**模块化开发**

**模块化：**

JS的由来就是为了实现页面的交互逻辑，随着CPU、浏览器性能的极大提升，很多的页面逻辑都迁移到了客户端（如表单验证），Ajax技术得到广泛应用，前端代码日益膨胀；JS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。既然JS无法驾驭海量的代码，那么可以借鉴Java的概念package，逻辑上相关的代码组织到同一个包内，包是一个相对独立的王国，不用担心命名冲突什么的，如果外部使用的话可以import对应的package即可。遗憾的是JS在设计时的定位原因，没有提供类似的功能，开发者需要模拟出来类似的功能，来隔离、组织复杂的JS代码，即成为模块化。

一个模块就是实现特定功能的文件，有了模块，就可以更方便的使用别人的代码，想要什么功能，就加在什么模块，模块开发需要遵循一定的规范，否则就乱套了。

最初的阶段就是函数封装，将实现特定逻辑的一组语句打包，而JS的作用域就是基于函数的，所以把函数作为模块化的第一步是很自然的事情，但是优缺点：污染了全局变量，无法保证不与其他模块发生变量名冲突，而且模块成员之间没什么关系。

进步到对象：为了解决上面的问题，对象的写法应运而生，可以把所有的模块成员封装在一个对象中，调用时引用对应文件然后调用对象的属性和方法。这样避免了变量污染，只要保证模块名唯一即可，同时同一模块内的成员也有了联系。但是这种方式也有缺陷，外部可以随意修改内部成员，这样就会产生意外的安全问题。可以通过立即执行函数（即把对象声明为一个自执行函数）来达到隐藏细节的目的，这样模块外部无法修改我们没有暴露出来的变量、函数。

上述做法就是我们模块化的基础，目前，同行的JS模块规范主要有两种：CommonJS和AMD

**CommonJS：**

因为网页端没有模块化编程，只是网页JS逻辑复杂，但也可以工作下去，在服务端却一定要有模块，所以虽然JS在Web端发展这么多年，第一个流行的模块化规范却是由服务端的JS应用带来的，CommonJS规范是有NodeJS发扬光大的，这标志着JS模块化编程正式登上舞台。

过程：定义模块→ 模块输出 → 加载模块

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，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。

模块输出：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。

加载模块：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。

Demo：

//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

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

nameModule.printName();

**AMD：**

Asynchronous Module Definition异步模块定义，它是一个在浏览器端模块开发的规范。由于不是JS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，也就是大名鼎鼎的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文件可能的依赖关系，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

**CMD：**

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加载时机上有所不同。

语法：

define（id？，deps？，function（require，exports，module）{ }）

require（id）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

exports 是一个对象，用来向外提供模块接口

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

Demo：

// 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和CMD区别：**

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