## **1. purgecss-webpack-plugin**

* [purgecss-webpack-plugin](https://www.npmjs.com/package/purgecss-webpack-plugin)
* [mini-css-extract-plugin](https://www.npmjs.com/package/mini-css-extract-plugin)
* [purgecss](https://www.purgecss.com/)
* 可以去除未使用的 css，一般与 glob、glob-all 配合使用
* 必须和mini-css-extract-plugin配合使用
* paths路径是绝对路径

npm i purgecss-webpack-plugin mini-css-extract-plugin css-loader glob -D

webpack.config.js

const path = require("path");+const glob = require("glob");+const PurgecssPlugin = require("purgecss-webpack-plugin");+const MiniCssExtractPlugin = require('mini-css-extract-plugin');

const PATHS = {

src: path.join(\_\_dirname, 'src')

}

module.exports = {

mode: "development",

entry: "./src/index.js",

module: {

rules: [

{

test: /\.js/,

include: path.resolve(\_\_dirname, "src"),

use: [

{

loader: "babel-loader",

options: {

presets: ["@babel/preset-env", "@babel/preset-react"],

},

},

],

},+ {+ test: /\.css$/,+ include: path.resolve(\_\_dirname, "src"),+ exclude: /node\_modules/,+ use: [+ {+ loader: MiniCssExtractPlugin.loader,+ },+ "css-loader",+ ],+ },

],

},

plugins: [+ new MiniCssExtractPlugin({+ filename: "[name].css",+ }),+ new PurgecssPlugin({+ paths: glob.sync(`${PATHS.src}/\*\*/\*`, { nodir: true }),+ })

],

};

## **3. 多进程处理**

### **3.1 thread-loader**

* 把这个 loader 放置在其他 loader 之前， 放置在这个 loader 之后的 loader 就会在一个单独的 worker 池(worker pool)中运行
* [thread-loader](https://webpack.js.org/loaders/thread-loader/)

npm i thread-loader- D

const path = require("path");

const glob = require("glob");

const PurgecssPlugin = require("purgecss-webpack-plugin");

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

const DllReferencePlugin = require("webpack/lib/DllReferencePlugin.js");

const PATHS = {

src: path.join(\_\_dirname, 'src')

}

module.exports = {

mode: "development",

entry: "./src/index.js",

module: {

rules: [

{

test: /\.js/,

include: path.resolve(\_\_dirname, "src"),

use: [+ {+ loader:'thread-loader',+ options:{+ workers:3+ }+ },

{

loader: "babel-loader",

options: {

presets: ["@babel/preset-env", "@babel/preset-react"],

},

},

],

},

{

test: /\.css$/,

include: path.resolve(\_\_dirname, "src"),

exclude: /node\_modules/,

use: [

{

loader: MiniCssExtractPlugin.loader,

},

"css-loader",

],

},

],

},

plugins: [

new MiniCssExtractPlugin({

filename: "[name].css",

}),

new PurgecssPlugin({

paths: glob.sync(`${PATHS.src}/\*\*/\*`, { nodir: true }),

}),

new DllReferencePlugin({

manifest: require("./dist/react.manifest.json"),

}),

],

};

## **4. CDN**

* [qiniu](https://www.qiniu.com/)
* CDN 又叫内容分发网络，通过把资源部署到世界各地，用户在访问时按照就近原则从离用户最近的服务器获取资源，从而加速资源的获取速度。



* HTML文件不缓存，放在自己的服务器上，关闭自己服务器的缓存，静态资源的URL变成指向CDN服务器的地址
* 静态的JavaScript、CSS、图片等文件开启CDN和缓存，并且文件名带上HASH值
* 为了并行加载不阻塞，把不同的静态资源分配到不同的CDN服务器上

### **4.1 使用缓存**

* 由于 CDN 服务一般都会给资源开启很长时间的缓存，例如用户从 CDN 上获取到了 index.html 这个文件后， 即使之后的发布操作把 index.html 文件给重新覆盖了，但是用户在很长一段时间内还是运行的之前的版本，这会新的导致发布不能立即生效 解决办法
* 针对 HTML 文件：不开启缓存，把 HTML 放到自己的服务器上，而不是 CDN 服务上，同时关闭自己服务器上的缓存。自己的服务器只提供 HTML 文件和数据接口。
* 针对静态的 JavaScript、CSS、图片等文件：开启 CDN 和缓存，上传到 CDN 服务上去，同时给每个文件名带上由文件内容算出的 Hash 值
* 带上 Hash 值的原因是文件名会随着文件内容而变化，只要文件发生变化其对应的 URL 就会变化，它就会被重新下载，无论缓存时间有多长。
* 启用CDN之后 相对路径，都变成了绝对的指向 CDN 服务的 URL 地址

### **4.2 域名限制**

* 同一时刻针对同一个域名的资源并行请求是有限制
* 可以把这些静态资源分散到不同的 CDN 服务上去
* 多个域名后会增加域名解析时间
* 可以通过在 HTML HEAD 标签中 加入<link rel="dns-prefetch" href="http://img.zhufengpeixun.cn">去预解析域名，以降低域名解析带来的延迟

### **4.3 接入CDN**

要给网站接入 CDN，需要把网页的静态资源上传到 CDN 服务上去，在服务这些静态资源的时候需要通过 CDN 服务提供的 URL 地址去访问

{

output: {

path: path.resolve(\_\_dirname, 'dist'),+ filename: '[name]\_[hash:8].js',+ publicPath: 'http://img.zhufengpeixun.cn'

},

}

### **4.4 文件指纹**

* 打包后输出的文件名和后缀
* hash一般是结合CDN缓存来使用，通过webpack构建之后，生成对应文件名自动带上对应的MD5值。如果文件内容改变的话，那么对应文件哈希值也会改变，对应的HTML引用的URL地址也会改变，触发CDN服务器从源服务器上拉取对应数据，进而更新本地缓存。

指纹占位符

| **占位符名称** | **含义** |
| --- | --- |
| ext | 资源后缀名 |
| name | 文件名称 |
| path | 文件的相对路径 |
| folder | 文件所在的文件夹 |
| hash | 每次webpack构建时生成一个唯一的hash值 |
| chunkhash | 根据chunk生成hash值，来源于同一个chunk，则hash值就一样 |
| contenthash | 根据内容生成hash值，文件内容相同hash值就相同 |

#### **4.4.1 hash**

* Hash 是整个项目的hash值，其根据每次编译内容计算得到，每次编译之后都会生成新的hash,即修改任何文件都会导致所有文件的hash发生改变

const path = require("path");

const glob = require("glob");

const PurgecssPlugin = require("purgecss-webpack-plugin");

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

const PATHS = {

src: path.join(\_\_dirname, 'src')

}

module.exports = {

mode: "production",+ entry: {+ main: './src/index.js',+ vender:['lodash']+ },

output:{

path:path.resolve(\_\_dirname,'dist'),+ filename:'[name].[hash].js'

},

devServer:{

hot:false

},

module: {

rules: [

{

test: /\.js/,

include: path.resolve(\_\_dirname, "src"),

use: [

{

loader:'thread-loader',

options:{

workers:3

}

},

{

loader: "babel-loader",

options: {

presets: ["@babel/preset-env", "@babel/preset-react"],

},

},

],

},

{

test: /\.css$/,

include: path.resolve(\_\_dirname, "src"),

exclude: /node\_modules/,

use: [

{

loader: MiniCssExtractPlugin.loader,

},

"css-loader",

],

},

],

},

plugins: [

new MiniCssExtractPlugin({+ filename: "[name].[hash].css"

}),

new PurgecssPlugin({

paths: glob.sync(`${PATHS.src}/\*\*/\*`, { nodir: true }),

}),

],

};

#### **4.4.2 chunkhash**

* chunkhash 采用hash计算的话，每一次构建后生成的哈希值都不一样，即使文件内容压根没有改变。这样子是没办法实现缓存效果，我们需要换另一种哈希值计算方式，即chunkhash,chunkhash和hash不一样，它根据不同的入口文件(Entry)进行依赖文件解析、构建对应的chunk，生成对应的哈希值。我们在生产环境里把一些公共库和程序入口文件区分开，单独打包构建，接着我们采用chunkhash的方式生成哈希值，那么只要我们不改动公共库的代码，就可以保证其哈希值不会受影响

const path = require("path");

const glob = require("glob");

const PurgecssPlugin = require("purgecss-webpack-plugin");

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

const PATHS = {

src: path.join(\_\_dirname, 'src')

}

module.exports = {

mode: "production",

entry: {

main: './src/index.js',

vender:['lodash']

},

output:{

path:path.resolve(\_\_dirname,'dist'),+ filename:'[name].[chunkhash].js'

},

devServer:{

hot:false

},

module: {

rules: [

{

test: /\.js/,

include: path.resolve(\_\_dirname, "src"),

use: [

{

loader:'thread-loader',

options:{

workers:3

}

},

{

loader: "babel-loader",

options: {

presets: ["@babel/preset-env", "@babel/preset-react"],

},

},

],

},

{

test: /\.css$/,

include: path.resolve(\_\_dirname, "src"),

exclude: /node\_modules/,

use: [

{

loader: MiniCssExtractPlugin.loader,

},

"css-loader",

],

},

],

},

plugins: [

new MiniCssExtractPlugin({+ filename: "[name].[chunkhash].css"

}),

new PurgecssPlugin({

paths: glob.sync(`${PATHS.src}/\*\*/\*`, { nodir: true }),

}),

],

};

#### **4.4.3 contenthash**

* 使用chunkhash存在一个问题，就是当在一个JS文件中引入CSS文件，编译后它们的hash是相同的，而且只要js文件发生改变 ，关联的css文件hash也会改变,这个时候可以使用mini-css-extract-plugin里的contenthash值，保证即使css文件所处的模块里就算其他文件内容改变，只要css文件内容不变，那么不会重复构建

const path = require("path");

const glob = require("glob");

const PurgecssPlugin = require("purgecss-webpack-plugin");

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

const PATHS = {

src: path.join(\_\_dirname, 'src')

}

module.exports = {

mode: "production",

entry: {

main: './src/index.js',

vender:['lodash']

},

output:{

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

filename:'[name].[chunkhash].js'

},

devServer:{

hot:false

},

module: {

rules: [

{

test: /\.js/,

include: path.resolve(\_\_dirname, "src"),

use: [

{

loader:'thread-loader',

options:{

workers:3

}

},

{

loader: "babel-loader",

options: {

presets: ["@babel/preset-env", "@babel/preset-react"],

},

},

],

},

{

test: /\.css$/,

include: path.resolve(\_\_dirname, "src"),

exclude: /node\_modules/,

use: [

{

loader: MiniCssExtractPlugin.loader,

},

"css-loader",

],

},

],

},

plugins: [

new MiniCssExtractPlugin({+ filename: "[name].[contenthash].css"

}),

new PurgecssPlugin({

paths: glob.sync(`${PATHS.src}/\*\*/\*`, { nodir: true }),

}),

],

};

## **5.Tree Shaking**

* 一个模块可以有多个方法，只要其中某个方法使用到了，则整个文件都会被打到bundle里面去，tree shaking就是只把用到的方法打入bundle,没用到的方法会uglify阶段擦除掉
* 原理是利用es6模块的特点,只能作为模块顶层语句出现,import的模块名只能是字符串常量

### **5.1 开启**

* webpack默认支持，在.babelrc里设置module:false即可在production mode下默认开启
* 还要注意把devtool设置为null 在 package.json 中配置：
* "sideEffects": false 所有的代码都没有副作用（都可以进行 tree shaking）
  + 可能会把 css / @babel/polyfill文件干掉 可以设置"sideEffects":["\*.css"]

webpack.config.js

const path = require("path");

const glob = require("glob");

const PurgecssPlugin = require("purgecss-webpack-plugin");

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

const PATHS = {

src: path.join(\_\_dirname, 'src')

}

module.exports = {+ mode: "production",+ devtool:false,

entry: {

main: './src/index.js'

},

output:{

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

filename:'[name].[hash].js'

},

module: {

rules: [

{

test: /\.js/,

include: path.resolve(\_\_dirname, "src"),

use: [

{

loader:'thread-loader',

options:{

workers:3

}

},

{

loader: "babel-loader",

options: {+ presets: [["@babel/preset-env",{"modules":false}], "@babel/preset-react"],

},

},

],

},

{

test: /\.css$/,

include: path.resolve(\_\_dirname, "src"),

exclude: /node\_modules/,

use: [

{

loader: MiniCssExtractPlugin.loader,

},

"css-loader",

],

},

],

},

plugins: [

new MiniCssExtractPlugin({

filename: "[name].[contenthash].css"

}),

new PurgecssPlugin({

paths: glob.sync(`${PATHS.src}/\*\*/\*`, { nodir: true }),

}),

],

};

### **5.2 没有导入和使用**

functions.js

**function** **func1**(){

**return** 'func1';

}**function** **func2**(){

**return** 'func2';

}**export** {

func1,

func2

}

**import** {func2} **from** './functions';**var** result2 = func2();console.log(result2);

### **5.3 代码不会被执行，不可到达**

**if**(false){

console.log('false')

}

### **5.4 代码执行的结果不会被用到**

**import** {func2} **from** './functions';

func2();

### **5.4 代码中只写不读的变量**

**var** aabbcc='aabbcc';

aabbcc='eeffgg';

## **6 代码分割**

* 对于大的Web应用来讲，将所有的代码都放在一个文件中显然是不够有效的，特别是当你的某些代码块是在某些特殊的时候才会被用到。
* webpack有一个功能就是将你的代码库分割成chunks语块，当代码运行到需要它们的时候再进行加载

### **6.1 入口点分割**

* Entry Points：入口文件设置的时候可以配置
* 这种方法的问题
  + 如果入口 chunks 之间包含重复的模块(lodash)，那些重复模块都会被引入到各个 bundle 中
  + 不够灵活，并且不能将核心应用程序逻辑进行动态拆分代码

entry: {

index: "./src/index.js",

login: "./src/login.js"

}

### **6.2 动态导入和懒加载**

* 用户当前需要用什么功能就只加载这个功能对应的代码，也就是所谓的按需加载 在给单页应用做按需加载优化时
* 一般采用以下原则：
  + 对网站功能进行划分，每一类一个chunk
  + 对于首次打开页面需要的功能直接加载，尽快展示给用户,某些依赖大量代码的功能点可以按需加载
  + 被分割出去的代码需要一个按需加载的时机

#### **6.2.1 hello.js**

hello.js

module.exports = "hello";

index.js

document.querySelector('#clickBtn').addEventListener('click',() => {

**import**('./hello').then(result => {

console.log(result.default);

});

});

index.html

<button id="clickBtn">点我</button>

#### **6.2.2 按需加载**

* 如何在react项目中实现按需加载？

##### **6.2.2.1 index.js**

index.js

**import** React, { Component, Suspense } **from** "react";**import** ReactDOM **from** "react-dom";**import** Loading **from** "./components/Loading";/\* function lazy(loadFunction) {

return class LazyComponent extends React.Component {

state = { Comp: null };

componentDidMount() {

loadFunction().then((result) => {

this.setState({ Comp: result.default });

});

}

render() {

let Comp = this.state.Comp;

return Comp ? <Comp {...this.props} /> : null;

}

};

} \*/**const** AppTitle = React.lazy(() =>

**import**(/\* webpackChunkName: "title" \*/ "./components/Title")

);

**class** **App** **extends** **Component** {

**constructor**(){

**super**();

**this**.state = {visible:false};

}

show(){

**this**.setState({ visible: true });

};

render() {

**return** (

<>

{this.state.visible && (

<Suspense fallback={<Loading />}>

<AppTitle />

</Suspense>

)}

<button onClick={this.show.bind(this)}>加载</button>

</>

);

}

}

ReactDOM.render(<App />, document.querySelector("#root"));

##### **6.2.2.2 Loading.js**

src\components\Loading.js

**import** React, { Component, Suspense } **from** "react";**export** **default** (props) => {

**return** <p>Loading</p>;

};

##### **6.2.2.3 Title.js**

src\components\Title.js

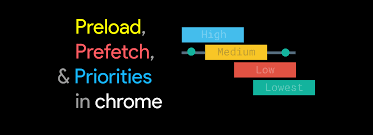
**import** React, { Component, Suspense } **from** "react";**export** **default** props=>{

**return** <p>Title</p>;

}

#### **6.2.3 preload(预先加载)**

* preload通常用于本页面要用到的关键资源，包括关键js、字体、css文件
* preload将会把资源得下载顺序权重提高，使得关键数据提前下载好,优化页面打开速度
* 在资源上添加预先加载的注释，你指明该模块需要立即被使用
* 一个资源的加载的优先级被分为五个级别,分别是
  + Highest 最高
  + High 高
  + Medium 中等
  + Low 低
  + Lowest 最低
* 异步/延迟/插入的脚本（无论在什么位置）在网络优先级中是 Low
* [preload-webpack-plugin](https://github.com/vuejs/preload-webpack-plugin)
* [preload-webpack-plugin](https://www.npmjs.com/package/preload-webpack-plugin)



<link rel="preload" **as**="script" href="utils.js">

**import**(

`./utils.js`

/\* webpackPreload: true \*/

/\* webpackChunkName: "utils" \*/

)

#### **6.2.4 prefetch(预先拉取)**

* prefetch 跟 preload 不同，它的作用是告诉浏览器未来可能会使用到的某个资源，浏览器就会在闲时去加载对应的资源，若能预测到用户的行为，比如懒加载，点击到其它页面等则相当于提前预加载了需要的资源

<link rel="prefetch" href="utils.js" as="script">

button.addEventListener('click', () => {

**import**(

`./utils.js`

/\* webpackPrefetch: true \*/

/\* webpackChunkName: "utils" \*/

).then(result => {

result.default.log('hello');

})

});

#### **6.2.5 preload vs prefetch**

* preload 是告诉浏览器页面必定需要的资源，浏览器一定会加载这些资源
* 而 prefetch 是告诉浏览器页面可能需要的资源，浏览器不一定会加载这些资源
* 所以建议：对于当前页面很有必要的资源使用 preload,对于可能在将来的页面中使用的资源使用 prefetch

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

<link rel="prefetch" href="prefetch.js?k=1" as="script">

<link rel="prefetch" href="prefetch.js?k=2" as="script">

<link rel="prefetch" href="prefetch.js?k=3" as="script">

<link rel="prefetch" href="prefetch.js?k=4" as="script">

<link rel="prefetch" href="prefetch.js?k=5" as="script">

</head>

<body>

</body>

<link rel="preload" href="preload.js" as="script">

</html>

#### **6.2.6 提取公共代码**

* [common-chunk-and-vendor-chunk](https://github.com/webpack/webpack/tree/master/examples/common-chunk-and-vendor-chunk)
* [split-chunks-plugin](https://www.webpackjs.com/plugins/split-chunks-plugin)
* 怎么配置单页应用?怎么配置多页应用?

##### **6.2.6.1 为什么需要提取公共代码**

* 大网站有多个页面，每个页面由于采用相同技术栈和样式代码，会包含很多公共代码，如果都包含进来会有问题
* 相同的资源被重复的加载，浪费用户的流量和服务器的成本；
* 每个页面需要加载的资源太大，导致网页首屏加载缓慢，影响用户体验。
* 如果能把公共代码抽离成单独文件进行加载能进行优化，可以减少网络传输流量，降低服务器成本

##### **6.2.6.2 如何提取**

* 基础类库，方便长期缓存
* 页面之间的公用代码
* 各个页面单独生成文件

##### **6.2.6.3 splitChunks**

###### **6.2.6.3.1 module chunk bundle**

* module：就是js的模块化webpack支持commonJS、ES6等模块化规范，简单来说就是你通过import语句引入的代码
* chunk: chunk是webpack根据功能拆分出来的，包含三种情况
  + 你的项目入口（entry）
  + 通过import()动态引入的代码
  + 通过splitChunks拆分出来的代码
* bundle：bundle是webpack打包之后的各个文件，一般就是和chunk是一对一的关系，bundle就是对chunk进行编译压缩打包等处理之后的产出

###### **6.2.6.3.2 默认配置**

webpack.config.js

**const** HtmlWebpackPlugin = require('html-webpack-plugin');

{

entry: {

page1: "./src/page1.js",

page2: "./src/page2.js",

page3: "./src/page3.js",

},

optimization: {

splitChunks: {

chunks: "all", //默认作用于异步chunk，值为all/initial/async

minSize: 0, //默认值是30kb,代码块的最小尺寸

minChunks: 1, //被多少模块共享,在分割之前模块的被引用次数

maxAsyncRequests: 3, //限制异步模块内部的并行最大请求数的，说白了你可以理解为是每个import()它里面的最大并行请求数量

maxInitialRequests: 5, //限制入口的拆分数量

name: true, //打包后的名称，默认是chunk的名字通过分隔符（默认是～）分隔开，如vendor~

automaticNameDelimiter: "~", //默认webpack将会使用入口名和代码块的名称生成命名,比如 'vendors~main.js'

cacheGroups: {

//设置缓存组用来抽取满足不同规则的chunk,下面以生成common为例

vendors: {

chunks: "all",

test: /node\_modules/, //条件

priority: -10, ///优先级，一个chunk很可能满足多个缓存组，会被抽取到优先级高的缓存组中,为了能够让自定义缓存组有更高的优先级(默认0),默认缓存组的priority属性为负值.

},

default: {

chunks: "all",

minSize: 0, //最小提取字节数

minChunks: 2, //最少被几个chunk引用

priority: -20,

reuseExistingChunk: false

}

},

runtimeChunk:true

},

plugins:[

**new** HtmlWebpackPlugin({

template:'./src/index.html',

chunks:["page1"],

filename:'page1.html'

}),

**new** HtmlWebpackPlugin({

template:'./src/index.html',

chunks:["page2"],

filename:'page2.html'

}),

**new** HtmlWebpackPlugin({

template:'./src/index.html',

chunks:["page3"],

filename:'page3.html'

})

]

}

src\page1.js

**import** module1 **from** "./module1";**import** module2 **from** "./module2";**import** $ **from** "jquery";console.log(module1, module2, $);**import**(/\* webpackChunkName: "asyncModule1" \*/ "./asyncModule1");

src\page2.js

**import** module1 **from** "./module1";**import** module2 **from** "./module2";**import** $ **from** "jquery";console.log(module1, module2, $);

src\page3.js

**import** module1 **from** "./module1";**import** module3 **from** "./module3";**import** $ **from** "jquery";console.log(module1, module3, $);

src\module1.js

console.log("module1");

src\module2.js

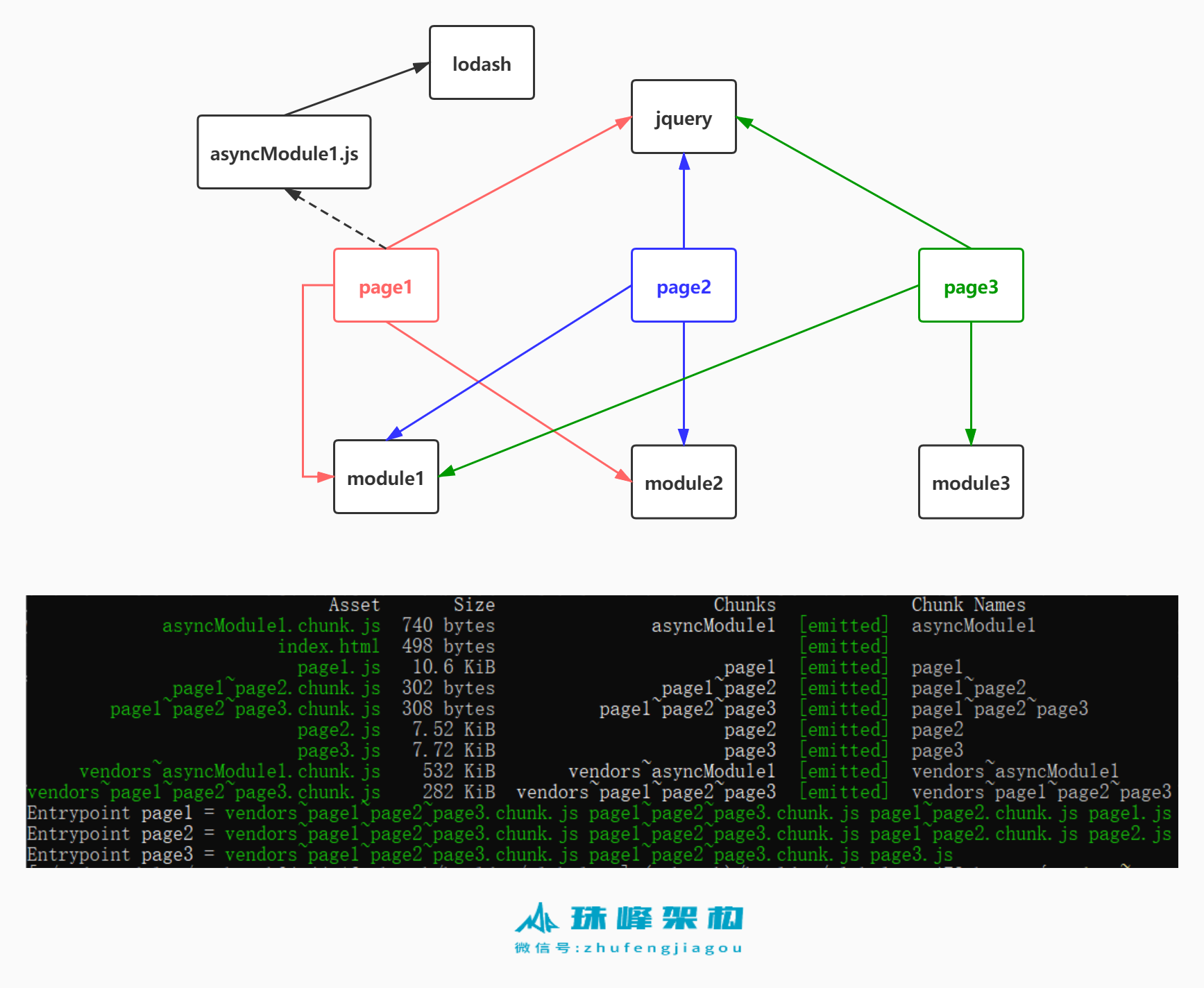
console.log("module2");

src\module3.js

console.log("module3");

src\asyncModule1.js

**import** \_ **from** 'lodash';console.log(\_);



## **7.开启 Scope Hoisting**

* Scope Hoisting 可以让 Webpack 打包出来的代码文件更小、运行的更快， 它又译作 "作用域提升"，是在 Webpack3 中新推出的功能。
* 初webpack转换后的模块会包裹上一层函数,import会转换成require
* 代码体积更小，因为函数申明语句会产生大量代码
* 代码在运行时因为创建的函数作用域更少了，内存开销也随之变小
* 大量作用域包裹代码会导致体积增大
* 运行时创建的函数作用域变多，内存开销增大
* scope hoisting的原理是将所有的模块按照引用顺序放在一个函数作用域里，然后适当地重命名一些变量以防止命名冲突
* 这个功能在mode为production下默认开启,开发环境要用 webpack.optimize.ModuleConcatenationPlugin插件
* 也要使用ES6 Module,CJS不支持

hello.js

**export** **default** 'Hello';

index.js

**import** str **from** './hello.js';console.log(str);

输出的结果main.js

"./src/index.js":

(**function**(module, \_\_webpack\_exports\_\_, \_\_webpack\_require\_\_) {

\_\_webpack\_require\_\_.r(\_\_webpack\_exports\_\_);**var** hello = ('hello');console.log(hello);

})

函数由两个变成了一个，hello.js 中定义的内容被直接注入到了 main.js 中

## **8.利用缓存**

* webpack中利用缓存一般有以下几种思路：
  + babel-loader开启缓存
  + 使用cache-loader

### **8.1 babel-loader**

* Babel在转义js文件过程中消耗性能较高，将babel-loader执行的结果缓存起来，当重新打包构建时会尝试读取缓存，从而提高打包构建速度、降低消耗

{

test: /\.js$/,

exclude: /node\_modules/,

use: [{

loader: "babel-loader",

options: {

cacheDirectory: true

}

}]

},

### **8.2 cache-loader**

* 在一些性能开销较大的 loader 之前添加此 loader,以将结果缓存到磁盘里
* 存和读取这些缓存文件会有一些时间开销,所以请只对性能开销较大的 loader 使用此 loader

npm i cache-loader -D

**const** loaders = ['babel-loader'];module.exports = {

module: {

rules: [

{

test: /\.js$/,

use: [

'cache-loader',

...loaders

],

include: path.resolve('src')

}

]

}

}

### **8.3 oneOf**

* 每个文件对于rules中的所有规则都会遍历一遍，如果使用oneOf就可以解决该问题，只要能匹配一个即可退出。(注意：在oneOf中不能两个配置处理同一种类型文件)

module.exports = {

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

//优先执行

enforce: 'pre',

loader: 'eslint-loader',

options: {

fix: true

}

},

{

// 以下 loader 只会匹配一个

oneOf: [

...,

{},

{}

]

}

]

}

}