第 3 章

前端构建工具

本章全面介绍前端开发中最流行和最常见的模块化构建工具，包括 Webpack、Rollup、Lerna、Vite 工具的原理和开发实践。通过本章读者可以全面掌握各种构建工具的使用场景、优缺点和用法。

3.1 前端构建工具介绍

前端构建工具能帮助前端开发人员把编写的 Less、SASS 等代码编译成原生 CSS，也可以将多个 JavaScript 文件合并及压缩成一个 JavaScript 文件，对前端不同的资源文件进行打包，它的作用就是通过将代码编译、压缩、合并等操作，来减少代码体积，减少网络请求，方便在服务器上运行。

3.1.1 为什么需要构建工具

随着前端开发项目的规模越来越大，业务模块和代码模块也越来越复杂，因此在项目开发过程中需要高效的构建工具帮助开发者解决项目中的痛点问题。下面列举几个企业项目开发中的痛点问题：

(1) 在大型的前端项目中，浏览器端的模块化存在两个主要问题，第一是效率问题，精细的模块化(更多的 JS 文件)带来大量的网络请求，从而降低了页面访问效率；第二是兼容性问题，浏览器端不支持 CommonJS 模块化，而很多第三方库使用了 CommonJS 模块化。

(2) 在大型前端项目开发中，需要考虑很多非业务问题，如执行效率、兼容性、代码的可维护性、可拓展性，团队协作、测试等工程问题。

(3) 在浏览器端，开发环境和线上环境的侧重点完全不一样。

开发环境：

- 模块划分得越精细越好；
- 不需要考虑兼容性问题；
- 支持多种模块化标准；
- 支持 NPM 和其他包管理器下载的模块；

■ 能解决其他工程化的问题。

线上环境：

■ 文件越少越好，减少网络请求；

■ 文件体积越小越好，传输速度快；

■ 兼容所有浏览器；

■ 代码内容越乱越好；

■ 执行效率越高越好。

开发环境和线上环境面临的情况有较大差异，因此需要一个工具能够让开发者专心地书写开发环境的代码，然后利用这个工具将开发时编写的所有代码转化为运行时所需要的资源文件。这样的工具称为构建工具，如图 3-1 所示。

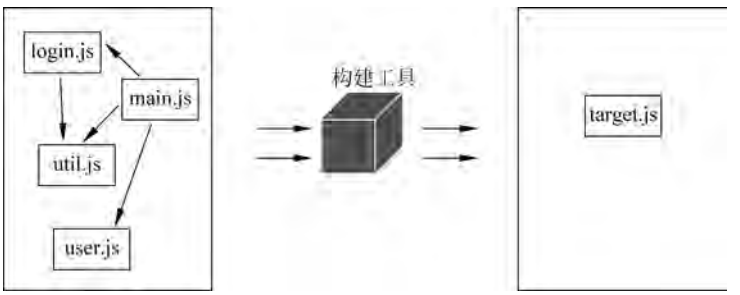

图 3-1　构建工具的作用

3.1.2　构建工具的功能需求

前端构建工具的本质是要解决前端整体资源文件的模块化，并不单指 JS 模块化，随着 JavaScript 在企业中大规模应用，复杂的前端项目越来越需要通过构建工具来帮助实现以下几方面的功能要求。

1. 模块打包器(Module Bundler)

(1) 解决模块 JS 打包问题。

(2) 可以将零散的 JS 代码整合到一个 JS 文件中。

2. 模块加载器(Loader)

(1) 对于存在兼容问题的代码，可以通过引入对应的解析编译模块进行编译。

(2) 对各种代码进行编译前的预处理。

3. 代码拆分(Code Splitting)

(1) 将应用中的代码按需求进行打包，避免因将所有的代码打包成一个文件而使文件过大的问题。

(2) 可以将应用加载初期所需代码打包在一起，而其余的代码在后续执行中按需加载，实现增量加载或渐进加载。

(3) 可以避免出现文件过大或文件太碎的问题。

4. 支持不同类型的资源模块

解决前端各种静态资源的打包。

3.1.3 前端构建工具演变

随着前端开发规模的增大，也不断推动前端构建工具链的向前发展，这里可以形象地总结为了以下几个阶段：石器时代、青铜时代、白银时代、黄金时代。

石器时代(纯手工)：需要纯手工打包构建、预览文件、刷新文件；代表是 Ant 脚本＋YUI Compressor，如图 3-2 所示，由 Yahoo 所发展的一套 JavaScript 与 CSS 压缩工具，可以协助网页开发者生成最小化的网页。

图 3-2 YUI Compressor

青铜时代(脚本式)：通过编写 bash 或 Node. js 任务脚本，实现命令式的热更新(HMR)和自动打包，代表为 Grunt、Gulp，如图 3-3 所示。

图 3-3 Grunt 和 Gulp

白银时代(Bundle)：通过集成式构建工具完成热更新(HMR)，处理兼容和编译打包。打包代表为 Babel、Webpack、Rollup、Parcel，如图 3-4 所示。

图 3-4 白银时代的打包工具代表

黄金时代(Bundleless)：通过浏览器解析原生 ESM 模块实现 Bundleless 的开发预览及热更新(HMR)，不打包发布或采用 Webpack 等集成式工具兼容打包，以保证兼容性，代表为 ESBuild、Snowpack、Vite，如图 3-5 所示。

图 3-5　黄金时代的打包工具代表

说明：ESBuild 是采用 Go 语言编写的 bundler，对标 tsc 和 Babel，只负责 ts 和 js 文件的转换。

3.1.4　NPM 与 Yarn、PNPM

NPM 是随同 Node 一起安装的包管理工具，用于 Node 模块管理(包括安装、卸载、管理依赖等)。

Yarn 是由 Facebook、Google、Exponent 和 Tilde 联合推出的一个新的 JS 包管理工具，用来代替 NPM 及其他模块管理器现有的工作流程，并且保持了对 NPM 代理的兼容性。它在与现有工作流程功能相同的情况下保证了操作更快、更安全和更可靠。

PNPM 是一个速度快、磁盘空间高效的软件包管理器。PNPM 使用内容可寻址文件系统将所有模块目录中的所有文件存储在磁盘上。当使用 NPM 或 Yarn 时，如果有 100 个项目使用 lodash，则磁盘上将有 100 个 lodash 副本。当使用 PNPM 时，lodash 将存储在内容可寻址的存储中。

这 3 个主流包管理器的图标如图 3-6 所示。

PNPM 官方网站为 https://pnpm.io/。Yarn 官方网站为 https://yarnpkg.com/，中文网站为 http://yarnpkg.cn/zh-Hans/。

1. 安装方式

Yarn 的安装方式，命令如下：

```
#全局安装 Yarn
npm install --global yarn
#全局安装 PNPM
npm install -g pnpm
```

图 3-6　主流包管理器

PNPM 的安装方法，命令如下：

```
#通过 NPM 安装
npm install -g pnpm
#通过 NPX 安装
npx pnpm add -g pnpm
#升级
#一旦安装了 PNPM，就无须再使用其他软件包管理器进行升级，可以使用 PNPM 升级自己
pnpm add -g pnpm
```

2. 常用命令比较

NPM、Yarn、PNPM 的操作命令差别不大，具体可以参考表 3-1。

表 3-1　NPM、Yarn、PNPM 命令对比

功能说明	NPM	Yarn	PNPM
创建 package.json	npm init	yarn init	pnpm init
安装本地依赖包	npm install、npm i	yarn	pnpm install、pnpm i
安装并运行依赖包，保存至 package.json 文件中	npm install --save	yarn add	pnpm add
安装开发依赖包，并保存至 package.json 文件中	npm install --save-dev	yarn add --dev	pnpm add -D
更新本地依赖包	npm update	yarn upgrade	pnpm up pnpm upgrade
卸载本地依赖包	npm uninstall	yarn remove	pnpm remove

续表

功 能 说 明	NPM	Yarn	PNPM
安装全局依赖包	npm install -g	yarn global add	pnpm add -g
更新全局依赖	npm update -g	yarn global upgrade	pnpm upgrade --global
查看全局依赖包	npm ls -g	yarn global list	pnpm list
卸载全局依赖包	yarn global remove	npm uninstall -g	pnpm remove --global
清除缓存	npm cache clean	yarn cache clean	—

3. 选择哪个包管理器

如何选择合适的包管理器，表 3-2 列举了部分内容的对比情况，供大家选择。

表 3-2 NPM 和 Yarn、PNPM 功能对比

功 能 说 明	NPM	Yarn	PNPM
团队	Node.js 官方	Facebook	Zoltan Kochan Full stack web developer
工作流	完整	完整	完整
包下载速度	NPM 5.0 版本后快	并行下载，快	快
monorepo	NPM v7.x 以上版本支持	支持	支持

3.2 Webpack

Webpack 是由德国开发者 Tobias Koppers 开发的模块加载器，如图 3-7 所示。Webpack 最初主要想解决代码拆分的问题，而这也是 Webpack 今天受欢迎的主要原因。随着 Web 应用规模越写越大，移动设备越来越普及，拆分代码的需求与日俱增。如果不拆分代码，就很难实现期望的性能。

图 3-7 Webpack 框架 Logo

2014 年，Facebook 的 Instagram 的前端团队分享了他们在对前端页面加载进行性能优化时用到了 Webpack 的 Code Splitting(代码拆分)功能。随即 Webpack 被广泛传播使用，同时开发者也给 Webpack 社区贡献了大量的 Plugin(插件)和 Loader(转换器)。

3.2.1 Webpack 介绍

Webpack 是一个现代 JavaScript 应用程序的静态模块打包器(Module Bundler)。当 Webpack 处理应用程序时，它会递归地构建一个依赖关系图(Dependency Graph)，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个包。

1. Webpack 的优点

Webpack 相比较其他构建工具，具有以下几个优点。

(1) 智能解析：对 CommonJS、AMD、CMD 等支持得很好。

(2) 代码拆分：创建项目依赖树，每个依赖都可拆分成一个模块，从而可以按需加载。

(3) Loader：Webpack 核心模块之一，主要处理各类型文件编译转换及 Babel 语法转换。

(4) Plugin(插件系统)：强大的插件系统，可实现对代码压缩、分包 chunk、模块热替换等，也可实现自定义模块、对图片进行 base64 编码等，文档非常全面，自动化工作都有直接的解决方案。

(5) 快速高效：开发配置可以选择不同环境的配置模式，可选择打包文件使用异步 I/O 和多级缓存提高运行效率。

(6) 微前端支持：Module Federation 也对项目中如何使用微型前端应用提供了一种解决方案。

(7) 功能全面：最主流的前端模块打包工具，支持流行的框架打包，如 React、Vue、Angular 等，社区全面。

2. Webpack 的工作方式

Webpack 的工作方式是把项目当作一个整体，通过一个给定的主文件(如 index.js)，Webpack 将从这个主文件开始找到项目的所有依赖文件，使用 Loader 处理它们，最后打包为一个(或多个)浏览器可识别的 JavaScript 文件，如图 3-8 所示。

图 3-8 Webpack 运行方式

3.2.2　Webpack 安装与配置

接下来，详细介绍如何安装和配置 Webpack 工具。

1. 安装 Node.js

推荐到 Node.js 官网下载 stable 版本。NPM 作为 Node.js 包管理工具在安装 Node.js 时已经顺带安装好了。

注意：保持 Node.js 和 Webpack 的版本尽量新，可以提升 Webpack 打包速度。

2. 更新 Node.js

如果想更新 Node.js 版本，则可以使用 n 模块，命令如下：

```
node -v                          # 首先查看当前 Node 版本
npm info node                    # 可以查看 Node 版本信息
npm install -g n                 # 安装 n 模块
sudo n stable                    # 安装稳定版本
sudo n latest                    # 或者安装最新版本
```

3. 更新 NPM

如果需要更新 NPM，则可以执行的命令如下：

```
sudo npm install npm@latest -g
```

4. 项目创建及初始化

初始化一个 Webpack 编译的项目，命令如下：

```
mkdir webpack-demo          # 首先创建一个文件夹
cd webpack-demo             # 进入文件夹
npm init                    # 初始化项目，使项目符合 Node.js 的规范，也可以用 npm init -y
                            # 这一步会在文件夹中生成 package.json 文件，这个文件描述了
                            # Node 项目的一些信息
```

目录结构如下：

```
webpack-demo/
├——— node_modules/
├——— src/
|    └——— main.js                # entry 入口文件
├——— webpack.config.js           # Webpack 配置文件
├——— package-lock.json
└——— package.json                # 已安装 Webpack、Webpack-cli
```

5. Webpack 安装与卸载

注意：Webpack 安装需要同时安装 Webpack 和 Webpack-cli 这两个模块。

```
npm install webpack webpack-cli -g                    # 全局安装
npm uninstall webpack webpack-cli -g                  # 全局卸载
npm install webpack webpack-cli -D                    # 在项目中安装
webpack npm install webpack@4.46.0 webpack-cli -D     # 安装指定版本
npx webpack -v                                        # 在项目中安装查看 Webpack 版本号
npm info webpack                                      # 查看 Webpack 历史版本
```

6. 通过配置文件使用 Webpack

配置文件规定了 Webpack 该如何打包，而执行 npx webpack ./main.js 进行打包使用的则是 Webpack 提供的默认配置文件。

配置 webpack.config.js 文件，配置如下：

```
//webpack.config.js
const path = require('path');
module.exports = {
  mode: 'production',
  entry: './src/main.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
}
```

默认模式为 mode:production，如果 mode 被配置为 production，则打包出的文件会被压缩，如果 mode 被配置为 development，则不会被压缩。

entry 的意思是这个项目要打包，以及从哪一个文件开始打包。打包输出中 Chunk Names 配置的 main 就是 entry 中的 main。简写模式如下：

```
entry: {
 main: './main.js'
}
```

output 的意思是打包后的文件放在哪里：

(1) output.filename 指打包后的文件名。

(2) output.path 指打包后的文件放到哪一个文件夹下，是一个绝对路径。需要引入 Node 中的 path 模块，然后调用这个模块的 resolve 方法。

Webpack 配置文件的作用是设置配置的参数，提供给 Webpack-cli，Webpack 从 main.js 文件开始打包，打包生成的文件放到 bundle 文件夹下，生成的文件名叫作 bundle.js。如果运行 npx webpack 命令，则会按照配置文件进行打包。

Webpack 默认的配置文件名为 webpack.config.js，如果要使用自定义名字(例如 my.webpack.js 作为配置文件名)，则可以用指令 npx webpack --config my.webpack.js 实现。

7. 配置 package.json

NPM scripts 原理：当执行 npm run xx 命令时，实际上运行的是 package.json 文件中的 xx 命令。在 scripts 标签中使用 Webpack，会优先到当前项目的 node_modules 中查找是否安装了 Webpack（和直接使用 Webpack 命令时到全局查找是否安装 Webpack 不同），命令如下：

```
"scripts": {
   "dev": "npx webpack"
 },
```

如果运行 dev 命令，则会自动执行 webpack 命令。最后可以直接运行 npm run dev 命令进行 Webpack 打包。

执行 npm run dev 命令后，在命令行中会输出编译完成后的提示信息，效果如图 3-9 所示。

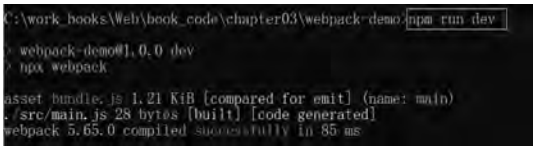

图 3-9　Webpack 编译提示

3.2.3　Webpack 基础

Webpack 的模块打包工具通过分析模块之间的依赖，最终将所有模块打包成一份或者多份代码包，供 HTML 直接引用。

1. 核心概念

Webpack 最核心的概念如下。

（1）Entry：入口文件，Webpack 会从该文件开始进行分析与编译。

（2）Output：出口路径，打包后创建包的文件路径及文件名。

（3）Module：模块，在 Webpack 中任何文件都可以作为一个模块，会根据配置使用不同的 Loader 进行加载和打包。

（4）Chunk：代码块，可以根据配置将所有模块代码合并成一个或多个代码块，以便按需加载，提高性能。

（5）Loader：模块加载器，进行各种文件类型的加载与转换。通过不同的 Loader，Webpack 有能力调用外部的脚本或工具，实现对不同格式文件的处理，例如分析及将 scss 转换为 css，或者把 ES6＋文件（ES6 和 ES7）转换为现代浏览器兼容的 JS 文件，对 React 的开发而言，合适的 Loader 可以把 React 中用到的 JSX 文件转换为 JS 文件。

（6）Plugin：拓展插件，可以通过 Webpack 相应的事件钩子，介入打包过程中的任意环

节，从而对代码按需修改。

2. 常见加载器(Loader)

Webpack 仅仅提供了打包功能和一套文件处理机制，然后通过生态中的各种 Loader 和 Plugin 对代码进行预编译和打包。

Loader 的作用：

(1) Loader 让 Webpack 能够去处理那些非 JavaScript 文件。

(2) Loader 专注实现资源模块加载从而实现模块的打包。

将常用的加载器分成以下三类。

(1) 编译转换类：将资源模块转换为 JS 代码。以 JS 形式工作的模块，如 css-loader。

(2) 文件操作类：将资源模块复制到输出目录，同时将文件的访问路径向外导出，如 file-loader。

(3) 代码检查类：对加载的资源文件(一般是代码)进行校验，以便统一代码风格，提高代码质量，一般不会修改生产环境的代码。

下面介绍最常用的几种加载器：解析 ES6+、处理 JSX、CSS/Less/SASS 样式、图片与字体。

3. 解析 ES6+

在 Webpack 中解析 ES6 需要使用 Babel，Babel 是一个 JavaScript 编译器，可以实现将 ES6+转换成浏览器能够识别的代码。

Babel 在执行编译时，可以依赖. babelrc 文件，当设置依赖文件时，会从项目的根目录下读取. babelrc 的配置项，. babelrc 配置文件主要是对预设(presets)和插件(plugins)进行配置。

下面介绍如何在 Webpack 中使用 Babel。

安装依赖，命令如下：

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

注意：Babel 7 推荐使用@babel/preset-env 套件来处理转译需求。顾名思义，preset 即“预制套件”，包含了各种可能用到的转译工具。之前的以年份为准的 preset 已经废弃了，现在统一用这个总包。同时，Babel 已经放弃开发 stage-* 包，以后的转译组件都只会放进 preset-env 包里。

配置 webpack. config. js 文件的 Loader，配置如下：

```
"scripts": {
module: {
      rules: [
          {
              test: /.js$/,
              use: 'babel-loader'
```

```
            }
        ]
    }
```

在根目录创建.babelrc,并配置 preset-env 对 ES6+语法特性进行转换,配置如下:

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
//对 ES6 模块文件不进行转化,以便使用 Tree Shaking、sideEffects 等
        "modules": false,
      }
    ]
  ],
  "plugins": []
}
```

注意:Babel 默认只转换新的 JavaScript 句法(syntax),而不转换新的 API,例如 Iterator、Generator、Set、Maps、Proxy、Reflect、Symbol、Promise 等全局对象,以及一些定义在全局对象上的方法(例如 Object.assign)都不会转码。

转译新的 API,需要借助 polyfill 方案去解决,可使用@babel/polyfill 或@babel/plugin-transform-runtime,二选一即可。

4. @babel/polyfill

本质上@babel/polyfill 是 core-js 库的别名,随着 core-js@3 的更新,@babel/polyfill 无法从 2 过渡到 3,所以@babel/polyfill 已经被放弃,可查看 corejs 3 的更新。

安装依赖包,命令如下:

```
npm i @babel/polyfill -D
```

.babelrc 文件需写入配置,而@babel/polyfill 不用写入配置,会根据 useBuiltIns 参数去决定如何被调用,配置如下:

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "useBuiltIns": "entry",
        "modules": false,
```

```
            "corejs": 2,     //新版本的@babel/polyfill 包含了 core-js@2 和 core-js@3 版本,
                             //所以需要声明版本,否则 Webpack 运行时会报 warning,此处暂时使用
                             //core-js@2 版本(末尾会附上@core-js@3 怎么用)
        }
      ]
    ]
  }
```

配置参数说明如下。

(1) Modules："amd"|"umd"|"systemjs"|"commonjs"|"cjs"|"auto"|false。

默认值为 auto。用来转换 ES6 的模块语法。如果使用 false,则不会对文件的模块语法进行转换。

如果要使用 Webpack 中的一些新特性,例如,Tree Shaking 和 sideEffects,就需要设置为 false,对 ES6 的模块文件不进行转换,因为这些特性只对 ES6 模块有效。

(2) useBuiltIns："usage"|"entry"|false,默认值为 false。

false：需要在 JS 代码的第一行主动通过 import'@babel/polyfill'语句将@babel/polyfill 整个包导入(不推荐,能覆盖到所有 API 的转译,但体积最大)。

entry：需要在 JS 代码的第一行主动通过 import'@babel/polyfill'语句将 browserslist 环境不支持的所有垫片导入(能够覆盖到'hello'.includes('h')这种句法,足够安全且代码体积不是特别大)。

usage：项目里不用主动通过 import 导入,会自动将代码里已用到的且 browserslist 环境不支持的垫片导入(但是检测不到'hello'.includes('h')这种句法,对这类原型链上的句法问题不会进行转译,书写代码需注意)。

(3) targets 用来配置需要支持的环境,不仅支持浏览器,还支持 Node。如果没有配置 targets 选项,就会读取项目中的 browserslist 配置项。

(4) loose 的默认值为 false,如果 preset-env 中包含的 Plugin 支持 loose 的设置,则可以通过这个字段来统一设置。

5. 解析 React JSX

JSX 是 React 框架中引入的一种 JavaScript XML 扩展语法,它能够支持在 JS 中编写类似 HTML 的标签,本质上来讲 JSX 就是 JS,所以需要 Babel 和 JSX 的插件 preset-react 支持解析。

安装 React 及@babel/preset-react,命令如下：

```
npm i react react-dom @babel/preset-react -D
```

配置解析 React 的 presets,配置如下：

```
module.exports = {
   entry: {},
```

```
    output: {},
    resolve: {
   //要解析的文件的扩展名
       extensions: [".js", ".jsx", ".json"],
    //解析目录时要使用的文件名
       mainFiles: ["index"],
    },
    module: {
       rules: [
          {
             test: /\.(js|jsx)$/,
             exclude: /(node_modules|bower_components)/,
             use: {
                loader: 'babel-loader',
                options: {
                   presets: ['@babel/preset-env', '@babel/preset-react']
                }
             }
          },
       ]
    },
};
```

6. 解析 CSS

传统上会在 HTML 文件中引入 CSS 代码，借助 webpack style-loader 和 css-loader 可以在.js 文件中引入 CSS 文件并让样式生效，如果需要使用预处理脚本，如 LESS，则需要安装 less-loader。

(1) css-loader：用于加载.css 文件并转换成 commonJS 对象。

(2) style-loader：将样式通过 style 标签插入 head 中。

安装依赖 css-loader 和 style-loader，命令如下：

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

Webpack 配置项添加 Loader 配置，其中由于 Loader 的执行顺序是从右向左执行的，所以会先进行 CSS 的样式解析后执行 style 标签的插入，配置如下：

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

less-loader 将.less 转换成.css。安装 less-loader 依赖并添加 Webpack 配置,配置如下:

```
npm i less less-loader -D
{
    test:/.less$/,
    use: [
        'style-loader',
        'css-loader',
        'less-loader'
]
}
```

7. 解析图片和字体

Webpack 提供了两个 Loader 来处理二进制格式的文件,如图片和字体等,url-loader 允许有条件地将文件转换为内联的 base-64 URL(当文件小于给定的阈值时),这会减少小文件的 HTTP 请求数。如果文件大于该阈值,则会自动交给 file-loader 处理。

1) file-loader

file-loader 用于处理文件及字体。安装 file-loader 依赖并配置,配置如下:

```
#安装 file-loader
npm i file-loader -D
#webapck.config 配置
{
  test: /\.(png|svg|jpg|jpeg|gif)$/,
  use: 'file-loader'},{
   test:/\.(woff|woff2|eot|ttf|otf|svg)/,
use:'file-loader'
}
```

2) url-loader

url-loader 也可以处理文件及字体,对比 file-loader 的优势是可以通过配置,将小资源自动转换为 base64。

安装 url-loader 依赖并配置 Webpack,配置如下:

```
{
    test: /\.(png|svg|jpg|jpeg|gif)$/,
     use: [
          {
              loader:'url-loader',
              options: {
                 limit:10240
              }
          }
     ]
}
```

8. 常见插件(Plugin)

插件的目的是为了增强 Webpack 的自动化能力,Plugin 可解决其他自动化工作,如清除 dist 目录、将静态文件复制至输出代码、压缩输出代码,常见的场景如下:

(1) 实现自动在打包之前清除 dist 目录(上次的打包结果)。

(2) 自动生成应用所需要的 HTML 文件。

(3) 根据不同环境为代码注入类似 API 地址这种可能变化的部分。

(4) 将不需要参与打包的资源文件复制到输出目录。

(5) 压缩 Webpack 打包完成后输出的文件。

(6) 自动将打包结果发布到服务器以实现自动部署。

9. 文件指纹

文件指纹的作用:

(1) 在前端发布体系中,为了实现增量发布,一般会对静态资源加上 md5 文件后缀,保证每次发布的文件都没有缓存,同时对于未修改的文件不会受发布的影响,最大限度地利用缓存。

(2) 简单地来讲"文件指纹"的应用场景是在项目打包时使用(上线),在项目开发阶段用不到。

这里简单介绍一下 3 种不同的 hash 表示方式。

(1) hash:与整个项目的构建相关,当有文件修改时,整个项目构建的 hash 值就会更新。

(2) chunkhash:和 Webpack 打包的 chunk 相关,不同的 entry 会生成不同的 chunkhash,一般用于.js 文件的打包。

(3) contenthash:根据文件内容来定义 hash,如果文件内容不变,则 contenthash 不变。例如.css 文件的打包,当修改了.js 或.html 文件但没有修改引入的.css 样式时,文件不需要生成新的 hash 值,所以可适用于.css 文件的打包。

注意:文件指纹不能和热更新一起使用。

1).js 文件指纹设置:chunkhash

代码如下:

```
#webpack.dev.js
module.export = {
    entry: {
         index: './src/demo.js',
         search: './src/search.js'
       },
       output: {
          path: path.resolve(__dirname,'dist'),
          filename: '[name][chunkhash:8].js'
       },
}
```

2）.css 文件指纹：contenthash

由于上面方式通过 style 标签将 css 插入 head 中并没有生成单独的.css 文件，因此可以通过 min-css-extract-plugin 插件将 css 提取成单独的.css 文件，并添加文件指纹。

安装依赖 mini-css-extract-plugin，命令如下：

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

配置.css 文件指纹，配置如下：

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
module.export = {
   module: {
      rules: [
         {
            test:/\.css$/,
            use: [
               MiniCssExtractPlugin.loader,
               'css-loader',
            ]
         },
      ]
   },
    plugins: [
      new MiniCssExtractPlugin({
        filename: '[name][contenthash:8].css'
      })
    ]
}
```

3）图片文件指纹设置：hash

其中，hash 对应的是文件内容的 hash 值，默认由 md5 生成，不同于前面所讲的 hash 值，配置如下：

```
module.export = {
   module:{
      rules: [
         {
               test: /\.(png|svg|jpg|jpeg|gif)$/,
               use: [{
                 loader:'file-loader',
                 options: {
                    name: 'img/[name][hash:8].[ext]'
                 }
               }],
```

```
            }
        ]
}
}
```

代码压缩，这里介绍两个插件：.css 文件压缩和.html 文件压缩。

（1）.css 文件压缩：optimize-css-assets-webpack-plugin。

安装 optimize-css-assets-webpack-plugin 和预处理器 cssnano，命令如下：

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

配置 Webpack，配置如下：

```
const OptimizeCssAssetsPlugin = require('optimize-css-assets-webpack-plugin')
module.export = {
   plugins: [
       new OptimizeCssAssetsPlugin({
          assetNameRegExp: /\.css$/g,
          cssProcessor: require('cssnano')
       })
]
}
```

（2）.html 文件压缩：html-webpack-plugin。

安装 html-webpack-plugin 插件，命令如下：

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

配置 Webpack，配置如下：

```
const HtmlWebpackPlugin = require('html-webpack-plugin')
module.export = {
   plugins: [
       new HtmlWebpackPlugin({
          template: path.join(__dirname,'src/search.html'),//使用模板
          filename: 'search.html',                          //打包后的文件名
          chunks: ['search'],                               //打包后需要使用的 chunk(文件)
          inject: true,                                     //默认注入所有静态资源
          minify: {
              html5:true,
              collapsableWhitespace: true,
              preserveLineBreaks: false,
              minifyCSS: true,
              minifyJS: true,
```

```
                removeComments: false
            }
        }),
    ]
    }
```

10. 跨应用代码共享(Module Federation)

Module Federation 使 JavaScript 应用得以在客户端或服务器上动态运行另一个包的代码。Module Federation 主要用来解决多个应用之间代码共享的问题,可以更加优雅地实现跨应用的代码共享。

1) 三个概念

首先,要理解三个重要的概念,如表 3-3 所示。

表 3-3 三个重要的概念

模块名称	属性描述
webpack	一个独立项目通过 Webpack 打包编译而产生资源包
remote	一个暴露模块供其他 Webpack 构建消费的 Webpack 构建
host	一个消费其他 remote 模块的 Webpack 构建

一个 Webpack 构建可以是 remote(服务的提供方),也可以是 host(服务的消费方),还可以同时扮演服务提供者和服务消费者的角色,这完全看项目的架构。

2) host 与 remote 两个角色的依赖关系

任何一个 Webpack 构建既可以作为 host 消费方,也可以作为 remote 提供方,区别在于职责和 Webpack 配置的不同。

3) 案例讲解

一共有三个微应用:lib-app、component-app、main-app,角色分别如表 3-4 所示。

表 3-4 三个微应用的关系

模块名称	属性描述
lib-app	作为 remote,暴露了两个模块 react 和 react-dom
component-app	作为 remote 和 host,依赖 lib-app 暴露了一些组件供 main-app 消费
main-app	作为 host,依赖 lib-app 和 component-app

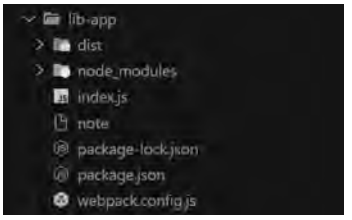

图 3-10 lib-app 模块目录结构

下面分别创建三个微应用的项目:

lib-app 模块提供其他模块所依赖的核心库,如 lib-app 对外提供 react 和 react-dom 两个类库模块。

步骤 1:创建 lib-app 模块的项目,目录结构如图 3-10 所示。

该模块需要安装两个类库,即 react 和 react-dom,命

令如下：

```
#安装核心模块
npm i react react-dom -S
#安装开发依赖模块
npm install concurrently serve webpack webpack-cli -D
```

步骤 2：配置 Webpack 编译配置，配置如下：

```
const {ModuleFederationPlugin} = require('webpack').container
const path = require('path');
module.exports = {
   entry: "./index.js",
   mode: "development",
   devtool:"hidden-source-map",
   output: {
      publicPath: "http://localhost:3000/",
      clean:true
   },
   module: {
   },
   plugins: [
      new ModuleFederationPlugin({
         name: "lib_app",
         filename: "remoteEntry.js",
         exposes: {
            "./react":"react",
            "./react-dom":"react-dom"
         }
      })
   ],
};
```

这里通过 ModuleFederationPlugin 插件设置暴露的库，详细信息如表 3-5 所示。

表 3-5　ModuleFederationPlugin 属性

属性名称	属 性 描 述
name	必选，唯一 ID，作为输出的模块名，使用时通过 ${name}/${expose}的方式使用
library	必选，这里的 library.name 作为 umd 的 library.name
remotes	可选，表示作为 host 时，去消费哪些 remote
exposes	可选，表示作为 remote 时，export 哪些属性被消费
shared	可选，优先用 host 的依赖，如果 host 没有，则再用自己的

步骤 3：配置 package.json 文件中的运行脚本，脚本如下：

```
"scripts": {
  "webpack": "webpack --watch",
  "serve": "serve dist -p 3000",
  "start": "concurrently \"npm run webpack\" \"npm run serve\""
},
```

步骤 4：除去生成的 map 文件，有 4 个文件，如图 3-11 所示，输出具体的编译文件如下：

```
main.js
remoteEntry.js
react_index.js
react-dom_index.js
```

步骤 5：在命令行中，输入 npm serve 命令启动 lib-app 项目，启动后在 3000 端口浏览。

```
http://localhost:3000/
```

component-app 模块对外提供组件库，如 Button、Dialog、Logo 基础组件。

步骤 1：创建 component-app 模块，目录结构如图 3-12 所示。

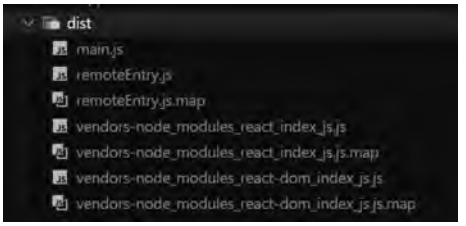

图 3-11　lib-app 模块打包输出目录

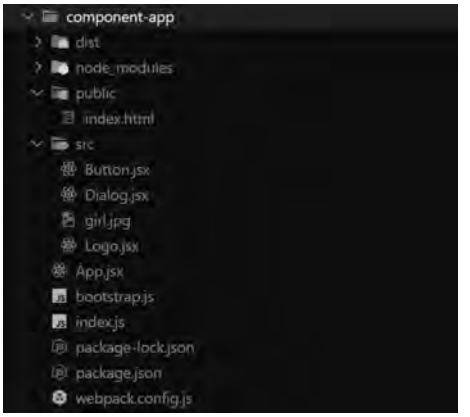

图 3-12　component-app 模块目录

步骤 2：创建 Button、Dialog、Logo 三个 React 组件。

Button.jsx 组件，代码如下：

```
//Button.jsx
import React from 'lib-app/react';
export default function(){
   return <button style={{color: "#fff",backgroundColor: "#f00",borderColor: "#fcc"}}>
按钮组件</button>
}
```

Dialog.jsx 组件，代码如下：

```
//Dialog.jsx
import React from 'lib-app/react';
export default class Dialog extends React.Component {
   constructor(props) {
      super(props);
   }
   render() {
      if(this.props.visible){
         return (
            <div style = {{ position:" fixed ", left: 0, right: 0, top: 0, bottom: 0,
backgroundColor:"rgba(0,0,0,.3)"}}>
               <button onClick = {( ) = > this. props. switchVisible ( false )} style =
{{position:"absolute",top:"10px",right:"10px"}}> X </button >
               <div style = {{ marginTop:"20 % ",textAlign:"center"}}>
                  <h1 >
                     输入你的昵称：
                  </h1 >
                  < input style = {{fontSize:"18px",lineHeight:2}} type = "text" />
               </div >

            </div >
            );
      }else{
         return null;
      }
   }
}
```

Logo.jsx 组件，代码如下：

```
//Logo.jsx
import React from 'lib-app/react';
import pictureData from './girl.jpg'
export default function(){
   return < img src = {pictureData} style = {{width:"100px",borderRadius:"10px"}}/>
}
```

步骤 3：需要以异步的方式导入 index.js 模块，所以这里创建了 bootstrap.js 模块，在 index.js 文件中通过 import 导入 bootstrap.js 文件，代码如下：

```
#Bootstrap.js
import App from "./App";
import ReactDOM from 'lib-app/react-dom';
import React from 'lib-app/react'
ReactDOM.render(< App />, document.getElementById("app"));
```

在 index.js 文件中导入 bootstrap.js 文件，代码如下：

```
import("./bootstrap.js")
```

步骤 4：webpack.config 文件的配置如下：

```
const {ModuleFederationPlugin} = require('webpack').container;
const HtmlWebpackPlugin = require("html-webpack-plugin");
const path = require('path');
module.exports = {
   entry: "./index.js",
   mode: "development",
   devtool:"hidden-source-map",
   output: {
      publicPath: "http://localhost:3001/",
      clean:true
   },
   resolve:{
      extensions: ['.jsx', '.js', '.json','.css','.scss','.jpg','jpeg','png',],
    },
   module: {
      rules: [
         {
           test:/\.(jpg|png|gif|jpeg)$/,
           loader:'url-loader'
         },
         {
           test: /\.css$/i,
           use: ["style-loader", "css-loader"],
         },
         {
           test: /\.jsx?$/,
           loader: "babel-loader",
           exclude: /node_modules/,
           options: {
            presets: ["@babel/preset-react"],
           },
         },
       ],
    },
   plugins: [
      new ModuleFederationPlugin({
         name: "component_app",
         filename: "remoteEntry.js",
         exposes: {
           "./Button":"./src/Button.jsx",
```

```
            "./Dialog":"./src/Dialog.jsx",
            "./Logo":"./src/Logo.jsx"
          },
          remotes:{
             "lib-app":"lib_app@http://localhost:3000/remoteEntry.js"
          }
        }),
        new HtmlWebpackPlugin({
          template: "./public/index.html",
         })
    ],
};
```

步骤 5：启动模块。

在该子项目下运行 npm run start 命令打开浏览器：localhost：3001，可以看到组件正常工作。

main-app 模块为三个模块中的主模块，该模块依赖 component-app 模块的基础组件，同时也依赖 lib-app 模块。

步骤 1：创建 main-app 模块，目录结构如图 3-13 所示。

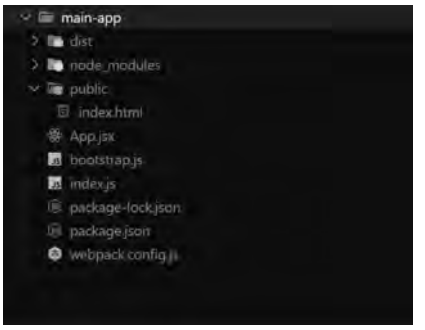

图 3-13　创建 main-app 模块

步骤 2：创建 App.jsx 文件，编写界面，代码如下：

```
#App.jsx
import React from 'lib-app/react';
import Button from 'component-app/Button'
import Dialog from 'component-app/Dialog'
import Logo from 'component-app/Logo'

export default class App extends React.Component{
   constructor(props) {
     super(props)
     this.state = {
       dialogVisible:false
     }
     this.handleClick = this.handleClick.bind(this);
     this.HanldeSwitchVisible = this.HanldeSwitchVisible.bind(this);
   }
   handleClick(ev){
     console.log(ev);
     this.setState({
       dialogVisible:true
```

```
      })
    }
    HanldeSwitchVisible(visible){
      this.setState({
        dialogVisible:visible
      })
    }
    render(){
      return (<div>
        <Logo></Logo>
       <h4>
        Buttons:
       </h4>
       <Button type="primary"/>
       <Button type="warning"/>
       <h4>
        Dialog:
       </h4>
       <button onClick={this.handleClick}>打开对话框</button>
       <Dialog switchVisible={this.HanldeSwitchVisible} visible={this.state.
dialogVisible}/>
      </div>)
    }
 }
```

步骤 3：配置 Webpack，配置如下：

```
const {ModuleFederationPlugin} = require('webpack').container
const HtmlWebpackPlugin = require("html-webpack-plugin");
const path = require('path');
module.exports = {
    entry: "./index.js",
    mode: "development",
    devtool:"hidden-source-map",
    output: {
        publicPath: "http://localhost:3002/",
        clean:true
    },
    resolve:{
        extensions: ['.jsx', '.js', '.json','.css','.scss','.jpg','jpeg','png',],
     },
    module: {
        rules: [
          {
            test:/\.(jpg|png|gif|jpeg)$/,
```

```
            loader:'url-loader'
          },
          {
            test: /\.jsx?$/,
            loader: "babel-loader",
            exclude: /node_modules/,
            options: {
              presets: ["@babel/preset-react"],
            },
          },
        ],
      },
    plugins: [
        new ModuleFederationPlugin({
            name: "main_app",
            remotes:{
                "lib-app":"lib_app@http://localhost:3000/remoteEntry.js",
                "component-app":"component_app@http://localhost:3001/remoteEntry.js"
            },
        }),
        new HtmlWebpackPlugin({
          template: "./public/index.html",
        })
    ],
};
```

步骤 4：启动编译器后，通过 localhost：3002 端口查看，效果如图 3-14 所示。

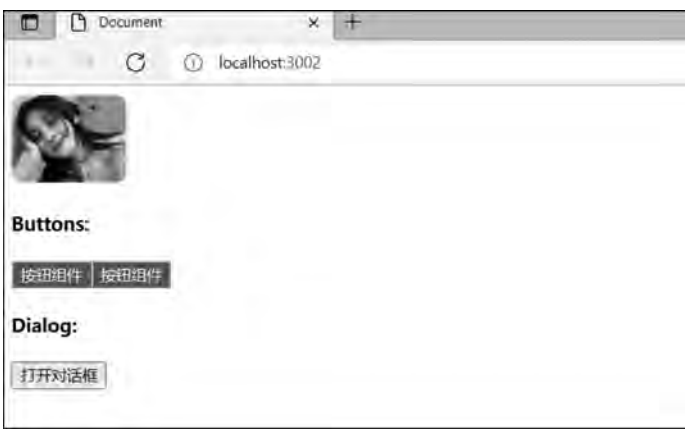

图 3-14　跨应用代码共享效果

11. 开发运行构建

使用 Webpack 的 Webpack-dev-server 插件可以帮助开发者快速搭建一个代码运行环

境，Webpack-dev-server 提供的热更新功能极大地方便了代码编译后进行预览。

1）文件监听：watch

在 Webpack-cli 中提供了 watch 工作模式，这种模式下项目中的文件会被监视，一旦这些文件发生变化就会自动重新运行打包任务。

Webpack 开启监听模式有以下两种方式：

（1）启动 Webpack 命令时带上--watch 参数。

（2）在配置 webpack.config.js 文件中设置 watch：true。

缺点是每次都需要手动刷新浏览器，需要自己使用一些 http 服务，例如使用 http-server 轮询判断文件的最后编辑时间是否发生变化，一开始有个文件的修改时间，这个修改时间先存储起来，下次再有修改时就会和上次修改时间进行比对，发现不一致时不会立即告诉监听者，而是把文件缓存起来，等待一段时间，等待期间内如果有其他变化，则会把变化列表一起构建，并生成到 bundle 文件夹。

可通过 Webpack 添加配置或者 CLI 添加配置的方式开启监听模式，该方式在源码变化时需要每次手动刷新浏览器。

Webpack 配置的代码如下：

```
module.export = {
watch: true
}
```

除了可通过 watch 参数的配置方式开启监听外，也可通过定制 watch 模式选项的形式 watchOptions 来定制监听配置，配置如下：

```
module.export = {
   watch: true,
   //只有开启了监听模式才有效
   watchOptions: {
      ignored: /node_modules/,          //默认为空，设置不监听的文件或文件夹
      aggregateTimeout: 300,            //默认为 300ms，即监听变化后需要等待的执行时间
      poll:1000                         //默认为 1000ms，通过轮询的方式询问系统指定文件
                                        //是否发生变化
}
}
```

Webpack-dev-server 是 Webpack 官方推出的一个开发工具，它提供了一个开发服务器，并且集成了自动编译和自动刷新浏览器等一系列功能的安装指令，如图 3-15 所示。

Webpack Compiler 将 JavaScript 编译成输出的 bundle.js 文件。

HMR Server 将热更新的文件输出到 HMR Runtime。

Bundle Server 通过提供服务器的形式，提供浏览器对文件的访问。

HMR Runtime 在开发打包阶段将构建输出文件注入浏览器，更新文件的变化。

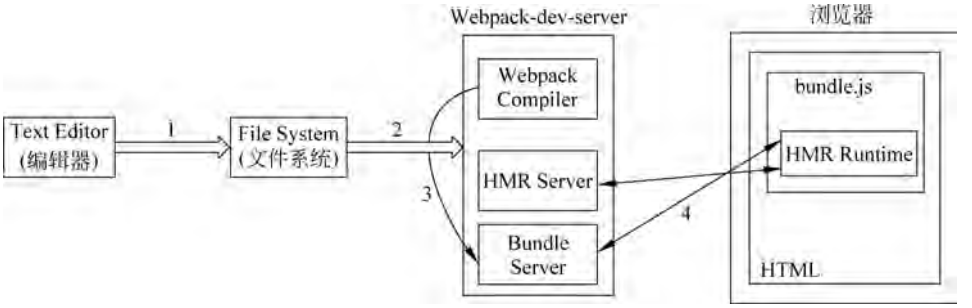

图 3-15　热更新的大概流程

当启动 Webpack-dev-server 阶段时，将源码在文件系统进行编译，通过 Webpack Compiler 编译器打包，并将编译好的文件提交给 Bundle Server 服务器，Bundle Server 即可以服务器的方式供浏览器访问。

当监听到源码发生变化时，经过 Webpack Compiler 的编译后提交给 HMR Server，一般通过 websocket 实现监听源码的变化，并通过 JSON 数据的格式通知 HMR Runtime，HMR Runtime 对 bundle.js 文件进行改变并刷新浏览器。

相比于 watch 不能自动刷新浏览器，Webpack-dev-server 的优势就明显了。Webpack-dev-server 构建的内容会存放在内存中，所以构建速度更快，并且可自动地实现浏览器的自动识别并做出变化，其中 Webpack-dev-server 需要配合 Webpack 内置的 HotModuleReplacementPlugin 插件一起使用。

安装依赖 Webpack-dev-server 并配置启动项，命令如下：

```
#安装
npm i webpack-dev-server -D
//package.json
"scripts": {
"dev": "webpack-dev-server --open"
}
```

配置 Webpack，其中 Webpack-dev-server 一般在开发环境中使用，所以需将 mode 模式设置为 development，配置如下：

```
const webpack = require('webpack')
plugins: [
    new webpack.HotModuleReplacementPlugin()
  ],
  devServer: {
     contentBase: path.join(__dirname,'dist'),     //监听 dist 文件夹下的内容
     hot: true                                     //启动热更新
}
```

2）清理构建目录：clean-webpack-plugin

由于每次构建项目前并不会自动地清理目录，会造成输出文件越来越多。这时就得手动清理输出目录的文件。

借助 clean-webpack-plugin 插件清除构建目录，默认会执行删除 output 值的输出目录。

安装 clean-webpack-plugin 插件并配置，命令如下：

```
npm i clean-webpack-plugin -D
```

在 webpack.config.js 文件中配置，代码如下：

```
const { CleanWebpackPlugin } = require('clean-webpack-plugin')
module.export = {
   plugins: [
      new CleanWebpackPlugin()
]
}
```

3.2.4 Webpack 进阶

可以通过 Webpack 插件进行打包内容分析，优化编译速度，减少构建包体积，同时通过接口编写自己的 Loader 和 Plugin 插件。

1. 项目分析

通过 Webpack-bundle-analyzer 可以看到项目各模块的大小，对各模块可以按需优化。

该插件通过读取输出文件夹（通常是 dist）中的 stats.json 文件，把该文件可视化展现。便于直观地比较各个包文件的大小，以达到优化性能的目的。

安装 Webpack-bundle-analyzer，命令如下：

```
npm install --save-dev webpack-bundle-analyzer
```

在 webpack.config.js 文件中导入 Webpack-bundle-analyzer 模块，在 plugins 数组中实例化插件，配置如下：

```
const path = require("path")
var BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin

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

```
    },
    plugins:[
        new BundleAnalyzerPlugin()
    ]
}
```

2. 编译阶段提速

编译模块阶段的效率提升，下面的操作都是在 Webpack 编译阶段实现的。

1）IgnorePlugin：忽略第三方包指定目录

IgnorePlugin 是 Webpack 的内置插件，其作用是忽略第三方包指定目录。

有的依赖包，除了项目所需的模块内容外，还会附带一些多余的模块，例如 moment 模块会将所有本地化内容和核心功能一起打包。

下面案例通过配置的 Webpack-bundle-analyzer 来查看使用 IgnorePlugin 后对 moment 模块的影响，Webpack 的配置如下：

```
const webpack = require('webpack')
//webpack.config.js
module.exports = {
    //...
    plugins: [
        //忽略 moment 模块下的 ./locale 目录
        new webpack.IgnorePlugin({
            resourceRegExp: /^\.\/locale$/,
            contextRegExp: /moment$/,
        }),
]
}
```

2）DllPlugin 和 DllReferencePlugin 提高构建速度

在使用 Webpack 进行打包时，对于依赖的第三方库，例如 React、Redux 等不会修改的依赖，可以让它和自己编写的代码分开打包，这样做的好处是每次更改本地代码文件时，Webpack 只需打包项目本身的文件代码，而不会再去编译第三方库，第三方库在第一次打包时只打包一次，以后只要不升级第三方包，Webpack 就不会对这些库打包，这样可以快速地提高打包速度，因此为了解决这个问题，DllPlugin 和 DllReferencePlugin 插件就产生了。

DLLPlugin 能把第三方库与自己的代码分离开，并且每次文件更改时，它只会打包该项目自身的代码，所以打包速度会更快。

DLLPlugin 插件在一个额外独立的 Webpack 设置中创建一个只有 dll 的 bundle，也就是说，在项目根目录下除了有 webpack.config.js 文件，还会新建一个 webpack.dll.config.js 文件。webpack.dll.config.js 的作用是除了把所有的第三方库依赖打包到一个 bundle 的 dll 文件里面，还会生成一个名为 manifest.json 的文件。该 manifest.json 文件的作用是

让 DllReferencePlugin 映射到相关的依赖上去。

DllReferencePlugin 插件在 webpack.config.js 文件中使用，该插件的作用是把刚刚在 webpack.dll.config.js 文件中打包生成的 dll 文件引用到需要的预编译的依赖上来。什么意思呢？就是说在 webpack.dll.config.js 文件中打包后会生成 vendor.dll.js 文件和 vendor-manifest.json 文件，vendor.dll.js 文件包含所有的第三方库文件，vendor-manifest.json 文件会包含所有库代码的一个索引，当在使用 webpack.config.js 文件打包 DllReferencePlugin 插件时，会使用该 DllReferencePlugin 插件读取 vendor-manifest.json 文件，看一看是否有该第三方库。vendor-manifest.json 文件只有一个第三方库的一个映射。

第一次使用 webpack.dll.config.js 文件时会对第三方库打包，打包完成后就不会再打包它了，然后每次运行 webpack.config.js 文件时，都会打包项目中本身的文件代码，当需要使用第三方依赖时，会使用 DllReferencePlugin 插件去读取第三方依赖库，所以说它的打包速度会得到一个很大的提升。

在项目中使用 DllPlugin 和 DllReferencePlugin，其使用步骤如下。

在使用之前，首先看一下项目现在的整个目录架构，架构如下：

```
Demo                                    # 工程名
|   |--- dist                           # 打包后生成的目录文件
|   |--- node_modules                   # 所有的依赖包
|   |--- js                             # 存放所有的 js 文件
|   |   |-- main.js                     # js 入口文件
|   |--- webpack.config.js              # Webpack 配置文件
|   |--- webpack.dll.config.js          # 打包第三方依赖的库文件
|   |--- index.html                     # html 文件
|   |--- package.json
```

因此需要在项目根目录下创建一个 webpack.dll.config.js 文件，配置的代码如下：

```
const path = require('path');
const webpack = require('webpack');

module.exports = {
    entry: {
        //library 中配置要处理的第三方库的名称
        library: [
            'react',
            'react-dom'
        ]
    },
    output: {
```

```
        filename: '[name]_[chunkhash].dll.js',       //library.dll.js 文件中暴露出的库的名称
        path: path.join(__dirname, 'build/library'),//打包后文件输出的位置
        library: '[name]_[hash]'                     //库暴露出来的名字,可以参考打包组件和
                                                     //基础库
    },
    plugins: [
        new webpack.DllPlugin({
            name: '[name]_[hash]',                   //生成一个文件映射 json 名字
            path: path.join(__dirname, 'build/library/[name].json')          //保存的位置
        })
    ]};
```

切换到 webpack.config.js 配置,引入文件,代码如下:

```
//引入 DllReferencePlugin
const DllReferencePlugin = require('webpack/lib/DllReferencePlugin');
```

然后在插件中使用该插件,代码如下:

```
plugins: [
    new webpack.DllReferencePlugin({
      context: __dirname,
      manifest: require('./build/library/')
    })
 ],
```

最后一步就是构建代码了,先生成第三方库文件,运行命令如下:

```
webpack --config webpack.dll.config.js
```

3. 优化构建体积

1）摇树优化(Tree Shaking)

Webpack 4.0 后通过开启 mode: production 即可开启摇树优化功能。

Tree Shaking 摇掉代码中未引用部分(dead-code),production 模式下会自动在使用 Tree Shaking 打包时除去未引用的代码,其作用是优化项目代码。

2）删除无效的 CSS

PurgeCSS 是一个能够通过字符串对比来决定移除不需要的 CSS 的工具。PurgeCSS 通过分析内容和 CSS 文件,首先将 CSS 文件中使用的选择器与内容文件中的选择器进行匹配,然后会从 CSS 中删除未使用的选择器,从而生成更小的 CSS 文件。对于 PurgeCSS 的配置因项目的不同而不同,它不仅可以作为 Webpack 的插件,还可以作为 postcss 的插件。一般与 glob、glob-all 配合使用。

安装 purgecss-webpack-plugin,命令如下:

```
npm i purgecss - webpack - plugin - D
```

配置 Webpack，配置如下：

```
//webpack.prod.js
const glob = require('glob')
const MiniCssExtractPlugin = require('mini - css - extract - plugin');
const PurgecssPlugin = require('purgecss - webpack - plugin')
const PATHS = {
   src: path.join(__dirname,'src')}
module.exports = {
   module:{
      rules: [
         {
             test: /.css $ /,
             use: [
               MiniCssExtractPlugin.loader,
               'css - loader'
             ]
         },
      ]
   },
   plugins: [
       new MiniCssExtractPlugin({
           filename: '[name]_[contenthash:8].css'
       }),
       new PurgecssPlugin({
           path: glob.sync(' $ {PATHS.src}/ ** / * ', {nodir:true})     //绝对路径
       }),
   ]}
```

4. 编写自定义 Loader

Loader 是一种打包的方案。可以定义一种规则，告诉 Webpack 当它遇到某种格式的文件后，去求助相应的 Loader。有些时候需要一些特殊的处理方式，这就需要自定义一些 Loader。

一个简单的 Loader 通过编写一个简单的 JavaScript 模块，并将模块导出即可。接收一个 source 当前源码，并返回新的源码即可。

Loader 分为同步 Loader 和异步 Loader。如果单个处理，则可以直接使用同步模式处理后直接返回，但如果需要多个处理，就必须在异步 Loader 中使用 this.async()告诉当前的上下文 context，这是一个异步的 Loader，需要 Loader Runner 等待异步处理的结果，在异步处理完之后再调用 this.callback()传递给下一个 Loader 执行。

编写同步 Loader，代码如下：

```
//同步 Loader
module.exports = function(source, sourceMap, meta) {
  return source
}
```

对于异步 Loader，使用 this.async 获取 callback()函数。

编写异步 Loader，代码如下：

```
//异步 Loader
module.exports = function(source) {
  const callback = this.async();
  setTimeout(() => {
    const output = source.replace(/World/g, 'Webpack5');
    callback(null, output);
  }, 1000);
}
```

执行构建，Webpack 会等待一秒，然后输出构建内容，通过 Node.js 执行构建后的文件，输出如下：

```
Webpack5
```

1）编写一个简单的 Loader

Webpack 默认只能识别 JavaScript 模块，在实际项目中会有.css .less .scss .txt .jpg .vue 等文件，这些都是 Webpack 无法直接识别打包的文件，都需要使用 Loader 来直接或者间接地进行转换成可以供 Webpack 识别的 JavaScript 文件。

创建一个简单的 Loader，命名为 hello-loader。hello-loader 的作用是让 Webpack 识别.hello 扩展名的模块，并进行转换打包。

第 1 步：创建目录 loader-demo。创建文件如图 3-16 所示，这里创建一个 test.hello 自定义文件，hello-loader 需要能够识别该模块，并进行打包。

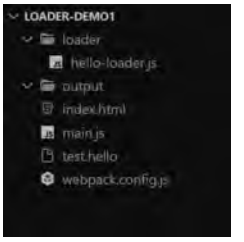

图 3-16　自定义 loader 目录

第 2 步：编写 hello-loader，代码如下：

```
module.exports = function loader(source) {
   source = "hello loader!"
   return 'export default ${ JSON.stringify(source) }'; //返回值
};
```

第 3 步：编写 main.js 打包入口代码。main.js 是打包的入口文件，因为要尽可能简单，所以这个文件只做一件事，即加载.hello 文件，并显示到页面，代码如下：

```
#main.js
import data from './test.hello';
function test() {
   let element = document.getElementById('app');
   console.log(data);
   element.innerText = data;
}
test();
```

第 4 步：编写 index.html 文件。main.js 文件的代码逻辑很简单，就是获取页面中的一个 id 为 app 的元素，并将.hello 中的值显示在元素中，代码如下：

```
#index.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>自定义 hello-loader</title>
</head>
<body>
    <div id="app">
        <script src="./output/bundle.js"></script>
    </div>
</body>
</html>
```

第 5 步：这里暂时不对 test.hello 文件进行特殊处理，所以 test.hello 文件里面的代码可以随便写，输出如下：

```
//里面随便写
```

第 6 步：在 webpack.config.js 文件中配置规则，完整的打包配置如下：

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

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

```
            {
                test: /\.hello$/,                    //需要加载的文件类型,正则匹配
                use: [path.resolve(__dirname, './loader/hello-loader.js'),]
                                                     //我们的 loader 文件
            }
        ]
    }
}
```

第 7 步：编译打包。在项目目录下直接执行 webpack 命令，如图 3-17 所示，打包的文件输出在 output 目录下，输出如下：

```
webpack
```

```
asset bundle.js 4.48 KiB [emitted] (name: main)
runtime modules 670 bytes 3 modules
cacheable modules 412 bytes
  ./main.js 380 bytes [built] [code generated]
  ./test.hello 32 bytes [built] [code generated]
webpack 5.37.1 compiled successfully in 78 ms
```

图 3-17　编译自定义 Loader

第 8 步：在浏览器中运行 index.html 文件，效果如图 3-18 所示。

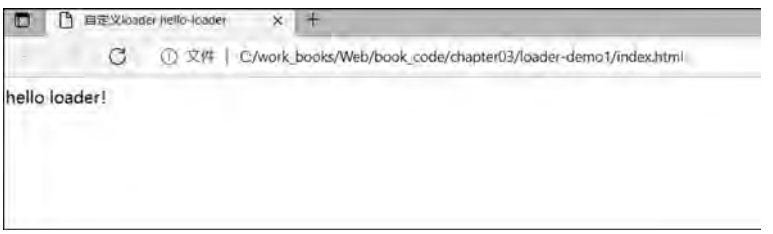

图 3-18　Loader 的运行效果

2）编写一个自定义的 less-loader

下面自定义一个 less-loader 和 style-loader。将编写的 less 经过这两个 Loader 处理之后使用 style 标签插入页面的 head 标签内。

为了测试自定义 Loader，创建测试目录，结构如图 3-19 所示。

第 1 步：定义一个.less 文件，这里命名为 index.less，代码如下：

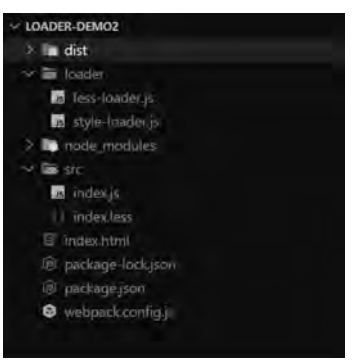

图 3-19　自定义 style-loadert 和 less-loader

```
@red:red;
@yellow:yellow;
@baseSize:20px;
body{
 background-color: @red;
 color:@yellow;
 font-size: @baseSize;
}
```

第 2 步：新建一个 less-loader.js 文件，用于处理.less 文件，代码如下：

```
let less = require('less');
function loader(source) {
   const callback = this.async();
   less.render(source)
      .then((output) =>{
         callback(null, output.css);
      }, (err) =>{
         //handler err
      })
}
module.exports = loader;
```

这里处理的业务很简单，就是获得原始的.less 文件的内容，将.less 文件通过 less.render 编译为.css 文件并传递给下一个 Loader 即可。

注意：这里需要安装 less 模块，命令为 npm install less -D。

关键的代码如下：

this.async 告诉当前上下文这是一个异步的 Loader，需要 loader runner 等待 less.render 异步处理的结果。

less.render 接收 less 源码，并返回一个 promise，在返回的 promise 中等待 less.render 处理完.less 文件之后，使用 callback 将处理的结果返给下一个 Loader。

第 3 步：新建一个 style-loader.js 文件，代码如下：

```
//Webpack 自定义 Loader
function loader(source) {
   source = JSON.stringify(source);
   const root = process.cwd();
   const resourcePath = this.resource;
   const origin = resourcePath.replace(root, '');
   let style = '
      let style = document.createElement('style');
      style.innerHTML = ${source};
```

```
        style.setAttribute('data-origin', '${origin}');
        document.head.appendChild(style);
    ';
    return style
}
module.exports = loader
```

使用 JSON.stringify 将接收到的.css 文件变为一个可编辑字符串。process.cwd 文件用于获取当前工程根目录。this.resource 通过 this 上的该属性可获取当前处理的源文件的绝对路径。

之后创建一个 style 标签，将编译完之后的.css 代码插入 style 标签内，并自定义一个 data-origin 属性，用来标记当前文件在工程中的相对路径。

最后返回一个可执行的 JS 字符串给 bundle.js。

第 4 步：新建 index.html 文件，引用 bundle.js 文件，代码如下：

```
<!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>自定义 less-loader</title>
    <script src="./dist/bundle.js"></script>
</head>
<body>
</body>
</html>
```

第 5 步：编译并运行。在项目目录下直接执行 webpack 命令，如图 3-20 所示，打包的文件输出在 dist 目录下。

```
webpack
```

```
asset bundle.js 2.73 KiB [emitted] (name: main)
./src/index.js 54 bytes [built] [code generated]
./src/index.less 266 bytes [built] [code generated]
webpack 5.37.1 compiled successfully in 193 ms
```

图 3-20　编译输出

通过浏览器查看效果，如图 3-21 所示。

5. 编写自定义插件 Plugin

插件件随 Webpack 构建的初始化到最后文件生成的整个生命周期，插件的目的是解决

图 3-21 使用自定义 style-loader 的效果

Loader 无法实现的其他事情。另外，插件没有像 Loader 那样的独立运行环境，所以插件只能在 Webpack 里面运行。

Webpack 通过 Plugin 机制让其更加灵活，以适应各种应用场景。在 Webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。

(1) 创建一个最基础的 Plugin，代码如下：

```
//采用 ES6
class ExamplePlugin {
  constructor(option) {
    this.option = option
  }
  apply(compiler) {}
}
```

以上就是一个最基本的 Plugin 结构。Webpack Plugin 最为核心的便是这个 apply()方法。

Webpack 执行时，首先会生成插件的实例对象，之后会调用插件上的 apply()方法，并将 compiler 对象(Webpack 实例对象，包含 Webpack 的各种配置信息等)作为参数传递给 apply()方法。

之后便可以在 apply()方法中使用 compiler 对象去监听 Webpack 在不同时刻触发的各种事件进行想要的操作了。

接下来看一个简单的示例，代码如下：

```
class plugin1 {
    constructor(option) {
      this.option = option
      console.log(option.name + '初始化')
    }
    apply(compiler) {
```

```
        console.log(this.option.name + 'apply 被调用')
        //在 Webpack 的 emit 生命周期上添加一种方法
        compiler.hooks.emit.tap('plugin1', (compilation) => {
          console.log('生成资源到 output 目录之前执行的生命周期')
        })
      }
    }

    class plugin2 {
      constructor(option) {
        this.option = option
        console.log(option.name + '初始化')
      }
      apply(compiler) {
        console.log(this.option.name + 'apply 被调用')

        //在 Webpack 的 afterPlugins 生命周期上添加一种方法
        compiler.hooks.afterPlugins.tap('plugin2', (compilation) => {
          console.log('Webpack 设置完初始插件之后执行的生命周期')
        })
      }
    }
    module.exports = {
        plugin1,
        plugin2
    }
```

定义 Webpack 配置文件，代码如下：

```
const path = require("path");
const {plugin1,plugin2} = require("./plugins/plugin1")

module.exports = {
   mode:"development",
   entry: {
      lib: "./src/index.js",
   },
   output: {
      path: path.join(__dirname, "build"),
      filename: "[name].js",
   },
   plugins: [
      new plugin1({ name: 'plugin1' }),
      new plugin2({ name: 'plugin2' })
   ],
};
```

编译后输出的结果如下：

```
//执行 Webpack 命令后输出的结果如下/*
plugin1 初始化
plugin2 初始化
plugin1 apply 被调用
plugin2 apply 被调用
Webpack 设置完初始插件之后执行的生命周期
生成资源到 output 目录之前执行的生命周期
*/
```

首先 Webpack 会按顺序实例化 plugin 对象，之后再依次调用 plugin 对象上的 apply() 方法。

也就是对应输出：plugin1 初始化、plugin2 初始化、plugin1 apply 被调用、plugin2 apply 被调用。

Webpack 在运行过程中会触发各种事件，而在 apply()方法中能接收一个 compiler 对象，可以通过这个对象监听到 Webpack 触发各种事件的时刻，然后执行对应的操作函数。这套机制类似于 Node.js 的 EventEmitter，总体来讲就是一个发布订阅模式。

在 compiler.hooks 中定义了各式各样的事件钩子，这些钩子会在不同的时机被执行，而上面代码中的 compiler.hooks.emit 和 compiler.hooks.afterPlugin 这两个生命周期钩子，分别对应了设置完初始插件及生成资源到 output 目录之前这两个时间节点，afterPlugin 是在 emit 之前被触发的，所以输出的顺序更靠前。

(2) 编写一个输出所有打包目录文件列表的插件，这个插件在构建完相关的文件后，会输出一个记录所有构建文件名的 list.md 文件，代码如下：

```
class myPlugin {
  constructor(option) {
    this.option = option
  }
  apply(compiler) {
    compiler.hooks.emit.tap('myPlugin', compilation => {
      let filelist = '构建后的文件：\n'
      for (var filename in compilation.assets) {
        filelist += '- ' + filename + '\n';
      }
      compilation.assets[list.md'] = {
        source: function() {
          return filelist
        },
        size: function() {
          return filelist.length
```

```
            }
          }
        })
      }
    }
```

在 Webpack 的 emit 事件被触发之后，插件会执行指定的工作，并将包含了编译生成资源的 compilation 作为参数传入函数。可以通过 compilation. assets 获得生成的文件，并获取其中的 filename 值。

(3) Compiler 和 Compilation。上面在开发 Plugin 时最常用的两个对象就是 Compiler 和 Compilation，它们是 Plugin 和 Webpack 之间的桥梁。

Compiler 和 Compilation 的含义如下：

- Compiler 对象包含了 Webpack 环境所有的配置信息，包含 options、loaders、plugins 信息，这个对象在 Webpack 启动时被实例化，它是全局唯一的，可以简单地把它理解为 Webpack 实例；
- Compilation 对象包含了当前的模块资源、编译生成资源、变化的文件等。当 Webpack 以开发模式运行时，每当检测到一个文件变化，一次新的 Compilation 将被创建。Compilation 对象也提供了很多事件回调供插件进行扩展。通过 Compilation 也能读取 Compiler 对象。

Compiler 和 Compilation 的区别在于：Compiler 代表整个 Webpack 从启动到关闭的生命周期，而 Compilation 只代表一次新的编译。

3.3　Rollup

Rollup 是下一代 ES6 模块化工具，它最大的亮点是利用 ES6 模块设计，生成更简洁、更简单的代码。Rollup 更适合构建 JavaScript 库，如图 3-22 所示。

图 3-22　Rollup 框架 Logo

3.3.1　Rollup 介绍

Rollup 是一个 JavaScript 模块打包器，可以将小块代码编译成大块复杂的代码，例如

library 或应用程序。

Rollup 对代码模块使用新的标准化格式，这些标准都包含在 JavaScript 的 ES6 版本中，而不是以前的特殊解决方案，如 CommonJS 和 AMD。ES6 模块可以使开发者自由、无缝地使用最喜爱的 library 中的那些最有用的独立函数，而项目不必携带其他未使用的代码。ES6 模块最终还是要由浏览器原生实现，但当前 Rollup 可以使开发者提前体验。

1. Rollup 的优缺点

Rollup 将所有资源放到同一个地方，一次性加载，利用 Tree Shaking 特性来剔除未使用的代码，减少冗余。它有以下优点：

(1) 配置简单，打包速度快。

(2) 自动移除未引用的代码(内置 Tree Shaking)。

但是它也有以下几个不可忽视的缺点：

(1) 开发服务器不能实现模块热更新，调试烦琐。

(2) 浏览器环境的代码分割时依赖 AMD。

(3) 加载第三方模块比较复杂。

2. Rollup 与 Webpack 及 Parcel 的区别

Rollup 的主要功能是对 JS 进行编译打包，这和 Webpack 有本质区别。Webpack 是一个通用的前端资源打包工具，它不仅可以处理 JS，也可以通过 Loader 来处理 CSS、图片、字体等各种前端资源，还提供了 Hot Reload 等方便前端项目开发的功能。如果不是开发一个 JS 框架，则 Webpack 显然会是一个更好的选择，Parcel 更适于开发和测试环境，开发者无须任何配置就可以使用 Parcel 进行打包，如表 3-6 所示。

表 3-6 Rollup 与 Webpack 及 Parcel 的区别

	Webpack	Rollup	Parcel
文件类型	JS/CSS/HTML 等多种类型的文件(配置 Loader)	主要是 JS 文件	JS/CSS/HTML 等多种类型的文件
多入口	支持(只能是 JS 文件)	支持(配置 Plugin)	支持. html
Library 类型	不支持 ESM	支持 ESM	不支持 ESM
Code Splitting	支持	支持	支持
Tree Shaking	支持	支持	支持
HMR	支持(需要配置)	支持(需要配置)	支持(开发环境自动启动)
TypeScript	支持	支持	支持
构建体积	Rollup＜Webpack＜Parcel		
构建耗时	长	一般	短
配置文件	复杂	复杂	零配置
官方文档	详细但复杂	清晰	不够详细
生态	非常丰富	丰富	一般
通用性	大型项目	JS 库	中小型项目

3.3.2　Rollup 安装与配置

安装 Rollup 的命令如下：

```
#yarn 安装
yarn global add rollup
#NPM 安装
npm install rollup -g
```

3.3.3　Rollup 基础

当 Rollup 不包含任何插件时只是一个模块语法的转换工具，它可以把 ES6 的模块语法编译成不同的模块标准输出，如表 3-7 所示。

表 3-7　支持编译的模块化标准

模块化标准	描　　述
cjs	CommonJS，适用于 Node 和 Browserify/Webpack
iife	IIFE(Immediately Invoked Function Expression)即立即执行函数表达式，所谓立即执行，就是声明一个函数时，声明完了立即执行。自执行模块，适用于浏览器环境 script 标签
amd	异步模块定义，用于像 RequireJS 这样的模块加载器
umd	通用模块定义，以 amd、cjs 和 iife 为一体
es	ES 模块文件

在下面的例子中，用 ES6 语法编写的模块代码需要运行在 Node 中，代码如下：

```
#foo.js
export default 'Hello World!'
```

main.js 入口，代码如下：

```
#main.js
import foo from './foo.js'
export default function () {
  console.log(foo)
}
```

这里 ES6 的模块语法无法在 Node 中运行，可使用 Rollup 进行打包，代码如下：

```
#针对浏览器环境打包
rollup ./src/main.js --file bundle.js --format iife
#针对 Node.js 环境打包
rollup ./src/main.js --file bundle.js --format cjs
#这里的 --format 是指定输出格式
#iife 是指浏览器中常用的自调用函数
```

打包后输出的结果如下：

```
'use strict';
var foo = 'Hello world!';
function main () {
  console.log(foo);
}
module.exports = main;
```

可以看到 Rollup 做了两件事：

(1) 把 ES6 模块语法编译成了 Node 的语法。

(2) 把两个文件的代码打包成了一份。

下面详细讲解 Rollup 编译配置及打包的流程和插件的使用。

1. 配置文件

创建一个 rollup.config.js 文件，配置文件格式如下：

```
export default {
   input:'src/main.js',
   output:{
      file:'dist/bundle.cjs.js',              //输出文件的路径和名称
      format:'cjs',                           //5 种输出格式:amd/es6/iife/umd/cjs
      name:'bundleName',                      //当 format 为 iife 和 umd 时必须提供,将作为全
                                              //局变量挂在 window 下
      sourcemap:true,
      globals:{
         lodash:'_',                          //告诉 Rollup 全局变量'_'即是 lodash
         jquery:'$'                           //告诉 Rollup 全局变量'$'即是 jquery
      }
   }
}
```

有了配置文件执行命令，便可使用 --config 参数，表明使用配置文件，命令如下：

```
yarn rollup --config
yarn rollup --config rollup.config.js       #指明使用的配置文件
```

2. rollup.config.js 配置文件包含选项

Rollup 的详细配置文件如下：

```
export default {
  //核心选项
  input,                 //必选
  external,
  plugins,
```

```
  //额外选项
  onwarn,
  //高危选项
  acorn,
  context,
  moduleContext,
  legacy
  //必选（如果要输出多个，可以是一个数组）
  output: {
    //核心选项
    file,           //必选
    format,         //必选
    name,
    globals,
    //额外选项
    paths,
    banner,
    footer,
    intro,
    outro,
    sourcemap,
    sourcemapFile,
    interop,
    //高危选项
    exports,
    amd,
    indent
    strict
  },
};
```

这里 input、output. file 和 output. format 都是必选的，因此，一个基础配置文件如下：

```
//rollup.config.js
export default {
  input: "main.js",
  output: {
    file: "bundle.js",
    format: "cjs",
  },
};
```

这里的 format 字段有 5 种选项，如表 3-8 所示。

表 3-8 支持编译的模块化标准

模块化标准	描　述
cjs	CommonJS，适用于 Node 和 Browserify/Webpack
iife	自执行模块，适用于浏览器环境 script 标签
amd	异步模块定义，用于像 RequireJS 这样的模块加载器
umd	通用模块定义，以 amd、cjs 和 iife 为一体
es	ES 模块文件

3. 使用插件

如果需要加载其他类型资源模块或者在代码中导入 CommonJS 模块，就需要使用 Rollup 插件。

插件拓展了 Rollup 处理其他类型文件的能力，它的功能有点类似于 Webpack 的 Loader 和 Plugin 的组合。不过配置比 Webpack 中要简单很多，不用逐个声明哪个文件用哪个插件处理，只需要在 Plugin 中声明，在引入对应文件类型时就会自动加载。接下来看几个常用的插件。

1）rollup-plugin-json 插件

rollup-plugin-json 插件让 Rollup 从 JSON 文件中读取数据，代码如下：

```
//在 main.js 文件中导入 json 中的值
import { name, version } from '../package.json'
console.log(name)
console.log(version)
```

配置如下：

```
#安装
yarn add rollup-plugin-json -D

#rollup.config.js
import json from 'rollup-plugin-json'
export default {
  input: 'src/index.js',
  output: {
    file: 'dist/bundle.js',
    format: 'iife'
  },
  plugins: [
    json()
  ]
}
```

package.json 文件的打包命令如下：

```
"build": "rollup -c --environment NODE_ENV:production && rollup -c",
"dev": "rollup -c  --watch",
```

2）rollup-plugin-commonjs 插件

Rollup 默认只能加载 ES6 模块的 JS，但是项目中通常也会用到 CommonJS 的模块，这样 Rollup 解析时就会出现问题，代码如下：

```
//add.js
let count = 1;
let add = () => {
  return count + 1;
};
module.exports = {
  count,
  add,
};

//main.js
//报错
import add from "./add";
console.log("foo", add.count);
```

由于 ES6 模块导入时默认会去找 default，因此这里打包会报错，这时就需要用到 rollup-plugin-commonjs 插件进行转换，配置如下：

```
import commonjs from "rollup-plugin-commonjs";

export default {
  input: "./main.js",
  output: {
    file: "bundle.js",
    format: "iife",
  },
  plugins: [commonjs()],
};
```

4. 模块热更新

Rollup 本身不支持启动开发服务器，可以通过 rollup-plugin-serve 第三方插件来启动一个静态资源服务器，代码如下：

```
import serve from "rollup-plugin-serve";
export default {
  input: "./main.js",
```

```
  output: {
    file: "dist/bundle.js",
    format: "iife",
  },
  plugins: [serve("dist")],
};
```

不过由于其本质上是一个静态资源的服务器，因此不支持模块热更新。

5. Tree Shaking

由于 Rollup 本身支持 ES6 模块化规范，因此不需要额外配置即可进行 Tree Shaking。

6. 代码分割

Rollup 代码分割和 Parcel 一样，也是通过按需导入的方式，但是输出的格式不能使用 iife，因为 iife 自执行函数会把所有模块放到一个文件中，可以通过 amd 或者 cjs 等其他规范，代码如下：

```
export default {
  input: "./main.js",
  output: {
    //输出文件夹
    dir: "dist",
    format: "amd",
  },
};
```

这样通过 import()动态导入的代码就会单独分割到独立的 JS 中，在调用时按需引入。不过对于这种 amd 模块的文件，不能直接在浏览器中引用，必须通过实现 AMD 标准的库加载，例如 Require.js。

7. 多入口打包

多入口打包默认会提取公共模块，意味着会执行代码拆分，所以格式不能是 iife 格式，需要使用多入口打包方式，配置如下：

```
export default {
  //这两种方式都可以
  //input: ['src/index.js', 'src/album.js'],
  input: {
    foo: 'src/index.js',
    bar: 'src/album.js'
  },
  output: {
    dir: 'dist',
    format: 'amd'
  }
```

```
}
<!-- 在浏览器端使用打包好的 dist 目录文件 -->
<!-- 需要引入 require.js --><script src="https://unpkg.com/requirejs@2.3.6/require.
js" data-main="foo.js"></script>
```

3.4　ESBuild

ESBuild 是一个用 Go 语言编写的 JavaScrip、TypeScript 打包工具，如图 3-23 所示。ESBuild 有两大功能，分别是 bundler 与 minifier，其中 bundler 用于代码编译，类似 babel-loader、ts-loader；minifier 用于代码压缩，类似 terser。

图 3-23　ESBuild 构建工具 Logo

大多数前端打包工具是基于 JavaScript 实现的，而 ESBuild 则选择使用 Go 语言编写，两种语言各自有其擅长的场景，但是在资源打包这种 CPU 密集型场景下，Go 语言天生具有多线程运行能力，因此更具性能优势。

ESBuild 官方网址为 https://esbuild.github.io/。

1. ESBuild 介绍

ESBuild 是由 Figma 的 CTO(Evan Wallace)基于 Go 语言开发的一款打包工具，相比传统的打包工具，主打性能优势在于构建速度上可以快 10～100 倍。

说明：Figma 是一个基于浏览器的协作式 UI 设计工具。Figma Inc. 创立于 2012 年 10 月 1 日，总部位于美国加州旧金山，开发了多人协作界面设计工具，可使整个团队的设计过程在一个在线工具中进行。

ESBuild 项目的主要目标是：开辟一个构建工具性能的新时代，创建一个易用的现代打包器。现在很多工具内置了它，例如熟知的 Vite、Snowpack。借助 ESBuild 优异的性能，Vite 更如虎添翼，编译打包速度显著提升。

ESBuild 的主要特征有以下几点：

(1) 打包时极致的速度，不需要缓存。

(2) 支持 Source Maps。

(3) 支持压缩、支持插件。

(4) 支持 ES6 和 CommonJS 模块。

(5) 支持 ES6 模块的 Tree Shaking。

(6) 提供 JavaScript 和 Go 的 API。

（7）支持 TypeScript 和 JSX 的语法。

2. ESBuild 使用场景

1）代码压缩工具

ESBuild 的代码压缩功能非常优秀，其性能比传统的压缩工具高一个量级以上。Vite 在 2.6 版本也官宣在生产环境中直接使用 ESBuild 来压缩 JS 和 CSS 代码。

2）第三方库 Bundler

Vite 在开发阶段使用 ESBuild 进行依赖的预打包，将所有用到的第三方依赖转换成 ESM 格式 Bundler 产物，并且未来有用到生产环境的打算。

3）小程序编译

对于小程序的编译场景，也可以使用 ESBuild 来代替 Webpack，大大提升编译速度，对于 AST 的转换则通过 ESBuild 插件嵌入 SWC 实现，从而实现快速编译。

4）Web 构建

Web 场景就显得比较复杂了，对于兼容性和周边工具生态的要求比较高，例如低浏览器语法降级、CSS 预编译器、HMR 等，如果要用纯 ESBuild 来做，则还需要补充很多能力。

3. ESBuild 安装与配置

```
#本地安装,或者全局安装
npm install esbuild
#查看版本
.\node_modules\.bin\esbuild --version
```

4. ESBuild 快速上手

下面通过一个简单的 Demo 演示 ESBuild 如何编译打包 React 项目。

首先需要安装 React、ReactDOM 库，命令如下：

```
npm install react react-dom -S
```

创建一个 App.jsx 组件页面，代码如下：

```
import React from 'react'
import ReactDOM from 'react-dom'
let Greet = () => <h1>Hello, ESBuild!</h1>
ReactDOM.render(<Greet />,document.querySelector("#app"))
```

在 package.json 文件中添加一个编译命令，命令如下：

```
"scripts": {
    "build": "esbuild App.jsx --bundle --outfile=out.js"
  },
```

执行脚本命令，命令如下：

```
npm run build
```

3.5　Vite

图 3-24　Vite 构建工具 Logo

Vite 是 Vue 框架的作者尤雨溪为 Vue 3 开发的新的构建工具，目的是替代 Webpack，其原理是利用现代浏览器已经支持 ES6 的动态 import，当遇到 import 时会发送一个 HTTP 请求去加载文件，Vite 会拦截这些请求，进行预编译，省去了 Webpack 冗长的打包时间，提升开发体验。Vite 构建工具的 Logo 如图 3-24 所示。

Vite 借鉴了 Snowpack，在生产环境使用 Rollup 打包。相比 Snowpack，它支持多页面、库模式、动态导入、自动 polyfill 等。

Vite 开源网址为 https://github.com/vitejs/vite，Vite 官方网址为 https://vitejs.dev/。

3.5.1　Vite 介绍

Vite 解决了 Webpack 开发阶段 Dev Server 冷启动时间过长，HMR（热更新）反应速度慢的问题。早期的浏览器基本上不支持 ES Module，这个时候需要使用 Webpack、Rollup、Parcel 等打包构建工具来提取、处理、连接和打包源码，但是当项目变得越来越复杂，模块数量越来越多时，特别是在开发过程中，启动一个 Dev Server 所需要的时间也会变得越来越长，当编辑代码、保存、使有 HRM 功能时，可能也要花费几秒才能反映到页面中。这种开发体验是非常耗时的，同时体验也非常差，而 Vite 就是为解决这种开发体验上的问题的。总体来讲 Vite 有以下优点：

（1）去掉打包步骤，快速地冷启动。

（2）及时进行模块热更新，不会随着模块变多而使热更新变慢。

（3）真正按需编译。

3.5.2　Vite 基本使用

Vite 不仅支持 Vue 3 项目构建，同时也支持其他的前端流行框架的项目构建，目前支持的框架有 vanilla、vanilla-ts、vue、vue-ts、react、react-ts、preact、preact-ts、lit、lit-ts、svelte、svelte-ts。

注意：Vite 需要 Node.js 版本不低于 12.0.0 版，然而，有些模板需要依赖更高的 Node 版本才能正常运行，当包管理器发出警告时，需要注意升级 Node 版本。

1. Vite 构建 Vue 项目

安装 Vite，同时使用 Vue 3 模板构建项目，命令如下：

```
#npm 6.x
npm create vite@latest my-vue-app -- --template vue
#npm 7+,extra double-dash is needed
npm create vite@latest my-vue-app -- -- --template vue
#yarn
yarn create vite my-vue-app -- --template vue
#pnpm
pnpm create vite my-vue-app -- -- --template vue
```

命令执行的结果如图 3-25 所示。

图 3-25　创建项目成功提示

接下来,进入 my-vue-app 目录,执行 yarn 命令安装依赖包,再执行 yarn dev 命令启动项目,如图 3-26 所示。

在浏览器中预览的效果如图 3-27 所示。

图 3-26　执行 yarn dev 命令启动项目

图 3-27　预览 Vue 3+Vite 项目效果

2. Vite 构建 React 项目

安装 Vite，同时使用 React 模板构建项目，命令如下：

```
#npm 6.x
npm create vite@latest my-vue-app -- template react
#npm 7+,extra double-dash is needed
npm create vite@latest my-vue-app --  -- template react
#yarn
yarn create vite my-vue-app -- template react
#pnpm
pnpm create vite my-vue-app --  -- template react
```

创建的项目结构如图 3-28 所示。

在项目目录下，执行 yarn dev 命令，启动项目，在浏览器中预览效果，如图 3-29 所示。

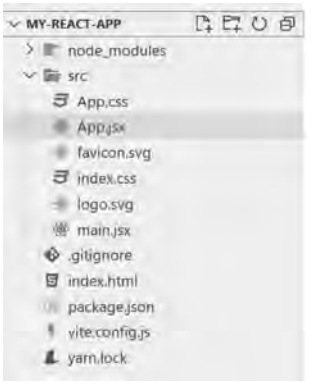

图 3-28　Vite＋React 项目的目录结构

图 3-29　预览 Vite＋React 项目效果

3.5.3　Vite 原理

在介绍 Vite 原理之前，需要先了解打包模式（Bundle）和无打包模式（Bundleless）。Vite 借鉴了 Snowpack，采用无打包模式。无打包模式只会编译代码，不会打包，因此构建速度极快，与打包模式相比时间缩短了 90％以上。

1. 打包 vs 无打包构建

2015 年之前，前端开发需要打包工具来解决前端工程化构建的问题，主要原因在于网络协议 HTTP 1.1 标准有并行连接限制，浏览器方面也不支持模块系统（如 CommonJS 包不能直接在浏览器运行），同时存在代码依赖关系与顺序管理问题。

但随着 2015 年 ESM 标准发布后，网络通信协议也发展到多路并用的 HTTP 2 标准，目前大部分浏览器已经支持了 HTTP 2 标准和浏览器的 ES Module，与此同时，随着前端工程体积的日益增长与亟待提升的构建性能之间的矛盾越来越突出，无打包模式逐渐发展兴起。

打包模式与无打包模式对比如表 3-9 所示。

表 3-9 打包模式与无打包模式对比

	Bundle(Webpack)	Bundleless(Vite/Snowpack)
启动时间	长,需要完成打包项目	短,只启动 Server 按需加载
构建时间	随项目体积线性增长	构建时间复杂度 $O(1)$
加载性能	打包后加载对应的 Bundle	请求映射至本地文件
缓存能力	缓存利用率一般,受 split 方式影响	缓存利用率近乎完美
文件更新	重新打包	重新请求单个文件
调试体验	通常需要 SourceMap 进行调试	不强依赖 SourceMap,可单文件调试
生态	非常完善	目前相对不成熟,但是发展很快

2. Vite 分为开发模式和生产模式

开发模式:Vite 提供了一个开发服务器,然后结合原生的 ESM,当代码中出现 import 时,发送一个资源请求,Vite 开发服务器拦截请求,根据不同的文件类型,在服务器端完成模块的改写(例如单文件的解析、编译等)和请求处理,实现真正的按需编译,然后返回浏览器。请求的资源在服务器端按需编译返回,完全跳过了打包这个概念,不需要生成一个大的包。服务器随启随用,所以开发环境下的初次启动是非常快的,而且热更新的速度不会随着模块增多而变慢,因为代码改动后,并不会有打包的过程。

Vite 本地开发服务器所有逻辑基本依赖中间件实现,如图 3-30 所示,中间件拦截请求之后,主要负责以下内容:

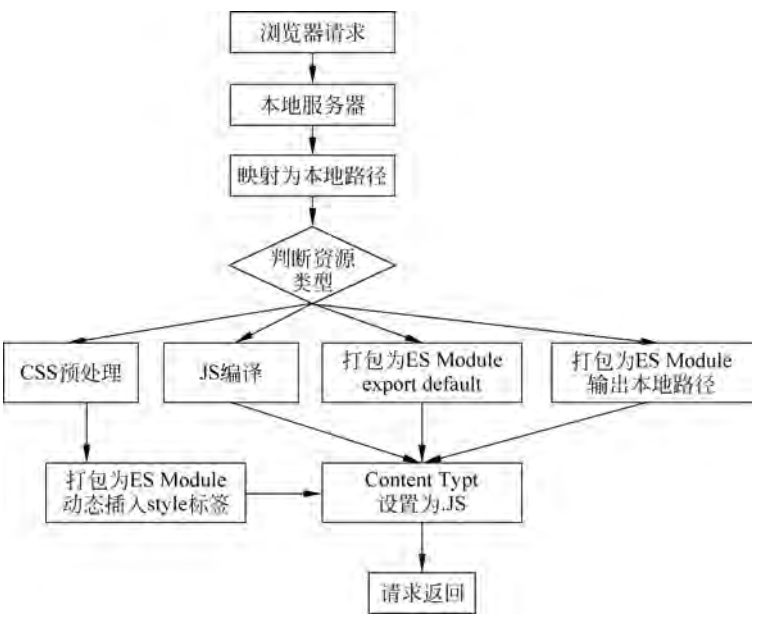

图 3-30 拦截不同的资源请求,实时编译转换

(1) 处理 ESM 语法,例如将业务代码中的 import 第三方依赖路径转换为浏览器可识别的依赖路径。

(2) 对.ts、.vue 等文件进行即时编译。

(3) 对 Sass/Less 等需要预编译的模块进行编译。

(4) 和浏览器端建立 socket 连接,实现 HMR。

生产模式:利用 Rollup 来构建源码,Vite 将需要处理的代码分为以下两大类。

第三方依赖:这类代码大部分是纯 JavaScript 代码,而且不会经常变化,Vite 会通过 pre-bundle 的方式来处理这部分代码。Vite 2 使用 ESBulid 来构建这部分代码,ESBuild 是基于 Go 语言实现的,处理速度会比用 JavaScript 写的打包器快 10～100 倍,这也是 Vite 为什么在开发阶段处理代码很快的一个原因。

业务代码:通常这部分代码不是纯的 JavaScript(例如 JSX、Vue 等)代码,经常会被修改,而且也不需要一次性全部加载(可以根据路由进行代码分割后加载)。

由于 Vite 使用了原生的 ESM,所以 Vite 本身只要按需编译代码,然后启动静态服务器就可以了。只有当浏览器请求这些模块时,这些模块才会被编译,以便动态加载到当前页面中。