webpack学习

一、webpack初始化和打包后的调试

```
npm install production 仅安装生产包
```

webpack初始化

1.新建文件夹 dkmidr webpack_demo。
2.初始化 webpack init 完成后会出现package.json。
3.项目内安装webpack npm install --save-dev webpck 若用webpack打包是出现wenpack不 是内部命令，则需要全局安装webpack。

webpack调试

在配置devtool时，webpack给我们提供了四种选项。

1、source-map:在一个单独文件中产生一个完整且功能完全的文件。这个文件具有最好的source map,但是它会减慢打包速度；

2、cheap-module-source-map:在一个单独的文件中产生一个不带列映射的map，不带列映射提高了打包速度，但是也使得浏览器开发者工具只能对应到具体的行，不能对应到具体的列（符号）,会对调试造成不便。

3、 eval-source-map:使用eval打包源文件模块，在同一个文件中生产干净的完整版的sourcemap，但是对打包后输出的JS文件的执行具有性能和安全的隐患。在开发阶段这是一个非常好的选项，在生产阶段则一定要不开启这个选项。

4、cheap-module-eval-source-map:这是在打包文件时最快的生产source map的方法，生产的 Source map 会和打包后的JavaScript文件同行显示，没有影射列，和eval-source-map选项具有相似的缺点。

四种打包模式，有上到下打包速度越来越快，不过同时也具有越来越多的负面作用，较快的打包速度的后果就是对执行和调试有一定的影响。

个人意见是，如果大型项目可以使用source-map，如果是中小型项目使用eval-source-map就完全可以应对，需要强调说明的是，source map只适用于开发阶段，上线前记得修改这些调试设置。

简单的配置：如

```
module.exports = {
                devtool: 'eval-source-map',
                entry:   __dirname + "/app/main.js",
                output: {
                  path:  __dirname + "/public",
                  filename: "bundle.js"
                }
            }
```

二、打包html

三、webpack.config.js目录结构

```
const path = require('path');
module.exports={
  //入口文件的配置项
    entry:{
        entry:'./src/entry.js'
    },
  //出口文件的配置项
```

```
    output:{
    //输出的路径，用了Node语法
        path:path.resolve(__dirname,'dist'),
    //输出的文件名称
        filename:'bundle.js'
    },
    //模块：例如解读CSS,图片如何转换，压缩
    module:{},
    //插件，用于生产模版和各项功能
    plugins:[],
    //配置webpack开发服务功能
    devServer:{}
}
```

四、配置入口和出口文件

单入口单出口

```
    entry:{
        entry:"./src/entry.js"
    },
    output:{
        path:path.resolve(__dirname,'dist'),
        filename:'bundle.[hash].js'
    },
```

多入口多出口

```
    entry:{
        entry:"./src/entry.js"
        entry2: "./src/entry.js"
    },
    output:{
        path:path.resolve(__dirname,'dist'),
        filename:'static/js/[name].[hash].[ext]'
    //把出口js文件打包到static下的js文件夹中 不写static/js 则是打包到根目录
    },
```

name 出口文件名和入口文件名一致
//ext 后缀名和入口文件一致

五、插件的使用

1、打包html文件的插件，能自动引入出口js文件 和css文件

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

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

```
new HtmlWebpackPlugin({ template:'./index.html'}),
```

2、每次打包清除dist目录

```
npm install --save-dev clean-webpack-plugin
```

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

```
new CleanWebpackPlugin({['dist']})
```

3.使用第三方模块

//使用webpaxk自带插件ProvidePlugin

```
 new webpack.ProvidePlugin({
            axios:"axios"
        })
```

4.压缩js

```
const uglify = require('uglifyjs-webpack-plugin')
new uglify(),
```

六、配置服务和热更新

```
  devServer:{
        //设置基本目录结构
        contentBase:path.resolve(__dirname,'dist'),
        //服务器的IP地址，可以使用IP也可以使用localhost
        host:'localhost',
        //服务端压缩是否开启
        compress:true,
        //配置服务端口号
        port:1717
    }
```

配置完成后直接webpack-dev-server可能会报错，此时，直接在package的script里配置"dev":"webpack-dev-server"，配置完成后直接用npm run dev运行即可！

注意：最近的3.6版本的webpack已经内置热更新不需要特别的配置，较低版本时需要配置插件webpack-hot-middlewar和webpack-dev-middleware

webpack内置的热更新插件，可以直接配置使用！

```
new webpack.HotModuleReplacementPlugin(),
```

六、分离css和其他相关配置

分离css

```
npm install --save-dev extract-text-webpack-plugin
npm install --save-dev style-loader
npm install --save-dev css-loader
const ExtractTextWbpackPlugin = require('extract-text-webpack-
plugin');
```

配置loader

```
{
    test: /\.css$/,
    use: ExtractTextWbpackPlugin.extract({
    fallback: "style-loader",
    use: "css-loader"
```

```
        })
}
```

配置plugins

```
 new ExtractTextWbpackPlugin("static/css/[name].[hash].css"),
```

statci/css 是打包到dist的目录

css自动添加前缀

```
npm install --save-dev post-loader
```

```
npm install --save-dev autoprefixer
```

1、先创建一个与webpcak.config.js同级的postcss.config.js

```
module.exports={
    plugins:[
        require('autoprefixer')
    ]
}
```

postcss的其他牛逼配置在https://github.com/postcss/postcss-loader

2、然后配置lodader，在原先配置的css中添加postcss-loader即可

```
{
    test: /\.css$/,
    use: ExtractTextWbpackPlugin.extract({
         fallback: "style-loader",
         use: [{
                 loader:"css-loader",
                 options: { importLoaders: 1 }
                  },{
                 loader:"postcss-loader"}
                  ]
                })
 },
```

消除未使用的CSS

1.安装PurifyCSS-webpack，purifycss-webpack要以来于purify-css这个包，这两个包都要安装

```
npm install --save-dev purifycss-webpack
```

```
npm install --save-dev purify-css
```

2.webpack.config.js中引用node的glob对象

```
const glob = require("glob")
```

```
const PurifyCSSPlugin = require("purifycss-webpack");
```

3.webpack.config.js里配置plugins

```
 new PurifyCSSPlugin({
    paths: glob.sync(path.join(__dirname, 'src/*.html')),
                    }),
```

//*.html是遍历所用的html文件

注意： 使用这个插件必须配合extract-text-webpack-plugin这个插件
这里配置了一个paths，主要是需找html模板，purifycss根据这个配置会遍历你的文件，查找哪些css被使用了。

七、打包图片和css引入图片中的问题解决

```
npm install --save-dev url-loader
```

```
npm install --save-dev file-loader
```

```
 {
                    test: /\.(png|jpg|gif)/,
```

//正则校验文件格式

```
                    use: [{
                        loader: 'url-loader',
                        options: {
                            limit: 5000,
```

loader后面 limit字段代表图片打包限制，这个限制并不是说超过了就不能打包，而是指当图片大小小于限制时会自动转成 base64 码引用。上例中大于5000字节的图片正常打包，小于5000字节的图片以 base64 的方式引用。

```
                            outputPath: 'static/images/',
```

//打包到dist/static/imgages文件夹中

```
                        }
                    }]
                },
```

问题1：打包html文件中通过<img/>标签引入的图片时，仅配置上述loader，打包不了图片；

解决：需要使用html-withimg-loader，配置完成就可以正常打包了

```
npm install --save-dev html-withimg-loader
```

```
 {
                    test: /\.(htm|html)$/i,
                    use: ['html-withimg-loader']
                }
```

问题2：css中用url引入的背景图，build之后引入路径出现错误

解决：build项目放在生产环境的时候,output处的publicPath改为生产地址，这样所有引入的文件使用的都是绝对路径，引入错误的问题就解决了；

```
 output: {
        path: buildPath,
        filename: 'static/js/[name].[hash].js',
        chunkFilename: "[id].js",
        publicPath: 'http://www.medcircle.cn'
    },
```

八、less和sass的配置

less

```
npm install --save-dev less
npm install --save-dev less-loader
```

其中css-loader和style-loader在配置css的时候已安装过

使用sass与less配置相同，只是loader不同，还有就是sass依赖于ruby，使用sass之前需要在电脑上安装ruby，配置sass的test正则校验使用的是 `/\.scss$/`,需要注意

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

less文件不分离

```
  {
      test: /\.less$/,
       use: [
            {loader: "style-loader"},
            {loader: "css-loader"},
            {loader: "less-loader"}]
//或use: ['style-loader','css-loader','less-loader',]
  }
```

less文件分离

```
    {
       test: /\.less$/,
        use: ExtractTextWbpackPlugin.extract({
            use: [
                {loader: "css-loader"},
                {loader: "less-loader"
            }],
             fallback: "style-loader"
                          })
    }
```

九、增加babel支持

```
npm install --save-dev babel-core babel-loader babel-preset-2015
babel-preset-react
```

方法1、webpack中直接配置

```
   {
       test:/\.(jsx|js)$/,
       use:{
           oader:'babel-loader',
```

```
            options:{
                presets:["es2015","react"]
                        }
            },
        exclude:/node_modules/
    }
```

//include 表示哪些目录中的 .js 文件需要进行 babel-loader

//exclude 表示哪些目录中的 .js 文件不要进行 babel-loader

方法2、.babelrc中配置

虽然Babel可以直接在webpack.config.js中进行配置，但是考虑到babel具有非常多的配置选项，如果卸载webapck.config.js中会非常的雍长不可阅读，所以我们经常把配置卸载.babelrc文件里。

根目录下新建.babelrc文件，配置写在文件中

```
{
    "presets": [
        "react","es2015"
    ]
}
```

然后在webpack.config.js中配置

```
{
   test:/\.(jsx|js)$/,
   use:{
        loader:'babel-loader',
        },
   exclude:/node_modules/
}
```

++

网络上已经不流行babel-preset-es2015，现在官方推荐使用的是babel-preset-env

```
npm install --save-dev babel-preset-env
```

只需要把

```
{
    "presets": [
        "react","es2015"
    ]
}
```

替换为

```
{
    "presets": [
        "react","env"
    ]
}
```

九、打包第三方库

方法1、import直接引用，

例如使用jquery，直接import $ from "jquery;可以在任何文件中引入多次，但是webpack只会打包一次；这种引入不是全局引用；

方法2、webpack全局引用

使用webpack自带的插件ProvidePlugin

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

```
  new webpack.ProvidePlugin({
                axios: "axios",
                $: "jquery"
                    })
```

注意：

- import引入方法：引用后不管你在代码中使用不使用该类库，都会把该类库打包起来，这样有时就会让代码产生冗余。
- ProvidePlugin引入方法：引用后只有在类库使用时，才按需进行打包，所以建议在工作使用插件的方式进行引入。

引入第三方插件的优化

1、在使用第二种方法的同时，修改入口文件

```
  entry: {
        entry: "./src/js/entry.js",
        entry2: "./src/js/entry2.js",
        jquery:'jquery'
    },
```

2.使用webpack自带的插件

```
 new webpack.optimize.CommonsChunkPlugin({
     //name对应入口文件中的名字，我们起的是jQuery
             name:'jquery',
     //把文件打包到哪里，是一个路径
             filename:"static/js/jquery.min.js",
     //最小打包的文件模块数，这里直接写2就好
             minChunks:2
         }),
```

minChunks一般都是固定配置，但是不写是不行的，你会打包失败。
filename是可以省略的，这是直接打包到了打包根目录下，我们这里直接打包到了dist文件夹下边。

抽离多个第三方库的写法如下

```
  entry: {
        entry: "./src/js/entry.js",
```

```
        entry2: "./src/js/entry2.js",
        jquery:'jquery'
        vue:'vue'
    },

 new webpack.optimize.CommonsChunkPlugin({
    //name对应入口文件中的名字，我们起的是jQuery
            name:['jquery','vue'],
    //把文件打包到哪里，是一个路径
            filename:"static/js/[name].min.js",
    //最小打包的文件模块数，这里直接写2就好
            minChunks:2
        }),
```

十、watch的配置

初级使用 --watch

高级：

```
   watchOptions: {
    //检测修改的时间，以毫秒为单位
        poll: 1000,
    //防止重复保存而发生重复编译错误。这里设置的500是半秒内重复保存，不进行打包操作
        aggregeateTimeout: 500,
    //不监听的目录
        ignored: /node_modules/,
    }
```

十一、webpack自动打开浏览器

```
npm install --save-dev open-browser-webpack-plugin
```

```
const OpenBrowserPlugin = require('open-browser-webpack-plugin');
```

```
 new OpenBrowserPlugin(
{
url: 'localhost:8080'
}
),
```

十一、静态资源集中输出

用途：项目中会用很多静态的资源，如图片，文档等，虽然有些可能不用或者以后用，需要把这些资源统一打包到一个文件中，方便以后在线上进行资源的替换

```
npm install --save-dev copy-webpack-plugin
```

```
const copyWebpackPlugin= require("copy-webpack-plugin");
 new copyWebpackPlugin([{
            from:__dirname+'/src/public',
            to:'./static/public'
        }]),
```

- from:要打包的静态资源目录地址，这里的__dirname是指项目目录下，是node的一种语法，可以直接定位到本机的项目目录中。
- to:要打包到的文件夹路径，跟随output配置中的目录。所以不需要再自己加__dirname。