学习前端工程化2——webpack入门演练

webpack 核心概念

- **entry（入口）：**Webpack创建bundle的起点。默认值是 `./src/index.js`。
- **output（输出）：**告诉 webpack在哪里**输出**它创建的bundles，以及如何命名这些文件。默认值是 `./dist/main.js`。
- **module（模块）：**在 webpack中，模块指的是构建在你的项目中的所有**依赖项**。这些模块会被映射到bundle中。
- **bundle（包）：**由 webpack打包的一组模块。
- **chunk（块）：**webpack构建的**中间产物**，用于管理bundles。一个chunk可以由多个模块组成，一个模块可以属于多个chunk。当你使用代码分割或加载器时，会用到chunks。
- **loader（加载器）：**webpack本身只能处理JavaScript和JSON文件。**加载器让 webpack能够处理其他类型的文件**，并将它们转换为有效的模块。
- **plugin（插件）：**插件用于执行范围更广泛的任务。从打包优化和压缩，到重新定义环境变量等等，插件的功能非常强大。

webpack 动手尝试

我们动手实现一遍基本的 webpack 项目的设置流程，对 webpack 不熟悉的读者可以打开一个新的空文件夹跟着操作一下：

1. **初始化项目**：在你的项目目录中打开终端，然后运行 `npm init -y` 来初始化你的项目。`-y` 表示 yes，即在初始化过程中所有的选项都选默认值，如果你想自定义配置可以根据文档修改 `package.json` 文件——package.json | npm Docs (npmjs.com)
2. **创建 `src/index.js`**：创建 `src` 文件夹，在 `src` 目录下创建一个 `index.js` 文件，你可以在这里写入一些JavaScript代码。

```
1 js
2 复制代码
3 console.log('Hello, Webpack!');
```

1. **创建 `public/index.html`**：在 `public` 目录下创建一个 `index.html` 文件，然后添加以下HTML代码：

```
1 html
2 复制代码
3 <!DOCTYPE html<html<head    <titleWebpack Project</title</head<body    <script
  src="../dist/main.js"</script</body</html
```

1. **创建 `webpack.config.js` 并填入配置**：在项目根目录下创建一个 `webpack.config.js` 文件，然后添加以下 webpack 配置：

```
1 js
2 复制代码
3 const path = require('path');module.exports = {    mode: 'development',
  entry: './src/index.js',      output:         filename: 'main.js',
  path: path.resolve(__dirname, 'dist'),    },};
```

1. **安装 webpack 和 webpack cli**： 运行 `npm install --save-dev webpack webpack-cli` 来安装 webpack 和 webpack cli。
2. **配置 build 命令为 webpack**：在你的 `package.json` 文件中，将 “scripts” 部分修改为以下内容：

```
1 json
2 复制代码
3 "scripts": {    "build": "webpack"}
```

1. **执行 npm run build 完成打包构建**：最后，运行 `npm run build` 来打包你的项目。如果成功，你应该会在 `dist` 目录下看到一个名为 `main.js` 的文件。

输出的 **main.js** 文件内容，这里笔者为了方便理解，删去了原来的注释并做了新的逐行解释：

```
1 js
2 复制代码
3 (() => {    // 这是一个立即执行的箭头函数，webpack的启动函数。     var
  webpack_modules = {        // 这是一个对象，其中包含了所有 webpack 打包的模块。
     // 这是模块的唯一标识符，对应于你的源代码文件。        './src/index.js': () => {
           // 这是一个函数，包含了模块的代码。            eval(
  "console.log('hello webpack');\r\n\n\n//# sourceURL=webpack://my-
  project/./src/index.js?"           ); // 这是模块的源代码，被包裹在一个eval函数中。
  eval函数会执行传入的JavaScript代码字符串。        },    };    var webpack_exports
  = {}; // 这是一个对象，用于存储模块的导出结果。
  __webpack_modules__['./src/index.js'](); // 这行代码加载并执行入口模块。})();
```

在浏览器中运行我们的 `index.html` 文件。我们先前通过 `<script src="../dist/main.js"></script>` 引入了这个文件，可以看到其中其实是一个**立即执行函数（IIFE）**，所以一旦这个文件被加载，它就会立即执行。如果成功，可以看到控制台打印出了 `'hello webpack'`。

至此我们就通过默认的配置，以最简单的形式过了一遍 webpack 工作的基本流程。

webpack devtool

前面一部分提及到，笔者删掉了一部分注释：

> - ATTENTION: The "eval" devtool has been used (maybe by default in mode: "development"). * This devtool is neither made for production nor for readable output files. * It uses "eval()" calls to create a separate source file in the browser devtools. * If you are trying to read the output file, select a different devtool (webpack.js.org/configurati…) * or disable the default devtool with "devtool: false". * If you are looking for production-ready output files, see mode: "production" (webpack.js.org/configurati…).

这部分注释告诉你，Webpack在开发模式下默认使用了"eval"作为devtool。“eval” devtool会使用 `eval()` 函数来创建浏览器开发者工具中的单独源文件。如果你想要阅读输出文件，或者寻找适合生产环境的输出文件，你可以选择一个不同的devtool或者通过设置 `devtool: false` 来禁用默认的devtool。

Devtool | webpack 中文文档

Devtool 控制是否生成，以及如何生成 **source map**。我们给 `webpack.config.js` 中添加一项配置属性 `devtool: source-map`，观察打包后的结果发现 `dist` 目录下多出了一个 `main.js.map` 文件：

```
1 json
2 复制代码
3 {    "version": 3,    "file": "main.js",    "mappings": ";;;;;AAAA",
  "sources": [        "webpack://my-project/./src/index.js"    ],
  "sourcesContent": [        "console.log('hello webpack');\r\n"    ],
  "names": [],    "sourceRoot": ""}
```

并且 `main.js` 中的内容也发生了变化：

```
1 js
2 复制代码
3 /******/ (() => { // webpackBootstrapvar webpack_exports =
  {};/*!**********************!*\  !*** ./src/index.js ***!
  \**********************/console.log('hello webpack');/******/ })();//#
  sourceMappingURL=main.js.map
```

先说这个新出现的 `.map` 文件，它是一个存储着代码位置信息的文件。Source Map的作用就在于能够帮你定位到错误的代码在那个文件哪一行。

具体来说，当你的代码出现错误时，浏览器控制台通常会显示出错的位置。但是，如果你的代码被压缩或转换（比如从ES6转换到ES5），那么显示的错误位置将会是转换后的代码，而不是你原始的源代码。这时候，Source Map就派上用场了。有了Source Map，浏览器在报错时可以直接显示出错源代码的位置，而不是转换后的代码。

在webpack中，如果devtool被设置为source-map，那么webpack会为每个模块生成对应的.map文件。并且在打包后的代码最后一行会有个注释，它会指向对应的.map文件。

⚠使用Source Map可能会暴露你的源代码，因此通常只在开发环境中使用。在生产环境中，应该选择其他devtool配置选项，或者完全禁用它。

阮一峰老师的博客详细介绍过 Source Map：JavaScript Source Map 详解 - 阮一峰的网络日志 (ruanyifeng.com)

webpack loader

默认情况下，webpack 只能理解 JavaScript 和 JSON 文件。**loader 用于把其他类型的文件转换成 js 文件**。（选择将所有类型的文件转换为JavaScript，主要是因为JavaScript是浏览器原生支持的语言。在前端开发中，无论是样式表（CSS）、图片、字体文件还是JavaScript模块，最终都需要通过浏览器来解析和渲染。）

这里通过引入最常见的**css-loader**，来加深大家对**loader**的理解：

引入 css 文件

我们在 `src` 目录下新建一个 `index.css` 文件，构建一个简单的样式：

```
1 css
2 复制代码
3 .test {    width: 100px;    height: 100px;    background-color: aqua;}
```

由于我们没有使用 MVVM 框架，所以还需要手动在 `index.html` 文件中插入这个元素：

```
1 html
2 复制代码
3 <div class="test"</div
```

那么如何引入我们的css文件呢？在正常的开发中我们只需要：

```
1 html
2 复制代码
3 <link rel="stylesheet" href="../src/index.css"
```

但这样引入的话其实不依赖于打包构建的，并且有违我们利用打包减少请求次数来提高性能的初衷——如果存在多个CSS文件，需要请求多次。

我们在 `index.js` 中引入 `index.css`：

```
1 js
2 复制代码
3 import './index.css';console.log('hello webpack');
```

此时直接执行 `npm run build` 会报错，提示我们需要正确的**loader**来处理css文件：

使用 css-loader

执行 `npm` 命令安装 `css-loader`：

```
1 terminal
2 复制代码
3 npm i -D css-loader
```

安装好以后，在 `webpack.config.js` 文件中添加**loader**的配置属性：

```
1 js
2 复制代码
3 module: {    rules: [        {            test: /.css$/,            use: ['css-
  loader'],        },    ],}
```

再次执行 `npm run build`，打包成功后在浏览器中运行 `index.html` 文件会发现，样式并没有生效。在控制台打开源代码中的 `index.css` 看看：

```
1 js
2 复制代码
3 // Importsimport CSS_LOADER_API_SOURCEMAP_IMPORT from "../node_modules/css-
  loader/dist/runtime/sourceMaps.js";import CSS_LOADER_API_IMPORT from
  "../node_modules/css-loader/dist/runtime/api.js";var CSS_LOADER_EXPORT =
  CSS_LOADER_API_IMPORT(CSS_LOADER_API_SOURCEMAP_IMPORT);//
  ModuleCSS_LOADER_EXPORT.push([module.id, `.test {    width: 100px;    height:
  100px;    background-color: aqua;}`, "",{"version":3,"sources":
  ["webpack://./src/index.css"],"names":
  [],"mappings":"AAAA;IACI,YAAY;IACZ,aAAa;IACb,sBAAsB;AAC1B","sourcesContent":
  [".test {\r\n    width: 100px;\r\n    height: 100px;\r\n    background-color:
  aqua;\r\n}\r\n"],"sourceRoot":""}]);// Exportsexport default CSS_LOADER_EXPORT;
```

css-loader的作用是将CSS文件转换为JavaScript模块。但是，它并不会将CSS样式应用到HTML文档中。 这就是为什么在控制台中看到的 `index.css` 文件内容看起来像JavaScript代码。

使用 stye-loader

style-loader的作用则是将css-loader处理后的结果（即CSS样式）通过创建style标签的方式添加到HTML页面上。 这就是为什么需要在使用了css-loader之后还需要使用style-loader。

这样的设计看起来有点奇怪，大多数情况下我们都需要 `css-loader` 和 `style-loader` 一起使用，为什么不合并成一个 loader 呢？Webpack 鼓励使用多个简单的、单一功能的 loader，而不是一个复杂的、多功能的 loader。这样设计的好处是可以保证每个 loader 的职责单一，也方便后期 loader 的组合和扩展。举两个例子：

- **只使用 `css-loader`：** 如果你想要将CSS作为字符串导入，而不是直接应用到DOM上，那么你可能只需要css-loader。例如，你可能想要在JavaScript中操作这些样式字符串，或者你可能想要在服务器端渲染(SSR)中使用它们。
- **只使用 `style-loader`：** 在某些情况下，你可能已经有了一些以JavaScript模块形式存在的CSS（例如，通过某种预处理器或者构建步骤生成），并且你想要将这些样式应用到DOM上。在这种情况下，你可能只需要style-loader。

安装 `style-loader`：

```
1 terminal
2 复制代码
3 npm i -D style-loader
```

在 `webpack.config.js` 文件中添加上配置：

```
1 js
2 复制代码
```

```
3 module: {    rules: [        {            test: /.css$/,            use:
  ['style-loader', 'css-loader'],        },    ],},
```

这里要注意了，配置中 `rule` 里面的内容是从下到上， `use` 里面的内容从右到左执行的。

此时再在浏览器中运行 `index.html` 文件会发现样式已经生效了，并且源代码中多了一个 `index.css` 文件：

- js
  复制代码
  // 导入style-loader的API，用于将样式插入到style标签中import API from "!../node_modules/style-loader/dist/runtime/injectStylesIntoStyleTag.js";// 导入style-loader的DOM API，用于操作DOMimport domAPI from "!../node_modules/style-loader/dist/runtime/styleDomAPI.js";// 导入style-loader的插入函数，用于将style标签插入到指定的位置import insertFn from "!../node_modules/style-loader/dist/runtime/insertBySelector.js";// 导入style-loader的设置属性函数，用于设置style标签的属性import setAttributes from "!../node_modules/style-loader/dist/runtime/setAttributesWithoutAttributes.js";// 导入style-loader的插入style元素函数，用于创建并插入style标签import insertStyleElement from "!../node_modules/style-loader/dist/runtime/insertStyleElement.js";// 导入style-loader的样式标签转换函数，用于转换样式标签import styleTagTransformFn from "!../node_modules/style-loader/dist/runtime/styleTagTransform.js";// 导入css-loader处理后的CSS内容和命名导出import content, * as namedExport from "!!../node_modules/css-loader/dist/cjs.js!./index.css";// 创建一个options对象，用于配置style-loadervar options = {};// 设置样式标签转换函数options.styleTagTransform = styleTagTransformFn;// 设置属性设置函数options.setAttributes = setAttributes;// 设置插入函数，将style标签插入到head中options.insert = insertFn.bind(null, "head");// 设置DOM APIoptions.domAPI = domAPI;// 设置插入style元素函数options.insertStyleElement = insertStyleElement;// 调用API函数，将CSS内容和配置选项传递给它，得到一个更新函数var update = API(content, options);// 从css-loader处理后的结果中导出所有命名导出exportfrom "!!../node_modules/css-loader/dist/cjs.js!./index.css";// 如果content有locals属性，则导出locals；否则导出undefinedexport default content && content.locals ? content.locals : undefined;

笔者添加了一些注释方便理解，这里最关键的就是 `API` 函数，它会根据 css 内容和配置来更新 DOM。检查一下我们的页面元素会发现， `style-loader` 在 `head` 标签内创建了 `style` 标签，并把 `index.css` 的样式都写入了这个 `style` 标签中。

实现一个自定义 loader

假设我们掘金的小伙伴设计了一种开发方式，以掘金的缩写 `jj` 命名了一个新的文件类型。在 `src` 目录下新建 `test.jj` ，并在 `index.js` 中引入：

```
```

```
1 js
2 复制代码
3 <script>  export default {    x: 1,    y: 2  }</script>
```

这里的设计类似于 `vue` 的单文件组件中的 `script` 标签。如果现在执行打包，也会像之前的 `css` 文件一样，提示我们需要正确的**loader**来处理 `jj` 文件。

在根目录下创建一个 `loader` 文件夹来存放我们的自定义 `loader`，创建 `jj-loader.js`：

```
1 js
2 复制代码
3 // 匹配 <script> 标签中的内容const REG = /<script>([\s\S]+?)<\/script>/;// 导出一个函数，参数是类型文件中的内容module.exports = function (src) {    const __src = src.match(REG);    return __src && __src[1] ? __src[1] : src;};
```

在 `webpack.config.js` 中添加对应的 `rule`：

```
1 js
2 复制代码
3 {    test: /.jj$/,    use: [path.resolve(__dirname, './loader/jj-loader.js')],}
```

此时执行 `npm run build`，查看 `main.js` 可以看到最后生成的结果：

```
1 js
2 复制代码
3 /* harmony default export */ const WEBPACK_DEFAULT_EXPORT = {    x: 1,    y: 2,};
```

同时查看控制台中的源代码也可以看到，`test.jj` 已经被转换成了可执行的 js 代码：

```
1 js
2 复制代码
3 export default {    x: 1,    y: 2,};
```

内联调用 loader

在Webpack中，除了在 `webpack.config.js` 文件中指定loader外，还可以在每个 `import` 语句中显式指定loader。

例如，你可以在 `import` 语句中这样使用 loader：

```
1 javascript
2 复制代码
3 import Styles from 'style-loader!css-loader?modules!./styles.css';
```

在这个例子中，`style-loader`、`css-loader` 和 `modules` 都是内联调用的。使用 `!` 将资源中的loader分开，每个部分都会相对于当前目录解析；问号 `?` 后面的部分被视为loader的选项。这些选项可以用来配置 `loader` 的行为；和 `module.rules.use` 一样，这里的 `loader` 也是从右往左执行的。

此外，你还可以通过添加前缀来覆盖配置中的所有loader：

- 使用 `!` 前缀，将禁用所有已配置的 normal loader：

```
1 javascript
2 复制代码
3 import Styles from '!style-loader!css-loader?modules!./styles.css';
```

- 使用 `!!` 前缀，将禁用所有已配置的loader（包括preLoader、loader和postLoader）：

```
1 javascript
2 复制代码
3 import Styles from '!!style-loader!css-loader?modules!./styles.css';
```

- 使用 `-!` 前缀，将禁用所有已配置的preLoader和loader，但不禁用 postLoaders：

```
1 javascript
2 复制代码
3 import Styles from '-!style-loader!css-loader?modules!./styles.css';
```

尽管内联调用loader提供了很大的灵活性，但Webpack官方文档仍推荐尽可能使用 `module.rules` 配置方式，因为这样可以减少源码中的代码量，并且可以在出错时更快地调试和定位loader中的问题。

webpack plugin

plugin 初体验

Webpack **插件（Plugin）** 是 Webpack 的支柱功能之一。Webpack 本身也是构建于你在 Webpack 配置中用到的相同的插件系统之上。插件的目的在于解决 Loader 无法实现的其他事。

Webpack 运行的生命周期中会广播出许多事件，插件可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。常见的有：打包优化，资源管理，注入环境变量等。

这里我们使用官方文档中 `plugin` 列表的第一位 BannerPlugin | webpack 中文文档，来研究一下 `plugin` 的工作原理。

首先在 `webpack.config.js` 中引入 `plugin` 的相关配置：

```
1 js
2 复制代码
3 ...const webpack = require('webpack');module.exports = {    ...,    plugins: [
        new webpack.BannerPlugin({            banner: 'Reese学习前端工程化',
  }),    ],};
```

此时再执行打包会发现，打包输出的 `main.js` 头部多出了一行 banner —— `/*! Reese学习前端工程化 */`。

为了搞清楚它是怎么工作的，我们找到 `node_modules\webpack\lib\BannerPlugin.js` 文件来看一下它的源码：

```
1 js
2 复制代码
3 const { ConcatSource } = require("webpack-sources");const Compilation =
  require("./Compilation");const ModuleFilenameHelpers =
  require("./ModuleFilenameHelpers");const Template = require("./Template");const
   createSchemaValidation = require("./util/create-schema-validation");// 创建一个
  用于验证选项对象的函数// 这个函数会根据提供的JSON schema来验证选项对象是否符合预期的格式
  const validate = createSchemaValidation(
  require('../schemas/plugins/BannerPlugin.check.js'), // 验证函数
  require('../schemas/plugins/BannerPlugin.json'), // JSON schema    {
  name: 'Banner Plugin', // 插件名称        baseDataPath: 'options', // 选项对象的基
  础路径    });// 定义一个函数，用于将字符串包装成注释// 如果字符串中不包含换行符，那么就直
  接将其转换为单行注释// 如果字符串中包含换行符，那么就将其转换为多行注释const
  wrapComment = (str) => {    if (!str.includes('\n')) {        return
  Template.toComment(str);    }    return `/*!\n * ${str
  .replace(/\*\//g, '* /')        .split('\n')        .join('\n * ')
  .replace(/\s+\n/g, '\n')        .trimEnd()}\n */`;};// 定义BannerPlugin类class
  BannerPlugin {    /**     * 构造函数，接收一个选项对象作为参数      * 如果选项是一个字
  符串或者函数，那么就将其作为banner选项      * 然后验证选项对象是否符合预期的格式      * 最
  后，根据banner选项的类型（函数或者字符串），设置this.banner属性      */
  constructor(options) {        // 如果选项是一个字符串或者函数，那么就将其作为banner选
  项        if (typeof options === 'string' || typeof options === 'function') {
```

```
        options = {            banner: options,        };      }    // 验证选项对象是否符合预期的格式      validate(options);      // 将处理过的选项对象保存到this.options属性中      this.options = options;      // 获取banner选项      const bannerOption = options.banner;      if (typeof bannerOption === 'function') {        // 如果banner选项是一个函数，那么就直接将其保存到this.banner属性中        // 如果设置了raw选项，那么就直接使用原始的banner函数        // 否则，就将banner函数的返回值包装成注释        const getBanner = bannerOption;        this.banner = this.options.raw    ? getBanner            : /** @type {BannerFunction} */ (data) =>          wrapComment(getBanner(data));      } else {        // 如果banner选项是一个字符串，那么就将其包装成注释后保存到this.banner属性中        // 如果设置了raw选项，那么就直接使用原始的banner字符串        // 否则，就将banner字符串包装成注释        const banner = this.options.raw            ? bannerOption            : wrapComment(bannerOption);      this.banner = () => banner;      }    }    /**     * apply方法，接收一个compiler对象作为参数     * 这个方法会在Webpack编译过程中被调用，用于注册插件需要监听的钩子事件     */    apply(compiler) {      const options = this.options; // 插件选项      const banner = this.banner; // banner内容      const matchObject = ModuleFilenameHelpers.matchObject.bind(        undefined,        options      ); // 用于匹配文件名的函数      const cache = new WeakMap(); // 用于缓存已处理的资源      // 监听compiler的compilation钩子      // 当新的编译创建时，这个钩子就会被触发      compiler.hooks.compilation.tap('BannerPlugin', (compilation) => {        // 监听compilation的processAssets钩子        // 在资源生成阶段，这个钩子就会被触发        compilation.hooks.processAssets.tap(          {            name: 'BannerPlugin', // 插件名称            stage: Compilation.PROCESS_ASSETS_STAGE_ADDITIONS, // 钩子触发阶段          },          () => {            for (const chunk of compilation.chunks) {              // 遍历所有的代码块              if (options.entryOnly && !chunk.canBeInitial()) {                // 如果只处理入口代码块，且当前代码块不是入口代码块，则跳过                continue;              }              for (const file of chunk.files) {                // 遍历代码块中的所有文件                if (!matchObject(file)) {                  // 如果文件名不匹配，则跳过                  continue;                }                const data = {                  // 创建一个包含当前代码块和文件名的对象，用于生成banner内容                  chunk,                  filename: file,                };                const comment = compilation.getPath(banner, data); // 获取banner内容                // 更新资源内容，添加banner注释                compilation.updateAsset(file, (old) => {                  let cached = cache.get(old); // 从缓存中获取已处理的资源                  if (!cached || cached.comment !== comment) {                    // 如果资源未被处理过，或者banner内容有变化，则重新处理资源并更新缓存                    const source = options.footer                      ? new ConcatSource(old, '\n', comment) // 如果是footer模式，则将banner
```

```
内容添加到资源内容的末尾                                                    : new
ConcatSource(comment, '\n', old); // 否则，将banner内容添加到资源内容的开头
                                cache.set(old, { source, comment });
                        return source;                                      }
                        return cached.source; // 如果资源已被处理过，并且banner内容未
变化，则直接返回缓存中的资源内容                                              });
    }                       }                   }               );          });     }}// 导
出BannerPlugin类，以便在其他模块中使用它module.exports = BannerPlugin;
```

笔者加上了详细的注释帮助理解代码的逻辑，重点来看一下这里用到的两个钩子：

- `compiler.hooks.compilation`：当新的编译创建时，这个钩子就会被触发。在这个钩子的回调函数中，插件会注册监听 `compilation.hooks.processAssets` 钩子。
- `compilation.hooks.processAssets`：在资源生成阶段，这个钩子就会被触发。在这个钩子的回调函数中，插件会遍历所有的代码块和文件，然后根据选项和文件名来决定是否需要给文件添加banner注释。

这里用到了 webpack 提供给插件的两个主要接口—— `compiler` 和 `compilation`，它们分别代表了Webpack环境和单次编译过程，并提供了一系列的钩子函数供插件使用。

- compiler 钩子 | webpack 中文文档
- compilation 钩子 | webpack 中文文档

具体来说，当你执行 `npm run build` 命令时，Webpack会创建一个 `compiler` 对象，这个对象代表了整个Webpack环境的配置。在Webpack生命周期内，`compiler` 对象只会被创建一次。

当你修改文件并保存时，如果你启用了Webpack的热更新功能（Hot Module Replacement），Webpack会创建一个新的 `compilation` 对象，这个对象代表了一次新的编译过程。每次文件变化都会触发一次新的编译过程，也就是说，每次文件变化都会创建一个新的 `compilation` 对象。

实现一个自定义 plugin

确保理解了 `BannerPlugin` 的实现之后，我们来照着它的实现，尝试自己动手实现一个简单版的，能够在 `main.js` 底部插入信息的自定义插件 `FooterPlugin`。

首先新建 `src` 目录下的 `plugin` 文件夹，在里面新建一个 `FooterPlugin.js` 文件。在 `webpack.config.js` 中设置相关配置：

```
1 js
2 复制代码
3 ...const footerPlugin = require('./plugin/FooterPlugin');module.exports = {
  ...    plugins: [        ...,        new footerPlugin({            footer:
  'Reese学习前端工程化',        }),    ],};
```

通过观察 `BannerPlugin` 的实现可以发现，实现一个简单的 webpack 插件，其核心思路主要包括以下几个步骤：

1. **定义插件类**：首先，我们需要定义一个插件类，这个类需要有一个 `apply` 方法。这个方法是 webpack 插件的主要方法，它接收一个 `compiler` 参数，这个参数是 webpack 的编译器实例。在我们的插件类中，构造函数接收一个名为 `options` 的参数。这个 `options` 参数是一个对象，它包含了用户在使用插件时传入的选项。
2. **监听事件**：在插件类的 `apply` 方法中，我们使用 `compiler.hooks.compilation.tap` 方法监听 `compilation` 事件。当 webpack 开始一个新的编译过程时，就会触发这个事件。在这个事件的回调函数中，我们又使用 `compilation.hooks.processAssets.tap` 方法监听 `processAssets` 事件。当 webpac k处理完所有的资源后，就会触发这个事件。
3. **处理资源**：在 `processAssets` 事件的回调函数中，我们遍历所有的 chunk 和它们的文件。对于每个文件，我们从插件的选项中获取 footer 属性，并将其作为注释添加到文件的末尾。这是通过使用 `compilation.updateAsset` 方法和 `ConcatSource` 类来实现的。`ConcatSource` 类可以将多个源连接起来，形成一个新的源。
4. **导出插件类**：最后，我们需要将插件类导出，以便在其他文件中使用。

```
1 js
2 复制代码
3 // 引入webpack-sources库中的ConcatSource模块const { ConcatSource } =
  require('webpack-sources');// 定义一个名为FooterPlugin的插件类class FooterPlugin
  {    // 构造函数，接收一个options参数    constructor(options) {        // 将
  options参数赋值给this.options，以便在类的其他方法中使用        this.options =
  options;    }    // apply是webpack插件的主要方法，接收一个compiler参数
  apply(compiler) {        // 使用compiler.hooks.compilation.tap方法监听compilation
  事件        compiler.hooks.compilation.tap('FooterPlugin', (compilation) => {
          // 在compilation事件中，使用compilation.hooks.processAssets.tap方法监听
  processAssets事件
  compilation.hooks.processAssets.tap('FooterPlugin', () => {                // 遍
  历compilation.chunks中的每个chunk                for (const chunk of
  compilation.chunks) {                    // 遍历每个chunk的files
     for (const file of chunk.files) {                        // 从this.options中
  获取footer属性，并赋值给comment变量                        const comment =
  this.options.footer;                    // 使用compilation.updateAsset方法更
  新资源                    compilation.updateAsset(
    file,                        // 使用ConcatSource将旧资源和注释连接起来，形成新
  资源                        (old) => new ConcatSource(old, '\n', comment)
                    );                    }                }            });
      });    }}// 将FooterPlugin类导出，以便在其他文件中使用module.exports =
  FooterPlugin;
```

此时执行打包就可以看到，`'Reese学习前端工程化'` 被添加到了页面底部。

上面提及的插件的 `apply` 方法中，都通过遍历 `compilation` 上的 `chunks` 来给资源文件做对应的处理。最后再补充一下 **chunk** 相关的一些知识：

- **chunk 从哪里来：** Webpack 会**根据 Module 引用关系生成 chunk 文件**。每个 chunk 包含多个模块。例如，从一个入口文件开始，该入口文件引用其他模块，这些模块再引用其他模块。Webpack 通过这种引用关系逐个打包模块，这些模块就形成了一个chunk。如果我们有多个入口文件，可能会产出多条打包路径，一条路径就会形成一个 chunk。
- **chunk 到哪里去：** Webpack 处理好 chunk 文件后，最后会输出 bundle 文件。这个 bundle 文件包含了经过加载和编译的最终源文件，它可以直接在浏览器中运行。

> Module、Chunk 和 Bundle 其实就是同一份逻辑代码在不同转换场景下的三个名字：**我们直接写出来的是 Module，Webpack 处理时是 Chunk，最后生成浏览器可以直接运行的 Bundle。**

总结

我以为，学习前端相关的知识，你的代码跑通了，你的思路也就通了大半了。首先代码跑通了就能用了，大体的认知就有了；其次把没看懂的代码和遇到的问题一查，理解也就深入了。**简单来说就是，动手，动手，还是 tmd 动手！**

我曾经很长一段时间呆在 `src` 目录下的舒适圈了，对工程化配置文件遇到不懂的就查一下，照搬网上的配置。这次也是痛下决心，决定拥抱 `src` 之外的世界，动手实践一遍曾经只会复制粘贴的内容。如果觉得这个系列对你有帮助，不妨点赞收藏评论，怎么方便怎么来。