版权注意事项：
1、书籍版权归作者和出版社所有
2、本PDF仅限用于个人获取知识，进行私底下的知识交流
3、PDF获得者不得在互联网上以任何目的进行传播
4、如觉得书籍内容很赞，请购买正版实体书，支持作者
5、请于下载PDF后24小时内删除本PDF。

Broadview®
www.broadview.com.cn

吴浩麟 / 著

深入浅出 Webpack

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

/ 作者简介 /

吴浩麟

- 一线前端工程师，曾就职于腾讯，现就职于美团。
- 专注于Web开发，参与过众多大型Web项目的构建、设计和开发，喜欢探索Web前沿技术。
- 也是Golang和音视频技术的爱好者，活跃于GitHub，ID为gwuhaolin。

深入浅出 Webpack

吴浩麟 / 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

随着 Web 开发技术的发展，Webpack 凭借其便于使用和涵盖面广的优势，成为目前非常流行的前端构建工具，是每位前端工程师的必备技能之一。

本书对 Webpack 进行了全面讲解，涵盖了 Webpack 入门、配置、实战、优化、原理等方面的内容。其中，第 1 章讲解 Webpack 入门所涉及的知识；第 2 章详细讲解 Webpack 提供的常用配置项；第 3 章结合实际项目中的常见场景进行实践；第 4 章给出优化 Webpack 的优秀方案；第 5 章剖析了 Webpack 的原理，并讲解如何开发 Plugin 和 Loader；附录汇总了常见的 Loader、Plugin 和 Webpack 的其他学习资源。除了深入讲解 Webpack，本书还介绍了 ES6、TypeScript、PostCSS、Prepack、离线缓存、单页应用、CDN 等 Web 开发相关的技能。

无论是对 Webpack 一无所知的初学者，还是经验丰富的前端工程师，相信都能够通过本书进一步提升对 Webpack 的理解，并在 Web 开发中更熟练地运用 Webpack。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

深入浅出 Webpack / 吴浩麟著. 一北京：电子工业出版社，2018.1

ISBN 978-7-121-33172-5

Ⅰ. ①深… Ⅱ. ①吴… Ⅲ. ①网页制作工具－程序设计 Ⅳ. ①TP392.092.2

中国版本图书馆 CIP 数据核字（2017）第 295678 号

策划编辑：张国霞
责任编辑：徐津平
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
开　　本：787×980　1/16　印张：18　　字数：375 千字
版　　次：2018 年 1 月第 1 版
印　　次：2018 年 1 月第 1 次印刷
印　　数：2500 册　　定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前言

Web 应用日益复杂，相关开发技术也百花齐放，这对前端构建工具提出了更高的要求。Webpack 从众多构建工具中脱颖而出，成为目前最流行的构建工具，也几乎成为目前前端开发里的必备工具之一，因此每位紧跟时代的前端工程师都应该掌握 Webpack。

本书从实践出发，用简单易懂的例子带领读者快速入门 Webpack，再结合实际工作中常用的场景给出实践案例，通过前 3 章的学习足以让我们解决工作中的常见问题；本书还介绍了如何优化构建的速度和输出，并解析了 Webpack 的工作原理，以及 Plugin 和 Loader 的编写方法，可帮助读者进一步学习 Webpack。本书按照入门、配置、实战、优化和原理的路线层层深入，涵盖了 Webpack 的方方面面。

在编写本书时，Webpack 已经迭代到了 3.8.1 版本，本书涵盖了对该版本的特性解析。由于 Webpack 从版本 1 到现在，其核心思想和 API 没有发生很大的变化，所以本书的大部分内容适用于 Webpack 的任何稳定版本，但部分实例代码只适用于最新版本。

本书的每一小节都会提供与之对应的完整项目代码，在每节的最后附有下载链接，它们都有详细的注释并且可以正常运行，我们可以在需要时下载这些代码。

在阅读本书前，我们需要掌握基本的 Web 开发技术，因为本书专注于 Webpack，不会详细介绍其他不相关的内容。

在阅读本书时，如果遇到任何不明白的地方，则都可以在本书的 GitHub 项目主页上（https://github.com/gwuhaolin/dive-into-webpack）以提 Issue 的方式提出问题，作者将详细解答。

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/33172

目 录

第1章 入门 1

1.1 前端的发展 …… 2

1.1.1 模块化 …… 2

1.1.2 新框架 …… 5

1.1.3 新语言 …… 6

1.2 常见的构建工具及对比 …… 8

1.2.1 Npm Script …… 9

1.2.2 Grunt …… 10

1.2.3 Gulp …… 11

1.2.4 Fis3 …… 12

1.2.5 Webpack …… 14

1.2.6 Rollup …… 15

1.2.7 为什么选择 Webpack …… 16

1.3 安装 Webpack …… 17

1.3.1 安装 Webpack 到本项目 …… 17

1.3.2 安装 Webpack 到全局 …… 18

1.3.3 使用 Webpack …… 18

1.4 使用 Loader ····· 20
1.5 使用 Plugin ····· 22
1.6 使用 DevServer ····· 24
1.6.1 实时预览 ····· 25
1.6.2 模块热替换 ····· 26
1.6.3 支持 Source Map ····· 26
1.7 核心概念 ····· 27

第 2 章 配置 29

2.1 Entry ····· 30
2.1.1 context ····· 30
2.1.2 Entry 类型 ····· 31
2.1.3 Chunk 的名称 ····· 31
2.1.4 配置动态 Entry ····· 32
2.2 Output ····· 32
2.2.1 filename ····· 32
2.2.2 chunkFilename ····· 33
2.2.3 path ····· 34
2.2.4 publicPath ····· 34
2.2.5 crossOriginLoading ····· 34
2.2.6 libraryTarget 和 library ····· 35
2.2.7 libraryExport ····· 37
2.3 Module ····· 38
2.3.1 配置 Loader ····· 38
2.3.2 noParse ····· 40
2.3.3 parser ····· 41

2.4 Resolve ····· 41
2.4.1 alias ····· 42
2.4.2 mainFields ····· 42
2.4.3 extensions ····· 43
2.4.4 modules ····· 43
2.4.5 descriptionFiles ····· 44
2.4.6 enforceExtension ····· 44
2.4.7 enforceModuleExtension ····· 44
2.5 Plugin ····· 44
2.6 DevServer ····· 45
2.6.1 hot ····· 45
2.6.2 inline ····· 46
2.6.3 historyApiFallback ····· 46
2.6.4 contentBase ····· 47
2.6.5 headers ····· 48
2.6.6 host ····· 48
2.6.7 port ····· 48
2.6.8 allowedHosts ····· 48
2.6.9 disableHostCheck ····· 49
2.6.10 https ····· 49
2.6.11 clientLogLevel ····· 50
2.6.12 compress ····· 50
2.6.13 open ····· 50
2.7 其他配置项 ····· 50
2.7.1 Target ····· 50
2.7.2 Devtool ····· 51
2.7.3 Watch 和 WatchOptions ····· 51
2.7.4 Externals ····· 52

2.7.5 ResolveLoader ····· 53
2.8 整体配置结构 ····· 54
2.9 多种配置类型 ····· 58
2.9.1 导出一个 Function ····· 58
2.9.2 导出一个返回 Promise 的函数 ····· 59
2.9.3 导出多份配置 ····· 60
2.10 总结 ····· 61

第 3 章 实战 62

3.1 使用 ES6 语言 ····· 63
3.1.1 认识 Babel ····· 63
3.1.2 接入 Babel ····· 67
3.2 使用 TypeScript 语言 ····· 67
3.2.1 认识 TypeScript ····· 67
3.2.2 减少代码冗余 ····· 69
3.2.3 集成 Webpack ····· 69
3.3 使用 Flow 检查器 ····· 70
3.3.1 认识 Flow ····· 70
3.3.2 使用 Flow ····· 71
3.3.3 集成 Webpack ····· 72
3.4 使用 SCSS 语言 ····· 73
3.4.1 认识 SCSS ····· 73
3.4.2 接入 Webpack ····· 74
3.5 使用 PostCSS ····· 75
3.5.1 认识 PostCSS ····· 75
3.5.2 接入 Webpack ····· 76

3.6 使用 React 框架 …… 77
3.6.1 React 的语法特征 …… 77
3.6.2 React 与 Babel …… 78
3.6.3 React 与 TypeScript …… 79
3.7 使用 Vue 框架 …… 80
3.7.1 认识 Vue …… 81
3.7.2 接入 Webpack …… 82
3.7.3 使用 TypeScript 编写 Vue 应用 …… 83
3.8 使用 Angular2 框架 …… 85
3.8.1 认识 Angular2 …… 85
3.8.2 接入 Webpack …… 88
3.9 为单页应用生成 HTML …… 89
3.9.1 引入问题 …… 89
3.9.2 解决方案 …… 90
3.10 管理多个单页应用 …… 94
3.10.1 引入问题 …… 94
3.10.2 解决方案 …… 96
3.11 构建同构应用 …… 100
3.11.1 认识同构应用 …… 100
3.11.2 解决方案 …… 101
3.12 构建 Electron 应用 …… 105
3.12.1 认识 Electron …… 105
3.12.2 接入 Webpack …… 108
3.13 构建 Npm 模块 …… 110
3.13.1 认识 Npm …… 110
3.13.2 抛出问题 …… 111
3.13.3 使用 Webpack 构建 Npm 模块 …… 112
3.13.4 发布到 Npm …… 117

3.14 构建离线应用 ····· 118
3.14.1 认识离线应用 ····· 118
3.14.2 认识 Service Workers ····· 119
3.14.3 接入 Webpack ····· 124
3.14.4 验证结果 ····· 126
3.15 搭配 Npm Script ····· 128
3.15.1 认识 Npm Script ····· 128
3.15.2 Webpack 为什么需要 Npm Script ····· 129
3.16 检查代码 ····· 130
3.16.1 代码检查具体是做什么的 ····· 130
3.16.2 怎么做代码检查 ····· 131
3.16.3 结合 Webpack 检查代码 ····· 133
3.17 通过 Node.js API 启动 Webpack ····· 136
3.17.1 安装和使用 Webpack 模块 ····· 136
3.17.2 以监听模式运行 ····· 137
3.18 使用 Webpack Dev Middleware ····· 138
3.18.1 Webpack Dev Middleware 支持的配置项 ····· 139
3.18.2 Webpack Dev Middleware 与模块热替换 ····· 140
3.19 加载图片 ····· 142
3.19.1 使用 file-loader ····· 142
3.19.2 使用 url-loader ····· 143
3.20 加载 SVG ····· 145
3.20.1 使用 raw-loader ····· 146
3.20.2 使用 svg-inline-loader ····· 147
3.21 加载 Source Map ····· 148
3.21.1 该如何选择 ····· 150
3.21.2 加载现有的 Source Map ····· 150
3.22 实战总结 ····· 151

第4章 优化 153

4.1 缩小文件的搜索范围 154
4.1.1 优化 Loader 配置 154
4.1.2 优化 resolve.modules 配置 155
4.1.3 优化 resolve.mainFields 配置 156
4.1.4 优化 resolve.alias 配置 157
4.1.5 优化 resolve.extensions 配置 159
4.1.6 优化 module.noParse 配置 159
4.2 使用 DllPlugin 160
4.2.1 认识 DLL 160
4.2.2 接入 Webpack 161
4.3 使用 HappyPack 166
4.3.1 使用 HappyPack 167
4.3.2 HappyPack 的原理 170
4.4 使用 ParallelUglifyPlugin 170
4.5 使用自动刷新 173
4.5.1 文件监听 173
4.5.2 自动刷新浏览器 176
4.6 开启模块热替换 180
4.6.1 模块热替换的原理 180
4.6.2 优化模块热替换 184
4.7 区分环境 186
4.7.1 为什么需要区分环境 186
4.7.2 如何区分环境 186
4.7.3 结合 UglifyJS 188
4.7.4 第三方库中的环境区分 188

4.8 压缩代码 …… 189
4.8.1 压缩 JavaScript …… 190
4.8.2 压缩 ES6 …… 192
4.8.3 压缩 CSS …… 193
4.9 CDN 加速 …… 195
4.9.1 什么是 CDN …… 195
4.9.2 接入 CDN …… 196
4.9.3 用 Webpack 实现 CDN 的接入 …… 198
4.10 使用 Tree Shaking …… 200
4.10.1 认识 Tree Shaking …… 200
4.10.2 接入 Tree Shaking …… 201
4.11 提取公共代码 …… 204
4.11.1 为什么需要提取公共代码 …… 204
4.11.2 如何提取公共代码 …… 205
4.11.3 如何通过 Webpack 提取公共代码 …… 206
4.12 分割代码以按需加载 …… 209
4.12.1 为什么需要按需加载 …… 209
4.12.2 如何使用按需加载 …… 209
4.12.3 用 Webpack 实现按需加载 …… 210
4.12.4 按需加载与 ReactRouter …… 212
4.13 使用 Prepack …… 215
4.13.1 认识 Prepack …… 215
4.13.2 接入 Webpack …… 216
4.14 开启 Scope Hoisting …… 217
4.14.1 认识 Scope Hoisting …… 217
4.14.2 使用 Scope Hoisting …… 218
4.15 输出分析 …… 219
4.15.1 官方的可视化分析工具 …… 220

4.15.2 webpack-bundle-analyzer ····· 224
4.16 优化总结 ····· 226

第 5 章 原理 233

5.1 工作原理概括 ····· 234
5.1.1 基本概念 ····· 234
5.1.2 流程概括 ····· 234
5.1.3 流程细节 ····· 235
5.2 输出文件分析 ····· 238
5.3 编写 Loader ····· 245
5.3.1 Loader 的职责 ····· 246
5.3.2 Loader 基础 ····· 247
5.3.3 Loader 进阶 ····· 247
5.3.4 其他 Loader API ····· 250
5.3.5 加载本地 Loader ····· 251
5.3.6 实战 ····· 253
5.4 编写 Plugin ····· 254
5.4.1 Compiler 和 Compilation ····· 255
5.4.2 事件流 ····· 256
5.4.3 常用的 API ····· 257
5.4.4 实战 ····· 261
5.5 调试 Webpack ····· 262
5.6 原理总结 ····· 265

附录 A 常用的 Loader ····· 266
附录 B 常用的 Plugin ····· 270
附录 C Webpack 的其他学习资源 ····· 273

第1章 入门

本章讲解当下最流行的前端构建工具 Webpack（https://webpack.js.org）。

在学习 Webpack 前，我们需要先知道为什么要用 Webpack，本章将通过两个小节进行讲解。

- 1.1 前端的发展：介绍近年来 Web 开发里出现的新技术和前端的发展趋势。
- 1.2 常见的构建工具及对比：讲解构建是什么，为什么需要构建，以及对常见构建工具的介绍和对比。

之后，本章以一个简单的项目“Hello，Webpack”为例，让我们一步步掌握 Webpack 的基础功能。

- 1.3 安装与使用：带我们踏入 Webpack 的大门，将 Webpack 正常运行起来。
- 1.4 使用 Loader：使用 Webpack 的 Loader 功能加载 CSS。
- 1.5 使用 Plugin：使用 Webpack 的 Plugin 功能提取 CSS。
- 1.6 使用 DevServer：使用 DevServer 提升开发体验。
- 1.7 核心概念：通过对以上几节的学习，我们已经掌握了 Webpack 里基础、核心的功能，本节通过总结来加深我们对 Webpack 的认识，同时约定一些专有名词为后面的深入学习做准备。

1.1 前端的发展

近年来 Web 应用变得更加复杂与庞大，Web 前端技术的应用范围也更加广泛。从复杂、庞大的管理后台，到对性能要求苛刻的移动网页，再到类似于 ReactNative 的原生应用开发方案，Web 前端工程师在面临更多机遇的同时也面临更大的挑战。通过直接编写 JavaScript、CSS、HTML 开发 Web 应用的方式已经无法应对当前 Web 应用的发展。近年来，前端社区涌现出许多新思想与框架，下面将一一介绍它们。

1.1.1 模块化

模块化是指将一个复杂的系统分解为多个模块以方便编码。

很久以前，开发网页要通过命名空间的方式来组织代码，例如 jQuery 库将它的 API 都放在了 `window.$`下，在加载完 jQuery 后，其他模块再通过 `window.$`去使用 jQuery。这样做有很多问题，其中包括：

- 命名空间冲突，两个库可能会使用同一个名称，例如 Zepto（http://zeptojs.com）也被放在 `window.$`下；
- 无法合理地管理项目的依赖和版本；
- 无法方便地控制依赖的加载顺序。

当项目变大时，这种方式将变得难以维护，需要用模块化的思想来组织代码。

1. CommonJS

CommonJS（http://www.commonjs.org）是一种被广泛使用的 JavaScript 模块化规范，其核心思想是通过 `require` 方法来同步加载依赖的其他模块，通过 `module.exports` 导出需要暴露的接口。CommonJS 规范的流行得益于 Node.js 采用了这种方式，后来这种方式被引入

到了网页开发中。

采用 CommonJS 导入及导出的代码如下：

```
// 导入
const moduleA = require('./moduleA');
// 导出
module.exports = moduleA.someFunc;
```

CommonJS 的优点在于：

- 代码可复用于 Node.js 环境下并运行，例如做同构应用；
- 通过 Npm 发布的很多第三方模块都采用了 CommonJS 规范。

CommonJS 的缺点在于，这样的代码无法直接运行在浏览器环境下，必须通过工具转换成标准的 ES5。

CommonJS 还可以细分为 CommonJS1 和 CommonJS2，区别在于 CommonJS1 只能通过 `exports.XX = XX` 的方式导出，而 CommonJS2 在 CommonJS1 的基础上加入了 `module.exports = XX` 的导出方式。CommonJS 通常指 CommonJS2。

2. AMD

AMD（https://en.wikipedia.org/wiki/Asynchronous_module_definition）也是一种 JavaScript 模块化规范，与 CommonJS 最大的不同在于，它采用了异步的方式去加载依赖的模块。AMD 规范主要用于解决针对浏览器环境的模块化问题，最具代表性的实现是 requirejs（http://requirejs.org）。

采用 AMD 导入及导出的代码如下：

```
// 定义一个模块
define('module', ['dep'], function(dep) {
  return exports;
});
// 导入和使用
require(['module'], function(module) {
});
```

AMD 的优点在于：

- 可在不转换代码的情况下直接在浏览器中运行；
- 可异步加载依赖；
- 可并行加载多个依赖；
- 代码可运行在浏览器环境和 Node.js 环境下。

AMD 的缺点在于 JavaScript 运行环境没有原生支持 AMD，需要先导入实现了 AMD 的库后才能正常使用。

3. ES6 模块化

ES6 模块化是国际标准化组织 ECMA 提出的 JavaScript 模块化规范，它在语言层面上实现了模块化。浏览器厂商和 Node.js 都宣布要原生支持该规范。它将逐渐取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。

采用 ES6 模块化导入及导出的代码如下：

```
// 导入
import { readFile } from 'fs';
import React from 'react';
// 导出
export function hello() {};
export default {
  // ...
};
```

ES6 模块虽然是终极模块化方案，但它的缺点在于目前无法直接运行在大部分 JavaScript 运行环境下，必须通过工具转换成标准的 ES5 后才能正常运行。

4. 样式文件中的模块化

除了 JavaScript 开始进行模块化改造，前端开发里的样式文件也支持模块化。以 SCSS 为例，将一些常用的样式片段放进一个通用的文件里，再在另一个文件里通过`@import`语句导入和使用这些样式片段：

```
// util.scss 文件
```

```
// 定义样式片段
@mixin center {
  // 水平竖直居中
  position: absolute;
  left: 50%;
  top: 50%;
  transform: translate(-50%,-50%);
}
// main.scss 文件
// 导入和使用 util.scss 中定义的样式片段
@import "util";
#box{
  @include center;
}
```

1.1.2　新框架

在 Web 应用变得庞大、复杂时，采用直接操作 DOM 的方式去开发会使代码变得复杂和难以维护，许多新思想被引入到网页开发中以减少开发难度和提升开发效率。

1. React

React（https://facebook.github.io/react/）框架引入了 JSX 语法到 JavaScript 语言层面中，可以更灵活地控制视图的渲染逻辑。

```
let has = true;
render(has ? <h1>hello,react</h1> : <div>404</div>);
```

这种语法无法直接在任何现成的 JavaScript 引擎里运行，必须经过转换。

2. Vue

Vue（https://vuejs.org）框架将与一个组件相关的 HTML 模板、JavaScript 逻辑代码、CSS 样式代码都写在一个文件里，这非常直观。

```
<!--HTML 模板-->
```

```
<template>
  <div class="example">{{ msg }}</div>
</template>
<!--JavaScript 组件逻辑-->
<script>
export default {
  data () {
    return {
      msg: 'Hello world!'
    }
  }
}
</script>
<!--CSS 样式-->
<style>
.example {
  font-weight: bold;
}
</style>
```

3. Angular2

Angular2（https://angular.io）推崇采用 TypeScript 语言开发应用，并且可以通过注解的语法描述组件的各种属性。

```
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>`
})
export class AppComponent {
  title = 'Tour of Heroes';
}
```

1.1.3 新语言

JavaScript 最初被设计用于完成一些简单的工作，在用它开发大型应用时会有一些语言缺陷暴露出来。CSS 只能用静态的语法描述元素的样式，无法像写 JavaScript 那样增加逻辑判断

与共享变量。为了解决这些问题，许多新语言诞生了。

1. ES6

ECMAScript 6.0（简称 ES6）是 JavaScript 语言的下一代标准。它在语言层面为 JavaScript 引入了很多新语法和 API，使得 JavaScript 语言可以用来编写复杂的大型应用程序。例如：

- 规范模块化；
- Class 语法；
- 用 `let` 声明代码块内有效的变量，用 `const` 声明常量；
- 箭头函数；
- async 函数；
- Set 和 Map 数据结构。

通过这些新特性，可以更高效地编写代码，并专注于解决问题本身。但遗憾的是，不同的浏览器对这些特性的支持不一致，使用了这些特性的代码可能会在部分浏览器下无法运行。为了解决兼容性的问题，需要将 ES6 代码转换成 ES5 代码，Babel（https://babeljs.io）是目前解决这个问题的优秀工具。Babel 的插件机制让它可灵活配置，支持将任何新语法转换成 ES5 的写法。

若想学习更多 ES6 的新特性，则可阅读阮一峰的《ECMAScript 6 入门》（http://es6.ruanyifeng.com）。

2. TypeScript

TypeScript（https://www.typescriptlang.org）是 JavaScript 的一个超集，由 Microsoft 开发并开源，除了支持 ES6 的所有功能，还提供了静态类型检查。采用 TypeScript 编写的代码可以被编译成符合 ES5、ES6 标准的 JavaScript。只有将 TypeScript 用于开发大型项目时，其优点才能体现出来，因为大型项目由多个模块组合而成，不同的模块可能又由不同的人编写，在对接不同的模块时，静态类型检查会在编译阶段找出可能存在的问题。TypeScript 的缺点在于语法相对于 JavaScript 更啰唆，并且无法在浏览器或 Node.js 环境下直接运行。

```
// 静态类型检查机制会检查传给 hello 函数的数据类型
function hello(content: string) {
  return `Hello, ${content}`;
}
let content = 'word';
hello(content);
```

3. Flow

Flow（https://flow.org）也是 JavaScript 的一个超集，它的主要特点是为 JavaScript 提供静态类型检查，和 TypeScript 相似但更灵活，可以让我们只在需要的地方加上类型检查。

4. SCSS

SCSS（http://sass-lang.com）可以让我们用程序员的方式写 CSS。它是一种 CSS 预处理器，其基本思想是用和 CSS 相似的编程语言写完后再编译成正常的 CSS 文件。

```
$blue: #1875e7;
div {
  color: $blue;
}
```

采用 SCSS 写 CSS 的好处在于可以方便地管理代码，抽离公共的部分，通过逻辑写出更灵活的代码。和 SCSS 类似的 CSS 预处理器还有 LESS（http://lesscss.org）等。

使用新语言可以提升编码效率，但是必须将源代码转换成可以直接在浏览器环境下运行的代码。

1.2 常见的构建工具及对比

在阅读完 1.1 节后，我们一定会感叹前端技术发展之快，各种可以提高开发效率的新思想和框架层出不穷。但是它们都有一个共同点：源代码无法直接运行，必须通过转换后才可以正常运行。

构建就是做这件事情，将源代码转换成可执行的 JavaScript、CSS、HTML 代码，包括如下内容。

- 代码转换：将 TypeScript 编译成 JavaScript、将 SCSS 编译成 CSS 等。
- 文件优化：压缩 JavaScript、CSS、HTML 代码，压缩合并图片等。
- 代码分割：提取多个页面的公共代码，提取首屏不需要执行部分的代码让其异步加载。
- 模块合并：在采用模块化的项目里会有很多个模块和文件，需要通过构建功能将模块分类合并成一个文件。
- 自动刷新：监听本地源代码的变化，自动重新构建、刷新浏览器。
- 代码校验：在代码被提交到仓库前需要校验代码是否符合规范，以及单元测试是否通过。
- 自动发布：更新代码后，自动构建出线上发布代码并传输给发布系统。

构建其实是工程化、自动化思想在前端开发中的体现，将一系列流程用代码去实现，让代码自动化地执行这一系列复杂的流程。构建为前端开发注入了更大的活力，解放了我们的生产力。

历史上先后出现了一系列构建工具，它们各有优缺点。由于前端工程师很熟悉 JavaScript，Node.js 又可以胜任所有构建需求，所以大多数构建工具都是用 Node.js 开发的。下面来一一介绍它们。

1.2.1　Npm Script

Npm Script（https://docs.npmjs.com/misc/scripts）是一个任务执行者。Npm 是在安装 Node.js 时附带的包管理器，Npm Script 则是 Npm 内置的一个功能，允许在 `package.json` 文件里面使用 `scripts` 字段定义任务：

```
{
  "scripts": {
    "dev": "node dev.js",
    "pub": "node build.js"
  }
```

```
}
```

里面的 scripts 字段是一个对象，每个属性对应一段 Shell 脚本，以上代码定义了两个任务：dev 和 pub。其底层实现原理是通过调用 Shell 去运行脚本命令，例如，执行 npm run pub 命令等同于执行 node build.js 命令。

Npm Script 的优点是内置，无须安装其他依赖。其缺点是功能太简单，虽然提供了 pre 和 post 两个钩子，但不能方便地管理多个任务之间的依赖。

1.2.2 Grunt

Grunt（https://gruntjs.com）和 Npm Script 类似，也是一个任务执行者。Grunt 有大量现成的插件封装了常见的任务，也能管理任务之间的依赖关系，自动化地执行依赖的任务，每个任务的具体执行代码和依赖关系写在配置文件 Gruntfile.js 里，例如：

```
module.exports = function(grunt) {
  // 所有插件的配置信息
  grunt.initConfig({
    // uglify 插件的配置信息
    uglify: {
      app_task: {
        files: {
          'build/app.min.js': ['lib/index.js', 'lib/test.js']
        }
      }
    },
    // watch 插件的配置信息
    watch: {
      another: {
        files: ['lib/*.js'],
      }
    }
  });
  // 告诉 Grunt 我们将使用这些插件
  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-contrib-watch');
  // 告诉 Grunt 我们在终端中启动 Grunt 时需要执行哪些任务
```

```
  grunt.registerTask('dev', ['uglify','watch']);
};
```

在项目根目录下执行命令 grunt dev，就会启动 JavaScript 文件压缩和自动刷新功能。

Grunt 的优点是：

- 灵活，它只负责执行我们定义的任务；
- 大量的可复用插件封装好了常见的构建任务。

Grunt 的缺点是集成度不高，要写很多配置后才可以用，无法做到开箱即用。

Grunt 相当于进化版的 Npm Script，它的诞生其实是为了弥补 Npm Script 的不足。

1.2.3 Gulp

Gulp（http://gulpjs.com）是一个基于流的自动化构建工具。除了可以管理和执行任务，还支持监听文件、读写文件。Gulp 被设计得非常简单，只通过下面 5 种方法就可以支持几乎所有构建场景：

- 通过 gulp.task 注册一个任务；
- 通过 gulp.run 执行任务；
- 通过 gulp.watch 监听文件的变化；
- 通过 gulp.src 读取文件；
- 通过 gulp.dest 写文件。

Gulp 的最大特点是引入了流的概念，同时提供了一系列常用的插件去处理流，流可以在插件之间传递，大致使用如下：

```
// 引入 Gulp
var gulp = require('gulp');
// 引入插件
var jshint = require('gulp-jshint');
var sass = require('gulp-sass');
var concat = require('gulp-concat');
```

```
var uglify = require('gulp-uglify');
// 编译 SCSS 任务
gulp.task('sass', function() {
  // 读取文件，通过管道喂给插件
  gulp.src('./scss/*.scss')
    // SCSS 插件将 scss 文件编译成 CSS 文件
    .pipe(sass())
    // 输出文件
    .pipe(gulp.dest('./css'));
});
// 合并压缩 JavaScript 文件
gulp.task('scripts', function() {
  gulp.src('./js/*.js')
    .pipe(concat('all.js'))
    .pipe(uglify())
    .pipe(gulp.dest('./dist'));
});
// 监听文件的变化
gulp.task('watch', function(){
  // 当 scss 文件被编辑时执行 SCSS 任务
  gulp.watch('./scss/*.scss', ['sass']);
  gulp.watch('./js/*.js', ['scripts']);
});
```

Gulp 的优点是好用又不失灵活，既可以单独完成构建，也可以和其他工具搭配使用。其缺点和 Grunt 类似，集成度不高，要写很多配置后才可以用，无法做到开箱即用。

可以将 Gulp 看作 Grunt 的加强版。相对于 Grunt，Gulp 增加了监听文件、读写文件、流式处理的功能。

1.2.4 Fis3

Fis3（https://fex.baidu.com/fis3/）是一个来自百度的优秀国产构建工具。相对于 Grunt、Gulp 这些只提供了基本功能的工具，Fis3 集成了 Web 开发中的常用构建功能，如下所述。

- 读写文件：通过 `fis.match` 读文件，`release` 配置文件的输出路径。

- 资源定位：解析文件之间的依赖关系和文件位置。
- 文件指纹：在通过 useHash 配置输出文件时为文件 URL 加上 md5 戳，来优化浏览器的缓存。
- 文件编译：通过 parser 配置文件解析器做文件转换，例如将 ES6 编译成 ES5。
- 压缩资源：通过 optimizer 配置代码压缩方法。
- 图片合并：通过 spriter 配置合并 CSS 里导入的图片到一个文件中，来减少 HTTP 请求数。

大致使用如下：

```
// 加 md5
fis.match('*.{js,css,png}', {
  useHash: true
});
// 通过 fis3-parser-typescript 插件可将 TypeScript 文件转换成 JavaScript 文件
fis.match('*.ts', {
  parser: fis.plugin('typescript')
});
// 对 CSS 进行雪碧图合并
fis.match('*.css', {
  // 为匹配到的文件分配属性 useSprite
  useSprite: true
});
// 压缩 JavaScript
fis.match('*.js', {
  optimizer: fis.plugin('uglify-js')
});
// 压缩 CSS
fis.match('*.css', {
  optimizer: fis.plugin('clean-css')
});
// 压缩图片
fis.match('*.png', {
  optimizer: fis.plugin('png-compressor')
});
```

可以看出 Fis3 很强大，内置了许多功能，无须做太多配置就能完成大量工作。

Fis3 的优点是集成了各种 Web 开发所需的构建功能，配置简单、开箱即用。其缺点是目前官方已经不再更新和维护，不支持最新版本的 Node.js。

Fis3 是一种专注于 Web 开发的完整解决方案，如果将 Grunt、Gulp 比作汽车的发动机，则可以将 Fis3 比作一辆完整的汽车。

1.2.5 Webpack

Webpack（https://webpack.js.org）是一个打包模块化 JavaScript 的工具，在 Webpack 里一切文件皆模块，通过 Loader 转换文件，通过 Plugin 注入钩子，最后输出由多个模块组合成的文件。Webpack 专注于构建模块化项目。

其官网的首页图很形象地展示了 Webpack 的定义，如图 1-1 所示。

图 1-1　Webpack 简介

一切文件如 JavaScript、CSS、SCSS、图片、模板，对于 Webpack 来说都是一个个模块，这样的好处是能清晰地描述各个模块之间的依赖关系，以方便 Webpack 对模块进行组合和打包。经过 Webpack 的处理，最终会输出浏览器能使用的静态资源。

Webpack 具有很大的灵活性，能配置处理文件的方式，使用方法大致如下：

```
module.exports = {
  // 所有模块的入口，Webpack 从入口开始递归解析出所有依赖的模块
  entry: './app.js',
  output: {
```

```
    // 将入口所依赖的所有模块打包成一个文件 bundle.js 输出
    filename: 'bundle.js'
  }
}
```

Webpack 的优点是：

- 专注于处理模块化的项目，能做到开箱即用、一步到位；
- 可通过 Plugin 扩展，完整好用又不失灵活；
- 使用场景不局限于 Web 开发；
- 社区庞大活跃，经常引入紧跟时代发展的新特性，能为大多数场景找到已有的开源扩展；
- 良好的开发体验。

Webpack 的缺点是只能用于采用模块化开发的项目。

1.2.6 Rollup

Rollup（https://rollupjs.org）是一个和 Webpack 很类似但专注于 ES6 的模块打包工具。它的亮点在于，能针对 ES6 源码进行 Tree Shaking，以去除那些已被定义但没被使用的代码并进行 Scope Hoisting，以减小输出文件的大小和提升运行性能。然而 Rollup 的这些亮点随后就被 Webpack 模仿和实现。由于 Rollup 的使用方法和 Webpack 差不多，所以这里就不详细介绍如何使用 Rollup 了，而是详细说明它们的差别：

- Rollup 是在 Webpack 流行后出现的替代品；
- Rollup 生态链还不完善，体验不如 Webpack；
- Rollup 的功能不如 Webpack 完善，但其配置和使用更简单；
- Rollup 不支持 Code Spliting，但好处是在打包出来的代码中没有 Webpack 那段模块的加载、执行和缓存的代码。

Rollup 在用于打包 JavaScript 库时比 Webpack 更有优势，因为其打包出来的代码更小、

更快。但它的功能不够完善，在很多场景下都找不到现成的解决方案。

1.2.7 为什么选择 Webpack

上面介绍的构建工具是按照它们诞生的时间排序的，它们是时代的产物，侧面反映出 Web 开发的发展趋势，如下所述：

- 在 Npm Script 和 Grunt 时代，Web 开发要做的事情变多，流程复杂，自动化思想被引入，用于简化流程；
- 在 Gulp 时代，开始出现一些新语言用于提高开发效率，流式处理思想的出现是为了简化文件转换的流程，例如将 ES5 转换成 ES6；
- 在 Webpack 时代，由于单页应用的流行，网页的功能和实现代码变得复杂、庞大，Web 开发向模块化改进。

这些构建工具都有各自的定位和专注点，它们之间既可以单独完成任务，也可以相互搭配来弥补各自的不足。在了解这些常见的构建工具后，我们需要根据自己的需求去判断应该如何选择和搭配它们才能更好地满足自己的需求。

经过多年的发展，Webpack 已经成为构建工具中的首选，这是有原因的：

- 大多数团队在开发新项目时会采用紧跟时代的技术，这些技术几乎都会采用“模块化+新语言+新框架”，Webpack 可以为这些新项目提供一站式的解决方案；
- Webpack 有良好的生态链和维护团队，能提供良好的开发体验并保证质量；
- Webpack 被全世界大量的 Web 开发者使用和验证，能找到各个层面所需的教程和经验分享。

下面，让我们开始进入 Webpack 的世界吧！

1.3 安装 Webpack

在用 Webpack 执行构建任务时，需要通过 webpack 可执行文件去启动构建任务，所以需要安装 webpack 可执行文件。在安装 Webpack 前请确保我们的系统安装了 5.0.0 及以上版本的 Node.js（https://nodejs.org）。

在开始为项目加入构建前，需要先新建一个 Web 项目，有如下方式：

- 新建一个目录，再进入项目根目录执行 npm init 来初始化最简单的采用了模块化开发的项目；
- 用脚手架工具 Yeoman（http://yeoman.io）直接、快速地生成一个最符合自己的需求的项目。

1.3.1 安装 Webpack 到本项目

安装 Webpack 到本项目时，可根据自己的需求选择以下任意命令运行：

```
# npm i -D 是 npm install --save-dev 的简写，是指安装模块并保存到 package.json 的 devDependencies
# 安装最新的稳定版
npm i -D webpack
# 安装指定版本
npm i -D webpack@<version>
# 安装最新的体验版本
npm i -D webpack@beta
```

安装完成后，我们可以通过以下途径运行安装到本项目的 Webpack：

- 在项目根目录下对应的命令行里通过 node_modules/.bin/webpack 运行 Webpack 的可执行文件。
- 在 Npm Script 里定义的任务会优先使用本项目下的 Webpack，代码如下：

```
json  "scripts": {      "start": "webpack --config webpack.config.js"   }
```

1.3.2 安装 Webpack 到全局

安装到全局后，我们可以在任何地方共用一个 Webpack 可执行文件，而不用各个项目重复安装，安装方式如下：

```
npm i -g webpack
```

虽然介绍了以上两种安装方式，但是我们推荐安装到本项目，原因是可防止不同的项目因依赖不同版本的 Webpack 而导致冲突。

1.3.3 使用 Webpack

下面通过 Webpack 构建一个采用了 CommonJS 模块化编写的项目，该项目中的某个网页会通过 JavaScript 显示 Hello,Webpack。

运行构建前，先将要完成该功能的最基础的 JavaScript 文件和 HTML 建立好，需要如下文件。

页面入口文件 index.html 如下：

```
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
<div id="app"></div>
<!--导入 Webpack 输出的 JavaScript 文件-->
<script src="./dist/bundle.js"></script>
</body>
</html>
```

存放工具函数的 show.js 文件的内容如下：

```
// 操作 DOM 元素，将 content 显示到网页上
function show(content) {
  window.document.getElementById('app').innerText = 'Hello,' + content;
}
// 通过 CommonJS 规范导出 show 函数
```

```
module.exports = show;
```

包含执行入口的 main.js 文件的内容如下：

```
// 通过 CommonJS 规范导入 show 函数
const show = require('./show.js');
// 执行 show 函数
show('Webpack');
```

Webpack 在执行构建时默认会从项目根目录下的 webpack.config.js 文件中读取配置，所以我们还需要新建它，其内容如下：

```
const path = require('path');
module.exports = {
  // JavaScript 执行入口文件
  entry: './main.js',
  output: {
    // 将所有依赖的模块合并输出到一个 bundle.js 文件
    filename: 'bundle.js',
    // 将输出文件都放到 dist 目录下
    path: path.resolve(__dirname, './dist'),
  }
};
```

由于 Webpack 构建运行在 Node.js 环境下，所以该文件最后需要通过 CommonJS 规范导出一个描述如何构建的 Object 对象。

此时，项目目录如下：

```
|-- index.html
|-- main.js
|-- show.js
|-- webpack.config.js
```

一切文件就绪，在项目根目录下执行 webpack 命令运行 Webpack 构建，我们会发现目录下多出一个 dist 目录，里面有个 bundle.js 文件，bundle.js 文件是一个可执行的 JavaScript 文件，它包含页面所依赖的两个模块 main.js、show.js，以及内置的 webpackBootstrap 启动函数。这时用浏览器打开 index.html 网页，将会看到 Hello, Webpack。

Webpack 是一个打包模块化 JavaScript 的工具，它会从 main.js 出发，识别出源码中的模块化导入语句，递归地找出入口文件的所有依赖，将入口和其所有依赖打包到一个单独的

文件中。从 Webpack 2 版本开始，Webpack 已经内置了对 ES6、CommonJS、AMD 模块化语句的支持。

至此我们已经学会了 Webpack 的基本功能，接下来我们将探索 Webpack 的更多功能。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/1-3 安装与使用.zip。

1.4 使用 Loader

在 1.3 节中使用 Webpack 构建了一个采用 CommonJS 规范的模块化项目，本节将继续优化这个网页的 UI，为项目引入 CSS 代码以让文字居中显示，main.css 的内容如下：

```
#app{
  text-align: center;
}
```

Webpack 将一切文件看作模块，CSS 文件也不例外。要引入 main.css，则需要像引入 JavaScript 文件那样，修改入口文件 main.js 如下：

```
// 通过 CommonJS 规范导入 CSS 模块
require('./main.css');
// 通过 CommonJS 规范导入 show 函数
const show = require('./show.js');
// 执行 show 函数
show('Webpack');
```

但是这样修改后去执行 Webpack 构建是会报错的，因为 Webpack 不原生支持解析 CSS 文件。要支持非 JavaScript 类型的文件，则需要使用 Webpack 的 Loader 机制。将 Webpack 的配置修改如下：

```
const path = require('path');
module.exports = {
  // JavaScript 执行入口文件
  entry: './main.js',
  output: {
    // 将所有依赖的模块合并输出到一个 bundle.js 文件中
```

```
    filename: 'bundle.js',
    // 将输出文件都放到 dist 目录下
    path: path.resolve(__dirname, './dist'),
  },
  module: {
    rules: [
      {
        // 用正则表达式去匹配要用该 Loader 转换的 CSS 文件
        test: /\.css$/,
        use: ['style-loader', 'css-loader?minimize'],
      }
    ]
  }
};
```

Loader 可以看作具有文件转换功能的翻译员，配置里的 module.rules 数组配置了一组规则，告诉 Webpack 在遇到哪些文件时使用哪些 Loader 去加载和转换。如上配置告诉 Webpack，在遇到以.css 结尾的文件时，先使用 css-loader 读取 CSS 文件，再由 style-loader 将 CSS 的内容注入 JavaScript 里。在配置 Loader 时需要注意：

- use 属性的值需要是一个由 Loader 名称组成的数组，Loader 的执行顺序是由后到前的；
- 每个 Loader 都可以通过 URL querystring 的方式传入参数，例如 css-loader?minimize 中的 minimize 告诉 css-loader 要开启 CSS 压缩。

想知道 Loader 具体支持哪些属性，则需要我们查阅文档，例如 css-loader 还有很多用法，我们可以在 css-loader 主页（https://github.com/webpack-contrib/css-loader）上查到。

在重新执行 Webpack 构建前，要先安装新引入的 Loader：

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

安装成功后重新执行构建时，我们会发现 bundle.js 文件被更新了，里面注入了在 main.css 中写的 CSS，而不会额外生成一个 CSS 文件。但是重新刷新 index.html 网页时，将会发现“Hello,Webpack”中了，样式生效了！也许你会对此感到奇怪，第一次看到 CSS 被写在了 JavaScript 里！这其实都是 style-loader 的功劳，它的工作原理大概是将 CSS 的内容用 JavaScript 里的字符串存储起来，在网页执行 JavaScript 时通过 DOM 操作，动态地

向 HTML head 标签里插入 HTML style 标签。也许你认为这样做会导致 JavaScript 文件变大并且加载网页的时间变长，想让 Webpack 单独输出 CSS 文件，这时你可以参考 1.5 节，1.5 节将讲解如何通过 Webpack Plugin 机制来实现。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/1-4 使用 Loader.zip。

向 Loader 传入属性的方式除了可以通过 querystring 实现，还可以通过 Object 实现，以上 Loader 配置可以修改为如下内容：

```
use: [
  'style-loader',
  {
    loader:'css-loader',
    options:{
      minimize:true,
    }
  }
]
```

除了在 webpack.config.js 配置文件中配置 Loader，还可以在源码中指定用什么 Loader 去处理文件。以加载 CSS 文件为例，修改上面例子中的 main.js 如下：

```
require('style-loader!css-loader?minimize!./main.css');
```

这样就能指定对 ./main.css 这个文件先采用 css-loader 再采用 style-loader 进行转换。

1.5 使用 Plugin

Plugin 是用来扩展 Webpack 功能的，通过在构建流程里注入钩子实现，它为 Webpack 带来了很大的灵活性。

在 1.4 节中通过 Loader 加载了 CSS 文件，本节通过 Plugin 将注入 bundle.js 文件里的 CSS 提取到单独的文件中，配置修改如下：

```
const path = require('path');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
module.exports = {
```

```
  // JavaScript 执行入口文件
  entry: './main.js',
  output: {
    // 将所有依赖的模块合并输出到一个 bundle.js 文件中
    filename: 'bundle.js',
    // 将输出文件都放到 dist 目录下
    path: path.resolve(__dirname, './dist'),
  },
  module: {
    rules: [
      {
        // 用正则去匹配要用该 loader 转换的 CSS 文件
        test: /\.css$/,
        loaders: ExtractTextPlugin.extract({
          // 转换.css 文件需要使用的 Loader
          use: ['css-loader'],
        }),
      }
    ]
  },
  plugins: [
    new ExtractTextPlugin({
      // 从.js 文件中提取出来的.css 文件的名称
      filename: `[name]_[contenthash:8].css`,
    }),
  ]
};
```

要让以上代码运行起来，需要先安装新引入的插件：

```
npm i -D extract-text-webpack-plugin
```

安装成功后重新执行构建，我们会发现 dist 目录下多出一个 main_1a87a56a.css 文件，bundle.js 文件里也没有 CSS 代码了，再将该 CSS 文件引入 index.html 里就完成了。

从以上代码可以看出，Webpack 是通过 plugins 属性来配置需要使用的插件列表的。plugins 属性是一个数组，里面的每一项都是插件的一个实例，在实例化一个组件时可以通过构造函数传入这个组件支持的配置属性。

例如，ExtractTextPlugin 插件的作用是提取出 JavaScript 代码里的 CSS 到一个单独的文件中。对此我们可以通过插件的 filename 属性，告诉插件输出的 CSS 文件名称是通过[name]_[contenthash:8].css 字符串模板生成的，里面的[name]代表文件的名称，[contenthash:8]代表根据文件内容算出的 8 位 Hash 值，还有很多配置选项可以在 ExtractTextPlugin（https://github.com/webpack-contrib/extract-text-webpack-plugin）的主页上查到。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/1-5 使用 Plugin.zip。

1.6 使用 DevServer

前面的几节只是让 Webpack 正常运行起来了，但在实际开发中我们可能会需要：

- 提供 HTTP 服务而不是使用本地文件预览；
- 监听文件的变化并自动刷新网页，做到实时预览；
- 支持 Source Map，以方便调试。

对于这些，Webpack 都为我们考虑好了。Webpack 原生支持上述第 2、3 点内容，再结合官方提供的开发工具 DevServer（https://webpack.js.org/configuration/dev-server/）也可以很方便地做到第 1 点。DevServer 会启动一个 HTTP 服务器用于服务网页请求，同时会帮助启动 Webpack，并接收 Webpack 发出的文件更变信号，通过 WebSocket 协议自动刷新网页做到实时预览。

下面为之前的小项目“Hello,Webpack”继续集成 DevServer。首先需要安装 DevServer：

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

安装成功后执行 webpack-dev-server 命令，DevServer 就启动了，这时我们会看到控制台有一串日志输出：

```
Project is running at http://localhost:8080/
webpack output is served from /
```

这意味着 DevServer 启动的 HTTP 服务器监听在 8080 端口，DevServer 启动后会一直驻留在后台保持运行，访问这个网址，就能获取项目根目录下的 index.html 了。用浏览器打开这个地址时我们会发现页面空白，错误的原因是./dist/bundle.js 加载 404 了。同时我们会发现并没有文件输出到 dist 目录，原因是 DevServer 会将 Webpack 构建出的文件保存在内存中，在要访问输出的文件时，必须通过 HTTP 服务访问。由于 DevServer 不会理会 webpack.config.js 里配置的 output.path 属性，所以要获取 bundle.js 的正确 URL 是 http://localhost:8080/bundle.js，对应的 index.html 应该修改为：

```
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
<div id="app"></div>
<!--导入 DevServer 输出的 JavaScript 文件-->
<script src="bundle.js"></script>
</body>
</html>
```

1.6.1 实时预览

接着上面的步骤，可以试试修改 main.js、main.css、show.js 中的任意文件，保存后我们会发现浏览器被自动刷新，运行出修改后的效果。

Webpack 在启动时可以开启监听模式，之后 Webpack 会监听本地文件系统的变化，在发生变化时重新构建出新的结果。Webpack 默认关闭监听模式，我们可以在启动 Webpack 时通过 webpack --watch 来开启监听模式。

通过 DevServer 启动的 Webpack 会开启监听模式，当发生变化时重新执行构建，然后通知 DevServer。DevServer 会让 Webpack 在构建出的 JavaScript 代码里注入一个代理客户端用于控制网页，网页和 DevServer 之间通过 WebSocket 协议通信，以方便 DevServer 主动向客户端发送命令。DevServer 在收到来自 Webpack 的文件变化通知时，通过注入的客户端控制

网页刷新。

如果尝试修改 index.html 文件并保存，则我们会发现这并不会触发以上机制，导致这个问题的原因是 Webpack 在启动时会以配置里的 entry 为入口去递归解析出 entry 所依赖的文件，只有 entry 本身和依赖的文件才会被 Webpack 添加到监听列表里。而 index.html 文件是脱离了 JavaScript 模块化系统的，所以 Webpack 不知道它的存在。

1.6.2 模块热替换

除了通过重新刷新整个网页来实现实时预览，DevServer 还有一种被称作模块热替换的刷新技术。模块热替换能做到在不重新加载整个网页的情况下，通过将已更新的模块替换老模块，再重新执行一次来实现实时预览。模块热替换相对于默认的刷新机制能提供更快的响应速度和更好的开发体验。模块热替换默认是关闭的，要开启模块热替换，我们只需在启动 DevServer 时带上--hot 参数，重启 DevServer 后再去更新文件就能体验到模块热替换的神奇了。

1.6.3 支持 Source Map

在浏览器中运行的 JavaScript 代码都是编译器输出的代码，这些代码的可读性很差。如果在开发过程中遇到一个不知道原因的 Bug，则我们可能需要通过断点调试去找出问题。在编译器输出的代码上进行断点调试是一件辛苦和不优雅的事情，调试工具可以通过 Source Map（https://www.html5rocks.com/en/tutorials/developertools/sourcemaps/）映射代码，让我们在源代码上断点调试。Webpack 支持生成 Source Map，只需在启动时带上--devtool source-map 参数。重启 DevServer 后刷新页面，再打开 Chrome 浏览器的开发者工具，就可以在 Sources 栏中看到可调试的源代码了，如图 1-2 所示。

图 1-2　在开发者工具中调试 Source Map

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/1-6 使用 DevServer.zip。

1.7　核心概念

通过之前几节的学习，相信我们已经对 Webpack 有了一个初步的认识。虽然 Webpack 功能强大且配置项多，但只要理解了其中的几个核心概念，就能随心应手地使用它。Webpack 有以下几个核心概念。

- Entry：入口，Webpack 执行构建的第一步将从 Entry 开始，可抽象成输入。
- Module：模块，在 Webpack 里一切皆模块，一个模块对应一个文件。Webpack 会从配置的 Entry 开始递归找出所有依赖的模块。
- Chunk：代码块，一个 Chunk 由多个模块组合而成，用于代码合并与分割。
- Loader：模块转换器，用于将模块的原内容按照需求转换成新内容。

- Plugin：扩展插件，在 Webpack 构建流程中的特定时机注入扩展逻辑，来改变构建结果或做我们想要的事情。
- Output：输出结果，在 Webpack 经过一系列处理并得出最终想要的代码后输出结果。

Webpack 在启动后会从 Entry 里配置的 Module 开始，递归解析 Entry 依赖的所有 Module。每找到一个 Module，就会根据配置的 Loader 去找出对应的转换规则，对 Module 进行转换后，再解析出当前 Module 依赖的 Module。这些模块会以 Entry 为单位进行分组，一个 Entry 及其所有依赖的 Module 被分到一个组也就是一个 Chunk。最后，Webpack 会将所有 Chunk 转换成文件输出。在整个流程中，Webpack 会在恰当的时机执行 Plugin 里定义的逻辑。

在实际应用中我们可能会遇到各种奇怪、复杂的场景，不知道从哪开始。根据以上总结，我们已对 Webpack 有了一个整体认识，这能让我们在以后使用 Webpack 时快速知道应该通过配置什么去完成自己想要的功能，而不是无从下手。

希望你能记住这 6 个核心概念，因为它们会在后面的章节中大量出现。

第 2 章将具体介绍 Webpack 常用的配置项及其具体含义。

第 2 章

配　置

第 1 章只是粗略讲解了 Webpack 的基础核心功能，本章会列举 Webpack 的常用功能所提供的配置选项，可以作为速查表使用。

配置 Webpack 的方式有如下两种：

（1）通过一个 JavaScript 文件描述配置，例如使用 webpack.config.js 文件里的配置；

（2）执行 Webpack 可执行文件时通过命令行参数传入，例如 webpack--devtool source-map。

这两种方式可以相互搭配，例如执行 Webpack 时通过命令 webpack--config webpack-dev.config.js 指定配置文件，再去 webpack-dev.config.js 文件里描述部分配置。

按照配置方式来划分，可分为：

- 只能通过命令行参数传入的选项，这种最为少见；
- 只能通过配置文件配置的选项；
- 通过两种方式都可以配置的选项。

按照配置所影响的功能来划分，可分为如下内容。

- Entry：配置模块的入口。

- Output：配置如何输出最终想要的代码。
- Module：配置处理模块的规则。
- Resolve：配置寻找模块的规则。
- Plugins：配置扩展插件。
- DevServer：配置 DevServer。
- 其他配置项：其他零散的配置项。
- 整体配置结构：整体地描述各配置项的结构。
- 多种配置类型：配置文件不止可以返回一个 Object，还可以返回其他形式。
- 配置总结：寻找配置 Webpack 的规律，减少思维负担。

2.1 Entry

entry 是配置模块的入口，可抽象成输入，Webpack 执行构建的第一步将从入口开始，搜寻及递归解析出所有入口依赖的模块。

entry 配置是必填的，若不填则将导致 Webpack 报错、退出。

2.1.1 context

Webpack 在寻找相对路径的文件时会以 context 为根目录，context 默认为执行启动 Webpack 时所在的当前工作目录。如果想改变 context 的默认配置，则可以在配置文件里这样设置它：

```
module.exports = {
  context: path.resolve(__dirname, 'app')
}
```

注意，context 必须是一个绝对路径的字符串。除此之外，还可以通过在启动 Webpack 时带上参数 webpack --context 来设置 context。

之所以在这里先介绍 context，是因为 Entry 的路径及其依赖的模块的路径可能采用相对于 context 的路径来描述，context 会影响到这些相对路径所指向的真实文件。

2.1.2　Entry 类型

Entry 类型可以是以下三种中的一种或者相互组合，如表 2-1 所示。

表 2-1　Entry 类型列表

类　型	例　子	含　义
string	`'./app/entry'`	入口模块的文件路径，可以是相对路径
array	`['./app/entry1', './app/entry2']`	入口模块的文件路径，可以是相对路径
object	`{ a: './app/entry-a', b: ['./app/entry-b1', './app/entry-b2']}`	配置多个入口，每个入口生成一个 Chunk

如果是 array 类型，则搭配 output.library 配置项使用时，只有数组里的最后一个入口文件的模块会被导出。

2.1.3　Chunk 的名称

Webpack 会为每个生成的 Chunk 取一个名称，Chunk 的名称和 Entry 的配置有关。

- 如果 entry 是一个 string 或 array，就只会生成一个 Chunk，这时 Chunk 的名称是 main。
- 如果 entry 是一个 object，就可能会出现多个 Chunk，这时 Chunk 的名称是 object 键值对中键的名称。

2.1.4 配置动态 Entry

假如项目里有多个页面需要为每个页面的入口配置一个 Entry，但这些页面的数量可能会不断增长，则这时 Entry 的配置会受到其他因素的影响，导致不能写成静态的值。其解决方法是将 Entry 设置成一个函数动态地返回上面所说的配置，代码如下：

```
// 同步函数
entry: () => {
  return {
    a:'./pages/a',
    b:'./pages/b',
  }
};
// 异步函数
entry: () => {
  return new Promise((resolve)=>{
    resolve({
      a:'./pages/a',
      b:'./pages/b',
    });
  });
};
```

2.2 Output

output 配置如何输出最终想要的代码。output 是一个 object，里面包含一系列配置项，下面分别介绍它们。

2.2.1 filename

output.filename 配置输出文件的名称，为 string 类型。如果只有一个输出文件，则可以将它写成静态不变的：

```
filename: 'bundle.js'
```

但是在有多个 Chunk 要输出时，就需要借助模板和变量了。前面讲到，Webpack 会为每个 Chunk 取一个名称，所以我们可以根据 Chunk 的名称来区分输出的文件名：

```
filename: '[name].js'
```

代码里的[name]代表用内置的 name 变量去替换[name]，这时我们可以将它看作一个字符串模块函数，每个要输出的 Chunk 都会通过这个函数去拼接出输出的文件名称。
内置变量除了包括 name，还包括如表 2-2 所示的变量。

表 2-2　内置变量列表

变量名	含　义
id	Chunk 的唯一标识，从 0 开始
name	Chunk 的名称
hash	Chunk 的唯一标识的 Hash 值
chunkhash	Chunk 内容的 Hash 值

其中，hash 和 chunkhash 的长度是可指定的，[hash:8]代表取 8 位 Hash 值，默认是 20 位。

注意，ExtractTextWebpackPlugin（https://github.com/webpack-contrib/extract-text-webpack-plugin）插件使用 contenthash 而不是 chunkhash 来代表哈希值，原因在于 ExtractTextWebpackPlugin 提取出来的内容是代码内容本身，而不是由一组模块组成的 Chunk。

2.2.2　chunkFilename

output.chunkFilename 配置无入口的 Chunk 在输出时的文件名称。chunkFilename 和上面的 filename 非常类似，但 chunkFilename 只用于指定在运行过程中生成的 Chunk 在输出时的文件名称。会在运行时生成 Chunk 的常见场景包括：使用 CommonChunkPlugin、使用 import('path/to/module')动态加载等。chunkFilename 支持和 filename 一致的内置变量。

2.2.3 path

output.path 配置输出文件存放在本地的目录，必须是 string 类型的绝对路径。通常通过 Node.js 的 path 模块去获取绝对路径：

```
path: path.resolve(__dirname, 'dist_[hash]')
```

2.2.4 publicPath

在复杂的项目里可能会有一些构建出的资源需要异步加载，加载这些异步资源需要对应的 URL 地址。

output.publicPath 配置发布到线上资源的 URL 前缀，为 string 类型。默认值是空字符串''，即使用相对路径。

这样说可能有点抽象，举个例子，需要将构建出的资源文件上传到 CDN 服务上，以利于加快页面的打开速度。配置代码如下：

```
filename:'[name]_[chunkhash:8].js'
publicPath: 'https://cdn.example.com/assets/'
```

这时发布到线上的 HTML 在引入 JavaScript 文件时就需要以下配置项：

```
<script src='https://cdn.example.com/assets/a_12345678.js'>
</script>
```

使用该配置项时要小心，稍有不慎将导致资源加载 404 错误。

output.path 和 output.publicPath 都支持字符串模板，内置变量只有一个，即 hash，代表一次编译操作的 Hash 值。

2.2.5 crossOriginLoading

Webpack 输出的部分代码块可能需要异步加载，而异步加载是通过 JSONP（https://zh.wikipedia.org/wiki/JSONP）方式实现的。JSONP 的原理是动态地向 HTML 中插入一个<script

src="url"></script>标签去加载异步资源。output.crossOriginLoading 则是用于配置这个异步插入的标签的 crossorigin 值。

script 标签的 crossorigin 属性可以取以下值：

- anonymous（默认），在加载此脚本资源时不会带上用户的 Cookies。
- use-credentials，在加载此脚本资源时会带上用户的 Cookies。

通常用设置 crossorigin 来获取异步加载的脚本执行时的详细错误信息。

2.2.6 libraryTarget 和 library

当用 Webpack 去构建一个可以被其他模块导入使用的库时，需要用到 libraryTarget 和 library。

- output.libraryTarget 配置以何种方式导出库。
- output.library 配置导出库的名称。

它们通常搭配在一起使用。

output.libraryTarget 是字符串的枚举类型，支持以下配置。

1. var（默认）

编写的库将通过 var 被赋值给通过 library 指定名称的变量。

假如配置了 output.library='LibraryName'，则输出和使用的代码如下：

```
// Webpack 输出的代码
var LibraryName = lb_code;
// 使用库的方法
LibraryName.doSomething();
```

假如 output.library 为空，则直接输出：

```
lib_code
```

其中，lib_code 是指导出库的代码内容，是有返回值的一个自执行函数。

2. commonjs

编写的库将通过 CommonJS 规范导出。

假如配置了 output.library='LibraryName'，则输出和使用的代码如下：

```
// Webpack 输出的代码
exports['LibraryName'] = lib_code;
// 使用库的方法
require('library-name-in-npm')['LibraryName'].doSomething();
```

其中，library-name-in-npm 是指模块被发布到 Npm 代码仓库时的名称。

3. commonjs2

编写的库将通过 CommonJS2 规范导出，输出和使用的代码如下：

```
// Webpack 输出的代码
module.exports = lib_code;
// 使用库的方法
require('library-name-in-npm').doSomething();
```

CommonJS2 和 CommonJS 规范相似，差别在于 CommonJS 只能用 exports 导出，而 CommonJS2 在 CommonJS 的基础上增加了 module.exports 的导出方式。

在 output.libraryTarget 为 commonjs2 时，配置 output.library 将没有意义。

4. this

编写的库将通过 this 被赋值给通过 library 指定的名称，输出和使用的代码如下：

```
// Webpack 输出的代码
this['LibraryName'] = lib_code;
// 使用库的方法
this.LibraryName.doSomething();
```

5. window

编写的库将通过 window 赋值给通过 library 指定的名称，输出和使用的代码如下：

```
// Webpack 输出的代码
window['LibraryName'] = lib_code;
// 使用库的方法
window.LibraryName.doSomething();
```

6. global

编写的库将通过 window 赋值给通过 library 指定的名称，输出和使用的代码如下：

```
// Webpack 输出的代码
global['LibraryName'] = lib_code;
// 使用库的方法
global.LibraryName.doSomething();
```

2.2.7 libraryExport

output.libraryExport 配置要导出的模块中哪些子模块需要被导出。它只有在 output.libraryTarget 被设置成 commonjs 或者 commonjs2 时使用才有意义。

假如要导出的模块源代码是：

```
export const a=1;
export default b=2;
```

而现在想让构建输出的代码只导出其中的 a，则可以将 output.libraryExport 设置成 a，那么构建输出的代码和使用方法将变成如下内容：

```
// Webpack 输出的代码
module.exports = lib_code['a'];
// 使用库的方法
require('library-name-in-npm')===1;
```

以上只是 output 中的常用配置项，还有部分几乎用不上的配置项没有在这里一一列举，可以在 Webpack 官方文档（https://webpack.js.org/configuration/output/）上查阅它们。

2.3 Module

module 配置处理模块的规则，下面对它进行详细讲解。

2.3.1 配置 Loader

rules 配置模块的读取和解析规则，通常用来配置 Loader。其类型是一个数组，数组里的每一项都描述了如何处理部分文件。配置一项 rules 时大致可通过以下方式来完成。

- 条件匹配：通过 test、include、exclude 三个配置项来选中 Loader 要应用规则的文件。
- 应用规则：对选中的文件通过 use 配置项来应用 Loader，可以只应用一个 Loader 或者按照从后往前的顺序应用一组 Loader，同时可以分别向 Loader 传入参数。
- 重置顺序：一组 Loader 的执行顺序默认是从右到左执行的，通过 enforce 选项可以将其中一个 Loader 的执行顺序放到最前或者最后。

下面通过一个例子来说明具体的使用方法：

```
module: {
  rules: [
    {
      // 命中 JavaScript 文件
      test: /\.js$/,
      // 用 babel-loader 转换 JavaScript 文件
      // ?cacheDirectory 表示传给 babel-loader 的参数，用于缓存 babel 的编译结果，
加快重新编译的速度
      use: ['babel-loader?cacheDirectory'],
      // 只命中 src 目录里的 JavaScript 文件，加快 Webpack 的搜索速度
      include: path.resolve(__dirname, 'src')
    },
    {
      // 命中 SCSS 文件
```

```
        test: /\.scss$/,
        // 使用一组 Loader 去处理 SCSS 文件
        // 处理顺序为从后到前，即先交给 sass-loader 处理，再将结果交给 css-loader，
最后交给 style-loader
        use: ['style-loader', 'css-loader', 'sass-loader'],
        // 排除 node_modules 目录下的文件
        exclude: path.resolve(__dirname, 'node_modules'),
      },
      {
        // 对非文本文件采用 file-loader 加载
        test: /\.(gif|png|jpe?g|eot|woff|ttf|svg|pdf)$/,
        use: ['file-loader'],
      },
    ]
  }
```

在 Loader 需要传入很多参数时，我们还可以通过一个 Object 来描述，例如在上面的 babel-loader 配置中有如下代码：

```
use: [
  {
    loader:'babel-loader',
    options:{
      cacheDirectory:true,
    },
    // enforce:'post'的含义是将该 Loader 的执行顺序放到最后
    // enforce 的值还可以是 pre，代表将 Loader 的执行顺序放到最前面
    enforce:'post'
  },
  // 省略其他 Loader
]
```

在上面的例子中，test、include、exclude 这三个命中文件的配置项只传入了一个字符串或正则，其实它们也支持数组类型，使用如下：

```
{
  test:[
    /\.jsx?$/,
    /\.tsx?$/
  ],
```

```
    include:[
      path.resolve(__dirname, 'src'),
      path.resolve(__dirname, 'tests'),
    ],
    exclude:[
      path.resolve(__dirname, 'node_modules'),
      path.resolve(__dirname, 'bower_modules'),
    ]
  }
```

数组里的每项之间是“或”的关系，即文件的路径只要满足数组中的任何一个条件，就会被命中。

2.3.2 noParse

noParse 配置项可以让 Webpack 忽略对部分没采用模块化的文件的递归解析和处理，这样做的好处是能提高构建性能。原因是一些库如 jQuery、ChartJS 庞大又没有采用模块化标准，让 Webpack 去解析这些文件既耗时又没有意义。

noParse 是可选的配置项，类型需要是 RegExp、[RegExp]、function 中的一种。

例如，若想要忽略 jQuery、ChartJS，则可以使用如下代码：

```
// 使用正则表达式
noParse: /jquery|chartjs/
// 使用函数，从 Webpack 3.0.0 开始支持
noParse: (content)=> {
  // content 代表一个模块的文件路径
  // 返回 true 或 false
  return /jquery|chartjs/.test(content);
}
```

注意，被忽略的文件里不应该包含 import、require、define 等模块化语句，不然会导致在构建出的代码中包含无法在浏览器环境下执行的模块化语句。

2.3.3 parser

因为 Webpack 是以模块化的 JavaScript 文件为入口的，所以内置了对模块化 JavaScript 的解析功能，支持 AMD、CommonJS、SystemJS、ES6。parser 属性可以更细粒度地配置哪些模块语法被解析、哪些不被解析。同 noParse 配置项的区别在于，parser 可以精确到语法层面，而 noParse 只能控制哪些文件不被解析。parser 的使用方法如下：

```
module: {
  rules: [
    {
      test: /\.js$/,
      use: ['babel-loader'],
      parser: {
      amd: false, // 禁用 AMD
      commonjs: false, // 禁用 CommonJS
      system: false, // 禁用 SystemJS
      harmony: false, // 禁用 ES6 import/export
      requireInclude: false, // 禁用 require.include
      requireEnsure: false, // 禁用 require.ensure
      requireContext: false, // 禁用 require.context
      browserify: false, // 禁用 browserify
      requireJs: false, // 禁用 requirejs
      }
    },
  ]
}
```

2.4 Resolve

Webpack 在启动后会从配置的入口模块出发找出所有依赖的模块，Resolve 配置 Webpack 如何寻找模块所对应的文件。Webpack 内置 JavaScript 模块化语法解析功能，默认会采用模块化标准里约定的规则去寻找，但我们也可以根据自己的需要修改默认的规则。

2.4.1 alias

resolve.alias 配置项通过别名来将原导入路径映射成一个新的导入路径。例如使用以下配置：

```
// Webpack alias 配置
resolve:{
  alias:{
    components: './src/components/'
  }
}
```

当通过 import Button from 'components/button'导入时，实际上被 alias 等价替换成了 import Button from './src/components/button'。

以上 alias 配置的含义是，将导入语句里的 components 关键字替换成./src/components/。

这样做可能会命中太多导入语句，alias 还支持通过$符号来缩小范围到只命中以关键字结尾的导入语句：

```
resolve:{
  alias:{
    'react$': '/path/to/react.min.js'
  }
}
```

react$只会命中以 react 结尾的导入语句，即只会将 import 'react'关键字替换成 import '/path/to/react.min.js'。

2.4.2 mainFields

有一些第三方模块会针对不同的环境提供几份代码。例如分别提供采用了 ES5 和 ES6 的两份代码，这两份代码的位置写在 package.json 文件里，代码如下：

```
{
  "jsnext:main": "es/index.js",// 采用 ES6 语法的代码入口文件
```

```
  "main": "lib/index.js" // 采用 ES5 语法的代码入口文件
}
```

Webpack 会根据 mainFields 的配置去决定优先采用哪份代码，mainFields 默认如下：

```
mainFields: ['browser', 'main']
```

Webpack 会按照数组里的顺序在 package.json 文件里寻找，只会使用找到的第 1 个文件。

假如我们想优先采用 ES6 的那份代码，则可以这样配置：

```
mainFields: ['jsnext:main', 'browser', 'main']
```

2.4.3　extensions

在导入语句没带文件后缀时，Webpack 会自动带上后缀后去尝试访问文件是否存在。resolve.extensions 用于配置在尝试过程中用到的后缀列表，默认是：

```
extensions: ['.js', '.json']
```

也就是说，当遇到 require('./data') 这样的导入语句时，Webpack 会先寻找 ./data.js 文件，如果该文件不存在，就去寻找 ./data.json 文件，如果还是找不到，就报错。

假如我们想让 Webpack 优先使用目录下的 TypeScript 文件，则可以这样配置：

```
extensions: ['.ts', '.js', '.json']
```

2.4.4　modules

resolve.modules 配置 Webpack 去哪些目录下寻找第三方模块，默认只会去 node_modules 目录下寻找。有时我们的项目里会有一些模块被其他模块大量依赖和导入，由于其他模块的位置不定，针对不同的文件都要计算被导入的模块文件的相对路径，这个路径有时会很长，就像 import '../../../components/button'，这时可以利用 modules 配置项优化。假如那些被大量导入的模块都在 ./src/components 目录下，则将 modules

配置成 modules: ['./src/components','node_modules']后，可以简单地通过 import 'button'导入。

2.4.5 descriptionFiles

resolve.descriptionFiles 配置描述第三方模块的文件名称，也就是 package.json 文件。默认如下：

```
descriptionFiles: ['package.json']
```

2.4.6 enforceExtension

如果 resolve.enforceExtension 被配置为 true，则所有导入语句都必须带文件后缀，例如开启前 import './foo'能正常工作，开启后就必须写成 import './foo.js'。

2.4.7 enforceModuleExtension

enforceModuleExtension 和 enforceExtension 的作用类似，但 enforceModuleExtension 只对 node_modules 下的模块生效。enforceModuleExtension 通常搭配 enforceExtension 使用，在 enforceExtension:true 时，因为安装的第三方模块中大多数导入语句都没带文件的后缀，所以这时通过配置 enforceModuleExtension:false 来兼容第三方模块。

2.5 Plugin

Plugin 用于扩展 Webpack 的功能，各种各样的 Plugin 几乎可以让 Webpack 做任何与构建相关的事情。

Plugin 的配置很简单，plugins 配置项接收一个数组，数组里的每一项都是一个要使用的 Plugin 的实例，Plugin 需要的参数通过构造函数传入。

```
const CommonsChunkPlugin = require('webpack/lib/optimize/
CommonsChunkPlugin');
module.exports = {
  plugins: [
    // 所有页面都会用到的公共代码被提取到 common 代码块中
    new CommonsChunkPlugin({
      name: 'common',
      chunks: ['a', 'b']
    }),
  ]
};
```

使用 Plugin 的难点在于掌握 Plugin 本身提供的配置项，而不是如何在 Webpack 中接入 Plugin。

几乎所有 Webpack 无法直接实现的功能都能在社区找到开源的 Plugin 去解决，我们需要善于使用搜索引擎去寻找解决问题的方法。

2.6 DevServer

在 1.6 节介绍过用来提高开发效率的 DevServer，它提供的一些配置项可以用于改变 DevServer 的默认行为。要配置 DevServer，除了可以在配置文件里通过 devServer 传入参数，还可以通过命令行参数传入。注意，只有在通过 DevServer 启动 Webpack 时，配置文件里的 devServer 才会生效，因为这些参数所对应的功能都是 DevServer 提供的，Webpack 本身并不认识 devServer 配置项。

2.6.1 hot

devServer.hot 配置是否启用 1.6 节提到的模块热替换功能。DevServer 的默认行为是在发现源代码被更新后通过自动刷新整个页面来做到实时预览，开启模块热替换功能后，将

在不刷新整个页面的情况下通过用新模块替换老模块来做到实时预览。

2.6.2 inline

DevServer 的实时预览功能依赖一个注入页面里的代理客户端，去接收来自 DevServer 的命令并负责刷新网页的工作。`devServer.inline` 用于配置是否将这个代理客户端自动注入将运行在页面中的 Chunk 里，默认自动注入。DevServer 会根据我们是否开启 `inline` 来调整它的自动刷新策略。

- 如果开启 `inline`，则 DevServer 会在构建变化后的代码时通过代理客户端控制网页刷新。
- 如果关闭 `inline`，则 DevServer 将无法直接控制要开发的网页。这时它会通过 iframe 的方式去运行要开发的网页。在构建完变化后的代码时，会通过刷新 `iframe` 来实现实时预览，但这时我们需要去 `http://localhost:8080/webpack-dev-server/`实时预览自己的网页。

如果想使用 DevServer 的模块热替换机制去实现实时预览，则最方便的方法是直接开启 `inline`。

2.6.3 historyApiFallback

`devServer.historyApiFallback` 用于方便地开发使用了 HTML5 History API（https://developer.mozilla.org/en-US/docs/Web/API/History）的单页应用。这类单页应用要求服务器在针对任何命中的路由时，都返回一个对应的 HTML 文件。例如在访问 `http://localhost/user` 和 `http://localhost/home` 时都返回 `index.html` 文件，浏览器端的 JavaScript 代码会从 URL 里解析出当前页面的状态，显示对应的界面。

配置 `historyApiFallback` 的最简单做法是：

```
historyApiFallback: true
```

这会导致任何请求都会返回 index.html 文件，这只能用于只有一个 HTML 文件的应用。

如果我们的应用由多个单页应用组成，则需要 DevServer 根据不同的请求返回不同的 HTML 文件，配置如下：

```
historyApiFallback: {
  // 使用正则匹配命中路由
  rewrites: [
    // /user 开头的都返回 user.html
    { from: /^\/user/, to: '/user.html' },
    { from: /^\/game/, to: '/game.html' },
    // 其他的都返回 index.html
    { from: /./, to: '/index.html' },
  ]
}
```

2.6.4 contentBase

devServer.contentBase 配置 DevServer HTTP 服务器的文件根目录。在默认情况下为当前的执行目录，通常是项目根目录，所以在一般情况下不必设置它，除非有额外的文件需要被 DevServer 服务。例如，若想将项目根目录下的 public 目录设置成 DevServer 服务器的文件根目录，则可以这样配置：

```
devServer:{
  contentBase: path.join(__dirname, 'public')
}
```

这里解释一下可能会让我们感到疑惑的地方。DevServer 服务器通过 HTTP 服务暴露文件的方式可分为两类：

- 暴露本地文件；
- 暴露 Webpack 构建出的结果，由于构建出的结果交给了 DevServer，所以我们在使用 DevServer 时，会在本地找不到构建出的文件。

contentBase 只能用来配置暴露本地文件的规则，可以通过 contentBase:false 来关闭暴露本地文件。

2.6.5 headers

devServer.headers 配置项可以在 HTTP 响应中注入一些 HTTP 响应头，使用如下：

```
devServer:{
  headers: {
    'X-foo':'bar'
  }
}
```

2.6.6 host

devServer.host 配置项用于配置 DevServer 服务监听的地址，只能通过命令行参数传入。例如，若想让局域网中的其他设备访问自己的本地服务，则可以在启动 DevServer 时带上 --host 0.0.0.0。host 的默认值是 127.0.0.1，即只有本地可以访问 DevServer 的 HTTP 服务。

2.6.7 port

devServer.host 配置项用于配置 DevServer 服务监听的端口，默认使用 8080 端口。如果 8080 端口已经被其他程序占用，就使用 8081；如果 8081 还是被占用，则使用 8082，以此类推。

2.6.8 allowedHosts

devServer.allowedHosts 配置一个白名单列表，只有 HTTP 请求的 HOST 在列表里才正常返回，使用如下：

```
allowedHosts: [
  // 匹配单个域名
```

```
    'host.com',
    'sub.host.com',
    // host2.com 和所有的子域名 *.host2.com 都将匹配
    '.host2.com'
  ]
```

2.6.9 disableHostCheck

devServer.disableHostCheck 配置项用于配置是否关闭用于 DNS 重新绑定的 HTTP 请求的 HOST 检查。DevServer 默认只接收来自本地的请求，关闭后可以接收来自任意 HOST 的请求。它通常用于搭配--host 0.0.0.0 使用，因为想让其他设备访问自己的本地服务，但访问时是直接通过 IP 地址访问而不是通过 HOST 访问，所以需要关闭 HOST 检查。

2.6.10 https

DevServer 默认使用 HTTP 服务，它也能使用 HTTPS 服务。在某些情况下我们必须使用 HTTPS，例如 HTTP2 和 Service Worker 就必须运行在 HTTPS 上。要切换成 HTTPS 服务，最简单的方式是：

```
devServer:{
  https: true
}
```

DevServer 会自动为我们生成一份 HTTPS 证书。

如果我们想用自己的证书，则可以这样配置：

```
devServer:{
  https: {
    key: fs.readFileSync('path/to/server.key'),
    cert: fs.readFileSync('path/to/server.crt'),
    ca: fs.readFileSync('path/to/ca.pem')
  }
}
```

2.6.11 clientLogLevel

devServer.clientLogLevel 配置客户端的日志等级，这会影响到我们在浏览器开发者工具控制台里看到的日志内容。clientLogLevel 是枚举类型，可取如下值之一：none、error、warning、info。默认为 info 级别，即输出所有类型的日志，设置成 none 时可以不输出任何日志。

2.6.12 compress

devServer.compress 配置是否启用 Gzip 压缩，为 boolean 类型，默认为 false。

2.6.13 open

devServer.open 用于在 DevServer 启动且第一次构建完时，自动用我们的系统的默认浏览器去打开要开发的网页。还提供了 devServer.openPage 配置项来打开指定 URL 的网页。

2.7 其他配置项

除了前面介绍到的配置项，Webpack 还提供了一些零散的配置项。下面介绍这些配置项中的常用部分。

2.7.1 Target

JavaScript 的应用场景越来越多，从浏览器到 Node.js，这些运行在不同环境中的 JavaScript 代码存在一些差异。target 配置项可以让 Webpack 构建出针对不同运行环境的代码。target 可以是如表 2-3 所示的值之一。

表 2-3　target 的值列表

target 值	描　述
web	针对浏览器（默认），所有代码都集中在一个文件里
node	针对 Node.js，使用 require 语句加载 Chunk 代码
`async-node`	针对 Node.js，异步加载 Chunk 代码
`webworker`	针对 WebWorker
`electron-main`	针对 Electron（http://electron.atom.io/）主线程
`electron-renderer`	针对 Electron 渲染线程

例如，在设置 `target:'node'`时，在源代码中导入 Node.js 原生模块的语句 `require('fs')`将会被保留，`fs` 模块的内容不会被打包到 Chunk 里。

2.7.2　Devtool

`devtool` 配置 Webpack 如何生成 Source Map，默认值是 `false`，即不生成 Source Map，若想为构建出的代码生成 Source Map 以方便调试，则可以这样配置：

```
module.export = {
  devtool: 'source-map'
}
```

2.7.3　Watch 和 WatchOptions

前面介绍过 Webpack 的监听模式，它支持监听文件更新，在文件发生变化时重新编译。在使用 Webpack 时，监听模式默认是关闭的，若想打开，则需要如下配置：

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

在使用 DevServer 时，监听模式默认是开启的。

除此之外，Webpack 还提供了 watchOptions 配置项去更灵活地控制监听模式，使用如下：

```
module.export = {
  // 只有在开启监听模式时，watchOptions 才有意义
  // 默认为 false，也就是不开启
  watch: true,
  // 监听模式运行时的参数
  // 在开启监听模式时，才有意义
  watchOptions: {
    // 不监听的文件或文件夹，支持正则匹配
    // 默认为空
    ignored: /node_modules/,
    // 监听到变化后会等 300ms 再去执行动作，防止文件更新太快导致重新编译频率太高
    // 默认为 300ms
    aggregateTimeout: 300,
    // 判断文件是否发生变化是通过不停地询问系统指定文件有没有变化实现的
    // 默认每秒询问 1000 次
    poll: 1000
  }
}
```

2.7.4 Externals

Externals 用来告诉在 Webpack 要构建的代码中使用了哪些不用被打包的模块，也就是说这些模板是外部环境提供的，Webpack 在打包时可以忽略它们。

有些 JavaScript 运行环境可能内置了一些全局变量或者模块，例如在我们的 HTML HEAD 标签里通过以下代码引入 jQuery：

```
<script src="path/to/jquery.js"></script>
```

这时，全局变量 jQuery 就会被注入网页的 JavaScript 运行环境里。

如果想在使用模块化的源代码里导入和使用 jQuery，则可能需要这样：

```
import $ from 'jquery';
$('.my-element');
```

构建后我们会发现输出的 Chunk 里包含的 jQuery 库的内容，这导致 jQuery 库出现了两

次，浪费加载流量，最好是 Chunk 里不会包含 jQuery 库的内容。

Externals 配置项就是用于解决这个问题的。

通过 externals 可以告诉 Webpack 在 JavaScript 运行环境中已经内置了哪些全局变量，不用将这些全局变量打包到代码中而是直接使用它们。要解决以上问题，可以这样配置 externals：

```
module.export = {
  externals: {
    // 将导入语句里的 jquery 替换成运行环境里的全局变量 jQuery
    jquery: 'jQuery'
  }
}
```

2.7.5 ResolveLoader

ResolveLoader 用来告诉 Webpack 如何去寻找 Loader，因为在使用 Loader 时是通过其包名称去引用的，Webpack 需要根据配置的 Loader 包名去找到 Loader 的实际代码，以调用 Loader 去处理源文件。

ResolveLoader 的默认配置如下：

```
module.exports = {
  resolveLoader:{
    // 去哪个目录下寻找 Loader
    modules: ['node_modules'],
    // 入口文件的后缀
    extensions: ['.js', '.json'],
    // 指明入口文件位置的字段
    mainFields: ['loader', 'main']
  }
}
```

该配置项常用于加载本地的 Loader。

2.8 整体配置结构

之前的章节分别讲述了每个配置项的具体含义，但没有描述它们所处的位置和数据结构，下面通过一份代码来描述清楚：

```
const path = require('path');
module.exports = {
  // entry 表示入口，Webpack 执行构建的第一步将从 Entry 开始，可抽象成输入
  // 类型可以是 string、object、array
  entry: './app/entry', // 只有 1 个入口，入口只有 1 个文件
  entry: ['./app/entry1', './app/entry2'], // 只有 1 个入口，入口有两个文件
  entry: { // 有两个入口
    a: './app/entry-a',
    b: ['./app/entry-b1', './app/entry-b2']
  },
  // 如何输出结果：在 Webpack 经过一系列处理后，如何输出最终想要的代码
  output: {
    // 输出文件存放的目录，必须是 string 类型的绝对路径
    path: path.resolve(__dirname, 'dist'),

    // 输出文件的名称
    filename: 'bundle.js', // 完整的名称
    filename: '[name].js', // 在配置了多个 entry 时，通过名称模板为不同的 entry 生成不同的文件名称
    filename: '[chunkhash].js', // 根据文件内容的 Hash 值生成文件的名称，用于浏览器长时间缓存文件
    // 发布到线上的所有资源的 URL 前缀，为 string 类型
    publicPath: '/assets/', // 放到指定目录下
    publicPath: '', // 放到根目录下
    publicPath: 'https://cdn.example.com/', // 放到 CDN 上
    // 导出库的名称，为 string 类型
    // 不填它时，默认的输出格式是匿名的立即执行函数
    library: 'MyLibrary',
    // 导出库的类型，为枚举类型，默认是 var
    // 可以是 umd、umd2、commonjs2、commonjs、amd、this、var、assign、window、global、jsonp
    libraryTarget: 'umd',
```

```
  // 是否包含有用的文件路径信息到生成的代码里，为 boolean 类型
  pathinfo: true,
  // 附加 Chunk 的文件名称
  chunkFilename: '[id].js',
  chunkFilename: '[chunkhash].js',
  // JSONP 异步加载资源时的回调函数名称，需要和服务端搭配使用
  jsonpFunction: 'myWebpackJsonp',
  // 生成的 Source Map 文件的名称
  sourceMapFilename: '[file].map',
  // 浏览器开发者工具里显示的源码模块名称
  devtoolModuleFilenameTemplate: 'webpack:///[resource-path]',
  // 异步加载跨域的资源时使用的方式
  crossOriginLoading: 'use-credentials',
  crossOriginLoading: 'anonymous',
  crossOriginLoading: false,
},
// 配置模块相关
module: {
  rules: [ // 配置 Loader
    {
      test: /\.jsx?$/, // 正则匹配命中要使用 Loader 的文件
      include: [ // 只会命中这里面的文件
        path.resolve(__dirname, 'app')
      ],
      exclude: [ // 忽略这里面的文件
        path.resolve(__dirname, 'app/demo-files')
      ],
      use: [ // 使用哪些 Loader，有先后次序，从后向前执行
        'style-loader', // 直接使用 Loader 的名称
        {
          loader: 'css-loader',
          options: { // 向 html-loader 传一些参数
          }
        }
      ]
    },
  ],
  noParse: [ // 不用解析和处理的模块
    /special-library\.js$/ // 用正则匹配
  ],
```

```
    },
    // 配置插件
    plugins: [
    ],
    // 配置寻找模块的规则
    resolve: {
      modules: [ // 寻找模块的根目录，为array类型，默认以node_modules为根目录
        'node_modules',
        path.resolve(__dirname, 'app')
      ],
      extensions: ['.js', '.json', '.jsx', '.css'], // 模块的后缀名
      alias: { // 模块别名配置，用于映射模块
        // 将'module'映射成'new-module'，同样，'module/path/file'也会被映射成'new-module/path/file'
        'module': 'new-module',
        // 使用结尾符号$后，将'only-module'映射成'new-module'，
        // 但是不像上面的，'module/path/file'不会被映射成'new-module/path/file'
        'only-module$': 'new-module',
      },
      alias: [ // alias还支持使用数组来更详细地进行配置
        {
          name: 'module', // 老模块
          alias: 'new-module', // 新模块
          // 是否只映射模块，如果是true，则只有'module'会被映射；如果是false，则'module/inner/path'也会被映射
          onlyModule: true,
        }
      ],
      symlinks: true, // 是否跟随文件的软链接去搜寻模块的路径
      descriptionFiles: ['package.json'], // 模块的描述文件
      mainFields: ['main'], // 模块的描述文件里描述入口的文件的字段名
      enforceExtension: false, // 是否强制导入语句写明文件后缀
    },
    // 输出文件的性能检查配置
    performance: {
      hints: 'warning', // 有性能问题时输出警告
      hints: 'error', // 有性能问题时输出错误
      hints: false, // 关闭性能检查
      maxAssetSize: 200000, // 最大文件的大小（单位为bytes）
      maxEntrypointSize: 400000, // 最大入口文件的大小（单位为bytes）
```

```
      assetFilter: function(assetFilename) { // 过滤要检查的文件
        return assetFilename.endsWith('.css') || assetFilename.endsWith
('.js');
      }
    ),
    devtool: 'source-map', // 配置 source-map 类型
    context: __dirname, // Webpack 使用的根目录，string 类型必须是绝对路径
    // 配置输出代码的运行环境
    target: 'web', // 浏览器，默认
    target: 'webworker', // WebWorker
    target: 'node', // Node.js，使用`require`语句加载 Chunk 代码
    target: 'async-node', // Node.js，异步加载 Chunk 代码
    target: 'node-webkit', // nw.js
    target: 'electron-main', // electron，主线程
    target: 'electron-renderer', // electron，渲染线程
    externals: { // 使用来自 JavaScript 运行环境提供的全局变量
    jquery: 'jQuery'
    },
    stats: { // 控制台输出日志控制
    assets: true,
    colors: true,
      errors: true,
      errorDetails: true,
      hash: true,
    },
    devServer: { // DevServer 相关的配置
      proxy: { // 代理到后端服务接口
        '/api': 'http://localhost:3000'
      },
      contentBase: path.join(__dirname, 'public'), // 配置 DevServer HTTP
服务器的文件根目录
      compress: true, // 是否开启 Gzip 压缩
      historyApiFallback: true, // 是否开发 HTML5 History API 网页
      hot: true, // 是否开启模块热替换功能
      https: false, // 是否开启 HTTPS 模式
      },
      profile: true, // 是否捕捉 Webpack 构建的性能信息，用于分析是什么原因导致构
建性能不佳
      cache: false, // 是否启用缓存来提升构建速度
      watch: true, // 是否开始
```

```
    watchOptions: { // 监听模式选项
    // 不监听的文件或文件夹，支持正则匹配。默认为空
    ignored: /node_modules/,
    // 监听到变化发生后，等 300ms 再执行动作，截流，防止文件更新太快导致重新编译频率太快。默认为 300ms
    aggregateTimeout: 300,
    //不停地询问系统指定的文件有没有发生变化，默认每秒询问 1000 次
    poll: 1000
  },
}
```

2.9 多种配置类型

除了通过导出一个 Object 来描述 Webpack 所需的配置，还有其他更灵活的方式，以简化不同场景的配置。下面来一一介绍它们。

2.9.1 导出一个 Function

在大多数时候，我们需要从同一份源代码中构建出多份代码，例如一份用于开发，一份用于发布到线上。

如果采用导出一个 Object 来描述 Webpack 所需配置的方法，则需要写两个文件，一个用于开发环境，一个用于线上环境。再在启动时通过 `webpack --config webpack.config.js` 指定使用哪个配置文件。

采用导出一个 Function 的方式，能通过 JavaScript 灵活地控制配置，做到只用写一个配置文件就能完成以上要求。

导出一个 Function 的使用方式如下：

```
const path = require('path');
const UglifyJsPlugin = require('webpack/lib/optimize/UglifyJsPlugin');
module.exports = function (env = {}, argv) {
  const plugins = [];
  const isProduction = env['production'];
```

```
  // 在生成环境中才压缩
  if (isProduction) {
    plugins.push(
      // 压缩输出的 JavaScript 代码
      new UglifyJsPlugin()
    )
  }
  return {
    plugins: plugins,
    // 在生成环境中不输出 Source Map
    devtool: isProduction ? undefined : 'source-map',
  };
}
```

在运行 Webpack 时，会向这个函数传入两个参数，如下所述。

- env：当前运行时的 Webpack 专属环境变量，env 是一个 Object。读取时直接访问 Object 的属性，将它设置为需要在启动 Webpack 时带上参数。例如启动命令是 webpack --env.production --env.bao=foo，则 env 的值是{"production":"true","bao":"foo"}。
- argv：代表在启动 Webpack 时通过命令行传入的所有参数，例如--config、--env、--devtool，可以通过 webpack -h 列出所有 Webpack 支持的命令行参数。

就以上配置文件而言，在开发时执行命令 webpack 构建出方便调试的代码，在需要构建出发布到线上的代码时执行 webpack --env.production 构建出压缩的代码。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/2-9 多种配置类型.zip。

2.9.2 导出一个返回 Promise 的函数

在某些情况下不能以同步的方式返回一个描述配置的 Object，Webpack 还支持导出一个返回 Promise 的函数，使用如下：

```
module.exports = function(env = {}, argv) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
```

```
      resolve({
        // ...
      })
    }, 5000)
  })
}
```

2.9.3 导出多份配置

除了只导出一份配置，Webpack 还支持导出一个数组，数组中可以包含每份配置，并且每份配置都会执行一遍构建。

注意，Webpack 从 3.1.0 版本才开始支持该特性。

使用如下：

```
module.exports = [
  // 采用 Object 描述的一份配置
  {
    // ...
  },
  // 采用函数描述的一份配置
  function() {
    return {
      // ...
    }
  },
  // 采用异步函数描述的一份配置
  function() {
    return Promise();
  }
]
```

以上配置会导致 Webpack 针对这三份配置执行三次不同的构建。

这特别适合用 Webpack 构建一个要上传到 Npm 仓库的库，因为库中可能需要包含多种模块化格式的代码，例如 CommonJS、UMD。

2.10 总结

从前面的配置看有很多选项，Webpack 内置了很多功能，我们不必都记住它们，只需要大概明白 Webpack 原理和核心概念，并判断选项大致属于哪个大模块下，再去查详细的使用文档即可。

通常我们可用如下经验去判断如何配置 Webpack：

- 若想让源文件加入构建流程中被 Webpack 控制，则配置 `entry`；
- 若想自定义输出文件的位置和名称，则配置 `output`；
- 若想自定义寻找依赖模块时的策略，则配置 `resolve`；
- 若想自定义解析和转换文件的策略，则配置 `module`，通常是配置 `module.rules` 里的 Loader；
- 若其他大部分需求可能通过 Plugin 去实现，则配置 `plugin`。

第3章

实　战

本章讲解如何通过 Webpack 去应对实际项目中的常见场景，根据不同的场景可划分为以下几部分。

- 使用新语言来开发项目，参见 3.1 节～3.5 节。
- 使用新框架来开发项目，参见 3.6 节～3.8 节。
- 用 Webpack 构建单页应用，参见 3.9 节～3.10 节。
- 用 Webpack 构建不同运行环境下的项目，参见 3.11 节～3.14 节。
- Webpack 结合其他工具搭配使用，各取所长，参见 3.15 节～3.18 节。
- 用 Webpack 加载特殊类型的资源，参见 3.19 节～3.22 节。

3.1　使用 ES6 语言

ECMAScript 6.0 是 2015 年发布的下一代 JavaScript 语言标准，它引入了新的语法和 API 来提升开发效率。虽然目前部分浏览器和 Node.js 已经支持 ES6，但由于它们对 ES6 的所有标准支持不全，会导致在开发中不能全面使用 ES6。

通常我们需要将采用 ES6 编写的代码转换成目前已经支持良好的 ES5 代码，包含如下两件事：

- 将新的 ES6 语法用 ES5 实现，例如 ES6 的 `class` 语法用 ES5 的 `prototype` 实现；
- 为新的 API 注入 polyfill，例如使用新的 `fetch` API 时在注入对应的 polyfill 后才能让低端浏览器正常运行。

3.1.1　认识 Babel

Babel（https://babeljs.io）可以方便地完成以上两件事。Babel 是一个 JavaScript 编译器，能将 ES6 代码转为 ES5 代码，让我们使用最新的语言特性而不用担心兼容性问题，并且可以通过插件机制根据需求灵活地扩展。在 Babel 执行编译的过程中，会从项目根目录下的`.babelrc` 文件中读取配置。`.babelrc` 是一个 JSON 格式的文件，内容大致如下：

```
{
  "plugins": [
    [
      "transform-runtime",
      {
        "polyfill": false
      }
    ]
  ],
  "presets": [
```

```
    [
      "es2015",
      {
        "modules": false
      }
    ],
    "stage-2",
    "react"
  ]
}
```

1. Plugins

plugins 属性告诉 Babel 要使用哪些插件，这些插件可以控制如何转换代码。

以上配置文件里的 transform-runtime 对应的插件全名叫作 babel-plugin-transform-runtime，即在前面加上了 babel-plugin-。要让 Babel 正常运行，我们必须先安装这个插件：

```
npm i -D babel-plugin-transform-runtime
```

babel-plugin-transform-runtime 是 Babel 官方提供的一个插件，作用是减少冗余的代码。Babel 在将 ES6 代码转换成 ES5 代码时，通常需要一些由 ES5 编写的辅助函数来完成新语法的实现，例如在转换 class extent 语法时会在转换后的 ES5 代码里注入 _extent 辅助函数用于实现继承：

```
function _extent(target) {
  for (var i = 1; i < arguments.length; i++) {
    var source = arguments[i];
    for (var key in source) {
      if (Object.prototype.hasOwnProperty.call(source, key)) {
        target[key] = source[key];
      }
    }
  }
  return target;
}
```

这会导致每个使用 class extent 语法的文件都被注入重复的_extent 辅助函数代

码，babel-plugin-transform-runtime的作用在于将原本注入JavaScript文件里的辅助函数替换成一条导入语句：

```
var _extent = require('babel-runtime/helpers/_extent');
```

这样能减小Babel编译出来的代码的文件大小。

同时需要注意的是，由于babel-plugin-transform-runtime注入了require('babel-runtime/helpers/_extent')语句到编译后的代码里，需要安装babel-runtime依赖到我们的项目后，代码才能正常运行。也就是说babel-plugin-transform-runtime和babel-runtime需要配套使用，在使用babel-plugin-transform-runtime后一定需要使用babel-runtime。

2. Presets

presets属性告诉Babel要转换的源码使用了哪些新的语法特性，一个Presets对一组新语法的特性提供了支持，多个Presets可以叠加。Presets其实是一组Plugins的集合，每个Plugin完成一个新语法的转换工作。Presets是按照ECMAScript草案来组织的，通常可以分为以下三大类。

(1)已经被写入ECMAScript标准里的特性，由于之前每年都有新特性被加入到标准里，所以又可细分如下。

- ES2015（https://babeljs.io/docs/plugins/preset-es2015/）：包含在2015年加入的新特性。
- ES2016（https://babeljs.io/docs/plugins/preset-es2016/）：包含在2016年加入的新特性。
- ES2017（https://babeljs.io/docs/plugins/preset-es2017/）：包含在2017年加入的新特性。
- Env（https://babeljs.io/docs/plugins/preset-env/），包含当前所有ECMAScript标准里的最新特性。

它们之间的关系如图3-1所示。

图 3-1　ECMAScript 标准里的特性的关系

（2）被社区提出来的但还未被写入 ECMAScript 标准里的特性，这其中又分为以下四种。

- stage0（https://babeljs.io/docs/plugins/preset-stage-0/）：只是一个美好激进的想法，一些 Babel 插件实现了对这些特性的支持，但是不确定是否会被定为标准。
- stage1（https://babeljs.io/docs/plugins/preset-stage-1/）：值得被纳入标准的特性。
- stage2（https://babeljs.io/docs/plugins/preset-stage-2/）：该特性规范已经被起草，将会被纳入标准里。
- stage3（https://babeljs.io/docs/plugins/preset-stage-3/）：该特性规范已经定稿，各大浏览器厂商和 Node.js 社区已开始着手实现。
- stage4：在接下来的一年里将会加入标准里。

它们之间的关系如图 3-2 所示。

图 3-2　stage 关系图

（3）用于支持一些特定应用场景下的语法的特性，和 ECMAScript 标准没有关系，例如 `babel-preset-react` 用于支持 React 开发里的 JSX 语法。

在实际应用中，我们需要根据项目源码所使用的语法去安装对应的 Plugins 或 Presets。

3.1.2 接入 Babel

在了解 Babel 后，下一步就需要知道如何在 Webpack 中使用它。由于 Babel 所做的事情是转换代码，所以应该通过 Loader 去接入 Babel。Webpack 的配置如下：

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        use: ['babel-loader'],
      },
    ]
  },
  // 输出 source-map 以方便直接调试 ES6 源码
  devtool: 'source-map'
};
```

以上配置命中了项目目录下的所有 JavaScript 文件，并通过 babel-loader 调用 Babel 完成转换工作。在重新执行构建前，需要先安装新引入的依赖：

```
# Webpack 接入 Babel 必须依赖的模块
npm i -D babel-core babel-loader
# 根据我们的需求选择不同的 Plugins 或 Presets
npm i -D babel-preset-env
```

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-1 使用 ES6 语言.zip。

3.2 使用 TypeScript 语言

3.2.1 认识 TypeScript

TypeScript（http://www.typescriptlang.org）是 JavaScript 的一个超集，主要提供了类型检查系统和对 ES6 语法的支持，但不支持新的 API。目前没有任何环境支持运行原生的

TypeScript 代码，必须通过构建将它转换成 JavaScript 代码后才能运行。

下面改造一下前面用过的例子 Hello,Webpack，用 TypeScript 重写 JavaScript。由于 TypeScript 是 JavaScript 的超集，直接将后缀.js 改成.ts 是可以的。但为了体现出 TypeScript 的不同，我们在这里重写 JavaScript 代码，并加入类型检查：

```
// show.ts
// 操作 DOM 元素，将 content 显示到网页上
// 通过 ES6 模块规范导出 show 函数
// 为 show 函数增加类型检查
export function show(content: string) {
  window.document.getElementById('app').innerText = 'Hello,' + content;
}
// main.ts
// 通过 ES6 模块规范导入 show 函数
import {show} from './show';
// 执行 show 函数
show('Webpack');
```

TypeScript 官方提供了能将 TypeScript 转换成 JavaScript 的编译器。我们需要在当前项目的根目录下新建一个用于配置编译选项的 tsconfig.json 文件，编译器默认会读取和使用这个文件，配置文件的内容大致如下：

```
{
  "compilerOptions": {
    "module": "commonjs", // 编译出的代码采用的模块规范
    "target": "es5", // 编译出的代码采用 ES 的哪个版本
    "sourceMap": true // 输出 Source Map 以方便调试
  },
  "exclude": [ // 不编译这些目录里的文件
    "node_modules"
  ]
}
```

通过 npm install -g typescript 安装编译器到全局后，可以通过 tsc hello.ts 命令编译出 hello.js 和 hello.js.map 文件。

3.2.2 减少代码冗余

TypeScript 编译器会有与 3.1 节中 Babel 同样的问题：在将 ES6 语法转换成 ES5 语法时需要注入辅助函数。为了不让同样的辅助函数重复出现在多个文件中，可以开启 TypeScript 编译器的 importHelpers 选项，需要修改 tsconfig.json 文件如下：

```
{
  "compilerOptions": {
    "importHelpers": true
  }
}
```

该选项的原理和 Babel 中介绍的 babel-plugin-transform-runtime 非常类似，会将辅助函数转换成如下导入语句：

```
var _tslib = require('tslib');
_tslib._extend(target);
```

这会导致编译出的代码依赖 tslib 这个迷你库，但避免了代码冗余。

3.2.3 集成 Webpack

要让 Webpack 支持 TypeScript，需要解决以下两个问题。

- 通过 Loader 将 TypeScript 转换成 JavaScript。
- Webpack 在寻找模块对应的文件时需要尝试 ts 后缀。

对于问题 1，社区已经出现了几个可用的 Loader，推荐速度更快的 awesome- typescript-loader（https://github.com/s-panferov/awesome-typescript-loader）。对于问题 2，根据 2.4 节中的 extensions，我们需要修改默认的 resolve.extensions 配置项。

综上所述，相关的 Webpack 配置如下：

```
const path = require('path');
module.exports = {
  // 执行入口文件
  entry: './main',
  output: {
```

```
    filename: 'bundle.js',
    path: path.resolve(__dirname, './dist'),
  },
  resolve: {
    // 先尝试以 ts 为后缀的 TypeScript 源码文件
    extensions: ['.ts', '.js']
  },
  module: {
    rules: [
      {
        test: /\.ts$/,
        loader: 'awesome-typescript-loader'
      }
    ]
  },
  devtool: 'source-map',// 输出 Source Map 以方便在浏览器里调试 TypeScript 代码
};
```

在运行构建前需要安装上面用到的依赖：

```
npm i -D typescript awesome-typescript-loader
```

安装成功后重新执行构建，我们将会在 dist 目录下看到输出的 JavaScript 文件 bundle.js，以及对应的 Source Map 文件 bundle.js.map。在浏览器里打开 index.html 页面后，可以在开发工具里看到和调试用 TypeScript 编写的源码。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-2 使用 TypeScript 语言.zip。

3.3 使用 Flow 检查器

3.3.1 认识 Flow

Flow（https://flow.org）是 Facebook 开源的一个 JavaScript 静态类型检测器，它是 JavaScript 语言的超集。我们所需要做的就是在需要的地方加上类型检查，例如在两个由不同的人开发的模块对接的接口处加上静态类型检查，就能在编译阶段指出部分模块使用不当的问题。同

时，Flow 能通过类型推断检查出在 JavaScript 代码中潜在的 Bug。

Flow 的使用效果如下：

```
// @flow
// 静态类型检查
function square1(n: number): number {
  return n * n;
}
square1('2'); // Error: square1 需要传入 number 作为参数
// 类型推断检查
function square2(n) {
  return n * n; // Error:传入的 string 类型不能做乘法运算
}
square2('2');
```

需要注意的是，该段代码的第 1 行//@flow 告诉 Flow 检查器这个文件需要被检查。

3.3.2 使用 Flow

以上只是让我们了解 Flow 的功能，下面讲解如何运行 Flow 来检查代码。Flow 检测器由高性能且跨平台的 OCaml（http://ocaml.org）语言编写，它的可执行文件可以通过 npm i -D flow-bin 安装，安装完成后可先配置 Npm Script：

```
"scripts": {
  "flow": "flow"
}
```

再通过 npm run flow 去调用 Flow 执行代码检查。

除此之外，我们还可以通过 npm i -g flow-bin 将 Flow 安装到全局，再直接通过 flow 命令执行代码检查。

安装成功后，在项目根目录下执行 Flow，Flow 会遍历出所有需要检查的文件并对其进行检查，输出错误结果到控制台，例如：

```
Error: show.js:6
  6: export function show(content) {
                      ^^^^^^^ parameter `content`. Missing annotation
```

```
Found 1 error
```

采用了 Flow 静态类型语法的 JavaScript，是无法直接在目前已有的 JavaScript 引擎中运行的，要让代码可以运行，需要将这些静态类型的语法去掉。例如：

```
// 采用 Flow 的源代码
function foo(one: any, two: number, three?): string {}
// 去掉静态类型语法后输出代码
function foo(one, two, three) {}
```

有两种方式可以做到这一点。

- flow-remove-types（https://github.com/flowtype/flow-remove-types）：可单独使用，速度快。
- babel-preset-flow（https://babeljs.io/docs/plugins/preset-flow/）：与 Babel 集成。

3.3.3 集成 Webpack

由于使用了 Flow 的项目一般都会使用 ES6 语法，所以将 Flow 集成到使用 Webpack 构建的项目里的最方便方法是借助 Babel。下面修改 3.1 节中的代码，为其加入 Flow 代码检查，改动如下。

（1）安装 `npm i -D babel-preset-flow` 依赖到项目。

（2）修改 `.babelrc` 配置文件，加入 Flow Preset：

```
"presets": [
  ...[],
  "flow"
]
```

向源码里加入静态类型后重新构建项目，我们会发现采用了 Flow 的源码还是能在浏览器中正常运行的。

要明确构建的目的只是去除源码中的 Flow 静态类型语法，而代码检查和构建无关。许多编辑器已经整合了 Flow，可以实时在代码中高亮显示 Flow 检查出的问题。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-3 使用 Flow 检查器.zip。

3.4 使用 SCSS 语言

3.4.1 认识 SCSS

SCSS（http://sass-lang.com）可以让我们用更灵活的方式写 CSS。它是一种 CSS 预处理器，语法和 CSS 相似，但加入了变量、逻辑等编程元素，代码类似这样：

```
$blue: #1875e7;
div {
  color: $blue;
}
```

SCSS 又叫作 SASS，区别在于 SASS 语法类似于 Ruby，而 SCSS 语法类似于 CSS，熟悉 CSS 的前端工程师会更喜欢 SCSS。

采用 SCSS 去写 CSS 的好处在于，可以方便地管理代码，抽离公共的部分，通过逻辑写出更灵活的代码。和 SCSS 类似的 CSS 预处理器还有 LESS（http://lesscss.org）等。

使用 SCSS 可以提升编码的效率，但是必须将 SCSS 源代码编译成可以直接在浏览器环境下运行的 CSS 代码。SCSS 官方提供了以多种语言实现的编译器，由于本书更倾向于前端工程师使用的技术栈，所以主要介绍 node-sass（https://github. com/sass/node-sass）。

node-sass 的核心模块是用 C++编写的，再用 Node.js 封装了一层，以提供给其他 Node.js 调用。node-sass 还支持通过命令行调用，先将它安装到全局：

```
npm i -g node-sass
```

再执行编译命令：

```
# 将 main.scss 源文件编译成 main.css
node-sass main.scss main.css
```

就能在源码同目录下看到编译后的 main.css 文件。

3.4.2 接入 Webpack

我们曾在 1.4 节介绍过将 SCSS 源代码转换成 CSS 代码的最佳方式是使用 Loader，Webpack 官方提供了对应的 sass-loader（https://github.com/webpack-contrib/sass-loader）。

Webpack 接入 sass-loader 的相关配置如下：

```
module.exports = {
  module: {
    rules: [
      {
        // 增加对 SCSS 文件的支持
        test: /\.scss/,
        // SCSS 文件的处理顺序为先 sass-loader，再 css-loader，再 style-loader
        use: ['style-loader', 'css-loader', 'sass-loader'],
      },
    ]
  },
};
```

以上配置通过正则/\.scss/匹配所有以.scss 为后缀的 SCSS 文件，再分别使用 3 个 Loader 去处理。具体处理流程如下。

- 通过 sass-loader 将 SCSS 源码转换为 CSS 代码，再将 CSS 代码交给 css-loader 处理。
- css-loader 会找出 CSS 代码中@import 和 url()这样的导入语句，告诉 Webpack 依赖这些资源。同时支持 CSS Modules、压缩 CSS 等功能。处理完后再将结果交给 style-loader 处理。
- style-loader 会将 CSS 代码转换成字符串后，注入 JavaScript 代码中，通过 JavaScript 向 DOM 增加样式。如果我们想将 CSS 代码提取到一个单独的文件中，而不是和 JavaScript 混在一起，则可以使用在 1.5 节中介绍过的 ExtractTextPlugin。

由于接入 sass-loader，所以项目需要安装这些新的依赖：

```
# 安装 Webpack Loader 依赖
npm i -D  sass-loader css-loader style-loader
# sass-loader 依赖 node-sass
npm i -D node-sass
```

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-4 使用 SCSS 语言.zip。

3.5 使用 PostCSS

3.5.1 认识 PostCSS

PostCSS（http://postcss.org）是一个 CSS 处理工具，和 SCSS 的不同之处在于它可以通过插件机制灵活地扩展其支持的特性，而不像 SCSS 那样语法是固定的。PostCSS 的用处非常多，包括向 CSS 自动加前缀、使用下一代 CSS 语法等。目前越来越多的人开始使用它，它很可能会成为 CSS 预处理器的最终赢家。

PostCSS 和 CSS 的关系就像 Babel 和 JavaScript 的关系，它们解除了语法上的禁锢，通过插件机制来扩展语言本身，用工程化手段为语言带来了更多的可能性。

PostCSS 和 SCSS 的关系就像 Babel 和 TypeScript 的关系，PostCSS 更灵活、可扩张性强，SCSS 内置了大量的功能而不能扩展。

为了更直观地展示 PostCSS，让我们来看一些例子。

为 CSS 自动加前缀，增加各浏览器的兼容性：

```
/*输入*/
h1 {
  display: flex;
}
/*输出*/
h1 {
  display: -webkit-box;
  display: -webkit-flex;
  display: -ms-flexbox;
  display: flex;
}
```

使用下一代 CSS 语法：

```
/*输入*/
:root {
  --red: #d33;
}
h1 {
  color: var(--red);
}
/*输出*/
h1 {
  color: #d33;
}
```

PostCSS 全部采用 JavaScript 编写，运行在 Node.js 之上，既提供了可在 JavaScript 中调用的 Node.js 模块，也提供了可直接通过命令行执行的程序。在 PostCSS 启动时，会从目录下的 postcss.config.js 文件中读取所需的配置，所以需要新建该文件，文件的内容大致如下：

```
module.exports = {
  plugins: [
    // 需要使用的插件列表
    require('postcss-cssnext')
  ]
}
```

其中的 postcss-cssnext（http://cssnext.io）插件可以让我们使用下一代 CSS 语法编写代码，再通过 PostCSS 转换成目前的浏览器可识别的 CSS，并且该插件包含为 CSS 自动加前缀的功能。

目前 Chrome 等现代浏览器已经能完全支持 cssnext 中的所有语法，也就是说按照 cssnext 语法写的 CSS 在不经过转换的情况下也能在浏览器中直接运行。

3.5.2 接入 Webpack

虽然使用 PostCSS 后，文件的后缀还是 .css，但必须将这些文件先交给 postcss-loader（https://github.com/postcss/postcss-loader）处理一遍后再交给 css-loader。

接入 PostCSS 相关的 Webpack 配置如下：

```
module.exports = {
  module: {
    rules: [
      {
        // 使用 PostCSS 处理 CSS 文件
        test: /\.css/,
        use: ['style-loader', 'css-loader', 'postcss-loader'],
      },
    ]
  },
};
```

接入 PostCSS 为项目带来了新的依赖需要安装，代码如下：

```
# 安装 Webpack Loader 依赖
npm i -D postcss-loader css-loader style-loader
# 根据我们使用的特性安装对应的 PostCSS 插件依赖
npm i -D postcss-cssnext
```

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-5 使用 PostCSS.zip。

3.6 使用 React 框架

3.6.1 React 的语法特征

在使用了 React 项目的代码中有 JSX 和 Class 语法，例如：

```
class Button extends Component {
  render() {
    return <h1>Hello,Webpack</h1>
  }
}
```

在使用了 React 的项目里，JSX 和 Class 语法并不是必需的，但使用新语法写出的代码看上去更优雅。

其中，JSX 语法是无法在任何现有的 JavaScript 引擎中运行的，所以在构建的过程中需

要将源码转换成可以运行的代码，例如：

```
// 原 JSX 语法代码
return <h1>Hello,Webpack</h1>
// 被转换成正常的 JavaScript 代码
return React.createElement('h1', null, 'Hello,Webpack')
```

目前 Babel 和 TypeScript 都提供了对 React 语法的支持，下面分别介绍如何在使用 Babel 或 TypeScript 的项目中接入 React 框架。

3.6.2 React 与 Babel

在使用 Babel 的项目中接入 React 框架很简单，只需要加入 React 所依赖的 Presets babel-preset-react（https://babeljs.io/docs/plugins/preset-react/）。接下来通过修改 3.1 节中的项目，为其接入 React 框架。

通过以下命令：

```
# 安装 React 基础依赖
npm i -D react react-dom
# 安装 Babel 完成语法转换所需的依赖
npm i -D babel-preset-react
```

安装新的依赖后，再修改.babelrc 配置文件，加入 React Presets：

```
"presets": [
    "react"
],
```

这样就完成了一切准备工作。

再修改 main.js 文件如下：

```
import * as React from 'react';
import { Component } from 'react';
import { render } from 'react-dom';
class Button extends Component {
  render() {
    return <h1>Hello,Webpack</h1>
  }
```

```
}
render(<Button/>, window.document.getElementById('app'));
```

重新执行构建，打开网页后我们将会发现由 React 渲染出来的 Hello,Webpack。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-6 使用 React 框架 Babel.zip。

3.6.3 React 与 TypeScript

与 Babel 相比，TypeScript 的优点在于，它原生支持 JSX 语法，不需要重新安装新的依赖，只需修改一行配置。但 TypeScript 的不同之处在于：

- 使用了 JSX 语法的文件后缀必须是 tsx；
- 由于 React 不是采用 TypeScript 编写的，所以需要安装 react 和 react-dom 对应的 TypeScript 接口描述模块@types/react 和@types/react-dom 才能通过编译。

接下来通过修改在 3.2 节中讲过的项目，为其接入 React 框架。修改 TypeScript 编译器的配置文件 tsconfig.json，增加对 JSX 语法的支持：

```
{
  "compilerOptions": {
    "jsx": "react" // 开启 JSX，支持 React
  }
}
```

由于 main.js 文件中存在 JSX 语法，所以这里再将 main.js 文件重命名为 main.tsx，同时修改文件的内容为上面 React 与 Babel 里采用的 React 代码。同时，为了让 Webpack 对项目里的 ts 与 tsx 原文件都采用 awesome-typescript-loader 去转换，我们需要注意，Webpack Loader 配置的 test 选项需要匹配到 tsx 类型的文件，并且在 extensions 中也要加上.tsx，配置如下：

```
const path = require('path');
module.exports = {
  // TS 执行入口文件
  entry: './main',
  output: {
```

```
    filename: 'bundle.js',
    path: path.resolve(__dirname, './dist'),
  },
  resolve: {
    // 先尝试 ts、tsx 后缀的 TypeScript 源码文件
    extensions: ['.ts', '.tsx', '.js',]
  },
  module: {
    rules: [
      {
        // 同时匹配 ts、tsx 后缀的 TypeScript 源码文件
        test: /\.tsx?$/,
        loader: 'awesome-typescript-loader'
      }
    ]
  },
  devtool: 'source-map',// 输出 Source Map 以方便在浏览器里调试 TypeScript 代码
};
```

通过以下代码安装新的依赖：

```
npm i react react-dom @types/react @types/react-dom
```

之后重启构建，重新打开网页，我们将会发现由 React 渲染出来的内容：`Hello, Webpack`。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-6 使用 React 框架 TypeScript.zip。

3.7 使用 Vue 框架

Vue（https://cn.vuejs.org）是一个渐进式的 MVVM 框架，比 React、Angular 更灵活、轻量。它不会强制性地内置一些功能和语法，我们可以根据自己的需要一点点地添加功能。虽然采用 Vue 的项目能用可直接运行在浏览器环境里的代码编写，但为了方便编码，大多数项目都会采用 Vue 官方的单文件组件（https://cn.vuejs.org/v2/guide/single-file-components.

html#介绍）的写法去编写项目。由于直接引用 Vue 是很成熟的做法，所以本书只专注于讲解如何用 Webpack 构建 Vue 单文件组件。

3.7.1 认识 Vue

Vue 和 React 一样，都推崇组件化和由数据驱动视图的思想，将视图和数据绑定在一起，这样数据改变时，视图会跟着改变，而无须直接操作视图。还是以前面的 Hello,Webpack 为例，来看看 Vue 版本的实现。

App.vue 文件代表一个单文件组件，它是项目唯一的组件，也是根组件：

```
<!--渲染模板-->
<template>
  <h1>{{ msg }}</h1>
</template>
<!--样式描述-->
<style scoped>
  h1 {
    color: red;
  }
</style>
<!--组件逻辑-->
<script>
  export default {
    data() {
      return {
        msg: 'Hello,Webpack'
      }
    }
  }
</script>
```

Vue 的单文件组件通过一个类似于 HTML 文件的 .vue 文件就能描述清楚一个组件所需的模板、样式、逻辑。

main.js 入口文件：

```
import Vue from 'vue'
```

```
import App from './App.vue'
new Vue({
  el: '#app',
  render: h => h(App)
});
```

入口文件创建 Vue 的一个根实例，在 ID 为 app 的 DOM 节点上渲染出上面定义的 App 组件。

3.7.2 接入 Webpack

目前最成熟和流行的开发 Vue 项目的方式是采用 ES6 加 Babel 转换，这和基本的采用 ES6 开发的项目很相似，差别在于要解析 .vue 格式的单文件组件。好在 Vue 官方提供了对应的 vue-loader（https://vue-loader.vuejs.org/zh-cn/），可以非常方便地完成单文件组件的转换。

修改 Webpack 的相关配置如下：

```
module: {
  rules: [
    {
      test: /\.vue$/,
      use: ['vue-loader'],
    },
  ]
}
```

安装新引入的依赖：

```
# Vue 框架运行需要的库
npm i -S vue
# 构建所需的依赖
npm i -D vue-loader css-loader vue-template-compiler
```

在这些依赖中，它们的作用分别如下。

- vue-loader：解析和转换 .vue 文件，提取出其中的逻辑代码 script、样式代码 style 及 HTML 模板 template，再分别将它们交给对应的 Loader 去处理。

- `css-loader`：加载由 `vue-loader` 提取出的 CSS 代码。
- `vue-template-compiler`：将 `vue-loader` 提取出的 HTML 模板编译成对应的可执行的 JavaScript 代码，这和 React 中的 JSX 语法被编译成 JavaScript 代码类似。预先编译好 HTML 模板相对于在浏览器中编译 HTML 模板，性能更好。

重新启动构建，我们就能看到由 Vue 渲染出的 `Hello,Webpack` 了。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-7 使用 Vue 框架 Babel.zip。

3.7.3 使用 TypeScript 编写 Vue 应用

Vue 从 2.5.0 版本开始，就提供了对 TypeScript 的良好支持。使用 TypeScript 编写 Vue 是一个很好的选择，因为 TypeScript 能检查出一些潜在的错误。下面讲解如何用 Webpack 构建使用 TypeScript 编写的 Vue 应用。

新增 `tsconfig.json` 配置文件，内容如下：

```
{
  "compilerOptions": {
    // 构建出 ES5 版本的 JavaScript，与 Vue 的浏览器支持保持一致
    "target": "es5",
    // 开启严格模式，这可以对`this`上的数据属性进行更严格的推断
    "strict": true,
    // TypeScript 编译器输出的 JavaScript 采用 ES2015 模块化，使 Tree Sharking
生效
    "module": "es2015",
    "moduleResolution": "node"
  }
}
```

在以上代码中，`"module": "es2015"`用于使 Tree Sharking 优化生效，可通过阅读 4.10 节进一步了解。

修改 `App.vue` 脚本部分的内容如下：

```
<!--组件逻辑-->
<script lang="ts">
```

```
  import Vue from "vue";
  // 通过 Vue.extend 启用 TypeScript 类型推断
  export default Vue.extend({
    data() {
      return {
        msg: 'Hello,Webpack',
      }
    },
  });
</script>
```

注意，script 标签中的 lang="ts"用于指明代码的语法是 TypeScript。

修改 main.ts 的执行入口文件如下：

```
import Vue from 'vue'
import App from './App.vue'
new Vue({
  el: '#app',
  render: h => h(App)
});
```

由于 TypeScript 不认识以.vue 结尾的文件，所以为了让其支持 import App from './App.vue'导入语句，还需要以下文件 vue-shims.d.ts 定义.vue 文件的类型：

```
// 告诉 TypeScript 编译器.vue 文件其实是一个 Vue
declare module "*.vue" {
  import Vue from "vue";
  export default Vue;
}
```

Webpack 配置需要两个地方，代码如下：

```
const path = require('path');
module.exports = {
  resolve: {
    // 增加对 TypeScript 的.ts 和.vue 文件的支持
    extensions: ['.ts', '.js', '.vue', '.json'],
  },
  module: {
    rules: [
      // 加载.ts 文件
```

```
        {
          test: /\.ts$/,
          loader: 'ts-loader',
          exclude: /node_modules/,
          options: {
            // 让 tsc 将 vue 文件当成一个 TypeScript 模块去处理，以解决 moudle not found 的问题，tsc 本身不会处理.vue 结尾的文件
            appendTsSuffixTo: [/\.vue$/],
          }
        },
      ]
    },
  };
```

除此之外，还需要安装新引入的依赖：

```
npm i -D ts-loader typescript
```

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-7 使用 Vue 框架 TypeScript.zip。

3.8 使用 Angular2 框架

3.8.1 认识 Angular2

Angular2（https://angular.io）是 AngularJS（https://angularjs.org）的下一个版本，它继承了 AngularJS 中的部分思想，又加入了一些新的改进。与 React 和 Vue 相比，Angular2 要复杂得多，这三者的出发点都是组件化和数据驱动视图，但 Angular2 多出了以下概念。

- 模块（NgModule）：这里的模块不是指 JavaScript 或者其他编程语言里的模块化，而是指 Angular2 里提出的独有用法。
- 注解（Decorator）：可通过注解语法@XXX 来为一个 Class 附加元数据。
- 服务（Service）：按照功能划分，将项目中可以复用的重复代码封装成一个个服务以方便为其他模块使用。服务可以包含函数、常值等，常见的有日志服务、数据服务、应用程序配置等。

- 依赖注入（Dependency Injection）：也叫作控制反转（Inversion of Control），是面向对象编程中的一种设计原则，可以用来降低代码之间的耦合度。

Angular2 引入的这些概念用于分解和简化大型项目的难度，但在小项目开发中可能都是累赘，初学者可能难以掌握。在语言选择上，虽然 Angular2 官方对 TypeScript 和 JavaScript 都提供了支持，但通常选择 Angular2 的项目都会使用 TypeScript，原因在于 Angular2 本身就是使用 TypeScript 开发的，在项目中使用 TypeScript 相对于 JavaScript 来说，开发体验会好很多。

看到这里，我们可能会被 Angular2 的复杂所吓到，所以我们先来看看如何用 Angular2 开发 `Hello,Webpack`。

这个应用只有一个视图组件 `AppComponent` 用于渲染 `Hello,Webpack`，组件的代码如下：

```
import {Component} from '@angular/core';
// 通过注解的方式描述清楚这个视图组件所需的模板、样式、数据、逻辑
@Component({
  // 标签的名称
  selector: 'app-root',
  // HTML 模板
  template: '<h1>{{msg}}</h1>',
  // CSS 样式
  styles: ['h1{ color:red; }']
})
export class AppComponent {
  msg = 'Hello,Webpack';
}
```

光有组件还不够，还需要实例化 `AppComponent` 视图组件，并将它渲染到 DOM 中。Angular2 规定可运行的应用至少有一个 NgModule，也就是需要一个根 NgModule。这个根 NgModule 描述了如何启动应用，代码如下：

```
// 让 Angular2 正常运行需要的 polyfill
import 'core-js/es6/reflect';
import 'core-js/es7/reflect';
import 'zone.js/dist/zone';
// Angular2 框架的核心模块
```

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from
'@angular/platform-browser-dynamic';
// 项目自定义视图组件
import {AppComponent} from './app.component';
@NgModule({
  // 该 NgModule 所依赖的视图组件
  declarations: [AppComponent],
  // 该 NgModule 所依赖的其他 NgModule
  imports: [BrowserModule],
  // 应用的根视图组件，只有根 NgModule 需要设置
  bootstrap: [AppComponent]
})
class AppModule {
}
// 从 AppModule 启动应用
platformBrowserDynamic().bootstrapModule(AppModule);
```

Angular2 应用在启动后会解析当前的 DOM 树，找出名为 app-root 的 HTML 标签，Angular2 应用会将这个找出的标签作为容器运行。为此还需要改造 index.html 文件，插入 HTML 标签 app-root，代码如下：

```
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
<app-root></app-root>
<!--导入 Webpack 输出的 JavaScript 文件-->
<script src="./dist/bundle.js"></script>
</body>
</html>
```

要让 Hello,Webpack 运行起来，需要安装以下模块：

```
# Angular2 框架的基础核心模块
npm i -S @angular/core @angular/common
# Angular2 框架的浏览器环境运行库，类似于 react-dom
npm i -S @angular/platform-browser
# 让 Angular2 正常运行时所依赖的运行环境和 polyfill
```

```
npm i -S core-js rxjs zone.js
# 在浏览器的运行过程中动态地编译 HTML 模板
npm i -S @angular/platform-browser-dynamic @angular/compiler
```

以上是一个最小的能正常运行的 Angular2 应用，可见 Angular2 的依赖有很多，使用起来很复杂。

3.8.2 接入 Webpack

由于 Angular2 应用采用 TypeScript 开发，构建与在 3.2 节中讲过的类似，不同之处在于 tsconfig.json 配置。由于在 Angular2 项目中采用了注解的语法，而且@angular/platform-browser 源码中有许多 DOM 操作，所以需要将配置修改如下：

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "sourceMap": true,
    // 开启对注解的支持
    "experimentalDecorators": true,
    // Angular2 依赖新的 JavaScript API 和 DOM 操作
    "lib": [
      "es2015",
      "dom"
    ]
  },
  "exclude": [
    "node_modules/*"
  ]
}
```

其他配置与在 3.2 节中讲到的配置保持一致，在安装好前面提到的 Angular2 框架依赖的模块后，重新执行构建并打开网页，我们会看到由 Angular2 渲染出来的 Hello,Webpack。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-8 使用 Angular2 框架.zip。

3.9 为单页应用生成 HTML

3.9.1 引入问题

在 3.6 节中是用最简单的 Hello,Webpack 作为例子让大家理解，在这个例子里因为只输出了一个 bundle.js 文件，所以手写了一个 index.html 文件去引入这个 bundle.js，才能让应用在浏览器中运行起来。

在实际项目中远比这复杂，一个页面常常有很多资源要加载。接下来举一个实战中的例子，要求如下。

- 项目采用 ES6 语言及 React 框架。
- 为页面加入 Google Analytics（https://analytics.google.com/analytics/web/），这部分代码需要内嵌到 HEAD 标签里。
- 为页面加入 Disqus（https://disqus.com）用户评论，这部分代码需要异步加载以提升首屏加载速度。
- 压缩和分离 JavaScript 和 CSS 代码，提升加载速度。

在开始前先来看看该应用最终发布到线上的代码：

```
<html>
<head>
  <meta charset="UTF-8">
  <!--注入 Chunk app 依赖的 CSS-->
  <style rel="stylesheet">h1{color:red}</style>
  <!--内嵌 google_analytics 中的 JavaScript 代码-->
  <script>
(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
(i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new
Date();a=s.createElement(o),
m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
})(window,document,'script','https://www.google-analytics.com/analytics.js','ga');
```

```
    ga('create', 'UA-XXXXX-Y', 'auto');
    ga('send', 'pageview');
      </script>
      <!--异步加载 Disqus 评论-->
      <script async=""
src="https://dive-into-webpack.disqus.com/embed.js"></script>
    </head>
    <body>
    <div id="app"></div>
    <!--导入 app 依赖的 JavaScript 文件-->
    <script src="app_746f32b2.js"></script>
    <!--Disqus 评论容器-->
    <div id="disqus_thread"></div>
    </body>
    </html>
```

HTML 应该是被压缩过的，这里为了方便大家阅读，格式化了 HTML 并且加入了注释。

构建出的目录结构为：

```
dist
├── app_792b446e.js
└── index.html
```

可以看到，部分代码被内嵌进了 HTML 的 HEAD 标签中，部分文件的名称被打上根据文件内容算出的 Hash 值，并且加载这些文件的 URL 地址也被正常注入 HTML 中了。如果我们还采用手写 `index.html` 文件去完成以上要求，就会使工作变得复杂、易错，项目难以维护。本节讲解如何自动化地生成这个符合要求的 `index.html`。

3.9.2 解决方案

这里推荐一个用于方便解决以上问题的 Webpack 插件 web-webpack-plugin（https://github.com/gwuhaolin/web-webpack-plugin）。该插件已经被社区中的许多人使用和验证，解决了大家的痛点并获得了很多好评，下面具体介绍如何用它来解决上面的问题。

首先，修改 Webpack 配置如下：

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

```
const UglifyJsPlugin = require('webpack/lib/optimize/UglifyJsPlugin');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
const DefinePlugin = require('webpack/lib/DefinePlugin');
const { WebPlugin } = require('web-webpack-plugin');
module.exports = {
  entry: {
    app: './main.js'// app 的 JavaScript 执行入口文件
  },
  output: {
    filename: '[name]_[chunkhash:8].js',// 为输出的文件名称加上 Hash 值
    path: path.resolve(__dirname, './dist'),
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        use: ['babel-loader'],
        // 排除 node_modules 目录下的文件，
        // 该目录下的文件都采用了 ES5 语法，没必要再通过 Babel 转换
        exclude: path.resolve(__dirname, 'node_modules'),
      },
      {
        test: /\.css/,// 增加对 CSS 文件的支持
        // 提取出 Chunk 中的 CSS 代码到单独的文件中
        use: ExtractTextPlugin.extract({
          use: ['css-loader?minimize'] // 压缩 CSS 代码
        }),
      },
    ]
  },
  plugins: [
    // 使用本文的主角 WebPlugin，一个 WebPlugin 对应一个 HTML 文件
    new WebPlugin({
      template: './template.html', // HTML 模板文件所在的文件路径
      filename: 'index.html' // 输出的 HTML 的文件名称
    }),
    new ExtractTextPlugin({
      filename: `[name]_[contenthash:8].css`,// 为输出的 CSS 文件名称加上
Hash 值
    }),
```

```
      new DefinePlugin({
        // 定义 NODE_ENV 环境变量为 production，以去除源码中只有开发时才需要的部分
'process.env': {
          NODE_ENV: JSON.stringify('production')
        }
      }),
      // 压缩输出的 JavaScript 代码
      new UglifyJsPlugin({
        // 最紧凑的输出
        beautify: false,
        // 删除所有的注释
        comments: false,
        compress: {
          // 在 UglifyJS 删除没有用到的代码时不输出警告
          warnings: false,
          // 删除所有`console`语句，可以兼容 IE 浏览器
          drop_console: true,
          // 内嵌已定义但是只用到了一次的变量
          collapse_vars: true,
          // 提取出出现多次但是没有定义成变量去引用的静态值
          reduce_vars: true,
        }
      }),
    ],
  };
```

以上大多数配置都是按照前面讲过的内容增加的配置，例如：

- 增加对 CSS 文件的支持，将 Chunk 中的 CSS 代码提取到单独的文件中，压缩 CSS 文件；
- 定义 NODE_ENV 环境变量为 production，以去除源码中只有开发时才需要的部分；
- 为输出的文件名称加上 Hash 值；
- 压缩输出的 JavaScript 代码。

但核心部分在于 plugins 里的内容：

```
new WebPlugin({
  template: './template.html', // HTML 模板文件所在的文件路径
```

```
  filename: 'index.html' // 输出的 HTML 的文件名称
})
```

其中 template: './template.html'所指的模板文件 template.html 的内容是：

```
<html>
<head>
  <meta charset="UTF-8">
  <!--注入 Chunk app 中的 CSS 代码-->
  <link rel="stylesheet" href="app?_inline">
  <!--注入 google_analytics 中的 JavaScript 代码-->
  <script src="./google_analytics.js?_inline"></script>
  <!--异步加载 Disqus 评论-->
  <script src="https://dive-into-webpack.disqus.com/embed.js"
async></script>
</head>
<body>
<div id="app"></div>
<!--导入 Chunk app 中的 JavaScript 代码-->
<script src="app"></script>
<!--Disqus 评论容器-->
<div id="disqus_thread"></div>
</body>
</html>
```

该文件描述了哪些资源需要被以某种方式加入到输出的 HTML 文件中。

以<link rel="stylesheet" href="app?_inline">为例，按照正常引入 CSS 文件一样的语法来引入 Webpack 生产的代码。href 属性中的 app?_inline 可以分为两部分，前面的 app 表示 CSS 代码来自名叫 app 的 Chunk，后面的_inline 表示这些代码需要被内嵌到这个标签所在的位置。

同样，<script src="./google_analytics.js?_inline"></script>表示 JavaScript 代码来自相对于当前模板文件 template.html 的本地文件./google_analytics.js，而且文件中的 JavaScript 代码也需要被内嵌到这个标签所在的位置。

也就是说，在资源链接 URL 字符串里问号前面的部分表示资源内容来自哪里，后面的 querystring 表示这些资源注入的方式。

该插件除了支持_inline 属性，表示内嵌资源到 HTML 中，还支持以下属性。

- _dist：只有在生产环境下才引入该资源。
- _dev：只有在开发环境下才引入该资源。
- _ie：只有在 IE 浏览器下才需要引入该资源，通过[if IE]>resource<![endif]注释实现。

这些属性之间可以搭配使用，互不冲突。例如 app?_inline&_dist 表示只在生产环境下才引入该资源，并且需要内嵌到 HTML 里。

WebPlugin 插件还支持一些更高级的用法，若想了解具体内容，则可以访问该项目主页（https://github.com/gwuhaolin/web-webpack-plugin）的阅读文档。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-9 为单页应用生成 HTML.zip。

3.10 管理多个单页应用

3.10.1 引入问题

在 3.9 节中只生成了一个 HTML 文件，但在实际应用中一个完整的系统不会将所有功能都做到一个网页中，因为这会导致网页性能不佳。实际的做法是按照功能模块划分成多个单页应用，每个单页应用生成一个 HTML 文件。并且随着业务的发展，更多的单页应用可能会被逐渐加入项目中。

虽然 3.9 节已经解决了自动化生成 HTML 的痛点，但是手动去管理多个单页应用的生成也是一件麻烦的事情。继续改造 3.9 节中的例子，要求如下。

- 该项目目前共有两个单页应用组成，一个是主页 index.html，一个是用户登入页 login.html。
- 多个单页应用之间会有公共的代码部分，需要将这些公共的部分抽离出来，放到单独的文件中以防止重复加载。例如多个页面都使用了一套 CSS 样式，都采用了 React 框架，这些公共的部分需要抽离到单独的文件中。

- 随着业务的发展，后面可能会不断加入新的单页应用，但是在加入新应用时都不能改动构建相关的代码。

在开始前先来看看该应用最终发布到线上的代码。

login.html 文件的内容如下：

```
<html>
<head>
<meta charset="UTF-8">
<!--从多个页面中抽离出的公共 CSS 代码-->
<link rel="stylesheet" href="common_7cc98ad0.css">
<!--只有这个页面需要的 CSS 代码-->
<link rel="stylesheet" href="login_e31e214b.css">
<!--注入 google_analytics 中的 JS 代码-->
<script>(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[
r]||function(){
    (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new
Date();a=s.createElement(o),
m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertB
efore(a,m)
})(window,document,'script','https://www.google-analytics.com/analyti
cs.js','ga');
ga('create', 'UA-XXXXX-Y', 'auto');
ga('send', 'pageview');</script>
<!--异步加载 Disqus 评论-->
<script async=""
src="https://dive-into-webpack.disqus.com/embed.js"></script>
</head>
<body>
<div id="app"></div>
<!--从多个页面中抽离出的公共 JavaScript 代码-->
<script src="common_a1d9142f.js"></script>
<!--只有这个页面需要的 JavaScript 代码-->
<script src="login_f926c4e6.js"></script>
<!--Disqus 评论容器-->
<div id="disqus_thread"></div>
</body>
</html>
```

构建出的目录结构为：

```
dist
├── common_029086ff.js
├── common_7cc98ad0.css
├── index.html
├── index_04c08fbf.css
├── index_b3d3761c.js
├── login.html
├── login_0a3feca9.js
└── login_e31e214b.css
```

如果按照 3.9 节的思路，可能需要为每个单页应用配置如下代码：

```
new WebPlugin({
  template: './template.html', // HTML 模板文件所在的文件路径
  filename: 'login.html' // 输出的 HTML 的文件名称
})
```

并且将页面对应的入口加入 enrty 配置项中，如下所示：

```
entry: {
  index: './pages/index/index.js',// 页面 index.html 的入口文件
  login: './pages/login/index.js',// 页面 login.html 的入口文件
}
```

当有新页面加入时，就需要修改 Webpack 的配置文件，新插入以上代码，这会导致构建代码难以维护且易错。

3.10.2 解决方案

在 3.9 节中讲到的 web-webpack-plugin（https://github.com/gwuhaolin/web-webpack-plugin）插件也内置了解决该问题的方法，在该节使用了它的 WebPlugin。本节将使用它的 AutoWebPlugin 来解决以上问题，使用起来非常简单，下面讲解具体用法。

项目源码的目录结构如下：

```
├── pages
│   ├── index
```

```
│   │   ├── index.css // 该页面单独需要的 CSS 样式
│   │   └── index.js // 该页面的入口文件
│   └── login
│       ├── index.css
│       └── index.js
├── common.css // 所有页面都需要的公共 CSS 样式
├── google_analytics.js
├── template.html
└── webpack.config.js
```

从目录结构中可以看出以下几点要求：

- 所有单页应用的代码都需要放到一个目录下，例如都放在 pages 目录下；
- 一个单页应用对应一个单独的文件夹，例如最后生成的 index.html 相关的代码都在 index 目录下，login.html 同理；
- 每个单页应用的目录下都有一个 index.js 文件作为入口执行文件。

虽然 AutoWebPlugin 强制性地规定了项目部分的目录结构，但从实战经验来看，这是一种优雅的目录规范，合理地拆分了代码，又能让新人快速看懂项目的结构，方便日后维护。

将 Webpack 配置文件修改如下：

```
const { AutoWebPlugin } = require('web-webpack-plugin');
// 使用本文的主角 AutoWebPlugin，自动寻找 pages 目录下的所有目录，将每一个目录看作一个单页应用
const autoWebPlugin = new AutoWebPlugin('pages', {
  template: './template.html', // HTML 模板文件所在的文件路径
  postEntrys: ['./common.css'],// 所有页面都依赖这份通用的 CSS 样式文件
  // 提取出所有页面的公共代码
  commonsChunk: {
    name: 'common',// 提取出公共代码 Chunk 的名称
  },
});
module.exports = {
  // AutoWebPlugin 会为寻找到的所有单页应用生成对应的入口配置，
  // autoWebPlugin.entry 方法可以获取所有由 autoWebPlugin 生成的入口配置
  entry: autoWebPlugin.entry({
    // 这里可以加入我们额外需要的 Chunk 入口
  }),
  plugins: [
```

```
    autoWebPlugin,
  ],
};
```

以上配置文件为了重点展示出本节侧重修改的部分，省略了部分和 3.9 节一致的代码，读者可以参照 3.9 节或者下载本项目的完整代码。

AutoWebPlugin 会找出 pages 目录下的两个文件夹 index 和 login，将这两个文件夹看作两个单页应用，并且分别为每个单页应用生成一个 Chunk 配置和 WebPlugin 配置。每个单页应用的 Chunk 名称等同于文件夹的名称，也就是说 autoWebPlugin.entry() 方法返回的内容其实是：

```
{
  "index":["./pages/index/index.js","./common.css"],
  "login":["./pages/login/index.js","./common.css"]
}
```

但 AutoWebPlugin 会自动为我们完成这些事情，我们明白大致原理即可。

template.html 模板文件如下：

```
<html>
<head>
  <meta charset="UTF-8">
  <!--在这里注入该页面所依赖但没有手动导入的 CSS-->
  <!--STYLE-->
  <!--注入 google_analytics 中的 JavaScript 代码-->
  <script src="./google_analytics.js?_inline"></script>
  <!--异步加载 Disqus 评论-->
  <script src="https://dive-into-webpack.disqus.com/embed.js" async>
</script>
</head>
<body>
<div id="app"></div>
<!--在这注入该页面所依赖但没有手动导入的 JavaScript-->
<!--SCRIPT-->
<!--Disqus 评论容器-->
<div id="disqus_thread"></div>
</body>
</html>
```

注意到在模板文件中出现了两个重要的新关键字：<!--STYLE-->和<!--SCRIPT-->，它们是什么意思呢？

由于该模板文件被当作项目中所有单页应用的模板，所以不能再像3.9节中那样直接写Chunk的名称去引入资源，因为需要被注入当前页面的Chunk名称是不固定的，每个单页应用都会有自己的名称。<!--STYLE-->和<!--SCRIPT-->的作用在于保证该页面所依赖的资源都会被注入生成的HTML模板里。

web-webpack-plugin能分析出每个页面依赖哪些资源，例如对于login.html来说，该插件可以确定该页面依赖以下资源：

- 所有页面都依赖的公共CSS代码common.css；
- 所有页面都依赖的公共JavaScript代码common.js；
- 只有这个页面依赖的CSS代码login.css；
- 只有这个页面依赖的JavaScript代码login.css。

由于在模板文件template.html里没有指出引入这些依赖资源的HTML语句，所以插件会自动将没有手动导入但页面依赖的资源按照不同的类型注入<!--STYLE-->和<!--SCRIPT-->所在的位置。

- 将CSS类型的文件注入<!--STYLE-->所在的位置，如果<!--STYLE-->不存在，就注入HTML HEAD标签的最后。
- 将JavaScript类型的文件注入<!--SCRIPT-->所在的位置，如果<!--SCRIPT-->不存在，就注入HTML BODY标签的最后。

如果后续有新的页面需要开发，就只需在pages目录下新建一个目录，该目录名为所输出HTML文件的名称，在目录下放这个页面相关的代码即可，无须改动构建代码。

由于AutoWebPlugin是间接地通过在3.9节提到的WebPlugin实现的，所以对于WebPlugin支持的功能，AutoWebPlugin全部支持。

AutoWebPlugin插件还支持一些更高级的用法，若想了解具体内容，则可以阅读该项目主页（https://github.com/gwuhaolin/web-webpack-plugin）的阅读文档。

本实例提供项目的完整代码，参见http://webpack.wuhaolin.cn/3-10管理多个单页应用.zip。

3.11 构建同构应用

同构应用是指写一份代码但可同时在浏览器和服务器中运行的应用。

3.11.1 认识同构应用

现在大多数单页应用的视图都是通过 JavaScript 代码在浏览器端渲染出来的，但在浏览器端渲染的坏处如下。

- 搜索引擎无法收录我们的网页，因为展示的数据都是在浏览器端异步渲染出来的，大部分爬虫无法获取这些数据。
- 对于复杂的单页应用，渲染过程的计算量大，对于低端移动设备来说可能会有性能问题，用户能明显感知首屏的渲染延迟。

为了解决以上问题，有人提出能否将原本只运行在浏览器中的 JavaScript 渲染代码也在服务器端运行，在服务器端渲染出带内容的 HTML 后再返回。这样就能让搜索引擎爬虫直接抓取带数据的 HTML，同时减少首屏渲染时间。Node.js 的流行和成熟，以及虚拟 DOM 的提出与实现，使这个假设成为可能。

实际上，现在主流的前端框架都支持同构，包括 React、Vue2、Angular2，其中最先支持也最成熟的同构方案是 React。由于 React 的使用者更多，所以它们之间很相似，本节只介绍如何用 Webpack 构建 React 同构应用。

同构应用的运行原理的核心在于虚拟 DOM，虚拟 DOM 的意思是不直接操作 DOM，而是通过 JavaScript Object 描述原本的 DOM 结构。在需要更新 DOM 时不直接操作 DOM 树，而是在更新 JavaScript Object 后再映射成 DOM 操作。

虚拟 DOM 的优点如下。

- 因为操作 DOM 树是高耗时的操作，所以尽量减少 DOM 树操作能优化网页的性能。而通过 DOM Diff 算法能找出两个不同 Object 的最小差异，得出最小的 DOM 操作。
- 虚拟 DOM 在渲染时不仅可以通过操作 DOM 树表示结果，也可以有其他表示方

式，例如将虚拟 DOM 渲染成字符串（服务器端渲染），或者渲染成手机 App 原生的 UI 组件（React Native）。

以 React 为例，核心模块 `react` 负责管理 React 组件的生命周期，而具体的渲染工作可以由 `react-dom` 模块负责。

`react-dom` 在渲染虚拟 DOM 树时有两种方式可以选择。

- 通过 `render()` 函数去操作浏览器 DOM 树来展示出结果。
- 通过 `renderToString()` 计算表示虚拟 DOM 的 HTML 形式的字符串。

构建同构应用的最终目的是从一份项目源码中构建出两份 JavaScript 代码，一份用于在浏览器端运行，一份用于在 Node.js 环境中运行并渲染出 HTML。对于要在 Node.js 环境中运行的 JavaScript 代码，需要注意：

- 不能包含浏览器环境提供的 API，例如使用 `document` 进行 DOM 操作，因为 Node.js 不支持这些 API；
- 不能包含 CSS 代码，因为服务端渲染的目的是渲染出 HTML 的内容，渲染出 CSS 代码会增加额外的计算量，影响服务端的渲染性能；
- 不能像用于浏览器环境的输出代码那样将 `node_modules` 里的第三方模块和 Node.js 原生模块（例如 fs 模块）打包进去，而是需要通过 CommonJS 规范引入这些模块；
- 需要通过 CommonJS 规范导出一个渲染函数，用于在 HTTP 服务器中执行这个渲染函数，渲染出 HTML 的内容后返回。

3.11.2 解决方案

接下来改造在 3.6 节中介绍的 React 项目，为它增加构建同构应用的功能。

由于要从一份源码中构建出两份不同的代码，所以需要有两份 Webpack 配置文件分别与之对应。构建用于浏览器环境的配置和前面讲的没有差别，本节侧重于讲解如何构建用于服务端渲染的代码。

用于构建浏览器环境代码的 webpack.config.js 配置文件保持不变，新建一个专门用于构建服务端渲染代码的配置文件 webpack_server.config.js，内容如下：

```
const path = require('path');
const nodeExternals = require('webpack-node-externals');
module.exports = {
  // JavaScript 执行入口文件
  entry: './main_server.js',
  // 为了不将 Node.js 内置的模块打包进输出文件中
  target: 'node',
  // 为了不将 node_modules 目录下的第三方模块打包进输出文件中
  externals: [nodeExternals()],
  output: {
    // 为了以 CommonJS2 规范导出渲染函数，以被采用 Node.js 编写的 HTTP 服务调用
    libraryTarget: 'commonjs2',
    // 将最终可在 Node.js 中运行的代码输出到 bundle_server.js 文件中
    filename: 'bundle_server.js',
    // 将输出文件都放到 dist 目录下
    path: path.resolve(__dirname, './dist'),
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        use: ['babel-loader'],
        exclude: path.resolve(__dirname, 'node_modules'),
      },
      {
        // CSS 代码不能被打包到用于服务端的代码中，忽略 CSS 文件
        test: /\.css/,
        use: ['ignore-loader'],
      },
    ]
  },
  devtool: 'source-map' // 输出 source-map，以方便直接调试 ES6 源码
};
```

以上代码有如下几个关键之处。

- target: 'node'：由于输出代码的运行环境是 Node.js，所以源码中依赖的

Node.js 原生模块没必要被打包进去。

- externals: *nodeExternals()（https://github.com/liady/webpack-node-externals）的目的是防止node_modules 目录下的第三方模块被打包进去，因为Node.js会默认去 node_modules 目录下寻找和使用第三方模块。
- {test: /\.css/, use: ['ignore-loader']}：忽略依赖的 CSS 文件，CSS 会影响服务端的渲染性能，也是做服务端渲染的不重要的部分。
- libraryTarget: 'commonjs2'：以 CommonJS2 规范导出渲染函数，以供采用 Node.js 编写的 HTTP 服务器代码调用。

为了最大限度地复用代码，需要调整目录结构。将页面的根组件放到一个单独的文件 AppComponent.js 中，该文件只能包含根组件的代码，不能包含渲染入口的代码，而且需要导出根组件以供渲染入口调用。AppComponent.js 的内容如下：

```
import React, { Component } from 'react';
import './main.css';
export class AppComponent extends Component {
  render() {
    return <h1>Hello,Webpack</h1>
  }
}
```

分别为不同环境的渲染入口写两份不同的文件，即用于浏览器端渲染 DOM 的 main_browser.js 文件和用于服务端渲染 HTML 字符串的 main_server.js 文件。

main_browser.js 文件的内容如下：

```
import React from 'react';
import { render } from 'react-dom';
import { AppComponent } from './AppComponent';
// 将根组件渲染到 DOM 树上
render(<AppComponent/>, window.document.getElementById('app'));
```

main_server.js 文件的内容如下：

```
import React from 'react';
import { renderToString } from 'react-dom/server';
import { AppComponent } from './AppComponent';
// 导出渲染函数，以供采用 Node.js 编写的 HTTP 服务器代码调用
```

```
export function render() {
  // 将根组件渲染成 HTML 字符串
  return renderToString(<AppComponent/>)
}
```

为了能将渲染的完整 HTML 文件通过 HTTP 服务返回给请求端，还需要用 Node.js 编写一个 HTTP 服务器。由于本节不注重于 HTTP 服务器的实现，所以这里采用 ExpressJS 实现。http_server.js 文件的内容如下：

```
const express = require('express');
const { render } = require('./dist/bundle_server');
const app = express();
// 调用构建出的 bundle_server.js 中暴露出的渲染函数，再拼接 HTML 模板，形成完整的 HTML 文件
app.get('/', function (req, res) {
  res.send(`
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
<div id="app">${render()}</div>
<!--导入 Webpack 输出的用于浏览器端渲染的 JavaScript 文件-->
<script src="./dist/bundle_browser.js"></script>
</body>
</html>
  `);
});
// 其他请求路径返回对应的本地文件
app.use(express.static('.'));
app.listen(3000, function () {
  console.log('app listening on port 3000!')
});
```

再安装新引入的第三方依赖：

```
# 安装 Webpack 构建依赖
npm i -D css-loader style-loader ignore-loader webpack-node-externals
# 安装 HTTP 服务器依赖
npm i -S express
```

以上所有准备工作都已经完成，接下来执行构建，编译出目标文件。

- 执行命令 webpack --config webpack_server.config.js，构建出用于服务端渲染的./dist/bundle_server.js 文件。
- 执行命令 webpack，构建出用于浏览器环境运行的./dist/bundle_browser.js 文件，默认的配置文件为webpack.config.js。

构建执行完成后，执行 node ./http_server.js 启动 HTTP 服务器，再用浏览器去访问 http://localhost:3000，就能看到 Hello,Webpack 了。但是为了验证服务端渲染的结果，需要打开浏览器的开发工具中的网络抓包一栏，再重新刷新浏览器，就能抓到请求 HTML 的包了，抓包效果如图 3-3 所示。

图 3-3 服务端渲染抓包

可以看到服务器返回的是渲染出内容后的 HTML，而不是 HTML 模板，这说明同构应用的改造完成了。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-11 构建同构应用.zip。

3.12 构建 Electron 应用

3.12.1 认识 Electron

Electron（https://electron.atom.io）可以让我们使用开发 Web 的技术去开发跨平台的桌面端

应用，由 GitHub 主导和开源，我们熟悉的 Atom 和 VSCode 编辑器就是使用 Electron 开发的。

Electron 是 Node.js 和 Chromium 浏览器的结合体，用 Chromium 浏览器显示出的 Web 页面作为应用的 GUI，通过 Node.js 和操作系统交互。当我们在 Electron 应用的一个窗口操作时，实际上是在操作一个网页。当我们的操作需要通过操作系统去完成时，网页会通过 Node.js 和操作系统交互。

采用这种方式开发桌面端应用的优点有：

- 降低了开发门槛，只需掌握网页开发技术和 Node.js，大量的 Web 开发技术和现成库可以复用于 Electron；
- 由于 Chromium 浏览器和 Node.js 都是跨平台的，所以 Electron 能做到在不同的操作系统中运行一份代码。

在运行 Electron 应用时，会从启动一个主进程开始。主进程的启动是通过 Node.js 执行一个 JavaScript 入口文件实现的，这个入口文件是 main.js，其内容如下：

```
const { app, BrowserWindow } = require('electron')
// 保持一个对 window 对象的全局引用，如果我们不这样做，
// 则当 JavaScript 对象被垃圾回收时，window 会被自动地关闭
let win
// 打开主窗口
function createWindow() {
  // 创建浏览器窗口
  win = new BrowserWindow({ width: 800, height: 600 })
  // 加载应用的 index.html
  const indexPageURL = `file://${__dirname}/dist/index.html`;
  win.loadURL(indexPageURL);
  // 当 Window 被关闭时，这个事件会被触发
  win.on('closed', () => {
    // 取消引用 window 对象
    win = null
  })
}
// Electron 会在创建浏览器窗口时调用这个函数
app.on('ready', createWindow)
// 当全部窗口关闭时退出
app.on('window-all-closed', () => {
```

```
  // 在 MacOS 上，除非用户用 Cmd + Q 确定地退出，否则绝大部分应用会保持激活状态
  if (process.platform !== 'darwin') {
    app.quit()
  }
})
```

主进程启动后会一直驻留在后台运行，我们所看到的和操作的窗口并不是主进程，而是由主进程新启动的窗口子进程。

应用从启动到退出有一系列生命周期事件，可通过 `electron.app.on()` 函数去监听生命周期事件，在特定的时刻做出反应。例如在 `app.on('ready')` 事件中通过 `BrowserWindow` 去展示应用的主窗口，若想了解具体用法，则可以参考 BrowserWindow 的 API 文档（https://github.com/electron/electron/blob/master/ docs-translations/zh-CN/api/browser- window.md）。

启动的窗口其实是一个网页，启动时会去加载在 `loadURL` 中传入的网页地址。每个窗口都是一个单独的网页进程，窗口之间需要借助主进程传递消息。

Electron 的应用架构如图 3-4 所示。

图 3-4 Electron 应用架构图

总体来说，开发 Electron 应用和开发 Web 应用很相似，区别在于 Electron 的运行环境同时内置了浏览器和 Node.js 的 API，在开发网页时除了可以使用浏览器提供的 API，还可以使用 Node.js 提供的 API。

3.12.2 接入 Webpack

接下来做一个简单的 Electron 应用，要求在应用启动后显示一个主窗口，在主窗口里有一个按钮，单击这个按钮后会重新显示一个窗口，并且通过使用 React 开发网页。

由于 Electron 应用中的每个窗口对应一个网页，所以需要开发两个网页，分别是主窗口的 index.html 和新打开的窗口 login.html。也就是说，项目由两个单页应用组成，这和 3.10 节中的项目非常相似，让我们来将它改造成一个 Electron 应用。

需要改动的地方如下。

（1）修改在项目根目录下新建主进程的入口文件 main.js，内容和上面提到的一致。

（2）修改主窗口网页的代码如下：

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import { remote } from 'electron';
import path from 'path';
import './index.css';
class App extends Component {
  // 在按钮被单击时
  handleBtnClick() {
    // 新窗口对应的页面的 URL 地址
    const modalPath = path.join('file://', remote.app.getAppPath(),
'dist/login.html');
    // 新窗口的大小
    let win = new remote.BrowserWindow({ width: 400, height: 320 })
    win.on('close', function () {
      // 在窗口被关闭时清空资源
      win = null
    })
    // 加载网页
```

```
    win.loadURL(modalPath)
    // 显示窗口
    win.show()
  }
  render() {
    return (
      <div>
        <h1>Page Index</h1>
        <button onClick={this.handleBtnClick}>Open Page Login</button>
      </div>
    )
  }
}
render(<App/>, window.document.getElementById('app'));
```

其中最关键的部分在于，在按钮单击事件里通过 electron 库里提供的 API 重新打开一个窗口，并加载网页文件所在的地址。

页面部分的代码已经修改完成，接下来修改构建方面的代码。这里，构建需要做到以下几点。

- 构建出两个可在浏览器里运行的网页，分别对应两个窗口的界面。
- 在网页的 JavaScript 代码里可能会调用 Node.js 原生模块或者 Electron 模块，输出的代码依赖这些模块，但由于这些模块都是内置支持的，所以构建出的代码不能将这些模块打包进去。

很容易完成以上要求，因为 Webpack 内置了对 Electron 的支持，只需要为 Webpack 配置文件加上一行代码即可：

```
target: 'electron-renderer',
```

在 2.7 节讲解 Target 时曾提到该配置，意思是让 Webpack 构建出用于 Electron 渲染进程用的 JavaScript 代码，即这两个窗口需要的网页代码。

在完成以上修改后重新执行 Webpack 构建，对应的网页需要的代码都输出到了项目根目录下的 dist 目录里。

为了以 Electron 应用的形式运行，还需要安装新的依赖：

```
# 安装 Electron 执行环境到项目中
```

```
npm i -D electron
```

安装成功后在项目目录下执行命令 `electron ./`，我们就能看到启动的桌面应用了，效果如图 3-5 所示。

图 3-5　Electron 应用运行效果图

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-12 构建 Electron 应用.zip。

3.13　构建 Npm 模块

3.13.1　认识 Npm

Npm（https://www.npmjs.com）是目前最大的 JavaScript 模块仓库，里面有全世界的开发者上传的可复用模块。虽然在大多数情况下我们都是这些开放模块的使用者，但我们也许会成为贡献者，会开发一个模块上传到 Npm 仓库。

发布到 Npm 仓库的模块有以下几个特点。

- 在每个模块根目录下都必须有一个描述该模块的 package.json 文件。该文件描述了模块的入口文件是哪个，该模块又依赖哪些模块等。若想深入了解，则可以阅读 package.json 文件（http://javascript.ruanyifeng.com/nodejs/packagejson.html）。
- 模块中的文件以 JavaScript 文件为主，但不限于 JavaScript 文件。例如一个 UI 组件可能同时需要 JavaScript、CSS、图片文件等。
- 模块中的代码大多采用模块化规范，因为我们的某个模块可能依赖其他模块，而别的模块又可能依赖我们的这个模块。目前支持比较广泛的是 CommonJS 模块化规范，上传到 Npm 仓库的代码最好遵守该规范。

3.13.2 抛出问题

Webpack 不仅用于构建可运行的应用，也用于构建可上传到 Npm 的模块。接下来讲解如何用 Webpack 构建一个可上传到 Npm 仓库的 React 组件，具体要求如下。

- 源代码采用 ES6 编写，但发布到 Npm 仓库时需要转换成 ES5 代码，并且遵守 CommonJS 模块化规范。如果发布到 Npm 上的 ES5 代码是经过转换的，则请同时提供 Source Map 以方便调试。
- 该 UI 组件依赖的其他资源文件如 CSS 文件也需要包含在发布的模块里。
- 尽量减少冗余代码，减少发布出去的组件的代码文件大小。
- 在发布出去的组件的代码中不能含有其依赖的模块的代码，而是让用户可选择性地安装。例如不能内嵌 React 库的代码，这样做的目的是，在其他组件也依赖 React 库时，防止 React 库的代码被重复打包。

在开始前先看看最终发布到 Npm 仓库的模块的目录结构：

```
node_modules/hello-webpack
├── lib
│   ├── index.css（组件所有依赖的 CSS 都在这个文件中）
│   ├── index.css.map
│   ├── index.js（符合 CommonJS 模块化规范的 ES5 代码）
│   └── index.js.map
```

```
├── src（ES6 源码）
│   ├── index.css
│   └── index.js
└── package.json（模块描述文件）
```

本节的重点不是 React，而是 Webpack，所以这里写一个简单的 React 组件，其代码放在 src/index.js 文件中，内容如下：

```
import React, { Component } from 'react';
import './index.css';
// 导出该组件以供其他模块使用
export default class HelloWebpack extends Component {
  render() {
    return <h1 className="hello-component">Hello,Webpack</h1>
  }
}
```

使用该模块时只需要这样做：

```
// 通过 ES6 语法导入
import HelloWebpack from 'hello-webpack';
import 'hello-webpack/lib/index.css';

// 或者通过 ES5 语法导入
var HelloWebpack = require('hello-webpack');
require('hello-webpack/lib/index.css');

// 使用 react-dom 渲染
render(<HelloWebpack/>);
```

3.13.3 使用 Webpack 构建 Npm 模块

接下来用 Webpack 一条一条地应对上面所抛出问题的 4 点要求。

（1）对于要求 1，可以这样做，如下所述。

- 使用 babel-loader 将 ES6 代码转换成 ES5 代码。
- 通过开启 devtool: 'source-map'输出 Source Map 以发布调试。

- 设置 output.libraryTarget='commonjs2'，使输出的代码符合 CommonJS2 模块化规范，以供其他模块导入使用。在 2.2 节介绍过该配置的含义。

相关的 Webpack 配置代码如下：

```
module.exports = {
  output: {
    // 输出的代码符合 CommonJS 模块化规范，以供其他模块导入使用。
    libraryTarget: 'commonjs2',
  },
  // 输出 Source Map
  devtool: 'source-map',
};
```

（2）对于要求 2，需要通过 css-loader 和 extract-text-webpack-plugin 实现，相关的 Webpack 配置代码如下：

```
const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  module: {
    rules: [
      {
        // 增加对 CSS 文件的支持
        test: /\.css/,
        // 提取出 Chunk 中的 CSS 代码到单独的文件中
        use: ExtractTextPlugin.extract({
          use: ['css-loader']
        }),
      },
    ]
  },
  plugins: [
    new ExtractTextPlugin({
      // 输出的 CSS 文件的名称
      filename: 'index.css',
    }),
  ],
};
```

此步引入了 3 个新依赖：

```
# 安装 Webpack 构建所需要的新依赖
npm i -D style-loader css-loader extract-text-webpack-plugin
```

（3）对于要求 3，需要注意的是 Babel 在将 ES6 代码转换成 ES5 代码时会注入一些辅助函数。

例如下面这段 ES6 代码：

```
class HelloWebpack extends Component{
}
```

这段代码在被转换成能正常运行的 ES5 代码时需要以下两个辅助函数。

- babel-runtime/helpers/createClass：用于实现 class 语法。
- babel-runtime/helpers/inherits：用于实现 extends 语法。

在默认情况下，Babel 会在每个输出文件中内嵌这些依赖的辅助函数的代码，如果多个源代码文件都依赖这些辅助函数，那么这些辅助函数的代码将会重复出现很多次，造成代码冗余。为了不让这些辅助函数的代码重复出现，可以在依赖它们时通过 require('babel-runtime/helpers/createClass'）的方式导入，这样就能做到只让它们出现一次。babel-plugin-transform-runtime（https://babeljs.io/docs/plugins/transform- runtime/）插件就是用来做这件事的。

修改.babelrc 文件，为其加入 transform-runtime 插件：

```
{
  "plugins": [
    [
      "transform-runtime",
      {
        // transform-runtime 默认会自动为我们使用的 ES6 API 注入 polyfill
        // 假如在源码中使用了 Promise，则输出的代码将会自动注入
require('babel-runtime/core-js/Promise')语句
        // polyfill 的注入应该交给模块使用者，因为使用者可能在其他地方注入了其他
Promise polyfill 库
        // 所以关闭该功能
        "polyfill": false
      }
    ]
```

```
  ]
}
```

由于加入 babel-plugin-transform-runtime 后，在生成的代码中会大量出现类似于 `require('babel-runtime/helpers/createClass')` 的语句，所以输出的代码将依赖 babel-runtime 模块。

此步引入了 3 个新的依赖：

```
# 安装 Webpack 构建所需的新依赖
npm i -D babel-plugin-transform-runtime
# 安装输出代码运行时所需的新依赖
npm i -S babel-runtime
```

（4）对于要求 4，需要通过在 2.7 节介绍过的 Externals 实现。

Externals 用来告诉在 Webpack 要构建的代码中使用了哪些不用被打包的模块，也就是说这些模板是外部环境提供的，Webpack 在打包时可以忽略它们。

相关的 Webpack 配置代码如下：

```
module.exports = {
  // 通过正则命中所有以 react 或者 babel-runtime 开头的模块
  // 这些模块通过注册在运行环境中的全局变量访问，不用被重复打包进输出的代码里
  externals: /^(react|babel-runtime)/,
};
```

开启以上配置后，在输出的代码中会存在导入 react 或者 babel-runtime 模块的代码，但是它们的 react 或者 babel-runtime 的内容不会被包含进去：

```
[
    (function (module, exports) {
        module.exports = require("babel-runtime/helpers/inherits");
    }),
    (function (module, exports) {
        module.exports = require("react");
    })
]
```

这样就做到了在保持代码正确性的情况下，输出文件不存放 react 或者 babel-runtime 模块的代码。

实际上，在开发 Npm 模块时，不只需要对 react 和 babel-runtime 模块进行这样的处理，还需要对所有正在开发的模块所依赖的模块进行这样的处理。因为正在开发的模块所依赖的模块也可能被其他模块所依赖。当一个项目中的一个模块被依赖多次时，Webpack 只会将其打包一次。若想更深入地了解这方面的内容，则可以阅读 5.2 节。

完成以上 4 步后，最终的 Webpack 的完整配置代码如下：

```
const path = require('path');
const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  // 模块的入口文件
  entry: './src/index.js',
  output: {
    // 输出文件的名称
    filename: 'index.js',
    // 输出文件的存放目录
    path: path.resolve(__dirname, 'lib'),
    // 输出的代码符合 CommonJS 模块化规范，以供给其他模块导入使用。
    libraryTarget: 'commonjs2',
  },
  // 通过正则命中所有以 react 或者 babel-runtime 开头的模块，
  // 这些模块通过注册在运行环境中的全局变量访问，不能被打包进输出的代码里，防止它们出现多次。
  externals: /^(react|babel-runtime)/,
  module: {
    rules: [
      {
        test: /\.js$/,
        use: ['babel-loader'],
        // 排除 node_modules 目录下的文件，
        // node_modules 目录下的文件都采用了 ES5 语法，没必要再通过 Babel 转换。
        exclude: path.resolve(__dirname, 'node_modules'),
      },
      {
        // 增加对 CSS 文件的支持
        test: /\.css/,
        // 提取 Chunk 中的 CSS 代码到单独的文件中
        use: ExtractTextPlugin.extract({
```

```
        use: ['css-loader']
      }),
    },
  ]
},
plugins: [
  new ExtractTextPlugin({
    // 输出的 CSS 文件名称
    filename: 'index.css',
  }),
],
// 输出 Source Map
devtool: 'source-map',
};
```

重新执行构建后，我们将会在项目目录下看到一个新目录 lib，里面放着要发布到 Npm 仓库的最终代码。

3.13.4 发布到 Npm

在将构建出的代码发布到 Npm 仓库前，还需要确保我们的模块描述文件 package.json 已正确配置。

由于构建出的代码的入口文件是./lib/index.js，所以需要修改 package.json 中的 main 字段：

```
{
  "main": "lib/index.js",
  "jsnext:main": "src/index.js"
}
```

其中，jsnext:main 字段用于指出采用 ES6 编写的模块入口文件所在的位置，这样做是为了便于实现在 4.10 节中介绍的 Tree Sharking。

修改完毕后在项目目录下执行 npm publish，就能将构建出的代码发布到 Npm 仓库中（确保已经 npm login）。

如果想让发布到 Npm 的代码同源码的目录结构保持一致，那么用 Webpack 将不再适合，因为源码是一个个分割的模块化文件，而 Webpack 会将这些模块组合在一起。虽然 Webpack 输出的文件也可以采用 CommonJS 模块化语法，但在某些场景下将所有模块打包成一个文件发布到 Npm 是不适合的。例如，像 Lodash 这样的工具函数库在项目中可能只用到了其中几个工具函数，如果将所有工具函数打包在一个文件中，那么所有工具函数都会被打包进去，而保持模块文件的独立能做到只打包要使用的；另外，像 UI 组件库这样的由大量独立组件组成的库也同 Lodash 类似。

所以，Webpack 适用于构建完整不可分割的 Npm 模块。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-13 构建 Npm 库.zip。

3.14 构建离线应用

3.14.1 认识离线应用

即使将网页的性能优化得非常好，如果网络不好，则也会导致网页的体验很差。离线应用是指通过离线缓存技术，让资源在第一次被加载后缓存在本地，在下次访问它时就直接返回本地的文件，即使没有网络连接。

离线应用有以下优点：

- 在没有网络的情况下也能打开网页；
- 由于部分被缓存的资源直接从本地加载，所以对用户来说可以加快网页的加载速度，对网站运营者来说可以减少服务器的压力及传输流量费用。

离线应用的核心是离线缓存技术，历史上曾先后出现两种离线离线缓存技术，如下所述。

- AppCache（https://developer.mozilla.org/zh-CN/docs/Web/HTML/Using_the_application_cache）：又叫作 Application Cache，目前已经从 Web 标准中删除，请尽量不要使用它。

- Service Workers（https://developer.mozilla.org/zh-CN/docs/Web/API/Service_Worker_API/Using_Service_Workers）：是目前最新的离线缓存技术，是 Web Worker（http://javascript.ruanyifeng.com/htmlapi/webworker.html）的一部分。它通过拦截网络请求实现离线缓存，比 AppCache 更灵活。它也是构建 PWA（https://developer. mozilla.org/zh-CN/Apps/Progressive）应用的关键技术之一。

与 AppCache 相比，Service Workers 更灵活，因为它可以通过 JavaScript 代码去控制缓存的逻辑。由于第 1 种技术已被废弃，所以本节只专注于讲解如何用 Webpack 构建使用了 Service Workers 的网页。

3.14.2 认识 Service Workers

Service Workers 是一个在浏览器后台运行的脚本，它的生命周期完全独立于网页。它无法直接访问 DOM，但可以通过 postMessage 接口发送消息来和 UI 进程通信。拦截网络请求是 Service Workers 的重要功能，通过 Service Workers 能完成离线缓存、编辑响应、过滤响应等功能。

若你想更深入地了解 Service Workers，则请阅读服务工作线程简介（https://developers.google.com/web/fundamentals/getting-started/primers/service-workers?hl=zh-cn）。

1. Service Workers 兼容性

目前 Chrome、Firefox 和 Opera 都已经全面支持 Service Workers，但只有高版本的 Android 支持移动端的浏览器。由于 Service Workers 无法通过注入 polyfill 实现兼容，所以在打算使用它前，请先弄明白自己的网页的运行场景。

判断浏览器是否支持 Service Workers 的最简单方法是通过以下代码：

```
// 如果 navigator 对象上存在 serviceWorker 对象，就表示支持
if (navigator.serviceWorker) {
  // 通过 navigator.serviceWorker 使用
}
```

2. 注册 Service Workers

要为网页接入 Service Workers，就需要在网页加载后注册一个描述 Service Workers 逻辑的脚本。代码如下：

```
if (navigator.serviceWorker) {
  window.addEventListener('DOMContentLoaded',function() {
    // 调用 serviceWorker.register 注册，参数/sw.js 为脚本文件所在的 URL 路径
      navigator.serviceWorker.register('/sw.js');
  });
}
```

一旦这个脚本文件被加载，Service Workers 的安装就开始了。在这个脚本被安装到浏览器中后，就算用户关闭了当前网页，它仍会存在。也就是说第一次打开该网页时，Service Workers 的逻辑不会生效，因为脚本还没有被加载和注册，但是以后再次打开该网页时脚本里的逻辑将会生效。

在 Chrome 中可以通过打开网址 `chrome://inspect/#service-workers` 来查看当前浏览器中所有已注册的 Service Workers。

3. 使用 Service Workers 实现离线缓存

Service Workers 在注册成功后会在其生命周期中派发一些事件，通过监听对应的事件在特点的时间节点上做一些事情。

在 Service Workers 脚本中引入了新的关键字 `self`，代表当前的 Service Workers 实例。

在 Service Workers 安装成功后会派发出 `install` 事件，需要在这个事件中执行缓存资源的逻辑，实现代码如下：

```
// 当前缓存版本的唯一标识符，用当前时间代替
var cacheKey = new Date().toISOString();

// 需要被缓存的文件的 URL 列表
var cacheFileList = [
  '/index.html',
  '/app.js',
  '/app.css'
];
```

```
// 监听 install 事件
self.addEventListener('install', function (event) {
  // 等待所有资源缓存完成时，才可以进行下一步
  event.waitUntil(
    caches.open(cacheKey).then(function (cache) {
      // 要缓存的文件 URL 列表
      return cache.addAll(cacheFileList);
    })
  );
});
```

接下来需要监听网络请求事件去拦截请求、复用缓存，代码如下：

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    // 去缓存中查询对应的请求
    caches.match(event.request).then(function(response) {
        // 如果命中本地缓存，就直接返回本地的资源
        if (response) {
          return response;
        }
        // 否则就用 fetch 下载资源
        return fetch(event.request);
      }
    )
  );
});
```

这样就实现了离线缓存。

4. 更新缓存

线上的代码有时需要更新和重新发布，如果这个文件被离线缓存了，就需要在 Service Workers 脚本中有对应的逻辑去更新缓存。这可以通过更新 Service Workers 脚本文件做到。

浏览器针对 Service Workers 有如下机制。

- 每次打开接入了 Service Workers 的网页时，浏览器都会重新下载 Service Workers 脚本文件（所以要注意该脚本文件不能太大）。如果发现和当前已经注册过的文件

存在字节差异，就将其视为“新服务工作线程”。

- 新的 Service Workers 线程将会启动，且将会触发其 install 事件。
- 当网站上当前打开的页面关闭时，旧的 Service Workers 线程将会被终止，新的 Service Workers 线程将会取得控制权。
- 新的 Service Workers 线程取得控制权后，将会触发其 activate 事件。

新的 Service Workers 线程中的 activate 事件就是清理旧缓存的最佳时间点，代码如下：

```
// 当前缓存的白名单，在新脚本的 install 事件里将使用白名单里的 key
var cacheWhitelist = [cacheKey];

self.addEventListener('activate', function(event) {
  event.waitUntil(
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        cacheNames.map(function(cacheName) {
          // 将不在白名单中的缓存全部清理掉
          if (cacheWhitelist.indexOf(cacheName) === -1) {
            // 删除缓存
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

最终，完整的 Service Workers 脚本代码如下：

```
// 当前缓存版本的唯一标识符，用当前时间代替
var cacheKey = new Date().toISOString();

// 当前缓存的白名单，在新脚本的 install 事件里将使用白名单里的 key
var cacheWhitelist = [cacheKey];

// 需要被缓存的文件的 URL 列表
var cacheFileList = [
  '/index.html',
```

```
  'app.js',
  'app.css'
];

// 监听 install 事件
self.addEventListener('install', function (event) {
  // 等所有资源缓存完成时，才可以进行下一步
  event.waitUntil(
    caches.open(cacheKey).then(function (cache) {
      // 要缓存的文件 URL 列表
      return cache.addAll(cacheFileList);
    })
  );
});

// 拦截网络请求
self.addEventListener('fetch', function (event) {
  event.respondWith(
    // 去缓存中查询对应的请求
    caches.match(event.request).then(function (response) {
      // 如果命中本地缓存，就直接返回本地的资源
      if (response) {
        return response;
      }
      // 否则就用 fetch 下载资源
      return fetch(event.request);
    }
  )
);
});

// 新的 Service Workers 线程取得控制权后，将会触发 activate 事件
self.addEventListener('activate', function (event) {
  event.waitUntil(
    caches.keys().then(function (cacheNames) {
      return Promise.all(
        cacheNames.map(function (cacheName) {
          // 将不在白名单中的缓存全部清理掉
          if (cacheWhitelist.indexOf(cacheName) === -1) {
            // 删除缓存
```

```
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

3.14.3 接入 Webpack

用 Webpack 构建接入 Service Workers 的离线应用时，要解决的关键问题在于如何生成上面提到的 sw.js 文件。并且 sw.js 文件中的 cacheFileList 变量，代表需要被缓存文件的 URL 列表，需要根据输出文件列表所对应的 URL 来决定，而不是像上面那样写成静态值。

假如构建输出的文件目录结构为：

```
├── app_4c3e186f.js
├── app_7cc98ad0.css
└── index.html
```

那么，sw.js 文件中 cacheFileList 的值应该是：

```
var cacheFileList = [
  '/index.html',
  'app_4c3e186f.js',
  'app_7cc98ad0.css'
];
```

Webpack 没有原生功能可以完成以上要求，幸好庞大的社区中已经有人为我们做好一个插件 serviceworker-webpack-plugin（https://github.com/oliviertassinari/serviceworker-webpack-plugin），通过它可以方便地解决以上问题。使用该插件后的 Webpack 配置如下：

```
const path = require('path');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
const { WebPlugin } = require('web-webpack-plugin');
const ServiceWorkerWebpackPlugin =
require('serviceworker-webpack-plugin');

module.exports = {
```

```
  entry: {
    app: './main.js'// Chunk app 的 JavaScript 执行入口文件
  },
  output: {
    filename: '[name].js',
    publicPath: '',
  },
  module: {
    rules: [
      {
        test: /\.css/,// 增加对 CSS 文件的支持
        // 提取出 Chunk 中的 CSS 代码到单独的文件中
        use: ExtractTextPlugin.extract({
          use: ['css-loader'] // 压缩 CSS 代码
        }),
      },
    ]
  },
  plugins: [
    // 一个 WebPlugin 对应一个 HTML 文件
    new WebPlugin({
      template: './template.html', // HTML 模板文件所在的文件路径
      filename: 'index.html' // 输出的 HTML 的文件名称
    }),
    new ExtractTextPlugin({
      filename: `[name].css`,// 为输出的 CSS 文件名称加上 Hash 值
    }),
    new ServiceWorkerWebpackPlugin({
      // 自定义的 sw.js 文件所在的路径
      // ServiceWorkerWebpackPlugin 会将文件列表注入生成的 sw.js 中
      entry: path.join(__dirname, 'sw.js'),
    }),
  ],
  devServer: {
    // Service Workers 依赖 HTTPS，使用 DevServer 提供的 HTTPS 功能。
    https: true,
  }
};
```

在以上配置中有两点需要注意：

- 由于 Service Workers 必须在 HTTPS 环境下才能拦截网络请求来实现离线缓存，所以这里通过在 2.6 节中提到的方式去实现 HTTPS 服务；
- serviceworker-webpack-plugin 插件为了保证灵活性，允许使用者自定义 sw.js，构建输出的 sw.js 文件中会在头部注入一个变量 serviceWorkerOption.assets 到全局，里面存放着所有需要被缓存的文件的 URL 列表。

需要将上面的 sw.js 文件中被写成了静态值的 cacheFileList 修改如下：

```
// 需要被缓存的文件的 URL 列表
var cacheFileList = global.serviceWorkerOption.assets;
```

以上已经完成所有文件的修改，在重新构建前先安装新引入的依赖：

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

安装成功后，在项目根目录下执行 webpack-dev-server 命令，DevServer 将以 HTTPS 模式启动，并输出如下日志：

```
> webpack-dev-server

Project is running at https://localhost:8080/
webpack output is served from /
Hash: 402ee6ce5bffb16dffe2
Version: webpack 3.5.5
Time: 619ms
     Asset       Size  Chunks                    Chunk Names
    app.js     325 kB       0  [emitted]  [big]  app
   app.css   21 bytes       0  [emitted]         app
index.html  235 bytes          [emitted]
     sw.js    4.86 kB          [emitted]
```

用 Chrome 浏览器打开网址 https://localhost:8080/index.html 后，就能访问接入 Service Workers 离线缓存的页面了。

3.14.4 验证结果

为了验证 Service Workers 和缓存是否生效，需要通过 Chrome 的开发者工具来查看。

通过打开开发者工具的 Application-Service Workers 一栏，就能看到当前页面注册的 Service Workers，正常的效果如图 3-6 所示。

图 3-6　查看当前页面注册的 Service Workers

打开开发者工具的 Application-Cache-Cache Storage 一栏，就能看到当前页面缓存的资源列表，正常的效果如图 3-7 所示。

图 3-7　查看当前页面的 Cache Storage

为了验证网页在离线时的访问能力，需要在开发者工具中的 Network 一栏通过 Offline 选项禁用网络，再刷新页面使其能正常访问，并且网络请求的响应都来自 Service Workers，正常的效果如图 3-8 所示。

图 3-8　在离线情况下访问页面

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-14 构建离线应用.zip。

3.15 搭配 Npm Script

3.15.1 认识 Npm Script

Npm Script（https://docs.npmjs.com/misc/scripts）是一个任务执行者。Npm 是在安装 Node.js 时附带的包管理器，Npm Script 则是 Npm 内置的一个功能，允许在 package.json 文件里使用 scripts 字段定义任务：

```
{
  "scripts": {
    "dev": "node dev.js",
    "pub": "node build.js"
  }
}
```

以上代码中的 scripts 字段是一个对象，每个属性对应一段脚本，以上代码定义了两个任务 dev 和 pub。Npm Script 的底层实现原理是通过调用 Shell 去运行脚本命令，例如执行 npm run pub 命令等同于执行 node build.js 命令。

Npm Script 还有一个重要的功能，是能运行安装到项目目录的 node_modules 里的可执行模块，例如在通过命令：

```
npm i -D webpack
```

将 Webpack 安装到项目中后，是无法直接在项目根目录下通过命令 webpack 去执行 Webpack 构建的，而是要通过如下命令去执行：

```
./node_modules/.bin/webpack
```

Npm Script 能方便地解决这个问题，只需要在 scripts 字段里定义一个任务，例如：

```
{
  "scripts": {
    "build": "webpack"
  }
}
```

Npm Script 会先去项目目录下的 node_modules 中寻找有没有可执行的 webpack 文件，如果有就使用本地的，如果没有就使用全局的。所以现在执行 Webpack 构建时，只需要通过执行 npm run build 实现。

3.15.2 Webpack 为什么需要 Npm Script

Webpack 只是一个打包模块化代码的工具，并没有提供任何任务管理相关的功能。但在实际场景中通常不会是只通过执行 webpack 就能完成所有任务的，而是需要多个任务才能完成。

举一个常见的例子，要求如下。

- 在开发阶段为了提高开发体验，使用 DevServer 做开发，并且需要输出 Source Map 以方便调试，同时需要开启自动刷新功能。
- 为了减小发布到线上的代码尺寸，在构建出发布到线上的代码时，需要压缩输出的代码。
- 在构建完发布到线上的代码后，需要将构建出的代码提交给发布系统。

可以看出要求 1 和要求 2 是相互冲突的，其中要求 3 又依赖要求 2。要满足以上三个要求，需要定义三个不同的任务。

接下来通过 Npm Script 定义上面的 3 个要求：

```
"scripts": {
  "dev": "webpack-dev-server --open",
  "dist": "NODE_ENV=production webpack --config webpack_dist.config.js",
  "pub": "npm run dist && rsync dist"
},
```

含义分别如下。

- dev 代表用于开发时执行的任务，通过 DevServer 启动构建。所以在开发项目时只需执行 npm run dev。
- dist 代表构建出用于发布到线上的代码，输出到 dist 目录中。其中的 NODE_ENV=production 用于在运行任务时注入环境变量。

- pub 代表先构建出用于发布到线上的代码，再同步 dist 目录中的文件到发布系统（如何同步文件，则需根据我们所使用的发布系统而定），所以在开发完成后需要发布时只需执行 npm run pub。

使用 Npm Script 的好处是将一连串复杂的流程简化成了一个简单的命令，在需要时只需执行对应的简短命令，而不用手动重复整个流程。这会大大提高我们的效率并降低出错率。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-15 搭配 NpmScript.zip。

3.16 检查代码

当项目代码变得日益庞大、复杂时，如何保障代码质量？如何保障多人协助开发时代码的可读性？

完全解决以上问题不是一件简单的事，但做代码检查能解决大部分问题。本节将讲解如何结合构建做代码检查。

3.16.1 代码检查具体是做什么的

检查代码和 Code Review 很相似，都是审视提交的代码可能存在的问题。但 Code Review 一般由人执行，而检查代码是通过机器执行一些自动化的检查。自动化地检查代码的成本更低，实施代价更小。

检查代码时主要检查以下几项。

- 代码风格：让项目成员强制遵守统一的代码风格，例如如何缩紧、如何写注释等，保障代码的可读性，不将时间浪费在争论如何使代码更好看上。
- 潜在问题：分析代码在运行过程中可能出现的潜在 Bug。

其中，检查代码风格相关的工具很多，也很成熟。由于情况复杂，对潜在问题的检查目前还没有成熟的工具。

目前已经有成熟的工具可以检验诸如 JavaScript、TypeScript、CSS、SCSS 等常用语言。

3.16.2 怎么做代码检查

在做代码风格检查时需要按照不同的文件类型来检查，下面分别介绍。

1. 检查 JavaScript

目前最常用的 JavaScript 检查工具是 ESlint（https://eslint.org），它不仅内置了大量的常用检查规则，还可以通过插件机制做到灵活扩展。

ESlint 的使用很简单，在通过：

```
npm i -g eslint
```

安装到全局后，再在项目目录下执行：

```
eslint init
```

来新建一个 ESlint 配置文件.eslintrc，该文件的格式为 JSON。

如果想覆盖默认的检查规则，或者想加入新的检查规则，则需要修改该文件，例如使用以下配置：

```
{
    // 从 eslint:recommended 中继承所有检查规则
    "extends": "eslint:recommended",
    // 再自定义一些规则
    "rules": {
        // 需要在每行结尾加;
        "semi": ["error", "always"],
        // 需要使用""包裹字符串
        "quotes": ["error", "double"]
    }
}
```

写好配置文件后，再执行：

```
eslint yourfile.js
```

去检查 yourfile.js 文件，如果我们的文件没有通过检查，则 ESlint 会输出出错的原因，例如：

```
/yourfile.js
  296:13  error  Strings must use doublequote  quotes
  298:7   error  Missing semicolon             semi

✖ 2 problems (2 errors, 0 warnings)
```

ESlint 还有很多功能和检查规则，由于篇幅有限，这里就不详细介绍，可以到其官网阅读相关文档。

2. 检查 TypeScript

TSLint（https://palantir.github.io/tslint/）是一个和 ESlint 相似的 TypeScript 代码检查工具，区别在于 TSLint 只专注于检查 TypeScript 代码。

TSLint 和 ESlint 的使用方法很相似，首先通过：

```
npm i -g tslint
```

安装到全局，再去项目根目录下执行：

```
tslint --init
```

生成配置文件 tslint.json，在配置好后，再执行：

```
tslint yourfile.ts
```

去检查 yourfile.ts 文件。

3. 检查 CSS

stylelint（https://stylelint.io）是目前最成熟的 CSS 检查工具，在内置了大量检查规则的同时，也提供了插件机制让用户自定义扩展。stylelint 基于 PostCSS，能检查任何 PostCSS 能解析的代码，例如 SCSS、Less 等。

首先通过：

```
npm i -g stylelint
```

安装到全局后，去项目根目录下新建.stylelintrc配置文件，该配置文件的格式为JSON，其格式和ESLint的配置相似，例如：

```
{
  // 继承stylelint-config-standard中所有的检查规则
  "extends": "stylelint-config-standard",
  // 再自定义检查规则
  "rules": {
    "at-rule-empty-line-before": null
  }
}
```

配置好后，再执行：

```
stylelint "yourfile.css"
```

去检查yourfile.css文件。

stylelint还有很多功能和配置项在这里没有介绍到，可以访问其官方进一步了解。

目前有很多编辑器如Webstorm、VSCode等已经集成了以上介绍的检查工具，编辑器会将检查工具输出的错误实时地显示到编辑的源码上。通过编辑器集成后，不用通过命令行的方式去定位错误。

3.16.3 结合Webpack检查代码

以上介绍的代码检查工具可以和Webpack结合，在开发过程中通过Webpack输出实时的检查结果。

1. 结合ESLint

eslint-loader（https://github.com/MoOx/eslint-loader）可以方便地将ESLint整合到Webpack中，使用方法如下：

```
module.exports = {
  module: {
    rules: [
      {
```

```
          test: /\.js$/,
          // 不用检查 node_modules 目录下的代码
          include: /node_modules/,
          loader: 'eslint-loader',
          // 将 eslint-loader 的执行顺序放在最前面，防止其他 Loader 将处理后的代码交给 eslint-loader 去检查
          enforce: 'pre',
        },
      ],
    },
  }
```

接入 eslint-loader 后，就能在控制台中看到 ESLint 输出的错误日志了。

2. 结合 TSLint

tslint-loader（https://github.com/wbuchwalter/tslint-loader）是一个和 eslint-loader 相似的 Webpack Loader，能方便地将 TSLint 整合到 Webpack 中，其使用方法如下：

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        //不用检查 node_modules 目录下的代码
        include: /node_modules/,
        loader: 'tslint-loader',
        // 将 tslint-loader 的执行顺序放到最前面，防止其他 Loader 将处理后的代码交给 tslint-loader 去检查
        enforce: 'pre',
      },
    ],
  },
}
```

3. 结合 stylelint

StyleLintPlugin（https://github.com/JaKXz/stylelint-webpack-plugin）能将 stylelint 整合到 Webpack 中，其使用方法很简单，代码如下：

```
const StyleLintPlugin = require('stylelint-webpack-plugin');

module.exports = {
  // ...
  plugins: [
    new StyleLintPlugin(),
  ],
}
```

4. 一些建议

将代码检查功能整合到 Webpack 中会导致以下问题：

- 由于执行检查步骤的计算量大，所以整合到 Webpack 中会导致构建变慢；
- 在整合代码检查到 Webpack 后，输出的错误信息是通过行号来定位错误的，没有编辑器集成显示错误直观。

为了避免以上问题，还可以这样做：

- 使用集成了代码检查功能的编辑器，让编辑器实时、直观地显示错误；
- 将代码检查步骤放到代码提交时，也就是说在代码提交前调用以上检查工具去检查代码，只有在检查都通过时才提交代码，这样就能保证提交到仓库的代码都通过了检查。

如果我们的项目是使用 Git 管理的，则 Git 提供了 Hook 功能做到在提交代码前触发执行脚本。

husky（https://github.com/typicode/husky）可以方便、快速地为项目接入 Git Hook，执行 `npm i -D husky` 安装 husky 时，husky 会通过 Npm Script Hook 自动配置好 Git Hook，我们需要做的只是在 `package.json` 文件中定义几个脚本，方法如下：

```
{
  "scripts": {
    // 在执行 git commit 前会执行的脚本
    "precommit": "npm run lint",
    // 在执行 git push 前会执行的脚本
    "prepush": "lint",
```

```
    // 调用 eslint、stylelint 等工具检查代码
    "lint": "eslint && stylelint"
  }
}
```

我们需要根据自己的情况选择设置 precommit 和 prepush 中的一个，无须对两个都设置。

3.17 通过 Node.js API 启动 Webpack

Webpack 除了提供了可执行的命令行工具，还提供了可在 Node.js 环境中调用的库。通过 Webpack 暴露的 API，可直接在 Node.js 程序中调用 Webpack 执行构建。

通过 API 去调用并执行 Webpack，比直接通过可执行文件启动更灵活，可用在一些特殊场景中，下面讲解如何使用 Webpack 提供的 API。

Webpack 其实是一个 Node.js 应用程序，全部通过 JavaScript 开发完成。在命令行中执行 webpack 命令其实等价于执行 node./node_modules/webpack/bin/webpack.js。

3.17.1 安装和使用 Webpack 模块

在调用 Webpack API 前，需要先安装它：

```
npm i -D webpack
```

安装成功后，可以采用以下代码导入 Webpack 模块：

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

// ES6 语法
import webpack from "webpack";
```

导出的 webpack 其实是一个函数，使用方法如下：

```
webpack({
  // Webpack 配置，和 webpack.config.js 文件一致
```

```
}, (err, stats) => {
  if (err || stats.hasErrors()) {
    // 构建过程出错
  }
  // 成功执行完构建
});
```

如果我们将 Webpack 配置写在 webpack.config.js 文件中，则可以这样使用：

```
// 读取 webpack.config.js 文件中的配置
const config = require('./webpack.config.js');
webpack(config , callback);
```

3.17.2 以监听模式运行

以上使用 Webpack API 的方法只能执行一次构建，无法以监听模式启动 Webpack，为了在使用 API 时以监听模式启动，则需要获取 Compiler 实例，方法如下：

```
// 如果不传 callback 回调函数作为第 2 个参数，就会返回一个 Compiler 实例，用于控制启动，而不是像上面那样立即启动
const compiler = webpack(config);

// 调用 compiler.watch 并以监听模式启动，返回的 watching 用于关闭监听
const watching = compiler.watch({
  // watchOptions
  aggregateTimeout: 300,
},(err, stats)=>{
  // 每次因文件发生变化而重新执行完构建后
});

// 调用 watching.close 关闭监听
watching.close(()=>{
  // 在监听关闭后
});
```

其中的 watchOptions 就是在 2.7 节中介绍过的 Watch 和 WatchOptions。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-17 通过 Node.jsAPI 启动 Webpack.zip。

3.18 使用 Webpack Dev Middleware

在 1.6 节中介绍过的 DevServer 是一个方便开发的小型 HTTP 服务器，DevServer 其实是基于 webpack-dev-middleware（https://github.com/webpack/webpack-dev-middleware）和 Expressjs（https://expressjs.com）实现的，而 webpack-dev-middleware 其实是 Expressjs 的一个中间件。

也就是说，实现 DevServer 基本功能的代码大致如下：

```
const express = require('express');
const webpack = require('webpack');
const webpackMiddleware = require('webpack-dev-middleware');

// 从 webpack.config.js 文件中读取 Webpack 配置
const config = require('./webpack.config.js');
// 实例化一个 Expressjs app
const app = express();

// 用读取到的 Webpack 配置实例化一个 Compiler
const compiler = webpack(config);
// 为 app 注册 webpackMiddleware 中间件
app.use(webpackMiddleware(compiler));
// 启动 HTTP 服务器，服务器监听在 3000 端口
app.listen(3000);
```

从以上代码可以看出，从 webpack-dev-middleware 中导出的 webpackMiddleware 是一个函数，该函数需要接收一个 Compiler 实例。在 3.17 节中曾提到，Webpack API 导出的 webpack 函数会返回一个 Compiler 实例。

webpackMiddleware 函数的返回结果是一个 Expressjs 的中间件，该中间件有以下功能。

- 接收来自 Webpack Compiler 实例输出的文件，但不会将文件输出到硬盘中，而会保存在内存中。
- 在 Expressjs app 上注册路由，拦截 HTTP 收到的请求，根据请求路径响应对应的文件内容。

通过 webpack-dev-middleware 能够将 DevServer 集成到现有的 HTTP 服务器中，让现有的 HTTP 服务器能返回 Webpack 构建出的内容，而不是在开发时启动多个 HTTP 服务器。这特别适用于后端接口服务采用 Node.js 编写的项目。

3.18.1 Webpack Dev Middleware 支持的配置项

在 Node.js 中调用 webpack-dev-middleware 提供的 API 时，还可以向它传入一些配置项，方法如下：

```
// webpackMiddleware 函数的第 2 个参数为配置项
app.use(webpackMiddleware(compiler, {
    // 在 webpack-dev-middleware 支持的所有配置项中
    // 只有 publicPath 属性为必填项，其他都是选填项

    // Webpack 输出资源绑定 HTTP 服务器上的根目录，
    // 和 Webpack 配置中的 publicPath 含义一致
    publicPath: '/assets/',

    // 不输出 info 类型的日志到控制台，只输出 warn 和 error 类型的日志
    noInfo: false,

    // 不输出任何类型的日志到控制台
    quiet: false,

    // 切换到懒惰模式，这意味着不监听文件的变化，只会在有请求时再编译对应的文件，
    // 这适合页面非常多的项目。
    lazy: true,

    // watchOptions
    // 只在非懒惰模式下才有效
    watchOptions: {
        aggregateTimeout: 300,
        poll: true
    },

    // 默认的 URL 路径，默认是'index.html'
    index: 'index.html',
```

```
    // 自定义 HTTP 头
    headers: {'X-Custom-Header': 'yes'},

    // 为特定后缀的文件添加 HTTP mimeTypes，作为文件类型映射表
    mimeTypes: {'text/html': ['phtml']},

    // 统计信息输出样式
    stats: {
      colors: true
    },

    // 自定义输出日志的展示方法
    reporter: null,

    // 开启或关闭服务端渲染
    serverSideRender: false,
  }));
```

3.18.2 Webpack Dev Middleware 与模块热替换

DevServer 提供了一个便捷的功能，可以做到在监听到文件发生变化时自动替换网页中的老模块，以做到实时预览。DevServer 虽然是基于 webpack-dev-middleware 实现的，但 webpack-dev-middleware 并没有实现模块热替换功能，而 DevServer 自己实现了该功能。

为了在使用 webpack-dev-middleware 时也能使用模块热替换功能去提升开发效率，需要额外接入 webpack-hot-middleware（https://github.com/glenjamin/webpack-hot-middleware）。需要做以下修改才能实现模块热替换。

第 1 步，修改 webpack.config.js 文件，加入 HotModuleReplacementPlugin 插件，修改如下：

```
  const HotModuleReplacementPlugin =
require('webpack/lib/HotModuleReplacementPlugin');

  module.exports = {
    entry: [
```

```
    // 为了支持模块热替换，注入代理客户端
    'webpack-hot-middleware/client',
    // JavaScript 执行入口文件
    './src/main.js'
  ],
  output: {
    // 将所有依赖的模块合并输出到一个 bundle.js 文件中
    filename: 'bundle.js',
  },
  plugins: [
    // 为了支持模块热替换，生成.hot-update.json 文件
    new HotModuleReplacementPlugin(),
  ],
  devtool: 'source-map',
};
```

该修改相当于完成了在 4.6 节中提到的 webpack-dev-server --hot 的工作。

第 2 步，修改 HTTP 服务器代码的 server.js 文件，接入 webpack-hot-middleware 中间件，修改如下：

```
const express = require('express');
const webpack = require('webpack');
const webpackMiddleware = require('webpack-dev-middleware');

// 从 webpack.config.js 文件中读取 Webpack 配置
const config = require('./webpack.config.js');
// 实例化一个 Expressjs app
const app = express();

// 用读取到的 Webpack 配置实例化一个 Compiler
const compiler = webpack(config);
// 为 app 注册 webpackMiddleware 中间件
app.use(webpackMiddleware(compiler));
// 为了支持模块热替换，响应用于替换老模块的资源
app.use(require('webpack-hot-middleware')(compiler));
// 将项目根目录作为静态资源目录，用于服务 HTML 文件
app.use(express.static('.'));
// 启动 HTTP 服务器，服务器监听在 3000 端口
app.listen(3000, () => {
```

```
  console.info('成功监听在 3000');
});
```

第 3 步，修改执行入口文件 main.js，加入替换逻辑，在文件末尾加入以下代码：

```
if (module.hot) {
  module.hot.accept();
}
```

第 4 步，安装新引入的依赖：

```
npm i -D webpack-dev-middleware webpack-hot-middleware express
```

安装成功后，通过 node ./server.js 就能启动一个类似于 DevServer 的支持模块热替换的自定义 HTTP 服务了。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-18 使用 WebpackDevMiddleware. zip。

3.19 加载图片

在网页中不可避免地会依赖图片资源，例如 PNG、JPG、GIF。下面讲解如何用 Webpack 加载图片资源。

3.19.1 使用 file-loader

file-loader（https://github.com/webpack-contrib/file-loader）可以将 JavaScript 和 CSS 中导入图片的语句替换成正确的地址，同时将文件输出到对应的位置。

例如，CSS 源码是这样写的：

```
#app {
  background-image: url(./imgs/a.png);
}
```

被 file-loader 转换后输出的 CSS 会变成下面这样：

```
#app {
  background-image: url(5556e1251a78c5afda9ee7dd06ad109b.png);
}
```

并且在输出目录 dist 中多出./imgs/a.png 对应的图片文件 5556e1251a78c5afda9ee7dd06ad109b.png，输出的文件名是根据文件的内容计算出的 Hash 值。

同理，在 JavaScript 中导入图片的源码如下：

```
import imgB from './imgs/b.png';

window.document.getElementById('app').innerHTML = `
<img src="${imgB}"/>
`;
```

经过 file-loader 处理后输出的 JavaScript 代码如下：

```
module.exports = __webpack_require__.p +
"0bcc1f8d385f78e1271ebfca50668429.png";
```

也就是说，imgB 的值就是图片对应的 URL 地址。

在 Webpack 中使用 file-loader 非常简单，相关配置如下：

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.png$/,
        use: ['file-loader']
      }
    ]
  }
};
```

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-19 加载图片 file-loader.zip。

3.19.2 使用 url-loader

url-loader（https://github.com/webpack-contrib/url-loader）可以将文件的内容经过 base64

编码后注入 JavaScript 或者 CSS 中。

例如，CSS 源码是这样写的：

```
#app {
  background-image: url(./imgs/a.png);
}
```

被 url-loader 转换后输出的 CSS 会变成下面这样：

```
#app {
  background-image: url(...); /* 结尾省略了剩下的 base64 编码后的数据 */
}
```

同理，在 JavaScript 中效果类似。

从上面的例子中可以看出，url-loader 会将根据图片内容计算出的 base64 编码的字符串直接注入代码中。由于一般的图片数据量巨大，会导致 JavaScript、CSS 文件也跟着变大，所以在使用 url-loader 时，一定要注意图片的体积不能太大，不然会导致因 JavaScript、CSS 文件过大而带来的网页加载缓慢问题。

一般利用 url-loader 将网页需要用到的小图片资源注入代码中，以减少加载次数。因为在 HTTP/1 协议中，每加载一个资源都需要建立一次 HTTP 链接，为了一个很小的图片而新建一次 HTTP 连接是不划算的。

url-loader 考虑到了以上问题，并提供了一个方便的选择：`limit`，该选项用于控制在文件的大小小于 limit 时才使用 url-loader，否则使用 `fallback` 选项中配置的 loader。相关的 Webpack 配置如下：

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.png$/,
        use: [{
          loader: 'url-loader',
          options: {
            // 30KB 以下的文件采用 url-loader
            limit: 1024 * 30,
```

```
            // 否则采用 file-loader，默认值是 file-loader
            fallback: 'file-loader',
          }
        }]
      }
    ]
  },
};
```

除此之外，还可以做以下优化。

- 通过 imagemin-webpack-plugin（https://www.npmjs.com/package/imagemin-webpack-plugin）压缩图片。
- 通过 webpack-spritesmith（https://www.npmjs.com/package/webpack-spritesmith）插件制作雪碧图。

以上加载图片的方法同样适用于其他二进制类型的资源，例如 PDF、SWF 等。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-19 加载图片 url-loader.zip。

3.20 加载 SVG

SVG 作为矢量图的一种标准格式，已经得到了各大浏览器的支持，也成为 Web 中矢量图的代名词。在网页中采用 SVG 代替位图有如下好处。

- SVG 比位图更清晰，在任意缩放的情况下都不会破坏图形的清晰度，能方便地解决高分辨率屏幕下图像显示不清楚的问题。
- 在图形线条比较简单的情况下，SVG 文件的大小要小于位图，在扁平化 UI 流行的今天，在大多数情况下 SVG 会更小。
- 图形相同的 SVG 比对应的高清图有更好的渲染性能。
- SVG 采用和 HTML 一致的 XML 语法描述，灵活性很高。

画图工具能导出一个个 .svg 文件，SVG 的导入方法和图片类似，既可以像下面这样在

CSS 中直接使用：

```
body {
  background-image: url(./svgs/activity.svg);
}
```

也可以在 HTML 中使用：

```
<img src="./svgs/activity.svg"/>
```

也就是说，可以直接将 SVG 文件当作一张图片来使用，方法和使用图片时完全一样。所以在 3.19 节中介绍的两种方法——使用 file-loader 和使用 url-loader，对 SVG 来说同样有效，只需将 Loader test 配置中的文件后缀改成 .svg，代码如下：

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.svg/,
        use: ['file-loader']
      }
    ]
  },
};
```

由于 SVG 是文本格式的文件，所以除了有以上两种方法，还有其他方法，下面一一说明。

3.20.1 使用 raw-loader

raw-loader（https://github.com/webpack-contrib/raw-loader）可以将文本文件的内容读取出来，注入 JavaScript 或 CSS 中。

例如，在 JavaScript 中这样写：

```
import svgContent from './svgs/alert.svg';
```

经过 raw-loader 处理后输出的代码如下：

```
module.exports = "<svg xmlns=\"http://www.w3.org/2000/svg\"... </svg>"
// 末尾省略了 SVG 的内容
```

也就是说，svgContent 的内容等同于字符串形式的 SVG，由于 SVG 本身就是 HTML 元素，所以在获取 SVG 的内容后，可以直接通过以下代码将 SVG 插入网页中：

```
window.document.getElementById('app').innerHTML = svgContent;
```

使用 raw-loader 时的相关 Webpack 配置如下：

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.svg$/,
        use: ['raw-loader']
      }
    ]
  }
};
```

由于 raw-loader 会直接返回 SVG 的文本内容，并且无法通过 CSS 展示 SVG 的文本内容，因此采用本方法后无法在 CSS 中导入 SVG。也就是说，在 CSS 中不可以出现 background-image:url(./svgs/activity.svg) 这样的代码，因为 background-image: url(<svg>...</svg>) 是不合法的。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-20 加载 SVG-raw-loader.zip。

3.20.2 使用 svg-inline-loader

svg-inline-loader（https://github.com/webpack-contrib/svg-inline-loader）和上面提到的 raw-loader 非常相似，不同之处在于 svg-inline-loader 会分析 SVG 的内容，去除其中不必要的部分代码，以减小 SVG 的文件大小。

在使用画图工具如 Adobe Illustrator、Sketch 制作 SVG 后，在导出时这些工具会生成对网页运行来说不必要的代码。举个例子，以下是 Sketch 导出的 SVG 的代码：

```
<svg class="icon" verison="1.1" xmlns="http://www.w3.org/2000/svg"
width="24" height="24" viewBox="0 0 24 24"
      stroke="#000">
```

```
  <circle cx="12" cy="12" r="10"/>
</svg>
```

被 svg-inline-loader 处理后会精简如下：

```
<svg viewBox="0 0 24 24" stroke="#000"><circle cx="12" cy="12"
r="10"/></svg>
```

也就是说，svg-inline-loader 增加了对 SVG 的压缩功能。

使用 svg-inline-loader 时相关的 Webpack 配置如下：

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.svg$/,
        use: ['svg-inline-loader']
      }
    ]
  }
};
```

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/3-20 加载 SVG-svg-inline-loader.zip。

3.21 加载 Source Map

在开发过程中会经常使用新语言开发项目，最后会将源码转换成能在浏览器中直接运行的 JavaScript 代码。这样做虽能提升开发效率，但在调试代码的过程中我们会发现，所生成代码的可读性非常差，这为代码调试带来了不便。

Webpack 支持为转换生成的代码输出对应的 Source Map 文件，以方便在浏览器中通过源码调试。控制 Source Map 输出的 Webpack 配置项是 devtool，它有很多选项，如表 3-1 所示。

表 3-1 devtool 的选项列表

devtool	含 义
空	不生成 Source Map
eval	每个 module 会封装到 eval 里包裹起来执行，并且会在每个 eval 语句的末尾追加注释 //# sourceURL=webpack:///./main.js
source-map	会额外生成一个单独的 Source Map 文件，并且会在 JavaScript 文件的末尾追加//# sourceMappingURL=bundle.js.map
hidden-source-map	和 source-map 类似，但不会在 JavaScript 文件的末尾追加//# sourceMappingURL= bundle.js.map
inline-source-map	和 source-map 类似，但不会额外生成一个单独的 Source Map 文件，而是将 Source Map 转换成 base64 编码内嵌到 JavaScript 中
eval-source-map	和 eval 类似，但会将每个模块的 Source Map 转换成 base64 编码内嵌到 eval 语句的末尾，例如//# sourceMappingURL=data:application/json;charset=utf-8; base64,eyJ2ZXJzaW...
cheap-source-map	和 source-map 类似，但生成的 Source Map 文件中没有列信息，因此生成速度更快
cheap-module-source-map	和 cheap-source-map 类似，但会包含 Loader 生成的 Source Map

其实以上表格只列举了 devtool 可能取值的一部分，它的取值可以由 source-map、eval、inline、hidden、cheap、module 这 6 个关键字随意组合而成。这 6 个关键字中的每一个都代表一种特性，它们的含义分别如下。

- eval：用 eval 语句包裹需要安装的模块。
- source-map：生成独立的 Source Map 文件。
- hidden：不在 JavaScript 文件中指出 Source Map 文件的所在，这样浏览器就不会自动加载 Source Map。
- inline：将生成的 Source Map 转换成 base64 格式内嵌在 JavaScript 文件中。
- cheap：在生成的 Source Map 中不会包含列信息，这样计算量更小，输出的 Source Map 文件更小；同时 Loader 输出的 Source Map 不会被采用。
- module：来自 Loader 的 Source Map 被简单处理成每行一个模块。

3.21.1 该如何选择

Devtool 配置项提供的这么多选项看似简单，却让很多人弄不明白它们之间的差别和应用场景。如果不关心细节和性能，只是想在不出任何差错的情况下调试源码，则可以直接设置成 source-map，但这样会造成以下两个问题。

- 在 source-map 模式下会输出质量最高且最详细的 Source Map，这会造成构建速度缓慢，特别是在开发过程中需要频繁修改时会增加等待时间。
- 在 source-map 模式下会将 Source Map 暴露，若构建发布到线上的代码的 Source Map 暴露，就等同于源码被泄露。

为了解决以上两个问题，可以这样做，如下所述。

- 在开发环境下将 devtool 设置成 cheap-module-eval-source-map，因为生成这种 Source Map 的速度最快，能加速构建。由于在开发环境下不会做代码压缩，所以在 Source Map 中即使没有列信息，也不会影响断点调试。
- 在生产环境下将 devtool 设置成 hidden-source-map，意思是生成最详细的 Source Map，但不会将 Source Map 暴露出去。由于在生产环境下会做代码压缩，一个 JavaScript 文件只有一行，所以需要列信息。

在生产环境下通常不会将 Source Map 上传到 HTTP 服务器让用户获取，而是上传到 JavaScript 错误收集系统，在错误收集系统上根据 Source Map 和收集到的 JavaScript 运行错误堆栈，计算出错误所在源码的位置。

不要在生产环境下使用 inline 模式的 Source Map，因为这会使 JavaScript 文件变得很大，而且会泄露源码。

3.21.2 加载现有的 Source Map

某些从 Npm 安装的第三方模块是采用 ES6 或者 TypeScript 编写的，它们在发布时会同时带上编译出来的 JavaScript 文件和对应的 Source Map 文件，以方便我们在使用它们出问题时进行调试。

在默认情况下，Webpack 是不会加载这些附加的 Source Map 文件的，Webpack 只会在转换的过程中生成 Source Map。为了让 Webpack 加载这些附加的 Source Map 文件，我们需要安装 source-map-loader（https://github.com/webpack-contrib/source-map-loader）。使用方法如下：

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        // 只加载我们关心的目录下的 Source Map，以提升构建速度
        include: [path.resolve(root, 'node_modules/some-components/')],
        use: ['source-map-loader'],
        // 要将 source-map-loader 的执行顺序放到最前面，如果在 source-map-loader 之前有 Loader 转换了该 JavaScript 文件，就会导致 Source Map 映射错误
        enforce: 'pre'
      }
    ]
  }
};
```

由于 source-map-loader 在加载 Source Map 时计算量很大，因此要避免让该 Loader 处理过多的文件，不然会导致构建速度缓慢。我们通常会采用 include 去命中自己关心的文件。

再安装新引入的依赖：

```
npm i -D source-map-loader
```

重启 Webpack 后，就能在浏览器中调试 node_modules/some-components/目录下的源码了。

3.22 实战总结

在实际应用中，我们会遇到各种各样的需求，虽然在前面的小节中已经给出了应对大部

分场景和需求的解决方案，但还是很难覆盖所有的可能性。所以我们需要有能力去分析遇到的问题，然后寻找对应的解决方案。我们可以按照以下思路分析和解决问题。

- 对所面临的问题本身要有所了解。例如在用 Webpack 构建 React 应用时，我们需要先掌握 React 的基础知识。
- 找出现实和目标之间的差异。例如在 React 应用的源码中用到了 JSX 语法和 ES6 语法，需要将源码转换成 ES5。
- 找出从现实到目标的可能路径。例如将新语法转换成 ES5 时，可以使用 Babel 转换源码。
- 寻找社区中现成的针对可能路径的 Webpack 集成方案。例如社区中已经有了 babel-loader。
- 如果找不到现成的方案，则说明自己的需求非常特别，这时就需要编写自己的 Loader 或者 Plugin 了。在第 5 章中会介绍如何编写它们。

在解决问题的过程中我们要拥有以下两个重要的能力。

- 通过一个知识尽可能多地联想到与其关联的知识，这有利于打通自己的知识体系，从经验中更快地得出答案。
- 善于使用搜索引擎去寻找自己所面临的问题，这有利于借助他人的经验更快地得出答案，而不是自己重新探索。

最重要的是需要多实战，自己去解决问题，这有利于加深理解，而不是只看不做。

第4章 优 化

经过前面的学习，我们已经能用 Webpack 解决常见的问题了，但还有很多关于优化的知识点需要我们了解。优化可以分为优化开发体验和优化输出质量两部分，本章进一步深入讲解如何优化 Webpack 构建。

（1）优化开发体验

优化开发体验的目的是提升开发效率，如下所述。

- 优化构建速度，如 4.1～4.4 节所述。项目庞大时构建的耗时可能会变得很长，每次等待构建的耗时加起来也会是个大数目。
- 优化使用体验，如 4.5～4.6 节所述。通过自动化手段完成一些重复的工作，让我们专注于解决问题本身。

（2）优化输出质量

优化输出质量的目的是为用户呈现体验更好的网页，例如减少首屏加载时间、提升性能流畅度等。这至关重要，因为在互联网行业竞争日益激烈的今天，这可能会关系到我们的产品的生死。

优化输出质量的本质是优化构建输出的要发布到线上的代码，分为以下几点。

- 减少用户能感知到的加载时间，也就是首屏加载时间。如 4.7～4.12 节所述。

- 提升流畅度，也就是提升代码性能。如 4.13 ~ 4.14 节所述。

优化的关键是找出问题所在，这样才能一针见血，4.15 节讲解如何利用工具快速找出问题的所在。

4.16 节会对以上优化方法做一个总结。

4.1 缩小文件的搜索范围

Webpack 在启动后会从配置的 Entry 出发，解析出文件中的导入语句，再递归解析。在遇到导入语句时，Webpack 会做以下两件事。

- 根据导入语句去寻找对应的要导入的文件。例如 `require('react')` 导入语句对应的文件是 `./node_modules/react/react.js`，`require('./util')` 对应的文件是 `./util.js`。
- 根据找到的要导入的文件的后缀，使用配置中的 Loader 去处理文件。例如使用 ES6 开发的 JavaScript 文件需要使用 babel-loader 处理。

虽然以上两件事对于处理一个文件来说非常快，但是在项目大了以后文件量会变得非常大，这时构建速度慢的问题就会暴露出来。虽然以上两件事情无法避免，但需要尽量减少以上两件事情的发生，以提高速度。

接下来一一介绍可以优化它们的途径。

4.1.1 优化 Loader 配置

由于 Loader 对文件的转换操作很耗时，所以需要让尽可能少的文件被 Loader 处理。

在 2.3 节中介绍过在使用 Loader 时，可以通过 `test`、`include`、`exclude` 三个配置项来命中 Loader 要应用规则的文件。为了尽可能少地让文件被 Loader 处理，可以通过 `include` 去命中只有哪些文件需要被处理。

以采用 ES6 的项目为例，在配置 babel-loader 时可以这样：

```
module.exports = {
  module: {
    rules: [
      {
        // 如果项目源码中只有 js 文件，就不要写成/\.jsx?$/，以提升正则表达式的性能
        test: /\.js$/,
        // babel-loader 支持缓存转换出的结果，通过 cacheDirectory 选项开启
        use: ['babel-loader?cacheDirectory'],
        // 只对项目根目录下的 src 目录中的文件采用 babel-loader
        include: path.resolve(__dirname, 'src'),
      },
    ]
  },
};
```

我们可以适当调整项目的目录结构，以方便在配置 Loader 时通过 include 缩小命中的范围。

4.1.2 优化 resolve.modules 配置

在 2.4 节中介绍过 resolve.modules，它用于配置 Webpack 去哪些目录下寻找第三方模块。

resolve.modules 的默认值是['node_modules']，含义是先去当前目录的./node_modules 目录下去找我们想找的模块，如果没找到，就去上一级目录../node_modules 中找，再没有就去../../node_modules 中找，以此类推，这和 Node.js 的模块寻找机制很相似。

当安装的第三方模块都放在项目根目录的./node_modules 目录下时，就没有必要按照默认的方式去一层层地寻找，可以指明存放第三方模块的绝对路径，以减少寻找，配置如下：

```
module.exports = {
  resolve: {
```

```
    // 使用绝对路径指明第三方模块存放的位置，以减少搜索步骤
    // 其中，__dirname 表示当前工作目录，也就是项目根目录
    modules: [path.resolve(__dirname, 'node_modules')]
  },
};
```

4.1.3 优化 resolve.mainFields 配置

在 2.4 节中介绍过 resolve.mainFields，它用于配置第三方模块使用哪个入口文件。

在安装的第三方模块中都会有一个 package.json 文件，用于描述这个模块的属性，其中的某些字段用于描述入口文件在哪里，resolve.mainFields 用于配置采用哪个字段作为入口文件的描述。

可以存在多个字段描述入口文件的原因是，某些模块可以同时用于多个环境中，针对不同的运行环境需要使用不同的代码。以 isomorphic-fetch（https://github.com/matthew-andrews/isomorphic-fetch）为例，它是 fetch API（https://developer.mozilla.org/zh-CN/docs/Web/API/Fetch_API）的一个实现，但可同时用于浏览器和 Node.js 环境。在它的 package.json 中就有两个入口文件描述字段：

```
{
  "browser": "fetch-npm-browserify.js",
  "main": "fetch-npm-node.js"
}
```

isomorphic-fetch 在不同的运行环境下使用不同的代码，是因为 fetch API 的实现机制不一样，在浏览器中通过原生的 fetch 或者 XMLHttpRequest 实现，在 Node.js 中通过 http 模块实现。

resolve.mainFields 的默认值和当前的 target 配置有关系，对应的关系如下。

- 当 target 为 web 或者 webworker 时，值是 ["browser", "module", "main"]。
- 当 target 为其他情况时，值是 ["module", "main"]。

以target等于web为例，Webpack会先采用第三方模块中的browser字段去寻找模块的入口文件，如果不存在，就采用module字段，以此类推。

为了减少搜索步骤，在明确第三方模块的入口文件描述字段时，我们可以将它设置得尽量少。由于大多数第三方模块都采用main字段去描述入口文件的位置，所以可以这样配置Webpack：

```
module.exports = {
  resolve: {
    // 只采用main字段作为入口文件的描述字段，以减少搜索步骤
    mainFields: ['main'],
  },
};
```

使用本方法优化时，需要考虑到所有运行时依赖的第三方模块的入口文件的描述字段，就算只有一个模块出错，也可能会造成构建出的代码无法正常运行。

4.1.4 优化resolve.alias配置

在2.4节中介绍过，resolve.alias配置项通过别名来将原导入路径映射成一个新的导入路径。

在实战项目中经常会依赖一些庞大的第三方模块，以React库为例，安装到node_modules目录下的React库的目录结构如下：

```
├── dist
│   ├── react.js
│   └── react.min.js
├── lib
│   ... 还有几十个文件被忽略
│   ├── LinkedStateMixin.js
│   ├── createClass.js
│   └── React.js
├── package.json
└── react.js
```

可以看到在发布出去的React库中包含两套代码。

- 一套是采用 CommonJS 规范的模块化代码，这些文件都放在 lib 目录下，以 package.json 中指定的入口文件 react.js 为模块的入口。
- 一套是将 React 的所有相关代码打包好的完整代码放到一个单独的文件中，这些代码没有采用模块化，可以直接执行。其中 dist/react.js 用于开发环境，里面包含检查和警告的代码。dist/react.min.js 用于线上环境，被最小化了。

在默认情况下，Webpack 会从入口文件./node_modules/react/react.js 开始递归解析和处理依赖的几十个文件，这会是一个很耗时的操作。通过配置 resolve.alias，可以让 Webpack 在处理 React 库时，直接使用单独、完整的 react.min.js 文件，从而跳过耗时的递归解析操作。

相关的 Webpack 配置如下：

```
module.exports = {
  resolve: {
    // 使用 alias 将导入 react 的语句换成直接使用单独、完整的 react.min.js 文件，
    // 减少耗时的递归解析操作
    alias: {
      'react': path.resolve(__dirname, './node_modules/react/dist/
react.min.js'),
    }
  },
};
```

除了 React 库，大多数库被发布到 Npm 仓库中时都会包含打包好的完整文件，对于这些库，也可以对它们配置 alias。

但是，对某些库使用本优化方法后，会影响到后面要讲的使用 Tree-Sharking 去除无效代码的优化，因为打包好的完整文件中有部分代码在我们的项目中可能永远用不上。一般对整体性比较强的库采用本方法优化，因为完整文件中的代码是一个整体，每一行都是不可或缺的。但是对于一些工具类的库如 lodash（https://github.com/lodash/lodash），我们的项目中可能只用到了其中几个工具函数，就不能使用本方法去优化了，因为这会导致在我们的输出代码中包含很多永远不会被执行的代码。

4.1.5 优化 resolve.extensions 配置

在导入语句没带文件后缀时，Webpack 会在自动带上后缀后去尝试询问文件是否存在。在 2.4 节中介绍过，resolve.extensions 用于配置在尝试过程中用到的后缀列表，默认是：

```
extensions: ['.js', '.json']
```

也就是说，当遇到 require('./data')这样的导入语句时，Webpack 会先去寻找./data.js 文件，如果该文件不存在，就去寻找./data.json 文件，如果还是找不到就报错。

如果这个列表越长，或者正确的后缀越往后，就会造成尝试的次数越多，所以 resolve.extensions 的配置也会影响到构建的性能。在配置 resolve.extensions 时需要遵守以下几点，以做到尽可能地优化构建性能。

- 后缀尝试列表要尽可能小，不要将项目中不可能存在的情况写到后缀尝试列表中。
- 频率出现最高的文件后缀要优先放在最前面，以做到尽快退出寻找过程。
- 在源码中写导入语句时，要尽可能带上后缀，从而可以避免寻找过程。例如在确定的情况下将 require('./data')写成 require('./data.json')。

相关的 Webpack 配置如下：

```
module.exports = {
  resolve: {
    // 尽可能减少后缀尝试的可能性
    extensions: ['js'],
  },
};
```

4.1.6 优化 module.noParse 配置

在 2.3 节中介绍过，module.noParse 配置项可以让 Webpack 忽略对部分没采用模块化的文件的递归解析处理，这样做的好处是能提高构建性能。原因是一些库如 jQuery、

ChartJS 庞大又没有采用模块化标准，让 Webpack 解析这些文件既耗时又没有意义。

在前面讲解优化 resolve.alias 配置时讲到，单独、完整的 react.min.js 文件没有采用模块化，让我们通过配置 module.noParse 忽略对 react.min.js 文件的递归解析处理，相关的 Webpack 配置如下：

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

module.exports = {
  module: {
    // 单独、完整的`react.min.js`文件没有采用模块化，忽略对`react.min.js`文件的递归解析处理
    noParse: [/react\.min\.js$/],
  },
};
```

注意，被忽略掉的文件里不应该包含 import、require、define 等模块化语句，不然会导致在构建出的代码中包含无法在浏览器环境下执行的模块化语句。

以上就是所有和缩小文件搜索范围相关的构建性能优化方面的内容了，在根据自己项目的需要按照以上方法改造后，构建速度一定会有所提升。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-1 缩小文件搜索范围.zip。

4.2 使用 DllPlugin

4.2.1 认识 DLL

在介绍 DllPlugin（https://webpack.js.org/plugins/dll-plugin/）前先为大家介绍一下 DLL。用过 Windows 系统的人应该会经常看到以 .dll 为后缀的文件，这些文件叫作动态链接库，在一个动态链接库中可以包含为其他模块调用的函数和数据。

要给 Web 项目构建接入动态链接库的思想，需要完成以下事情。

- 将网页依赖的基础模块抽离出来，打包到一个个单独的动态链接库中。在一个动态链接库中可以包含多个模块。
- 当需要导入的模块存在于某个动态链接库中时，这个模块不能被再次打包，而是去动态链接库中获取。
- 页面依赖的所有动态链接库都需要被加载。

为什么为 Web 项目构建接入动态链接库的思想后，会大大提升构建速度呢？原因在于，包含大量复用模块的动态链接库只需被编译一次，在之后的构建过程中被动态链接库包含的模块将不会重新编译，而是直接使用动态链接库中的代码。由于动态链接库中大多数包含的是常用的第三方模块，例如 react、react-dom，所以只要不升级这些模块的版本，动态链接库就不用重新编译。

4.2.2 接入 Webpack

Webpack 已经内置了对动态链接库的支持，需要通过以下两个内置的插件接入。

- DllPlugin 插件：用于打包出一个个单独的动态链接库文件。
- DllReferencePlugin 插件：用于在主要的配置文件中引入 DllPlugin 插件打包好的动态链接库文件。

下面以基本的 React 项目为例，为其接入 DllPlugin。在开始前先来看看最终构建出的目录结构：

```
├── main.js
├── polyfill.dll.js
├── polyfill.manifest.json
├── react.dll.js
└── react.manifest.json
```

其中包含两个动态链接库文件，如下所述。

- `polyfill.dll.js`：里面包含项目所有依赖的 polyfill，例如 Promise、fetch 等 API。
- `react.dll.js`：里面包含 React 的基础运行环境，即 react 和 react-dom 模块。

以 react.dll.js 文件为例，其文件内容大致如下：

```
var _dll_react = (function(modules) {
  // ... 此处省略 webpackBootstrap 函数的代码
}([
  function(module, exports, __webpack_require__) {
    // ID 为 0 的模块对应的代码
  },
  function(module, exports, __webpack_require__) {
    // ID 为 1 的模块对应的代码
  },
  // ... 此处省略剩下的模块对应的代码
]));
```

可见，一个动态链接库文件中包含了大量模块的代码，这些模块被存放在一个数组里，用数组的索引号作为 ID。并且通过 _dll_react 变量将自己暴露在全局中，即可以通过 window._dll_react 访问到其中包含的模块。

其中，polyfill.manifest.json 和 react.manifest.json 文件也是由 DllPlugin 生成的，用于描述在动态链接库文件中包含哪些模块，以 react.manifest.json 文件为例，其文件的内容大致如下：

```
{
  // 描述该动态链接库文件暴露在全局中的变量名称
  "name": "_dll_react",
  "content": {
    "./node_modules/process/browser.js": {
      "id": 0,
      "meta": {}
    },
    // ... 此处省略部分模块
    "./node_modules/react-dom/lib/ReactBrowserEventEmitter.js": {
      "id": 42,
      "meta": {}
    },
    "./node_modules/react/lib/lowPriorityWarning.js": {
      "id": 47,
      "meta": {}
    },
    // ... 此处省略部分模块
```

```
    "./node_modules/react-dom/lib/SyntheticTouchEvent.js": {
      "id": 210,
      "meta": {}
    },
    "./node_modules/react-dom/lib/SyntheticTransitionEvent.js": {
      "id": 211,
      "meta": {}
    },
  }
}
```

可见 manifest.json 文件清楚地描述了与其对应的 dll.js 文件中包含哪些模块，以及每个模块的路径和 ID。

main.js 文件是被编译出来的执行入口文件，在遇到其依赖的模块在 dll.js 文件中时，会直接通过 dll.js 文件暴露的全局变量获取打包在 dll.js 文件中的模块，所以在 index.html 文件中需要将依赖的两个 dll.js 文件加载进去。index.html 的内容如下：

```
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
<div id="app"></div>
<!--导入依赖的动态链接库文件-->
<script src="./dist/polyfill.dll.js"></script>
<script src="./dist/react.dll.js"></script>
<!--导入执行入口文件-->
<script src="./dist/main.js"></script>
</body>
</html>
```

以上就是所有接入 DllPlugin 后最终编译出来的代码，接下来讲解如何实现。

1. 构建出动态链接库文件

构建输出的以下 4 个文件：

```
├── polyfill.dll.js
├── polyfill.manifest.json
```

```
├── react.dll.js
└── react.manifest.json
```

和以下这个文件：

```
├── main.js
```

是由两份不同的构建分别输出的。

动态链接库文件相关的文件需要由一份独立的构建输出，用于为主构建使用。新建一个 Webpack 配置文件 webpack_dll.config.js 专门用于构建它们，文件的内容如下：

```
const path = require('path');
const DllPlugin = require('webpack/lib/DllPlugin');

module.exports = {
  // JavaScript 执行入口文件
  entry: {
    // 将 React 相关的模块放到一个单独的动态链接库中
    react: ['react', 'react-dom'],
    // 将项目需要所有的 polyfill 放到一个单独的动态链接库中
    polyfill: ['core-js/fn/object/assign', 'core-js/fn/promise',
'whatwg-fetch'],
  },
  output: {
    // 输出的动态链接库的文件名称，[name]代表当前动态链接库的名称，
    // 也就是 entry 中配置的 react 和 polyfill
    filename: '[name].dll.js',
    // 将输出的文件都放到 dist 目录下
    path: path.resolve(__dirname, 'dist'),
    // 存放动态链接库的全局变量名称，例如对于 react 来说就是_dll_react
    // 之所以在前面加上_dll_，是为了防止全局变量冲突
    library: '_dll_[name]',
  },
  plugins: [
    // 接入 DllPlugin
    new DllPlugin({
      // 动态链接库的全局变量名称，需要和 output.library 中的保持一致
      // 该字段的值也就是输出的 manifest.json 文件中 name 字段的值
      // 例如在 react.manifest.json 中就有"name": "_dll_react"
      name: '_dll_[name]',
      // 描述动态链接库的 manifest.json 文件输出时的文件名称
```

```
    path: path.join(__dirname, 'dist', '[name].manifest.json'),
  }),
],
};
```

2. 使用动态链接库文件

构建出的动态链接库文件用于在其他地方使用，在这里用于在执行入口使用。

用于输出 main.js 的主 Webpack 配置文件的内容如下：

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

module.exports = {
  entry: {
    // 定义入口 Chunk
    main: './main.js'
  },
  output: {
    // 输出文件的名称
    filename: '[name].js',
    // 将输出文件都放到 dist 目录下
    path: path.resolve(__dirname, 'dist'),
  },
  module: {
    rules: [
      {
        // 项目源码使用了 ES6 和 JSX 语法，需要使用 babel-loader 转换
        test: /\.js$/,
        use: ['babel-loader'],
        exclude: path.resolve(__dirname, 'node_modules'),
      },
    ]
  },
  plugins: [
    // 告诉 Webpack 使用了哪些动态链接库
    new DllReferencePlugin({
      // 描述 react 动态链接库的文件内容
      manifest: require('./dist/react.manifest.json'),
```

```
    }),
    new DllReferencePlugin({
      // 描述 polyfill 动态链接库的文件内容
      manifest: require('./dist/polyfill.manifest.json'),
    }),
  ],
  devtool: 'source-map'
};
```

注意：在 webpack_dll.config.js 文件中，DllPlugin 中的 name 参数必须和 output.library 中的保持一致。原因在于 DllPlugin 中的 name 参数会影响输出的 manifest.json 文件中 name 字段的值。而在 webpack.config.js 文件中，DllReferencePlugin 会去 manifest.json 文件中读取 name 字段的值，将值的内容作为在从全局变量中获取动态链接库的内容时的全局变量名。

3. 执行构建

在修改好以上两个 Webpack 配置文件后，需要重新执行构建。重新执行构建时要注意的是，需要先将动态链接库相关的文件编译出来，因为主 Webpack 配置文件中定义的 DllReferencePlugin 依赖这些文件。

执行构建时的流程如下。

- 如果动态链接库相关的文件还没有编译出来，就需要先将它们编译出来。方法是执行 webpack --config webpack_dll.config.js 命令。
- 在确保动态链接库存在时，才能正常编译入口执行文件。方法是执行 webpack 命令。这时我们会发现构建速度有了非常大的提升。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-2 使用 DllPlugin.zip。

4.3 使用 HappyPack

由于有大量文件需要解析和处理，所以构建是文件读写和计算密集型的操作，特别是当

文件数量变多后，Webpack 构建慢的问题会显得更为严重。运行在 Node.js 之上的 Webpack 是单线程模型的，也就是说 Webpack 需要一个一个地处理任务，不能同时处理多个任务。

文件读写和计算操作是无法避免的，那能不能让 Webpack 在同一时刻处理多个任务，发挥多核 CPU 电脑的功能，以提升构建速度呢？

HappyPack（https://github.com/amireh/happypack）就能让 Webpack 做到这一点，它将任务分解给多个子进程去并发执行，子进程处理完后再将结果发送给主进程。

由于 JavaScript 是单线程模型，所以要想发挥多核 CPU 的功能，就只能通过多进程实现，而无法通过多线程实现。

4.3.1 使用 HappyPack

对于分解任务和管理线程的事情，HappyPack 都会帮我们做好，我们所需要做的只是接入 HappyPack。接入 HappyPack 的相关代码如下：

```
const path = require('path');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
const HappyPack = require('happypack');

module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        // 将对.js 文件的处理转交给 id 为 babel 的 HappyPack 实例
        use: ['happypack/loader?id=babel'],
        // 排除 node_modules 目录下的文件，node_modules 目录下的文件都采用了 ES5
语法，没必要再通过 Babel 去转换
        exclude: path.resolve(__dirname, 'node_modules'),
      },
      {
        // 将对.css 文件的处理转交给 id 为 css 的 HappyPack 实例
        test: /\.css$/,
        use: ExtractTextPlugin.extract({
          use: ['happypack/loader?id=css'],
```

```
      },
    ],
  },
  plugins: [
    new HhappyPack({
      // 用唯一的标识符 id，来代表当前的 HappyPack 是用来处理一类特定的文件的
      id: 'babel',
      // 如何处理.js 文件，用法和 Loader 配置中的一样
      loaders: ['babel-loader?cacheDirectory'],
      // ... 其他配置项
    }),
    new HhappyPack({
      id: 'css',
      // 如何处理.css 文件，用法和 Loader 配置中的一样
      loaders: ['css-loader'],
    }),
    new ExtractTextPlugin({
      filename: `[name].css`,
    }),
  ],
};
```

在以上代码中有以下两项重要的修改。

- 在 Loader 配置中，对所有文件的处理都交给了 happypack/loader，使用紧跟其后的 querystring ?id=babel 去告诉 happypack/loader 选择哪个 HappyPack 实例处理文件。
- 在 Plugin 配置中新增了两个 HappyPack 实例，分别用于告诉 happypack/loader 如何处理.js 和.css 文件。选项中的 id 属性的值和上面 querystring 中的?id=babel 对应，选项中的 loaders 属性和 Loader 配置中的一样。

在实例化 HappyPack 插件时，除了可以传入 id 和 loaders 两个参数，HappyPack 还支持传入如下参数。

- threads：代表开启几个子进程去处理这一类型的文件，默认是 3 个，必须是整数。
- verbose：是否允许 HappyPack 输出日志，默认是 true。

- threadPool：代表共享进程池，即多个 HappyPack 实例都使用同一个共享进程池中的子进程去处理任务，以防止资源占用过多，相关代码如下：

```
const HappyPack = require('happypack');
// 构造出共享进程池，在进程池中包含 5 个子进程
const happyThreadPool = HappyPack.ThreadPool({ size: 5 });

module.exports = {
  plugins: [
    new HappyPack({
      // 用唯一的标识符 id 来代表当前的 HappyPack 用来处理一类特定的文件
      id: 'babel',
      // 如何处理 .js 文件，用法和 Loader 配置中一样
      loaders: ['babel-loader?cacheDirectory'],
      // 使用共享进程池中的子进程去处理任务
      threadPool: happyThreadPool,
    }),
    new HappyPack({
      id: 'css',
      // 如何处理 .css 文件，用法和 Loader 配置中的一样
      loaders: ['css-loader'],
      // 使用共享进程池中的子进程去处理任务
      threadPool: happyThreadPool,
    }),
    new ExtractTextPlugin({
      filename: `[name].css`,
    }),
  ],
};
```

接入 HappyPack 后，需要为项目安装新的依赖：

```
npm i -D happypack
```

安装成功后重新执行构建，就会看到由 HappyPack 输出的以下日志：

```
Happy[babel]: Version: 4.0.0-beta.5. Threads: 3
Happy[babel]: All set; signaling webpack to proceed.
Happy[css]: Version: 4.0.0-beta.5. Threads: 3
Happy[css]: All set; signaling webpack to proceed.
```

这说明 HappyPack 配置生效了，并且可以得知 HappyPack 分别启动了 3 个子进程去并行处理任务。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-3 使用 HappyPack.zip。

4.3.2 HappyPack 的原理

在整个 Webpack 构建流程中，最耗时的流程可能就是 Loader 对文件的转换操作了，因为要转换的文件数据量巨大，而且这些转换操作都只能一个一个地处理。HappyPack 的核心原理就是将这部分任务分解到多个进程中去并行处理，从而减少总的构建时间。

从前面的使用中可以看出，所有需要通过 Loader 处理的文件都先交给了 `happypack/loader` 去处理，在收集到了这些文件的处理权后，HappyPack 就可以统一分配了。

每通过 `new HappyPack()` 实例化一个 HappyPack，其实就是告诉 HappyPack 核心调度器如何通过一系列 Loader 去转换一类文件，并且可以指定如何为这类转换操作分配子进程。

核心调度器的逻辑代码在主进程中，也就是运行着 Webpack 的进程中，核心调度器会将一个个任务分配给当前空闲的子进程，子进程处理完毕后将结果发送给核心调度器，它们之间的数据交换是通过进程间的通信 API 实现的。

核心调度器收到来自子进程处理完毕的结果后，会通知 Webpack 该文件已处理完毕。

4.4 使用 ParallelUglifyPlugin

在使用 Webpack 构建出用于发布到线上的代码时，都会有压缩代码这一流程。最常见的 JavaScript 代码压缩工具是 UglifyJS（https://github.com/mishoo/UglifyJS2），并且 Webpack 也内置了它。

若用过 UglifyJS，则我们一定会发现能很快通过它构建用于开发环境的代码，但在构建用于线上的代码时会卡在一个时间点迟迟没有反应，其实在这个卡住的时间点正在进行的就

是代码压缩。

由于压缩 JavaScript 代码时，需要先将代码解析成用 Object 抽象表示的 AST 语法树，再去应用各种规则分析和处理 AST，所以导致这个过程的计算量巨大，耗时非常多。

为什么不将在 4.3 节中介绍过的多进程并行处理的思想也引入到代码压缩中呢？Parallel UglifyPlugin（https://github.com/gdborton/webpack-parallel-uglify-plugin）就做了这件事情。当 Webpack 有多个 JavaScript 文件需要输出和压缩时，原本会使用 UglifyJS 去一个一个压缩再输出，但是 ParallelUglifyPlugin 会开启多个子进程，将对多个文件的压缩工作分配给多个子进程去完成，每个子进程其实还是通过 UglifyJS 去压缩代码，但是变成了并行执行。所以 ParallelUglifyPlugin 能更快地完成对多个文件的压缩工作。

ParallelUglifyPlugin 的使用也非常简单，将原来 Webpack 配置文件中内置的 UglifyJsPlugin 去掉后，再替换成 ParallelUglifyPlugin 即可，相关代码如下：

```
const path = require('path');
const DefinePlugin = require('webpack/lib/DefinePlugin');
const ParallelUglifyPlugin = require('webpack-parallel-uglify-plugin');

module.exports = {
  plugins: [
    // 使用ParallelUglifyPlugin并行压缩输出的JavaScript代码
    new ParallelUglifyPlugin({
      // 传递给UglifyJS的参数
      uglifyJS: {
        output: {
          // 最紧凑的输出
          beautify: false,
          // 删除所有注释
          comments: false,
        },
        compress: {
          // 在UglifyJS删除没有用到的代码时不输出警告
          warnings: false,
          // 删除所有的`console`语句，可以兼容IE浏览器
          drop_console: true,
          // 内嵌已定义但是只用到一次的变量
          collapse_vars: true,
```

```
            // 提取出出现多次但是没有定义成变量去引用的静态值
            reduce_vars: true,
        }
    },
  }),
],
};
```

在通过 new ParallelUglifyPlugin()实例化时，支持以下参数。

- test：使用正则去匹配哪些文件需要被 ParallelUglifyPlugin 压缩，默认为 /.js$/，也就是默认压缩所有的.js 文件。
- include：使用正则去命中需要被 ParallelUglifyPlugin 压缩的文件，默认为[]。
- exclude：使用正则去命中不需要被 ParallelUglifyPlugin 压缩的文件，默认为 []。
- cacheDir：缓存压缩后的结果，下次遇到一样的输入时直接从缓存中获取压缩后的结果并返回。cacheDir 用于配置缓存存放的目录路径。默认不会缓存，若想开启缓存，则请设置一个目录路径。
- workerCount：开启几个子进程去并发执行压缩。默认为当前运行的计算机的 CPU 核数减 1。
- sourceMap：是否输出 Source Map，这会导致压缩过程变慢。
- uglifyJS：用于压缩 ES5 代码时的配置，为 Object 类型，被原封不动地传递给 UglifyJS 作为参数。
- uglifyES：用于压缩 ES6 代码时的配置，为 Object 类型，被原封不动地传递给 UglifyES 作为参数。

其中的 test、include、exclude 与配置 Loader 时的思想和用法一样。

UglifyES（https://github.com/mishoo/UglifyJS2/tree/harmony）是 UglifyJS 的变种，专门用于压缩 ES6 代码，它们都出自同一个项目，并且不能同时使用。

UglifyES 一般用于为比较新的 JavaScript 运行环境压缩代码，例如用于 ReactNative 的代码运行在兼容性较好的 JavaScriptCore 引擎中，为了得到更好的性能和尺寸，可采用 UglifyES 压缩。

ParallelUglifyPlugin 同时内置了 UglifyJS 和 UglifyES，也就是说 ParallelUglifyPlugin 支持并行压缩 ES6 代码。

接入 ParallelUglifyPlugin 后，项目需要安装新的依赖：

```
npm i -D webpack-parallel-uglify-plugin
```

安装成功后重新执行构建，会发现速度变快了许多。如果设置 cacheDir 开启缓存，则在之后的构建中速度会更快。

本实例提供项目的完整代码，参见http://webpack.wuhaolin.cn/4-4使用ParallelUglifyPlugin.zip。

4.5 使用自动刷新

在开发阶段，修改源码是不可避免的操作。对于开发网页来说，要想看到修改后的效果，就需要刷新浏览器让其重新运行最新的代码。虽然这相对于开发原生 iOS 和 Android 应用来说要方便很多，因为那需要重新编译这个项目再运行，但我们可以将这个体验优化得更好。借助自动化的手段，可以将这些重复的操作交给代码去帮我们完成，在监听到本地源码文件发生变化时，自动重新构建出可运行的代码后再控制浏览器刷新。

Webpack 将这些功能都内置了，并且提供了多种方案供我们选择。

4.5.1 文件监听

文件监听是在发现源码文件发生变化时，自动重新构建出新的输出文件。

Webpack 官方提供了两大模块，一个是核心的 webpack（https://www.npmjs.com/package/webpack），一个是在 1.6 节中提到的 webpack-dev-server。而文件监听功能是 Webpack 提供的。

在 2.7 节中曾介绍过 Webpack 支持文件监听相关的配置项，代码如下：

```
module.export = {
  // 只有在开启监听模式时，watchOptions才有意义
  // 默认为false，也就是不开启
```

```
  watch: true,
  // 监听模式运行时的参数
  // 在开启监听模式时才有意义
  watchOptions: {
    // 不监听的文件或文件夹，支持正则匹配
    // 默认为空
    ignored: /node_modules/,
    // 监听到变化发生后等300ms再去执行动作，截流，
    // 防止文件更新太快而导致重新编译频率太快。默认为300ms
    aggregateTimeout: 300,
    // 判断文件是否发生变化是通过不停地询问系统指定文件有没有变化实现的
    // 默认每秒询问1000次
    poll: 1000
  }
}
```

让 Webpack 开启监听模式时，有如下两种方式。

- 在配置文件 webpack.config.js 中设置 watch: true。
- 在执行启动 Webpack 的命令时带上--watch 参数，完整的命令是 webpack--watch。

1. 文件监听的工作原理

在 Webpack 中监听一个文件发生变化的原理，是定时获取这个文件的最后编辑时间，每次都存下最新的最后编辑时间，如果发现当前获取的和最后一次保存的最后编辑时间不一致，就认为该文件发生了变化。配置项中的 watchOptions.poll 用于控制定时检查的周期，具体含义是每秒检查多少次。

当发现某个文件发生了变化时，并不会立刻告诉监听者，而是先缓存起来，收集一段时间的变化后，再一次性告诉监听者。配置项中的 watchOptions.aggregateTimeout 用于配置这个等待时间。这样做的目的是，我们在编辑代码的过程中可能会高频地输入文字，导致文件变化的事件高频地发生，如果每次都重新执行构建，就会让构建卡死。

对于多个文件来说，其原理相似，只不过会对列表中的每个文件都定时执行检查。但是怎么确定这个需要监听的文件列表呢？在默认情况下，Webpack 会从配置的 Entry 文件出发，

递归解析出 Entry 文件所依赖的文件，将这些依赖的文件都加入监听列表中。可见，Webpack 这一点还是做得很智能的，而不是粗暴地直接监听项目目录下的所有文件。

由于保存文件的路径和最后的编辑时间需要占用内存，定时检查周期检查需要占用 CPU 及文件 I/O，所以最好减少需要监听的文件数量和降低检查频率。

2. 优化文件监听的性能

在明白文件监听的工作原理后，就可以分析如何优化文件监听的性能了。

在开启监听模式时，默认情况下会监听配置的 Entry 文件和所有 Entry 递归依赖的文件。在这些文件中会有很多存在于 node_modules 下，因为如今的 Web 项目会依赖大量的第三方模块，所以在大多数情况下我们都不可能去编辑 node_modules 下的文件，而是编辑自己建立的源码文件，而一个很大的优化点就是忽略 node_modules 下的文件，不监听它们。相关配置如下：

```
module.export = {
  watchOptions: {
    // 不监听的 node_modules 目录下的文件
    ignored: /node_modules/,
  }
}
```

采用这种方法优化后，Webpack 消耗的内存和 CPU 将会大大减少。

有时我们可能会觉得 node_modules 目录下的第三方模块有 Bug，想修改第三方模块的文件，然后在自己的项目中尝试。如果在这种情况下使用以上优化方法，就需要重启构建以看到最新效果，但这种情况是非常少见的。

除了忽略部分文件的优化，还有如下两种方法。

- watchOptions.aggregateTimeout 的值越大性能越好，因为这能降低重新构建的频率。
- watchOptions.poll 的值越小越好，因为这能降低检查的频率。

但两种优化方法的后果是监听模式的反应和灵敏度降低了。

4.5.2 自动刷新浏览器

监听到文件更新后的下一步是刷新浏览器，webpack 模块负责监听文件，webpack-dev-server 模块则负责刷新浏览器。在使用 webpack-dev-server 模块去启动 webpack 模块时，webpack 模块的监听模式默认会被开启。webpack 模块会在文件发生变化时通知 webpack-dev-server 模块。

1. 自动刷新的原理

控制浏览器刷新有如下三种方法。

- 借助浏览器扩展去通过浏览器提供的接口刷新，WebStorm IDE 的 LiveEdit 功能就是这样实现的。
- 向要开发的网页中注入代理客户端代码，通过代理客户端去刷新整个页面。
- 将要开发的网页装进一个 iframe 中，通过刷新 iframe 去看到最新效果。

DevServer 支持第 2、3 种方法，第 2 种是 DevServer 默认采用的刷新方法。

通过 DevServer 启动构建后，会看到如下日志：

```
> webpack-dev-server

Project is running at http://localhost:8080/
webpack output is served from /
Hash: e4e2f9508ac286037e71
Version: webpack 3.5.5
Time: 1566ms
        Asset     Size  Chunks                    Chunk Names
    bundle.js  1.07 MB       0  [emitted]  [big]  main
bundle.js.map  1.27 MB       0  [emitted]         main
   [115] multi (webpack)-dev-server/client?http://localhost:8080 ./main.js
40 bytes {0} [built]
   [116] (webpack)-dev-server/client?http://localhost:8080 5.83 kB {0}
[built]
   [117] ./node_modules/url/url.js 23.3 kB {0} [built]
   [120] ./node_modules/querystring-es3/index.js 127 bytes {0} [built]
```

```
[123] ./node_modules/strip-ansi/index.js 161 bytes {0} [built]
[125] ./node_modules/loglevel/lib/loglevel.js 6.74 kB {0} [built]
[126] (webpack)-dev-server/client/socket.js 856 bytes {0} [built]
[158] (webpack)-dev-server/client/overlay.js 3.6 kB {0} [built]
[159] ./node_modules/ansi-html/index.js 4.26 kB {0} [built]
[163] (webpack)/hot nonrecursive ^\.\/log$ 170 bytes {0} [built]
[165] (webpack)/hot/emitter.js 77 bytes {0} [built]
[167] ./main.js 2.28 kB {0} [built]
  + 255 hidden modules
```

细心的你会观察到输出的 bundle.js 中包含了以下 7 个模块：

```
[116] (webpack)-dev-server/client?http://localhost:8080 5.83 kB {0}
[built]
[117] ./node_modules/url/url.js 23.3 kB {0} [built]
[120] ./node_modules/querystring-es3/index.js 127 bytes {0} [built]
[123] ./node_modules/strip-ansi/index.js 161 bytes {0} [built]
[125] ./node_modules/loglevel/lib/loglevel.js 6.74 kB {0} [built]
[126] (webpack)-dev-server/client/socket.js 856 bytes {0} [built]
[158] (webpack)-dev-server/client/overlay.js 3.6 kB {0} [built]
```

这 7 个模块就是代理客户端的代码，它们被打包到了要开发的网页代码中。

在浏览器中打开网址 http://localhost:8080/后，在浏览器的开发者工具中会发现由代理客户端向 DevServer 发起的 WebSocket 连接，如图 4-1 所示。

图 4-1 代理客户端向 DevServer 发起的 WebSocket 连接

2. 优化自动刷新的性能

在 2.6 节中曾介绍过 devServer.inline 配置项，它用来控制是否向 Chunk 中注入代理客户端，默认会注入。事实上，在开启 inline 时，DevServer 会向每个输出的 Chunk 中注入代理客户端的代码，当我们的项目需要输出很多 Chunk 时，就会导致构建缓慢。其实要完成自动刷新，一个页面只需要一个代理客户端，DevServer 之所以粗暴地为每个 Chunk 都注入，是因为它不知道某个网页依赖哪几个 Chunk，索性全部都注入一个代理客户端。网页只要依赖了其中任何一个 Chunk，代理客户端就被注入网页中。

这里的优化思路是关闭还不够优雅的 inline 模式，只注入一个代理客户端。为了关闭 inline 模式，在启动 DevServer 时可以通过执行命令 webpack-dev-server --inline false（也可以在配置文件中设置）来完成，这时输出的日志如下：

```
> webpack-dev-server --inline false

Project is running at http://localhost:8080/webpack-dev-server/
webpack output is served from /
Hash: 5a43fc44b5e85f4c2cf1
Version: webpack 3.5.5
Time: 1130ms
     Asset     Size  Chunks                    Chunk Names
   bundle.js  750 kB       0  [emitted]  [big]  main
bundle.js.map  897 kB       0  [emitted]         main
  [81] ./main.js 2.29 kB {0} [built]
    + 169 hidden modules
```

和前面的不同在于：

- 入口网址变成了 http://localhost:8080/webpack-dev-server/；
- bundle.js 中不再包含代理客户端的代码。

在浏览器中打开网址 http://localhost:8080/webpack-dev-server/后，会看到如图 4-2 所示的效果。

图 4-2　通过 iframe 自动刷新

要开发的网页被放进了一个 iframe 中，编辑源码后，iframe 会被自动刷新。同时我们会发现构建的时间从 1566ms 减少到了 1130ms，说明优化生效了。要输出的 Chunk 数量越多，构建性能提升的效果越明显。

在关闭 inline 后，DevServer 会自动提示通过新网址 http://localhost:8080/ webpack-dev-server/去访问，这一点做得很人性化。

如果不想以 iframe 的方式去访问，但同时想让网页保持自动刷新的功能，则需要手动向网页中注入代理客户端的脚本，向 index.html 中插入以下标签：

```
<!--注入 DevServer 提供的代理客户端脚本，这个服务是 DevServer 内置的-->
<script src="http://localhost:8080/webpack-dev-server.js"></script>
```

向网页注入以上脚本后，独立打开的网页就能自动刷新了。但是要注意在发布到线上时删掉这段用于开发环境的代码。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-5 使用自动刷新.zip。

4.6 开启模块热替换

要做到实时预览，除了可以用 4.5 中介绍的刷新整个网页的方法，DevServer 还支持一种叫做模块热替换（Hot Module Replacement）的技术可在不刷新整个网页的情况下做到超灵敏实时预览。原理是在一个源码发生变化时，只需重新编译发生变化的模块，再用新输出的模块替换掉浏览器中对应的老模块。

模块热替换技术的优势如下。

- 实时预览反应更快，等待时间更短。
- 不刷新浏览器时能保留当前网页的运行状态，例如在使用 Redux 管理数据的应用中搭配模块热替换能做到在代码更新时 Redux 中的数据保持不变。

总的来说，模块热替换技术在很大程度上提升了开发效率和体验。

4.6.1 模块热替换的原理

模块热替换的原理和自动刷新的原理类似，都需要向要开发的网页中注入一个代理客户端来连接 DevServer 和网页，不同在于模块热替换的独特的模块替换机制。

DevServer 默认不会开启模块热替换模式，要开启该模式，则只需在启动时带上参数 `--hot`，完整的命令是 `webpack-dev-server --hot`。

除了通过在启动时带上`--hot` 参数，还可以通过接入 Plugin 实现，相关代码如下：

```
const HotModuleReplacementPlugin =
require('webpack/lib/HotModuleReplacementPlugin');

module.exports = {
  entry:{
    // 为每个入口都注入代理客户端
    main:['webpack-dev-server/client?http://localhost:8080/',
'webpack/hot/dev-server','./src/main.js'],
  },
```

```
  plugins: [
    // 该插件的作用就是实现模块热替换，实际上若启动时带上`--hot`参数，就会注入该插件，生成.hot-update.json 文件。
    new HotModuleReplacementPlugin(),
  ],
  devServer:{
    // 告诉 DevServer 要开启模块热替换模式
    hot: true,
  }
};
```

在启动 Webpack 时带上参数--hot，其实就是自动完成以上配置。

启动后的日志如下：

```
> webpack-dev-server --hot

Project is running at http://localhost:8080/
webpack output is served from /
webpack: wait until bundle finished: /
webpack: wait until bundle finished: /bundle.js
Hash: fe62ac6b753c1d98961b
Version: webpack 3.5.5
Time: 3563ms
      Asset     Size  Chunks                    Chunk Names
    bundle.js  1.11 MB       0  [emitted]  [big]  main
bundle.js.map  1.33 MB       0  [emitted]         main
   [50] (webpack)/hot/log.js 1.04 kB {0} [built]
   [118] multi (webpack)-dev-server/client?http://localhost:8080 webpack/hot/dev-server ./main.js 52 bytes {0} [built]
   [119] (webpack)-dev-server/client?http://localhost:8080 5.83 kB {0} [built]
   [120] ./node_modules/url/url.js 23.3 kB {0} [built]
   [126] ./node_modules/strip-ansi/index.js 161 bytes {0} [built]
   [128] ./node_modules/loglevel/lib/loglevel.js 6.74 kB {0} [built]
   [129] (webpack)-dev-server/client/socket.js 856 bytes {0} [built]
   [161] (webpack)-dev-server/client/overlay.js 3.6 kB {0} [built]
   [166] (webpack)/hot nonrecursive ^\.\/log$ 170 bytes {0} [built]
   [168] (webpack)/hot/dev-server.js 1.61 kB {0} [built]
   [169] (webpack)/hot/log-apply-result.js 1.31 kB {0} [built]
   [170] ./main.js 2.35 kB {0} [built]
```

```
    + 262 hidden modules
```

可以看出，bundle.js 代理客户端相关的代码包含 9 个文件：

```
    [119] (webpack)-dev-server/client?http://localhost:8080 5.83 kB {0}
[built]
    [120] ./node_modules/url/url.js 23.3 kB {0} [built]
    [126] ./node_modules/strip-ansi/index.js 161 bytes {0} [built]
    [128] ./node_modules/loglevel/lib/loglevel.js 6.74 kB {0} [built]
    [129] (webpack)-dev-server/client/socket.js 856 bytes {0} [built]
    [161] (webpack)-dev-server/client/overlay.js 3.6 kB {0} [built]
    [166] (webpack)/hot nonrecursive ^\.\/log$ 170 bytes {0} [built]
    [168] (webpack)/hot/dev-server.js 1.61 kB {0} [built]
    [169] (webpack)/hot/log-apply-result.js 1.31 kB {0} [built]
```

与自动刷新的代理客户端相比，最后多出了三个用于模块热替换的文件，也就是说代理客户端更大了。

修改源码 main.css 文件后，重新输出了如下日志：

```
webpack: Compiling...
Hash: 18f81c959118f6230623
Version: webpack 3.5.5
Time: 551ms
                    Asset       Size  Chunks             Chunk Names
                bundle.js    1.11 MB       0  [emitted]  [big]  main
  0.ea11a51f97f2b52bca7d.hot-update.js  353 bytes  0  [emitted]  main
  ea11a51f97f2b52bca7d.hot-update.json   43 bytes     [emitted]
                bundle.js.map    1.33 MB       0  [emitted]        main
0.ea11a51f97f2b52bca7d.hot-update.js.map  577 bytes  0  [emitted]  main
   [68] ./node_modules/css-loader!./main.css 217 bytes {0} [built]
  [166] (webpack)/hot nonrecursive ^\.\/log$ 170 bytes {0} [built]
    + 275 hidden modules
webpack: Compiled successfully.
```

DevServer 重新生成了一个用于替换老模块的补丁文件 0.ea11a51f97f2b52bca7d.hot-update.js，同时在浏览器开发工具中也能看到请求这个补丁的抓包，如图 4-3 所示。

图 4-3 模块热替换模式下的补丁

可见补丁中包含了 main.css 文件新编译出来的 CSS 代码，网页中的样式也立刻变成了源码中描述的那样。

但在修改 main.js 文件时，我们会发现模块热替换没有生效，而是整个页面被刷新了，为什么修改 main.js 文件时会有这样的效果呢？

为了让使用者在使用模块热替换功能时能灵活地控制老模块被替换时的逻辑，Webpack 允许在源码中定义一些代码去做相应的处理。

将 main.js 文件修改如下：

```
import React from 'react';
import { render } from 'react-dom';
import { AppComponent } from './AppComponent';
import './main.css';

render(<AppComponent/>, window.document.getElementById('app'));
```

```
// 只有当开启了模块热替换时 module.hot 才存在
if (module.hot) {
  // accept 函数的第 1 个参数指出当前文件接收哪些子模块的替换，这里表示只接收./AppComponent 这个子模块
  // 第 2 个参数用于在新的子模块加载完毕后需要执行的逻辑
  module.hot.accept(['./AppComponent'], () => {
    // 在新的 AppComponent 加载成功后重新执行组建渲染逻辑
    render(<AppComponent/>, window.document.getElementById('app'));
  });
}
```

其中的 module.hot 是当开启模块热替换后注入全局的 API，用于控制模块热替换的逻辑。

现在修改 AppComponent.js 文件，将 Hello,Webpack 改成 Hello,World，我们会发现模块热替换生效了。但是在编辑 main.js 时，我们会发现整个网页被刷新了。为什么修改这两个文件会有不一样的表现呢？

其原因在于当子模块发生更新时，更新事件会一层层地向上传递，也就是从 AppComponent.js 文件传递到 main.js 文件，直到有某层的文件接收了当前变化的模块，即 main.js 文件中定义的 module.hot.accept(['./AppComponent'], callback)，这时就会调用 callback 函数去执行自定义逻辑。如果事件一直往上抛，到最外层都没有文件接收它，则会直接刷新网页。

那为什么没有地方接收.css 文件，但是修改所有.css 文件都会触发模块热替换呢？原因在于 style-loader 会注入用于接收 CSS 的代码。

请不要将模块热替换技术用于线上环境，它是专门为提升开发效率而生的。

4.6.2 优化模块热替换

在发生模块热替换时，我们会在浏览器的控制台中看到类似图 4-4 所示的日志。

图 4-4　模块热替换浏览器日志

其中的 `Updated modules: 68` 是指 ID 为 68 的模块被替换了，这对开发者来说很不友好，因为开发者不知道 ID 和模块之间的对应关系，最好是将替换了的模块的名称输出。Webpack 内置的 NamedModulesPlugin 插件可以解决该问题，修改 Webpack 配置文件接入该插件：

```
const NamedModulesPlugin = require('webpack/lib/NamedModulesPlugin');

module.exports = {
  plugins: [
    // 显示出被替换模块的名称
    new NamedModulesPlugin(),
  ],
};
```

重启构建后，我们会发现浏览器中的日志更友好了，如图 4-5 所示。

图 4-5　显示被替换模块的浏览器日志

除此之外，模块热替换还面临和自动刷新一样的性能问题，因为它们都需要监听文件的变化和注入客户端。优化模块热替换的构建性能的思路和在 4.5 节中提到的类似：监听更少

的文件，忽略 node_modules 目录下的文件。但是其中提到的关闭默认的 inline 模式且手动注入代理客户端的优化方法，不能用于使用模块热替换的情况，原因在于模块热替换的运行依赖在每个 Chunk 中都包含代理客户端的代码。

4.7 区分环境

4.7.1 为什么需要区分环境

在开发网页的时候，一般都会有多套运行环境，例如：

- 在开发过程中方便开发调试的环境；
- 发布到线上为用户使用的运行环境。

这两套不同的环境虽然都是由同一套源代码编译而来的，但是代码的内容不一样，其差异如下：

- 线上代码已通过在 4.8 节中提到的方法进行了压缩；
- 开发用的代码包含一些用于提示开发者的日志，普通用户不可能去看这些日志；
- 开发用的代码所连接的后端数据的接口地址也可能和线上环境不同，因为要避免在开发过程中造成对线上数据的影响。

为了尽可能复用代码，在构建的过程中需要根据目标代码要运行的环境输出不同的代码，我们需要一套机制在源码中区分环境。幸运的是，Webpack 已经为我们实现了这一点。

4.7.2 如何区分环境

具体的区分方法很简单，在源码中通过如下方式即可：

```
if (process.env.NODE_ENV === 'production') {
  console.log('你正在线上环境');
```

```
} else {
  console.log('你正在使用开发环境');
}
```

其大概原理是通过环境变量的值去判断执行哪个分支。

当代码中出现了使用 process（https://nodejs.org/api/process.html）模块的语句时，Webpack 就会自动打包进 process 模块的代码以支持非 Node.js 的运行环境。当代码中没有使用 process 时就不会打包进 process 模块的代码。这个注入的 process 模块的作用是模拟 Node.js 中的 process，以支持上面使用的 `process.env.NODE_ENV === 'production'` 语句。

在构建线上环境代码时，需要为当前的运行环境设置环境变量 `NODE_ENV = 'production'`，Webpack 的相关配置如下：

```
const DefinePlugin = require('webpack/lib/DefinePlugin');

module.exports = {
  plugins: [
    new DefinePlugin({
      // 定义 NODE_ENV 环境变量为 production
      'process.env': {
        NODE_ENV: JSON.stringify('production')
      }
    }),
  ],
};
```

注意，在定义环境变量的值时用 `JSON.stringify` 包裹字符串的原因是，环境变量的值需要是一个由双引号包裹的字符串，而 `JSON.stringify('production')` 的值正好等于 `'"production"'`。

执行构建后，我们会在输出的文件中发现如下代码：

```
if (true) {
  console.log('你正在使用线上环境');
} else {
  console.log('你正在使用开发环境');
}
```

Webpack 定义的环境变量的值被代入了源码中，`process.env.NODE_ENV ===`

'production'被直接替换成了 true，并且由于此时访问 process 的语句被替换且不存在了，Webpack 也不会将 process 模块包含到输出文件中了。

DefinePlugin 定义的环境变量只对 Webpack 需要处理的代码有效，而不会影响 Node.js 运行时的环境变量的值。

通过 Shell 脚本的方式定义的环境变量如 NODE_ENV=production webpack，Webpack 是不认识的，对 Webpack 需要处理的代码中的环境区分语句是没有作用的。

也就是说，只需要通过 DefinePlugin 定义环境变量，就能使上面介绍的环境区分语句正常工作，没必要再次通过 Shell 脚本的方式定义一遍。

如果想让 Webpack 使用通过 Shell 脚本的方式定义的环境变量，则可以使用 Environment Plugin，代码如下：

```
new webpack.EnvironmentPlugin(['NODE_ENV'])
```

以上这句代码实际上等价于：

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

4.7.3 结合 UglifyJS

其实前面输出的代码还可以进一步优化，因为 if(true)语句永远只会执行前一个分支中的代码，也就是说最佳的输出应该直接是：

```
console.log('你正在线上环境');
```

Webpack 没有实现去除死代码的功能，但是 UglifyJS 可以做这件事情，若想了解使用方法，请阅读 4.8 节中关于压缩 JavaScript 的内容。

4.7.4 第三方库中的环境区分

除了在自己写的源码中可以有环境区分的代码，很多第三方库也做了环境区分的优化。

以 React 为例，它做了两套环境区分，分别如下。

- 开发环境：包含类型检查、HTML 元素检查等针对开发者的警告日志代码。
- 线上环境：去掉了所有针对开发者的代码，只保留了让 React 能正常运行的部分，以优化大小和性能。

例如，在 React 源码中有大量的类似下面这样的代码：

```
if (process.env.NODE_ENV !== 'production') {
  warning(false, '%s(...): Can only update a mounted or mounting
component.... ')
}
```

如果不定义 NODE_ENV=production，那么这些警告日志会被包含到输出的代码中，输出的文件将会非常大。

process.env.NODE_ENV !== 'production'中的 NODE_ENV 和'production'两个值是社区的约定，通常使用这条判断语句来区分开发环境和线上环境。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-7 区分环境.zip。

4.8 压缩代码

浏览器通过服务器访问网页时获取的 JavaScript、CSS 资源都是文本形式的，文件越大，网页加载的时间越长。为了提升网页加载速度和减少网络传输流量，可以对这些资源进行压缩。除了可以通过 GZIP 算法对文件进行压缩，还可以对文本本身进行压缩。

对文本本身进行压缩，除了可以提升网页加载的速度，还有混淆源码的作用。由于压缩后的代码可读性非常差，所以就算别人下载了网页的代码，也很难进行代码分析和改造。

下面一一介绍如何在 Webpack 中压缩代码。

4.8.1 压缩 JavaScript

目前最成熟的 JavaScript 代码压缩工具是 UglifyJS（https://github.com/mishoo/UglifyJS2），它会分析 JavaScript 代码语法树，理解代码的含义，从而做到去掉无效代码、去掉日志输出代码、缩短变量名等优化。

我们需要通过插件的形式在 Webpack 中接入 UglifyJS。目前有两个成熟的插件，如下所述。

- UglifyJsPlugin：通过封装 UglifyJS 实现压缩。
- ParallelUglifyPlugin：多进程并行处理压缩，在 4.4 节中有详细介绍。

由于在 4.4 节中已介绍过 ParallelUglifyPlugin，所以这里重点介绍如何配置 UglifyJS 以达到最优的压缩效果。

UglifyJS 提供了非常多的选择，用于配置在压缩过程中采用哪些规则，可以在其官方文档（https://github.com/mishoo/UglifyJS2#minify-options）中看到所有选项说明。由于选项非常多，所以这里挑出一些常用的选项，来详细讲解其应用方式。

- `sourceMap`：是否为压缩后的代码生成对应的 Source Map，默认为不生成，开启后耗时会大大增加。一般不会将压缩后的代码的 Source Map 发送给网站用户的浏览器，而是在内部开发人员调试线上代码时使用。
- `beautify`：是否输出可读性较强的代码，即会保留空格和制表符，默认为输出，为了达到更好的压缩效果，可以设置为 `false`。
- `comments`：是否保留代码中的注释，默认为保留，为了达到更好的压缩效果，可以设置为 `false`。
- `compress.warnings`：是否在 UglifyJS 删除没有用到的代码时输出警告信息，默认为输出，可以设置为 `false` 以关闭这些作用不大的警告。
- `drop_console`：是否删除代码中的所有 `console` 语句，默认为不删除。开启后不仅可以提升代码压缩的效果，也可以兼容不支持 `console` 语句的 IE 浏览器。
- `collapse_vars`：是否内嵌虽然已定义了但是只用到一次的变量，例如将 `var x = 5;`

y = x 转换成 y = 5，默认为不转换。为了达到更好的压缩效果，可以设置为 false。

- reduce_vars：是否提取出现了多次但是没有定义成变量去引用的静态值，例如将 x = 'Hello'; y = 'Hello'转换成 var a = 'Hello'; x = a; y = b，默认为不转换。为了达到更好的压缩效果，可以设置为 false。

也就是说，在不影响代码正确执行的前提下，最优化的代码压缩配置如下：

```
const UglifyJSPlugin = require('webpack/lib/optimize/UglifyJsPlugin');

module.exports = {
  plugins: [
    // 压缩输出的 JavaScript 代码
    new UglifyJSPlugin({
      compress: {
        // 在 UglifyJS 删除没有用到的代码时不输出警告
        warnings: false,
        // 删除所有`console`语句，可以兼容 IE 浏览器
        drop_console: true,
        // 内嵌已定义但是只用到一次的变量
        collapse_vars: true,
        // 提取出现了多次但是没有定义成变量去引用的静态值
        reduce_vars: true,
      },
      output: {
        // 最紧凑的输出
        beautify: false,
        // 删除所有注释
        comments: false,
      }
    }),
  ],
};
```

从以上配置可以看出，Webpack 内置了 UglifyJsPlugin。需要指出的是，UglifyJsPlugin 当前采用的是 UglifyJS2（https://github.com/mishoo/UglifyJS2），而不是老版本的 UglifyJS1（https://github.com/mishoo/UglifyJS）。这两个版本的 UglifyJS 在配置上有所区别，看文档时要注意版本。

除此之外，Webpack 还提供了一个更简便的方法来接入 UglifyJSPlugin，直接在启动 Webpack 时带上`--optimize-minimize`参数，即`webpack --optimize-minimize`，这样 Webpack 会自动为我们注入一个带有默认配置的 UglifyJSPlugin。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-8 压缩代码-ES5.zip。

4.8.2 压缩 ES6

虽然当前大多数 JavaScript 引擎还不完全支持 ES6 中的新特性，但在一些特定的运行环境下已经可以直接执行 ES6 代码了，例如最新版的 Chrome、ReactNative 的引擎 JavaScriptCore。

运行 ES6 的代码相对于转换后的 ES5 代码有如下优点。

- 对于一样的逻辑，用 ES6 实现的代码量比 ES5 更少。
- JavaScript 引擎对 ES6 中的语法做了性能优化，例如针对`const`申明的变量有更快的读取速度。

所以在运行环境允许的情况下，我们要尽可能地使用原生的 ES6 代码去运行，而不是使用转换后的 ES5 代码。

在用上面所讲的压缩方法去压缩 ES6 代码时，我们会发现 UglifyJS 报错退出，原因是 UglifyJS 只理解 ES5 语法的代码。为了压缩 ES6 代码，需要使用专门针对 ES6 代码的 UglifyES（https://github.com/mishoo/UglifyJS2/tree/harmony）。

UglifyES 和 UglifyJS 来自同一个项目的不同分支，它们的配置项基本相同，只是接入 Webpack 时有所区别。在为 Webpack 接入 UglifyES 时，不能使用内置的 UglifyJsPlugin，而是需要单独安装和使用最新版本的 uglifyjs-webpack-plugin（https://github.com/webpack-contrib/uglifyjs-webpack-plugin）。安装方法如下：

```
npm i -D uglifyjs-webpack-plugin@beta
```

Webpack 相关配置的代码如下：

```
const UglifyESPlugin = require('uglifyjs-webpack-plugin')

module.exports = {
```

```
plugins: [
  new UglifyESPlugin({
    // 多嵌套了一层
    uglifyOptions: {
      compress: {
        // 在 UglifyJS 删除没有用到的代码时不输出警告
        warnings: false,
        // 删除所有`console` 语句，可以兼容 IE 浏览器
        drop_console: true,
        // 内嵌已定义但是只用到了一次的变量
        collapse_vars: true,
        // 提取出现多次但是没有定义成变量去引用的静态值
        reduce_vars: true,
      },
      output: {
        // 最紧凑的输出
        beautify: false,
        // 删除所有注释
        comments: false,
      }
    }
  })
]
}
```

同时，为了不让 babel-loader 输出 ES5 语法的代码，需要去掉 `.babelrc` 配置文件中的 `babel-preset-env`，但还是要保留其他 Babel 插件如 `babel-preset-react`，因为正是 `babel-preset-env` 负责将 ES6 代码转换为 ES5 代码。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-8 压缩代码-ES6.zip。

4.8.3 压缩 CSS

CSS 代码也可以像 JavaScript 那样被压缩，以达到提升加载速度和代码混淆的作用。目前比较成熟、可靠的 CSS 压缩工具是 cssnano（http://cssnano.co），基于 PostCSS。

cssnano 能理解 CSS 代码的含义，而不仅仅是删掉空格，如下所述。

- margin:10px 20px 10px 20px 被压缩成 margin: 10px 20px。
- color:#ff0000 被压缩成 color:red。

可以到其官网查看更多的压缩规则，通常压缩率能达到 60%。

将 cssnano 接入 Webpack 中也非常简单，因为 css-loader 已经将其内置了，要开启 cssnano 去压缩代码，则只需开启 css-loader 的 minimize 选项。相关的 Webpack 配置如下：

```
const path = require('path');
const {WebPlugin} = require('web-webpack-plugin');
const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  module: {
    rules: [
      {
        test: /\.css/,// 增加对 CSS 文件的支持
        // 提取 Chunk 中的 CSS 代码到单独的文件中
        use: ExtractTextPlugin.extract({
          // 通过 minimize 选项压缩 CSS 代码
          use: ['css-loader?minimize']
        }),
      },
    ]
  },
  plugins: [
    // 用 WebPlugin 生成对应的 HTML 文件
    new WebPlugin({
      template: './template.html', // HTML 模板文件所在的文件路径
      filename: 'index.html' // 输出的 HTML 的文件名称
    }),
    new ExtractTextPlugin({
      filename: `[name]_[contenthash:8].css`,// 为输出的 CSS 文件名称加上
Hash 值
    }),
  ],
};
```

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-8 压缩代码-CSS.zip。

4.9 CDN 加速

4.9.1 什么是 CDN

虽然在前面通过压缩代码的手段减小了网络传输的大小，但实际上最影响用户体验的还是网页首次打开时的加载等待，其根本原因是网络传输过程耗时较大。CDN（内容分发网络）的作用就是加速网络传输，通过将资源部署到世界各地，使用户在访问时按照就近原则从离其最近的服务器获取资源，来加快资源的获取速度。CDN 其实是通过优化物理链路层传输过程中的光速有限、丢包等问题来提升网速的，其大致原理如图 4-6 所示。

图 4-6　CDN 的原理

在本节中，我们不必理解 CDN 的具体运行流程和实现原理，可以简单地将 CDN 服务看作成速度更快的 HTTP 服务。并且目前很多大公司都会建立自己的 CDN 服务，就算我们没有资源去搭建一套 CDN 服务，各大云服务提供商也都为我们提供了按量收费的 CDN 服务。

4.9.2 接入 CDN

要为网站接入 CDN，需要将网页的静态资源上传到 CDN 服务上，在服务这些静态资源时需要通过 CDN 服务提供的 URL 地址去访问。

举个详细的例子，有一个单页应用，其构建出的代码结构如下：

```
dist
|-- app_9d89c964.js
|-- app_a6976b6d.css
|-- arch_ae805d49.png
`-- index.html
```

其中，index.html 的内容如下：

```
<html>
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="app_a6976b6d.css">
</head>
<body>
<div id="app"></div>
<script src="app_9d89c964.js"></script>
</body>
</html>
```

app_a6976b6d.css 的内容如下：

```
body{background:url(arch_ae805d49.png) repeat}h1{color:red}
```

可以看出，在导入资源之前都是通过相对路径去访问的，当将这些资源都放到同一个 CDN 服务上时，网页能正常使用。但需要注意的是，由于 CDN 服务一般都会为资源开启很长时间的缓存，例如用户从 CDN 上获取 index.html 这个文件后，即使之后的发布操作将 index.html 文件重新覆盖了，但是用户在很长一段时间内还是会运行之前的版本，这会导致新的发布不能立即生效。

要避免以上问题，业界比较成熟的做法如下。

- 针对 HTML 文件：不开启缓存，将 HTML 放到自己的服务器上，而不是 CDN 服务上，同时关闭自己服务器上的缓存。自己的服务器只提供 HTML 文件和数据接口。

- 针对静态的 JavaScript、CSS、图片等文件：开启 CDN 和缓存，上传到 CDN 服务上，同时为每个文件名带上由文件内容算出的 Hash 值，例如上面的 app_a6976b6d.css 文件。带上 Hash 值的原因是文件名会随着文件的内容而变化，只要文件的内容发生变化，其对应的 URL 就会变化，它就会被重新下载，无论缓存时间有多长。

采用以上方案后，也需要将 HTML 文件中的资源引入地址换成 CDN 服务提供的地址，例如以上 index.html 变为：

```
<html>
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="//cdn.com/id/app_a6976b6d.css">
</head>
<body>
<div id="app"></div>
<script src="//cdn.com/id/app_9d89c964.js"></script>
</body>
</html>
```

并且 app_a6976b6d.css 的内容也应该变为：

```
body{background:url(//cdn.com/id/arch_ae805d49.png)
repeat}h1{color:red}
```

也就是说，之前的相对路径都变成了绝对的指向 CDN 服务的 URL 地址。

如果对形如//cdn.com/id/app_a6976b6d.css 这样的 URL 感到陌生，则我们需要知道这种 URL 省掉了前面的 http:或者 https:前缀。这样做的好处是，在访问这些资源时会自动根据当前 HTML 的 URL 采用了什么模式去决定是采用 HTTP 还是 HTTPS 模式。

除此之外，如果我们还知道浏览器有一个规则是，在同一时刻针对同一个域名的资源的并行请求有限制（大概 4 个左右，不同的浏览器可能不同），则会发现上面的做法有很大的问题。由于所有静态资源都被放到了同一个 CDN 服务的域名下，也就是上面的 cdn.com 下，如果网页的资源很多，例如有很多图片，就会导致资源的加载被阻塞，因为同时只能加载几个，必须等其他资源加载完才能继续加载。要解决这个问题，我们可以将这些静态资源分散到不同的 CDN 服务上，例如将 JavaScript 文件放到 js.cdn.com 域名下，将 CSS 文件放到 css.cdn.com 域名下，将图片文件放到 img.cdn.com 域名下，这样 index.html

需要变成：

```
<html>
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="//css.cdn.com/id/app_a6976b6d.css">
</head>
<body>
<div id="app"></div>
<script src="//js.cdn.com/id/app_9d89c964.js"></script>
</body>
</html>
```

使用多个域名后又会带来一个新的问题：增加域名解析时间。对于是否采用多域名分散资源，需要根据自己的需求去衡量得失。当然，可以通过在 HTML HEAD 标签中加入`<link rel="dns-prefetch" href="//js.cdn.com">`预解析域名，以减少域名解析带来的延迟。

4.9.3 用 Webpack 实现 CDN 的接入

总之，构建需要实现以下几点。

- 静态资源的导入 URL 需要变成指向 CDN 服务的绝对路径的 URL，而不是相对于 HTML 文件的 URL。
- 静态资源的文件名需要带上由文件内容算出来的 Hash 值，以防止被缓存。
- 将不同类型的资源放到不同域名的 CDN 服务上，以防止资源的并行加载被阻塞。

先来看看要实现以上要求的最终 Webpack 配置：

```
const path = require('path');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
const {WebPlugin} = require('web-webpack-plugin');

module.exports = {
  // 省略 entry 配置...
  output: {
```

```
    // 为输出的 JavaScript 文件名加上 Hash 值
    filename: '[name]_[chunkhash:8].js',
    path: path.resolve(__dirname, './dist'),
    // 指定存放 JavaScript 文件的 CDN 目录 URL
    publicPath: '//js.cdn.com/id/',
  },
  module: {
    rules: [
      {
        // 增加对 CSS 文件的支持
        test: /\.css/,
        // 提取 Chunk 中的 CSS 代码到单独的文件中
        use: ExtractTextPlugin.extract({
          // 压缩 CSS 代码
          use: ['css-loader?minimize'],
          // 指定存放 CSS 中导入的资源（例如图片）的 CDN 目录 URL
          publicPath: '//img.cdn.com/id/'
        }),
      },
      {
        // 增加对 PNG 文件的支持
        test: /\.png/,
        // 为输出的 PNG 文件名加上 Hash 值
        use: ['file-loader?name=[name]_[hash:8].[ext]'],
      },
      // 省略其他 Loader 配置...
    ]
  },
  plugins: [
    // 使用 WebPlugin 自动生成 HTML
    new WebPlugin({
      // HTML 模板文件所在的文件路径
      template: './template.html',
      // 输出的 HTML 的文件名
      filename: 'index.html',
      // 指定存放 CSS 文件的 CDN 目录 URL
      stylePublicPath: '//css.cdn.com/id/',
    }),
    new ExtractTextPlugin({
      // 为输出的 CSS 文件名加上 Hash 值
```

```
      filename: `[name]_[contenthash:8].css`,
    }),
    // 省略代码压缩插件配置...
  ],
};
```

在以上代码中最核心的部分是通过 publicPath 参数设置存放静态资源的 CDN 目录 URL。为了让不同类型的资源输出到不同的 CDN，需要分别进行如下设置。

- 在 output.publicPath 中设置 JavaScript 的地址。
- 在 css-loader.publicPath 中设置被 CSS 导入的资源的地址。
- 在 WebPlugin.stylePublicPath 中设置 CSS 文件的地址。

设置好 publicPath 后，WebPlugin 在生成 HTML 文件并将 css-loader 转换 CSS 代码时，会考虑到配置中的 publicPath，用对应的线上地址替换原来的相对地址。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-9 CDN 加速.zip。

4.10 使用 Tree Shaking

4.10.1 认识 Tree Shaking

Tree Shaking 可以用来剔除 JavaScript 中用不上的死代码。它依赖静态的 ES6 模块化语法，例如通过 import 和 export 导入、导出。Tree Shaking 最先在 Rollup 中出现，Webpack 在 2.0 版本中将其引入。

为了更直观地理解它，来看一个具体的例子。假如有一个文件 util.js 里存放了很多工具函数和常量，在 main.js 中会导入和使用 util.js，代码如下：

util.js 源码如下：

```
export function funcA() {
}
```

```
export function funB() {
}
```

main.js 源码如下：

```
import {funcA} from './util.js';
funcA();
```

Tree Shaking 后的 util.js 如下：

```
export function funcA() {
}
```

由于只用到了 util.js 中的 funcA，所以剩下的都被 Tree Shaking 当作死代码剔除了。

需要注意，要让 Tree Shaking 正常工作的前提是，提交给 Webpack 的 JavaScript 代码必须采用了 ES6 的模块化语法，因为 ES6 模块化语法是静态的（在导入、导出语句中的路径必须是静态的字符串，而且不能放入其他代码块中），这让 Webpack 可以简单地分析出哪些 export 的被 import 了。如果采用了 ES5 中的模块化，例如 module.export={...}、require(x+y)、if(x){require('./util')}，则 Webpack 无法分析出可以剔除哪些代码。

4.10.2 接入 Tree Shaking

前面讲了 Tree Shaking 是做什么的，接下来讲解如何配置 Webpack 让 Tree Shaking 生效。

首先，为了将采用 ES6 模块化的代码提交给 Webpack，需要配置 Babel 以让其保留 ES6 模块化语句。修改.babelrc 文件如下：

```
{
  "presets": [
    [
      "env",
      {
        "modules": false
      }
    ]
  ]
```

```
}
```

其中，"modules": false 的含义是关闭 Babel 的模块转换功能，保留原本的 ES6 模块化语法。

配置好 Babel 后，重新运行 Webpack，在启动 Webpack 时带上--display-used-exports 参数，以方便追踪 Tree Shaking 的工作。这时我们会发现在控制台中输出了如下日志：

```
> webpack --display-used-exports
bundle.js  3.5 kB       0  [emitted]  main
   [0] ./main.js 41 bytes {0} [built]
   [1] ./util.js 511 bytes {0} [built]
       [only some exports used: funcA]
```

其中，[only some exports used: funcA]提示了 util.js 只导出了用到的 funcA，说明 Webpack 确实正确分析出了如何剔除死代码。

若打开 Webpack 输出的 bundle.js 文件并查看，则会发现用不上的代码还在里面：

```
/* harmony export (immutable) */
__webpack_exports__["a"] = funcA;

/* unused harmony export funB */

function funcA() {
  console.log('funcA');
}

function funB() {
  console.log('funcB');
}
```

Webpack 只是指出了哪些函数被用上了，而哪些函数没被用上，要剔除用不上的代码，则还得经过 UglifyJS 处理一遍。要接入 UglifyJS，也很简单，不仅可以通过 4.8 节中介绍的加入 UglifyJSPlugin 去实现，也可以简单地通过在启动 Webpack 时带上--optimize-minimize 参数来实现，为了快速验证 Tree Shaking，我们采用较简单的后者来实验一下。

通过 webpack --display-used-exports --optimize-minimize 重启 Webpack 后，打开新输出的 bundle.js，内容如下：

```
function r() {
  console.log("funcA")
}

t.a = r
```

可以看出 Tree Shaking 确实做到了，用不上的代码都被剔除了。

在项目中使用大量的第三方库时，我们会发现 Tree Shaking 似乎不生效了，原因是大部分 Npm 中的代码都采用了 CommonJS 语法，这导致 Tree Shaking 无法正常工作而降级处理。但幸运的是，有些库考虑到了这一点，这些库在发布到 Npm 上时会同时提供两份代码，一份采用 CommonJS 模块化语法，一份采用 ES6 模块化语法。并且在 package.json 文件中分别指出这两份代码的入口。

以 redux 库为例，其发布到 Npm 上的目录结构为：

```
node_modules/redux
|-- es
|   |-- index.js # 采用 ES6 模块化语法
|-- lib
|   |-- index.js # 采用 ES5 模块化语法
|-- package.json
```

在 package.json 文件中有两个字段：

```
{
  "main": "lib/index.js", // 指明采用 CommonJS 模块化的代码入口
  "jsnext:main": "es/index.js" // 指明采用 ES6 模块化的代码入口
}
```

在 2.4 节中曾介绍过 mainFields 用于配置采用哪个字段作为模块的入口描述。为了让 Tree Shaking 对 redux 生效，需要配置 Webpack 的文件寻找规则如下：

```
module.exports = {
  resolve: {
    // 针对 Npm 中的第三方模块优先采用 jsnext:main 中指向的 ES6 模块化语法的文件
    mainFields: ['jsnext:main', 'browser', 'main']
  },
};
```

以上配置的含义是优先使用 jsnext:main 作为入口，如果不存在，jsnext:main 就

会采用 browser 或者 main 并将其作为入口。虽然并不是每个 Npm 中的第三方模块都会提供 ES6 模块化语法的代码，但对于已提供了的代码要尽量优化。

目前越来越多的 Npm 中的第三方模块都考虑到了 Tree Shaking，并对其提供了支持。采用 jsnext:main 作为 ES6 模块化代码的入口是社区的一个约定，假如将来要发布一个库到 Npm，则我们希望该库能支持 Tree Shaking，以让 Tree Shaking 发挥更大的优化效果，让更多的人受益。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-10 使用 TreeShaking.zip。

4.11 提取公共代码

4.11.1 为什么需要提取公共代码

大型网站通常由多个页面组成，每个页面都是一个独立的单页应用。但由于所有页面都采用同样的技术栈及同一套样式代码，就导致这些页面之间有很多相同的代码。

如果每个页面的代码都将这些公共的部分包含进去，则会造成以下问题。

- 相同的资源被重复加载，浪费用户的流量和服务器的成本。
- 每个页面需要加载的资源太大，导致网页首屏加载缓慢，影响用户体验。

如果将多个页面的公共代码抽离成单独的文件，就能优化以上问题。原因是假如用户访问了某网站的其中一个网页，那么访问这个网站下的其他网页的概率将非常大。在用户第一次访问后，这些页面的公共代码的文件已经被浏览器缓存起来，在用户切换到其他页面时，就不会再重新加载存放公共代码的文件，而是直接从缓存中获取。这样做有如下好处。

- 减少网络传输流量，降低服务器成本。
- 虽然用户第一次打开网站的速度得不到优化，但之后访问其他页面的速度将大大提升。

4.11.2 如何提取公共代码

已经知道了提取公共代码有什么好处，那么在实战中具体要怎么做，才能达到最佳效果呢？通常可以采用以下原则为网站提取公共代码。

- 根据网站所使用的技术栈，找出网站的所有页面都需要用到的基础库，以采用 React 技术栈的网站为例，所有页面都会依赖 react、react-dom 等库，将它们提取到一个单独的文件 base.js 中，该文件包含了所有网页的基础运行环境。
- 在剔除了各个页面中被 base.js 包含的部分代码后，再找出所有页面都依赖的公共部分的代码，将它们提取出来并放到 common.js 中。
- 再为每个网页都生成一个单独的文件，在这个文件中不再包含 base.js 和 common.js 中包含的部分，而只包含各个页面单独需要的部分代码。

文件之间的结构图如图 4-7 所示。

图 4-7 提取的公共代码文件的结构图

读到这里我们可能会有疑问：既然能找出所有页面都依赖的公共代码，并提取出来放到 common.js 中，那为什么还需要再将网站的所有页面都需要用到的基础库提取到 base.js 中呢？答案是为了长期缓存 base.js 这个文件。

发布到线上的文件都会采用在 4.9 节中介绍过的方法，对静态文件的文件名都附加根据文件内容计算出来的 Hash 值，即最终 base.js 的文件名会变成 base_3b1682ac.js，以长期缓存文件。网站通常会不断地更新发布，每次发布都会导致 common.js 和各个网页的 JavaScript 文件因为文件的内容发生变化而导致其 Hash 值被更新，也就是缓存被更新。

将所有页面都需要用到的基础库提取到 base.js 中的好处在于，只要不升级基础库的版本，base.js 的文件内容就不会变化，Hash 值不会被更新，缓存就不会被更新。每次发布时浏览器都会使用被缓存的 base.js 文件，而不用重新下载 base.js 文件。由于 base.js 通常会很大，所以对提升网页的响应速度能起到很大的效果。

4.11.3 如何通过 Webpack 提取公共代码

我们通过前面的内容已经知道了如何提取公共代码，接下来讲解如何用 Webpack 提取公共代码。

Webpack 内置了专门用于提取多个 Chunk 中的公共部分的插件 CommonsChunkPlugin，CommonsChunkPlugin 的大致使用方法如下：

```
const CommonsChunkPlugin =
require('webpack/lib/optimize/CommonsChunkPlugin');

new CommonsChunkPlugin({
  // 从哪些 Chunk 中提取
  chunks: ['a', 'b'],
  // 提取出的公共部分形成一个新的 Chunk
  name: 'common'
})
```

通过以上配置就能从网页 A 和网页 B 中抽离出公共部分，放到 common 中。

每个 CommonsChunkPlugin 实例都会生成一个新的 Chunk，在这个 Chunk 中包含了被提取的代码，在使用的过程中必须指定 name 属性，以告诉插件新生成的 Chunk 的名称。其中 chunks 属性指明从哪些已有的 Chunk 中提取，如果不填该属性，则默认会从所有已知的 Chunk 中提取。

Chunk 是一系列文件的集合，在一个 Chunk 中会包含这个 Chunk 的入口文件和该入口文件依赖的文件。

在通过以上配置输出的 common Chunk 中会包含所有页面都依赖的基础运行库 react、react-dom，为了将基础运行库从 common 中抽离到 base 中，还需要做一些处理。

首先需要配置一个 Chunk，在这个 Chunk 中只依赖所有页面都依赖的基础库及所有页面都使用的样式，为此需要在项目中写一个文件 base.js 来描述 base Chunk 所依赖的模块，文件的内容如下：

```
// 所有页面都依赖的基础库
import 'react';
import 'react-dom';
// 所有页面都使用的样式
import './base.css';
```

接着修改 Webpack 的配置，在 entry 中加入 base，相关修改如下：

```
module.exports = {
  entry: {
    base: './base.js'
  },
};
```

这样就完成了对新 Chunk base 的配置。

为了从 common 中提取出 base 也包含的部分，还需要配置一个 CommonsChunkPlugin，相关代码如下：

```
new CommonsChunkPlugin({
  // 从 common 和 base 两个现成的 Chunk 中提取公共部分
  chunks: ['common', 'base'],
  // 将公共部分放到 base 中
  name: 'base'
})
```

由于 common 和 base 的公共部分就是 base 目前已经包含的部分，所以这样配置后 common 将会变小，而 base 将保持不变。

在以上内容都配置好后重新执行构建，将会得到以下 4 个文件。

- base.js：所有网页都依赖的基础库组成的代码。
- common.js：网页 A、B 都需要的但没在 base.js 文件中出现过的代码。
- a.js：网页 A 单独需要的代码。
- b.js：网页 B 单独需要的代码。

为了让网页正常运行，以网页 A 为例，我们需要在其 HTML 中按照以下顺序引入以下文件，才能让网页正常运行：

```
<script src="base.js"></script>
<script src="common.js"></script>
<script src="a.js"></script>
```

这样就完成了提取公共代码所需的所有步骤。

针对 CSS 资源，以上理论和方法同样有效，也就是说可以对 CSS 文件做同样的优化。

采用以上方法后可能会出现 common.js 中没有代码的情况，原因是去掉基础运行库后，很难再找到所有页面都会用上的模块。在出现这种情况时可以采取以下做法。

- CommonsChunkPlugin 提供了一个选项 minChunks，表示文件要被提取出来时需要在指定的 Chunks 中出现的最小次数。假如 minChunks=2、chunks=['a','b','c','d']，则任何一个文件只要在['a','b','c','d']中两个以上的 Chunk 中都出现过，这个文件就会被提取出来。我们可以根据自己的需求去调整 minChunks 的值，minChunks 越小，被提取到 common.js 中的文件就会越多，但这也会导致部分页面加载的不相关的资源越多；minChunks 越大，被提取到 common.js 中的文件就会越少，但这会导致 common.js 变小、效果变弱。
- 根据各个页面之间的相关性选取其中的部分页面时，可用 CommonsChunkPlugin 提取这部分被选出的页面的公共部分，而不是提取所有页面的公共部分，而且这样的操作可以叠加多次。这样做的效果会很好，但缺点是配置复杂，需要根据页面之间的关系去思考如何配置，该方法并不通用。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-11 提取公共代码.zip。

4.12 分割代码以按需加载

4.12.1 为什么需要按需加载

随着互联网的发展，一个网页需要承载的功能越来越多。采用单页应用作为前端架构的网站会面临着网页需要加载的代码量很大的问题，因为许多功能都被集中做到了一个 HTML 里，这会导致网页加载缓慢、交互卡顿，使用户体验非常糟糕。

导致这个问题的根本原因在于一次性加载所有功能对应的代码，但其实用户在每个阶段只可能使用其中一部分功能，所以解决以上问题的方法就是用户当前需要用什么功能就只加载这个功能对应的代码，也就是所谓的按需加载。

4.12.2 如何使用按需加载

在为单页应用做按需加载优化时，一般采用以下原则。

- 将整个网站划分成一个个小功能，再按照每个功能的相关程度将它们分成几类。
- 将每一类合并为一个 Chunk，按需加载对应的 Chunk。
- 不要按需加载用户首次打开网站时需要看到的画面所对应的功能，将其放到执行入口所在的 Chunk 中，以减少用户能感知的网页加载时间。
- 对于不依赖大量代码的功能点，例如依赖 Chart.js 去画图表、依赖 flv.js 去播放视频的功能点，可再对其进行按需加载。

被分割出去的代码的加载需要一定的时机去触发，即当用户操作到了或者即将操作到对

应的功能时再去加载对应的代码。被分割出去的代码的加载时机需要开发者根据网页的需求去衡量和确定。

由于被分割出去进行按需加载的代码在加载的过程中也需要耗时，所以可以预估用户接下来可能会进行的操作，并提前加载对应的代码，让用户感知不到网络加载。

4.12.3 用 Webpack 实现按需加载

Webpack 内置了强大的分割代码的功能去实现按需加载，实现起来非常简单。举个例子，现在需要做这样一个进行了按需加载优化的网页。

- 网页首次加载时只加载 main.js 文件，网页会展示一个按钮，在 main.js 文件中只包含监听按钮事件和加载按需加载的代码。
- 在按钮被单击时才去加载被分割出去的 show.js 文件，在加载成功后再执行 show.js 里的函数。

其中，main.js 文件的内容如下：

```
window.document.getElementById('btn').addEventListener('click',
function () {
  // 在按钮被单击后才去加载 show.js 文件，文件加载成功后执行文件导出的函数
  import(/* webpackChunkName: "show" */ './show').then((show) => {
    show('Webpack');
  })
});
```

show.js 文件的内容如下：

```
module.exports = function (content) {
  window.alert('Hello ' + content);
};
```

其中最关键的一句是：

```
import(/* webpackChunkName: "show" */ './show')
```

Webpack 内置了对 import(*)语句的支持，当 Webpack 遇到了类似的语句时会这样

处理：

- 以./show.js 为入口重新生成一个 Chunk；
- 当代码执行到 import 所在的语句时才去加载由 Chunk 对应生成的文件；
- import 返回一个 Promise，当文件加载成功时可以在 Promise 的 then 方法中获取 show.js 导出的内容。

在使用 import()分割代码后，浏览器要支持 Promise API（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise）才能使代码正常运行，因为 import()返回一个 Promise，它依赖 Promise。对于不原生支持 Promise 的浏览器，可以注入 Promise polyfill。

/* webpackChunkName: "show" */的含义是为动态生成的 Chunk 赋予一个名称，以方便我们追踪和调试代码。如果不指定动态生成的 Chunk 的名称，则其默认的名称将会是[id].js。/* webpackChunkName: "show" */是在 Webpack 3 中引入的新特性，在 Webpack 3 之前是无法为动态生成的 Chunk 赋予名称的。

为了正确输出在/* webpackChunkName: "show" */中配置的 ChunkName，还需要配置 Webpack：

```
module.exports = {
  // JavaScript 执行入口文件
  entry: {
    main: './main.js',
  },
  output: {
    // 为从 entry 中配置生成的 Chunk 配置输出文件的名称
    filename: '[name].js',
    // 为动态加载的 Chunk 配置输出文件的名称
    chunkFilename: '[name].js',
  }
};
```

其中，最关键的一行是 chunkFilename: '[name].js'，它专门指定动态生成的 Chunk 在输出时的文件名称。如果没有这一行，则分割出的代码的文件名称将会是[id].js。chunkFilename 的具体含义参见 2.2 节。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-12 分割代码按需加载.zip。

4.12.4 按需加载与 ReactRouter

在实战中不可能会有上面那么简单的场景。接下来讲解一个实战案例，对采用了 ReactRouter（https://reacttraining.com/react-router/web）的应用进行按需加载优化。这个案例由一个单页应用构成，这个单页应用由两个子页面构成，通过 ReactRouter 在两个子页面之间切换和管理路由。

这个单页应用的入口文件 main.js 如下：

```
import React, {PureComponent, createElement} from 'react';
import {render} from 'react-dom';
import {HashRouter, Route, Link} from 'react-router-dom';
import PageHome from './pages/home';

/**
 * 异步加载组件
 * @param load 组件加载函数，load 函数会返回一个 Promise，在文件加载完成时 resolve
 * @returns {AsyncComponent}返回一个高阶组件用于封装需要异步加载的组件
 */
function getAsyncComponent(load) {
  return class AsyncComponent extends PureComponent {

    componentDidMount() {
      // 在高阶组件 DidMount 时才去执行网络加载步骤
      load().then(({default: component}) => {
        // 代码加载成功，获取了代码导出的值，调用 setState，通知高阶组件重新渲染子
组件
        this.setState({
          component,
        })
      });
    }

    render() {
      const {component} = this.state || {};
```

```
        // component 是 React.Component 类型，需要通过 React.createElement 生产
一个组件实例
        return component ? createElement(component) : null;
      }
    }
  }

  // 根组件
  function App() {
    return (
      <HashRouter>
        <div>
          <nav>
            <Link to='/'>Home</Link> | <Link to='/about'>About</Link> |
<Link to='/login'>Login</Link>
          </nav>
          <hr/>
          <Route exact path='/' component={PageHome}/>
          <Route path='/about' component={getAsyncComponent(
            // 异步加载函数，异步加载 PageAbout 组件
            () => import(/* webpackChunkName: 'page-about'
*/'./pages/about')
          )}
          />
          <Route path='/login' component={getAsyncComponent(
            // 异步加载函数，异步加载 PageAbout 组件
            () => import(/* webpackChunkName: 'page-login'
*/'./pages/login')
          )}
          />
        </div>
      </HashRouter>
    )
  }

  // 渲染根组件
  render(<App/>, window.document.getElementById('app'));
```

在以上代码中最关键的部分是 getAsyncComponent 函数，它的作用是配合 ReactRouter 按需加载组件，具体含义请看代码中的注释。

由于以上源码需要通过 Babel 转换后才能在浏览器中正常运行，所以需要在 Webpack 中配置好对应的 babel-loader，将源码先提交给 babel-loader 处理，再提交给 Webpack 处理其中的 import(*)语句。但这样做会有一个问题：Babel 报错，说不理解 import(*)语法。其原因是 import(*)语法还没有被加入在 3.1 节中提到的 ECMAScript 标准里。为此，我们需要安装一个 Babel 插件 babel-plugin-syntax-dynamic-import，并且将其加入.babelrc 中：

```
{
  "presets": [
    "env",
    "react"
  ],
  "plugins": [
    "syntax-dynamic-import"
  ]
}
```

执行 Webpack 构建后，我们会发现输出了以下三个文件。

- main.js：执行入口所在的代码块，同时包括 PageHome 所需的代码，因为用户首次打开网页时就需要看到 PageHome 的内容，所以不对其进行按需加载，以减少用户能感知到的加载时间。
- page-about.js：在用户访问/about 时才会加载的代码块。
- page-login.js：在用户访问/login 时才会加载的代码块。

同时我们会发现，在首页不会加载 page-about.js 和 page-login.js 这两个文件，而是在切换到了对应的子页面后才会加载。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-12 分割代码按需加载 ReactRouter.zip。

4.13 使用 Prepack

4.13.1 认识 Prepack

在前面的优化方法中提到了代码压缩和分块，这些都是在网络加载层面的优化，除此之外，还可以优化代码在运行时的效率，Prepack（https://prepack.io）就是为此而生的。

Prepack 由 Facebook 开源，采用了较为激进的方法：在保持运行结果一致的情况下，改变源代码的运行逻辑，输出性能更好的 JavaScript 代码。实际上，Prepack 就是一个部分求值器，编译代码时提前将计算结果放到编译后的代码中，而不是在代码运行时才去求值。

以如下源码为例：

```
import React, {Component} from 'react';
import {renderToString} from 'react-dom/server';

function hello(name) {
  return 'hello ' + name;
}

class Button extends Component {
  render() {
    return hello(this.props.name);
  }
}

console.log(renderToString(<Button name='webpack'/>));
```

被 Prepack 转化后竟然直接输出：

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

可以看出 Prepack 通过在编译阶段预先执行源码来得到执行结果，再直接将运行结果输出以提升性能。

Prepack 的工作原理和流程大致如下：

- 通过 Babel 将 JavaScript 源码解析成抽象语法树（AST），以更细粒度地分析源码；
- Prepack 实现了一个 JavaScript 解释器，用于执行源码。借助这个解释器，Prepack 才能理解源码具体是如何执行的，并将执行过程中的结果返回到输出中。

从表面上看这似乎非常美好，但实际上 Prepack 还不够成熟与完善。Prepack 目前还处于初期开发阶段，局限性也很大，例如：

- 不能识别 DOM API 和部分 Node.js API，如果在源码中有调用依赖运行环境的 API，就会导致 Prepack 报错；
- 代码在优化后性能可能更差；
- 代码在优化后，文件的尺寸可能大大增加。

总之，现在将 Prepack 用于线上环境还为时过早。

4.13.2 接入 Webpack

Prepack 需要在 Webpack 输出最终的代码之前，对这些代码进行优化，就像 UglifyJS 那样，因此需要通过新接入一个插件来为 Webpack 接入 Prepack。幸运的是社区中已经有人做好了这个插件：prepack-webpack-plugin（https://github.com/gajus/prepack-webpack-plugin）。

接入该插件非常简单，相关配置代码如下：

```
const PrepackWebpackPlugin = require('prepack-webpack-plugin').default;

module.exports = {
  plugins: [
    new PrepackWebpackPlugin()
  ]
};
```

重新执行构建，就会看到输出了被 Prepack 优化后的代码。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-13 使用 Prepack.zip。

4.14 开启 Scope Hoisting

Scope Hoisting 可以让 Webpack 打包出来的代码文件更小、运行更快，它又被译作“作用域提升”，是在 Webpack 3 中新推出的功能。下面详细讲解 Scope Hoisting。

4.14.1 认识 Scope Hoisting

让我们先来看看在没有 Scope Hoisting 之前 Webpack 的打包方式。

假如现在有两个文件，分别是 util.js：

```
export default 'Hello,Webpack';
```

和入口文件 main.js：

```
import str from './util.js';
console.log(str);
```

以上源码用 Webpack 打包后输出的部分代码如下：

```
[
  (function (module, __webpack_exports__, __webpack_require__) {
    var __WEBPACK_IMPORTED_MODULE_0__util_js__ =
__webpack_require__(1);
    console.log(__WEBPACK_IMPORTED_MODULE_0__util_js__["a"]);
  }),
  (function (module, __webpack_exports__, __webpack_require__) {
    __webpack_exports__["a"] = ('Hello,Webpack');
  })
]
```

在开启 Scope Hoisting 后，以同样的源码输出的部分代码如下：

```
[
  (function (module, __webpack_exports__, __webpack_require__) {
    var util = ('Hello,Webpack');
    console.log(util);
  })
]
```

可以看出开启 Scope Hoisting 后，函数申明由两个变成了一个，util.js 中定义的内容被直接注入 main.js 对应的模块中。这样做的好处是：

- 代码体积更小，因为函数申明语句会产生大量的代码；
- 代码在运行时因为创建的函数作用域变少了，所以内存开销也变小了。

Scope Hoisting 的实现原理其实很简单：分析模块之间的依赖关系，尽可能将被打散的模块合并到一个函数中，但前提是不能造成代码冗余。因此只有那些被引用了一次的模块才能被合并。

由于 Scope Hoisting 需要分析模块之间的依赖关系，因此源码必须采用 ES6 模块化语句，不然它将无法生效，其原因和在 4.10 节中提到的原因类似。

4.14.2 使用 Scope Hoisting

在 Webpack 中使用 Scope Hoisting 非常简单，因为这是 Webpack 内置的功能，只需要配置一个插件，相关代码如下：

```
const ModuleConcatenationPlugin =
require('webpack/lib/optimize/ModuleConcatenationPlugin');

module.exports = {
  plugins: [
    // 开启 Scope Hoisting
    new ModuleConcatenationPlugin(),
  ],
};
```

同时，考虑到 Scope Hoisting 依赖源码时需采用 ES6 模块化语法，还需要配置 mainFields。因为大部分 Npm 中的第三方库采用了 CommonJS 语法，但部分库会同时提供 ES6 模块化的代码，所以为了充分发挥 Scope Hoisting 的作用，需要增加以下配置：

```
module.exports = {
  resolve: {
    // 针对 Npm 中的第三方模块优先采用 jsnext:main 中指向的 ES6 模块化语法的文件
```

```
    mainFields: ['jsnext:main', 'browser', 'main']
  },
};
```

对于采用了非 ES6 模块化语法的代码，Webpack 会降级处理且不使用 Scope Hoisting 优化。为了知道 Webpack 对哪些代码做了降级处理，我们可以在启动 Webpack 时带上 --display-optimization-bailout 参数，这样在输出日志中就会包含类似如下的日志：

```
[0] ./main.js + 1 modules 80 bytes {0} [built]
  ModuleConcatenation bailout: Module is not an ECMAScript module
```

其中的 ModuleConcatenation bailout 告诉我们哪个文件因为什么原因导致了降级处理。

也就是说，开启 Scope Hoisting 并发挥最大作用的配置如下：

```
const ModuleConcatenationPlugin =
require('webpack/lib/optimize/ModuleConcatenationPlugin');

module.exports = {
  resolve: {
    // 针对 Npm 中的第三方模块优先采用 jsnext:main 中指向的 ES6 模块化语法的文件
    mainFields: ['jsnext:main', 'browser', 'main']
  },
  plugins: [
    // 开启 Scope Hoisting
    new ModuleConcatenationPlugin(),
  ],
};
```

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-14 开启 ScopeHoisting.zip。

4.15 输出分析

前面虽然介绍了非常多的优化方法，但这些方法无法应对所有场景，为此我们需要对输出结果进行分析，以决定下一步的优化方向。

最直接的分析方法是阅读 Webpack 输出的代码，但 Webpack 输出的代码可读性非常差而且文件非常大，让我们非常头疼。为了更简单、直观地分析输出结果，社区中出现了许多可视化分析工具。这些工具以图形的方式将结果更直观地展示出来，让我们快速了解问题所在。接下来讲解如何使用这些工具。

在启动 Webpack 时支持如下两个参数。

- `--profile`：记录构建过程中的耗时信息。
- `--json`：以 JSON 的格式输出构建结果，最后只输出一个 `.json` 文件，这个文件中包括所有构建相关的信息。

在启动 Webpack 时带上以上两个参数，启动命令如下：

```
webpack --profile --json > stats.json
```

我们会发现项目中多出了一个 `stats.json` 文件。这个 `stats.json` 文件是为后面介绍的可视化分析工具使用的。

`webpack --profile --json` 会输出字符串形式的 JSON，`> stats.json` 是 UNIX/Linux 系统中的管道命令，其含义是将 `webpack --profile --json` 输出的内容通过管道输出到 `stats.json` 文件中。

4.15.1 官方的可视化分析工具

Webpack 官方提供了一个可视化分析工具 Webpack Analyse（http://webpack.github.io/analyse/），它是一个在线 Web 应用。

打开 Webpack Analyse 链接的网页后，我们就会看到一个弹窗，提示我们上传 JSON 文件，也就是需要上传前面讲到的 `stats.json` 文件，如图 4-8 所示。

Webpack Analyse 不会将我们选择的 `stats.json` 文件发送给服务器，而是在浏览器本地解析，我们不用担心自己的代码为此而泄露。选择文件后，马上就能看到如图 4-9 所示的效果。

图 4-8　Webpack Analyse 上传文件弹窗

图 4-9　Webpack Analyse 主页

该主页被分为如下 6 大板块。

- **Modules**：展示所有模块，每个模块对应一个文件，并且包含所有模块之间的依赖关系图、模块路径、模块 ID、模块所属的 Chunk、模块的大小。
- **Chunks**：展示所有代码块，在一个代码块中包含多个模块，并且包含代码块的 ID、名称、大小、每个代码块包含的模块数量，以及代码块之间的依赖关系图。
- **Assets**：展示所有输出的文件资源，包括.js、.css、图片等，并且包括文件的名

称、大小及该文件来自哪个代码块。

- **Warnings**：展示构建过程中出现的所有警告信息。
- **Errors**：展示构建过程中出现的所有错误信息。
- **Hints**：展示处理每个模块所耗费的时间。

下面以在 3.10 节中使用的项目为例，分析其 `stats.json` 文件。

单击 **Modules**，查看模块的信息，效果如图 4-10 所示。

图 4-10　Webpack Analyse Modules

由于依赖了大量第三方模块，文件数量大，所以导致模块之间的依赖关系图太密集而无法看清，但我们可以进一步放大查看。

单击 **Chunks**，查看代码块的信息，效果如图 4-11 所示。

图 4-11 Webpack Analyse Chunks

通过代码块之间的依赖关系图，我们可以看到两个页面级的代码块 login 和 index 依赖提取出来的公共代码块 common。

单击 **Assets**，查看输出的文件资源，效果如图 4-12 所示。

assets	size	chunks	names
index.html	1050 bytes		
login.html	1050 bytes		
index_b3d3761c.js	1346 bytes	1	index
login_0a3feca9.js	1350 bytes	0	login
common_193d77a3.js	143 KiB	2	common
index_04c08fbf.css	25 bytes	1	index
login_e31e214b.css	26 bytes	0	login
common_7cc98ad0.css	13 bytes	2	common

图 4-12 Webpack Analyse Assets

单击 **Hints**，查看输出过程中的耗时分布，效果如图 4-13 所示。

module	name	time	finished ⊕
79	multi ./pages/index ./common.css	2 ms	2 ms
80	./pages/index/index.js	1002 ms	1004 ms
51	./node_modules/react-dom/index.js	127 ms	1131 ms
96	./node_modules/react-dom/lib/ReactDOM.js	14 ms	1145 ms
97	./node_modules/react-dom/lib/ReactDefaultInjection.js	55 ms	1200 ms
147	./node_modules/react-dom/lib/ReactReconcileTransaction.js	146 ms	1346 ms
74	./node_modules/react-dom/lib/ReactInputSelection.js	155 ms	1501 ms
150	./node_modules/fbjs/lib/containsNode.js	86 ms	1587 ms
151	./node_modules/fbjs/lib/isTextNode.js	23 ms	1610 ms
152	./node_modules/fbjs/lib/isNode.js	8 ms	1618 ms

图 4-13　Webpack Analyse Hints

从 Hints 可以看出每个文件在处理过程中的开始时间和结束时间，从而找出是哪个文件导致构建缓慢的。

4.15.2　webpack-bundle-analyzer

webpack-bundle-analyzer（https://www.npmjs.com/package/webpack-bundle-analyzer）是另一个可视化分析工具，它虽然没有 Webpack Analyse 那么多的功能，但比 Webpack Analyse 更直观。

先来看看它的效果图（见图 4-14）。

图 4-14 Webpack Analyse Assets

它能很方便地让我们知道：

- 打包出的文件中都包含了什么；
- 每个文件的尺寸在总体中的占比，让我们一眼看出哪些文件的尺寸大；
- 模块之间的包含关系；
- 每个文件的 Gzip 后的大小。

接入 webpack-bundle-analyzer 的方法很简单，步骤如下：

- 安装 webpack-bundle-analyzer 到全局，执行命令 `npm i -g webpack-bundle-analyzer`；
- 按照上面提到的方法生成 `stats.json` 文件；
- 在项目根目录中执行 `webpack-bundle-analyzer` 后，浏览器会打开对应的网页并展现以上效果。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-15 输出分析.zip。

4.16 优化总结

本章从开发体验和输出质量两个角度讲解了如何优化项目中的 Webpack 配置，这些优化方法都来自于项目实战中的经验积累。虽然每个小节都是一个个独立的优化方法，但是有些优化方法并不冲突且可以相互组合，以达到最佳的效果。

下面给出一个结合了本章所有优化方法的实例项目，由于构建速度和输出质量不能兼得，所以按照开发环境和线上环境为该项目配置了如下两份文件。

（1）配置侧重优化开发体验的文件 webpack.config.js：

```
const path = require('path');
const CommonsChunkPlugin =
require('webpack/lib/optimize/CommonsChunkPlugin');
const {AutoWebPlugin} = require('web-webpack-plugin');
const HappyPack = require('happypack');

// 自动寻找 pages 目录下的所有目录，将每个目录看作一个单页应用
const autoWebPlugin = new AutoWebPlugin('./src/pages', {
  // HTML 模板文件所在的文件路径
  template: './template.html',
  // 提取所有页面的公共代码
  commonsChunk: {
    // 提取公共代码 Chunk 的名称
    name: 'common',
  },
});

module.exports = {
  // AutoWebPlugin 会为寻找到的所有单页应用生成对应的入口配置，
  // 通过 autoWebPlugin.entry 方法可以获取生成入口的配置
  entry: autoWebPlugin.entry({
    // 这里可以加入我们额外需要的 Chunk 入口
    base: './src/base.js',
  }),
  output: {
    filename: '[name].js',
  },
```

```
  resolve: {
    // 使用绝对路径指明第三方模块存放的位置，以减少搜索步骤
    // 其中，__dirname 表示当前工作目录，也就是项目根目录
    modules: [path.resolve(__dirname, 'node_modules')],
    // 针对 Npm 中的第三方模块，优先采用 jsnext:main 中指向的 ES6 模块化语法的文件，
使用 Tree Shaking 优化
    // 只采用 main 字段作为入口文件描述字段，以减少搜索步骤
    mainFields: ['jsnext:main', 'main'],
  },
  module: {
    rules: [
      {
        // 如果项目源码中只有 js 文件，就不要写成/\.jsx?$/，以提升正则表达式的性能
        test: /\.js$/,
        // 使用 HappyPack 加速构建
        use: ['happypack/loader?id=babel'],
        // 只对项目根目录下 src 目录中的文件采用 babel-loader
        include: path.resolve(__dirname, 'src'),
      },
      {
        test: /\.js$/,
        use: ['happypack/loader?id=ui-component'],
        include: path.resolve(__dirname, 'src'),
      },
      {
        // 增加对 CSS 文件的支持
        test: /\.css/,
        use: ['happypack/loader?id=css'],
      },
    ]
  },
  plugins: [
    autoWebPlugin,
    // 使用 HappyPack 加速构建
    new HappyPack({
      id: 'babel',
      // babel-loader 支持缓存转换出的结果，通过 cacheDirectory 选项开启
      loaders: ['babel-loader?cacheDirectory'],
    }),
    new HappyPack({
```

```
        // UI 组件加载拆分
        id: 'ui-component',
        loaders: [{
          loader: 'ui-component-loader',
          options: {
            lib: 'antd',
            style: 'style/index.css',
            camel2: '-'
          }
        }],
      }),
      new HappyPack({
        id: 'css',
        // 如何处理.css 文件，用法和 Loader 配置中的一样
        loaders: ['style-loader', 'css-loader'],
      }),
      // 4.11 提取公共代码
      new CommonsChunkPlugin({
        // 从 common 和 base 两个现成的 Chunk 中提取公共的部分
        chunks: ['common', 'base'],
        // 将公共的部分放到 base 中
        name: 'base'
      }),
    ],
    watchOptions: {
      // 4.5 使用自动刷新：不监听的 node_modules 目录下的文件
      ignored: /node_modules/,
    }
  };
```

（2）配置侧重优化输出质量的文件 webpack-dist.config.js：

```
const path = require('path');
const DefinePlugin = require('webpack/lib/DefinePlugin');
const ModuleConcatenationPlugin =
require('webpack/lib/optimize/ModuleConcatenationPlugin');
const CommonsChunkPlugin =
require('webpack/lib/optimize/CommonsChunkPlugin');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
```

```
const {AutoWebPlugin} = require('web-webpack-plugin');
const HappyPack = require('happypack');
const ParallelUglifyPlugin = require('webpack-parallel-uglify-plugin');

// 自动寻找 pages 目录下的所有目录，将每个目录看作一个单页应用
const autoWebPlugin = new AutoWebPlugin('./src/pages', {
  // HTML 模板文件所在的文件路径
  template: './template.html',
  // 提取所有页面的公共代码
  commonsChunk: {
    // 提取公共代码 Chunk 的名称
    name: 'common',
  },
  // 指定存放 CSS 文件的 CDN 目录 URL
  stylePublicPath: '//css.cdn.com/id/',
});

module.exports = {
  // AutoWebPlugin 会为寻找到的所有单页应用生成对应的入口配置，
  // 通过 autoWebPlugin.entry 方法可以获取生成入口配置
  entry: autoWebPlugin.entry({
    // 这里可以加入我们额外需要的 Chunk 入口
    base: './src/base.js',
  }),
  output: {
    // 为输出的文件名称加上 Hash 值
    filename: '[name]_[chunkhash:8].js',
    path: path.resolve(__dirname, './dist'),
    // 指定存放 JavaScript 文件的 CDN 目录 URL
    publicPath: '//js.cdn.com/id/',
  },
  resolve: {
    // 使用绝对路径指明第三方模块存放的位置，以减少搜索步骤
    // 其中，__dirname 表示当前工作目录，也就是项目根目录
    modules: [path.resolve(__dirname, 'node_modules')],
    // 只采用 main 字段作为入口文件描述字段，以减少搜索步骤
    mainFields: ['jsnext:main', 'main'],
  },
  module: {
    rules: [
```

```
    {
      // 如果项目源码中只有 js 文件，就不要写成/\.jsx?$/，以提升正则表达式的性能
      test: /\.js$/,
      // 使用 HappyPack 加速构建
      use: ['happypack/loader?id=babel'],
      // 只对项目根目录下 src 目录中的文件采用 babel-loader
      include: path.resolve(__dirname, 'src'),
    },
    {
      test: /\.js$/,
      use: ['happypack/loader?id=ui-component'],
      include: path.resolve(__dirname, 'src'),
    },
    {
      // 增加对 CSS 文件的支持
      test: /\.css/,
      // 提取 Chunk 中的 CSS 代码到单独的文件中
      use: ExtractTextPlugin.extract({
        use: ['happypack/loader?id=css'],
        // 指定存放 CSS 中导入的资源（例如图片）的 CDN 目录 URL
        publicPath: '//img.cdn.com/id/'
      }),
    },
  ]
},
plugins: [
  autoWebPlugin,
  // 4.14 开启 ScopeHoisting
  new ModuleConcatenationPlugin(),
  // 4.3 使用 HappyPack
  new HappyPack({
    // 用唯一的标识符 id 来代表当前的 HappyPack 用来处理一类特定的文件
    id: 'babel',
    // babel-loader 支持缓存转换出的结果，通过 cacheDirectory 选项开启
    loaders: ['babel-loader?cacheDirectory'],
  }),
  new HappyPack({
    // UI 组件加载拆分
    id: 'ui-component',
    loaders: [{
```

```
    loader: 'ui-component-loader',
    options: {
      lib: 'antd',
      style: 'style/index.css',
      camel2: '-'
    }
  }],
},
new HappyPack({
  id: 'css',
  // 如何处理.css 文件，用法和 Loader 配置中的一样
  // 通过 minimize 选项压缩 CSS 代码
  loaders: ['css-loader?minimize'],
},
new ExtractTextPlugin({
  // 为输出的 CSS 文件名加上 Hash 值
  filename: `[name]_[contenthash:8].css`,
},
// 4-11 提取公共代码
new CommonsChunkPlugin({
  // 从 common 和 base 两个现成的 Chunk 中提取公共部分
  chunks: ['common', 'base'],
  // 将公共部分放到 base 中
  name: 'base'
},
new DefinePlugin({
  // 定义 NODE_ENV 环境变量为 production，去除在 react 代码中开发时才需要的部分
  'process.env': {
    NODE_ENV: JSON.stringify('production')
  }
},
// 使用 ParallelUglifyPlugin 并行压缩输出的 JavaScript 代码
new ParallelUglifyPlugin({
  // 传递给 UglifyJS 的参数
  uglifyJS: {
    output: {
      // 最紧凑的输出
      beautify: false,
      // 删除所有注释
      comments: false,
```

```
        },
        compress: {
          // 在 UglifyJS 删除没有用到的代码时不输出警告
          warnings: false,
          // 删除所有`console`语句，可以兼容 IE 浏览器
          drop_console: true,
          // 内嵌已定义但是只用到一次的变量
          collapse_vars: true,
          // 提取出现多次但是没有定义成变量去引用的静态值
          reduce_vars: true,
        }
      },
    }),
  ]
};
```

本章介绍的优化方法虽然难以涵盖 Webpack 的方方面面，但足以解决实战中的常见场景。对于本书没有介绍到的场景，我们需要根据自己的需求按照以下思路去优化：

- 找出问题的成因；
- 找出解决问题的方法；
- 找出解决方法对应的 Webpack 集成方案。

同时我们需要跟紧社区的步伐，学习他人的优化方法，了解最新的 Webpack 特性和新涌现出的插件、Loader。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/4-16 优化总结.zip。

第5章 原理

虽然通过前 4 章的学习，我们已经能用 Webpack 解决常见的问题了，但当在实战中遇到比较特殊的需求及在社区中找不到解决方案时，我们需要编写自己的 Loader 或 Plugin。要做到这一点，我们需要先了解 Webpack 的工作原理，这样我们才能对 Webpack 有更深的认识，使用它时更得心应手。

本章包含以下内容：

- 了解 Webpack 的整体架构和工作流程，明白是通过 Loader 还是 Plugin 去实现某个功能，见 5.1 ~ 5.2 节；
- 如何开发、调试 Loader 和 Plugin，见 5.3 ~ 5.5 节；
- 对本章的总结，见 5.6 节。

5.1 工作原理概括

Webpack 以其使用方法简单著称，在使用它的过程中，使用者只需将它当作一个黑盒，只需关心它暴露出来的配置。本节将带我们走进这个黑盒，看看 Webpack 是如何运行的。

5.1.1 基本概念

在了解 Webpack 原理前，我们需要掌握以下几个核心概念，以方便后面的理解。

- **Entry**：入口，Webpack 执行构建的第一步将从 Entry 开始，可抽象成输入。
- **Module**：模块，在 Webpack 里一切皆模块，一个模块对应一个文件。Webpack 会从配置的 Entry 开始，递归找出所有依赖的模块。
- **Chunk**：代码块，一个 Chunk 由多个模块组合而成，用于代码合并与分割。
- **Loader**：模块转换器，用于将模块的原内容按照需求转换成新内容。
- **Plugin**：扩展插件，在 Webpack 构建流程中的特定时机会广播对应的事件，插件可以监听这些事件的发生，在特定的时机做对应的事情。

5.1.2 流程概括

Webpack 的运行流程是一个串行的过程，从启动到结束会依次执行以下流程。

- 初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数。
- 开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，通过执行对象的 run 方法开始执行编译。
- 确定入口：根据配置中的 entry 找出所有入口文件。

- 编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理。
- 完成模块编译：在经过第 4 步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容及它们之间的依赖关系。
- 输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再将每个 Chunk 转换成一个单独的文件加入输出列表中，这是可以修改输出内容的最后机会。
- 输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，将文件的内容写入文件系统中。

在以上过程中，Webpack 会在特定的时间点广播特定的事件，插件在监听到感兴趣的事件后会执行特定的逻辑，并且插件可以调用 Webpack 提供的 API 改变 Webpack 的运行结果。

5.1.3 流程细节

Webpack 的构建流程可以分为以下三大阶段。

- 初始化：启动构建，读取与合并配置参数，加载 Plugin，实例化 Compiler。
- 编译：从 Entry 发出，针对每个 Module 串行调用对应的 Loader 去翻译文件的内容，再找到该 Module 依赖的 Module，递归地进行编译处理。
- 输出：将编译后的 Module 组合成 Chunk，将 Chunk 转换成文件，输出到文件系统中。

如果只执行一次构建，则以上阶段将会按照顺序各执行一次。但在开启监听模式下，流程将变为如图 5-1 所示。

图 5-1　监听模式的构建流程

在每个大阶段中又会发生很多事件，Webpack 会将这些事件广播出来供 Plugin 使用，下面进行一一介绍。

1. 初始化阶段

初始化阶段会发生的事件及解释如表 5-1 所示。

表 5-1　初始化阶段会发生的事件及解释

事 件 名	解　　释
初始化参数	从配置文件和 Shell 语句中读取与合并参数，得出最终的参数。在这个过程中还会执行配置文件中的插件实例化语句 new Plugin()
实例化 Compiler	用上一步得到的参数初始化 Compiler 实例，Compiler 负责文件监听和启动编译。在 Compiler 实例中包含了完整的 Webpack 配置，全局只有一个 Compiler 实例
加载插件	依次调用插件的 apply 方法，让插件可以监听后续的所有事件节点。同时向插件传入 compiler 实例的引用，以方便插件通过 compiler 调用 Webpack 提供的 API
environment	开始应用 Node.js 风格的文件系统到 compiler 对象，以方便后续的文件寻找和读取
entry-option	读取配置的 Entrys，为每个 Entry 实例化一个对应的 EntryPlugin，为后面该 Entry 的递归解析工作做准备
after-plugins	调用完所有内置的和配置的插件的 apply 方法
after-resolvers	根据配置初始化 resolver，resolver 负责在文件系统中寻找指定路径的文件

2. 编译阶段

编译阶段会发生的事件及解释如表 5-2 所示。

表 5-2 编译阶段会发生的事件及解释

事件名	解 释
run	启动一次新的编译
watch-run	和 run 类似，区别在于它是在监听模式下启动编译，在这个事件中可以获取是哪些文件发生了变化从而导致重新启动一次新的编译
compile	该事件是为了告诉插件一次新的编译将要启动，同时会给插件带上 compiler 对象
compilation	当 Webpack 以开发模式运行时，每当检测到文件的变化，便有一次新的 Compilation 被创建。一个 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。Compilation 对象也提供了很多事件回调给插件进行扩展
make	一个新的 Compilation 创建完毕，即将从 Entry 开始读取文件，根据文件的类型和配置的 Loader 对文件进行编译，编译完后再找出该文件依赖的文件，递归地编译和解析
after-compile	一次 Compilation 执行完成
invalid	当遇到文件不存在、文件编译错误等异常时会触发该事件，该事件不会导致 Webpack 退出

在编译阶段中，最重要的事件是 compilation，因为在 compilation 阶段调用了 Loader，完成了每个模块的转换操作。在 compilation 阶段又会发生很多小事件，如表 5-3 所示。

表 5-3 compilation 阶段会发生的小事件

事件名	解 释
build-module	使用对应的 Loader 去转换一个模块
normal-module-loader	在用 Loader 转换完一个模块后，使用 acorn 解析转换后的内容，输出对应的抽象语法树（AST），以方便 Webpack 在后面对代码进行分析
program	从配置的入口模块开始，分析其 AST，当遇到 require 等导入其他模块的语句时，便将其加入依赖的模块列表中，同时对新找出的依赖模块递归分析，最终弄清楚所有模块的依赖关系
seal	所有模块及其依赖的模块都通过 Loader 转换完成，根据依赖关系开始生成 Chunk

3. 输出阶段

输出阶段会发生的事件及解释如表 5-4 所示。

表 5-4 输出阶段会发生的事件及解释

事件名	解释
should-emit	所有需要输出的文件已经生成，询问插件有哪些文件需要输出，有哪些不需要输出
emit	确定好要输出哪些文件后，执行文件输出，可以在这里获取和修改输出的内容
after-emit	文件输出完毕
done	成功完成一次完整的编译和输出流程
failed	如果在编译和输出的流程中遇到异常，导致 Webpack 退出，就会直接跳转到本步骤，插件可以在本事件中获取具体的错误原因

在输出阶段已经得到了各个模块经过转换后的结果和其依赖关系，并且将相关模块组合在一起形成一个个 Chunk。在输出阶段会根据 Chunk 的类型，使用对应的模板生成最终要要输出的文件内容。

至于如何将 Chunk 输出为具体的文件，可以参考 5.2 节。

5.2 输出文件分析

虽然在前面的章节中我们学会了如何使用 Webpack，也大致知道其工作原理，可是 Webpack 输出的 bundle.js 是什么样子的呢？为什么原来一个个的模块文件被合并成了一个单独的文件？为什么 bundle.js 能直接运行在浏览器中？本节将解释以上问题。

先来看看由 1.3 节中最简单的项目构建出的 bundle.js 文件的内容，代码如下：

```
(
    // webpackBootstrap 启动函数
    // modules 即存放所有模块的数组，数组中的每个元素都是一个函数
    function (modules) {
        // 安装过的模块都存放在这里面
        // 作用是将已经加载过的模块缓存在内存中，提升性能
        var installedModules = {};

        // 去数组中加载一个模块，moduleId 为要加载模块在数组中的 index
        // 作用和 Node.js 中的 require 语句相似
        function __webpack_require__(moduleId) {
```

```
            // 如果需要加载的模块已经被加载过，就直接从缓存中返回
            if (installedModules[moduleId]) {
                return installedModules[moduleId].exports;
            }

            // 如果缓存中不存在需要加载的模块，就新建一个模块，并将它存在缓存中
            var module = installedModules[moduleId] = {
                // 模块在数组中的 index
                i: moduleId,
                // 该模块是否已经加载完毕
                l: false,
                // 该模块的导出值
                exports: {}
            };

            // 从 modules 中获取 index 为 moduleId 的模块对应的函数
            // 再调用这个函数，同时将函数需要的参数传入
            modules[moduleId].call(module.exports, module, module.
exports, __webpack_require__);
            // 将这个模块标记为已加载
            module.l = true;
            // 返回这个模块的导出值
            return module.exports;
        }

        // Webpack 配置中的 publicPath，用于加载被分割出去的异步代码
        __webpack_require__.p = "";

        // 使用__webpack_require__去加载 index 为 0 的模块，并且返回该模块导出的
内容
        // index 为 0 的模块就是 main.js 对应的文件，也就是执行入口模块
        // __webpack_require__.s 的含义是启动模块对应的 index
        return __webpack_require__(__webpack_require__.s = 0);

    })(

    // 所有的模块都存放在一个数组里，根据每个模块在数组的 index 来区分和定位模块
    [
        /* 0 */
        (function (module, exports, __webpack_require__) {
```

```
            // 通过__webpack_require__规范导入show函数，show.js对应的模块
index为1
            const show = __webpack_require__(1);
            // 执行show函数
            show('Webpack');
        }),
        /* 1 */
        (function (module, exports) {
            function show(content) {
                window.document.getElementById('app').innerText =
'Hello,' + content;
            }
            // 通过CommonJS规范导出show函数
            module.exports = show;
        })
    ]
);
```

以上看上去复杂的代码其实是一个立即执行函数，可以简写如下：

```
(function(modules) {

  // 模拟require语句
  function __webpack_require__() {
  }

  // 执行存放所有模块数组中的第0个模块
  __webpack_require__(0);

})([/*存放所有模块的数组*/])
```

bundle.js 能直接运行在浏览器中的原因是，在输出的文件中通过__webpack_require__函数，定义了一个可以在浏览器中执行的加载函数，来模拟 Node.js 中的 require 语句。

原来一个个独立的模块文件被合并到了一个单独的 bundle.js 的原因是，浏览器不能像 Node.js 那样快速地在本地加载一个个模块文件，而必须通过网络请求去加载还未得到的文件。如果模块的数量很多，则加载时间会很长，因此将所有模块都存放在了数组中，执行一次网络加载。

仔细分析__webpack_require__函数的实现，我们还会发现 Webpack 做了缓存优化：执行加载过的模块不会再执行第 2 次，执行结果会缓存在内存中，当某个模块第 2 次被访问时会直接去内存中读取被缓存的返回值。

在采用了 4.2 节中介绍过的优化方法时，Webpack 的输出文件会发生变化。

例如，将源码中的 main.js 修改如下：

```
// 异步加载 show.js
import('./show').then((show) => {
  // 执行 show 函数
  show('Webpack');
});
```

重新构建后会输出两个文件，分别是执行入口文件 bundle.js 和异步加载文件 0.bundle.js。

其中 0.bundle.js 的内容如下：

```
// 加载在本文件（0.bundle.js）中包含的模块
webpackJsonp(
  // 在其他文件中存放的模块的 ID
  [0],
  // 本文件所包含的模块
  [
    // show.js 所对应的模块
    (function (module, exports) {
      function show(content) {
        window.document.getElementById('app').innerText = 'Hello,' +
content;
      }

      module.exports = show;
    })
  ]
);
```

bundle.js 的内容如下：

```
(function (modules) {
  /***
```

```
     * webpackJsonp 用于从异步加载的文件中安装模块。
     * 将 webpackJsonp 挂载到全局是为了方便在其他文件中调用。
     *
     * @param chunkIds 异步加载的文件中存放的需要安装的模块对应的 Chunk ID
     * @param moreModules 异步加载的文件中存放的需要安装的模块列表
     * @param executeModules 在异步加载的文件中存放的需要安装的模块都安装成功后，
需要执行的模块对应的 index
     */
    window["webpackJsonp"] = function webpackJsonpCallback(chunkIds,
moreModules, executeModules) {
      // 将 moreModules 添加到 modules 对象中
      // 将所有 chunkIds 对应的模块都标记成已经加载成功
      var moduleId, chunkId, i = 0, resolves = [], result;
      for (; i < chunkIds.length; i++) {
        chunkId = chunkIds[i];
        if (installedChunks[chunkId]) {
          resolves.push(installedChunks[chunkId][0]);
        }
        installedChunks[chunkId] = 0;
      }
      for (moduleId in moreModules) {
        if (Object.prototype.hasOwnProperty.call(moreModules, moduleId)) {
          modules[moduleId] = moreModules[moduleId];
        }
      }
      while (resolves.length) {
        resolves.shift()();
      }
    };

    // 缓存已经安装的模块
    var installedModules = {};

    // 存储每个 Chunk 的加载状态；
    // 键为 Chunk 的 ID，值为 0 代表已经加载成功
    var installedChunks = {
      1: 0
    };

    // 模拟 require 语句，和上面介绍的一致
```

```
function __webpack_require__(moduleId) {
  // ... 省略和上面一样的内容
}

/**
 * 用于加载被分割出去的需要异步加载的 Chunk 对应的文件
 * @param chunkId 需要异步加载的 Chunk 对应的 ID
 * @returns {Promise}
 */
__webpack_require__.e = function requireEnsure(chunkId) {
  // 从上面定义的 installedChunks 中获取 chunkId 对应的 Chunk 的加载状态
  var installedChunkData = installedChunks[chunkId];
  // 如果加载状态为 0，则表示该 Chunk 已经加载成功了，直接返回 resolve Promise
  if (installedChunkData === 0) {
    return new Promise(function (resolve) {
      resolve();
    });
  }

  // installedChunkData 不为空且不为 0 时，表示该 Chunk 正在网络加载中
  if (installedChunkData) {
    // 返回存放在 installedChunkData 数组中的 Promise 对象
    return installedChunkData[2];
  }

  // installedChunkData 为空，表示该 Chunk 还没有加载过，去加载该 Chunk 对应的
文件
  var promise = new Promise(function (resolve, reject) {
    installedChunkData = installedChunks[chunkId] = [resolve, reject];
  });
  installedChunkData[2] = promise;

  // 通过 DOM 操作，向 HTML head 中插入一个 script 标签去异步加载 Chunk 对应的
JavaScript 文件
  var head = document.getElementsByTagName('head')[0];
  var script = document.createElement('script');
  script.type = 'text/javascript';
  script.charset = 'utf-8';
  script.async = true;
  script.timeout = 120000;
```

```
        // 文件的路径由配置的 publicPath、chunkId 拼接而成
        script.src = __webpack_require__.p + "" + chunkId + ".bundle.js";

        // 设置异步加载的最长超时时间
        var timeout = setTimeout(onScriptComplete, 120000);
        script.onerror = script.onload = onScriptComplete;

        // 在 script 加载和执行完成时回调
        function onScriptComplete() {
          // 防止内存泄露
          script.onerror = script.onload = null;
          clearTimeout(timeout);

          // 去检查 chunkId 对应的 Chunk 是否安装成功，安装成功时才会存在于
installedChunks 中
          var chunk = installedChunks[chunkId];
          if (chunk !== 0) {
            if (chunk) {
chunk[1](/Users/halwu/WebstormProjects/dive-into-webpack/docs/5 原理/new
Error('Loading chunk ' + chunkId + ' failed.'));
            }
            installedChunks[chunkId] = undefined;
          }
        };
        head.appendChild(script);

        return promise;
      };

      // 加载并执行入口模块，和上面介绍的一致
      return __webpack_require__(__webpack_require__.s = 0);
    })
    (
      // 存放所有没有经过异步加载的，随着执行入口文件加载的模块
      [
        // main.js 对应的模块
        (function (module, exports, __webpack_require__) {
          // 通过__webpack_require__.e 异步加载 show.js 对应的 Chunk
```

```
        __webpack_require__.e(0).then(__webpack_require__.bind(null,
1)).then((show) => {
          // 执行 show 函数
          show('Webpack');
        });
      })
    ]
  );
```

这里的 bundle.js 和上面所讲的 bundle.js 非常相似，区别在于：

- 多了一个 __webpack_require__.e，用于加载被分割出去的需要异步加载的 Chunk 对应的文件；
- 多了一个 webpackJsonp 函数，用于从异步加载的文件中安装模块。

使用 CommonsChunkPlugin 提取公共代码时输出的文件和使用异步加载时输出的文件是一样的，都会有 __webpack_require__.e 和 webpackJsonp。原因在于提取公共代码和异步加载在本质上都是代码分割。

本实例提供项目的完整代码，参见 http://webpack.wuhaolin.cn/5-2 输出文件分析.zip。

5.3 编写 Loader

Loader 就像一个翻译员，能将源文件经过转化后输出新的结果，并且一个文件还可以链式地经过多个翻译员翻译。

以处理 SCSS 文件为例：

- 先将 SCSS 源代码提交给 sass-loader，将 SCSS 转换成 CSS；
- 将 sass-loader 输出的 CSS 提交给 css-loader 处理，找出 CSS 中依赖的资源、压缩 CSS 等；
- 将 css-loader 输出的 CSS 提交给 style-loader 处理，转换成通过脚本加载的 JavaScript 代码。

可以看出，以上处理过程需要有顺序地链式执行，先 sass-loader，再 css-loader，再 style-loader。以上处理的 Webpack 的相关配置如下：

```
module.exports = {
  module: {
    rules: [
      {
        // 增加对 SCSS 文件的支持
        test: /\.scss/,
        // SCSS 文件的处理顺序为先 sass-loader，再 css-loader，再 style-loader
        use: [
          'style-loader',
          {
            loader:'css-loader',
            // 向 css-loader 传入配置项
            options:{
              minimize:true,
            }
          },
          'sass-loader'],
      },
    ]
  },
};
```

5.3.1 Loader 的职责

由上面的例子可以看出，一个 Loader 的职责是单一的，只需要完成一种转换。如果一个源文件需要经历多步转换才能正常使用，就通过多个 Loader 去转换。在调用多个 Loader 去转换一个文件时，每个 Loader 都会链式地顺序执行。第 1 个 Loader 将会拿到需处理的原内容，上一个 Loader 处理后的结果会被传给下一个 Loader 接着处理，最后的 Loader 将处理后的最终结果返回给 Webpack。

所以，在开发一个 Loader 时，请保持其职责的单一性，我们只需关心输入和输出。

5.3.2 Loader 基础

Webpack 是运行在 Node.js 上的，一个 Loader 其实就是一个 Node.js 模块，这个模块需要导出一个函数。这个导出的函数的工作就是获得处理前的原内容，对原内容执行处理后，返回处理后的内容。

一个最简单的 Loader 的源码如下：

```
module.exports = function(source) {
  // source 为 compiler 传递给 Loader 的一个文件的原内容
  // 该函数需要返回处理后的内容，这里为了简单起见，直接将原内容返回了，相当于该
Loader 没有做任何转换
  return source;
};
```

由于 Loader 运行在 Node.js 中，所以我们可以调用任意 Node.js 自带的 API，或者安装第三方模块进行调用：

```
const sass = require('node-sass');
module.exports = function(source) {
  return sass(source);
};
```

5.3.3 Loader 进阶

以上只是一个最简单的 Loader，Webpack 还提供了一些 API 供 Loader 调用，下面进行一一介绍。

1. 获得 Loader 的 options

在最上面处理 SCSS 文件的 Webpack 配置中，将 options 参数传给了 css-loader，以控制 css-loader。如何在自己编写的 Loader 中获取用户传入的 options 呢？需要这样做：

```
const loaderUtils = require('loader-utils');
module.exports = function(source) {
  // 获取用户为当前 Loader 传入的 options
```

```
  const options = loaderUtils.getOptions(this);
  return source;
};
```

2. 返回其他结果

上面的 Loader 都只是返回了原内容转换后的内容，但在某些场景下还需要返回除了内容之外的东西。

以用 babel-loader 转换 ES6 代码为例，它还需要输出转换后的 ES5 代码对应的 Source Map，以方便调试源码。为了将 Source Map 也一起随着 ES5 代码返回给 Webpack，还可以这样写：

```
module.exports = function(source) {
  // 通过 this.callback 告诉 Webpack 返回的结果
  this.callback(null, source, sourceMaps);
  // 当我们使用 this.callback 返回内容时，该 Loader 必须返回 undefined，
  // 以让 Webpack 知道该 Loader 返回的结果在 this.callback 中，而不是 return 中
  return;
};
```

其中的 this.callback 是 Webpack 向 Loader 注入的 API，以方便 Loader 和 Webpack 之间通信。this.callback 的详细使用方法如下：

```
this.callback(
    // 当无法转换原内容时，为 Webpack 返回一个 Error
    err: Error | null,
    // 原内容转换后的内容
    content: string | Buffer,
    // 用于通过转换后的内容得出原内容的 Source Map，以方便调试
    sourceMap?: SourceMap,
    // 如果本次转换为原内容生成了 AST 语法树，则可以将这个 AST 返回，
    // 以方便之后需要 AST 的 Loader 复用该 AST，避免重复生成 AST，提升性能
    abstractSyntaxTree?: AST
);
```

Source Map 的生成很耗时，通常在开发环境下才会生成 Source Map，在其他环境下不用生成，以加速构建。因此，Webpack 为 Loader 提供了 this.sourceMap API 去告诉 Loader 在当前构建环境下用户是否需要 Source Map。如果我们编写的 Loader 会生成 Source Map，

则请考虑这一点。

3. 同步与异步

Loader 有同步和异步之分，上面介绍的 Loader 都是同步的 Loader，因为它们的转换流程都是同步的，转换完成后再返回结果。但在某些场景下转换的步骤只能是异步完成的，例如我们需要通过网络请求才能得出结果，如果采用同步的方式，则网络请求会阻塞整个构建，导致构建非常缓慢。

如果是异步转换，则我们可以这样做：

```
module.exports = function(source) {
    // 告诉 Webpack 本次转换是异步的，Loader 会在 callback 中回调结果
    var callback = this.async();
    someAsyncOperation(source, function(err, result, sourceMaps, ast) {
        // 通过 callback 返回异步执行后的结果
        callback(err, result, sourceMaps, ast);
    });
};
```

4. 处理二进制数据

在默认情况下，Webpack 传给 Loader 的原内容都是 UTF-8 格式编码的字符串。但在某些场景下 Loader 不会处理文本文件，而会处理二进制文件如 file-loader，这时就需要 Webpack 为 Loader 传入二进制格式的数据。为此，我们需要这样编写 Loader：

```
module.exports = function(source) {
    // 在 exports.raw === true 时，Webpack 传给 Loader 的 source 是 Buffer 类型的
    source instanceof Buffer === true;
    // Loader 返回的类型也可以是 Buffer 类型的
    // 在 exports.raw !== true 时，Loader 也可以返回 Buffer 类型的结果
    return source;
};
// 通过 exports.raw 属性告诉 Webpack 该 Loader 是否需要二进制数据
module.exports.raw = true;
```

在以上代码中最关键的代码是最后一行 `module.exports.raw = true;`，若没有该

行代码，则 Loader 只能拿到字符串。

5. 缓存加速

在某些情况下，有些转换操作需要大量的计算，非常耗时，如果每次构建都重新执行重复的转换操作，则构建将会变得非常缓慢。为此，Webpack 会默认缓存所有 Loader 的处理结果，也就是说在需要被处理的文件或者其依赖的文件没有发生变化时，是不会重新调用对应的 Loader 去执行转换操作的。

如果我们不想让 Webpack 不缓存该 Loader 的处理结果，则可以这样：

```
module.exports = function(source) {
  // 关闭该 Loader 的缓存功能
  this.cacheable(false);
  return source;
};
```

5.3.4 其他 Loader API

除了以上提到的在 Loader 中能调用的 Webpack API，还存在以下常用的 API。

- `this.context`：当前处理的文件所在的目录，假如当前 Loader 处理的文件是 `/src/main.js`，则 `this.context` 等于`/src`。
- `this.resource`：当前处理的文件的完整请求路径，包括 querystring，例如`/src/main.js?name=1`。
- `this.resourcePath`：当前处理的文件的路径，例如`/src/main.js`。
- `this.resourceQuery`：当前处理的文件的 querystring。
- `this.target`：等于 Webpack 配置中的 Target，具体内容请参见 2.7 节。
- `this.loadModule`：但 Loader 在处理一个文件时，如果依赖其他文件的处理结果才能得出当前文件的结果，就可以通过 `this.loadModule(request: string, callback: function(err, source, sourceMap, module))` 去获取 request

对应的文件的处理结果。

- this.resolve：像 require 语句一样获得指定文件的完整路径，使用方法为 resolve(context: string, request: string, callback: function (err, result: string))。
- this.addDependency：为当前处理的文件添加其依赖的文件，以便其依赖的文件发生发生变化时，重新调用 Loader 处理该文件。使用方法为 addDependency (file: string)。
- this.addContextDependency：和 addDependency 类似，但 addContext Dependency 是将整个目录加入当前正在处理的文件的依赖中。使用方法为 add ContextDependency(directory: string)。
- this.clearDependencies：清除当前正在处理的文件的所有依赖，使用方法为 clearDependencies()。
- this.emitFile：输出一个文件，使用方法为 emitFile(name: string, content: Buffer|string, sourceMap: {...})。

对于其他没有提到的 API，可以在 Webpack 官网（https://webpack.js.org/api/loaders/）查看。

5.3.5 加载本地 Loader

在开发 Loader 的过程中，为了测试编写的 Loader 能否正常工作，需要将它配置到 Webpack 中，才可能会调用该 Loader。在前面的章节中使用的 Loader 都是通过 Npm 安装的，在使用 Loader 时会直接使用 Loader 的名称，代码如下：

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css/,
        use: ['style-loader'],
      },
    ]
```

```
  },
};
```

如果还采取以上方法去使用本地开发的 Loader，将会很麻烦，因为我们需要确保编写的 Loader 的源码在 node_modules 目录下。为此需要先将编写的 Loader 发布到 Npm 仓库，再安装到本地项目中使用。

解决以上问题的便捷方法有如下两种。

1. Npm link

Npm link 专门用于开发和调试本地的 Npm 模块，能做到在不发布模块的情况下，将本地的一个正在开发的模块的源码链接到项目的 node_modules 目录下，让项目可以直接使用本地的 Npm 模块。由于是通过软链接的方式实现的，编辑了本地的 Npm 模块的代码，所以在项目中也能使用到编辑后的代码。

完成 Npm link 的步骤如下：

- 确保正在开发的本地 Npm 模块（也就是正在开发的 Loader）的 package.json 已经正确配置好；
- 在本地的 Npm 模块根目录下执行 npm link，将本地模块注册到全局；
- 在项目根目录下执行 npm link loader-name，将第 2 步注册到全局的本地 Npm 模块链接到项目的 node_moduels 下，其中的 loader-name 是指在第 1 步的 package.json 文件中配置的模块名称。

链接好 Loader 到项目后我们就可以像使用一个真正的 Npm 模块一样使用本地的 Loader 了。

2. ResolveLoader

在 2.7 节中曾介绍过，ResolveLoader 用于配置 Webpack 如何寻找 Loader，它在默认情况下只会去 node_modules 目录下寻找。为了让 Webpack 加载放在本地项目中的 Loader，需要修改 resolveLoader.modules。

假如本地项目中的 Loader 在项目目录的./loaders/loader-name 下，则需要如下

配置：

```
module.exports = {
  resolveLoader:{
    // 去哪些目录下寻找 Loader，有先后顺序之分
    modules: ['node_modules','./loaders/'],
  }
}
```

加上以上配置后，Webpack 会先去 node_modules 项目下寻找 Loader，如果找不到，则再去./loaders/目录下寻找。

5.3.6 实战

上面讲了许多理论，接下来从实际出发，编写一个解决实际问题的 Loader。

该 Loader 名叫 comment-require-loader，作用是将在 JavaScript 代码中的注释语法：

```
// @require '../style/index.css'
```

转换成：

```
require('../style/index.css');
```

该 Loader 的使用场景是正确加载针对 Fis3（http://fis.baidu.com/fis3/docs/user-dev/require.html）编写的 JavaScript，这些 JavaScript 中存在通过注释的方式加载依赖的 CSS 文件。

该 Loader 的使用方法如下：

```
module.exports = {
  module: {
    loaders: [
      {
        test: /\.js$/,
        loaders: ['comment-require-loader'],
        // 针对采用了 fis3 CSS 导入语法的 JavaScript 文件，通过 comment-require-loader 转换
        include: [path.resolve(__dirname, 'node_modules/imui')]
      }
```

```
    ]
  }
};
```

该 Loader 的实现非常简单，完整代码如下：

```
function replace(source) {
    // 使用正则表达式将// @require '../style/index.css'转换成
require('../style/index.css');
    return source.replace(/(\/\/ *@require) +(('|").+('|")).*/,
'require($2);');
}

module.exports = function (content) {
    return replace(content);
};
```

本实例提供项目的完整代码，参见 https://github.com/gwuhaolin/comment-require-loader。

5.4 编写 Plugin

Webpack 通过 Plugin 机制让其更灵活，以适应各种应用场景。在 Webpack 运行的生命周期中会广播许多事件，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。

一个最基础的 Plugin 的代码是这样的：

```
class BasicPlugin{
  // 在构造函数中获取用户为该插件传入的配置
  constructor(options){
  }

  // Webpack 会调用 BasicPlugin 实例的 apply 方法为插件实例传入 compiler 对象
  apply(compiler){
    compiler.plugin('compilation',function(compilation) {
    })
  }
```

```
}

// 导出 Plugin
module.exports = BasicPlugin;
```

在使用这个 Plugin 时，相关的配置代码如下：

```
const BasicPlugin = require('./BasicPlugin.js');
module.export = {
  plugins:[
    new BasicPlugin(options),
  ]
}
```

Webpack 启动后，在读取配置的过程中会先执行 new BasicPlugin(options)，初始化一个 BasicPlugin 并获得其实例。在初始化 compiler 对象后，再调用 basicPlugin.apply(compiler)为插件实例传入 compiler 对象。插件实例在获取到 compiler 对象后，就可以通过 compiler.plugin（事件名称，回调函数）监听到 Webpack 广播的事件，并且可以通过 compiler 对象去操作 Webpack。

通过以上最简单的 Plugin，相信我们大概明白了 Plugin 的工作原理，但在实际开发中还有很多细节需要注意，下面进行详细介绍。

5.4.1 Compiler 和 Compilation

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 环境的所有配置信息，包含 options、loaders、plugins 等信息。这个对象在 Webpack 启动时被实例化，它是全局唯一的，可以简单地将它理解为 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发生变化，便有一次新的 Compilation 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

也能读取到 Compiler 对象。

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

5.4.2 事件流

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，多个流程之间存在依赖关系，只有在完成当前处理后才能提交给下一个流程去处理。插件就像插入生产线中的某个功能，在特定的时机对生产线上的资源进行处理。

Webpack 通过 Tapable（https://github.com/webpack/tapable）来组织这条复杂的生产线。Webpack 在运行的过程中会广播事件，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，使得整个系统的扩展性良好。

Webpack 的事件流机制应用了观察者模式，和 Node.js 中的 EventEmitter 非常相似。Compiler 和 Compilation 都继承自 Tapable，可以直接在 Compiler 和 Compilation 对象上广播和监听事件，方法如下：

```
/**
* 广播事件
* event-name 为事件名称，注意不要和现有的事件重名
* params 为附带的参数
*/
compiler.apply('event-name',params);

/**
* 监听名称为 event-name 的事件，当 event-name 事件发生时，函数就会被执行。
* 同时函数中的 params 参数为广播事件时附带的参数。
*/
compiler.plugin('event-name',function(params) {

});
```

同理，compilation.apply 和 compilation.plugin 的使用方法和前面讲解一致。

在开发插件时，我们可能会不知道如何下手，因为不知道该监听哪个事件才能完成任务。

在开发插件时，还需要注意以下两点：只要能拿到 Compiler 或 Compilation 对象，就能广播新的事件，所以在新开发的插件中也能广播事件，为其他插件监听使用。传给每个插件的 Compiler 和 Compilation 对象都是同一个引用。也就是说，若在一个插件中修改了 Compiler 或 Compilation 对象上的属性，就会影响到后面的插件。有些事件是异步的，这些异步的事件会附带两个参数，第 2 个参数为回调函数，在插件处理完任务时需要调用回调函数通知 Webpack，才会进入下一个处理流程。例如：

```
```js compiler.plugin('emit',function(compilation, callback) { // 支持处理逻辑
    // 处理完毕后执行 callback 以通知 Webpack
    // 如果不执行 callback，运行流程将会一直卡在这里而不往后执行
    callback();
  });
```
```

5.4.3 常用的 API

插件可以用来修改输出文件和增加输出文件，甚至可以提升 Webpack 的性能，等等。总之，插件可以通过调用 Webpack 提供的 API 完成很多事情。由于 Webpack 提供的 API 非常多，有很多 API 很少用得上，所以本节只介绍一些常用的 API。

1. 读取输出资源、代码块、模块及其依赖

某些插件可能需要读取 Webpack 的处理结果，例如输出资源、代码块、模块及其依赖，以便做下一步处理。

emit 事件发生时，代表源文件的转换和组装已经完成，在这里可以读取到最终将输出的资源、代码块、模块及其依赖，并且可以修改输出资源的内容。插件的代码如下：

```
class Plugin {
  apply(compiler) {
```

```
compiler.plugin('emit', function (compilation, callback) {
  // compilation.chunks 存放所有代码块，是一个数组
  compilation.chunks.forEach(function (chunk) {
    // chunk 代表一个代码块
    // 代码块由多个模块组成，通过 chunk.forEachModule 能读取代码块的每个模块
    chunk.forEachModule(function (module) {
      // module 代表一个模块
      // module.fileDependencies 存放当前模块的所有依赖的文件路径，是一个数组
      module.fileDependencies.forEach(function (filepath) {
      });
    });

    // Webpack 会根据 Chunk 生成输出的文件资源，每个 Chunk 都对应一个及以上的
输出文件
    // 例如在 Chunk 中包含 CSS 模块并且使用了 ExtractTextPlugin 时，
    // 该 Chunk 就会生成.js 和.css 两个文件
    chunk.files.forEach(function (filename) {
      // compilation.assets 存放当前即将输出的所有资源
      // 调用一个输出资源的 source()方法能获取输出资源的内容
      let source = compilation.assets[filename].source();
    });
  });

  // 这是一个异步事件，要记得调用 callback 来通知 Webpack 本次事件监听处理结束
  // 如果忘记了调用 callback，则 Webpack 将一直卡在这里而不会往后执行
  callback();
})
}
}
```

2. 监听文件的变化

在 4.5 节中介绍过，Webpack 会从配置的入口模块出发，依次找出所有依赖模块，当入口模块或者其依赖的模块发生变化时，就会触发一次新的 Compilation。

在开发插件时经常需要知道是哪个文件发生的变化导致了新的 Compilation，为此可以使用如下代码：

```
// 当依赖的文件发生变化时会触发 watch-run 事件
```

```
compiler.plugin('watch-run', (watching, callback) => {
    // 获取发生变化的文件列表
    const changedFiles =
watching.compiler.watchFileSystem.watcher.mtimes;
    // changedFiles 格式为键值对，键为发生变化的文件路径。
    if (changedFiles[filePath] !== undefined) {
      // filePath 对应的文件发生了变化
    }
    callback();
});
```

在默认情况下，Webpack 只会监视入口和其依赖的模块是否发生了变化，在某些情况下项目可能需要引入新的文件，例如引入一个 HTML 文件。由于 JavaScript 文件不会导入 HTML 文件，所以 Webpack 不会监听 HTML 文件的变化，编辑 HTML 文件时就不会重新触发新的 Compilation。为了监听 HTML 文件的变化，我们需要将 HTML 文件加入依赖列表中，为此可以使用如下代码：

```
compiler.plugin('after-compile', (compilation, callback) => {
  // 将 HTML 文件添加到文件依赖列表中，好让 Webpack 监听 HTML 模板文件，在 HTML 模
板文件发生变化时重新启动一次编译
    compilation.fileDependencies.push(filePath);
    callback();
});
```

3. 修改输出资源

在某些场景下插件需要修改、增加、删除输出的资源，要做到这一点，则需要监听 emit 事件，因为发生 emit 事件时所有模块的转换和代码块对应的文件已经生成好，需要输出的资源即将输出，因此 emit 事件是修改 Webpack 输出资源的最后时机。

所有需要输出的资源都会被存放在 compilation.assets 中，compilation.assets 是一个键值对，键为需要输出的文件名称，值为文件对应的内容。

设置 compilation.assets 的代码如下：

```
compiler.plugin('emit', (compilation, callback) => {
  // 设置名称为 fileName 的输出资源
  compilation.assets[fileName] = {
```

```
    // 返回文件内容
    source: () => {
      // fileContent 既可以是代表文本文件的字符串，也可以是代表二进制文件的 Buffer
      return fileContent;
    },
    // 返回文件的大小
    size: () => {
      return Buffer.byteLength(fileContent, 'utf8');
    }
  };
  callback();
});
```

读取 compilation.assets 的代码如下：

```
compiler.plugin('emit', (compilation, callback) => {
  // 读取名称为 fileName 的输出资源
  const asset = compilation.assets[fileName];
  // 获取输出资源的内容
  asset.source();
  // 获取输出资源的文件大小
  asset.size();
  callback();
});
```

4. 判断 Webpack 使用了哪些插件

在开发一个插件时，我们可能需要根据当前配置是否使用了其他插件来做下一步决定，因此需要读取 Webpack 当前的插件配置情况。比如，若想判断当前是否使用了 ExtractTextPlugin，则可以使用如下代码：

```
// 判断当前配置是否使用了 ExtractTextPlugin，
// compiler 参数为 Webpack 在 apply(compiler) 中传入的参数
function hasExtractTextPlugin(compiler) {
  // 当前配置使用的所有插件列表
  const plugins = compiler.options.plugins;
  // 去 plugins 中寻找有没有 ExtractTextPlugin 的实例
  return plugins.find(plugin=>plugin.__proto__.constructor ===
ExtractTextPlugin) != null;
}
```

5.4.4 实战

下面举一个实际案例，带我们一步步实现一个插件。

该插件的名称为 EndWebpackPlugin，作用是在 Webpack 即将退出时再附加一些额外的操作，例如在 Webpack 成功编译和输出了文件后执行发布操作，将输出的文件上传到服务器。同时该插件还能区分 Webpack 构建是否执行成功。使用该插件时的方法如下：

```
module.exports = {
  plugins:[
    // 在初始化 EndWebpackPlugin 时传入了两个参数，分别是成功时的回调函数和失败时的回调函数：
    new EndWebpackPlugin(() => {
      // Webpack 构建成功，并且在文件输出后会执行到这里，在这里可以做发布文件操作
    }, (err) => {
      // Webpack 构建失败，err 是导致错误的原因
      console.error(err);
    })
  ]
}
```

要实现该插件，需要借助以下两个事件。

- **done**：在成功构建并且输出文件后，Webpack 即将退出时发生。
- **failed**：在构建出现异常时导致构建失败，Webpack 即将退出时发生。

该插件的实现非常简单，完整代码如下：

```
class EndWebpackPlugin {

  constructor(doneCallback, failCallback) {
    // 保存在构造函数中传入的回调函数
    this.doneCallback = doneCallback;
    this.failCallback = failCallback;
  }

  apply(compiler) {
    compiler.plugin('done', (stats) => {
      // 在 done 事件中回调 doneCallback
```

```
            this.doneCallback(stats);
        });
        compiler.plugin('failed', (err) => {
            // 在 failed 事件中回调 failCallback
            this.failCallback(err);
        });
    }
}
// 导出插件
module.exports = EndWebpackPlugin;
```

在开发这个插件时可以看出，找到合适的事件点去完成功能在开发插件时显得尤为重要。在 5.1 节中详细介绍过 Webpack 在运行过程中广播的常用事件，我们可以从中找到自己需要的事件。

本实例提供项目的完整代码，参见 https://github.com/gwuhaolin/end-webpack-plugin。

5.5 调试 Webpack

在编写 Webpack 的 Plugin 和 Loader 时，执行结果可能会和我们预期的不一样，就和我们平时写代码遇到了奇怪的 Bug 一样。对于无法一眼看出问题的 Bug，通常需要调试程序源码才能找出问题所在。

虽然可以通过 `console.log` 的方式完成调试，但这种方法非常不方便，也不优雅。本节将讲解如何断点调试 5.1 节（http://webpack.wuhaolin.cn/5-1 工作原理概括.zip）中的插件代码。由于 Webpack 运行在 Node.js 上，所以调试 Webpack 就相当于调试 Node.js 程序。

Webstorm 集成了 Node.js 的调试工具，因此使用 Webstorm 调试 Webpack 将非常简单，下面进行详细讲解。

1. 设置断点

在我们认为可能出现问题的地方设下断点，单击编辑区，在代码左侧出现红点时表示设置了断点，如图 5-2 所示。

```
webpack.config.js
const path = require('path');

class TestPlugin {
  constructor() {
    console.log('@plugin constructor');
  }

  apply(compiler) {
    console.log('@plugin apply');

    compiler.plugin('environment', (options) => {
      console.log('@environment');
    });

    compiler.plugin('after-environment', (options) => {
      console.log('@after-environment');
    });
```

图 5-2　设置了断点

2. 配置执行入口

告诉 Webstorm 如何启动 Webpack，由于 Webpack 实际上是一个 Node.js 应用，因此需要新建一个 Node.js 类型的执行入口，如图 5-3 所示。

图 5-3　新建一个 Node.js 类型的执行入口

在以上配置中有以下三点需要注意：

- Name 被设置为 debug webpack，就像设置了一个别名，以方便记忆和区分；
- Working directory 被设置为需要调试的插件所在的项目的根目录；
- JavaScript file 是 Node.js 的执行入口文件，被设置为 Webpack 的执行入口文

件 node_modules/webpack/bin/webpack.js。

3. 启动调试

经过以上两步，准备工作已经完成了，下面启动调试，在启动时选中前面设置的 debug webpack，如图 5-4 所示。

图 5-4　启动 Webpack

4. 执行到断点

启动后程序就会停在断点所在的位置，在这里我们可以方便地查看变量的当前状态，找出问题，如图 5-5 所示。

图 5-5　执行到断点

VSCode 的断点调试方法和 Webstorm 很类似，这里不再赘述。

除了以上调试方法，我们还可以通过 Node.js 自带的* Debugger *（https://nodejs.org/api/debugger.html）进行断点调试。

5.6 原理总结

Webpack 是一个庞大的 Node.js 应用，如果阅读过它的源码，则我们会发现实现一个完整的 Webpack 需要编写非常多的代码。但我们无须了解所有细节，只需了解其整体架构和部分细节。

对 Webpack 的使用者来说，Webpack 是一个简单强大的工具；对 Webpack 的开发者来说，Webpack 是一个扩展性高的系统。

Webpack 之所以能成功，在于它将复杂的实现隐藏了起来，向用户暴露的只是一个简单的工具，让用户能快速达成目的。同时其整体架构设计合理且扩展性高，开发扩展难度不大，通过社区补足了大量缺失的功能，使 Webpack 几乎能胜任任何场景。

希望通过对本章的学习，我们不仅能学会如何编写 Webpack 扩展，也能从中领悟到如何设计好的系统架构。

附录 A

常用的 Loader

本节将对本书用到的 Loader 及其他常用的 Loader 进行一个汇总，以方便我们快速找到自己所需的 Loader。

1. 加载文件

- raw-loader（https://github.com/webpack-contrib/raw-loader）：将文本文件的内容加载到代码中，在 3.20 节中有介绍。
- file-loader（https://github.com/webpack-contrib/file-loader）：将文件输出到一个文件夹中，在代码中通过相对 URL 去引用输出的文件，在 3.19 节、3.20 节和 4.9 节中有介绍。
- url-loader（https://github.com/webpack-contrib/url-loader）：和 file-loader 类似，但是能在文件很小的情况下以 base64 方式将文件的内容注入代码中，在 3.19 节和 3.20 节中有介绍。
- source-map-loader（https://github.com/webpack-contrib/source-map-loader）：加载额外的 SourceMap 文件，以方便断点调试，在 3.21 节中有介绍。

- svg-inline-loader（https://github.com/webpack-contrib/svg-inline-loader）：将压缩后的 SVG 内容注入代码中，在 3.20 节中有介绍。
- node-loader（https://github.com/webpack-contrib/node-loader）：加载 Node.js 原生模块的.node 文件。
- image-loader（https://github.com/tcoopman/image-webpack-loader）：加载并且压缩图片文件。
- json-loader（https://github.com/webpack-contrib/json-loader）：加载 JSON 文件。
- yaml-loader（https://github.com/okonet/yaml-loader）：加载 YAML 文件。

2. 编译模版

- pug-loader（https://github.com/pugjs/pug-loader）：将 Pug 模版转换成 JavaScript 函数并返回。
- handlebars-loader（https://github.com/pcardune/handlebars-loader）：将 Handlebars 模版编译成函数并返回。
- ejs-loader（https://github.com/okonet/ejs-loader）：将 EJS 模版编译成函数并返回。
- haml-loader（https://github.com/AlexanderPavlenko/haml-loader）：将 HAML 代码转换成 HTML。
- markdown-loader（https://github.com/peerigon/markdown-loader）：将 Markdown 文件转换成 HTML。

3. 转换脚本语言

- babel-loader（https://github.com/babel/babel-loader）：将 ES6 转换成 ES5，在 3.1 节中有介绍。
- ts-loader（https://github.com/TypeStrong/ts-loader）：将 TypeScript 转换成 JavaScript，在 3.2 节中有介绍。

- awesome-typescript-loader（https://github.com/s-panferov/awesome-typescript-loader）：将 TypeScript 转换成 JavaScript，性能要比 ts-loader 好。
- coffee-loader（https://github.com/webpack-contrib/coffee-loader）：将 CoffeeScript 转换成 JavaScript。

4. 转换样式文件

- css-loader（https://github.com/webpack-contrib/css-loader）：加载 CSS，支持模块化、压缩、文件导入等特性。
- style-loader（https://github.com/webpack-contrib/style-loader）：将 CSS 代码注入 JavaScript 中，通过 DOM 操作去加载 CSS。
- sass-loader（https://github.com/webpack-contrib/sass-loader）：将 SCSS/SASS 代码转换成 CSS，在 3.4 节中有介绍。
- postcss-loader（https://github.com/postcss/postcss-loader）：扩展 CSS 语法，使用下一代 CSS，在 3.5 节中有介绍。
- less-loader（https://github.com/webpack-contrib/less-loader）：将 Less 代码转换成 CSS 代码。
- stylus-loader（https://github.com/shama/stylus-loader）：将 Stylus 代码转换成 CSS 代码。

5. 检查代码

- eslint-loader（https://github.com/MoOx/eslint-loader）：通过 ESLint 检查 JavaScript 代码，在 3.16 节中有介绍。
- tslint-loader（https://github.com/wbuchwalter/tslint-loader）：通过 TSLint 检查 TypeScript 代码。
- mocha-loader（https://github.com/webpack-contrib/mocha-loader）：加载 Mocha 测试用例的代码。
- coverjs-loader（https://github.com/webpack-contrib/coverjs-loader）：计算测试的覆盖率。

6. 其他 Loader

- vue-loader（https://github.com/vuejs/vue-loader）：加载 Vue.js 单文件组件，在 3.7 节中有介绍。
- i18n-loader（https://github.com/webpack-contrib/i18n-loader）：加载多语言版本，支持国际化。
- ignore-loader（https://github.com/cherrry/ignore-loader）：忽略部分文件，在 3.11 节中有介绍。
- ui-component-loader（https://github.com/gwuhaolin/ui-component-loader）：按需加载 UI 组件库，例如在使用 antdUI 组件库时，不会因为只用到了 Button 组件而打包进所有的组件。

附录 B
常用的 Plugin

本节将对本书用到的 Plugin 及其他常用 Plugin 进行一个汇总，以方便我们快速找到自己所需的 Plugin。

1. 用于修改行为

- define-plugin（https://webpack.js.org/plugins/define-plugin/）：定义环境变量，在 4.7 节中有介绍。
- context-replacement-plugin（https://webpack.js.org/plugins/context-replacement-plugin/）：修改 require 语句在寻找文件时的默认行为。
- ignore-plugin（https://webpack.js.org/plugins/ignore-plugin/）：用于忽略部分文件。

2. 用于优化

- commons-chunk-plugin（https://webpack.js.org/plugins/commons-chunk-plugin/）：提取公共代码，在 4.11 节中有介绍。
- extract-text-webpack-plugin（https://github.com/webpack-contrib/extract-text-webpack-

plugin)：提取 JavaScript 中的 CSS 代码到单独的文件中，在 1.5 节中有介绍。

- prepack-webpack-plugin（https://github.com/gajus/prepack-webpack-plugin）：通过 Facebook 的 Prepack 优化输出的 JavaScript 代码的性能，在 4.13 节中有介绍。
- uglifyjs-webpack-plugin（https://github.com/webpack-contrib/uglifyjs-webpack-plugin）：通过 UglifyES 压缩 ES6 代码，在 4.8 节中有介绍。
- webpack-parallel-uglify-plugin（https://github.com/gdborton/webpack-parallel-uglify-plugin）：多进程执行 UglifyJS 代码压缩，提升构建的速度。
- imagemin-webpack-plugin（https://www.npmjs.com/package/imagemin-webpack-plugin）：压缩图片文件。
- webpack-spritesmith（https://www.npmjs.com/package/webpack-spritesmith）：用插件制作雪碧图。
- ModuleConcatenationPlugin（https://webpack.js.org/plugins/module-concatenation-plugin/）：开启 WebpackScopeHoisting 功能，在 4.14 节中有介绍。
- dll-plugin（https://webpack.js.org/plugins/dll-plugin/）：借鉴 DDL 的思想大幅度提升构建速度，在 4.2 节中有介绍。
- hot-module-replacement-plugin（https://webpack.js.org/plugins/hot-module-replacement-plugin/）：开启模块热替换功能。

3. 其他 Plugin

- serviceworker-webpack-plugin（https://github.com/oliviertassinari/serviceworker-webpack-plugin）：为网页应用增加离线缓存功能，在 3.14 节中有介绍。
- stylelint-webpack-plugin（https://github.com/JaKXz/stylelint-webpack-plugin）：将 stylelint 集成到项目中，在 3.16 节中有介绍。
- i18n-webpack-plugin（https://github.com/webpack-contrib/i18n-webpack-plugin）：使网页支持国际化。

- provide-plugin（https://webpack.js.org/plugins/provide-plugin/）：从环境中提供的全局变量中加载模块，而不用导入对应的文件。
- web-webpack-plugin（https://github.com/gwuhaolin/web-webpack-plugin）：可方便地为单页应用输出 HTML，比 html-webpack-plugin 好用。

附录 C
Webpack 的其他学习资源

除了本书，还有以下非常优秀的 Webpack 学习资源可供参考。

- Webpack 官网（https://webpack.js.org）：官方网站，涵盖入门教程及详细的 API 文档。
- Webpack 中文文档（https://doc.webpack-china.org/）：翻译自官方网站。
- Webpack 官方博客（https://medium.com/webpack）：在 Medium 上的 Webpack 官方博客，能看到 Webpack 的最新特性和未来计划。
- SurviveJS-Webpack（https://survivejs.com/webpack/）：来自 SurviveJS 的 Webpack 英文电子书，从入门到精通。
- webpack-examples（https://github.com/webpack/webpack/tree/master/examples）：来自 Webpack 官方的案例集合。
- webpack-demos（https://github.com/ruanyf/webpack-demos）：来自阮一峰的 Webpack 案例集合。
- awesome-webpack（https://github.com/webpack-contrib/awesome-webpack）：Webpack 的优秀周边生态集合。
- awesome-webpack-cn（https://github.com/webpack-china/awesome-webpack-cn）：Webpack 的优秀中文文章。

/ 好书分享 /

推荐选题，自荐翻译请联系：
zhanggx@phei.com.cn
微信：zgx228