

test

chenyiqiao

Published
with GitBook



Table of Contents

Introduction	0
我们为什么需要Webpack？	1
Webpack是什么？	2
安装Webpack	3
模块化方案	4
使用webpack	5
使用教程和示例	5.1
使用加载器	5.2
使用插件	5.3
加载CSS	5.4
怎样写一个加载器？	5.5
代码分割	6

Webpack官方文档中文翻译

我们为什么需要Webpack？

现在的网站都在演变成为Web Apps:

- 页面上的JavaScript越来越多。
- 在现代浏览器上用户可以做更多的事情了。
- 整个页面重新加载的情况更少了，与此同时，页面上的代码量更大了。

结果就是：客户端的代码量变得越来越庞大，庞大的代码量意味着我们需要适当地组织代码，而模块系统则提供了把代码分割成不同模块的功能。

模块系统实现的演变

关于怎么定义依赖和导出接口有多种标准：

- Script标签形式
- CommonJS
- AMD和一些变种实现
- ES6模块
- 其它

script标签形式

如果不对代码进行模块化，那么通常的做法就是使用标签引入script文件：

```
<script src="src=module1.js"></script>
<script src="src=module2.js"></script>
<script src="librariesA.js"></script>
<script src="src=module3.js"></script>
```

模块把接口导出到全局对象上，比方说window对象上。也就是说模块可以在全局对象上获取到依赖的接口。

这样做的问题是：

- 全局对象上可能会出现命名冲突
- 加载的顺序很重要
- 程序员必须要手动处理模块/库的依赖问题
- 在大型项目里面，这个列表可能会很长、很难管理

CommonJS:同步的require

CommonJS采用同步的require方法来加载依赖并返回导出的接口。一个模块可以通过往exports对象上添加属性或者设置module.exports的值来确定导出哪些接口。

```
require("module");
require("../file.js");
exports.doStuff = function(){};
module.exports = someValue;
```

后端的nodejs用的就是这个方法。

优点：

- 服务端的模块可以被复用
- 已经有很多模块供使用了（npm）
- 很容易上手使用

缺点：

- 阻塞式的调用不能适用于网络请求，因为网络请求是异步的
- 无法同步require多个模块

使用CommonJS的项目：

- Nodejs -服务端
- Browserify
- Module-webmake -编译成一个bundle
- Wreq -服务端

AMD:异步require

之前提到的模块系统只能同步require，而AMD则采用异步的实现形式。

```
require(["module", "../file.js"], function(module, file){/*...*/});
define("mymodule", ["dep1", "dep2"], function(d1, d2){
    return someExportValue;
});
```

优点：

- 能满足网络请求的异步需求
- 能同步加载多个模块

缺点：

- 代码复杂，更难写也更难读
- 似乎是一种绕路的笨办法

使用AMD的项目：

- Require.js -客户端
- Curl -客户端

ES6模块

EcmaScript6给JavaScript加了一些语言特性，其中就包括了模块系统。

```
import "jQuery";
export function doStuff(){}
module "localModule"{}
```

优点：

- 静态分析变得更简单了
- 作为ES标准，这个实现未来肯定会成为主流

缺点：

- 浏览器原生支持还需要时间
- 目前只有很少的一些模块可用

最合适（unbiased）的解决办法

让开发人员选择合适的模块系统，让代码能跑起来，也能使得添加自定义模块更简单易行。

模块的传输

模块需要在客户端执行，因此它们需要从服务器传输到浏览器上。

有两个极端的方法来传输模块：

- 每传输一个模块发起一个请求
- 所有的模块都放在一个请求里

这两种方法都有人在用，但是都不是最优解：

- 每传输一个模块发起一个请求
 - 优点：只会请求目前需要的模块
 - 缺点：很多请求意味着会有很多额外的消耗

- 缺点：请求延迟会使得程序启动变慢
- 所有的模块放在一个请求里
 - 优点：更少的请求意味着更低的延迟
 - 缺点：无论目前需要与否，所有的模块都一次性传输过来了

分块传输

我们需要一个更灵活的传输方式，在大多数情况下，如果能在极端中间找到一个平衡会是最好的选择。比方说，在编译所有的模块的时候可以把模块分割成很多小模块。这样一来，我们就可以把请求分成很多小请求，而分割后的模块只有在需要的时候才会被请求。所以初始的请求不会包含所有的代码，从而减小传输压力。至于代码怎么分割由程序员决定。

这个想法来源于Google's GWT. 更多的信息在code splitting.

为什么只处理JavaScript？

为什么模块系统只能处理JavaScript？还有很多其他的静态资源需要处理：

- 样式表
- 图片
- 字体
- html模板
- 等等

还有：

- coffeescript -> javascript
- less 样式表 -> css 样式表
- jade 模板 -> javascript which generates html
- i18n files -> something
- 等等

这些都应该很容易处理才对：

```
require("./style.css");
require("./style.less");
require("./template.jade");
require("./image.png");
```

这些都可以做到，更多相关的信息在Using loader和loader章节。

静态分析

在编译所有的模块的时候，静态分析会试图去找到依赖。通常来说在没有表达式的情况下只能找到一些简单的东西，但是像

```
require("../template/" + templateName + ".jade")
```

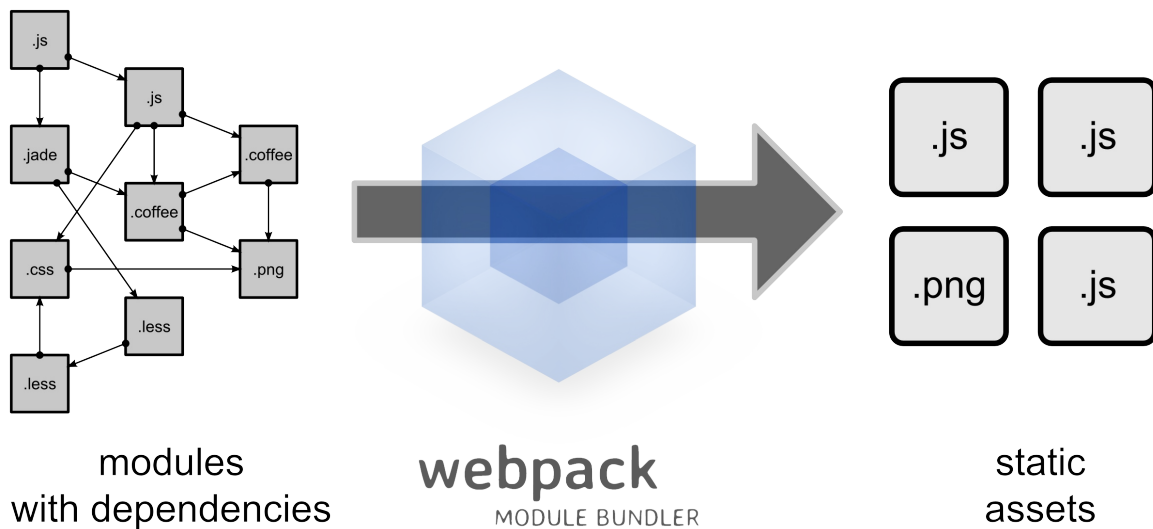
这样的表示方法是很常见的形式。不同的库会用不同的形式表示，有一些看起来很奇怪...

分析策略

一个智能的解析器会使得大多数代码能跑起来。即便程序员写了些很奇怪的代码，它还是能找到最兼容的办法。

Webpack是什么？

Webpack是一个模块打包的工具，它的作用是把互相依赖的模块处理成静态资源。



为什么又来一个新的模块打包工具？

现有的模块打包工具不适合大型项目（大型的SPA）的开发。当然最重要的还是因为缺少代码分割功能，以及静态资源需要通过模块化来无缝衔接。Webpack的作者曾经试图对原有的打包工具进行扩展，但是没能成功。Webpack的目标：

- 把依赖树按需分割
- 把初始加载时间控制在较低的水平
- 每个静态资源都应该能成为一个模块
- 能把第三方库继承到项目里来成为一个模块
- 能定制模块打包器的每个部分
- 能适用于大型项目

Webpack有哪些特别的地方？

代码分割

在Webpack的依赖树里有两种类型的依赖：同步依赖和异步依赖。异步依赖会成为一个代码分割点，并且组成一个新的代码块。在代码块组成的树被优化之后，每个代码块都会在一个单独的文件里。

更多关于code splitting的信息。

加载器

Webpack原生是只能处理JavaScript的，而加载器的作用把其它的代码转换成JavaScript代码，这样一来所有种类的代码都能组成一个模块。

更多关于loaders的信息。

智能解析

Webpack的智能解析器能处理几乎所有的第三方库，它甚至允许依赖里出现这样的表达式：

```
require("../templates/"+ name + ".jade")
```

它能处理大多数的模块系统，比如说CommonJS和AMD.

插件系统

Webpack有丰富的插件系统，大多数内部的功能都是基于这个插件系统。这也使得我们可以定制Webpack，把它打造成能满足我们的需求的工具，并且把自己做的插件开源出去。

安装Webpack

node.js

安装node.js，同时npm也会随node.js一起安装。

Webpack

Webpack可以通过npm安装：

```
$npm install webpack -g
```

然后Webpack就全局安装了，可以在命令行里用webpack命令了。

在项目里使用webpack

最好是把Webpack本身也当作一个依赖放到项目里去，这样的话就可以选择一个本地的Webpack版本而不用使用全局的了。

添加一个配置文件package.json:

```
$npm init
```

如果不打算把这个项目发布到npm上去的话，这里面的问题全部跳过就好。然后安装Webpack并把它加到package.json里:

```
$npm install webpack --save-dev
```

Webpack分成稳定版和β版，β版里会有一些试验性的东西，如果是用于生产的项目，那就用稳定版本。如果要用开发工具也可以安装:

```
$npm install webpack-dev-server --save-dev
```

模块化方案

CommonJS

CommonJS的实现很简单。在CommonJS里面，文件和模块就是一一对应的关系。比方说，在同一个文件夹下，foo.js加载circle.js是这么写的：

foo.js:

```
var circle = require("./circle.js");
console.log('The area of a circle of radius 4 is' + circle.area(4));
```

circle.js :

```
var PI = Math.PI;
exports.area = function(r){
    return PI * r * r;
};

exports.circumference = function(r){
    return 2 * PI * r;
};
```

在上面这个例子里面，circle.js这个模块导出了它的两个方法。注意，模块里的局部变量是私有的，比如circle.js里的PI.如果需要导出多个变量或者方法，可以采用module.exports的写法：

```
module.exports = function(){
    return {
        //返回需要导出的变量或方法
    }
}
```

AMD

AMD实现了JavaScript模块的异步加载，它的实现形式是：

```
define(id?: String, dependencies?: String[], factory: Function|Object);
```

id定义了模块的名字，是选填项。

dependencies指定了该模块所依赖的模块，它也是选填项，如果缺省了，那么默认为：
["require", "exports", "module"]。

factory定义了模块的方法，它可以是个函数，也可以是个对象。如果是函数的话，它所返回的值将会被这个模块导出。

具名模块：

```
define('myModule', ['jquery'], function($) {  
    // $ is the export of the jquery module.  
    $('body').text('hello world');  
});  
// and use it  
define(['myModule'], function(myModule) {});
```

多个依赖：

```
define(['jquery', './math.js'], function($, math) {  
    // $ and math are the exports of the jquery module.  
    $('body').text('hello world');  
});
```

导出自身：

```
define(['jquery'], function($) {  
    var HelloWorldize = function(selector){  
        $(selector).text('hello world');  
    };  
    return HelloWorldize;  
});
```

使用webpack

Webpack提供了一个配置对象，基于你使用webpack的方式可以有两种方式来传递这个配置对象。

CLI

如果你用CLI来读取webpack.config.js里的内容，这个文件应该要导出这个配置对象：

```
module.export = {  
  //configuration  
};
```

node.js API

如果你用node.js API，你需要把这个配置对象当作参数传递：

```
webpack({  
  //configuration  
},callback);
```

配置对象的内容

提示：这个配置对象就是纯javascript，它就是个node.js模块，这里举个很简单的配置文件的例子：

```
{  
  context : _dirname + "/app",  
  entry : "/entry",  
  output : {  
    path : _dirname + "/dist",  
    filename : "bundle.js"  
  }  
}
```

context

context就是解析入口点的绝对路径。如果输出路径信息设定好了，那么与之相关的路径信息就被缩成了这个目录。

entry

entry point就是文件打包的入口点，如果传递的是一个字符串，那么这个字符串就会被在初始化时被当作一个模块处理。如果你传递的是一个字符串数组，所有的模块都立即加载，最后一个会被导出：

```
entry : ["../entry1", "./entry2"]
```

如果你传递的是一个对象，那就会创建多个入口，也就是打了多个包。键就是包的名字，值可以是一个字符串或者一个数组。

```
{
  entry :{
    page1:"./page1",
    page2:["./page1", "./page2"]
  },
  output:{
    //make sure to use [name] or [id] in output.filename
    //when using multiple entry points
    filename:"[name].bundle.js",
    chunkFilename:"[id].bundle.js"
  }
}
```

output

选项影响输出。如果你用任何哈希([hash]或者[chunkhash])，注意模块要有统一的顺序，使用OccurenceOrderPlugin或者recordsPath.

output.path

这个属性对应的是输出目录的绝对路径,[hash]被编译的哈希值所替代。

output.filename

这个属性代表每个包的相对路径。当然，它是在output.path目录下的。[id]被包的id所替代，[name]被包的名字替代（包的名字不存在的时候就用id顶替），[hash]被编译的哈希值替代，[chunkhash]被包的哈希值替代。

使用教程和示例

安装webpack

首先需要安装nodejs(npm也会一起安装)。然后:

```
npm install webpack -g
```

这个命令敲过之后就可以在命令行使用webpack命令了。

组织编译

在一个文件夹下添加两个文件：

entry.js：

```
document.write("It works.");
```

index.html:

```
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <script type="text/javascript" src="bundle.js" charset="utf-8"></script>
  </body>
</html>
```

然后在命令行：

```
webpack ./entry.js bundle.js
```

它会编译你的文件然后创建一个打包文件，成功之后会显示这些信息：


```
Time: 43ms
   Asset      Size  Chunks             Chunk Names
bundle.js  1.42 kB       0  [emitted]  main
chunk      {0} bundle.js (main) 28 bytes [rendered]
```

打开index.html会显示It works.

然后在文件夹下添加一个content.js:

```
module.exports = "It works from content.js";
```

并修改entry.js:

```
document.write(require("./content.js"));
```

再敲入：

```
webpack ./entry.js bundle.js
```

刷新浏览器会看到显示:

```
It works from content.js
```

Webpack会分析入口文件来寻找依赖的模块，这些模块会被添加到bundle.js，webpack会给每个模块一个id，并且在bundle.js可以通过id来找到这些模块。在初始阶段只会执行入口模块，然后按需执行依赖的模块。

Loader

Webpack本身是只能处理JavaScript的，要是需要在应用里加CSS文件，那么我们就需要css-loader来处理CSS文件，另外还需要style-loader来把样式加到CSS文件里面。

局部安装Loader：

```
npm install css-loader style-loader
```

添加style.css：

```
body{  
  background:yellow;  
}
```

修改entry.js:

```
require("./style.css");  
document.write(require("./content.js"));
```

编译运行：

```
webpack ./entry.js bundle.js --module-bind "css=style!css"
```

刷新浏览器会看到背景颜色变成黄色了。

配置文件

也可以把所有的配置都放到一个配置文件里，添加一个webpack.config.js:

```
module.exports = {  
  entry : "./entry.js",  
  output : {  
    path : __dirname,  
    filename : "bundle.js"  
  },  
  module : {  
    loaders : [  
      {test : /\.css$/, loader:"style!css"}  
    ]  
  }  
};
```

然后只要运行：

```
webpack
```

就会得到结果：

```
Time: 354ms  
   Asset      Size  Chunks             Chunk Names  
bundle.js  10.7 kB          0 [emitted]  main  
chunk      {0} bundle.js (main) 8.86 kB [rendered]
```

Watch Mode

如果运行这个命令：

```
webpack --watch
```

webpack就会在编译阶段对每个文件进行监视，如果监测到变化了，就会重新编译并加到缓存，如果没有变化就直接使用缓存。

Dev Server

用开发服务器就更方便了：

```
npm install webpack-dev-server -g
```

```
webpack-dev-server
```

这个配置会在Localhost:8080绑定一个express服务器，这个服务器会提供静态资源和打包文件，当打包文件重新编译的时候，它会自动个更新浏览器页面。

使用加载器

什么是加载器？

加载器是对你的应用的源文件进行转换的工具。比如说你可以用加载器来加载coffeescript、CSS或者JSX.

加载器的特征

- 可以对源文件进行链式操作，最后一个加载器将返回js文件，中间的加载器则返回一些特定格式的文件。
- 可以是异步操作，也可以是同步操作。
- 在Nodejs下跑。
- 可以接收查询参数，这一点可以用来向加载器传递配置参数。
- 可以在配置文件里限定文件后缀/正则表达式。
- 可以通过npm发布和下载。
- 可以access到配置文件。
- 插件可以给加载器更多功能。
- etc

使用加载器

通过配置文件操作

首先安装加载器：

```
npm install jade-loader css-loader style-loader --save-dev
```

在配置文件里用正则表达式来绑定加载器：

```
{
  module: {
    loaders: [
      { test: /\.jade$/, loader: "jade" },
      // => "jade" loader is used for ".jade" files

      { test: /\.css$/, loader: "style!css" },
      // => "style" and "css" loader is used for ".css" files
      // Alternative syntax:
      { test: /\.css$/, loaders: ["style", "css"] },
    ]
  }
}
```

通过CLI操作

以上的方法是通过配置文件来操作，如果直接在CLI上操作，那么可以这样写：

```
webpack --module-bind jade --module-bind 'css=style!css'
```

查询参数

加载器也可以附上查询参数，就跟HTTP请求里一样。

配置文件的方法：

```
{test : /\.png$/, loader: "url-loader?mimetype=image/png"}
```

CLI：

```
webpack --module-bind "png=url-loader?mimetype=image/png"
```


注：这一章省略了不少内容，只提取了常用的一些东西。

使用插件

内置插件

在配置文件webpack.config.js里把插件属性加进去之后就能使用插件了：

```
var webpack = require("webpack");
module.exports = {
  plugins: [
    new webpack.ResolverPlugin([
      new webpack.ResolverPlugin.DirectoryDescriptionFilePlugin("bower.json",
        ], ["normal", "loader"])
    ]
  ];
};
```



如果不是内置的插件，那就先要安装：

```
npm install component-webpack-plugin
```

然后再用：

```
var ComponentPlugin = require("component-webpack-plugin");
module.exports = {
  plugins: [
    new ComponentPlugin()
  ]
}
```

加载CSS

用style-loader和css-loader就可以把样式表加到JS打包文件里。这样一来，就能把样式表和其他的模块相嵌了。配置好之后，加载CSS只需要一行代码：

```
require("./stylesheet.css")
```

安装加载器

```
npm install style-loader css-loader --save-dev
```

配置

```
{
  //...something up there
  module :{
    loaders : [
      {test : /\.css$/, loader : "style-loader!css-loader"}
    ]
  }
}
```

如果要用`less`、`sass`这些语言写CSS，也有对应的加载器。

怎样写一个加载器？

关于加载器

Loader是一个帮你预处理JS之外的文件的工具，它有点像是gulp这些构建工具里的Tasks，它可以把coffee转成js，也可以把images转成url形式的数据。通过使用Loader，我们可以在JS文件里require css文件。

要用Loader转换模块，可以在一个require调用里完成：

```
var moduleWithOneLoader = require("my-loader!./my-module");
```

上面这个例子里面Loader使用名字的形式指定的，当然我们也可以用相对路径来指定Loader：

```
require("./loaders/my-module!./my-module");
```

Loader也可以进行链式操作：

```
require("style-loader!css-loader!less-loader!./my-styles.less");
```

链式操作是从右至左进行的，在上面这个例子里面，my-style.less会首先通过less-loader来转换成css，再通过css-loader处理诸如urls，字体等资源，然后传到style-loader来放到一个style标签里去。

另外，Loader还能用查询参数：

```
require("loader?with=parameter!./file");
```

在配置文件里配置Loader

通过require来配置Loader意味着需要几个Loader就要写几个require，这显然不是个好办法。

我们还可以用在配置文件里使用Loader:


```
{
  module:{
    loaders:[
      {test:/\.coffee$/,loader:"coffee-loader"}
    ],
    preLoaders:[
      {test:/\.coffee$/,loader:"coffee-hint-loader"}
    ]
  }
};
```

Loader的顺序

当文件被读取的时候，Loaders以以下的顺序执行：

1. preLoaders
2. loaders
3. require里指定的loaders
4. postLoaders

你也可以修改require的写法来修改Loader的加载顺序：

- 在require里添加！会使得配置文件里的预加载器失效
 - `require("!raw!./script/coffee");`
- 在require里添加！！会使得配置文件里所有的加载器失效
 - `require("!!raw!./script/coffee");`
- 在require里添加-会使得配置文件里的加载器和预加载器失效
 - `require("-!raw!./script.coffe");`

建议

暂空

写一个加载器

写一个加载器是件挺简单的事，一个加载器就是一个导出一个方法的文件（这个方法就是处理源文件的方法），编译器把上一个加载器的结构当作参数传进去然后调用这个函数，编译器会往这个函数的this绑定一些诸如把加载从同步改成异步、获取查询参数之类的方法。第一个加载器的参数就是源文件，最后一个加载器则输出期望的结果，这个结果应该是个字符串或者buffer(被转成字符串)，也就是这个模块经过处理之后得到的JS代码。还有一个可选的结果是SourceMap,一个JSON对象。

如果只有一个结果，那就会以同步形式返回。有多个结果的话，`this.callback`一定会被调用。在异步模式下则会调用`this.async()`，如果异步模式是可行的，它会返回`this.callback`，然后加载器就必须返回`undefined`并且调用`callback`。

同步加载器

```
module.exports = function(){
  return someSyncOperation(content);
};
```

异步加载器

```
module.exports = function(){
  var callback = this.async();
  if(!callback) return someSyncOperation(content);
  someAsyncOperation(content, function(err, result){
    if(err) return callback(err);
    callback(null, result);
  })
}
```

注意：最好在异步不可行的时候`fallback`到同步的模式

原始加载器

pitching加载器

加载器的一些参数

代码分割

对于大型web app来说，把所有的代码打包到一个文件里面显然是不高效的，如果有一些代码块只在特定情况下才需要加载的话，这样做就更不合适了。Webpack针对这个问题有一个叫做代码分割的方法，这个方法把你的代码分割成块，按需加载。

代码分割是一个可选的功能，由你来决定分割点，webpack会解决依赖、输出、运行时的问题。必须要澄清一个误会，代码分割不是把共用的代码提取出来打成一个包。Webpack更强大的地方是把代码分割成块之后按需加载，这样一来初始加载时间会更短，而且所有代码块都只在请求的时候下载。

确定分割点

AMD和CommonJS有不同的方法来加载代码。

CommonJS

```
require.ensure(dependencies, callback)
```

require.ensure方法可以确保在调用callback的时候能同步加载依赖，callback以require为参数。举个例子：

```
require.ensure(['module-a', 'module-b'], function(require){
  var a = require("module-a");
  //...
})
```

注意：*require.ensure*只会加载模块，不会计算模块。

require(AMD)

```
require(dependencies, callback)
```

调用的时候，所有的依赖都被加载。举个例子：

```
require(["module-a", "module-b"], function(a, b){
  //...
})
```

注意：AMD的`require`方法会加载并计算模块，从左至右计算。`callback`可以省略。

代码块的内容

所有在分割点上的依赖都会变成一个新的代码块。如果你把你一个函数表达式作为分割点的回调，webpack会自动把函数里所有的依赖都加到代码块里。

代码块的优化

- 如果两个代码块包含同一个模块，它们将会合并成一个代码块。
- 如果一个模块在它的所有的父代码块都可以`access`，那么模块将被从移除。

代码块的加载

代码块的类型