了解webpack插件？写过webpack插件吗？

前言

面试官: **了解webpack插件吗？**

> 我：**有了解过一点**

面试官: **那你说说你了解的webpack插件**

> 我：**就是一个类，然后有一个apply函数作为入口函数，接受一个compiler参数，然后就在这个插件内完成具体的功能**

面试官: **写过webpack插件吗？**

> 我：**嗯...，这个...,** 😅

面试官：**em...**

是不是很熟悉的场景，因为有时候我们确实没机会去对项目进行 `webpack` 配置或者去优化 `webpack` 配置，所以就谈不上对 `webpack` `plugin` 、 `loader` 、整体流程有了解，那么在面试的时候当面试官一问到 `webpack` 相关问题的时候，可能就答不上来了

那么面试官为什么想要问 `webpack` 相关的问题？无外乎

1. `webpack` 几乎是目前前端项目构建的标配，所以 `webpack` 的掌握程度与工程化息息相关
2. `webpack` 提供了很多特性和功能，例如多入口、 `chunk` 提取、懒加载等，这些功能可以显著提升前端项目的性能。因此掌握 `webpack` 的各种功能能够让开发者更好地构建出高效、健壮的前端应用
3. `webpack` 够难，有一定的门槛

所以在面试中，会涉及到 `webpack` 的基础概念、配置、原理、性能优化等方面的问题，需要面试者有一定的实践经验和理论知识。同时，面试官也会通过对 `webpack` 相关问题的考察，来评估面试者的技术深度和解决问题的能力。

那么本篇就是帮助面试者，攻克 `webpack` `plugin` 相关的问题，帮助面试者对 `webpack` `plugin` 有一个更好的认识，不仅能够帮助我们面试，更能够帮助我们在日常的开发中更好的使用 `webpack`

看完本篇之后，希望小伙伴面试的时候是这样回答的

面试官：**了解webpack插件吗？**

> **你会这么回答**

> - **了解，webpack的大致流程是，初始化配置 => 从入口模块开始解析 => 经过loader处理 => 遍历ast => 找到依赖 => 继续解析依赖，直到所有的子模块都解析完成 => 优化chunk => 生成assets => 根据assets生成最终的产物**
> - **而这个过程中webpack不能满足所有的场景，为了webpack更加灵活与拓展，设计了插件机制，webpack的插件机制，基于 `tapable` 实现，而 `tapable` 提供了多种类型的hook，比如同步串行hook，异步并行hook等**
> - **然后webpack目前提供的hook大概有5类，第一类是compiler上的hook，这类hook是大流程上的节点；第二类complation上的hook，这类hook是构建模块实例、优化chunk等流程上的节点；第三类NormalModuleFactory上的hook，这类hook是模块创建、超找模块等流程上节点；第四类是JavascriptParser上的hook，这类hook就是遍历ast流程上的节点；第五类就是ContextModuleFactory上的hook与NormalModuleFactory上的hook类似，但是用的少**
> - **最后一个插件以apply方法作为入口函数，入口函数会接受一个compiler参数，接下来就是根据webpack在compiler，compilation等对象上爆料的hooks上注册callback，在callback内完成拓展功能**

面试官：**写过webpack插件吗?**

> **你会这么回答**
>
> - **写过，我写过约定式路由插件(任何自己写的插件)，写这个插件的目的是为了解决手写routes配置文件，做到自动生成routes文件，以高开发效率**
> - **为了使生成routes文件生效，我选择在webpack编译之前的hooks内完成routes文件的生成，而编译之前的hooks有，environment、initialize等hook，我这里选择initialize hook，这一个同步串行hook**
> - **最后在initialize hook上注册callback，在callback内读取目录及相关的配置，生成路由配置文件**

下面开始正文，在这篇文章中，我们将探讨如何编写 `webpack` 插件。 `webpack` 是一个非常流行的 `JavaScript` 模块打包工具，使用它可以将多个 `JavaScript` 模块打包成一个或多个 `bundle` 文件。 `webpack` 有很多功能和特点，其中一项非常重要的特点就是其可扩展性，通过编写 `webpack` 插件可以实现各种自定义功能。插件就是 `webpack` 的基石。所以掌握 `webpack` 插件，能过让我们成为一个更熟练的 `webpack` 配置工程师，同时也能解决面试中碰到的 `webpack` `plugin` 问题

本文将从以下几个方面介绍 `webpack` 插件：

- `webpack` **插件是什么**？帮助我们了解 `webpack` 为什么设计插件机制
- **项目内如何使用插件**？帮助我们快速搭建项目的 `webpack` 配置
- **常用** `webpack` **插件及原理**？帮助我们了解常用的 `webpack` 插件内部是怎么写的，哪些值得我们借鉴
- **编写自己的** `webpack` **插件**？根据我们的业务场景，编写一些提升开发效率的插件

插件是什么

为什么设计插件机制

`webpack` 的设计理念是万物皆模块，然后将所有模块打包成一个或者多个 `bundle` 文件，但是这并不意味着 `webpack` 自身就能满足所有功能，比如 `chunk` 提取、代码压缩、 `css` 文件提取等等，所以为了使 `webpack` 更加了灵活与便于拓展， `webpack` 设计了插件机制，通过插件可以实现各种拓展功能，满足业务需求

怎么定义插件

`webpack` 规定插件必须是一个 `javascript` 对象，且对象上必须有一个 `apply` 方法，这个 `apply` 方法会在 `webpack` 进行编译的时候进行调用，插件定义如下所示

```
const pluginName = 'ConsoleLogOnBuildWebpackPlugin';

class ConsoleLogOnBuildWebpackPlugin {
  apply() {}
}

module.exports = ConsoleLogOnBuildWebpackPlugin;

```

到这里我们已经已经知道怎么定义一个 `webpack plugin` ，虽然这个插件能过被执行，但是不会有任何作用，原因是插件内没有做任何处理，那么插件内怎么去介入 `webpack` 的构建流程，影响最终的构建结果呢？其实 `webpack` 在执行插件 `apply` 方法的时候，会传入一个 `compiler` 对象，这个 `compiler` 对象上会暴露 `webpack` 构建阶段不同节点的 `hook` ，然后在这个 `hook` 上允许注册对应的 `callback` ，如下所示

```
const pluginName = 'ConsoleLogOnBuildWebpackPlugin';

class ConsoleLogOnBuildWebpackPlugin {
  apply(compiler) {
    // compiler.hooks 包含一组节点，这些节点，都允许注册对应的callback,webpack执行过程中，会调用这个callback，然后在这个callback
    // 调用的时候传入一些参数，然后callback内借助传入的参数修改webpack构建过程中的一些内容，最终影响webpack的构建结果
    compiler.hooks.run.tap(pluginName, (compilation) => {
      console.log('The webpack build process is starting!');
    });
  }
```

```
}

module.exports = ConsoleLogOnBuildWebpackPlugin;

```

`webpack` 构建流程可以简单总结如下图所示

那么 `webpack` 只要在处理的过程中，在各个阶段，执行我们注册的 `callback` ，那么我们的插件就可以介入 `webpack` 构建流程，我们从 `webpack` 源码看下， `webpack` 是怎么触发我们注册的 `hook callback` 执行的

```
const {
        SyncHook,
        SyncBailHook,
        AsyncParallelHook,
        AsyncSeriesHook
} = require("tapable");

this.hooks = Object.freeze({
  /** @type {SyncHook<[]>} */
  initialize: new SyncHook([]),
  /** @type {SyncHook<[CompilationParams]>} */
  compile: new SyncHook(["params"]),
  /** @type {AsyncParallelHook<[Compilation]>} */
  make: new AsyncParallelHook(["compilation"]),
  ...
});


compile(callback) {
  const params = this.newCompilationParams();
  // 调用beforeCompile hook，传入参数params，callback
  this.hooks.beforeCompile.callAsync(params, err => {
    if (err) return callback(err);

    // 调用compile hook，传入参数params
    this.hooks.compile.call(params);

    const compilation = this.newCompilation(params);

    // 调用compile hook 传入参数compilation，callback
```

```
    this.hooks.make.callAsync(compilation, err => {
      if (err) return callback(err);

      // 调用finishMake hook 传入参数compilation, callback
      this.hooks.finishMake.callAsync(compilation, err => {
        if (err) return callback(err);

        process.nextTick(() => {
          compilation.finish(err => {
            if (err) return callback(err);

            compilation.seal(err => {
              if (err) return callback(err);

              // 调用afterCompile hook 传入参数compilation, callback
              this.hooks.afterCompile.callAsync(compilation, err => {
                if (err) return callback(err);
                return callback(null, compilation);
              });
            });
          });
        });
      });
    });
  });
}

```

从源码我们可以看到，`webpack` 在编译的过程中，会在各个节点调用对应的 `hook`，从而执行对应的 `callback`，以达到功能拓展的目的

目前 `webpack` 暴露的 `hook` 有5类

- `compiler` 类 `hook`
  - `run`
  - `compiler`
  - `compilation`
  - `shouldEmit`
  - `emit`
  - `done`
- `compilation` 类 `hook`
  - `buildModule`

    - `succeedModule`
    - `finishModules`
    - `normalModuleLoader`
- `ContextModuleFactory` 类 `hook`
    - `beforeResolve`
    - `afterResolve`
    - `createModule`
    - `module`
- `NormalModuleFactory` 类 `hook`
    - `beforeResolve`
    - `afterResolve`
    - `createModule`
    - `module`
- `JavascriptParser` 类 `hook`
    - `import`
    - `call`

更多 `hook` 直接查看文档即可compiler hooks

我们只需要每个 `hook` 代表执行的哪个阶段，并且该 `hook` 属于哪种类型的 `hook` 即可在插件中通过该 `hook` 注册 `callback` ，完成对应的逻辑，如所示 我想在编译模块之前做些事情


```
1 javascript
2 复制代码
3 compiler.hooks.beforeRun.tapAsync('MyPlugin', (compiler, callback) => {  /* 处理
  逻辑 */        callback()});
```

比如我想在模块解析之前做些事情

finishModules

`AsyncSeriesHook`

Called when all modules have been built without errors.

- Callback Parameters: `modules`

```
1 javascript
2 复制代码
3 compiler.hooks.compilation.tap('MyPlugin', (compilation, compilationParams) =>
  {   compilation.hooks.finishModules.tapAsync(
  'SourceMapDevToolModuleOptionsPlugin',      (modules, callback) => {        //
  modules 就是包含所有module处理完之后的module实例        callback()      }    );});
```

比如我想在所有模块处理之后做一些事情

beforeResolve

`AsyncSeriesBailHook`

Called when a new dependency request is encountered. A dependency can be ignored by returning `false`. Otherwise, it should return `undefined` to proceed.

- Callback Parameters: `resolveData`

```
1 compiler.hooks.compilation.tap('MyPlugin', (compilation, compilationParams) =>
  {
2    compilation.hooks.finishModules.tapAsync(
3       'SourceMapDevToolModuleOptionsPlugin',
4       (modules, callback) => {
5
6         NormalModuleFactory.hooks.someHook.tap(/* ... */)
7       }
8     );
9 });
10
```

到这里我们基本上知道，应该怎么去注册 `callback` 了，但是我们仔细看的话， `hook` 的类型有很多种，比如

- `Synchook`：同步串行钩子

- `AsyncSerieshook`:异步串行钩子
- `AsyncParallelhook`：异步并发钩子
- `SyncBailhook`：同步熔断钩子，也就是当有一个返回非`undefined`的值时，会中断后续`callback`的执行

那为什么会有这么多种类型，不要这种类型行不行，比如我们注册`callback`，将这些注册的`callback`放到一个数组里，然后执行数组内所有的`callback`不行吗？伪代码如下所示

```
compiler.hooks.compilation.tap('MyPlugin', callback1)

compiler.hooks.compilation.tap('MyPlugin', callback2)

compiler.hooks.compilation.tap('MyPlugin', callback2)

// 用数组来介绍callback
handles = [callback1, callback2, callback2]

// 然后到执行节点的时候，按照注册顺序执行callback
handles.forEach((handle) => {
  handle(params)
})

```

这种简单的发布订阅方式实现的插件机制不行吗？不是不行，而是`webpack`场景更复杂，单纯的通过注册顺序执行无法满足所有需求，所以才设计了更多的插件执行模式

- 比如`callback`串行执行
- 比如`callback`并行执行
- 比如`callback`串行执行，将前一个的结果，传给后一个
- 比如`callback`串行执行，只要有一个返回不是`undefined`的值，就立马返回，中断后面的`callback`执行等等

`webpack`把这一套插件机制封装成了一个单独的npm包tapable，tapable提供的`hook`如下所示

- `SyncHook` 同步钩子
- `SyncBailHook` 同步熔断钩子
- `SyncWaterfallHook` 同步流水钩子
- `SyncLoopHook` 同步循环钩子
- `AsyncParalleHook` 异步并发钩子
- `AsyncParalleBailHook` 异步并发熔断钩子

- `AsyncSeriesHook` 异步串行钩子
- `AsyncSeriesBailHook` 异步串行熔断钩子
- `AysncSeriesWaterfallHook` 异步串行流水钩子

从功能对 `Hook` 分类

Type	Function
Waterfall	同步方法，传值下一个函数
Bail	当函数有任何返回值，则在当前执行函数停止
Loop	监听函数返回true则继续循环，返回undefined表示循环结束
Series	串行钩子
Paralle	并行钩子

从类型对 `Hook` 分类

Async*	Sync*
绑定: tapAsync/ tapPromise/ tap	绑定：tap
执行: callAsync/ promise	执行：call

tapable的实现原理也相当有意思，使用的是字符串拼接 + new Function的方式生成函数体，感兴趣的可以自己通过vscode断点调试的方式去看看源码

所以到这里我们可以知道，写一个 `webpack` 插件，需要

- 插件是一个 `javascript` 对象，且该对象必须包含入口 `apply` 方法
- `webpack` 暴露了5类 `hook` ，我们必须要知道我们要介入的节点是哪个 `hook`

- 注册 `callback` 的方式，有同步与异步的方式，一定要根据文档的提示 `hook` 类型来决定

项目内如何使用插件

上面我们已经知道插件应该怎么定义，那么这一节，我们将会学习怎么在项目内使用插件，本文的案例都是围绕 `webpack5` 来讲的

假设我们写的插件如下所示 使用类定义我们的插件

```
class MyPlugin {
  constructor(opts) {
    this.options = opts
  }
  apply(compiler) {
    compiler.hooks.compilation.tap('MyPlugin', (compilation, compilationParams) => {
       compilation.hooks.finishModules.tapAsync(
          'MyPlugin',
          (modules, callback) => {
                  for (let item of [...modules]) {
              //         打印每个module的路径
              console.log('item', item.resource);
            }
            callback()
          }
        );
    });
  }
}

```

使用 `function` 定义我们的插件

```
function MyPlugin2(opts) {
  this.options = opts
}

MyPlugin2.prototype.apply = function(compiler) {
  compiler.hooks.compilation.tap('MyPlugin', (compilation, compilationParams) => {
    compiler.hooks.finishModules.tapAsync(
       'MyPlugin',
       (modules, callback) => {
         for (let item of [...modules]) {
```

```
            // 打印每个module的路径
            console.log('item 2', item.resource);
          }
          callback()
        }
      );
  });
}

```

一份简单的 `webpack` 配置

```
const path = require('path');
const { ProgressPlugin } = require('webpack')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const Myplugin = require('./myPlugin')


const config = {
  mode: 'production',
  output: {
    path: path.join(__dirname, '../dist'),
    filename: 'js/[name].[chunkhash].js',
    chunkFilename: 'chunk/[name].[chunkhash].js',
    publicPath: './'
  },
  plugins: [
    // 使用我们自己的插件
    new Myplugin({
      test: 1
    }),
    // 使用我们自己的插件
    new Myplugin2({
      test: 2
    }),
    // 使用webpack提供的插件
    new ProgressPlugin(
      {
        percentBy: 'entries',
      }
    ),
    // 使用社区提供的插件
    new HtmlWebpackPlugin(
      {
        filename: 'index.html',
```

```
        template: path.join(__dirname, '../public/index.html'),
      }
    ),
  ],
  entry: {
    app: path.join(__dirname, '../src/app')
  },
}

module.exports = config

```

所以其实插件使用只要注意两点

- 插件本身要是 `javascript` 对象，且包含 `apply` 方法
- 插件通过 `webpack` 的 `plugins` 字段传入

常用插件

以 `react` 项目为例，我们一个项目可能会包含哪些插件，这些插件分别是怎么实现功能的，我们通过对这些常用插件的了解，进一步掌握 `webpack` 插件原理

基础插件

对插件原理的阅读可能需要一定的 `webpack` 基础，可选择跳过原理部分

html-webpack-plugin@5.5.1 html处理插件

使用

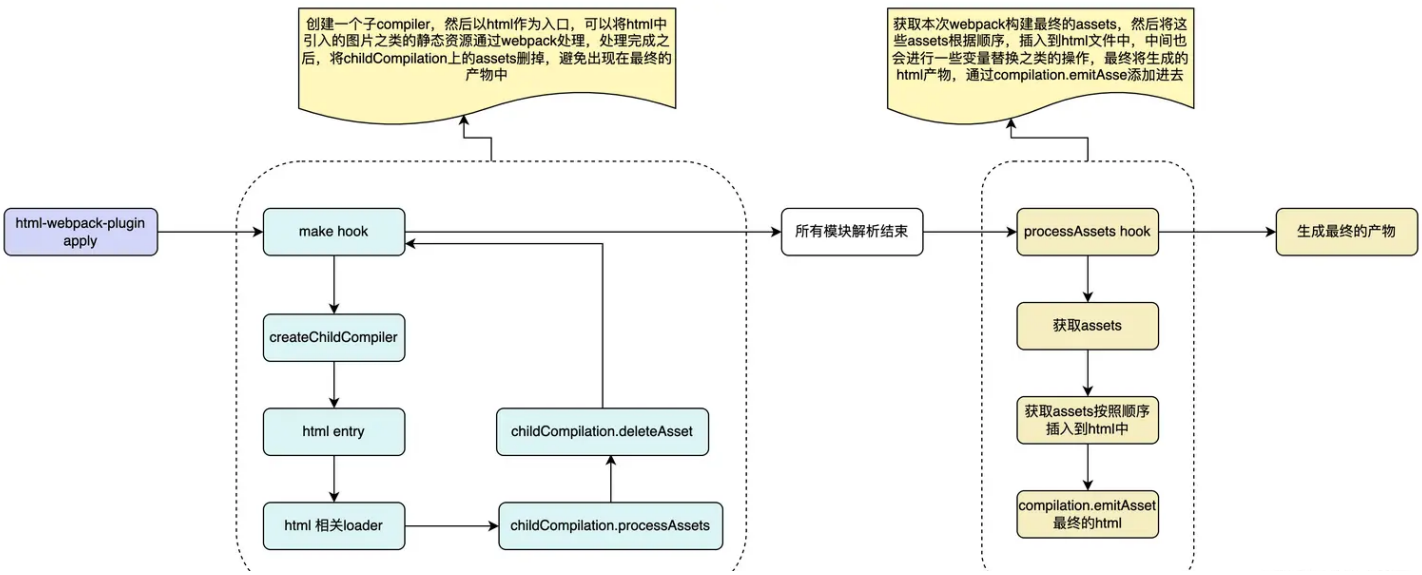
```
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  plugins: [
    new HtmlWebpackPlugin(
      {
        filename: 'index.html',
        template: path.join(__dirname, '../public/index.html'),
        minify: {
          collapseWhitespace: true,
          minifyJS: true,
          html5: true,
          minifyCSS: true,
          removeComments: true,
          removeTagWhitespace: false
```

```
        },
      }
    ),
  ]
}

```

原理 `html-webpack-plugin` 主要做的事情是

- 在 `make` 钩子的 `callback` 内创建 `child compiler` ，然后通过 `child compiler` 去处理传入的 `filename` ，最终 `child compiler` 处理完之后，会将 `asset` 保存到一个对象上，等待最终处理，这里需要注意下 `child compiler` 内是删除了 `asset` ，这样做的原因是，后续还需要对 `html` 进一步处理，比如插入 `js` 、 `css` 等，避免 `child compiler` 处里完之后直接赋值给了父 `compiler` 的 `assets` 里面
- 在父 `compilation` 的 `processAssets` 钩子的 `callback` 内，对之前 `child compiler` 处理完之后生成的 `asset` ，做进一步处理，比如添加 `js` 、 `css` 、处理 `publicPath` 、处理一些变量转化等，然后最终在通过 `compilation.emitAsset` 输出最终的 `html` 文件

流程图如下图所示



下面是精简过的代码

```
apply (compiler) {
  // 在compiler实例创建之后，注册第一个hook
  compiler.hooks.initialize.tap('HtmlWebpackPlugin', () => {

    entryOptions.forEach((instanceOptions) => {
      hookIntoCompiler(compiler, instanceOptions, this);
    });
  });
}
```

```
10
11
12 function hookIntoCompiler (compiler, options, plugin) {
13   compiler.hooks.thisCompilation.tap('HtmlWebpackPlugin',
14     (compilation) => {
15       // 使用子编译器处理入口html
16       compiler.hooks.make.tapAsync(
17       'PersistentChildCompilerSingletonPlugin',
18       (mainCompilation, callback) => {
19         // 创建子compiler，处理html文件，这里使用子编译器的目的是，html内可能还需要处理
   src等静态资源路径等
20         const childCompiler =
   mainCompilation.createChildCompiler(compilerName, outputOptions, [
21           new webpack.library.EnableLibraryPlugin('var')
22         ]);
23
24         childCompiler.hooks.thisCompilation.tap('HtmlWebpackPlugin',
   (compilation) => {
25           compilation.hooks.processAssets.tap(
26             {
27               name: 'HtmlWebpackPlugin',
28               stage: Compilation.PROCESS_ASSETS_STAGE_ADDITIONS
29             },
30             (assets) => {
31               temporaryTemplateNames.forEach((temporaryTemplateName) => {
32                 if (assets[temporaryTemplateName]) {
33                   // 用extractedAssets保存html经过webpack处理之后的内容
34                   extractedAssets.push(assets[temporaryTemplateName]);
35                   // 删除子compiler上的asset，避免赋值到父compiler的asset上
36                   compilation.deleteAsset(temporaryTemplateName);
37                 }
38               });
39             }
40           );
41         });
42       })
43
44       // webpack处理assets时注册callback
45       compilation.hooks.processAssets.tapAsync(
46         {
47           name: 'HtmlWebpackPlugin',
48           stage:
49           // 开始优化assets的时候执行
50           webpack.Compilation.PROCESS_ASSETS_STAGE_OPTIMIZE_INLINE
51         },
52         (compilationAssets, callback) => {
```

```
53            // compilationAssets包含所有最终要生成的文件，html-webpack-plugin在这个
    callback
54            // 内回去拿入口文件，以及处理publicpath等事情，还有html本身提供的一些替换等逻
    辑，保证最终生成html的时候，html内有准确的js、css地址
55            const entryNames = Array.from(compilation.entrypoints.keys());
56            const filteredEntryNames = filterChunks(entryNames, options.chunks,
    options.excludeChunks);
57            const sortedEntryNames = sortEntryChunks(filteredEntryNames,
    options.chunksSortMode, compilation);
58
59            const htmlPublicPath = getPublicPath(compilation, options.filename,
    options.publicPath);
60
61            const assets = htmlWebpackPluginAssets(compilation,
    sortedEntryNames, htmlPublicPath);
62            const emitHtmlPromise = injectedHtmlPromise
63              .then(html => {
64                const filename = options.filename.replace(/\
    [templatehash([^\]]*)\]/g, require('util').deprecate(
65                  (match, options) => `[contenthash${options}]`,
66                  '[templatehash] is now [contenthash]')
67                );
68
69                // 将最终的html文件输出的assets中去
70                compilation.emitAsset(replacedFilename.path, new
    webpack.sources.RawSource(html, false), replacedFilename.info);
71                return replacedFilename.path;
72              }).then(() => null));
73
74            emitHtmlPromise.then(() => {
75              callback();
76            });
77          });
78      });
79  }
80
```

其实简单总结就是，创建一个 `child compiler` 处理 `html` ，然后在父 `compiler` 处理 `assets` 的时候，在将 `child compiler` 处理的 `html` 内容经过一系列处理之后，通过 `compilation.emitAsset` 输出到最终的 `assets` 里面

mini-css-extract-plugin@2.7.5 css提取插件

使用

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
module.exports = {
  module: {
    rules: [
      {
        test: /\.(css|less|s[a|c]ss)(\?.*)?$/,
        use: [
          {
            loader: MiniCssExtractPlugin.loader
          }
        ]
      },
    ]
  }
  plugins: [
    new MiniCssExtractPlugin(
      {
        filename: 'css/[name].[contenthash].css',
        chunkFilename: 'css/[name].[contenthash].css'
      }
    )
  ]
}

```

原理 `mini-css-extract-plugin` 插件主要做的事情是

- 在 `loader` 钩子的 `callback` 内向 `loaderContext` 添加属性，用来配合 `MinicssExtractPlugin.loader` 是否需要使用 `importModule` 方法
- 在 `thisCompilation` 钩子注册 `callback`，完成 `cssModuleFactory`、`cssDependencyTemplate` 的注册，便于正确解析 `css` 模块与生成最终的 `css` 内容
- `MinicssExtractPlugin.loader` 内如果支持 `importModule` 方法，则会用 `importModule` 方法处理 `css` 模块，如果不支持 `importModule` 则会创建 `child compiler`，然后通过 `child compiler` 去处理 `css` 文件，`child compiler` 处理完之后，删除 `child compiler` 内的 `asset`，然后最终在父 `compiler` 统一处理所有的 `css module`
- 在 `renderManifest` 钩子注册 `callback`，目的是构造合并 `chunk` 内 `css module` 的 `manifest`，即将所有的 `css module` 合并到对应的 `css asset` 里面，创建出 `css asset`

更多内容可以查看面试官：生产环境构建时为什么要提取css文件?

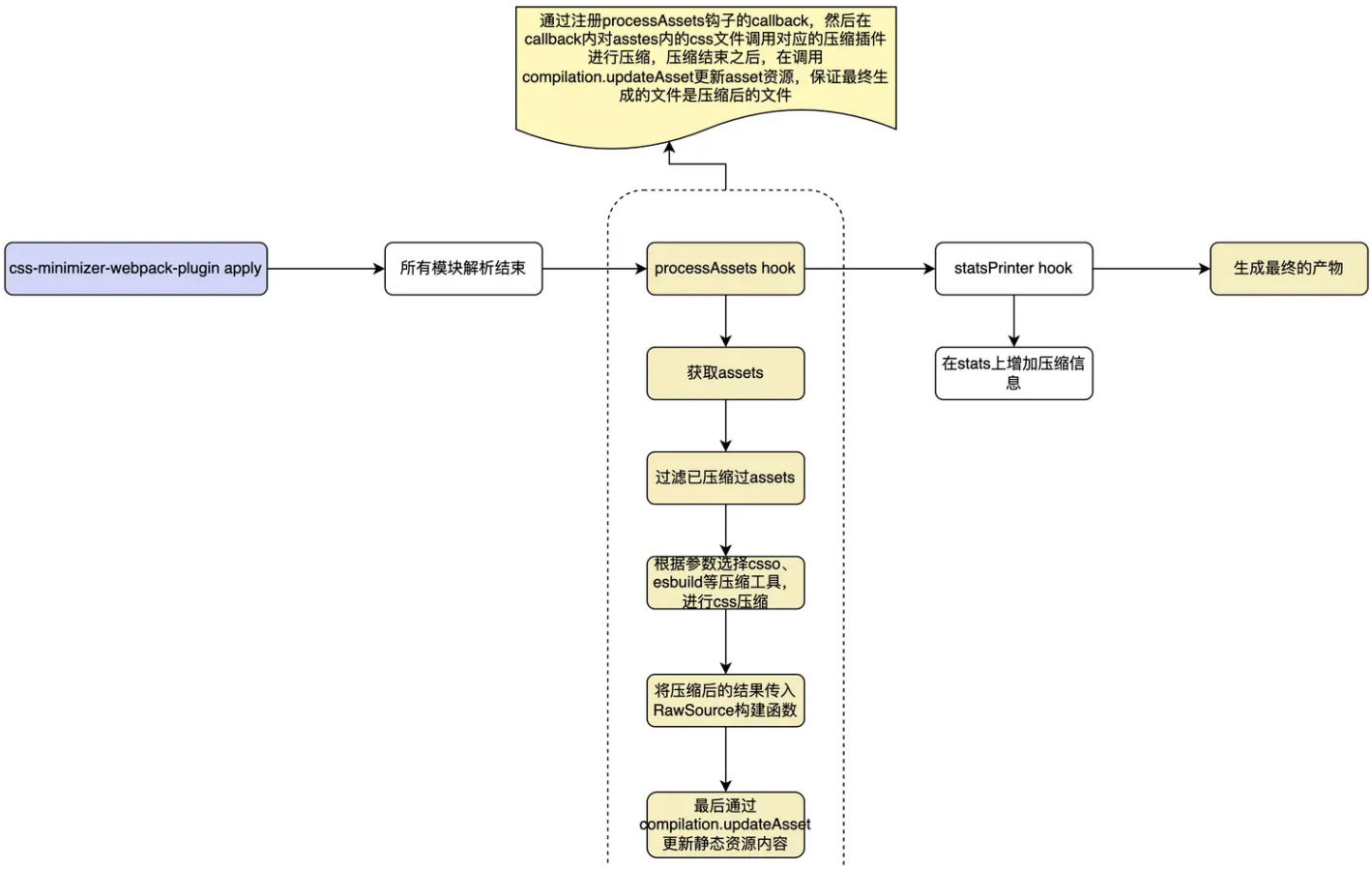
css-minimizer-webpack-plugin@5.0.0 css压缩插件

使用

```
javascript
复制代码
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");module.exports = {  optimization: {    minimize: true,    minimizer: [      new CssMinimizerPlugin(),    ]  },}
```

原理 `cssMinimizerPlugin` 插件的核心原理就是通过注册 `processAssets` 钩子的 `callback`，然后在 `callback` 内对 `asstes` 内的 `css` 文件调用对应的压缩插件进行压缩，压缩结束之后，在调用 `compilation.updateAsset` 更新 `asset` 资源，保证最终生成的文件是压缩后的文件

流程图如下图所示



精简后的代码

```
class CssMinimizerPlugin {

  constructor(options) {
          // 传入使用的压缩插件，默认使用cssnano插件进行压缩，里面还有csso、esbuild、swc、lightningCss等压缩方式
    const {
      minify = cssnanoMinify,
    } = options || {};

```

```
    this.options = {
      minimizer: {
        implementation: /** @type {MinimizerImplementation<T>} */minify,
      }
    };
  }


  async optimize(compiler, compilation, assets, optimizeOptions) {
    const cache = compilation.getCache("CssMinimizerWebpackPlugin");
    let numberOfAssetsForMinify = 0;
    // 遍历assets，过滤已压缩过与不需要压缩的asset
    const assetsForMinify = await Promise.all(Object.keys(typeof assets ===
"undefined" ? compilation.assets : assets).filter(name => {
                        return true;
    }).map(async name => {
      const {
        info,
        source
      } = /** @type {Asset} */
      compilation.getAsset(name);
      return {
        name,
        info,
        inputSource: source,
        output,
        cacheItem
      };
    }));

          // 借助webpack的RawSource生成最终的source
    const {
      SourceMapSource,
      RawSource
    } = compiler.webpack.sources;
    const scheduledTasks = [];
    for (const asset of assetsForMinify) {
      scheduledTasks.push(async () => {
        const {
          name,
          inputSource,
          cacheItem
        } = asset;
        let {
          output
        } = asset;
        if (!output) {
```

```
55            let input;
56            /** @type {RawSourceMap | undefined} */
57            let inputSourceMap;
58            const {
59              source: sourceFromInputSource,
60              map
61            } = inputSource.sourceAndMap();
62            input = sourceFromInputSource;
63
64            const options = {};
65            let result;
66            try {
67              // 调用压缩方法，比如teser等
68              result = await minifyWorker(options);
69            } catch (error) {
70
71              compilation.errors.push( /** @type {WebpackError} */
72
73              return;
74            }
75
76            for (const item of result.outputs) {
77              // 将压缩后的结果，传入RawSource构造函数
78              output.source = new RawSource(item.code);
79            }
80            await cacheItem.storePromise({});
81          }
82
83          const newInfo = {
84            minimized: true
85          };
86          const {
87            source
88          } = output;
89          // 最终通过compilation.updateAsset方法更新asset内容
90          compilation.updateAsset(name, source, newInfo);
91        });
92      }
93      const limit = getWorker && numberOfAssetsForMinify > 0 ? /** @type
    {number} */numberOfWorkers : scheduledTasks.length;
94      await throttleAll(limit, scheduledTasks);
95
96    }
97
98    apply(compiler) {
99      const pluginName = this.constructor.name;
```

```
    const availableNumberOfCores = CssMinimizerPlugin.getAvailableNumberOfCores(this.options.parallel);
    compiler.hooks.compilation.tap(pluginName, compilation => {
      // 在processAssets hook注册callback
      compilation.hooks.processAssets.tapPromise({
        name: pluginName,
        stage: compiler.webpack.Compilation.PROCESS_ASSETS_STAGE_OPTIMIZE_SIZE,
        additionalAssets: true
      }, assets => this.optimize(compiler, compilation, assets, {
        availableNumberOfCores
      }));

      compilation.hooks.statsPrinter.tap(pluginName, stats => {
        stats.hooks.print.for("asset.info.minimized").tap("css-minimizer-webpack-plugin", (minimized, {
          green,
          formatFlag
        }) =>
        // eslint-disable-next-line no-undefined
        minimized ? /** @type {Function} */green( /** @type {Function} */formatFlag("minimized")) : "");
      });
    });
  }
}

```

terser-webpack-plugin@5.3.7 js压缩插件

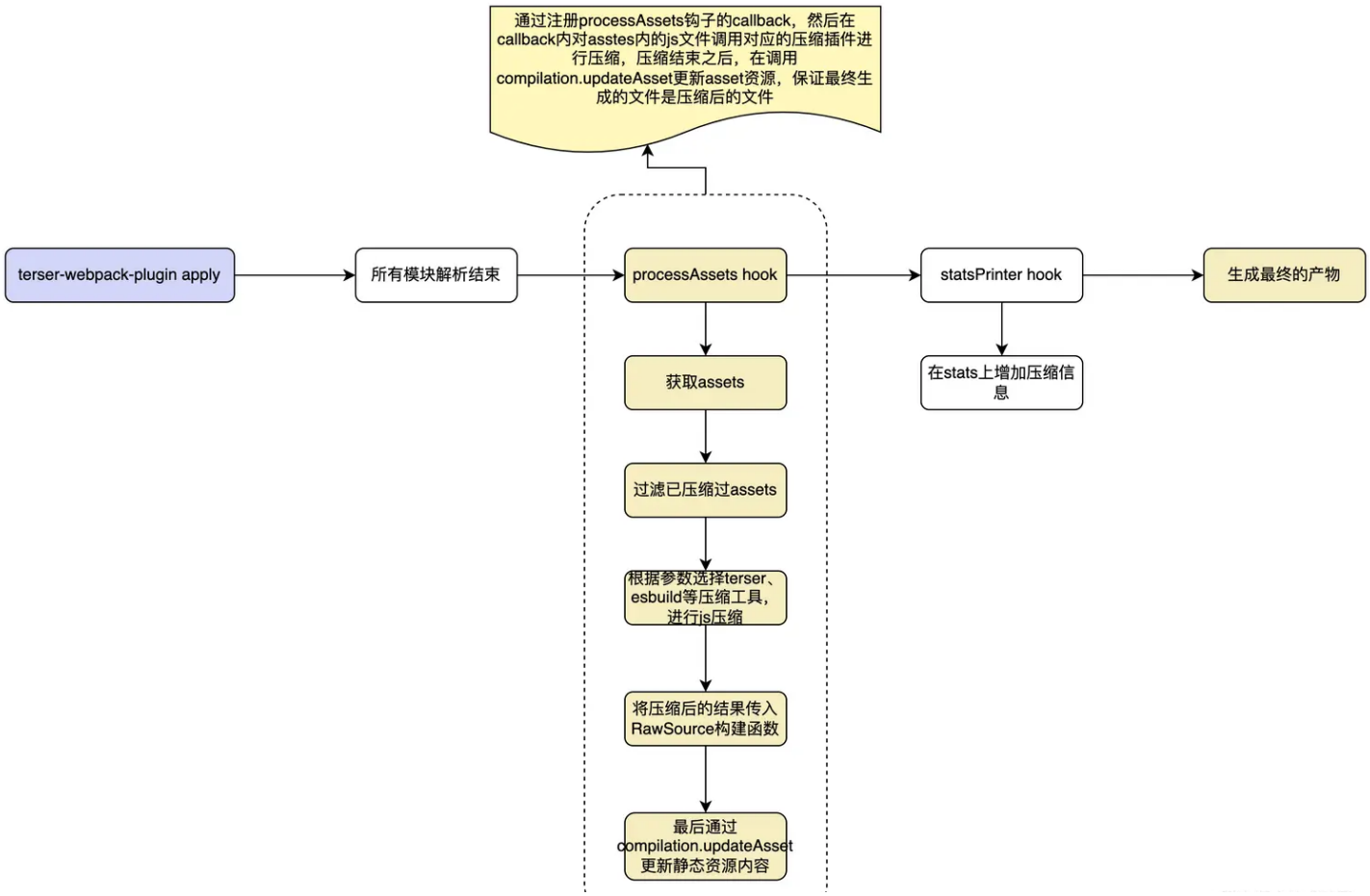
使用

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  optimization: {
    minimize: true,
    minimizer: [
      new TerserPlugin({
        parallel: false,
        terserOptions: {
          // https://github.com/webpack-contrib/terser-webpack-plugin#teroptions
        },
      }),
    ]
```

```
  },
}

```

原理 `TerserPlugin` 插件的核心原理就是通过注册 `processAssets` 钩子的 `callback` ，然后在 `callback` 内对 `asstes` 内的 `js` 文件调用对应的压缩插件进行压缩，压缩结束之后，在调用 `compilation.updateAsset` 更新 `asset` 资源，保证最终生成的文件是压缩后的文件

流程图如下所示



精简后的代码如下所示

```
class TerserPlugin {
  constructor(options) {

    // 选择压缩的插件，默认时terser，里面还有uglifyJs、swc、esbuild，可以通过传入的参数控制
    const {
      minify = /** @type {MinimizerImplementation<T>} */terserMinify,
    } = options || {};

    this.options = {
      minimizer: {
```

```
        implementation: minify,
        options: terserOptions
      }
    };
  }

  async optimize(compiler, compilation, assets, optimizeOptions) {
    const cache = compilation.getCache("TerserWebpackPlugin");
    let numberOfAssets = 0;
    // 遍历assets，然后返回一个新数组
    const assetsForMinify = await Promise.all(Object.keys(assets).filter(name
=> {
      return true;
    }).map(async name => {
      const {
        info,
        source
      } = compilation.getAsset(name);
      return {
        name,
        info,
        inputSource: source,
        output,
        cacheItem
      };
    }));

    const {
      SourceMapSource,
      ConcatSource,
      RawSource
    } = compiler.webpack.sources;

    const scheduledTasks = [];
    for (const asset of assetsForMinify) {
      scheduledTasks.push(async () => {
        const {
          name,
          inputSource,
          info,
          cacheItem
        } = asset;
        let {
          output
        } = asset;
        // 表示没有被压缩过
        if (!output) {
```

```
57            let input;
58            let inputSourceMap;
59            const {
60              source: sourceFromInputSource,
61              map
62            } = inputSource.sourceAndMap();
63            input = sourceFromInputSource;
64
65            const options = {};
66
67            try {
68              // 调用压缩插件进行压缩
69              output = await minify(options);
70            } catch (error) {
71
72              compilation.errors.push( /** @type {WebpackError} */
73              return;
74            }
75
76            // 使用webpack提供的RawSource构造函数
77            output.source = new RawSource(output.code);
78
79            await cacheItem.storePromise({});
80          }
81
82
83          /** @type {Record<string, any>} */
84          const newInfo = {
85            minimized: true
86          };
87          const {
88            source,
89            extractedCommentsSource
90          } = output;
91
92          // 更新压缩后的内容
93          compilation.updateAsset(name, source, newInfo);
94        });
95      }
96      const limit = getWorker && numberOfAssets > 0 ? /** @type {number}
    */numberOfWorkers : scheduledTasks.length;
97      await throttleAll(limit, scheduledTasks);
98
99    }
100
101   apply(compiler) {
102     const pluginName = this.constructor.name;
```

```
103     const availableNumberOfCores =
    TerserPlugin.getAvailableNumberOfCores(this.options.parallel);
104     compiler.hooks.compilation.tap(pluginName, compilation => {
105       const hooks =
    compiler.webpack.javascript.JavascriptModulesPlugin.getCompilationHooks(compila
    tion);
106       const data = serialize({
107         minimizer: typeof
    this.options.minimizer.implementation.getMinimizerVersion !== "undefined" ?
    this.options.minimizer.implementation.getMinimizerVersion() || "0.0.0" :
    "0.0.0",
108         options: this.options.minimizer.options
109       });
110
111       // 注册processAssets钩子的callback，在这里压缩assets
112       compilation.hooks.processAssets.tapPromise({
113         name: pluginName,
114         stage: compiler.webpack.Compilation.PROCESS_ASSETS_STAGE_OPTIMIZE_SIZE,
115         additionalAssets: true
116       }, assets => this.optimize(compiler, compilation, assets, {
117         availableNumberOfCores
118       }));
119
120       // 注册statsPrinter钩子的callback，优化控制台输出
121       compilation.hooks.statsPrinter.tap(pluginName, stats => {
122         stats.hooks.print.for("asset.info.minimized").tap("terser-webpack-
    plugin", (minimized, {
123           green,
124           formatFlag
125         }) => minimized ? /** @type {Function} */green( /** @type {Function}
    */formatFlag("minimized")) : "");
126       });
127     });
128   }
129 }
130
```

看完 `cssMinimizerPlugin` 与 `TerserPlugin` 插件之后，发现两个压缩插件实现基本上是一样的

辅助插件

speed-measure-webpack-plugin@1.5.0 耗时统计插件

使用

```
const SpeedMeasurePlugin = require('speed-measure-webpack-plugin')

const smp = new SpeedMeasurePlugin();

module.exports = smp.wrap({
  plugins: []
})

```

原理 `speed-measure-webpack-plugin` 注册 `compile` 、 `done` 钩子的 `callback` 统计 `webpack` 本次构建耗时，注册 `build-module` 、 `succeed-module` 钩子的 `callback` ，统计 `loader` 链耗时

精简代码如下所示

```
apply(compiler) {
    tap(compiler, "compile", () => {
      this.addTimeEvent("misc", "compile", "start", { watch: false });
    });
    tap(compiler, "done", () => {
      clear();
      this.addTimeEvent("misc", "compile", "end", { fillLast: true });
    });
}

tap(compilation, "build-module", (module) => {
  // 获取模块的userRequest
  const name = getModuleName(module);
  if (name) {
    this.addTimeEvent("loaders", "build", "start", {
      name,
      fillLast: true,
      loaders: getLoaderNames(module.loaders), //获取处理当前module的loaders数组，用于最终的分组统计与展示
    });
  }
});

tap(compilation, "succeed-module", (module) => {
  const name = getModuleName(module);
  if (name) {
    this.addTimeEvent("loaders", "build", "end", {
      name,
      fillLast: true,
    });
```

```
  }
});

```

webpack-bundle-analyzer@4.8.0 产物大小分析插件

使用

```
const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin;

module.exports = {
  plugins: [
    new BundleAnalyzerPlugin()
  ]
}

```

原理 `bundleAnalyzerPlugin` 注册 `done` 钩子的 `callback` ，然后在 `callback` 内通过 `stats` 获取最终的信息，然后起服务，分析 `stats` 内的信息，并最终呈现出来

精简后的代码如下所示

```
class BundleAnalyzerPlugin {
  constructor(opts = {}) {
    this.opts = {};
    this.server = null;
    this.logger = new Logger(this.opts.logLevel);
  }

  apply(compiler) {
    this.compiler = compiler;

    const done = (stats, callback) => {
      callback = callback || (() => {});

      const actions = [];

      if (this.opts.generateStatsFile) {
        // 是否生成stats文件
        actions.push(() => this.generateStatsFile(stats.toJson(this.opts.statsOptions)));
      } // Handling deprecated `startAnalyzer` flag

```

```
21
22        if (this.opts.analyzerMode === 'server' && !this.opts.startAnalyzer) {
23          this.opts.analyzerMode = 'disabled';
24        }
25
26        if (this.opts.analyzerMode === 'server') {
27          // 是否起本地服务方式分析
28          actions.push(() => this.startAnalyzerServer(stats.toJson()));
29        } else if (this.opts.analyzerMode === 'static') {
30          // 是否以静态文件的方式分析
31          actions.push(() => this.generateStaticReport(stats.toJson()));
32        } else if (this.opts.analyzerMode === 'json') {
33          // 是否以生成json的方式分析
34          actions.push(() => this.generateJSONReport(stats.toJson()));
35        }
36
37        if (actions.length) {
38          // Making analyzer logs to be after all webpack logs in the console
39          setImmediate(async () => {
40            try {
41              await Promise.all(actions.map(action => action()));
42              callback();
43            } catch (e) {
44              callback(e);
45            }
46          });
47        } else {
48          callback();
49        }
50      };
51
52      if (compiler.hooks) {
53        compiler.hooks.done.tapAsync('webpack-bundle-analyzer', done);
54      } else {
55        compiler.plugin('done', done);
56      }
57    }
58  }
59
```

@soda/friendly-errors-webpack-plugin@1.8.1 美化错误插件

使用

```
1  const FriendlyErrorsWebpackPlugin = require('@soda/friendly-errors-webpack-
```

```
plugin')

module.exports = {
  plugins: [
    new FriendlyErrorsWebpackPlugin(),
  ]
}

```

原理 `FriendlyErrorswebpackPlugin` 插件在注册 `done` 及 `invalid` 钩子上注册 `callback`，然后在 `done` 钩子对应的 `callback` 内根据 `stats` 获取错误及警告，然后在进行对应的美化打印；在 `invalid` 钩子注册的 `callback` 内处理错误

精简后的代码如下所示

```
class FriendlyErrorsWebpackPlugin {

  constructor(options) {}

  apply(compiler) {

    const doneFn = stats => {
      this.clearConsole();

      const hasErrors = stats.hasErrors();
      const hasWarnings = stats.hasWarnings();

      if (!hasErrors && !hasWarnings) {
        this.displaySuccess(stats);
        return;
      }

      if (hasErrors) {
        this.displayErrors(extractErrorsFromStats(stats, 'errors'), 'error');
        return;
      }

      if (hasWarnings) {
        this.displayErrors(extractErrorsFromStats(stats, 'warnings'),
'warning');
      }
    };

    const invalidFn = () => {
      this.clearConsole();
```

```
30        output.title('info', 'WAIT', 'Compiling...');
31      };
32
33      if (compiler.hooks) {
34        const plugin = { name: 'FriendlyErrorsWebpackPlugin' };
35
36        compiler.hooks.done.tap(plugin, doneFn);
37        compiler.hooks.invalid.tap(plugin, invalidFn);
38      } else {
39        compiler.plugin('done', doneFn);
40        compiler.plugin('invalid', invalidFn);
41      }
42    }
43  }
44
```

编写自己的webpack插件

了解了上面的常用插件原理之后，我们知道，写一个 `webpack` 插件，最关键的点就是需要知道 `webpack` 大致的构建流程， `webpack` 流程中暴露了哪些 `hook` ，而我们的真实场景又是需要在哪个阶段介入，比如上面我们看到的

- `html-webpack-plugin` 插件主要目的是，根据传入的模版 `html` 文件，生成最终带js、css等静态资源的 `html` 文件，那么 `html-webpack-plugin` 就在编译开始阶段的 `make hook` 上注册 `callback` ，然后在 `callback` 内创建 `child compiler` 完成对 `html` 文件的编译，然后又在生成 `asset` 阶段的 `hook processAssets` 上注册 `callback` ，在这个 `callback` 内获取已经生成的 `assets` ，然后插入到 `html` 内
- `terser-webpack-plugin` 插件的主要目的是压缩js代码，那么要压缩肯定也是编译结束生成 `assets` 之后，然后对 `assets` 内的js进行压缩，所以是在 `assets` 生成阶段的 `processAssets hook` 上注册 `callback` ，然后在 `callback` 内对js文件进行压缩

是以我们要编写自己的 `webpack` 插件

1. 先确定自己的目的
2. 根据目的选择介入的 `webpack` 阶段
3. 然后在该阶段内，找 `webpack` 暴露的 `hook`
4. 然后注册对应 `hook` 的 `callback` ，在 `callback` 内完成对应的目的
5. 有些复杂场景可能会涉及到多个阶段的不同 `hook` ，那就需要自己多翻翻 `webpack` 文档

下面用几个具体的例子带领大家一起写 `webpack` 插件

约定式路由插件

我们不想要在项目内些routes配置，而是通过一些目录约定自动帮助我们生成路由配置文件，我们只需要加载这个路由文件即可

根据我们的目的来确定怎么写这个插件

目的：自动生成routes配置

webpack阶段：肯定要在编译之前，不然`webpack`会构建二次

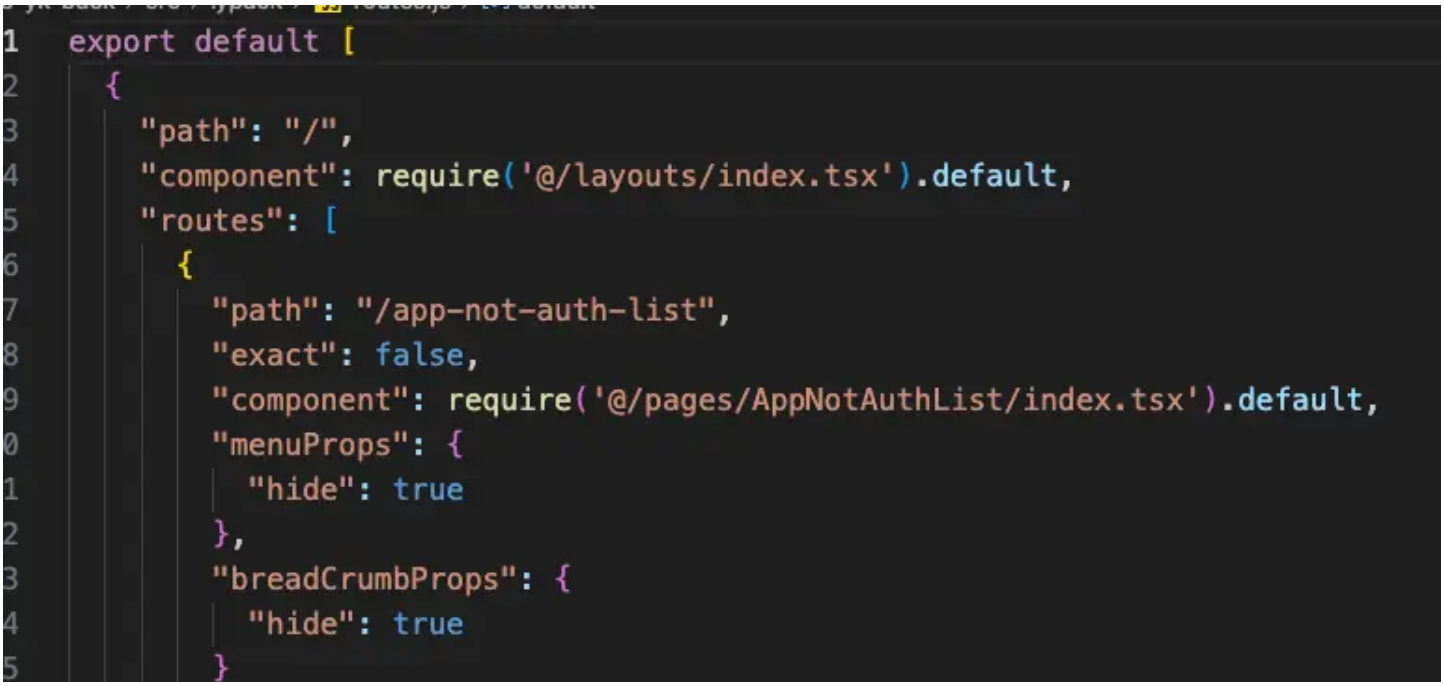
webpack编译之前的hook有： `environment`、`initialize`等很多`hook`，我们这里选择`initialize`

initialize callback内逻辑：处理路由文件生成与文件监听的逻辑

代码如下所示

```
const pluginName = 'ReactAutoRoutePlugin'
class ReactAutoRoutePlugin extends BaseRoute {

  constructor(options: IGetRoutesOpts) {
    super(options);
    this.options = options;
    this.isWriting = false;
  }

  apply(compiler: Compiler) {
    if (process.env.NODE_ENV === 'production') {
      compiler.hooks.run.tapPromise(pluginName, async () => {
        await this.writeFile();
      })
    } else {
      compiler.hooks.initialize.tap(pluginName, async () => {
        await this.writeFile();
        this.watchAndWriteFile();
      })
    }
  }
}

```

最终产物如下图所示



```
export default [
  {
    "path": "/",
    "component": require('@/layouts/index.tsx').default,
    "routes": [
      {
        "path": "/app-not-auth-list",
        "exact": false,
        "component": require('@/pages/AppNotAuthList/index.tsx').default,
        "menuProps": {
          "hide": true
        },
        "breadCrumbProps": {
          "hide": true
        }
```

注意点：这里只需要保证 `callback` 执行在 `webpack` 处理模块之前生成路由文件，避免在 `webpack` 处理模块之后生成，导致 `webpack` 重新编译或者最终的产物不包含最新的route内容

生成zip包插件

比如混合 `app` 场景， `app` 想要更新内容，我们使用的方式，就是在构建的时候将静态资源打包成一个zip包，然后远程通过拉取这个 `zip` 包达到资源更新的目的

根据我们的目的来确定怎么写这个插件

目的：生成 `zip` 包与一些辅助验证文件 webpack阶段：肯定要在编译之后，也就是 `assets` 生成阶段

webpack assets生成阶段的hook有： `emit` 、 `processAssets` 、 `afterProcessAssets` 等很多 `hook` ，我们这里选择 `emit`

emit callback内逻辑：处理 `zip` 包压缩逻辑与创建新的 `asset` 逻辑

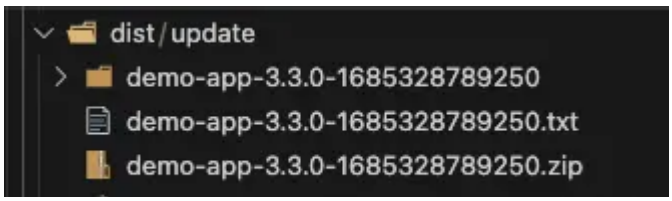
代码如下所示

```
export default class BuildZip {
  private options: Opts;
  constructor(options: Opts) {
    this.options = options;
  }
  async handlerZip(compilation: Compilation, callback: any) {
    if (compilation.compiler.isChild()) {
      return callback();
    }
    const { versionConfig, zipConfig } = this.options;
    const assetsCache = compilation.assets;
    // 将静态资源通过yazl处理成zip包
```

```
13      const [zip, config] = await Promise.all([
14        doZip(compilation, this.options),
15        generateConfig(compilation, this.options),
16      ]);
17
18      // 兼容webpack5与webpack4,webpack5可以直接在compiler.webpack.sources上直接获取操作source相关的构造函数
19      const { RawSource } = compilation.compiler.webpack
20        ? compilation.compiler.webpack.sources
21        : require('webpack-sources');
22
23      // 将zip合并成一个
24      const zipContent = new RawSource(Buffer.concat(zip as any) as any);
25      if (zipConfig.removeBundle === true) {
26        // 清空assets准备重新赋值
27        compilation.assets = {};
28      } else if (typeof zipConfig.removeBundle === 'function') {
29        const assets = {} as { [key: string]: any };
30        for (const name in compilation.assets) {
31          if (compilation.assets.hasOwnProperty(name)) {
32            if (!zipConfig.removeBundle(name, compilation.assets[name])) {
33              assets[name] = compilation.assets[name];
34            }
35          }
36        }
37
38        compilation.assets = assets;
39      }
40
41      const zipFileName = zipConfig.filename.replace('.zip', '');
42      const fileKeys = Object.keys(assetsCache);
43      // 保留原来的js、css等静态资源
44      fileKeys.map((key) => {
45        compilation.assets[`${zipFileName}/${key}`] = assetsCache[key];
46      });
47
48      // 添加一个包含文件目录的txt
49      compilation.assets[`${zipFileName}.txt`] = new RawSource(fileKeys.join('\n'));
50
51      // 生成zip包
52      compilation.assets[zipConfig.filename] = zipContent;
53
54      const content = JSON.stringify(config, null, '\t');
55
56      // 生成版本信息json文件
57      compilation.assets[versionConfig.filename] = new RawSource(content);
```

```
58
59      callback();
60    }
61    apply(compiler: Compiler) {
62      const { pass } = this.options;
63      if (!pass) {
64        // webpack5注册hook
65        if (compiler.hooks) {
66          compiler.hooks.emit.tapAsync('BuildZipPlugin',
   this.handlerZip.bind(this));
67        } else {
68          // webpack4之前注册hook方式
69          // @ts-ignore
70          compiler.plugin('emit', this.handlerZip.bind(this));
71        }
72      }
73    }
74  }
75
```

最终产物如下图所示



tree-shaking插件

在项目内引入一些 `npm` 包，但是一些 `npm` 包没有主动声明 `sideEffects` ，这时候看了代码之后，确定是没有副作用的，那么可以通过自动插件控制这个包的 `sideEffects` ，另外就是有些包设置了 `sideEffects` 但是在一些极端场景下会导致 `tree-shaking` 不对，项目运行的时候报错

根据我们的目的来确定怎么写这个插件

目的：为一些 `npm` 包自动添加 `sideEffects` 及去掉一些npm包的 `sideEffects`

webpack阶段：肯定要在编译过中，也就是 `module` 实例生成阶段

webpack module生成阶段的hook有： `createModule` 、 `module` 等很多 `hook` ，我们这里选择 `module`

module callback内逻辑：控制 `module` 上的元数据逻辑

代码如下所示

```
1  class ControlNpmPackageSideEffectsPlugin {
```

```
  noSideEffectsPackages: string[]
  hadSideEffectsPackages: string[]
  constructor({
    noSideEffectsPkgs = [], // 传入需要处理的npm package name
    hadSideEffectsPkgs = [], // 传入需要处理的npm package name
  } = {}) {
    this.noSideEffectsPackages = noSideEffectsPkgs;
    this.hadSideEffectsPackages = hadSideEffectsPkgs;
  }
  apply(compiler: Compiler) {
    if (!this.noSideEffectsPackages.length &&
!this.hadSideEffectsPackages.length) return;
    const name = this.constructor.name;

    compiler.hooks.normalModuleFactory.tap(name, (normalModuleFactory) => {
      // 在module实例创建之后，通过修改module相关的meta数据信息，改变模块或者包的
sideEffects配置
      normalModuleFactory.hooks.module.tap(name, (module, data) => {
        const resolveData = data.resourceResolveData;

        // 如果npm包没有设置sideEffects，且满足includePackages，就设置
sideEffectFree: true,表示该模块是纯的
        if (
          this.noSideEffectsPackages.some((item) =>
data?.resource?.includes(item)) &&
          resolveData &&
          resolveData.descriptionFileData &&
          resolveData.descriptionFileData.sideEffects === void 0
        ) {
          // 处理npm包没有标记了sideEffects的场景
          module.factoryMeta.sideEffects = false;
        }

        if (
          this.hadSideEffectsPackages.some((item) =>
data?.resource?.includes(item)) &&
          resolveData &&
          resolveData.descriptionFileData &&
          resolveData.descriptionFileData.sideEffects !== void 0
        ) {
          // 处理npm包标记了sideEffects的场景
          resolveData.descriptionFileData.sideEffects = undefined;
        }
      });
    });
  }
}
```

```
44
```

本地服务插件

有这样的场景，开发环境没有问题，但是上到测试环境之后，运行的时候报错，这时候没有`source-map`,代码又是压缩过后的，不太方便排查问题，如果这个时候我们可以直接在本地`build`一份构建后的产物，然后又可以请求测试环境的数据就很方便，我们可以通过一个自定义插件帮助我们达成这个目的

根据我们的目的来确定怎么写这个插件

目的：为本地build之后的产物，可以直接通过服务访问，并且可以请求到测试环境的数据

webpack阶段：肯定要在编译结束之后起服务，与编译过程中替换一些`html`中的占位符

webpack 结束阶段的hook有： `done`，我们这里选择`done`

webpack 编译过程中的hook有： `compilation`、`thisCompilation`，我们这里选择`compilation` done callback内逻辑：起server服务逻辑

compilation callback内逻辑：**注册`html-webpack-plugin` `hook`修改`html`内占位字符

代码如下所示

```
1  export default class LocalDebugSettingPlugin {
2    local_debug: string | undefined;
3    constructor({ userConfig }) {
4      this.local_debug = process.env.LOCAL_DEBUG;
5      this.userConfig = userConfig;
6    }
7
8    apply(compiler: Compiler) {
9      if (this.local_debug) {
10       if (envs.includes(this.local_debug)) {
11         this.registerReplace(compiler);
12         !process.env.LOCAL_DEBUG_NO_SERVER && this.addService(compiler);
13       } else {
14         console.log('当前process.env.LOCAL_DEBUG的值不是支持的类型，目前支持',
   envs.join(','));
15         process.exit(1);
16       }
17     }
18   }
19
20   getHtml(html: string) {
21     if (typeof html !== 'string') return html;
22     const OSS_HOST = 'https://xxxx.com';
23     const ENV = this.local_debug as string;
```

```
24      const DEPLOY_ENV = this.local_debug as string;
25      return html.replace(/__OSS_HOST__/gm, OSS_HOST).replace(/__ENV__/gm,
    YUNKE_ENV).replace(/__DEPLOY_ENV__/gm, DEPLOY_ENV);
26    }
27
28    replaceHtml(htmlPluginData, callback) {
29      if (typeof htmlPluginData.html === 'string') {
30        htmlPluginData.html = this.getHtml(htmlPluginData.html);
31      }
32      callback(null, htmlPluginData);
33    }
34
35    registerReplace(compiler: Compiler) {
36      if (compiler.hooks) {
37        compiler.hooks.compilation.tap('LocalDebugSettingPlugin', (compilation)
    => {
38          if (compilation.hooks.htmlWebpackPluginAfterHtmlProcessing) {
39            compilation.hooks.htmlWebpackPluginAfterHtmlProcessing.tapAsync(
40              'EnvReplaceWebpackPlugin',
41              this.replaceHtml.bind(this),
42            );
43          } else {
44            const htmlWebpackPlugin = compiler.options.plugins.filter((plugin)
    => plugin.constructor.name === 'HtmlWebpackPlugin');
45            if (htmlWebpackPlugin.length) {
46              htmlWebpackPlugin.forEach((item) => {
47
    item.constructor.getHooks(compilation).beforeEmit.tapAsync('LocalDebugSettingPl
    ugin', this.replaceHtml.bind(this));
48              });
49            } else {
50              const HtmlWebpackPlugin = require('html-webpack-plugin');
51              if (!HtmlWebpackPlugin) {
52                throw new Error('Please ensure that `html-webpack-plugin` was
    placed before `html-replace-webpack-plugin` in your Webpack config if you were
    working with Webpack 4.x!');
53              }
54              HtmlWebpackPlugin.getHooks(compilation).beforeEmit.tapAsync(
55                'EnvReplaceWebpackPlugin',
56                this.replaceHtml.bind(this),
57              );
58            }
59          }
60        });
61      } else {
62        compiler.plugin('compilation', (compilation) => {
```

```
63          compilation.plugin('html-webpack-plugin-before-html-processing', this.replaceHtml.bind(this));
64        });
65      }
66    }
67
68
69    addService(compiler) {
70      const { outputRoot = '/dist', devServer = {}, publicPath = '/' } = this.userConfig;
71      const contentBase = `${path.join(process.cwd(), outputRoot)}`;
72      const devServerOptions = Object.assign({}, {
73        publicPath,
74        contentBase: [contentBase],
75        historyApiFallback: true,
76      }, devServer, { inline: false, lazy: true, writeToDisk: true, watchContentBase: false, filename: /not-to-match/ });
77
78      if (!compiler.outputPath) {
79        compiler.outputPath = path.join(process.cwd(), outputRoot);
80      }
81
82      compiler.hooks.done.tap('LocalDebugSettingPlugin', (stats) => {
83        server.listen(devServerOptions.port, devServerOptions.host, (err: Error) => {
84          if (err) {
85            throw err;
86          }
87          console.log();
88          console.log('- 已开启本地生产调试模式，可以直接使用上面的链接地址进行访问');
89          console.log();
90        });
91      });
92
93      const server = new WebpackDevServer(compiler, devServerOptions);
94    }
95  }
96
```

总结

`webpack` 插件看似很难掌握，但其实只要掌握关键点，完成大部分业务需求还是问题不大，同时对于面试中问题的 `webpack plugin` 问题相信也能够有自己的准确回答 最后插件架构或者微内核架构目前是前端很常用的一种架构模式，在 `babel` 、 `rollup` 、 `esbuild` 、 `vite` 中都能看到这种架

构，只不过插件插件定义与运行略有不同，但是这种思想是想通的，所以掌握插件架构能够对我们的日常开发有一定的好处