**[前端模块化，AMD与CMD的区别](http://blog.csdn.net/jackwen110200/article/details/52105493)**

在[JavaScript](http://lib.csdn.net/base/javascript)发展初期就是为了实现简单的页面交互逻辑，寥寥数语即可；如今CPU、浏览器性能得到了极大的提升，很多页面逻辑迁移到了客户端（表单验证等），随着web2.0时代的到来，Ajax技术得到广泛应用，[jQuery](http://lib.csdn.net/base/jquery" \t "_blank" \o "jQuery知识库)等前端库层出不穷，前端代码日益膨胀

这时候JavaScript作为[嵌入式](http://lib.csdn.net/base/embeddeddevelopment" \t "_blank" \o "嵌入式开发知识库)的脚本语言的定位动摇了，JavaScript却没有为组织代码提供任何明显帮助，甚至没有类的概念，更不用说模块（module）了，JavaScript极其简单的代码组织规范不足以驾驭如此庞大规模的代码

## 模块

既然JavaScript不能handle如此大规模的代码，我们可以借鉴一下其它语言是怎么处理大规模程序设计的，在[Java](http://lib.csdn.net/base/javase" \t "_blank" \o "Java SE知识库)中有一个重要带概念——package，逻辑上相关的代码组织到同一个包内，包内是一个相对独立的王国，不用担心命名冲突什么的，那么外部如果使用呢？直接import对应的package即可

import java.util.ArrayList;   
遗憾的是JavaScript在设计时定位原因，没有提供类似的功能，开发者需要模拟出类似的功能，来隔离、组织复杂的JavaScript代码，我们称为模块化。

一个模块就是实现特定功能的文件，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。模块开发需要遵循一定的规范，各行其是就都乱套了

规范形成的过程是痛苦的，前端的先驱在刀耕火种、茹毛饮血的阶段开始，发展到现在初具规模，简单了解一下这段不凡的历程

**CommonJS(代表：NodeJS)（** Node.js其实是CommonJS的一个部分实现）

CommonJS是一种规范，NodeJS是这种规范的实现。

1、定义模块   
根据CommonJS规范，一个单独的文件就是一个模块。每一个模块都是一个单独的作用域，也就是说，在该模块内部定义的变量，无法被其他模块读取，除非定义为global对象的属性

2、模块输出（接口）：（自由变量，它包含模块希望提供给其它模块的对象）   
模块只有一个出口，module.exports对象，我们需要把模块希望输出的内容放入该对象

3、加载模块（模块引入）：   
加载模块使用require方法，该方法读取一个文件并执行，返回文件内部的module.exports对象

看个例子

//模块定义 myModel.js

var name = 'Byron';

function printName(){

console.log(name);

}

function printFullName(firstName){

console.log(firstName + name);

}

module.exports = {

printName: printName,

printFullName: printFullName

}

//加载模块,需要引入模块（其他js文件）的js文件，在nodejs命令窗口中打开观察效果（首先要安装nodejs）

var nameModule = require('./myModel.js');

nameModule.printName();

**注意** js文件名前面需要加上路径，可以是相对路径（相对于使用require方法的文件），也可以是绝对路径。如果省略路径，node.js会认为，你要加载一个核心模块，或者已经安装在本地 node\_modules 目录中的模块。如果加载的是一个目录，node.js会首先寻找该目录中的 package.json 文件，加载该文件 main 属性提到的模块，否则就寻找该目录下的 index.js 文件。

**尴尬的浏览器**

仔细看上面的代码，会发现require是同步的。模块系统需要同步读取模块文件内容，并编译执行以得到模块接口。

这在服务器端实现很简单，也很自然，然而， 想在浏览器端实现问题却很多。

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。

## AMD（代表：RequireJS）

AMD 即Asynchronous Module Definition，中文名是异步模块定义的意思。它是一个在浏览器端模块化开发的规范

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。

由于一个重大的局限，使得CommonJS规范不适用于浏览器环境。

RequrieJS其实就是AMD现在用的最广泛，最流行的实现

由于不是JavaScript原生支持，使用AMD规范进行页面开发需要用到对应的库函数，也就是大名鼎鼎RequireJS，实际上AMD 是 RequireJS 在推广过程中对模块定义的规范化的产出

requireJS主要解决两个问题

1、多个js文件可能有依赖关系，被依赖的文件需要早于依赖它的文件加载到浏览器   
2、js加载的时候浏览器会停止页面渲染，加载文件越多，页面失去响应时间越长   
看一个使用requireJS的例子

// 定义模块 myModule.js

define(['dependency'], function(){

var name = 'Byron';

function printName(){

console.log(name);

}

return {

printName: printName

};

});

// 加载模块

require(['myModule'], function (my){

　 my.printName();

});

**语法**

requireJS定义了一个函数 define，它是全局变量，用来定义模块

**define** 是AMD规范用来声明模块的接口，示例中的第一个参数是一个数组，表示当前模块的依赖。第二个参数是一个回调函数，表示此模块的执行体。只有当依赖数组中的所有依赖模块都是可用的时，AMD模块加载器（比如RequireJS）才会去执行回调函数并返回此模块的暴露接口。

注意，回调函数中参数的顺序与依赖数组中的依赖顺序一致。（即： **jquery** -> **$** ， **underscore** -> **\_** ）

当然，在这里我可以将回调函数的参数名称改成任何我们想用的可用变量名，这并不会对模块的声明造成任何影响。

除此之外，你不能在模块声明的外部使用 **$** 或者 **\_** ，因为他们只在模块的回调函数体中才有定义。

define(id?, dependencies?, factory);

1. id：可选参数，用来定义模块的标识，如果没有提供该参数，脚本文件名（去掉拓展名）
2. dependencies：是一个当前模块依赖的模块名称数组
3. factory：工厂方法，模块初始化要执行的函数或对象。如果为函数，它应该只被执行一次。如果是对象，此对象应该为模块的输出值   
   在页面上使用require函数加载模块

require([dependencies], function(){});   
require()函数接受两个参数

1. 第一个参数是一个数组，表示所依赖的模块
2. 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()函数在加载依赖的函数的时候是异步加载的，这样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。

## CMD（代表：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   
**语法**   
Sea.js 推崇一个模块一个文件，遵循统一的写法   
define(id?, deps?, factory)   
因为CMD推崇

1. 一个文件一个模块，所以经常就用文件名作为模块id
2. CMD推崇依赖就近，所以一般不在define的参数中写依赖，在factory中写

factory是一个函数，有三个参数，function(require, exports, module)

1. require 是一个方法，接受 模块标识 作为唯一参数，用来获取其他模块提供的接口：require(id)
2. exports 是一个对象，用来向外提供模块接口
3. 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

看个例子：

// 定义模块 myModule.js

define(function(require, exports, module) {

var $ = require('jquery.js')

$('div').addClass('active');

});

// 加载模块

seajs.use(['myModule.js'], function(my){

});

## AMD与CMD区别

关于这两个的区别网上可以搜出一堆文章，简单总结一下

最明显的区别就是在模块定义时对依赖的处理不同

**1、AMD推崇依赖前置，在定义模块的时候就要声明其依赖的模块**   
**2、CMD推崇就近依赖，只有在用到某个模块的时候再去require**   
这种区别各有优劣，只是语法上的差距，而且requireJS和SeaJS都支持对方的写法

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

为什么我们说两个的区别是依赖模块执行时机不同，为什么很多人认为ADM是异步的，CMD是同步的（除了名字的原因。。。）

同样都是异步加载模块，AMD在加载模块完成后就会执行改模块，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