
Table of Contents

Introduction	1.1
简介	1.2
webpack与其他工具的比较	1.3
使用webpack进行开发	1.4
开始	1.4.1
分解配置文件	1.4.2
浏览器自动刷新	1.4.3
自动刷新CSS	1.4.4
使用Sourcemaps	1.4.5
使用webpack进行构建	1.5
最小化构建	1.5.1
设置环境变量	1.5.2
分解Bundle	1.5.3
在文件名上添加hash	1.5.4
清理构建	1.5.5
分解CSS	1.5.6
分析构建统计数据	1.5.7
托管于Github pages	1.5.8
加载资源	1.6
支持格式	1.6.1
定义loader	1.6.2
加载样式	1.6.3
加载图片	1.6.4
加载字体	1.6.5
进阶	1.7
理解块	1.7.1
在webpack中使用linting	1.7.2
编写包	1.7.3
编写loader	1.7.4
配置React	1.7.5

Webpack - 从初学者到大师

by Juho Vepsäläinen, Tobias Koppers and Jesús Rodríguez Rodríguez from SurviveJS

webpack通过解决web开发中的资源打包问题，极大地提高了web开发的效率。它的作用是将各种资源如JavaScript, CSS, 和HTML打包成一个浏览器可以接受的形式。

如果你熟练地运用webpack, 它可以帮助你将在web开发中大量的痛点消除，这本书就是为了帮助你熟练地掌握webpack而设计的。

你可以浏览[gitbook地址](#)来阅读。

简介

Webpack 通过解决web开发中的打包问题，极大地提高了web开发的效率。它可以将各种资源如JavaScript, CSS, 和HTML打包成一个浏览器可以接受的形式。如果你熟练地运用webpack, 它可以将你在web开发中遇到的大量的痛点消除。由于webpack是一个配置驱动的工具，所以它并不非常容易上手，这本书的目的就是帮助你熟练地掌握webpack。

什么是webpack？

浏览器是用来解析HTML,JavaScript和CSS文件的。一个最简单的开发方式是直接编写许多文件，然后把它们用浏览器来直接解析，这种资源加载方式显得非常笨重，然而这个场景在我们开发web应用的时候非常常见。

一个解决这个问题的原始方法是将这些文件打包成一个文件，很显然这不是一个很好的解决方案，你有时需要将它们分解开来，甚至需要动态的加载你所需要的依赖，并且这个问题的复杂性随着项目的开发而逐渐提高。

Webpack就是来解决这个问题的. 它可以允许你使用不同的工作流和不同的构建工具来得到同样的结果。事实上，这些已经足够了，一些构建工具，如Grunt和Gulp，也可以帮助你做这些事情，但是需要你编写许多配置。

webpack改变了什么？

Webpack采用了另外的一条路径，它允许你将你的项目看作是一个依赖图，你的项目需要包含一个 index.js 文件，它使用标准的 import 语句描述了项目所需要的依赖，而且你可以描述你的样式文件和其他资源文件通过同样的方式。

Webpack为你做了所有预处理的事情，然后把符合你之前配置的bundle交给你。这些声明的配置是非常强大的，但是很难学习。然而，如果你理解了webpack是如何工作的，它会成为你不可或缺的工具。

你将如何学习？

这本书作为[webpack 官方文档](#)的补充，由于官方文档包含了非常多的内容，它可能不太适合初学者，本书可以简化webpack的学习曲线同时给有经验的开发者一些灵感。

你将会学习到如何根据不同的环境（如开发和生产）来配置webpack，你还会学习到webpack的一些进阶的用法，它将会让你十分受益。

这本书是如何组织的？

最开始的两章为你介绍webpack和它的一些基本的内容，你将使用基本的配置来适应项目的不同环境，前面的章节都是任务导向性的，你可以收藏它，防止你以后忘记了webpack的一些具体的用法。

书的后面一部分还讨论了一些进阶的话题，你将了解到如何加载具体类型的文件，你也将会加深对webpack打包机制的理解，利用它创建一些包，编写一些loader等。

谁适合阅读本书？

我希望你有一些基本的JavaScript和Node.js的知识，并且使用过npm。如果你还了解一些webpack，那就更好了，通过学习，你会加深对它们的理解。

如果你已经了解了webpack，这本书还是有一些知识可以提供给你，浏览此书看看有没有适合你的部分，我最大努力的在本书中提供了使用webpack的会遇到的棘手问题。

如果你学习地很困难，尝试着围绕着本书向社区寻求帮助，我们在那里等着你，任何关于本书的回应将使得书的内容变得更好。

我将如何开始？

如果你不太了解webpack，前两章你要仔细地学习，你可以略过其他的内容只选择你所感兴趣的。如果你已经了解了webpack，挑选你认为有价值的内容。

书的版本

本书收到了大量的更新，很难在这里罗列出版本，我维护了更新内容在本书的[博客](#)中，他可以让你了解到每个版本所更新的内容，同样的，使用Github来了解更新，也是一个不错地方法，我推荐使用Github，你可以这样使用，

```
https://github.com/survivejs/webpack/compare/v1.2.0...v1.3.1
```

这个页面可以让你方便地查看到指定版本之间的提交，它虽然排除了私有的章节，但也足够了。

目前的版本是 1.3.1

本书还在创作中，欢迎提交反馈和讨论，通过反馈我可以将书创作地更好，你甚至可以将本书修改成适合你的版本，这都是允许的。

书的一部分收入将会用来支持本书和webpack本身。

获得帮助

没有任何地书是完美的，你可以提交issue来对书的内容提问，下面是提issue的一些途径，

- 通过[github](#)的issue联系我。
- 通过[Gitter](#)来聊天。
- 在Twitter关注官方账号[@survivejs](#)来获取官方的新闻，也可以通过[@bebraw](#)直接联系我。
- 发送邮件到info@survivejs.com
- 通过[SurviveJS AmA](#)来问我任何关于webpack和react的问题

如果你在stack Overflow上提问，将这些问题加上 **survivejs** 的标签，我可以很方便地注意到它们，你可以在Twitter使用**#survivejs**话题来达到同样的效果。

声明

我声明 SurviveJS 的信息通过下面的渠道发布：

- [邮件列表](#)
- [Twitter](#)
- [Blog RSS](#)

感谢

感谢 Christian Alfoni 帮助我编写了本书的第一版，他鼓舞了我整个SurviveJS的项目，你看到的版本已经修改了很多了。

这本书也离不开耐心编写和整理反馈的我的编辑Jesús Rodríguez Rodríguez，感谢他。

这本书同样也离不开SurviveJS - Webpack and React这个项目的支持，我对曾经贡献过这个项目的所有人表示感谢，你可以通过本书获得更多的贡献。

感谢 Mike "Pomax" Kamermans, Cesar Andreu, Dan Palmer, Viktor Jančík, Tom Byrer, Christian Hettlage, David A. Lee, Alexandar Castaneda, Marcel Olszewski, Steve Schwartz, Chris Sanders和其他直接贡献本书的人。

webpack与其他工具的比较

至此，你了解到了webpack功能的强大，在此之前的开发中，我们将一些脚本机械地连接起来就感觉已经足够了，然而，时代变化了，分发你的JavaScript越来越复杂。

这个问题随着单页应用（SPAs）的发展变得更加突出，这些单页应用通常会依赖许多库，我们现在有着许多如何加载这些库的解决方案，你可以一次性地全部加载完成，你也可以按需加载，webpack支持了许多这些灵活的解决方案。

Node.js和npm的发展，给了我们更多的想象空间，在npm之前，我们很难去处理依赖，现在，npm成为前端开发流行地解决依赖的方案，它使得管理依赖更加容易了。

构建工具和打包工具

在实际工作中，构建工具有很多，[Make](#)就是其中的代表，它至今仍然具有无限地活力。为了将构建变得更容易，一些任务执行器（*task runners*），如[Grunt](#)和[Gulp](#)出现了，并且它们通过在npm上得插件而变得更加强大。

构建工具属于一个更高层次地工具，他们允许你做一些跨平台的事情。你最初使用构建工具遇到的问题就是分拣出各种资源并且将他们打包（*bundle*），这就是我们需要打包工具（*bundlers*）的原因，如[Browserify](#),[Brunch](#),[webpack](#)等都是比较流行的打包工具。

让我们继续，[JSPM](#)将对浏览器进行直接地包管理，他依赖于一个动态地模块加载器[System.js](#)。[webpack2](#) 将会支持System.js的语法。

与Browserify,Brunch和webpack不一样的是，JSPM跳过了开发时候的打包环节，你可以生成一个生产的包来使用他，你可以通过Glen Maddern的[视频](#)来详细了解JSPM。

Make

你可以说Make已经成为历史了，不错，这个诞生于1977年的工具，经历了各种各样的维护。Make允许你针对不同的目的编写出不同的任务，例如，你可能需要针对不同的环境如构建生产包，压缩文件，执行测试编写不同地任务，这个思想在其它的工具中也会体验出来。

虽然Make经常应用于C的项目中，但是这并不妨碍它用于其它语言，James Coglan在《[如何使用Make在JavaScript](#)》中详细介绍了如何在JavaScript项目中使用Make，如下面地代码

```
PATH := node_modules/.bin:$(PATH)
SHELL := /bin/bash

source_files := $(wildcard lib/*.coffee)
build_files := $(source_files:%.coffee=build/%.js)
app_bundle := build/app.js
spec_coffee := $(wildcard spec/*.coffee)
spec_js := $(spec_coffee:%.coffee=build/%.js)

libraries := vendor/jquery.js

.PHONY: all clean test

all: $(app_bundle)

build/%.js: %.coffee
    coffee -co $(dir $@) $<

$(app_bundle): $(libraries) $(build_files)
    uglifyjs -cmo $@ $^

test: $(app_bundle) $(spec_js)
    phantomjs phantom.js

clean:
    rm -rf build
```

通过使用Make，你可以将你的任务通过使用make特定的语法和终端命令来进行模块化，然后很轻松地用webpack进行连接。

Grunt



GRUNT
The JavaScript Task Runner

Grunt是Gulp之前的主流，这得益于他的插件的设计思想，尽管插件的本身有些复杂。当这些插件的配置越来越复杂的时候，会变得特别难于理解。

这里是一个来自于[Grunt官方文档](#)的例子，在这个配置中，我们定义了一个linting和一个watcher的任务，通过这个任务，我们可以在编写代码的时候实时地得到错误的警告。

Gruntfile.js

```
module.exports = function(grunt) {
  grunt.initConfig({
    lint: {
      files: ['Gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
      options: {
        globals: {
          jQuery: true
        }
      }
    },
    watch: {
      files: ['<%= lint.files %>'],
      tasks: ['lint']
    }
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-watch');

  grunt.registerTask('default', ['lint']);
};
```

实际上，你可以编写需要许多小任务来完成各种需求，Grunt的一个强大之处就是它隐藏了许多任务之间的连接操作。然而，时间长了，将会出现问题，你会对于任务地执行细节存在困惑。

[grunt-webpack](#)允许你在Grunt环境中使用webpack,你可以将关注点转移到使用webpack上。

Gulp



Gulp采用了不同的方式，它并不是通过配置每一个插件，而是直接地操作代码，Gulp依赖于底层的piping，如果你对于Unix非常熟悉，这里与Unix中的非常相似，它包括了加载器，过滤器和接收器三个组成部分。

加载器的作用是匹配文件，过滤器来操作这些文件（如生成JavaScript），最终，将这些文件传递给接收器（如一个文件夹）。下面是一个简单的*Gulpfile*示例，

Gulpfile.js

```
var paths = {
  scripts: ['client/js/**/*.coffee', '!client/external/**/*.coffee']
};

// Not all tasks need to use streams
// A gulpfile is just another node program
// and you can use all packages available on npm
gulp.task('clean', function(cb) {
  // You can use multiple globbing patterns as
  // you would with `gulp.src`
  del(['build'], cb);
});

gulp.task('scripts', ['clean'], function() {
  // Minify and copy all JavaScript (except vendor scripts)
  // with sourcemaps all the way down
  return gulp.src(paths.scripts)
    .pipe(sourcemaps.init())
    .pipe(coffee())
    .pipe(uglify())
    .pipe(concat('all.min.js'))
    .pipe(sourcemaps.write())
    .pipe(gulp.dest('build/js'));
});

// Rerun the task when a file changes
gulp.task('watch', function() {
  gulp.watch(paths.scripts, ['scripts']);
});

// The default task (called when you run `gulp` from CLI)
gulp.task('default', ['watch', 'scripts']);
```

通过上面的配置代码，你可以在遇到困难的时候很容易地修改，你还可以包装已有的Node.js包成为Gulp插件。对比Grunt，你非常清晰这里面发生了什么。你可以对一些不具有共性的任务编写模块，这些也是新思想经常出现的地方。

[gulp-webpack](#)允许你在Gulp环境下使用webpack

[fly](#)是一个与Gulp相似的工具，它依赖于ES6的generator

Browserify




处理JavaScript模块是一个非常麻烦的事情，这门语言直到ES6的时候才诞生了模块化。因此，在这之前出现了许多解决方案，如AMD等。

实际上，我们通常使用CommonJS和Node.js的形式，这个优势很明显就是你可以发送到npm上，避免重复地制造轮子。

Browserify是一个解决模块化问题的方案，他可以将CommonJS的模块打包起来，你可以将它与Gulp一起使用。这里有一些小的转换工具作为进阶的用法，如watchify提高了一个文件watcher允许你在开发的过程中打包。虽然browserify还需要一些完善但无疑它目前是一个比较好的工具。

Browserify的生态系统诞生了许多小模块，在这个方面，Browserify得益于Unix的设计思想，Browserify比webpack更易于接受，也是Webpack一个好的替代品。

Brunch



See your build tool in nightmares?
Try Brunch!

It gets out of your way and lets you focus on what matters most to you — solving real problems, not messing around with the gluing.

[Get Started](#)

对比Gulp, Brunch在更高的抽象层进行操作，他像webpack一样，声明一个应用。下面是Brunch官网的一个示例

```
module.exports = {
  files: {
    javascripts: {
      joinTo: {
        'vendor.js': /^(?!app)/,
        'app.js': /^app/
      }
    },
    stylesheets: {
      joinTo: 'app.css'
    }
  },
  plugins: {
    babel: {
      presets: ['es2015', 'react']
    },
    postcss: {
      processors: [require('autoprefixer')]
    }
  }
};
```

Brunch通过使用一些命令，如 `brunch new`，`brunch watch --server` 和 `brunch build --production` 来进行任务的操作，这里面包含了许多内容而且可以使用插件扩展。

这里有一个Brunch热替换的方案。

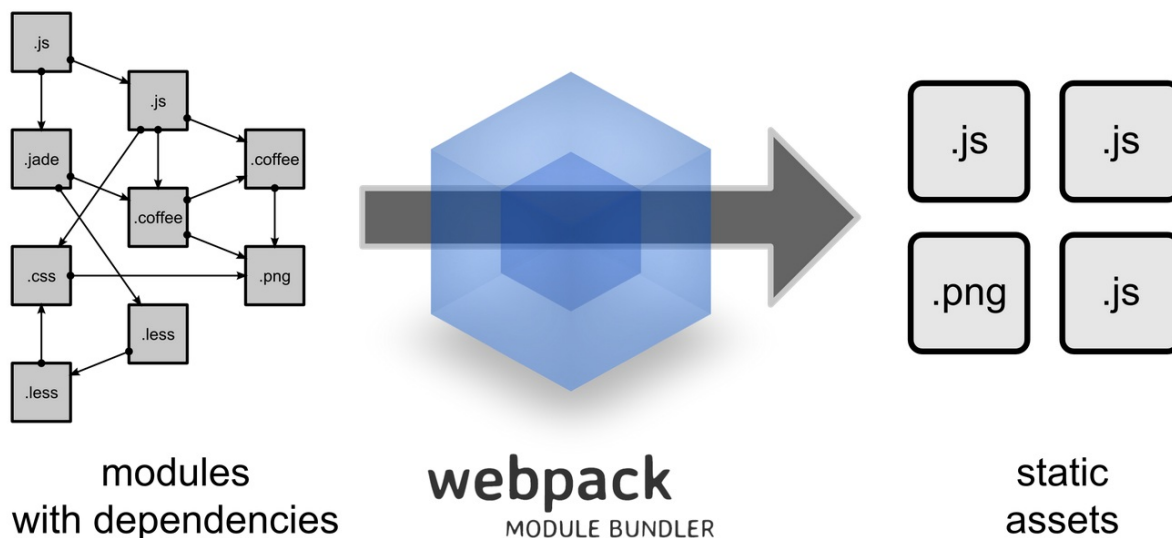
JSPM



使用JSPM将不同于之前的工具，他通过在项目上安装一个很小的命令行实现，并且，他支持SystemJS 插件，允许你在你的项目中加载各种格式的文件。

考虑到JSPM仍然是一个年轻的项目，还有一些缺陷，如果你想体验可以试用一下，正如你所知道的，工具的目的是改变前端开发的环境，而且JSPM是其中强有力的竞争者。

Webpack



你可以说webpack是一个比browserify更加强大的工具。Browserify包含了众多小的工具，而webpack包含了一个核心功能，而且加了许多功能性的接口，这个核心的功能可以通过特定的loaders和plugins进行扩展。

webpack可以通过你项目中的 `require` 语句来生成你所需要的bundles。这个loader的机制可以使得CSS文件正常运行，并且支持 `@import` 语法。还有一些具体任务特定的插件，如压缩代码，本地化，热替换等等。

以一段代码作为示例，`require('style!css!./main.css')` 加载了`main.css`中的文件，并且通过CSS和style loaders从右向左进行处理。通过这个声明，将源文件提供给了webpack，更推荐使用webpack的配置来完成这一功能，下面是一个从[webpack官方指南](#)中得到的一个例子。

```
var webpack = require('webpack');

module.exports = {
  entry: './entry.js',
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css']
      }
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin()
  ]
};
```

这个配置是使用JavaScript编写的，它的可扩展性非常的好。

这个配置的模型可能会使你感到一点迷惑，你可能会很难理解这里发生了什么，这在一些复杂的场景下显得更加突出，这些原因也是本书存在的价值。

为什么是Webpack？

我们为什么在Gulp和Grunt之上使用Webpack？webpack的强大之处在与处理复杂的打包问题，然而这个其他的工具也可能涉及。我推荐webpack是因为它支持热替换(HMR)，它还可以通过[babel-plugin-react-transform](#)这个插件来使用React，我接下来会教你如何设置。

热替换

你可以已经对这些工具如[LiveReload](#)或者[Browsersync](#)非常熟悉了，这些工具在你更改代码的时候可以自动的刷新页面。HMR把这个特性变得更加强大。在react的环境下，webpack允许程序维护自己的state,这听起来很简单，但是使得开发过程产生了巨大的变化。

需要注意的是，HMR在Browserify中通过[livereload](#)进行实现，所以它不属于webpack的一个独有的特性。

分解bundle

除了HMR的特性，webpack的分解bundle的功能是十分强大的，他允许你通过多种的方式来分解bundles.你甚至可以在程序的执行过程中动态地加载他们。这种懒加载尤其在大的项目中十分有用，你可以加载你所需要的依赖在任何需要的时候。

资源哈希

通过webpack，你非常容易的将hash注入到bundle名称中（如app.d587bbd6e38337f5accd.js），当源码发生改变后，你可以验证浏览器是否发生了对应的改变。在理想状况下，分解bundle的特性允许浏览器只刷新一小部分。

Loaders和Plugins

这些小的功能叠加起来，不出意外的话，你可以完成所有你想要的功能。而且如果你遗忘了一些东西，你可以通过loaders和plugins完成这些功能。

webpack有一个很复杂的学习曲线，即使是这样，这仍然是一个值得学习的工具，长时间学习它，你可以在今后的开发中节省大量的时间。如果你想了解更多与其他工具的对比，看一下[官方的对比](#)。

结论

我希望通过本章帮助你了解到为什么webpack是一个值得学习的工具，他解决了许多web开发的通用问题，如果你很熟悉他，将会节省很多时间。

在接下来的章节中，我们将会详细学习webpack，你将会学习如何完成一个基础的开发和一个基本的配置，再往后的章节你将会探索一些高级的用法。

你应该讲webpack配合其他工具一起使用，因为他只是解决了打包的问题。当然，这些事情你不需要关心，使用package.json，scripts 和webpack吧，接下来我们将进行详细的学习。

使用webpack进行开发

webpack是通过一个配置文件来驱动的，这个文件通常被命名为`webpack.config.js`，它是所有操作的核心。这个配置文件定义了你的项目的输入和输出，更重要的是，他描述了你所进行的所有转换，这些转换使用了`loaders`和`plugins`来实现。

这本节中，我将会带着你基于webpack做一个最基本的开发，你将会学习到很多基础的内容。

开始

确保你正在使用最新版的[Node.js](#)，我推荐使用最新的LTS（长时间支持）版本，并且你还要确保 `node` 和 `npm` 可以正常在你的终端中使用。

上面完整的配置可以参照[GitHub](#)，如果你不了解，可以作为参照。

创建你的程序

在开始之前，你应该为你的程序建立一个文件夹，并且创建一个`package.json`文件，`npm`通过这个文件来管理依赖，下面是一个基本的命令。

```
mkdir webpack-demo
cd webpack-demo
npm init -y # -y 生成了 *package.json*
```

你可以手动编写`package.json`文件来获得更多的灵活性，虽然我们也可以通过`npm`来进行大部分间接地配置该文件，但是我们还是需要了解该文件的具体用法，你可以通过查看[package.json选项](#)来获得更多信息。

安装Webpack

虽然`webpack`可以全局安装(`npm i webpack -g`)，我仍然推荐将其作为你项目的一个依赖，这个可以避免一些版本上的错误。

这个项目应该与持续集成（**CI**）更好地结合起来，一个持续集成系统可以本地安装依赖，编译并且使用，把最终结果推送到服务器上。

安装一个`Webpack`到你的项目，执行

```
npm i webpack --save-dev # 或者使用 -D 作为缩写。
```

你应该看到`webpack`在你的`package.json`中的 `devDependencies` 字段中，这时候，`webpack`被安装到了本地的`node_modules`目录下，并且`npm`为其生成了一个可以执行的入口文件。

执行webpack

你可以通过使用 `npm bin` 来查看可执行文件的路径，大多数情况下，他应该存在于 `./node_modules/.bin` 目录下，。尝试在这个路径执行 `webpack` 通过 `node_modules/.bin/webpack` 。

当你执行过上面片段之后，你应该看到一个版本，一个接口导航的地址和一个很长的参数说明。我们不需要使用其中的大多数，但是我们可以知道这个工具已经被正确执行了。

```
webpack-demo $ node_modules/.bin/webpack
webpack 1.13.0
Usage: https://webpack.github.io/docs/cli.html

Options:
  --help, -h, -?
  --config
  --context
  --entry
  ...
  --display-cached-assets
  --display-reasons, --verbose, -v

Output filename not configured.
```

文件夹结构

一个程序不能单单的只有 `package.json`，我们需要丰富它的内容。让我们一开始先实现一个小的 `web` 站点，他可以加载我们通过 `webpack` 打包的 `JavaScript` 代码，当我们完成这个功能的时候，文件目录看起来是这个样子：

```
- app/
  - index.js
  - component.js
- build/
- package.json
- webpack.config.js
```

我们可以将 `app/` 下面的文件打包放在 `build/` 下面，为了能达到这个目的，我们需要配置对应的 `webpack.config.js` 文件。

配置资源

我想你并不会厌倦 `Hello world` 的示例，我们可以像这样设置一个模块

app/component.js

```
module.exports = function () {  
  var element = document.createElement('h1');  
  
  element.innerHTML = 'Hello world';  
  
  return element;  
};
```

下一步，我们需要对我们的应用配置一个入口，他通过 `require`，包含了我们的刚才的模块然后在DOM上渲染。

app/index.js

```
var component = require('./component');  
  
document.body.appendChild(component());
```

配置webpack

我们需要告诉webpack如何操作我们刚才的资源，我们创建了一个`webpack.config.js`文件，webpack和它的开发服务器（`webpack-dev-server`）可以检测到这个文件。

为了让代码可以更好的维护，我们使用[html-webpack-plugin](#)来为我们的程序生成`index.html`，在项目中安装它：

```
npm i html-webpack-plugin --save-dev
```

下面是完整的配置来在build目录下生成bundle

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const PATHS = {
  app: path.join(__dirname, 'app'),
  build: path.join(__dirname, 'build')
};

module.exports = {
  // Entry accepts a path or an object of entries.
  // We'll be using the latter form given it's
  // convenient with more complex configurations.
  entry: {
    app: PATHS.app
  },
  output: {
    path: PATHS.build,
    filename: '[name].js'
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: 'Webpack demo'
    })
  ]
};
```

这个 `entry` 路径可以是一个相对的地址，这个时候可以使用 `context` 字段来配置查找的上下文。大部分的配置字段最好都用一个绝对的路径，我建议在任何时候都使用绝对路径来避免冲突。

如果你执行 `node_modules/.bin/webpack`，你将会看到下面的输出：

```
Hash: 2a7a7bccea1741de9447
Version: webpack 1.13.0
Time: 813ms
   Asset      Size  Chunks             Chunk Names
   app.js    1.69 kB       0  [emitted]  app
index.html  157 bytes             [emitted]
    [0] ./app/index.js 80 bytes {0} [built]
    [1] ./app/component.js 136 bytes {0} [built]
Child html-webpack-plugin for "index.html":
   + 3 hidden modules
```

这里告诉了我们很多，下面将进行注释：

- `Hash: 2a7a7bccea1741de9447` 本次build的hash值，你可以使用这个通过 `[hash]` 字段来验证资源。我们将详细的在 *Adding Hashes to Filenames* 章节中详细讨论
- `Version: webpack 1.13.0` webpack的版本

- `Time: 813ms` 打包所需要的时间
- `app.js 1.69 kB 0 [emitted] app` 打包后资源的名称，大小，打包后关联的块的id，状态信息和块名称。
- `[0] ./app/index.js 80 bytes {0} [built]` 被打包资源的id，名称，大小，被打到块的id以及状态
- `Child html-webpack-plugin for "index.html":` 这个是插件的输出，在这个例子中是`html-webpack-plugin`的输出。
- `+ 3 hidden modules` 告诉你webpack生成的东西，在 `node_modules` 目录下，你可以使用 `webpack --display-modules` 展现详细的信息，也可以参照[Stack Overflow](#)来查看更多的解释。

观察 `build/` 下面生成的内容，如果你观察仔细，你会发现这些内容就是webpack打包出来的内容，你可以通过打开 `build/index.html` 文件来运行，在OS X的环境下，通过 `open ./build/index.html` 来打开。

添加一个打包的快捷方式

通过执行 `node_modules/.bin/webpack` 的命令来执行webpack看起来有点多余，我们可以解决这个问题。我们可以通过npm和`package.json`作为一个task runner, 让我们来做一些如下的配置。

package.json

```
...  
"scripts": {  
  "build": "webpack"  
},  
...
```

你可以执行这些脚本通过`npm run`。例如，在这个例子中，使用`npm run build`，你将会得到与刚才同样的结果。这个生效是因为npm把`node_modules/.bin`这个文件路径加载了起来，所以我们才可以通过 `"build": "webpack"` 来达到执行 `"node_modules/.bin/webpack"` 同样的效果。

结论

我们已经完成了一个非常基本的webpack配置，尽管他看起来并不是很好，每次我们修改，我们都需要手动执行 `npm build` ,然后刷新浏览器，体验十分不好。

这个时候我们就需要webpack的进阶用法了，为了更好的学习进阶的用法，我们将会在下一节中先讨论如何分解webpack的配置。

分解配置文件

最开始的时候，webpack的配置文件可以包含在一个单独的文件中，但是随着程序复杂性的增加，你可能需要分解这个配置才可以更好地对它进行管理。根据已有经验，你很有必要将他们根据运行环境来进行分类，然后你就很好的针对不同的运行环境来构建不同的包。如何的分类并没有绝对正确的方法，但是一般有下面几个方法：

- 在多个文件中维护配置，通过Webpack的 `--config` 选项来连接，共同的配置通过模块间的imports。你可以在[webpack/react-starter](#)这个项目中看到具体的使用。
- 将配置放入一个库中，示例[HenrikJoreteg/hjs-webpack](#)
- 维护一个文件的配置，然后设立分支。通过npm的脚本（如 `npm run test`）来决定使用哪个配置。npm在他的环境变量中设置了这些信息，你可以匹配他们来做你想做的事情。

我推荐使用最后的方案，我们可以很清楚的知道这里发生了什么，我开发了一个小的工具名为[webpack-merge](#)，我将会展示如何使用它。

设置webpack-merge

首先执行

```
npm i webpack-merge --save-dev
```

在项目中添加了*webpack-merge*

webpack.config.js


```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const merge = require('webpack-merge');

const PATHS = {
  app: path.join(__dirname, 'app'),
  build: path.join(__dirname, 'build')
};

// module.exports = {

const common = {

  // Entry accepts a path or an object of entries.
  // We'll be using the latter form given it's
  // convenient with more complex configurations.
  entry: {
    app: PATHS.app
  },
  output: {
    path: PATHS.build,
    filename: '[name].js'
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: 'Webpack demo'
    })
  ]
};

var config;

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(common, {});
    break;
  default:
    config = merge(common, {});
}

module.exports = config;
```

当我们做了这个改变之后，虽然说实际的功能仍然与之前一样，但是我们有了更多的扩展空间。我们可以将热替换接入进来使得浏览器自动刷新使得开发模式更好地提高了我们的效率。

连接 *webpack-validator*

为了使得我们更容易地开发配置，我们需要引入一个`webpack-validator`在我们的项目中，它可以提醒我们的配置与标准配置规则不合适的地方，这对我们学习和使用`webpack`起到了重要的作用。

首先安装：

```
npm i webpack-validator --save-dev
```

我们之后就可以将其引入到项目中了。

`webpack.config.js`

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const merge = require('webpack-merge');
const validate = require('webpack-validator');
// ...
// module.exports = config;
// module.exports = validate(config);
```

如果你编写了错误的`webpack`配置，它将会给你提示和一下优雅的解决方法。

结论

尽管分解配置是一个非常微不足道的技术方案，但是它极大地扩展了我们的使用`webpack`的适用场景。每一种方案都有优点和缺点，我发现这个方案对于中小型的项目很有帮助，对于大型项目可能会有其他的解决方案。

现在我们已经拥有了足够的扩展空间，在下一章节中，我将会告诉你如何来利用`webpack`完成浏览器的自动刷新。

浏览器自动刷新

一些像[LiveReload](#)和[Browsersync](#)的工具允许我们修改的应用的时候，浏览器自行刷新，他们甚至可以检测CSS变动时候的自动刷新。

首先，我们需要开启webpack的`watch`模式，你可以通过 `webpack --watch` 来开启。一旦开启了这个功能，它将会检测文件发生的变化，然后自行编译打包。一个我们下面使用一个叫 `webpack-dev-server` 的工具在更高的层次上完成了这个功能。

`webpack-dev-server`是一个运行在内存中的开发服务器。当你对应用做出改变的时候，浏览器会自行刷新。并且支持webpack的一个更高级的功能——热替换(HMR)，他使得浏览器不会整个页面的状态都进行刷新，这极大的提高了开发像React之类应用的效率。

开始使用 `webpack-dev-server`

首先执行

```
npm i webpack-dev-server --save-dev
```

通过上面的载入，你可以将命令放置在 `npm bin` 指定的目录中，在那里，你可以运行 `webpack-dev-server`，最容易的方法是运行 `webpack-dev-server --inline`。 `--inline` 其中了一个服务器的`inline`模式，可以重写结果的url地址，并且直接自行刷新。

在项目中 使用 `webpack-dev-server`

在项目中 使用 `webpack-dev-server`，我们可以遵循之前的思路，在 `package.json` 文件中加入新的 `scripts`。

package.json

```
...  
"scripts": {  
  "start": "webpack-dev-server",  
  "build": "webpack"  
},  
...
```

我们可以将 `--inline` 的部分写在webpack的配置里面，我建议你的 `npm scripts` 要保持足够的简洁，将复杂的地方都交给配置文件来完成，你可以把注意力直接集中到配置文件本身上来。

如果你现在执行`npm run start`或者`npm start`，你可以在你的终端下看到这些内容，

```
> webpack-dev-server

[webpack-validator] Config is valid.
http://localhost:8080/webpack-dev-server/
webpack result is served from /
content is served from ../webpack-demo
Hash: 2dca5a3850ce5d2de54c
Version: webpack 1.13.0
```

这意味着开发服务器已经其中，你可以在你的浏览器访问<http://localhost:8080>看到结果。

如果你修改了代码，你应该会在终端上看到输出，问题是这时候浏览器并不会自动刷新，它需要我们接下来的配置。

Hello world

如果你没有在浏览器上看到任何东西，尝试着改变端口，之前的端口可能被占用了，你可以使用 `netstat -na | grep 8080` 来查看8080端口是否被占用。这个命令可能会根据你的平台而有相应的改变。

配置热替换(HMR)

热替换的功能处于`webpack-dev-server`的上层，它通过一个接口来替换当前的模块。例如，`style-loader`可以不刷新页面更新CSS，通过HMR来替换CSS显得很容易，因为它不会包含任何的状态。

HMR也可以应用于JavaScript，但是考虑到JavaScript的状态，它比CSS要复杂。在配置`React`的章节中，我们将会讨论如何将它与`React`结合起来，并且你可以把这个思想利用在其他的其他地方。

你可以通过 `webpack-dev-server --inline --hot` 来使用这个功能，`--hot` 通过一个为特定目的设计的插件来激活HMR的模块，并且编写一个JavaScript文件接入的入口。

为HMR定义配置

为了让我们的配置可以更好的维护，我将会分离HMR的配置，我们将保持`webpack.config.js`简单并且可重用，我们可以将它发布至npm的平台上，这个时候就可以重用了。

我们将配置的部分放至`libs/parts.js`，

`libs/parts.js`

```
const webpack = require('webpack');

exports.devServer = function(options) {
  return {
    devServer: {
      // Enable history API fallback so HTML5 History API based
      // routing works. This is a good default that will come
      // in handy in more complicated setups.
      historyApiFallback: true,

      // Unlike the cli flag, this doesn't set
      // HotModuleReplacementPlugin!
      hot: true,
      inline: true,

      // Display only errors to reduce the amount of output.
      stats: 'errors-only',

      // Parse host and port from env to allow customization.
      //
      // If you use Vagrant or Cloud9, set
      // host: options.host || '0.0.0.0';
      //
      // 0.0.0.0 is available to all network devices
      // unlike default `localhost`.
      host: options.host, // Defaults to `localhost`
      port: options.port // Defaults to 8080
    },
    plugins: [
      // Enable multi-pass compilation for enhanced performance
      // in larger projects. Good default.
      new webpack.HotModuleReplacementPlugin({
        multiStep: true
      })
    ]
  };
}
```

我们可以将这段代码以后重用起来，然后在主配置里连接它。

`webpack.config.js`

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const merge = require('webpack-merge');
const validate = require('webpack-validator');

const parts = require('./libs/parts');
...
// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(common, {});
    break;
  default:
    // config = merge(common, {});
    config = merge(
      common,
      parts.devServer({
        // Customize host/port here if needed
        host: process.env.HOST,
        port: process.env.PORT
      })
    );
}

module.exports = validate(config);
```

执行 `npm start` 之后浏览 **localhost:8080**，试着去修改 `app/component.js`，它应该会自动刷新浏览器，你要知道刷新JavaScript是复杂的，而CSS是比较简单的，你可以在下一章节了解到刷新css。

在Windows, Ubuntu和Vagrant上使用HMR

你可能会在Windows, Ubuntu和Vagrant遇到一些问题，尝试着这样去解决，

libs/parts.js

```
const webpack = require('webpack');

exports.devServer = function(options) {
  return {

    watchOptions: {
      // Delay the rebuild after the first change
      aggregateTimeout: 300,
      // Poll using interval (in ms, accepts boolean too)
      poll: 1000
    },

    devServer: {
      ...
    },
    ...
  };
}
```

如果默认的配置不能工作的话，就使用上面这个，他根据文件系统，变得更加资源敏感。

从网络上连接开发服务器

我们很有必要根据我们的环境（如windows上是 `SET PORT=3000`，Unix上是 `export PORT=3000`）来定制主机和端口号，这个当你使用其他的设备来访问时显得很重要，默认的配置在大多数情况下已经足够了。

为了连接你的服务器，你需要知道你的机器上的ip，在Unix上使用 `ifconfig | grep inet`，在Windows上，使用 `ipconfig`。也可以通过一些如node-ip的npm包。特殊的在windows机器上你需要设置 `HOST` 来配置。

其他配置`webpack-dev-server`的方法

我们可以通过`webpack-dev-server`的配置在终端中使用。但是我发现在webpack的配置中来配置可以增加可读行性，而且便于发现问题。

另外一种选择是，我们可以使用一个Express的服务器，并且将`webpack-dev-server`充当它的一个中间件。还可以通过Node.js API来配置。这是一个好的方案如果你想更灵活的来控制它。

`dotenv`允许你通过`.env`文件来定义你的环境变量，这个可以给你的开发带来一定的方便。

CLI与Node.js API有一定的区别，这就是为什么有一些人喜欢完全的使用Node.js的API的原因。

结论

在本章节中，我们讨论了如何设置`webpack`来自动刷新你的浏览器，你可以把这个特性运用在CSS上，我们将会在下一节来讨论它。

自动刷新CSS

在本节中我们将会讨论如何让我们项目中的CSS自动刷新，很容易想到的是，webpack在这个场景中不需要强制性的全部刷新，并且，我们可以做一些更智能的事情。

加载CSS

在开始之前，我们需要安装两个loader

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

我们现在拥有了需要的loaders，接下来，我们需要它它们与webpack连接起来。

libs/part.js

```
...
exports.setupCSS = function(paths) {
  return {
    module: {
      loaders: [
        {
          test: /\.css$/,
          loaders: ['style', 'css'],
          include: paths
        }
      ]
    }
  };
}
```

我们需要将其与我们的主配置文件连接起来

webpack.config.js

```
...
switch(process.env.npm_lifecycle_event) {
  case 'build':
    // config = merge(common, {});
    config = merge(
      common,
      parts.setupCSS(PATHS.app)
    );
    break;
  default:
    config = merge(
      common,
      parts.setupCSS(PATHS.app),
      ...
    );
}
module.exports = validate(config);
```

这个配置文件告诉我们凡是以 `.css` 结尾的文件都将会传入到给定的loader中，`test` 字段并没有匹配到js文件，并且这些loaders是从右向左执行的。

在这个例子中`css-loader`首先被执行，接着是`style-loader`。其中`css-loader`可以处理css文件中的 `@import` 和 `url` 语句，`style-loader`可以处理JavaScript中的 `require` 语句，就想类似于Sass和LESS一样的CSS预处理器。

loaders的作用是将一些资源转换为另一些资源，loaders可以被串行化，就想Unix中的piping一样，你可以参照文章[什么是loaders](#)，和[loaders列表](#)来进一步了解。

如果 `include` 字段没有设置，webpack将转化项目中所有符合的文件，这极大地降低了性能，所以，在配置loaders的时候，配置 `include` 字段是非常必要的。当然，还有一个 `exclude` 字段用于排除相关文件，不过我推荐使用 `include` 来指定文件。

初始化CSS

事实上CSS文件是这个样子的，

app/main.css

```
body {
  background: cornsilk;
}
```

我们接下来需要让webpack连接到它，但是webpack只能通过 `require` 语句来连接一个文件。

app/index.js

```
require('./main.css');  
...
```

现在执行 `npm start`，如果你使用了默认端口，在浏览器上打开 `http://localhost:8080`。

打开这个CSS文件，改变背景颜色为 `lime (background: lime)`，样式变得更加美观了一点。

Hello world

另外一个加载CSS的方法是针对CSS文件定义一个单独的入口。

理解CSS范围和CSS组件

当你使用这种方式来引用了一个CSS文件，webpack将会在你引用的位置将样式打包进这个文件的bundle中来。假设你在使用 `style-loader`，webpack将把他包含在html的 `<style>` 标签中，这意味着默认情况下这个样式的默认范围就是全局的。

特殊的，你可以使用 `CSS Modules` 来定义非全局的CSS。webpack的CSS loader支持这个特性，需要你通过 `css?modules` 来激活这个功能，然后将你的全局css通过 `:global(body){ ... }` 来进行声明。

关于query的语法 `css?modules` 将会在 *Loader Definition* 的章节中详细描述，并且在webpack中有多重方法可以达到同样的效果。

在这个例子中，可以通过 `require` 语句来带给你非全局的class，你可以将它针对元素进行绑定，就像下面这样。

app/main.css

```
:local(.redButton) {  
  background: red;  
}
```

我们接下来可以这样绑定元素

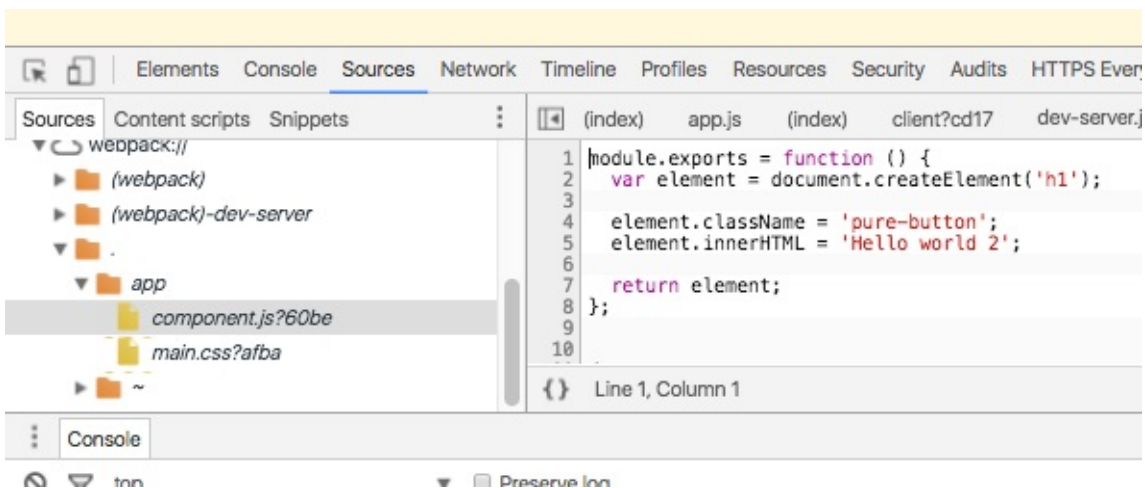
app/component.js

```
var styles = require('./main.css');  
...  
// Attach the generated class name  
element.className = styles.redButton;
```

尽管这种用法感到有点奇怪，使用非全局的方式可以减少你许多处理CSS的错误，我们将会在我们的项目中使用一些传统的写法，但是这种方式我们也需要了解。我想一种组合模式一样，值得我们学习。

结论

在本节中，你学习了如何在开发过程中搭建webpack并且自动刷新浏览器，在下一章节中，我们将讨论一个能极大提高开发效率的功能——sourcemaps。



为了让我们更好的调试程序，我们可以设置sourcemap方便对代码和样式进行调试。

sourcemap可以帮助你快速定位问题发生的地方，webpack可以生成两种sourcemap,包括在bundle中的行内sourcemaps和单独的sourcemap文件两种形式.行内的sourcemap在开发过程中有着非常好的调试体验，单独的sourmap用于生产环境中，可以使得bundle的大小降低。

我下面将会给你演示如何使用JavaScript的sourcemaps。当你想查看一个细节的时候，最好的方法是看它的文档。比如使用TypeScript你可能需要一个特定的标志来让它生效。

在开发过程中使用Sourcemap

为了能在开发中使用sourcemap，我们需要配置一个 `eval-source-map` 的配置。而在生产过程中，我们使用规范的单独的sourcemap文件。

webpack.config.js

```
...
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(
      common,
      {
        devtool: 'source-map'
      },
      parts.setupCSS(PATHS.app)
    );
  default:
    config = merge(
      common,
      {
        devtool: 'eval-source-map'
      },
      parts.setupCSS(PATHS.app),
      ...
    );
}
module.exports = validate(config);
```

`eval-source-map` 初始化构建的时候非常的缓慢，但是再次构建和生成文件的时候速度就会加快，你也可以使用类似 `cheap-module-eval-source-map` 和 `eval` 来加快速度，只不过他们生成了一个质量不高的sourcemap。所有的 `eval` 选项将会生成sourcemap作为你javascript代码的一部分。

你可以在你的工作中使用这些生成好的sourcemap，参照[Chrome](#)，[Firefox](#)，[IE Edge](#)和[Safari](#)来获得更多的细节。

webpack 支持的sourcemap类型

尽管两种类型的sourcemap，包括 `eval-source-map` 和 `eval` 在我们的开发过程中已经足够了，webpack给我们提供了更多类型的sourcemap，这些类型可以在你的bundles里自动生成，并且在生产环境中不可用。

下面的表格是从[文档](#)中根据打包速度排列的支持sourcemap的列表。sourcemap质量越低，打包速度越快。

Sourcemap type	Quality	Notes
<code>eval</code>	生成代码	每个模块都被 <code>eval</code> 执行，并且存在 <code>@sourceURL</code>
<code>cheap-eval-source-map</code>	转换代码（行内）	每个模块被 <code>eval</code> 执行，并且sourcemap作为 <code>eval</code> 的一个dataurl
<code>cheap-module-eval-source-map</code>	原始代码（只有行内）	同样道理，但是更高的质量和更低性能
<code>eval-source-map</code>	原始代码	同样道理，但是最高的质量和最低性能

你可以从 `eval-source-map` 开始试用，逐渐地转变成其他的选项感受一下性能的差距。

webpack还可以生成生产环境下友好的sourcemaps。它可有在浏览器需要的时候引用单独的文件。在这个场景下你依然可以很轻松地进行调试。下面是他们的功能特点。

Sourcemap type	Quality	Notes
<code>cheap-source-map</code>	转换代码（行内）	生成的sourcemap没有列映射，从loaders生成的sourcemap没有被使用
<code>cheap-module-source-map</code>	原始代码（只有行内）	与上面一样除了每行特点的从loader中进行映射
<code>source-map</code>	原始代码	最好的sourcemap质量有完整的结果，但是会很慢

`source-map` 在这里是一个默认值，尽管使用他生成代码会花费很长时间，换来了更高的质量。如果你不考虑生产环境下的sourcemap，你可以忽略此项。

还有一些其他影响sourcemap生成的参数。

```
const config = {
  output: {
    // 你可以使用[file], [id]和[hash]来修改生成sourcemap的文件名
    // 默认的选项在大多数情况下足够了
    sourceMapFilename: '[file].map', // Default

    // 这个是 sourcemap 文件名模板，依赖于devtool的选项，你一般情况下不要修改它
    devtoolModuleFilenameTemplate: 'webpack:///[resource-path]?[loaders]'
  },
  ...
};
```

这里有devtool的[官方文档](#)

SourceMapDevToolPlugin

如果你想更好的控制sourcemap的生成，你可能需要使用 `SourceMapDevToolPlugin` 作为替代。通过这个你可以控制生成SourceMap的范围。当你使用插件的时候，你可以跳过 `devtool` 的选项。

下面是一个来自[官方文档](#)的例子

```
const config = {
  plugins: [
    new webpack.SourceMapDevToolPlugin({
      // 像loaders一样匹配文件
      test: string | RegExp | Array,
      include: string | RegExp | Array,

      // `exclude` 匹配文件名，而不是包名
      exclude: string | RegExp | Array,

      // 如果文件名设置了，输出这个文件
      // 参考 `sourceMapFileName`
      filename: string,

      // This line is appended to the original asset processed. For
      // instance '[url]' would get replaced with an url to the
      // sourcemap.
      append: false | string,

      // See `devtoolModuleFilenameTemplate` for specifics.
      moduleFilenameTemplate: string,
      fallbackModuleFilenameTemplate: string,

      module: bool, // If false, separate sourcemaps aren't generated.
      columns: bool, // If false, column mappings are ignored.

      // Use simpler line to line mappings for the matched modules.
      lineToLine: bool | {test, include, exclude}
    }),
    ...
  ],
  ...
};
```

为样式文件的Sourcemaps

你可以通过一个query参数使用样式的SourceMap文件。就想`css-loader`,`sass-loader`,`less-loader`，除了一个 `?sourceMap` （例如 `css?sourceMap` ）

但是并没有这么简单，`css-loader`的官方文档建议我们进行css声明的时候使用官方文档，使用 `output.publicPath` 来替代server url。

结论

使用sourcemap可以给我们的开发带来巨大的方便，它可以在从打包之后的代码贯穿到原始代码，他还可以用于生产过程中，可以方便的让你进行调试。

我们的打包配置目前还没有进行得很完善，接下来我们会讨论许多与打包相关的技术。