Javascript 高级模块化

在前面，我们已经学习了 javascript 和 jQuery，我们发现在一个正常的项目中，我们要在页面上导入大量 js 文件，jQuery 文件，jQuery 各种插件的文件，甚至是其他 javascript 库文件，这样大量的 javascript 文件的导入，使得我们的 javascript 异常的复杂、难以管理和维护，大大的增加了我们项目的管理和维护成本，那么有什么办法可以很友好的管理我们页面的各种各样的 js 代码吗？有，就是 js 的模块化管理。

Require.js

随着网址功能的日渐丰富，页面的 js 也变的越来越复杂和臃肿，原本通过 script 标签来导入 js 文件的方式已经不能满足我们的开发需求了，我们需要团队合作、模块复用、单元测试等等一系列复杂的需要，所以 Require.js 就应运而出了。

RequireJS 是一个非常小巧的 javascript 模块载入框架，是 AMD（Asynchronous Module Definition，异步模块加载机制）规范最好的实现之一。最新版的 requireJS 压缩后只有 14k，堪称非常轻量。它还同时可以和其他的框架协调工作，使用 requireJS 必将使我们的前端代码质量得以提升。

RequireJS 的好处

Requires 官方网站这样说的：（RequireJS is a JavaScript file and module loader. It is optimized for in-browser use, but it can be used in other JavaScript environments, like Rhino and Node. Using a modular script loader like RequireJS will improve the speed and quality of your code.）RequireJS 是一个 JavaScript 模块加载器。它非常适合在浏览器中使用,但它也可以用在其他脚本环境,就像 Rhino and Node. 使用 RequireJS 加载模块化脚本将提高代码的加载速度和质量。

RequireJS 的初识~~不胜风情

首先我们先使用一个普通的页面来看看：

```
<!DOCTYPE html>
```

```
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
        <script type="text/javascript" src="js/test01.js" ></script>
    </head>
    <body>
        <h1>Hello requireJS</h1>
    </body>
</html>
```

test01.js 文件：

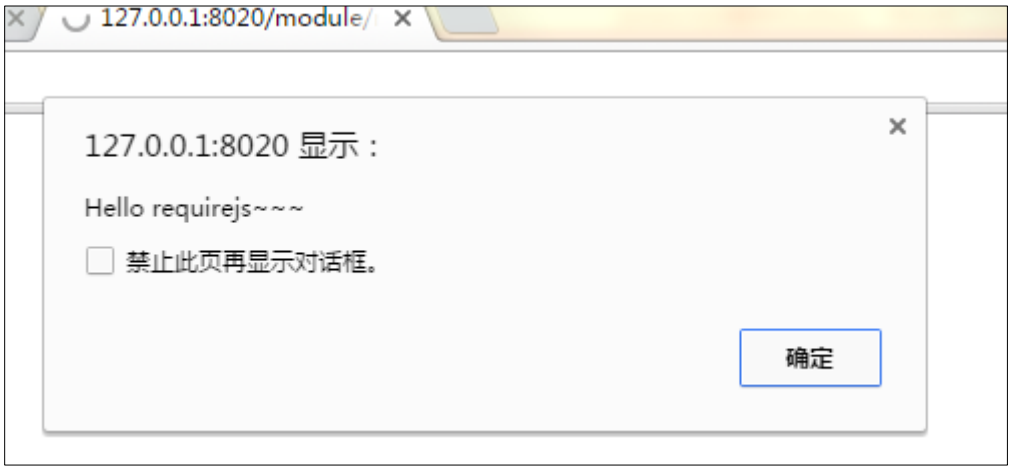
```
(function(){

    function fn1() {
        alert("Hello requirejs~~~");
    }

    fn1();
})();
```

使用闭包写了一个最简单的函数，弹出一句话，我们使用闭包的好处就是避免全局变量的出现，这样就防止了全局变量的污染。

在这段代码中，我们导入了一个 test01.js 文件，当我们打开页面时，是这个情况：



我们发现页面上的内容没有输出，js 弹出框在等待用户的点击后，页面才会继续运行，这样的结果我们不能接受，因为 js 一般和页面的加载无关，我们应该让页面继续加载，js 执行自己就行了，但是现在阻塞住了页面的继续加载，如果某个 js 文件报错，很有可能导致页面的正常加载和运行，这是我们不能接受的。

那么如果我们使用 requireJS 呢？首先我们来演示下，大家来看看：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
        <script type="text/javascript" src="js/require2.1.11.js"></script>
        <script type="text/javascript">
            require(["js/test02"]);
        </script>
    </head>
    <body>
        <h1>Hello requireJS</h1>
    </body>
</html>
```

test02.js 文件：

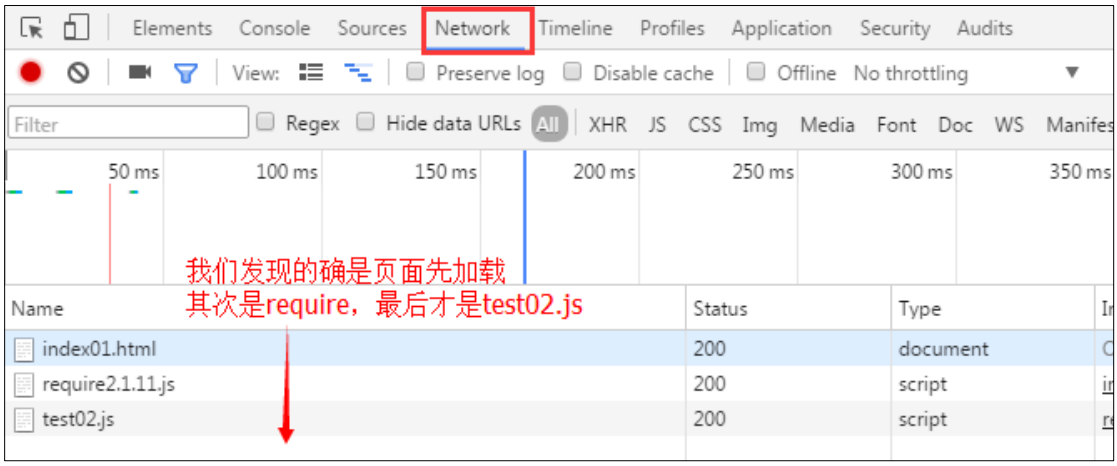
```
define(function(){

    function fn1() {
        alert("Hello requirejs~~~");
    }

    fn1();
});
```

我们看出，首先页面上不再导入 test02.js，只是导入了 requireJS，其次在 javascript 中我们使用 require()方法，在其中传递了一个数组的参数，实参为我们要导入的 js 文件的【路径+文件名称（后缀名可带可不带，不建议写，因为后面的网络访问时不能写，为了保持一致，不建议带）】，这样就完成了 requireJS 的使用，那么运行效果如何呢？

我们发现此时的页面加载已经完成了，并不像前面我们传统的一样在等待 js 运行完成后页面再加载，而是页面加载完成后才运行 js 代码，这样在运行效率上就大大的提高了。



回过头来我们看看 test02.js，我们发现和 test01.js 写法不同的是闭包—()();的写法改成了 define();在 define 函数中传递匿名函数完成操作，这个变化不大，大家应该很好能接受。

基本 API

requireJS 会定义三个变量：define、require、requirejs。

- require === requirejs，一般使用 require 更简短。
- define 从名字就可以看出这个 api 是用来定义一个模块。
- require 加载依赖模块，并执行加载完后的回调函数。

前面我们写的 test02.js：

```
define(function(){

    function fn1() {
        alert("Hello requirejs~~~");
    }

    fn1();
});
```

就是使用 define 来定义了一个模块，然后使用 require 来引用

```
<script type="text/javascript">
    require(["js/test02"]);//参数为一个数组
</script>
```

来加载该模块(注意 require 中的依赖是一个数组，即使只有一个依赖，你也必须使用数组来定义)，requir API 的第二个参数是 callback，一个回调函数，是用来处理加载完毕后的逻辑，如：

```
require(["js/test02.js"],function(){
    alert("js加载完成");
});
```

当所有的模块都加载完成后就会触发这个函数。同样存在了第三个参数，也是一个 callback 函数，这个是来处理加载失败后的情况的，如：

```
require(["js/test03"],function(){
    alert("js加载完成");
},function(){
    //当没有加载成功后就会触发
    //失败的话，在这个函数中处理
    alert("加载失败~~~");
});
```

此时 test03.js 不存在，页面肯定加载不成功，所以会触发第三个参数。

加载网络文件

之前的例子中加载模块都是本地 js，但是大部分情况下网页需要加载的 JS 可能来自本地服务器、其他网站或 CDN，这样就不能通过这种方式来加载了，我们以加载一个 jquery 库为例：

```
//百度cdn公共库jQuery地址：
http://apps.bdimg.com/libs/jquery/2.1.1/jquery.js
//jQuery官方地址：          https://code.jquery.com/jquery-3.1.1.js
//注意：网络上去取时不能加后缀，否则取不到
require.config({
    paths : {
        //为网络上的库去一个名字：jquery
        "jquery" : ["https://code.jquery.com/jquery-3.1.1"]
    }
});

//
require(["jquery","js/test01","js/test02"],function(){
    alert("页面加载成功~~");
},function(){
    alert("页面加载失败~~")
});
```

在取网络上的文件时注意：

1、 config 方法的参数是一个对象

2、 paths 的值也是一个对象

3、 当我们为网络上的库取名字是任意，但是建议取有意义的名字，别人可以通过名称知道你的网络资源是什么资源

4、 库的值是一个数组，意味着可以多个同时写，防止网络异常取不到

5、 特别注意：网络资源路径不能带后缀名，否则取不到

6、 我们也可以先让去网络中去取，如果取不到，再在本地取，减轻本地服务的压力（属于项目优化）。

```
require.config({
    paths : {
        //这样配置，减轻本地服务器的压力
        "jquery" : ["https://code.jquery.com/jquery-3.1.1.js","js/jquery-1.8.3"]
    }
});

//
require(["jquery","js/test01","js/test02"],function(){
    alert("页面加载成功~~");
},function(){
    alert("页面加载失败~~")
});
```

同样我们也可以将本地的配置到 paths 中：

```
require.config({
    paths : {
        //这样配置，减轻本地服务器的压力
        "jquery" : ["https://code.jquery.com/jquery-3.1.1","js/jquery-1.8.3"],
        //将本地的js文件同样配置，之后引用
        "test01" : ["js/test01"],
        "test02" : ["js/test02"]
    }
});

//
require(["jquery","test01","test02"],function(){
    alert("页面加载成功~~");
},function(){
    alert("页面加载失败~~")
});
```

全局配置

上面的例子中重复出现了 require.config 配置，如果每个页面中都加入配置，必然显得十分不雅，requirejs 提供了一种叫"主数据"的功能，我们首先创建一个 main.js：

```
require.config({
    paths : {
        //这样配置，减轻本地服务器的压力
        "jquery" : ["https://code.jquery.com/jquery-3.1.1","js/jquery-1.8.3"],
        //将本地的js文件同样配置，之后引用
        "test01" : ["js/test01"],
        "test02" : ["js/test02"]
    }
});
```

然后再页面中使用下面的方式来使用 requirejs：

```
<script type="text/javascript" src="js/require2.1.11.js" ></script>
<script type="text/javascript" src="js/main.js" ></script>
<script type="text/javascript">

    require(["jquery","t1","t2"],function(){
        alert("页面加载成功~~");
    },function(){
        alert("页面加载失败~~")
    });

</script>
```

在官方提供了一种基于标签属性的方式：

```
<script data-main="js/main" src="js/require2.1.11.js" ></script>
```

将所有的配置和导入 js 都放在了 main.js 中，这样在页面只要这样一个标签就行了。

第三方模块

通过 require 加载的模块一般都需要符合 AMD 规范即使用 define 来申明模块，但是部分时候需要加载非 AMD 规范的 js，这时候就需要用到另一个功能：shim，shim 解释起来也比较难理解，shim 直接翻译为"垫"，其实也是有这层意思的，目前我们主要用在两个地方

非 AMD 模块输出，将非标准的 AMD 模块"垫"成可用的模块，例如：在老版本的 jquery 中，是没有继承 AMD 规范的，所以不能直接 require["jquery"],这时候就需要 shim，比如我

要是用 underscore 类库，但是他并没有实现 AMD 规范，那我们可以这样配置：

```
require.config({
    shim: {
        "underscore" : {
            exports : "_";
        }
    }
})
```

这样配置后，我们就可以在其他模块中引用 underscore 模块：

```
require(["underscore"], function(_){
    _.each([1,2,3], alert);
})
```

插件形式的非 AMD 模块，我们经常会用到 jquery 插件，而且这些插件基本都不符合 AMD 规范，比如 jquery.form 插件，这时候就需要将 form 插件"垫"到 jquery 中：

```
require.config({
    shim: {
        "underscore" : {
            exports : "_";
        },
        "jquery.form" : {
            deps : ["jquery"]
        }
    }
})
```

也可以简写为：

```
require.config({
    shim: {
        "underscore" : {
            exports : "_";
        },
        "jquery.form" : ["jquery"]
    }
})
```

这样配置之后我们就可以使用加载插件后的 jquery 了

```
require.config(["jquery", "jquery.form"], function($){
    $(function(){
        $("#form").ajaxSubmit({...});
    })
})
```

其他内容

在回调函数中，我们可以再对应的参数得到加载的 js 对象，如第一个就是 jQuery 对象，第二个和第三个我们自己的 js 中没有对象，所以为 undefined。

```
<script type="text/javascript" src="js/require2.1.11.js" ></script>
<script type="text/javascript" src="js/main.js" ></script>
<script type="text/javascript">

    require(["jquery","t1","t2"],function($,t1,t2){
        /**
         * 我们在回调函数中可以传递参数，这些参数就是
         * require方法中的js对象，如jQuery对象
         */
        alert($("body").html());
    },function(){
        alert("页面加载失败~~")
    });

</script>
```

还是就是在 define 函数中同样可以传递一个数组参数，这个数组参数就是在前面我们已经 config 过的 js 库或者我们本地的 js 文件，如：

```
//将要使用的库可以再这儿引入
define(["jquery"],function($){

    function fn1() {
        alert("Hello requirejs~~~***");
    }
    //alert($(window).scrollTop());
    fn1();
});
```

这样我们在 define 中制作 jQuery 的插件的话，就可以直接使用了

Sea.js

SeaJS 是一个遵循 CommonJS 规范的 JavaScript 模块加载框架，可以实现 JavaScript 的模块化开发及加载机制。与 jQuery 等 JavaScript 框架不同，SeaJS 不会扩展封装语言特性，而只是实现 JavaScript 的模块化及按模块加载。SeaJS 的主要目的是令 JavaScript 开发模块化并可以轻松愉悦进行加载，将前端工程师从繁重的 JavaScript 文件及对象依赖处理中解放出来，可以专注于代码本身的逻辑。SeaJS 可以与 jQuery 这类框架完美集成（骗人的~~）。使用 SeaJS 可以提高 JavaScript 代码的可读性和清晰度，解决目前 JavaScript 编程中普遍存在的依赖关系混乱和代码纠缠等问题，方便代码的编写和维护。SeaJS 的作者是淘宝前端工程师玉伯。

对于前端们来说，后端的一些语言非常让我们感觉到和蔼可亲，没有浏览器的烦恼，可以随便 include、随便 require，又有 class，用起来非常的爽。我们羡慕了这么久，前端的发展也正从这条路上熙熙攘攘的走来，各种思想和潮流蜂拥而至，不过是 js 还是 css 都非常革命性的的影响了我们。

seaJS 的初识

定义一个 js 文件 test01.js，这样写：

```
define(function(require,exports,module){

    alert("hello seajs~~~");
});
```

在页面上引入我们定义的 js 文件，注意在之前一定要先引入 sea.js。

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title></title>
        <script type="text/javascript" src="js/sea.js" ></script>
        <script type="text/javascript">

            //注意，路径问题，seajs.user()中第一个参数是一个数组
            //只有一个是可以省略数组，但是路径是相当于sea.js路径，
            //而不是相对于页面路径
            seajs.use("test01",function() {
                alert("加载完成");
```

```
        })

    </script>

</head>
<body>
    <h1>seajs Demo</h1>
</body>
</html>
```

当然回调函数可以不写。

```
seajs.use("test01");
```

工厂函数 factory 解析

在我们自定义的 js 文件中，我们使用 define 来定义一个模块，define 方法和 require 中的一样，第一个参数可以传递一个模块，当然也可以不写（可选），第二个参数是一个匿名函数，不同于 requireJS 的是 seaJS 的匿名函数存在三个参数，这三个参数有不同的意义，建议不要省略，并且名称也不建议修改（形参可以修改名称的嘛），下面我们来看看这三个参数的含义：

1、 require——模块加载函数，用于记载依赖模块。

2、 exports——接口点，将数据或方法定义在其上则将其暴露给外部调用。

3、 module——模块的元数据。

test02.js：

```
define(function(require,exports,module){

    var date1 = "哈哈，我是test02.js中的变量date1，我是私有的";
    //exports定义的变量是公共，在其他模块中可以调用
    exports.date2 = "哈哈，我是test02.js中的变量date2，我是公共的";

    var fn1 = function() {
        alert("哈哈，我是test02.js的函数fn1，我是私有的");
    }

    fn1();

    var fn2 = function() {
```

```
        alert("哈哈，我是test02.js的函数fn2，我是私有的");
    }

    //exports定义的函数（方法）是公共，在其他模块中可以调用
    exports.fn3 = function() {
        alert("哈哈，我是test02.js的函数fn3，我是公共的");
    }
});
```

test03.js：

```
define(function(require,exports,module){

    //require用于导入所需的模块
    //载入a模块，我们可能在这个文件中使用test02.js
    var test02 = require("test02");
    alert("--------------------------------");
    /**
     * 虽然变量也是使用exports声明的
     * 但是变量的话，只有在seajs.use方法中可以调用
     * 在其他模块中照样不能调用，注意！！
     */
    alert(test02.data2);
    /**
     * 在其他模块（test03.js）中，可以调用
     * test02.js中的exports声明的方法
     * 方法一旦被exports声明，代表着公开
     */
    test02.fn3()
    alert("--------------------------------");

    var date1 = "哈哈，我是test03.js中的变量date1，我是私有的";
    //exports定义的变量是公共，在seajs.use方法中可以调用
    exports.date2 = "哈哈，我是test03.js中的变量date2，我是公共的";

    var fn1 = function() {
        alert("哈哈，我是test03.js的函数fn1，我是私有的");
    }

    var fn2 = function() {
        alert("哈哈，我是test03.js的函数fn2，我是私有的");
    }

    //exports定义的函数（方法）是公共，在其他模块中可以调用
    exports.fn3 = function() {
```

```
        alert("哈哈，我是test03.js的函数fn3，我是公共的");
    }

    fn1();
    fn2();
});
```

页面：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title></title>
        <script type="text/javascript" src="js/sea.js" ></script>
        <script type="text/javascript">
            seajs.use("test03",function(t3) {
//              t3.fn1();//报错，因为没有公共化
//              t3.fn2();//报错，因为没有公共化
                t3.fn3();//正确，因为使用了exports声明，此时这个方法公共化
                //在这里可以调用使用exports声明的变量
                alert(t3.date2)
            });

        </script>
    </head>
    <body>
        <h1>seajs Demo</h1>
    </body>
</html>
```

模块的载入和引用

模块的寻址算法

一个模块对应一个 js 文件，而载入模块时一般都是提供一个字符串参数告诉载入函数需要的模块，所以就需要有一套从字符串标识到实际模块所在文件路径的解析算法。SeaJS 支持如下标识：

绝对地址——给出 js 文件的绝对路径。如:

```
require("http://example/js/a");
```

就代表载入 http://example/js/a.js 。

相对地址——用相对调用载入函数所在 js 文件的相对地址寻找模块。例如在 http://example/js/b.js 中载入：

```
require("./c");
```

则载入 http://example/js/c.js 。

基址地址——如果载入字符串标识既不是绝对路径也不是以"./"开头，则相对 SeaJS 全局配置中的"base"来寻址，这种方法稍后讨论。注意上面在载入模块时都不用传递后缀名".js"，SeaJS 会自动添加".js"。但是下面三种情况下不会添加：

载入 css 时，如:

```
require("./style.css");
```

路径中含有"?"时，如:

```
require(<a href="http://example/js/a.json?cb=func">http://example/js/a.json?cb=func</a>);
```

路径以"#"结尾时，如:

```
require("http://example/js/a.json#");
```

根据应用场景的不同，SeaJS 提供了三个载入模块的 API，分别是 seajs.use，require 和 require.async。

其他两个咱们都用过了，没什么可说的，seajs.use 方法就是主载入口，seaJS 就是从这里开始运行的。require 方法在模块中调用模块时使用。那么 require.async 是 SeaJS 会在 html 页面打开时通过静态分析一次性记载所有需要的 js 文件，如果想要某个 js 文件在用到时才下载，可以使用 require.async：

```
require.async('/path/to/module/file', function(m) {
    //code of callback...
});
```

这样只有在用到这个模块时，对应的 js 文件才会被下载，也就实现了 JavaScript 代码的按需加载。

SeaJS 的全局配置

SeaJS 提供了一个 seajs.config 方法可以设置全局配置，接收一个表示全局配置的配置对象。具体使用方法如下：

```
seajs.config({
    base: 'path/to/jslib/',
    alias: {
      'app': 'path/to/app/'
    },
    charset: 'utf-8',
    timeout: 20000,
    debug: false
});
```

其中 base 表示基址寻址时的基址路径。例如 base 设置为 http://example.com/js/3-party/ ，则:

```
var $ = require('jquery');
```

会载入 http://example.com/js/3-party/jquery.js 。

- alias 可以对较长的常用路径设置缩写。
- charset 表示下载 js 时 script 标签的 charset 属性。
- timeout 表示下载文件的最大时长，以毫秒为单位。
- debug 表示是否工作在调试模式下。

与 jQuery 的整合

要将现有 JS 库如 jQuery 与 SeaJS 一起使用，只需根据 SeaJS 的的模块定义规则对现有库进行一个封装。例如，下面是对 jQuery 的封装方法：

```
define(function() {

    //jQuery原有代码开始
    jQuery原有代码结束

    return $.noConflict();
});
```

就是将 jQuery 的源代码包裹在一个定义的模块中，最后加入一个 return $.noConflict(); 就可以了。

Sea 的配置文件

https://github.com/seajs/seajs/issues/262