vue-cli中的webpack配置

编辑模式下显示正常，打开的时候不知道为啥排版有问题。
segementfalut链接在这里

版本号
vue-cli 2.8.1 (终端通过vue -V 可查看)

vue 2.2.2

webpack 2.2.1

目录结构

```
├── README.md
├── build
│   ├── build.js
│   ├── check-versions.js
│   ├── dev-client.js
│   ├── dev-server.js
│   ├── utils.js
│   ├── vue-loader.conf.js
│   ├── webpack.base.conf.js
│   ├── webpack.dev.conf.js
│   └── webpack.prod.conf.js
├── config
│   ├── dev.env.js
│   ├── index.js
│   └── prod.env.js
├── index.html
├── package.json
├── src
│   ├── App.vue
│   ├── assets
│   │   └── logo.png
│   ├── components
│   │   └── Hello.vue
│   └── main.js
└── static
```

webpack配置
主要对build目录下的webpack配置做详细分析

```
webpack.base.conf.js
```

入口文件entry

```
entry: {
  app: '.src/main.js'
}
```

输出文件output

config的配置在config/index.js文件中

```
output: {
  path: config.build.assetsRoot, //导出目录的绝对路径
  filename: '[name].js', //导出文件的文件名
  publicPath: process.env.NODE_ENV === 'production'? config.build.a
ssetsPublicPath : config.dev.assetsPublicPath //生产模式或开发模式下htm
l、js等文件内部引用的公共路径
}
```

文件解析resolve

主要设置模块如何被解析。

```
resolve: {
  extensions: ['.js', '.vue', '.json'], //自动解析确定的拓展名,使导入模
块时不带拓展名
  alias: {   // 创建import或require的别名
    'vue$': 'vue/dist/vue.esm.js',
    '@': resolve('src')
  }
}
```

模块解析module

如何处理项目不同类型的模块。

```
module: {
  rules: [
    {
      test: /\.vue$/, // vue文件后缀
      loader: 'vue-loader', //使用vue-loader处理
      options: vueLoaderConfig //options是对vue-loader做的额外选项配置
    },
    {
      test: /\.js$/, // js文件后缀
      loader: 'babel-loader', //使用babel-loader处理
      include: [resolve('src'), resolve('test')] //必须处理包含src和te
st文件夹
    },
    {
      test: /\.(png|jpe?g|gif|svg)(\?.*)?$/, //图片后缀
      loader: 'url-loader', //使用url-loader处理
      query: {  // query是对loader做额外的选项配置
        limit: 10000, //图片小于10000字节时以base64的方式引用
        name: utils.assetsPath('img/[name].[hash:7].[ext]') //文件名
为name.7位hash值.拓展名
      }
```

```
    },
    {
      test: /\.(woff2?|eot|ttf|otf)(\?.*)?$/, //字体文件
      loader: 'url-loader', //使用url-loader处理
      query: {
        limit: 10000,  //字体文件小于1000字节的时候处理方式
        name: utils.assetsPath('fonts/[name].[hash:7].[ext]') //文件名为name.7位hash值.拓展名
      }
    }
  ]
}
```

注：关于query 仅由于兼容性原因而存在。请使用 options 代替。

webpack.dev.conf.js

开发环境下的webpack配置，通过merge方法合并webpack.base.conf.js基础配置

```
var merge = require('webpack-merge')
var baseWebpackConfig = require('./webpack.base.conf')
module.exports = merge(baseWebpackConfig, {})
```

模块配置

```
module: {
  //通过传入一些配置来获取rules配置，此处传入了sourceMap: false,表示不生成sourceMap
  rules: utils.styleLoaders({ sourceMap: config.dev.cssSourceMap })
}
```

在util.styleLoaders中的配置如下

```
exports.styleLoaders = function (options) {
  var output = [] //定义返回的数组，数组中保存的是针对各类型的样式文件的处理方式
  var loaders = exports.cssLoaders(options) // 调用cssLoaders方法返回各类型的样式对象(css: loader)
  for (var extension in loaders) {  //循环遍历loaders
    var loader = loaders[extension] //根据遍历获得的key(extension)来得到value(loader)
    output.push({      //
      test: new RegExp('\\.' + extension + '$'), // 处理的文件类型
      use: loader  //用loader来处理，loader来自loaders[extension]
    })
  }
  return output
}
```

上面的代码中调用了exports.cssLoaders(options),用来返回针对各类型的样式文件的处理方式,具体实现如下

```
exports.cssLoaders = function (options) {
  options = options || {}
```

```
  var cssLoader = {
    loader: 'css-loader',
    options: {  //options是loader的选项配置
      minimize: process.env.NODE_ENV === 'production', //生成环境下压缩文件
      sourceMap: options.sourceMap  //根据参数是否生成sourceMap文件
    }
  }
  function generateLoaders (loader, loaderOptions) {  //生成loader
    var loaders = [cssLoader] // 默认是css-loader
    if (loader) { // 如果参数loader存在
      loaders.push({
        loader: loader + '-loader',
        options: Object.assign({}, loaderOptions, { //将loaderOptions和sourceMap组成一个对象
          sourceMap: options.sourceMap
        })
      })
    }
    if (options.extract) { // 如果传入的options存在extract且为true
      return ExtractTextPlugin.extract({  //ExtractTextPlugin分离js中引入的css文件
        use: loaders,  //处理的loader
        fallback: 'vue-style-loader' //没有被提取分离时使用的loader
      })
    } else {
      return ['vue-style-loader'].concat(loaders)
    }
  }
  return {  //返回css类型对应的loader组成的对象 generateLoaders()来生成loader
    css: generateLoaders(),
    postcss: generateLoaders(),
    less: generateLoaders('less'),
    sass: generateLoaders('sass', { indentedSyntax: true }),
    scss: generateLoaders('sass'),
    stylus: generateLoaders('stylus'),
    styl: generateLoaders('stylus')
  }
}
```

插件配置

```
plugins: [
  new webpack.DefinePlugin({ // 编译时配置的全局变量
    'process.env': config.dev.env //当前环境为开发环境
  }),
  new webpack.HotModuleReplacementPlugin(), //热更新插件
  new webpack.NoEmitOnErrorPlugin(), //不触发错误,即编译后运行的包正常运行
  new HtmlWebpackPlugin({  //自动生成html文件,比如编译后文件的引入
```

```
    filename: 'index.html', //生成的文件名
    template: 'index.html', //模板
    inject: true
  }),
  new FriendlyErrorsPlugin() //友好的错误提示
]
```

webpack.prod.conf.js

生产环境下的webpack配置，通过merge方法合并webpack.base.conf.js基础配置

module的处理,主要是针对css的处理

同样的此处调用了utils.styleLoaders

```
module: {
  rules: utils.styleLoaders({
    sourceMap: config.build.productionSourceMap,
    extract: true
  })
}
```

输出文件output

```
output: {
  //导出文件目录
  path: config.build.assetsRoot,
  //导出的文件名
  filename: utils.assetsPath('js/[name].[chunkhash].js'),
  //非入口文件的文件名，而又需要被打包出来的文件命名配置,如按需加载的模块
  chunkFilename: utils.assetsPath('js/[id].[chunkhash].js')
}
```

插件plugins

```
var path = require('path')
var utils = require('./utils')
var webpack = require('webpack')
var config = require('../config')
var merge = require('webpack-merge')
var baseWebpackConfig = require('./webpack.base.conf')
var CopyWebpackPlugin = require('copy-webpack-plugin')
var HtmlWebpackPlugin = require('html-webpack-plugin')
var ExtractTextPlugin = require('extract-text-webpack-plugin')
var OptimizeCSSPlugin = require('optimize-css-assets-webpack-plugi
n')
var env = config.build.env
plugins: [
  new webpack.DefinePlugin({
    'process.env': env //配置全局环境为生产环境
  }),
  new webpack.optimize.UglifyJsPlugin({ //js文件压缩插件
    compress: {  //压缩配置
      warnings: false  // 不显示警告
    },
    sourceMap: true //生成sourceMap文件
```

```
  }),
  new ExtractTextPlugin({ //将js中引入的css分离的插件
    filename: utils.assetsPath('css/[name].[contenthash].css') //分离出的css文件名
  }),
  //压缩提取出的css，并解决ExtractTextPlugin分离出的js重复问题(多个文件引入同一css文件)
  new OptimizeCSSPlugin(),
  //生成html的插件,引入css文件和js文件
  new HtmlWebpackPlugin({
    filename: config.build.index, //生成的html的文件名
    template: 'index.html', //依据的模板
    inject: true, //注入的js文件将会被放在body标签中,当值为'head'时，将被放在head标签中
    minify: {  //压缩配置
      removeComments: true, //删除html中的注释代码
      collapseWhitespace: true,  //删除html中的空白符
      removeAttributeQuotes: true  //删除html元素中属性的引号
    },
    chunksSortMode: 'dependency' //按dependency的顺序引入
  }),
  //分离公共js到vendor中
  new webpack.optimize.CommonsChunkPlugin({
    name: 'vendor',  //文件名
    minChunks: functions(module, count) { // 声明公共的模块来自node_modules文件夹
      return (module.resource && /\.js$/.test(module.resource) && module,resource.indexOf(path.join(__dirname, '../node_modules')) === 0)
    }
  }),
  //上面虽然已经分离了第三方库,每次修改编译都会改变vendor的hash值，导致浏览器缓存失效。原因是vendor包含了webpack在打包过程中会产生一些运行时代码，运行时代码中实际上保存了打包后的文件名。当修改业务代码时,业务代码的js文件的hash值必然会改变。一旦改变必然会导致vendor变化。vendor变化会导致其hash值变化。
  //下面主要是将运行时代码提取到单独的manifest文件中，防止其影响vendor.js
  new webpack.optimize.CommonsChunkPlugin({
    name: 'mainifest',
    chunks: ['vendor']
  }),
  // 复制静态资源,将static文件内的内容复制到指定文件夹
  new CopyWebpackPlugin([{
    from: path.resolve(__dirname, '../static'),
    to: config.build.assetsSubDirectory,
    ignore: ['.*']  //忽视.*文件
  }])
]
```

额外配置

```
if (config.build.productionGzip) { //配置文件开启了gzip压缩
```

```
  //引入压缩文件的组件,该插件会对生成的文件进行压缩，生成一个.gz文件
  var CompressionWebpackPlugin = require('compression-webpack-plugi
n')

  webpackConfig.plugins.push(
    new CompressionWebpackPlugin({
      asset: '[path].gz[query]', //目标文件名
      algorithm: 'gzip', //使用gzip压缩
      test: new RegExp( //满足正则表达式的文件会被压缩
        '\\.(' +
        config.build.productionGzipExtensions.join('|') +
        ')$'
      ),
      threshold: 10240, //资源文件大于10240B=10kB时会被压缩
      minRatio: 0.8 //最小压缩比达到0.8时才会被压缩
    })
  )
}
```

npm run dev

有了上面的配置之后，下面看看运行命令npm run dev发生了什么

在package.json文件中定义了dev运行的脚本

```
"scripts": {
   "dev": "node build/dev-server.js",
   "build": "node build/build.js"
},
```

当运行npm run dev命令时，实际上会运行dev-server.js文件

该文件以express作为后端框架

```
// nodejs环境配置
var config = require('../config')
if (!process.env.NODE_ENV) {
  process.env.NODE_ENV = JSON.parse(config.dev.env.NODE_ENV)
}
var opn = require('opn') //强制打开浏览器
var path = require('path')
var express = require('express')
var webpack = require('webpack')
var proxyMiddleware = require('http-proxy-middleware') //使用代理的中
间件
var webpackConfig = require('./webpack.dev.conf') //webpack的配置

var port = process.env.PORT || config.dev.port //端口号
var autoOpenBrowser = !!config.dev.autoOpenBrowser //是否自动打开浏览
器
var proxyTable = config.dev.proxyTable //http的代理url
```

```
var app = express() //启动express
var compiler = webpack(webpackConfig) //webpack编译

//webpack-dev-middleware的作用
//1.将编译后的生成的静态文件放在内存中,所以在npm run dev后磁盘上不会生成文件
//2.当文件改变时,会自动编译。
//3.当在编译过程中请求某个资源时，webpack-dev-server不会让这个请求失败，而是会一直阻塞它，直到webpack编译完毕
var devMiddleware = require('webpack-dev-middleware')(compiler, {
  publicPath: webpackConfig.output.publicPath,
  quiet: true
})

//webpack-hot-middleware的作用就是实现浏览器的无刷新更新
var hotMiddleware = require('webpack-hot-middleware')(compiler, {
  log: () => {}
})
//声明hotMiddleware无刷新更新的时机:html-webpack-plugin 的template更改之后
compiler.plugin('compilation', function (compilation) {
  compilation.plugin('html-webpack-plugin-after-emit', function (data, cb) {
    hotMiddleware.publish({ action: 'reload' })
    cb()
  })
})

//将代理请求的配置应用到express服务上
Object.keys(proxyTable).forEach(function (context) {
  var options = proxyTable[context]
  if (typeof options === 'string') {
    options = { target: options }
  }
  app.use(proxyMiddleware(options.filter || context, options))
})

//使用connect-history-api-fallback匹配资源
//如果不匹配就可以重定向到指定地址
app.use(require('connect-history-api-fallback')())

// 应用devMiddleware中间件
app.use(devMiddleware)
// 应用hotMiddleware中间件
app.use(hotMiddleware)

// 配置express静态资源目录
var staticPath = path.posix.join(config.dev.assetsPublicPath, config.dev.assetsSubDirectory)
```

```
app.use(staticPath, express.static('./static'))

var uri = 'http://localhost:' + port

//编译成功后打印uri
devMiddleware.waitUntilValid(function () {
  console.log('> Listening at ' + uri + '\n')
})
//启动express服务
module.exports = app.listen(port, function (err) {
  if (err) {
    console.log(err)
    return
  }
  // 满足条件则自动打开浏览器
  if (autoOpenBrowser && process.env.NODE_ENV !== 'testing') {
    opn(uri)
  }
})
npm run build
由于package.json中的配置，运行此命令后会执行build.js文件

process.env.NODE_ENV = 'production' //设置当前环境为production
var ora = require('ora') //终端显示的转轮loading
var rm = require('rimraf')  //node环境下rm -rf的命令库
var path = require('path')  //文件路径处理库
var chalk = require('chalk')  //终端显示带颜色的文字
var webpack = require('webpack')
var config = require('../config')
var webpackConfig = require('./webpack.prod.conf') //生产环境下的webp
ack配置

// 在终端显示ora库的loading效果
var spinner = ora('building for production...')
spinner.start()

// 删除已编译文件
rm(path.join(config.build.assetsRoot, config.build.assetsSubDirecto
ry), err => {
  if (err) throw err
  //在删除完成的回调函数中开始编译
  webpack(webpackConfig, function (err, stats) {
    spinner.stop() //停止loading
    if (err) throw err

    // 在编译完成的回调函数中,在终端输出编译的文件
    process.stdout.write(stats.toString({
      colors: true,
      modules: false,
```

```
        children: false,
        chunks: false,
        chunkModules: false
      }) + '\n\n')
    })
  })
```