**AMD 规范使用总结**

[](http://www.jianshu.com/u/4e938340e243)

[Northerner](http://www.jianshu.com/u/4e938340e243) 关注

2016.04.15 16:25\* 字数 2389 阅读 3390评论 2喜欢 3

**AMD模式**

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。

**define方法：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。  
按照是否依赖其他模块，可以分成两种情况讨论。第一种情况是定义独立模块，即所定义的模块不依赖其他模块；第二种情况是定义非独立模块，即所定义的模块依赖于其他模块。

**（1）AMD实例**

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。  
　　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。

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

　　exports.verb = function() {

　　 return beta.verb();

　　 //或者:

　　 return require("beta").verb();

　　}

});

**（2）匿名模块**

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。  
　　下面的代码就定义了一个依赖于alpha模块的匿名模块：

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

　　return {

　　 verb: function(){

　　 return alpha.verb() + 2;

　　 }

　　};

});

**（3）独立模块**

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。

define({

method1: function() {},

method2: function() {},

});

上面代码生成了一个拥有method1、method2两个方法的模块。另一种等价的写法是，把对象写成一个函数，该函数的返回值就是输出的模块。

define(function () {

return {

method1: function() {},

method2: function() {},

};

});

后一种写法的自由度更高一点，可以在函数体内写一些模块初始化代码。

值得指出的是，define定义的模块可以返回任何值，不限于对象。

**（4）非独立模块**

如果被定义的模块需要依赖其他模块，则define方法必须采用下面的格式。

define(['module1', 'module2'], function(m1, m2) {

...

});

define方法的第一个参数是一个数组，它的成员是当前模块所依赖的模块。比如，['module1', 'module2']表示我们定义的这个新模块依赖于module1模块和module2模块，只有先加载这两个模块，新模块才能正常运行。一般情况下，module1模块和module2模块指的是，当前目录下的module1.js文件和module2.js文件，等同于写成['./module1', './module2']。  
define方法的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，比如function(m1, m2)就表示，这个函数的第一个参数m1对应module1模块，第二个参数m2对应module2模块。这个函数必须返回一个对象，供其他模块调用。

define(['module1', 'module2'], function(m1, m2) {

return {

method: function() {

m1.methodA();

m2.methodB();

}

};

});

上面代码表示新模块返回一个对象，该对象的method方法就是外部调用的接口，menthod方法内部调用了m1模块的methodA方法和m2模块的methodB方法。

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。

define(

[ 'dep1', 'dep2', 'dep3', 'dep4', 'dep5', 'dep6', 'dep7', 'dep8'],

function(dep1, dep2, dep3, dep4, dep5, dep6, dep7, dep8){

...

}

);

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');

}

});

下面是一个define实际运用的例子。

define(['math', 'graph'],

function ( math, graph ) {

return {

plot: function(x, y){

return graph.drawPie(math.randomGrid(x,y));

}

}

};

);

上面代码定义的模块依赖math和graph两个库，然后返回一个具有plot接口的对象。  
另一个实际的例子是，通过判断浏览器是否为IE，而选择加载zepto或jQuery。

define(('\_\_proto\_\_' in {} ? ['zepto'] : ['jquery']), function($) {

return $;

});

上面代码定义了一个中间模块，该模块先判断浏览器是否支持proto属性（除了IE，其他浏览器都支持），如果返回true，就加载zepto库，否则加载jQuery库。

**require方法：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。这个数组可以写得很灵活，请看下面的例子。

require( [ window.JSON ? undefined : 'util/json2' ], function ( JSON ) {

JSON = JSON || window.JSON;

console.log( JSON.parse( '{ "JSON" : "HERE" }' ) );

});

上面代码加载JSON模块时，首先判断浏览器是否原生支持JSON对象。如果是的，则将undefined传入回调函数，否则加载util目录下的json2模块。  
require方法也可以用在define方法内部。

define(function (require) {

var otherModule = require('otherModule');

});

下面的例子显示了如何动态加载模块。

define(function ( require ) {

var isReady = false, foobar;

require(['foo', 'bar'], function (foo, bar) {

isReady = true;

foobar = foo() + bar();

});

return {

isReady: isReady,

foobar: foobar

};

});

上面代码所定义的模块，内部加载了foo和bar两个模块，在没有加载完成前，isReady属性值为false，加载完成后就变成了true。因此，可以根据isReady属性的值，决定下一步的动作。  
下面的例子是模块的输出结果是一个promise对象。

define(['lib/Deferred'], function( Deferred ){

var defer = new Deferred();

require(['lib/templates/?index.html','lib/data/?stats'],

function( template, data ){

defer.resolve({ template: template, data:data });

}

);

return defer.promise();

});

上面代码的define方法返回一个promise对象，可以在该对象的then方法，指定下一步的动作。  
如果服务器端采用JSONP模式，则可以直接在require中调用，方法是指定JSONP的callback参数为define。

require( [

"http://someapi.com/foo?callback=define"

], function (data) {

console.log(data);

});

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) {

// ...

};

**实际应用**

//定义M模块，本申明一个全局变量

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

window.M={};

return M;

})

//定义模块a 依赖模块 M,b,c

define('a',['M','b','c'],function(M){

alert(M.ob);

alert(M.oc);

})

//定义b模块

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

M.ob = 2;

return M;

})

//定义c模块

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

M.oc = 3;

return M;

})

//引入a模块

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

})

此例子里边需要注意:

M模块的作用是定义一个全局的变量；

a模块依赖了其他模块，但是function 参数缺只有M一个参数，但是也可以用使用b，c两个模块里边的变量，这是因为 M 模块里边的全局变量。可以看得出 b，c模块里边的变量在申明的时候就是挂在全局变量下的。所以a模块依赖了b，c模块缺没有使用参数接口，也可以用的原因。

反之，变量没有挂在全局变量下，引用了模块，没有用参数接受，这样是不可以使用其他模块下边的变量和方法的，因为很简单是作用局的问题。

还有就是b，c模块没有引用M模块缺使用了，M模块下边的变量，也是因为此变量是全局变量。如果是局部变量，那就要依赖于M模块才可以使用了。

假如说，反正b，c两个模块也不用参加接收，都挂在了全局变量下边，是不是不引也可以，直接引一个M模块好了。这肯定是不行的，会弹出 undefined