基本概念：用JavaScript编写的JS框架，主要功能是可以按不同的先后依赖关系对 JavaScript 等文件的进行加载工作，可简单理解为JS文件的加载器，它非常适合在浏览器中使用，它可以确保所依赖的JS文件加载完成之后再加载当前的JS文件，这在大量使用JS文件的项目中可确保各个JS文件的先后加载顺序，确保避免了以前因某些原因某个文件加载慢而导致其它加载快的文件需要依赖其某些功能而出现某函数或某变量找不到的问题，RequireJS是一个非常小巧的JavaScript模块载入框架（模块化加载器），是AMD规范最好的实现者之一。最新版本的RequireJS压缩后只有14K。它还同时可以和其他的框架协同工作

AMD模块的写法

a:无依赖项

define(function() {

return {

mix: function(source, target) { }

};

});

b:有依赖项

define(['data', 'ui'], function(data, ui) {

// init here

});

c:直接一个对象

define({

data: [],

ui: []

});

d:有名模块

define('index', ['data','base'], function(data, base) {

// todo

});

e:包装模块

define(function(require, exports, module) {

var base = require('base');

exports.show = function() {

// todo with module base

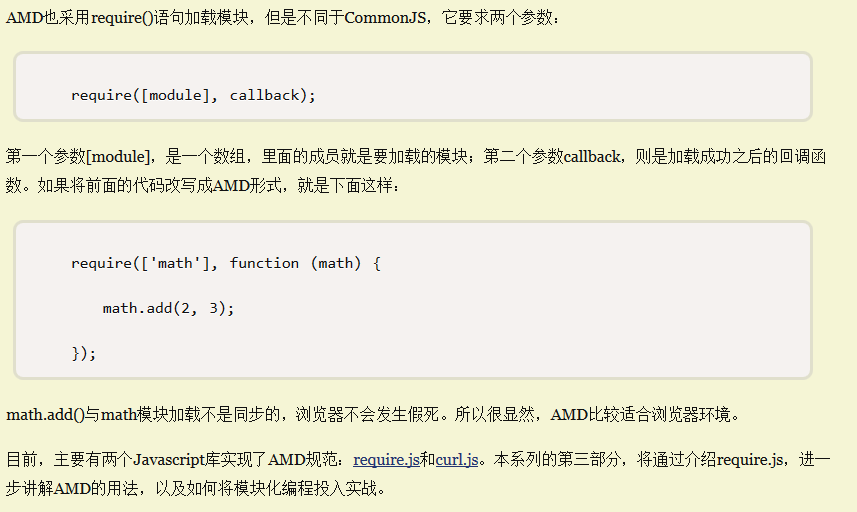
}

});

requireJS 官网 <http://requirejs.org> 中文翻译网 <http://www.requirejs.cn>

Javascript模块化编程：AMD规范

有了模块化编程我们就可以更方便地使用别人的代码，想要什么功能，就加载什么模块。但是，这样做有一个前提，那就是大家必须以同样的方式编写模块，否则你有你的写法，我有我的写法，岂不是乱了套！考虑到Javascript模块现在还没有官方规范，这一点就更重要了。目前，通行的Javascript模块规范共有两种：[CommonJS](http://wiki.commonjs.org/wiki/Modules/1.1)和[AMD](https://github.com/amdjs/amdjs-api/wiki/AMD)。我主要介绍AMD

[AMD](https://github.com/amdjs/amdjs-api/wiki/AMD)是"Asynchronous Module Definition"的缩写，意思就是"异步模块定义"。它采用异步方式加载模块，模块的加载不影响它后面语句的运行。所有依赖这个模块的语句，都定义在一个回调函数中，等到加载完成之后，这个回调函数才会运行

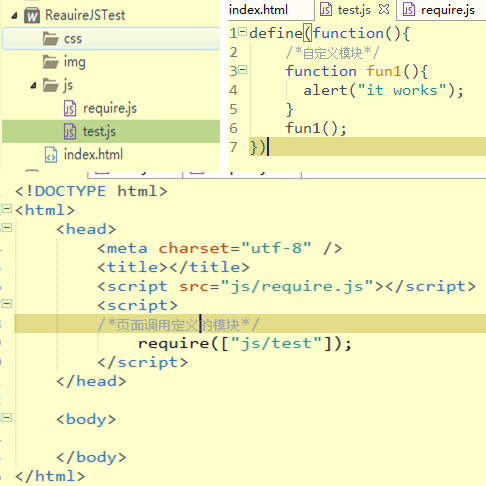
基本API

require会定义三个变量：define,require,requirejs，其中require === requirejs，一般使用require更简短

define：定义一个模块

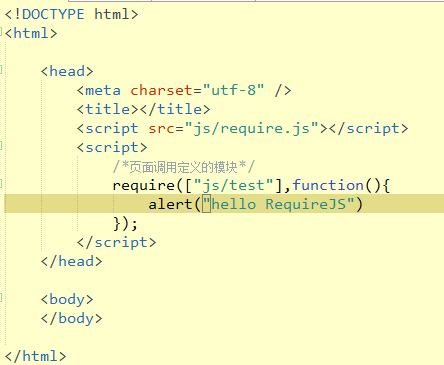
每个模块拥有一个唯一的模块ID，它被用于RequireJS的运行时函数，define函数是一个全局函数，不需要使用requirejs命名空间

require（requirejs）：加载依赖模块并在加载完之后执行回调函数



关于加载函数require的说明

Require（[],fn）用来加载模块读取依赖(注意require中的依赖是一个数组，即使只有一个依赖，你也必须使用数组来定义)，require API的第二个参数是callback，一个function，是用来处理加载完毕后的逻辑同样它是一个全局函数

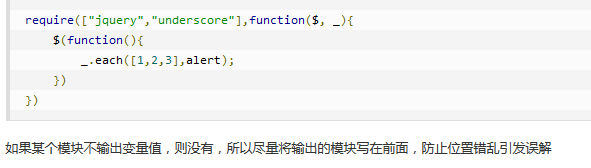


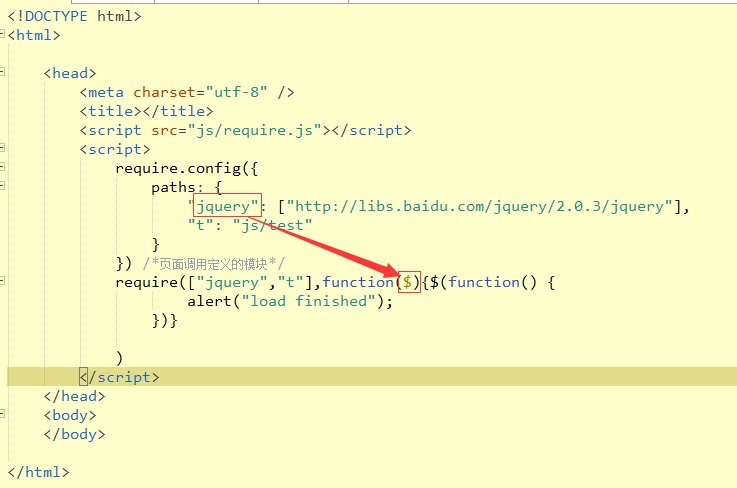
加载文件：之前的例子中加载模块都是本地js，但是大部分情况下网页需要加载的JS可能来自本地服务器、其他网站或CDN，这样就不能通过这种方式来加载了，我们以加载一个jquery库为例

require.config是用来配置模块加载位置，简单点说就是给模块起一个别名，比如将百度的jquery库地址标记为jquery，这样在require时只需要写["jquery"]就可以加载该js，本地的js我们也可以这样配置

paths的配置会使我们的模块名字更精炼，paths还有一个重要的功能，就是可以配置多个路径，如果远程库没有加载成功，可以加载本地的库

这样配置后，当百度的jquery没有加载成功后，会加载本地js目录下的jquery

上面例子中的callback函数中发现有$参数，这个就是依赖的jquery模块的输出变量，如果你依赖多个模块，可以依次写入多个参数来使用



全局配置

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

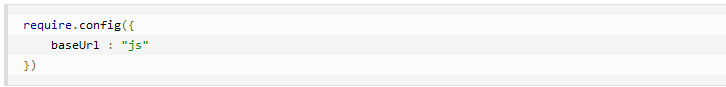


///////////////////////////////////////无法验证main.js

解释一下，加载requirejs脚本的script标签加入了data-main属性，这个属性指定的js将在加载完reuqire.js后处理，我们把require.config的配置加入到data-main后，就可以使每一个页面都使用这个配置，然后页面中就可以直接使用require来加载所有的短模块名

data-main还有一个重要的功能，当script标签指定data-main属性时，require会默认的将data-main指定的js为根路径，是什么意思呢？如上面的data-main="js/main"设定后，我们在使用require(['jquery'])后(不配置jquery的paths)，require会自动加载js/jquery.js这个文件，而不是jquery.js，相当于默认配置了：

///////////////////////////////////////无法验证main.js

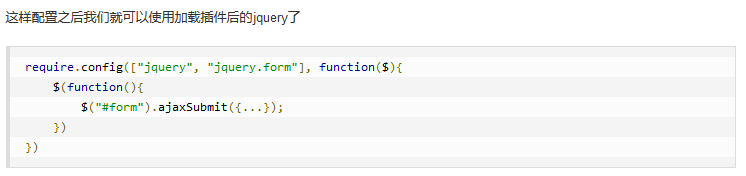
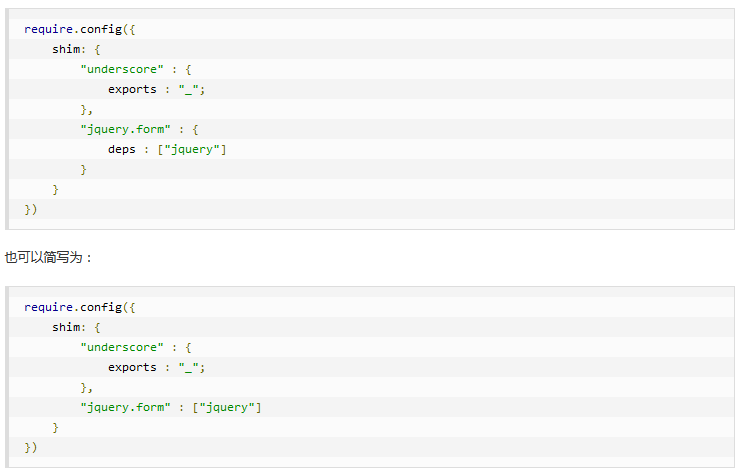


第三方模块

通过require加载的模块一般都需要符合AMD规范即使用define来申明模块，但是部分时候需要加载非AMD规范的js，这时候就需要用到另一个功能：shim，目前我主要用在两个地方

1. 非AMD模块输出，将非标准的AMD模块"垫"成可用的模块，例如：在老版本的jquery中，是没有继承AMD规范的，所以不能直接require["jquery"],这时候就需要shim，比如我要是用underscore类库，但是他并没有实现AMD规范，那我们可以这样配置



2插件形式的非AMD模块，我们经常会用到jquery插件，而且这些插件基本都不符合AMD规范，比如jquery.form插件，这时候就需要将form插件"垫"到jquery中：

shim：{

a：{

b：{

deps:[],

exports:‘name’}

}

}

exports就是模块的返回值

deps：用来定义对其他模块的依赖关系

requireJS的加载

<script src="js/require.js"></script>加载这个文件，也可能造成网页失去响应。解决办法有两个，一个是把它放在网页底部加载，另一个是写成下面这样

<script src="js/require.js" **defer async="true"** ></script>

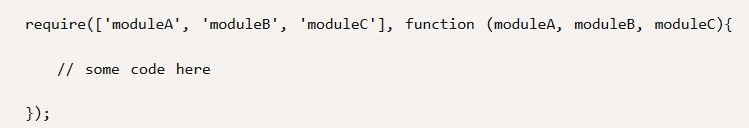
async属性表明这个文件需要异步加载，避免网页失去响应。IE不支持这个属性，只支持defer，所以把defer也写上

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

主模块的写法

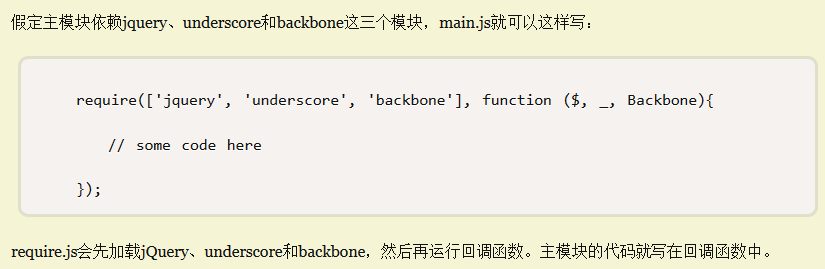
上一节的main.js，我把它称为"主模块"，意思是整个网页的入口代码。它有点像C语言的main()函数，所有代码都从这儿开始运行

常见的情况是，主模块依赖于其他模块，这时就要使用AMD规范定义的的require()函数



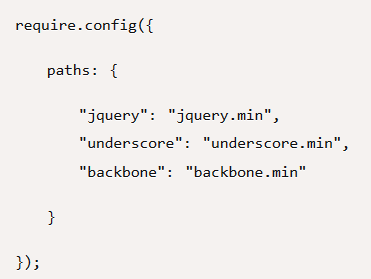
require()函数接受两个参数。第一个参数是一个数组，表示所依赖的模块，上例就是['moduleA', 'moduleB', 'moduleC']，即主模块依赖这三个模块；第二个参数是一个回调函数，当前面指定的模块都加载成功后，它将被调用。加载的模块会以参数形式传入该函数，从而在回调函数内部就可以使用这些模块。

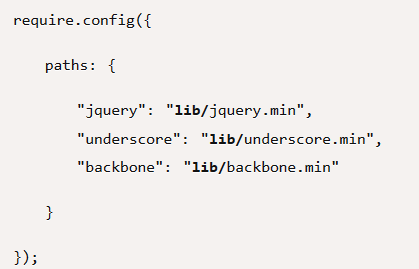
require()异步加载moduleA，moduleB和moduleC，浏览器不会失去响应；它指定的回调函数，只有前面的模块都加载成功后，才会运行

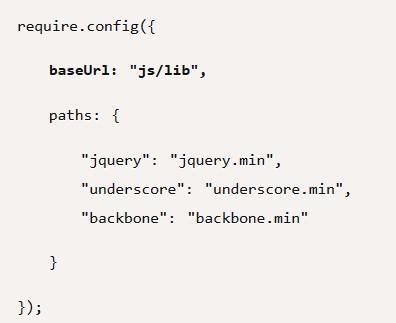


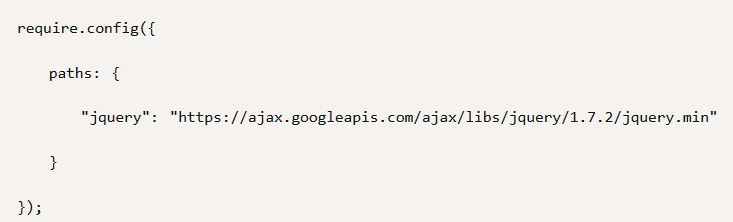
模块的加载

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



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

另一种则是直接改变基目录（baseUrl）

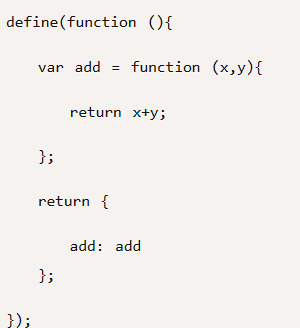
如果某个模块在另一台主机上，也可以直接指定它的网址

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

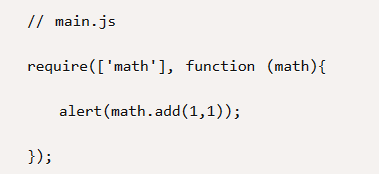
**AMD模块的写法**

模块必须按照AMD的规定来写具体来说就是模块必须采用特定的define()函数来定义

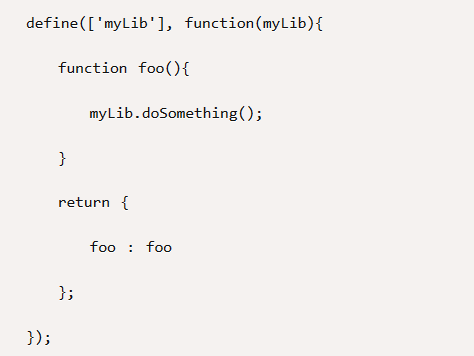
如果一个模块不依赖其他模块，那么可以直接定义在define()函数当中



加载方法如下



如果这个模块还依赖其他模块，那么define()函数的第一个参数，必须是一个数组，指明该模块的依赖性，如下面的例子当require()函数加载上面这个模块的时候，就会先加载myLib.js文件



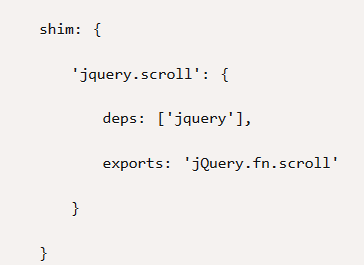
加载非规范模块

理论上，require.js加载的模块，必须是按照AMD规范、用define()函数定义的模块。但是实际上，虽然已经有一部分流行的函数库（比如jQuery）符合AMD规范，更多的库并不符合举例来说，underscore和backbone这两个库，都没有采用AMD规范编写。如果要加载它们的话，必须先定义它们的特征



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

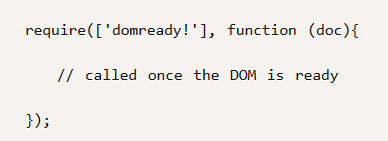
deps数组，表明该模块的依赖性。比如，jQuery的插件可以这样定义：

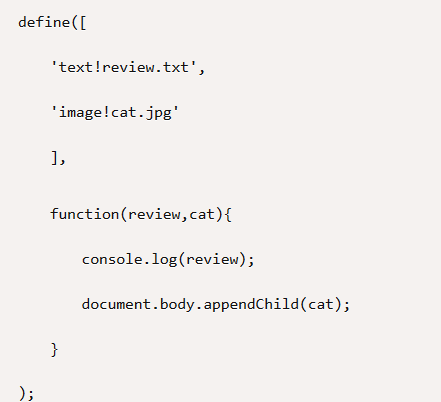


**require.js插件**

require.js还提供一系列[插件](https://github.com/jrburke/requirejs/wiki/Plugins)，实现一些特定的功能。

domready插件，可以让回调函数在页面DOM结构加载完成后再运行



text和image插件，则是允许require.js加载文本和图片文件

官方文档

RequireJS加载所有相对于baseUrl的代码。 baseUrl通常设置为与用于加载页面的顶级脚本的data-main属性中使用的脚本相同的目录。 data-main属性是一个特殊的属性，require.js将检查以启动脚本加载。 这个例子将以一个baseUrl脚本结束

这将baseUrl设置为“scripts”目录，并且 加载一个模块ID为'main'的脚本

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

或者，可以通过RequireJS配置手动设置baseUrl。 如果没有明确的配置并且没有使用data-main，那么默认的baseUrl是包含运行RequireJS的HTML页面的目录。RequireJS在默认情况下也假定所有依赖项都是脚本，因此它不会在模块ID上看到尾随的“.js”后缀。 在将模块ID转换为路径时，RequireJS会自动添加它。

有时您可能会直接引用脚本，并且不符合“baseUrl +路径”规则以查找它。 如果模块ID具有以下特征之一，则该ID不会通过“baseUrl +路径”配置传递，而只会被视为与文档相关的常规URL

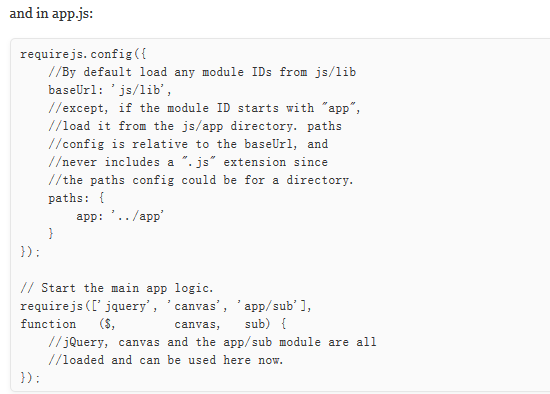
**Ends in ".js"**

**Starts with a "/"**

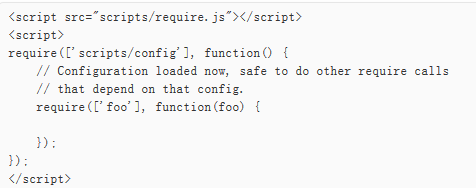
**包含URL协议，如“http：”或“https：**

一般来说，最好使用baseUrl和“paths”配置为模块ID设置路径。 通过这样做，可以更灵活地重命名并配置通往不同位置的路径以实现优化构建。同样，为了避免大量配置，最好避免使用脚本的深层文件夹层次结构，而是将所有脚本保存在baseUrl中，或者如果要将库/供应商提供的代码与应用程序代码分开，请使用 目录布局如下所示：





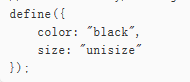
data-main：数据的主要入口点，data-main属性是一个特殊的属性，require.js将以启动脚本加载

如果你想在HTML页面中进行require（）调用，那么最好不要使用data-main。 data-main仅用于页面只有一个主入口点，即数据主脚本。 对于想要执行内联require（）调用的页面，最好将这些嵌套在require（）调用中以进行配置

定义一个模块

模块与传统的脚本文件不同之处在于它定义了一个良好范围的对象，避免污染全局名称空间。 它可以显式地列出它的依赖关系，并获得这些依赖关系的句柄，而不需要引用全局对象，而是接收依赖关系作为定义模块的函数的参数。 RequireJS中的模块是模块模式的扩展，其优点是不需要全局变量来引用其他模块

简单的K:V模式定义



如果模块没有依赖性，但需要使用某个函数来完成一些设置工作，那么定义它自己，传递一个函数define（）：

define(function () {

//这里是设置工作

return {

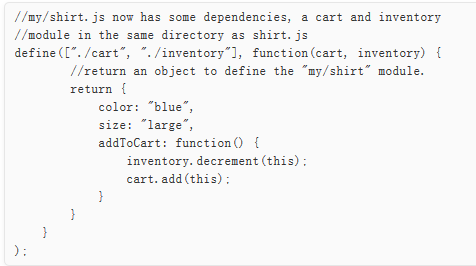
color: "black",

size: "unisize"

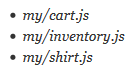
}

});

定义具有依赖关系函数

如果模块具有依赖关系，则第一个参数是一个依赖项名称数组，而第二个参数定义函数。 一旦加载了所有依赖关系，该函数将被调用来定义模块。 该函数应该返回一个定义该模块的对象。 依赖性将作为函数参数传递给定义函数，按照与依赖性数组中的顺序相同的顺序列出

在这个例子中，创建了一个my / shirt模块。 这依赖于my/cart my/inventory。 在磁盘上，文件的结构如下所示：



上面的函数调用指定了两个参数“cart”和“inventory”。 这些是由“./cart”和“./inventory”模块名称表示的模块。直到my / cart和my / inventory模块被加载后才能调用该函数，并且该函数接收模块作为“cart”和“inventory”参数。定义全局变量的模块显然不鼓励，因此一次可以在一个页面中存在多个模块版本（请参阅高级用法）。 另外，函数参数的顺序应该与依赖关系的顺序相匹配。函数调用的返回对象定义了“my / shirt”模块。 通过以这种方式定义模块，“my / shirt”不作为全局对象存在。

将模块定义为函数

模块不必返回对象。 任何有效的函数返回值都是允许的。

define(["my/cart", "my/inventory"],

function(cart, inventory) {

//它获取或设置窗口标题。

return function(title) {

return title ? (window.title = title) :

inventory.storeName + ' ' + cart.name;

}

}

);

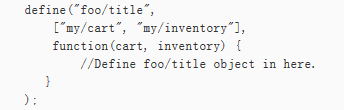
在foo / title.js中的模块定义。 它依赖"my/cart", "my/inventory模块 ，但由于foo / title.js与其目录不同它使用模块依赖关系中的“"my/cart", "my/inventory模块。 名字的“my”部分可以被映射到任何目录，但默认情况下，它被假定为同级到“foo”目录。

用名称定义一个模块

您可能会遇到一些define（）调用，其中包含模块的名称作为define（）的第一个参数：

定义一个名称为foo/title的模块（不提倡）

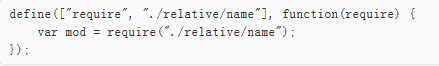
这些通常由优化工具生成。 你可以明确地命名模块，但它会使模块的可移植性降低 - 如果将文件移动到另一个目录，则需要更改名称。 通常最好避免使用模块名称进行编码，只需让优化工具在模块名称中刻录即可。 优化工具需要添加名称，以便可以在一个文件中捆绑多个模块，以便在浏览器中加载速度更快。



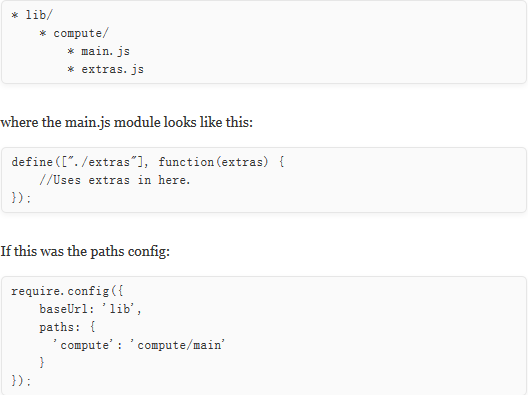
几点注意

每个文件定义一个模块，每个JavaScript文件只应定义一个模块。您只应使用优化工具将多个模块分组为优化文件。

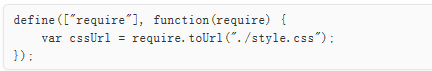
define（）中的相对路径模块名称：对于在define（）函数调用中可能发生的require（“./ relative / name”）调用，请务必要求“require”作为依赖项，以便相对名称被解析正确：



使用相对模块定义



生成相对于模块的URL：您可能需要生成一个与模块相关的URL。 为此，请求“require”作为依赖项，然后使用require.toUrl（）生成URL：



指定JSONP服务依赖项

JSONP是一种在JavaScript中调用某些服务的方式。 它可以跨域使用，并且它是一种通过脚本标记调用仅需要HTTP GET的服务的既定方法。

要在RequireJS中使用JSONP服务，请将“define”指定为回调参数的值。 这意味着您可以像获取模块定义一样获取JSONP URL的值。

这是一个调用JSONP API端点的例子。 在这个例子中，JSONP回调参数被称为“callback”，所以“callback = define”告诉API将JSON响应包装在“define（）”包装中

require(["http://example.com/api/data.json?callback=define"],

function (data) {

//数据对象将是JSONP数据调用的API响应。

console.log(data);

}

);

对于初始应用程序设置，JSONP的这种使用应限于JSONP服务。 如果JSONP服务超时，则意味着您通过define（）定义的其他模块可能无法执行，因此错误处理不健全。仅支持JSON对象的JSONP返回值。 一个数组，字符串或数字的JSONP响应将不起作用。此功能不应用于长轮询JSONP连接 - 处理实时流的API。 这些类型的API在接收到每个响应后应该做更多的脚本清理工作，并且RequireJS只会获取一次JSONP URL--随后在require（）或define（）调用中使用同一个URL作为依赖关系会得到一个缓存值。加载JSONP服务的错误通常通过服务的超时来浮出水面，因为脚本标记加载并没有给出网络问题的详细信息。 要检测错误，可以覆盖requirejs.onError（）以获取错误。 处理错误部分有更多信息。

定义一个模块

RequireJS使用head.appendChild（）将每个依赖项加载为脚本标记。

配置对象的加载

在顶级HTML页面（或未定义模块的顶级脚本文件）中使用require（）时，可以将配置对象作为第一个选项定义

<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"],

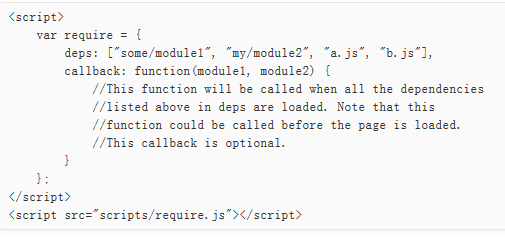
function(someModule, myModule) {

//code

}

);

</script>

可以从数据主入口点调用require.config，但要注意数据主脚本是异步加载的。 避免其他入口点脚本错误地假定data-main及其require.config始终在其脚本加载之前执行。此外，您可以在require.js加载之前将配置对象定义为全局变量require，并自动应用这些值。 这个例子指定了require.js定义require（）时要加载的一些依赖关系：

注意：最好使用var require = {}并且不要使用window.require = {}，它在IE支持

支持配置选项

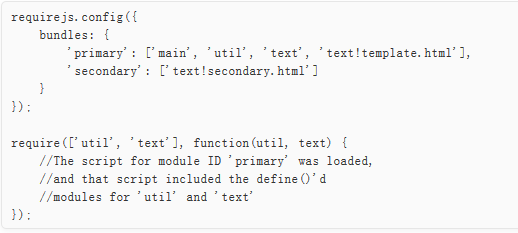
baseUrl：用于所有模块查找的根路径。在上面的例子中，“my / module”的脚本标签将有一个src =“/ another / path / my / module.js”。

在加载plain .js文件（由以斜杠开始，具有协议或以.js结尾的依赖项字符串表示）时，不会使用baseUrl，因此这些字符串将按原样使用，因此a.js和b.js将是 从与包含上述片段的HTML页面相同的目录加载。

如果在配置中没有显式设置baseUrl，则默认值将是加载require.js的HTML页面的位置。 如果使用data-main属性，则该路径将成为baseUrl。

paths：使用与模块名称的路径映射不能直接在baseUrl下找到。用于模块名称的路径不应包含扩展名，因为路径映射可能适用于目录。 将模块名称映射到路径时，路径映射代码将自动添加.js扩展名。 如果使用require.toUrl（），它将添加适当的扩展名，

[**bundles**](http://requirejs.org/docs/api.html#config-bundles): 捆绑：在RequireJS 2.1.10中引入：允许在另一个脚本中配置多个模块ID。 例：



当模块ID为'primary'的脚本被加载，该脚本包括 'util'和'text'模块

该配置指出：通过加载模块ID'primary'可以找到模块的'main'，'util'，'text'和'

模块'text！secondary.html'可以通过加载模块ID'secondary'找到。

这只设置在脚本中找到一个模块的位置，该脚本中有多个define（）'模块。它不会自动将这些模块绑定到软件包的模块ID。该包的模块ID仅用于定位该组模块。

如果执行构建并且构建目标不是现有的模块ID，或者在构建的JS文件中有不应由加载器插件加载的加载器插件资源，则bundles config非常有用。请注意，键和值是模块ID，而不是路径段。它们是绝对的模块ID，而不是像路径配置或映射配置的模块ID前缀。此外，bundle config与map config的不同之处在于，map config是一对一的模块ID关系，其中bundle config用于将多个模块ID指向一个bundle的模块ID。

从RequireJS 2.2.0开始，优化器可以生成bundle config并将其插入到顶层requirejs.config（）调用中。有关更多详细信息，请参阅bundlesConfigOutFile构建配置选项。

Map：对于给定的模块前缀，而不是使用给定的ID加载模块，而是替换不同的模块ID。

这种能力对于可能有两组模块需要使用两个不同版本的'foo'的大型项目来说非常重要，但他们仍然需要相互合作。

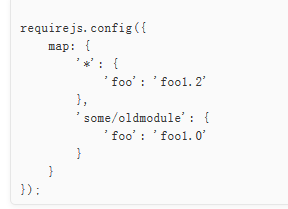
这种背景支持多版本支持是不可能的。 此外，路径配置仅用于为模块ID设置根路径，而不是将一个模块ID映射到另一个模块ID。



当'some / newmodule'确实需要（'foo'）`它会得到foo1.2.js文件，而'some / oldmodule'做'require（'foo'）`它会得到foo1.0。 js文件。

此功能仅适用于调用define（）并注册为匿名模块的真正AMD模块的脚本。 另外，只能使用map config的绝对模块ID。 相对ID（如'../some/thing'）不起作用。

还支持“\*”这意味着“对于所有加载的模块，使用此配置”。 如果有更具体的配置，那将优先于星形配置。 例：



注意：在使用map config进行构建时，地图配置需要被馈送到优化器，构建输出必须包含一个设置地图配置的requirejs配置调用。 优化器不会在构建过程中进行ID重命名，因为项目中的某些依赖项引用可能取决于运行时变量状态。 所以优化器在构建之后使地图配置无效。

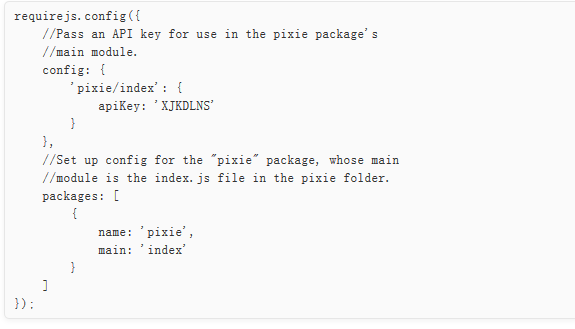
[**config**](http://requirejs.org/docs/api.html#config-moduleconfig)**的相关参数**

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

deps——加载依赖关系数组

**设置“pixie”包的配置，其主要模块是pixie文件夹中的index.js文件。**



[**packages**](http://requirejs.org/docs/api.html#config-packages)**：从CommonJS包配置加载模块。 有关更多信息，请参阅软件包主题**

[**waitSeconds**](http://requirejs.org/docs/api.html#config-waitSeconds)**：在放弃加载脚本之前等待的秒数。 将其设置为0将禁用超时。 默认值是7秒。**

[**context**](http://requirejs.org/docs/api.html#config-context)

[**deps**](http://requirejs.org/docs/api.html#config-deps)**：要加载的依赖关系数组**

callback：在deps加载后执行的函数。 当require.js被加载之前require被定义为一个配置对象时，并且你想要在配置的deps数组被加载后指定一个要求的函数。

enforceDefine：捕获加载失败

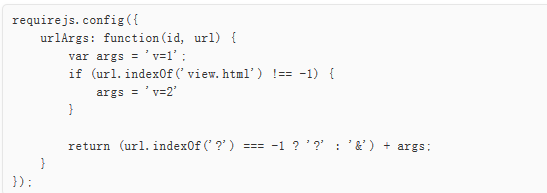
xhtml：如果设置为true，则将使用document.createElementNS（）来创建脚本元素。

urlArgs：追加到RequireJS用于提取资源的URL的额外查询字符串参数。 当浏览器或服务器配置不正确时，最有用的功能是缓存。 urlArgs的缓存大小设置示例：



从RequireJS 2.2.0开始，urlArgs可以是一个函数。 如果是函数，它将接收模块ID和URL作为参数，并且它应该返回一个字符串，该字符串将被添加到URL的末尾。 如果没有参数，则返回一个空字符串。 一定要注意添加'？' 或'＆'取决于URL的现有状态。

在开发过程中使用它可能很有用，但是在部署代码之前一定要将其删除。

例：

[**scriptType**](http://requirejs.org/docs/api.html#config-scriptType): 指定用于由RequireJS插入到文档中的脚本标记的type =“”属性的值。 默认是“text / javascript”。 要使用Firefox的JavaScript 1.8功能，请使用“text / javascript; version = 1.8”。

[skipDataMain](http://requirejs.org/docs/api.html#config-skipDataMain): 在RequireJS 2.1.9中引入：如果设置为true，则跳过完成的data-main属性扫描以启动模块加载。 如果RequireJS嵌入在可能与页面上的其他RequireJS库交互的实用程序库中，并且嵌入式版本不应该执行数据主加载，则这很有用。

从包中加载模块

包的要求

包可以与模块名称/前缀关联

软件包配置可以为特定软件包指定以下属性：

Name：包的名称（用于模块名称/前缀映射）

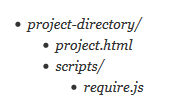
Location：磁盘上的位置。 位置与baseUrl配置值有关，除非它们包含协议或以前斜杠（/）开头

Main: 当某人需要“packageName”时应该使用的包内模块的名称。 默认值是“main”，所以只有当它与默认值不同时才指定它。 该值是相对于包文件夹

说明

虽然软件包可以具有CommonJS目录布局，但模块本身应该是RequireJS能够理解的模块格式。 规则的例外：如果您使用的是r.js节点适配器，则模块可以采用传统的CommonJS模块格式。 如果您需要将传统CommonJS模块转换为RequireJS使用的异步模块格式，则可以使用CommonJS转换器工具。

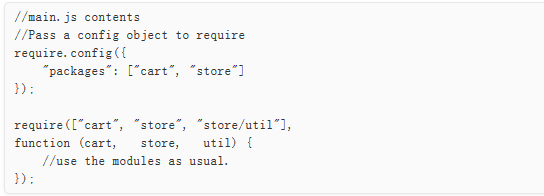
一次只能在项目上下文中使用一个软件包版本。 你可以使用RequireJS多版本支持来加载两个不同的模块上下文，但是如果你想在一个上下文中使用包A和B，并且它们依赖于包C的不同版本，那么这将是一个问题。 这可能在未来发生变化。如果您使用“开始指南”中指定的类似项目布局，则Web项目的开始部分将如下所示（基于Node / Rhino的项目相似，只需使用脚本目录的内容作为顶级项目目录）：



下面是示例目录布局与两个包，购物车和商店的关系：

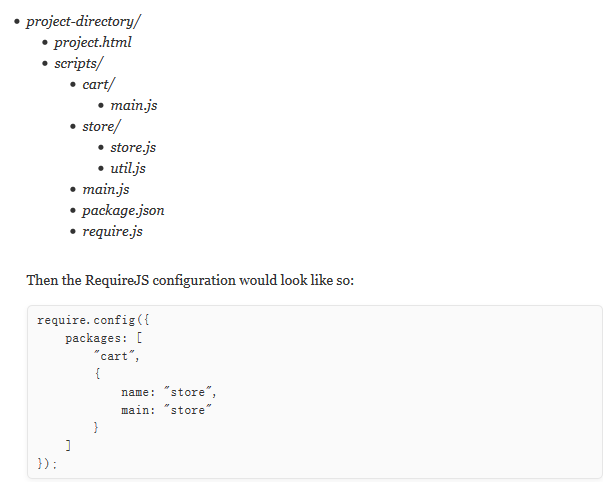


这将指示require.js加载scripts / main.js。 main.js使用“packages”配置来设置与require.js相关的包，在这种情况下，它是源包“cart”和“store”：



由于“main”是RequireJS支持的默认主模块设置，因此“cart”的要求意味着它将从scripts / cart / main.js加载。 “store / util”的需求将从scripts / store / util.js加载。

如果“store”包不遵循“main.js”约定，并且看起来更像这样：



为避免冗长，强烈建议始终使用在其结构中使用“主要”约定的包。

RequireJS的优化

RequireJS有一个优化工具，可以执行以下操作

通过UglifyJS（默认）或Closure Compiler（使用Java时的选项）将相关脚本组合到构建层中并缩小它们。

通过内嵌@import引用的CSS文件并删除注释来优化CSS。

优化器是Node和Nashorn的r.js适配器的一部分，它被设计成在完成开发之后作为构建或打包步骤的一部分运行，并准备为您的用户部署代码

优化器只会将在传递给顶层require和定义调用的字符串文字数组中指定的模块或简化的CommonJS包装中的require（'name'）字符串文字调用组合在一起。 所以，它不会找到通过变量加载的模块

优化器可以使用Node，Java与Rhino或Nashorn或浏览器运行。 每个选项的要求：

**Node:** (preferred) [Node](http://nodejs.org) 0.4.0 or later.（[Node](http://nodejs.org)0.4.0或更高版本）

**Java:** [Java 1.6](http://java.com/) or later.

对于命令行使用，Node是首选的执行环境。 优化器使用Node运行得更快

环境准备

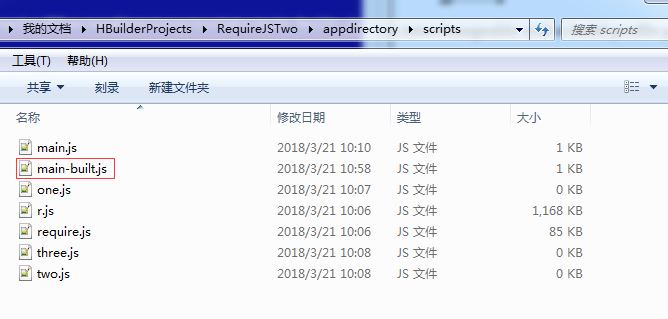
Node.js软件

r.js从下载页面手动下载。 它通常是使用优化器的最清晰，最便携的方式

build.js（优化器配置文件）的创建方式：通过命令行（不推介）

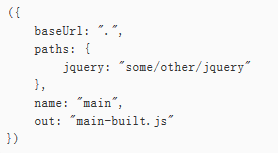
在项目目录下命令行执行（运行优化器（optimizer）命令）node r.js -o baseUrl=. paths.jquery=some/other/jquery name=main out=main-built.js

结果如下



方式二：手动构建配置文件**build.js**

**文件内容如下**



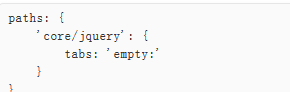
然后执行：node r.js -o build.js达到和方式一同样的效果

不推介通过命令行创建build.js的原因

命令行参数语法有一个限制。 点被视为对象属性分隔符

如paths.jquery = lib / jquery 内容转换为以下内容：



但对于paths.core/jquery.tabs=empty这样的命令会出现如下的错误结果

所以手动构建build.js非常必要

build.js中的参数设置参考如下文档<https://github.com/requirejs/r.js/blob/master/build/example.build.js>

相对路径解析规则：

一般来说，使用命令行参数如果是一个路径，它相对于用于存放构建选项的build.js文件或者相对于当前工作目录而言。

属于文件路径的属性示例：appDir，dir，mainConfigFile，out，wrap.startFile wrap.endFile。

对于baseUrl，它与appDir相关。 如果没有appDir，那么baseUrl是相对于build.js文件

对于路径和包，它们与baseUrl相关

对于属于模块ID的属性，它们应该是模块ID，而不是文件路径例如 name, include, exclude, excludeShallow, deps

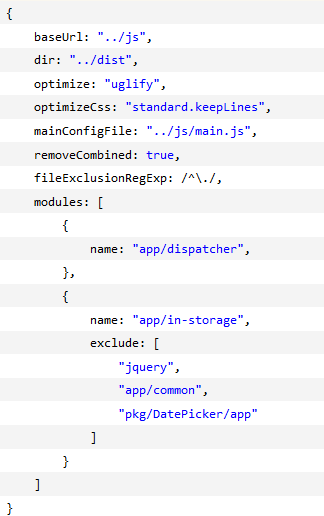
默认情况下，优化器不会读取运行时加载到浏览器中的主JS模块中的配置设置

这是因为构建的配置设置可能会非常不同，并具有多个优化目标。 因此需要为优化器指定一组单独的配置选项。

在优化器的1.0.5+版本中，mainConfigFile选项可用于指定运行时配置的位置。 如果指定了主JS文件的路径，则会解析该文件中的第一个requirejs（{}），requirejs.config（{}），require（{}）或require.config（{}） 用作传递给优化器的配置选项的一部分：

config：命令行，构建配置文件，高于mainConfigFile的优先级。 换句话说，mainConfigFile配置具有最低的优先级。

简单的build.js文件



参数说明

appDir

应用程序的最顶层目录。可选的，如果设置了的话，r.js 会认为脚本在这个路径的子目录中，应用程序的文件都会被拷贝到输出目录（dir 定义的路径）。如果不设置，则使用下面的 baseUrl 路径。

baseUrl

默认情况下，所有的模块都是相对于这个路径的。如果没有设置，则模块的加载是相对于 build 文件所在的目录。另外，如果设置了appDir，那么 baseUrl 应该定义为相对于 appDir 的路径。

dir

输出目录的路径。如果不设置，则默认为和 build 文件同级的 build 目录。

Optimize：JavaScript 代码优化方式。可设置的值：

"uglify：使用 UglifyJS 压缩代码，默认值；

"uglify2"：使用 2.1.2+ 版本进行压缩；

"closure"： 使用 Google's Closure Compiler 进行压缩合并，需要 Java 环境；

"closure.keepLines"：使用 Closure Compiler 进行压缩合并并保留换行；

"none"：不做压缩合并；

optimizeCss：CSS 代码优化方式，可选的值有：

"standard"：标准的压缩方式；

"standard.keepLines"：保留换行；

"standard.keepComments"：保留注释；

"standard.keepComments.keepLines"：保留换行；

"none"：不压缩；

mainConfigFile

如果不想重复定义的话，可以使用这个参数配置 RequireJS 的配置文件路径。

removeCombined：删除之前压缩合并的文件，默认值 false。

fileExclusionRegExp：要排除的文件的正则匹配的表达式。

modules：定义要被优化的模块数组。每一项是模块优化的配置，常用的几个参数如下：

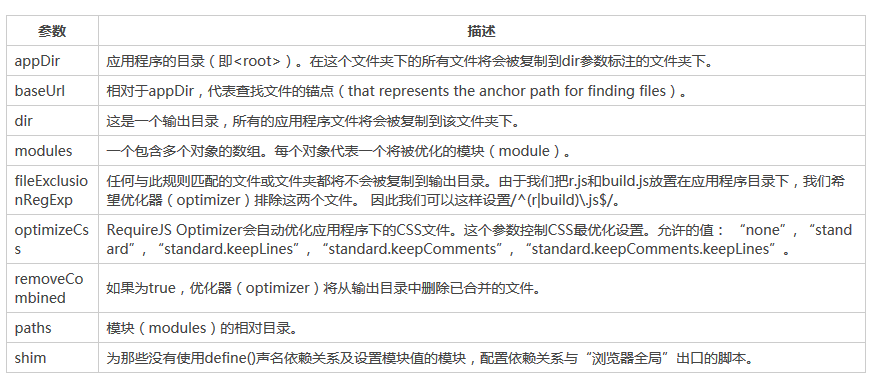
　　　　name：模块名；

　　　　create：如果不存在，是否创建。默认 false；

　　　　include：额外引入的模块，和 name 定义的模块一起压缩合并；

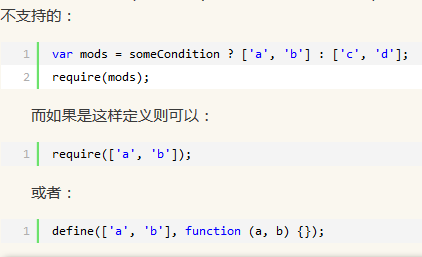
　　　　exclude：要排除的模块。有些模块有公共的依赖模块，在合并的时候每个都会压缩进去，例如一些基础库。使用 exclude 就可以把这些模块在压缩在一个更早之前加载的模块中，其它模块不用重复引入。

RequireJS 配置也可以放到 RequireJS Optimizer 配置文件里面



相关配置参数参考

<https://github.com/jrburke/r.js/blob/master/build/example.build.js>

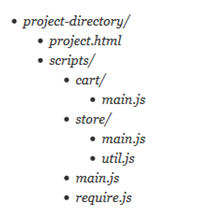
注意RequireJS Optimizer 只支持使用 require 和 define 语法定义的模块，因此下面这种通过变量定义的方式是不支持的：

项目被优化后的缺点

显然，在优化之后，我们再也没有必要引用require.js文件了。因为已经没有被分离的脚本文件了并且所有具有依赖关系的文件也已被加载。 尽管如此，优化过程将我们所有的脚本合并生成了一个优化后的脚本文件，其中包含了很多次define() 和require()调用。 因此，为了保证应用程序能够正常运行，define()和require()必须指定并实施到应用程序的某处（即包含这些文件）。 这会导致一个众所周知的开销：我们总是会有一些代码实现define()和require()。**这些代码并不是应用程序的一部分，它们的存在仅仅是为我们的基础建设考虑（infrastructure considerations）。** 当我们开发一个JavaScript库（JavaScript library）时，这个问题变得尤为巨大。相比RequireJS，这些库通常都很小，因此在库中包含它会造成一笔巨大的开销

解决方案：使用[almond](https://github.com/jrburke/almond)来缓解这个问题。Almond是一个极简单的AMD加载器，它实现了RequireJS接口（API）。因此，可以用来在已优化过的代码中替代RequireJS实现，我们可以在项目中包含almond。

优化一个javasc文件



如果只想优化main.js，则可以从appdirectory / scripts目录内使用此命令：

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

这将创建一个名为appdirectory / scripts / main-built.js的文件，其中包含main.js，one.js，two.js和three.js的内容。通常情况下，不应该使用原始项目源保存优化文件。 通常将它们保存到的项目副本中，将out =选项更改为您喜欢的任何目录，其中包含源代码的副本。 然后，您可以将main-built.js文件名更改为main.js，以便HTML页面将加载文件的优化版本。如果你想在main.js源文件中包含require.js，你可以使用这种命令：

node ../../r.js -o baseUrl=. paths.requireLib=../../require name=main include=requireLib out=main-built.js

细粒度优化：优化时指定某个js 文件为单独文件并参与浏览器调试

excludeShallow选项来完成此操作，上面的示例设置，假设您正在构建或调试two.js（不想把two.js一起优化）你可以使用这个优化命令

node ../../r.js -o name=main **excludeShallow=two** out=main-built.js baseUrl=.

不希望main-build.js文件缩小，请在上面的命令中传递optimize = none

如果想要从网络或其他域中加载脚本。优化程序无法加载网络资源，但希望它包含在构建中，请确保创建一个路径配置，以将该文件映射到模块名称。然后，为了运行优化器，请下载CDN脚本并将路径配置传递给将模块名称映射到本地文件路径的优化器。但是，您更不可能将该资源包含在构建中。如果脚本没有任何依赖关系，或者您不想包含它的依赖关系，或者以其他方式包含它们，那么您可以在路径配置中使用特殊的'empty：'方案，以便在执行优化

在main.js文件中，创建一个路径配置，为该脚本提供一个模块名称。即使脚本没有通过调用define（）来定义模块，也可以这样做。路径配置仅用于将短模块/脚本ID映射到URL。这使您可以使用不同的路径配置进行优化。在main.js中：



运行优化器时，使用'empty：'作为路径config：

node ../../r.js -o name=main out=main-built.js baseUrl=. paths.jquery=empty:

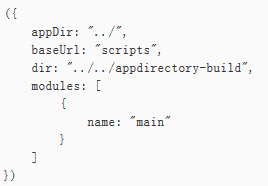


优化一个css文件

使用上面的示例设置，如果您只想优化main.css，则可以从appdirectory / css目录内使用此命令：node ../../r.js -o cssIn=main.css out=main-built.css

优化整个项目

优化器可以通过使用构建配置文件来优化项目中的所有CSS和JS文件。

创建一个构建配置文件，将其称为app.build.js，并将其放入脚本目录中。 app.build.js文件可以存在于任何地方，但只要确保在下面的示例中相应地调整路径 - 所有路径都将相对于app.build.js所在的位置。 示例app.build.js：

此构建配置文件告诉RequireJS将所有appdirectory复制到称为appdirectory-build的同级目录，并在appdirectory-build目录中应用所有优化。强烈建议您使用与源目录不同的输出目录 - 否则在优化器覆盖源代码时可能会发生不好的事情。

RequireJS将使用baseUrl来解析任何模块名称的路径。 baseUrl应该与appDir相关。

在模块数组中，指定要优化的模块名称，例如“main”。 “main”将被映射到项目中的appdirectory / scripts / main.js。然后构建系统将跟踪main.js的依赖关系并将它们注入到appdirectory-build / scripts / main.js文件中。

它还会优化在appdirectory-build中找到的任何CSS文件。

优化多页面项目

requirejs / example-multipage是一个具有多个页面的项目示例，但共享一个通用配置和一个通用优化构建层。

该项目具有以下布局：

     tool：requirejs优化器r.js和优化器配置build.js。

     www：在开发模式下运行在浏览器中的代码。

     www-built：在优化器构建之后生成。 包含可以部署到现场的构建代码。

这个www有以下布局：

     page1.html：应用的第1页。

     page2.html：第2页的应用程序。

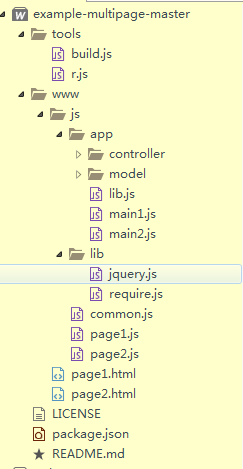
     JS 应用程序：存储应用程序特定模块的目录。

     lib：用来存放第三方模块的目录，比如jQuery。

     common.js：包含requirejs配置，它将成为该组公共模块的构建目标。

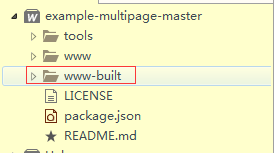
         page1.js：用于page1.html的data-main。 加载公共模块，然后加载app / main1，第1页的主要模块。

         page2.js：用于page2.html的data-main。 加载公共模块，然后加载app / main2，第2页的主要模块。



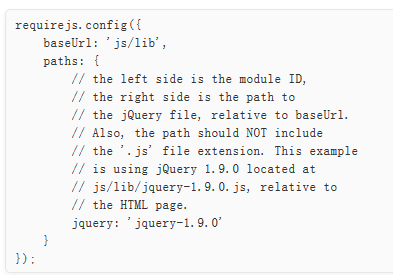
执行如下命令：node tools/r.js -o tools/build.js

该构建命令在www-built目录中创建项目的优化版本。 js / common.js文件将包含所有通用模块。 js / page1.js将包含page1特定的模块，js / page2.js将包含page2特定的模块。



如何在jQuery中使用RequireJS

jQuery在检测到AMD / RequireJS时定义了命名的AMD模块“jquery”（全部小写）。 为了减少混淆，我们建议在你的requirejs.config中使用'jquery'作为模块名称。



另一种（推荐的）解决方案是只命名文件'jquery.js'并将其放在baseUrl目录中。 那么上面的路径条目是不需要的。通过根据baseUrl + moduleID +'.js'的默认ID到路径约定放置文件，可以避免大量配置行。 以下示例显示如何将baseUrl设置为第三方库代码的目录，并为您的应用程序代码使用一个额外的路径配置。

