery, YAHOO);

上面的module1模块需要使用jQuery库和YUI库，就把这两个库（其实是两个模块）当作参数输入module1。这样做除了保证模块的独立性，还使得模块之间的依赖关系变得明显。

**七、模块的规范**

先想一想，为什么模块很重要？

因为有了模块，我们就可以更方便地使用别人的代码，想要什么功能，就加载什么模块。

但是，这样做有一个前提，那就是大家必须以同样的方式编写模块，否则你有你的写法，我有我的写法，岂不是乱了套！考虑到Javascript模块现在还没有官方规范，这一点就更重要了。

目前，通行的Javascript模块规范共有两种：[CommonJS](http://wiki.commonjs.org/wiki/Modules/1.1)和[AMD](https://github.com/amdjs/amdjs-api/wiki/AMD)。我主要介绍AMD，但是要先从CommonJS讲起。

**八、CommonJS**

2009年，美国程序员Ryan Dahl创造了[node.js](http://nodejs.org/)项目，将javascript语言用于服务器端编程。

这标志"Javascript模块化编程"正式诞生。因为老实说，在浏览器环境下，没有模块也不是特别大的问题，毕竟网页程序的复杂性有限；但是在服务器端，一定要有模块，与操作系统和其他应用程序互动，否则根本没法编程。

node.js的[模块系统](http://nodejs.org/docs/latest/api/modules.html)，就是参照[CommonJS](http://wiki.commonjs.org/wiki/Modules/1.1)规范实现的。在CommonJS中，有一个全局性方法require()，用于加载模块。假定有一个数学模块math.js，就可以像下面这样加载。var math = require('math');

然后，就可以调用模块提供的方法：

var math = require('math');

　　math.add(2,3); // 5

因为这个系列主要针对浏览器编程，不涉及node.js，所以对CommonJS就不多做介绍了。我们在这里只要知道，require()用于加载模块就行了。

**九、浏览器环境**

有了服务器端模块以后，很自然地，大家就想要客户端模块。而且最好两者能够兼容，一个模块不用修改，在服务器和浏览器都可以运行。

但是，由于一个重大的局限，使得CommonJS规范不适用于浏览器环境。还是上一节的代码，如果在浏览器中运行，会有一个很大的问题，你能看出来吗？

　var math = require('math');

　　math.add(2, 3);

第二行math.add(2, 3)，在第一行require('math')之后运行，因此必须等math.js加载完成。也就是说，如果加载时间很长，整个应用就会停在那里等。

这对服务器端不是一个问题，因为所有的模块都存放在本地硬盘，可以同步加载完成，等待时间就是硬盘的读取时间。但是，对于浏览器，这却是一个大问题，因为模块都放在服务器端，等待时间取决于网速的快慢，可能要等很长时间，浏览器处于"假死"状态。

因此，浏览器端的模块，不能采用"同步加载"（synchronous），只能采用"异步加载"（asynchronous）。这就是AMD规范诞生的背景。

**十、AMD**

[AMD](https://github.com/amdjs/amdjs-api/wiki/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。

AMD也采用require()语句加载模块，但是不同于CommonJS，它要求两个参数：

require([module], callback);

第一个参数[module]，是一个数组，里面的成员就是要加载的模块；第二个参数callback，则是加载成功之后的回调函数。如果将前面的代码改写成AMD形式，就是下面这样：

require(['math'], function (math) {

　　　　math.add(2, 3);

　　});

math.add()与math模块加载不是同步的，浏览器不会发生假死。所以很显然，AMD比较适合浏览器环境。

目前，主要有两个Javascript库实现了AMD规范：[require.js](http://requirejs.org/)和[curl.js](https://github.com/cujojs/curl)。本系列的第三部分，将通过介绍require.js，进一步讲解AMD的用法，以及如何将模块化编程投入实战。

**十一、为什么要用require.js？**

最早的时候，所有Javascript代码都写在一个文件里面，只要加载这一个文件就够了。后来，代码越来越多，一个文件不够了，必须分成多个文件，依次加载。下面的网页代码，相信很多人都见过。

<script src="1.js"></script>  
　　<script src="2.js"></script>  
　　<script src="3.js"></script>  
　　<script src="4.js"></script>  
　　<script src="5.js"></script>  
　　<script src="6.js"></script>

这段代码依次加载多个js文件。

这样的写法有很大的缺点。首先，加载的时候，浏览器会停止网页渲染，加载文件越多，网页失去响应的时间就会越长；其次，由于js文件之间存在依赖关系，因此必须严格保证加载顺序（比如上例的1.js要在2.js的前面），依赖性最大的模块一定要放到最后加载，当依赖关系很复杂的时候，代码的编写和维护都会变得困难。

（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。

**十二、require.js的加载**

使用require.js的第一步，是先去官方网站[下载](http://requirejs.org/docs/download.html)最新版本。

下载后，假定把它放在js子目录下面，就可以加载了。

<script src="js/require.js"></script>

有人可能会想到，加载这个文件，也可能造成网页失去响应。解决办法有两个，一个是把它放在网页底部加载，另一个是写成下面这样：

<script src="js/require.js" **defer async="true"** ></script>

async属性表明这个文件需要异步加载，避免网页失去响应。IE不支持这个属性，只支持defer，所以把defer也写上。

加载require.js以后，下一步就要加载我们自己的代码了。假定我们自己的代码文件是main.js，也放在js目录下面。那么，只需要写成下面这样就行了：

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

data-main属性的作用是，指定网页程序的主模块。在上例中，就是js目录下面的main.js，这个文件会第一个被require.js加载。由于require.js默认的文件后缀名是js，所以可以把main.js简写成main。

**十三、主模块的写法**

上一节的main.js，我把它称为"主模块"，意思是整个网页的入口代码。它有点像C语言的main()函数，所有代码都从这儿开始运行。

下面就来看，怎么写main.js。

如果我们的代码不依赖任何其他模块，那么可以直接写入javascript代码。

require(['jquery', 'underscore', 'backbone'], function ($, \_, Backbone){

　　　　// some code here

　　});

require()函数接受两个参数。第一个参数是一个数组，表示所依赖的模块，上例就是['moduleA', 'moduleB', 'moduleC']，即主模块依赖这三个模块；第二个参数是一个回调函数，当前面指定的模块都加载成功后，它将被调用。加载的模块会以参数形式传入该函数，从而在回调函数内部就可以使用这些模块。

require()异步加载moduleA，moduleB和moduleC，浏览器不会失去响应；它指定的回调函数，只有前面的模块都加载成功后，才会运行，解决了依赖性的问题。

下面，我们看一个实际的例子。

假定主模块依赖jquery、underscore和backbone这三个模块，main.js就可以这样写：

require(['jquery', 'underscore', 'backbone'], function ($, \_, Backbone){

　　　　// some code here

　　});

require.js会先加载jQuery、underscore和backbone，然后再运行回调函数。主模块的代码就写在回调函数中。

**十四、模块的加载**

**上一节最后的示例中，主模块的依赖模块是['jquery', 'underscore', 'backbone']。默认情况下，require.js假定这三个模块与main.js在同一个目录，文件名分别为jquery.js，underscore.js和backbone.js，然后自动加载。**

**使用require.config()方法，我们可以对模块的加载行为进行自定义。require.config()就写在主模块（main.js）的头部。参数就是一个对象，这个对象的paths属性指定各个模块的加载路径。**

**require.config({**

**paths: {**

**"jquery": "jquery.min",**

**"underscore": "underscore.min",**

**"backbone": "backbone.min"**

**}**

**});**

**上面的代码给出了三个模块的文件名，路径默认与main.js在同一个目录（js子目录）。如果这些模块在其他目录，比如js/lib目录，则有两种写法。一种是逐一指定路径。**

**require.config({**

**paths: {**

**"jquery": "lib/jquery.min",**

**"underscore": "lib/underscore.min",**

**"backbone": "lib/backbone.min"**

**}**

**});**

**另一种则是直接改变基目录（baseUrl）。**

**require.config({**

**baseUrl: "js/lib",**

**paths: {**

**"jquery": "jquery.min",**

**"underscore": "underscore.min",**

**"backbone": "backbone.min"**

**}**

**});**

**如果某个模块在另一台主机上，也可以直接指定它的网址，比如：**

**require.config({**

**paths: {**

**"jquery": "https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min"**

**}**

**});**

**require.js要求，每个模块是一个单独的js文件。这样的话，如果加载多个模块，就会发出多次HTTP请求，会影响网页的加载速度。因此，require.js提供了一个优化工具，当模块部署完毕以后，可以用这个工具将多个模块合并在一个文件中，减少HTTP请求数。**

**十五、AMD模块的写法**

**require.js加载的模块，采用AMD规范。也就是说，模块必须按照AMD的规定来写。**

**具体来说，就是模块必须采用特定的define()函数来定义。如果一个模块不依赖其他模块，那么可以直接定义在define()函数之中。**

**假定现在有一个math.js文件，它定义了一个math模块。那么，math.js就要这样写：**

**// math.js**

**define(function (){**

**var add = function (x,y){**

**return x+y;**

**};**

**return {**

**add: add**

**};**

**});**

**加载方法如下：**

**// main.js**

**require(['math'], function (math){**

**alert(math.add(1,1));**

**});**

**如果这个模块还依赖其他模块，那么define()函数的第一个参数，必须是一个数组，指明该模块的依赖性。**

**define(['myLib'], function(myLib){**

**function foo(){**

**myLib.doSomething();**

**}**

**return {**

**foo : foo**

**};**

**});**

**当require()函数加载上面这个模块的时候，就会先加载myLib.js文件。**

**十六、加载非规范的模块**

**理论上，require.js加载的模块，必须是按照AMD规范、用define()函数定义的模块。但是实际上，虽然已经有一部分流行的函数库（比如jQuery）符合AMD规范，更多的库并不符合。那么，require.js是否能够加载非规范的模块呢？**

**回答是可以的。**

**这样的模块在用require()加载之前，要先用require.config()方法，定义它们的一些特征。**

**举例来说，underscore和backbone这两个库，都没有采用AMD规范编写。如果要加载它们的话，必须先定义它们的特征。**

**require.config({**

**shim: {**

**'underscore':{**

**exports: '\_'**

**},**

**'backbone': {**

**deps: ['underscore', 'jquery'],**

**exports: 'Backbone'**

**}**

**}**

**});**

**require.config()接受一个配置对象，这个对象除了有前面说过的paths属性之外，还有一个shim属性，专门用来配置不兼容的模块。具体来说，每个模块要定义（1）exports值（输出的变量名），表明这个模块外部调用时的名称；（2）deps数组，表明该模块的依赖性。**

**比如，jQuery的插件可以这样定义：**

**shim: {**

**'jquery.scroll': {**

**deps: ['jquery'],**

**exports: 'jQuery.fn.scroll'**

**}**

**}**

**十七、require.js插件**

**require.js还提供一系列插件，实现一些特定的功能。**

**domready插件，可以让回调函数在页面DOM结构加载完成后再运行。**

**require(['domready!'], function (doc){**

**// called once the DOM is ready**

**});**

**text和image插件，则是允许require.js加载文本和图片文件。**

**define([**

**'text!review.txt',**

**'image!cat.jpg'**

**],**

**function(review,cat){**

**console.log(review);**

**document.body.appendChild(cat);**

**}**

**);**

**类似的插件还有json和mdown，用于加载json文件和markdown文件。**

1. js模块化（seajs）

[SeaJS](http://seajs.com/)是一个遵循[CommonJS](http://wiki.commonjs.org/)规范的[**JavaScript**](http://lib.csdn.net/base/javascript)模块加载框架，**可以实现JavaScript的模块化开发及加载机制**。与[jQuery](http://jquery.com/)等[**JavaScript框架**](http://lib.csdn.net/base/angularjs)不同，SeaJS不会扩展封装语言特性，而只是实现JavaScript的模块化及按模块加载。SeaJS的主要目的是令JavaScript开发模块化并可以轻松愉悦进行加载，将前端工程师从繁重的JavaScript文件及对象依赖处理中解放出来，可以专注于代码本身的逻辑。SeaJS可以与[**jQuery**](http://lib.csdn.net/base/jquery)这类框架完美集成。使用SeaJS可以提高JavaScript代码的可读性和清晰度，解决目前JavaScript编程中普遍存在的依赖关系混乱和代码纠缠等问题，方便代码的编写和维护。

一、SeaJS模块化开发

<!DOCTYPE HTML>

<html lang="zh-CN">

<head>

<meta charset="UTF-8">

<title>TinyApp</title>

</head>

<body>

<p class="content"></p>

<script src="./sea.js"></script>

<script>

seajs.use('./init', function(init) {

init.initPage();

});

</script>

</body>

</html>

可以看到html页面不再需要引入所有依赖的js文件，而只是引入一个sea.js，sea.js会处理所有依赖，加载相应的js文件，加载策略可以选择在渲染页面时一次性加载所有js文件，也可以按需加载（用到时才加载响应js），具体加载策略使用方法下文讨论。

index.html加载了init模块，并使用此模块的initPage方法初始化页面数据，这里先不讨论代码细节。

# html

# java

# android

# 部署

# 数据库

# 命令

# 其他