RequireJs

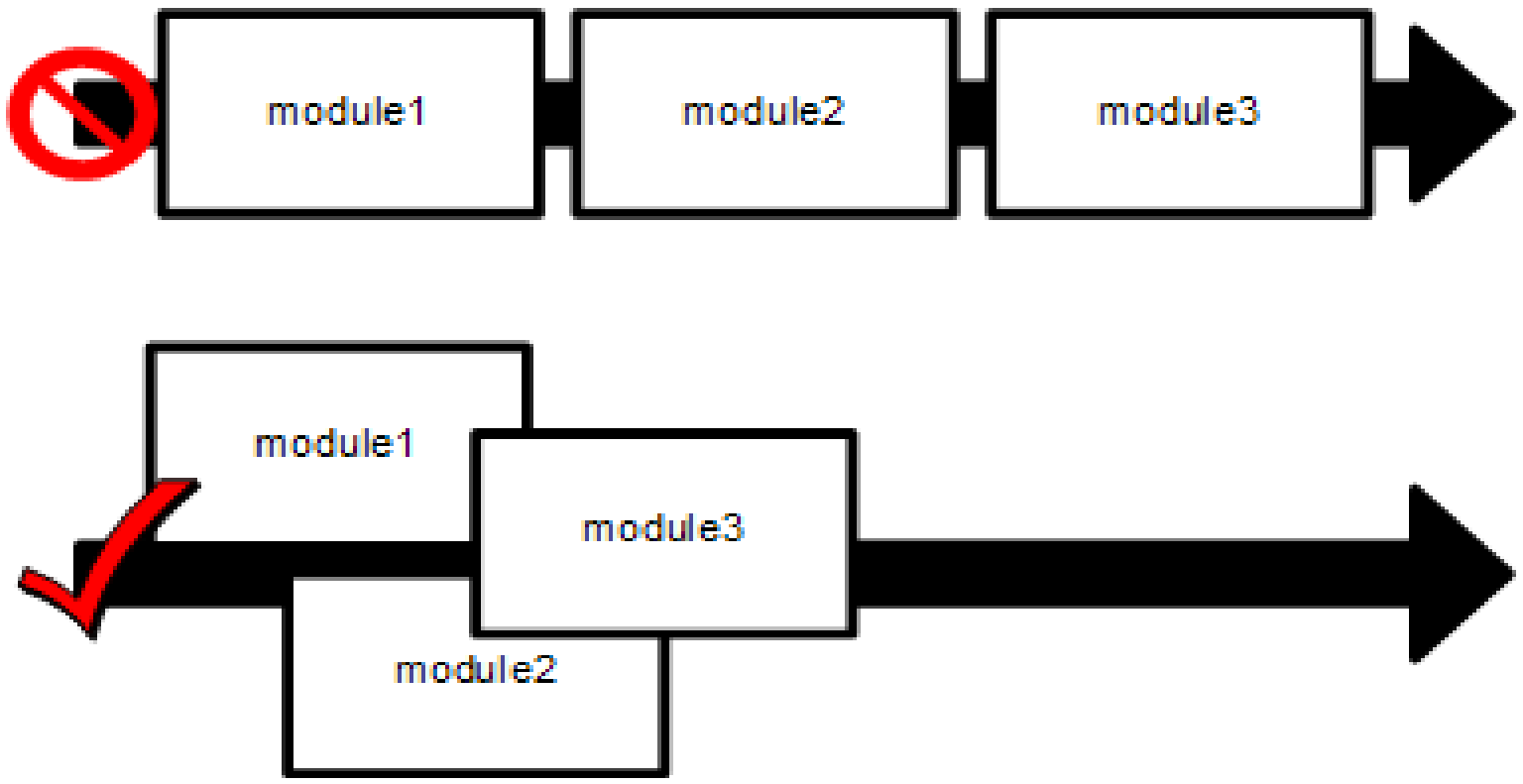
了解异步模块定义（AMD）

JavaScript 模块只是遵循 SRP(Single Responsibility Principle 单一职责原则）的代码段，它暴露了一个公开的 API 。在现今 JavaScript 开发中，你可以在模块中封装许多功能，而且在大多数项目中，每个模块都有其自己的文件。这使得 JavaScript 开发难度加大，因为它们需要持续不断的关注模块之间的依赖性，按照一个特定的顺序加载这些模块，否则运行时将会发生错误。

当你要加载 JavaScript 模块时，就会使用 script 标签。为了加载依赖的模块，你就要先加载被依赖的，之后再加载依赖的。使用 script 标签时，你需要按照此特定顺序安排它们的加载，而且脚本的加载是同步的。可以使用 async 和 defer 关键字使得加载异步，但可能因此在加载过程中丢失加载的顺序。另一个选择是将所有的脚本捆绑打包在一起，但在捆绑的时候你仍然需要把它们按照正确的顺序排序。

AMD 就是这样一种对模块的定义，使模块和它的依赖可以被异步的加载，但又按照正确的顺序。如下图形象地展现了文件同步加载和异步加载的区别。

Asynchronous Module Definition (AMD)



RequireJs 简介

RequireJS 是一个 Javascript 文件和模块框架，可以从 http://requirejs.org/ 下载，使用 RequireJS, 你可以顺序读取仅需要相关依赖模块。 RequireJS 所做的是，在你使用 script 标签加载你所定义的依赖时，将这些依赖通过 head.appendChild() 函数来加载他们。当依赖加载以后， RequireJS 计算出模块定义的顺序，并按正确的顺序进行调用。这意味着你需要做的仅仅是使用一个“根”来读取你需要的所有功能，然后剩下的事情只需要交给 RequireJS 就行了。为了正确的使用这些功能，你定义的所有模块都需要使用 RequireJS 的 API，否者它不会像期望的那样工作。

RequireJS API 存在于 RequireJS 载入时创建的命名空间 requirejs 下。其主要 API 主要是下面三个函数：

define– 该函数用户创建模块。每个模块拥有一个唯一的模块 ID，它被用于 RequireJS 的运行时函数， define 函数是一个全局函数，不需要使用 requirejs 命名空间 .

require– 该函数用于读取依赖。同样它是一个全局函数，不需要使用 requirejs 命名空间 .

config– 该函数用于配置 RequireJS.

□ 用背景

最早的时候，所有 Javascript 代码都写在一个文件里面，只要加载一个文件就够了。后来，代码越来越多，一个文件不够了，必须分成多个文件，依次加载。如下加载方式需要按顺序添加 js 文件。

```
<script src="1.js"></script>
<script src="2.js"></script>
<script src="3.js"></script>
```

这段代码依次加载多个 js 文件。这样的写法有很大的缺点。首先，加载的时候，浏览器会停止网页渲染，加载文件越多，网页失去响应的时间就会越长；其次，由于 js 文件之间存在依赖关系，因此必须严格保证加载顺序（比如上例的 1.js 要在 2.js 的前面），依赖性最大的模块一定要放到最后加载，当依赖关系很复杂的时候，代码的编写和维护都会变得困难。

required.js 能很好地解决 js 文件之间的依赖问题。

(1) 实现 js 文件的异步加载，避免网页失去响应；

(2) 管理模块之间的依赖性，便于代码的编写和维护。

RequireJs 的加载

如果想使用 Require.js，首先下载 require.js。下载后，假定把它放在 js 子目录下面，就可以加载了。<script src="js/require.js"></script>。加载这个文件可能造成网页失去响应。解决办法有两个，一个是把它放在网页底部加载，另一个是写成下面这样：<script src="js/require.js" defer async="true" ></script>async 属性表明这个文件需要异步加载，避免网页失去响应。IE 不支持这个属性，只支持 defer，所以把 defer 也写上。

加载 require.js 以后，下一步就要加载我们自己的代码了。假定我们自己的代码文件是 main.js，也放在 js 目录下面。那么，只需要写成下面这样就行了<script src="js/require.js" data-main="js/main"></script>

data-main 属性的作用是，指定网页程序的主模块。在上例中，就是 js 目录下面的 main.js

RequireJs 依赖加载

RequireJs 如果依赖其他模块可以使用以下代码进行 js 文件的加载。require(['moduleA', 'moduleB', 'moduleC'], function (moduleA, moduleB, moduleC){// some code here});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，上例就是['moduleA', 'moduleB', 'moduleC']，即主模块依赖这三个模块；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() 异步加载 moduleA， moduleB 和 moduleC，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。

例子：

假定主模块依赖 jquery 、 underscore 和 backbone 这三个模块， main.js 就可以这样写：

```
Require(['jquery', 'underscore', 'backbone'], function ($, _, Backbone){// some code here});
```

require.js 会先加载 jQuery 、 underscore 和 backbone ， 然后再运行回调函数。主模块的代码就写在回调函数中。

AMD 模块的定义

require.js 加载的模块，采用 AMD 规范。也就是说，模块必须按照 AMD 的规定来写。具体来说，就是模块必须采用特定的 define() 函数来定义。如果一个模块不依赖其他模块，那么可以直接定义在 define() 函数之中。假定现在有一个 math.js 文件，它定义了一个 math 模块。那么， math.js 就要这样写：

```
// math.js
define(function (){
    var add = function (x,y){
        return x+y;
    };
    return {
        add: add
    };
});
```

如果这个模块还依赖其他模块，那么 define() 函数的第一个参数，必须是一个数组，指明该模块的依赖性。

```
define(['module1','module2','module3',...],function('module1','module2','module3',...){
        function foo1(){
            module1.doSomething();
        }
        function foo2(){
            module2.doSomething();
        }
        return {
            foo1 : foo1,
            foo2 : foo2
        };
    });
```

当 require() 函数加载上面这个模块的时候，就会先加载他依赖的 js 文件。在 requireJS 中，通过向全局变量注册 define 函数来声明一个模块。在定义模块时，我们要遵循一下的规范：

1) 一个 JavaScrip 文件即为一个模块，即一个 JavaScrip 文件只能定义一个 define 函数。

2)RequireJS 最佳实践推荐，定义模块时不要指定模块标识。这样方便后期压缩。

3)RequireJS 最佳实践推荐推行尽量将代码封装到 define 函数里面。尽量不要使用 shim 配置项。

RequireJs 兼容性

IE 6+ ..... compatible ✔
Firefox 2+ ..... compatible ✔
Safari 3.2+ .... compatible ✔
Chrome 3+ ..... compatible ✔
Opera 10+ ..... compatible ✔

配置函数 config

如果你想改变 RequireJS 的默认配置来使用自己的配置，你可以使用 require.configh 函数。 config 函数需要传入一个可选参数对象，这个可选参数对象包括了许多的配置参数选项。下面是一些你可以使用的配置：
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

以下通过例子来介绍 config 函数使用的例子。

假如有一个 main.js 主模块的依赖模块是 ['jquery', 'underscore', 'backbone']。默认情况下， require.js 假定这三个模块与 main.js 在同一个目录，文件名分别为 jquery.js ， underscore.js 和 backbone.js ，然后自动加载。

使用 require.config() 方法，我们可以对模块的加载行为进行自定义。 require.config() 就写在主模块（ main.js ）的头部。参数就是一个对象，这个对象的 paths 属性指定各个模块的加载路径。

```
require.config({
    paths: {
        "jquery": "jquery.min",
        "underscore": "underscore.min",
        "backbone": "backbone.min"
    }
});
```

上面的代码给出了三个模块的文件名，路径默认与 main.js 在同一个目录（ js 子目录）。如果这些模块在其他目录，比如 js/lib 目录，则有两种写法。一种是逐一指定路径。

```
require.config({
    paths: {
        "jquery": "lib/jquery.min",
        "underscore": "lib/underscore.min",
        "backbone": "lib/backbone.min"
    }
});
```

另一种则是直接改变基目录（baseUrl）。

```
    require.config({
        baseUrl: "js/lib",
        paths: {
            "jquery": "jquery.min",
            "underscore": "underscore.min",
            "backbone": "backbone.min"
        }
    });
```

如果某个模块在另一台主机上，也可以直接指定它的网址，比如：

```
    require.config({
        paths: {
            "jquery":
"https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min"
        }
    });
```

require.js 要求，每个模块是一个单独的 js 文件。这样的话，如果加载多个模块，就会发出多次 HTTP 请求，会影响网页的加载速度。因此， require.js 提供了一个优化工具，当模块部署完毕以后，可以用这个工具将多个模块合并在一个文件中，减少 HTTP 请求数。

在 config 函数中对于不符合 AMD 规范的 js 库，我们可以通过使用 config 函数中的 shim 参数来配置使得 requireJs 可以使用非标准的 js 库。这样的模块在用 require() 加载之前，要先用 require.config() 方法，定义它们的一些特征。举例来说， underscore 和 backbone 这两个库，都没有采用 AMD 规范编写。如果要加载它们的话，必须先定义它们的特征。

```
require.config({
shim: {

'underscore':{
exports: '_'
},
'backbone': {
deps: ['underscore', 'jquery'],
exports: 'Backbone'
}
}
});
```

require.config() 接受一个配置对象，这个对象除了有前面说过的 paths 属性之外，还有一个 shim 属性，专门用来配置不兼容的模块。具体来说，每个模块要定义（1） exports 值（输出的变量名），表明这个模块外部

调用时的名称；（２） deps 数组，表明该模块的依赖性。
比如， jQuery 的插件可以这样定义：

```
shim: {
'jquery.scroll': {
deps: ['jquery'],
exports: 'jQuery.fn.scroll'
  }
}
```

js 合并压缩

需在目录中放置 r.js, 同时有一个合并规则配置文件
例如 build.js:

```
({
    appDir: './',
    baseUrl: './js',
    dir: './dist',
    modules: [
        {
            name: 'config'              // □ 面中 data-main 引入文件
            exclude:['other/b']         // 不合并压缩的文件
        },{
            name: 'config2'
        },{
            name: 'main'
        }
    ],
```

```
fileExclusionRegExp: /^(r|build)\.js$/,
    optimizeCss: 'standard',
    removeCombined: true,
    paths: {
        a:'empty:',      //empty:指不被压缩合并，单独请求
        b:'other/b',
        c:'some/c',
        d:'some/d',
        e:'other/e',
        f:'other/f'
    }
})
```

r.js 依赖于 nodejs 环境，使用方法：进入文件所在目录，运行 node r.js -o build.js

Thank you !