Step08. 常见插件 Plugins

重点插件 plugin 介绍

插件与模块解析的功能不一样，模块解析是为了导入非 es5 格式 js 或其它资源类型文件，定制了一些 loader。插件是对最后的打包文件进行处理的，也可以理解 loader 是为打包前做准备，plugin 是打包再进行处理

- **官方插件的使用步骤(内置插件 2 步)**
  ① 配置文件中导入 XxxxPlugin， const wp= require(XxxxPlugin)
  ② 在 plugins 这个数组中加入一个插件实例,new wp.XxxxPlugin({对象})

- **第三方插件的使用步骤(第 3 方 3 步，多一次安装)**
  ① 安装(第三方插件要安装) 根目录>cnpm i -D XxxxPlugin
  ② 配置文件中导入插件 const XxxxPlugin = require('xxxx-plugin')
  ③ 在 plugins 这个数组中加入一个插件实例,new XxxxPlugin({对象})

- **官方插件有**

可以在配置中打印查看

```
const webpack = require('webpack')
console.log(webpack ) //这里可以看到哪些是 webpack 内置的插件
```

待讲插件清单

01: webpack.BannerPlugin 加注释
02: terser-webpack-plugin 代码缩小
03: html-webpack-plugin 生成 html 页
04: 以前的 extract-text-webpack-plugin，mini-css-extract-plugin 提取 css 等
05. DefinePlugin //定义一个全局常量，如 new wp.DefinePlugin({BJ: JSON.stringify('北京'),})，在待打包的 js 文件中可以直接使用，如在./src/main.js 中 console.log('我是在配置文件中定义的'+BJ)
06.Dllplugins

- **BannerPlugin**

作用: 在打包的文件中添加注释

```
 //这是 webpack 内置的插件，所以不用 require 导入，但是对于第三方插件要先导入
plugins: [
        new webpack.BannerPlugin({
            banner: '永远要记得，成功的决心远胜于任何东西'
        })

    ]
```

● 打包复制

作用：在 webpack 中拷贝文件和文件夹

安装： yarn add copy-webpack-plugin --dev

from	定义要拷贝的源文件	from: __dirname+'/src/components'
to	定义要拷贝到的目标文件夹	to: __dirname+'/dist'
toType	file 或者 dir	可选，默认是文件
force	强制覆盖前面的插件	可选，默认是文件
context		可选，默认 base context 可用 specific context
flatten	只拷贝指定的文件	可以用模糊匹配
ignore	忽略拷贝指定的文件	可以模糊匹配

插件引入和配置：

```
const CopyWebpackPlugin = require('copy-webpack-plugin');
new CopyWebpackPlugin([
    {
            from: path.resolve(__dirname, './static'),
            to: buildPath+'/static',
            ignore: ['.*']
            }
])
```

● js 压缩插件

作用：压缩 js 代码，减小体积
这个插件不是内置的，要先安装

不引荐运用 webpack-parallel-uglify-plugin
项目基础处于没人保护的阶段，issue 没人处置惩罚，pr 没人兼并。

引荐运用 terser-webpack-plugin

terser-webpack-plugin 是一个运用 terser 紧缩 js 的 webpack 插件。

紧缩是宣布前处置惩罚最耗时候的一个步骤，如果是你是在 webpack 4 中，只需几行代码，即可加快你的构建宣布速率。

安装：因为是第三方的 ， 所以需要安装 ， https://github.com/webpack-contrib/terser-webpack-plugin

yarn add terser-webpack-plugin -D
或者 npm install terser-webpack-plugin --save-dev

用法

```
module.exports = {
  optimization: {
    minimize: true,
    minimizer: [
      new TerserPlugin({
        parallel: 4,     //多线程
      }),
    ],
  },
};
```

- **抽离 css 样式,**

抽离 css 的目的是防止将样式打包在 js 中方便缓存静态资源

方式一：extract-text-webpack-plugin

安装： yarn add extract-text-webpack-plugin^4.0.0-beta.0 --dev

注意：如果用这个 需要 指定版本："extract-text-webpack-plugin": "^4.0.0-beta.0"

配置：

```
const ExtractTextPlugin = require("extract-text-webpack-plugin");
//module 部分
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
```

```
            use: ExtractTextPlugin.extract({
              fallback: "style-loader",
              use: "css-loader"
            })
          }
        ]
    },
//插件部分
    plugins: [
        new ExtractTextPlugin("styles.css"),
    ]
}
```

use:指需要什么样的 loader 去编译文件,这里由于源文件是.css 所以选择 css-loader
fallback:编译后用什么 loader 来提取 css 文件

方式二：mini-css-extract-plugin

相比 extract-text-webpack-plugin：

- 异步加载
- 没有重复的编译（性能）
- 更容易使用
- 特定于 CSS

安装： yarn add mini-css-extract-plugin -D

抽出 css ：

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

modle 部分

```
//mini-css-extract-plugin 示例
 {
 test: /\.(sa|sc|c)ss$/,
 use: [
 isProduction ? MiniCssExtractPlugin.loader:'style-loader',
 'css-loader',
 'postcss-loader',
 'sass-loader',
 ],
 },
```

插件部分：

```
// 这里的配置和 webpackOptions.output 中的配置相似
```

// 即可以通过在名字前加路径，来决定打包后的文件存在的路径

```
new MiniCssExtractPlugin({
 filename: isProduction ? 'styles/'+outFilename+'.css' : 'styles/[name].[contenthash].css',
 chunkFilename: isProduction ? 'styles/[id].[hash].css' : 'styles/[id].css',
 });
```

[id]和[name]在 webpack 中被称做 placeholder
用来在 webpack 构建之后动态得替换内容的（本质上是正则替换）。
chunkFilename 是构建应用的时候生成的。

- **压缩 css: optimize-css-assets-webpack-plugin**

✧ ·生产环境的配置，默认开启 tree-shaking 和 js 代码压缩；
✧ ·通过 optimize-css-assets-webpack-plugin 插件可以对 css 进行压缩，与此同时，必须指定 js 压缩插件（例子中使用 terser-webpack-plugin 插件），否则 webpack 不再对 js 文件进行压缩；
✧ 设置 optimization.splitChunks.cacheGroups，可以将 css 代码块提取到单独的文件中。

安装：yarn add optimize-css-assets-webpack-plugin -D

引入插件与配置：

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

module.exports = merge(common, {
   mode: 'production',
   optimization: {
      minimizer: [new TerserJSPlugin({}), new OptimizeCSSAssetsPlugin({})],
      splitChunks: {
         cacheGroups: {
            styles: {
               name: 'styles',
               test: /\.css$/,
               chunks: 'all',
               enforce: true,
            },
         }
      },
   }})
```

总结

① 不同环境下的打包，如果出现图片显示不了时（特别是 css 中的图片），请检查 publicPath 的配置。

② mode: 'production'会开启 tree-shaking 和 js 代码压缩，但配置 optimization.minimizer 会使默认的压缩功能失效。所以，指定 css 压缩插件的同时，务必指定 js 的压缩插件。

③ mini-css-extract-plugin 插件，结合 optimization.splitChunks.cacheGroups 配置，可以把 css 代码打包到单独的 css 文件，且可以设置存放路径（通过设置插件的 filename 和 chunkFilename）。

- **生成 json 文件的列表索引插件**

安装： yarn add assets-webpack-plugin --dev

```
var AssetsPlugin = require('assets-webpack-plugin');
//生成 json 文件的列表索引插件
new AssetsPlugin({
    filename: 'assets-resources.json',
    fullPath: false,
    includeManifest: 'manifest',
    prettyPrint: true,
    update: true,
    path: buildPath,
    metadata: {version: 123}
});
```

sourceMap

源代码与打包后的代码的映射关系
在 dev 模式中，默认开启，关闭的话 可以在配置文件里

devtool:"none"

devtool 的介绍：https://webpack.js.org/configuration/devtool#devtool
eval:速度最快
cheap:较快，不用用管列的报错

Module：第三方模块
开发环境推荐
devtool:"cheap-module-eval-source-map"

线上环境可以不不开启：如果要看到一一些错误信息，推荐；
devtool:"cheap-module-source-map"

配置别名快捷方式 resolve

```
resolve: {
       extensions: ['.js', '.vue', '.json'],
       alias: {
         'src': srcPath,
         'styles': srcPath+'/styles',
         'images':srcPath+'/images',
         'config':path.resolve(srcPath,'js/config.js')
       }
     },
```

resolve.alias ： 设置别名
resolve.enforceExtension :默认是 false
如果是 true，将不允许无扩展名(extension-less)文件。如果启用此选项，只有 require('./foo.js') 能够正常工作。
resolve.extensions 自动解析确定的扩展。

设置参数 通过 script

很久以前这样用过：
script:{
"start": "export NODE_ENV='development' && node app.js"

```
    // 在 Mac 和 Linux 上使用 export， 在 windows 上 export 要换成 set
}
```

目前推荐 cross-env

cross-env

安装：yarn add cross-env --dev

注意：cross-env scene=dev　cross-env scene=prod　后面不能添加&&

```
"scripts": {
  "dev": "cross-env scene=dev webpack-dev-server --config webpack.config.js",
  "build": "cross-env scene=prod webpack --mode production --config
webpack.config.js"
  },
```

需求：　根部不同的开发环境，来设置一些变量

```
console.log(process.env.scene);
var isProduction = (process.env.scene=='prod');
```

DefinePlugin

DefinePlugin 允许创建一个在编译时可以配置的全局常量

在配置文件里获取变量并把值设置，DefinePlugin 是 webpack 自带的插件，无法安装

```
new webpack.DefinePlugin({
   'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV),
   'process.env.DEBUG': JSON.stringify(process.env.DEBUG)
})
```

```
pluginsAll.push(new webpack.DefinePlugin({
    'sceneParam': JSON.stringify(process.env.scene),
    'laney':JSON.stringify('laney'),
    'test':'"kkkkk"'
```

```
}));
```

然后在项目中，根据不同的环境来调用不是的样式

```
if(sceneParam=='prod') {
  import './styles/test03.scss';
} else if(sceneParam=='dev'){
  import './styles/test02.less';
}
```

如果这么使用会报错：

`import` 和 `export` 只能在顶级用，不能在代码块中用。否则会报 `'import' and 'export' may only appear at the top level`。

下面的写法可以

```
switch(sceneParam){
   case 'prod':
     console.log('pppp11');
     import('styles/test03.scss');
     break;
   case 'dev':
     import('styles/login.scss');
     break;
 }
```

Require 按需加载

https://segmentfault.com/a/1190000013630936
https://github.com/jiang43605/multiple-mini-css-extract-plugin

EnvironmentPlugin

EnvironmentPlugin 是一个通过 DefinePlugin 来设置 process.env 环境变量的快捷方式。
用法：
new webpack.EnvironmentPlugin(['NODE_ENV', 'DEBUG'])
不同于 DefinePlugin，默认值将被 EnvironmentPlugin 执行 JSON.stringify。

上面的写法和下面这样使用 DefinePlugin 的效果相同：

```
new webpack.DefinePlugin({
  'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV),
  'process.env.DEBUG': JSON.stringify(process.env.DEBUG)
})

new webpack.EnvironmentPlugin({
  NODE_ENV: 'development',
// 除非有定义 process.env.NODE_ENV，否则就使用 'development'
  DEBUG: false})
```