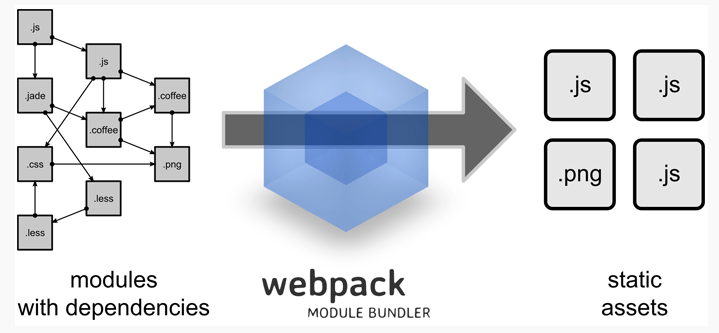
## 一、什么是 Webpack

Webpack 是一个模块打包器。它将根据模块的依赖关系进行静态分析，然后将这些模块按照指定的规则生成对应的静态资源。



## 二、为什么重复造轮子

市面上已经存在的模块管理和打包工具并不适合大型的项目，尤其单页面 Web 应用程序。最紧迫的原因是如何在一个大规模的代码库中，维护各种模块资源的分割和存放，维护它们之间的依赖关系，并且无缝的将它们整合到一起生成适合浏览器端请求加载的静态资源。

这些已有的模块化工具并不能很好的完成如下的目标：

1. 将依赖树拆分成按需加载的块
2. 初始化加载的耗时尽量少
3. 各种静态资源都可以视作模块
4. 将第三方库整合成模块的能力
5. 可以自定义打包逻辑的能力
6. 适合大项目，无论是单页还是多页的 Web 应用

## 三、Webpack的特点和优势

Webapck 和其他模块化工具有什么区别呢？

代码拆分

Webpack 有两种组织模块依赖的方式，同步和异步。异步依赖作为分割点，形成一个新的快。在优化了依赖树后，每一个异步区块都作为一个文件被打包。

Loader

Webpack 本身只能处理原生的 JavaScript 模块，但是 loader 转换器可以将各种类型的资源转换成 JavaScript 模块。这样，任何资源都可以成为 Webpack 可以处理的模块。

智能解析

Webpack 有一个智能解析器，几乎可以处理任何第三方库，无论它们的模块形式是 CommonJS、 AMD 还是普通的 JS 文件。甚至在加载依赖的时候，允许使用动态表达式 require("./templates/" + name + ".jade")。

插件系统

Webpack 还有一个功能丰富的插件系统。大多数内容功能都是基于这个插件系统运行的，还可以开发和使用开源的 Webpack 插件，来满足各式各样的需求。

快速运行

Webpack 使用异步 I/O 和多级缓存提高运行效率，这使得 Webpack 能够以令人难以置信的速度快速增量编译。

## 四、使用

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | 第一步：先要全局安装webpack,再在项目中独立安装webpack  npm install webpack -g  建立项目目录  mkdir webpack\_03 //建立项目文件夹  cd webpack-demo //进入项目文件夹  npm init -y //生成默认package.json文件  npm install --save-dev webpack //项目中安装webpack  如果没全局安装，可以cd到项目目录，命令行执行node\_module/.bin/webpack hello.js hello.bundle.js.  atom ./ //打开当前目录(使用atom打开) |

Note:未全局安装webpack可采用方法

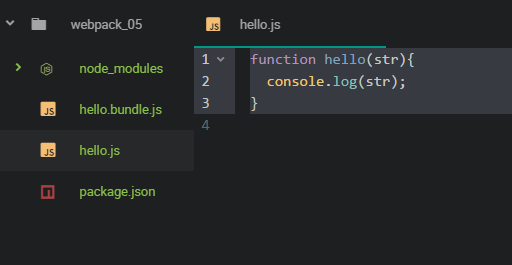
配置npm的package.json文件，在script里面配置上script{start: webpack}，然后在命令行执行npm start(或npm run start),实际上会执行webpack指令,npm会默认加上node\_module/.bin/,和上面一样的效果。

1. 测试webpack

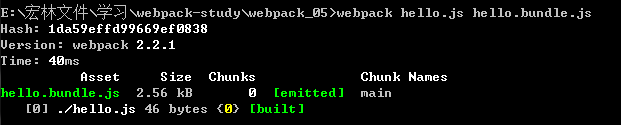
5.1.1文件打包

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | 目录下新建hello.js文件，文件内容：  function hello(str){  console.log(str);  }  在目录下运行打包命令：  webpack hello.js hello.bundle.js //前一个为原始已存在文件，后一个为命令要生成的打包文件 |

5.1.2生成目录结构：



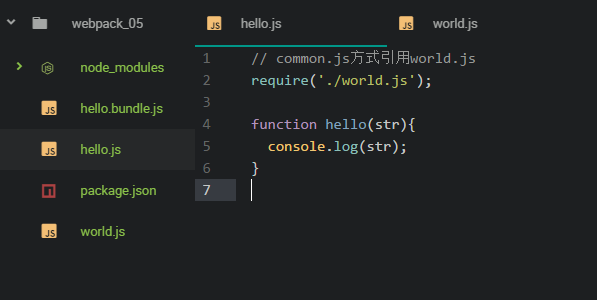
5.1.3打包成功生成信息：

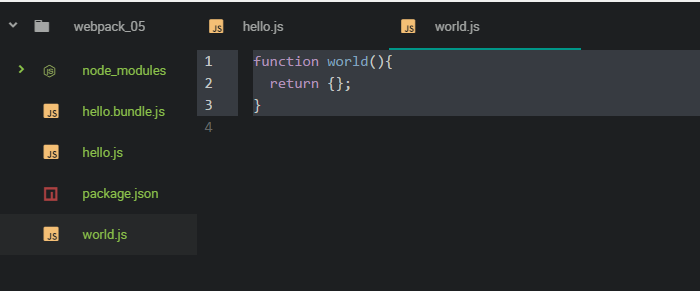


5.2.1文件引用

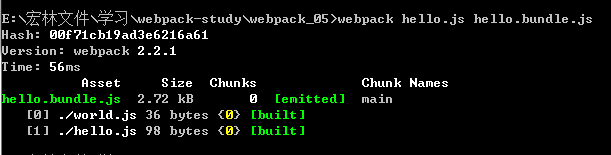
|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | 目录下新建world.js文件，文件内容：  function world(){  return {}; //返回一个空对象  }  在hello.js中以common.js方式引用world.js  + require('./world.js');  function hello(str){  console.log(str);  }  在目录下运行打包命令：  webpack hello.js hello.bundle.js //前一个为原始已存在文件，后一个为命令要生成的打包文件 |

5.2.2生成目录与文件截图





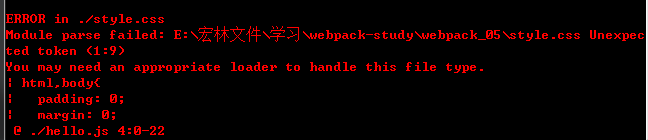
5.2.3打包成功生成信息



5.3 webpack处理css文件类型

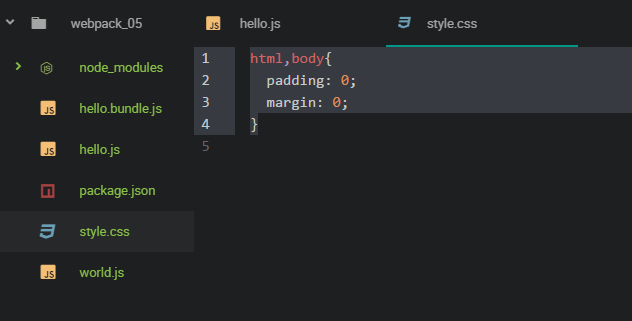
5.3.1Webpack本身不支持处理.css文件类型，需要依赖loader

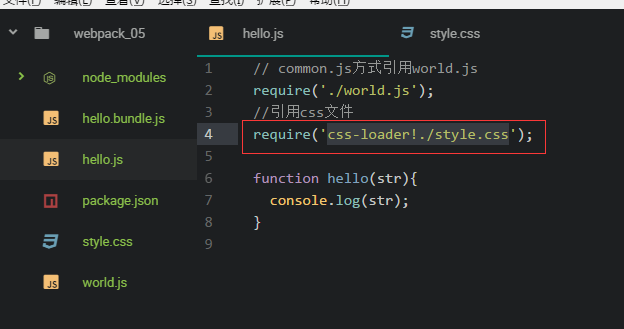
未安装css-loader依赖，直接运行打包命令，会出现以下报错：



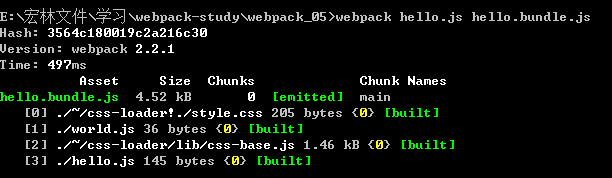
|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | npm install css-loader style-loader --save-dev //在项目目录中安装loader  目录下新建style.css文件，文件内容：  html,body{  padding: 0;  margin: 0;  }  hello.js中引入模块：  //引用css文件  require('css-loader!./style.css'); //在引用style.css前用css-loader先处理，未引用此语法直接进行打包也会出现相应报错：    在目录下运行打包命令：  webpack hello.js hello.bundle.js //前一个为原始已存在文件，后一个为命令要生成的打包文件 |

5.3.2生成目录与文件截图





5.3.3打包成功生成信息

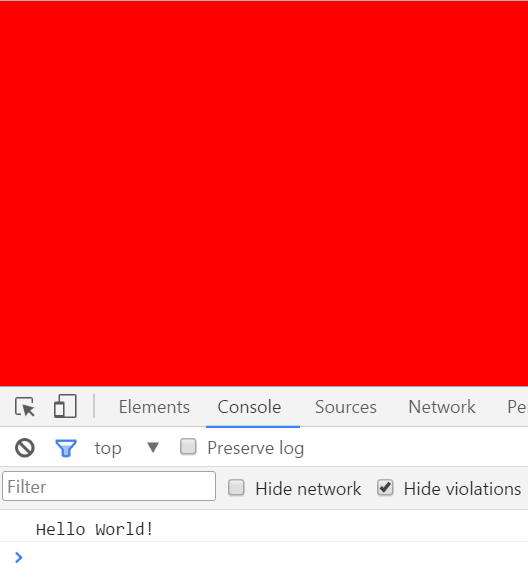


5.4 测试index文件

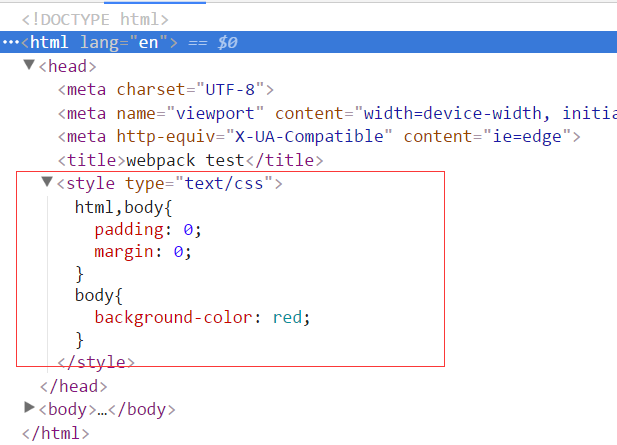
5.4.1生成相关文件

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | 在目录下新建index.html文件  Index.html文件中直接引用打包后的入口文件 hello.bundle.js    为增强效果，在style.css文件中新增样式  body{  background-color: red;  }    Hello.js中新增函数执行语句：  +hello("Hello World!");  require('style-loader!css-loader!./style.css'); //添加style-loader,使样式生效    在目录下运行打包命令：  webpack hello.js hello.bundle.js //前一个为原始已存在文件，后一个为命令要生成的打包文件 |

直接运行index.html文件，背景色生效，控制台输出：



Note:css-loader使webpack能处理.css文件，style-loader处理css-loader处理后的文件，将样式插入到html文件style标签中：



5.4.2 命令行添加loader

如果入口文件中require不引用css-loader与style-loader,可用命令行方式引用处理

入口文件不引用相关loader:



使用命令行引用：

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | webpack hello.js hello.bundle.js --module-bind “css=style-loader!css-loader” //切记此处引用一定用双引号，不然无法识别会报错 |

每一次修改文件都要执行一次这个命令，十分繁琐，可让其文件改变的时候自动更新，自动打包

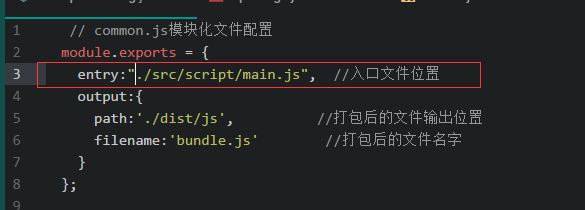
|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | webpack hello.js hello.bundle.js --module-bind “css=style-loader!css-loader” --watch //切记此处引用一定用双引号，不然无法识别会报错    其他参数命令：  webpack hello.js hello.bundle.js --module-bind “css=style-loader!css-loader” --progress //能看到打包过程    webpack hello.js hello.bundle.js --module-bind “css=style-loader!css-loader” --progress --display-modules //看到打包模块过程，模块通过那个loader处理的也会列出来    webpack hello.js hello.bundle.js --module-bind “css=style-loader!css-loader” --progress --display-modules --display-reasons //为什么打包的原因也会列出来    webpack hello.js hello.bundle.js --module-bind “css=style-loader!css-loader” --progress --display-modules --display-reasons --watch //通过命令行工具能够更清晰的打包过程 |

## webpack配置文件

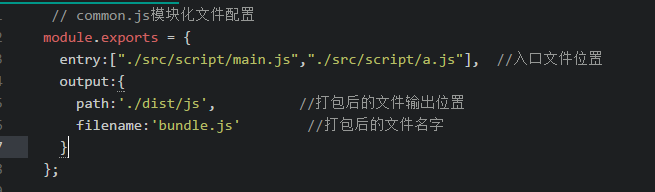
1、在项目目录中建立相关文件目录

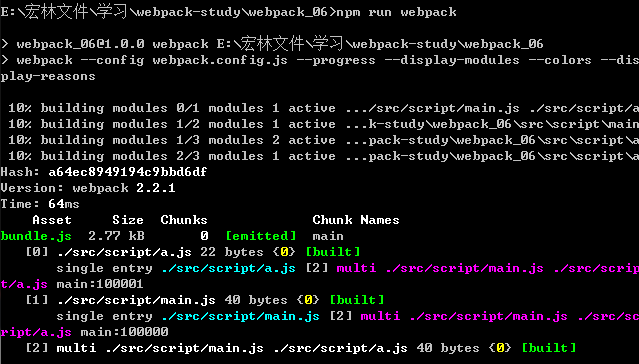
|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | mkdir src //建立源码目录  mkdir dist //建立打包后的静态资源目录  Atom ./ //直接文件夹中打开  建立index.html首页文件，文件中引入打包后的文件：  <!-- 引入打包后的文件 -->  <script src="boundle.js"></script>    项目根目录下配置webpack.config.js文件  // common.js模块化文件配置  module.exports = {  entry:"./src/script/main.js", //入口文件位置  output:{  path:'./dist/js', //打包后的文件输出位置  filename:'bundle.js' //打包后的文件名字  }  };  在script文件夹下建立main.js文件，文件内容：  function helloWord(){  return {};  }  生成目录结构：    脚本窗口中运行webpack命令：  webpack    dist目录下多了相关打包文件：    运行自定义配置文件：  webpack --config webpack.dev.config.js    在package.json文件中配置scripts参数，自定义命令：  {  "name": "webpack\_06",  "version": "1.0.0",  "description": "",  "main": "index.js",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1",  "webpack":"webpack --config webpack.config.js --progress --display-modules --colors --display-reasons"  },  "author": "",  "license": "ISC",  "devDependencies": {  "webpack": "^2.2.1"  }  }    窗口命令执行：  npm run webpack |

1. webpack相关配置文件参数
2. webpack.config.js文件中entry参数：



添加一个两个互不依赖的入口文件（数组）：





传入一个对象

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | // common.js模块化文件webpack.config.js配置  module.exports = {  entry:{  main:"./src/script/main.js",  a:"./src/script/a.js"  },  output:{  path:'./dist/js', //打包后的文件输出位置  filename:'[name].js' //打包后的文件名字  }  };    Note:创建的chunk多于一个，filename的value参数要使用占位符，输出的文件就不会相互覆盖，也不会报错。    不同占位符输出结果：  [name].js  [name]-[hash].js //hash为本次打包的hash  [name]-[chunkhash].js //chunkhash为文件版本号，或者MD5值，每次文件有改动，chunkhash值就会改变 |

1. 自动化生成项目中的HTML文件

动态生成的打包文件名是不确定的，可使用html-webpack-plugin插件解决这个事情

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | npm install html-webpack-plugin --save-dev //使用npm安装这个插件  在webpack.config.js配置文件中配置使用这个插件：  var htmlWebpackPlugin = require('html-webpack-plugin'); //建立对插件的引用  // common.js模块化文件配置  module.exports = {  entry:{  main:"./src/script/main.js",  a:"./src/script/a.js"  },  output:{  path:'./dist/js', //打包后的文件输出位置  filename:'[name]-[hash].js' //打包后的文件名字  },  plugins:[  new htmlWebpackPlugin() //插件初始化  ]  };    运行：  npm run webpack //插件生效    生成效果： |

根目录index.html与打包生成的index.html文件如何关联？

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | 在插件中传参  var htmlWebpackPlugin = require('html-webpack-plugin'); //建立对插件的引用  // common.js模块化文件配置  module.exports = {  entry:{  main:"./src/script/main.js",  a:"./src/script/a.js"  },  output:{  path:'./dist/js', //打包后的文件输出位置  filename:'[name].js' //打包后的文件名字  },  plugins:[  new htmlWebpackPlugin({  template:'index.html'  }) //插件初始化,传参可把根目录index.html与动态打包生成的index.html建立联系  ]  };    根目录下index.html文件改变，且无需引入bundle.js文件：    运行打包命令后，打包文件夹下index.html文件也相应改变： |

Html和js文件分文件夹打包：

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | 修改webpack.config.js配置文件：  var htmlWebpackPlugin = require('html-webpack-plugin'); //建立对插件的引用  // common.js模块化文件配置  module.exports = {  entry:{  main:"./src/script/main.js",  a:"./src/script/a.js"  },  output:{  path:'./dist', //打包后的文件输出位置  filename:'js/[name].js' //打包后的文件名字  },  plugins:[  new htmlWebpackPlugin({  template:'index.html'  }) //插件初始化,传参可把根目录index.html与动态打包生成的index.html建立联系  ]  };    运行打包文件命令后输出结果：    Plugin其他参数：  filename:'index-[hash].html', //文件名  inject:'head', //脚本嵌入位置，可head或body  title:'webpack is good!' //title传值    根目录index.html文件title获值：  <title> <%= htmlWebpackPlugin.options.title %></title> //title中加入模板语言 |

配置插件参数，通过模板语言传参

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | 根目录index.html文件用模板语言获取值：  <%= htmlWebpackPlugin.options.date%>    运行打包命令生成的index.html文件： |

模板语言获取值：

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | 第一步：先要全局安装webpack,再在项目中独立安装webpack  <% for (var key in htmlWebpackPlugin.files) {%>  <%= key %>:<%= JSON.stringify(htmlWebpackPlugin.files[key]) %>  <% } %>  <% for (var key in htmlWebpackPlugin.options) {%>  <%= key %> : <%= JSON.stringify(htmlWebpackPlugin.options[key])%>  <% } %>  // <%= %>直接获取值语法  //<% %> js语法  // JSON.stringify() 将可能产生的对象转化成字符串输出  //使用for(var key in array){ } 遍历输出 |

JS文件分不同区域引入，也可以使用模板语言：

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | Head中引入：  <script type="text/javascript" src="<%= htmlWebpackPlugin.files.chunks.main.entry %>">  Body中引入：  <script type="text/javascript" src="<%= htmlWebpackPlugin.files.chunks.a.entry %>">  webpack.config.js中inject要设置成false:  inject:false, |

其他参数说明：

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | publicPath:'http://cdn.com/' //项目上线后的打包文件的相对地址    结果：    压缩配置：  minify:{  removeComments:true, //删除注释  collapseWhitespace:true //删除空格  }    其他参数详见插件官网说明：  <https://www.npmjs.com/package/html-webpack-plugin> |

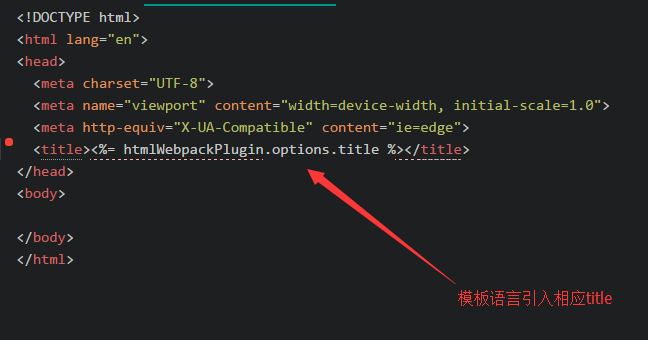
minify使用会报错,先删除

多页面应用：

Webpack.config.js文件配置：



Index.html模板文件删除其他模板语言：



|  |  |
| --- | --- |
| 1. | 执行打包命令：  npm run webpack |

## Loader的使用

要在项目根目录下安装相应的loader：

|  |  |
| --- | --- |
| 1. | 安装babel-loader,将es6语法转化成浏览器能识别语法  npm install --save-dev babel-loader babel-core  三种方式引入babel:   1. 在webpack.config.js中配置：   module:{  loader:[  {  test:/\.js$/,  loader:'babel',  query:{  presets:['latest']  }  }  ]  }     1. package.json中配置：   "babel":{  "presets":["lastest"]  }     1. 根目录下建立 .babelrc 文件，文件内容：   {  "presets": ["env"]  } |

解决babel-loader语法转换时间久，导致打包慢问题：



|  |  |
| --- | --- |
| 1. | exclude:'./node\_modules/',  include:'./src/',    效果：    Exclude参数正确配置方式：  var path = require('path');  exclude:path.resolve(\_\_dirname,'node\_modules'),  include:path.resolve(\_\_dirname,'src')    效果（速度快很多）： |

## 如何处理预处理语言

命令行安装相应loader:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | npm i css-loader style-loader  入口文件中以模块引入css:  import '../css/common.css';  Webpack.config.js中配置loader参数：  context:\_\_dirname,  {  test:/\.css$/,  loader:'style-loader!css-loader'  } |

自动添加不兼容属性的浏览器前缀：

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | 安装loader插件：  npm install postcss-loader --save-dev  安装postcss-loader中的插件autoprefixer:  npm install autoprefixer --save-dev |