## 6 Source Maps 调试

打包后的文件很难找到出错的位置，利用source maps可以解决。

devtool的配置如下：

devtool: 'eval-source-map',

可选项如下：

source-map：

在一个单独的文件中产生一个完整且功能完全的文件，会减慢打包速度；

cheap-module-source-map：

在一个单独的文件中生成一个不带列映射的map，提高了打包速度，但是浏览器开发 者工具只能对应到具体的行，不能对应到具体的列（符号）

eval-source-map：

使用eval打包源文件模块，在同一个文件中生成干净的完整的source map。这个选项 可以在不影响构建速度的前提下生成完整的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 定不要启用这个选项；

cheap-module-eval-source-map：

这是在打包文件时最快的生成source map的方法，生成的Source Map 会和打包后的 JavaScript文件同行显示，没有列映射，和eval-source-map选项具有相似的缺点；

官方文档中常用关键字的解释：

eval： 使用eval包裹模块代码

source-map： 产生.map文件

cheap： 不包含列信息，也不包含loader的sourcemap

module： 包含loader的sourcemap（比如jsx to js ，babel的sourcemap）

inline： 将.map作为DataURI嵌入，不单独生成.map文件（这个配置项比较少见）

eval模式是使用eval将webpack中每个模块包裹，然后在模块末尾添加模块来源//# souceURL， 依靠souceURL找到原始代码的位置。

“在cheap-source-map模式下sourcemap不包含列信息，也不包含loaders的sourcemap”这里的“column-mappings”就是代码列数的意思



# 一 开发环境

## 1.1 前端开发服务器介绍

代码都是运行在服务器上的，如果前端拥有自己的本地开发服务器，更能准确的测试代码的正确率，同样也能解决跨域等困扰前端的测试问题。

webpack提供了三种方式来搭建前端开发服务器：

方式一：watch参数

webpack提供了 -watch 参数 来监听文件，（可以简写为-w），使用该命令打包后，webpack将启动一个内部的服务器，监听文件改变，每次改变都会触发打包。

方式二：（推荐）webpack-dev-server

webpack将启动一个内部服务器，自动打包，且向外提供一系列配置，如端口、定向、https、接口代理、模块热更新、自动刷新浏览器、浏览器显示编译错误等完善的设置。

## 1.2 webpack-dev-server

### 1.2.1 webpack-dev-server作用

由于该模块是webpack官方的，安装后直接在配置中，添加 devServer字段即可。

主要作用：

开启一个本地服务器监控文件变化，自动刷新浏览器；

可以实现热更新打包；

可以实现请求代理：比如ajax的测试地址是：test.\*\*\*.com，上线后是api.\*\*\*.com。那么可以在代码中直接书写api.\*\*\*.com，让webpack-dev-server转发到test上；

安装：npm i webpack-dev-server

### 1.2.2 webpack-dev-server参数

inline： 模式，默认true，false时浏览器会显示打包进度

contentBase：指内容提供路径，默认为当前工作目录

port： 监听端口

historyApiFallback：指定redirect规则，比如前端路由使用#，支持正则路由

https： 默认false，设为true时可以自动生成证书

proxy： 生成代理，用于ajax代理

hot： 热更新，默认false

open: 打包后是否打开网页，默认为true，默认打开localhost

openPage： 指定初识打开页面

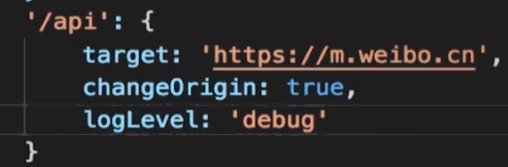
lazy： 只打包访问到的资源，多页面需要配置，增加打包速度

overlay： 在打开的页面中提供一个遮罩，遮罩中显示错误页面

### 1.2.3 proxy与ajax

proxy配置，比如接口地址为：<http://www.test.com/api/getOrderLists>

proxy配置如下：



在项目根目录创建多个webpack配置文件

（1）webpack.base.config.js //公用的配置（通过插件继承）

（2）webpack.dev.config.js //开发环境中用到的配置文件

（3）webpack.prod.config.js //生产环境中用到的配置文件

由于跨域问题，本地文件无法发送ajax，webpack可以将本地文件的ajax进行转发：

## 2 webpack-dev-server

使用原生的webpack命令，每次修改完代码都要重新打包，需要服务器来实现自动化：

npm i webpack-dev-server -D

注意：该工具和webpack的用法完全一样，由于只有全局安装的工具才能在终端使用，该命令本地安装后不能直接使用，需要在package.json文件中配置一个命令：

"dev": "webpack-dev-server --open --port 3000 --contentBase src --hot"

命令与参数解释：

webpack-dev-server 在 localhost:8080 建立一个 Web 服务器

--devtool eval 为代码创建源地址，报错时精确地定位到文件和行号

--progress 显示合并代码进度

--colors 命令行中显示颜色！

--content-base 默认webpack-dev-server会为根文件夹提供本地服务器，设置了该目 录，则从该目录提供本地服务

--open 自动打开浏览器

--inline 设置为true，当源文件改变时会自动刷新页面

--historyApiFallback依赖于HL5 history API，设置为true，在开发单页应用时所有的跳转 将指向index.html

注意： 打包生成的bundle.js文件位于内存中，可以认为位置是和dist src平级！

所以html的引入方式也要相应修改为 /bundle.js

脚本+配置文件方式实现上述命令：

"dev": "webpack-dev-server --mode development"

配置文件webpack.config.js内容：

const webpack = require('webpack');

const path = require('path');

module.exports = {

entry: path.resolve(\_\_dirname, 'src/index.js'),

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'bundle.js'

},

devServer:{

hot: true,

open: true,

port: 3000,

inline: true

}

}

*//设置hot为ture后才能开启热更新插件*

new webpack.HotModuleReplacementPlugin();

# 二 第三方库引用

场景一：

通过npm安装jquery，或者通过远程地址引入的jquery，利用providePlugin插件：

安装jquery：npm install jquery -S

配置$:

new webpack.ProvidePlugin({

$: 'jquery',

jQuery: 'jquery'

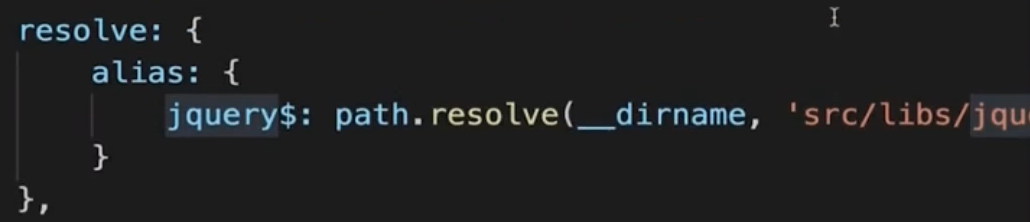
})

在index.js中直接使用jquery：

$('#box').css("background-color","red");

场景二：

通过本地目录引入的方式，在上述配置中，当文件使用jquery时候会去node\_modules中查找jquery模块，但是jquery是在本地，所以需要额外告知webpack去哪查找：



qjeury$带$的意思是解析到文件下，而不是解析到目录。

当然也可以使用import-loader 替代上述插件。

# 三 抽离与压缩

## 2.1抽离公共JS webpack4版

场景一：有个公用JS文件module.js同时被pageA.js pageB.js引用，打包后，需要将该公共JS文件也被打包为一个文件。

上述无需配置，是webpack4的默认行为，不过需要注意，默认配置只有多入口文件才可以打包公共代码。

optimization: {

splitChunks: {

cacheGroups: {

commons: {

name: "module",

chunks: "initial",

minChunks: 2 *//出现多少次将会被打包*

}

}

}

},

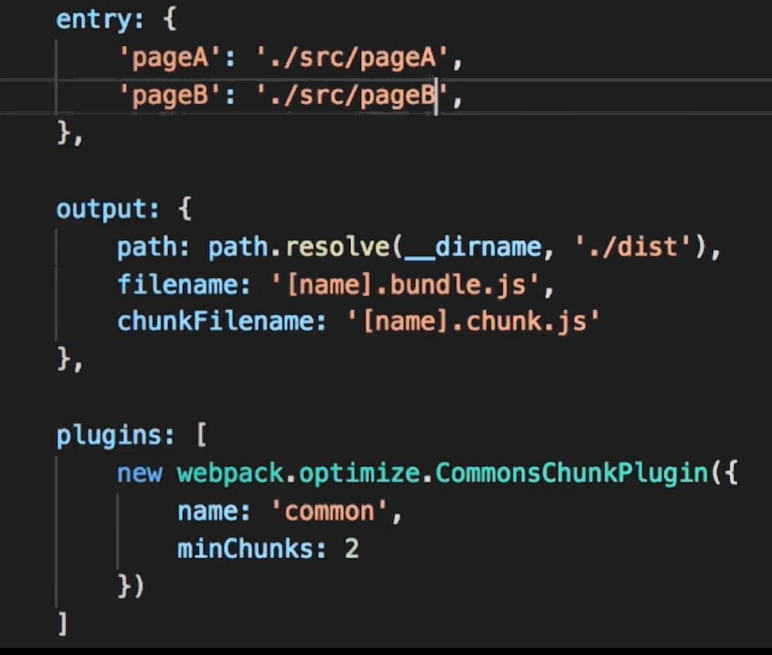
minChunks:出现多少次会被打包，不打包则直接写入pageA.js这样的文件中。

场景二：

## 2.2 抽离公共JS webpack3版

场景一：

一个js文件被多次引用，希望被被打包在一个公共的JS文件中；



name：打包后的额外生成的文件名

minChunks：出现多少次将会被打包为额外的JS

场景二：在一个js文件中引入了jquery，那么打包时，jquery也被打包进了bundle文件中，这不是我们想要的，我们需要的是第三方的包都放在类似libs这样的文件夹中。

首先在入口处配置：

**entry**: {  
 **app** : path.resolve(\_\_dirname,**'src/js/app.js'**),  
 **vendors**:[**'jquer'**]  
},

插件处配置为：

**plugins**: [**new** webpack.**optimize**.CommonsChunkPlugin({

**name**: **'vendors'**,

**filename**: **'vendors.js'**

}),  
]

这时候我们在dist文件中的index.html文件添加：

<**script src="vendors.js"**></**script**>

执行命令：npm run publish

直接右键打开html页面就可以看到部署后的效果。vebdors.js就是被部署的入口文件。

注意：如果vendors.js引入顺序错误，会报错：

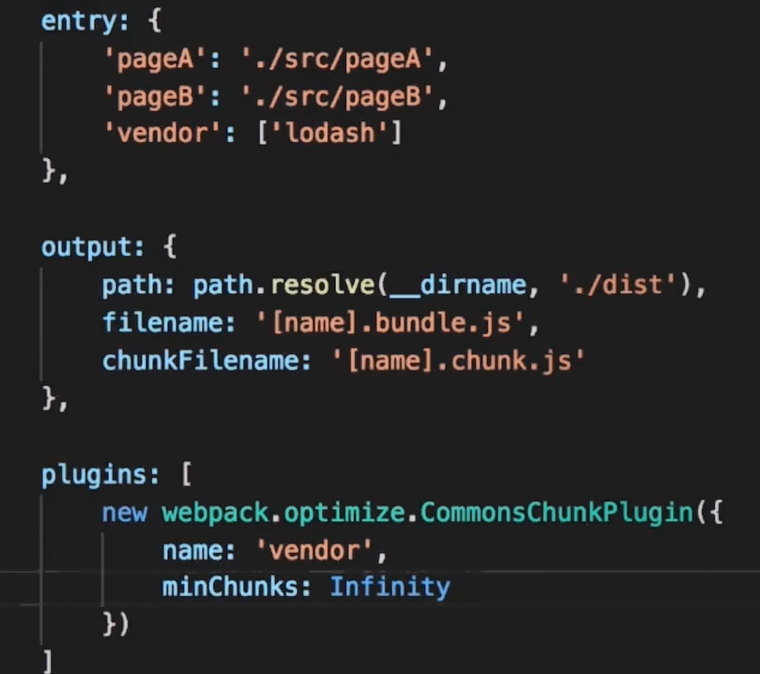
Uncaught ReferenceError: webpackJsonp is not defined。

因为我们需要调整index文件内JS的顺序如下：

<script src=”vendors.js”></script>

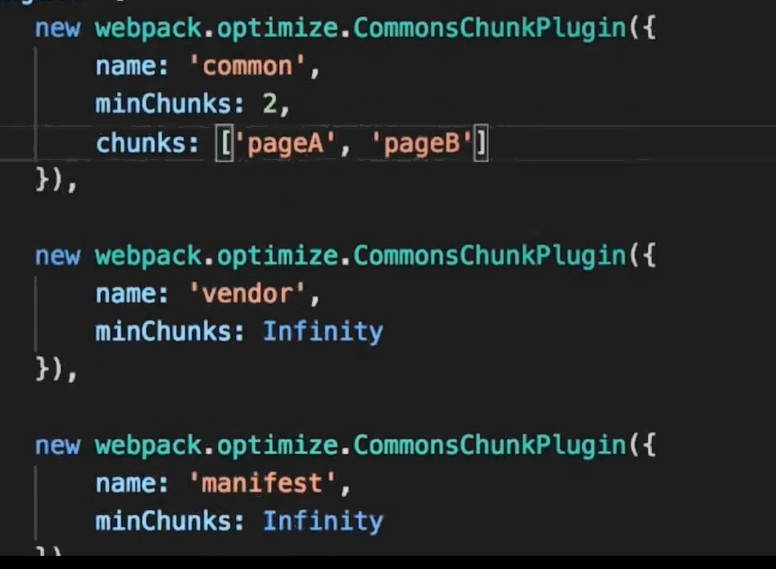
<script src=”bundle.js”></script>

场景三：如果我们希望jquery与自己的公共代码都被打包到同一个文件中：

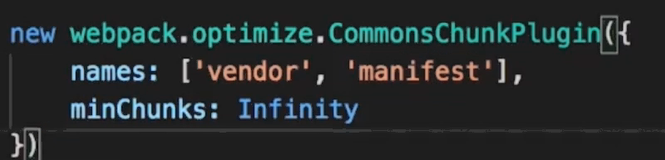


那么打包后，lodash包以及开发者自己的公共JS都被打包到了vendor.bundle.js中

场景：如果项目中有 pageA.js pageB.js 二者同时引用了loadsh，且内部也有公共代码，我们不希望公共代码和lodash打包在一起，想要将公共代码打包为一个文件，lodash本身打包为一个文件：

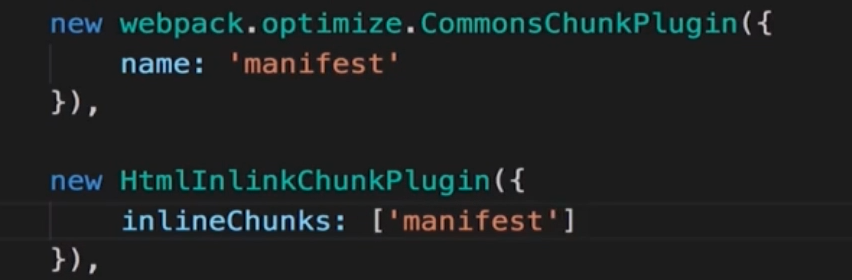


注意：图上的后2个插件可以写为：



## 1.2 页面src引入文件的方式，改成用script标签嵌入的方式，减少http请求( 提高加载性能）每个页面加载的时候，都会去加载公共代码，这里值得优化。我们可以引入的script文件直接作为代码插入html中，则可以解决。

利用插件：html-webpack-inline-chunk-plugin



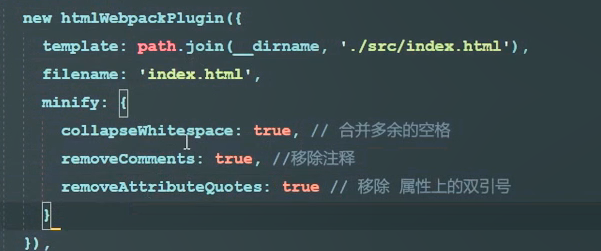
## 代码分割与异步加载

## 1.3 JS压缩



但是webpack4不需要上述第二行的插件配置

## 1.4 html压缩



Webpack4指定了生产环境后，无需上述配置

## 1.5 抽取CSS



加上css/前缀表示公共css放置位置

## 1.6 压缩抽取出来的css

