WebApp开发实战 第11章：Javascript模块化规范

作者：徐礼文 2015/8/21 7:10:11

一、前端js模块化由来与演变

> CommonJS 原来叫 ServerJS，推出 Modules/1.0 规范后，在 Node.js 等环境下取得了很不错的实践。09年下半年这帮充满干劲的小伙子们想把 ServerJS 的成功经验进一步推广到浏览器端，于是将社区改名叫 CommonJS，同时激烈争论 Modules 的下一版规范。分歧和冲突由此诞生，逐步形成了三大流派：

1. Modules/1.x 流派。这个观点觉得 1.x 规范已经够用，只要移植到浏览器端就好。要做的是新增 Modules/Transport 规范，即在浏览器上运行前，先通过转换工具将模块转换为符合 Transport 规范的代码。主流代表是服务端的开发人员。现在值得关注的有两个实现：越来越火的 component 和走在前沿的 es6 module transpiler。
2. Modules/Async 流派。这个观点觉得浏览器有自身的特征，不应该直接用 Modules/1.x 规范。这个观点下的典型代表是 AMD 规范及其实现 RequireJS。
3. Modules/2.0 流派。这个观点觉得浏览器有自身的特征，不应该直接用 Modules/1.x 规范，但应该尽可能与 Modules/1.x 规范保持一致。这个观点下的典型代表是 BravoJS 和 FlyScript 的作者。BravoJS 作者对 CommonJS 的社区的贡献很大，这份 Modules/2.0-draft 规范花了很多心思。 FlyScript 的作者提出了 Modules/Wrappings 规范，这规范是 CMD 规范的前身。可惜的是 BravoJS 太学院派，FlyScript 后来做了自我阉割，将整个网站（flyscript.org）下线了。

二、js模块化演变过程

1. 函数封装

```
function fn1(){
  statement
}

function fn2(){
  statement
}
```

> 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，而且模块成员之间没什么关系。

对象

```
var myModule = {
var1: 1,

var2: 2,

fn1: function(){

  },

fn2: function(){

  }
}
```

> 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关系
>
> 看似不错的解决方案，但是也有缺陷，外部可以随意修改内部成员
>
> myModel.var1 = 100; 这样就会产生意外的安全问题

立即执行函数

```
var myModule = (function(){
  var var1 = 1;
  var var2 = 2;

  function fn1(){

  }

  function fn2(){
```

```
  }

  return {
    fn1: fn1,
    fn2: fn2
  };
})();
```

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

CommonJS

> 一，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。
>
> 二，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。
>
> 三，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。

```
//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();
```

> 不同的实现对require时的路径有不同要求，一般情况可以省略js拓展名，可以使用相对路径，也可以使用绝对路径，甚至可以省略路径直接使用模块名（前提是该模块是系统内置模块）

浏览器问题

> 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。
>
> 浏览器端，加载JavaScript最佳、最容易的方式是在document中插入script 标签。但脚本标签天生异步，传统CommonJS模块在浏览器环境中无法正常加载。

解决思路之一是，开发一个服务器端组件，对模块代码作静态分析，将模块与它的依赖列表一起返回给浏览器端。 这很好使，但需要服务器安装额外的组件，并因此要调整一系列底层架构。

另一种解决思路是，用一套标准模板来封装模块定义，但是对于模块应该怎么定义和怎么加载，又产生的分歧：

AMD

> 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。
>
> 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

语法

requireJS定义了一个函数 define，它是全局变量，用来定义模块。

define(id?, dependencies?, factory);

> ------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）；
>
> ------dependencies：是一个当前模块依赖的模块名称数组
>
> ------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；

```
    // 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();
   });
```

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

> 第一个参数是一个数组，表示所依赖的模块
>
> 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()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

> 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

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

   define(id?, deps?, factory)

   //factory有三个参数

   function(require, exports, module)

> 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
>
> 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

```
// 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){

});
```

http://seajs.org/docs/

AMD与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
>
> 一、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
>
> 二、CMD推崇就近依赖，只有在用到某个模块的时候再去require
>
> 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

```
http://div.io/topic/430
```

RequireJs 实战

> RequireJS由James Burke创建，他也是AMD规范的创始人。
>
> RequireJS会让你以不同于往常的方式去写JavaScript。你将不再使用script标签在HTML中引入JS文件，以及不用通过script标签顺序去管理依赖关系。
>
> 当然也不会有阻塞（blocking）的情况发生。

Require.js安装与配置

1. 全局安装requirejs：使用r.js工具

```
bower install require.js
npm install requirejs -g
```

2. data-main属性

```
<script src="scripts/require.js" data-main="scripts/app.js"></script>
```

3. 配置函数（require.config({})）

- baseUrl——用于加载模块的根路径。
- paths——用于映射不存在根路径下面的模块路径。
- shims——配置在脚本/模块外面并没有使用RequireJS的函数依赖并且初始化函数。假设underscore并没有使用 RequireJS定义，但是你还是想通过RequireJS来使用它，那么你就需要在配置中把它定义为一个shim。
- deps——加载依赖关系数组

配置require 全局的配置

注意：paths: 是复数 path[s],没有s会报各种错误！

```
require.config({
//By default load any module IDs from scripts/app
baseUrl: 'scripts/app',
//except, if the module ID starts with "lib"
 paths: {
    lib: '../lib'
},
// load backbone as a shim
shim: {
    'backbone': {
        //The underscore script dependency should be loaded before loading backbone.js
```

```
        deps: ['underscore'],
        // use the global 'Backbone' as the module name.
        exports: 'Backbone'
    }
}
});
```

配置用于合并压缩的配置文件 : app.build.config.js

```
({
   baseUrl:"../js",
   dir:"../dist",
   mainConfigFile:"../js/main.js",
   name:"main"
})
```

合并压缩命令: r_js -o build/app.build.config.js