# 一 常用加载器

## 1.1 常见module rules

注意：loader是从后往前处理

// 处理 CSS 文件的 loader

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

// 处理 less 文件的 loader

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

// 处理 scss 文件的 loader

{ test: /\.scss$/, use: ['style-loader', 'css-loader', 'sass-loader'] },

// 处理 图片路径的 loader

{ test: /\.(jpg|png|gif|bmp|jpeg)$/, use: 'url-loader?limit=7631&name=[hash:8]-[name].[ext]' },

// 处理 字体文件的 loader

{ test: /\.(ttf|eot|svg|woff|woff2)$/, use: 'url-loader' },

// 配置 Babel 来转换高级的ES语法

{ test: /\.js$/, use: 'babel-loader', exclude: /node\_modules/ }

## 1.2 处理CSS style-loader css-loader

### 1.2.1 常用处理方式

style-loader： 负责在加载的页面创建样式标签，引入css

css-loader： 让JS支持import css模块

加载器：

rules: [

{

test: /\.css$/,

use: [

{loader: 'style-loader'},

{loader: 'css-loader'}

]

}

]

我们在app.js中可以直接使用import引入css，此时打包的结果：

css样式被以<style>标签的形式直接写入了html中，这样的好处是CSS 完全包含在合并的应用中，再也不需要重新下载。

### 1.2.2 第二种方式file-loader

另外一种打包方式：不使用css-loader，此时import的css不会被以标签形式直接加入html，而是额外打包成了css文件，通过link标签引入该css：

{

test: /\.css$/,

use: [

{loader: 'style-loader/url'},

{loader: 'file-loader'}

]

}

但是这样打包后，如果import了多个css，那么会生成多个link，这样会造成多次请求，所以很少使用。

### 1.2.3 CSS样式显示控制

use: [

{loader: 'style-loader/useable'},

{loader: 'css-loader'}

]

这样配置后，我们可以在js中控制CSS的显示与否，比如在app.js中加入如下代码：

import baseCSS from './css/index.css';

baseCSS.use();

setInterval(function(){

baseCSS.unuse();

},300);

### 1.2.4 style-loader选项

常见选项：

insertAt： 插入位置

insertInto： 插入到某个dom节点

singleton： 是否只使用一个style标签

transform： 浏览器中转化，插入也面前执行一些列操作

案例：

{

loader: 'style-loader',

options: {

insertInfo: '#box',

singleton: true

}

},

上述配置案例中代表，style会直接插入到 #box这个标签中，singleton为true代表多个不同的style将会被合并到一个style中。

关于transform使用： transform: ./css.transform.js'' ##相对于webpack配置的路径

此时编译完成后，打开浏览器时执行这个js中的函数。

css.transform.js：

module.exports = function(css){ return css}

在该函数中，可以使用window对象，那么我们就可以拿到一系列浏览器相关信息，可以依据不同的浏览器输出不同的css到style标签中。

### 1.2.5 css-loader选项

常见选项：

alias： 解析的别名

importLoader： 也作 @import

minimize： 是否压缩，true为要压缩

modules： 是否启用css-modules，true为启用css模块化

localIdentName: CSS的class名重新生成，常见设定为：



## 1.3 处理Less/Sass

只需要安装和配置相应的loader即可：sass-loader less-loader

## 1.4 加载资源文件 file-loader

webpack无法处理background中的url地址等等类似的资源文件都需要加载器支持。

npm i -D file-loader

配置如下所示：

{

test: /\.(png|jpg|jpeg|gif)$/,

use: [

{

loader: 'file-loader',

options: {

publicPath: '',

outputPath: 'dist/',

useRelativePath: true

}

}

]

}

## 1.5 base64压缩图片 url-loader

npm i -D url-loader

配置：

{

test: /\.(png|jpg|jpeg|gif)$/,

use: [

{

loader: 'url-loader',

options: {

limit: 10000 *//小于该大小的图片转变为base64*

}

}

]

}

注意：url-loader具备和file-loader一样的图片打包功能，但是额外多了base64压缩功能

在第一章 第0节中的配置中加入[hash]的原因：



webpack复制到打包后的文件目录的文件都是按照文件名复制的，如果源文件文件名出现重复，则打包后的不同图片变成了相同图片，所以需要加入hash前缀，8指取前8位。

## 1.6 压缩图片 img-loader

直接添加该加载器进入配置后即可压缩图片，如果需要配置压缩率，可以添加可选项：

pngquant:{

quality: 80

}

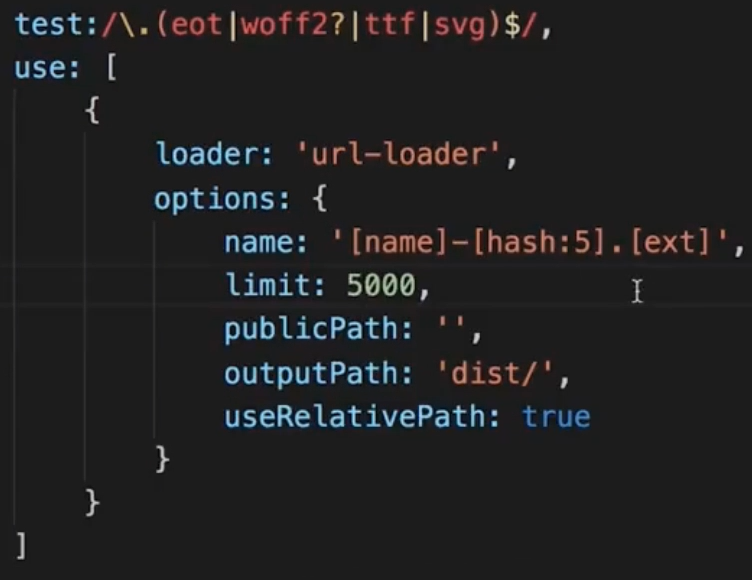
## 1.7 图片合并 postcss-loader



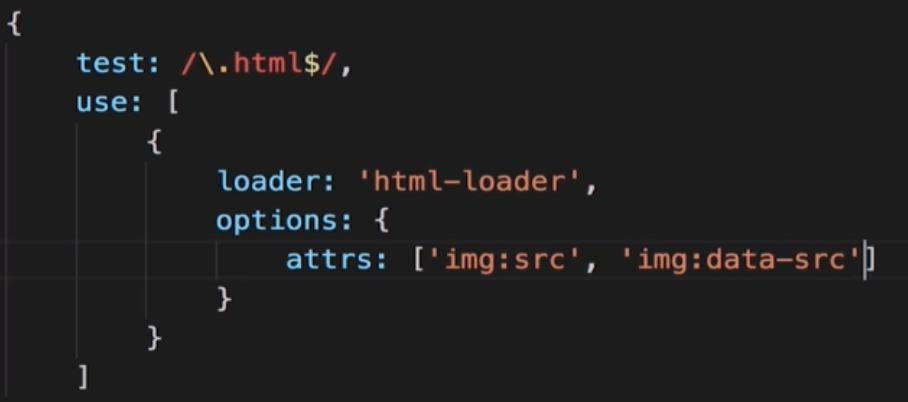
retina:true时候可以支持视网膜屏，比如有些图是\*\*@2x.png

## 1.8 处理字体文件 url-loader

推荐将字体放置在本地目录。



## 1.9 处理img标签 html-loader



## 1.10 处理typescript

安装loader：

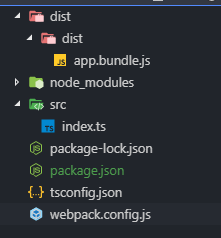
推荐安装官方的：npm i typescript ts-loader -D

或者安装第三方：npm i awesome-typescript-loader -D #作者声称利用了缓存，更快

在根目录新建ts配置文件：tsconfig.json (详见官网 /docs/handbook/compiler-options.html)

常用选项：compilerOptions，include，exclude

配置目录如图：（配置npm脚本：）



tsconfig.json配置：

{

"compilerOptions": {

"module": "commonjs",

"target": "es5",

"allowJs": true

},

"include": [

"./src/\*"

],

"exclude": [

"./node\_module"

]

}

webpack配置：

{

test: /\.tsx?$/,

use: {

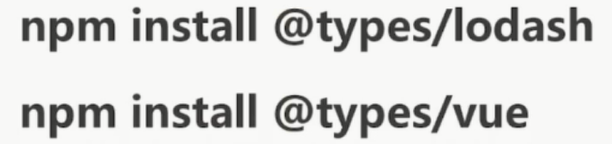
loader: 'ts-loader'

}

}

关于ts的一些小贴士：ts虽然对类型进行了限制，但是不会制动提示哪个地方出现了类型问题。

可以安装一些声明文件：

 如果需要vue的话。

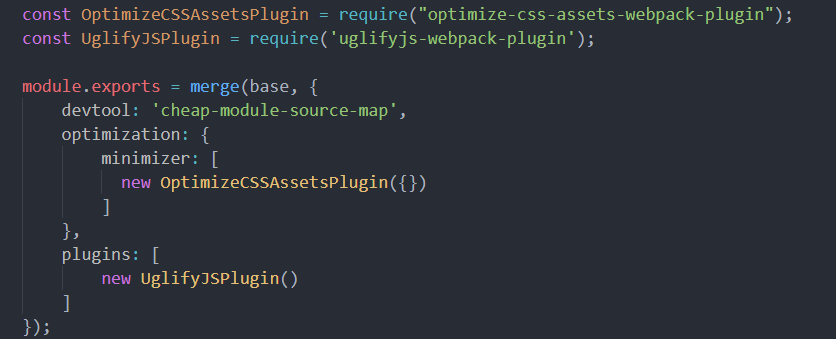
在文件中引入，即可识别：



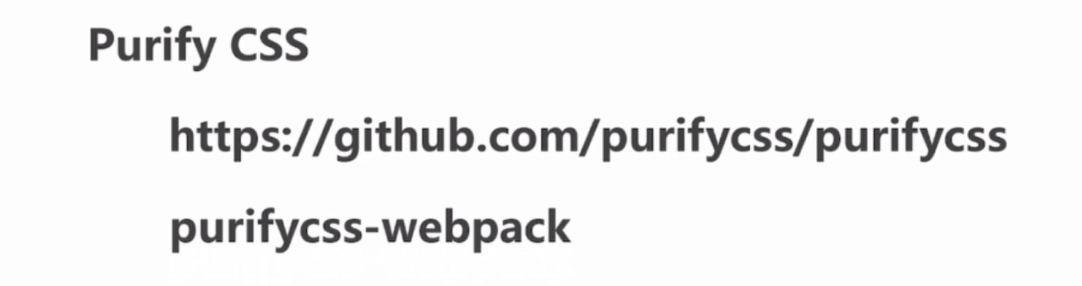
## 1.11 其他

JS去除不被利用的代码：

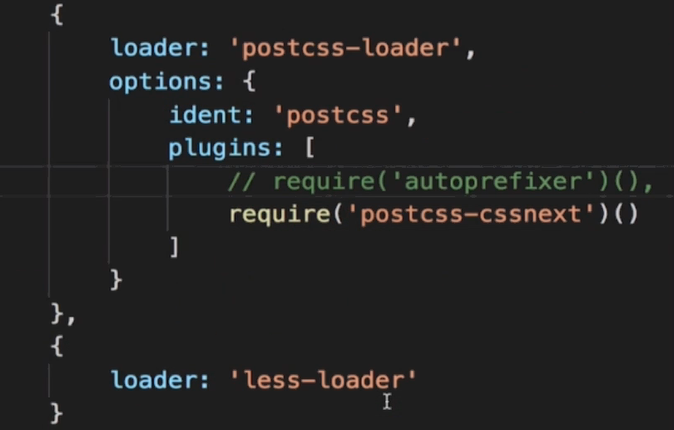
webpack 在打包时，会将没被使用的代码注释掉，但是只注释仍然引起了打包文件过大，这时候可以配置如下，去除注释：



CSS去除不被利用代码：

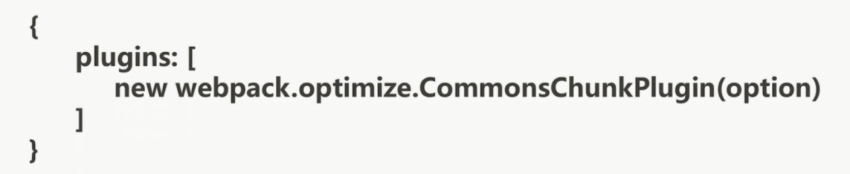


CSS自动加入浏览器前缀加载器：postcss



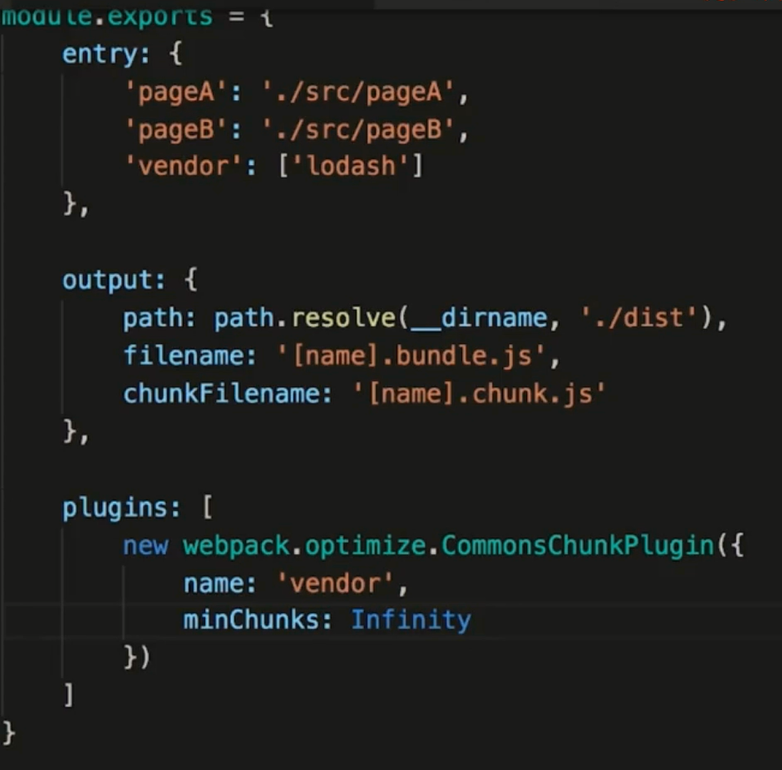
# 二 常用插件介绍

## 0 内置插件提取公共代码



注意：该插件主要针对多页面

配置如下：

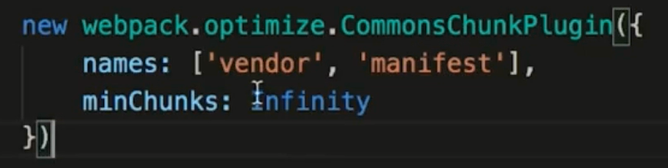


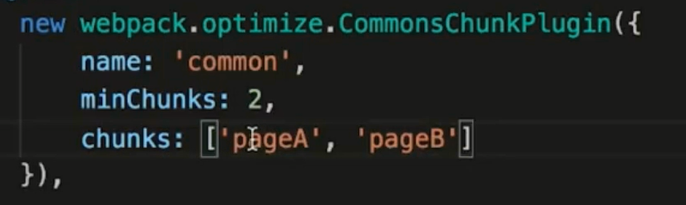
此时 入口pageA和入口pageB引入的共同文件将会被打包到一个公共js中，名为插件配置中的name，如果name的值和入口中某个文件相同，那么会被打包到该入口文件中，如上所示：

vendor是默认该项目引入的一系列第三方包，其他入口的功用代码也会被打包到该文件中。

如果不想被打包如某个入口文件，可以在plugins中修改name，或者再配置一个optimize。

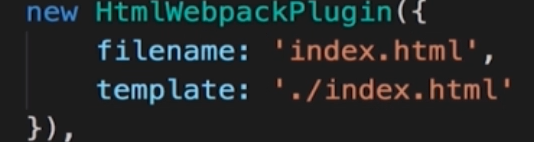
也可以这样配置：





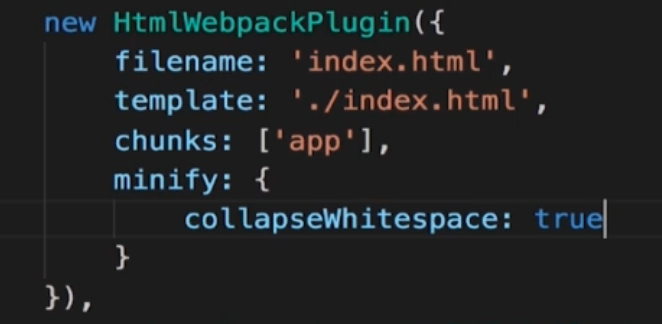
## 1 打包html插件

npm i html-webpack-plugin -D



打包完毕后，我们会发现依赖的CSS,JS都被加入到了dist目录下生成的index.html中。

问题：如果有多个html，每个html的入口不一致，如何处理？



注意：minify是让html压缩，也是使用了第三方的压缩包

## 2 提取公共CSS插件

### 2.2.1 基本使用

首先需要安装该插件：extract-text-webpack-plugin

注意：该插件需要依赖本地的webpack，并且需要注意插件本身的版本

通过该插件提取的css并未自动添加到html中，需要手动引入。

使用步骤：

在webpac中引入：

const ExtractTextWebpackPlugin = require('extract-text-webpack-plugin');

定义plugins：

plugins: [

new ExtractTextWebpackPlugin({

filename: '[name].min.css' *//提取后的css名称*

})

]

修改css加载器：

rules: [

{

test: /\.less$/,

use: ExtractTextWebpackPlugin.extract({

fallback: { *//不提取时，采用哪种方式加载样式到页面*

loader: 'style-loader',

options: {

singleton: true

}

},

use: [ *//继续处理loader*

{loader: 'css-loader'},

{loader: 'less-loader'}

]

})

}

]

### 2.2.2 allChunks参数

在plugin配置参数中，除了filename，还可以配置allChunks，默认值是false。

该插件指定了插件提取css的范围。如果值设置为true，则会把所有import进来的css提取到一个css中，为false则只会提取初始化的css，比如异步加载的css则不会被提取（异步的css即不同组件里引入的css）。

### 2.2.3 懒加载的详细使用

allChunks懒加载可以发挥多重入口文件的优势，你可以在每个入口点包含各自的 CSS。模块用文件夹分离，每个文件夹有各自的 CSS 和 JavaScript 文件。当应用再次发布的时候，导入的 CSS 会加载到每个入口文件中。

app/main.js import ‘./style.css’;

app/entryA/main.js import ‘./style.css’;

app/entryB/main.js import ‘./style.css’;

可以根据这个策略为每个组件创建 CSS 文件，可以让组件名和 CSS 中的 class 使用一个命名空间，来避免一个组件中的一些 class 干扰到另外一些组件的 class。



在 “React Native” 中不再需要使用任何 CSS 文件，你只需要使用 *style 属性*，可以把 CSS 定义成一个对象，那样就可以根据项目重新来考略 CSS 策略。

注意：这点的样式都没有-



## 3 压缩插件

这个插件是webpack自带的，如上所示

**new** webpack.**optimize**.UglifyJsPlugin({  
 **compress**: {  
 **warnings**: **false** }  
 })

## 5 其他插件

删除目录插件 [clean-webpack-plugin](https://github.com/johnagan/clean-webpack-plugin)

提取公共JS插件 CommonsChunkPlugin

拷贝文件插件 [copy-webpack-plugin](https://github.com/kevlened/copy-webpack-plugin)