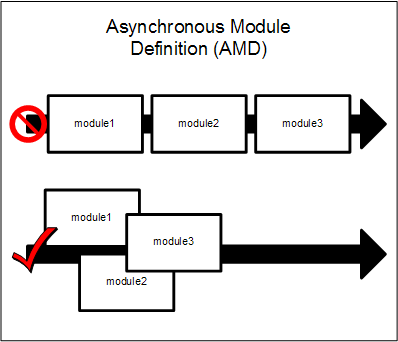
### 异步模块定义(AMD)

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](http://blogs.microsoft.co.il/blogs/gilf/archive/2011/12/29/the-async-and-defer-script-attributes-in-html5.aspx)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**](http://www.oschina.net/p/commonjs), 是对通用的JavaScript模式的标准化尝试，它包含有 [**AMD 定义**](http://wiki.commonjs.org/wiki/Modules/AsynchronousDefinition) ，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](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：

|  |  |
| --- | --- |
| 1 | <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': { | |

|  |  |  |
| --- | --- | --- |
| 11 | //The underscore script dependency should be loaded before loading backbone.js | |
| 12 | deps: ['underscore'], |

|  |  |  |
| --- | --- | --- |
| 13 | // use the global 'Backbone' as the module 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](http://www.oschina.net/translate/getting-started-with-the-requirejs-library#viewSource)

[print](http://www.oschina.net/translate/getting-started-with-the-requirejs-library#printSource)[?](http://www.oschina.net/translate/getting-started-with-the-requirejs-library#about)

|  |  |
| --- | --- |
| 1 | require(['jquery'], function ($) { |
| 2 | //jQuery was loaded and can be used now | |

|  |  |
| --- | --- |
| 3 | }); |

### 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。

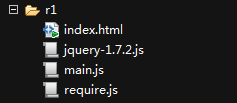
## [RequireJS入门（一）](http://www.cnblogs.com/snandy/archive/2012/05/22/2513652.html)

RequireJS由[James Burke](http://tagneto.blogspot.com/)创建，他也是[AMD](https://github.com/amdjs/amdjs-api/wiki/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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <!doctype html>  <html>      <head>          <title>requirejs入门（一）</title>          <meta charset="utf-8">          <script data-main="main" src="require.js"></script>      </head>      <body>        </body>  </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  2  3  4  5  6  7  8  9 | require.config({      paths: {          jquery: 'jquery-1.7.2'      }  });    require(['jquery'], function($) {      alert($().jquery);  }); |

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](http://requirejs.org/docs/api.html#config)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

|  |  |
| --- | --- |
| 1  2  3 | 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

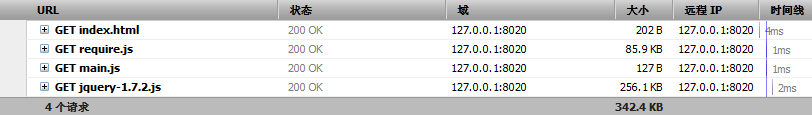
|  |  |
| --- | --- |
| 1  2 | // Expose jQuery to the global object  window.jQuery = window.$ = jQuery; |

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



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](http://www.cnblogs.com/snandy/archive/2012/05/23/2513712.html) 。

[r1.zip](http://files.cnblogs.com/snandy/r1.zip)



## [RequireJS入门（二）](http://www.cnblogs.com/snandy/archive/2012/05/23/2513712.html)

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <!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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | 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.length)) :                  context;            if(regId.test(s)) {              return doc.getElementById(s.substr(1,s.length));          }          // 略...      }        return query;  }); |
|  |  |

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。

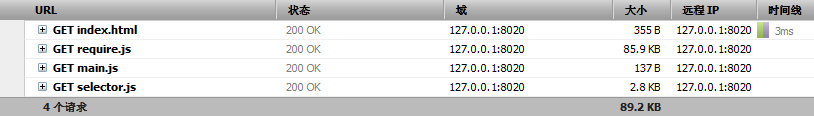
main.js 如下

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

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。

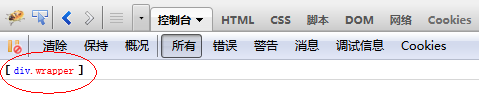
把目录r2放到apache或其它web服务器上，访问index.html。

网络请求如下



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](http://files.cnblogs.com/snandy/r2.zip)



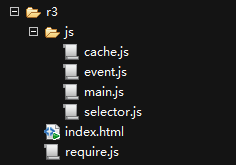
## [RequireJS入门（三）](http://www.cnblogs.com/snandy/archive/2012/05/24/2514700.html)

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](http://www.cnblogs.com/snandy/archive/2011/06/10/2077298.html)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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <!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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | 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 element');              }                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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | 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, false); } :              function(el, type, fn) { el.attachEvent('on' + type, fn); },          removeListener = w3c ?              function(el, type, fn) { el.removeEventListener(type, fn, false); } :              function(el, type, fn) { el.detachEvent('on' + type, fn); };        // 略去...        return {          bind : bind,          unbind : unbind,          trigger : trigger      };  }); |

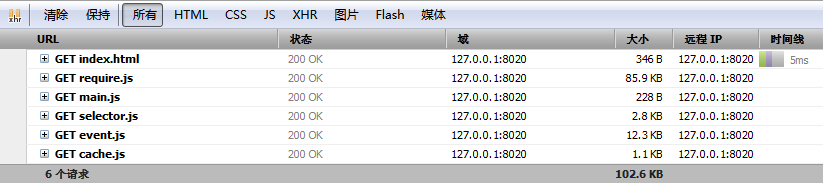
event依赖于cache，定义时第一个参数数组中放入“cache”即可。第二个参数是为函数类型，它的参数就是cache模块对象。  
这样定义后，当require事件模块时，requirejs会自动将event依赖的cache.js也下载下来。

main.js 如下

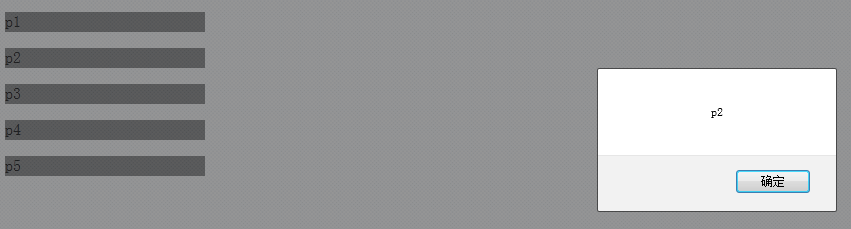
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | require.config({      baseUrl: 'js'  });    require(['selector', 'event'], function($, E) {      var els = $('p');      for (var i=0; i<els.length; i++) {          E.bind(els[i], 'click', function() {              alert(this.innerHTML);          });      }  }); |

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](http://files.cnblogs.com/snandy/r3.zip)



[**RequireJS 2.0 正式发布**](http://www.cnblogs.com/snandy/archive/2012/06/04/2532997.html)

就在前天晚上RequireJS发布了一个大版本，直接从version[1.0.8](http://requirejs.org/docs/1.0/)升级到了[2.0](http://requirejs.org/docs/api.html)。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](https://github.com/jrburke/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](http://www.cnblogs.com/snandy/archive/2012/03/12/2390782.html)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](http://requirejs.org/docs/api.html#order)插件。其实在1.0版本中就曾经有人开发过[use](https://github.com/tbranyen/use.js/)和[wrap](https://github.com/geddesign/wrapjs)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  2  3  4  5  6 | require.config({      shim: {          'jquery-slide': ['jquery']      }  });  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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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

|  |  |
| --- | --- |
| 1  2  3  4 | 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

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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](https://github.com/cujojs/curl)的作者[John Hann](https://github.com/unscriptable)。

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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](http://livedocs.dojotoolkit.org/loader/amd)，RequireJS取名undef。如下

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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。

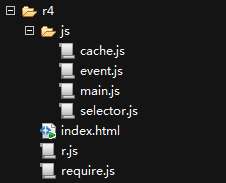
## [RequireJS进阶（一）](http://www.cnblogs.com/snandy/archive/2012/06/06/2536969.html)

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](http://requirejs.org/docs/download.html#rjs)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](https://github.com/mishoo/UglifyJS)或[Closure Compiler](http://code.google.com/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](http://www.cnblogs.com/snandy/archive/2012/05/24/2514700.html)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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <!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>          <p>p1</p><p>p2</p><p>p3</p><p>p4</p><p>p5</p>          <script data-main="built" src="require.js"></script>      </body>  </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。

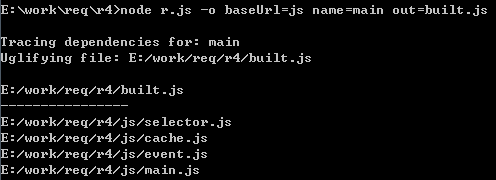
好，让我们开始合并压缩吧。

1，打开windows命令窗口，cd到r4目录中

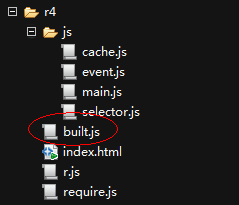


2，输入命令

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

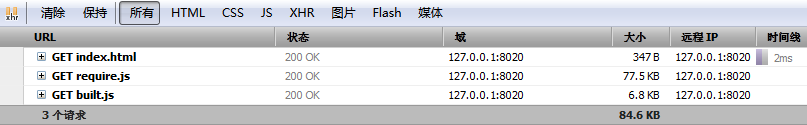


命令行信息可以看到已经将各个js文件合并成功了。这时回到r4目录会发现多了一个built.js文件。

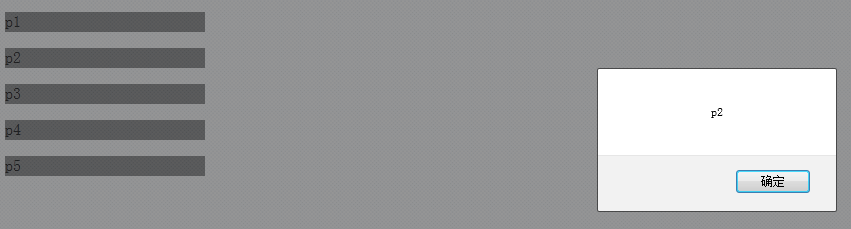


好了，合并压缩过程完成了。

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。

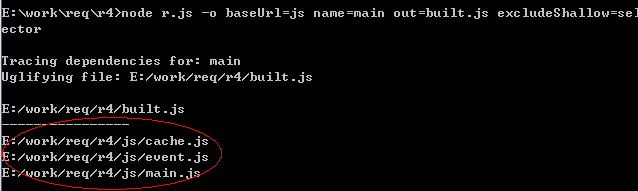
下面对命令行做个简单解释。  
node r.js -o baseUrl=js name=main out=built.js

-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

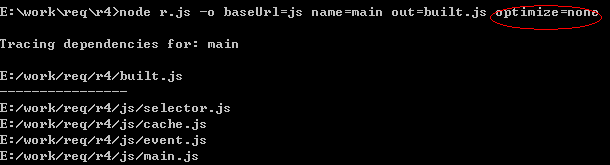
node r.js -o baseUrl=js name=main out=built.js excludeShallow=selector



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



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](http://files.cnblogs.com/snandy/r4.zip)



# RequireJS和AMD规范

## 目录

* [**概述**](http://javascript.ruanyifeng.com/tool/requirejs.html#toc0)
* [**define方法：定义模块**](http://javascript.ruanyifeng.com/tool/requirejs.html#toc1)
* [**require方法：调用模块**](http://javascript.ruanyifeng.com/tool/requirejs.html#toc2)
* [**AMD模式小结**](http://javascript.ruanyifeng.com/tool/requirejs.html#toc3)
* [**配置require.js：config方法**](http://javascript.ruanyifeng.com/tool/requirejs.html#toc4)
* [**插件**](http://javascript.ruanyifeng.com/tool/requirejs.html#toc5)
* [**优化器r.js**](http://javascript.ruanyifeng.com/tool/requirejs.html#toc6)
* [**参考链接**](http://javascript.ruanyifeng.com/tool/requirejs.html#toc7)

## 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](https://github.com/amdjs/amdjs-api/wiki/AMD)（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/?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) {**

**// ...**

**};**

### 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](https://github.com/jrburke/requirejs/wiki/Plugins)。

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](https://github.com/jrburke/r.js/blob/master/build/example.build.js)。

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](http://www.webdeveasy.com/optimize-requirejs-projects/)
* Jonathan Creamer, [Deep dive into Require.js](http://tech.pro/tutorial/1300/deep-dive-into-requirejs)
* Addy Osmani, [Writing Modular JavaScript With AMD, CommonJS & ES Harmony](http://addyosmani.com/writing-modular-js/)
* Jim Cowart, [Five Helpful Tips When Using RequireJS](http://tech.pro/blog/1561/five-helpful-tips-when-using-requirejs)
* Jim Cowart, [Using r.js to Optimize Your RequireJS Project](http://tech.pro/blog/1639/using-rjs-to-optimize-your-requirejs-project)

# RequireJS API中文版

PRE: 3  
update: 2013-05-13

这是 [RequireJS 2.0](https://github.com/jrburke/requirejs/wiki/Upgrading-to-RequireJS-2.0) 的API，对应的[官方文档](http://requirejs.org/docs/api.html)版本号是2.1.6

目录

* [用法 ------------------------------------------------ §§ 1-1.2](http://makingmobile.org/docs/tools/requirejs-api-zh/#usage)
  + [加载JavaScript文件 ------------------------------- § 1.1](http://makingmobile.org/docs/tools/requirejs-api-zh/#jsfiles)
  + [定义模块 ----------------------------------------- § 1.2](http://makingmobile.org/docs/tools/requirejs-api-zh/#define)
    - [简单的值对 ----------------------------------- § 1.2.1](http://makingmobile.org/docs/tools/requirejs-api-zh/#defsimple)
    - [函数式定义 ----------------------------------- § 1.2.2](http://makingmobile.org/docs/tools/requirejs-api-zh/#deffunc)
    - [存在依赖的函数式定义 -------------------------- § 1.2.3](http://makingmobile.org/docs/tools/requirejs-api-zh/#defdep)
    - [将模块定义为一个函数 -------------------------- § 1.2.4](http://makingmobile.org/docs/tools/requirejs-api-zh/#funcmodule)
    - [简单包装CommonJS来定义模块 -------------------- § 1.2.5](http://makingmobile.org/docs/tools/requirejs-api-zh/#cjsmodule)
    - [定义一个命名模块 ------------------------------ § 1.2.6](http://makingmobile.org/docs/tools/requirejs-api-zh/#modulename)
    - [其他注意事项 --------------------------------- § 1.2.7](http://makingmobile.org/docs/tools/requirejs-api-zh/#modulenotes)
    - [循环依赖 ------------------------------------- § 1.2.8](http://makingmobile.org/docs/tools/requirejs-api-zh/#circular)
    - [JSONP服务依赖 -------------------------------- § 1.2.9](http://makingmobile.org/docs/tools/requirejs-api-zh/#jsonp)
    - [undefine一个模块 ----------------------------- § 1.2.10](http://makingmobile.org/docs/tools/requirejs-api-zh/#undef)
* [机理 ------------------------------------------------ §§ 2](http://makingmobile.org/docs/tools/requirejs-api-zh/#mechanics)
* [配置项 ---------------------------------------------- § 3](http://makingmobile.org/docs/tools/requirejs-api-zh/#config)
* [进阶应用 --------------------------------------------- §§ 4-4.6](http://makingmobile.org/docs/tools/requirejs-api-zh/#advanced)
  + [从包中加载模块 ------------------------------------ § 4.1](http://makingmobile.org/docs/tools/requirejs-api-zh/#packages)
  + [多版本支持 --------------------------------------- § 4.2](http://makingmobile.org/docs/tools/requirejs-api-zh/#multiversion)
  + [在页面加载之后加载代码 ---------------------------- § 4.3](http://makingmobile.org/docs/tools/requirejs-api-zh/#afterload)
  + [对Web Worker的支持 ------------------------------ § 4.4](http://makingmobile.org/docs/tools/requirejs-api-zh/#webworker)
  + [对Rhino的支持 ----------------------------------- § 4.5](http://makingmobile.org/docs/tools/requirejs-api-zh/#rhino)
  + [处理错误 ---------------------------------------- § 4.6](http://makingmobile.org/docs/tools/requirejs-api-zh/#errors)
    - [在IE中捕获加载错 ----------------------------- § 4.6.1](http://makingmobile.org/docs/tools/requirejs-api-zh/#ieloadfail)
    - [require([]) errbacks ----------------------- § 4.6.2](http://makingmobile.org/docs/tools/requirejs-api-zh/#errbacks)
    - [paths备错配置 ------------------------------- § 4.6.3](http://makingmobile.org/docs/tools/requirejs-api-zh/#pathsfallbacks)
    - [全局的 requirejs.onError -------------------- § 4.6.4](http://makingmobile.org/docs/tools/requirejs-api-zh/#requirejsonerror)
* [加载器插件 ------------------------------------------- §§ 5-5.3](http://makingmobile.org/docs/tools/requirejs-api-zh/#plugins)
  + [指定文本文件依赖 ---------------------------------- § 5.1](http://makingmobile.org/docs/tools/requirejs-api-zh/#text)
  + [页面加载事件及DOM Ready --------------------------- § 5.2](http://makingmobile.org/docs/tools/requirejs-api-zh/#pageload)
  + [define I18N bundle ------------------------------ § 5.3](http://makingmobile.org/docs/tools/requirejs-api-zh/#i18n)

# § 1用法

## § 1.1[加载JavaScript文件](http://makingmobile.org/docs/tools/requirejs-api-zh/#jsfiles)

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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-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](http://makingmobile.org/docs/tools/requirejs-api-zh/#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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-paths)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](https://github.com/volojs/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()](http://makingmobile.org/docs/tools/requirejs-api-zh/#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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-shim)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](http://makingmobile.org/docs/tools/requirejs-api-zh/#define)

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](http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html)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](http://makingmobile.org/docs/tools/requirejs-api-zh/#multiversion)。

(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](http://requirejs.org/docs/optimization.html)将其组织打包。

### § 1.2.1 [简单的值对](http://makingmobile.org/docs/tools/requirejs-api-zh/#defsimple)

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](http://makingmobile.org/docs/tools/requirejs-api-zh/#deffunc)

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](http://makingmobile.org/docs/tools/requirejs-api-zh/#defdep)

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](http://makingmobile.org/docs/tools/requirejs-api-zh/#funcmodule)

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](http://makingmobile.org/docs/tools/requirejs-api-zh/#cjsmodule)

如果你现有一些以[CommonJS模块格式](http://wiki.commonjs.org/wiki/Modules/1.1.1)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](http://requirejs.org/docs/commonjs.html)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](http://requirejs.org/docs/optimization.html)将依赖导出为数组形式。

更多的信息可参看[CommonJS Notes](http://requirejs.org/docs/commonjs.html)页面，以及"[Why AMD](http://requirejs.org/docs/whyamd.html)"页面的["Sugar"](http://requirejs.org/docs/whyamd.html#sugar)段落。

### § 1.2.6 [定义一个命名模块](http://makingmobile.org/docs/tools/requirejs-api-zh/#modulename)

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](http://requirejs.org/docs/optimization.html)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，加快到浏览器的载人速度。

### § 1.2.7 [其他注意事项](http://makingmobile.org/docs/tools/requirejs-api-zh/#modulenotes)

**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](http://requirejs.org/docs/optimization.html)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](http://requirejs.org/docs/commonjs.html)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](http://makingmobile.org/docs/tools/requirejs-api-zh/#circular)

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](https://github.com/jrburke/requirejs/wiki/Differences-between-the-simplified-CommonJS-wrapper-and-standard-AMD-define#wiki-magic)"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](http://makingmobile.org/docs/tools/requirejs-api-zh/#jsonp)

[JSONP](http://en.wikipedia.org/wiki/JSON#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](http://makingmobile.org/docs/tools/requirejs-api-zh/#errors)部分。

### § 1.2.10 [undefine一个模块](http://makingmobile.org/docs/tools/requirejs-api-zh/#undef)

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)](http://makingmobile.org/docs/tools/requirejs-api-zh/#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](https://github.com/jrburke/requirejs/wiki/Internal-API:-onResourceLoad)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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-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](http://makingmobile.org/docs/tools/requirejs-api-zh/#jsfiles))，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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-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()](http://makingmobile.org/docs/tools/requirejs-api-zh/#modulenotes-urls)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](http://makingmobile.org/docs/tools/requirejs-api-zh/#pathsfallbacks))，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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-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'exports'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](http://requirejs.org/docs/optimization.html#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](http://makingmobile.org/docs/tools/requirejs-api-zh/#map-config): 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](http://makingmobile.org/docs/tools/requirejs-api-zh/#multiversion)实现中很难做到这一点。而且，[paths配置](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-paths)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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-moduleconfig)：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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-packages)：从CommonJS包(package)中加载模块。参见[从包中加载模块](http://makingmobile.org/docs/tools/requirejs-api-zh/#packages)。

[waitSeconds](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-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](http://makingmobile.org/docs/tools/requirejs-api-zh/#multiversion)一节。

[deps](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-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](http://makingmobile.org/docs/tools/requirejs-api-zh/##ieloadfail)一节。

[xhtml](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-xhtml)：如果设置为true，则使用document.createElementNS()去创建script元素。

[urlArgs](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-baseUrl)：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](http://makingmobile.org/docs/tools/requirejs-api-zh/#packages)

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](http://makingmobile.org/docs/tools/requirejs-api-zh/#multiversion)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](http://requirejs.org/docs/start.html)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](http://makingmobile.org/docs/tools/requirejs-api-zh/#multiversion)

如[配置项](http://makingmobile.org/docs/tools/requirejs-api-zh/#config)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](http://makingmobile.org/docs/tools/requirejs-api-zh/#afterload)

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](http://makingmobile.org/docs/tools/requirejs-api-zh/#webworker)

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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config)来确保require()可找到待加载脚本。  
你可以在[unit test](https://github.com/jrburke/requirejs/blob/master/tests/workers.js)使用的一个文件中找到一个例子。

## § 4.5[对Rhino的支持](http://makingmobile.org/docs/tools/requirejs-api-zh/#rhino)

RequireJS可通过[r.js适配器](http://requirejs.org/docs/download.html#rjs)用在Rhino中。参见r.js的README。

## § 4.6[处理错误](http://makingmobile.org/docs/tools/requirejs-api-zh/#errors)

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](http://makingmobile.org/docs/tools/requirejs-api-zh/#ieloadfail)

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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-shim)配置指定了导出字串，则如果你将[enforceDefine](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-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](https://github.com/jrburke/almond)而不是require.js来build你的代码，记得在build配置项中使用[insertRequire](https://github.com/jrburke/r.js/blob/master/build/example.build.js#L289)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](http://makingmobile.org/docs/tools/requirejs-api-zh/#errbacks)

当与[requirejs.undef()](http://makingmobile.org/docs/tools/requirejs-api-zh/#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](http://makingmobile.org/docs/tools/requirejs-api-zh/#pathsfallbacks)

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](http://makingmobile.org/docs/tools/requirejs-api-zh/#requirejsonerror)

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](http://requirejs.org/docs/plugins.html)。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](https://github.com/jrburke/requirejs/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](http://makingmobile.org/docs/tools/requirejs-api-zh/#text)

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](https://github.com/requirejs/text)。

## § 5.2[页面加载事件及DOM Ready](http://makingmobile.org/docs/tools/requirejs-api-zh/#pageload)

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](http://requirejs.org/docs/download.html#domReady)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](http://requirejs.org/docs/plugins.html)，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](http://makingmobile.org/docs/tools/requirejs-api-zh/#config-waitSeconds)配置项的值，或在require()使用domReady()调用（将其当做是一个模块）。

## § 5.3[define I18N bundle](http://makingmobile.org/docs/tools/requirejs-api-zh/#i18n)

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贅的。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](http://requirejs.org/docs/download.html#i18n)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"

});