



webpack性能优化

- 优化开发体验
- 优化输出质量

优化开发体验

- 提升效率
- 优化构建速度
- 优化使用体验

优化输出质量

- 优化要发布到线上的代码，减少用户能感知到的加载时间
- 提升代码性能，性能好，执行就快

缩小文件范围

优化loader配置

- test include exclude三个配置项来缩小loader的处理范围
- 推荐include

```
include: path.resolve(__dirname, "./src"),
```

优化resolve.modules配置

resolve.modules用于配置webpack去哪些目录下寻找第三方模块，默认是['node_modules']

寻找第三方模块，默认是在当前项目目录下的node_modules里面去找，如果没有找到，就会去上一级目录../node_modules找，再没有会去../../node_modules中找，以此类推，和Node.js的模块寻找机制很类似。

如果我们的第三方模块都安装在了项目根目录下，就可以直接指明这个路径。

```
module.exports={
  resolve:{
    modules: [path.resolve(__dirname, "./node_modules")]
  }
}
```

优化resolve.alias配置

resolve.alias配置通过别名来将原导入路径映射成一个新的导入路径

拿react为例，我们引入的react库，一般存在两套代码

- cjs
 - 采用commonJS规范的模块化代码
- umd
 - 已经打包好的完整代码，没有采用模块化，可以直接执行

默认情况下，webpack会从入口文

件./node_modules/bin/react/index开始递归解析和处理依赖的文件。我们可以直接指定文件，避免这处的耗时。

```
alias: {
  "@": path.join(__dirname, "./pages"),
  react: path.resolve(
    __dirname,
    "./node_modules/react/umd/react.production.min.js"
  ),
  "react-dom": path.resolve(
    __dirname,
    "./node_modules/react-dom/umd/react-dom.production.min.js"
  )
}
```

优化resolve.extensions配置

resolve.extensions在导入语句没带文件后缀时，webpack会自动带上后缀后，去尝试查找文件是否存在。

默认值：

```
extensions:['.js','.json','.jsx','.ts']
```

- 后缀尝试列表尽量的小
- 导入语句尽量带上后缀。

使用静态资源路径publicPath(CDN)

CDN通过将资源部署到世界各地，使得用户可以就近访问资源，加快访问速度。要接入CDN，需要把网页的静态资源上传到CDN服务上，在访问这些资源时，使用CDN服务提供的URL。

```
##webpack.config.js
output:{
  publicPath: '//cdnURL.com', //指定存放JS文件的CDN地址
}
```

- 咱们公司得有cdn服务器地址
- 确保静态资源文件的上传与否

css文件的处理

- 使用less或者sass当做css技术栈

```
$ npm install less less-loader --save-dev
```

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

- 使用postcss为样式自动补齐浏览器前缀
 - <https://caniuse.com/>

```
npm i postcss-loader autoprefixer -D
```

##新建postcss.config.js

```
module.exports = {
  plugins: [
    require("autoprefixer")({
      overrideBrowserslist: ["last 2 versions",
">1%"]
    })
  ]
};
```

##index.less

```
body {
  div {
    display: flex;
    border: 1px red solid;
  }
}
```

##webpack.config.js

```
{
  test: /\.less$/,
```

```
include: path.resolve(__dirname, "../src"),
use: [
  "style-loader",
  {
    loader: "css-loader",
    options: {}
  },
  "less-loader",
  "postcss-loader"
],
},
```

如果不做抽取配置，我们的 `css` 是直接打包进 `js` 里面的，我们希望能单独生成 `css` 文件。因为单独生成css,css可以和js并行下载，提高页面加载效率

借助MiniCssExtractPlugin 完成抽离css

```
npm install mini-css-extract-plugin -D
```

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

```
{
  test: /\.scss$/,
  use: [
    // "style-loader", // 不再需要style-loader, 用
    MiniCssExtractPlugin.loader代替
    MiniCssExtractPlugin.loader,
    "css-loader", // 编译css
  ],
}
```

```

        "postcss-loader",
        "sass-loader" // 编译scss
    ]
},

plugins: [
    new MiniCssExtractPlugin({
        filename: "css/[name]_[contenthash:6].css",
        chunkFilename: "[id].css"
    })
]

```

压缩css

- 借助 optimize-css-assets-webpack-plugin
- 借助cssnano

##安装

```
npm install cssnano -D
```

```
npm i optimize-css-assets-webpack-plugin -D
```

```
const OptimizeCSSAssetsPlugin = require("optimize-css-
assets-webpack-plugin");
```

```

new OptimizeCSSAssetsPlugin({
    cssProcessor: require("cssnano"), //引入cssnano配置压缩
    选项
    cssProcessorOptions: {
        discardComments: { removeAll: true }
    }
})

```

压缩HTML

- 借助html-webpack-plugin

```
new htmlWebpackPlugin({
  title: "京东商城",
  template: "./index.html",
  filename: "index.html",
  minify: {
    // 压缩HTML文件
    removeComments: true, // 移除HTML中的注释
    collapseWhitespace: true, // 删除空白符与换行符
    minifyCSS: true // 压缩内联css
  }
}),
```

优化文件监听的性能

在开启监听模式时，默认情况下会监听配置的 Entry 文件和所有 Entry 递归依赖的文件，在这些文件中会有很多存在于 node_modules 下，因为如今的 Web 项目会依赖大量的第三方模块，所以在大多数情况下我们都不可能去编辑 node_modules 下的文件，而是编辑自己建立的源码文件，而一个很大的优化点就是忽略 node_modules 下的文件，不监听它们。


```
module.export = {
  watchOptions : {
    //不监听的 node_modules 目录下的文件
    ignored : /node_modules/,
  }
}
```

采用这种方法优化后，Webpack 消耗的内存和 CPU 将会大大减少。

development vs Production模式区分打包

```
npm install webpack-merge -D
```

案例

```
const merge = require("webpack-merge")
const commonConfig = require("../webpack.common.js")
const devConfig = {
  ...
}

module.exports = merge(commonConfig, devConfig)

//package.js
"scripts":{
  "dev":"webpack-dev-server --config
./build/webpack.dev.js",
  "build":"webpack --config ./build/webpack.prod.js"
}
```

基于环境变量区分

- 借助cross-env

```
npm i cross-env -D
```

package里面配置命令脚本，传入参数

```
##package.json
```

```
"test": "cross-env NODE_ENV=test webpack --config  
./webpack.config.test.js",
```

在webpack.config.js里拿到参数

```
process.env.NODE_ENV
```

```
//外部传入的全局变量  
module.exports = (env)=>{  
  if(env && env.production){  
    return merge(commonConfig,prodConfig)  
  }else{  
    return merge(commonConfig,devConfig)  
  }  
}
```

```
//外部传入变量  
scripts:" --env.production"
```

tree Shaking

webpack2.x开始支持 tree shaking概念，顾名思义，"摇树"，清除无用 css,js(Dead Code)

Dead Code 一般具有以下几个特征

- 代码不会被执行，不可到达
- 代码执行的结果不会被用到
- 代码只会影响死变量（只写不读）
- Js tree shaking只支持ES module的引入方式！！！！，

Css tree shaking

```
npm i glob-all purify-css purifycss-webpack --save-dev

const PurifyCSS = require('purifycss-webpack')
const glob = require('glob-all')
plugins:[
  // 清除无用 css
  new PurifyCSS({
    paths: glob.sync([
      // 要做 CSS Tree Shaking 的路径文件
      path.resolve(__dirname, './src/*.html'), // 请注意，我们同样需要对 html 文件进行 tree shaking
      path.resolve(__dirname, './src/*.js')
    ])
  })
]
```

JS tree shaking

只支持import方式引入，不支持commonjs的方式引入

案例：

```
//expo.js
export const add = (a, b) => {
  return a + b;
};

export const minus = (a, b) => {
  return a - b;
};
```

```
//index.js
import { add } from "../expo";
add(1, 2);
```

```
//webpack.config.js
optimization: {
  usedExports: true // 哪些导出的模块被使用了，再做打包
}
```

只要mode是production就会生效，development的tree shaking是不生效的，因为webpack为了方便你的调试

可以查看打包后的代码注释以辨别是否生效。

生产模式不需要配置，默认开启

副作用

```
//package.json
"sideEffects":false //正常对所有模块进行tree shaking , 仅生产模式有效, 需要配合usedExports
```

或者 在数组里面排除不需要tree shaking的模块

```
"sideEffects":["*.css",'@babel/polyfill']
```

代码分割 code Splitting

打包完后, 所有页面只生成了一个bundle.js

- 代码体积变大, 不利于下载
- 没有合理利用浏览器资源

```
import _ from "lodash";

console.log(_.join(['a', 'b', 'c', '****']))
```

假如我们引入一个第三方的工具库，体积为1mb，而我们的业务逻辑代码也有1mb，那么打包出来的体积大小会在2mb

导致问题：

体积大，加载时间长

业务逻辑会变化，第三方工具库不会，所以业务逻辑一变更，第三方工具库也要跟着变。

其实code Splitting概念 与 webpack并没有直接的关系，只不过webpack中提供了一种更加方便的方法供我们实现代码分割

基于 <https://webpack.js.org/plugins/split-chunks-plugin/>

```
optimization: {
  splitChunks: {
    chunks: "all", // 所有的 chunks 代码公共的部分分离出来
    成为一个单独的文件
  },
},
```

```
optimization: {
  splitChunks: {
    chunks: 'async', //对同步 initial, 异步 async, 所有的
    模块有效 all
  }
}
```

```

minSize: 30000, //最小尺寸, 当模块大于30kb
maxSize: 0, //对模块进行二次分割时使用, 不推荐使用
minChunks: 1, //打包生成的chunk文件最少有几个chunk引用了
这个模块
maxAsyncRequests: 5, //最大异步请求数, 默认5
maxInitialRequests: 3, //最大初始化请求书, 入口文件同步
请求, 默认3
automaticNameDelimiter: '-', //打包分割符号
name: true, //打包后的名称, 除了布尔值, 还可以接收一个函数
function
cacheGroups: { //缓存组
  vendors: {
    test: /[\\/]node_modules[\\/]$/,
    name: "vendor", // 要缓存的 分隔出来的 chunk 名称
    priority: -10 //缓存组优先级 数字越大, 优先级越高
  },
  other: {
    chunks: "initial", // 必须三选一: "initial" |
    "all" | "async" (默认就是async)
    test: /react|lodash/, // 正则规则验证, 如果符合就
提取 chunk,
    name: "other",
    minSize: 30000,
    minChunks: 1,
  },
  default: {
    minChunks: 2,
    priority: -20,
    reuseExistingChunk: true //可设置是否重用该chunk
  }
}
}
}
}

```


使用下面配置即可：

```
optimization:{  
  //帮我们自动做代码分割  
  splitChunks:{  
    chunks:"all", //默认是支持异步，我们使用all  
  }  
}
```

DllPlugin插件打包第三方类库 优化构建性能

Dll动态链接库 其实就是做缓存

.dll文件称为动态链接库，在windows系统会经常看到。

百度百科:<https://baike.baidu.com/item/.dll/2133451?fr=aladdin>

项目中引入了很多第三方库，这些库在很长的一段时间内，基本不会更新，打包的时候分开打包来提升打包速度，而DllPlugin动态链接库插件，其原理就是把网页依赖的基础模块抽离出来打包到dll文件中，当需要导入的模块存在于某个dll中时，这个模块不再被打包，而是去dll中获取。

- 动态链接库只需要被编译一次，项目中用到的第三方模块，很稳定，例如react,react-dom，只要没有升级的需求

webpack已经内置了对动态链接库的支持

- DllPlugin:用于打包出一个个单独的动态链接库文件
- DllReferencePlugin: 用于在主要的配置文件中引入DllPlugin插件打包好的动态链接库文件

新建webpack.dll.config.js文件，打包基础模块

我们在 `index.js` 中使用了第三方库 `react`、`react-dom`，接下来，我们先对这两个库先进行打包。

```
const path = require("path");
const { DllPlugin } = require("webpack");
module.exports = {
  mode: "development",
  entry: {
    react: ["react", "react-dom"] //!! node_modules?
  },
  output: {
    path: path.resolve(__dirname, "./dll"),
    filename: "[name].dll.js",
    library: "react"
  },
  plugins: [
    new DllPlugin({
      // manifest.json文件的输出位置
      path: path.join(__dirname, "./dll", "[name]-manifest.json"),
      // 定义打包的公共vendor文件对外暴露的函数名
      name: "react"
    })
  ]
};
```

在package.json中添加

```
"dev:dll": "webpack --config  
./build/webpack.dll.config.js",
```

运行

```
npm run dev:dll
```

你会发现多了一个dll文件夹，里边有dll.js文件，这样我们就把我们的React这些已经单独打包了

- dll文件包含了大量模块的代码，这些模块被存放在一个数组里。用数组的索引号为ID,通过变量讲自己暴露在全局中，就可以在window.xxx访问到其中的模块
- Manifest.json 描述了与其对应的dll.js包含了哪些模块，以及ID和路径。

接下来怎么使用呢？

要给web项目构建介入动态链接库，需要完成以下事情：

- 将网页依赖的基础模块抽离，打包到单独的动态链接库，一个动态链接库是可以包含多个模块的。
- 当需要导入的模块存在于某个动态链接库中时，不要再次打包，直接使用构建好的动态链接库即可。

```
##webpack.dev.config.js
new DllReferencePlugin({
  manifest: require("../dll/react-manifest.json")
}),
```

- 页面依赖的所有动态链接库都需要被加载。

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>webpack</title>
    <link href="css/main_e2bf39.css" rel="stylesheet"></head>
    <body>
      <div id="app"></div>
      <script type="text/javascript" src="react.dll.js"></script>
      <script type="text/javascript" src="js/main_142e6c.js"></script></body>
</html>
```

手动添加使用，体验不好，这里推荐使用add-asset-html-webpack-plugin插件帮助我们做这个事情。

安装一个依赖 `npm i add-asset-html-webpack-plugin`，它会将我们打包后的 `dll.js` 文件注入到我们生成的 `index.html` 中。在 `webpack.base.config.js` 文件中进行更改。

```
new AddAssetHtmlWebpackPlugin({
  filepath: path.resolve(__dirname,
    '../dll/react.dll.js') // 对应的 dll 文件路径
}),
```

运行：

```
npm run dev
```

这个理解起来不费劲，操作起来很费劲。所幸，在Webpack5中已经不用它了，而是用 `HardSourceWebpackPlugin`，一样的优化效果，但是使用却及其简单

```
const HardSourceWebpackPlugin = require('hard-source-
webpack-plugin')

const plugins = [
  new HardSourceWebpackPlugin()
]
```

使用happypack并发执行任务

运行在 Node.之上的Webpack是单线程模型的，也就是说Webpack需要一个一个地处理任务，不能同时处理多个任务。 **Happy Pack** 就能让Webpack做到这一点，它将任务分解给多个子进程去并发执行，子进程处理完后再将结果发送给主进程。从而发挥多核 CPU 电脑的威力。

```
npm i -D happypack

var happyThreadPool = HappyPack.ThreadPool({ size: 5
});

//const happyThreadPool = HappyPack.ThreadPool({ size:
os.cpus().length })

// webpack.config.js
rules: [
  {
    test: /\.jsx?$/,
    exclude: /node_modules/,
    use: [
      {
        // 一个loader对应一个id
        loader: "happypack/loader?id=babel"
      }
    ]
  },
  {
    test: /\.css$/,
    include: path.resolve(__dirname, "./src"),
    use: ["happypack/loader?id=css"]
  },
]
```

```
]

//在plugins中增加
plugins:[
  new HappyPack({
    // 用唯一的标识符id，来代表当前的HappyPack是用来处理一类
    特定的文件
    id:'babel',
    // 如何处理.js文件，用法和Loader配置中一样
    loaders:['babel-loader?cacheDirectory'],
    threadPool: happyThreadPool,
  }),
  new HappyPack({
    id: "css",
    loaders: ["style-loader", "css-loader"]
  }),
]
```

<https://github.com/webpack-contrib/mini-css-extract-plugin/issues/273>

<https://github.com/amireh/happypack/issues/242>

多入口打包配置通用方案

```
entry:{
  index:"./src/index",
  list:"./src/list",
  detail:"./src/detail"
}

new HtmlWebpackPlugin({
  title: "index.html",
  template: path.join(__dirname,
    "./src/index/index.html"),
  filename:"index.html",
  chunks:[index]
})
```

1.目录结构调整

- src
 - index
 - index.js
 - index.html
 - list
 - index.js
 - index.html
 - detail
 - index.js
 - index.html
- 2.使用 glob.sync 第三方库来匹配路径


```
npm i glob -D
```

```
const glob = require("glob")
```

//MPA多页面打包通用方案

```
const setMPA = () => {  
  const entry = {};  
  const htmlWebpackPlugins = [];  
  
  return {  
    entry,  
    htmlWebpackPlugins  
  };  
};  
  
const { entry, htmlWebpackPlugins } = setMPA();
```

```
const setMPA = () => {  
  const entry = {};  
  const htmlWebpackPlugins = [];  
  
  const entryFiles = glob.sync(path.join(__dirname,  
    "./src/*/index.js"));  
  
  entryFiles.map((item, index) => {
```

```

    const entryFile = entryFiles[index];
    const match =
entryFile.match(/src\/(.*)\/index\.js$/);
    const pageName = match && match[1];
    entry[pageName] = entryFile;
    htmlWebpackPlugin.push(
      new htmlWebpackPlugin({
        title: pageName,
        template: path.join(__dirname,
`src/${pageName}/index.html`),
        filename: `${pageName}.html`,
        chunks: [pageName],
        inject: true
      })
    );
  });
  return {
    entry,
    htmlWebpackPlugin
  };
};

const { entry, htmlWebpackPlugin } = setMPA();

module.exports = {
  entry,
  output:{
    path: path.resolve(__dirname, "../dist"),
    filename: "[name].js"
  }
  plugins: [

```

```
// ...  
...htmlWebpackPlugin//展开数组  
]  
}
```

文件监听

轮询判断文件的最后编辑时间是否变化，某个文件发生了变化，并不会立刻告诉监听者，先缓存起来

webpack开启监听模式，有两种

1.启动webpack命令式 带上`--watch` 参数，启动监听后，需要手动刷新浏览器

```
scripts:{  
  "watch":"webpack --watch"  
}
```

2.在配置文件里设置 `watch:true`

```
watch: true, //默认false,不开启  
//配合watch,只有开启才有作用  
watchOptions: {  
  //默认为空，不监听的文件或者目录，支持正则  
  ignored: /node_modules/,
```

```
//监听到文件变化后，等300ms再去执行，默认300ms，
aggregateTimeout: 300,
//判断文件是否发生变化是通过不停的询问系统指定文件有没有变化，
默认每秒问1次
poll: 1000 //ms
}
```

如何自己编写一个Loader

自己编写一个Loader的过程是比较简单的，

Loader就是一个函数，**声明式函数**，不能用箭头函数

拿到源代码，作进一步的修饰处理，再返回处理后的源码就可以了

官方文档：<https://webpack.js.org/contribute/writing-a-loader/>

接口文档：<https://webpack.js.org/api/loaders/>

简单案例

- 创建一个替换源码中字符串的loader

```
//index.js
console.log("hello kkb");
```

```
//replaceLoader.js
module.exports = function(source) {
  console.log(source, this, this.query);
  return source.replace('kkb', '开课吧')
};
```

//需要用声明式函数，因为要上到上下文的this,用到this的数据，该函数接受一个参数，是源码

- 在配置文件中使用的loader

```
//需要使用node核心模块path来处理路径
const path = require('path')
module: {
  rules: [
    {
      test: /\.js$/,
      use: path.resolve(__dirname,
        "../loader/replaceLoader.js")
    }
  ]
},
```

- 如何给loader配置参数，loader如何接受参数？
 - this.query
 - loader-utils

```
//webpack.config.js
module: {
  rules: [
    {
      test: /\.js$/,
      use: [
        {
          loader: path.resolve(__dirname,
            './loader/replaceLoader.js'),
          options: {
            name: "开课吧"
          }
        }
      ]
    }
  ],
},
```

```
//replaceLoader.js
//const loaderUtils = require("loader-utils");//官方推荐
处理loader,query的工具
```

```
module.exports = function(source) {
  //this.query 通过this.query来接受配置文件传递进来的参数

  //return source.replace("kkb", this.query.name);
  const options = loaderUtils.getOptions(this);
  const result = source.replace("kkb", options.name);
  return source.replace("kkb", options.name);
}
```

```
}
```

- **this.callback**: 如何返回多个信息, 不止是处理好的源码呢, 可以使用this.callback来处理

```
//replaceLoader.js
const loaderUtils = require("loader-utils");//官方推荐处理
loader, query的工具

module.exports = function(source) {
  const options = loaderUtils.getOptions(this);
  const result = source.replace("kkb", options.name);
  this.callback(null, result);
};

//this.callback(
  err: Error | null,
  content: string | Buffer,
  sourceMap?: SourceMap,
  meta?: any
);
```

- **this.async**: 如果loader里面有异步的事情要怎么处理呢

```
const loaderUtils = require("loader-utils");

module.exports = function(source) {
  const options = loaderUtils.getOptions(this);
  setTimeout(() => {
    const result = source.replace("kkb", options.name);

    return result;
  }, 1000);
};

//先用setTimeout处理下试试，发现会报错
```

我们使用this.async来处理，他会返回this.callback

```
const loaderUtils = require("loader-utils");

module.exports = function(source) {
  const options = loaderUtils.getOptions(this);

  //定义一个异步处理，告诉webpack，这个loader里有异步事件，在里面
  //调用下这个异步
  //callback 就是 this.callback 注意参数的使用
  const callback = this.async();
  setTimeout(() => {
    const result = source.replace("kkb", options.name);
    callback(null, result);
  }, 3000);
};
```

- 多个loader的使用 注意顺序

```
//replaceLoader.js
```



```

module.exports = function(source) {
  return source.replace("开课吧", "word");
};
//replaceLoaderAsync.js
const loaderUtils = require("loader-utils");
module.exports = function(source) {
  const options = loaderUtils.getOptions(this);
  //定义一个异步处理, 告诉webpack, 这个loader里有异步事件, 在里面
  调用下这个异步
  const callback = this.async();
  setTimeout(() => {
    const result = source.replace("kkb", options.name);
    callback(null, result);
  }, 3000);
};

//webpack.config.js
module: {
  rules: [
    {
      test: /\.js$/,
      use: [
        path.resolve(__dirname,
"./loader/replaceLoader.js"),
        {
          loader: path.resolve(__dirname,
"./loader/replaceLoaderAsync.js"),
          options: {
            name: "开课吧"
          }
        }
      ]
    }
  ]
}

```

```
        // use: [path.resolve(__dirname,  
        "./loader/replaceLoader.js")]  
    }  
]  
},
```

顺序，自下而上，自右到左

- 处理loader的路径问题

```
resolveLoader: {  
    modules: ["node_modules", "./loader"]  
},  
module: {  
    rules: [  
        {  
            test: /\.js$/,  
            use: [  
                "replaceLoader",  
                {  
                    loader: "replaceLoaderAsync",  
                    options: {  
                        name: "开课吧"  
                    }  
                }  
            ]  
            // use: [path.resolve(__dirname,  
            "./loader/replaceLoader.js")]  
        }  
    ]  
},  
},
```

参考：loader API

<https://webpack.js.org/api/loaders>

如何自己编写一个Plugin

Plugin: 开始打包，在某个时刻，帮助我们处理一些什么事情的机制

plugin要比loader稍微复杂一些，在webpack的源码中，用plugin的机制还是占有非常大的场景，可以说plugin是webpack的灵魂

设计模式

事件驱动

发布订阅

plugin是一个类，里面包含一个apply函数，接受一个参数，compiler

官方文档： <https://webpack.js.org/contribute/writing-a-plugin/>

案例：

- 创建copyright-webpack-plugin.js

```
class CopyrightWebpackPlugin {
  constructor() {
  }

  //compiler: webpack实例
  apply(compiler) {

  }
}
module.exports = CopyrightWebpackPlugin;
```

- 配置文件里使用

```
const CopyrightWebpackPlugin =
require("../plugin/copyright-webpack-plugin");

plugins: [new CopyrightWebpackPlugin()]
```

- 如何传递参数

```
//webpack配置文件
plugins: [
  new CopyrightWebpackPlugin({
    name: "开课吧"
  })
]

//copyright-webpack-plugin.js
class CopyrightWebpackPlugin {
  constructor(options) {
```

```
//接受参数
console.log(options);
}

apply(compiler) {}
}
module.exports = CopyrightWebpackPlugin;
```

- 配置plugin在什么时刻进行

```
class CopyrightWebpackPlugin {
  constructor(options) {
    // console.log(options);
  }

  apply(compiler) {
    //hooks.emit 定义在某个时刻
    compiler.hooks.emit.tapAsync(
      "CopyrightWebpackPlugin",
      (compilation, cb) => {
        compilation.assets["copyright.txt"] = {
          source: function() {
            return "hello copy";
          },
          size: function() {
            return 20;
          }
        };
        cb();
      }
    );
  }
}
```

```
);

//同步的写法

//compiler.hooks.compile.tap("CopyrightWebpackPlugin",
  compilation => {
    // console.log("开始了");
    //});
  }
}
module.exports = CopyrightWebpackPlugin;
```

参考： compiler-hooks

<https://webpack.js.org/api/compiler-hooks>