# 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'

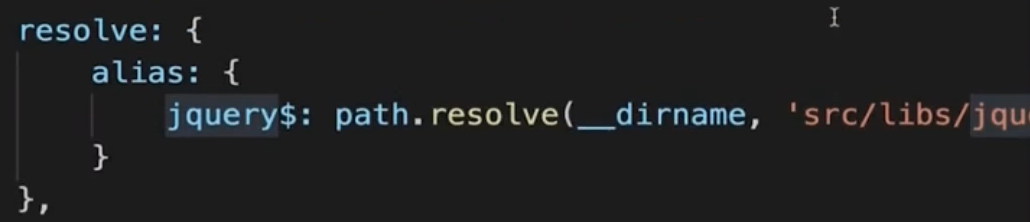
})

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*

}

}

}

},

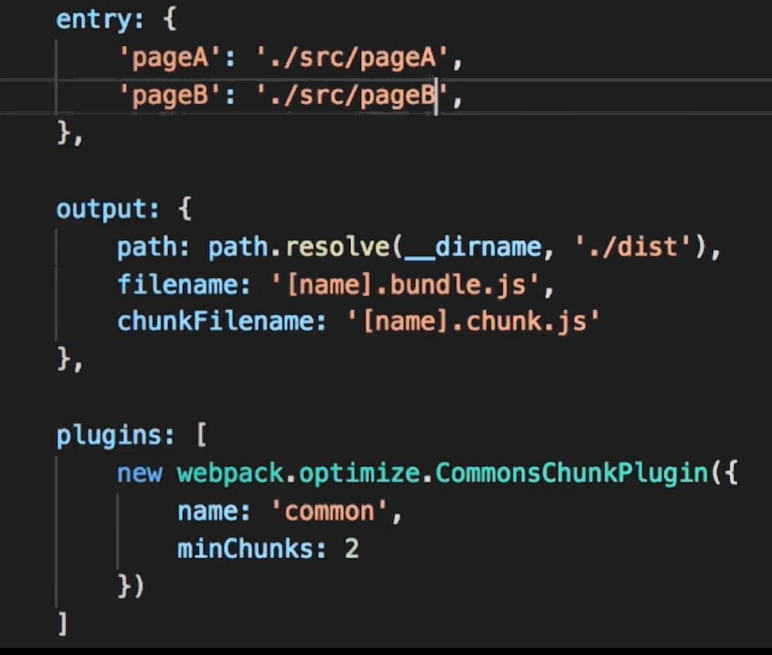
minChunks:出现多少次会被打包，不打包则直接写入pageA.js这样的文件中。

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：

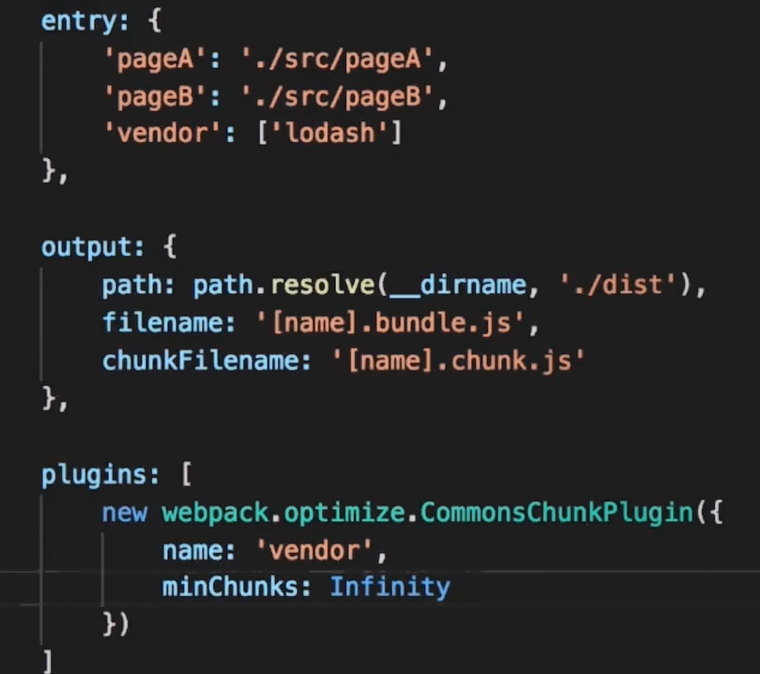
Uncaught ReferenceError: webpackJsonp is not defined。

因为我们需要调整index文件内JS的顺序如下：

<script src=”vendors.js”></script>

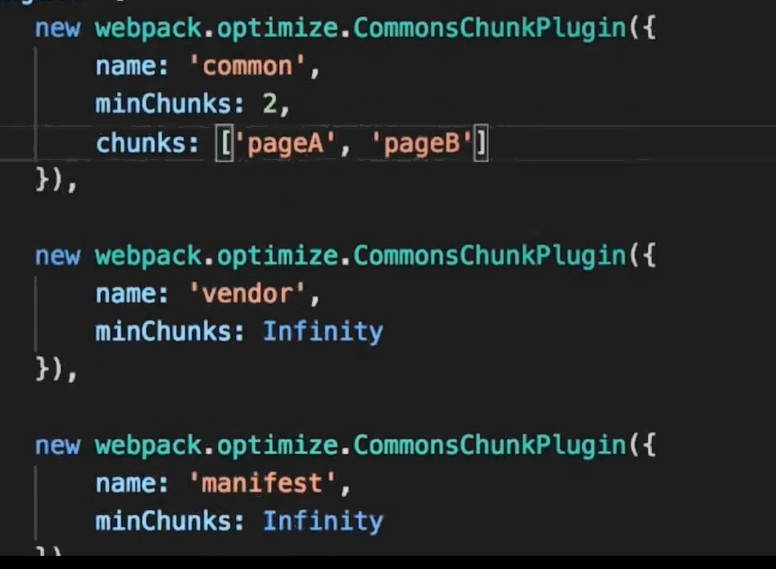
<script src=”bundle.js”></script>

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：

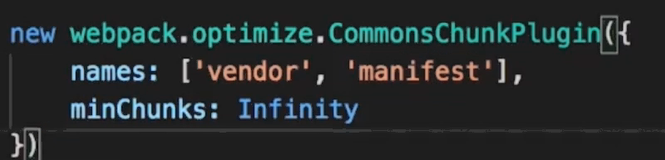


那么打包后，lodash包以及开发者自己的公共JS都被打包到了vendor.bundle.js中

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：

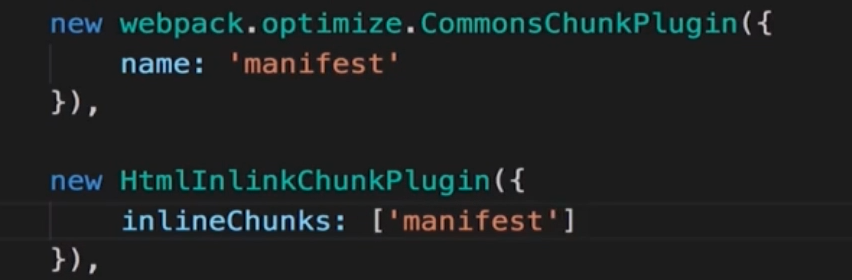


注意：图上的后2个插件可以写为：



## 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



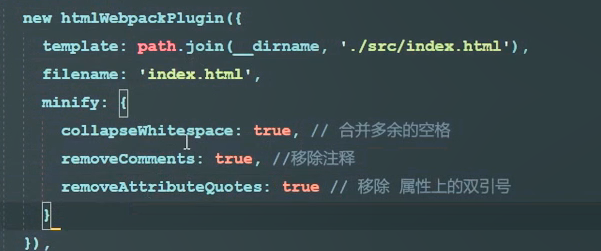
## 代码分割与异步加载

## 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



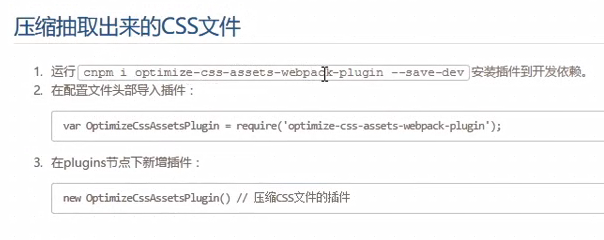
Webpack4指定了生产环境后，无需上述配置

## 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



## 1 缓存

### 1.1生成缓存

缓存无处不在，使用缓存的最好方法是保证你的文件名和文件内容是匹配的（内容改变，名称相应改变）

webpack可以把一个哈希值添加到打包的文件名中，使用方法如下,添加特殊的字符串混合体（[name], [id] and [hash]）到输出文件名前

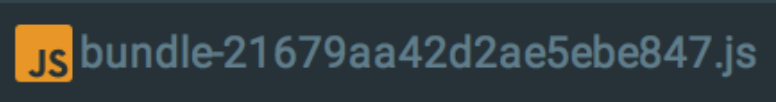
output: {

path: \_\_dirname + "/build",

filename: "bundle-[hash].js"

}

现在用户会有合理的缓存了。



### 1.2 去除多余缓存文件 clean-webpack-plugin

添加了hash之后，会导致改变文件内容后重新打包时，文件名不同而内容越来越多，使用插件：npm i -Dl clean-webpack-plugin

new CleanWebpackPlugin('build/\*.\*', {

root: \_\_dirname,

verbose: true,

dry: false

})

## 2 Resolve属性

webpack在构建包的时候会按目录进行文件的查找，resolve属性中的extensions数组用于配置程序可以自行补全哪些文件后缀，这样require的文件就不需要再书写后缀名了：

resolve: {

root: '/src', //绝对路径

extensions: ['', '.js', '.json', '.scss',’jsx’], //空字符串对应无需后缀名

alias: { //给模块定义一个别名

AppStore : 'js/stores/AppStores.js', //使用时 require('AppStore')

ActionType : 'js/actions/ActionType.js',

}

}

## 3 Externals属性

外部依赖不需要打包进bundle，当我们想在项目中require一些其他的类库或者API，而又不想让这些类库的源码被构建到运行时文件中，这在实际开发中很有必要。

比如：你在页面里通过script标签引用了jQuery：

<script src="//code.jquery.com/jquery-1.12.0.min.js"></script>，所以并不想在其他js里再打包进入一遍，比如你的其他js代码类似：



其作用类似 部署时移除依赖包的插件。

其实就是不是通过require或者import引入的，而是直接写在html中的js地址，但是这里有个问题：没有引入react，那么构建完毕后，如何运行呢

答案：使用cdn来直接在html中引入



## 4 noParse属性



## 5 重点：多文件入口

<http://fakefish.github.io/react-webpack-cookbook/Multiple-entry-points.html>



https://blog.csdn.net/DeepLies/article/details/79005507?utm\_source=blogxgwz3

## 6 重点Chunk

http://webpack.github.io/docs/code-splitting.html

## 7 懒加载

（1）在react中如何使用

<http://fakefish.github.io/react-webpack-cookbook/Lazy-loaded-entry-points.html>

（2）在react-router中用到动态加载路由可以实现

# webpack优化：

## 问题

首先，我们先大致看下我们都有什么问题，然后一步步进行解决

* 项目频繁进行修改，冗余文件过多
* 部分第三方依赖滥用，想去除但是不知道在哪个文件中。或没用，但是遗留在package.json里，
* 项目庞大，打包的结果过大，时间过长

## 删除冗余文件

由于项目的频繁改动，有很多文件已经不被使用并且没有被删除。由于项目的不断扩大，只会影响我们定位功能和问题的速度，因此对冗余文件进行清理，是很重要的。但是我们单凭肉眼很难识别哪个文件是否被依赖的，因此还要通过webpack来解决。

### 1.获取项目依赖的所有文件

我们来看一下webpack的输出文件格式:

{

...

chunks: [{

name: 'chunk-name',

modules: [

// 每个chunk中所有的依赖文件

]

}]

...

}

所以说，根据这个stats.json，我们可以拿到在整个项目中拿到的所有项目文件:

/\*\*

\* 查询依赖的模块

\*/**function** **findSrcModules** () {

**return** **new** Promise((resolve, reject) => {

fs.readFile(statPath, (err, data) => {

**if** (err) **return**

**const** json = JSON.parse(data)

**const** assetsList = json.chunks

**let** ret = []

// 拿到所有chunk的所有依赖文件

assetsList.forEach(chunk => {

**const** modules = chunk.modules.map(item => item.name)

ret = ret.concat(modules)

})

// 去除node\_modules中的文件

ret = ret.filter(item => item.indexOf('node\_modules') < 0)

resolve(ret)

})

})

}

通过这一步，我们可以拿到项目中，所有打包依赖的文件。

### 2.获取项目中所有的文件

通过glob,我们可以获取所有的文件:

**function** **getAllFilesInSrc** () {

**const** pattern = './src/\*\*'

**return** **new** Promise((resolve, reject) => {

glob(pattern, {

nodir: true

}, (err, files) => {

**const** ret = files.map(item => {

**return** item.replace('./src', '.')

})

resolve(ret)

})

})

}

### 3.将两个文件数组进行对比，然后进行删除等操作：

将两个数组进行对比，没有出现在依赖中的文件，就是冗余文件。我们可以一键删除

findSrcModules().then(ret => {

getAllFilesInSrc().then(allFiles => {

**const** unUsed = allFiles.filter(item => {

**return** ret.indexOf(item) < 0

})

**const** join = p => path.join('./src', p)

unUsed.forEach(file => {

shelljs.rm(join(file))

})

})

})

## 分析第三方依赖

根据上述冗余文件的思路，我们同样可以对第三方依赖进行处理，大致思路如下

1. 获取所有包含node\_modules的依赖
2. 将文件名进行截取、去重。获取到所有的依赖
3. 与package.json进行对比，拿到没有使用的依赖
4. 将对比结果进行分析，将不想使用的依赖保存下来
5. 再次查找stat.json，查找该依赖的reson字段，获取再哪里引用了该依赖，进行输出
6. 将依赖进行手动替换、删除等操作

可以说，拿到了所有依赖及依赖关系，我们可以很灵活的对其进行处理，拿到我们想要的结果。

该功能后续也会更新到[webpack-unused-files](https://github.com/callmedadaxin/webpack-unused-files)中去。

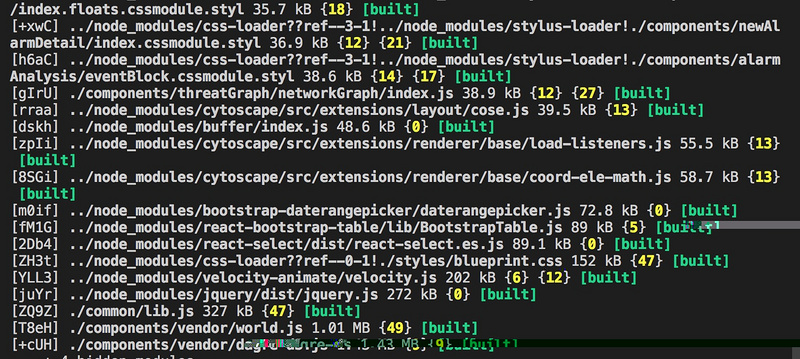
## 优化打包大小

让人震惊的是，整个项目由于种种原因，打包后的大小有近20M的大小！虽然并不是TO C项目，并且针对页面进行了代码拆分和懒加载，但是作为一个“合格的前端”，这种现象是一定要修改的（没错！）。该如何下手呢？一个个的翻代码，看看我们都引用了什么大依赖，看哪些项目过大未免太复杂了。我们看看webpack给我吗提供了什么方案：

### 1.展示打包结果

我们知道，在webpack打包结束后，会自动在控制台显示打包结果。同时，他也提供了输出依赖及大小的功能，我们执行以下参数, 便可将所有的依赖进行展示，并且看到他们的大小了。

webpack --**display**-modules --**sort**-modules-**by** size

结果类似这样：  


我们可以很快的定位到排名前几的js文件或者第三方依赖，决定该如何对其进行处置。

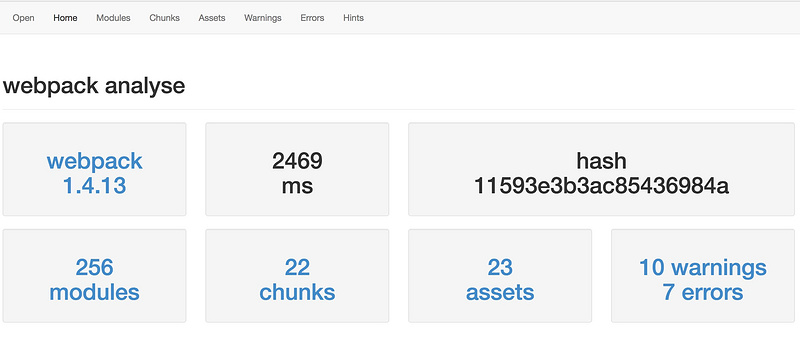
### 2.可视化分析依赖

webpack提供了一个功能，将打包的所有依赖文件以及关系，以json格式进行输出：

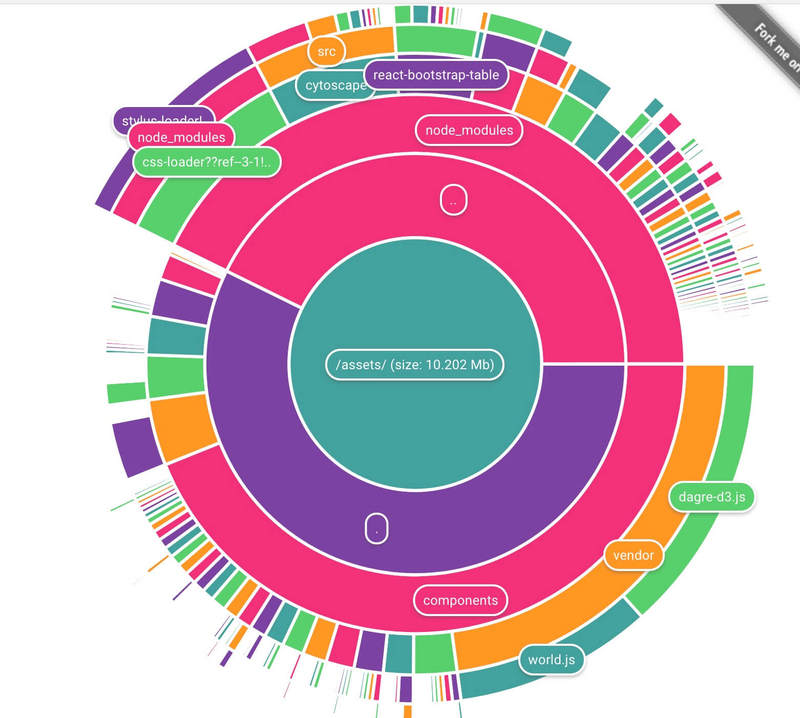
webpack --profile --json > stats.json

这是我们整篇文章的一个基础，很多人基于此封装了不少可视化分析的工具,可以直观的看到各个  
文件、chunk之间的依赖关系以及大小等，快速定位到大文件、大模块

#### [webpack analyse](http://webpack.github.io/analyse/)



#### [webpack chart](http://alexkuz.github.io/webpack-chart/)



### 3.优化方案

通过以上两种方法，我们可以很好的对内容文件和依赖进行定位和分析，针对打包大小的优化方案网上已经有很多了，在此不再进行赘述，提供几个思路及参考：

* CommonsChunkPlugin提取公共代码
* [dll-plugin进行大文件单独打包，缓存](https://segmentfault.com/a/1190000005969643)
* 删除无用的依赖（后面会提到
* 选择性的弃用一些依赖
* 代码压缩
* babel-polyfill
* Scope Hoisting

## 优化打包时间

针对打包时间的优化的文章其实也很多了，我们在此仅提供一些思路。我们主要提一点，通过构建会发现，项目中引用了大量的svg图标以及国旗图标，每次在静态资源处理中，打包时间就会变的特别慢。

我们在项目中使用的svg-sprite-loader，自动将各个svg图标进行svg-spirte。但是我们知道，这些图标一旦引用，我们很少进行修改。尤其是像国旗图标这种，但是每次构建我们都需要进行重复打包。因此，我们可以提前把这些图标进行svg-sprite。推荐一个网站，将各种svg图标提前进行sprite并自动进行引用：

[iconmoon](https://icomoon.io/)

### 日常打包时间优化点

* externals 避免打包大的第三方依赖
* dll-plugin 预打包第三方依赖
* happypack 多进程处理，缓存

缓存与增量构建

* + babel-loader?cacheDirectory
  + webpack cache:true
* 减少构建搜索或编译路径 alias resolve
* 具象打包的范围 include exclude

## 总结

通过对webpack输出依赖关系的json的分析，我们可以直观的拿到以下数据：

* 所有依赖文件及其大小
* 每个依赖文件是被哪些文件引用的
* 项目依赖的第三方依赖

通过这些数据，我们可以很方便的对现有项目进行优化。

## 问题归纳

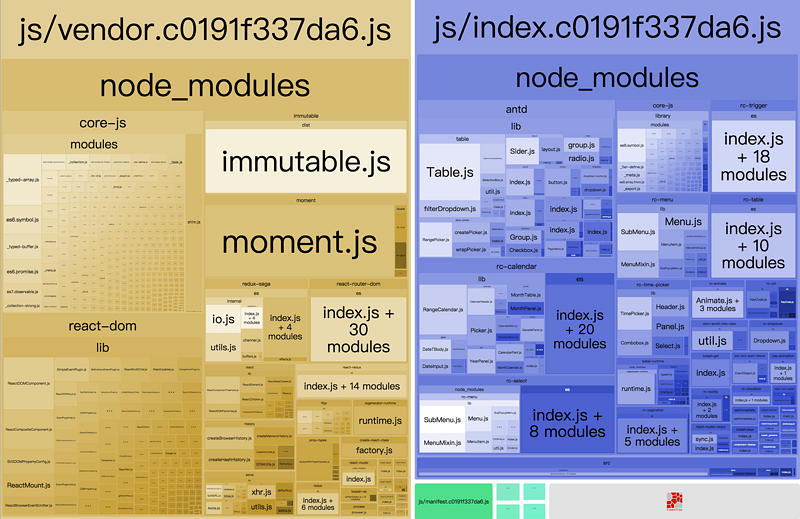
历经了多个web项目的实战检验，我们对webapck在构建中逐步暴露出来的性能问题归纳主要有如下几个方面：

* 代码全量构建速度过慢，即使是很小的改动，也要等待长时间才能查看到更新与编译后的结果（引入HMR热更新后有明显改进）；
* 随着项目业务的复杂度增加，工程模块的体积也会急剧增大，构建后的模块通常要以M为单位计算；
* 多个项目之间共用基础资源存在重复打包，基础库代码复用率不高；
* node的单进程实现在耗cpu计算型loader中表现不佳；

针对以上的问题，我们来看看怎样利用webpack现有的一些机制和第三方扩展插件来逐个击破。

## 慢在何处

作为工程师，我们一直鼓励要理性思考，用数据和事实说话，“我觉得很慢”，“太卡了”，“太大了”之类的表述难免显得太笼统和太抽象，那么我们不妨从如下几个方面来着手进行分析：



* 从项目结构着手，代码组织是否合理，依赖使用是否合理；
* 从webpack自身提供的优化手段着手，看看哪些api未做优化配置；
* 从webpack自身的不足着手，做有针对性的扩展优化，进一步提升效率；

在这里我们推荐使用一个wepback的可视化资源分析工具：[webpack-bundle-analyzer](https://github.com/webpack-contrib/webpack-bundle-analyzer)，在webpack构建的时候会自动帮你计算出各个模块在你的项目工程中的依赖与分布情况，方便做更精确的资源依赖和引用的分析。

从上图中我们不难发现大多数的工程项目中，依赖库的体积永远是大头，通常体积可以占据整个工程项目的7-9成，而且在每次开发过程中也会重新读取和编译对应的依赖资源，这其实是很大的的资源开销浪费，而且对编译结果影响微乎其微，毕竟在实际业务开发中，我们很少会去主动修改第三方库中的源码，改进方案如下：

### 方案一、合理配置 CommonsChunkPlugin

webpack的资源入口通常是以entry为单元进行编译提取，那么当多entry共存的时候，CommonsChunkPlugin的作用就会发挥出来，对所有依赖的chunk进行公共部分的提取，但是在这里可能很多人会误认为抽取公共部分指的是能抽取某个代码片段，其实并非如此，它是以module为单位进行提取。

假设我们的页面中存在entry1，entry2，entry3三个入口，这些入口中可能都会引用如utils，loadash，fetch等这些通用模块，那么就可以考虑对这部分的共用部分机提取。通常提取方式有如下四种实现：

**1、传入字符串参数，由chunkplugin自动计算提取**

**new** webpack.optimize.CommonsChunkPlugin('common.js')

这种做法默认会把所有入口节点的公共代码提取出来, 生成一个common.js

**2、有选择的提取公共代码**

**new** webpack.optimize.CommonsChunkPlugin('common.js',['entry1','entry2']);

只提取entry1节点和entry2中的共用部分模块, 生成一个common.js

**3、将entry下所有的模块的公共部分（可指定引用次数）提取到一个通用的chunk中**

**new** webpack.optimize.CommonsChunkPlugin({

name: 'vendors',

minChunks: **function** (module, count) {

**return** (

module.resource &&

/\.js$/.test(module.resource) &&

module.resource.indexOf(

path.join(\_\_dirname, '../node\_modules')

) === 0

)

}

});

提取所有node\_modules中的模块至vendors中，也可以指定minChunks中的最小引用数；

**4、抽取enry中的一些lib抽取到vendors中**

entry = {

vendors: ['fetch', 'loadash']

};**new** webpack.optimize.CommonsChunkPlugin({

name: "vendors",

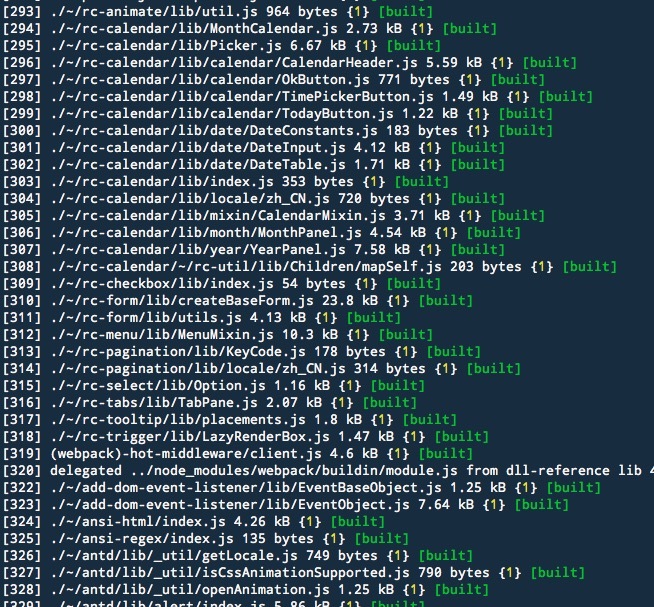
minChunks: Infinity

});

添加一个entry名叫为vendors，并把vendors设置为所需要的资源库，CommonsChunk会自动提取指定库至vendors中。

### 方案二、通过 externals 配置来提取常用库

在实际项目开发过程中，我们并不需要实时调试各种库的源码，这时候就可以考虑使用external选项了。



简单来说external就是把我们的依赖资源声明为一个外部依赖，然后通过script外链脚本引入。这也是我们早期页面开发中资源引入的一种翻版，只是通过配置后可以告知webapck遇到此类变量名时就可以不用解析和编译至模块的内部文件中，而改用从外部变量中读取，这样能极大的提升编译速度，同时也能更好的利用CDN来实现缓存。

external的配置相对比较简单，只需要完成如下三步：

**1、在页面中加入需要引入的lib地址**，如下：

<head><script src="//cdn.bootcss.com/jquery.min.js"></script><script src="//cdn.bootcss.com/underscore.min.js"></script><script src="/static/common/react.min.js"></script><script src="/static/common/react-dom.js"></script><script src="/static/common/react-router.js"></script><script src="/static/common/immutable.js"></script></head>

**2、在webapck.config.js中加入external配置项：**

module.export = {

externals: {

'react-router': {

amd: 'react-router',

root: 'ReactRouter',

commonjs: 'react-router',

commonjs2: 'react-router'

},

react: {

amd: 'react',

root: 'React',

commonjs: 'react',

commonjs2: 'react'

},

'react-dom': {

amd: 'react-dom',

root: 'ReactDOM',

commonjs: 'react-dom',

commonjs2: 'react-dom'

}

}

}

这里要提到的一个细节是：此类文件在配置前，构建这些资源包时需要采用amd/commonjs/cmd相关的模块化进行兼容封装，即打包好的库已经是umd模式包装过的，如在node\_modules/react-router中我们可以看到umd/ReactRouter.js之类的文件，只有这样webpack中的require和import \* from 'xxxx'才能正确读到该类包的引用，在这类js的头部一般也能看到如下字样：

**if** (**typeof** exports === 'object' && **typeof** module === 'object') {

module.exports = factory(require("react"));

} **else** **if** (**typeof** define === 'function' && define.amd) {

define(["react"], factory);

} **else** **if** (**typeof** exports === 'object') {

exports["ReactRouter"] = factory(require("react"));

} **else** {

root["ReactRouter"] = factory(root["React"]);

}

**3、非常重要的是一定要在output选项中加入如下一句话：**

output: {

libraryTarget: 'umd'

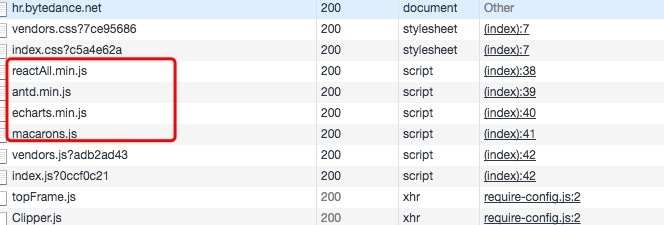
}

由于通过external提取过的js模块是不会被记录到webapck的chunk信息中，通过libraryTarget可告知我们构建出来的业务模块，当读到了externals中的key时，需要以umd的方式去获取资源名，否则会有出现找不到module的情况。

通过配置后，我们可以看到对应的资源信息已经可以在浏览器的source map中读到了。



对应的资源也可以直接由页面外链载入，有效地减小了资源包的体积。



### 方案三、利用 DllPlugin 和 DllReferencePlugin 预编译资源模块

我们的项目依赖中通常会引用大量的npm包，而这些包在正常的开发过程中并不会进行修改，但是在每一次构建过程中却需要反复的将其解析，如何来规避此类损耗呢？这两个插件就是干这个用的。

简单来说DllPlugin的作用是预先编译一些模块，而DllReferencePlugin则是把这些预先编译好的模块引用起来。这边需要注意的是DllPlugin必须要在DllReferencePlugin执行前先执行一次，dll这个概念应该也是借鉴了windows程序开发中的dll文件的设计理念。

相对于externals，dllPlugin有如下几点优势：

* dll预编译出来的模块可以作为静态资源链接库可被重复使用，尤其适合多个项目之间的资源共享，如同一个站点pc和手机版等；

dll资源能有效地解决资源循环依赖的问题，部分依赖库如：react-addons-css-transition-group这种原先从react核心库中抽取的资源包，整个代码只有一句话：

module.exports = require('react/lib/ReactCSSTransitionGroup');

却因为重新指向了react/lib中，这也会导致在通过externals引入的资源只能识别react,寻址解析react/lib则会出现无法被正确索引的情况。

* 由于externals的配置项需要对每个依赖库进行逐个定制，所以每次增加一个组件都需要手动修改，略微繁琐，而通过dllPlugin则能完全通过配置读取，减少维护的成本；

**1、配置dllPlugin对应资源表并编译文件**

那么externals该如何使用呢，其实只需要增加一个配置文件：webpack.dll.config.js：

**const** webpack = require('webpack');**const** path = require('path');**const** isDebug = process.env.NODE\_ENV === 'development';**const** outputPath = isDebug ? path.join(\_\_dirname, '../common/debug') : path.join(\_\_dirname, '../common/dist');**const** fileName = '[name].js';

// 资源依赖包，提前编译**const** lib = [

'react',

'react-dom',

'react-router',

'history',

'react-addons-pure-render-mixin',

'react-addons-css-transition-group',

'redux',

'react-redux',

'react-router-redux',

'redux-actions',

'redux-thunk',

'immutable',

'whatwg-fetch',

'byted-people-react-select',

'byted-people-reqwest'

];

**const** plugin = [

**new** webpack.DllPlugin({

/\*\*

\* path

\* 定义 manifest 文件生成的位置

\* [name]的部分由entry的名字替换

\*/

path: path.join(outputPath, 'manifest.json'),

/\*\*

\* name

\* dll bundle 输出到那个全局变量上

\* 和 output.library 一样即可。

\*/

name: '[name]',

context: \_\_dirname

}),

**new** webpack.optimize.OccurenceOrderPlugin()

];

**if** (!isDebug) {

plugin.push(

**new** webpack.DefinePlugin({

'process.env.NODE\_ENV': JSON.stringify('production')

}),

**new** webpack.optimize.UglifyJsPlugin({

mangle: {

except: ['$', 'exports', 'require']

},

compress: { warnings: false },

output: { comments: false }

})

)

}

module.exports = {

devtool: '#source-map',

entry: {

lib: lib

},

output: {

path: outputPath,

filename: fileName,

/\*\*

\* output.library

\* 将会定义为 window.${output.library}

\* 在这次的例子中，将会定义为`window.vendor\_library`

\*/

library: '[name]',

libraryTarget: 'umd',

umdNamedDefine: true

},

plugins: plugin

};

然后执行命令：

$ NODE\_ENV=development webpack --config webpack.dll.lib.js --progress

$ NODE\_ENV=production webpack --config webpack.dll.lib.js --progress

即可分别编译出支持调试版和生产环境中lib静态资源库，在构建出来的文件中我们也可以看到会自动生成如下资源：

common

├── debug

│ ├── lib.js

│ ├── lib.js.map

│ └── manifest.json

└── dist

├── lib.js

├── lib.js.map

└── manifest.json

文件说明：

* lib.js可以作为编译好的静态资源文件直接在页面中通过src链接引入，与externals的资源引入方式一样，生产与开发环境可以通过类似charles之类的代理转发工具来做路由替换；
* manifest.json中保存了webpack中的预编译信息，这样等于提前拿到了依赖库中的chunk信息，在实际开发过程中就无需要进行重复编译；

**2、dllPlugin的静态资源引入**

lib.js和manifest.json存在一一对应的关系，所以我们在调用的过程也许遵循这个原则，如当前处于开发阶段，对应我们可以引入common/debug文件夹下的lib.js和manifest.json，切换到生产环境的时候则需要引入common/dist下的资源进行对应操作，这里考虑到手动切换和维护的成本，我们推荐使用[add-asset-html-webpack-plugin](https://github.com/SimenB/add-asset-html-webpack-plugin)进行依赖资源的注入，可得到如下结果：

<head><script src="/static/common/lib.js"></script></head>

在webpack.config.js文件中增加如下代码：

**const** isDebug = (process.env.NODE\_ENV === 'development');**const** libPath = isDebug ? '../dll/lib/manifest.json' : '../dll/dist/lib/manifest.json';

// 将mainfest.json添加到webpack的构建中

module.export = {

plugins: [

**new** webpack.DllReferencePlugin({

context: \_\_dirname,

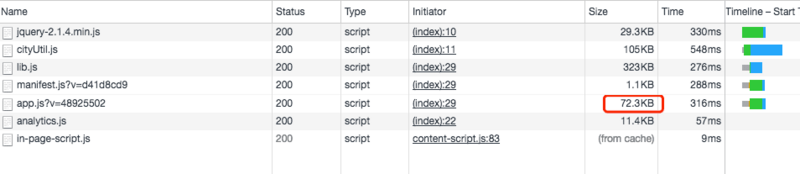
manifest: require(libPath),

})

]

}

配置完成后我们能发现对应的资源包已经完成了纯业务模块的提取



多个工程之间如果需要使用共同的lib资源，也只需要引入对应的lib.js和manifest.js即可，plugin配置中也支持多个webpack.DllReferencePlugin同时引入使用，如下：

module.export = {

plugins: [

**new** webpack.DllReferencePlugin({

context: \_\_dirname,

manifest: require(libPath),

}),

**new** webpack.DllReferencePlugin({

context: \_\_dirname,

manifest: require(ChartsPath),

})

]

}

### 方案四、使用 Happypack 加速你的代码构建

以上介绍均为针对webpack中的chunk计算和编译内容的优化与改进，对资源的实际体积改进上也较为明显，那么除此之外，我们能否针对资源的编译过程和速度优化上做些尝试呢？

众所周知，webpack中为了方便各种资源和类型的加载，设计了以loader加载器的形式读取资源，但是受限于node的编程模型影响，所有的loader虽然以async的形式来并发调用，但是还是运行在单个 node的进程以及在同一个事件循环中，这就直接导致了当我们需要同时读取多个loader文件资源时，比如babel-loader需要transform各种jsx，es6的资源文件。在这种同步计算同时需要大量耗费cpu运算的过程中，node的单进程模型就无优势了，那么happypack就针对解决此类问题而生。

#### 开启happypack的线程池

happypack的处理思路是将原有的webpack对loader的执行过程从单一进程的形式扩展多进程模式，原本的流程保持不变，这样可以在不修改原有配置的基础上来完成对编译过程的优化，具体配置如下：

**const** HappyPack = require('happypack');

**const** os = require('os')

**const** HappyThreadPool = HappyPack.ThreadPool({ size: os.cpus().length}); // 启动线程池});

module:{

rules: [

{

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

// use: ['babel-loader?cacheDirectory'],

use: 'happypack/loader?id=jsx',

exclude: /^node\_modules$/

}

]

},

plugins:[

**new** HappyPack({

id: 'jsx',

cache: true,

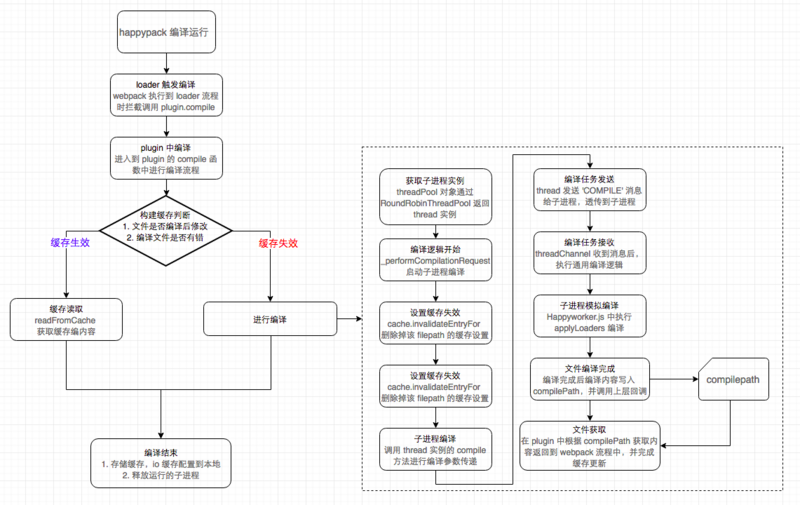
threadPool: HappyThreadPool,

loaders: ['babel-loader']

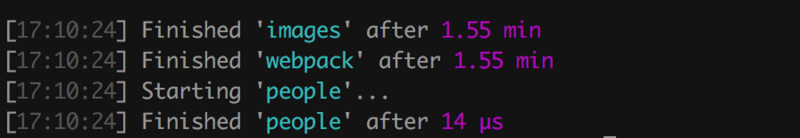
})

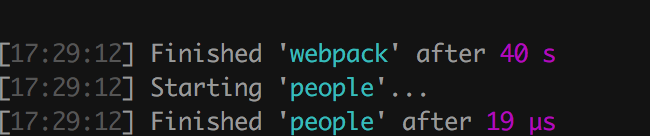
]

我们可以看到通过在loader中配置直接指向happypack提供的loader，对于文件实际匹配的处理 loader，则是通过配置在plugin属性来传递说明，这里happypack提供的loader与plugin的衔接匹配，则是通过id=happybabel来完成。配置完成后，laoder的工作模式就转变成了如下所示：



happypack在编译过程中除了利用多进程的模式加速编译，还同时开启了cache计算，能充分利用缓存读取构建文件，对构建的速度提升也是非常明显的，经过测试，最终的构建速度提升如下：

优化前：  


优化后：  


关于happyoack的更多介绍可以查看：

* [happypack](https://github.com/amireh/happypack)
* [happypack 原理解析](http://taobaofed.org/blog/2016/12/08/happypack-source-code-analysis/?utm_source=tuicool&utm_medium=referral)

### 方案五、增强 uglifyPlugin

uglifyJS凭借基于node开发，压缩比例高，使用方便等诸多优点已经成为了js压缩工具中的首选，但是我们在webpack的构建中观察发现，当webpack build进度走到80%前后时，会发生很长一段时间的停滞，经测试对比发现这一过程正是uglfiyJS在对我们的output中的bunlde部分进行压缩耗时过长导致，针对这块我们可以使用[webpack-uglify-parallel](https://github.com/tradingview/webpack-uglify-parallel)来提升压缩速度。

从插件源码中可以看到，webpack-uglify-parallel的是实现原理是采用了多核并行压缩的方式来提升我们的压缩速度。

plugin.nextWorker().send({

input: input,

inputSourceMap: inputSourceMap,

file: file,

options: options

});

plugin.\_queue\_len++;

**if** (!plugin.\_queue\_len) {

callback();

}

**if** (**this**.workers.length < **this**.maxWorkers) {

**var** worker = fork(\_\_dirname + '/lib/worker');

worker.on('message', **this**.onWorkerMessage.bind(**this**));

worker.on('error', **this**.onWorkerError.bind(**this**));

**this**.workers.push(worker);

}

**this**.\_next\_worker++;**return** **this**.workers[**this**.\_next\_worker % **this**.maxWorkers];

使用配置也非常简单，只需要将我们原来webpack中自带的uglifyPlugin配置：

**new** webpack.optimize.UglifyJsPlugin({

exclude:/\.min\.js$/

mangle:true,

compress: { warnings: false },

output: { comments: false }

})

修改成如下代码即可：

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

**const** UglifyJsParallelPlugin = require('webpack-uglify-parallel');

**new** UglifyJsParallelPlugin({

workers: os.cpus().length,

mangle: true,

compressor: {

warnings: false,

drop\_console: true,

drop\_debugger: true

}

})

目前webpack官方也维护了一个支持多核压缩的UglifyJs插件：[uglifyjs-webpack-plugin](https://webpack.js.org/plugins/uglifyjs-webpack-plugin/),使用方式类似，优势在于完全兼容webpack.optimize.UglifyJsPlugin中的配置，可以通过uglifyOptions写入，因此也做为**推荐使用**，参考配置如下：

**const** UglifyJsPlugin = require('uglifyjs-webpack-plugin');

**new** UglifyJsPlugin({

uglifyOptions: {

ie8: false,

ecma: 8,

mangle: true,

output: { comments: false },

compress: { warnings: false }

},

sourceMap: false,

cache: true,

parallel: os.cpus().length \* 2

})

### 方案六、Tree-shaking & Scope Hoisting

wepback在2.X和3.X中从rolluo中借鉴了[tree-shaking](https://webpack.js.org/guides/tree-shaking)和[Scope Hoisting](https://webpack.js.org/plugins/module-concatenation-plugin)，利用es6的module特性，利用AST对所有引用的模块和方法做了静态分析，从而能有效地剔除项目中的没有引用到的方法，并将相关方法调用归纳到了独立的webpack\_module中，对打包构建的体积优化也较为明显，但是前提是所有的模块写法必须使用ES6 Module进行实现，具体配置参考如下：

// .babelrc: 通过配置减少没有引用到的方法

{

"presets": [

["env", {

"targets": {

"browsers": ["last 2 versions", "safari >= 7"]

}

}],

// https://www.zhihu.com/question/41922432

["es2015", {"modules": false}] // tree-shaking

]

}

// webpack.config: Scope Hoisting

{

plugins:[

// https://zhuanlan.zhihu.com/p/27980441

**new** webpack.optimize.ModuleConcatenationPlugin()

]

}

#### 适用场景

在实际的开发过程中，可灵活地选择适合自身业务场景的优化手段。

| **优化手段** | **开发环境** | **生产环境** |
| --- | --- | --- |
| CommonsChunk | √ | √ |
| externals |  | √ |
| DllPlugin | √ | √ |
| Happypack | √ |  |
| uglify-parallel |  | √ |