RequireJS 入门指南

如今最常用的 JavaScript 库之一是 RequireJS。最近我参与的每个项目，都用到了 RequireJS，或者是我向它们推荐了增加 RequireJS。在这篇文章中，我将描述 RequireJS 是什么，以及它的一些基础场景。

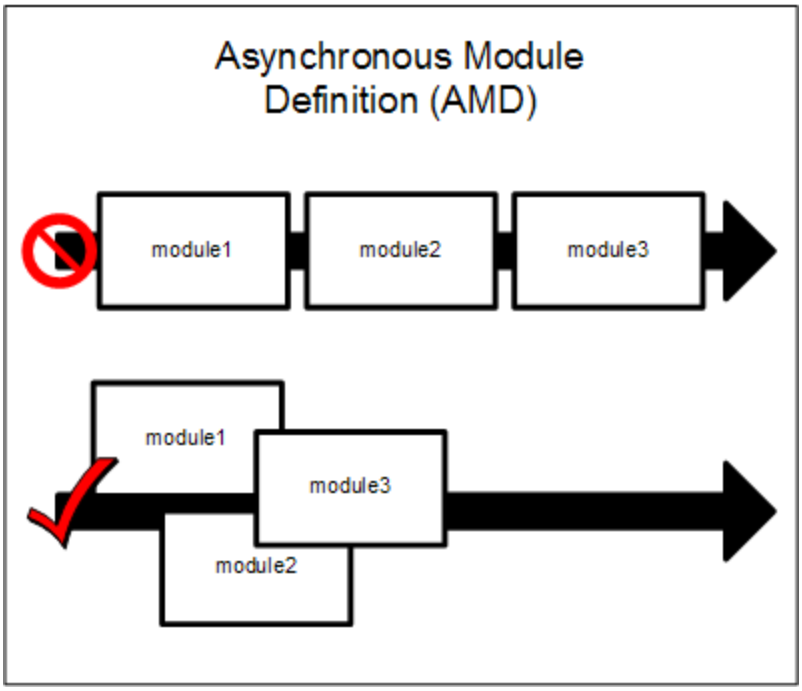
异步模块定义(AMD)

谈起 RequireJS，你无法绕过提及 JavaScript 模块是什么，以及 AMD 是什么。

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 开发者日子有点难过，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，否则运行时将会放生错误。

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。



CommonJS，是对通用的 JavaScript 模式的标准化尝试，它包含有 **AMD 定义** ，我建议你在继续本文之前先读一下。在 ECMAScript 6 这个下一版本 JavaScript 规范中，有关于输出，输入以及模块的规范定义，这些将成为 JavaScript 语言的一部分，而且这不会太久。这也是关于 RequireJS 我们想说的东西。

RequireJS?

RequireJS 是一个 Javascript 文件和模块框架，可以从 http://requirejs.org/下载，如果你使用 Visual Studio 也可以通过 Nuget 获取。它支持浏览器和像 node.js 之类的服务器环境。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.

在后面，我们将教你如果使用这些函数，但首先让我们先了解下 RequireJS 的加载流程。

data-main 属性

当你下载 RequireJS 之后，你要做的第一件事情就是理解 RequireJS 是怎么开始工作的。当 RequireJS 被加载的时候，它会使用 data-main 属性去搜寻一个脚本文件（它应该是与使用 src 加载 RequireJS 是相同的脚本）。data-main 需要给所有的脚本文件设置一个根路径。根据这个根路径，RequireJS 将会去加载所有相关的模块。下面的脚本是一个使用 data-main 例子：

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

另外一种方式定义根路劲是使用配置函数，后面我们将会看到。requireJs 假设所有的依赖都是脚本，那么当你声明一个脚本依赖的时候你不需要使用.js 后缀。

配置函数

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

下面是使用配置的一个例子：

```
01 require.config({
02         //By default load any module IDs from scripts/app
03         baseUrl:  'scripts/app',
04         //except, if the module ID starts with "lib"
05           paths: {
06                 lib:  '../lib'
07         },
08         // load backbone as a shim
09         shim: {
10                 'backbone': {
                           //The underscore script dependency should be
11 loaded before loading backbone.js
12                         deps: ['underscore'],
                           // use the global 'Backbone' as the module
13 name.
14                         exports:  'Backbone'
15                 }
16         }
17 });
```

在这个例子中把根路径设置为了 *scripts/app* ,由 *lib* 开始的每个模块都被配置在 *scripts/lib* 文件夹下面，*backbone* 加载的是一个 *shim* 依赖。

用 RequireJS 定义模块

模块是进行了内部实现封装、暴露接口和合理限制范围的对象。ReuqireJS 提供了 define 函数用于定义模块。按章惯例每个 Javascript 文件只应该定义一个模块。define 函数接受一个依赖数组和一个包含模块定义的函数。通常模块定义函数会把前面的数组中的依赖模块按顺序做为参数接收。例如，下面是一个简单的模块定义:

```
01 define(["logger"], function(logger) {
02                 return {
03                           firstName: "John",
04                           lastName: "Black",
05                           sayHello: function() {
06                                 logger.log('hello');
07                           }
08                 }
09         }
10);
```

我们看，一个包含了 logger 的模块依赖数组被传给了 define 函数,该模块后面会被调用。同样我们看所定义的模块中有一个名为 logger 的参数，它会被设置为 logger 模块。每一个模块都应该返回它的 API.这个示例中我们有两个属性(firstName 和 lastName)和一个函数(sayHello)。然后，只要你后面定义的模块通过 ID 来引用这个模块，你就可以使用其暴露的 API。

使用 require 函数

在 RequireJS 中另外一个非常有用的函数是 require 函数。require 函数用于加载模块依赖但并不会创建一个模块。例如 :下面就是使用 require 定义了能够使用 jQuery 的一个函数。

```
view source
print?
```

```
require(['jquery'],  function  ($) {
        //jQuery was loaded and can be used now
});
```

小结

在这篇文章中我介绍了 RequireJS 库,它是我创建每个 Javascript 项目都会用到的库函数之一。它不仅仅用于加载模块依赖和相关的命令，RequireJS 帮助我们写出模块化的 JavaScript 代码，这非常有利于代码的可扩展性和重用性。

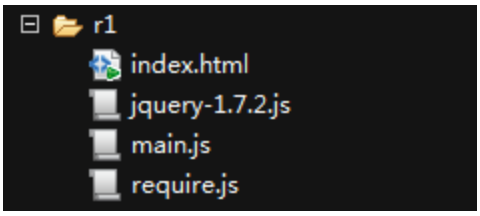
RequireJS 入门（一）

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。好，以一个简单示例开始。

新建一个目录，结构如下



目录 r1 下有 index.html、jquery-1.7.2.js、main.js、require.js。require.js 和 jquery-1.7.2.js 去各自官网下载即可。

index.html 如下

```
<!doctype html>
<html>
        <head>
                <title>requirejs 入门（一）</title>
```

```
5                   <meta charset="utf-8">
6                   <script data-main="main"  src="require.js"></script>
7           </head>
8           <body>
9
10          </body>
11  </html>
```

使用 requirejs 很简单，只需要在 head 中通过 script 标签引入它（实际上除了 require.js，其它文件模块都不再使用 script 标签引入）。

细心的同学会发现 script 标签上了多了一个自定义属性：data-main="main"，等号右边的 main 指的 main.js。当然可以使用任意的名称。这个 main 指主模块或入口模块，好比 c 或 java 的主函数 main。

main.js 如下

```
1   require.config({
2           paths: {
3                   jquery: 'jquery-1.7.2'
4           }
5   });
6
7   require(['jquery'], function($) {
8           alert($().jquery);
9   });
```

main.js 中就两个函数调用 require.config 和 require。

require.config 用来配置一些参数，它将影响到 requirejs 库的一些行为。

require.config 的参数是一个 JS 对象，常用的配置有 baseUrl，paths 等。

这里配置了 paths 参数，使用模块名"jquery"，其实际文件路径 jquery-1.7.2.js（后缀.js 可以省略）。

我们知道 jQuery 从 1.7 后开始支持 AMD 规范，即如果 jQuery 作为一个 AMD 模块运行时，它的模块名是"jquery"。注意"jquery"是固定的，不能写"jQuery"或其它。

注：如果文件名“jquery-1.7.2.js”改为“jquery.js”就不必配置 paths 参数了。

jQuery 中的支持 AMD 代码如下

```
if  ( typeof  define === "function"  && define.amd && define.amd.jQuery ) {
        define( "jquery", [], function  () { return  jQuery; } );
}
```

我们知道 jQuery 最终向外暴露的是全局的 jQuery 和 $。如下

```
// Expose jQuery to the global object
window.jQuery = window.$ = jQuery;
```

如果将 jQuery 应用在模块化开发时，其实可以不使用全局的，即可以不暴露出来。需要用到 jQuery 时使用 require 函数即可，

这里 require 函数的第一个参数是数组，数组中存放的是模块名（字符串类型），数组中的模块与回调函数的参数一一对应。这里的例子则只有一个模块“jquery”。

把目录 r1 放到 apache 或其它 web 服务器上，访问 index.html。

网络请求如下

URL	状态	域	大小	远程 IP	时间线
GET index.html	200 OK	127.0.0.1:8020	202 B	127.0.0.1:8020	4ms
GET require.js	200 OK	127.0.0.1:8020	85.9 KB	127.0.0.1:8020	1ms
GET main.js	200 OK	127.0.0.1:8020	127 B	127.0.0.1:8020	1ms
GET jquery-1.7.2.js	200 OK	127.0.0.1:8020	256.1 KB	127.0.0.1:8020	2ms
4 个请求			342.4 KB		

我们看到除了 require.js 外 main.js 和 jquery-1.7.2.js 也请求下来了。而它们正是通过 requirejs 请求的。

页面上会弹出 jQuery 的版本

这是一个很简单的示例，使用 requirejs 动态加载 jquery。使用到了以下知识点

1、data-main 属性

2、require.config 方法

3、require 函数

感兴趣的请继续阅读 下一篇 。

r1.zip

RequireJS 入门（二）

上一篇是把整个 jQuery 库作为一个模块。这篇来写一个自己的模块：选择器。

为演示方便这里仅实现常用的三种选择器 id，className，attribute。RequireJS 使用 define 来定义模块。

新建目录结构如下

这次新建了一个子目录 js，把 main.js 和 selctor.js 放入其中，require.js 仍然和 index.html 在同一级目录。

HTML 如下

```
<!doctype html>
<html>
        <head>
                <title>requirejs 入门（二）</title>
                <meta charset="utf-8">
                <style type="text/css">
                        .wrapper {
                                width: 200px;
                                height: 200px;
                                background: gray;
                        }
                </style>
        </head>
        <body>
                <div class="wrapper"></div>
                <script data-main="js/main"  src="require.js"></script>
        </body>
</html>
```

这次把 script 标签放到了 div 的后面，因为要用选择器去获取页面 dom 元素，而这要等到 dom ready 后。

因为把 main.js 放到 js 目录中，这里 data-main 的值须改为“js/main”。

selector.js 如下

```
define(function() {

        function  query(selector,context) {
                var  s = selector,
                        doc = document,
                        regId = /^#[\w\-]+/,
                        regCls = /^([\w\-]+)?\.([\w\-]+)/,
                        regTag = /^([\w\*]+)$/,
                        regNodeAttr = /^([\w\-]+)?\[([\w]+)(=(\w+))?\]/;

                var  context =
                                context == undefined ?
                                document :
                                typeof  context == 'string'  ?
                                doc.getElementById(context.substr(1,context.leng
```

```
16                                  context;
17
18                  if(regId.test(s)) {
19                          return  doc.getElementById(s.substr(1,s.length));
20                  }
21                  // 略...
22          }
23
24          return  query;
25  });
```

define 的参数为一个匿名函数，该匿名函数执行后返回 query，query 为函数类型。query 就是选择器的实现函数。

main.js 如下

```
1   require.config({
2           baseUrl: 'js'
3   });
4
5   require(['selector'], function(query) {
6           var  els = query('.wrapper');
7           console.log(els)
8   });
```

require.config 方法执行配置了 baseUrl 为"js"，baseUrl 指的模块文件的根目录，可以是绝对路径或相对路径。这里用的是相对路径。相对路径指引入 require.js 的页面为参考点，这里是 index.html。

把目录 r2 放到 apache 或其它 web 服务器上，访问 index.html。

网络请求如下

URL	状态	域	大小	远程 IP	时间线
GET index.html	200 OK	127.0.0.1:8020	355 B	127.0.0.1:8020	3ms
GET require.js	200 OK	127.0.0.1:8020	85.9 KB	127.0.0.1:8020	
GET main.js	200 OK	127.0.0.1:8020	137 B	127.0.0.1:8020	
GET selector.js	200 OK	127.0.0.1:8020	2.8 KB	127.0.0.1:8020	
4 个请求			89.2 KB		

main.js 和 selector.js 都请求下来了。

selector.js 下载后使用 query 获取页面 class 为“.wrapper”的元素，控制台输出了该元素。如下

总结：

1、使用 baseUrl 来配置模块根目录，baseUrl 可以是绝对路径也可以是相对路径。

2、使用 define 定义一个函数类型模块，RequireJS 的模块可以是 JS 对象，函数或其它任何类型（CommonJS/SeaJS 则只能是 JS 对象）。

r2.zip

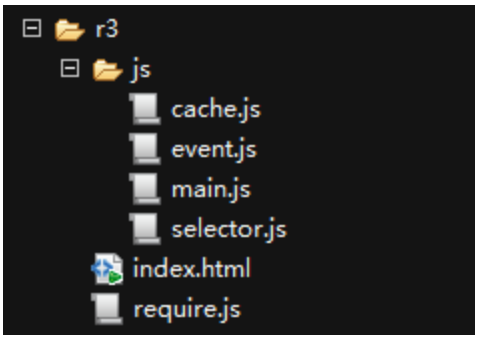
RequireJS 入门（三）

这篇来写一个具有依赖的事件模块 event。event 提供三个方法 bind、unbind、trigger 来管理 DOM 元素事件。

event 依赖于 cache 模块，cache 模块类似于 jQuery 的$.data 方法。提供了 set、get、remove 等方法用来管理存放在 DOM 元素上的数据。

示例实现功能：为页面上所有的段落 P 元素添加一个点击事件，响应函数会弹出 P 元素的 innerHTML。

创建的目录如下



为了获取元素，用到了上一篇写的 selector.js。不在贴其代码。

index.html 如下

```
<!doctype html>
<html>
        <head>
                <title>requirejs 入门（三）</title>
                <meta charset="utf-8">
                <style type="text/css">
                        p {
                                width: 200px;
                                background: gray;
                        }
                </style>
        </head>
        <body>
                <p>p1</p><p>p2</p><p>p3</p><p>p4</p><p>p5</p>
                <script data-main="js/main"  src="require.js"></script>
        </body>
</html>
```

cache.js 如下

```
define(function() {
        var  idSeed = 0,
                cache = {},
                id = '_ guid _';

        // @private
        function  guid(el) {
                return  el[id] || (el[id] = ++idSeed);
        }

        return  {
                set: function(el, key, val) {

                        if  (!el) {
                                throw  new  Error('setting failed, invalid eleme
                        }

                        var  id = guid(el),
                                c = cache[id] || (cache[id] = {});
```

```
                        if  (key) c[key] = val;

                        return  c;
                },

                // 略去...
        };
});
```

cache 模块的写法没啥特殊的，与 selector 不同的是返回的是一个 JS 对象。

event.js 如下

```
define(['cache'], function(cache) {
        var  doc = window.document,
                w3c = !!doc.addEventListener,
                expando = 'snandy'  + (''+Math.random()).replace(/\D/g, ''),
                triggered,
                addListener = w3c ?
                        function(el, type, fn) { el.addEventListener(type, fn, fa
                        function(el, type, fn) { el.attachEvent('on'  + type, fn)
                removeListener = w3c ?
                        function(el, type, fn) { el.removeEventListener(type, fn,
                        function(el, type, fn) { el.detachEvent('on'  + type, fn)

        // 略去...

        return  {
                bind : bind,
                unbind : unbind,
                trigger : trigger
        };
});
```

event 依赖于 cache，定义时第一个参数数组中放入"cache"即可。第二个参数是为函数类型，
它的参数就是 cache 模块对象。
这样定义后，当 require 事件模块时，requirejs 会自动将 event 依赖的 cache.js 也下载下来。

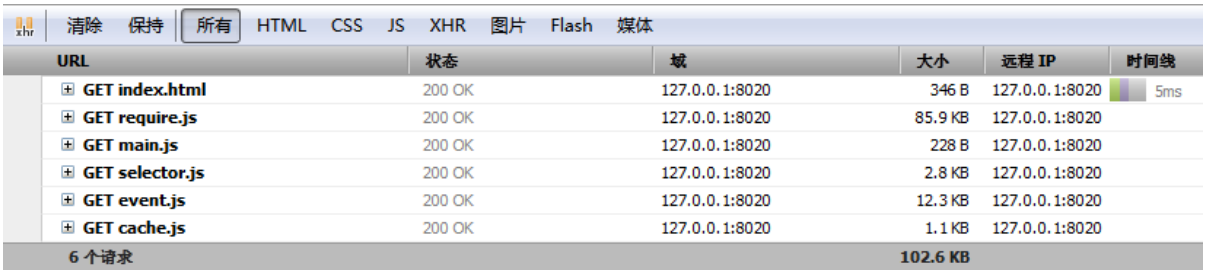
main.js 如下

```
require.config({
        baseUrl: 'js'
```

```
3       });
4
5       require(['selector', 'event'], function($, E) {
6               var  els = $('p');
7               for  (var  i=0; i<els.length; i++) {
8                       E.bind(els[i], 'click', function() {
9                               alert(this.innerHTML);
10                      });
11              }
12      });
```

依然先配置了下模块的根目录 js，然后使用 require 获取 selector 和 event 模块。
回调函数中使用选择器$(别名)和事件管理对象 E(别名)给页面上的所有 P 元素添加点击事件。
注意：require 的第一个参数数组内的模块名必须和回调函数的形参一一对应。

把目录 r3 放到 apache 或其它 web 服务器上，访问 index.html。网络请求如下



我们看到当 selector.js 和 event.js 下载后，event.js 依赖的 cache.js 也被自动下载了。这时点击页面上各个 P 元素，会弹出对应的 innerHTML。如下

总结：
当一个模块依赖（a）于另一个模块（b）时，定义该模块时的第一个参数为数组，数组中的模块名（字符串类型）就是它所依赖的模块。

当有多个依赖模时，须注意回调函数的形参顺序得和数组元素一一对应。此时 requirejs 会自动识别依赖，且把它们都下载下来后再进行回调。

r3.zip

RequireJS 2.0 正式发布

就在前天晚上 RequireJS 发布了一个大版本，直接从 version1.0.8 升级到了 2.0。随后的几小时 James Burke 又迅速的将版本调整为 2.0.1，当然其配套的打包压缩工具 r.js 也同时升级到了 2.0.1。此次变化较大，代码也进行了重构，层次更清晰可读。功能上主要变化如下：

1，延迟模块的执行。

这是一个很大变化，以前模块加载后 factory 立马执行。性能上肯定有一些损耗。2.0 修改实现，再没人诟病 AMD 的模块是立即执行的。现在也可以等到 require 的时候才执行。

2，config 增加了 shim，map，module，enforceDefine。

shim 参数解决了使用非 AMD 方式定义的模块（如 jQuery 插件）及其载入顺序。使用 shim 参数来取代 1.0 版本的 order 插件。其实在 1.0 版本中就曾经有人开发过 use 和 wrap 插件来解决此类问题。考虑到很多开发者有此类需求（比如某些 JS 模块是较早时候其他人开发的，非 AMD 方式）此次 2.0 版本直接将其内置其中。

下面是一个使用 jQuery 插件形式配置的参数。我们知道 jQuery 插件本质上是将命名空间挂在全局的 jQuery 或 jQuery.fn 上而非使用 define 定义的模块。而 jQuery 插件都依赖于 jQuery，即在 require 插件时得保证 jQuery 先下载下来。可以如下配置

```
1  require.config({
2          shim: {
3                  'jquery-slide': ['jquery']
4          }
5  });
6  require(['jquery-slide']);
```

这时会保证先下载 jquery.js，然后再下载 jquery-slide.js。

map 参数用来解决同一个模块的不同版本问题，这一灵感来自于 Dojo 的 packageMap。有这样的场景：开发初期使用了的 jquery-1.6.4，后期升级到了 1.7.2。但担心有些依赖 jquery-1.6.4 的代码升级到 1.7.2 后有问题。因此保守的让这部分代码继续使用 1.6.4 版本。这时 map 参数将派上用场。

假如 A，B 模块中使用了 jquery-1.6.4.js，C，D 模块中使用了 jquery-1.7.2.js。如下

```
requirejs.config({
        map: {
                'A': {
                        'jquery': 'jquery-1.6.4'
                },
                'B': {
                        'jquery': 'jquery-1.7.2'
                }
        }
});
require(['A']); // download jquery-1.6.4.js
require(['B']); // download jquery-1.7.2.js
```

这时 require(['A'])将会下载 jquery-1.6.4.js，require(['B'])会下载 jquery-1.7.2.js。模块"A"如果写成"*"则表示除了 B 模块使用 jquery-1.7.2 之外其它模块都使用 jquery-1.6.4。map 参数解决了模块的各个版本问题，很好的向下兼容了老代码。

config 参数用来给指定的模块传递一些有用的数据。如下

```
require.config({
        config: {
                'A': {
                        info: {name: 'jack'}
                }
        }
});
```

使用 A 的模块中可以通过 A.config().info 获取到该数据信息。如

```
require(['A'], function(A) {
        var info = a.config().info;
        console.log(info);
});
```

enforceDefine 用来强制模块使用 define 定义，默认为 false。如 underscore 不再支持 AMD 后，其代码移除了 define。此时如果仍然使用 requirejs 来载入它，它就是普通的 js 文件了。此时如果 enforceDefine 设为 true，虽然 underscore.js 能下载但 requirejs 会报错。如

```
require.config({
        enforceDefine: true
});
require(['underscore'], function(_){
        console.log(_)
})
```

错误信息

4，require 函数增加了第三个参数 errbacks。

很明显该函数指模块文件没有载入成功时的回调。这个也是应一些开发者得要求而增加，其中还包括另一个著名 AMD 的实现 curl 的作者 John Hann。

```
require(['b'], function(){
        console.log('success');
},function(err){
        console.log(err)
});
```

err 会给出一些出错提示信息。

5，更强大的 paths 参数。

requirejs 1.x 版本中已经有 paths 参数，用来映射模块别名。requirejs2.0 更加强大，可以配置为一个数组，顺序映射。当前面的路径没有成功载入时可接着使用后面的路径。如下

```
requirejs.config({
        enforceDefine: true,
        paths: {
                jquery: [
                        'http://ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min',
                        'lib/jquery'
                ]
```

```
        }
});

require(['jquery'], function ($) {
});
```

当 google cdn 上的 jquery.min.js 没有获取时（假如 google 宕机），可以使用本地的 lib/jquery.js。

6，在模块载入失败回调中可以使用 undef 函数移除模块的注册。

这个灵感来自 dojo AMD loader，RequireJS 取名 undef。如下

```
require(['jquery'], function ($) {
        //Do something with $ here
}, function (err) {
        var failedId = err.requireModules && err.requireModules[0];
        if (failedId === 'jquery') {
                requirejs.undef(failedId);
        }
});
```

7，删除了 jQuery domready 相关代码。

这次没人再诟病 RequireJS 和 jQuery 耦合的太紧密。

8，删除了 priority，packagePaths，catchError.define。

这几个参数已经有相应的替代品。

最后需要注意的是，虽然功能增加了不少。但代码量却减少了近 60 行。主要是去掉了 jQuery ready 相关代码。另外 newContext 函数依然有 1000 多行。

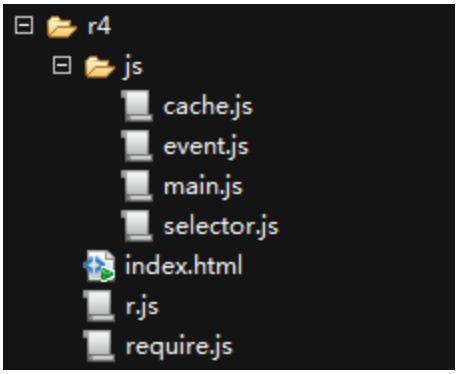
RequireJS 进阶（一）

为了应对日益复杂，大规模的 JavaScript 开发。我们化整为零，化繁为简。将复杂的逻辑划分一个个小单元，各个击破。这时一个项目可能会有几十个甚至上百个 JS 文件，每个文件为一个模块单元。如果上线时都是这些小文件，那将对性能造成一定影响。

RequireJS 提供了一个打包压缩工具 r.js 来对模块进行合并压缩。r.js 非常强大，不但可以压缩 js，css，甚至可以对整个项目进行打包。

r.js 的压缩工具使用 UglifyJS 或 Closure Compiler。默认使用 UglifyJS（jQuery 也是使用它压缩）。此外 r.js 需要 node.js 环境，当然它也可以运行在 Java 环境中如 Rhino。

这篇以一个简单的示例来感受下 r.js，创建的目录如下



和入门之三目录结构一样，写了三个模块 cache，event，selector。这三个模块的代码就不在贴了。main.js 也未做修改，实现的功能仍然是为页面上的所有段落 P 元素添加个点击事件，点击后弹出 P 的 innerHTML。唯一的区别是目录中**多了个 r.js**。

index.html 做了修改，如下

```
<!doctype html>
<html>
        <head>
                <title>requirejs 进阶（一）</title>
                <meta charset="utf-8"/>
                <style type="text/css">
                        p {
                                background:  #999;
                                width: 200px;
                        }
                </style>
        </head>
        <body>
```

```
14                  <p>p1</p><p>p2</p><p>p3</p><p>p4</p><p>p5</p>
15                  <script data-main="built"  src="require.js"></script>
16          </body>
17  </html>
```

注意，**data-main 改为了“built”**，上一篇的是“main”。我们将使用 r.js 把 js 目录下的 cache.js，event.js，selector.js，main.js 合并压缩后写到 r4 目录中。

好，让我们开始合并压缩吧。

1，打开 windows 命令窗口，cd 到 r4 目录中

```
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\taozhou>e:

E:\>cd work/req/r4

E:\work\req\r4>
```

2，输入命令

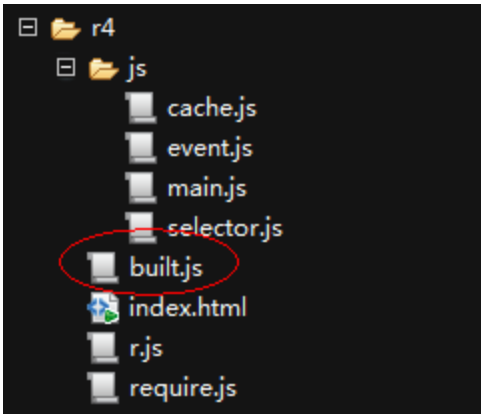
```
node r.js -o baseUrl=js  name=main  out=built.js
```

```
E:\work\req\r4>node r.js -o baseUrl=js name=main out=built.js

Tracing dependencies for: main
Uglifying file: E:/work/req/r4/built.js

E:/work/req/r4/built.js
----------------
E:/work/req/r4/js/selector.js
E:/work/req/r4/js/cache.js
E:/work/req/r4/js/event.js
E:/work/req/r4/js/main.js
```

命令行信息可以看到已经将各个 js 文件合并成功了。这时回到 r4 目录会发现多了一个 built.js 文件。



好了，合并压缩过程完成了。

把目录 r4 放到 apache 或其它 web 服务器上，访问 index.html。网络请求如下

可以看到除了 require.js，就只有 built.js 了。大大减少了 JS 文件的 http 请求。这时点击页面上各个 P 元素，会弹出对应的 innerHTML

这说明功能完损无缺。

下面对命令行做个简单解释。

```
node r.js -o baseUrl=js  name=main  out=built.js
```

-o　　表示优化，该参数是固定的，必选。

baseUrl　指存放模块的根目录，这里是 r4/js，因为 cd 到 r4 中了，只需设置为 js。可选，如果没有设置将从 r4 中查找 main.js。

name　　模块的入口文件，这里设置成“main”，那么 r.js 会从 baseUrl+main 去查找。这里实际是 r4/js/main.js。r.js 会分析 main.js，找出其所依赖的所有其它模块，然后合并压缩。

out　　指合并压缩后输出的文件路径，这里直接是 built.js，那么将输出到根目录 r4 下。

好了，再介绍两个参数

1，excludeShallow　合并时将排除该文件

```
node r.js -o baseUrl=js  name=main  out=built.js  excludeShallow=selector
```

```
E:\work\req\r4>node r.js -o baseUrl=js name=main out=built.js excludeShallow=sel
ector

Tracing dependencies for: main
Uglifying file: E:/work/req/r4/built.js

E:/work/req/r4/built.js
----------------
E:/work/req/r4/js/cache.js
E:/work/req/r4/js/event.js
E:/work/req/r4/js/main.js
```

可以看到输出信息中不再包括 selector.js。这时运行 index.html 页面，会发现 selector.js 被单独请求下来了。

2，optimize（none/uglify/closure）　指定是否压缩，默认为 uglify

不传该参数时 r.js 默认以 UglifyJS 压缩。设置为 none 则不会压缩，仅合并，这在开发阶段是很用用的。

```
node r.js -o baseUrl=js  name=main  out=built.js  optimize=none
```

```
E:\work\req\r4>node r.js -o baseUrl=js name=main out=built.js optimize=none

Tracing dependencies for: main

E:/work/req/r4/built.js
----------------
E:/work/req/r4/js/selector.js
E:/work/req/r4/js/cache.js
E:/work/req/r4/js/event.js
E:/work/req/r4/js/main.js
```

这时生成的 built.js 是没有压缩的。

总结：
这篇演示了采用模块开发后上线前的一个小示例：把所有模块文件合并为一个文件。

先下载 r.js 放到开发目录中，然后使用命令行来合并压缩。其中演示了命令行参数-o、baseUrl、name、out 及 excludeShallow、optimize 的使用。-o、name、out 是必选的，其它为可选。

r4.zip

RequireJS 和 AMD 规范

目录

概述

RequireJS 是一个工具库，主要用于客户端的模块管理。它可以让客户端的代码分成一个个模块，实现异步或动态加载，

从而提高代码的性能和可维护性。它的模块管理遵守 AMD 规范（Asynchronous Module Definition）。

RequireJS 的基本思想是，通过 define 方法，将代码定义为模块；通过 require 方法，实现代码的模块加载。

首先，将 require.js 嵌入网页，然后就能在网页中进行模块化编程了。

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

上面代码的 data-main 属性不可省略，用于指定主代码所在的脚本文件，在上例中为 scripts 子目录下的 main.js 文件。用户自定义的代码就放在这个 main.js 文件中。

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）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。

（2）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/?st
ats'],

        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) {

    // ...

};
```

AMD 模式小结

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。

配置 require.js：config 方法

require 方法本身也是一个对象，它带有一个 config 方法，用来配置 require.js 运行参数。config 方法接受一个对象作为参数。

```
require.config({

    paths: {

        jquery: [


'//cdnjs.cloudflare.com/ajax/libs/jquery/2.0.0/jquery.min.js',

            'lib/jquery'

        ]

    }

});
```

config 方法的参数对象有以下主要成员：

（1）paths

paths 参数指定各个模块的位置。这个位置可以是同一个服务器上的相对位置，也可以是外部网址。可以为每个模块定义多个位置，如果第一个位置加载失败，则加载第二个位置，上面的示例就表示如果 CDN 加载失败，则加载服务器上的备用脚本。需要注意的是，指定本地文件路径时，可以省略文件最后的 js 后缀名。

```
require(["jquery"], function($) {

    // ...

});
```

上面代码加载 jquery 模块，因为 jquery 的路径已经在 paths 参数中定义了，所以就会到事先设定的位置下载。

（2）baseUrl

baseUrl 参数指定本地模块位置的基准目录，即本地模块的路径是相对于哪个目录的。该属性通常由 require.js 加载时的 data-main 属性指定。

（3）shim

有些库不是 AMD 兼容的，这时就需要指定 shim 属性的值。shim 可以理解成“垫片”，用来帮助 require.js 加载非 AMD 规范的库。

```
require.config({

    paths: {

        "backbone": "vendor/backbone",

        "underscore": "vendor/underscore"

    },

    shim: {

        "backbone": {

            deps: [ "underscore" ],

            exports: "Backbone"

        },

        "underscore": {

            exports: "_"
```

```
        }

    }

});
```

上面代码中的 backbone 和 underscore 就是非 AMD 规范的库。shim 指定它们的依赖关系（backbone 依赖于 underscore），以及输出符号（backbone 为“Backbone”，underscore 为“_”）。

插件

RequireJS 允许使用插件，加载各种格式的数据。完整的插件清单可以查看官方网站。

下面是插入文本数据所使用的 text 插件的例子。

```
define([

    'backbone',

    'text!templates.html'

], function( Backbone, template ){

   // ...
```

```
});
```

上面代码加载的第一个模块是 backbone，第二个模块则是一个文本，用'text!'表示。该文本作为字符串，存放在回调函数的 template 变量中。

优化器 r.js

RequireJS 提供一个基于 node.js 的命令行工具 r.js，用来压缩多个 js 文件。它的主要作用是将多个模块文件压缩合并成一个脚本文件，以减少网页的 HTTP 请求数。

第一步是安装 r.js（假设已经安装了 node.js）。

```
npm install -g requirejs
```

然后，使用的时候，直接在命令行键入以下格式的命令。

```
node r.js -o <arguments>
```

<argument>表示命令运行时，所需要的一系列参数，比如像下面这样：

```
node r.js -o baseUrl=. name=main out=main-built.js
```

除了直接在命令行提供参数设置，也可以将参数写入一个文件，假定文件名为 build.js。

```
({
    baseUrl: ".",
    name: "main",
    out: "main-built.js"
})
```

然后，在命令行下用 r.js 运行这个参数文件，就 OK 了，不需要其他步骤了。

```
node r.js -o build.js
```

下面是一个参数文件的范例，假定位置就在根目录下，文件名为 build.js。

```
({
    appDir: './',
    baseUrl: './js',
    dir: './dist',
    modules: [
```

```
        {
            name: 'main'
        }
    ],
    fileExclusionRegExp: /^(r|build)\.js$/,
    optimizeCss: 'standard',
    removeCombined: true,
    paths: {
        jquery: 'lib/jquery',
        underscore: 'lib/underscore',
        backbone: 'lib/backbone/backbone',
        backboneLocalstorage:
'lib/backbone/backbone.localStorage',
        text: 'lib/require/text'
    },
```

```
shim: {
    underscore: {
        exports: '_'
    },
    backbone: {
        deps: [
            'underscore',
            'jquery'
        ],
        exports: 'Backbone'
    },
    backboneLocalstorage: {
        deps: ['backbone'],
        exports: 'Store'
```

```
        }

    }

})
```

上面代码将多个模块压缩合并成一个 main.js。

参数文件的主要成员解释如下：

- **appDir**：项目目录，相对于参数文件的位置。
- **baseUrl**：js 文件的位置。
- **dir**：输出目录。
- **modules**：一个包含对象的数组，每个对象就是一个要被优化的模块。
- **fileExclusionRegExp**：凡是匹配这个正则表达式的文件名，都不会被拷贝到输出目录。
- **optimizeCss**：自动压缩 CSS 文件，可取的值包括“none”, “standard”, “standard.keepLines”, “standard.keepComments”, “standard.keepComments.keepLines”。
- **removeCombined**：如果为 true，合并后的原文件将不保留在输出目录中。

- **paths**：各个模块的相对路径，可以省略 js 后缀名。
- **shim**：配置依赖性关系。如果某一个模块不是 AMD 模式定义的，就可以用 shim 属性指定模块的依赖性关系和输出值。
- **generateSourceMaps**：是否要生成 source map 文件。

更详细的解释可以参考官方文档。

运行优化命令后，可以前往 dist 目录查看优化后的文件。

下面是另一个 build.js 的例子。

```
({
    mainConfigFile : "js/main.js",
    baseUrl: "js",
    removeCombined: true,
    findNestedDependencies: true,
    dir: "dist",
    modules: [
        {
```

```
            name: "main",

            exclude: [

                "infrastructure"

            ]

        },

        {

            name: "infrastructure"

        }

    ]

})
```

上面代码将模块文件压缩合并成两个文件，第一个是 main.js（指定排除 infrastructure.js），第二个则是 infrastructure.js。

参考链接

- NaorYe, Optimize (Concatenate and Minify) RequireJS Projects

- Jonathan Creamer, Deep dive into Require.js
- Addy Osmani, Writing Modular JavaScript With AMD, CommonJS & ES Harmony
- Jim Cowart, Five Helpful Tips When Using RequireJS
- Jim Cowart, Using r.js to Optimize Your RequireJS Project

RequireJS API 中文版

PRE: 3

update: 2013-05-13

这是 RequireJS 2.0 的 API，对应的官方文档版本号是 2.1.6

目录

---

§ 1 用法

§ 1.1 加载 JavaScript 文件

RequireJS 的目标是鼓励代码的模块化，它使用了不同于传统<script>标签的脚本加载步骤。可以用它来加速、优化代码，但其主要目的还是为了代码的模块化。它鼓励在使用脚本时以 module ID 替代 URL 地址。

RequireJS 以一个相对于 baseUrl 的地址来加载所有的代码。 页面顶层<script>标签含有一个特殊的属性 data-main，require.js 使用它来启动脚本加载过程，而 baseUrl 一般设置到与该属性相一致的目录。下列示例中展示了 baseUrl 的设置：

```
<!-- 将 baseUrl 设置到"scripts"目录，并且加载 module ID 为'main'的一个脚本'-->
<script data-main="scripts/main.js" src="scripts/require.js"></script>
```

baseUrl亦可通过RequireJS config手动设置。如果没有显式指定config及data-main，则默认的 baseUrl 为包含 RequireJS 的那个 HTML 页面的所属目录。

RequireJS 默认假定所有的依赖资源都是 js 脚本，因此无需在 module ID 上再加".js"后缀，RequireJS 在进行 module ID 到 path 的解析时会自动补上后缀。你可以通过 paths config 设置一组脚本，这些有助于我们在使用脚本时码更少的字。

有时候你想避开"baseUrl + paths"的解析过程，而是直接指定加载某一个目录下的脚本。此时可以这样做：如果一个 module ID 符合下述规则之一，其 ID 解析会避开常规的"baseUrl + paths"配置，而是直接将其加载为一个相对于当前 HTML 文档的脚本：

- 以".js"结尾；
- 以"/"开头；
- 包含 URL 协议，如"http:"、"https:"

一般来说，最好还是使用 baseUrl 及"paths" config 去设置 module ID。它会给你带来额外的灵活性，如便于脚本的重命名、重定位等。 同时，为了避免凌乱的配置，最好不要使用多级嵌套的目录层次来组织代码，而是要么将所有的脚本都放置到 baseUrl 中，要么分置为项目库/第三方库的一个扁平结构，如下：

- www/
  - index.html
  - js/
    - app/
    - sub.js
    - lib/
    - jquery.js
    - canvas.js
    - app.js

在 index.html 中：

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

在 app.js 中：

```
requirejs.config({
    // 默认从 js/lib 加载所有的 module ID
    baseUrl: 'js/lib',
    // 除非，module ID 以"app"开头，则
```

```
    // 从 js/app 目录加载。
    // 注意，paths config 是相对于 baseUrl 的，
    // 而且不要包含".js"的后缀，因为一个 path
    // 有可能是个目录
    paths: {
        app: '../app'
    }
});

// 启动 main app
requirejs(['jquery', 'canvas', 'app/sub'],
function   ($,        canvas,   sub) {
    //自此，jQuery，canvas 以及 app/sub 模块
    //都已加载并可开始使用了。
});
```

注意在示例中，三方库如 jQuery 没有将版本号包含在他们的文件名中。我们建议将版本信息放置在单独的文件中来进行跟踪。使用诸如 volo 这类的工具，可以将 package.json 打上版本信息，并在磁盘上保持文件名为"jquery.js"。这有助于你保持配置的最小化，避免为每个库版本设置一条 path。例如，将"jquery"配置为"jquery-1.7.2"。

理想状况下，每个加载的脚本都是通过 define()来定义的一个模块；但有些"浏览器全局变量注入"型的传统/遗留库并没有使用 define()来定义它们的依赖关系，你必须为此使用 shim config 来指明它们的依赖关系。 如果你没有指明依赖关系，加载可能报错。这是因为基于速度的原因，RequireJS 会异步地以无序的形式加载这些库。

§ 1.2 定义模块

模块不同于传统的脚本文件，它良好地定义了一个作用域来避免全局名称空间污染。它可以显式地列出其依赖关系，并以函数(定义此模块的那个函数)参数的形式将这些依赖进行注入，而无需引用全局变量。RequireJS 的模块是模块模式的一个扩展，其好处是无需全局地引用其他模块。

RequireJS 的模块语法允许它尽快地加载多个模块，虽然加载的顺序不定，但依赖的顺序最终是正确的。同时因为无需创建全局变量，甚至可以做到在同一个页面上同时加载同一模块的不同版本。

(如果你熟悉 ConmmonJS，可参看 CommonJS 的注释信息以了解 RequireJS 模块到 CommonJS 模块的映射关系)。

一个磁盘文件应该只定义 **1** 个模块。多个模块可以使用内置优化工具将其组织打包。

§ 1.2.1 简单的值对

如果一个模块仅含值对，没有任何依赖，则在 define()中定义这些值对就好了：

```
//my/shirt.js:
define({
    color: "black",
    size: "unisize"
});
```

§ 1.2.2 函数式定义

如果一个模块没有任何依赖，但需要一个做 setup 工作的函数，则在 define()中定义该函数，并将其传给 define()：

```
//my/shirt.js 现在在返回其模块定义之前做
//了一些 setup 工作
define(function () {
    //Do setup work here

    return {
        color: "black",
        size: "unisize"
    }
});
```

§ 1.2.3 存在依赖的函数式定义

如果模块存在依赖：则第一个参数是依赖的名称数组；第二个参数是函数，在模块的所有依赖加载完毕后，该函数会被调用来定义该模块，因此该模块应该返回一个定义了本模块的 object。依赖关系会以参数的形式注入到该函数上，参数列表与依赖名称列表一一对应。

```
//my/shirt.js 现在对同目录下的 cart 及 inventory 存在依赖
define(["./cart", "./inventory"], function(cart, inventory) {
        //返回一个定义了该"my/shirt"模块的 object
        return {
            color: "blue",
            size: "large",
            addToCart: function() {
                inventory.decrement(this);
                cart.add(this);
            }
```

```
        }
    }
);
```

本示例创建了一个 my/shirt 模块，它依赖于 my/cart 及 my/inventory。磁盘上各文件分布如下：

- my/cart.js
- my/inventory.js
- my/shirt.js

模块函数以参数"cart"及"inventory"使用这两个以"./cart"及"./inventory"名称指定的模块。在这两个模块加载完毕之前，模块函数不会被调用。

严重不鼓励模块定义全局变量。遵循此处的定义模式，可以使得同一模块的不同版本并存于同一个页面上(参见 **高级用法**)。另外，函参的顺序应与依赖顺序保存一致。

返回的 object 定义了"my/shirt"模块。这种定义模式下，"my/shirt"不作为一个全局变量而存在。

§ 1.2.4 将模块定义为一个函数

对模块的返回值类型并没有强制为一定是个 object，任何函数的返回值都是允许的。此处是一个返回了函数的模块定义：

```
//foo/title.js 像之前一样使用了 my/cart 及 my/inventory 模块，
//但 foo/bar.js 位于不同于"my"模块的目录下，它在模块依赖名称
//中使用"my"来定位它们。依赖名称中的"my"可能映射到任意一个目录，
//但默认地，假定它邻接着"foo"目录。
define(["my/cart", "my/inventory"],
    function(cart, inventory) {
        //返回一个函数以定义"foo/title".
        //它获取/设置 window 的 title
        return function(title) {
            return title ? (window.title = title) :
                  inventory.storeName + ' ' + cart.name;
        }
    }
);
```

§ 1.2.5 简单包装 CommonJS 来定义模块

如果你现有一些以 CommonJS 模块格式编写的代码，而这些代码难于使用上述依赖名称数组参数的形式来重构，你可以考虑直接将这些依赖对应到一些本地变量中进行使用。你可以使用一个 CommonJS 的简单包装来实现：

```
define(function(require, exports, module) {
       var a = require('a'),
           b = require('b');

       //Return the module value
       return function () {};
   }
);
```

该包装方法依靠 Function.prototype.toString()将函数内容赋予一个有意义的字串值，但在一些设备如 PS3 及一些老的 Opera 手机浏览器中不起作用。考虑在这些设备上使用优化器将依赖导出为数组形式。

更多的信息可参看 CommonJS Notes 页面，以及"Why AMD"页面的"Sugar"段落。

§ 1.2.6 定义一个命名模块

你可能会看到一些 define()中包含了一个模块名称作为首个参数：

```
   //显式地定义"foo/title"模块：
   define("foo/title",
       ["my/cart", "my/inventory"],
       function(cart, inventory) {
           //此处定义 foo/title object
      }
   );
```

这些常由优化工具生成。你也可以自己显式指定模块名称，但这使模块更不具备移植性——就是说若你将文件移动到其他目录下，你就得重命名。一般最好避免对模块硬编码，而是交给优化工具去生成。优化工具需要生成模块名以将多个模块打成一个包，加快到浏览器的载入速度。

§ 1.2.7 其他注意事项

一个文件一个模块: 每个 Javascript 文件应该只定义一个模块，这是模块名-至-文件名查找机制的自然要求。多个模块会被优化工具组织优化，但你在使用优化工具时应将多个模块放置到一个文件中。

define()中的相对模块名: 为了可以在 define()内部使用诸如 require("./relative/name")的调用以正确解析相对名称，记得将"require"本身作为一个依赖注入到模块中：

```
define(["require", "./relative/name"], function(require) {
    var mod = require("./relative/name");
});
```

或者更好地，使用下述为转换 CommonJS 模块所设的更短的语法：

```
define(function(require) {
    var mod = require("./relative/name");
});
```

该形式利用了 Function.prototype.toString()去查找 require()调用，然后将其与"require"一起加入到依赖数组中，这样代码可以正确地解析相对路径了。

相对路径在一些场景下格外有用，例如：为了以便于将代码共享给其他人或项目，你在某个目录下创建了一些模块。你可以访问模块的相邻模块，无需知道该目录的名称。

生成相对于模块的 URL 地址: 你可能需要生成一个相对于模块的 URL 地址。你可以将"require"作为一个依赖注入进来，然后调用 require.toUrl()以生成该 URL:

```
define(["require"], function(require) {
    var cssUrl = require.toUrl("./style.css");
});
```

控制台调试: 如果你需要处理一个已通过 require(["module/name"], function(){})调用加载了的模块，可以使用模块名作为字符串参数的 require()调用来获取它：

```
require("module/name").callSomeFunction()
```

注意这种形式仅在"module/name"已经由其异步形式的 require(["module/name"])加载了后才有效。只能在 define 内部使用形如"./module/name"的相对路径。

§ 1.2.8 循环依赖

如果你定义了一个循环依赖(a 依赖 b，b 同时依赖 a)，则在这种情形下当 b 的模块函数被调用的时候，它会得到一个 undefined 的 a。b 可以在模块已经定义好后用 require()方法再获取(记得将 require 作为依赖注入进来)：

```
//b.js:
define(["require", "a"],
    function(require, a) {
        //"a"将是 null，如果 a/b 间是循环依赖
        return function(title) {
            return require("a").doSomething();
        }
    }
);
```

一般说来你无需使用 require()去获取一个模块，而是应当使用注入到模块函数参数中的依赖。循环依赖比较罕见，它也是一个重构代码重新设计的警示灯。但不管怎样，有时候还是要用到循环依赖，这种情形下就使用上述的 require()方式来解决。

如果你熟悉 CommonJS，你可以考虑使用 **exports** 为模块建立一个空 object，该 object 可以立即被其他模块引用。在循环依赖的两头都如此操作之后，你就可以安全地持有其他模块了。这种方法仅在每个模块都是输出 object 作为模块值的时候有效，换成函数无效。

```
//b.js:
define(function(require, exports, module) {
    //若"a"使用了 exports，则此处我们就拥有了一个真正的 object 引用。
    //但在 b 返回值之前我们无法使用 a 的任何属性。
    var a = require("a");

    exports.foo = function () {
        return a.bar();
    };
});
```

或者，如果你使用依赖注入数组的步骤，则可用注入特殊的"exports"来解决：

```
//b.js:
define(['a', 'exports'], function(a, exports) {
    //若"a"使用了 exports，则此处我们就拥有了一个真正的 object 引用。
    //但在 b 返回值之前我们无法使用 a 的任何属性。

    exports.foo = function () {
```

```
        return a.bar();
    };
});
```

§ 1.2.9 JSONP 服务依赖

JSONP 是在 javascript 中服务调用的一种方式。它仅需简单地通过一个 script 标签发起 HTTP GET 请求，是实现跨域服务调用一种公认手段。

为了在 RequireJS 中使用 JSON 服务，须要将 callback 参数的值指定为"define"。这意味着你可将获取到的 JSONP URL 的值看成是一个模块定义。

下面是一个调用 JSONP API 端点的示例。该示例中，JSONP 的 callback 参数为"callback"，因此"callback=define"告诉 API 将 JSON 响应包裹到一个"define()"中：

```
require(["http://example.com/api/data.json?callback=define"],
    function (data) {
        //data 将作为此条 JSONP data 调用的 API 响应
        console.log(data);
    }
);
```

JSONP 的这种用法应仅限于应用的初始化中。一旦 JSONP 服务超时，其他通过 define() 定义了的模块也可能得不得执行，错误处理不是十分健壮。

仅支持返回值类型为 JSON object 的 JSONP 服务，其他返回类型如数组、字串、数字等都不能支持。

这种功能不该用于 long-polling 类的 JSONP 连接——那些用来处理实时流的 API。这些 API 在接收响应后一般会做 script 的清理，而 RequireJS 则只能获取该 JSONP URL 一次——后继使用 require()或 define()发起的的对同一 URL 的依赖(请求)只会得到一个缓存过的值。

JSONP 调用错误一般以服务超时的形式出现，因为简单加载一个 script 标签一般不会得到很 详细的网络错误信息。你可以 override requirejs.onError()来过去错误。更多的信息请参看错误处理部分。

§ 1.2.10 undefine 一个模块

有一个全局函数 requirejs.undef()用来 undefine 一个模块。它会重置 loader 的内部状态以使其忘记之前定义的一个模块。

但是若有其他模块已将此模块作为依赖使用了，该模块就不会被清除，所以该功能仅在无其他模块持有该模块时的错误处理中，或者当未来需要加载该模块时有点用。参见备错(errbacks)段的示例。

如果你打算在 undefine 时做一些复杂的依赖图分析，则半私有的 onResourceLoad API 可能对你有用。

---

§ 2 机理

RequireJS 使用 head.appendChild()将每一个依赖加载为一个 script 标签。

RequireJS 等待所有的依赖加载完毕，计算出模块定义函数正确调用顺序，然后依次调用它们。

在同步加载的服务端 JavaScript 环境中，可简单地重定义 require.load()来使用 RequireJS。build 系统就是这么做的。该环境中的 require.load 实现可在 build/jslib/requirePatch.js 中找到。

未来可能将该部分代码置入 require/目录下作为一个可选模块，这样你可以在你的宿主环境中使用它来获得正确的加载顺序。

---

§ 3 配置项

当在顶层 HTML 页面(或不作为一个模块定义的顶层脚本文件)中，可将配置作为首项放入：

```
<script src="scripts/require.js"></script>
<script>
  require.config({
    baseUrl: "/another/path",
    paths: {
        "some": "some/v1.0"
    },
    waitSeconds: 15
  });
  require( ["some/module", "my/module", "a.js", "b.js"],
    function(someModule,    myModule) {
        //该函数会在上述所有的依赖加载完毕后调用。
        //注意该函数可在页面加载完毕前被调用。
        //本回调函数是可选的。
    }
```

```
  );
</script>
```

或者，你将配置作为全局变量"require"在 require.js 加载之前进行定义，它会被自动应用。下面的示例定义的依赖会在 require.js 一旦定义了 require()之后即被加载：

```
<script>
    var require = {
        deps: ["some/module1", "my/module2", "a.js", "b.js"],
        callback: function(module1, module2) {
            //该函数会在上述所有的依赖加载完毕后调用。
            //注意该函数可在页面加载完毕前被调用。
            //本回调函数是可选的。
        }
    };
</script>
<script src="scripts/require.js"></script>
```

注意：最好使用 var require = {} 的形式而不是 window.require = {} 的形式。后者在 IE 中运行不正常。

支持的配置项：

baseUrl ：所有模块的查找根路径。所以上面的示例中，"my/module"的标签 src 值是"/another/path/my/module.js"。当加载纯.js 文件(依赖字串以/开头，或者以.js 结尾，或者含有协议)，不会使用 baseUrl。因此 a.js 及 b.js 都在包含上述代码段的 HTML 页面的同目录下加载。

如未显式设置 baseUrl，则默认值是加载 require.js 的 HTML 所处的位置。如果用了 **data-main** 属性，则该路径就变成 baseUrl。

baseUrl 可跟 require.js 页面处于不同的域下，RequireJS 脚本的加载是跨域的。唯一的限制是使用 text! plugins 加载文本内容时，这些路径应跟页面同域，至少在开发时应这样。优化工具会将 text! plugin 资源内联，因此在使用优化工具之后你可以使用跨域引用 text! plugin 资源的那些资源。

paths ：path 映射那些不直接放置于 baseUrl 下的模块名。设置 path 时起始位置是相对于 baseUrl 的，除非该 path 设置以"/"开头或含有 URL 协议（如 http:）。在上述的配置下，"some/module"的 script 标签 src 值是"/another/path/some/v1.0/module.js"。

用于模块名的 path 不应含有.js 后缀，因为一个 path 有可能映射到一个目录。路径解析机制会自动在映射模块名到 path 时添加上.js 后缀。在文本模版之类的场景中使用 require.toUrl()时它也会添加合适的后缀。

在浏览器中运行时，可指定路径的备选(fallbacks)，以实现诸如首先指定了从 CDN 中加载，一旦 CDN 加载失败则从本地位置中加载这类的机制。

shim：为那些没有使用 define()来声明依赖关系、设置模块的"浏览器全局变量注入"型脚本做依赖和导出配置。

下面有个示例，它需要 RequireJS 2.1.0+，并且假定 backbone.js、underscore.js 、jquery.js 都装于 baseUrl 目录下。如果没有，则你可能需要为它们设置 paths config：

```
requirejs.config({
    shim: {
        'backbone': {
            //下述依赖脚本应在 backbone.js 之前加载
            deps: ['underscore', 'jquery'],
            //一旦加载，使用全局变量'Backbone'作为模块值
            exports: 'Backbone'
        },
        'underscore': {
            exports: '_'
        },
        'foo': {
            deps: ['bar'],
            exports: 'Foo',
            init: function (bar) {
                //使用该函数允许你调用库所支持的 noConflict 方法，或其他的清理工作。
                //但是这些库的一些插件们可能依然需要一个全局引用，函数中的"this"提供这个全局
引用。
                //依赖会以函数参数的形式被注入。
                //如果本函数具备返回值，则该值会被用做模块的 export 值，而不是使用上述'expor
ts'中的字串。
                return this.Foo.noConflict();
            }
        }
    }
});

//然后，在一个单独的文件中，如'MyModel.js'，定义一个模块，
//指定'backbone'作为依赖。RequireJS 会使用 shim 配置去合理
//地加载'backbone'并给予该模块一个本地的引用。全局的 Backbone 引用一并
//存在于页面上。
define(['backbone'], function (Backbone) {
  return Backbone.Model.extend({});
});
```

RequireJS 2.0.*中，shim 配置中的"exports"属性可以是一个函数而不是字串。这种情况下它就起到上述示例中的"init"属性的功能。 RequireJS 2.1.0+中加入了"init"承接库加载后的初始工作，以使 exports 作为字串值被 enforceDefine 所使用。

那些仅作为 jQuery 或 Backbone 的插件存在而不导出任何模块变量的"模块"们，shim 配置可简单设置为依赖数组：

```
requirejs.config({
    shim: {
        'jquery.colorize': ['jquery'],
        'jquery.scroll': ['jquery'],
        'backbone.layoutmanager': ['backbone']
    }
});
```

但请注意，若你想在 IE 中使用 404 加载检测以启用 path 备选(fallbacks)或备错(errbacks)，则需要给定一个字串值的 exports 以使 loader 能够检查出脚本是否实际加载了(init 中的返回值不会用于 enforceDefine 检查中)：

```
requirejs.config({
    shim: {
        'jquery.colorize': {
            deps: ['jquery'],
            exports: 'jQuery.fn.colorize'
        },
        'jquery.scroll': {
            deps: ['jquery'],
            exports: 'jQuery.fn.scroll'
        },
        'backbone.layoutmanager': {
            deps: ['backbone']
            exports: 'Backbone.LayoutManager'
        }
    }
});
```

"shim"配置的重要注意事项:

- shim 配置仅设置了代码的依赖关系，想要实际加载 shim 指定的或涉及的模块，仍然需要一个常规的 require/define 调用。设置 shim 本身不会触发代码的加载。

- 请仅使用其他"shim"模块作为 shim 脚本的依赖，或那些没有依赖关系，并且在调用 define()之前定义了全局变量(如 jQuery 或 lodash)的 AMD 库。否则，如果你使用了一个 AMD 模块作为一个 shim 配置模块的依赖，在 build 之后，AMD 模块可能在 shim 托管代码执行之前都不会被执行，这会导致错误。终极的解决方案是将所有 shim 托管代码都升级为含有可选的 AMD define()调用。

"shim"配置的优化器重要注意事项:

- 您应当使用 mainConfigFile build 配置项来指定含有 shim 配置的文件位置，否则优化器不会知晓 shim 配置。另一个手段是将 shim 配置复制到 build profile 中。
- 不要在一个 build 中混用 CDN 加载和 shim 配置。示例场景，如：你从 CDN 加载 jQuery 的同时使用 shim 配置加载依赖于 jQuery 的原版 Backbone。不要这么做。您应该在 build 中将 jQuery 内联而不是从 CDN 加载，否则 build 中内联的 Backbone 会在 CDN 加载 jQuery 之前运行。这是因为 shim 配置仅延时加载到所有的依赖已加载，而不会做任何 define 的自动装裹(auto-wrapping)。在 build 之后，所有依赖都已内联，shim 配置不能延时执行非 define()的代码。define()的模块可以在 build 之后与 CDN 加载代码一并工作，因为它们已将自己的代码合理地用 define 装裹了，在所有的依赖都已加载之前不会执行。因此记住：shim 配置仅是个处理非模块(non-modular)代码、遗留代码的将就手段，如可以应尽量使用 define()的模块。

- 对于本地的多文件 build，上述的 CDN 加载建议仍然适用。任何 shim 过的脚本，它们的依赖**必须**加载于该脚本执行之前。这意味着要么直接在含有 shim 脚本的 build 层 build 它的依赖，要么先使用 require([], function (){})调用来加载它的依赖，然后对含有 shim 脚本的 build 层发出一个嵌套的 require([])调用。
- 如果您使用了 uglifyjs 来压缩代码，**不要**将 uglify 的 toplevel 选项置为 true，或在命令行中**不要**使用 -mt。 该选项会破坏 shim 用于找到 exports 的全局名称。

map：对于给定的模块前缀，使用一个不同的模块 ID 来加载该模块。
该手段对于某些大型项目很重要：如有两类模块需要使用不同版本的"foo"，但它们之间仍需要一定的协同。
在那些基于上下文的多版本实现中很难做到这一点。而且，paths 配置仅用于为模块 ID 设置 root paths，而不是为了将一个模块 ID 映射到另一个。
map 示例：

```
requirejs.config({
    map: {
        'some/newmodule': {
            'foo': 'foo1.2'
        },
        'some/oldmodule': {
            'foo': 'foo1.0'
        }
    }
});
```

如果各模块在磁盘上分布如下：

- foo1.0.js
- foo1.2.js
- some/
  - newmodule.js

- oldmodule.js

  当“some/newmodule”调用了“require('foo')”，它将获取到 foo1.2.js 文件；而当“some/oldmodule”调用“`require('foo')”时它将获取到 foo1.0.js。

  该特性仅适用于那些调用了 define()并将其注册为匿名模块的真正 AMD 模块脚本。并且，请在 map 配置中仅使用**绝对模块 ID**，“../some/thing”之类的相对 ID 不能工作。
  另外在 map 中支持“*”，意思是“对于所有的模块加载，使用本 map 配置”。如果还有更细化的 map 配置，会优先于“*”配置。示例：

```
requirejs.config({
    map: {
        '*': {
            'foo': 'foo1.2'
        },
        'some/oldmodule': {
            'foo': 'foo1.0'
        }
    }
});
```

  意思是除了“some/oldmodule”外的所有模块，当要用“foo”时，使用“foo1.2”来替代。对于“some/oldmodule”自己，则使用“foo1.0”。

  config：常常需要将配置信息传给一个模块。这些配置往往是 application 级别的信息，需要一个手段将它们向下传递给模块。在 RequireJS 中，基于 requirejs.config()的 **config** 配置项来实现。要获取这些信息的模块可以加载特殊的依赖“module”，并调用 **module.config()**。示例：

```
requirejs.config({
    config: {
        'bar': {
            size: 'large'
        },
        'baz': {
            color: 'blue'
        }
    }
});

//bar.js 用了最简单的 CJS 装裹:
//http://requirejs.org/docs/whyamd.html#sugar
define(function (require, exports, module) {
    //其值是'large'
```

```
    var size = module.config().size;
});

//baz.js 使用了一个依赖数组，
//并要求一个特殊的依赖“module”：
//https://github.com/jrburke/requirejs/wiki/Differences-between-the-simplified-CommonJS-wrapper-and-standard-AMD-define#wiki-magic
define(['module'], function (module) {
    //Will be the value 'blue'
    var color = module.config().color;
});
```

若要将 config 传给包，将目标设置为包的主模块而不是包 ID：

```
requirejs.config({
    //将 API key 传递给包的主模块：
    config: {
        'pixie/index': {
            apiKey: 'XJKDLNS'
        }
    },
    //设置“pixie”包的主模块为 pixie 目录下的 index.js
    packages: [
        {
            name: 'pixie',
            main: 'index'
        }
    ]
});
```

packages：从 CommonJS 包(package)中加载模块。参见从包中加载模块。

waitSeconds：在放弃加载一个脚本之前等待的秒数。设为 0 禁用等待超时。默认为 7 秒。

context：命名一个加载上下文。这允许 require.js 在同一页面上加载模块的多个版本，如果每个顶层 require 调用都指定了一个唯一的上下文字符串。想要正确地使用，请参考多版本支持一节。

deps：指定要加载的一个依赖数组。当将 require 设置为一个 config object 在加载 require.js 之前使用时很有用。一旦 require.js 被定义，这些依赖就已加载。使用 deps 就像调用 require([])，但它在 loader 处理配置完毕之后就立即生效。它并不阻塞其他的 require()调用，它仅是指定某些模块作为 config 块的一部分而异步加载的手段而已。

callback：在 deps 加载完毕后执行的函数。当将 require 设置为一个 config object 在加载 require.js 之前使用时很有用，其作为配置的 deps 数组加载完毕后为 require 指定的函数。

enforceDefine：如果设置为 true，则当一个脚本不是通过 define()定义且不具备可供检查的 shim 导出字串值时，就会抛出错误。参考在 IE 中捕获加载错误一节。

xhtml：如果设置为 true，则使用 document.createElementNS()去创建 script 元素。

urlArgs：RequireJS 获取资源时附加在 URL 后面的额外的 query 参数。作为浏览器或服务器未正确配置时的"cache bust"手段很有用。使用 cache bust 配置的一个示例：

```
urlArgs: "bust=" +  (new Date()).getTime()
```

在开发中这很有用，但请记得在部署到生成环境之前移除它。

scriptType：指定 RequireJS 将 script 标签插入 document 时所用的 type=""值。默认为"text/javascript"。想要启用 Firefox 的 JavaScript 1.8 特性，可使用值"text/javascript;version=1.8"。

---

§ 4 进阶应用

§ 4.1 从包中加载模块

RequireJS 支持从 CommonJS 包结构中加载模块，但需要一些额外的配置。具体地，支持如下的 CommonJS 包特性：

- 一个包可以关联一个模块名/前缀。
- package config 可为特定的包指定下述属性：
  - **name**：包名（用于模块名/前缀映射）
  - **location**：磁盘上的位置。位置是相对于配置中的 baseUrl 值，除非它们包含协议或以"/"开头
  - **main**：当以"包名"发起 require 调用后，所应用的一个包内的模块。默认为"main"，除非在此处做了另外设定。该值是相对于包目录的。

重要事项：

- 虽然包可以有 CommonJS 的目录结构，但模块本身应为 RequireJS 可理解的模块格式。例外是：如果你在用 r.js Node 适配器，模块可以是传统的 CommonJS 模块格式。你可

以使用 CommonJS 转换工具来将传统的 CommonJS 模块转换为 RequireJS 所用的异步模块格式。

- 一个项目上下文中仅能使用包的一个版本。你可以使用 RequireJS 的多版本支持来加载两个不同的模块上下文；但若你想在同一个上下文中使用依赖了不同版本的包 C 的包 A 和 B，就会有问题。未来可能会解决此问题。

如果你使用了类似于入门指导中的项目布局，你的 web 项目应大致以如下的布局开始（基于 Node/Rhino 的项目也是类似的，只不过使用 scripts 目录中的内容作为项目的顶层目录）：

- project-directory/
  - project.html
  - scripts/
    - require.js

而下面的示例中使用了两个包，**cart** 及 **store**：

- project-directory/
  - project.html
  - cart/
    - main.js
  - store/
    - main.js
    - util.js
  - main.js
  - require.js

project.html 会有如下的一个 script 标签：

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

这会指示 require.js 去加载 scripts/main.js。main.js 使用“packages”配置项来设置相对于 require.js 的各个包，此例中是源码包“cart”及“store”：

```
//main.js 的内容
//传递一个 config object 到 require
```

```
require.config({
    "packages": ["cart", "store"]
});

require(["cart", "store", "store/util"],
function (cart,   store,   util) {
    //正常地使用模块
});
```

对“cart”的依赖请求会从 **scripts/cart/main.js** 中加载，因为“main”是 RequireJS 默认的包主模块。对“store/util”的依赖请求会从 **scripts/store/util.js** 加载。

如果“store”包不采用“main.js”约定，如下面的结构：

- project-directory/
  - project.html
  - scripts/
    - cart/
    - main.js
    - store/
    - store.js
    - util.js
    - main.js
    - package.json
    - require.js

则 RequireJS 的配置应如下：

```
require.config({
    packages: [
        "cart",
        {
            name: "store",
            main: "store"
        }
    ]
});
```

减少麻烦期间，强烈建议包结构遵从“main.js”约定。

§ 4.2 多版本支持

如配置项一节中所述，可以在同一页面上以不同的“上下文”配置项加载同一模块的不同版本。require.config()返回了一个使用该上下文配置的 require 函数。下面是一个加载不同版本（alpha 及 beta）模块的示例（取自 test 文件中）：

```
<script src="../require.js"></script>
<script>
var reqOne = require.config({
  context: "version1",
  baseUrl: "version1"
});

reqOne(["require", "alpha", "beta",],
function(require,   alpha,   beta) {
  log("alpha version is: " + alpha.version); //prints 1
  log("beta version is: " + beta.version); //prints 1

  setTimeout(function() {
    require(["omega"],
      function(omega) {
        log("version1 omega loaded with version: " +
            omega.version); //prints 1
      }
    );
  }, 100);
});

var reqTwo = require.config({
      context: "version2",
      baseUrl: "version2"
    });

reqTwo(["require", "alpha", "beta"],
function(require,   alpha,   beta) {
  log("alpha version is: " + alpha.version); //prints 2
  log("beta version is: " + beta.version); //prints 2

  setTimeout(function() {
    require(["omega"],
      function(omega) {
        log("version2 omega loaded with version: " +
```

```
            omega.version); //prints 2
        }
      );
    }, 100);
});
</script>
```

注意"require"被指定为模块的一个依赖，这就允许传递给函数回调的 require()使用正确的上下文来加载多版本的模块。如果"require"没有指定为一个依赖，则很可能会出现错误。

§ 4.3 在页面加载之后加载代码

上述多版本示例中也展示了如何在嵌套的 require()中迟后加载代码。

§ 4.4 对 Web Worker 的支持

从版本 0.12 开始，RequireJS 可在 Web Worker 中运行。可以通过在 web worker 中调用 importScripts()来加载 require.js（或包含 require()定义的 JS 文件），然后调用 require 就好了。

你可能需要设置 **baseUrl** 配置项来确保 require()可找到待加载脚本。
你可以在 unit test 使用的一个文件中找到一个例子。

§ 4.5 对 Rhino 的支持

RequireJS 可通过 r.js 适配器用在 Rhino 中。参见 r.js 的 README。

§ 4.6 处理错误

通常的错误都是 404（未找到）错误，网络超时或加载的脚本含有错误。RequireJS 有些工具来处理它们：require 特定的错误回调（errback），一个"paths"数组配置，以及一个全局的 requirejs.onError 事件。

传入 errback 及 requirejs.onError 中的 error object 通常包含两个定制的属性：

- requireType：含有类别信息的字串值，如"timeout"，"nodefine"， "scripterror"
- requireModules： 超时的模块名/URL 数组。

如果你得到了 requireModules 错，可能意味着依赖于 requireModules 数组中的模块的其他模块未定义。

§ 4.6.1 在 IE 中捕获加载错

Internet Explorer 有一系列问题导致检测 errbacks/paths fallbacks 中的加载错 比较困难：

- IE 6-8 中的 script.onerror 无效。没有办法判断是否加载一个脚本会导致 404 错；更甚地，在 404 中依然会触发 state 为 complete 的 onreadystatechange 事件。
- IE 9+中 script.onerror 有效，但有一个 bug：在执行脚本之后它并不触发 script.onload 事件句柄。因此它无法支持匿名 AMD 模块的标准方法。所以 script.onreadystatechange 事件仍被使用。但是，state 为 complete 的 onreadystatechange 事件会在 script.onerror 函数触发之前触发。

因此 IE 环境下很难两全其美：匿名 AMD（AMD 模块机制的核心优势）和可靠的错误检测。但如果你的项目里使用了 define()来定义所有模块，或者为其他非 define()的脚本使用 shim 配置指定了导出字串，则如果你将 enforceDefine 配置项设为 true，loader 就可以通过检查 define()调用或 shim 全局导出值来确认脚本的加载无误。
因此如果你打算支持 Internet Explorer，捕获加载错，并使用了 define()或 shim，则记得将 **enforceDefine** 设置为 true。参见下节的示例。

注意：如果你设置了 enforceDefine: true，而且你使用 data-main="" 来加载你的主 JS 模块，则该主 JS 模块**必须调用 define()**而不是 require()来加载其所需的代码。主 JS 模块仍然可调用 require/requirejs 来设置 config 值，但对于模块加载必须使用 define()。

如果你使用了 almond 而不是 require.js 来 build 你的代码，记得在 build 配置项中使用 insertRequire 来在主模块中插入一个 require 调用 —— 这跟 data-main 的初始化 require()调用起到相同的目的。

§ 4.6.2 require([]) errbacks

当与 requirejs.undef()一同使用 errback 时，允许你检测模块的一个加载错，然后 undefine 该模块，并重置配置到另一个地址来进行重试。
一个常见的应用场景是先用库的一个 CDN 版本，如果其加载出错，则切换到本地版本：

```
requirejs.config({
    enforceDefine: true,
    paths: {
        jquery: 'http://ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min'
    }
```

```
});

//Later
require(['jquery'], function ($) {
    //使用$
}, function (err) {
    //errback
    //error 含有出错的模块列表
    var failedId = err.requireModules && err.requireModules[0],
    if (failedId === 'jquery') {
        //undef 是全局的 requirejs object 上的一个函数。
        //用它来清空 jQuery 的信息。 任何依赖于 jQuery 或处于加载中的模块都不再
        //加载，它们会等待有效的 jQuery 加载完毕。
        requirejs.undef(failedId);

        //将 jQuery 设置到本地版本上
        requirejs.config({
            paths: {
                jquery: 'local/jquery'
            }
        });

        //重试。注意上述含有“使用$”一句的 require 回调会在新的
        //jQuery 加载成功后被调用。
        require(['jquery'], function () {});
    } else {
        //其他错。考虑报错给用户。
    }
});
```

使用“requirejs.undef()”，如果你配置到不同的位置并重新尝试加载同一模块，则 loader 会将依赖于该模块的那些模块记录下来并在该模块重新加载成功后去加载它们。

注意：errback 仅适用于回调风格的 require 调用，而不是 define()调用。

define()仅用于声明模块。

§ 4.6.3 paths 备错配置

上述模式（检错，undef()模块，修改 paths，重加载）是一个常见的需求，因此有一个快捷设置方式。paths 配置项允许数组值：

```
requirejs.config({
    //为了在 IE 中正确检错，强制 define/shim 导出检测
    enforceDefine: true,
    paths: {
        jquery: [
            'http://ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min',
            //若 CDN 加载错，则从如下位置重试加载
            'lib/jquery'
        ]
    }
});

//后面
require(['jquery'], function ($) {
});
```

上述代码先尝试加载 CDN 版本，如果出错，则退回到本地的 lib/jquery.js。

注意：paths 备错仅在模块 ID 精确匹配时工作。这不同于常规的 paths 配置，常规配置可匹配模块 ID 的任意前缀部分。备错主要用于非常的错误恢复，而不是常规的 path 查找解析，因为那在浏览器中是低效的。

§ 4.6.4 全局的 requirejs.onError

为了捕获在局域的 errback 中未捕获的异常，你可以重载 requirejs.onError()：

```
requirejs.onError = function (err) {
    console.log(err.requireType);
    if (err.requireType === 'timeout') {
        console.log('modules: ' + err.requireModules);
    }

    throw err;
};
```

§ 5 加载器插件

RequireJS 支持加载器插件。使用它们能够加载一些对于脚本正常工作很重要的非 JS 文件。RequireJS 的 wiki 有一个插件的列表。本节讨论一些由 RequireJS 一并维护的特定插件：

§ 5.1 指定文本文件依赖

如果都能用 HTML 标签而不是基于脚本操作 DOM 来构建 HTML，是很不错的。但没有好的办法在 JavaScript 文件中嵌入 HTML。所能做的仅是在 js 中使用 HTML 字串，但这一般很难维护，特别是多行 HTML 的情况下。
RequireJS 有个 text.js 插件可以帮助解决这个问题。如果一个依赖使用了 text!前缀，它就会被自动加载。参见 text.js 的 README 文件。

§ 5.2 页面加载事件及 DOM Ready

RequireJS 加载模块速度很快，很有可能在页面 DOM Ready 之前脚本已经加载完毕。需要与 DOM 交互的工作应等待 DOM Ready。现代的浏览器通过 DOMContentLoaded 事件来知会。
但是，不是所有的浏览器都支持 DOMContentLoaded。domReady 模块实现了一个跨浏览器的方法来判定何时 DOM 已经 ready。下载并在你的项目中如此用它：

```
require(['domReady'], function (domReady) {
  domReady(function () {
    //一旦 DOM 准备就绪，本回调就执行。
    //在此函数中查询及处理 DOM 是安全的。
  });
});
```

基于 DOM Ready 是个常规需求，像上述 API 中的嵌套调用方式，理想情况下应避免。domReady 模块也实现了 Loader Plugin API，因此你可以使用 loader plugin 语法（注意 domReady 依赖的!前缀）来强制 require()回调函数在执行之前等待 DOM Ready。当用作 loader plugin 时，domReady 会返回当前的 document：

```
require(['domReady!'], function (doc) {
    //本函数会在 DOM ready 时调用。
    //注意'domReady!'的值为当前的 document
});
```

注意：如果 document 需要一段时间来加载（也许是因为页面较大，或加载了较大的 js 脚本阻塞了 DOM 计算），使用 domReady 作为 loader plugin 可能会导致 RequireJS“超时”错。如果这是个问题，则考虑增加 waitSeconds 配置项的值，或在 require()使用 domReady()调用（将其当做是一个模块）。

§ 5.3define I18N bundle

一旦你的 web app 达到一定的规模和流行度，提供本地化的接口和信息是十分有用的，但实现一个扩展良好的本地化方案又是很繁赘的。RequireJS 允许你先仅配置一个含有本地化信息的基本模块，而不需要将所有的本地化信息都预先创建起来。后面可以将这些本地化相关的变化以值对的形式慢慢加入到本地化文件中。
i18n.js 插件提供 i18n bundle 支持。在模块或依赖使用了 i18n!前缀的形式（详见下）时它会自动加载。下载该插件并将其放置于你 app 主 JS 文件的同目录下。
将一个文件放置于一个名叫"nls"的目录内来定义一个 bundle——i18n 插件当看到一个模块名字含有"nls"时会认为它是一个 i18n bundle。名称中的"nls"标记告诉 i18n 插件本地化目录（它们应当是 nls 目录的直接子目录）的查找位置。如果你想要为你的"my"模块集提供颜色名的 bundle，应像下面这样创建目录结构：

- my/nls/colors.js

  该文件的内容应该是：

```
//my/nls/colors.js 文件内容:
define({
    "root": {
        "red": "red",
        "blue": "blue",
        "green": "green"
    }
});
```

  以一个含有"root"属性的 object 直接量来定义该模块。这就是为日后启用本地化所需的全部工作。你可以在另一个模块中，如 my/lamps.js 中使用上述模块：

```
//my/lamps.js 内容
define(["i18n!my/nls/colors"], function(colors) {
    return {
        testMessage: "The name for red in this locale is: " + colors.red
    }
});
```

  my/lamps 模块具备一个"testMessage"属性，它使用了 colors.red 来显示红色的本地化值。
  日后，当你想要为文件再增加一个特定的翻译，如 fr-fr，可以改变 my/nls/colors 内容如下：

```
//my/nls/colors.js 内容
define({
```

```
    "root": {
        "red": "red",
        "blue": "blue",
        "green": "green"
    },
    "fr-fr": true
});
```

然后再定义一个 my/nls/fr-fr/colors.js 文件，含有如下内容：

```
//my/nls/fr-fr/colors.js 的内容
define({
    "red": "rouge",
    "blue": "bleu",
    "green": "vert"
});
```

RequireJS 会使用浏览器的 navigator.language 或 navigator.userLanguage 属性来判定 my/nls/colors 的本地化值，因此你的 app 不需要更改。如果你想指定一个本地化方式，你可使用模块配置将该方式传递给插件：

```
requirejs.config({
    config: {
        //为 i18n 做配置
        //module ID
        i18n: {
            locale: 'fr-fr'
        }
    }
});
```

注意 RequireJS 总是使用小写版本的 locale 值来避免大小写问题，因此磁盘上 i18n 的所有目录和文件都应使用小写的本地化值。 RequireJS 有足够智能去选取合适的本地化 bundle，使其尽量接近 my/nls/colors 提供的那一个。例如，如果 locale 值时“en-us”，则会使用“root” bundle。如果 locale 值是“fr-fr-paris”，则会使用“fr-fr” bundle。RequireJS 也会将 bundle 合理组合，例如，若 french bundle 如下定义（忽略 red 的值）：

```
//my/nls/fr-fr/colors.js 内容：
define({
    "blue": "bleu",
    "green": "vert"
});
```

则会应用“root”下的 red 值。所有的 locale 组件是如此。如果如下的所有 bundle 都已定义，则 RequireJS 会按照如下的优先级顺序（最顶的最优先）应用值：

- my/nls/fr-fr-paris/colors.js
- my/nls/fr-fr/colors.js
- my/nls/fr/colors.js
- my/nls/colors.js

如果你不在模块的顶层中包含 root bundle，你可像一个常规的 locale bundle 那样定义它。这种情形下顶层模块应如下：

```
//my/nls/colors.js 内容:
define({
    "root": true,
    "fr-fr": true,
    "fr-fr-paris": true
});
```

root bundle 应看起来如下：

```
//my/nls/root/colors.js 内容:
define({
    "red": "red",
    "blue": "blue",
    "green": "green"
});
```