REQUIREJS 2.0 API

用法

加载 JS 文件

RequireJS 采用了不同于传统<script>标签的方式加载 script。它的目标是鼓励模块化代码。虽然它运行的很快，能够足够的优化，但是它主要的目标是鼓励模块化代码。作为其中的一部分，它鼓励使用模块 ID，而不是 script 标签的 URL。

RequireJS 加载的所有代码都是相对于 baseUrl 的，baseUrl 通常设置成和最外层页面加载的 script 标签的 data-main 属性所在的相同目录。Data-main 属性是一个特殊的属性，require.js 将检查初始脚本的加载。

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

或者可以通过 RequireJS config 对象手动设置 baseUrl，如果没有准确的 config 对象和没有使用 data-main 属性，那么默认的 baseUrl 是运行 RequireJS 的 HTML 页面目录。

RequireJS 默认认为所有的依赖都是脚本文件，因此不希望看到有.js 后缀的模块 ID。RequireJS 将模块 ID 转换成路径时会自动加上它。在 paths config，你可以指定一组脚本位置。和传统<script>标签相比，所有的这些特性都将使用更少的脚本字符串。

有时候你想直接一个脚本，而不是符合'baseUrl+paths'的规则。如果一个模块 ID 符合如下规则，这个 ID 将不传递到‘baseUrl+paths’配置中，而是直接作为一个相对于文档的正常的 url

- 以.js 结尾的
- 以‘/’开头的
- 包含 URL 协议的，例如 http，https 等

总体上来说，这是最好给模块使用 baseUrl 和 paths 配置。通过这个，它在重命名和给优化构建配置不同的路径提供了更多的可移植性。

类似的，为了避免一堆配置，最好避免那些深层次的目录结构脚本，相反吧所有脚本在 baseUrl 中，或者和你的应用代码分离开放到 library/vendor-supplied 中，目录就像这样:

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

index.html

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

app.js

```
requirejs.config({
    //By default load any module IDs from js/lib
    baseUrl: 'js/lib',
    //except, if the module ID starts with "app",
    //load it from the js/app directory. paths
    //config is relative to the baseUrl, and
    //never includes a ".js" extension since
    //the paths config could be for a directory.
    paths: {
        app: '../app'
    }
});

// Start the main app logic.
requirejs(['jquery', 'canvas', 'app/sub'],
function   ($,        canvas,   sub) {
    //jQuery, canvas and the app/sub module are all
    //loaded and can be used here now.
});
```

注意：上面的示例中，想 jQuery 的第三方类库文件名中没有版本号。推荐将版本信息放到一个单独的文本中，如果你想跟踪，或者使用下 volo 这样的工具，他将建一个包含版本信息的 package.json 文件，而磁盘上仍然叫 jquery.js。这个允许你有一个最小的配置，而不是为每个库在 paths 配置中放一个实体。例如，将 jquery 配置成 jquery-1.7.2。

更完美地，加载的那些脚本都是欧诺个过调用 define()定义的。然而，你可以使用传统的浏览器全局对象，通过 define()没有申明他们的依赖。对于这些，你可以使用 shim config 来申明它们的依赖。

如果你不申明依赖，有可能加载失败，因为 RequireJS 加载脚本是异步的，而且为了速度是无序的。

数据主入口点

Data-main 是一个特殊的属性，require.js 将检查启动时加载的脚本。

```
<!--when require.js loads it will inject another script tag
    (with async attribute) for scripts/main.js-->
<script data-main="scripts/main" src="scripts/require.js"></script>
```

通常使用一个 data-main 脚本来设置配置选项， 那么就会加载第一个应用模块。

注意：require.js 的 script 标签会为 data-main 的模块包含一个 async 属性。这意味着你不能认为 data-main 脚本的加载和执行将优先于该页面上的其他脚本。

例如：**当没有 require.config 中 foo 模块配置，并且在优先于 require()之后，这种写法可能会出现随机的失败，**

```
// contents of main.js:
require.config({
    paths: {
        foo: 'libs/foo-1.1.3'
    }
});
```

```
// contents of other.js:

// This code might be called before the require.config() in main.js
// has executed. When that happens, require.js will attempt to
// load 'scripts/foo.js' instead of 'scripts/libs/foo-1.1.3.js'
require( ['foo'], function( foo ) {

});
```

通过改变使用的路径查找依赖的信息，请参阅"配置选项"章节。

虽然你可以在 HTML 文件<script>标签中使用 require()，但是我们强烈推荐你将其放到一个 RequireJS 加载后的可执行文件中。这样就可以更容易地过优化工具优化通，同时这种模式在 HTML 中使用是更加简洁的。

上面说所的示例结构如下：

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

注意：

在 RequireJS 0.23 中 data-main 的路径规则已经改变。在之前的版本中，上面的示例应改成 data-main='main'

data-main 属性通知 RequireJS 取里面的值，相当于调用 reqire([])一样。因此，在这种情况下，它会加载 scripts/ main.js ，并且该文件应具有最高优先级的调用：

```
//Inside scripts/main.js
require(["some/module", "a.js", "b.js"], function(someModule) {
    //...
```

```
});
```

data-main 元素所在脚本自然的成为了 URL 根路径（在 RequireJS 中成为 baseUrl），通过模块命名规则来加载其他脚本，这些规则不是以".js"后缀结尾的，在上述示例中'some/module'是采用的模块命名规则，而'a.js''b.js'不是。

由于使用了 data-main 元素，main.js 在 scripts 目录下，RequireJS 中的 baseUrl 变成了 scripts 目录。"some/module"脚本将映射在相对于 scripts 目录下。因此，它在 scripts/some/module.js

请注意，和第一个示例不同的是"some/module"是在 some/module.js 下。如果没有使用 data-main，同时在 RequireJS 配置选项中没有显示传递 baserUrl 值，那么 baseUrl 的默认值是加载 RequireJS 的 HTML 页面的所在目录。

脚本名称以常规的.js 结尾，或以'/'开头，或以协议在其中总是直接映射到 HTML 页面中，而不是相对于 baseUrl，就是 HTML <script> scr="" 中的值。你可以使用"模块命名规则"，即使脚本中没有定义模块，既然那样它仅仅意味着路径是相对于 baseUrl 而言。

定义模块

模块不同于传统意义上的脚本文件，因为它定义了一个良好作用域的对象，能够避免污染全局的命名空间。它能够明确地列出其依赖，和全局对象无关地操作这些依赖，相反的，接收这些依赖作为定义模块函数的参数。RequireJS 中的模块是模块模式的一个扩展，具有不需要参考其他模块的全局设置的优势。

对于模块的 RequireJS 语法允许它们尽可能快的被加载，甚至是无序的，但是能够以正确的依赖顺序执行，由于没有创建全局变量，所以说在一个页面中加载多个版本的模块成为了可能。

（如果你熟悉或正在使用 CommonJS 模块，那么你也可以参考下在 CommonJS 注解中的关于如何将 RequireJS 模块映射到 CommonJS 模块中的相关资料）

磁盘上的每个文件应当只有一个模块定义。各个模块可以通过优化工具划分为优化的包。

简单键值对

如果模块不依赖于任何模块，同时只是传递一些 name/value 对，那么只是传递了一个原始的对象给 define():

```
//Inside file my/shirt.js:
```

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

定义函数

如果模块不依赖于任何模块，但是需要使用函数做一些设置工作，那么定义它自己，传递一个函数给 define():

```
//my/shirt.js now does setup work
//before returning its module definition.
define(function () {
    //Do setup work here

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

定义带依赖的函数

如果模块有依赖，那么第一个参数应该是依赖名称的数组集合，第二个参数应该是一个定义的函数。一旦所有依赖都已加载，那么将调用这个函数来定义模块。这个函数应该返回一个定义模块的对象。这些依赖将作为参数传递给定义的参数，同时按照依赖顺序在参数中列出来。

```
//my/shirt.js now has some dependencies, a cart and inventory
//module in the same directory as shirt.js
define(["./cart", "./inventory"], function(cart, inventory) {
        //return an object to define the "my/shirt" module.
        return {
            color: "blue",
            size: "large",
            addToCart: function() {
                inventory.decrement(this);
                cart.add(this);
            }
        }
    }
```

```
);
```

在上述案例中，my/shirt 模块已经创建。它依赖于 my/cart 和 my/inventory。磁盘上文件结构是这样的：

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

上述函数调用中指定了两个参数，’cart’ ’inventory’.这些代表了’./car’ ‘./inventory’模块。

上述函数直到 my/cart,my/inventory 模块加载完成后才被调用，它接收模块作为 cart 和 inventory 参数

定义全局的模块是明确不鼓励使用的，以便一个模块的多个版本能够在一个页面上同时存在（参考高级用法）。同时函数的参数顺序应该和依赖的顺序相匹配。

函数调用的返回值定义了’my/shirt’模块。通过这种方式定义模块，’my/shirt’并不是作为一个全局对象而存在的

定义一个模块作为一个函数

模块没有必要一定返回对象。函数中的任何合法返回值都是允许的。下面是一个返回一个函数作为它的模块定义的模块示例：

```
//A module definition inside foo/title.js. It uses
//my/cart and my/inventory modules from before,
//but since foo/bar.js is in a different directory than
//the "my" modules, it uses the "my" in the module dependency
//name to find them. The "my" part of the name can be mapped
//to any directory, but by default, it is assumed to be a
//sibling to the "foo" directory.
define(["my/cart", "my/inventory"],
    function(cart, inventory) {
        //return a function to define "foo/title".
        //It gets or sets the window title.
        return function(title) {
            return title ? (window.title = title) :
                    inventory.storeName + ' ' + cart.name;
        }
    }
);
```

用简单的 CommonJS 包装器定义模块

如果你希望继续使用用 CommonJS 模块格式的的代码，在上述情形下可能很难正常工作，你可能更喜欢对依赖关系直接用局部变量代替依赖的名称。在这种情形下，你可以使用简单的 CommonJS 包装器

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

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

这个包装器依赖 Function.prototype.toString()返回一个函数内容的有效字符串。在一些像 PS3、老版本 Opera 移动浏览器上不能正常运行。如果想在这些设备上运行，可以使用优化工具移除这些以数组形式的依赖。

CommonJS 页面和 “为什么使用 AMD 页面的语法糖章节” 的上有更多的详细信息。

自定义名称方式定义模块

您可能会遇到的一些 define()函数调用，函数的第一个参数是模块名称：

```
//Explicitly defines the "foo/title" module:
    define("foo/title",
        ["my/cart", "my/inventory"],
        function(cart, inventory) {
            //Define foo/title object in here.
        }
    );
```

这些通常是由优化工具生成的。你可以明确地命名模块名称，但是这样会导致模块的可移植性很差。假如你把文件移到另外一个目录，这时你需要修改名称。最好的实践是避免编码模块名称，让优化工具自行优化模块名。优化工具需要添加模块名，以便在一个文件中可以绑定多个模块，这样在浏览器中能够加载的更快。

其他模块说明

- **一个文件一个模块：**

按照模块到文件路径查找算法，每个 JavaScript 文件应该只定义一个模块。优化工具会将多个模块分组到优化的文件中，但是你只能通过优化工具将多个模块放到一个文件中。

- **Define()相对模块命名：**

在 define()函数内部调用 require('./relative/name')之前，确保'require'已经作为一个依赖，只有这样相对名称才能被正确的解析：

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

或更好的，对于解析 CommonJS 模块使用简洁的语法是可行的。

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

这种形式，将使用 Function.prototype.toStrin()找到 require()调用，并把它们添加依赖数组中，由于"require"已加载 ，因此这样的代码会正常解析相对路径。

相对路径真的非常有用，如果你在一个目录下创建了一些模块，那么你可以好其他人或其他工程共享这些目录，同时你能够在那个目录下的兄弟模块获取到一个句柄，而不需要知道目录的名称。

- **生成相对路径模块的 URL**

你可能需要生成相对于模块的 URL。要做到这一点，将'require'作为依赖，使用 require.toUrl()生成 URL：

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

- **控制台代码调试**

如果你需要在 JavaScript 控制台和一个已经通过 require(["module/name"], function(){})加载好的模块进行调试，那么你需要使用 require()函数通过模块名查询它

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

注意：这个只有先通过 require 异步加载模块 require(['module/name'])才有效。如果使用的相对路径，比如'./module/name'，这类的只有在 define 内部有效。

循环依赖

如果你定义了一循环依赖（比如 a 模块需要 b 模块，同时 b 模块需要 a 模块），那么在这样情形下，当调用 b 模块函数时，它取到 a 的值为 undefined。B 模块能够查询到通过 require() 后定义的模块（确保指定 require 作为一个依赖，那么可以使用正确的上下文查找 a）。

```
//Inside b.js:
define(["require", "a"],
    function(require, a) {
        //"a" in this case will be null if a also asked for b,
        //a circular dependency.
        return function(title) {
            return require("a").doSomething();
        }
    }
);
```

正常情况下你不应该需要使用 require()查询一个模块，相反，依赖的模块应该作为一个参数传递到函数中。循环依赖是不多见的，通常情况下是你需要重新思考设计的一个标志。然而，有时候这些是必须的，在这中情形下，使用上述的 require()

如果你熟悉 CommonJS 模块，你可以使用 exports 为模块创建一个空的对象，从而可以被其他模块立即使用。通过在一个循环依赖两侧做这个，那么你可以安全地保存到其他模块中。这个只有每个模块导出的模块返回值是一个对象时才有效，而非一个函数：

```
//Inside b.js:
define(function(require, exports, module) {
    //If "a" has used exports, then we have a real
    //object reference here. However, we cannot use
    //any of a's properties until after b returns a value.
    var a = require("a");

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

或者，如果你使用的是依赖数组的方式，可以参考 exports 依赖

```
//Inside b.js:
define(['a', 'exports'], function(a, exports) {
```

```
    //If "a" has used exports, then we have a real
    //object reference here. However, we cannot use
    //any of a's properties until after b returns a value.

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

指定一个 JSONP 服务依赖

JSONP 是一种在 JavaScript 中调用服务的方法。它能够跨域使用，它是一种通过<script>标签发送一个 HTTP GET 请求来调用服务的既定途径。

为了在 RequireJS 中使用 JSONP，指定'define'作为回调参数 callback 的值。这意味这你可以从 JSONP URL 就像一个模块定义一样中获取值。

下面是一段调用 JSONP API 端点的示例。在这个例子中，

```
require(["http://example.com/api/data.json?callback=define"],
    function (data) {
        //The dta object will be the API response for the
        //JSONP data call.
        console.log(data);
    }
);
```

这种情况下 JSONP 的用法，应该系统初始化时设置。如果 JSONP 服务调用超时，那么意味着通过 define()定义的模块都不会执行，因此错误的处理不是很健壮。

目前只支持返回值是 JSON 对象的 JSONP，不支持数组，字符串，或数字。

这样的功能不能用于处理实时的、长连接的 JSONP API。这类的 API 应当在接受到请求后进行清除，因为 RequireJS 只做了一次 JSONP URL 查询操作，在 require() 或者 define()中调用相同的 URL 依赖将从缓存中取值。

通过设置超时时间，经常遇到 JSONP 服务加载出错，因此加载<script>标签时并没有给出太多的关于连接网络的问题。为了检测错误信息，你可以覆盖 requirejs.onError()方法获取更多的错误信息。更多详细信息参考错误处理章节

取消一个模块定义

有个全局函数 require.undef()，允许取消一个模块定义。它将充值加载器的内部状态，废弃该模块旧的定义。

但是，它不会从其他已经定义的模块中移除模块，当他们执行的时候有个模块句柄依赖。

因此，在报错的情况下是非常有用到的，当其他模块已经取到模块的句柄，或者作为未加载模块的一部分可能会用到这个模块。可以参考 errback section 中的示例。

如果你想为取消定义做更多复杂的依赖图分析，半公开的 onResourceLoad API 可能有用

机制原理

RequireJS 使用 head.appendChild()加载每一个依赖作为一个 script 标签。

RequireJS 等待所有的依赖加载完成，计算出各个模块调用执行顺序，然后依照这个顺序调用各个模块的调用函数。

在可以同步加载的服务端 JS 环境中使用 RequireJS，应该和重新定义 requrie.load()一样简单。构建系统也是如此，环境中 require.load()方法可以在 build/jslib/requirePatch.js 中找到。

配置选项

在最外层 HTML 页面（或不含模块的最顶层脚本文件）中使用 requre()，可以传递一个配置项对象作为第一个参数：

```
<script type="text/javascript" src="scripts/require.js"></script>
<script type="text/javascript">
  require.config({
    baseUrl: "/another/path",
    paths: {
        "some": "some/v1.0"
    },
    waitSeconds: 15,
    locale: "fr-fr"
  });
  require( ["some/module", "my/module", "a.js", "b.js"],
    function(someModule,      myModule) {
        //This function will be called when all the dependencies
        //listed above are loaded. Note that this function could
        //be called before the page is loaded.
        //This callback is optional.
    }
  );
</script>
```

或者，在加载 require.js 之前，你可以定义一个 require 对象，这些值将会生效。下面的

示例中，一旦 require.js

```
<script type="text/javascript">
    var require = {
        deps: ["some/module1", "my/module2", "a.js", "b.js"],
        callback: function(module1, module2) {
            //This function will be called when all the dependencies
            //listed above in deps are loaded. Note that this
            //function could be called before the page is loaded.
            //This callback is optional.
        }
    };
</script>
<script type="text/javascript" src="scripts/require.js"></script>
```

注意：最好使用 var require = {};不要使用 window.require = {};这个在 IE 下不能正确执行

支持的配置项：

baseUrl：所有模块查询的根路径。因此在上述的示例中，‘my/module’的<script>标签的 src 值为‘/another/path/my/module.js’。当加载后缀为.js 文件（像以斜杠开头、协议开头，.js 结尾的依赖字符串）时不能使用 baseUrl，像 a.js,b.js 等直接使用的字符串将被从包含上述片段的 HTML 页面相同目录结构中加载

如果配置项中没有明确设置 baseUrl 参数，默认值是加载 require.js 的 HTML 文件的目录。如果使用了 data-main 属性，那个路径将变成 baseUrl。

在加载 require.js 的页面上的 baseUrl 可以是一个不同域名的 URL。RequireJS 可以在跨域下加载脚本。唯一的限制是加载文本的 text！插件：至少在开发阶段，这些路径应该和页面有相同的域名。在使用优化工具之后，它们将在 text!插件里，你从其它的域就可以引用 text!插件资源来使用该资源了

paths：给在 baseUrl 中没有路径映射的模块。如果路径没有以’/’开头或者有 URL 协议（例如：http:），那么路径的设置是假定成相对 baseUrl 的。用上面的例子，’some/module’的<script>标签的 src 值为’another/path/some/v1.0/module.js’。

由于路径映射可能是一个目录，因此使用模块名作为路径就不应该包括扩展名。当将模块名映射成路径时，路径映射代码将自动加上.js 扩展名。如果使用 require.toUrl()，它将添加适当的扩展名，类似于一个文本模板。

当在浏览器中运行时，可以指定 paths fallbacks，允许从 CDN 节点上加载，但是如果从 CDN 加载失败可以回退到本地路径。

shim：可以为旧的，传统的浏览器没有使用 define()声明依赖和定义模块的，配置依赖关系，导出和自定义初始化。

这里有个示例，它需要 RequireJS 2.1.0 以上版本，同时在 baseUrl 路径下安装 backbone.js，underscore.js，jquery.js。如果没有，你可能需要为它们设置一个路径配置。

```
requirejs.config({
```

```
    //Remember: only use shim config for non-AMD scripts,
    //scripts that do not already call define(). The shim
    //config will not work correctly if used on AMD scripts,
    //in particular, the exports and init config will not
    //be triggered, and the deps config will be confusing
    //for those cases.
    shim: {
        'backbone': {
            //These script dependencies should be loaded before loading
            //backbone.js
            deps: ['underscore', 'jquery'],
            //Once loaded, use the global 'Backbone' as the
            //module value.
            exports: 'Backbone'
        },
        'underscore': {
            exports: '_'
        },
        'foo': {
            deps: ['bar'],
            exports: 'Foo',
            init: function (bar) {
                //Using a function allows you to call noConflict for
                //libraries that support it, and do other cleanup.
                //However, plugins for those libraries may still want
                //a global. "this" for the function will be the global
                //object. The dependencies will be passed in as
                //function arguments. If this function returns a value,
                //then that value is used as the module export value
                //instead of the object found via the 'exports' string.
                //Note: jQuery registers as an AMD module via define(),
                //so this will not work for jQuery. See notes section
                //below for an approach for jQuery.
                return this.Foo.noConflict();
            }
        }
    }
});

//Then, later in a separate file, call it 'MyModel.js', a module is
//defined, specifying 'backbone' as a dependency. RequireJS will use
//the shim config to properly load 'backbone' and give a local
//reference to this module. The global Backbone will still exist on
//the page too.
```

```
define(['backbone'], function (Backbone) {
   return Backbone.Model.extend({});
});
```

在 RequireJS 2.0.*，shim 配置项中的 exports 属性已经是一个函数，而不是字符串。在这种情形下，他和上述的 init 属性有相同的功能。在 RequireJS 2.1.0 以上版本中使用 init 模式，因此 enforceDefine 可以使用 exports 的字符串值，一旦加载了类库就允许执行

对于 JQuery 或 Backbone 插件的模块，没用必要导出任何模块值，shim 的配置仅仅是一个数组依赖：

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

如果想在 IE 中检测 404，你可以在 paths 中使用 fallbacks 和 errbacks，这时你需要指定一个 exports 值，来检测脚本是否已正确加载。

```
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'
        }
    }
});
```

重要的 shim 配置说明：

- Shim 配置只建立了代码关系。为了加载使用 shim 配置项的模块，还是需要调用 require 或者 define。单独的设置 shim 是不会触发加载代码的。
- 仅在 script 片段上使用 shim 模块作为依赖，像 AMD 类库这类的没有依赖的，当他们创建一个全局对象（像 jQuery 或 lodash）后调用 define()。否则的话，如果你在构建之后使用一个 AMD 模块作为 shim 配置模块的依赖。AMD 模块直到代码片段构建执行和出

错后才被执行。最终的解决方案是升级了所有的调用 AMD define()的代码片段。

- 对于 AMD 模块，初始化函数是不会被调用的。例如。你不能使用 shim 初始化配置来调用 jqueyr 的 noConflict。参考另外一种方法—“使用 noConflit 映射模块”。
- 通过 RequreJS 运行的 AMD 模块的节点是不支持 shim 配置的（尽管在优化工具下可以使用）。由于模块被 shim 之后，可能在 Node 中运行事变，因为 Node 中没有和浏览器相同的全局运行环境。到 RequireJS 2.1.7 为止，它将在控制台上告警你不支持 shim 配置，或许可以或许不可以正常运行。如果你洗完不显示这些信息，可以使用 requirejs.config({ suppress: { nodeShim: true }});控制。

重要的 Shim 配置优化说明

- 你应该使用 mainConfigFile 构建选项去指定寻找 shim 配置的文件。否则的话，优化工具不知道 shim 配置。另外一种方式是在构建文件中复制一份 shim 配置。
- 在构建是不要用 shim 配置混合 CDN 加载。示例场景：如果你通过 CDN 加载 jquery，但是通过 shim 配置加载项 Backbone 之类依赖 jquery 的版本。当你构建时，确保构建文建中使用内嵌的 jquery 而不是从 CDN 中加载。否则话 Backbone 内嵌在构建文件中，它将在 CDN 上 jquery 加载前执行。这是因为 shim 配置仅仅是延迟了文件加载直到它们的依赖被加载，但是不做任何的 define 自动包装。构建之后，依赖文件已被嵌入，shim 配置不在延迟执行费 define 部分的代码。通过 define 定义的模块能在通过 CDN 加载的代码下正常工作，就是因为它们包装了在定义函数中包装了代码直到依赖部分加载完成后才执行。所以结论就是：shim 配置模块对于非模块代码，遗留代码是表面上的解决方法。通过 define 定义的模块化是更好的选择。
- 对于本地的，多文件构建，上述的 CDN 建议也同样适用。对于任何脚本片段，必须在脚本片段执行前将依赖加载完成。这意味着在构建层直接构建脚本片段的依赖，或者通过调用 require([],function(){})加载它们的依赖，然后在已有脚本片段的构建层做一个嵌套的 requre([])调用。
- 如果你正在使用 uglifyjs 压缩代码，不要设置 uglify 的选项 toplevel 为 true，或者在命令行中不要使用 pass –mt。这个选项会损害使用 shim 导出的全局名称。

map：对于已有的模块前缀，而不是通过给出 ID 来加载模块，而是用新的模块 ID。
这类功能对于大项目而言非常实用，也许有两个不同的模块需要加载不同的 foo 版本，但是他们依然能够正常运行。

在基于上下文多版本加载中这是不可能支持的。另外，paths config 只能用在设置根路径的模块，不能用于将一个模块 ID 映射到另外一个。

Map 示例：

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

```
    }
});
```

磁盘上的模块目录如下

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

当‘some/newmodule’做 require(‘foo’)时，它将取到 foo1.2.js。当‘some/oldmodule’做 require(‘foo’)时，它将取到 foo1.1.js

此特性只有在脚本是真正的 AMD 模块通过调用 define()同时注册为匿名模块时才有效。不过在 map 配置中使用绝对路径。在相对路径（像‘../some/thing’）中不能使用。

同样支持通配符’*’，这意味着使用这个映射，加载所有模块。如果有一个明确的映射配置，那么将优先于’*’通配符。

例如：

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

出了‘some/module’的任何模块如果想加载 foo，将获得 foo1.2。对于‘some/module’模块，请求 foo 时将获取到 foo1.0

Config：传递一些配置信息给模块是一个常见的需求。那些配置信息通畅作为应用程序的一部分，需要有一中方式传递给模块。在 RequireJS 中，是通过在 require.config()中配置 config 选项。模块可以通过‘module’依赖或者调用 module.config()读取到指定的信息

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

```
            }
        }
});

//bar.js, which uses simplified CJS wrapping:
//http://requirejs.org/docs/whyamd.html#sugar
define(function (require, exports, module) {
        //这里将读取 bar 中的 size 信息
        var size = module.config().size;
});

//baz.js which uses a dependency array,
//it asks for the special module ID, 'module':
//https://github.com/jrburke/requirejs/wiki/Differences-between-the-simplified-CommonJS-wrapper-and-standard-AMD-define#wiki-magic
define(['module'], function (module) {
        //这里将读取到 baz.js 模块信息
        var color = module.config().color;
});
```

对于传递 config 对象到 package 中，是针对于 package 中的主模块，而不是 package ID

```
requirejs.config({
        //Pass an API key for use in the pixie package's
        //main module.
        config: {
                'pixie/index': {
                        apiKey: 'XJKDLNS'
                }
        },
        //Set up config for the "pixie" package, whose main
        //module is the index.js file in the pixie folder.
        packages: [
                {
                        name: 'pixie',
                        main: 'index'
                }
        ]
});
```

packages：配置从 CommonJS 包中加载的模块。更多信息可以参考包相关主题

waitSeconds：脚本加载超时时间。设置为 0 禁用超时。默认值为 7s

context：用来定义加载上下文的名称。这个机制允许 require.js 在一个页面中加载多个版本的模块，只要在最外层调用一个唯一的上下文名称。如何正确的使用它，请参考多版本支持章节。

deps：需要加载的依赖数组。在加载 require.js 之前，require 被定义成一个配置对象，同时你又想一旦 require()被定义时指定依赖的情况下，这时是非常有用的。使用 deps 就像调用 require([])一样，但是加载器一旦处理到配置项就立刻加载依赖。它不会阻塞来自模块的任何 require()调用，她仅仅是一种用来异步的加载模块作为配置项块方法

callback：当所有的模块加载完成后执行的函数。在加载 require.js 之前，require 被定义成一个配置对象，同时你想要在配置的依赖数组都被加载后指定一个函数去执行。

enforceDefine：如果设置成 true，如果加载的脚本没有调用 define()或者校验 shim 导出的字符串值时将抛出异常。更多信息参考 IE 中捕获加载失败。

xhtml：如果设置成 true，将使用 document.createElementNS()创建 script 元素

urlArgs：在 RequireJS 用来查询资源的 URL 后添加的额外参数值。当浏览器或服务器配置没有正确配置时，是缓存失效。

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

在开发阶段这个非常有用，确保在部署前移除这个。

scriptType：通过 RequireJS 指定<script>标签的 type 属性值。默认值是 text/javascript。如果想使用 FireFox 的 Javascript 1.8 特性，需改成’text/javascript;version=1.8’

高级用法

从包中加载模块

RequireJS 支持加载以 CommonJS 包目录结构的模块，但是需要指定额外的配置才能使用，具体地支持如下 CommonJS 包特性：

- 包名可以和模块名相关联
- 可以给包指定如下属性：
  - name：包名，用于映射模块名
  - location：文件路径。相对于 baseUrl 配置的路径，除非它们包含协议或者以’/’开头。

- main：当有人需要使用包名时，应该使用包内的模块名。默认值是‘main‘，如果和默认值不同需要重新指定。这个值相对于包文件夹

重要的注意事项：

- 当包具有 CommonJS 目录结构时，模块自身应该是 RequireJS 能够理解的模块格式。例外情形：如果你正在使用 r.js，模块应该是传统的 CommonJS 模块格式。如果你需要将传统的 CommonJS 模块转换成 RequireJS 方式的异步模块，你可以用 CommonJS 转换工具。
- 在一个工程上下文中同一时刻只能用一个版本的包。你可以通过 RequireJS 多版本支持加载两个不同的模块上下文，如果在同一个上下文中使用包 A 和包 B，而它们同时依赖于不同版本的包 C，这点还是一个问题，这点可能在未来的版本中修改。

如果你使用和快速指导类似的工程目录，你的 web 工程目录看起来就像这样（基于 Node、Rhino 的工程很类似，仅仅使用 scripts 目录作为顶层工程目录）

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

这里展示了带有两个包的目录结构，cart 和 store

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

project.html 页面内有像这样的<script>

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

这个要求 require.js 加在 script/main.js。在 main.js 中使用 packages 配置项建立相对于 require.js 的包，这个场景中是包 cart 和包 store。

```
//main.js contents
//Pass a config object to require
require.config({
    "packages": ["cart", "store"]
```

```
});

require(["cart", "store", "store/util"],
function (cart,     store,     util) {
    //use the modules as usual.
});
```

需要 cart 意味这从 scripts/cart/main.js 中加载，由于 main 是 RequireJS 支持默认的主模块设置。需要 store/util 将从 scripts/store/util.js 中加载。

如果 store 包没有采用 main.js 的约定，而是像下面的这样

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

RequireJS 的配置更像这样

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

为了避免冗长，强烈推荐在目录结构中使用 main 约定

多版本支持

配置项中提到的，在一个页面中通过不同的上下文配置选项加载多个版本的模块。Require.config()返回一个可以使用上下文配置的 require 函数。下面是加载 alapha 和 beta 模块两个不同版本（可以从最新的测试文件中取到）

```
<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：为模块指定一个 require 作为一个依赖。这将允许传递 require()函数到回调函数中，然后使用正确的上下文正确的加载模块为多版本提供支持。如果没有指定 require 作为一个依赖，那么可能出现错误。

页面加载完成后加载代码

在上述多版本支持章节中展示了代码如何通过内嵌的 require()调用实现延迟加载。

WebWorker 支持

这部分基本上使用不到，只做部分解释
tests/workers.html 有使用示例

当在 HTML 页面中执行脚本时，页面的状态是不可响应的，直到脚本已完成。
web worker 是运行在后台的 JavaScript，独立于其他脚本，不会影响页面的性能。您可以继续做任何愿意做的事情：点击、选取内容等等，而此时 web worker 在后台运行。

IE 10 开始支持，其他主流浏览器都支持
http://www.w3school.com.cn/html5/html_5_webworkers.asp

http://www.w3school.com.cn/tiy/t.asp?f=html5_webworker
https://www.ibm.com/developerworks/cn/web/1112_sunch_webworker/

Rhino 支持

Rhino 引擎，由网景公司的 Norris Boyd 开发，由 Java 实现
由 Mozilla 基金会管理，开放源代码，完全以 Java 编写。

错误处理

常见的错误类型：404（未找到），网络超时或其他脚本加载错误。RequireJS 有一系列用户处理这些的方法：指定 require 的 errbacks，paths 的配置项数组，全局的 requirejs.onError

传递一个 error 对象给 errbacks，全局的 requirejs.onError()函数通常包含两个自定义属性：

- **requireType**：一个常见的字符串类型值，像'timeout','nodefine','scripterror'
- **requireModules**：超时的模块或 URL 数组

如果你遇到了 requireModules 错误，很有可能是依赖模块的模块没有定义。

IE 中捕获加载失败

IE 有一些列的问题导致很难检测到 errbacks 或者 paths fallbacks 的加载失败。

- script.onerror 在 IE6-8 中不能正常运行。没有方法知道加载脚本中是否产生 404 错误，更糟糕的是，即使在 404 状态下也触发了 onreadystatechange 的完整状态。
- script.onerror 可运行在 IE9 中，但是有一个 bug，脚本执行后它不能够触发 script.onload 事件，因此它不支持使用匿名 AMD 模块的标准方法。Script.onreadystatechange 仍然在使用中。然而，onreadystatechange 在 script.onerror 函数触发之前以完整状态触发

因此，对于 IE 同时支持匿名的 AMD 模块（AMD 模块的核心用处），和可靠的检错机制是非常困难的。

然而可是，如果在你已知的项目中使用 define()来声明所有的模块，或者使用 shim 配置指定那些没有使用 define()的字符串导出，那么，如果你吧 enforceDefine 设置成 true，加载器能够确认通过检查 define()调用或者 shim 导出的全局值是否存在来判断脚本是否已加载。

因此，如果你想支持 IE，捕获加载失败，直接使用 define()或者 shim 配置的模块化代码，那么应该设置 enforceDefine 成 true。下一章节中有相关示例。

注意：如果你设置 enforceDefine 成 true，同时使用 data-main 加载 main JS 模块，那么 main JS 模块必须调用 define()而不是使用 require()加载它需要的代码，在 main JS 模块中仍然可以调用 require/requirejs 设置配置项，但是应该使用 define()加载模块。

如果你使用 almond 而不是用 require.js 来构建你的代码，确保在主模块中使用了 insertRequire 来构建调用 require()，这个和配置 data-main 来初始化 require()是一样的。

require([]) errbacks

当使用 requirejs.undef()，失败回调函数可以用来检测模块加载失败，模块未定义。可以重新尝试定位到另外一个地址。
常见的情形是使用 CDN 托管的版本库，如果加载失败了，将切换到本地文件加载。

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

```
    }
});

//Later
require(['jquery'], function ($) {
    //Do something with $ here
}, function (err) {
    //The errback, error callback
    //The error has a list of modules that failed
    var failedId = err.requireModules && err.requireModules[0];
    if (failedId === 'jquery') {
        //undef is function only on the global requirejs object.
        //Use it to clear internal knowledge of jQuery. Any modules
        //that were dependent on jQuery and in the middle of loading
        //will not be loaded yet, they will wait until a valid jQuery
        //does load.
        requirejs.undef(failedId);

        //Set the path to jQuery to local path
        requirejs.config({
            paths: {
                jquery: 'local/jquery'
            }
        });

        //Try again. Note that the above require callback
        //with the "Do something with $ here" comment will
        //be called if this new attempt to load jQuery succeeds.
        require(['jquery'], function () {});
    } else {
        //Some other error. Maybe show message to the user.
    }
});
```

使用 require.undef()，如果之后你配置了一个不同的配置项，尝试加载相同的模块，加载器仍然可以记住以前依赖的模块，当最新的配置模块加载时完成加载。

注意：errbacks 只适合在调用 require 回调方式，而非 define()。Define()只是用来声明模块的。

paths config fallbacks

上面检测加载失败，取消模块，修改路径，重新加载的模式是一种非常普遍的请求，同样也

有简写的方式。Paths config 允许一个数组值

```
requirejs.config({
    //To get timely, correct error triggers in IE, force a define/shim exports check.
    enforceDefine: true,
    paths: {
        jquery: [
            'http://ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min',
            //If the CDN location fails, load from this location
            'lib/jquery'
        ]
    }
});

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

上面的示例会先尝试 CDN 地址，如果失败了将加载本地的 lib/jquery.js 地址
注意：paths fallbacks 只有在准确的模块 ID 想匹配时才能正常运行。这个不同于可以应用与任何模块 ID 前缀的 paths config。Fallbacks 的设计目标是为了不常见的错误恢复，由于在浏览器中效率低下，因此不是一个通用的路径查找路径方案。

Global requirejs.onError function

为了检测那些没有被 errbacks 捕获的错误，你可以覆盖 requirejs.onError():

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

    throw err;
};
```

加载器插件

RequireJS 支持加载器插件。这是一种支持那些不是 JS 文件的依赖的方法，但是在它能够正常运行之前加载 script 是非常重要的。RequireJS Wiki 有一些列的插件。本章将讨论一些由 RequireJS 维护的插件。

指定一个文本文件依赖

使用常规的 HTML 标签来构建 HTML 是一种非常好的方式，而不是在 script 中构建 DOM。然而可是，在 JS 文件中没有好的方式嵌入 HTML。最好的方式是使用 HTML 字符串，但是这可能导致很难管理，尤其像多行 HTML

RequireJS 有个插件，text.js，对于这个问题有帮助。它可以自动的加载 text!前缀的依赖。详细请参考 text.js README。

一个简单的示例：

```
require({
        baseUrl: './',
        paths: {
            'text': 'frm/core/require/plugins/text'
        }
    }, ['text!test/text/a.txt', 'text!test/text/1.html'],
function(txt, html){
        console.log('txt file content is ' + txt);
        console.log('html file content is \r\n' + html);
    });
```

页面加载事件/DOM 就绪

由于使用 RequireJS 加载脚本是非常快的，所以在 DOM 就绪已经加载好了，这是有可能的。任何想和 DOM 进行交互的操作必须在 DOM 就绪后才能进行。对于现代的浏览器，直到 DOMContextLoaded 事件触发才完成。

然而，不是所有的浏览器支持 DOMContentLoaded 事件，domReady 模式实现了实现了跨浏览器的方法来判断 DOM 是否已经就绪。下载该模块，在项目中像这使用：

```
require(['domReady'], function (domReady) {
  domReady(function () {
    //This function is called once the DOM is ready.
    //It will be safe to query the DOM and manipulate
    //DOM nodes in this function.
  });
});
```

由于 DOM 就绪是一个常见的应用需求，最好避免在 API 上函数嵌套。domReady 模块同样也实现了加载器插件 API。因此你可以使用加载器插件语法（!在 domReady 中的依赖）强制使用是 require()的回调函数执行前必须等待 DOM 就绪。当 domReady 作为加载器插件时，将返回当前的文档 document 对象。

```
require(['domReady!'], function (doc) {
    //This function is called once the DOM is ready,
    //notice the value for 'domReady!' is the current
    //document.
});
```

注意：如果花费了很长时间去加载一个文档（很有可能是一个很大的文档，或者 HTML script 脚本加载大的 JS 文件阻塞了 DOM 就绪一直到加载完成）使用 domReady 作为加载器插件可能导致 RequireJS 超时错误。增加 waitSeconds 配置，或使用 domReady 作为一个模块，同时在 require()回调函数内调用 domReady()那么这就不是问题了。

定义一个 I18n 捆绑

nls， Native Language Support，国际化支持

一旦 Web 应用程序上了规模和知名度，在界面上提供国际化信息是十分有用的。但是，对于多语言支持，可能变得繁琐。

RequireJS 允许你设置一个基本的模块，这个模块有国际化信息，而不需要强制提供所有的资源信息。在区域文件中随着时间的推移只有添加字符串和值。

通过 i18n.js 提供 I18n 国际化信息支持。当一个模块或依赖指定的 i18n!前缀时，就会自动加载。下载插件，把它放到你应用程主入口 js 文件的相同目录。

定义一个捆绑，把它放在一个叫‘nls’的目录下，i18n 插件指定一个带 nls 模块将加载 i18n 捆绑，nls 标识告诉 i18n 插件去找本地目录（他们应该是 nls 的子目录）。如果你希望为‘my’模块提供颜色名称，像如下创建目录：

- *my/nls/colors.js*

这个文件的内容就像这样：

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

一个对象的 root 属性定义了这个模块。这是你为以后的本地化工作不得不做的所有设置。

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

my/lamps 模块有个叫 testMessage 的属性，使用 colors.red 来显示国际化中 color red 的值。

稍后，当你想添加一个翻译后的文件，比如 fr-fr 国际化，修改 my/nls/colors 内容如下：

```
//Contents of my/nls/colors.js
define({
    "root": {
        "red": "red",
        "blue": "blue",
        "green": "green"
    },
    "fr-fr": true
});
```

在 my/nls/fr-fr/colors.js 中定义如下内容：

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

RequireJS 将使用浏览器中的 navigator.language 或者 navigator.userLanguage 属性来确定 my/nls/colors 使用哪种资源文件。因此你的应用程序不需要做任何改变。如果你想设置语言区域，你可以使用 module config 传递区域语言给插件。

```
requirejs.config({
    config: {
        //Set the config for the i18n
        //module ID
        i18n: {
            locale: 'fr-fr'
        }
    }
```

```
});
```

注意：为了避免大小写问题，RequireJS 始终是使用小写区域语言，因此所有的国际化资源目录和文件都应该使用小写的区域语言。

RequireJS 在选择正确的区域语言文件同样也很智能，my/nls/colors 提供最近匹配原则。例如：如果区域语言是 en-us，那么将使用 root 捆绑。如果区域语言是 fr-fr-paris，那么将使用 fr-fr 捆绑。

RequireJS 同时会合并资源文件，例如，如果法文的资源文件如下（省略了 red）

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

那么将使用 root 中的 red 值，这个适用于所有的区域语言片段。如果定义如下的资源文件，那么 RequireJS 依照如下优先级顺序使用其中的值：

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

如果你喜欢在顶层模块不使用 root 绑定，那么你可以定义一个正常区域语言文件。这种情形，顶层模块可能像这样：

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

Root 捆绑将变成这样：

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