### 1、目录结构

├── .babelrc babel转义配置

├── .editorconfig 编辑格式配置

├── .eslintignore eslint检查忽略的文件配置

├── .eslintrc.js eslint规则配置

├── .gitignore git管理忽略

├── .postcssrc.js postcss配置

├── index.html 页面基础layout

├── package.json npm包管理文档

├── package-lock.json 当前实际环境的npm包文档

├── build webpack打包配置管理

│ ├── build.js 线上打包配置

│ ├── check-versions.js 检车依赖node,npm版本号

│ ├── dev-client.js 本地开发连接客户端

│ ├── dev-server.js 本地开发服务

│ ├── utils 打包常用工具函数

│ ├── vue-loader.conf vue-loader配置

│ ├── webpack.base.conf webpack基础配置

│ ├── webpack.dev.conf webpack dev配置

│ └── webpack.prod.conf webpack prod配置

├── config 配置相关

│ ├── dev.env.js dev环境配置

│ ├── index.js webpack配置文件

│ ├── mock-filter.js mock数据过滤接口配置

│ └── prod.env.js prod环境配置

├── mock mock数据

│ └── user

│ └── listpage.json

├── static 不需要编译涉及的静态资源

│ └── ...

└── src 前端项目主要入口

├── components 公共vue组件

│ └── breadcrumb

│ └── index.vue

├── imgs 公共图片资源

│ └── element-logo

├── router vue路由相关

│ └── index.js

├── store vuex相关

│ └── index.js

├── styles 公共样式相关

│ └── index.sass

├── views 页面相关

│ └── 页面名称

│ ├── components 页面级组件

│ │ └── index.vue

│ ├── imgs 页面级imgs

│ │ └── xxx.jpg

│ └── 页面.vue

├── app.vue vue页面layout

└── main.js vue打包入口

### .babelrc

Babel配置文件，一个转码器，可以使用ES6的语法写代码，通过babel转码为Es5的代码实现兼容性，从而在现有环境执行

[https://babeljs.cn](https://babeljs.cn/docs/usage/babelrc/)

### .editorconfig

统一代码风格使用，如定义换行，缩进

<https://www.cnblogs.com/xiaohuochai/p/7160067.html>

### .eslintignore

管理eslint不做代码检查的文件

### .eslintrc.js

eslint代码检查规则配置

基于VUE的规则：https://blog.csdn.net/qq\_30817073/article/details/78043553

普通JS规则：<http://eslint.cn/>

### .gitignore

git管理忽略，该文档内描述的目录或文件将不会上传到git中

### .postcssrc.js

postcss配置，利用JS插件来对CSS进行转换的工具，Autoprefixer就是众多PostCSS插件中最流行的一个

<https://segmentfault.com/a/1190000011595620>

### index.html

页面基础layout，这里定义了所有页面输出的公共html

### package.json

一个json格式的文件，主要是npm的配置文件，主要管理javascript包，作用相当于maven的.pom文件

{

name ： 当前npm包名，

version ： 当前npm版本号，Name+Version组合形成唯一的标识，

description ： 当前npm的描述，

author ： 作者，

private ： 是否私有，如果设置true，则该包不会发布到npm中，

scripts ： 由脚本命令组成的字典，这些命令运行在包的各个生命周期中，如配置中的 dev ： node build/dev-server.js 当运行npm run dev时，实际就是执行的node build/dev-server.js命令，

dependencies：指定了依赖的包名和其版本范围的映射，该处描述的应该是发布需要的依赖，如果是本地开发需要的依赖，请放在devDependencies中，

devDependencies ：如果有人计划在他们的项目中下载和使用你的模块，但他们可能并不想或并不需要你开发所使用的外部测试和文档框架。在这种情况下，最好将这些附加的项放在devDependencies中，如代码检查工具eslint，

engines ： node运行需要的版本信息，

browserslist ： 浏览器兼容的信息

}

其他内容参考：https://www.cnblogs.com/nullcc/p/5829218.html

### package-lock.json

在package.json中配置依赖的时候，不一定是精确到小版本号的依赖，如^0.16.2，表示兼容0.16.2版本，而npm实际下载时，就会根据规则下载完整版本号的依赖，如0.16.7，而package-lock.json就是记录依赖对应的完整版本号

### build

webpack打包需要的配置

#### 、build.js

线上打包需要的配置，实际正式环境的编译，就是执行的node build/build.js命令

require('./check-versions')()

引入检查版本模块，具体含义可参见[check-version.js](#_、check-versions.js)

process.env.NODE\_ENV = 'production'

设置当前环境为生产环境

**var** ora = require('ora')

ora，一个可以在终端显示spinner的插件，可以在命令行窗口显示提示信息

**var** rm = require('rimraf')

rm，用于删除文件或文件夹的插件，每次build之前先删除

**var** path = require('path')

node提供的操作文件的帮助类，一般用于获取当前路径位置等

**var** chalk = require('chalk')

chalk，用于在控制台输出带颜色字体的插件，比如错误信息用红色字体显示

**var** webpack = require('webpack')

webpack打包工具

**var** config = require('../config')

引入config文件里的配置内容，具体含义可参见[10、config](#_config)

**var** webpackConfig = require('./webpack.prod.conf')

引入webpack生产环境配置，具体含义参见[webpack.prod.conf](#_、webpack.prod.conf.js)

**var** spinner = ora('building for production...')  
// 开启loading动画  
spinner.start()

显示启动中的动画

rm(path.join(config.build.assetsRoot, config.build.assetsSubDirectory), err => {  
 **if** (err) **throw** err  
 // 执行webpack构建打包，完成之后在终端输出构建完成的相关信息或者输出报错信息并退出程序  
 webpack(webpackConfig, **function** (err, stats) {  
 spinner.stop()  
 **if** (err) **throw** err  
 process.stdout.write(stats.toString({  
 colors: **true**,  
 modules: **false**,  
 children: **false**,  
 chunks: **false**,  
 chunkModules: **false** }) + '\n\n')  
  
 console.log(chalk.cyan(' Build complete.\n'))  
 console.log(chalk.yellow(  
 ' Tip: built files are meant to be served over an HTTP server.\n' +  
 ' Opening index.html over file:// won\'t work.\n'  
 ))  
 })  
})

根据config获取发布目录，删除该目录下已发布的内容，使用webpack根据webpackConfig配置内容进行打包，如果打包过程中有错误，则通过chalk插件，显示有颜色的错误或警告信息

#### 、check-versions.js

检查当前环境node及npm的版本是否满足package.json中配置的需求版本

**var** chalk = require('chalk')

chalk，用于在控制台输出带颜色字体的插件，比如错误信息用红色字体显示

**var** semver = require('semver')

引入的是一个语义化版本文件的npm包，其实它就是用来控制版本的。“semver”的常用方法如下：

semver.valid('1.2.3') // '1.2.3'，返回符合版本格式的版本

semver.valid('a.b.c') // null，如果不符合版本返回null

semver.clean(' =v1.2.3 ') // '1.2.3'，返回一个标准的版本号，且去掉两边的空格

semver.satisfies('1.2.3', '1.x || >=2.5.0 || 5.0.0 - 7.2.3') // true，第一个参数是测试的版本号，第二个参数是匹配的版本，如果匹配则返回true

semver.gt('1.2.3', '9.8.7') // false，第一个参数的版本是否比第二个大

semver.lt('1.2.3', '9.8.7') // true ，第一个参数的版本您是否比第二个小

参考：https://npm.taobao.org/package/semver

**var** packageConfig = require('../package.json')

读取npm包配置，用来获取node及npm的版本

**var** shell = require('shelljs')

shelljs, 执行Unix命令行的插件

**function** exec (cmd) {  
 **return** require('child\_process').execSync(cmd).toString().trim()  
}

定义方法，开辟子进程执行指令cmd并返回结果

**var** versionRequirements = [  
 {  
 name: 'node',  
 currentVersion: semver.clean(process.version),  
 versionRequirement: packageConfig.engines.node  
 },  
]

定义node和npm版本需求变量

**if** (shell.which('npm')) {  
 versionRequirements.push({  
 name: 'npm',  
 currentVersion: exec('npm --version'),  
 versionRequirement: packageConfig.engines.npm  
 })  
}

如果npm命令存在，则调用子进程返回npm的版本信息，将结果添加到versionRequirements中（node和npm版本需求变量）

参考：https://www.npmjs.com/package/shelljs

module.exports = **function** () {  
 **var** warnings = []  
 // 依次判断版本是否符合要求  
 **for** (**var** i = 0; i < versionRequirements.length; i++) {  
 **var** mod = versionRequirements[i]  
 **if** (!semver.satisfies(mod.currentVersion, mod.versionRequirement)) {  
 warnings.push(mod.name + ': ' +  
 chalk.red(mod.currentVersion) + ' should be ' +  
 chalk.green(mod.versionRequirement)  
 )  
 }  
 }  
 // 如果有警告则将其输出到控制台  
 **if** (warnings.length) {  
 console.log('')  
 console.log(chalk.yellow('To use this template, you must update following to modules:'))  
 console.log()  
 **for** (**var** i = 0; i < warnings.length; i++) {  
 **var** warning = warnings[i]  
 console.log(' ' + warning)  
 }  
 console.log()  
 process.exit(1)  
 }  
}

分别检查node和npm的当前版本，是否匹配需求版本（package.json中配置的engines），如果版本不匹配，则通过chalk显示错误信息

#### 、dev-client.js

开发时使用，无刷新热部署客户端，在webpack开发编译配置文件webpack.dev.conf.js中引用，当修改代码后，无需刷新即可通知页面显示最新代码，与dev-server上的热重载插件之间需要进行双向通信，服务端webpack重新编译后，会向客户端推送信息，告诉客户端进行更新

require('eventsource-polyfill')

加载eventsource-polyfill，polyfill解决API的兼容问题的，eventsource就是解决事件源兼容问题的

**var** hotClient = require('webpack-hot-middleware/client?noInfo=true&reload=true')

webpack热重载客户端

hotClient.subscribe(**function** (event) {  
 **if** (event.action === 'reload') {  
 window.location.reload()  
 }  
})

客户端收到更新动作，执行页面刷新

#### 、dev-server.js

开发时使用，当编译开发环境时，实际就是执行的node build/dev-server.js命令

该包的主要作用：

1. 根据mock规则启用本地mock
2. 根据代理配置规则启用代理
3. 启用热部署
4. 打开浏览器

require('./check-versions')()

检查node和npm的版本，具体含义可参见[check-version.js](#_、check-versions.js)

**var** config = require('../config')

获取配置信息，具体含义可参见[10、config](#_config)

**if** (!process.env.NODE\_ENV) {  
 process.env.NODE\_ENV = JSON.parse(config.dev.env.NODE\_ENV)  
}

如果node无法判断当前是开发环境还是生产环境，则使用config.dev.env.NODE\_ENV作为当前的环境，具体含义参见[dev.env.js](#_、dev.env.js)

**var** opn = require('opn')

打开浏览器

参考：<https://blog.csdn.net/xmloveth/article/details/58350496>

**var** path = require('path')

node提供的操作文件的帮助类，一般用于获取当前路径位置等

**var** express = require('express')

express框架 本地开发服务器

参考：<http://www.runoob.com/nodejs/nodejs-express-framework.html>

**var** webpack = require('webpack')

webpack打包工具

**var** utils = require('./utils')

引入帮助类，具体含义参见[utils.js](#_、utils.js)

**var** proxyMiddleware = require('http-proxy-middleware')

proxy插件可以将前端开发中涉及到的请求代理到提供服务的后台服务器上，方便与服务器对接

参见：<https://blog.csdn.net/xmloveth/article/details/56847456>

**var** webpackConfig = require('./webpack.dev.conf')

开发环境下的webpack配置，具体含义参见[webpack.dev.conf](#_、webpack.dev.conf.js)

**var** port = process.env.PORT || config.dev.port

dev-server 监听的端口，如果没有在命令行传入端口号，则使用config.dev.port设置的端口，例如8080，具体含义参见[config/index.js](#_、index.js)

**var** autoOpenBrowser = !!config.dev.autoOpenBrowser

用于判断是否要自动打开浏览器的布尔变量，当配置文件中没有设置自动打开浏览器的时候其值为 false，具体含义参见[config/index.js](#_、index.js)

**var** proxyTable = config.dev.proxyTable

HTTP代理表，指定规则，将某些API请求代理到相应的服务器，具体含义参见[config/index.js](#_、index.js)

**var** app = express()

创建express服务器

**var** compiler = webpack(webpackConfig)

webpack根据配置开始编译打包源码并返回compiler对象

**var** devMiddleware = require('webpack-dev-middleware')(compiler, {  
 // 设置访问路径为webpack配置中的output里面所对应的路径  
 publicPath: webpackConfig.output.publicPath  
})

/webpack-dev-middleware将webpack编译打包后得到的产品文件存放在内存中而没有写进磁盘， 将这个中间件挂到express上使用之后即可提供这些编译后的产品文件服务

参见：<https://segmentfault.com/a/1190000011761306>

**var** hotMiddleware = require('webpack-hot-middleware')(compiler, {  
 log: () => {}  
})

用于实现热重载功能的中间件

参见：<https://segmentfault.com/a/1190000011761345>

compiler.plugin('compilation', **function** (compilation) {  
 compilation.plugin('html-webpack-plugin-after-emit', **function** (data, cb) {  
 hotMiddleware.publish({ action: 'reload' })  
 cb()  
 })  
})

webpack(重新)编译打包完成后并将js、css等文件inject到html文件之后，通过热重载中间件强制页面刷新

参见：<https://www.cnblogs.com/cqhaibin/p/6581308.html>

app.use(utils.mock)

启用本地mock

Object.keys(proxyTable).forEach(**function** (context) {  
 **var** options = proxyTable[context]  
 **if** (**typeof** options === 'string') {  
 options = { target: options }  
 }  
 app.use(proxyMiddleware(options.filter || context, options))  
})

根据 proxyTable 中的代理请求配置来设置express服务器的http代理规则

app.use(devMiddleware)

挂载webpack-dev-middleware中间件，提供webpack编译打包后的产品文件服务

app.use(hotMiddleware)

挂载热重载中间件

**var** staticPath = path.posix.join(config.dev.assetsPublicPath, config.dev.assetsSubDirectory)  
app.use(staticPath, express.static('./static'))

提供static文件夹上的静态文件服务

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

访问链接默认localhsot

**var** \_resolve  
**var** readyPromise = **new** Promise(resolve => {  
 \_resolve = resolve  
})

创建promise，在应用服务启动之后resolve，便于外部文件require了这个dev-server之后的代码编写

promise参见：<https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Using_promises>

console.log('> Starting dev server...')

devMiddleware.waitUntilValid(() => {  
 console.log('> Listening at ' + uri + '\n')  
 // when env is testing, don't need open it  
 **if** (autoOpenBrowser && process.env.NODE\_ENV !== 'testing') {  
 opn(uri)  
 }  
 \_resolve()  
})

webpack-dev-middleware等待webpack完成所有编译打包之后输出提示语到控制台，表明服务正式启动，服务正式启动才自动打开浏览器进入页面

**var** server = app.listen(port)

启动express服务器并监听相应的端口

module.exports = {  
 ready: readyPromise,  
 close: () => {  
 server.close()  
 }  
}

暴露本模块的功能给外部使用，例如下面这种用法

var devServer = require('./build/dev-server')

devServer.ready.then(() => {...})

if (...) { devServer.close() }

#### 、utils.js

常用工具包

**var** path = require('path')

node提供的操作文件的帮助类，一般用于获取当前路径位置等

**var** config = require('../config')

获取配置信息，具体含义可参见[10、config](#_config)

**var** ExtractTextPlugin = require('extract-text-webpack-plugin')

压缩css，该插件的主要是为了抽离css样式,防止将样式打包在js中引起页面样式加载错乱的现象

参考：<https://www.cnblogs.com/EnSnail/p/6927211.html>

exports.assetsPath = **function** (\_path) {  
 **var** assetsSubDirectory = process.env.NODE\_ENV === 'production'  
 ? config.build.assetsSubDirectory  
 : config.dev.assetsSubDirectory  
 **return** path.posix.join(assetsSubDirectory, \_path)  
}

根据当前环境变量来分别决定资源文件存放路径，如果是开发环境，则读取config.dev.assetsSubDirectory，具体含义参见[dev.env.js](#_、dev.env.js)，如果是生成环境，则读取config.build.assetsSubDirectory，具体含义参见[prod.env.js](#_、prod.env.js)

exports.cssLoaders = **function** (options) {  
 options = options || {}  
 // css-loader配置  
 **var** cssLoader = {  
 loader: 'css-loader',  
 options: {  
 // 是否最小化  
 minimize: process.env.NODE\_ENV === 'production',  
 // 是否使用source-map  
 sourceMap: options.sourceMap  
 }  
 }  
  
 // 生成各种loader配置，并且配置了extract-text-pulgin  
 **function** generateLoaders (loader, loaderOptions) {  
 // 默认是css-loader  
 **var** loaders = [cssLoader]  
 // 如果非css，则增加一个处理预编译语言的loader并设好相关配置属性  
 // 例如generateLoaders('less')，这里就会push一个less-loader  
 // less-loader先将less编译成css，然后再由css-loader去处理css  
 // 其他sass、scss等语言也是一样的过程  
 **if** (loader) {  
 loaders.push({  
 loader: loader + '-loader',  
 options: Object.assign({}, loaderOptions, {  
 sourceMap: options.sourceMap  
 })  
 })  
 }  
  
 // 配置extract-text-plugin提取样式  
 **if** (options.extract) {  
 **return** ExtractTextPlugin.extract({  
 use: loaders,  
 fallback: 'vue-style-loader'  
 })  
 } **else** {  
 // 无需提取样式则简单使用vue-style-loader配合各种样式loader去处理<style>里面的样式  
 **return** ['vue-style-loader'].concat(loaders)  
 }  
 }  
  
 // 得到各种不同处理样式的语言所对应的loader  
 **return** {  
 css: generateLoaders(),  
 postcss: generateLoaders(),  
 less: generateLoaders('less'),  
 sass: generateLoaders('sass', { indentedSyntax: **true** }),  
 scss: generateLoaders('sass'),  
 stylus: generateLoaders('stylus'),  
 styl: generateLoaders('stylus')  
 }  
}

生成css、sass、scss等各种用来编写样式的语言所对应的loader配置，生成各种loader配置，并且配置了extract-text-pulgin，具体代码含义可以参见上方注释

loader的解释参见：<http://www.css88.com/doc/webpack/concepts/loaders/>

exports.styleLoaders = **function** (options) {  
 **var** output = []  
 **var** loaders = exports.cssLoaders(options)  
 **for** (**var** extension **in** loaders) {  
 **var** loader = loaders[extension]  
 output.push({  
 test: **new** RegExp('\\.' + extension + '$'),  
 use: loader  
 })  
 }  
 **return** output  
}

生成处理单独的.css、.sass、.scss等样式文件的规则

**const** fs = require('fs');

node内置的文件模块

**const** vm = require('vm');

VM模块是NodeJS里面的核心模块，支撑了require方法和NodeJS的运行机制

exports.mock = **function**(req, res, next){  
 // 支持热更新  
 **let** mockFilter = parseConfig(path.join(\_\_dirname, '../config/mock-filter.json'));  
  
 **const** url = req.originalUrl;  
 // 如果配置略过mock, 或者关闭所有mock则直接跳过  
 **if**(mockFilter['IS\_CLOSE\_ALL\_MOCK'] || mockFilter[url]) **return** next();  
  
 **const** mockPath = path.join(\_\_dirname, '../mock', url + '.json');  
 **const** data = parseConfig(mockPath);  
 **try** {  
 **if**(JSON.stringify(data) !== '{}'){  
 res.json(data);  
 **return**;  
 }  
 } **catch** (error) {}  
  
 next();  
};

mock数据的中间件，根据[config/mockfilter.json](#_、mockfilter.json)的配置，来决定路由是走本地mock，还是服务器，支持全局开关及单个请求的开关

**function** parseConfig (filePath) {  
 **let** content;  
 **try** {  
 content = fs.readFileSync(filePath);  
 } **catch** (e) {  
 **return** {};  
 }  
 **const** code = '(' + content + ')';  
 **const** sandbox = {};  
 **try** {  
 **return** vm.runInThisContext(code, sandbox, {  
 filename: filePath,  
 displayErrors: **false**,  
 timeout: 1000  
 });  
 } **catch** (e) {  
 console.error(e);  
 **return** {};  
 }  
}

使用vm执行代码，解释json文件，将js文件内容，转换为JSON变量

#### 、vue-loader.conf.js

处理.vue文件中的样式，在[webpack.base.conf.js](#_、webpack.base.conf.js)中被调用

**var** utils = require('./utils')

node提供的操作文件的帮助类，一般用于获取当前路径位置等

**var** config = require('../config')

引入config中的配置

**var** isProduction = process.env.NODE\_ENV === 'production'

是否生产环境

module.exports = {  
 // 处理.vue文件中的样式  
 loaders: utils.cssLoaders({  
 sourceMap: isProduction  
 ? config.build.productionSourceMap  
 : config.dev.cssSourceMap,  
 // 是否提取样式到单独的文件  
 extract: isProduction  
 })  
}

调用帮助类的方法加载loader

#### 、webpack.base.conf.js

webpack基本配置，或者叫公共配置，不论开发环境，生产环境，都要使用该配置合并

**var** path = require('path')

node提供的操作文件的帮助类，一般用于获取当前路径位置等

**var** utils = require('./utils')

加载帮助包

**var** config = require('../config')

加载config配置

**var** vueLoaderConfig = require('./vue-loader.conf')

加在vue配置

**function** resolve (dir) {  
 **return** path.join(\_\_dirname, '..', dir)  
}

获取绝对路径

entry: {  
 // webpack入口文件  
 app: './src/main.js'  
}

定义webpack的入口文件，webpack打包时，会解析依赖树，根据依赖关系，打包文件，所以必须指定一个入口文件，告诉webpack，从哪个文件开始解析

webpack参见：<http://www.css88.com/doc/webpack/concepts/entry-points/>

output: {  
 // webpack输出的目标文件夹路径（例如：/dist）  
 path: config.build.assetsRoot,  
 // webpack输出bundle文件命名格式  
 filename: '[name].js',  
 // webpack编译输出的发布路径（例如部署到cdn就可以设置'//cdn.xxx.com/app/'）  
 publicPath: process.env.NODE\_ENV === 'production'  
 ? config.build.assetsPublicPath  
 : config.dev.assetsPublicPath  
},

webpack输出路径和命名规则

resolve: {  
 // 别名，方便引用模块，例如有了别名之后  
 extensions: ['.js', '.vue', '.json'],  
 alias: {  
 // fix vue的package.json里面的main的指引  
 'vue$': 'vue/dist/vue.esm.js',  
 '@': resolve('src')  
 }  
},

模块resolve的规则

module: {  
 rules: [  
 {// 对所有.vue文件使用vue-loader进行编译  
 test: /\.vue$/,  
 loader: 'vue-loader',  
 options: vueLoaderConfig  
 },  
 {// 对src和test文件夹下的.js和.vue文件使用eslint-loader进行代码规范检查  
 test: /\.js$/,  
 loader: 'babel-loader',  
 include: [resolve('src'), resolve('test')]  
 },  
 {// 对图片资源文件使用url-loader  
 test: /\.(png|jpe?g|gif|svg)(\?.\*)?$/,  
 loader: 'url-loader',  
 options: {  
 // 小于10K的资源转成base64编码的dataURL字符串写到代码中  
 limit: 10000,  
 // 其他的资源转移到静态资源文件夹  
 name: utils.assetsPath('img/[name].[hash:7].[ext]')  
 }  
 },  
 {// 对字体资源文件使用url-loader  
 test: /\.(woff2?|eot|ttf|otf)(\?.\*)?$/,  
 loader: 'url-loader',  
 options: {  
 // 小于10K的资源转成base64编码的dataURL字符串写到代码中  
 limit: 10000,  
 // 其他的资源转移到静态资源文件夹  
 name: utils.assetsPath('fonts/[name].[hash:7].[ext]')  
 },

{// 编译时，对代码规范进行检查

test: /\.(js|vue)$/,

loader: 'eslint-loader',

enforce: 'pre',

include: [resolve('src'), resolve('test')],

options: {

formatter: require('eslint-friendly-formatter'),

// 不符合Eslint规则时只警告(默认运行出错)

// emitWarning: !config.dev.showEslintErrorsInOverlay

}

}  
 }  
 ]  
}

符合规则test（正则表达式）的文件使用对应的loader进行处理，其中最后一项是编译时，对.js和.vue文件使用eslint-loader进行代码检查，不符合代码规范的，编译不通过

#### 、webpack.dev.conf.js

webpack开发环境配置

**var** utils = require('./utils')

引入帮助工具包

**var** webpack = require('webpack')

webpack打包工具

**var** config = require('../config')

引入config配置，具体含义参见[10、config](#_config)

**var** merge = require('webpack-merge')

webpack-merge是一个可以合并数组和对象的插件，用于混合webpack配置，这里的配置分为基本配置[webpack.base.conf.js](#_、webpack.base.conf.js)，当开发环境的时候混合基本配置和本配置，当生产环境的时候混合基本配置和[webpack.prod.conf.js](#_、webpack.prod.conf.js)

**var** baseWebpackConfig = require('./webpack.base.conf')

引入webpack基本配置

**var** HtmlWebpackPlugin = require('html-webpack-plugin')

将webpack编译打包后的产品文件注入到html模板中，即自动在index.html里面加上<link>和<script>标签引用webpack打包后的文件

参见：<https://segmentfault.com/a/1190000007294861>

**var** FriendlyErrorsPlugin = require('friendly-errors-webpack-plugin')

更友好地输出webpack的警告、错误等信息

参见：<http://npm.taobao.org/package/friendly-errors-webpack-plugin>

Object.keys(baseWebpackConfig.entry).forEach(**function** (name) {  
 baseWebpackConfig.entry[name] = ['./build/dev-client'].concat(baseWebpackConfig.entry[name])  
})

给每个入口页面(应用)加上[dev-client](#_、dev-client.js)，用于跟[dev-server](#_、dev-server.js)的热重载插件通信，实现热更新

module.exports = merge(baseWebpackConfig, {  
 module: {  
 // 样式文件的处理规则，对css/sass/scss等不同内容使用相应的styleLoaders  
 // 由utils配置出各种类型的预处理语言所需要使用的loader，例如sass需要使用sass-loader  
 rules: utils.styleLoaders({ sourceMap: config.dev.cssSourceMap, usePostCSS: **true** })  
 },  
 // 打包规则 详细查看https://www.webpackjs.com/configuration/devtool/#devtool  
 devtool: '#cheap-module-eval-source-map',  
 plugins: [  
 // webpack插件  
 **new** webpack.DefinePlugin({  
 'process.env': config.dev.env  
 }),  
 // 开启webpack热更新功能  
 **new** webpack.HotModuleReplacementPlugin(),  
 // webpack编译过程中出错的时候跳过报错阶段，不会阻塞编译，在编译结束后报错  
 **new** webpack.NoEmitOnErrorsPlugin(),  
 // 自动将依赖注入html模板，并输出最终的html文件到目标文件夹  
 **new** HtmlWebpackPlugin({  
 filename: 'index.html',  
 template: 'index.html',  
 inject: **true** }),  
 **new** FriendlyErrorsPlugin()  
 ]  
})

定义当前配置，合并基础配置和定义当前配置

#### 、webpack.prod.conf.js

### config

#### 、dev.env.js

开发环境变量配置包

**var** merge = require('webpack-merge')

引入webpack混合包

**var** prodEnv = require('./prod.env')

加载开发环境变量配置，具体含义参见[prod.env.js](#_、prod.env.js)

module.exports = merge(prodEnv, {  
 NODE\_ENV: '"development"'  
})

将开发环境变量配置中的NODE环境变量改为development（设置当前环境为开发环境）

#### 、index.js

主配置文件，文件内部分别定义了生产环境和开发环境需要的配置

**var** path = require('path')

node提供的操作文件的帮助类，一般用于获取当前路径位置等

build: {  
 // 环境变量  
 env: require('./prod.env'),  
 // html入口文件打包后之后的文件名称和格式  
 index: path.resolve(\_\_dirname, '../dist/index.html'),  
 // 打包文件的生产路径  
 assetsRoot: path.resolve(\_\_dirname, '../dist'),  
 // 二级目录，存放静态资源文件的目录，位于dist文件夹下  
 assetsSubDirectory: 'static',  
 // 发布路径，如果构建后的产品文件有用于发布CDN或者放到其他域名的服务器，可以在这里进行设置例如(//cdn.xxx.com)  
 // 设置之后构建的产品文件在注入到index.html中的时候就会带上这里的发布路径  
 assetsPublicPath: '/',  
 // 是否使用source-map  
 productionSourceMap: **true**,  
 // 是否开启gzip压缩  
 productionGzip: **false**,  
 // gzip模式下需要压缩的文件的扩展名，设置js、css之后就只会对js和css文件进行压缩  
 productionGzipExtensions: ['js', 'css'],  
 // 是否展示webpack构建打包之后的分析报告  
 // 如果需要请执行`npm run build --report`  
 bundleAnalyzerReport: process.env.npm\_config\_report  
}

生产环境配置，其中重要的参数有：assetsRoot打包后生成的路径，assetsSubDirectory静态资源文件的目录，位于dist文件夹下，assetsPublicPath发布目录可使用前缀域名，如xxx.com，设置之后构建的产品文件在注入到index.html中的时候就会带上这里的发布路径，其余的可参见上方代码注释

dev: {  
 // 环境变量  
 env: require('./dev.env'),  
 // dev-server监听的端口  
 port: 8080,  
 // 是否自动打开浏览器  
 autoOpenBrowser: **true**,  
 // 静态资源文件夹  
 assetsSubDirectory: 'static',  
 // 发布路径  
 assetsPublicPath: '/',  
 // 代理配置表，在这里可以配置特定的请求代理到对应的API接口  
 proxyTable: {  
 '/v2': {  
 target: 'https://api.douban.com',  
 changeOrigin: **true** }  
 },  
 // 是否开启 cssSourceMap  
 cssSourceMap: **false**}

开发环境配置，重要的参数有：port端口，assetsSubDirectory静态资源目录，assetsPublicPath发布路径，proxyTable代理配置表，所有符合配置规则的，可以代理到不同的服务器上，如这里配置的，所有请求后方有/v2的，全部代理到https://api.douban.com这个服务器上

#### 、mockfilter.json

mock路由开关配置

{  
 "IS\_CLOSE\_ALL\_MOCK":**false**, // 是否关闭所有的mock  
 "/api/demo":**true**}

IS\_CLOSE\_ALL\_MOCK 全局开关，当为true时，所有的请求都走服务器，当该值为false时，再查找下面请求的白名单,如api.demo这个请求就是走服务端，其他的走本地mock

#### 、prod.env.js

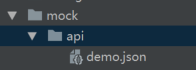
生产环境变量配置包

module.exports = {  
 NODE\_ENV: '"production"'  
}

设置当前环境为生产环境

### mock

本地模拟数据使用，目录名/json文件名，对应请求的路由，例如api/demo，则在mock目录下创建api目录，在api目录下创建demo.json文件，如图：



demo.json中就是返回的json数据

{  
 "success":**true**,  
 "code":0,  
 "data":1  
}

### static

静态资源存放目录，发布时会将该目录下的文件 copy到assetsPublicPath中的assetsSubDirectory目录中

### src

开发源代码目录，具体规则参见《开发篇》