1. AMD概念

1、定义：

AMD规范全称是Asynchronous Module Definition，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。简单而优雅的方式统一了JavaScript的模块定义和加载机制

2、AMD语法：

作为一个规范，只需定义其语法API，而不关心其实现。AMD规范简单到只有一个API，即define函数：

* define([module-name?], [array-of-dependencies?], [module-factory-or-object]);

　　其中：

　　module-name: 模块标识，可以省略。

　　array-of-dependencies: 所依赖的模块，可以省略。

module-factory-or-object: 模块的实现，或者一个JavaScript对象。

第一个参数和第二个参数都是可以省略的，第三个参数则是模块的具体实现本身。

3、异步性

当define函数执行时，它首先会异步的去调用第二个参数中列出的依赖模块，当所有的模块被载入完成之后，如果第三个参数是一个回调函数则执行，然后告诉系统模块可用，也就通知了依赖于自己的模块自己已经可用。

* 与dojo对比，如果对应到dojo1.6之前的实现，那么在功能上可以有如下对应关系：

　　module-name: dojo.provide

　　dependencies: dojo.require

　　module-factory: dojo.declare

不同的是，在加载依赖项时，AMD用的是异步，而dojo.require是同步。异步和同步的区别显而易见，前者不会阻塞浏览器，有更好的性能和灵活性。而对于NodeJs这样的服务器端AMD，则模块载入无需阻塞服务器进程，同样提高了性能。

4、AMD实例

* 定义一个模块，具体代码：

define("alpha", ["require", "exports", "beta"], function (require, exports, beta) {

　　exports.verb = function() { //定义exports的verb方法

　　return beta.verb();//调用外部模块beta的verb方法

　　//或者:

　　return require("beta").verb();

　　}

});

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。

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。

5、匿名模块 – 省略第一个参数，第二个第三个不省略

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。

具体代码：

define(["alpha"], function (alpha) {

　　return {

　　verb: function(){ //return一个verb方法

　　return alpha.verb() + 2;

　　}

　};

});

6、仅有一个参数的define – 省略第一个第二个参数，第三个不省略

三个参数有两种情况，一种是一个JavaScript对象，另一种是一个函数。

* 一个JavaScript对象：

可能是一个包含方法具有功能的一个对象；也有可能是仅提供数据。后者和JSON-P非常类似，因此AMD也可以认为包含了一个完整的 JSON-P实现。模块演变为一个简单的数据对象，这样的数据对象是高度可用的，而且因为是静态对象，它也是CDN友好的，可以提高JSON-P的性能。

define({

　　provinces: [

　　{

　　name: '上海',

　　areas: ['浦东新区', '徐汇区']},

　　{

　　name: '江苏',

　　cities: ['南京', '南通']}

　　//.....

　　]

});

假设这个文件名为china.js，那么如果某个模块需要这个数据，只需要：

define(['china', function(china){

　　//在这里使用中国省市数据

　　});

通过这种方式，这个模块是真正高度可复用的，无论是用远程的，还是Copy到本地项目，都节约了开发时间和维护时间。

* 一个函数：

如果参数是一个函数，其用途之一是快速开发实现。适用于较小型的应用，你无需提前关注自己需要什么模块，自己给谁用。**在函数中，可以随时require自己需要的模块。**

define(function(){

var p = require('china');

//使用china这个模块

});

即你省略了模块名，以及自己需要依赖的模块。这不意味着你无需依赖于其他模块，而是可以让你在需要的时候去require这些模块。define方法在执行的时候，会调用函数的toString方法，并扫描其中的require调用，提前帮助你载入这些模块，载入完成之后再执行。这使得快速开发成为可能。

* 需要注意的一点是，Opera不能很好的支持函数的toString方法，因此，在浏览器中它的适用性并不是很强。但如果你是通过build工具打包所有的 JavaScript文件，这将不是问题，构建工具会帮助你扫描require并强制载入依赖的模块。

7、Dojo中的AMD

　　Dojo 的1.6版本，其中一个重要的变化就是引入了AMD机制，取代了原来的dojo.provide和dojo.require方法。但是现在仍然保持了向后兼容性，你仍然可以用dojo.provide和dojo.require来定义和加载模块。需要注意的是：在 Dojo 1.6 中， 针对 AMD 的重构仍然属于一个过渡期的改动 , 用户自己开发的 AMD 模块还不能被 Dojo 的加载器和 Build 系统支持 . 1.6 中现有的编译系统对AMD的支持还非常局限。 如果你自己开发了 AMD 格式的模块，并且你仍然在使用默认的 Dojo 同步模块加载器，那么你必须严格遵循 Dojo 模块的格式 ( 包括换行的格式 ) 来保证你自己的模块能够成功编译。总结起来有以下三点：

　　用传统的方法 (dojo.require()/dojo.provide()) – 这些模块，只能被 Dojo 同步加载器 加载，但可以被 Dojo 编译系统(Build System )正确的编译

　　用 Dojo 同步加载器来加载 AMD 格式 （ define ()） 模块 – 这些模块可以被正常的加载，并且可以被其他兼容 AMD 格式的加载器加载 . 现在虽然 Dojo1.6 还没有正式支持这种用法， 但在目前的 Dojo1.6 编译系统中，是可以正常工作的 . ( 前提是你必须严格遵循 Dojo 模块定义的代码规范 )

　　使用第三方加载器来加载 AMD 格式（ define ()）模块 – 模块可以被正常加载，并且可以被其他加载器所使用 . 这些模块可以使用 RequireJS 或 Backdraft 提供的编译系统正常编译，但是 Dojo 还没有正式的测试过和其他加载器的兼容性 .

　 以Calendar为例，用define方法来定义这个模块：

　　define("dijit/Calendar",

　　["dojo", "dijit", "text!dijit/templates/Calendar.html",

　　"dojo/cldr/supplemental", "dojo/date", "dojo/date/locale",

　　"dijit/\_Widget", "dijit/\_Templated", "dijit/\_CssStateMixin", "dijit/form/DropDownButton"],

　　function(dojo, dijit) {

　　dojo.declare(

　　"dijit.Calendar",

　　[dijit.\_Widget, dijit.\_Templated, dijit.\_CssStateMixin],

　　{...}

　　);

　　return dijit.Calendar;

　　}

　　);

　　可以看到，模块标识就是模块文件的路径，模块本身一般都是dojo.declare定义的类。Dojo1.6中的dojo和dijit命名空间下的模块均已经用AMD的形式进行了重构，但dojox下仍然延用了传统的dojo.provide和dojo.require形式。对AMD的引入是Dojo走向自动化包管理的重要一步，在后续文章中我们也将继续关注Dojo在这方面的进展。