--

[webpack 1](#_Toc12990)

[webpack 1](#_Toc15994)

[用 babel-loader 转换 ES2015 2](#_Toc12049)

[webpack-dev-server 3](#_Toc3548)

[Automatic Refresh 3](#_Toc23543)

[Hot Module Replacement 4](#_Toc32160)

[与本地API Server 5](#_Toc21090)

[react热替换 5](#_Toc19925)

[开发环境 生产环境 使用案例 6](#_Toc14722)

[demo 6](#_Toc15069)

webpack

|  |  |
| --- | --- |
|  | Gulp和Webpack对比  <http://chping.website/2016/10/18/gulp%E5%92%8Cwebpack%E5%AF%B9%E6%AF%94/>  认识Webpack  <http://aztack.wang/post/about-webpack.html>  webpack + gulp 构建完整前端工作流  <https://zhuanlan.zhihu.com/p/21312474>  简易：[webpack的配置文件entry与output](http://www.cnblogs.com/zhubei/p/6479454.html)  <http://www.cnblogs.com/zhubei/p/6479454.html>  完美教程 react vue webpack  <https://www.geekjc.com/post/58b307e4f8251573a1d2dfa5>  <https://www.geekjc.com/book/58b3027ef8251573a1d2dfa3>  <https://www.geekjc.com/book/58b4e761c8c6ea16bd30445e>  中文教程（完全版本）  <https://doc.webpack-china.org/guides/getting-started/> |
|  | html-webpack-plugin  <http://blog.csdn.net/keliyxyz/article/details/51513114>  Webpack——解决疑惑,让你明白  <http://www.jianshu.com/p/dcb28b582318>  <https://github.com/chemdemo/chemdemo.github.io/issues/13>  搜罗一切webpack的好文章好工具  <https://github.com/webpack-china/awesome-webpack-cn> |
|  | [webpack使用webpack-dev-middleware进行热重载](http://www.cnblogs.com/dreamless/p/6008362.html)  <http://www.cnblogs.com/dreamless/p/6008362.html> |
| 优化 | [Webpack 打包优化之速度篇](https://jeffjade.com/2017/08/12/125-webpack-package-optimization-for-speed/)  <https://jeffjade.com/2017/08/12/125-webpack-package-optimization-for-speed/>  -配置 resolve.modules，减少搜索范围  -设置 test & include & exclude  -增强代码代码压缩工具；Webpack 默认提供的 UglifyJS 插件，由于采用单线程压缩，速度颇慢 ；推荐采用 [webpack-parallel-uglify-plugin](https://www.npmjs.com/package/webpack-parallel-uglify-plugin" \t "https://jeffjade.com/2017/08/12/125-webpack-package-optimization-for-speed/_blank) 插件，  -设置 babel 的 cacheDirectory 为true  [Webpack 打包优化之体积篇](https://jeffjade.com/2017/08/06/124-webpack-packge-optimization-for-volume/)  <https://jeffjade.com/2017/08/06/124-webpack-packge-optimization-for-volume/>  -引入 DllPlugin 和 DllReferencePlugin  -外部引入模块(CDN)；考虑外部引入之，再借助 externals 予以指定  -按需异步加载模块；把import Foo from './Foo.vue' 改为如下写法:   const Foo = () => import('./Foo.vue')  -生产环境，压缩混淆并移除console  new webpack.optimize.UglifyJsPlugin({  compress: {  warnings: false,  drop\_console: true,  pure\_funcs: ['console.log']  },  sourceMap: false  }) |
| 中文简易教程 | webpack 2  <http://www.css88.com/doc/webpack2/>  webpack 3  <http://www.css88.com/doc/webpack/> |
| react结合中文的使用 | [React 最佳实践——那些 React 没告诉你但很重要的事](https://segmentfault.com/a/1190000005013207)  <https://segmentfault.com/a/1190000005013207>  [React从入门到精通系列之（](https://segmentfault.com/a/1190000007797584)1-22）  <https://segmentfault.com/blog/zhangyatao?page=2>  [webpack-es6-react （为系统学习React布一个良好的开局）](https://segmentfault.com/a/1190000004579183)  <https://segmentfault.com/a/1190000004579183?_ea=676957>  <https://github.com/minooo/React-Study/tree/master/step-01>  [React+ES6+Webpack深入浅出](http://www.cnblogs.com/chenziyu-blog/p/5675086.html)  <http://www.cnblogs.com/chenziyu-blog/p/5675086.html> |

webpack

|  |  |
| --- | --- |
|  | Webpack的构建过程需要一个配置文件，一个典型的配置文件大概就是这样：  entry：指定打包的入口文件，每有一个键值对，就是一个入口文件  output：配置打包结果，path定义了输出的文件夹，filename则定义了打包结果文件的名称，filename里面的[name]会由entry中的键（这里是entry1和entry2）替换  resolve：定义了解析模块路径时的配置，常用的就是extensions，可以用来指定模块的后缀，这样在引入模块时就不需要写后缀了，会自动补全  module：定义了对模块的处理逻辑，这里可以用loaders定义了一系列的加载器，以及一些正则。当需要加载的文件匹配test的正则时，就会调用后面的loader对文件进行处理，这正是webpack强大的原因。比如这里定义了凡是.js结尾的文件都是用babel-loader做处理，而.jsx结尾的文件会先经过jsx-loader处理，然后经过babel-loader处理。当然这些loader也需要通过npm install安装  plugins: 这里定义了需要使用的插件，比如commonsPlugin在打包多个入口文件时会提取出公用的部分，生成common.js |
|  |
| 与gulp | 执行打包  如果通过npm install -g webpack方式安装webpack的话，可以通过命令行直接执行打包命令，比如这样：  $webpack --config webpack.config.js  这样就会读取当前目录下的webpack.config.js作为配置文件执行打包操作  如果是通过gulp插件gulp-webpack，则可以在gulpfile中写上gulp任务：  var gulp  =  require('gulp');  var webpack  =  require('gulp-webpack');  var webpackConfig  =  require('./webpack.config');  gulp.task("webpack",  function()  {  return gulp          .src('./')          .pipe(webpack(webpackConfig))          .pipe(gulp.dest('./build'));  }); |
| css-loader | 样式编写  我是一个强烈地Less依赖患者，脱离了Less直接写CSS就会出现四肢乏力、不想干活、心情烦躁等现象，而且还不喜欢在写Less时候加前缀，平常都是gulp+less+autoprefixer直接处理的，那么在Webpack组织的React组件中要怎么写呢？  没错，依旧是使用loader  可以在webpack.config.js的loaders中增加Less的配置：  {    test: /\.less$/,    loader: 'style-loader!css-loader!autoprefixer-loader!less-loader'  }  通过这样的配置，就可以直接在模块代码中引入Less样式了：  import React from  'react';  require('./HelloWorldComponent.less');  export default class HelloWorldComponent extends React.Component  {  constructor()  {  super();  this.state  =   {};  }  render()  {  return ( < div > Hello World < / div > );  }  }  其他  Webpack的loader为React组件化提供了很多帮助，像图片也提供了相关的loader：  { test: /\.png$/, loader: "url-loader?mimetype=image/png" }  更多地loader可以移步[webpack的wiki](https://github.com/webpack/docs/wiki/list-of-loaders" \t "https://my.oschina.net/bosscheng/blog/_blank) |
| w-d-s | 在Webpack下实时调试React组件  Webpack和React结合的另一个强大的地方就是，在修改了组件源码之后，不刷新页面就能把修改同步到页面上。这里需要用到两个库webpack-dev-server和react-hot-loader。  首先需要安装这两个库，npm install --save-dev webpack-dev-server react-hot-loader  安装完成后，就要开始配置了，首先需要修改entry配置：  entry:   {    helloworld:  [  'webpack-dev-server/client?http://localhost:3000',  'webpack/hot/only-dev-server',  './helloworld'    ]  },  通过这种方式指定资源热启动对应的服务器，然后需要配置react-hot-loader到loaders的配置当中，比如我的所有组件代码全部放在scripts文件夹下：  {    test: /\.js?$/,    loaders: ['react-hot', 'babel'],    include: [path.join(\_\_dirname, 'scripts')]  }  最后配置一下plugins，加上热替换的插件和防止报错的插件：  plugins: [  new webpack.HotModuleReplacementPlugin(),  new webpack.NoErrorsPlugin()  ]  这样配置就完成了，但是现在要调试需要启动一个服务器，而且之前配置里映射到http://localhost:3000，所以就在本地3000端口起个服务器吧，在项目根目录下面建个server.js：  var webpack  =  require('webpack');  var WebpackDevServer  =  require('webpack-dev-server');  var config  =  require('./webpack.config');  new WebpackDevServer(webpack(config),   {    publicPath:  config.output.publicPath,    hot:  true,    historyApiFallback:  true  }).listen(3000,   'localhost',  function (err,  result)  {    if (err) console.log(err);    console.log('Listening at localhost:3000');  });  这样就可以在本地3000端口开启调试服务器了，比如我的页面是根目录下地index.html，就可以直接通过http://localhost:3000/index.html访问页面，修改React组件后页面也会被同步修改，这里貌似使用了websocket来同步数据。  图是一个简单的效果：  IMG_256 |
| w-d-s | 下面我们通过简单的例子来学习webpack-dev-server的用法（[代码下载](https://github.com/xianfeishen/Webpack/tree/master/sample3)）。项目目录和结构仍然与教程一中的基本一致，不同的是要增加一个webpack.config.js文件：  module.exports = {  devtool: 'eval-source-map',  entry: \_\_dirname + "/app/main.js",  output: {  path: \_\_dirname + "/public",  filename: "bundle.js"  },  devServer: {  contentBase: "./public", //以public为根目录提供文件  colors: true,  historyApiFallback: true,  inline: true  }  };  然后运行webpack-dev-server命令，顺利启动服务器后，在浏览器中输入http://localhost:8080/index.html就可以看到页面了。[代码下载](https://github.com/xianfeishen/Webpack/tree/master/sample3) |
| 用 babel-loader 转换 ES2015 | 用 babel-loader 转换 ES2015  在这个例子中， 我们会告诉 webpack　用　babel 跑我们的源码，这样就可以用 ES2015新特性了。  1.先安装 babel 和 预设。  npm intall --save-dev babel-core babel-preset-es2015  2.安装 babel-loader  npm install --save-dev babel-loader  3.添加 .babelrc 文件来配置 babel 使用 规范。  npm install --save-dev babel-loader  4.修改 webpack.config.js 用babel-loader 处理所以 .js文件。  module.exports = {  entry: './src/app.js',  output: {  path: './bin',  filename: 'app.bundle.js',  },  module: {  loaders: [{  test: /\.js$/,  exclude: /node\_modules/,  loader: 'babel-loader',  }]  }  }  我们排除 node\_modules中的文件，因为要不然的话所有外部库都会通过babel编译，将会降低 编译速度。  5.安装你想用的第三方库。（例如 jQuery）  npm install --save jquery babel-polyfill  这里用 –save 而不是 –save-dev，　是因为这些库在执行时会用到。我们也安装了babel-polyfill 这样在一些老式浏览中可以用 ES2015了。   1. 编辑 src/app.js   import 'babel-polyfill';  import cats from './cats';  import $ from 'jquery';  $('<h1>Cats</h1>').appendTo('body');  const ul = $('<ul></ul>').appendTo('body');  for (const cat of cats) {  $('<li></li>').text(cat).appendTo(ul);  }  7.用webpack 打包  webpack  8.添加 index.html，这样app能跑起来。  <!DOCTYPE html><body>  <script src="bin/app.bundle.js"></script>  有许多不同的loders你可以用在app打包中，包括 css 和 image　loader。 |
| 小结 | 另外.vue文件也要vue-loader来处理。  所以，我们还需要安装几个webpack的模块(module)  npm install vue-loader vue-html-loader css-loader style-loader file-loader --save -dev  最最核心的必须要下载vue：  npm install vue vue-template-compiler --save -dev  下载babel需要的  npm install babel-cli babel-loader babel-core babel-plugin-transform-runtime babel-preset-es2015 babel-runtime --save -dev  根目录新建.babelrc文件，内容如下：  {  "presets":["es2015"]  }  webpack-merge：开发环境和生产环节的webpaak配置文件的配置合并  file-loader：编译写入文件，默认情况下生成文件的文件名是文件名与MD5哈希值的组合  vue-laoder：编译写入.vue文件  vue-html-loader：编译vue的template部分  vue-style-loader：编译vue的样式部分  vue-hot-reload-api：webpack对vue实现热替换  babel-core：ES2015编译核心  babel-loader：编译写入ES2015文档  babel-preset-es2015：ES2015语法  babel-preset-stage-0：开启测试功能  babel-runtime：babel执行环境 |
|  |  |
|  |  |
|  | 在webpack.config文件中添加  var OpenBrowserPlugin = require('open-browser-webpack-plugin');  plugins: [  new OpenBrowserPlugin({ url: 'http://localhost:8080' })  ]  则在webpack 启动成功后会打开浏览器  参考文档 [https://www.npmjs.com/package/open-browser-webpack-plugin](https://www.npmjs.com/package/open-browser-webpack-plugin" \t "http://blog.csdn.net/isaisai/article/details/_blank) |
|  |  |
|  | context: path.resolve(\_\_dirname, './src'), |
|  |  |

webpack-dev-server

|  |  |  |
| --- | --- | --- |
|  | [详解webpack-dev-server的使用](https://segmentfault.com/a/1190000006964335)  <https://segmentfault.com/a/1190000006964335>  [webpack-dev-server使用方法，看完还不会的来找我~](https://segmentfault.com/a/1190000006670084)  <https://segmentfault.com/a/1190000006670084>  [【webpack】webpack-dev-server生猛上手——让我们来搭一个webpack的微服务器吧！](http://www.cnblogs.com/penghuwan/p/6941616.html)  <http://www.cnblogs.com/penghuwan/p/6941616.html> | |
|  | 这是一个基于Express.js框架开发的web server，默认监听8080端口。server内部调用Webpack，这样做的好处是提供了额外的功能如热更新“Live Reload”以及热替换“Hot Module Replacement”（即HMR）。 | |
|  | 记录下webpack-dev-server的用法.  首先，我们来看看基本的webpack.config.js的写法  module.exports = {  entry: './src/js/index.js',  output: {  path: './dist/js',  filename: 'bundle.js'  }  }  配置文件提供一个入口和一个出口，webpack根据这个来进行js的打包和编译工作。虽然webpack提供了webpack --watch的命令来动态监听文件的改变并实时打包，输出新bundle.js文件，这样文件多了之后打包速度会很慢，此外这样的打包的方式不能做到hot replace，即每次webpack编译之后，你还需要手动刷新浏览器。  webpack-dev-server其中部分功能就能克服上面的2个问题。webpack-dev-server主要是启动了一个使用express的Http服务器。它的作用主要是用来伺服资源文件。此外这个Http服务器和client使用了websocket通讯协议，原始文件作出改动后，webpack-dev-server会实时的编译，但是最后的编译的文件并没有输出到目标文件夹，即上面配置的:  output: {  path: './dist/js',  filename: 'bundle.js'  }  注意：你启动webpack-dev-server后，你在目标文件夹中是看不到编译后的文件的,实时编译后的文件都保存到了内存当中。因此很多同学使用webpack-dev-server进行开发的时候都看不到编译后的文件 | |
|  | 下面来结合webpack的文档和webpack-dev-server里部分源码来说明下如何使用：  启动  启动webpack-dev-server有3种方式： [ cmd line | Node.js API | npm script]  方式 1:  // 全局安装  npm install webpack-dev-server --save// 终端输入  $ webpack-dev-server --inline --hot  用法 3:  // 添加到package.json scripts  "scripts": {  "start": "webpack-dev-server --inline --hot",  ...  }  // 运行：  $ npm start  // 浏览器预览：  http://localhost:8080  配置  目录结构是:  app  |\_\_dist  | |\_\_styles  | |\_\_js  | |\_\_bundle.js  | |\_\_index.html  |\_\_src  | |\_\_styles  | |\_\_js  | |\_\_index.js  |\_\_node\_modules  |\_\_package.json  |\_\_webpack.config.js  webpack-dev-server --content-base ./dist  // content-base设定webpack-dev-server伺服的directory。如果不进行设定的话，默认是在当前目录下。 | |
|  | 这个时候还要注意的一点就是在webpack.config.js文件里面，如果配置了output的publicPath这个字段的值的话，在index.html文件里面也应该做出调整。  因为webpack-dev-server伺服的文件是相对publicPath这个路径的。 | |
| 因此，如果你的webpack.config.js配置成这样的：  module.exports = {  entry: './src/js/index.js',  output: {  path: './dist/js',  filename: 'bundle.js'，  publicPath: '/assets/'  }  }  那么，在index.html文件当中引入的路径也发生相应的变化:  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <title>Demo</title>  </head>  <body>  <script src="assets/bundle.js"></script>  </body>  </html> | 如果在webpack.config.js里面没有配置output的publicPath的话，那么index.html最后引入的文件js文件路径应该是下面这样的。  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <title>Demo</title>  </head>  <body>  <script src="bundle.js"></script>  </body>  </html> |
| Automatic Refresh | webpack-dev-server支持2种自动刷新的方式：  1. iframe模式(页面放在iframe中,当发生改变时重载)  2. inline模式(将webpack-dev-sever的客户端入口添加到包(bundle)中)  两种模式都支持热模块替换(Hot Module Replacement).热模块替换的好处是只替换更新的部分,而不是页面重载.  这2种模式配置的方式和访问的路径稍微有点区别，最主要的区别还是Iframe mode是在网页中嵌入了一个iframe，将我们自己的应用注入到这个iframe当中去，因此每次你修改的文件后，都是这个iframe进行了reload。  通过查看webpack-dev-server的源码，lib路径下的Server.js文件，第38-48行，分别新建几个流，这几个流保存了client文件夹下的相关文件：  // Prepare live html page  var livePage = this.livePage = new StreamCache();  fs.createReadStream(path.join(\_\_dirname, "..", "client", "live.html")).pipe(livePage);  // Prepare the live js file  var liveJs = new StreamCache();  fs.createReadStream(path.join(\_\_dirname, "..", "client", "live.bundle.js")).pipe(liveJs);  // Prepare the inlined js file  var inlinedJs = new StreamCache();  fs.createReadStream(path.join(\_\_dirname, "..", "client", "index.bundle.js")).pipe(inlinedJs);  // Init express server  var app = this.app = new express();  // middleware for serving webpack bundle  this.middleware = webpackDevMiddleware(compiler, options);  app.get("/\_\_webpack\_dev\_server\_\_/live.bundle.js", function(req, res) {  res.setHeader("Content-Type", "application/javascript");  liveJs.pipe(res);  });  app.get("/webpack-dev-server.js", function(req, res) {  res.setHeader("Content-Type", "application/javascript");  inlinedJs.pipe(res);  });  app.get("/webpack-dev-server/\*", function(req, res) {  res.setHeader("Content-Type", "text/html");  this.livePage.pipe(res);  }.bind(this)); | |
| iframe模式 | 当使用Iframe mode时，请求/webpack-dev-server/index.html路径时，会返回client/index.html文件，这个文件的内容就是：  <!DOCTYPE html><html><head><meta http-equiv="X-UA-Compatible" content="IE=edge"/><meta charset="utf-8"/><meta name="viewport" content="width=device-width, height=device-height, initial-scale=1.0, user-scalable=no, minimum-scale=1.0, maximum-scale=1.0"/><script type="text/javascript" charset="utf-8"src="/\_\_webpack\_dev\_server\_\_/live.bundle.js"></script></head><body></body></html>  这个页面会请求live.bundle.js,其中里面会新建一个Iframe，你的应用就被注入到了这个Iframe当中。同时live.bundle.js中含有socket.io的client代码，这样它就能和webpack-dev-server建立的http server进行websocket通讯了。并根据返回的信息完成相应的动作。  使用这种模式不需要额外的配置,只需要以下面这种URL格式访问即可  http://«host»:«port»/webpack-dev-server/«path» | |
| inline模式 | inline模式下我们访问的URL不用发生变化。  配置方式1:  而Inline-mode，是webpack-dev-server会在你的webpack.config.js的入口配置文件中再添加一个入口:  module.exports = {  entry: {  app: [  'webpack-dev-server/client?http://localhost:8080/',  './src/js/index.js'  ]  },  output: {  path: './dist/js',  filename: 'bundle.js'  }  }  这样就完成了将inlinedJS打包进bundle.js里的功能，同时inlinedJS里面也包含了socket.io的client代码，可以和webpack-dev-server进行websocket通讯。  配置方式2：  当然你也可以直接在你index.html引入这部分代码:  <script src="http://localhost:8080/webpack-dev-server.js"></script>  启用方式1：  当以命令行启动webpack-dev-server时,需要做两点：  在命令行中添加--inline命令  在webpack.config.js中添加devServer:{inline:true}  启用方式2：  当以Node.js API启动webpack-dev-server时,我们也需要做两点:  由于webpack-dev-server的配置中无inline选项,我们需要添加webpack-dev-server/client?http://«path»:«port»/到webpack配置的entry入口点中.  将<script src="http://localhost:8080/webpack-dev-server.js"></script>添加到html文件中  var config = require("./webpack.config.js");  var webpack = require('webpack');  var WebpackDevServer = require('webpack-dev-server');  config.entry.app.unshift("webpack-dev-server/client?http://localhost:8080/");  var compiler = webpack(config);  new WebpackDevServer(compiler, {  contentBase: 'build/',  publicPath: "/assets/"  }).listen(8080);  在Node中运行上面的代码即可。    启动方式3：  "dev": "webpack-dev-server --devtool eval-source-map --progress --colors --hot --inline --content-base ./assets",  **注意：webpack配置中的devSever配置项只对在命令行模式webpack-dev-server有效。** | |
| 其实entry不需要搞那么麻烦：  entry: {  app: './assets/main.js'  },  只要在把inline也启动起来就好了：  new WebpackDevServer(webpack(config), {  publicPath : config.output.publicPath,  hot : true,  inline : true,//启动inline  noInfo : true,  historyApiFallback: true  }).listen(8080, '127.0.0.1', function (err, res) {  if (err) console.log(err);  console.log('At 3000');  }); | |
| 比较 | 不过Iframe mode和Inline mode最后达到的效果都是一样的，都是监听文件的变化，然后再将编译后的文件推送到前端，完成页面的reload的。  Iframe mode(默认)  Iframe mode下cmd line不需要添加其他的内容: webpack-dev-server  浏览器访问的路径是: localhost:8080/webpack-dev-server/index.html。  这个时候这个页面的header部分会出现整个reload消息的状态。当时改变源文件的时候，即可以完成自动编译打包，页面自动刷新的功能。  Inline mode  使用inline mode的时候，cmd line需要写成: webpack-dev-server --inline --content-base ./dist  这个时候访问的路径是: localhost:8080/index.html  也能完成自动编译打包，页面自动刷新的功能。但是没有的header部分的reload消息的显示，不过在控制台中会显示reload的状态。 | |
| Hot Module Replacement | 在命令行中运行inline模式，并启用热模块替换  这里只需要多增加 --hot指令就OK了:  webpack-dev-server --hot --inline --content-base ./dist  "scripts": {  "dev": "webpack-dev-server --devtool eval-source-map --progress --colors --hot --inline --content-base ./dist",  "build": "webpack --progress --colors"  },  注意:命令行模式下,webpack.config.js中一定要配置output.publicPath来指定编译后的包(bundle)的访问位置.  在Nodejs API中运行inline模式，并启用热模块替换  这里需要做以下三点:  在webpack.config.js的entry选项中添加:webpack/hot/dev-server  在webpack.config.js的plugins选项中添加:new webpack.HotModuleReplacementPlugin()  在webpack-dev-server的配置中添加：hot:true | |
|  | webpack-dev-server的“hot” 和 “inline”选项  注意像inline和hot这些选项是Webpack-dev-server特有的，而另外的如hide-modules则是CLI模式特有的选项。  “inline”选项会为入口页面添加“热加载”功能，“hot”选项则开启“热替换（Hot Module Reloading）”，即尝试重新加载组件改变的部分（而不是重新加载整个页面）。如果两个参数都传入，当资源改变时，webpack-dev-server将会先尝试HRM（即热替换），如果失败则重新加载整个入口页面。  // 当资源发生改变，以下三种方式都会生成新的bundle，但是又有区别：  // 1. 不会刷新浏览器  $ webpack-dev-server  //2. 刷新浏览器  $ webpack-dev-server --inline  //3. 重新加载改变的部分，HRM失败则刷新页面  $ webpack-dev-server --inline --hot  另外值得注意的是你可以通过以下两种方式向webpack-dev-server传入参数：  // 通过CLI传参  webpack-dev-server --hot --inline  // 通过webpack.config.js文件的"devServer"对象  devServer: {  inline: true,  hot:true  }  我发现有时devServer配置项（hot: true 和inline: true）不生效，我更偏向使用如下的方式向CLI传递参数：  "scripts": "webpack-dev-server --hot --inline" // package.json  注意：确定你没有同时传入hot:true、--hot | |
|  | webpack-dev-server中的配置选项  var WebpackDevServer = require("webpack-dev-server");  var webpack = require("webpack");  var compiler = webpack({  // configuration  });  var server = new WebpackDevServer(compiler, {  // webpack-dev-server options  contentBase: "/path/to/directory",  // Can also be an array, or: contentBase: "http://localhost/",  hot: true,  // Enable special support for Hot Module Replacement  // Page is no longer updated, but a "webpackHotUpdate" message is send to the content  // Use "webpack/hot/dev-server" as additional module in your entry point  // Note: this does \_not\_ add the `HotModuleReplacementPlugin` like the CLI option does.  // Set this as true if you want to access dev server from arbitrary url.  // This is handy if you are using a html5 router.  historyApiFallback: false,  // Set this if you want to enable gzip compression for assets  compress: true,  // Set this if you want webpack-dev-server to delegate a single path to an arbitrary server.  // Use "\*\*" to proxy all paths to the specified server.  // This is useful if you want to get rid of 'http://localhost:8080/' in script[src],  // and has many other use cases (see https://github.com/webpack/webpack-dev-server/pull/127 ).  proxy: {  "\*\*": "http://localhost:9090"  },  setup: function(app) {  // Here you can access the Express app object and add your own custom middleware to it.  // For example, to define custom handlers for some paths:  // app.get('/some/path', function(req, res) {  // res.json({ custom: 'response' });  // });  },  // pass [static options](http://expressjs.com/en/4x/api.html#express.static) to inner express server  staticOptions: {  },  // webpack-dev-middleware options  quiet: false,  noInfo: false,  lazy: true,  filename: "bundle.js",  watchOptions: {  aggregateTimeout: 300,  poll: 1000  },  // It's a required option.  publicPath: "/assets/",  headers: { "X-Custom-Header": "yes" },  stats: { colors: true }  });  server.listen(8080, "localhost", function() {});  // server.close(); | |
|  | 配置参数说明如下：  --content-base <file/directory/url/port>: base path for the content.  --quiet: don’t output anything to the console.  --no-info: suppress boring information.  --colors: add some colors to the output.  --no-colors: don’t use colors in the output.  --compress: use gzip compression.  --host <hostname/ip>: hostname or IP. 0.0.0.0 binds to all hosts.  --port <number>: port.  --inline: embed the webpack-dev-server runtime into the bundle.  --hot: adds the HotModuleReplacementPlugin and switch the server to hot mode. Note: make sure you don’t add HotModuleReplacementPlugin twice.  --hot --inline also adds the webpack/hot/dev-server entry.  --public: overrides the host and port used in --inline mode for the client (useful for a VM or Docker).  --lazy: no watching, compiles on request (cannot be combined with --hot).  --https: serves webpack-dev-server over HTTPS Protocol. Includes a self-signed certificate that is used when serving the requests.  --cert, --cacert, --key: Paths the certificate files.  --open: opens the url in default browser (for webpack-dev-server versions > 2.0).  --history-api-fallback: enables support for history API fallback.  --client-log-level: controls the console log messages shown in the browser. Use error, warning, info or none.  webpack-dev-server相关介绍请参见这里 | |
|  | devServer.overlay  这个配置属性用来在编译出错的时候，在浏览器页面上显示错误，默认是false，可设置为true  devServer.historyApiFallback  在文档里面说的很清楚，这个配置属性是用来应对返回404页面时定向到特定页面用的（the index.html page will likely have to be served in place of any 404 responses)  historyApiFallback:{  rewrites:[  {from:/./,to:'/404.html'}  ]  }  }  }  devServer.stats（字符串）  这个配置属性用来控制编译的时候shell上的输出内容  stats: "errors-only"表示只打印错误：  我们把配置改成：  devServer: {  contentBase: path.join(\_\_dirname, "dist"),  stats: "errors-only"  }  devServer.compress  这是一个布尔型的值，当它被设置为true的时候对所有的服务器资源采用gzip压缩  采用gzip压缩的优点和缺点：  优点：对JS，CSS资源的压缩率很高，可以极大得提高文件传输的速率，从而提升web性能  缺点：服务端要对文件进行压缩，而客户端要进行解压，增加了两边的负载    devServer.hot和devServer.inline  在这之前，首先要说一下的是webpack-dev-server的自动刷新和模块热替换机制 | |
| 与本地API Server | 如果你在本地还启动了一个api server,port为3000,这个server主要和你的前端应用进行数据交互。这个时候很显然会出现跨域的问题，那么这个时候，你前端应用的入口文件应当是用你自己启动的api server提供的。  var express = require('express');  var app = express();  app.get('/', function(req, res) {  res.send('xxx/xxx/index.html'); //这个地方填写dist/index.html的路径  })  此外webpack.config.js:  module.exports = {  entry: './src/js/index.js',  output: {  path: './dist/js',  filename: 'bundle.js',  publicPath: 'localhost:8080/dist'  },  devServer: {  '/get': {  targer: 'localhost:3000',  secure: false  }  }  }  将publicPath字段的内容配置为绝对路径。同时index.html文件中对js引用的路径也改为绝对路径  <script src="localhost:8080/dist/bundle.js"></script>  如果对web-dev-server还有其他问题的话，请留言告知。  另外2篇关于webpack的文章:  [webpack1.x分包及异步加载套路](https://segmentfault.com/a/1190000007962830)  [webpack2分包及异步加载套路](https://segmentfault.com/a/1190000008279471" \t "https://segmentfault.com/a/_blank) | |
| react热替换 | Webpack有一个很实用的功能叫做热替换（Hot-replace），尤其是结合React Hot Loader插件，开发过程中都不需要刷新浏览器，任何前端代码的更改都会实时的在浏览器中表现出来。  首先需要安装Webpack-dev-server,一个轻量的node.js express服务器。  **npm install webpack-dev-server --save-dev**  Webpack-dev-server十分小巧，这里的作用是用来伺服资源文件，不能替代后端的服务器，因此如果你还要进行后端开发，就要采用双服务器模式：一个后端服务器和一个资源服务器（即Webpack-dev-server)  配置双服务器热替换模式首先需要修改Wepack.config.js的entry条目  entry: [      'webpack-dev-server/client?http://0.0.0.0:9090',//资源服务器地址      'webpack/hot/only-dev-server',//react的热加载      './static/js/entry.coffee'  ] | |
|  | output条目,将publicPath设置为webpack-dev-server资源服务器服务器下资源目录的绝对路径  output: {      publicPath: "http://127.0.0.1:9090/static/dist/",      path: './static/dist/',      filename: "bundle.js"  }  在Plugins中增加  new webpack.DefinePlugin({      'process.env.NODE\_ENV': '"development"'  }),  new webpack.HotModuleReplacementPlugin()  webpack变量需要在Webpack.config.js的开始处声明  var webpack = require('webpack');  接着中页面文件中插入打包后的资源文件，注意这里要采用资源服务器的绝对路径:  <script src="http://127.0.0.1:9090/static/dist/bundle.js"></script>  双服务器的原理其实就是让后端服务器伺服的页面加载资源服务伺服的资源。 | |
|  | 如果要实现react的热加载，还需要[React Hot Loader插件](http://gaearon.github.io/react-hot-loader/getstarted/" \t "http://www.jianshu.com/p/_blank) ，  **npm install react-hot-loader --save-dev**  然后将react-hot loader放在jsx文件loader链的第一位  {  test: /\.jsx?$/,  loaders: ['react-hot', 'jsx?harmony'],  include: path.join(\_\_dirname, 'src')  }  最后就是启动Webpack-dev-server服务器，注意这里要采用inline模式， 如果采用hot模式会出现错误。--content-base 为资源文件夹。另外如果你想用另一台电脑做调试，需要增加--host参数为开发服务器的IP地址，因为默认只能通过localhost进行方式。如果IP地址经常变动，可以写一个脚本方便每次启动服务器  #!/bin/sh  ip=`awk '/inet / && $2 != "127.0.0.1"{print $2}' <(ifconfig)`  node webpack-dev-server.js --port 9090 --inline --host $ip --content-base static/dist/  当资源文件修改时，webpack-dev-server将通过socket.io通知客户端更新． | |

开发环境 生产环境 使用案例

|  |  |
| --- | --- |
| demo | 1. 开发模式和生产模式(<http://www.jianshu.com/p/dcb28b582318>)  首先要知道的是Webpack有许许多多的特性，一些是”开发模式“下才有的，一些是”生产模式“下才有的，还有一些是两种模式下都有的。  IMG_256 |
|  | 值得注意的是，Webpack作为模块打包工具，提供两种用户交互接口：  Webpack CLI tool：默认的交互方式（已随Webpack本身安装到本地）  webpack-dev-server：一个Node.js服务器（需要开发者从npm自行安装）  通常使用到Webpack如此多特性的项目都会有两个比较大的Webpack配置文件  为了生成bundles文件你可能在package.json文件加入如下的scripts项：  "scripts": {  // 运行npm run build 来编译生成生产模式下的bundles  "build": "webpack --config webpack.config.prod.js",  // 运行npm run dev来生成开发模式下的bundles以及启动本地server  "dev": "webpack-dev-server"  } |
| demo | Webpack给本地开发提供了一个可选的服务器webpack-dev-server。webpack-dev-server是一个很小的express应用，使用前需要用npm安装，它根据webpack.config.js文件中的选项构建。常见的选项如下：   |  |  | | --- | --- | | webpack-dev-server选项 | 选项说明 | | content-Base | 默认情况下，webpack-dev-server会从项目的根目录提供文件，可以通过此选项设置文件的目录名 | | port | 服务器使用的端口，默认情况下为8080 | | inline | 设为true时可以在文件发生变化时，更新页面 | | colors | 设置终端输出字体颜色 | | historyApiFallback | 当设置为true时，访问所有服务器上不存在的文件，都会被重定向到/，也就是index.html文件 |    下面我们通过简单的例子来学习webpack-dev-server的用法（[代码下载](https://github.com/xianfeishen/Webpack/tree/master/sample3)）。项目目录和结构仍然与教程一中的基本一致，不同的是要增加一个webpack.config.js文件：  然后运行webpack-dev-server命令，顺利启动服务器后，在浏览器中输入http://localhost:8080/index.html就可以看到页面了。[代码下载](https://github.com/xianfeishen/Webpack/tree/master/sample3)  module.exports = {  devtool: 'eval-source-map',  entry: \_\_dirname + "/app/main.js",  output: {  path: \_\_dirname + "/public",  filename: "bundle.js"  },  devServer: {  contentBase: "./public", //以public为根目录提供文件  colors: true,  historyApiFallback: true,  inline: true  }  }; |
| demo | <https://zhuanlan.zhihu.com/p/20367175>  npm install webpack-dev-server --save-dev  安装完毕后 在config中添加配置  module.exports = {  ....  devServer: {  historyApiFallback: true,  hot: true,  inline: true,  progress: true,  },  ...  }  然后再package.json里面配置一下运行的命令,npm支持自定义一些命令  ...  "scripts": {  "start": "webpack-dev-server --hot --inline"  },  ... |
| demo | 实际上 Webpack 有它自己的开发服务器，所以无论你正在开发一个静态网站，或者只是正在原型化前端阶段，这个服务器都是完美可用的。想要运行它，只需要在 webpack.config.js 里添加一个 devServer 对象：  module.exports = {  context: path.resolve(\_\_dirname, './src'),  entry: {  app: './app.js',  },  output: {  filename: '[name].bundle.js',  path: path.resolve(\_\_dirname, './dist/assets'),  publicPath: '/assets', // New  },  devServer: {  contentBase: path.resolve(\_\_dirname, './src'), // New  },  };  现在新建一个 src/index.html 文件，加入下面这行：  <script src="/assets/app.bundle.js"></script>  运行webpack-dev-server，服务器现在就运行在了 localhost:8080 上。注意 script 标签中的 /assets 对应的是 output.publicPath 的值 - 可以随便填成你想要的命名（如果需要一个 CDN，这就很有用了）。  当你更改 JavaScript 代码的时候，Webpack 就会实时更新页面而无需手动刷新浏览器。但是，任何对 webpack.config.js 的更改都需要重启服务器才可以生效。 |
|  | [给vue项目添加ESLint](http://www.cnblogs.com/hahazexia/p/6393212.html) 并 配置说明  <http://www.cnblogs.com/hahazexia/p/6393212.html>    相关配置 |
|  | 自动启动浏览器open-browser-webpack-plugin  var OpenBrowserPlugin = require('open-browser-webpack-plugin');  new OpenBrowserPlugin({ url: 'http://localhost:8080' }); |
|  | 提取公共js插件 webpack.optimize.CommonsChunkPlugin  var webpack = require('webpack');  var commonsPlugin = new webpack.optimize.CommonsChunkPlugin('common.js');  module.exports = {  entry: {  entry1: './entry/entry1.js',  entry2: './entry/entry2.js'  },  output: {  path: \_\_dirname,  filename: '[name].entry.js'  },  resolve: {  extensions: ['', '.js', '.jsx']  },  module: {  loaders: [{  test: /\.js$/,  loader: 'babel-loader'  }, {  test: /\.jsx$/,  loader: 'babel-loader!jsx-loader?harmony'  }  ]  },  plugins: [commonsPlugin]  }; |
|  | HotModuleReplacementPlugin  entry: [  'webpack-dev-server/client?http://localhost:8080',  'webpack/hot/dev-server',  './scripts/index'  plugins: [  new webpack.HotModuleReplacementPlugin(), |
|  | optimize 优化类  UglifyJsPlugin  用途：js压缩  用法： new webpack.optimize.UglifyJsPlugin()  OccurrenceOrderPlugin  用途：排序输出  通过模块调用次数给模块分配ids，常用的ids就会分配更短的id，使ids可预测，减小文件大小，推荐使用  用法： new webpack.optimize.OccurrenceOrderPlugin(true)  CommonsChunkPlugin  用途：合并公共模块为单独文件，比如全局通用的js等，长期不会修改的，从而可以从缓存中取，优化网页性能  用法：  new webpack.optimize.CommonsChunkPlugin({  names: ['vendor']  }) |
|  | 为了创建最高效的Webpack生产版本，需要在生产版本的配置中添加这些插件：  new webpack.DefinePlugin({  'process.env': {  NODE\_ENV: JSON.stringify('production')  }  }),  new webpack.optimize.UglifyJsPlugin() |