Webpack5 快速入门

基础配置

• 前言

- 为什么需要打包工具?

开发时，我们会使用框架（React、Vue），ES6 模块化语法，Less/Sass 等 css 预处理器等语法进行开发。

这样的代码要想在浏览器运行必须经过编译成浏览器能识别的 JS、Css 等语法，才能运行。

所以我们需要打包工具帮我们做完这些事。

除此之外，打包工具还能压缩代码、做兼容性处理、提升代码性能等。

- 有哪些打包工具?

- Grunt
- Gulp
- Parcel
- Webpack
- Rollup
- Vite
- ...

目前市面上最流量的是 Webpack，所以我们主要以 Webpack 来介绍使用打包工具

• 基本使用

`Webpack` 是一个静态资源打包工具。

它会以一个或多个文件作为打包的入口，将我们整个项目所有文件编译组合成一个或多个文件输出出去。

输出的文件就是编译好的文件，就可以在浏览器段运行了。

我们将 `Webpack` 输出的文件叫做 `bundle`。

功能介绍

Webpack 本身功能是有限的:

- 开发模式：仅能编译 JS 中的 `ES Module` 语法
- 生产模式：能编译 JS 中的 `ES Module` 语法，还能压缩 JS 代码

开始使用

1. 资源目录

```
webpack_code # 项目根目录（所有指令必须在这个目录运行）
    └── src # 项目源码目录
        ├── js # js文件目录
        │   ├── count.js
        │   └── sum.js
        └── main.js # 项目主文件
```

2. 创建文件

- count.js

```
export default function count(x, y) {
    return x - y;
}
```

- sum.js

```
export default function sum(...args) {
    return args.reduce((p, c) => p + c, 0);
}
```

- main.js

```
import count from "./js/count";
import sum from "./js/sum";

console.log(count(2, 1));
console.log(sum(1, 2, 3, 4));
```

3. 下载依赖

打开终端，来到项目根目录。运行以下指令：

- 初始化 `package.json`

```
npm init -y
```

此时会生成一个基础的 `package.json` 文件。

需要注意的是 `package.json` 中 `name` 字段不能叫做 `webpack`，否则下一步会报错

- 下载依赖

```
npm i webpack webpack-cli -D
```

4. 启用 Webpack

- 开发模式

```
npx webpack ./src/main.js --mode=development
```

- 生产模式

```
npx webpack ./src/main.js --mode=production
```

`npx webpack`：是用来运行本地安装 `Webpack` 包的。

`./src/main.js`：指定 `Webpack` 从 `main.js` 文件开始打包，不但会打包 `main.js`，还会将其依赖也一起打包进来。

`--mode=xxx`：指定模式（环境）。

5. 观察输出文件

默认 `Webpack` 会将文件打包输出到 `dist` 目录下，我们查看 `dist` 目录下文件情况就好了。

小结

`Webpack` 本身功能比较少，只能处理 `js` 资源，一旦遇到 `css` 等其他资源就会报错。

所以我们学习 `Webpack`，就是主要学习如何处理其他资源。

• 基本配置

在开始使用 `Webpack` 之前，我们需要对 `Webpack` 的配置有一定的认识。

- 5 大核心概念

1. entry（入口）

指示 Webpack 从哪个文件开始打包

2. output（输出）

指示 Webpack 打包完的文件输出到哪里去，如何命名等

3. loader（加载器）

webpack 本身只能处理 js、json 等资源，其他资源需要借助 loader，Webpack 才能解析

4. plugins（插件）

扩展 Webpack 的功能

5. mode（模式）

主要由两种模式：

- 开发模式：development
- 生产模式：production

- 准备 Webpack 配置文件

在项目根目录下新建文件：`webpack.config.js`

```
module.exports = {
    // 入口
    entry: "", // 相对路径
    // 输出
    output: {
        // 文件的输出路径
        path: "", // 绝对路径
        // 文件名
        filename: "",
    },
    // 加载器
    module: {
        rules: [
            // loader的配置
        ],
    },
    // 插件
    plugins: [
        // 插件的配置
    ],
```

```
    // 模式
    mode: "",
};
```

Webpack 是基于 Node.js 运行的，所以采用 Common.js 模块化规范

修改配置文件

1. 配置文件

```
// Node.js的核心模块，专门用来处理文件路径
const path = require("path");

module.exports = {
    // 入口
    // 相对路径和绝对路径都行
    entry: "./src/main.js",
    // 输出
    output: {
        // path: 文件输出目录，必须是绝对路径
        // path.resolve()方法返回一个绝对路径
        // __dirname 当前文件的文件夹绝对路径
        path: path.resolve(__dirname, "dist"),
        // filename: 输出文件名
        filename: "main.js",
    },
    // 加载器
    module: {
        rules: [],
    },
    // 插件
    plugins: [],
    // 模式
    mode: "development", // 开发模式
};
```

2. 运行指令

```
npx webpack
```

此时功能和之前一样，也不能处理样式资源

小结

Webpack 将来都通过 `webpack.config.js` 文件进行配置，来增强 Webpack 的功能

我们后面会以两个模式来分别搭建 Webpack 的配置，先进行开发模式，再完成生产模式

开发模式介绍

开发模式顾名思义就是我们开发代码时使用的模式。

这个模式下我们主要做两件事：

1. 编译代码，使浏览器能识别运行

开发时我们有样式资源、字体图标、图片资源、html 资源等，webpack 默认都不能处理这些资源，所以我们要加载配置来编译这些资源

2. 代码质量检查，树立代码规范

提前检查代码的一些隐患，让代码运行时能更加健壮。

提前检查代码规范和格式，统一团队编码风格，让代码更优雅美观。

处理样式资源

本章节我们学习使用 Webpack 如何处理 Css、Less、Sass、Scss、Styl 样式资源

介绍

Webpack 本身是不能识别样式资源的，所以我们需要借助 Loader 来帮助 Webpack 解析样式资源

我们找 Loader 都应该去官方文档中找到对应的 Loader，然后使用

官方文档找不到的话，可以从社区 Github 中搜索查询

Webpack 官方 Loader 文档：https://webpack.docschina.org/loaders/

处理 Css 资源

1. 下载包

```
npm i css-loader style-loader -D
```

注意：需要下载两个 loader

2. 功能介绍

- `css-loader`：负责将 Css 文件编译成 Webpack 能识别的模块
- `style-loader`：会动态创建一个 Style 标签，里面放置 Webpack 中 Css 模块内容

此时样式就会以 Style 标签的形式在页面上生效

3. 配置

```
const path = require("path");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "main.js",
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: ["style-loader", "css-loader"],
      },
    ],
  },
  plugins: [],
  mode: "development",
};
```

4. 添加 Css 资源

- src/css/index.css

```
.box1 {
  width: 100px;
  height: 100px;
  background-color: pink;
}
```

- src/main.js

```
import count from "./js/count";
import sum from "./js/sum";
// 引入 Css 资源，Webpack才会对其打包
import "./css/index.css";

console.log(count(2, 1));
console.log(sum(1, 2, 3, 4));
```

- public/index.html

```
```

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>webpack5</title>
  </head>
  <body>
    <h1>Hello Webpack5</h1>
    <!-- 准备一个使用样式的 DOM 容器 -->
    <div class="box1"></div>
    <!-- 引入打包后的js文件，才能看到效果 -->
    <script src="../dist/main.js"></script>
  </body>
</html>
```

5. 运行指令

```
npx webpack
```

打开 index.html 页面查看效果

处理 Less 资源

1. 下载包

```
npm i less less-loader -D
```

2. 功能介绍

- `less-loader`：负责将 Less 文件编译成 Css 文件

3. 配置

```
const path = require("path");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "main.js",
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
```

```
          test: /\.css$/,
          // use 数组里面 Loader 执行顺序是从右到左
          use: ["style-loader", "css-loader"],
        },
        {
          test: /\.less$/,
          use: ["style-loader", "css-loader", "less-loader"],
        },
      ],
    },
    plugins: [],
    mode: "development",
  };
```

4. 添加 Less 资源

- src/less/index.less

```
.box2 {
  width: 100px;
  height: 100px;
  background-color: deeppink;
}
```

- src/main.js

```
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/index.css";
import "./less/index.less";

console.log(count(2, 1));
console.log(sum(1, 2, 3, 4));
```

- public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>webpack5</title>
  </head>
  <body>
```

```
    <h1>Hello Webpack5</h1>
    <div class="box1"></div>
    <div class="box2"></div>
    <script src="../dist/main.js"></script>
  </body>
</html>
```

5. 运行指令

```
npx webpack
```

打开 index.html 页面查看效果

处理 Sass 和 Scss 资源

1. 下载包

```
npm i sass-loader sass -D
```

注意：需要下载两个

2. 功能介绍

- `sass-loader`：负责将 Sass 文件编译成 css 文件
- `sass`：`sass-loader` 依赖 `sass` 进行编译

3. 配置

```
const path = require("path");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "main.js",
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.less$/,
        use: ["style-loader", "css-loader", "less-loader"],
```

```
        },
        {
          test: /\.s[ac]ss$/,
          use: ["style-loader", "css-loader", "sass-loader"],
        },
      ],
    },
    plugins: [],
    mode: "development",
  };
```

4. 添加 Sass 资源

- src/sass/index.sass

```
/* 可以省略大括号和分号 */
.box3
  width: 100px
  height: 100px
  background-color: hotpink
```

- src/sass/index.scss

```
.box4 {
  width: 100px;
  height: 100px;
  background-color: lightpink;
}
```

- src/main.js

```
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";

console.log(count(2, 1));
console.log(sum(1, 2, 3, 4));
```

- public/index.html

```
```

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>webpack5</title>
  </head>
  <body>
    <h1>Hello Webpack5</h1>
    <div class="box1"></div>
    <div class="box2"></div>
    <div class="box3"></div>
    <div class="box4"></div>
    <script src="../dist/main.js"></script>
  </body>
</html>
```

5. 运行指令

```
npx webpack
```

打开 index.html 页面查看效果

处理 Styl 资源

1. 下载包

```
npm i stylus-loader -D
```

2. 功能介绍

- `stylus-loader`：负责将 Styl 文件编译成 Css 文件

3. 配置

```
const path = require("path");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "main.js",
  },
  module: {
    rules: [
      {
```

```
          // 用来匹配 .css 结尾的文件
          test: /\.css$/,
          // use 数组里面 Loader 执行顺序是从右到左
          use: ["style-loader", "css-loader"],
        },
        {
          test: /\.less$/,
          use: ["style-loader", "css-loader", "less-loader"],
        },
        {
          test: /\.s[ac]ss$/,
          use: ["style-loader", "css-loader", "sass-loader"],
        },
        {
          test: /\.styl$/,
          use: ["style-loader", "css-loader", "stylus-loader"],
        },
      ],
    },
    plugins: [],
    mode: "development",
  };
```

4. 添加 Styl 资源

- src/styl/index.styl

```
/* 可以省略大括号、分号、冒号 */
.box
  width 100px
  height 100px
  background-color pink
```

- src/main.js

```
import { add } from "./math";
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";
import "./styl/index.styl";

console.log(count(2, 1));
console.log(sum(1, 2, 3, 4));
```

- public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>webpack5</title>
  </head>
  <body>
    <h1>Hello Webpack5</h1>
    <!-- 准备一个使用样式的 DOM 容器 -->
    <div class="box1"></div>
    <div class="box2"></div>
    <div class="box3"></div>
    <div class="box4"></div>
    <div class="box5"></div>
    <script src="../dist/main.js"></script>
  </body>
</html>
```

5. 运行指令

```
npx webpack
```

打开 index.html 页面查看效果

• 处理图片资源

过去在 Webpack4 时，我们处理图片资源通过 `file-loader` 和 `url-loader` 进行处理

现在 Webpack5 已经将两个 Loader 功能内置到 Webpack 里了，我们只需要简单配置即可处理图片资源

- 1. 配置

```
const path = require("path");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "main.js",
  },
  module: {
```

```
      rules: [
        {
          // 用来匹配 .css 结尾的文件
          test: /\.css$/,
          // use 数组里面 Loader 执行顺序是从右到左
          use: ["style-loader", "css-loader"],
        },
        {
          test: /\.less$/,
          use: ["style-loader", "css-loader", "less-loader"],
        },
        {
          test: /\.s[ac]ss$/,
          use: ["style-loader", "css-loader", "sass-loader"],
        },
        {
          test: /\.styl$/,
          use: ["style-loader", "css-loader", "stylus-loader"],
        },
        {
          test: /\.(png|jpe?g|gif|webp)$/,
          type: "asset",
        },
      ],
    },
    plugins: [],
    mode: "development",
  };
```

2. 添加图片资源

- src/images/1.jpeg
- src/images/2.png
- src/images/3.gif

3. 使用图片资源

- src/less/index.less

```
.box2 {
  width: 100px;
  height: 100px;
  background-image: url("../images/1.jpeg");
  background-size: cover;
}
```

- src/sass/index.sass

```
.box3
  width: 100px
  height: 100px
  background-image: url("../images/2.png")
  background-size: cover
```

- src/styl/index.styl

```
.box5
  width 100px
  height 100px
  background-image url("../images/3.gif")
  background-size cover
```

4. 运行指令

```
npx webpack
```

打开 index.html 页面查看效果

5. 输出资源情况

此时如果查看 dist 目录的话，会发现多了三张图片资源

因为 Webpack 会将所有打包好的资源输出到 dist 目录下

- 为什么样式资源没有呢?

因为经过 `style-loader` 的处理，样式资源打包到 main.js 里面去了，所以没有额外输出出来

6. 对图片资源进行优化

将小于某个大小的图片转化成 data URI 形式（Base64 格式）

```
const path = require("path");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "main.js",
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
```

```
          test: /\.css$/,
          // use 数组里面 Loader 执行顺序是从右到左
          use: ["style-loader", "css-loader"],
        },
        {
          test: /\.less$/,
          use: ["style-loader", "css-loader", "less-loader"],
        },
        {
          test: /\.s[ac]ss$/,
          use: ["style-loader", "css-loader", "sass-loader"],
        },
        {
          test: /\.styl$/,
          use: ["style-loader", "css-loader", "stylus-loader"],
        },
        {
          test: /\.(png|jpe?g|gif|webp)$/,
          type: "asset",
          parser: {
            dataUrlCondition: {
              maxSize: 10 * 1024 // 小于10kb的图片会被base64处理
            }
          }
        },
      ],
    },
    plugins: [],
    mode: "development",
  };
```

- 优点：减少请求数量
- 缺点：体积变得更大

此时输出的图片文件就只有两张，有一张图片以 data URI 形式内置到 js 中了
（注意：需要将上次打包生成的文件清空，再重新打包才有效果）

• 修改输出资源的名称和路径

1. 配置

```
const path = require("path");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
  },
```

```
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.less$/,
        use: ["style-loader", "css-loader", "less-loader"],
      },
      {
        test: /\.s[ac]ss$/,
        use: ["style-loader", "css-loader", "sass-loader"],
      },
      {
        test: /\.styl$/,
        use: ["style-loader", "css-loader", "stylus-loader"],
      },
      {
        test: /\.(png|jpe?g|gif|webp)$/,
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
          },
        },
        generator: {
          // 将图片文件输出到 static/imgs 目录中
          // 将图片文件命名 [hash:8][ext][query]
          // [hash:8]: hash值取8位
          // [ext]: 使用之前的文件扩展名
          // [query]: 添加之前的query参数
          filename: "static/imgs/[hash:8][ext][query]",
        },
      },
    ],
  },
  plugins: [],
  mode: "development",
};
```

2. 修改 index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>webpack5</title>
  </head>
  <body>
    <h1>Hello Webpack5</h1>
    <div class="box1"></div>
    <div class="box2"></div>
    <div class="box3"></div>
    <div class="box4"></div>
    <div class="box5"></div>
    <!-- 修改 js 资源路径 -->
    <script src="../dist/static/js/main.js"></script>
  </body>
</html>
```

3. 运行指令

```
npx webpack
```

- 此时输出文件目录：

（注意：需要将上次打包生成的文件清空，再重新打包才有效果）

```
├── dist
    └── static
        ├── imgs
        │   └── 7003350e.png
        └── js
            └── main.js
```

自动清空上次打包资源

1. 配置

```
const path = require("path");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "static/js/main.js",
    clean: true, // 自动将上次打包目录资源清空
  },
  module: {
    rules: [
      {
```

```
          // 用来匹配 .css 结尾的文件
          test: /\.css$/,
          // use 数组里面 Loader 执行顺序是从右到左
          use: ["style-loader", "css-loader"],
        },
        {
          test: /\.less$/,
          use: ["style-loader", "css-loader", "less-loader"],
        },
        {
          test: /\.s[ac]ss$/,
          use: ["style-loader", "css-loader", "sass-loader"],
        },
        {
          test: /\.styl$/,
          use: ["style-loader", "css-loader", "stylus-loader"],
        },
        {
          test: /\.(png|jpe?g|gif|webp)$/,
          type: "asset",
          parser: {
            dataUrlCondition: {
              maxSize: 40 * 1024, // 小于40kb的图片会被base64处理
            },
          },
          generator: {
            // 将图片文件输出到 static/imgs 目录中
            // 将图片文件命名 [hash:8][ext][query]
            // [hash:8]: hash值取8位
            // [ext]: 使用之前的文件扩展名
            // [query]: 添加之前的query参数
            filename: "static/imgs/[hash:8][ext][query]",
          },
        },
      ],
    },
    plugins: [],
    mode: "development",
  };
```

2. 运行指令

```
npx webpack
```

观察 dist 目录资源情况

• 处理字体图标资源

- 1. 下载字体图标文件

1. 打开阿里巴巴矢量图标库
2. 选择想要的图标添加到购物车，统一下载到本地

- 2. 添加字体图标资源

- src/fonts/iconfont.ttf
- src/fonts/iconfont.woff
- src/fonts/iconfont.woff2
- src/css/iconfont.css
    - 注意字体文件路径需要修改
- src/main.js

```
import { add } from "./math";
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/iconfont.css";
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";
import "./styl/index.styl";

console.log(count(2, 1));
console.log(sum(1, 2, 3, 4));
```

- public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>webpack5</title>
  </head>
  <body>
    <h1>Hello Webpack5</h1>
    <div class="box1"></div>
    <div class="box2"></div>
    <div class="box3"></div>
    <div class="box4"></div>
```

```
    <div class="box5"></div>
    <!-- 使用字体图标 -->
    <i class="iconfont icon-arrow-down"></i>
    <i class="iconfont icon-ashbin"></i>
    <i class="iconfont icon-browse"></i>
    <script src="../dist/static/js/main.js"></script>
  </body>
</html>
```

3. 配置

```
const path = require("path");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true, // 自动将上次打包目录资源清空
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.less$/,
        use: ["style-loader", "css-loader", "less-loader"],
      },
      {
        test: /\.s[ac]ss$/,
        use: ["style-loader", "css-loader", "sass-loader"],
      },
      {
        test: /\.styl$/,
        use: ["style-loader", "css-loader", "stylus-loader"],
      },
      {
        test: /\.(png|jpe?g|gif|webp)$/,
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
          },
        },
        generator: {
          // 将图片文件输出到 static/imgs 目录中
```

```
            // 将图片文件命名 [hash:8][ext][query]
            // [hash:8]: hash值取8位
            // [ext]: 使用之前的文件扩展名
            // [query]: 添加之前的query参数
            filename: "static/imgs/[hash:8][ext][query]",
          },
        },
        {
          test: /\.(ttf|woff2?)$/,
          type: "asset/resource",
          generator: {
            filename: "static/media/[hash:8][ext][query]",
          },
        },
      ],
    },
    plugins: [],
    mode: "development",
  };
```

`type: "asset/resource"` 和 `type: "asset"` 的区别：

1. `type: "asset/resource"` 相当于 `file-loader`，将文件转化成 Webpack 能识别的资源，其他不做处理
2. `type: "asset"` 相当于 `url-loader`，将文件转化成 Webpack 能识别的资源，同时小于某个大小的资源会处理成 data URI 形式

4. 运行指令

```
npx webpack
```

打开 index.html 页面查看效果

处理其他资源

开发中可能还存在一些其他资源，如音视频等，我们也一起处理了

1. 配置

```
const path = require("path");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
```

```
    clean: true, // 自动将上次打包目录资源清空
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.less$/,
        use: ["style-loader", "css-loader", "less-loader"],
      },
      {
        test: /\.s[ac]ss$/,
        use: ["style-loader", "css-loader", "sass-loader"],
      },
      {
        test: /\.styl$/,
        use: ["style-loader", "css-loader", "stylus-loader"],
      },
      {
        test: /\.(png|jpe?g|gif|webp)$/,
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
          },
        },
        generator: {
          // 将图片文件输出到 static/imgs 目录中
          // 将图片文件命名 [hash:8][ext][query]
          // [hash:8]: hash值取8位
          // [ext]: 使用之前的文件扩展名
          // [query]: 添加之前的query参数
          filename: "static/imgs/[hash:8][ext][query]",
        },
      },
      {
        test: /\.(ttf|woff2?|map4|map3|avi)$/,
        type: "asset/resource",
        generator: {
          filename: "static/media/[hash:8][ext][query]",
        },
      },
    ],
  },
  plugins: [],
  mode: "development",
};
```

就是在处理字体图标资源基础上增加其他文件类型，统一处理即可

2. 运行指令

```
npx webpack
```

打开 index.html 页面查看效果

处理 js 资源

有人可能会问，js 资源 Webpack 不能已经处理了吗，为什么我们还要处理呢?

原因是 Webpack 对 js 处理是有限的，只能编译 js 中 ES 模块化语法，不能编译其他语法，导致 js 不能在 IE 等浏览器运行，所以我们希望做一些兼容性处理。

其次开发中，团队对代码格式是有严格要求的，我们不能由肉眼去检测代码格式，需要使用专业的工具来检测。

- 针对 js 兼容性处理，我们使用 Babel 来完成
- 针对代码格式，我们使用 Eslint 来完成

我们先完成 Eslint，检测代码格式无误后，在由 Babel 做代码兼容性处理

Eslint

可组装的 JavaScript 和 JSX 检查工具。

这句话意思就是：它是用来检测 js 和 jsx 语法的工具，可以配置各项功能

我们使用 Eslint，关键是写 Eslint 配置文件，里面写上各种 rules 规则，将来运行 Eslint 时就会以写的规则对代码进行检查

1. 配置文件

配置文件由很多种写法：

- `.eslintrc.*`：新建文件，位于项目根目录
  - `.eslintrc`
  - `.eslintrc.js`
  - `.eslintrc.json`
  - 区别在于配置格式不一样
- `package.json` 中 `eslintConfig`：不需要创建文件，在原有文件基础上写

ESLint 会查找和自动读取它们，所以以上配置文件只需要存在一个即可

2. 具体配置

我们以 `.eslintrc.js` 配置文件为例：

```
module.exports = {
  // 解析选项
  parserOptions: {},
  // 具体检查规则
  rules: {},
  // 继承其他规则
  extends: [],
  // ...
  // 其他规则详见：https://eslint.bootcss.com/docs/user-guide/configuring
};
```

1. parserOptions 解析选项

```
parserOptions: {
  ecmaVersion: 6, // ES 语法版本
  sourceType: "module", // ES 模块化
  ecmaFeatures: { // ES 其他特性
    jsx: true // 如果是 React 项目，就需要开启 jsx 语法
  }
}
```

2. rules 具体规则

- `"off"` 或 `0` - 关闭规则
- `"warn"` 或 `1` - 开启规则，使用警告级别的错误： `warn` (不会导致程序退出)
- `"error"` 或 `2` - 开启规则，使用错误级别的错误： `error` (当被触发的时候，程序会退出)

```
rules: {
  semi: "error", // 禁止使用分号
  'array-callback-return': 'warn', // 强制数组方法的回调函数中有 return 语句，否则警告
  'default-case': [
    'warn', // 要求 switch 语句中有 default 分支，否则警告
    { commentPattern: '^no default$' } // 允许在最后注释 no default，就不会有警告了
  ],
  eqeqeq: [
    'warn', // 强制使用 === 和 !==，否则警告
    'smart' // https://eslint.bootcss.com/docs/rules/eqeqeq#smart 除了少数情况下不会有警告
  ],
}
```

更多规则详见：规则文档

3. extends 继承

开发中一点点写 rules 规则太费劲了，所以有更好的办法，继承现有的规则。

现有以下较为有名的规则：

- Eslint 官方的规则：`eslint:recommended`
- Vue Cli 官方的规则：`plugin:vue/essential`
- React Cli 官方的规则：`react-app`

```
// 例如在React项目中，我们可以这样写配置
module.exports = {
  extends: ["react-app"],
  rules: {
    // 我们的规则会覆盖掉react-app的规则
    // 所以想要修改规则直接改就是了
    eqeqeq: ["warn", "smart"],
  },
};
```

3. 在 Webpack 中使用

1. 下载包

```
npm i eslint-webpack-plugin eslint -D
```

2. 定义 Eslint 配置文件

- .eslintrc.js

```
module.exports = {
  // 继承 Eslint 规则
  extends: ["eslint:recommended"],
  env: {
    node: true, // 启用node中全局变量
    browser: true, // 启用浏览器中全局变量
  },
  parserOptions: {
    ecmaVersion: 6,
    sourceType: "module",
  },
  rules: {
    "no-var": 2, // 不能使用 var 定义变量
  },
};
```

3. 修改 js 文件代码

- main.js

```
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/iconfont.css";
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";
import "./styl/index.styl";

var result1 = count(2, 1);
console.log(result1);
var result2 = sum(1, 2, 3, 4);
console.log(result2);
```

1. 配置

- webpack.config.js

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true, // 自动将上次打包目录资源清空
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.less$/,
        use: ["style-loader", "css-loader", "less-loader"],
      },
      {
        test: /\.s[ac]ss$/,
        use: ["style-loader", "css-loader", "sass-loader"],
      },
```

```
        {
          test: /\.styl$/,
          use: ["style-loader", "css-loader", "stylus-loader"],
        },
        {
          test: /\.(png|jpe?g|gif|webp)$/,
          type: "asset",
          parser: {
            dataUrlCondition: {
              maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
            },
          },
          generator: {
            // 将图片文件输出到 static/imgs 目录中
            // 将图片文件命名 [hash:8][ext][query]
            // [hash:8]: hash值取8位
            // [ext]: 使用之前的文件扩展名
            // [query]: 添加之前的query参数
            filename: "static/imgs/[hash:8][ext][query]",
          },
        },
        {
          test: /\.(ttf|woff2?)$/,
          type: "asset/resource",
          generator: {
            filename: "static/media/[hash:8][ext][query]",
          },
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "src"),
      }),
    ],
    mode: "development",
  };
```

5. 运行指令

```
npx webpack
```

在控制台查看 Eslint 检查效果

4. VSCode Eslint 插件

打开 VSCode，下载 Eslint 插件，即可不用编译就能看到错误，可以提前解决

但是此时就会对项目所有文件默认进行 Eslint 检查了，我们 dist 目录下的打包后文件就会报错。但是我们只需要检查 src 下面的文件，不需要检查 dist 下面的文件。

所以可以使用 Eslint 忽略文件解决。在项目根目录新建下面文件:

- `.eslintignore`

```
# 忽略dist目录下所有文件
dist
```

Babel

JavaScript 编译器。

主要用于将 ES6 语法编写的代码转换为向后兼容的 JavaScript 语法，以便能够运行在当前和旧版本的浏览器或其他环境中

1. 配置文件

配置文件由很多种写法：

- `babel.config.*`：新建文件，位于项目根目录
  - `babel.config.js`
  - `babel.config.json`
- `.babelrc.*`：新建文件，位于项目根目录
  - `.babelrc`
  - `.babelrc.js`
  - `.babelrc.json`
- `package.json` 中 `babel`：不需要创建文件，在原有文件基础上写

Babel 会查找和自动读取它们，所以以上配置文件只需要存在一个即可

2. 具体配置

我们以 `babel.config.js` 配置文件为例：

```
module.exports = {
  // 预设
  presets: [],
};
```

1. presets 预设

简单理解：就是一组 Babel 插件, 扩展 Babel 功能

- `@babel/preset-env`：一个智能预设，允许您使用最新的 JavaScript。
- `@babel/preset-react`：一个用来编译 React jsx 语法的预设
- `@babel/preset-typescript`：一个用来编译 TypeScript 语法的预设

3. 在 Webpack 中使用

1. 下载包

```
npm i babel-loader @babel/core @babel/preset-env -D
```

2. 定义 Babel 配置文件

- babel.config.js

```
module.exports = {
  presets: ["@babel/preset-env"],
};
```

3. 修改 js 文件代码

- main.js

```
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/iconfont.css";
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";
import "./styl/index.styl";

const result1 = count(2, 1);
console.log(result1);
const result2 = sum(1, 2, 3, 4);
console.log(result2);
```

4. 配置

- webpack.config.js

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");

module.exports = {
  entry: "./src/main.js",
```

```
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true, // 自动将上次打包目录资源清空
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.less$/,
        use: ["style-loader", "css-loader", "less-loader"],
      },
      {
        test: /\.s[ac]ss$/,
        use: ["style-loader", "css-loader", "sass-loader"],
      },
      {
        test: /\.styl$/,
        use: ["style-loader", "css-loader", "stylus-loader"],
      },
      {
        test: /\.(png|jpe?g|gif|webp)$/,
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
          },
        },
        generator: {
          // 将图片文件输出到 static/imgs 目录中
          // 将图片文件命名 [hash:8][ext][query]
          // [hash:8]: hash值取8位
          // [ext]: 使用之前的文件扩展名
          // [query]: 添加之前的query参数
          filename: "static/imgs/[hash:8][ext][query]",
        },
      },
      {
        test: /\.(ttf|woff2?)$/,
        type: "asset/resource",
        generator: {
          filename: "static/media/[hash:8][ext][query]",
        },
      },
      {
        test: /\.js$/,
```

```
          exclude: /node_modules/, // 排除node_modules代码不编译
          loader: "babel-loader",
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "src"),
      }),
    ],
    mode: "development",
  };
```

5. 运行指令

```
npx webpack
```

打开打包后的 `dist/static/js/main.js` 文件查看，会发现箭头函数等 ES6 语法已经转换了

• 处理 Html 资源

- 1. 下载包

```
npm i html-webpack-plugin -D
```

- 2. 配置

- webpack.config.js

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true, // 自动将上次打包目录资源清空
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
```

```
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.less$/,
        use: ["style-loader", "css-loader", "less-loader"],
      },
      {
        test: /\.s[ac]ss$/,
        use: ["style-loader", "css-loader", "sass-loader"],
      },
      {
        test: /\.styl$/,
        use: ["style-loader", "css-loader", "stylus-loader"],
      },
      {
        test: /\.(png|jpe?g|gif|webp)$/,
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
          },
        },
        generator: {
          // 将图片文件输出到 static/imgs 目录中
          // 将图片文件命名 [hash:8][ext][query]
          // [hash:8]: hash值取8位
          // [ext]: 使用之前的文件扩展名
          // [query]: 添加之前的query参数
          filename: "static/imgs/[hash:8][ext][query]",
        },
      },
      {
        test: /\.(ttf|woff2?)$/,
        type: "asset/resource",
        generator: {
          filename: "static/media/[hash:8][ext][query]",
        },
      },
      {
        test: /\.js$/,
        exclude: /node_modules/, // 排除node_modules代码不编译
        loader: "babel-loader",
      },
    ],
  },
  plugins: [
    new ESLintWebpackPlugin({
      // 指定检查文件的根目录
      context: path.resolve(__dirname, "src"),
```

```
    }),
    new HtmlWebpackPlugin({
      // 以 public/index.html 为模板创建文件
      // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
      template: path.resolve(__dirname, "public/index.html"),
    }),
  ],
  mode: "development",
};
```

3. 修改 index.html

去掉引入的 js 文件，因为 HtmlWebpackPlugin 会自动引入

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>webpack5</title>
  </head>
  <body>
    <h1>Hello Webpack5</h1>
    <div class="box1"></div>
    <div class="box2"></div>
    <div class="box3"></div>
    <div class="box4"></div>
    <div class="box5"></div>
    <i class="iconfont icon-arrow-down"></i>
    <i class="iconfont icon-ashbin"></i>
    <i class="iconfont icon-browse"></i>
  </body>
</html>
```

4. 运行指令

```
npx webpack
```

此时 dist 目录就会输出一个 index.html 文件

• 开发服务器&自动化

每次写完代码都需要手动输入指令才能编译代码，太麻烦了，我们希望一切自动化

1. 下载包

```
npm i webpack-dev-server -D
```

2. 配置

- webpack.config.js

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true, // 自动将上次打包目录资源清空
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.less$/,
        use: ["style-loader", "css-loader", "less-loader"],
      },
      {
        test: /\.s[ac]ss$/,
        use: ["style-loader", "css-loader", "sass-loader"],
      },
      {
        test: /\.styl$/,
        use: ["style-loader", "css-loader", "stylus-loader"],
      },
      {
        test: /\.(png|jpe?g|gif|webp)$/,
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
          },
        },
        generator: {
          // 将图片文件输出到 static/imgs 目录中
```

```
            // 将图片文件命名 [hash:8][ext][query]
            // [hash:8]: hash值取8位
            // [ext]: 使用之前的文件扩展名
            // [query]: 添加之前的query参数
            filename: "static/imgs/[hash:8][ext][query]",
          },
        },
        {
          test: /\.(ttf|woff2?)$/,
          type: "asset/resource",
          generator: {
            filename: "static/media/[hash:8][ext][query]",
          },
        },
        {
          test: /\.js$/,
          exclude: /node_modules/, // 排除node_modules代码不编译
          loader: "babel-loader",
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "src"),
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "public/index.html"),
      }),
    ],
    // 开发服务器
    devServer: {
      host: "localhost", // 启动服务器域名
      port: "3000", // 启动服务器端口号
      open: true, // 是否自动打开浏览器
    },
    mode: "development",
  };
```

3. 运行指令

```
npx webpack serve
```

注意运行指令发生了变化

并且当你使用开发服务器时，所有代码都会在内存中编译打包，并不会输出到 dist 目录下。

开发时我们只关心代码能运行，有效果即可，至于代码被编译成什么样子，我们并不需要知道。

• 生产模式介绍

生产模式是开发完成代码后，我们需要得到代码将来部署上线。

这个模式下我们主要对代码进行优化，让其运行性能更好。

优化主要从两个角度出发:

1. 优化代码运行性能
2. 优化代码打包速度

– 生产模式准备

我们分别准备两个配置文件来放不同的配置

1. 文件目录

```
├── webpack-test（项目根目录）
    ├── config（Webpack配置文件目录）
    │    ├── webpack.dev.js(开发模式配置文件)
    │    └── webpack.prod.js(生产模式配置文件)
    ├── node_modules（下载包存放目录）
    ├── src（项目源码目录，除了html其他都在src里面）
    │    └── 略
    ├── public（项目html文件）
    │    └── index.html
    ├── .eslintrc.js(Eslint配置文件)
    ├── babel.config.js(Babel配置文件)
    └── package.json（包的依赖管理配置文件）
```

2. 修改 webpack.dev.js

因为文件目录变了，所以所有绝对路径需要回退一层目录才能找到对应的文件

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: undefined, // 开发模式没有输出，不需要指定输出目录
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    // clean: true, // 开发模式没有输出，不需要清空输出结果
  },
  module: {
    rules: [
      {
```

```
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.less$/,
        use: ["style-loader", "css-loader", "less-loader"],
      },
      {
        test: /\.s[ac]ss$/,
        use: ["style-loader", "css-loader", "sass-loader"],
      },
      {
        test: /\.styl$/,
        use: ["style-loader", "css-loader", "stylus-loader"],
      },
      {
        test: /\.(png|jpe?g|gif|webp)$/,
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
          },
        },
        generator: {
          // 将图片文件输出到 static/imgs 目录中
          // 将图片文件命名 [hash:8][ext][query]
          // [hash:8]: hash值取8位
          // [ext]: 使用之前的文件扩展名
          // [query]: 添加之前的query参数
          filename: "static/imgs/[hash:8][ext][query]",
        },
      },
      {
        test: /\.(ttf|woff2?)$/,
        type: "asset/resource",
        generator: {
          filename: "static/media/[hash:8][ext][query]",
        },
      },
      {
        test: /\.js$/,
        exclude: /node_modules/, // 排除node_modules代码不编译
        loader: "babel-loader",
      },
    ],
  },
  plugins: [
    new ESLintWebpackPlugin({
      // 指定检查文件的根目录
```

```
      context: path.resolve(__dirname, "../src"),
    }),
    new HtmlWebpackPlugin({
      // 以 public/index.html 为模板创建文件
      // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
      template: path.resolve(__dirname, "../public/index.html"),
    }),
  ],
  // 其他省略
  devServer: {
    host: "localhost", // 启动服务器域名
    port: "3000", // 启动服务器端口号
    open: true, // 是否自动打开浏览器
  },
  mode: "development",
};
```

运行开发模式的指令：

```
npx webpack serve --config ./config/webpack.dev.js
```

3. 修改 webpack.prod.js

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true,
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.less$/,
        use: ["style-loader", "css-loader", "less-loader"],
      },
      {
        test: /\.s[ac]ss$/,
```

```
          use: ["style-loader", "css-loader", "sass-loader"],
        },
        {
          test: /\.styl$/,
          use: ["style-loader", "css-loader", "stylus-loader"],
        },
        {
          test: /\.(png|jpe?g|gif|webp)$/,
          type: "asset",
          parser: {
            dataUrlCondition: {
              maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
            },
          },
          generator: {
            // 将图片文件输出到 static/imgs 目录中
            // 将图片文件命名 [hash:8][ext][query]
            // [hash:8]: hash值取8位
            // [ext]: 使用之前的文件扩展名
            // [query]: 添加之前的query参数
            filename: "static/imgs/[hash:8][ext][query]",
          },
        },
        {
          test: /\.(ttf|woff2?)$/,
          type: "asset/resource",
          generator: {
            filename: "static/media/[hash:8][ext][query]",
          },
        },
        {
          test: /\.js$/,
          exclude: /node_modules/, // 排除node_modules代码不编译
          loader: "babel-loader",
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
    ],
    // devServer: {
    //   host: "localhost", // 启动服务器域名
    //   port: "3000", // 启动服务器端口号
```

```
    //   open: true, // 是否自动打开浏览器
    // },
    mode: "production",
  };
```

运行生产模式的指令：

```
npx webpack --config ./config/webpack.prod.js
```

4. 配置运行指令

为了方便运行不同模式的指令，我们将指令定义在 package.json 中 scripts 里面

```
// package.json
{
  // 其他省略
  "scripts": {
    "start": "npm run dev",
    "dev": "npx webpack serve --config ./config/webpack.dev.js",
    "build": "npx webpack --config ./config/webpack.prod.js"
  }
}
```

以后启动指令：

- 开发模式：`npm start` 或 `npm run dev`
- 生产模式：`npm run build`

Css 处理

提取 Css 成单独文件

Css 文件目前被打包到 js 文件中，当 js 文件加载时，会创建一个 style 标签来生成样式

这样对于网站来说，会出现闪屏现象，用户体验不好

我们应该是单独的 Css 文件，通过 link 标签加载性能才好

1. 下载包

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

2. 配置

- webpack.prod.js

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true,
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: [MiniCssExtractPlugin.loader, "css-loader"],
      },
      {
        test: /\.less$/,
        use: [MiniCssExtractPlugin.loader, "css-loader", "less-loader"],
      },
      {
        test: /\.s[ac]ss$/,
        use: [MiniCssExtractPlugin.loader, "css-loader", "sass-loader"],
      },
      {
        test: /\.styl$/,
        use: [MiniCssExtractPlugin.loader, "css-loader", "stylus-loader"],
      },
      {
        test: /\.(png|jpe?g|gif|webp)$/,
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
          },
        },
        generator: {
          // 将图片文件输出到 static/imgs 目录中
          // 将图片文件命名 [hash:8][ext][query]
          // [hash:8]: hash值取8位
          // [ext]: 使用之前的文件扩展名
          // [query]: 添加之前的query参数
          filename: "static/imgs/[hash:8][ext][query]",
```

```
          },
        },
        {
          test: /\.(ttf|woff2?)$/,
          type: "asset/resource",
          generator: {
            filename: "static/media/[hash:8][ext][query]",
          },
        },
        {
          test: /\.js$/,
          exclude: /node_modules/, // 排除node_modules代码不编译
          loader: "babel-loader",
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      // 提取css成单独文件
      new MiniCssExtractPlugin({
        // 定义输出文件名和目录
        filename: "static/css/main.css",
      }),
    ],
    // devServer: {
    //   host: "localhost", // 启动服务器域名
    //   port: "3000", // 启动服务器端口号
    //   open: true, // 是否自动打开浏览器
    // },
    mode: "production",
  };
```

3. 运行指令

```
npm run build
```

Css 兼容性处理

1. 下载包

```
npm i postcss-loader postcss postcss-preset-env -D
```

2. 配置

- webpack.prod.js

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true,
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: [
          MiniCssExtractPlugin.loader,
          "css-loader",
          {
            loader: "postcss-loader",
            options: {
              postcssOptions: {
                plugins: [
                  "postcss-preset-env", // 能解决大多数样式兼容性问题
                ],
              },
            },
          },
        ],
      },
      {
        test: /\.less$/,
        use: [
          MiniCssExtractPlugin.loader,
          "css-loader",
          {
            loader: "postcss-loader",
```

```
              options: {
                postcssOptions: {
                  plugins: [
                    "postcss-preset-env", // 能解决大多数样式兼容性问题
                  ],
                },
              },
            },
            "less-loader",
          ],
        },
        {
          test: /\.s[ac]ss$/,
          use: [
            MiniCssExtractPlugin.loader,
            "css-loader",
            {
              loader: "postcss-loader",
              options: {
                postcssOptions: {
                  plugins: [
                    "postcss-preset-env", // 能解决大多数样式兼容性问题
                  ],
                },
              },
            },
            "sass-loader",
          ],
        },
        {
          test: /\.styl$/,
          use: [
            MiniCssExtractPlugin.loader,
            "css-loader",
            {
              loader: "postcss-loader",
              options: {
                postcssOptions: {
                  plugins: [
                    "postcss-preset-env", // 能解决大多数样式兼容性问题
                  ],
                },
              },
            },
            "stylus-loader",
          ],
        },
        {
          test: /\.(png|jpe?g|gif|webp)$/,
          type: "asset",
          parser: {
```

```
            dataUrlCondition: {
              maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
            },
          },
          generator: {
            // 将图片文件输出到 static/imgs 目录中
            // 将图片文件命名 [hash:8][ext][query]
            // [hash:8]: hash值取8位
            // [ext]: 使用之前的文件扩展名
            // [query]: 添加之前的query参数
            filename: "static/imgs/[hash:8][ext][query]",
          },
        },
        {
          test: /\.(ttf|woff2?)$/,
          type: "asset/resource",
          generator: {
            filename: "static/media/[hash:8][ext][query]",
          },
        },
        {
          test: /\.js$/,
          exclude: /node_modules/, // 排除node_modules代码不编译
          loader: "babel-loader",
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      // 提取css成单独文件
      new MiniCssExtractPlugin({
        // 定义输出文件名和目录
        filename: "static/css/main.css",
      }),
    ],
    // devServer: {
    //   host: "localhost", // 启动服务器域名
    //   port: "3000", // 启动服务器端口号
    //   open: true, // 是否自动打开浏览器
    // },
    mode: "production",
  };
```

3. 控制兼容性

我们可以在 `package.json` 文件中添加 `browserslist` 来控制样式的兼容性做到什么程度。

```
{
  // 其他省略
  "browserslist": ["ie ≥ 8"]
}
```

想要知道更多的 `browserslist` 配置，查看browserslist 文档

以上为了测试兼容性所以设置兼容浏览器 ie8 以上。

实际开发中我们一般不考虑旧版本浏览器了，所以我们可以这样设置：

```
{
  // 其他省略
  "browserslist": ["last 2 version", "> 1%", "not dead"]
}
```

4. 合并配置

- webpack.prod.js

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

// 获取处理样式的Loaders
const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

```

```
module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true,
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: getStyleLoaders(),
      },
      {
        test: /\.less$/,
        use: getStyleLoaders("less-loader"),
      },
      {
        test: /\.s[ac]ss$/,
        use: getStyleLoaders("sass-loader"),
      },
      {
        test: /\.styl$/,
        use: getStyleLoaders("stylus-loader"),
      },
      {
        test: /\.(png|jpe?g|gif|webp)$/,
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
          },
        },
        generator: {
          // 将图片文件输出到 static/imgs 目录中
          // 将图片文件命名 [hash:8][ext][query]
          // [hash:8]: hash值取8位
          // [ext]: 使用之前的文件扩展名
          // [query]: 添加之前的query参数
          filename: "static/imgs/[hash:8][ext][query]",
        },
      },
      {
        test: /\.(ttf|woff2?)$/,
        type: "asset/resource",
        generator: {
          filename: "static/media/[hash:8][ext][query]",
        },
      },
```

```
        {
          test: /\.js$/,
          exclude: /node_modules/, // 排除node_modules代码不编译
          loader: "babel-loader",
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      // 提取css成单独文件
      new MiniCssExtractPlugin({
        // 定义输出文件名和目录
        filename: "static/css/main.css",
      }),
    ],
    // devServer: {
    //   host: "localhost", // 启动服务器域名
    //   port: "3000", // 启动服务器端口号
    //   open: true, // 是否自动打开浏览器
    // },
    mode: "production",
  };
```

5. 运行指令

```
npm run build
```

Css 压缩

1. 下载包

```
npm i css-minimizer-webpack-plugin -D
```

2. 配置

- webpack.prod.js

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
```

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");

// 获取处理样式的Loaders
const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true,
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: getStyleLoaders(),
      },
      {
        test: /\.less$/,
        use: getStyleLoaders("less-loader"),
      },
      {
        test: /\.s[ac]ss$/,
        use: getStyleLoaders("sass-loader"),
      },
      {
        test: /\.styl$/,
        use: getStyleLoaders("stylus-loader"),
      },
      {
```

```
        test: /\.(png|jpe?g|gif|webp)$/,
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
          },
        },
        generator: {
          // 将图片文件输出到 static/imgs 目录中
          // 将图片文件命名 [hash:8][ext][query]
          // [hash:8]: hash值取8位
          // [ext]: 使用之前的文件扩展名
          // [query]: 添加之前的query参数
          filename: "static/imgs/[hash:8][ext][query]",
        },
      },
      {
        test: /\.(ttf|woff2?)$/,
        type: "asset/resource",
        generator: {
          filename: "static/media/[hash:8][ext][query]",
        },
      },
      {
        test: /\.js$/,
        exclude: /node_modules/, // 排除node_modules代码不编译
        loader: "babel-loader",
      },
    ],
  },
  plugins: [
    new ESLintWebpackPlugin({
      // 指定检查文件的根目录
      context: path.resolve(__dirname, "../src"),
    }),
    new HtmlWebpackPlugin({
      // 以 public/index.html 为模板创建文件
      // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
      template: path.resolve(__dirname, "../public/index.html"),
    }),
    // 提取css成单独文件
    new MiniCssExtractPlugin({
      // 定义输出文件名和目录
      filename: "static/css/main.css",
    }),
    // css压缩
    new CssMinimizerPlugin(),
  ],
  // devServer: {
  //   host: "localhost", // 启动服务器域名
  //   port: "3000", // 启动服务器端口号
```

```
105     //   open: true, // 是否自动打开浏览器
106     // },
107     mode: "production",
108   };
```

3. 运行指令

```
1   npm run build
```

• html 压缩

默认生产模式已经开启了：html 压缩和 js 压缩

不需要额外进行配置

• 总结

本章节我们学会了 Webpack 基本使用，掌握了以下功能：

1. 两种开发模式

- 开发模式：代码能编译自动化运行
- 生产模式：代码编译优化输出

2. Webpack 基本功能

- 开发模式：可以编译 ES Module 语法
- 生产模式：可以编译 ES Module 语法，压缩 js 代码

3. Webpack 配置文件

- 5 个核心概念
    - entry
    - output
    - loader
    - plugins
    - mode
- devServer 配置

4. Webpack 脚本指令用法

- `webpack` 直接打包输出
- `webpack serve` 启动开发服务器，内存编译打包没有输出

高级优化

• 介绍

本章节主要介绍 Webpack 高级配置。

所谓高级配置其实就是进行 Webpack 优化，让我们代码在编译/运行时性能更好~

我们会从以下角度来进行优化：

1. 提升开发体验
2. 提升打包构建速度
3. 减少代码体积
4. 优化代码运行性能

• 提升开发体验

- SourceMap

为什么

开发时我们运行的代码是经过 webpack 编译后的，例如下面这个样子：

```
/*
 * 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 (https://webpack.js.org/configuration/devtool/)
 * or disable the default devtool with "devtool: false".
 * If you are looking for production-ready output files, see mode: "production" (https://webpack.js.org/configuration/mode/).
 */
/******/ (() => { // webpackBootstrap
/******/ 	"use strict";
/******/ 	var __webpack_modules__ = ({

/***/ "./node_modules/css-loader/dist/cjs.js!./node_modules/less-loader/dist/cjs.js!./src/less/index.less":
/*!*****************************************************************************************************!*\
  !*** ./node_modules/css-loader/dist/cjs.js!./node_modules/less-loader/dist/cjs.js!./src/less/index.less ***!
  \*****************************************************************************************************/
/***/ ((module, __webpack_exports__, __webpack_require__) => {

```

```
19  eval("__webpack_require__.r(__webpack_exports__);\n/* harmony export */
    __webpack_require__.d(__webpack_exports__, {\n/* harmony export */
    \"default\": () ⇒ (__WEBPACK_DEFAULT_EXPORT__)\n/* harmony export */ });\n/*
    harmony import */ var
    _node_modules_css_loader_dist_runtime_noSourceMaps_js__WEBPACK_IMPORTED_MODUL
    E_0__ = __webpack_require__(/*! ../../node_modules/css-
    loader/dist/runtime/noSourceMaps.js */ \"./node_modules/css-
    loader/dist/runtime/noSourceMaps.js\");\n/* harmony import */ var
    _node_modules_css_loader_dist_runtime_noSourceMaps_js__WEBPACK_IMPORTED_MODUL
    E_0___default =
    /*#__PURE__*/__webpack_require__.n(_node_modules_css_loader_dist_runtime_noSo
    urceMaps_js__WEBPACK_IMPORTED_MODULE_0__);\n/* harmony import */ var
    _node_modules_css_loader_dist_runtime_api_js__WEBPACK_IMPORTED_MODULE_1__ =
    __webpack_require__(/*! ../../node_modules/css-loader/dist/runtime/api.js */
    \"./node_modules/css-loader/dist/runtime/api.js\");\n/* harmony import */ var
    _node_modules_css_loader_dist_runtime_api_js__WEBPACK_IMPORTED_MODULE_1___def
    ault =
    /*#__PURE__*/__webpack_require__.n(_node_modules_css_loader_dist_runtime_api_
    js__WEBPACK_IMPORTED_MODULE_1__);\n// Imports\n\n\nvar
    ___CSS_LOADER_EXPORT___ =
    _node_modules_css_loader_dist_runtime_api_js__WEBPACK_IMPORTED_MODULE_1___def
    ault()
    ((_node_modules_css_loader_dist_runtime_noSourceMaps_js__WEBPACK_IMPORTED_MOD
    ULE_0___default()));\n// Module\n___CSS_LOADER_EXPORT___.push([module.id,
    \".box2 {\\n  width: 100px;\\n  height: 100px;\\n  background-color:
    deeppink;\\n}\\n\", \"\"]);\n// Exports\n/* harmony default export */ const
    __WEBPACK_DEFAULT_EXPORT__ = (___CSS_LOADER_EXPORT___);\n\n\n//#
    sourceURL=webpack://webpack5/./src/less/index.less?./node_modules/css-
    loader/dist/cjs.js!./node_modules/less-loader/dist/cjs.js");
20
21  /***/ }),
22  // 其他省略
```

所有 css 和 js 合并成了一个文件，并且多了其他代码。此时如果代码运行出错那么提示代码错误位置我们是看不懂的。一旦将来开发代码文件很多，那么很难去发现错误出现在哪里。

所以我们需要更加准确的错误提示，来帮助我们更好的开发代码。

是什么

SourceMap（源代码映射）是一个用来生成源代码与构建后代码一一映射的文件的方案。

它会生成一个 xxx.map 文件，里面包含源代码和构建后代码每一行、每一列的映射关系。当构建后代码出错了，会通过 xxx.map 文件，从构建后代码出错位置找到映射后源代码出错位置，从而让浏览器提示源代码文件出错位置，帮助我们更快的找到错误根源。

怎么用

通过查看Webpack DevTool 文档可知，SourceMap 的值有很多种情况.

但实际开发时我们只需要关注两种情况即可：

- 开发模式：`cheap-module-source-map`
  - 优点：打包编译速度快，只包含行映射
  - 缺点：没有列映射

```
module.exports = {
  // 其他省略
  mode: "development",
  devtool: "cheap-module-source-map",
};
```

- 生产模式：`source-map`
  - 优点：包含行/列映射
  - 缺点：打包编译速度更慢

```
module.exports = {
  // 其他省略
  mode: "production",
  devtool: "source-map",
};
```

提升打包构建速度

HotModuleReplacement

为什么

开发时我们修改了其中一个模块代码，Webpack 默认会将所有模块全部重新打包编译，速度很慢。

所以我们需要做到修改某个模块代码，就只有这个模块代码需要重新打包编译，其他模块不变，这样打包速度就能很快。

是什么

HotModuleReplacement（HMR/热模块替换）：在程序运行中，替换、添加或删除模块，而无需重新加载整个页面。

怎么用

1. 基本配置

```
module.exports = {
  // 其他省略
  devServer: {
    host: "localhost", // 启动服务器域名
    port: "3000", // 启动服务器端口号
    open: true, // 是否自动打开浏览器
    hot: true, // 开启HMR功能（只能用于开发环境，生产环境不需要了）
  },
};
```

此时 css 样式经过 style-loader 处理，已经具备 HMR 功能了。
但是 js 还不行。

2. JS 配置

```
// main.js
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/iconfont.css";
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";
import "./styl/index.styl";

const result1 = count(2, 1);
console.log(result1);
const result2 = sum(1, 2, 3, 4);
console.log(result2);

// 判断是否支持HMR功能
if (module.hot) {
  module.hot.accept("./js/count.js", function (count) {
    const result1 = count(2, 1);
    console.log(result1);
  });

  module.hot.accept("./js/sum.js", function (sum) {
    const result2 = sum(1, 2, 3, 4);
    console.log(result2);
  });
}
```

上面这样写会很麻烦，所以实际开发我们会使用其他 loader 来解决。

比如：vue-loader，react-hot-loader。

OneOf

为什么

打包时每个文件都会经过所有 loader 处理，虽然因为 `test` 正则原因实际没有处理上，但是都要过一遍。比较慢。

是什么

顾名思义就是只能匹配上一个 loader, 剩下的就不匹配了。

怎么用

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: undefined, // 开发模式没有输出，不需要指定输出目录
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    // clean: true, // 开发模式没有输出，不需要清空输出结果
  },
  module: {
    rules: [
      {
        oneOf: [
          {
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
            use: ["style-loader", "css-loader"],
          },
          {
            test: /\.less$/,
            use: ["style-loader", "css-loader", "less-loader"],
          },
          {
            test: /\.s[ac]ss$/,
            use: ["style-loader", "css-loader", "sass-loader"],
          },
          {
            test: /\.styl$/,
            use: ["style-loader", "css-loader", "stylus-loader"],
          },
          {
            test: /\.(png|jpe?g|gif|webp)$/,
            type: "asset",
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
```

```
              },
            },
            generator: {
              // 将图片文件输出到 static/imgs 目录中
              // 将图片文件命名 [hash:8][ext][query]
              // [hash:8]: hash值取8位
              // [ext]: 使用之前的文件扩展名
              // [query]: 添加之前的query参数
              filename: "static/imgs/[hash:8][ext][query]",
            },
          },
          {
            test: /\.(ttf|woff2?)$/,
            type: "asset/resource",
            generator: {
              filename: "static/media/[hash:8][ext][query]",
            },
          },
          {
            test: /\.js$/,
            exclude: /node_modules/, // 排除node_modules代码不编译
            loader: "babel-loader",
          },
        ],
      },
    ],
  },
  plugins: [
    new ESLintWebpackPlugin({
      // 指定检查文件的根目录
      context: path.resolve(__dirname, "../src"),
    }),
    new HtmlWebpackPlugin({
      // 以 public/index.html 为模板创建文件
      // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
      template: path.resolve(__dirname, "../public/index.html"),
    }),
  ],
  // 开发服务器
  devServer: {
    host: "localhost", // 启动服务器域名
    port: "3000", // 启动服务器端口号
    open: true, // 是否自动打开浏览器
    hot: true, // 开启HMR功能
  },
  mode: "development",
  devtool: "cheap-module-source-map",
};
```

生产模式也是如此配置。

Include/Exclude

为什么

开发时我们需要使用第三方的库或插件，所有文件都下载到 node_modules 中了。而这些文件是不需要编译可以直接使用的。

所以我们在对 js 文件处理时，要排除 node_modules 下面的文件。

是什么

- include

包含，只处理 xxx 文件

- exclude

排除，除了 xxx 文件以外其他文件都处理

怎么用

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: undefined, // 开发模式没有输出，不需要指定输出目录
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    // clean: true, // 开发模式没有输出，不需要清空输出结果
  },
  module: {
    rules: [
      {
        oneOf: [
          {
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
            use: ["style-loader", "css-loader"],
          },
          {
            test: /\.less$/,
            use: ["style-loader", "css-loader", "less-loader"],
          },
          {
            test: /\.s[ac]ss$/,
            use: ["style-loader", "css-loader", "sass-loader"],
          },
          {
            test: /\.styl$/,
            use: ["style-loader", "css-loader", "stylus-loader"],
```

```
            },
            {
              test: /\.(png|jpe?g|gif|webp)$/,
              type: "asset",
              parser: {
                dataUrlCondition: {
                  maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
                },
              },
              generator: {
                // 将图片文件输出到 static/imgs 目录中
                // 将图片文件命名 [hash:8][ext][query]
                // [hash:8]: hash值取8位
                // [ext]: 使用之前的文件扩展名
                // [query]: 添加之前的query参数
                filename: "static/imgs/[hash:8][ext][query]",
              },
            },
            {
              test: /\.(ttf|woff2?)$/,
              type: "asset/resource",
              generator: {
                filename: "static/media/[hash:8][ext][query]",
              },
            },
            {
              test: /\.js$/,
              // exclude: /node_modules/, // 排除node_modules代码不编译
              include: path.resolve(__dirname, "../src"), // 也可以用包含
              loader: "babel-loader",
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules", // 默认值
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
    ],
    // 开发服务器
    devServer: {
      host: "localhost", // 启动服务器域名
      port: "3000", // 启动服务器端口号
```

```
    open: true, // 是否自动打开浏览器
    hot: true, // 开启HMR功能
  },
  mode: "development",
  devtool: "cheap-module-source-map",
};
```

生产模式也是如此配置。

Cache

为什么

每次打包时 js 文件都要经过 Eslint 检查 和 Babel 编译，速度比较慢。

我们可以缓存之前的 Eslint 检查 和 Babel 编译结果，这样第二次打包时速度就会更快了。

是什么

对 Eslint 检查 和 Babel 编译结果进行缓存。

怎么用

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  entry: "./src/main.js",
  output: {
    path: undefined, // 开发模式没有输出，不需要指定输出目录
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    // clean: true, // 开发模式没有输出，不需要清空输出结果
  },
  module: {
    rules: [
      {
        oneOf: [
          {
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
            use: ["style-loader", "css-loader"],
          },
          {
            test: /\.less$/,
            use: ["style-loader", "css-loader", "less-loader"],
          },
          {
            test: /\.s[ac]ss$/,
            use: ["style-loader", "css-loader", "sass-loader"],
          },
```

```
            {
              test: /\.styl$/,
              use: ["style-loader", "css-loader", "stylus-loader"],
            },
            {
              test: /\.(png|jpe?g|gif|webp)$/,
              type: "asset",
              parser: {
                dataUrlCondition: {
                  maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
                },
              },
              generator: {
                // 将图片文件输出到 static/imgs 目录中
                // 将图片文件命名 [hash:8][ext][query]
                // [hash:8]: hash值取8位
                // [ext]: 使用之前的文件扩展名
                // [query]: 添加之前的query参数
                filename: "static/imgs/[hash:8][ext][query]",
              },
            },
            {
              test: /\.(ttf|woff2?)$/,
              type: "asset/resource",
              generator: {
                filename: "static/media/[hash:8][ext][query]",
              },
            },
            {
              test: /\.js$/,
              // exclude: /node_modules/, // 排除node_modules代码不编译
              include: path.resolve(__dirname, "../src"), // 也可以用包含
              loader: "babel-loader",
              options: {
                cacheDirectory: true, // 开启babel编译缓存
                cacheCompression: false, // 缓存文件不要压缩
              },
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules", // 默认值
        cache: true, // 开启缓存
        // 缓存目录
        cacheLocation: path.resolve(
          __dirname,
```

```
          "../node_modules/.cache/.eslintcache"
        ),
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
    ],
    // 开发服务器
    devServer: {
      host: "localhost", // 启动服务器域名
      port: "3000", // 启动服务器端口号
      open: true, // 是否自动打开浏览器
      hot: true, // 开启HMR功能
    },
    mode: "development",
    devtool: "cheap-module-source-map",
  };
```

Thread

为什么

当项目越来越庞大时，打包速度越来越慢，甚至于需要一个下午才能打包出来代码。这个速度是比较慢的。

我们想要继续提升打包速度，其实就是要提升 js 的打包速度，因为其他文件都比较少。

而对 js 文件处理主要就是 eslint 、babel、Terser 三个工具，所以我们要提升它们的运行速度。

我们可以开启多进程同时处理 js 文件，这样速度就比之前的单进程打包更快了。

是什么

多进程打包：开启电脑的多个进程同时干一件事，速度更快。

⚠需要注意：请仅在特别耗时的操作中使用，因为每个进程启动就有大约为 600ms 左右开销。

怎么用

我们启动进程的数量就是我们 CPU 的核数。

1. 如何获取 CPU 的核数，因为每个电脑都不一样。

```
// nodejs核心模块，直接使用
const os = require("os");
// cpu核数
const threads = os.cpus().length;
```

2. 下载包

```
npm i thread-loader -D
```

3. 使用

```
const os = require("os");
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");

// cpu核数
const threads = os.cpus().length;

// 获取处理样式的Loaders
const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true,
  },
  module: {
    rules: [
      {
        oneOf: [
          {
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
```

```
            use: getStyleLoaders(),
          },
          {
            test: /\.less$/,
            use: getStyleLoaders("less-loader"),
          },
          {
            test: /\.s[ac]ss$/,
            use: getStyleLoaders("sass-loader"),
          },
          {
            test: /\.styl$/,
            use: getStyleLoaders("stylus-loader"),
          },
          {
            test: /\.(png|jpe?g|gif|webp)$/,
            type: "asset",
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
              },
            },
            generator: {
              // 将图片文件输出到 static/imgs 目录中
              // 将图片文件命名 [hash:8][ext][query]
              // [hash:8]: hash值取8位
              // [ext]: 使用之前的文件扩展名
              // [query]: 添加之前的query参数
              filename: "static/imgs/[hash:8][ext][query]",
            },
          },
          {
            test: /\.(ttf|woff2?)$/,
            type: "asset/resource",
            generator: {
              filename: "static/media/[hash:8][ext][query]",
            },
          },
          {
            test: /\.js$/,
            // exclude: /node_modules/, // 排除node_modules代码不编译
            include: path.resolve(__dirname, "../src"), // 也可以用包含
            use: [
              {
                loader: "thread-loader", // 开启多进程
                options: {
                  workers: threads, // 数量
                },
              },
              {
                loader: "babel-loader",
```

```
                options: {
                  cacheDirectory: true, // 开启babel编译缓存
                },
              },
            ],
          },
        ],
      },
    ],
  },
  plugins: [
    new ESLintWebpackPlugin({
      // 指定检查文件的根目录
      context: path.resolve(__dirname, "../src"),
      exclude: "node_modules", // 默认值
      cache: true, // 开启缓存
      // 缓存目录
      cacheLocation: path.resolve(
        __dirname,
        "../node_modules/.cache/.eslintcache"
      ),
      threads, // 开启多进程
    }),
    new HtmlWebpackPlugin({
      // 以 public/index.html 为模板创建文件
      // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
      template: path.resolve(__dirname, "../public/index.html"),
    }),
    // 提取css成单独文件
    new MiniCssExtractPlugin({
      // 定义输出文件名和目录
      filename: "static/css/main.css",
    }),
    // css压缩
    // new CssMinimizerPlugin(),
  ],
  optimization: {
    minimize: true,
    minimizer: [
      // css压缩也可以写到optimization.minimizer里面，效果一样的
      new CssMinimizerPlugin(),
      // 当生产模式会默认开启TerserPlugin，但是我们需要进行其他配置，就要重新写了
      new TerserPlugin({
        parallel: threads // 开启多进程
      })
    ],
  },
  // devServer: {
  //   host: "localhost", // 启动服务器域名
  //   port: "3000", // 启动服务器端口号
  //   open: true, // 是否自动打开浏览器
```

```
148     // },
149     mode: "production",
150     devtool: "source-map",
151   };
```

我们目前打包的内容都很少，所以因为启动进程开销原因，使用多进程打包实际上会显著的让我们打包时间变得很长。

• 减少代码体积

– Tree Shaking

为什么

开发时我们定义了一些工具函数库，或者引用第三方工具函数库或组件库。

如果没有特殊处理的话我们打包时会引入整个库，但是实际上可能我们可能只用上极小部分的功能。

这样将整个库都打包进来，体积就太大了。

是什么

`Tree Shaking` 是一个术语，通常用于描述移除 JavaScript 中的没有使用上的代码。

⚠**注意：它依赖 `ES Module`。**

怎么用

Webpack 已经默认开启了这个功能，无需其他配置。

– Babel

为什么

Babel 为编译的每个文件都插入了辅助代码，使代码体积过大！

Babel 对一些公共方法使用了非常小的辅助代码，比如 `_extend`。默认情况下会被添加到每一个需要它的文件中。

你可以将这些辅助代码作为一个独立模块，来避免重复引入。

是什么

`@babel/plugin-transform-runtime`：禁用了 Babel 自动对每个文件的 runtime 注入，而是引入 `@babel/plugin-transform-runtime` 并且使所有辅助代码从这里引用。

怎么用

1. 下载包

```
1   npm i @babel/plugin-transform-runtime -D
```

2. 配置

```
const os = require("os");
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");

// cpu核数
const threads = os.cpus().length;

// 获取处理样式的Loaders
const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true,
  },
  module: {
    rules: [
      {
        oneOf: [
          {
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
            use: getStyleLoaders(),
          },
          {
            test: /\.less$/,
```

```
            use: getStyleLoaders("less-loader"),
          },
          {
            test: /\.s[ac]ss$/,
            use: getStyleLoaders("sass-loader"),
          },
          {
            test: /\.styl$/,
            use: getStyleLoaders("stylus-loader"),
          },
          {
            test: /\.(png|jpe?g|gif|webp)$/,
            type: "asset",
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
              },
            },
            generator: {
              // 将图片文件输出到 static/imgs 目录中
              // 将图片文件命名 [hash:8][ext][query]
              // [hash:8]: hash值取8位
              // [ext]: 使用之前的文件扩展名
              // [query]: 添加之前的query参数
              filename: "static/imgs/[hash:8][ext][query]",
            },
          },
          {
            test: /\.(ttf|woff2?)$/,
            type: "asset/resource",
            generator: {
              filename: "static/media/[hash:8][ext][query]",
            },
          },
          {
            test: /\.js$/,
            // exclude: /node_modules/, // 排除node_modules代码不编译
            include: path.resolve(__dirname, "../src"), // 也可以用包含
            use: [
              {
                loader: "thread-loader", // 开启多进程
                options: {
                  workers: threads, // 数量
                },
              },
              {
                loader: "babel-loader",
                options: {
                  cacheDirectory: true, // 开启babel编译缓存
                  cacheCompression: false, // 缓存文件不要压缩
```

```
                    plugins: ["@babel/plugin-transform-runtime"], // 减少代码体积
                  },
                },
              ],
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules", // 默认值
        cache: true, // 开启缓存
        // 缓存目录
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
        threads, // 开启多进程
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      // 提取css成单独文件
      new MiniCssExtractPlugin({
        // 定义输出文件名和目录
        filename: "static/css/main.css",
      }),
      // css压缩
      // new CssMinimizerPlugin(),
    ],
    optimization: {
      minimizer: [
        // css压缩也可以写到optimization.minimizer里面，效果一样的
        new CssMinimizerPlugin(),
        // 当生产模式会默认开启TerserPlugin，但是我们需要进行其他配置，就要重新写了
        new TerserPlugin({
          parallel: threads, // 开启多进程
        }),
      ]
    ],
    // devServer: {
    //   host: "localhost", // 启动服务器域名
    //   port: "3000", // 启动服务器端口号
    //   open: true, // 是否自动打开浏览器
    // },
```

```
150     mode: "production",
151     devtool: "source-map",
152   };
```

Image Minimizer

为什么

开发如果项目中引用了较多图片，那么图片体积会比较大，将来请求速度比较慢。

我们可以对图片进行压缩，减少图片体积。

⚠注意：如果项目中图片都是在线链接，那么就不需要了。本地项目静态图片才需要进行压缩。

是什么

`image-minimizer-webpack-plugin`：用来压缩图片的插件

怎么用

1. 下载包

```
1   npm i image-minimizer-webpack-plugin imagemin -D
```

还有剩下包需要下载，有两种模式：

- 无损压缩

```
1   npm install imagemin-gifsicle imagemin-jpegtran imagemin-optipng imagemin-svgo
    -D
```

- 有损压缩

```
1   npm install imagemin-gifsicle imagemin-mozjpeg imagemin-pngquant imagemin-svgo
    -D
```

> “
> 有损/无损压缩的区别

2. 配置

我们以无损压缩配置为例：

```
1   const os = require("os");
2   const path = require("path");
3   const ESLintWebpackPlugin = require("eslint-webpack-plugin");
```

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");

// cpu核数
const threads = os.cpus().length;

// 获取处理样式的Loaders
const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true,
  },
  module: {
    rules: [
      {
        oneOf: [
          {
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
            use: getStyleLoaders(),
          },
          {
            test: /\.less$/,
            use: getStyleLoaders("less-loader"),
          },
          {
            test: /\.s[ac]ss$/,
```

```
55                use: getStyleLoaders("sass-loader"),
56              },
57              {
58                test: /\.styl$/,
59                use: getStyleLoaders("stylus-loader"),
60              },
61              {
62                test: /\.(png|jpe?g|gif|svg)$/,
63                type: "asset",
64                parser: {
65                  dataUrlCondition: {
66                    maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
67                  },
68                },
69                generator: {
70                  // 将图片文件输出到 static/imgs 目录中
71                  // 将图片文件命名 [hash:8][ext][query]
72                  // [hash:8]: hash值取8位
73                  // [ext]: 使用之前的文件扩展名
74                  // [query]: 添加之前的query参数
75                  filename: "static/imgs/[hash:8][ext][query]",
76                },
77              },
78              {
79                test: /\.(ttf|woff2?)$/,
80                type: "asset/resource",
81                generator: {
82                  filename: "static/media/[hash:8][ext][query]",
83                },
84              },
85              {
86                test: /\.js$/,
87                // exclude: /node_modules/, // 排除node_modules代码不编译
88                include: path.resolve(__dirname, "../src"), // 也可以用包含
89                use: [
90                  {
91                    loader: "thread-loader", // 开启多进程
92                    options: {
93                      workers: threads, // 数量
94                    },
95                  },
96                  {
97                    loader: "babel-loader",
98                    options: {
99                      cacheDirectory: true, // 开启babel编译缓存
100                     cacheCompression: false, // 缓存文件不要压缩
101                     plugins: ["@babel/plugin-transform-runtime"], // 减少代码体
    积
102                   },
103                 },
104               ],
```

```
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules", // 默认值
        cache: true, // 开启缓存
        // 缓存目录
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
        threads, // 开启多进程
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      // 提取css成单独文件
      new MiniCssExtractPlugin({
        // 定义输出文件名和目录
        filename: "static/css/main.css",
      }),
      // css压缩
      // new CssMinimizerPlugin(),
    ],
    optimization: {
      minimizer: [
        // css压缩也可以写到optimization.minimizer里面，效果一样的
        new CssMinimizerPlugin(),
        // 当生产模式会默认开启TerserPlugin，但是我们需要进行其他配置，就要重新写了
        new TerserPlugin({
          parallel: threads, // 开启多进程
        }),
        // 压缩图片
        new ImageMinimizerPlugin({
          minimizer: {
            implementation: ImageMinimizerPlugin.imageminGenerate,
            options: {
              plugins: [
                ["gifsicle", { interlaced: true }],
                ["jpegtran", { progressive: true }],
                ["optipng", { optimizationLevel: 5 }],
                [
                  "svgo",
                  {
```

```
                    plugins: [
                      "preset-default",
                      "prefixIds",
                      {
                        name: "sortAttrs",
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
      ],
    },
    // devServer: {
    //   host: "localhost", // 启动服务器域名
    //   port: "3000", // 启动服务器端口号
    //   open: true, // 是否自动打开浏览器
    // },
    mode: "production",
    devtool: "source-map",
  };
```

3. 打包时会出现报错：

```
Error: Error with 'src\images\1.jpeg':
'"C:\Users\86176\Desktop\webpack\webpack_code\node_modules\jpegtran-
bin\vendor\jpegtran.exe"'
Error with 'src\images\3.gif': spawn
C:\Users\86176\Desktop\webpack\webpack_code\node_modules\optipng-
bin\vendor\optipng.exe ENOENT
```

我们需要安装两个文件到 node_modules 中才能解决, 文件可以从课件中找到：

- jpegtran.exe

需要复制到 `node_modules\jpegtran-bin\vendor` 下面

> “
> jpegtran 官网地址

- optipng.exe

需要复制到 `node_modules\optipng-bin\vendor` 下面

> “
>
> OptiPNG 官网地址

• 优化代码运行性能

– Code Split

为什么

打包代码时会将所有 js 文件打包到一个文件中，体积太大了。我们如果只要渲染首页，就应该只加载首页的 js 文件，其他文件不应该加载。

所以我们需要将打包生成的文件进行代码分割，生成多个 js 文件，渲染哪个页面就只加载某个 js 文件，这样加载的资源就少，速度就更快。

是什么

代码分割（Code Split）主要做了两件事：

1. 分割文件：将打包生成的文件进行分割，生成多个 js 文件。
2. 按需加载：需要哪个文件就加载哪个文件。

怎么用

代码分割实现方式有不同的方式，为了更加方便体现它们之间的差异，我们会分别创建新的文件来演示

1. 多入口

1. 文件目录

```
├── public
├── src
|   ├── app.js
|   └── main.js
├── package.json
└── webpack.config.js
```

2. 下载包

```
npm i webpack webpack-cli html-webpack-plugin -D
```

3. 新建文件

内容无关紧要，主要观察打包输出的结果

- app.js

```
console.log("hello app");
```

- main.js

```
console.log("hello main");
```

4. 配置

```
// webpack.config.js
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  // 单入口
  // entry: './src/main.js',
  // 多入口
  entry: {
    main: "./src/main.js",
    app: "./src/app.js",
  },
  output: {
    path: path.resolve(__dirname, "./dist"),
    // [name]是webpack命名规则，使用chunk的name作为输出的文件名。
    // 什么是chunk？打包的资源就是chunk，输出出去叫bundle。
    // chunk的name是啥呢？ 比如： entry中xxx: "./src/xxx.js", name就是xxx。注意是前面的xxx，和文件名无关。
    // 为什么需要这样命名呢？如果还是之前写法main.js，那么打包生成两个js文件都会叫做main.js会发生覆盖。(实际上会直接报错的)
    filename: "js/[name].js",
    clear: true,
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: "./public/index.html",
    }),
  ],
  mode: "production",
};
```

5. 运行指令

```
npx webpack
```

此时在 dist 目录我们能看到输出了两个 js 文件。

总结：配置了几个入口，至少输出几个 js 文件。

2. 提取重复代码

如果多入口文件中都引用了同一份代码，我们不希望这份代码被打包到两个文件中，导致代码重复，体积更大。

我们需要提取多入口的重复代码，只打包生成一个 js 文件，其他文件引用它就好。

1. 修改文件

- app.js

```
import { sum } from "./math";

console.log("hello app");
console.log(sum(1, 2, 3, 4));
```

- main.js

```
import { sum } from "./math";

console.log("hello main");
console.log(sum(1, 2, 3, 4, 5));
```

- math.js

```
export const sum = (...args) => {
  return args.reduce((p, c) => p + c, 0);
};
```

2. 修改配置文件

```
// webpack.config.js
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  // 单入口
  // entry: './src/main.js',
  // 多入口
  entry: {
    main: "./src/main.js",
    app: "./src/app.js",
  },
  output: {
```

```
14      path: path.resolve(__dirname, "./dist"),
15      // [name]是webpack命名规则，使用chunk的name作为输出的文件名。
16      // 什么是chunk？打包的资源就是chunk，输出出去叫bundle。
17      // chunk的name是啥呢？ 比如： entry中xxx: "./src/xxx.js", name就是xxx。注意是
    前面的xxx，和文件名无关。
18      // 为什么需要这样命名呢？如果还是之前写法main.js，那么打包生成两个js文件都会叫做
    main.js会发生覆盖。（实际上会直接报错的）
19      filename: "js/[name].js",
20      clean: true,
21    },
22    plugins: [
23      new HtmlWebpackPlugin({
24        template: "./public/index.html",
25      }),
26    ],
27    mode: "production",
28    optimization: {
29      // 代码分割配置
30      splitChunks: {
31        chunks: "all", // 对所有模块都进行分割
32        // 以下是默认值
33        // minSize: 20000, // 分割代码最小的大小，单位是byte
34        // minRemainingSize: 0, // 类似于minSize，最后确保提取的文件大小不能为0
35        // minChunks: 1, // 至少被引用的次数，满足条件才会代码分割
36        // maxAsyncRequests: 30, // 按需加载时并行加载的文件的最大数量
37        // maxInitialRequests: 30, // 入口js文件最大并行请求数量
38        // enforceSizeThreshold: 50000, // 超过50kb一定会单独打包（此时会忽略
    minRemainingSize、maxAsyncRequests、maxInitialRequests)
39        // cacheGroups: { // 组，哪些模块要打包到一个组
40        //   defaultVendors: { // 组名
41        //     test: /[\\/]node_modules[\\/]/, // 需要打包到一起的模块
42        //     priority: -10, // 权重（越大越高）
43        //     reuseExistingChunk: true, // 如果当前 chunk 包含已从主 bundle 中拆分
    出的模块，则它将被重用，而不是生成新的模块
44        //   },
45        //   default: { // 其他没有写的配置会使用上面的默认值
46        //     minChunks: 2, // 这里的minChunks权重更大
47        //     priority: -20,
48        //     reuseExistingChunk: true,
49        //   },
50        // },
51        // 修改配置
52        cacheGroups: {
53          // 组，哪些模块要打包到一个组
54          // defaultVendors: { // 组名
55          //   test: /[\\/]node_modules[\\/]/, // 需要打包到一起的模块
56          //   priority: -10, // 权重（越大越高）
57          //   reuseExistingChunk: true, // 如果当前 chunk 包含已从主 bundle 中拆分
    出的模块，则它将被重用，而不是生成新的模块
58          // },
59          default: {
```

```
            // 其他没有写的配置会使用上面的默认值
            minSize: 0, // 我们定义的文件体积太小了，所以要改打包的最小文件体积
            minChunks: 2,
            priority: -20,
            reuseExistingChunk: true,
          },
        },
      },
    },
  };
```

3. 运行指令

```
npx webpack
```

此时我们会发现生成 3 个 js 文件，其中有一个就是提取的公共模块。

3. 按需加载，动态导入

想要实现按需加载，动态导入模块。还需要额外配置：

1. 修改文件

- main.js

```
console.log("hello main");

document.getElementById("btn").onclick = function () {
  // 动态导入 --> 实现按需加载
  // 即使只被引用了一次，也会代码分割
  import("./math.js").then(({ sum }) => {
    alert(sum(1, 2, 3, 4, 5));
  });
};
```

- app.js

```
console.log("hello app");
```

- public/index.html

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Code Split</title>
  </head>
  <body>
    <h1>hello webpack</h1>
    <button id="btn">计算</button>
  </body>
</html>
```

2. 运行指令

```
npx webpack
```

我们可以发现，一旦通过 import 动态导入语法导入模块，模块就被代码分割，同时也能按需加载了。

4. 单入口

开发时我们可能是单页面应用（SPA），只有一个入口（单入口）。那么我们需要这样配置：

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
  // 单入口
  entry: "./src/main.js",
  // 多入口
  // entry: {
  //   main: "./src/main.js",
  //   app: "./src/app.js",
  // },
  output: {
    path: path.resolve(__dirname, "./dist"),
    // [name]是webpack命名规则，使用chunk的name作为输出的文件名。
    // 什么是chunk？打包的资源就是chunk，输出出去叫bundle。
    // chunk的name是啥呢？ 比如： entry中xxx: "./src/xxx.js", name就是xxx。注意是前面的xxx，和文件名无关。
    // 为什么需要这样命名呢？如果还是之前写法main.js，那么打包生成两个js文件都会叫做main.js会发生覆盖。（实际上会直接报错的）
    filename: "js/[name].js",
    clean: true,
  },
  plugins: [
    new HtmlWebpackPlugin({
```

```
      template: "./public/index.html",
    }),
  ],
  mode: "production",
  optimization: {
    // 代码分割配置
    splitChunks: {
      chunks: "all", // 对所有模块都进行分割
      // 以下是默认值
      // minSize: 20000, // 分割代码最小的大小
      // minRemainingSize: 0, // 类似于minSize，最后确保提取的文件大小不能为0
      // minChunks: 1, // 至少被引用的次数，满足条件才会代码分割
      // maxAsyncRequests: 30, // 按需加载时并行加载的文件的最大数量
      // maxInitialRequests: 30, // 入口js文件最大并行请求数量
      // enforceSizeThreshold: 50000, // 超过50kb一定会单独打包（此时会忽略minRemainingSize、maxAsyncRequests、maxInitialRequests)
      // cacheGroups: { // 组，哪些模块要打包到一个组
      //   defaultVendors: { // 组名
      //     test: /[\\/]node_modules[\\/]/, // 需要打包到一起的模块
      //     priority: -10, // 权重（越大越高）
      //     reuseExistingChunk: true, // 如果当前 chunk 包含已从主 bundle 中拆分出的模块，则它将被重用，而不是生成新的模块
      //   },
      //   default: { // 其他没有写的配置会使用上面的默认值
      //     minChunks: 2, // 这里的minChunks权重更大
      //     priority: -20,
      //     reuseExistingChunk: true,
      //   },
      // },
    },
};
```

5. 更新配置

最终我们会使用单入口+代码分割+动态导入方式来进行配置。更新之前的配置文件。

```
// webpack.prod.js
const os = require("os");
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");

// cpu核数
const threads = os.cpus().length;

// 获取处理样式的Loaders
```

```
const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    filename: "static/js/main.js", // 将 js 文件输出到 static/js 目录中
    clean: true,
  },
  module: {
    rules: [
      {
        oneOf: [
          {
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
            use: getStyleLoaders(),
          },
          {
            test: /\.less$/,
            use: getStyleLoaders("less-loader"),
          },
          {
            test: /\.s[ac]ss$/,
            use: getStyleLoaders("sass-loader"),
          },
          {
            test: /\.styl$/,
            use: getStyleLoaders("stylus-loader"),
          },
          {
            test: /\.(png|jpe?g|gif|svg)$/,
            type: "asset",
            parser: {
```

```
                dataUrlCondition: {
                  maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
                },
              },
              generator: {
                // 将图片文件输出到 static/imgs 目录中
                // 将图片文件命名 [hash:8][ext][query]
                // [hash:8]: hash值取8位
                // [ext]: 使用之前的文件扩展名
                // [query]: 添加之前的query参数
                filename: "static/imgs/[hash:8][ext][query]",
              },
            },
            {
              test: /\.(ttf|woff2?)$/,
              type: "asset/resource",
              generator: {
                filename: "static/media/[hash:8][ext][query]",
              },
            },
            {
              test: /\.js$/,
              // exclude: /node_modules/, // 排除node_modules代码不编译
              include: path.resolve(__dirname, "../src"), // 也可以用包含
              use: [
                {
                  loader: "thread-loader", // 开启多进程
                  options: {
                    workers: threads, // 数量
                  },
                },
                {
                  loader: "babel-loader",
                  options: {
                    cacheDirectory: true, // 开启babel编译缓存
                    cacheCompression: false, // 缓存文件不要压缩
                    plugins: ["@babel/plugin-transform-runtime"], // 减少代码体
积
                  },
                },
              ],
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules", // 默认值
```

```
        cache: true, // 开启缓存
        // 缓存目录
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
        threads, // 开启多进程
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      // 提取css成单独文件
      new MiniCssExtractPlugin({
        // 定义输出文件名和目录
        filename: "static/css/main.css",
      }),
      // css压缩
      // new CssMinimizerPlugin(),
    ],
    optimization: {
      minimizer: [
        // css压缩也可以写到optimization.minimizer里面，效果一样的
        new CssMinimizerPlugin(),
        // 当生产模式会默认开启TerserPlugin，但是我们需要进行其他配置，就要重新写了
        new TerserPlugin({
          parallel: threads, // 开启多进程
        }),
        // 压缩图片
        new ImageMinimizerPlugin({
          minimizer: {
            implementation: ImageMinimizerPlugin.imageminGenerate,
            options: {
              plugins: [
                ["gifsicle", { interlaced: true }],
                ["jpegtran", { progressive: true }],
                ["optipng", { optimizationLevel: 5 }],
                [
                  "svgo",
                  {
                    plugins: [
                      "preset-default",
                      "prefixIds",
                      {
                        name: "sortAttrs",
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
```

```
                  },
                ],
              ],
            },
          },
        }),
      ],
      // 代码分割配置
      splitChunks: {
        chunks: "all", // 对所有模块都进行分割
        // 其他内容用默认配置即可
      },
    },
    // devServer: {
    //   host: "localhost", // 启动服务器域名
    //   port: "3000", // 启动服务器端口号
    //   open: true, // 是否自动打开浏览器
    // },
    mode: "production",
    devtool: "source-map",
  };
```

6. 给动态导入文件取名称

1. 修改文件

- main.js

```
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/iconfont.css";
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";
import "./styl/index.styl";

const result2 = sum(1, 2, 3, 4);
console.log(result2);

// 以下代码生产模式下会删除
if (module.hot) {
  module.hot.accept("./js/sum.js", function (sum) {
    const result2 = sum(1, 2, 3, 4);
    console.log(result2);
  });
}

document.getElementById("btn").onClick = function () {
  // eslint会对动态导入语法报错，需要修改eslint配置文件
```

```
    // webpackChunkName: "math": 这是webpack动态导入模块命名的方式
    // "math"将来就会作为[name]的值显示。
    import(/* webpackChunkName: "math" */ "./js/math.js").then(({ count }) => {
      console.log(count(2, 1));
    });
  };
```

2. eslint 配置

- 下载包

```
npm i eslint-plugin-import -D
```

- 配置

```
// .eslintrc.js
module.exports = {
  // 继承 Eslint 规则
  extends: ["eslint:recommended"],
  env: {
    node: true, // 启用node中全局变量
    browser: true, // 启用浏览器中全局变量
  },
  plugins: ["import"], // 解决动态导入import语法报错问题 ⟶ 实际使用eslint-plugin-import的规则解决的
  parserOptions: {
    ecmaVersion: 6,
    sourceType: "module",
  },
  rules: {
    "no-var": 2, // 不能使用 var 定义变量
  },
};
```

1. 统一命名配置

```
const os = require("os");
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");

// cpu核数
```

```
const threads = os.cpus().length;

// 获取处理样式的Loaders
const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    filename: "static/js/[name].js", // 入口文件打包输出资源命名方式
    chunkFilename: "static/js/[name].chunk.js", // 动态导入输出资源命名方式
    assetModuleFilename: "static/media/[name].[hash][ext]", // 图片、字体等资源命名方式（注意用hash）
    clean: true,
  },
  module: {
    rules: [
      {
        oneOf: [
          {
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
            use: getStyleLoaders(),
          },
          {
            test: /\.less$/,
            use: getStyleLoaders("less-loader"),
          },
          {
            test: /\.s[ac]ss$/,
            use: getStyleLoaders("sass-loader"),
          },
          {
            test: /\.styl$/,
```

```
            use: getStyleLoaders("stylus-loader"),
          },
          {
            test: /\.(png|jpe?g|gif|svg)$/,
            type: "asset",
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
              },
            },
            // generator: {
            //   // 将图片文件输出到 static/imgs 目录中
            //   // 将图片文件命名 [hash:8][ext][query]
            //   // [hash:8]: hash值取8位
            //   // [ext]: 使用之前的文件扩展名
            //   // [query]: 添加之前的query参数
            //   filename: "static/imgs/[hash:8][ext][query]",
            // },
          },
          {
            test: /\.(ttf|woff2?)$/,
            type: "asset/resource",
            // generator: {
            //   filename: "static/media/[hash:8][ext][query]",
            // },
          },
          {
            test: /\.js$/,
            // exclude: /node_modules/, // 排除node_modules代码不编译
            include: path.resolve(__dirname, "../src"), // 也可以用包含
            use: [
              {
                loader: "thread-loader", // 开启多进程
                options: {
                  workers: threads, // 数量
                },
              },
              {
                loader: "babel-loader",
                options: {
                  cacheDirectory: true, // 开启babel编译缓存
                  cacheCompression: false, // 缓存文件不要压缩
                  plugins: ["@babel/plugin-transform-runtime"], // 减少代码体
积
                },
              },
            ],
          },
        ],
      },
    ],
```

```
  },
  plugins: [
    new ESLintWebpackPlugin({
      // 指定检查文件的根目录
      context: path.resolve(__dirname, "../src"),
      exclude: "node_modules", // 默认值
      cache: true, // 开启缓存
      // 缓存目录
      cacheLocation: path.resolve(
        __dirname,
        "../node_modules/.cache/.eslintcache"
      ),
      threads, // 开启多进程
    }),
    new HtmlWebpackPlugin({
      // 以 public/index.html 为模板创建文件
      // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
      template: path.resolve(__dirname, "../public/index.html"),
    }),
    // 提取css成单独文件
    new MiniCssExtractPlugin({
      // 定义输出文件名和目录
      filename: "static/css/[name].css",
      chunkFilename: "static/css/[name].chunk.css",
    }),
    // css压缩
    // new CssMinimizerPlugin(),
  ],
  optimization: {
    minimizer: [
      // css压缩也可以写到optimization.minimizer里面，效果一样的
      new CssMinimizerPlugin(),
      // 当生产模式会默认开启TerserPlugin，但是我们需要进行其他配置，就要重新写了
      new TerserPlugin({
        parallel: threads, // 开启多进程
      }),
      // 压缩图片
      new ImageMinimizerPlugin({
        minimizer: {
          implementation: ImageMinimizerPlugin.imageminGenerate,
          options: {
            plugins: [
              ["gifsicle", { interlaced: true }],
              ["jpegtran", { progressive: true }],
              ["optipng", { optimizationLevel: 5 }],
              [
                "svgo",
                {
                  plugins: [
                    "preset-default",
                    "prefixIds",
```

```
                      {
                        name: "sortAttrs",
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
      ],
      // 代码分割配置
      splitChunks: {
        chunks: "all", // 对所有模块都进行分割
        // 其他内容用默认配置即可
      },
    },
    // devServer: {
    //   host: "localhost", // 启动服务器域名
    //   port: "3000", // 启动服务器端口号
    //   open: true, // 是否自动打开浏览器
    // },
    mode: "production",
    devtool: "source-map",
  };
```

3. 运行指令

```
npx webpack
```

观察打包输出 js 文件名称。

Preload / Prefetch

为什么

我们前面已经做了代码分割，同时会使用 import 动态导入语法来进行代码按需加载（我们也叫懒加载，比如路由懒加载就是这样实现的）。

但是加载速度还不够好，比如：是用户点击按钮时才加载这个资源的，如果资源体积很大，那么用户会感觉到明显卡顿效果。

我们想在浏览器空闲时间，加载后续需要使用的资源。我们就需要用上 `Preload` 或 `Prefetch` 技术。

是什么

- `Preload`：告诉浏览器立即加载资源。
- `Prefetch`：告诉浏览器在空闲时才开始加载资源。

它们共同点：

- 都只会加载资源，并不执行。
- 都有缓存。

它们区别：

- `Preload` 加载优先级高，`Prefetch` 加载优先级低。
- `Preload` 只能加载当前页面需要使用的资源，`Prefetch` 可以加载当前页面资源，也可以加载下一个页面需要使用的资源。

总结：

- 当前页面优先级高的资源用 `Preload` 加载。
- 下一个页面需要使用的资源用 `Prefetch` 加载。

它们的问题：兼容性较差。

- 我们可以去 Can I Use 网站查询 API 的兼容性问题。
- `Preload` 相对于 `Prefetch` 兼容性好一点。

怎么用

1. 下载包

```
npm i @vue/preload-webpack-plugin -D
```

2. 配置 webpack.prod.js

```
const os = require("os");
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");
const PreloadWebpackPlugin = require("@vue/preload-webpack-plugin");

// cpu核数
const threads = os.cpus().length;

// 获取处理样式的Loaders
const getStyleLoaders = (preProcessor) => {
  return [
```

```
      MiniCssExtractPlugin.loader,
      "css-loader",
      {
        loader: "postcss-loader",
        options: {
          postcssOptions: {
            plugins: [
              "postcss-preset-env", // 能解决大多数样式兼容性问题
            ],
          },
        },
      },
      preProcessor,
    ].filter(Boolean);
  };

  module.exports = {
    entry: "./src/main.js",
    output: {
      path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
      filename: "static/js/[name].js", // 入口文件打包输出资源命名方式
      chunkFilename: "static/js/[name].chunk.js", // 动态导入输出资源命名方式
      assetModuleFilename: "static/media/[name].[hash][ext]", // 图片、字体等资源命名方式（注意用hash）
      clean: true,
    },
    module: {
      rules: [
        {
          oneOf: [
            {
              // 用来匹配 .css 结尾的文件
              test: /\.css$/,
              // use 数组里面 Loader 执行顺序是从右到左
              use: getStyleLoaders(),
            },
            {
              test: /\.less$/,
              use: getStyleLoaders("less-loader"),
            },
            {
              test: /\.s[ac]ss$/,
              use: getStyleLoaders("sass-loader"),
            },
            {
              test: /\.styl$/,
              use: getStyleLoaders("stylus-loader"),
            },
            {
              test: /\.(png|jpe?g|gif|svg)$/,
              type: "asset",
```

```
              parser: {
                dataUrlCondition: {
                  maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
                },
              },
              // generator: {
              //   // 将图片文件输出到 static/imgs 目录中
              //   // 将图片文件命名 [hash:8][ext][query]
              //   // [hash:8]: hash值取8位
              //   // [ext]: 使用之前的文件扩展名
              //   // [query]: 添加之前的query参数
              //   filename: "static/imgs/[hash:8][ext][query]",
              // },
            },
            {
              test: /\.(ttf|woff2?)$/,
              type: "asset/resource",
              // generator: {
              //   filename: "static/media/[hash:8][ext][query]",
              // },
            },
            {
              test: /\.js$/,
              // exclude: /node_modules/, // 排除node_modules代码不编译
              include: path.resolve(__dirname, "../src"), // 也可以用包含
              use: [
                {
                  loader: "thread-loader", // 开启多进程
                  options: {
                    workers: threads, // 数量
                  },
                },
                {
                  loader: "babel-loader",
                  options: {
                    cacheDirectory: true, // 开启babel编译缓存
                    cacheCompression: false, // 缓存文件不要压缩
                    plugins: ["@babel/plugin-transform-runtime"], // 减少代码体
积
                  },
                },
              ],
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
```

```
        exclude: "node_modules", // 默认值
        cache: true, // 开启缓存
        // 缓存目录
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
        threads, // 开启多进程
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      // 提取css成单独文件
      new MiniCssExtractPlugin({
        // 定义输出文件名和目录
        filename: "static/css/[name].css",
        chunkFilename: "static/css/[name].chunk.css",
      }),
      // css压缩
      // new CssMinimizerPlugin(),
      new PreloadWebpackPlugin({
        rel: "preload", // preload兼容性更好
        as: "script",
        // rel: 'prefetch' // prefetch兼容性更差
      }),
    ],
    optimization: {
      minimizer: [
        // css压缩也可以写到optimization.minimizer里面，效果一样的
        new CssMinimizerPlugin(),
        // 当生产模式会默认开启TerserPlugin，但是我们需要进行其他配置，就要重新写了
        new TerserPlugin({
          parallel: threads, // 开启多进程
        }),
        // 压缩图片
        new ImageMinimizerPlugin({
          minimizer: {
            implementation: ImageMinimizerPlugin.imageminGenerate,
            options: {
              plugins: [
                ["gifsicle", { interlaced: true }],
                ["jpegtran", { progressive: true }],
                ["optipng", { optimizationLevel: 5 }],
                [
                  "svgo",
                  {
                    plugins: [
                      "preset-default",
                      "prefixIds",
```

```
                      {
                        name: "sortAttrs",
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
      ],
      // 代码分割配置
      splitChunks: {
        chunks: "all", // 对所有模块都进行分割
        // 其他内容用默认配置即可
      },
    },
    // devServer: {
    //   host: "localhost", // 启动服务器域名
    //   port: "3000", // 启动服务器端口号
    //   open: true, // 是否自动打开浏览器
    // },
    mode: "production",
    devtool: "source-map",
  };
```

Network Cache

为什么

将来开发时我们对静态资源会使用缓存来优化，这样浏览器第二次请求资源就能读取缓存了，速度很快。

但是这样的话就会有一个问题，因为前后输出的文件名是一样的，都叫 main.js，一旦将来发布新版本，因为文件名没有变化导致浏览器会直接读取缓存，不会加载新资源，项目也就没法更新了。

所以我们从文件名入手，确保更新前后文件名不一样，这样就可以做缓存了。

是什么

它们都会生成一个唯一的 hash 值。

- fullhash（webpack4 是 hash）

每次修改任何一个文件，所有文件名的 hash 至都将改变。所以一旦修改了任何一个文件，整个项目的文件缓存都将失效。

- chunkhash

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 js 和 css 是同一个引入，会共享一个 hash 值。

- contenthash

根据文件内容生成 hash 值，只有文件内容变化了，hash 值才会变化。所有文件 hash 值是独享且不同的。

怎么用

```
const os = require("os");
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");
const PreloadWebpackPlugin = require("@vue/preload-webpack-plugin");

// cpu核数
const threads = os.cpus().length;

// 获取处理样式的Loaders
const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    // [contenthash:8]使用contenthash，取8位长度
    filename: "static/js/[name].[contenthash:8].js", // 入口文件打包输出资源命名方式
    chunkFilename: "static/js/[name].[contenthash:8].chunk.js", // 动态导入输出资源命名方式
```

```
    assetModuleFilename: "static/media/[name].[hash][ext]", // 图片、字体等资源命名方式（注意用hash）
    clean: true,
  },
  module: {
    rules: [
      {
        oneOf: [
          {
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
            use: getStyleLoaders(),
          },
          {
            test: /\.less$/,
            use: getStyleLoaders("less-loader"),
          },
          {
            test: /\.s[ac]ss$/,
            use: getStyleLoaders("sass-loader"),
          },
          {
            test: /\.styl$/,
            use: getStyleLoaders("stylus-loader"),
          },
          {
            test: /\.(png|jpe?g|gif|svg)$/,
            type: "asset",
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
              },
            },
            // generator: {
            //   // 将图片文件输出到 static/imgs 目录中
            //   // 将图片文件命名 [hash:8][ext][query]
            //   // [hash:8]: hash值取8位
            //   // [ext]: 使用之前的文件扩展名
            //   // [query]: 添加之前的query参数
            //   filename: "static/imgs/[hash:8][ext][query]",
            // },
          },
          {
            test: /\.(ttf|woff2?)$/,
            type: "asset/resource",
            // generator: {
            //   filename: "static/media/[hash:8][ext][query]",
            // },
          },
          {
```

```
              test: /\.js$/,
              // exclude: /node_modules/, // 排除node_modules代码不编译
              include: path.resolve(__dirname, "../src"), // 也可以用包含
              use: [
                {
                  loader: "thread-loader", // 开启多进程
                  options: {
                    workers: threads, // 数量
                  },
                },
                {
                  loader: "babel-loader",
                  options: {
                    cacheDirectory: true, // 开启babel编译缓存
                    cacheCompression: false, // 缓存文件不要压缩
                    plugins: ["@babel/plugin-transform-runtime"], // 减少代码体
积
                  },
                },
              ],
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules", // 默认值
        cache: true, // 开启缓存
        // 缓存目录
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
        threads, // 开启多进程
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      // 提取css成单独文件
      new MiniCssExtractPlugin({
        // 定义输出文件名和目录
        filename: "static/css/[name].[contenthash:8].css",
        chunkFilename: "static/css/[name].[contenthash:8].chunk.css",
      }),
      // css压缩
      // new CssMinimizerPlugin(),
```

```
      new PreloadWebpackPlugin({
        rel: "preload", // preload兼容性更好
        as: "script",
        // rel: 'prefetch' // prefetch兼容性更差
      }),
    ],
    optimization: {
      minimizer: [
        // css压缩也可以写到optimization.minimizer里面，效果一样的
        new CssMinimizerPlugin(),
        // 当生产模式会默认开启TerserPlugin，但是我们需要进行其他配置，就要重新写了
        new TerserPlugin({
          parallel: threads, // 开启多进程
        }),
        // 压缩图片
        new ImageMinimizerPlugin({
          minimizer: {
            implementation: ImageMinimizerPlugin.imageminGenerate,
            options: {
              plugins: [
                ["gifsicle", { interlaced: true }],
                ["jpegtran", { progressive: true }],
                ["optipng", { optimizationLevel: 5 }],
                [
                  "svgo",
                  {
                    plugins: [
                      "preset-default",
                      "prefixIds",
                      {
                        name: "sortAttrs",
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
      ],
      // 代码分割配置
      splitChunks: {
        chunks: "all", // 对所有模块都进行分割
        // 其他内容用默认配置即可
      },
    },
    // devServer: {
    //   host: "localhost", // 启动服务器域名
```

```
    //   port: "3000", // 启动服务器端口号
    //   open: true, // 是否自动打开浏览器
    // },
    mode: "production",
    devtool: "source-map",
};
```

- 问题：

当我们修改 math.js 文件再重新打包的时候，因为 contenthash 原因，math.js 文件 hash 值发生了变化（这是正常的）。

但是 main.js 文件的 hash 值也发生了变化，这会导致 main.js 的缓存失效。明明我们只修改 math.js，为什么 main.js 也会变身变化呢?

- 原因：
  - 更新前：math.xxx.js, main.js 引用的 math.xxx.js
  - 更新后：math.yyy.js, main.js 引用的 math.yyy.js, 文件名发生了变化，间接导致 main.js 也发生了变化
- 解决：

将 hash 值单独保管在一个 runtime 文件中。

我们最终输出三个文件：main、math、runtime。当 math 文件发送变化，变化的是 math 和 runtime 文件，main 不变。

runtime 文件只保存文件的 hash 值和它们与文件关系，整个文件体积就比较小，所以变化重新请求的代价也小。

```
const os = require("os");
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");
const PreloadWebpackPlugin = require("@vue/preload-webpack-plugin");

// cpu核数
const threads = os.cpus().length;

// 获取处理样式的Loaders
const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
    "css-loader",
    {
      loader: "postcss-loader",
```

```
        options: {
          postcssOptions: {
            plugins: [
              "postcss-preset-env", // 能解决大多数样式兼容性问题
            ],
          },
        },
      },
      preProcessor,
    ].filter(Boolean);
  };

  module.exports = {
    entry: "./src/main.js",
    output: {
      path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
      // [contenthash:8]使用contenthash，取8位长度
      filename: "static/js/[name].[contenthash:8].js", // 入口文件打包输出资源命名方式
      chunkFilename: "static/js/[name].[contenthash:8].chunk.js", // 动态导入输出资源命名方式
      assetModuleFilename: "static/media/[name].[hash][ext]", // 图片、字体等资源命名方式（注意用hash）
      clean: true,
    },
    module: {
      rules: [
        {
          oneOf: [
            {
              // 用来匹配 .css 结尾的文件
              test: /\.css$/,
              // use 数组里面 Loader 执行顺序是从右到左
              use: getStyleLoaders(),
            },
            {
              test: /\.less$/,
              use: getStyleLoaders("less-loader"),
            },
            {
              test: /\.s[ac]ss$/,
              use: getStyleLoaders("sass-loader"),
            },
            {
              test: /\.styl$/,
              use: getStyleLoaders("stylus-loader"),
            },
            {
              test: /\.(png|jpe?g|gif|svg)$/,
              type: "asset",
              parser: {
```

```
                dataUrlCondition: {
                  maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
                },
              },
              // generator: {
              //   // 将图片文件输出到 static/imgs 目录中
              //   // 将图片文件命名 [hash:8][ext][query]
              //   // [hash:8]: hash值取8位
              //   // [ext]: 使用之前的文件扩展名
              //   // [query]: 添加之前的query参数
              //   filename: "static/imgs/[hash:8][ext][query]",
              // },
            },
            {
              test: /\.(ttf|woff2?)$/,
              type: "asset/resource",
              // generator: {
              //   filename: "static/media/[hash:8][ext][query]",
              // },
            },
            {
              test: /\.js$/,
              // exclude: /node_modules/, // 排除node_modules代码不编译
              include: path.resolve(__dirname, "../src"), // 也可以用包含
              use: [
                {
                  loader: "thread-loader", // 开启多进程
                  options: {
                    workers: threads, // 数量
                  },
                },
                {
                  loader: "babel-loader",
                  options: {
                    cacheDirectory: true, // 开启babel编译缓存
                    cacheCompression: false, // 缓存文件不要压缩
                    plugins: ["@babel/plugin-transform-runtime"], // 减少代码体
积
                  },
                },
              ],
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules", // 默认值
```

```
        cache: true, // 开启缓存
        // 缓存目录
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
        threads, // 开启多进程
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      // 提取css成单独文件
      new MiniCssExtractPlugin({
        // 定义输出文件名和目录
        filename: "static/css/[name].[contenthash:8].css",
        chunkFilename: "static/css/[name].[contenthash:8].chunk.css",
      }),
      // css压缩
      // new CssMinimizerPlugin(),
      new PreloadWebpackPlugin({
        rel: "preload", // preload兼容性更好
        as: "script",
        // rel: 'prefetch' // prefetch兼容性更差
      }),
    ],
    optimization: {
      minimizer: [
        // css压缩也可以写到optimization.minimizer里面，效果一样的
        new CssMinimizerPlugin(),
        // 当生产模式会默认开启TerserPlugin，但是我们需要进行其他配置，就要重新写了
        new TerserPlugin({
          parallel: threads, // 开启多进程
        }),
        // 压缩图片
        new ImageMinimizerPlugin({
          minimizer: {
            implementation: ImageMinimizerPlugin.imageminGenerate,
            options: {
              plugins: [
                ["gifsicle", { interlaced: true }],
                ["jpegtran", { progressive: true }],
                ["optipng", { optimizationLevel: 5 }],
                [
                  "svgo",
                  {
                    plugins: [
                      "preset-default",
                      "prefixIds",
                      {
```

```
                        name: "sortAttrs",
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
      ],
      // 代码分割配置
      splitChunks: {
        chunks: "all", // 对所有模块都进行分割
        // 其他内容用默认配置即可
      },
      // 提取runtime文件
      runtimeChunk: {
        name: (entrypoint) => `runtime~${entrypoint.name}`, // runtime文件命名
规则
      },
    },
    // devServer: {
    //   host: "localhost", // 启动服务器域名
    //   port: "3000", // 启动服务器端口号
    //   open: true, // 是否自动打开浏览器
    // },
    mode: "production",
    devtool: "source-map",
  };
```

Core-js

为什么

过去我们使用 babel 对 js 代码进行了兼容性处理，其中使用@babel/preset-env 智能预设来处理兼容性问题。

它能将 ES6 的一些语法进行编译转换，比如箭头函数、点点点运算符等。但是如果是 async 函数、promise 对象、数组的一些方法（includes）等，它没办法处理。

所以此时我们 js 代码仍然存在兼容性问题，一旦遇到低版本浏览器会直接报错。所以我们想要将 js 兼容性问题彻底解决

是什么

`core-js` 是专门用来做 ES6 以及以上 API 的 `polyfill`。

`polyfill` 翻译过来叫做垫片/补丁。就是用社区上提供的一段代码，让我们在不兼容某些新特性的浏览器上，使用该新特性。

怎么用

1. 修改 main.js

```
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/iconfont.css";
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";
import "./styl/index.styl";

const result1 = count(2, 1);
console.log(result1);
const result2 = sum(1, 2, 3, 4);
console.log(result2);
// 添加promise代码
const promise = Promise.resolve();
promise.then(() => {
  console.log("hello promise");
});
```

此时 Eslint 会对 Promise 报错。

2. 修改配置文件

- 下载包

```
npm i @babel/eslint-parser -D
```

- .eslintrc.js

```
module.exports = {
  // 继承 Eslint 规则
  extends: ["eslint:recommended"],
  parser: "@babel/eslint-parser", // 支持最新的最终 ECMAScript 标准
  env: {
    node: true, // 启用node中全局变量
    browser: true, // 启用浏览器中全局变量
  },
  plugins: ["import"], // 解决动态导入import语法报错问题 ⟶ 实际使用eslint-plugin-import的规则解决的
  parserOptions: {
    ecmaVersion: 6, // es6
    sourceType: "module", // es module
  },
```

```
    rules: {
      "no-var": 2, // 不能使用 var 定义变量
    },
  };
```

3. 运行指令

```
npm run build
```

此时观察打包输出的 js 文件，我们发现 Promise 语法并没有编译转换，所以我们需要使用 `core-js` 来进行 `polyfill`。

4. 使用 `core-js`

- 下载包

```
npm i core-js
```

- 手动全部引入

```
import "core-js";
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/iconfont.css";
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";
import "./styl/index.styl";

const result1 = count(2, 1);
console.log(result1);
const result2 = sum(1, 2, 3, 4);
console.log(result2);
// 添加promise代码
const promise = Promise.resolve();
promise.then(() => {
  console.log("hello promise");
});
```

这样引入会将所有兼容性代码全部引入，体积太大了。我们只想引入 promise 的 `polyfill`。

- 手动按需引入

```
```

```
import "core-js/es/promise";
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/iconfont.css";
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";
import "./styl/index.styl";

const result1 = count(2, 1);
console.log(result1);
const result2 = sum(1, 2, 3, 4);
console.log(result2);
// 添加promise代码
const promise = Promise.resolve();
promise.then(() => {
  console.log("hello promise");
});
```

只引入打包 promise 的 `polyfill`，打包体积更小。但是将来如果还想使用其他语法，我需要手动引入库很麻烦。

- 自动按需引入
  - main.js

```
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/iconfont.css";
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";
import "./styl/index.styl";

const result1 = count(2, 1);
console.log(result1);
const result2 = sum(1, 2, 3, 4);
console.log(result2);
// 添加promise代码
const promise = Promise.resolve();
promise.then(() => {
  console.log("hello promise");
});
```

  - babel.config.js

```
module.exports = {
  // 智能预设：能够编译ES6语法
  presets: [
    [
      "@babel/preset-env",
      // 按需加载core-js的polyfill
      { useBuiltIns: "usage", corejs: { version: "3", proposals: true } }
    ],
  ],
};
```

此时就会自动根据我们代码中使用的语法，来按需加载相应的 `polyfill` 了。

PWA

为什么

开发 Web App 项目，项目一旦处于网络离线情况，就没法访问了。

我们希望给项目提供离线体验。

是什么

渐进式网络应用程序(progressive web application - PWA)：是一种可以提供类似于 native app(原生应用程序) 体验的 Web App 的技术。

其中最重要的是，在 **离线(offline)** 时应用程序能够继续运行功能。

内部通过 Service Workers 技术实现的。

怎么用

1. 下载包

```
npm i workbox-webpack-plugin -D
```

2. 修改配置文件

```
const os = require("os");
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");
const PreloadWebpackPlugin = require("@vue/preload-webpack-plugin");
const WorkboxPlugin = require("workbox-webpack-plugin");

```

```
// cpu核数
const threads = os.cpus().length;

// 获取处理样式的Loaders
const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"), // 生产模式需要输出
    // [contenthash:8]使用contenthash，取8位长度
    filename: "static/js/[name].[contenthash:8].js", // 入口文件打包输出资源命名方式
    chunkFilename: "static/js/[name].[contenthash:8].chunk.js", // 动态导入输出资源命名方式
    assetModuleFilename: "static/media/[name].[hash][ext]", // 图片、字体等资源命名方式（注意用hash）
    clean: true,
  },
  module: {
    rules: [
      {
        oneOf: [
          {
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
            use: getStyleLoaders(),
          },
          {
            test: /\.less$/,
            use: getStyleLoaders("less-loader"),
          },
          {
            test: /\.s[ac]ss$/,
```

```
            use: getStyleLoaders("sass-loader"),
          },
          {
            test: /\.styl$/,
            use: getStyleLoaders("stylus-loader"),
          },
          {
            test: /\.(png|jpe?g|gif|svg)$/,
            type: "asset",
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
              },
            },
            // generator: {
            //   // 将图片文件输出到 static/imgs 目录中
            //   // 将图片文件命名 [hash:8][ext][query]
            //   // [hash:8]: hash值取8位
            //   // [ext]: 使用之前的文件扩展名
            //   // [query]: 添加之前的query参数
            //   filename: "static/imgs/[hash:8][ext][query]",
            // },
          },
          {
            test: /\.(ttf|woff2?)$/,
            type: "asset/resource",
            // generator: {
            //   filename: "static/media/[hash:8][ext][query]",
            // },
          },
          {
            test: /\.js$/,
            // exclude: /node_modules/, // 排除node_modules代码不编译
            include: path.resolve(__dirname, "../src"), // 也可以用包含
            use: [
              {
                loader: "thread-loader", // 开启多进程
                options: {
                  workers: threads, // 数量
                },
              },
              {
                loader: "babel-loader",
                options: {
                  cacheDirectory: true, // 开启babel编译缓存
                  cacheCompression: false, // 缓存文件不要压缩
                  plugins: ["@babel/plugin-transform-runtime"], // 减少代码体
积
                },
              },
            ],
```

```
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        // 指定检查文件的根目录
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules", // 默认值
        cache: true, // 开启缓存
        // 缓存目录
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
        threads, // 开启多进程
      }),
      new HtmlWebpackPlugin({
        // 以 public/index.html 为模板创建文件
        // 新的html文件有两个特点：1. 内容和源文件一致 2. 自动引入打包生成的js等资源
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      // 提取css成单独文件
      new MiniCssExtractPlugin({
        // 定义输出文件名和目录
        filename: "static/css/[name].[contenthash:8].css",
        chunkFilename: "static/css/[name].[contenthash:8].chunk.css",
      }),
      // css压缩
      // new CssMinimizerPlugin(),
      new PreloadWebpackPlugin({
        rel: "preload", // preload兼容性更好
        as: "script",
        // rel: 'prefetch' // prefetch兼容性更差
      }),
      new WorkboxPlugin.GenerateSW({
        // 这些选项帮助快速启用 ServiceWorkers
        // 不允许遗留任何“旧的” ServiceWorkers
        clientsClaim: true,
        skipWaiting: true,
      }),
    ],
    optimization: {
      minimizer: [
        // css压缩也可以写到optimization.minimizer里面，效果一样的
        new CssMinimizerPlugin(),
        // 当生产模式会默认开启TerserPlugin，但是我们需要进行其他配置，就要重新写了
        new TerserPlugin({
          parallel: threads, // 开启多进程
        }),
```

```
        // 压缩图片
        new ImageMinimizerPlugin({
          minimizer: {
            implementation: ImageMinimizerPlugin.imageminGenerate,
            options: {
              plugins: [
                ["gifsicle", { interlaced: true }],
                ["jpegtran", { progressive: true }],
                ["optipng", { optimizationLevel: 5 }],
                [
                  "svgo",
                  {
                    plugins: [
                      "preset-default",
                      "prefixIds",
                      {
                        name: "sortAttrs",
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
      ],
      // 代码分割配置
      splitChunks: {
        chunks: "all", // 对所有模块都进行分割
        // 其他内容用默认配置即可
      },
    },
    // devServer: {
    //   host: "localhost", // 启动服务器域名
    //   port: "3000", // 启动服务器端口号
    //   open: true, // 是否自动打开浏览器
    // },
    mode: "production",
    devtool: "source-map",
  };
```

3. 修改 main.js

```
import count from "./js/count";
import sum from "./js/sum";
// 引入资源，Webpack才会对其打包
import "./css/iconfont.css";
```

```
import "./css/index.css";
import "./less/index.less";
import "./sass/index.sass";
import "./sass/index.scss";
import "./styl/index.styl";

const result1 = count(2, 1);
console.log(result1);
const result2 = sum(1, 2, 3, 4);
console.log(result2);
// 添加promise代码
const promise = Promise.resolve();
promise.then(() => {
  console.log("hello promise");
});

const arr = [1, 2, 3, 4, 5];
console.log(arr.includes(5));

if ("serviceWorker" in navigator) {
  window.addEventListener("load", () => {
    navigator.serviceWorker
      .register("/service-worker.js")
      .then((registration) => {
        console.log("SW registered: ", registration);
      })
      .catch((registrationError) => {
        console.log("SW registration failed: ", registrationError);
      });
  });
}
```

4. 运行指令

```
npm run build
```

此时如果直接通过 VSCode 访问打包后页面，在浏览器控制台会发现 `SW registration failed`。

因为我们打开的访问路径是：`http://127.0.0.1:5500/dist/index.html`。此时页面会去请求 `service-worker.js` 文件，请求路径是：`http://127.0.0.1:5500/service-worker.js`，这样找不到会 404。

实际 `service-worker.js` 文件路径是：`http://127.0.0.1:5500/dist/service-worker.js`。

5. 解决路径问题

- 下载包

```
npm i serve -g
```

serve 也是用来启动开发服务器来部署代码查看效果的。

- 运行指令

```
serve dist
```

此时通过 serve 启动的服务器我们 service-worker 就能注册成功了。

总结

我们从 4 个角度对 webpack 和代码进行了优化：

1. 提升开发体验

- 使用 `Source Map` 让开发或上线时代码报错能有更加准确的错误提示。

2. 提升 webpack 提升打包构建速度

- 使用 `HotModuleReplacement` 让开发时只重新编译打包更新变化了的代码，不变的代码使用缓存，从而使更新速度更快。
- 使用 `OneOf` 让资源文件一旦被某个 loader 处理了，就不会继续遍历了，打包速度更快。
- 使用 `Include/Exclude` 排除或只检测某些文件，处理的文件更少，速度更快。
- 使用 `Cache` 对 eslint 和 babel 处理的结果进行缓存，让第二次打包速度更快。
- 使用 `Thead` 多进程处理 eslint 和 babel 任务，速度更快。（需要注意的是，进程启动通信都有开销的，要在比较多代码处理时使用才有效果）

3. 减少代码体积

- 使用 `Tree Shaking` 剔除了没有使用的多余代码，让代码体积更小。
- 使用 `@babel/plugin-transform-runtime` 插件对 babel 进行处理，让辅助代码从中引入，而不是每个文件都生成辅助代码，从而体积更小。
- 使用 `Image Minimizer` 对项目中图片进行压缩，体积更小，请求速度更快。（需要注意的是，如果项目中图片都是在线链接，那么就不需要了。本地项目静态图片才需要进行压缩。）

4. 优化代码运行性能

- 使用 `Code Split` 对代码进行分割成多个 js 文件，从而使单个文件体积更小，并行加载 js 速度更快。并通过 import 动态导入语法进行按需加载，从而达到需要使用时才加载该资源，不用时不加载资源。
- 使用 `Preload / Prefetch` 对代码进行提前加载，等未来需要使用时就能直接使用，从而用户体验更好。
- 使用 `Network Cache` 能对输出资源文件进行更好的命名，将来好做缓存，从而用户体验更好。

- 使用 `Core-js` 对 js 进行兼容性处理，让我们代码能运行在低版本浏览器。
- 使用 `PWA` 能让代码离线也能访问，从而提升用户体验。

项目配置

React 脚手架

开发模式配置

```
// webpack.dev.js
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const ReactRefreshWebpackPlugin = require("@pmmmwh/react-refresh-webpack-plugin");
const CopyPlugin = require("copy-webpack-plugin");

const getStyleLoaders = (preProcessor) => {
  return [
    "style-loader",
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: undefined,
    filename: "static/js/[name].js",
    chunkFilename: "static/js/[name].chunk.js",
    assetModuleFilename: "static/js/[hash:10][ext][query]",
  },
  module: {
    rules: [
      {
        oneOf: [
          {
```

```
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
            use: getStyleLoaders(),
          },
          {
            test: /\.less$/,
            use: getStyleLoaders("less-loader"),
          },
          {
            test: /\.s[ac]ss$/,
            use: getStyleLoaders("sass-loader"),
          },
          {
            test: /\.styl$/,
            use: getStyleLoaders("stylus-loader"),
          },
          {
            test: /\.(png|jpe?g|gif|svg)$/,
            type: "asset",
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
              },
            },
          },
          {
            test: /\.(ttf|woff2?)$/,
            type: "asset/resource",
          },
          {
            test: /\.(jsx|js)$/,
            include: path.resolve(__dirname, "../src"),
            loader: "babel-loader",
            options: {
              cacheDirectory: true,
              cacheCompression: false,
              plugins: [
                // "@babel/plugin-transform-runtime", // presets中包含了
                "react-refresh/babel", // 开启js的HMR功能
              ],
            },
          },
        ],
      },
    ],
  },
  plugins: [
    new ESLintWebpackPlugin({
      context: path.resolve(__dirname, "../src"),
      exclude: "node_modules",
```

```
        cache: true,
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
      }),
      new HtmlWebpackPlugin({
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      new ReactRefreshWebpackPlugin(), // 解决js的HMR功能运行时全局变量的问题
      // 将public下面的资源复制到dist目录去 (除了index.html)
      new CopyPlugin({
        patterns: [
          {
            from: path.resolve(__dirname, "../public"),
            to: path.resolve(__dirname, "../dist"),
            toType: "dir",
            noErrorOnMissing: true, // 不生成错误
            globOptions: {
              // 忽略文件
              ignore: ["**/index.html"],
            },
            info: {
              // 跳过terser压缩js
              minimized: true,
            },
          },
        ],
      }),
    ],
    optimization: {
      splitChunks: {
        chunks: "all",
      },
      runtimeChunk: {
        name: (entrypoint) => `runtime~${entrypoint.name}`,
      },
    },
    resolve: {
      extensions: [".jsx", ".js", ".json"], // 自动补全文件扩展名，让jsx可以使用
    },
    devServer: {
      open: true,
      host: "localhost",
      port: 3000,
      hot: true,
      compress: true,
      historyApiFallback: true, // 解决react-router刷新404问题
    },
    mode: "development",
    devtool: "cheap-module-source-map",
```

```
};
```

生产模式配置

```
// webpack.prod.js
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const TerserWebpackPlugin = require("terser-webpack-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");
const CopyPlugin = require("copy-webpack-plugin");

const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "../dist"),
    filename: "static/js/[name].[contenthash:10].js",
    chunkFilename: "static/js/[name].[contenthash:10].chunk.js",
    assetModuleFilename: "static/js/[hash:10][ext][query]",
    clean: true,
  },
  module: {
    rules: [
      {
        oneOf: [
          {
            // 用来匹配 .css 结尾的文件
            test: /\.css$/,
            // use 数组里面 Loader 执行顺序是从右到左
            use: getStyleLoaders(),
```

```
            },
            {
              test: /\.less$/,
              use: getStyleLoaders("less-loader"),
            },
            {
              test: /\.s[ac]ss$/,
              use: getStyleLoaders("sass-loader"),
            },
            {
              test: /\.styl$/,
              use: getStyleLoaders("stylus-loader"),
            },
            {
              test: /\.(png|jpe?g|gif|svg)$/,
              type: "asset",
              parser: {
                dataUrlCondition: {
                  maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
                },
              },
            },
            {
              test: /\.(ttf|woff2?)$/,
              type: "asset/resource",
            },
            {
              test: /\.(jsx|js)$/,
              include: path.resolve(__dirname, "../src"),
              loader: "babel-loader",
              options: {
                cacheDirectory: true,
                cacheCompression: false,
                plugins: [
                  // "@babel/plugin-transform-runtime" // presets中包含了
                ],
              },
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules",
        cache: true,
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
```

```
    }),
    new HtmlWebpackPlugin({
      template: path.resolve(__dirname, "../public/index.html"),
    }),
    new MiniCssExtractPlugin({
      filename: "static/css/[name].[contenthash:10].css",
      chunkFilename: "static/css/[name].[contenthash:10].chunk.css",
    }),
    // 将public下面的资源复制到dist目录去（除了index.html）
    new CopyPlugin({
      patterns: [
        {
          from: path.resolve(__dirname, "../public"),
          to: path.resolve(__dirname, "../dist"),
          toType: "dir",
          noErrorOnMissing: true, // 不生成错误
          globOptions: {
            // 忽略文件
            ignore: ["**/index.html"],
          },
          info: {
            // 跳过terser压缩js
            minimized: true,
          },
        },
      ],
    }),
  ],
  optimization: {
    // 压缩的操作
    minimizer: [
      new CssMinimizerPlugin(),
      new TerserWebpackPlugin(),
      new ImageMinimizerPlugin({
        minimizer: {
          implementation: ImageMinimizerPlugin.imageminGenerate,
          options: {
            plugins: [
              ["gifsicle", { interlaced: true }],
              ["jpegtran", { progressive: true }],
              ["optipng", { optimizationLevel: 5 }],
              [
                "svgo",
                {
                  plugins: [
                    "preset-default",
                    "prefixIds",
                    {
                      name: "sortAttrs",
                      params: {
                        xmlnsOrder: "alphabetical",
```

```
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
      ],
      splitChunks: {
        chunks: "all",
      },
      runtimeChunk: {
        name: (entrypoint) => `runtime~${entrypoint.name}`,
      },
    },
    resolve: {
      extensions: [".jsx", ".js", ".json"],
    },
    mode: "production",
    devtool: "source-map",
  };
```

其他配置

- package.json

```
{
  "name": "react-cli",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "npm run dev",
    "dev": "cross-env NODE_ENV=development webpack serve --config ./config/webpack.dev.js",
    "build": "cross-env NODE_ENV=production webpack --config ./config/webpack.prod.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.17.10",
    "@pmmmwh/react-refresh-webpack-plugin": "^0.5.5",
    "babel-loader": "^8.2.5",
    "babel-preset-react-app": "^10.0.1",
    "copy-webpack-plugin": "^10.2.4",
    "cross-env": "^7.0.3",
```

```
    "css-loader": "^6.7.1",
    "css-minimizer-webpack-plugin": "^3.4.1",
    "eslint-config-react-app": "^7.0.1",
    "eslint-webpack-plugin": "^3.1.1",
    "html-webpack-plugin": "^5.5.0",
    "image-minimizer-webpack-plugin": "^3.2.3",
    "imagemin": "^8.0.1",
    "imagemin-gifsicle": "^7.0.0",
    "imagemin-jpegtran": "^7.0.0",
    "imagemin-optipng": "^8.0.0",
    "imagemin-svgo": "^10.0.1",
    "less-loader": "^10.2.0",
    "mini-css-extract-plugin": "^2.6.0",
    "postcss-loader": "^6.2.1",
    "postcss-preset-env": "^7.5.0",
    "react-refresh": "^0.13.0",
    "sass-loader": "^12.6.0",
    "style-loader": "^3.3.1",
    "stylus-loader": "^6.2.0",
    "webpack": "^5.72.0",
    "webpack-cli": "^4.9.2",
    "webpack-dev-server": "^4.9.0"
  },
  "dependencies": {
    "antd": "^4.20.2",
    "react": "^18.1.0",
    "react-dom": "^18.1.0",
    "react-router-dom": "^6.3.0"
  },
  "browserslist": ["last 2 version", "> 1%", "not dead"]
}
```

- .eslintrc.js

```
module.exports = {
  extends: ["react-app"], // 继承 react 官方规则
  parserOptions: {
    babelOptions: {
      presets: [
        // 解决页面报错问题
        ["babel-preset-react-app", false],
        "babel-preset-react-app/prod",
      ],
    },
  },
};
```

- babel.config.js

```
module.exports = {
  // 使用react官方规则
  presets: ["react-app"],
};
```

合并开发和生产配置

- webpack.config.js

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserWebpackPlugin = require("terser-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");
const ReactRefreshWebpackPlugin = require("@pmmmwh/react-refresh-webpack-plugin");

// 需要通过 cross-env 定义环境变量
const isProduction = process.env.NODE_ENV === "production";

const getStyleLoaders = (preProcessor) => {
  return [
    isProduction ? MiniCssExtractPlugin.loader : "style-loader",
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: [
            "postcss-preset-env", // 能解决大多数样式兼容性问题
          ],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: isProduction ? path.resolve(__dirname, "../dist") : undefined,
    filename: isProduction
      ? "static/js/[name].[contenthash:10].js"
      : "static/js/[name].js",
    chunkFilename: isProduction
      ? "static/js/[name].[contenthash:10].chunk.js"
```

```
        : "static/js/[name].chunk.js",
      assetModuleFilename: "static/js/[hash:10][ext][query]",
      clean: true,
    },
    module: {
      rules: [
        {
          oneOf: [
            {
              // 用来匹配 .css 结尾的文件
              test: /\.css$/,
              // use 数组里面 Loader 执行顺序是从右到左
              use: getStyleLoaders(),
            },
            {
              test: /\.less$/,
              use: getStyleLoaders("less-loader"),
            },
            {
              test: /\.s[ac]ss$/,
              use: getStyleLoaders("sass-loader"),
            },
            {
              test: /\.styl$/,
              use: getStyleLoaders("stylus-loader"),
            },
            {
              test: /\.(png|jpe?g|gif|svg)$/,
              type: "asset",
              parser: {
                dataUrlCondition: {
                  maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
                },
              },
            },
            {
              test: /\.(ttf|woff2?)$/,
              type: "asset/resource",
            },
            {
              test: /\.(jsx|js)$/,
              include: path.resolve(__dirname, "../src"),
              loader: "babel-loader",
              options: {
                cacheDirectory: true, // 开启babel编译缓存
                cacheCompression: false, // 缓存文件不要压缩
                plugins: [
                  // "@babel/plugin-transform-runtime",  // presets中包含了
                  !isProduction && "react-refresh/babel",
                ].filter(Boolean),
              },
```

```
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        extensions: [".js", ".jsx"],
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules",
        cache: true,
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
      }),
      new HtmlWebpackPlugin({
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      isProduction &&
        new MiniCssExtractPlugin({
          filename: "static/css/[name].[contenthash:10].css",
          chunkFilename: "static/css/[name].[contenthash:10].chunk.css",
        }),
      !isProduction && new ReactRefreshWebpackPlugin(),
    ].filter(Boolean),
    optimization: {
      minimize: isProduction,
      // 压缩的操作
      minimizer: [
        // 压缩css
        new CssMinimizerPlugin(),
        // 压缩js
        new TerserWebpackPlugin(),
        // 压缩图片
        new ImageMinimizerPlugin({
          minimizer: {
            implementation: ImageMinimizerPlugin.imageminGenerate,
            options: {
              plugins: [
                ["gifsicle", { interlaced: true }],
                ["jpegtran", { progressive: true }],
                ["optipng", { optimizationLevel: 5 }],
                [
                  "svgo",
                  {
                    plugins: [
                      "preset-default",
                      "prefixIds",
                      {
                        name: "sortAttrs",
```

```
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
      ],
      // 代码分割配置
      splitChunks: {
        chunks: "all",
        // 其他都用默认值
      },
      runtimeChunk: {
        name: (entrypoint) => `runtime~${entrypoint.name}`,
      },
    },
    resolve: {
      extensions: [".jsx", ".js", ".json"],
    },
    devServer: {
      open: true,
      host: "localhost",
      port: 3000,
      hot: true,
      compress: true,
      historyApiFallback: true,
    },
    mode: isProduction ? "production" : "development",
    devtool: isProduction ? "source-map" : "cheap-module-source-map",
  };
```

- 修改运行指令 package.json

```
{
  "name": "react-cli",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "npm run dev",
    "dev": "cross-env NODE_ENV=development webpack serve --config ./config/webpack.config.js",
    "build": "cross-env NODE_ENV=production webpack --config ./config/webpack.config.js"
  },
```

```
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.17.10",
    "@pmmmwh/react-refresh-webpack-plugin": "^0.5.5",
    "babel-loader": "^8.2.5",
    "babel-preset-react-app": "^10.0.1",
    "cross-env": "^7.0.3",
    "css-loader": "^6.7.1",
    "css-minimizer-webpack-plugin": "^3.4.1",
    "eslint-config-react-app": "^7.0.1",
    "eslint-webpack-plugin": "^3.1.1",
    "html-webpack-plugin": "^5.5.0",
    "image-minimizer-webpack-plugin": "^3.2.3",
    "imagemin": "^8.0.1",
    "imagemin-gifsicle": "^7.0.0",
    "imagemin-jpegtran": "^7.0.0",
    "imagemin-optipng": "^8.0.0",
    "imagemin-svgo": "^10.0.1",
    "less-loader": "^10.2.0",
    "mini-css-extract-plugin": "^2.6.0",
    "react-refresh": "^0.13.0",
    "sass-loader": "^12.6.0",
    "style-loader": "^3.3.1",
    "stylus-loader": "^6.2.0",
    "webpack": "^5.72.0",
    "webpack-cli": "^4.9.2",
    "webpack-dev-server": "^4.9.0"
  },
  "dependencies": {
    "react": "^18.1.0",
    "react-dom": "^18.1.0",
    "react-router-dom": "^6.3.0"
  },
  "browserslist": ["last 2 version", "> 1%", "not dead"]
}
```

优化配置

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserWebpackPlugin = require("terser-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");
const ReactRefreshWebpackPlugin = require("@pmmmwh/react-refresh-webpack-plugin");
const CopyPlugin = require("copy-webpack-plugin");
```

```
10
11  const isProduction = process.env.NODE_ENV === "production";
12
13  const getStyleLoaders = (preProcessor) => {
14    return [
15      isProduction ? MiniCssExtractPlugin.loader : "style-loader",
16      "css-loader",
17      {
18        loader: "postcss-loader",
19        options: {
20          postcssOptions: {
21            plugins: [
22              "postcss-preset-env",
23            ],
24          },
25        },
26      },
27      preProcessor && {
28        loader: preProcessor,
29        options:
30          preProcessor === "less-loader"
31            ? {
32                // antd的自定义主题
33                lessOptions: {
34                  modifyVars: {
35                    // 其他主题色: https://ant.design/docs/react/customize-
    theme-cn
36                    "@primary-color": "#1DA57A", // 全局主色
37                  },
38                  javascriptEnabled: true,
39                },
40              }
41            : {},
42      },
43    ].filter(Boolean);
44  };
45
46  module.exports = {
47    entry: "./src/main.js",
48    output: {
49      path: isProduction ? path.resolve(__dirname, "../dist") : undefined,
50      filename: isProduction
51        ? "static/js/[name].[contenthash:10].js"
52        : "static/js/[name].js",
53      chunkFilename: isProduction
54        ? "static/js/[name].[contenthash:10].chunk.js"
55        : "static/js/[name].chunk.js",
56      assetModuleFilename: "static/js/[hash:10][ext][query]",
57      clean: true,
58    },
59    module: {
```

```
      rules: [
        {
          oneOf: [
            {
              test: /\.css$/,
              use: getStyleLoaders(),
            },
            {
              test: /\.less$/,
              use: getStyleLoaders("less-loader"),
            },
            {
              test: /\.s[ac]ss$/,
              use: getStyleLoaders("sass-loader"),
            },
            {
              test: /\.styl$/,
              use: getStyleLoaders("stylus-loader"),
            },
            {
              test: /\.(png|jpe?g|gif|svg)$/,
              type: "asset",
              parser: {
                dataUrlCondition: {
                  maxSize: 10 * 1024,
                },
              },
            },
            {
              test: /\.(ttf|woff2?)$/,
              type: "asset/resource",
            },
            {
              test: /\.(jsx|js)$/,
              include: path.resolve(__dirname, "../src"),
              loader: "babel-loader",
              options: {
                cacheDirectory: true,
                cacheCompression: false,
                plugins: [
                  // "@babel/plugin-transform-runtime",  // presets中包含了
                  !isProduction && "react-refresh/babel",
                ].filter(Boolean),
              },
            },
          ],
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
```

```
        extensions: [".js", ".jsx"],
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules",
        cache: true,
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
      }),
      new HtmlWebpackPlugin({
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      isProduction &&
        new MiniCssExtractPlugin({
          filename: "static/css/[name].[contenthash:10].css",
          chunkFilename: "static/css/[name].[contenthash:10].chunk.css",
        }),
      !isProduction && new ReactRefreshWebpackPlugin(),
      // 将public下面的资源复制到dist目录去 (除了index.html)
      new CopyPlugin({
        patterns: [
          {
            from: path.resolve(__dirname, "../public"),
            to: path.resolve(__dirname, "../dist"),
            toType: "dir",
            noErrorOnMissing: true, // 不生成错误
            globOptions: {
              // 忽略文件
              ignore: ["**/index.html"],
            },
            info: {
              // 跳过terser压缩js
              minimized: true,
            },
          },
        ],
      }),
    ].filter(Boolean),
    optimization: {
      minimize: isProduction,
      // 压缩的操作
      minimizer: [
        // 压缩css
        new CssMinimizerPlugin(),
        // 压缩js
        new TerserWebpackPlugin(),
        // 压缩图片
        new ImageMinimizerPlugin({
          minimizer: {
            implementation: ImageMinimizerPlugin.imageminGenerate,
            options: {
```

```
              plugins: [
                ["gifsicle", { interlaced: true }],
                ["jpegtran", { progressive: true }],
                ["optipng", { optimizationLevel: 5 }],
                [
                  "svgo",
                  {
                    plugins: [
                      "preset-default",
                      "prefixIds",
                      {
                        name: "sortAttrs",
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
      ],
      // 代码分割配置
      splitChunks: {
        chunks: "all",
        cacheGroups: {
          // layouts通常是admin项目的主体布局组件，所有路由组件都要使用的
          // 可以单独打包，从而复用
          // 如果项目中没有，请删除
          layouts: {
            name: "layouts",
            test: path.resolve(__dirname, "../src/layouts"),
            priority: 40,
          },
          // 如果项目中使用antd，此时将所有node_modules打包在一起，那么打包输出文件会比较大。
          // 所以我们将node_modules中比较大的模块单独打包，从而并行加载速度更好
          // 如果项目中没有，请删除
          antd: {
            name: "chunk-antd",
            test: /[\\/]node_modules[\\/]antd(.*)/,
            priority: 30,
          },
          // 将react相关的库单独打包，减少node_modules的chunk体积。
          react: {
            name: "react",
            test: /[\\/]node_modules[\\/]react(.*)?[\\/]/,
            chunks: "initial",
            priority: 20,
```

```
          },
          libs: {
            name: "chunk-libs",
            test: /[\\/]node_modules[\\/]/,
            priority: 10, // 权重最低，优先考虑前面内容
            chunks: "initial",
          },
        },
      },
      runtimeChunk: {
        name: (entrypoint) => `runtime~${entrypoint.name}`,
      },
    },
    resolve: {
      extensions: [".jsx", ".js", ".json"],
    },
    devServer: {
      open: true,
      host: "localhost",
      port: 3000,
      hot: true,
      compress: true,
      historyApiFallback: true,
    },
    mode: isProduction ? "production" : "development",
    devtool: isProduction ? "source-map" : "cheap-module-source-map",
    performance: false, // 关闭性能分析，提示速度
};
```

- **Vue 脚手架**

  - **开发模式配置**

```
// webpack.dev.js
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { VueLoaderPlugin } = require("vue-loader");
const { DefinePlugin } = require("webpack");
const CopyPlugin = require("copy-webpack-plugin");

const getStyleLoaders = (preProcessor) => {
  return [
    "vue-style-loader",
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
```

```
            plugins: [
              "postcss-preset-env", // 能解决大多数样式兼容性问题
            ],
          },
        },
      },
      preProcessor,
    ].filter(Boolean);
  };

  module.exports = {
    entry: "./src/main.js",
    output: {
      path: undefined,
      filename: "static/js/[name].js",
      chunkFilename: "static/js/[name].chunk.js",
      assetModuleFilename: "static/js/[hash:10][ext][query]",
    },
    module: {
      rules: [
        {
          // 用来匹配 .css 结尾的文件
          test: /\.css$/,
          // use 数组里面 Loader 执行顺序是从右到左
          use: getStyleLoaders(),
        },
        {
          test: /\.less$/,
          use: getStyleLoaders("less-loader"),
        },
        {
          test: /\.s[ac]ss$/,
          use: getStyleLoaders("sass-loader"),
        },
        {
          test: /\.styl$/,
          use: getStyleLoaders("stylus-loader"),
        },
        {
          test: /\.(png|jpe?g|gif|svg)$/,
          type: "asset",
          parser: {
            dataUrlCondition: {
              maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
            },
          },
        },
        {
          test: /\.(ttf|woff2?)$/,
          type: "asset/resource",
        },
```

```
        {
          test: /\.(jsx|js)$/,
          include: path.resolve(__dirname, "../src"),
          loader: "babel-loader",
          options: {
            cacheDirectory: true,
            cacheCompression: false,
            plugins: [
              // "@babel/plugin-transform-runtime" // presets中包含了
            ],
          },
        },
        // vue-loader不支持oneOf
        {
          test: /\.vue$/,
          loader: "vue-loader", // 内部会给vue文件注入HMR功能代码
          options: {
            // 开启缓存
            cacheDirectory: path.resolve(
              __dirname,
              "node_modules/.cache/vue-loader"
            ),
          },
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules",
        cache: true,
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
      }),
      new HtmlWebpackPlugin({
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      new CopyPlugin({
        patterns: [
          {
            from: path.resolve(__dirname, "../public"),
            to: path.resolve(__dirname, "../dist"),
            toType: "dir",
            noErrorOnMissing: true,
            globOptions: {
              ignore: ["**/index.html"],
            },
            info: {
              minimized: true,
```

```
            },
          },
        ],
      }),
      new VueLoaderPlugin(),
      // 解决页面警告
      new DefinePlugin({
        __VUE_OPTIONS_API__: "true",
        __VUE_PROD_DEVTOOLS__: "false",
      }),
    ],
    optimization: {
      splitChunks: {
        chunks: "all",
      },
      runtimeChunk: {
        name: (entrypoint) => `runtime~${entrypoint.name}`,
      },
    },
    resolve: {
      extensions: [".vue", ".js", ".json"], // 自动补全文件扩展名，让vue可以使用
    },
    devServer: {
      open: true,
      host: "localhost",
      port: 3000,
      hot: true,
      compress: true,
      historyApiFallback: true, // 解决vue-router刷新404问题
    },
    mode: "development",
    devtool: "cheap-module-source-map",
  };
```

生产模式配置

```
// webpack.prod.js
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserWebpackPlugin = require("terser-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");
const { VueLoaderPlugin } = require("vue-loader");
const { DefinePlugin } = require("webpack");

const getStyleLoaders = (preProcessor) => {
  return [
    MiniCssExtractPlugin.loader,
```

```
      "css-loader",
      {
        loader: "postcss-loader",
        options: {
          postcssOptions: {
            plugins: [
              "postcss-preset-env", // 能解决大多数样式兼容性问题
            ],
          },
        },
      },
      preProcessor,
    ].filter(Boolean);
  };

  module.exports = {
    entry: "./src/main.js",
    output: {
      path: undefined,
      filename: "static/js/[name].[contenthash:10].js",
      chunkFilename: "static/js/[name].[contenthash:10].chunk.js",
      assetModuleFilename: "static/js/[hash:10][ext][query]",
      clean: true,
    },
    module: {
      rules: [
        {
          // 用来匹配 .css 结尾的文件
          test: /\.css$/,
          // use 数组里面 Loader 执行顺序是从右到左
          use: getStyleLoaders(),
        },
        {
          test: /\.less$/,
          use: getStyleLoaders("less-loader"),
        },
        {
          test: /\.s[ac]ss$/,
          use: getStyleLoaders("sass-loader"),
        },
        {
          test: /\.styl$/,
          use: getStyleLoaders("stylus-loader"),
        },
        {
          test: /\.(png|jpe?g|gif|svg)$/,
          type: "asset",
          parser: {
            dataUrlCondition: {
              maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
            },
```

```
          },
        },
        {
          test: /\.(ttf|woff2?)$/,
          type: "asset/resource",
        },
        {
          test: /\.(jsx|js)$/,
          include: path.resolve(__dirname, "../src"),
          loader: "babel-loader",
          options: {
            cacheDirectory: true,
            cacheCompression: false,
            plugins: [
              // "@babel/plugin-transform-runtime" // presets中包含了
            ],
          },
        },
        // vue-loader不支持oneOf
        {
          test: /\.vue$/,
          loader: "vue-loader", // 内部会给vue文件注入HMR功能代码
          options: {
            // 开启缓存
            cacheDirectory: path.resolve(
              __dirname,
              "node_modules/.cache/vue-loader"
            ),
          },
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules",
        cache: true,
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
      }),
      new HtmlWebpackPlugin({
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      new CopyPlugin({
        patterns: [
          {
            from: path.resolve(__dirname, "../public"),
            to: path.resolve(__dirname, "../dist"),
            toType: "dir",
```

```
            noErrorOnMissing: true,
            globOptions: {
              ignore: ["**/index.html"],
            },
            info: {
              minimized: true,
            },
          },
        ],
      }),
      new MiniCssExtractPlugin({
        filename: "static/css/[name].[contenthash:10].css",
        chunkFilename: "static/css/[name].[contenthash:10].chunk.css",
      }),
      new VueLoaderPlugin(),
      new DefinePlugin({
        __VUE_OPTIONS_API__: "true",
        __VUE_PROD_DEVTOOLS__: "false",
      }),
    ],
    optimization: {
      // 压缩的操作
      minimizer: [
        new CssMinimizerPlugin(),
        new TerserWebpackPlugin(),
        new ImageMinimizerPlugin({
          minimizer: {
            implementation: ImageMinimizerPlugin.imageminGenerate,
            options: {
              plugins: [
                ["gifsicle", { interlaced: true }],
                ["jpegtran", { progressive: true }],
                ["optipng", { optimizationLevel: 5 }],
                [
                  "svgo",
                  {
                    plugins: [
                      "preset-default",
                      "prefixIds",
                      {
                        name: "sortAttrs",
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
```

```
        }),
      ],
      splitChunks: {
        chunks: "all",
      },
      runtimeChunk: {
        name: (entrypoint) => `runtime~${entrypoint.name}`,
      },
    },
    resolve: {
      extensions: [".vue", ".js", ".json"],
    },
    mode: "production",
    devtool: "source-map",
  };
```

其他配置

- package.json

```
{
  "name": "vue-cli",
  "version": "1.0.0",
  "description": "",
  "main": "main.js",
  "scripts": {
    "start": "npm run dev",
    "dev": "cross-env NODE_ENV=development webpack serve --config
./config/webpack.dev.js",
    "build": "cross-env NODE_ENV=production webpack --config
./config/webpack.prod.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.17.10",
    "@babel/eslint-parser": "^7.17.0",
    "@vue/cli-plugin-babel": "^5.0.4",
    "babel-loader": "^8.2.5",
    "copy-webpack-plugin": "^10.2.4",
    "cross-env": "^7.0.3",
    "css-loader": "^6.7.1",
    "css-minimizer-webpack-plugin": "^3.4.1",
    "eslint-plugin-vue": "^8.7.1",
    "eslint-webpack-plugin": "^3.1.1",
    "html-webpack-plugin": "^5.5.0",
    "image-minimizer-webpack-plugin": "^3.2.3",
    "imagemin": "^8.0.1",
    "imagemin-gifsicle": "^7.0.0",
```

```
    "imagemin-jpegtran": "^7.0.0",
    "imagemin-optipng": "^8.0.0",
    "imagemin-svgo": "^10.0.1",
    "less-loader": "^10.2.0",
    "mini-css-extract-plugin": "^2.6.0",
    "postcss-preset-env": "^7.5.0",
    "sass-loader": "^12.6.0",
    "stylus-loader": "^6.2.0",
    "vue-loader": "^17.0.0",
    "vue-style-loader": "^4.1.3",
    "vue-template-compiler": "^2.6.14",
    "webpack": "^5.72.0",
    "webpack-cli": "^4.9.2",
    "webpack-dev-server": "^4.9.0"
  },
  "dependencies": {
    "vue": "^3.2.33",
    "vue-router": "^4.0.15"
  },
  "browserslist": ["last 2 version", "> 1%", "not dead"]
}
```

- .eslintrc.js

```
module.exports = {
  root: true,
  env: {
    node: true,
  },
  extends: ["plugin:vue/vue3-essential", "eslint:recommended"],
  parserOptions: {
    parser: "@babel/eslint-parser",
  },
};
```

- babel.config.js

```
module.exports = {
  presets: ["@vue/cli-plugin-babel/preset"],
};
```

合并开发和生产配置

```
// webpack.config.js
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
```

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
const TerserWebpackPlugin = require("terser-webpack-plugin");
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");
const { VueLoaderPlugin } = require("vue-loader");
const { DefinePlugin } = require("webpack");
const CopyPlugin = require("copy-webpack-plugin");

// 需要通过 cross-env 定义环境变量
const isProduction = process.env.NODE_ENV === "production";

const getStyleLoaders = (preProcessor) => {
  return [
    isProduction ? MiniCssExtractPlugin.loader : "vue-style-loader",
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: ["postcss-preset-env"],
        },
      },
    },
    preProcessor,
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: isProduction ? path.resolve(__dirname, "../dist") : undefined,
    filename: isProduction
      ? "static/js/[name].[contenthash:10].js"
      : "static/js/[name].js",
    chunkFilename: isProduction
      ? "static/js/[name].[contenthash:10].chunk.js"
      : "static/js/[name].chunk.js",
    assetModuleFilename: "static/js/[hash:10][ext][query]",
    clean: true,
  },
  module: {
    rules: [
      {
        // 用来匹配 .css 结尾的文件
        test: /\.css$/,
        // use 数组里面 Loader 执行顺序是从右到左
        use: getStyleLoaders(),
      },
      {
        test: /\.less$/,
```

```
          use: getStyleLoaders("less-loader"),
        },
        {
          test: /\.s[ac]ss$/,
          use: getStyleLoaders("sass-loader"),
        },
        {
          test: /\.styl$/,
          use: getStyleLoaders("stylus-loader"),
        },
        {
          test: /\.(png|jpe?g|gif|svg)$/,
          type: "asset",
          parser: {
            dataUrlCondition: {
              maxSize: 10 * 1024, // 小于10kb的图片会被base64处理
            },
          },
        },
        {
          test: /\.(ttf|woff2?)$/,
          type: "asset/resource",
        },
        {
          test: /\.(jsx|js)$/,
          include: path.resolve(__dirname, "../src"),
          loader: "babel-loader",
          options: {
            cacheDirectory: true,
            cacheCompression: false,
            plugins: [
              // "@babel/plugin-transform-runtime" // presets中包含了
            ],
          },
        },
        // vue-loader不支持oneOf
        {
          test: /\.vue$/,
          loader: "vue-loader", // 内部会给vue文件注入HMR功能代码
          options: {
            // 开启缓存
            cacheDirectory: path.resolve(
              __dirname,
              "node_modules/.cache/vue-loader"
            ),
          },
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
```

```
106         context: path.resolve(__dirname, "../src"),
107         exclude: "node_modules",
108         cache: true,
109         cacheLocation: path.resolve(
110           __dirname,
111           "../node_modules/.cache/.eslintcache"
112         ),
113       }),
114       new HtmlWebpackPlugin({
115         template: path.resolve(__dirname, "../public/index.html"),
116       }),
117       new CopyPlugin({
118         patterns: [
119           {
120             from: path.resolve(__dirname, "../public"),
121             to: path.resolve(__dirname, "../dist"),
122             toType: "dir",
123             noErrorOnMissing: true,
124             globOptions: {
125               ignore: ["**/index.html"],
126             },
127             info: {
128               minimized: true,
129             },
130           },
131         ],
132       }),
133       isProduction &&
134         new MiniCssExtractPlugin({
135           filename: "static/css/[name].[contenthash:10].css",
136           chunkFilename: "static/css/[name].[contenthash:10].chunk.css",
137         }),
138       new VueLoaderPlugin(),
139       new DefinePlugin({
140         __VUE_OPTIONS_API__: "true",
141         __VUE_PROD_DEVTOOLS__: "false",
142       }),
143     ].filter(Boolean),
144     optimization: {
145       minimize: isProduction,
146       // 压缩的操作
147       minimizer: [
148         new CssMinimizerPlugin(),
149         new TerserWebpackPlugin(),
150         new ImageMinimizerPlugin({
151           minimizer: {
152             implementation: ImageMinimizerPlugin.imageminGenerate,
153             options: {
154               plugins: [
155                 ["gifsicle", { interlaced: true }],
156                 ["jpegtran", { progressive: true }],
```

```
                ["optipng", { optimizationLevel: 5 }],
                [
                  "svgo",
                  {
                    plugins: [
                      "preset-default",
                      "prefixIds",
                      {
                        name: "sortAttrs",
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
      ],
      splitChunks: {
        chunks: "all",
      },
      runtimeChunk: {
        name: (entrypoint) => `runtime~${entrypoint.name}`,
      },
    },
    resolve: {
      extensions: [".vue", ".js", ".json"],
    },
    devServer: {
      open: true,
      host: "localhost",
      port: 3000,
      hot: true,
      compress: true,
      historyApiFallback: true, // 解决vue-router刷新404问题
    },
    mode: isProduction ? "production" : "development",
    devtool: isProduction ? "source-map" : "cheap-module-source-map",
  };
```

优化配置

```
const path = require("path");
const ESLintWebpackPlugin = require("eslint-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");
```

```
const ImageMinimizerPlugin = require("image-minimizer-webpack-plugin");
const TerserWebpackPlugin = require("terser-webpack-plugin");
const CopyPlugin = require("copy-webpack-plugin");
const { VueLoaderPlugin } = require("vue-loader");
const { DefinePlugin } = require("webpack");
const AutoImport = require("unplugin-auto-import/webpack");
const Components = require("unplugin-vue-components/webpack");
const { ElementPlusResolver } = require("unplugin-vue-
components/resolvers");
// 需要通过 cross-env 定义环境变量
const isProduction = process.env.NODE_ENV === "production";

const getStyleLoaders = (preProcessor) => {
  return [
    isProduction ? MiniCssExtractPlugin.loader : "vue-style-loader",
    "css-loader",
    {
      loader: "postcss-loader",
      options: {
        postcssOptions: {
          plugins: ["postcss-preset-env"],
        },
      },
    },
    preProcessor && {
      loader: preProcessor,
      options:
        preProcessor === "sass-loader"
          ? {
              // 自定义主题：自动引入我们定义的scss文件
              additionalData: `@use "@/styles/element/index.scss" as *;`,
            }
          : {},
    },
  ].filter(Boolean);
};

module.exports = {
  entry: "./src/main.js",
  output: {
    path: isProduction ? path.resolve(__dirname, "../dist") : undefined,
    filename: isProduction
      ? "static/js/[name].[contenthash:10].js"
      : "static/js/[name].js",
    chunkFilename: isProduction
      ? "static/js/[name].[contenthash:10].chunk.js"
      : "static/js/[name].chunk.js",
    assetModuleFilename: "static/js/[hash:10][ext][query]",
    clean: true,
  },
  module: {
```

```
    rules: [
      {
        test: /\.css$/,
        use: getStyleLoaders(),
      },
      {
        test: /\.less$/,
        use: getStyleLoaders("less-loader"),
      },
      {
        test: /\.s[ac]ss$/,
        use: getStyleLoaders("sass-loader"),
      },
      {
        test: /\.styl$/,
        use: getStyleLoaders("stylus-loader"),
      },
      {
        test: /\.(png|jpe?g|gif|svg)$/,
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024,
          },
        },
      },
      {
        test: /\.(ttf|woff2?)$/,
        type: "asset/resource",
      },
      {
        test: /\.(jsx|js)$/,
        include: path.resolve(__dirname, "../src"),
        loader: "babel-loader",
        options: {
          cacheDirectory: true,
          cacheCompression: false,
          plugins: [
            // "@babel/plugin-transform-runtime" // presets中包含了
          ],
        },
      },
      // vue-loader不支持oneOf
      {
        test: /\.vue$/,
        loader: "vue-loader", // 内部会给vue文件注入HMR功能代码
        options: {
          // 开启缓存
          cacheDirectory: path.resolve(
            __dirname,
            "node_modules/.cache/vue-loader"
```

```
            ),
          },
        },
      ],
    },
    plugins: [
      new ESLintWebpackPlugin({
        context: path.resolve(__dirname, "../src"),
        exclude: "node_modules",
        cache: true,
        cacheLocation: path.resolve(
          __dirname,
          "../node_modules/.cache/.eslintcache"
        ),
      }),
      new HtmlWebpackPlugin({
        template: path.resolve(__dirname, "../public/index.html"),
      }),
      new CopyPlugin({
        patterns: [
          {
            from: path.resolve(__dirname, "../public"),
            to: path.resolve(__dirname, "../dist"),
            toType: "dir",
            noErrorOnMissing: true,
            globOptions: {
              ignore: ["**/index.html"],
            },
            info: {
              minimized: true,
            },
          },
        ],
      }),
      isProduction &&
        new MiniCssExtractPlugin({
          filename: "static/css/[name].[contenthash:10].css",
          chunkFilename: "static/css/[name].[contenthash:10].chunk.css",
        }),
      new VueLoaderPlugin(),
      new DefinePlugin({
        __VUE_OPTIONS_API__: "true",
        __VUE_PROD_DEVTOOLS__: "false",
      }),
      // 按需加载element-plus组件样式
      AutoImport({
        resolvers: [ElementPlusResolver()],
      }),
      Components({
        resolvers: [
          ElementPlusResolver({
```

```
            importStyle: "sass", // 自定义主题
          }),
        ],
      }),
    ].filter(Boolean),
    optimization: {
      minimize: isProduction,
      // 压缩的操作
      minimizer: [
        new CssMinimizerPlugin(),
        new TerserWebpackPlugin(),
        new ImageMinimizerPlugin({
          minimizer: {
            implementation: ImageMinimizerPlugin.imageminGenerate,
            options: {
              plugins: [
                ["gifsicle", { interlaced: true }],
                ["jpegtran", { progressive: true }],
                ["optipng", { optimizationLevel: 5 }],
                [
                  "svgo",
                  {
                    plugins: [
                      "preset-default",
                      "prefixIds",
                      {
                        name: "sortAttrs",
                        params: {
                          xmlnsOrder: "alphabetical",
                        },
                      },
                    ],
                  },
                ],
              ],
            },
          },
        }),
      ],
      splitChunks: {
        chunks: "all",
        cacheGroups: {
          // layouts通常是admin项目的主体布局组件，所有路由组件都要使用的
          // 可以单独打包，从而复用
          // 如果项目中没有，请删除
          layouts: {
            name: "layouts",
            test: path.resolve(__dirname, "../src/layouts"),
            priority: 40,
          },
```

```
          // 如果项目中使用element-plus，此时将所有node_modules打包在一起，那么打包输出文件会比较大。
          // 所以我们将node_modules中比较大的模块单独打包，从而并行加载速度更好
          // 如果项目中没有，请删除
          elementUI: {
            name: "chunk-elementPlus",
            test: /[\\/]node_modules[\\/]_?element-plus(.*)/,
            priority: 30,
          },
          // 将vue相关的库单独打包，减少node_modules的chunk体积。
          vue: {
            name: "vue",
            test: /[\\/]node_modules[\\/]vue(.*)[\\/]/,
            chunks: "initial",
            priority: 20,
          },
          libs: {
            name: "chunk-libs",
            test: /[\\/]node_modules[\\/]/,
            priority: 10, // 权重最低，优先考虑前面内容
            chunks: "initial",
          },
        },
      },
      runtimeChunk: {
        name: (entrypoint) => `runtime~${entrypoint.name}`,
      },
    },
    resolve: {
      extensions: [".vue", ".js", ".json"],
      alias: {
        // 路径别名
        "@": path.resolve(__dirname, "../src"),
      },
    },
    devServer: {
      open: true,
      host: "localhost",
      port: 3000,
      hot: true,
      compress: true,
      historyApiFallback: true, // 解决vue-router刷新404问题
    },
    mode: isProduction ? "production" : "development",
    devtool: isProduction ? "source-map" : "cheap-module-source-map",
    performance: false,
  };
```

原理分析

• Loader 原理

- loader 概念

帮助 webpack 将不同类型的文件转换为 webpack 可识别的模块。

- loader 执行顺序

1. 分类

- pre： 前置 loader
- normal： 普通 loader
- inline： 内联 loader
- post： 后置 loader

2. 执行顺序

- 4 类 loader 的执行优级为： `pre > normal > inline > post` 。
- 相同优先级的 loader 执行顺序为： `从右到左，从下到上` 。

例如：

```
// 此时loader执行顺序: loader3 - loader2 - loader1
module: {
  rules: [
    {
      test: /\.js$/,
      loader: "loader1",
    },
    {
      test: /\.js$/,
      loader: "loader2",
    },
    {
      test: /\.js$/,
      loader: "loader3",
    },
  ],
},
```

```
// 此时loader执行顺序: loader1 - loader2 - loader3
module: {
  rules: [
    {
      enforce: "pre",
      test: /\.js$/,
      loader: "loader1",
    },
    {
```

```
        // 没有enforce就是normal
        test: /\.js$/,
        loader: "loader2",
      },
      {
        enforce: "post",
        test: /\.js$/,
        loader: "loader3",
      },
    ],
  },
```

3. 使用 loader 的方式

- 配置方式：在 `webpack.config.js` 文件中指定 loader。（pre、normal、post loader）
- 内联方式：在每个 `import` 语句中显式指定 loader。（inline loader）

4. inline loader

用法：`import Styles from 'style-loader!css-loader?modules!./styles.css';`

含义：

- 使用 `css-loader` 和 `style-loader` 处理 `styles.css` 文件
- 通过 `!` 将资源中的 loader 分开

`inline loader` 可以通过添加不同前缀，跳过其他类型 loader。

- `!` 跳过 normal loader。

`import Styles from '!style-loader!css-loader?modules!./styles.css';`

- `-!` 跳过 pre 和 normal loader。

`import Styles from '-!style-loader!css-loader?modules!./styles.css';`

- `!!` 跳过 pre、 normal 和 post loader。

`import Styles from '!!style-loader!css-loader?modules!./styles.css';`

• 开发一个 loader

– 1. 最简单的 loader

```
// loaders/loader1.js
module.exports = function loader1(content) {
  console.log("hello loader");
  return content;
};
```

它接受要处理的源码作为参数，输出转换后的 js 代码。

2. loader 接受的参数

- `content` 源文件的内容
- `map` SourceMap 数据
- `meta` 数据，可以是任何内容

loader 分类

1. 同步 loader

```
module.exports = function (content, map, meta) {
  return content;
};
```

`this.callback` 方法则更灵活，因为它允许传递多个参数，而不仅仅是 `content`。

```
module.exports = function (content, map, meta) {
  // 传递map，让source-map不中断
  // 传递meta，让下一个loader接收到其他参数
  this.callback(null, content, map, meta);
  return; // 当调用 callback() 函数时，总是返回 undefined
};
```

2. 异步 loader

```
module.exports = function (content, map, meta) {
  const callback = this.async();
  // 进行异步操作
  setTimeout(() => {
    callback(null, result, map, meta);
  }, 1000);
};
```

> “
>
> 由于同步计算过于耗时，在 Node.js 这样的单线程环境下进行此操作并不是好的方案，我们建议尽可能地使你的 loader 异步化。但如果计算量很小，同步 loader 也是可以的。

3. Raw Loader

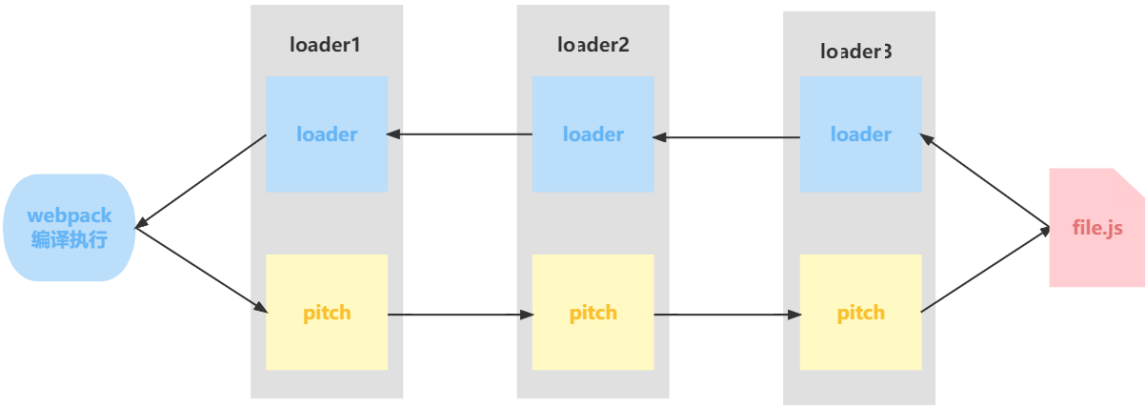
默认情况下，资源文件会被转化为 UTF-8 字符串，然后传给 loader。通过设置 raw 为 true，loader 可以接收原始的 Buffer。

```
module.exports = function (content) {
  // content是一个Buffer数据
  return content;
};
module.exports.raw = true; // 开启 Raw Loader
```

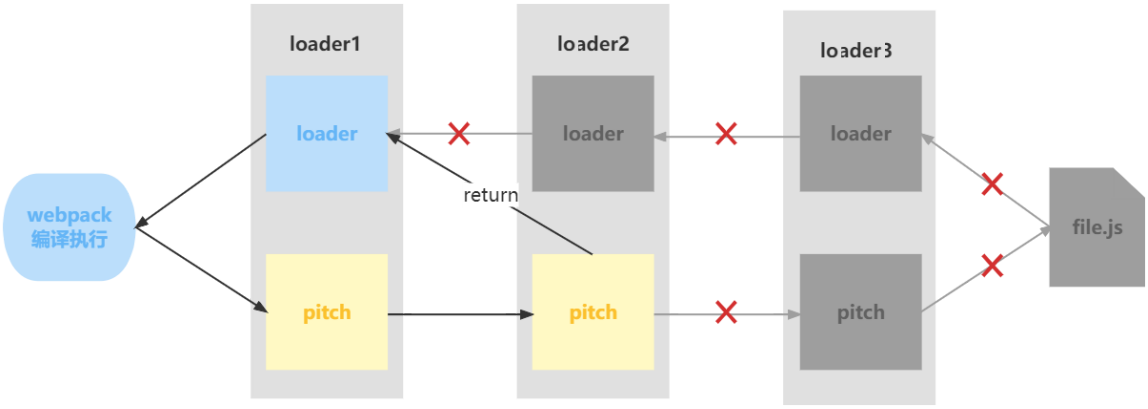
4. Pitching Loader

```
module.exports = function (content) {
  return content;
};
module.exports.pitch = function (remainingRequest, precedingRequest, data) {
  console.log("do somethings");
};
```

webpack 会先从左到右执行 loader 链中的每个 loader 上的 pitch 方法（如果有），然后再从右到左执行 loader 链中的每个 loader 上的普通 loader 方法。



在这个过程中如果任何 pitch 有返回值，则 loader 链被阻断。webpack 会跳过后面所有的的 pitch 和 loader，直接进入上一个 loader 。



• loader API

方法名	含义	用法
this.async	异步回调 loader。返回 this.callback	const callback = this.async()
this.callback	可以同步或者异步调用的并返回多个结果的函数	this.callback(err, content, sourceMap?, meta?)
this.getOptions(schema)	获取 loader 的 options	this.getOptions(schema)
this.emitFile	产生一个文件	this.emitFile(name, content, sourceMap)
this.utils.contextify	返回一个相对路径	this.utils.contextify(context, request)
this.utils.absolutify	返回一个绝对路径	this.utils.absolutify(context, request)

> “
>
> 更多文档，请查阅 webpack 官方 loader api 文档

• 手写 clean-log-loader

作用：用来清理 js 代码中的 `console.log`

```
// loaders/clean-log-loader.js
module.exports = function cleanLogLoader(content) {
  // 将console.log替换为空
  return content.replace(/console\.log\(.*\);?/g, "");
};
```

• 手写 banner-loader

作用：给 js 代码添加文本注释

- loaders/banner-loader/index.js

```
const schema = require("./schema.json");

module.exports = function (content) {
  // 获取loader的options，同时对options内容进行校验
  // schema是options的校验规则（符合 JSON schema 规则）
  const options = this.getOptions(schema);
```

```
    const prefix = `
      /*
      * Author: ${options.author}
      */
    `;

    return `${prefix} \n ${content}`;
  };
```

- loaders/banner-loader/schema.json

```
{
  "type": "object",
  "properties": {
    "author": {
      "type": "string"
    }
  },
  "additionalProperties": false
}
```

• 手写 babel-loader

作用：编译 js 代码，将 ES6+语法编译成 ES5-语法。

- 下载依赖

```
npm i @babel/core @babel/preset-env -D
```

- loaders/babel-loader/index.js

```
const schema = require("./schema.json");
const babel = require("@babel/core");

module.exports = function (content) {
  const options = this.getOptions(schema);
  // 使用异步loader
  const callback = this.async();
  // 使用babel对js代码进行编译
  babel.transform(content, options, function (err, result) {
    callback(err, result.code);
  });
};
```

- loaders/banner-loader/schema.json

```
{
  "type": "object",
  "properties": {
    "presets": {
      "type": "array"
    }
  },
  "additionalProperties": true
}
```

手写 file-loader

作用：将文件原封不动输出出去

- 下载包

```
npm i loader-utils -D
```

- loaders/file-loader.js

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

function fileLoader(content) {
  // 根据文件内容生产一个新的文件名称
  const filename = loaderUtils.interpolateName(this, "[hash].[ext]", {
    content,
  });
  // 输出文件
  this.emitFile(filename, content);
  // 暴露出去，给js引用。
  // 记得加上''
  return `export default '${filename}'`;
}

// loader 解决的是二进制的内容
// 图片是 Buffer 数据
fileLoader.raw = true;

module.exports = fileLoader;
```

- loader 配置

```
{
  test: /\.(png|jpe?g|gif)$/,
  loader: "./loaders/file-loader.js",
  type: "javascript/auto", // 解决图片重复打包问题
},
```

• 手写 style-loader

作用：动态创建 style 标签，插入 js 中的样式代码，使样式生效。

- loaders/style-loader.js

```
const styleLoader = () => {};

styleLoader.pitch = function (remainingRequest) {
  /*
    remainingRequest: C:\Users\86176\Desktop\source\node_modules\css-loader\dist\cjs.js!C:\Users\86176\Desktop\source\src\css\index.css
      这里是inline loader用法，代表后面还有一个css-loader等待处理

    最终我们需要将remainingRequest中的路径转化成相对路径，webpack才能处理
      希望得到：../../node_modules/css-loader/dist/cjs.js!./index.css

    所以：需要将绝对路径转化成相对路径
    要求：
      1. 必须是相对路径
      2. 相对路径必须以 ./ 或 ../ 开头
      3. 相对路径的路径分隔符必须是 / ，不能是 \
  */
  const relativeRequest = remainingRequest
    .split("!")
    .map((part) => {
      // 将路径转化为相对路径
      const relativePath = this.utils.contextify(this.context, part);
      return relativePath;
    })
    .join("!");

  /*
    !!${relativeRequest}
      relativeRequest：../../node_modules/css-loader/dist/cjs.js!./index.css
      relativeRequest是inline loader用法，代表要处理的index.css资源，使用css-loader处理
      !!代表禁用所有配置的loader，只使用inline loader。（也就是外面我们style-loader和css-loader），它们被禁用了，只是用我们指定的inline loader，也就是css-loader

    import style from "!!${relativeRequest}"
      引入css-loader处理后的css文件
      为什么需要css-loader处理css文件，不是我们直接读取css文件使用呢?
```

```
35          因为可能存在@import导入css语法，这些语法就要通过css-loader解析才能变成一个css文件，否则我们引入的css资源会缺少
36        const styleEl = document.createElement('style')
37          动态创建style标签
38        styleEl.innerHTML = style
39          将style标签内容设置为处理后的css代码
40        document.head.appendChild(styleEl)
41          添加到head中生效
42      */
43      const script = `
44        import style from "!!${relativeRequest}"
45        const styleEl = document.createElement('style')
46        styleEl.innerHTML = style
47        document.head.appendChild(styleEl)
48      `;
49
50      // style-loader是第一个loader，由于return导致熔断，所以其他loader不执行了（不管是normal还是pitch）
51      return script;
52    };
53
54    module.exports = styleLoader;
```

Plugin 原理

• Plugin 的作用

通过插件我们可以扩展 webpack，加入自定义的构建行为，使 webpack 可以执行更广泛的任务，拥有更强的构建能力。

• Plugin 工作原理

> webpack 就像一条生产线，要经过一系列处理流程后才能将源文件转换成输出结果。 这条生产线上的每个处理流程的职责都是单一的，多个流程之间有存在依赖关系，只有完成当前处理后才能交给下一个流程去处理。
> 插件就像是一个插入到生产线中的一个功能，在特定的时机对生产线上的资源做处理。webpack 通过 Tapable 来组织这条复杂的生产线。 webpack 在运行过程中会广播事件，插件只需要监听它所关心的事件，就能加入到这条生产线中，去改变生产线的运作。
> webpack 的事件流机制保证了插件的有序性，使得整个系统扩展性很好。
> ——「深入浅出 Webpack」

站在代码逻辑的角度就是：webpack 在编译代码过程中，会触发一系列 `Tapable` 钩子事件，插件所做的，就是找到相应的钩子，往上面挂上自己的任务，也就是注册事件，这样，当 webpack 构建的时候，插件注册的事件就会随着钩子的触发而执行了。

• Webpack 内部的钩子

- 什么是钩子

钩子的本质就是：事件。为了方便我们直接介入和控制编译过程，webpack 把编译过程中触发的各类关键事件封装成事件接口暴露了出来。这些接口被很形象地称做：`hooks`（钩子）。开发插件，离不开这些钩子。

- Tapable

`Tapable` 为 webpack 提供了统一的插件接口（钩子）类型定义，它是 webpack 的核心功能库。webpack 中目前有十种 `hooks`，在 `Tapable` 源码中可以看到，他们是：

```
// https://github.com/webpack/tapable/blob/master/lib/index.js
exports.SyncHook = require("./SyncHook");
exports.SyncBailHook = require("./SyncBailHook");
exports.SyncWaterfallHook = require("./SyncWaterfallHook");
exports.SyncLoopHook = require("./SyncLoopHook");
exports.AsyncParallelHook = require("./AsyncParallelHook");
exports.AsyncParallelBailHook = require("./AsyncParallelBailHook");
exports.AsyncSeriesHook = require("./AsyncSeriesHook");
exports.AsyncSeriesBailHook = require("./AsyncSeriesBailHook");
exports.AsyncSeriesLoopHook = require("./AsyncSeriesLoopHook");
exports.AsyncSeriesWaterfallHook = require("./AsyncSeriesWaterfallHook");
exports.HookMap = require("./HookMap");
exports.MultiHook = require("./MultiHook");
```

`Tapable` 还统一暴露了三个方法给插件，用于注入不同类型的自定义构建行为：

- `tap`：可以注册同步钩子和异步钩子。
- `tapAsync`：回调方式注册异步钩子。
- `tapPromise`：Promise 方式注册异步钩子。

• Plugin 构建对象

- Compiler

compiler 对象中保存着完整的 Webpack 环境配置，每次启动 webpack 构建时它都是一个独一无二，仅仅会创建一次的对象。

这个对象会在首次启动 Webpack 时创建，我们可以通过 compiler 对象上访问到 Webapck 的主环境配置，比如 loader 、 plugin 等等配置信息。

它有以下主要属性：

- `compiler.options` 可以访问本次启动 webpack 时候所有的配置文件，包括但不限于 loaders 、 entry 、 output 、 plugin 等等完整配置信息。

- `compiler.inputFileSystem` 和 `compiler.outputFileSystem` 可以进行文件操作，相当于 Nodejs 中 fs。
- `compiler.hooks` 可以注册 tapable 的不同种类 Hook，从而可以在 compiler 生命周期中植入不同的逻辑。

> “
> compiler hooks 文档

Compilation

compilation 对象代表一次资源的构建，compilation 实例能够访问所有的模块和它们的依赖。

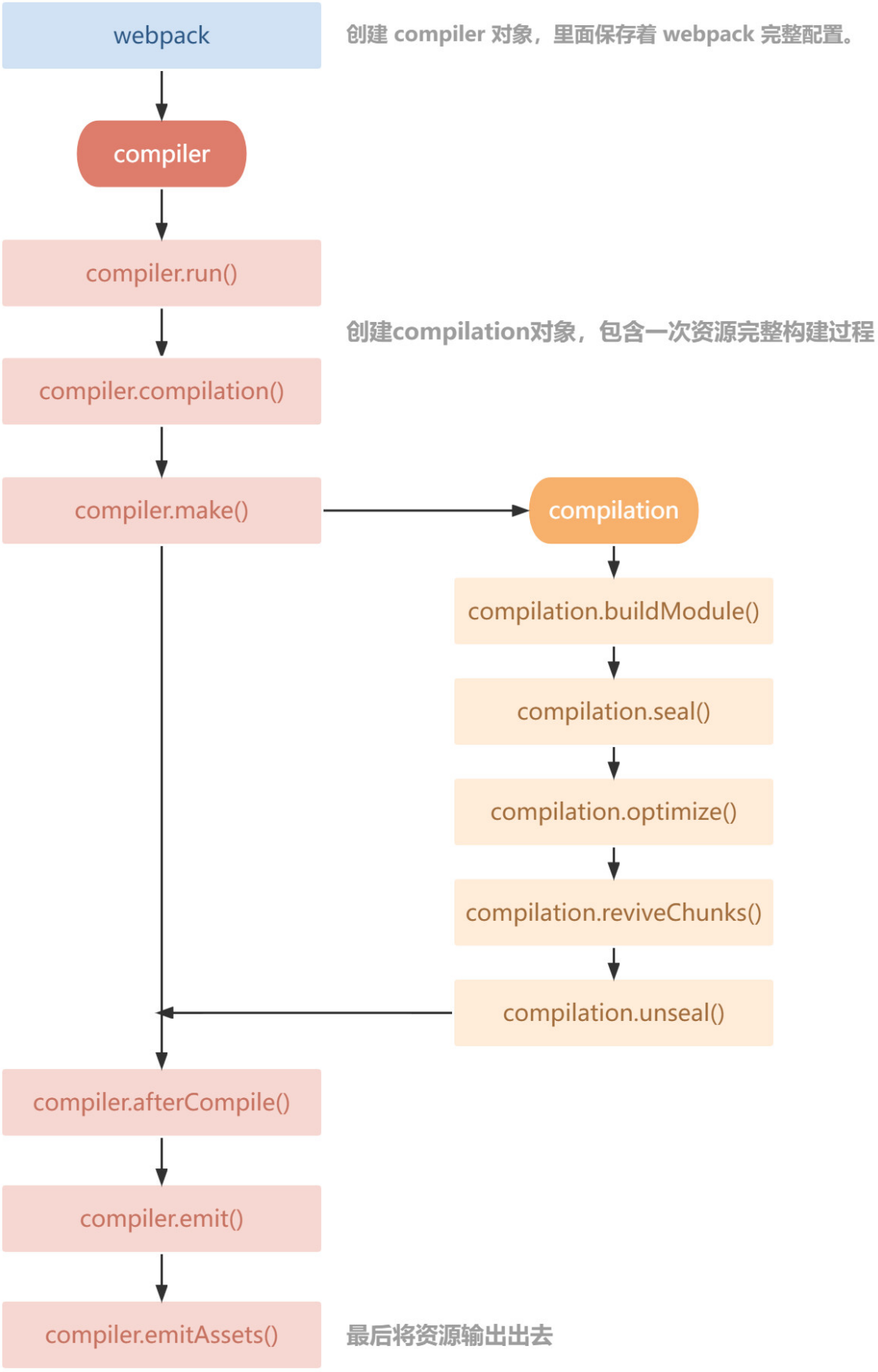
一个 compilation 对象会对构建依赖图中所有模块，进行编译。 在编译阶段，模块会被加载(load)、封存(seal)、优化(optimize)、 分块(chunk)、哈希(hash)和重新创建(restore)。

它有以下主要属性：

- `compilation.modules` 可以访问所有模块，打包的每一个文件都是一个模块。
- `compilation.chunks` chunk 即是多个 modules 组成而来的一个代码块。入口文件引入的资源组成一个 chunk，通过代码分割的模块又是另外的 chunk。
- `compilation.assets` 可以访问本次打包生成所有文件的结果。
- `compilation.hooks` 可以注册 tapable 的不同种类 Hook，用于在 compilation 编译模块阶段进行逻辑添加以及修改。

> “
> compilation hooks 文档

生命周期简图


webpack
创建 compiler 对象，里面保存着 webpack 完整配置。
compiler
compiler.run()
创建compilation对象，包含一次资源完整构建过程
compiler.compilation()
compiler.make()
compilation
compilation.buildModule()
compilation.seal()
compilation.optimize()
compilation.reviveChunks()
compilation.unseal()
compiler.afterCompile()
compiler.emit()
compiler.emitAssets()
最后将资源输出出去

• 开发一个插件

- 最简单的插件

- plugins/test-plugin.js

```
class TestPlugin {
  constructor() {
    console.log("TestPlugin constructor()");
  }
  // 1. webpack读取配置时，new TestPlugin() ，会执行插件 constructor 方法
  // 2. webpack创建 compiler 对象
  // 3. 遍历所有插件，调用插件的 apply 方法
  apply(compiler) {
    console.log("TestPlugin apply()");
  }
}

module.exports = TestPlugin;
```

- 注册 hook

```
class TestPlugin {
  constructor() {
    console.log("TestPlugin constructor()");
  }
  // 1. webpack读取配置时，new TestPlugin() ，会执行插件 constructor 方法
  // 2. webpack创建 compiler 对象
  // 3. 遍历所有插件，调用插件的 apply 方法
  apply(compiler) {
    console.log("TestPlugin apply()");

    // 从文档可知，compile hook 是 SyncHook，也就是同步钩子，只能用tap注册
    compiler.hooks.compile.tap("TestPlugin", (compilationParams) => {
      console.log("compiler.compile()");
    });

    // 从文档可知，make 是 AsyncParallelHook，也就是异步并行钩子，特点就是异步任务同时执行
    // 可以使用 tap、tapAsync、tapPromise 注册。
    // 如果使用tap注册的话，进行异步操作是不会等待异步操作执行完成的。
    compiler.hooks.make.tap("TestPlugin", (compilation) => {
      setTimeout(() => {
        console.log("compiler.make() 111");
      }, 2000);
    });

    // 使用tapAsync、tapPromise注册，进行异步操作会等异步操作做完再继续往下执行
    compiler.hooks.make.tapAsync("TestPlugin", (compilation, callback) => {
```

```
      setTimeout(() => {
        console.log("compiler.make() 222");
        // 必须调用
        callback();
      }, 1000);
    });

    compiler.hooks.make.tapPromise("TestPlugin", (compilation) => {
      console.log("compiler.make() 333");
      // 必须返回promise
      return new Promise((resolve) => {
        resolve();
      });
    });

    // 从文档可知，emit 是 AsyncSeriesHook，也就是异步串行钩子，特点就是异步任务顺序执行
    compiler.hooks.emit.tapAsync("TestPlugin", (compilation, callback) => {
      setTimeout(() => {
        console.log("compiler.emit() 111");
        callback();
      }, 3000);
    });

    compiler.hooks.emit.tapAsync("TestPlugin", (compilation, callback) => {
      setTimeout(() => {
        console.log("compiler.emit() 222");
        callback();
      }, 2000);
    });

    compiler.hooks.emit.tapAsync("TestPlugin", (compilation, callback) => {
      setTimeout(() => {
        console.log("compiler.emit() 333");
        callback();
      }, 1000);
    });
  }
}

module.exports = TestPlugin;
```

启动调试

通过调试查看 `compiler` 和 `compilation` 对象数据情况。

1. package.json 配置指令

```
{
  "name": "source",
```

```
  "version": "1.0.0",
  "scripts": {
    "debug": "node --inspect-brk ./node_modules/webpack-cli/bin/cli.js"
  },
  "keywords": [],
  "author": "xiongjian",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.17.10",
    "@babel/preset-env": "^7.17.10",
    "css-loader": "^6.7.1",
    "loader-utils": "^3.2.0",
    "webpack": "^5.72.0",
    "webpack-cli": "^4.9.2"
  }
}
```

2. 运行指令

```
npm run debug
```

此时控制台输出以下内容：

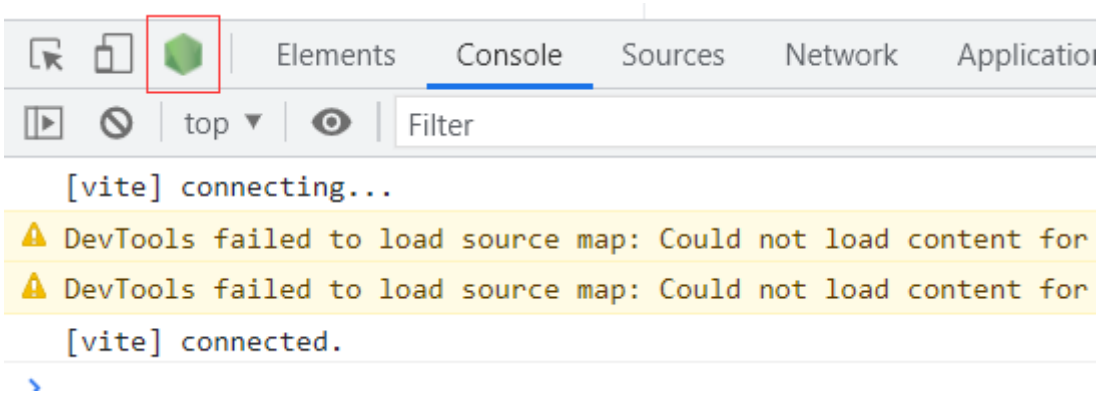
```
PS C:\Users\86176\Desktop\source> npm run debug

> source@1.0.0 debug
> node --inspect-brk ./node_modules/webpack-cli/bin/cli.js

Debugger listening on ws://127.0.0.1:9229/629ea097-7b52-4011-93a7-02f83c75c797
For help, see: https://nodejs.org/en/docs/inspecto
```

3. 打开 Chrome 浏览器，F12 打开浏览器调试控制台。

此时控制台会显示一个绿色的图标



4. 点击绿色的图标进入调试模式。
5. 在需要调试代码处用 `debugger` 打断点，代码就会停止运行，从而调试查看数据情况。

- **BannerWebpackPlugin**

1. 作用：给打包输出文件添加注释。
2. 开发思路:

- 需要打包输出前添加注释：需要使用 `compiler.hooks.emit` 钩子, 它是打包输出前触发。
- 如何获取打包输出的资源？ `compilation.assets` 可以获取所有即将输出的资源文件。

3. 实现：

```
// plugins/banner-webpack-plugin.js
class BannerWebpackPlugin {
  constructor(options = {}) {
    this.options = options;
  }

  apply(compiler) {
    // 需要处理文件
    const extensions = ["js", "css"];

    // emit是异步串行钩子
    compiler.hooks.emit.tapAsync("BannerWebpackPlugin", (compilation, callback) => {
      // compilation.assets包含所有即将输出的资源
      // 通过过滤只保留需要处理的文件
      const assetPaths = Object.keys(compilation.assets).filter((path) => {
        const splitted = path.split(".");
        return extensions.includes(splitted[splitted.length - 1]);
      });

      assetPaths.forEach((assetPath) => {
        const asset = compilation.assets[assetPath];

        const source = `/*
* Author: ${this.options.author}
*/\n${asset.source()}`;

        // 覆盖资源
        compilation.assets[assetPath] = {
          // 资源内容
          source() {
            return source;
          },
          // 资源大小
          size() {
            return source.length;
          },
        };
      });

```

```
        callback();
      });
    }
  }

  module.exports = BannerWebpackPlugin;
```

- **CleanWebpackPlugin**

1. 作用：在 webpack 打包输出前将上次打包内容清空。
2. 开发思路：

- 如何在打包输出前执行？需要使用 `compiler.hooks.emit` 钩子，它是打包输出前触发。
- 如何清空上次打包内容？
  - 获取打包输出目录：通过 compiler 对象。
  - 通过文件操作清空内容：通过 `compiler.outputFileSystem` 操作文件。

3. 实现：

```
// plugins/clean-webpack-plugin.js
class CleanWebpackPlugin {
  apply(compiler) {
    // 获取操作文件的对象
    const fs = compiler.outputFileSystem;
    // emit是异步串行钩子
    compiler.hooks.emit.tapAsync("CleanWebpackPlugin", (compilation, callback) => {
      // 获取输出文件目录
      const outputPath = compiler.options.output.path;
      // 删除目录所有文件
      const err = this.removeFiles(fs, outputPath);
      // 执行成功err为undefined，执行失败err就是错误原因
      callback(err);
    });
  }

  removeFiles(fs, path) {
    try {
      // 读取当前目录下所有文件
      const files = fs.readdirSync(path);

      // 遍历文件，删除
      files.forEach((file) => {
        // 获取文件完整路径
        const filePath = `${path}/${file}`;
        // 分析文件
        const fileStat = fs.statSync(filePath);
        // 判断是否是文件夹
```

```
          if (fileStat.isDirectory()) {
            // 是文件夹需要递归遍历删除下面所有文件
            this.removeFiles(fs, filePath);
          } else {
            // 不是文件夹就是文件，直接删除
            fs.unlinkSync(filePath);
          }
        });

        // 最后删除当前目录
        fs.rmdirSync(path);
      } catch (e) {
        // 将产生的错误返回出去
        return e;
      }
    }
  }

  module.exports = CleanWebpackPlugin;
```

- **AnalyzeWebpackPlugin**

1. 作用：分析 webpack 打包资源大小，并输出分析文件。
2. 开发思路:

- 在哪做? `compiler.hooks.emit`，它是在打包输出前触发，我们需要分析资源大小同时添加上分析后的 md 文件。

3. 实现：

```
// plugins/analyze-webpack-plugin.js
class AnalyzeWebpackPlugin {
  apply(compiler) {
    // emit是异步串行钩子
    compiler.hooks.emit.tap("AnalyzeWebpackPlugin", (compilation) => {
      // Object.entries将对象变成二维数组。二维数组中第一项值是key，第二项值是value
      const assets = Object.entries(compilation.assets);

      let source = "# 分析打包资源大小 \n| 名称 | 大小 |\n| --- | --- |";

      assets.forEach(([filename, file]) => {
        source += `\n| ${filename} | ${file.size()} |`;
      });

      // 添加资源
      compilation.assets["analyze.md"] = {
        source() {
          return source;
        },
```

```
          size() {
            return source.length;
          },
        };
      });
    }
  }

  module.exports = AnalyzeWebpackPlugin;
```

- **InlineChunkWebpackPlugin**

1. 作用：webpack 打包生成的 runtime 文件太小了，额外发送请求性能不好，所以需要将其内联到 js 中，从而减少请求数量。
2. 开发思路:

- 我们需要借助 `html-webpack-plugin` 来实现
  - 在 `html-webpack-plugin` 输出 index.html 前将内联 runtime 注入进去
  - 删除多余的 runtime 文件
- 如何操作 `html-webpack-plugin` ？ 官方文档

3. 实现：

```
// plugins/inline-chunk-webpack-plugin.js
const HtmlWebpackPlugin = require("safe-require")("html-webpack-plugin");

class InlineChunkWebpackPlugin {
  constructor(tests) {
    this.tests = tests;
  }

  apply(compiler) {
    compiler.hooks.compilation.tap("InlineChunkWebpackPlugin", (compilation) => {
      const hooks = HtmlWebpackPlugin.getHooks(compilation);

      hooks.alterAssetTagGroups.tap("InlineChunkWebpackPlugin", (assets) => {
        assets.headTags = this.getInlineTag(assets.headTags, compilation.assets);
        assets.bodyTags = this.getInlineTag(assets.bodyTags, compilation.assets);
      });

      hooks.afterEmit.tap("InlineChunkHtmlPlugin", () => {
        Object.keys(compilation.assets).forEach((assetName) => {
          if (this.tests.some((test) => assetName.match(test))) {
            delete compilation.assets[assetName];
          }
```

```
          });
        });
      });
    }

    getInlineTag(tags, assets) {
      return tags.map((tag) => {
        if (tag.tagName !== "script") return tag;

        const scriptName = tag.attributes.src;

        if (!this.tests.some((test) => scriptName.match(test))) return tag;

        return { tagName: "script", innerHTML: assets[scriptName].source(),
closeTag: true };
      });
    }
}

module.exports = InlineChunkWebpackPlugin;
```