初学教程1部分- Webpack介绍Webpack

我希望这个教程可以使Webpack易学。

# 必要条件

至少你会知道Node.js和npm的基础知识。

# 目录

1为什么Webpack？

2基础知识

安装

命令行

打包

装载

插件

3您的配置文件

一个小的例子

引入插件

4一个更完整的例子

介绍了装载机

添加更多的插件

开发服务器

开始编码

5结论

6最后的想法

# 为什么要学Webpack？

因为每一个react/redux教程都假设我们知道，但是往往我们有一些更现实的原因使我们想使用webpack。

可能有这些：

压缩我们的js文件合并成一个文件

在我们的前端代码项目中使用NPM包管理工具

使用ES6/ES7规范书写代码（借助babel）

缩小/优化代码

编译LESS/SCSS成CSS

使用HMR（Hot Module Replacement/实时的模块监听改变）

把任何类型的文件放进我们的javascript中

我们需要了解、掌握更多先进和潮流的技术

## 为什么我们想要上面这些？

压缩js文件——让我们可以编写模块化的javascript代码，却不需要为把它放进单独的js文件来被单个的<script>标签所引用。（如果我们需要配置多个js文件时）

使用NPM包管理在我们的前端代码项目——NPM是互联网上最大的开源代码生态系统。我们可以试试把写好的代码保存上传到NPM看看，把想要的库放进你的前端项目。

ES6/ES7——为javascript增加了很多新特性，使它更具有潜力又容易编写。看看这里的介绍：<https://github.com/DrkSephy/es6-cheatsheet>

缩小/优化代码——减少我们发布文件的大小，就有助于让我们页面更快地加载的。

编译LESS/SCSS成CSS——更好的方式去编写CSS代码。如果你不熟悉可以看这里的介绍：<http://alistapart.com/article/why-sass>

使用HMR-生产力的提高，每次保存代码的时候，只要他是被引入到该页面那么不需要完整的页面刷新。这在我们编辑代码时候是非常方便的。

把任何类型的文件包含进我们的javascript中——减少其他构建工具的需要，并允许我们用代码的方式修改和使用这些文件。

# 基础知识

## 安装

想使用Webpack大多数的功能只需要我们全局安装一下：

npm install -g webpack

然而Webpack的一些功能，比如优化插件，需要我们把它安装到本地。在这种情况下我们需要：

npm install --save-dev webpack

## 命令行

运行Webpack只需要下面一行命令：

webpack

webpack --display-error-details

如果想要Webpack在我们改变文件的同时监听改变并重新构建：

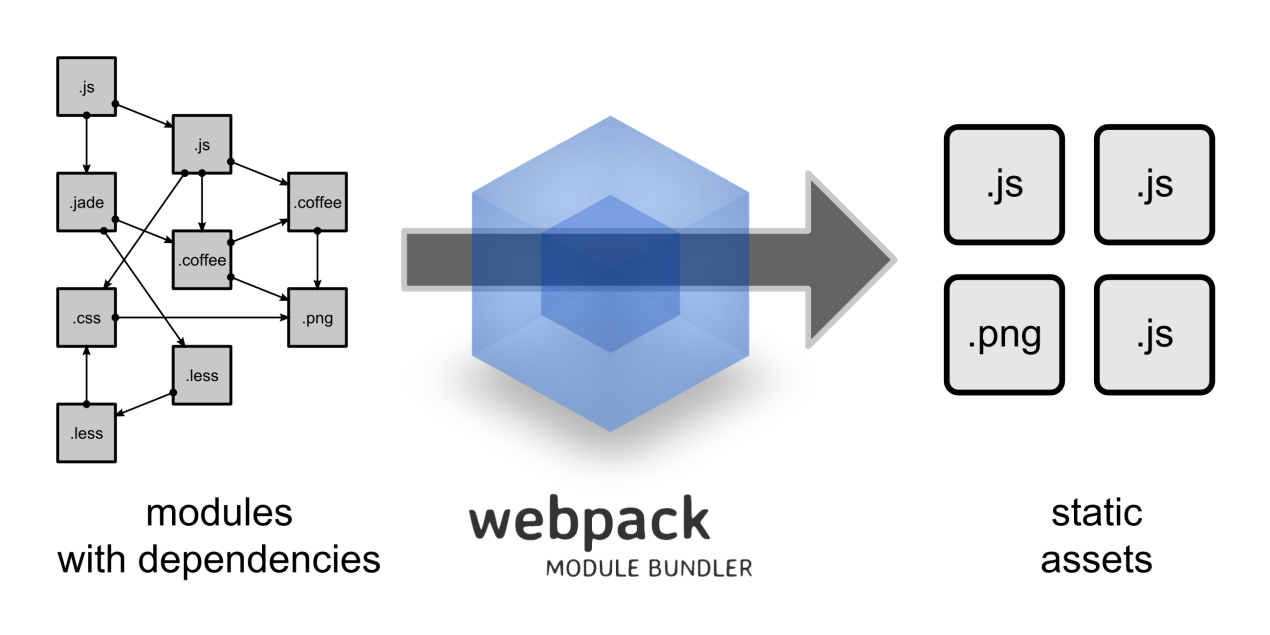
webpack –watch

如果想要使用自定义名称的Webpack的配置文件:

webpack --config myconfig.js

## 打包

第一个例子



Webpack被正式称为一个模块打包工具。如果你想对模块和模块化打包有深入且易于理解的解释，肯定要看看这两篇文章：

<https://medium.freecodecamp.com/javascript-modules-a-beginner-s-guide-783f7d7a5fcc#.jw1txw6uh>

<https://medium.com/@preethikasireddy/javascript-modules-part-2-module-bundling-5020383cf306#.lfnspler2>

我们保持它的简单。它的工作方式是，我们制定一个单独的文件为入口点。这个文件将是根节点。然后我们每次都需要引用一个文件都是被添加到节点上的文件。当我们运行Webpack时，所有这些文件会被打包成一个文件。

下面是一个简单的例子：

Dependency Tree

鉴于这张图片，我们可以有目录：

mydirectory  
| - index.js  
| - uistuff.js  
| - apistuff.js  
| - styles.css  
| - extrafile.js

这可能是你的文件的内容

// index.js

require('./styles.css')

require('./UIStuff.js')

require('./APIStuff.js')

// UIStuff.js

var React = require('React')

React.createClass({

// stuff

})

// APIStuff.js

var fetch = require('fetch') // fetch polyfill

fetch('https://google.com')

/\* styles.css \*/

body {

background-color: rgb(200, 56, 97);

}

当我们运行Webpack时，我们将得到一个节点内容的打包，其中没有extraFile.js，虽然文件都在同一个目录，结果却不会打包是因为我们没有引用。

打包的js文件将会像这样

// contents of styles.css

// contents of UIStuff.js + React

// contents of APIStuff.js + fetch

被打包压缩的东西只会是我们在文件中明确要求的东西

## 加载器

正如你可能注意到的，我在上面的例子做了依稀奇怪的事情。我引用了一个css文件到我需要的javascript文件中。

这是很酷的，这是Webpack的一桩非常有趣的功能，就是我们可以引用不仅仅只是javascript文件。

这种功能在Webpack中被叫加载器。使用这些加载器，我们可以引用css、html、png文件。

如下上面图片中一样，我这样做：

/\*index.js\*/

require('./styles.css')

如果我引用了style-loader和css-loader在我跑得webpack配置，这不仅是完全有效的，而且也会应用css到我的页面中来。

这只是我们可以使用Webpack的许多加载器（loader）的其中一个简单例子。

## 插件

插件，正如其名，可以为Webpack添加额外的功能。一个经常使用的插件UglifyJsPlugin，

可以缩小你的javascript代码。我们等下会讨论怎么使用。

# 你的配置文件

Webpack不会在盒子外自动运行，所以你必须按你的需要。为此，你需要创建一个

//webpack.config.js

这是Webpack默认的配置名称，如果你选择使用自定义的名称，必须使用“配置标识命令”来自定该特殊的配置文件名称。

## 一个小例子

例2

你的目录结构是这样的

|- dist

|- src

|- index.js

|- webpack.config.js

然后我们可以使用一个很小的webpack配置：

// webpack.config.js

var path = require('path')

module.exports = {

entry: ['./src/index'], // file extension after index is optional for .js files

output: {

path: path.join(\_\_dirname, 'dist'),

filename: 'bundle.js'

}

}

我们来一个一个看这些新特性：

* entry – 打包入口，我们想要打包部分文件。这个配置是一个数组，因为Webpack允许有个入口文件，如果我们想要生成多个打包文件。
* output – Webpack的输出形式
* path – 打包文件地址
* filename - 打包文件名

当我们运行Webpack时，这就会创建一个叫bundle.js的文件在我们的dist文件夹。

## 引入插件

例3

想象一下，我们使用Webpack来一起打包我们全部的文件，现在我们知道它们全部打包在一起会有900KB。去缩小我们的捆版好的打包文件是一个问题。因为我们需要使用一个之前提到过的叫UglifyJsPlugin的插件。

此外，我们必须得在本地安装Webpack以使用插件。

npm install --save-dev webpack

现在我们可以引入webpack来缩小我们的代码

// webpack.config.js

var path = require('path')

var webpack = require('webpack')

module.exports = {

entry: ['./src/index'],

output: {

path: path.join(\_\_dirname, 'dist'),

filename: 'bundle.js'

},

plugins: [

new webpack.optimize.UglifyJsPlugin({

compressor: {

warnings: false,

},

})

]

}

我们来看看这个新特性：

* plugins – 保存我们插件的数组
* webpack.optimize.UglifyJsPlugin – 压缩我们的代码，并抑制警告信息。

这一次，当我们运行Webpack时，现在因为我们有UglifyJsPlugin插件，这可以通过比如删除所有空格，可以减少我们本来的900KB文件到200KB。

我们还可以添加OccurenceOderPlugin插件

通过计算模块出现次数来分配模块。这个经常被使用可以较快地获得模块。这使得模块可以预读，建议这样可以减少总文件大小。

事实上，我们是不知道它背后的机制的，但是在目前的测试的webpack2中已经被默认包含了，所以我们不妨引用它。

// webpack.config.js

var path = require('path')

var webpack = require('webpack')

module.exports = {

entry: ['./src/index'],

output: {

path: path.join(\_\_dirname, 'dist'),

filename: 'bundle.js'

},

plugins: [

new webpack.optimize.UglifyJsPlugin({

compressor: {

warnings: false,

},

}),

new webpack.optimize.OccurenceOrderPlugin()

]

}

所以现在我们已经谢了一个配置，使用我们能够缩小和打包我们的javascript。这个打包文件可以复制并粘贴到另一个项目文件中，并被扔进一个<script>标签中。你可以跳到这个结论，如果你只在乎这个使用webpack仅仅是javascript的基础知识。

# 一个更完整例子

另外，因为webpack还能做不仅仅只是关于javascript的工作，我们可以避免复制粘贴和使用webpack管理我们的整个项目。

在接下来的部分中，我们将使用webpack来创建一个非常简单的网站。如果你希望跟随这个例子，可以创建一个如下的目录结构：

MyDirectory

|- dist

|- src

|- index.js

|- index.html

|- styles.css

|- package.json

|- webpack.config.js

## 内容

1.介绍加载器（loaders）

我们将增加加载器，来雨荨我们添加css到我们的打包中。

2.添加更多插件（plugins）

我们将添加一个插件来帮助我们 创建/使用 一个HTML文件。

3.开发服务器

我们会把我们的webpack配置分割独立的开发和生产文件。然后使用webpack–dev-server 来查看我们的网站，使它能够HMR。

4.开始编码

我们将会写一写javascript。

## 介绍装载器

例4

在前面的教程中，我提到了加载器。这将有助于在我们的代码中引用非js文件。在这种情况下，我们将需要style-loader和css-loader。首先 我们需要安装这些加载器：

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

现在，安装好了它们之后我们可以调整我们的的配置把这些css的加载器包含进来。

// webpack.config.js

var path = require('path')

var webpack = require('webpack')

module.exports = {

entry: ['./src/index'],

output: {

path: path.join(\_\_dirname, 'dist'),

filename: 'bundle.js'

},

plugins: [

new webpack.optimize.UglifyJsPlugin({

compressor: {

warnings: false,

},

}),

new webpack.optimize.OccurenceOrderPlugin()

],

module: {

loaders: [{

test: /**\.**css$/,

loaders: ['style', 'css']

}]

}

}

我们来理解这些新特性。

* module-影响我们的文件的选项
* loaders-一个我们为应用程序指定的一个加载器的数组
* test-一个正则表达式，将加载器和文件匹配
* loaders-该正则匹配的文件应用的加载器

当你运行WebPACK，如果你rquire一个.css文件，那么我们将请用style和css的加载器，来打包添加CSS。

如果没有加载器，那么我们会得到一个这样的错误。

ERROR in ./test.css

Module parse failed: /Users/Developer/workspace/tutorials/webpack/part1/example1/test.css

Line 1: Unexpected token {

You may need an appropriate loader to handle this file type.

### **可选**

如果你想使用SCSS来代替CSS，你需要运行：

npm install --save-dev sass-loader node-sass webpack

你的加载器将改写成这个样子

{

test: /**\.**scss$/,

loaders: ["style", "css", "sass"]

}

这个过程和LESS差不多。

识别的一个重要的点是，这些加载器需要指定一个特定顺序。在上面的例子中，这个sass-loader是优先作用于我们的.scss文件，然后是css-loader，最后是style-loader。正如所见，该模式是应用加载器的优先级从右向左的。

## 添加更多的插件

例5

现在我们有基础的构建在我们的网站，我们需要一个实际的页面样式。我们将会通过HtmlWebpackPlugin插件，来生成一个HTML页面或者一个已经存在的。我们将使用一个已存在的文件。

首先我们安装插件：

npm install --save-dev html-webpack-plugin@2

然后我们需要添加它到我们的配置

// webpack.config.js

var path = require('path')

var webpack = require('webpack')

var HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {

entry: ['./src/index'],

output: {

path: path.join(\_\_dirname, 'dist'),

filename: 'bundle.js'

},

plugins: [

new webpack.optimize.UglifyJsPlugin({

compressor: {

warnings: false,

},

}),

new webpack.optimize.OccurenceOrderPlugin(),

new HtmlWebpackPlugin({

template: './src/index.html'

})

],

module: {

loaders: [{

test: /**\.**css$/,

loaders: ['style', 'css']

}]

}

}

这一次，当我们运行webpack，因为我们指定了一个HtmlWebpackPlugin 的插件和./src/index.html模版，他将生成一个index.html的文件在我们的dist文件夹有./src/index.html的内容。

如果我使用index.html作为模版，并且他是空的。实际上现在就是一个很好填充它的机会。

<html>

<head>

<title>Webpack Tutorial</title>

</head>

<body>

<h1>Very Website</h1>

<section id="color"></section>

<button id="button">Such Button</button>

</body>

</html>

请注意，我们没有把bundle.js作为一个<script>标签放进我们的html当中。这个插件实际上会自动帮我们做这个。如果你插入了script标签，你将会导致你相同的代码加载两次。

接下来让我们在style.css中添加一些基本的样式：“

h1 {

color: rgb(114, 191, 190);

text-align: center;

}

#color {

width: 300px;

height: 300px;

margin: 0 auto;

}

button {

cursor: pointer;

display: block;

width: 100px;

outline: 0;

border: 0;

margin: 20px auto;

}

## 开发服务器

例6

现在我们要在浏览器中看到我们的网站，他需要一个网络服务器来服务我们的代码。方便的是webpack自带的 webpack-dev-server ，你需要在本地和全局都安装。

npm install -g webpack-dev-server

npm install --save-dev webpack-dev-server

开发服务器是一个非常有用的资源让你的王章看起来想在浏览器中，以便更快的开发。默认情况下，你可以访问 <http://localhost:8080> 来访问。不幸的是，如果箱子的热加载功能不工作，则需要一些更多的配置。

这是一个很好的实例来分割我们的webpack配置在一个用于开发一个用于生产。因为我们我们这是一个简单的教程，这不会是一个巨大的差异，但是这是一个介绍webpack差异配置性的介绍。我们将命名他们为webpack.config.dev.js和webpack.config.prod.js。

// webpack.config.dev.js

var path = require('path')

var webpack = require('webpack')

var HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {

devtool: 'cheap-eval-source-map',

entry: [

'webpack-dev-server/client?http://localhost:8080',

'webpack/hot/dev-server',

'./src/index'

],

output: {

path: path.join(\_\_dirname, 'dist'),

filename: 'bundle.js'

},

plugins: [

new webpack.HotModuleReplacementPlugin(),

new HtmlWebpackPlugin({

template: './src/index.html'

})

],

module: {

loaders: [{

test: /**\.**css$/,

loaders: ['style', 'css']

}]

},

devServer: {

contentBase: './dist',

hot: true

}

}

### 变化

开发配置省略了优化（optimize）配置，因为在我们不断地重构改造我们的项目时，这些优化配置都是不必要的开销。所以开发环境的配置没有webpack.optimize插件。

开发配置是开发服务器所必要的配置，你可以阅读更多关于这里：<https://webpack.github.io/docs/webpack-dev-server.html>

### 总结

entry：两个2新的连接服务器的入口配置到浏览器允许HMR。

devServer

contentBase：服务器文件位置

hot：是否使用HMR

生产环境的配置变化不大：

// webpack.config.prod.js

var path = require('path')

var webpack = require('webpack')

var HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {

devtool: 'source-map',

entry: ['./src/index'],

output: {

path: path.join(\_\_dirname, 'dist'),

filename: 'bundle.js'

},

plugins: [

new webpack.optimize.UglifyJsPlugin({

compressor: {

warnings: false,

},

}),

new webpack.optimize.OccurenceOrderPlugin(),

new HtmlWebpackPlugin({

template: './src/index.html'

})

],

module: {

loaders: [{

test: /**\.**css$/,

loaders: ['style', 'css']

}]

}

}

我此次增加了一个全新的属性包括开发配置和生产配置：

devtool – 这是一个调试工具。基本上，当你出现一个错误，他将帮助你看清楚关于错误的信息，比如chrome开发者控制台。不同的是source-map比cheap-eval-source-map更难从文档中搜集。我肯以肯定的说source-map对生产意味着是一个很大的开销。而cheap-eval-source-map相较是更少的开销，意味着只能开发中使用。

运行dev开发服务器必须要运行

webpack-dev-server --config webpack.config.dev.js

构建生产代码我们需要运行

webpack --config webpack.config.prod.js

让我们使用得更简单一点，我们现在可以使用一个简单的任务让package.json运行，我们不需要打任何命令。

我们将脚本属性添加到配置。

// package.json

{

//...

"scripts": {

"build": "webpack --config webpack.config.prod.js",

"dev" : "webpack-dev-server --config webpack.config.dev.js"

}

//...

}

我们可以运行这些命令：

npm run build

npm run dev

现在我们可以通过运行npm run dev来看到你美丽的网站，导航到<http://localhost:8080>。

注意：当我测试这部门时我发现服务器不会实时重载我修的index.html文件。解决这个问题的办法是 html-reload。这是一些有用的信息，包括一些webpack的配置选项，我推荐你看，我把它们分开了，因为我觉得它延长了教程可能导致很琐碎。

<https://github.com/AriaFallah/WebpackTutorial/tree/master/part1/html-reload>

## 开始编码

例7：

人们之所以对webpack看起来很紧张是因为我们需要懂得这一些过程才能让我们开始写javascript代码。好了，现在我们已经完成了这个过程，到了教程的高潮部分。

万一你还没有：运行 npm run dev，并导航到 <http://localhost:8080>。构建一个实时重载服务器不能显示。每一次都可以节省你修改你项目任何部分时的时间，浏览器将会重新加载来显示你的改变。

我们也需要一个 新的npm package来说明你可以使用他们在你的前端当中。

npm install --save pleasejs

PleaseJs是一个随机颜色生成器，我们要安装我们的按钮来改变div的颜色

// index.js

// Accept hot module reloading

if (module.hot) {

module.hot.accept()

}

require('./styles.css') // The page is now styled

var Please = require('pleasejs')

var div = document.getElementById('color')

var button = document.getElementById('button')

function changeColor() {

div.style.backgroundColor = Please.make\_color()

}

button.addEventListener('click', changeColor)

有趣的是，为了使实时加载模块工作，我们需要包含这些代码：

if (module.hot) {

module.hot.accept()

}

在一个模块或者这个模块的父级中。

我们已经完成了！

注意：你可能注意到了你的css被应用事后的延迟，你可能会讨厌你的css被包含在js文件当中。我留出一个例子，css-extract（提取），描述如何把你的css放进一个不同的文件中。

<https://github.com/AriaFallah/WebpackTutorial/tree/master/part1/css-extract>

# 结论

我希望能帮助到你们。

webpack首先是一个模块打包器。它是一个高度模块化和易用的工具，在事实上，他不限用于ES6和React。

now：

第二部分将使用webpack通过Babel去转译ES6到ES5的语法。

第三部分将使用webpack和React+Babel，因为它们都是最常见的用例。

# 最后的想法

祝贺你！你做了一个按钮去改变div的颜色！webpack是不是很强大！

是的。但是，如果你做的是一个按钮，去改变div的颜色，他是不可能匹配写配置的价值的。如果你这样做，你可能会…很累。