# 深入浅出 Webpack

## 目录

* *前言*

### *第1章 入门*

* *1-1 前端的发展*
* *1-2 常见的构建工具及对比*
* *1-3 安装与使用*
* *1-4 使用 Loader*
* *1-5 使用 Plugin*
* *1-6 使用 DevServer*
* *1-7 核心概念*

### *第2章 配置*

* *2-1 Entry*
* *2-2 Output*
* *2-3 Module*
* *2-4 Resolve*
* *2-5 Plugins*
* *2-6 DevServer*
* *2-7 其它配置项*
* *2-8 整体配置结构*
* *2-9 多种配置类型*
* *2-10 配置总结*

### *第3章 实战*

* *3-1 使用 ES6 语言*
* *3-2 使用 TypeScript 语言*
* *3-3 使用 Flow 检查器*
* *3-4 使用 SCSS*
* *3-5 使用 PostCSS*
* *3-6 使用 React 框架*
* *3-7 使用 Vue 框架*
* *3-8 使用 Angular2 框架*
* *3-9 为单页应用生成 HTML*
* *3-10 管理多个单页应用*
* *3-11 构建同构应用*
* *3-12 构建 Electron 应用*
* *3-13 构建 Npm 模块*
* *3-14 构建离线应用*
* *3-15 搭配 Npm Script*
* *3-16 检查代码*
* *3-17 通过 Node.js API 启动 Webpack*
* *3-18 使用 Webpack Dev Middleware*
* *3-19 加载图片*
* *3-20 加载SVG*
* *3-21 加载 Source Map*
* *3-22 实战总结*

### *第4章 优化*

* *4-1 缩小文件搜索范围*
* *4-2 使用 DllPlugin*
* *4-3 使用 HappyPack*
* *4-4 使用 ParallelUglifyPlugin*
* *4-5 自动刷新与模块热替换*
* *4-6 开启模块热替换*
* *4-7 区分环境*
* *4-8 压缩代码*
* *4-9 CDN 加速*
* *4-10 使用 Tree Sharking*
* *4-11 提取公共代码*
* *4-12 按需加载*
* *4-13 使用 Prepack*
* *4-14 开启 Scope Hoisting*
* *4-15 输出分析*
* *4-16 优化总结*

### *第5章 原理*

* *5-1 工作原理概括*
* *5-2 输出文件分析*
* *5-3 编写 Loader*
* *5-4 编写 Plugin*
* *5-5 调试 Webpack*
* *5-6 原理总结*

# 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，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你快速入门 Webpack，再结合实际工作中常用的场景给出最佳实战，通过前3章的学习足以让你能解决工作中常见的问题。 对于想深一步学习 Webpack 的人士，本书还介绍了如何优化构建的速度和输出，解析了 Webpack 的工作原理和拓展的编写。 本书按照入门、配置、实战、优化、原理的路线层层深入，涵盖了 Webpack 的方方面面。 对于小白，本书可以作为 Webpack 快速入门教程；对于老手，本书可以进一步加深你对 Webpack 的理解。

在篡写本书时 Webpack 已经迭代到了 3.8.1 版本，涵盖了最新版本的特性解析。 由于 Webpack 从版本1到现在核心思想和 API 没有发生很大的变化，本书大部分内容适用于 Webpack 的任何一个稳定版本，但部分实例代码只适用于最新版本。

本书每一小节都会提供与之对应的完整项目代码，在每节的最后会有下载链接，它们都是可以正常运行的并且有详细的注释，你可以在需要时对照它们。

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

在阅读本书的时如你遇到任何不明白的地方，可以通过去*本书的 Github 项目主页(https://github.com/gwuhaolin/dive-into-webpack)*上提 Issue 的方式提出你的问题，作者将详细解答。

# 第1章 入门

本章带你入门当下最流行的前端构建工具 *Webpack(https://webpack.js.org)*。

学习 Webpack 前需要先知道为什么需要它：

* *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* ：告诉你构建是什么，为什么需要构建，以及常见构建工具的介绍和对比。

之后再以一个简单的项目 Hello,Webpack 为例子，带你一步步掌握 Webpack 的基础功能：

* *1-3 安装与使用* ：带你踏入 Webpack 的大门，把 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* ：通过以上几节的学习，你已经掌握 Webpack 里最基础核心的功能，做一个总结来加深对 Webpack 的认识，同时约定好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 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会面临更大的挑战。 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。

## 模块化

模块化是指把一个复杂的系统分解到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 库把它的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。

### CommonJS

*CommonJS(http://www.commonjs.org)* 是一种使用广泛的 JavaScript 模块化规范，核心思想是通过 require 方法来同步地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 导入及导出时的代码如下：

// 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，CommonJS2 在 CommonJS1 的基础上加入了 module.exports = XX 的导出方式。 CommonJS 通常指 CommonJS2。

### 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异步的方式去加载依赖的模块。 AMD 规范主要是为了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 导入及导出时的代码如下：

// 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。

### 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，它在语言的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 模块化导入及导出时的代码如下：

// 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。

### 样式文件中的模块化

除了 JavaScript 开始模块化改造，前端开发里的样式文件也支持模块化。 以 SCSS 为例，把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 语句去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;  
}

## 新框架

在 Web 应用变得庞大复杂时，采用直接操作 DOM 的方式去开发将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、提升开发效率。

### React

*React(https://facebook.github.io/react/)* 框架引入 JSX 语法到 JavaScript 语言层面中，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。

### Vue

*Vue(https://vuejs.org)* 框架把一个组件相关的 HTML 模版、JavaScript 逻辑代码、CSS 样式代码都写在一个文件里，这非常直观。

<!--HTML 模版-->  
<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>

### Angular2

*Angular2(https://angular.io)* 推崇采用 TypeScript 语言去开发应用，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';  
}

## 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一些语言缺陷会暴露出来。 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。

### 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，可以更加高效地编写代码，专注于解决问题本身。但遗憾的是不同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。为了解决兼容性问题，需要把 ES6 代码转换成 ES5 代码，*Babel(https://babeljs.io)* 是目前解决这个问题最好的工具。 Babel 的插件机制让它可灵活配置，支持把任何新语法转换成 ES5 的写法。

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

### 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。 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，不同模块可能又由不同人编写，在对接不同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 更加啰嗦，并且无法直接运行在浏览器或 Node.js 环境下。

// 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);

### 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，可以让你只在需要的地方加上类型检查。

### SCSS

*SCSS(http://sass-lang.com)* 可以让你用程序员的方式写 CSS。它是一种 CSS 预处理器，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 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。

使用新语言可以提升编码效率，但是必须把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

在阅读完上一节 *1-1前端的发展* 后你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被发明。但是这些东西都有一个共同点：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，把源代码转换成发布到线上的可执行 JavaScrip、CSS、HTML 代码，包括如下内容。

* 代码转换：TypeScript 编译成 JavaScript、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，需要构建功能把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：更新完代码后，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，把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。 构建给前端开发注入了更大的活力，解放了我们的生产力。

历史上先后出现一系列构建工具，它们各有其优缺点。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。

## 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 命令等同于执行命令 node build.js。

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。

## 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当我们在终端中启动 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，它只负责执行你定义的任务；
* 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。

## 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种个方法就可以胜任几乎所有构建场景：

* 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 插件把 scss 文件编译成 CSS 文件  
 .pipe(sass())  
 // 输出文件  
 .pipe(gulp.dest('./css'));  
});  
  
// 合并压缩 JS  
gulp.task('scripts', function() {  
 gulp.src('./js/\*.js')  
 .pipe(concat('all.js'))  
 .pipe(uglify())  
 .pipe(gulp.dest('./dist'));  
});  
  
// 监听文件变化  
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它工具搭配使用。其缺点是和 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。

## 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：通过 useHash 配置输出文件时给文件 URL 加上 md5 戳来优化浏览器缓存。
* 文件编译：通过 parser 配置文件解析器做文件转换，例如把 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来减少 HTTP 请求数。

大致使用如下：

// 加 md5  
fis.match('\*.{js,css,png}', {  
 useHash: true  
});  
  
// fis3-parser-typescript 插件把 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', {  
 // 给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。

## 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的画出了 Webpack 是什么，如下：

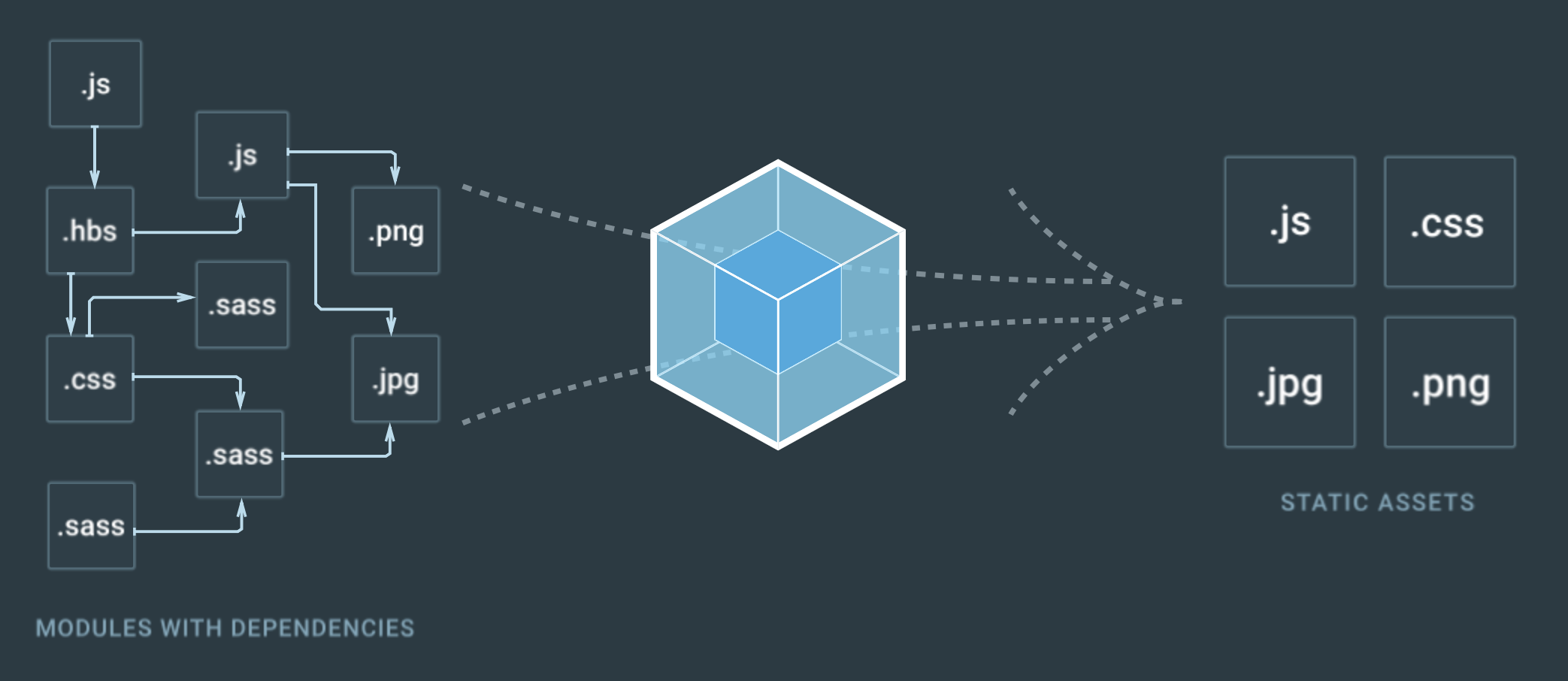


图1-2 Webpack 简介

一切文件：JavaScript、CSS、SCSS、图片、模板，在 Webpack 眼中都是一个个模块，这样的好处是能清晰的描述出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，能配置如何处理文件，大致使用如下：

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: {  
 // 把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；
* 通过 Plugin 扩展，完整好用又不失灵活；
* 使用场景不仅限于 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。

## 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。 Rollup 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，以及 Scope Hoisting 以减小输出文件大小提升运行性能。 然而 Rollup 的这些亮点随后就被 Webpack 模仿和实现。 由于 Rollup 的使用和 Webpack 差不多，这里就不详细介绍如何使用了，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 功能不如 Webpack 完善，但其配置和使用更加简单；
* Rollup 不用支持 Code Spliting，但好处是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 更加有优势，因为其打包出来的代码更小更快。 但功能不够完善，很多场景都找不到现成的解决方案。

## 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：

1. 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；
2. 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 ES6 转换成 ES6。
3. 在 Webpack 时代由于单页应用的流行，一个网页的功能和实现代码变得庞大，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地完成任务，也可以相互搭配起来弥补各自的不足。 在了解这些常见的构建工具后，你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完成自己的需求。

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和保证质量；
* Webpack 被全世界的大量 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。

下面开始跨入 Webpack 的大门吧！

# 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 前请确保你的系统安装了5.0.0及以上版本的 *Node.js(https://nodejs.org)*。

在开始给项目加入构建前，你需要先新建一个 Web 项目，方式包括：

* 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你的需求的项目。

## 安装 Webpack 到本项目

要安装 Webpack 到本项目，可按照你的需要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  
# 安装最新稳定版  
npm i -D webpack  
  
# 安装指定版本  
npm i -D webpack@<version>  
  
# 安装最新体验版本  
npm i -D webpack@beta

安装完后你可以通过这些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 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" }

## 安装 Webpack 到全局

安装到全局后你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，原因是可防止不同项目依赖不同版本的 Webpack 而导致冲突。

## 使用 Webpack

下面通过 Webpack 构建一个采用 CommonJS 模块化编写的项目，该项目有个网页会通过 JavaScript 在网页中显示 Hello,Webpack。

运行构建前，先把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

<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>

JS 工具函数文件 show.js

// 操作 DOM 元素，把 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;

JS 执行入口文件 main.js

// 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 文件读取配置，所以你还需要新建它，其内容如下：

const path = require('path');  
  
module.exports = {  
 // JavaScript 执行入口文件  
 entry: './main.js',  
 output: {  
 // 把所有依赖的模块合并输出到一个 bundle.js 文件  
 filename: 'bundle.js',  
 // 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，你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 和 show.js 及内置的 webpackBootstrap 启动函数。 这时你用浏览器打开 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， 递归的寻找出入口文件的所有依赖，把入口和其所有依赖打包到一个单独的文件中。 从 Webpack2 开始，已经内置了对 ES6、CommonJS、AMD 模块化语句的支持。

至此你已经学会了 Webpack，接下来我们将探索 Webpack 的更多功能。

本实例 *提供项目完整代码(http://webpack.wuhaolin.cn/1-3安装与使用.zip)*

# 1-4 使用 Loader

在上一节中使用 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 代码让文字居中显示，main.css 的内容如下：

#app{  
 text-align: center;  
}

Webpack 把一切文件看作模块，CSS 文件也不例外，要引入 main.css 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，需要使用 Webpack 的 Loader 机制。Webpack的配置修改使用如下：

const path = require('path');  
  
module.exports = {  
 // JavaScript 执行入口文件  
 entry: './main.js',  
 output: {  
 // 把所有依赖的模块合并输出到一个 bundle.js 文件  
 filename: 'bundle.js',  
 // 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$/,  
 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，再交给 style-loader 把 CSS 内容注入到 JavaScript 里。 在配置 Loader 时需要注意的是：

* use 属性需要传入一个数组，Loader 的执行顺序是由后到前的；
* 每一个 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，你会发现 bundle.js 文件被更新了，里面注入了在 main.css 中写的 CSS，而不是会额外生成一个 CSS 文件。 但是重新刷新 index.html 网页时将会发现 Hello,Webpack 居中了，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把 CSS 内容用 JavaScript 里的字符串存储起来，在页面 JavaScript 执行后通过 DOM 操作动态地往 HTML head 标签里插入 HTML style 标签。 也许你认为这样做会导致 JavaScript 文件变大并导致加载网页时间变长，想让 Webpack 单独输出 CSS 文件。下一节 *1-5 使用Plugin* 将教你如何通过 Webpack Plugin 机制来实现。

本实例 *提供项目完整代码(http://webpack.wuhaolin.cn/1-4使用Loader.zip)*

给 Loader 传入属性的方式除了有 querystring 外，还可以通过 Object 传入，以上的 Loader 配置可以修改为如下：

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

除了在 webpack.config.js 配置文件中配置 Loader 外，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 在采用 style-loader 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，它给 Webpack 带来了很大的灵活性。

在上一节中通过 Loader 加载了 CSS 文件，本节将通过 Plugin 把注入到 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: {  
 // 把所有依赖的模块合并输出到一个 bundle.js 文件  
 filename: 'bundle.js',  
    // 把输出文件都放到 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，你会发现 dist 目录下多出一个 main\_1a87a56a.css 文件，bundle.js 里也没有 CSS 代码了，再把该 CSS 文件引入到 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。 对此你可以通过插件的 filename 属性，告诉插件输出的 CSS 文件名称是通过 [name]\_[contenthash:8].css 字符串模版生成的，里面的 [name] 代表文件名称， [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 *提供项目完整代码(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，但在实际开发中你可能会需要：

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

对于这些， Webpack 都为你考虑好了。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

按照成功后执行 webpack-dev-server 命令， DevServer 就启动了，这时你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 http://localhost:8080/ ，DevServer 启动后会一直驻留在后台保持运行，访问这个网址你就能获取项目根目录下的 index.html。 用浏览器打开这个地址你会发现页面空白错误原因是 ./dist/bundle.js 加载404了。 同时你会发现并没有文件输出到 dist 目录，原因是 DevServer 会把 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>

## 实时预览

接着上面的步骤，你可以试试修改 main.js main.css show.js 中的任何一个文件，保存后你会发现浏览器会被自动刷新，运行出修改后的效果。

Webpack 在启动时可以开启监听模式，开启监听模式后 Webpack 会监听本地文件系统的变化，发生变化时重新构建出新的结果。Webpack 默认是关闭监听模式的，你可以在启动 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完构建后通知 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，你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。

## 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，通过将被更新过的模块替换老的模块，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，你只需在启动 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。

## 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，则你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，让你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。 加上参数重启 DevServer 后刷新页面，再打开 Chrome 浏览器的开发者工具，就可在 Sources 栏中看到可调试的源代码了。

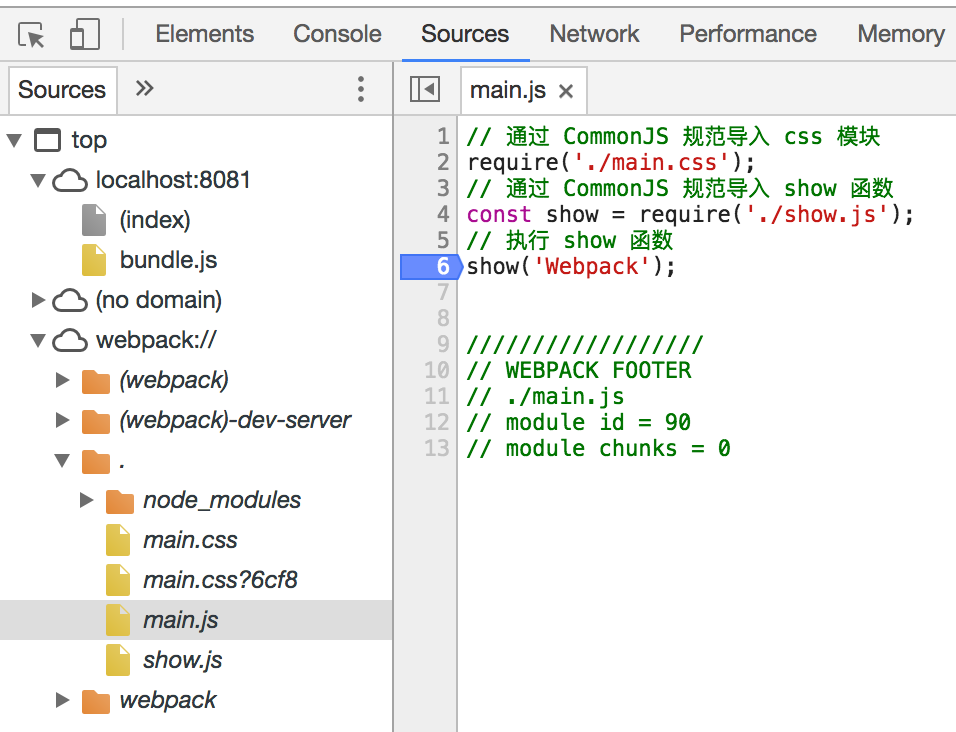


图1.6.1 在开发者工具中调试 Source Map

本实例 *提供项目完整代码(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，相信你已经对 Webpack 有了一个初步的认识。虽然Webpack 功能强大且配置项多，但只要你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，一个模块对应着一个文件。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，用于把模块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你想要的事情。
* **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 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 和其所有依赖的 Module 被分到一个组也就是一个 Chunk。最后 Webpack 会把所有 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。

在实际应用中你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，你会对 Webpack 有一个整体的认识，这能让你在以后使用 Webpack 的过程中快速知道应该通过配置什么去完成你想要的功能，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。

下一章 *2 配置* 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 时用webpack --config webpack-dev.config.js 指定配置文件，再去 webpack-dev.config.js 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：

* *2-1 Entry* 配置模块的入口；
* *2-2 Output* 配置如何输出最终想要的代码；
* *2-3 Module* 配置模块相关；
* *2-4 Resolve* 配置寻找模块的规则；
* *2-5 Plugins* 配置扩展插件；
* *2-6 DevServer* 配置 DevServer；
* *2-7 其它配置项* 其它零散的配置项；
* *2-8 整体配置结构* 整体地描述各配置项的结构；
* *2-9 多种配置类型* 配置文件不止可以返回一个 Object，还有其他返回形式；
* *2-10 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。

## 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 的路径和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。

## Entry 类型

Entry 类型可以是以下三种中的一种或者相互组合：

|  |  |  |
| --- | --- | --- |
| 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。

## Chunk 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 键值对里键的名称。

## 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 的配置会受到到其他因素的影响导致不能写成静态的值。其解决方法是把 Entry 设置成一个函数去动态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。

## 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，则可以把它写成静态不变的：

filename: 'bundle.js'

但是在有多个 Chunk 要输出时，就需要借助模版和变量了。前面说到 Webpack 会为每个 Chunk取一个名称，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]，这时你可以把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 name 还包括：

|  |  |
| --- | --- |
| 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)* 插件是使用 contenthash 来代表哈希值而不是 chunkhash， 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。

## 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。 常见的会在运行时生成 Chunk 场景有在使用 CommonChunkPlugin、使用 import('path/to/module') 动态加载等时。 chunkFilename 支持和 filename 一致的内置变量。

## 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]')

## 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，需要把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：

<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 都支持字符串模版，内置变量只有一个：hash 代表一次编译操作的 hash 值。

## 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。

## 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它们。

* 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。

### 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 = lib\_code;  
  
// 使用库的方法  
LibraryName.doSomething();

假如 output.library 为空，则将直接输出：

lib\_code

其中 lib\_code 代指导出库的代码内容，是有返回值的一个自执行函数。

### 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 是指模块发布到 Npm 代码仓库时的名称。

### 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 规范很相似，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。

### 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();

### window

编写的库将通过 window 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  
window['LibraryName'] = lib\_code;  
  
// 使用库的方法  
window.LibraryName.doSomething();

### global

编写的库将通过 window 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  
global['LibraryName'] = lib\_code;  
  
// 使用库的方法  
global.LibraryName.doSomething();

## 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;

现在你想让构建输出的代码只导出其中的 a，可以把 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：

// 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 里常用的配置项，还有部分几乎用不上的配置项没有一一列举，你可以在 *Webpack 官方文档(https://webpack.js.org/configuration/output/)* 上查阅它们。

# 2-3 Module

module 配置如何处理模块。

## 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，数组里每一项都描述了如何去处理部分文件。 配置一项 rules 时大致通过以下方式：

1. 条件匹配：通过 test 、 include 、 exclude 三个配置项来命中 Loader 要应用规则的文件。
2. 应用规则：对选中后的文件通过 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，同时还可以分别给 Loader 传入参数。
3. 重置顺序：一组 Loader 的执行顺序默认是从右到左执行，通过 enforce 选项可以让其中一个 Loader 的执行顺序放到最前或者最后。

下面来通过一个例子来说明具体使用方法：

module: {  
 rules: [  
 {  
 // 命中 JavaScript 文件  
 test: /\.js$/,  
 // 用 babel-loader 转换 JavaScript 文件  
 // ?cacheDirectory 表示传给 babel-loader 的参数，用于缓存 babel 编译结果加快重新编译速度  
 use: ['babel-loader?cacheDirectory'],  
 // 只命中src目录里的js文件，加快 Webpack 搜索速度  
 include: path.resolve(\_\_dirname, 'src')  
 },  
 {  
 // 命中 SCSS 文件  
 test: /\.scss$/,  
 // 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，再把结果交给 css-loader 最后再给 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，你还可以通过一个 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' 的含义是把该 Loader 的执行顺序放到最后  
 // enforce 的值还可以是：代表把 Loader 的执行顺序放到最前的 pre  
 enforce:'post'  
 },  
 // 省略其它 Loader  
]

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，其实它们还都支持数组类型，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，即符合数组中的任何一个就会被命中。

## 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。 原因是一些库例如 jQuery 、ChartJS 它们庞大又没有采用模块化标准，让 Webpack 去解析这些文件耗时又没有意义。

noParse 是可选配置项，类型需要是 RegExp | [RegExp] | function 其中一个。

例如想要忽略掉 jQuery 、ChartJS，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 or false  
 return /jquery|chartjs/.test(content);  
}

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。

## parser

因为 Webpack 是以模块化的 JavaScript 文件为入口，所以内置了对模块化 JavaScript 的解析功能，支持 AMD CommonJS SystemJS ES6。 parser 属性可以更细粒度的配置哪些模块语法要解析哪些不解析，和 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 使用如下：

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好的规则去寻找，但你也可以根据自己的需要修改默认的规则。

## alias

resolve.alias 配置项通过别名来把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/'  
 }  
}

当你通过 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 配置的含义是把导入语句里的 components 关键字替换成 ./src/components/。

这样做可能会命中太多的导入语句，alias 还支持 $ 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，即只会把 import 'react' 关键字替换成 import '/path/to/react.min.js'。

## mainFields

有一些第三方模块会针对不同环境提供几分代码。 例如分别提供采用 ES5 和 ES6 的2份代码，这2份代码的位置写在 package.json 文件里，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那份代码，mainFields 默认如下：

mainFields: ['browser', 'main']

Webpack 会按照数组里的顺序去package.json 文件里寻找，只会使用找到的第一个。

假如你想优先采用 ES6 的那份代码，可以这样配置：

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

## 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。

假如你想让 Webpack 优先使用目录下的 TypeScript 文件，可以这样配置：

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

## modules

resolve.modules 配置 Webpack 去哪些目录下寻找第三方模块，默认是只会去 node\_modules 目录下寻找。 有时你的项目里会有一些模块会大量被其它模块依赖和导入，由于其它模块的位置分布不定，针对不同的文件都要去计算被导入模块文件的相对路径， 这个路径有时候会很长，就像这样 import '../../../components/button' 这时你可以利用 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，把 modules 配置成

modules:['./src/components','node\_modules']

后，你可以简单通过 import 'button' 导入。

## 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']

## enforceExtension

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

## enforceModuleExtension

enforceModuleExtension 和 enforceExtension 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没带文件后缀， 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 功能，各种各样的 Plugin 几乎让 Webpack 可以做任何构建相关的事情。

## 配置 Plugin

Plugin 的配置很简单，plugins 配置项接受一个数组，数组里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代码到 common  
 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，你需要善于使用搜索引擎去寻找解决问题的方法。

# 2-6 devServer

*1-6 使用DevServer* 介绍过用来提高开发效率的 DevServer ，它提供了一些配置项可以改变 DevServer 的默认行为。 要配置 DevServer ，除了在配置文件里通过 devServer 传入参数外，还可以通过命令行参数传入。 注意只有在通过 DevServer 去启动 Webpack 时配置文件里 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。

## hot

devServer.hot 配置是否启用 *1-6 使用DevServer* 中提到的模块热替换功能。 DevServer 默认的行为是在发现源代码被更新后会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。

## inline

DevServer 的实时预览功能依赖一个注入到页面里的代理客户端去接受来自 DevServer 的命令和负责刷新网页的工作。 devServer.inline 用于配置是否自动注入这个代理客户端到将运行在页面里的 Chunk 里去，默认是会自动注入。 DevServer 会根据你是否开启 inline 来调整它的自动刷新策略：

* 如果开启 inline，DevServer 会在构建完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，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，当构建完变化后的代码时通过刷新 iframe 来实现实时预览。 但这时你需要去 http://localhost:8080/webpack-dev-server/ 实时预览你的网页了。

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

## historyApiFallback

devServer.historyApiFallback 用于方便的开发使用了 *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，显示出对应的界面。

配置 historyApiFallback 最简单的做法是：

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。

如果你的应用由多个单页应用组成，这就需要 DevServer 根据不同的请求来返回不同的 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' },  
 // 其它的都返回 index.html  
 { from: /./, to: '/index.html' },  
 ]  
}

## contentBase

devServer.contentBase 配置 DevServer HTTP 服务器的文件根目录。 默认情况下为当前执行目录，通常是项目根目录，所有一般情况下你不必设置它，除非你有额外的文件需要被 DevServer 服务。 例如你想把项目根目录下的 public 目录设置成 DevServer 服务器的文件根目录，你可以这样配置：

devServer:{  
 contentBase: path.join(\_\_dirname, 'public')  
}

这里需要指出可能会让你疑惑的地方，DevServer 服务器通过 HTTP 服务暴露出的文件分为两类：

* 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，所以你在使用了 DevServer 时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，你可以通过 contentBase:false 来关闭暴露本地文件。

## 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'  
 }  
}

## 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。 例如你想要局域网中的其它设备访问你本地的服务，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。

## 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 端口已经被其它程序占有就使用 8081，如果 8081 还是被占用就使用 8082，以此类推。

## 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'  
]

## disableHostCheck

devServer.disableHostCheck 配置项用于配置是否关闭用于 DNS 重绑定的 HTTP 请求的 HOST 检查。 DevServer 默认只接受来自本地的请求，关闭后可以接受来自任何 HOST 的请求。 它通常用于搭配 --host 0.0.0.0 使用，因为你想要其它设备访问你本地的服务，但访问时是直接通过 IP 地址访问而不是 HOST 访问，所以需要关闭 HOST 检查。

## https

DevServer 默认使用 HTTP 协议服务，它也能通过 HTTPS 协议服务。 有些情况下你必须使用 HTTPS，例如 HTTP2 和 Service Worker 就必须运行在 HTTPS 之上。 要切换成 HTTPS 服务，最简单的方式是：

devServer:{  
 https: true  
}

DevServer 会自动的为你生成一份 HTTPS 证书。

如果你想用自己的证书可以这样配置：

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

## clientLogLevel

devServer.clientLogLevel 配置在客户端的日志等级，这会影响到你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，可取如下之一的值 none | error | warning | info。 默认为 info 级别，即输出所有类型的日志，设置成 none 可以不输出任何日志。

## compress

devServer.compress 配置是否启用 gzip 压缩。boolean 类型，默认 false。

## open

devServer.open 用于在 DevServer 启动且第一次构建完时自动用你系统上默认的浏览器去打开要开发的网页。 同时还提供 devServer.openPage 配置项用于打开指定 URL 的网页。

# 2-7 其它配置项

除了前面介绍到的配置项外，Webpack 还提供了一些零散的配置项。下面来介绍它们中常用的部分。

## 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 可以是以下之一：

|  |  |
| --- | --- |
| 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 |

例如当你设置 target:'node' 时，源代码中导入 Node.js 原生模块的语句 require('fs') 将会被保留，fs 模块的内容不会打包进 Chunk 里。

## Devtool

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

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

## 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，想打开需要如下配置：

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 配置项去更灵活的控制监听模式，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/,  
 // 监听到变化发生后等 300ms 再去执行动作，截流，  
 // 防止文件更新太快导致重新编译频率太快。默认 300ms   
 aggregateTimeout: 300,  
 // 判断文件是否发生变化是通过不停的去询问系统指定文件有没有变化实现的  
 // 默认每秒问 1000 次  
 poll: 1000  
 }  
}

## Externals

Externals 用来告诉 Webpack 要构建的代码中使用了哪些不用被打包的模块，也就是说这些模版是外部环境提供的，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，例如在你的 HTML HEAD 标签里通过以下代码：

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

引入 jQuery 后，全局变量 jQuery 就会被注入到网页的 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，可能需要这样：

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

构建后你会发现输出的 Chunk 里包含的 jQuery 库的内容，这导致 jQuery 库出现了2次，浪费加载流量，最好是 Chunk 里不会包含 jQuery 库的内容。

Externals 配置项就是为了解决这个问题。

通过 externals 可以告诉 Webpack JavaScript 运行环境已经内置了那些全局变量，针对这些全局变量不用打包进代码中而是直接使用全局变量。 要解决以上问题，可以这样配置 externals：

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

## 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2个文件  
 entry: { // 有2个入口  
 a: './app/entry-a',  
 b: ['./app/entry-b1', './app/entry-b2']  
 },  
  
 // 输出结果，在 Webpack 经过一系列处理出最终想要的代码后，如何输出它们。  
 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', // 当配置了多个 entry 时，通过名称模版为不同的 entry 生成不同的文件名称  
 filename: '[chunkhash].js', // 根据文件内容 hash 值生成文件名称，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，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 上去  
   
 // 导出库的名称，string 类型  
 // 不填它时，默认输出格式是匿名的立即执行函数  
 library: 'MyLibrary',  
   
 // 导出库的类型，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去，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?$/, // 正则匹配命中要使用 Loader 的文件  
 include: [ // 只会命中这里面的文件  
 path.resolve(\_\_dirname, 'app')  
 ],  
 exclude: [ // 忽略这里面的文件  
 path.resolve(\_\_dirname, 'app/demo-files')  
 ],  
 use: [ // 使用那些 Loader，有先后次序，从后往前执行  
 'style-loader', // 直接使用 Loader 的名称  
 {  
 loader: 'css-loader',   
 options: { // 给 html-loader 传一些参数  
 }  
 }  
 ]  
 },  
 ],  
 noParse: [ // 不用解析和处理的模块  
 /special-library\.js$/ // 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，array 类型，默认以 node\_modules 为根  
 '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  
 // 把 'module' 映射 'new-module'，同样的 'module/path/file' 也会被映射成 'new-module/path/file'  
 'module': 'new-module',  
 // 使用结尾符号 $ 后，把 '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的配置  
 {  
 name: 'module', // 老的模块  
 alias: 'new-module', // 新的模块  
 // 是否是只映射模块，如果是 true 只有 'module' 会被映射，如果是 false 'module/inner/path' 也会被映射  
 onlyModule: true,   
 }  
 ],  
 symlinks: true, // 是否跟随文件软链接去搜寻模块的路径  
 descriptionFiles: ['package.json'], // 模块的描述文件  
 mainFields: ['main'], // 模块的描述文件里的描述入口的文件的字段名称  
 enforceExtension: false, // 是否强制导入语句必须要写明文件后缀  
 },  
   
 // 输出文件性能检查配置  
 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, // 最大文件大小 (单位 bytes)  
 maxEntrypointSize: 400000, // 最大入口文件大小 (单位 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, // 是否是否 HTTPS  
 },  
   
 profile: true, // 是否捕获 Webpack 构建性能信息，用于分析什么原因导致构建性能不佳  
   
 cache: false, // 是否启用缓存提升构建速度  
   
 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 再去执行动作，截流，防止文件更新太快导致重新编译频率太快。默认 300ms   
 aggregateTimeout: 300,  
 // 判断文件是否发生变化是不停的去询问系统指定文件有没有变化，默认每秒问 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 所需的配置外，还有其它更灵活的方式，以简化不同场景的配置。 下面来一一介绍它们。

## 导出一个 Function

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

如果采用导出一个 Object 来描述 Webpack 所需的配置的方法，需要写量个文件。 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 灵活的控制配置，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'];  
  
 // 在生成环境才压缩  
 if (isProduction) {  
 plugins.push(  
 // 压缩输出的 JS 代码  
 new UglifyJsPlugin()  
 )  
 }  
  
 return {  
 plugins: plugins,  
 // 在生成环境不输出 Source Map  
 devtool: isProduction ? undefined : 'source-map',  
 };  
}

在运行 Webpack 时，会给这个函数传入2个参数，分别是：

1. 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，设置它需要在启动 Webpack 时带上参数。例如启动命令是 webpack --env.production --env.bao=foo时，则 env 的值是 {"production":"true","bao":"foo"}。
2. argv：代表在启动 Webpack 时所有通过命令行传入的参数，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 *提供项目完整代码(http://webpack.wuhaolin.cn/2-9多种配置类型.zip)*

## 导出一个返回 Promise 的函数

在有些情况下你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)  
 })  
}

## 导出多份配置

除了只导出一份配置外，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。

注意本特性从 Webpack 3.1.0 版本才开始支持。

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。

这特别适合于用 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

从前面的配置看来选项很多，Webpack 内置了很多功能。 你不必都记住它们，只需要大概明白 Webpack 原理和核心概念去判断选项大致属于哪个大模块下，再去查详细的使用文档。

通常你可用如下经验去判断如何配置 Webpack：

* 想让源文件加入到构建流程中去被 Webpack 控制，配置 entry。
* 想自定义输出文件的位置和名称，配置 output。
* 想自定义寻找依赖模块时的策略，配置 resolve。
* 想自定义解析和转换文件的策略，配置 module，通常是配置 module.rules 里的 Loader。
* 其它的大部分需求可能要通过 Plugin 去实现，配置 plugin。

# 第3章 实战

本章教你如何用 Webpack 去解决实际项目中常见的场景。

按照不同场景划分成以下几类：

* 使用新语言来开发项目：
* *3-1 使用 ES6 语言*
* *3-2 使用 TypeScript 语言*
* *3-3 使用 Flow 检查器*
* *3-4 使用 SCSS 语言*
* *3-5 使用 PostCSS*
* 使用新框架来开发项目：
* *3-6 使用 React 框架*
* *3-7 使用 Vue 框架*
* *3-8 使用 Angular2 框架*
* 用 Webpack 构建单页应用：
* *3-9 为单页应用生成 HTML*
* *3-10 管理多个单页应用*
* 用 Webpack 构建不同运行环境的项目：
* *3-11 构建同构应用*
* *3-12 构建 Electron 应用*
* *3-13 构建 Npm 模块*
* *3-14 构建离线应用*
* Webpack 结合其它工具搭配使用，各取所长：
* *3-15 搭配 Npm Script*
* *3-16 检查代码*
* *3-17 通过 Node.js API 启动 Webpack*
* *3-18 使用 Webpack Dev Middleware*
* 用 Webpack 加载特殊类型的资源：
* *3-19 加载图片*
* *3-20 加载SVG*
* *3-21 加载 Source Map*
* *3-22 实战总结*

# 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 所有的标准支持不全，这导致在开发中不敢全面地使用 ES6。

通常我们需要把采用 ES6 编写的代码转换成目前已经支持良好的 ES5 代码，这包含2件事：

1. 把新的 ES6 语法用 ES5 实现，例如 ES6 的 class 语法用 ES5 的 prototype 实现。
2. 给新的 API 注入 polyfill ，例如使用新的 fetch API 时注入对应的 polyfill 后才能让低端浏览器正常运行。

## Babel

*Babel(https://babeljs.io)* 可以方便的完成以上2件事。 Babel 是一个 JavaScript 编译器，能将 ES6 代码转为 ES5 代码，让你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的扩展。 在 Babel 执行编译的过程中，会从项目根目录下的 .babelrc 文件读取配置。.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"  
 ]  
}

### Plugins

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

以上配置文件里的 transform-runtime 对应的插件全名叫做 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它：

npm i -D babel-plugin-transform-runtime

babel-plugin-transform-runtime 是 Babel 官方提供的一个插件，作用是减少冗余代码。 Babel 在把 ES6 代码转换成 ES5 代码时通常需要一些 ES5 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;  
}

这会导致每个使用了 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 的作用在于不把辅助函数内容注入到文件里，而是注入一条导入语句：

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 依赖到你的项目后的代码才能正常运行， 也就是说 babel-plugin-transform-runtime 和 babel-runtime 需要配套使用，使用了 babel-plugin-transform-runtime 后一定需要 babel-runtime。

### 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 对一组新语法特性提供支持，多个 Presets 可以叠加。 Presets 其实是一组 Plugins 的集合，每一个 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类： - 已经被写入 ECMAScript 标准里的特性，由于每年都有特性被加入到标准里，这其中又分为：

* *es2015(https://babeljs.io/docs/plugins/preset-es2015/)* 包含在2015里加入的新特性；
* *es2016(https://babeljs.io/docs/plugins/preset-es2016/)* 包含在2016里加入的新特性；
* *es2017(https://babeljs.io/docs/plugins/preset-es2017/)* 包含在2017里加入的新特性；
* *es2017(https://babeljs.io/docs/plugins/preset-es2017/)* 包含在2017里加入的新特性；
* *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：

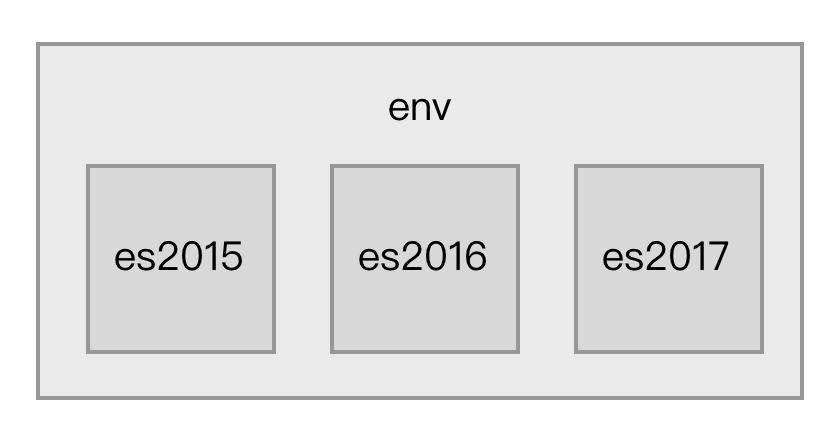


图3.1.1 ECMAScript 标准里的特性关系图

* 被社区提出来的但还未被写入 ECMAScript 标准里特性，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，有 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 社区开始着手实现；
* stage4 在下一年就加入到标准里去。

它们之间的关系如图：

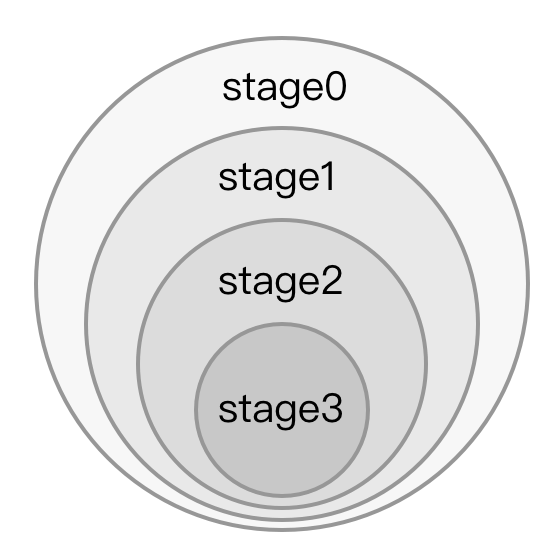


图3.1.2 stage 关系图

* 为了支持一些特定的语法，例如 babel-preset-react 是为了支持 React 开发里的 JSX 语法。

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

## 接入 Babel

在了解 Babel 后，下一步要知道如何在 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 配置如下：

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

配置命中了项目目录下所有的 JavaScript 文件，通过 babel-loader 去调用 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   
# 根据你的需求选择不同的 Plugins 或 Presets  
npm i -D babel-preset-env

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-1使用ES6语言.zip)*

# 3-2 使用 TypeScript 语言

## 认识 TypeScript

*TypeScript(http://www.typescriptlang.org)* 是 JavaScript 的一个超集，主要提供了类型系统和对 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，必须通过构建把它转换成 JavaScript 代码后才能运行。

改造下前面用过的例子 Hello,Webpack，用 TypeScript 重写 JavaScript。由于 TypeScript 是 JavaScript 的超集，直接把后缀 .js 改成 .ts 是可以的。 但为了体现出 TypeScript 的不同，我们重写 JavaScript 代码为如下，加入类型检查：

// show.ts  
// 操作 DOM 元素，把 content 显示到网页上  
// 通过 ES6 模块规范导出 show 函数  
// 给 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 官方提供了能把 TypeScript 转换成 JavaScript 的编译器。 你需要在当前项目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 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，你可以通过 tsc hello.ts 命令编译出 hello.js 和 hello.js.map 文件。

## 减少代码冗余

TypeScript 编译其会有和在 *3-1 使用ES6语言*中 Babel 一样的问题：在把 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的出现在多个文件中，可以开启 TypeScript 编译器的 importHelpers 选项，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，会把辅助函数换成如下导入语句：

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。

## 集成 Webpack

要让 Webpack 支持 TypeScript，需要解决以下2个问题：

1. 通过 Loader 把 TypeScript 转换成 JavaScript。
2. 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 Resolve 中的 extensions* 我们需要修改默认的 resolve.extensions 配置项。

综上，相关 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 ts 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 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，你将会在 dist 目录看到输出的 JavaScript 文件 bundle.js，和对应的 Source Map 文件 bundle.js.map。 在浏览器里打开 index.html 页面后，来开发工具里可以看到和调试用 TypeScript 编写的源码。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-2使用TypeScript语言.zip)*

# 3-3 使用 Flow 检查器

## 认识 Flow

*Flow(https://flow.org)* 是一个 Facebook 开源的 JavaScript 静态类型检测器，它是 JavaScript 语言的超集。 你所需要做的就是在需要的地方加上类型检查，例如在两个由不同人开发的模块对接的接口出加上静态类型检查，能在编译阶段就指出部分模块使用不当的问题。 同时 Flow 也能通过类型推断检查出 JavaScript 代码中潜在的 Bug。

Flow 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');

需要注意的时代码中的第一行 // @flow 告诉 Flow 检查器这个文件需要被检查。

## 使用 Flow

以上只是让你了解 Flow 的功能，下面教你任何运行 Flow 去检查代码。 Flow 检测器由高性能跨平台的 *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，安装完成后通过先配置 Npm Script

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

再通过 npm run flow 去调用 Flow 执行代码检查。

除此之外你还可以通过

npm i -g flow-bin

把 Flow 安装到全局后，再直接通过 flow 命令去执行代码检查。

安装成功后，在项目根目录下执行 Flow 后，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把这些静态类型语法去掉。 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点：

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

## 集成 Webpack

由于使用了 Flow 项目一般都会使用 ES6 语法，所以把 Flow 集成到使用 Webpack 构建的项目里最方便的方法是借助 Babel，下面来修改前面讲过的*3-1 使用 ES6 语言*，为其加入 Flow 代码检查。 改动很少，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"  
  ]
* 往源码里加入静态类型后重新构建项目，你会发现采用了 Flow 的源码还是能正常在浏览器中运行。

要明确构建的目的只是为了去除源码中的 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 Flow，可以实时在代码中高亮指出 Flow 检查出的问题。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-3使用Flow检查器.zip)*

# 3-4 使用 SCSS 语言

## 认识 SCSS

*SCSS(http://sass-lang.com)* 可以让你用更灵活的方式写 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 SASS，区别在于 SASS 语法类似 Ruby，而 SCSS 语法类似 CSS，对于熟悉 CSS 的前端工程师来说会更喜欢 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 可以提升编码效率，但是必须把 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，所以主要来介绍下 *node-sass(https://github.com/sass/node-sass)*。

node-sass 核心模块是由 C++ 编写，再用 Node.js 封装了一层，以供给其它 Node.js 调用。 node-sass 还支持通过命令行调用，先安装它到全局：

npm i -g node-sass

再执行编译命令：

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

你就能在源码同目录下看到编译后的 main.css 文件。

## 接入 Webpack

由于需要把 SCSS 源代码转换成 CSS 代码，在*1-4 使用Loader](../1入门/1-4使用Loader.md)中曾介绍过转换文件最适合的方式是使用 Loader，Webpack 官方提供了对应的 [sass-loader(https://github.com/webpack-contrib/sass-loader)*。

Webpack 接入 sass-loader 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：

1. 通过 sass-loader 把 SCSS 源码转换为 CSS 代码，再把 CSS 代码交给 css-loader 去处理。
2. css-loader 会找出 CSS 代码中的 @import 和 url() 这样的导入语句，告诉 Webpack 依赖这些资源。同时还支持 CSS Modules、压缩 CSS 等功能。处理完后再把结果交给 style-loader 去处理。
3. style-loader 会把 CSS 代码转换成字符串后，注入到 JavaScript 代码中去，通过 JavaScript 去给 DOM 增加样式。如果你想把 CSS 代码提取到一个单独的文件而不是和 JavaScript 混在一起，可以使用*1-5 使用Plugin* 中介绍过的 ExtractTextPlugin。

由于接入 sass-loader，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*提供项目完整代码(http://webpack.wuhaolin.cn/3-4使用SCSS语言.zip)*

# 3-5 使用 PostCSS

## 认识 PostCSS

*PostCSS(http://postcss.org)* 是一个 CSS 处理工具，和 SCSS 不同的地方在于它通过插件机制可以灵活的扩展其支持的特性，而不是像 SCSS 那样语法是固定的。 PostCSS 的用处非常多，包括给 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，用工程化手段给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 更加灵活、可扩张性强，而 SCSS 内置了大量功能而不能扩展。

为了更直观的展示 PostCSS，让我们来看一些例子。

给 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，即提供了给 JavaScript 代码调用的模块，也提供了可执行的文件。 在 PostCSS 启动时，会从目录下的 postcss.config.js 文件中读取所需配置，所以需要新建该文件，文件内容大致如下：

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

其中的 *postcss-cssnext(http://cssnext.io)* 插件可以让你使用下一代 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，并且该插件还包含给 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。

## 接入 Webpack

虽然使用 PostCSS 后文件后缀还是 .css 但这些文件必须先交给 *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 给项目带来了新的依赖需要安装，如下：

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

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-5使用PostCSS.zip)*

# 3-6 使用 React 框架

## React 语法特征

使用了 React 项目的代码特征有 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 语法并不是必须的，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，所以在构建过程中需要把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，下面分别来介绍如何在使用 Babel 或 TypeScript 的项目中接入 React 框架。

## React 与 Babel

要在使用 Babel 的项目中接入 React 框架是很简单的，只需要加入 React 所依赖的 Presets *babel-preset-react(https://babeljs.io/docs/plugins/preset-react/)*。 接下来通过修改前面讲过的*3-1 使用 ES6 语言*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"  
],

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你将会发现由 React 渲染出来的 Hello,Webpack。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-6使用React框架Babel.zip)*

## React 与 TypeScript

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

* 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，需要安装 react 和 react-dom 对应的 TypeScript 接口描述模块 @types/react 和 @types/react-dom 后才能通过编译。

接下来通过修改*3-2 使用 TypeScript 语言*中讲过的的项目，为其接入 React 框架。 修改 TypeScript 编译器配置文件 tsconfig.json 增加对 JSX 语法的支持，如下：

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

由于 main.js 文件中存在 JSX 语法，再把 main.js 文件重命名为 main.tsx，同时修改文件内容为在上面 *React 与 Babel* 里所采用的 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， 需要注意的是 Webpack Loader 配置的 test 选项需要匹配到 tsx 类型的文件，并且 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 方便在浏览器里调试 TypeScript 代码  
};

通过

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

安装新的依赖后重启构建，重新打开网页你将会发现由 React 渲染出来的 Hello,Webpack。

本实例*提供项目完整代码(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，相比于 React、Angular 它更灵活轻量。 它不会强制性地内置一些功能和语法，你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 是古老和成熟的做法，本书只专注于讲解如何用 Webpack 构建 Vue 单文件组件。

## 认识 Vue

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

App.vue 单文件组件，唯一的组件，也是根组件：

<!--渲染模版-->  
<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 HTML 文件的 .vue 文件就能描述清楚一个组件所需的模版、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)  
});

入口文件创建一个 Vue 的根实例，在 ID 为 app 的 DOM 节点上渲染出上面定义的 App 组件。

## 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/)* 可以非常方便的完成单文件组件的转换。

修改 Webpack 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，它们的作用分别是：

* 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、以及 HTML 模版 template，再分别把它们交给对应的 Loader 去处理。
* css-loader：加载由 vue-loader 提取出的 CSS 代码。
* vue-template-compiler：把 vue-loader 提取出 HTML 模版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 模版相对于在浏览器中再去编译 HTML 模版的好处在于性能更好。

重新启动构建你就能看到由 Vue 渲染出的 Hello,Webpack 了。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-7使用Vue框架Babel.zip)*

## 使用 TypeScript 编写 Vue 应用

从 Vue 2.5.0+ 版本开始，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。

用 TypeScript 重写上面的项目，改动如下：

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,  
 // 输出的 JavaScript 采用 es2015 模块化，使 Tree Sharking 生效  
 "module": "es2015",  
 "moduleResolution": "node"  
 }  
}

以上代码中的 "module": "es2015" 是为了 Tree Sharking 优化生效，阅读 *4-10 使用 TreeSharking* 进一步了解。

修改 App.vue 脚本部分内容如下：

<!--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" 是为了指明代码的语法是 TypeScript。

修改 main.ts 执行入口文件为如下：

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

由于 TypeScript 不认识 .vue 结尾的文件，为了让其支持 import App from './App.vue' 导入语句，还需要以下类型定义文件 vue-shims.d.ts：

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

除此之外还需要按照新引入的依赖：

npm i -D ts-loader typescript

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-7使用Vue框架TypeScript.zip)*

# 3-8 使用 Angular2 框架

## 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。 Angular2 相比于 React 和 Vue 来说要复杂很多，这三者的出发点都是组件化和数据驱动视图，但 Angular2 多出了这些思想：

* 模块（NgModule）这里的模块不是指 JavaScript 或者其它编程语言里的模块化，而是指 Angular2 里提出的独有的用法。
* 注解（Decorator）通过注解语法 @XXX 来给一个 Class 附加元数据。
* 服务（Service）按照功能划分，把可以复用的重复代码分装成一个服务以方便给其它模块使用。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，可以用来减低代码之间的耦合度。

Angular2 引入的这些思想都是为了分解和简化大型项目的难度，但对于开发小项目来说这些可能都是累赘，对于初学者来说可能会难以掌握。 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，在项目中使用 TypeScript 相比 JavaScript 开发体验会好很多。

看到这你可能会被 Angular2 的复杂所吓到，先来看看如何用 Angular2 来开发 Hello,Webpack。

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

import {Component} from '@angular/core';  
  
// 通过注解的方式描述清楚了这个视图组件所需的模版、样式、数据、逻辑。  
@Component({  
 // 标签名称  
 selector: 'app-root',  
 // HTML 模版  
 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，并把它渲染到 DOM 中去。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 所依赖的其它 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 应用启动后会去解析当前 DOM 树，找出名叫 app-root 的 HTML 标签，并以这个标签为 Angular2 应用的运行容器， 所以还需要改造 index.html 加入 app-root HTML 标签，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 跑起来，需要安装以下模块：

# Angular2 框架基础核心模块  
npm i -S @angular/core @angular/common  
# Angular2 框架浏览器环境运行库，类似于 react-dom  
npm i -S @angular/platform-browser   
# 要让 Angular2 正常跑起来所依赖的运行环境和 polyfill  
npm i -S core-js rxjs zone.js   
# 在浏览器里运行过程中动态的编译 HTML 模版  
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 依赖很多，使用繁杂。

## 接入 Webpack

由于 Angular2 应用采用 TypeScript 开发，构建和前面讲过的*(3.2 使用 TypeScript 语言* 类似，不同在于 tsconfig.json 配置。 由于 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，配置需要修改为如下：

{  
 "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/\*"  
 ]  
}

其它配置保持和在*(3.2 使用 TypeScript 语言*的一致，安装好前面提到的 Angular2 框架依赖的模块后，重新执行构建打开网页，你会看到由 Angular2 渲染出来的 Hello,Webpack。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-8使用Angular2框架.zip)*

# 3-9 为单页应用生成 HTML

## 引入问题

在*3-6 使用 React 框架*中，是用最简单的 Hello,Webpack 作为例子让大家理解， 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。接下来举一个实际应用，要求如下：

1. 项目采用 ES6 语言加 React 框架。
2. 给页面加入 *Google Analytics(https://analytics.google.com/analytics/web/)*，这部分代码需要内嵌进 HEAD 标签里去。
3. 给页面加入 *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。
4. 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 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 JS-->  
<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而格式化了 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，部分文件的文件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 地址也被正常的注入到了 HTML 中。 如果你还采用手写 index.html 文件去完成以上要求，这就会使工作变得复杂、易错，项目难以维护。 本节教你如何自动化的生成这个符合要求的 index.html。

## 解决方案

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

首先，修改 Webpack 配置为如下：

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',// 给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，  
 // 该目录下的文件都是采用的 ES5 语法，没必要再通过 Babel 去转换  
 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 模版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`,// 给输出的 CSS 文件名称加上 Hash 值  
 }),  
 new DefinePlugin({  
 // 定义 NODE\_ENV 环境变量为 production 去除代码中的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,  
 // 删除所有的 `console` 语句，可以兼容ie浏览器  
 drop\_console: true,  
 // 内嵌定义了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,  
 }  
 }),  
 ],  
};

以上配置中，大多数都是按照前面已经讲过的内容增加的配置，例如：

* 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，压缩 CSS 文件；
* 定义 NODE\_ENV 环境变量为 production 去除代码中的开发时才需要的部分；
* 给输出的文件名称加上 Hash 值；
* 压缩输出的 JavaScript 代码。

但最核心的部分在于 plugins 里的：

new WebPlugin({  
 template: './template.html', // HTML 模版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' 所指的模版文件 template.html 的内容是：

<html>  
<head>  
 <meta charset="UTF-8">  
 <!--注入 Chunk app 中的 CSS-->  
 <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 JS-->  
<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何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 中，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。

同样的 <script src="./google\_analytics.js?\_inline"></script> 表示 JavaScript 代码来自相对于当前模版文件 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。

除了 \_inline 表示内嵌外，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IE浏览器才需要引入的资源，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 里去。
* WebPlugin 插件还支持一些其它更高级的用法，详情可以访问该*项目主页(https://github.com/gwuhaolin/web-webpack-plugin)*阅读文档。
* -->
* 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 ```

如果按照上节的思路，可能需要为每个单页应用配置一段如下代码：

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

并且把页面对应的入口加入到 enrty 配置项中，就像这样：

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 配置文件，新插入一段以上代码，这会导致构建代码难以维护而且易错。

## 解决方案

上一节中的 *web-webpack-plugin(https://github.com/gwuhaolin/web-webpack-plugin)* 插件也内置了解决这个问题的方法，上一节中只使用了它的 WebPlugin， 这节将使用它的 AutoWebPlugin 来解决以上问题，使用方法非常简单，下面来教你具体如何使用。

项目源码目录结构如下：

├── 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成出下几点要求：

* 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 强制性的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，合理的拆分了代码，又能让新人快速的看懂项目结构，也方便日后的维护。

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，把每一个目录看成一个单页应用  
const autoWebPlugin = new AutoWebPlugin('pages', {  
 template: './template.html', // HTML 模版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  
 // 提取出所有页面公共的代码  
 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 方法可以获取到生成入口配置  
 entry: autoWebPlugin.entry({  
 // 这里可以加入你额外需要的 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，省略了部分和上一节一致的代码，完整代码可以参照上一节或者下载本项目完整代码。

AutoWebPlugin 会找出 pages 目录下的2个文件夹 index 和 login，把这两个文件夹看成两个单页应用。 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 名称就等于文件夹的名称，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"]  
}

但这些事情 AutoWebPlugin 都会自动为你完成，你不用操心，明白大致原理即可。

template.html 模版文件如下：

<html>  
<head>  
 <meta charset="UTF-8">  
 <!--该页面所依赖的其它剩下的 CSS 注入的地方-->  
 <!--STYLE-->  
 <!--注入 google\_analytics 中的 JS 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>  
<!--该页面所依赖的其它剩下的 JavaScript 注入的地方-->  
<!--SCRIPT-->  
<!--Disqus 评论容器-->  
<div id="disqus\_thread"></div>  
</body>  
</html>

注意到模版文件中出现了2个重要的新关键字 <!--STYLE--> 和 <!--SCRIPT-->，它们是什么意思呢？

由于这个模版文件被当作项目中所有单页应用的模版，就不能再像上一节中直接写 Chunk 的名称去引入资源，因为需要被注入到当前页面的 Chunk 名称是不定的，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到生成的 HTML 模版里去。

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，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 JavaScrip 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 JavaScrip 代码 login.css。

由于模版文件 template.html 里没有指出引入这些依赖资源的 HTML 语句，插件会自动的把剩下依赖的资源按照不同类型注入到 <!--STYLE--> 和 <!--SCRIPT--> 所在的位置。

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

如果后续有新的页面需要开发，只需要在 pages 目录下新建一个目录，目录名称取为输出 HTML 文件的名称，目录下放这个页面相关的代码即可，无需改动构建代码。

由于 AutoWebPlugin 是间接的通过上一节提到的 WebPlugin 实现的，WebPlugin 支持的功能 AutoWebPlugin 都支持。

AutoWebPlugin 插件还支持一些其它更高级的用法，详情可以访问该*项目主页(https://github.com/gwuhaolin/web-webpack-plugin)*阅读文档。

本实例*提供项目完整代码(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。

## 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有：

* 搜索引擎无法收录你的网页，因为展示出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到这些数据。
* 对于复杂的单页应用，渲染过程计算量大，对低端移动设备来说可能会有性能问题，用户能明显感知到首屏的渲染延迟。

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到带数据的 HTML，同时也能降低首屏渲染时间。 由于 Node.js 的流行和成熟，以及虚拟 DOM 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，其中最先支持也是最成熟的同构方案是 React。 由于 React 使用者更多，它们之间又很相似，本节只介绍如何用 Webpack 构建 React 同构应用。

同构应用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 去描述原本的 DOM 结构。 在需要更新 DOM 时不直接操作 DOM 树，而是通过更新 JavaScript Object 后再映射成 DOM 操作。

虚拟 DOM 的优点在于：

* 因为操作 DOM 树是高耗时的操作，尽量减少 DOM 树操作能优化网页性能。而 DOM Diff 算法能找出2个不同 Object 的最小差异，得出最小 DOM 操作；
* 虚拟 DOM 的在渲染的时候不仅仅可以通过操作 DOM 树来表示出结果，也能有其它的表示方式，例如把虚拟 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交给 react-dom 模块来负责。

react-dom 在渲染虚拟 DOM 树时有2中方式可选：

* 通过 render() 操作浏览器 DOM 树来表示出结果。
* 通过 renderToString() 计算出表示虚拟 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2份 JavaScript 代码，一份用于在浏览器端运行，一份用于在 Node.js 环境中运行渲染出 HTML。 其中用于在 Node.js 环境中运行的 JavaScript 代码需要注意以下几点：

* 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 内容，渲染出 CSS 代码会增加额外的计算量，影响服务端渲染性能；
* 不能像用于浏览器环境的输出代码那样把 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 规范去引入这些模块。
* 需要通过 CommonJS 规范导出一个渲染函数，以用于在 HTTP 服务器中去执行这个渲染函数，渲染出 HTML 内容返回。

## 解决方案

接下来改造在*3-6使用 React 框架*中介绍的 React 项目，为它增加构建同构应用的功能。

由于要从一份源码构建出2份不同的代码，需要有2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，本节侧重于讲如何构建用于服务端渲染的代码。

配置用于构建浏览器环境代码的 webpack.config.js 文件保留，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 = {  
 // JS 执行入口文件  
 entry: './main\_server.js',  
 // 为了不打包进 Node.js 内置的模块，例如 fs net 模块等  
 target: 'node',  
 // 为了不打包进 node\_modules 目录下的第三方模块  
 externals: [nodeExternals()],  
 output: {  
 // 为了以 CommonJS2 规范导出渲染函数，以给采用 Node.js 编写的 HTTP 服务调用  
 libraryTarget: 'commonjs2',  
 // 把最终可在 Node.js 中运行的代码输出到一个 bundle\_server.js 文件  
 filename: 'bundle\_server.js',  
 // 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 代码不能被打包进用于服务端的代码中去，忽略掉 CSS 文件  
 test: /\.css/,  
 use: ['ignore-loader'],  
 },  
 ]  
 },  
 devtool: 'source-map' // 输出 source-map 方便直接调试 ES6 源码  
};

以上代码有几个关键的地方，分别是：

* target: 'node' 由于输出代码的运行环境是 Node.js，源码中依赖的 Node.js 原生模块没必要打包进去；
* externals: \*nodeExternals()] [webpack-node-externals(https://github.com/liady/webpack-node-externals)\* 的目的是为了防止 node\_modules 目录下的第三方模块被打包进去，因为 Node.js 默认会去 node\_modules 目录下寻找和使用第三方模块；
* {test: /\.css/, use: ['ignore-loader']} 忽略掉依赖的 CSS 文件，CSS 会影响服务端渲染性能，又是做服务端渲没必要的部分；
* libraryTarget: 'commonjs2' 以 CommonJS2 规范导出渲染函数，以供给采用 Node.js 编写的 HTTP 服务器代码调用。

为了最大限度的复用代码，需要调整下目录结构：

把页面的根组件放到一个单独的文件 AppComponent.js，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给渲染入口调用，AppComponent.js 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，分别是用于浏览器端渲染 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 文件内容如下：

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

main\_server.js 文件内容如下：

import React from 'react';  
import { renderToString } from 'react-dom/server';  
import { AppComponent } from './AppComponent';  
  
// 导出渲染函数，以给采用 Node.js 编写的 HTTP 服务器代码调用  
export function render() {  
 // 把根组件渲染成 HTML 字符串  
 return renderToString(<AppComponent/>)  
}

为了能把渲染的完整 HTML 文件返回通过 HTTP 服务返回给请求端，还需要通过用 Node.js 编写一个 HTTP 服务器。 由于本节不专注于将 HTTP 服务器的实现，就采用了 ExpressJS 来实现，http\_server.js 文件内容如下：

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，再拼接下 HTML 模版，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 JS 文件-->  
<script src="./dist/bundle\_browser.js"></script>  
</body>  
</html>  
 `);  
});  
  
// 其它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 服务器后，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，你需要打开浏览器的开发工具中的网络抓包一栏，再重新刷新浏览器后，就能抓到请求 HTML 的请求了，抓包效果图如下：



图3.9.1 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 模版，这说明同构应用的改造完成。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-11构建同构应用.zip)*

# 3-12 构建 Electron 应用

## 认识 Electron

*Electron(https://electron.atom.io)* 可以让你使用开发 Web 的技术去开发跨平台的桌面端应用，由 Github 主导和开源，大家熟悉的 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 去和操作系统交互。 当你在 Electron 应用中的一个窗口操作时，实际上是在操作一个网页。当你的操作需要通过操作系统去完成时，网页会通过 Node.js 去和操作系统交互。

采用这种方式开发桌面端应用的优点有：

* 降低开发门槛，只需掌握网页开发技术和 Node.js 即可，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，Electron 能做到写一份代码在不同的操作系统运行。

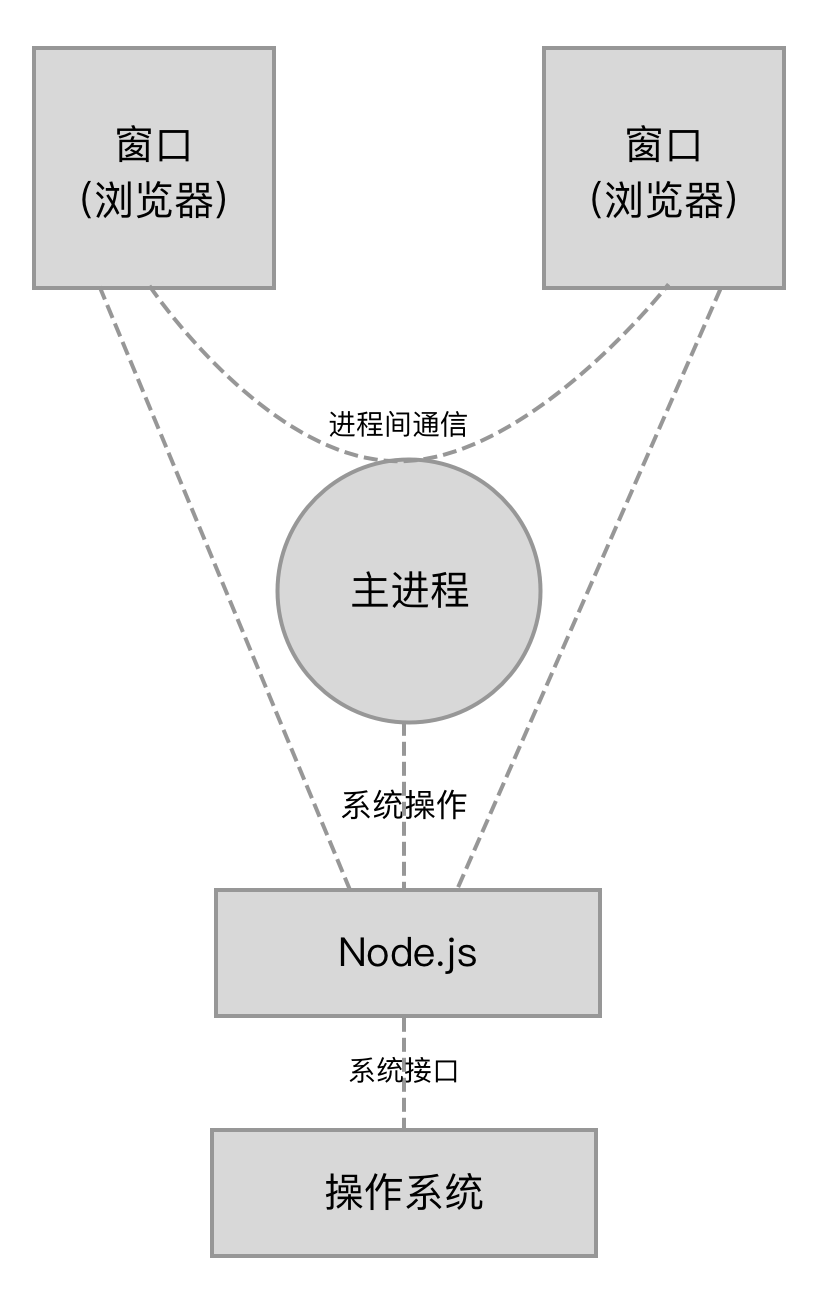
在运行 Electron 应用时，会从启动一个主进程开始。主进程的启动是通过 Node.js 去执行一个入口 JavaScript 文件实现的，这个入口文件 main.js 内容如下：

const { app, BrowserWindow } = require('electron')  
  
// 保持一个对于 window 对象的全局引用，如果你不这样做，  
// 当 JavaScript 对象被垃圾回收， 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，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  
 if (process.platform !== 'darwin') {  
 app.quit()  
 }  
})

主进程启动后会一直驻留在后台运行，你眼睛所看得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，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，具体用法见 *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，窗口之间的通信需要借助主进程传递消息。



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 外，还可以使用 Node.js 提供的 API。

## 接入 Webpack

接下来做一个简单的 Electron 应用，要求为应用启动后显示一个主窗口，在主窗口里有一个按钮，点击这个按钮后新显示一个窗口，且使用 React 开发网页。

由于 Electron 应用中的每一个窗口对应一个网页，所以需要开发2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2个单页应用组成，这和*3-10管理多个单页应用* 中的项目非常相似，让我们来把它改造成一个 Electron 应用。

需要改动的地方如下：

* 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；
* 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 {  
  
 // 在按钮被点击时  
 handleBtnClick() {  
 // 新窗口对应的页面的 URI 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 () {  
 // 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点击事件里通过 electron 库里提供的 API 去新打开一个窗口，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2个可在浏览器里运行的网页，分别对应2个窗口的界面；
* 由于在网页的 JavaScript 代码里可能会有调用 Node.js 原生模块或者 electron 模块，也就是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，构建出的代码不能把这些模块打包进去。

要完成以上要求非常简单，因为 Webpack 内置了对 Electron 的支持。 只需要给 Webpack 配置文件加上一行代码即可，如下：

target: 'electron-renderer',

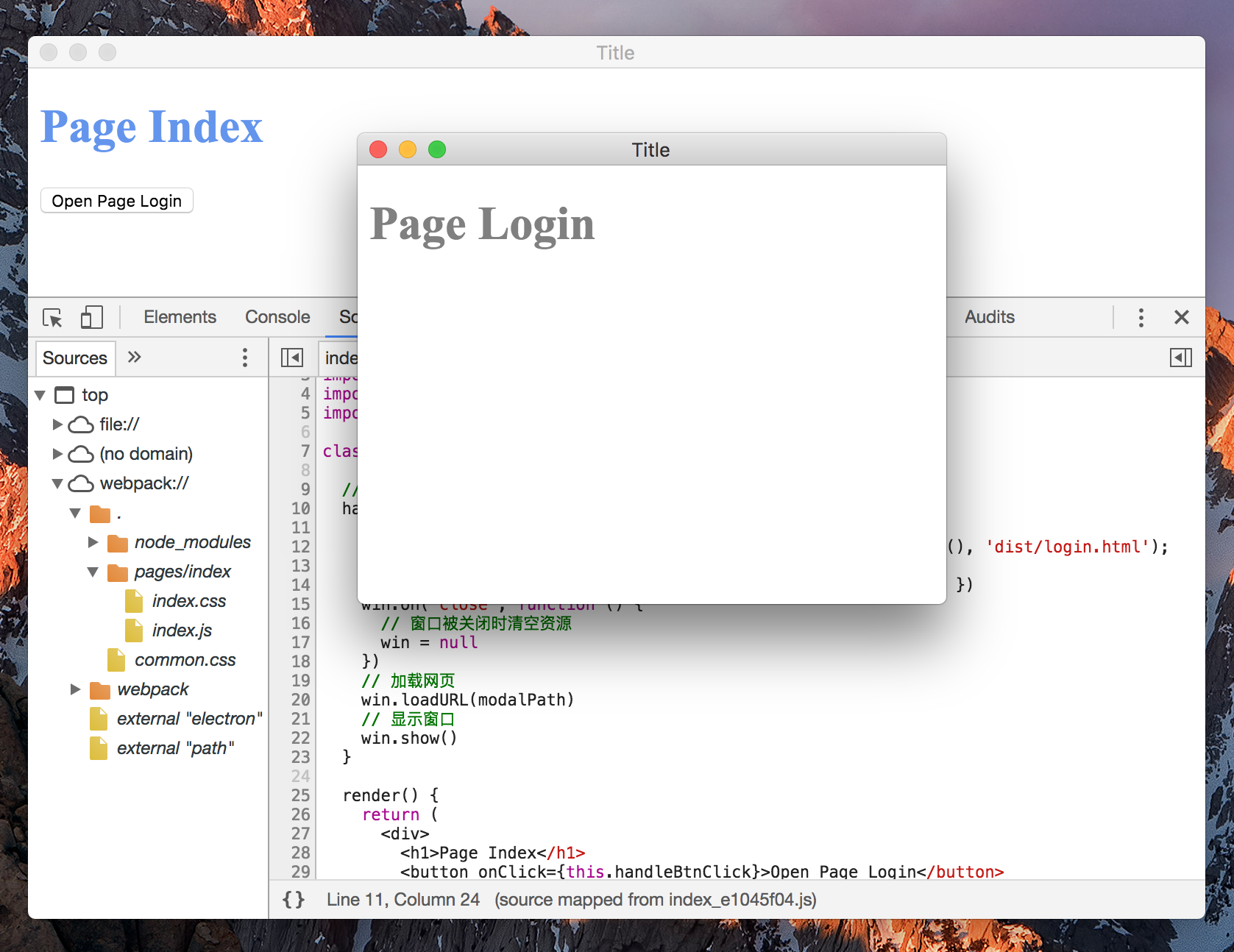
这句配置曾在*2-7其它配置项-Target*中提到，意思是指让 Webpack 构建出用于 Electron 渲染进程用的 JavaScript 代码，也就是这2个窗口需要的网页代码。

以上修改都完成后重新执行 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 electron . 你就能成功看到启动的桌面应用了，效果如图：



Electron 应用运行效果图

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-12构建Electron应用.zip)*

# 3-13 构建 Npm 模块

## 认识 Npm

*Npm(https://www.npmjs.com)* 是目前最大的 JavaScript 模块仓库，里面有来自全世界开发者上传的可复用模块。 虽然大多数情况下你都是这些开放模块的使用者，但有时候你也许会成为贡献者，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：

* 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。想深入了解可以阅读文章 *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 组件模块可能同时需要 JavaScript、CSS、图片文件等。
* 模块中的代码大多采用模块化规范，因为你的这个模块可能依赖其它模块，而且别的模块又可能依赖你的这个模块。因为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。

## 抛出问题

Webpack 不仅可用于构建运行的应用，也可用于构建上传到 Npm 的模块。 接下来用教大家如何用 Webpack 构建一个可上传的 Npm 仓库的 React 组件，具体要求如下：

1. 源代码采用 ES6 写，但发布到 Npm 仓库的需要是 ES5 的，并且遵守 CommonJS 模块化规范。如果发布到 Npm 上去的 ES5 代码是经过转换的，请同时提供 Source Map 以方便调试。
2. 该 UI 组件依赖的其它资源文件例如 CSS 文件也需要包含在发布的模块里。
3. 尽量减少冗余代码，减少发布出去的组件的代码文件大小。
4. 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的去安装。例如不能内嵌 React 库的代码，这样做的目的是在其它组件也依赖 React 库时，防止 React 库的代码被重复打包。

在开始前先看下最终发布到 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)

由于本节的重点不在于 React 而在于 Webpack，所以写一个最简单的 React 组件，src/index.js 内容如下：

import React, { Component } from 'react';  
import './index.css';  
  
// 导出该组件供给其它模块使用  
export default class HelloWebpack extends Component {  
 render() {  
 return <h1 className="hello-component">Hello,Webpack</h1>  
 }  
}

要使用该模块时只需要这样：

// 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/>);

## 使用 Webpack 构建 Npm 模块

接下来用 Webpack 一条条来解决上面抛出问题的4点要求。

#### 对于要求1，可以这样做到：

* 使用 babel-loader 把 ES6 代码转换成 ES5 的代码。
* 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，以供给其它模块导入使用。在*2-2 Output-libraryTarget 和 library* 有介绍这个配置的含义。

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，以供给其它模块导入使用。  
 libraryTarget: 'commonjs2',  
 },  
 // 输出 Source Map  
 devtool: 'source-map',  
};

#### 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

#### 对于要求3，需要注意的是 Babel 在把 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{  
}

在被转换成能正常运行的 ES5 代码时需要以下2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

默认的情况下 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的出现很多次，造成代码冗余。 为了不让这些辅助函数的代重复出现，可以在依赖它们的时候通过 require('babel-runtime/helpers/createClass') 的方式去导入，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/)* 插件就是用来做这个事情的。

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

{  
 "plugins": [  
 [  
 "transform-runtime",  
 {  
 // transform-runtime 默认会自动的为你使用的 ES6 API 注入 polyfill  
 // 假如你在源码中使用了 Promise，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它地方已经注入的其它的 Promise polyfill 库  
 // 所以关闭该功能  
 "polyfill": false  
 }  
 ]  
 ]  
}

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

此步引入了3个新依赖：

# 安装 Webpack 构建所需要的新依赖  
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

#### 对于要求4，需要通过在 *2-7其它配置项* 中介绍过的 Externals 来实现。

Externals 用来告诉 Webpack 要构建的代码中使用了哪些不用被打包的模块，也就是说这些模版是外部环境提供的，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  
 // 这些模块使用外部的，不能被打包进输出的代码里  
 externals: /^(react|babel-runtime)/,  
};

开启以上配置后，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，如下：

[  
 (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不存放打包 react 或者 babel-runtime 模块。

实际上当你在开发 Npm 模块时，不只需要对 react 和 babel-runtime 模块做这样的处理，而是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它模块所依赖。 当一个项目中一个模块被依赖多次时，Webpack 只会将其打包一次，想更深入的了解可以阅读 *5-2 输出文件分析*。

完成以上4步后最终的 Webpack 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，以供给其它模块导入使用。  
 libraryTarget: 'commonjs2',  
 },  
 // 通过正则命中所有以 react 或者 babel-runtime 开头的模块，  
 // 这些模块使用外部的，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 目录下的文件都是采用的 ES5 语法，没必要再通过 Babel 去转换。  
 exclude: path.resolve(\_\_dirname, 'node\_modules'),  
 },  
 {  
 // 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 文件名称  
 filename: 'index.css',  
 }),  
 ],  
 // 输出 Source Map  
 devtool: 'source-map',  
};

重新执行构建后，你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。

## 发布到 Npm

在把构建出的代码发布到 Npm 仓库前，还需要确保你的模块描述文件 package.json 是正确配置的。

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

{  
 "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，这样做的目的是为了方便在 *4-10 中介绍的 Tree Sharking*。

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

如果你想让发布到 Npm 上去的代码保持和源码的目录结构一致，那么用 Webpack 将不在适合。 因为源码中一个个分割的模块化文件，而 Webpack 会把这些模块组合在一起。 虽然 Webpack 输出的文件也可以是采用 CommonJS 模块化语法的，但在有些场景下把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，而保存模块文件的独立能做到只打包进使用到的。 还有就是像 UI 组件库这样由大量独立组件组成的库也和 Lodash 类似。

Webpack 适合于构建完整不可分割的 Npm 模块。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-13构建Npm库.zip)*

# 3-14 构建离线应用

## 认识离线应用

你的网页性能优化的再好，如果网络不好那也会导致网页的体验差。 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，下次访问它时就直接返回本地的文件，就算没有网络连接。

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，对用户来说可以加速网页加载速度，对网站运营者来说可以减少服务器压力以及传输流量费用。

离线应用的核心是离线缓存技术，历史上曾先后出现2种离线离线缓存技术，它们分别是：

1. *AppCache(https://developer.mozilla.org/zh-CN/docs/Web/HTML/Using\_the\_application\_cache)* 又叫 Application Cache，目前已经从 Web 标准中删除，请尽量不要使用它。
2. *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 更加灵活。它也是构建 *PWA(https://developer.mozilla.org/zh-CN/Apps/Progressive)* 应用的关键技术之一。

Service Workers 相比于 AppCache 来说更加灵活，因为它可以通过 JavaScript 代码去控制缓存的逻辑。 由于第1种技术已经废弃，本节只专注于讲解如何用 Webpack 构建使用了 Service Workers 的网页。

## 认识 Service Workers

Service Workers 是一个在浏览器后台运行的脚本，它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 的一个重要功能，通过它能完成离线缓存、编撰响应、过滤响应等功能。

想更深入的了解 Service Workers，推荐阅读文章*服务工作线程：简介(https://developers.google.com/web/fundamentals/getting-started/primers/service-workers?hl=zh-cn)*。

### Service Workers 兼容性

目前 Chrome、Firefox、Opera 都已经全面支持 Service Workers，但对于移动端浏览器就不太乐观了，只有高版本的 Android 支持。 由于 Service Workers 无法通过注入 polyfill 去实现兼容，所以在你打算使用它前请先调查清楚你的网页的运行场景。

判断浏览器是否支持 Service Workers 的最简单的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  
}

### 注册 Service Workers

要给网页接入 Service Workers，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。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注册了的 Service Workers。

### 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出一些事件，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;  
 }  
 // 否则就去用 fetch 下载资源  
 return fetch(event.request);  
 }  
 )  
 );  
});

以上就实现了离线缓存。

### 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，那就需要 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：

1. 每次打开接入了 Service Workers 的网页时，浏览器都会去重新下载 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”。
2. 新 Service Workers 线程将会启动，且将会触发其 install 事件。
3. 当网站上当前打开的页面关闭时，旧 Service Workers 线程将会被终止，新 Service Workers 线程将会取得控制权。
4. 新 Service Workers 线程取得控制权后，将会触发其 activate 事件。

新 Service Workers 线程中的 activate 事件就是最佳的清理旧缓存的时间点，代码如下：

// 当前缓存白名单，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) {  
 // 不在白名单的缓存全部清理掉  
 if (cacheWhitelist.indexOf(cacheName) === -1) {  
 // 删除缓存  
 return caches.delete(cacheName);  
 }  
 })  
 );  
 })  
 );  
});

最终完整的代码 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);  
 })  
 );  
});  
  
// 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;  
 }  
 // 否则就去用 fetch 下载资源  
 return fetch(event.request);  
 }  
 )  
 );  
});  
  
// 新 Service Workers 线程取得控制权后，将会触发其 activate 事件  
self.addEventListener('activate', function (event) {  
 event.waitUntil(  
 caches.keys().then(function (cacheNames) {  
 return Promise.all(  
 cacheNames.map(function (cacheName) {  
 // 不在白名单的缓存全部清理掉  
 if (cacheWhitelist.indexOf(cacheName) === -1) {  
 // 删除缓存  
 return caches.delete(cacheName);  
 }  
 })  
 );  
 })  
 );  
});

## 接入 Webpack

用 Webpack 构建接入 Service Workers 的离线应用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 文件中的代表需要被缓存的文件的 URL 列表 cacheFileList 的值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 没有原生功能能完成以上要求，幸好庞大的社区中已经有人为我们做好了一个插件 *serviceworker-webpack-plugin(https://github.com/oliviertassinari/serviceworker-webpack-plugin)* 可以方便的解决以上问题。 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 JS 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 模版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`,// 给输出的 CSS 文件名称加上 hash 值  
 }),  
 new ServiceWorkerWebpackPlugin({  
 // 自定义的 sw.js 文件所在路径  
 // ServiceWorkerWebpackPlugin 会把文件列表注入到生成的 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,  
 }  
};

以上配置有2点需要注意：

* 由于 Service Workers 必须在 HTTPS 环境下才能拦截网络请求实现离线缓存，使用在 *2-6 DevServer https* 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。

需要修改上面的 sw.js 文件中写成了静态值的 cacheFileList 为如下：

// 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 命令后，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，就能服务接入了 Service Workers 离线缓存的页面了。

## 验证结果

为了验证 Service Workers 和缓存生效了，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：

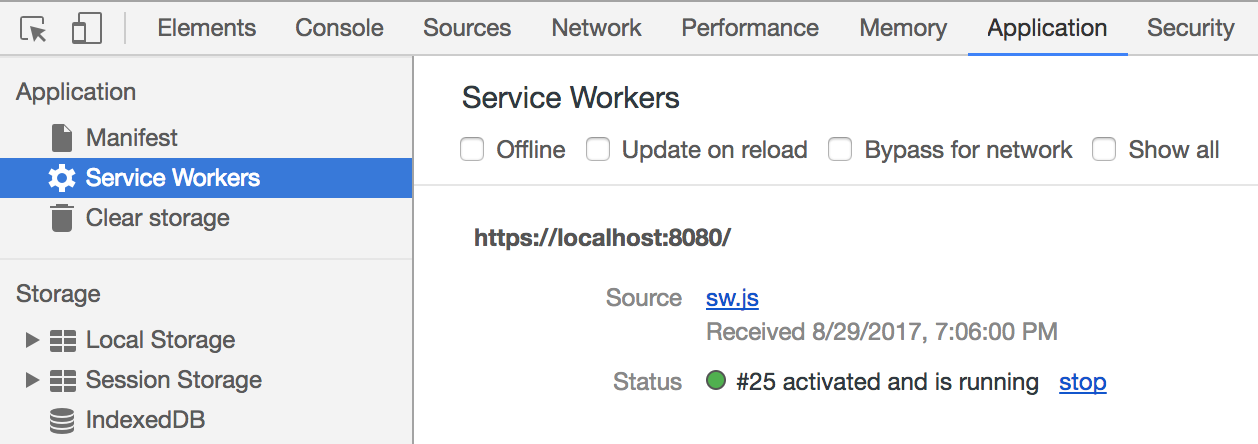


图3.12.1 查看当前页面注册的 Service Workers

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

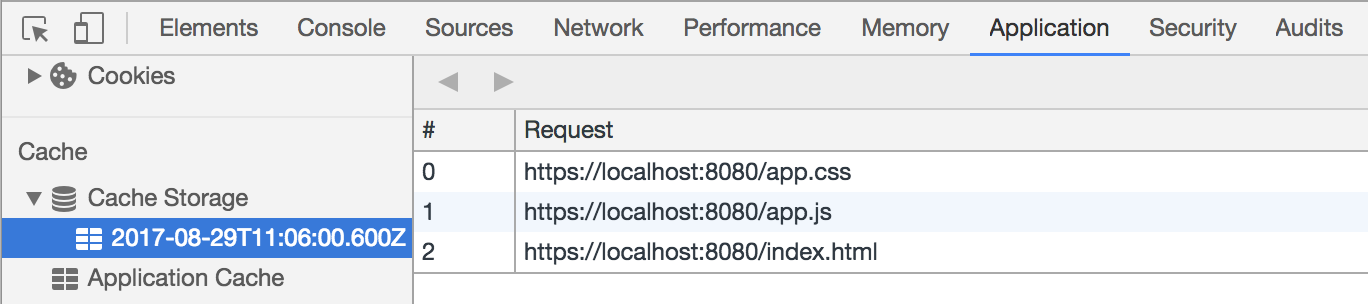


图3.12.2 查看当前页面的 Cache Storage

为了验证网页在离线时能访问的能力，需要在开发者工具中的 Network 一栏中通过 Offline 选项禁用掉网络，再刷新页面能正常访问，并且网络请求的响应都来自 Service Workers，正常的效果如图：

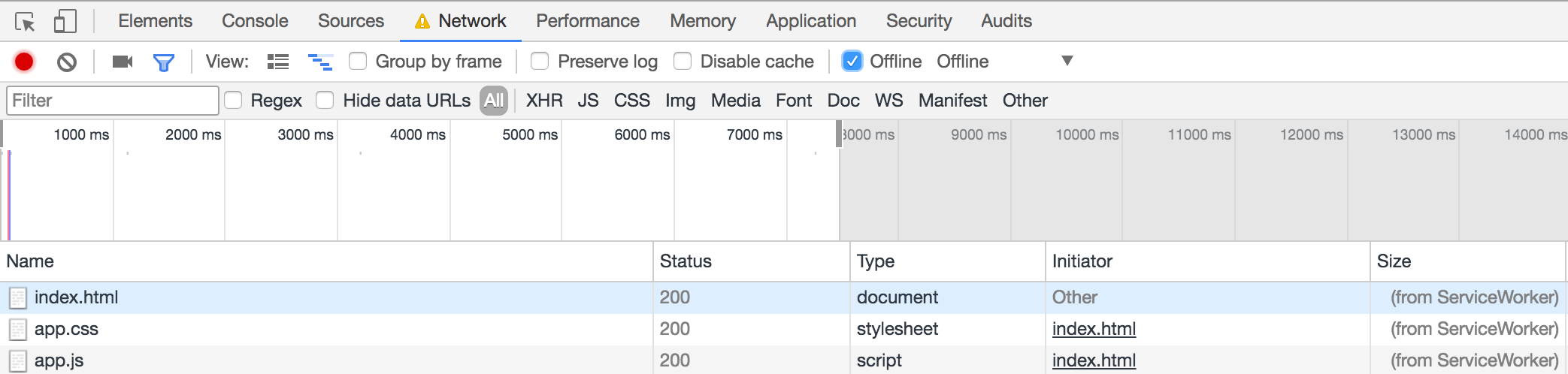


图3.12.3 离线情况下访问页面

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-14构建离线应用.zip)*

# 3-15 搭配 Npm Script

## 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，每一个属性对应一段脚本，以上定义了两个任务 dev 和 pub。 Npm Script 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 命令等同于执行命令 node build.js。

Npm Script 还有一个重要的功能是能运行安装到项目目录里的 node\_modules 里的可执行模块，例如在通过

npm i -D 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，而是要通过命令 ./node\_modules/.bin/webpack 去执行。

Npm Script 能方便的解决这个问题，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 构建只需要通过执行 npm run build 去实现。

## 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：

1. 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，同时还需要开启自动刷新功能。
2. 为了减小发布到线上的代码尺寸，在构建发布到线上的代码需要压缩。
3. 在构建完发布到线上的代码后，需要把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，其中任务3又依赖任务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 来定义上面的3个任务：

"scripts": {  
 "dev": "webpack-dev-server --open",  
 "dist": "NODE\_ENV=production webpack --config webpack\_dist.config.js",  
 "pub": "npm run dist && rsync dist"  
},

含义分别是：

* dev 代表用于开发时执行的任务，通过 DevServer 去启动构建。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去的代码，输出到 dist 目录中。其中的 NODE\_ENV=production 是为了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去的代码，再同步 dist 目录中的文件到发布系统(如何同步文件需根据你所使用的发布系统而定)。所以在开发完后需要发布时只需执行 npm run pub。

使用 Npm Script 的好处是把一连串复杂的流程简化成了一个简单的命令，需要时只需要执行对应的那个简短的命令，而不用去手动的重复整个流程。 这会大大的提高我们的效率和降低出错率。

本实例*提供项目完整代码(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个简单的事，但做检查代码能解决大部分问题。本节将教你如何结合构建做代码检查。

## 代码检查具体是做什么

检查代码和 Code Review 很相似，都是去审视提交的代码可能存在的问题。 但 Code Review 一般通过人去执行，而检查代码是通过机器去执行一些自动化的检查。 自动化的检查代码成本更低，实施代价更小。

检查代码主要检查以下几项：

* 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，保障代码可读性，不把时间浪费在争论如何写代码更好看上；
* 潜在问题：分析出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，分析潜在问题的检查由于情况复杂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。

## 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，下面来分别介绍。

### 检查 JavaScript

目前最常用的 JavaScript 检查工具是 *ESlint(https://eslint.org)* ，它不仅内置了大量常用的检查规则，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

按照到全局后，再在项目目录下执行

eslint init

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

如果你想覆盖默认的检查规则，或者想加入新的检查规则，你需要修改该文件，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，如果你的文件没有通过检查，ESlint 会输出错误原因，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，可以去其官网阅读文档。

### 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

按照到全局，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。

### 检查 CSS

*stylelint(https://stylelint.io)* 是目前最成熟的 CSS 检查工具，内置了大量检查规则的同时也提供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，诸如 SCSS、Less 等。

首先通过

npm i -g stylelint

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

{  
 // 继承 stylelint-config-standard 中的所有检查规则  
 "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没有介绍到，可以访问其官方进一步了解。

目前很多编辑器，例如 Webstorm、VSCode 等，已经集成了以上介绍过的检查工具，编辑器会实时地把检查工具输出的错误显示编辑的源码上。 通过编辑器集成后，你不用通过命令行的方式去定位错误。

## 结合 Webpack 检查代码

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

### 结合 ESLint

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

module.exports = {  
 module: {  
 rules: [  
 {  
 test: /\.js$/,  
 // node\_modules 目录的下的代码不用检查  
 include: /node\_modules/,  
 loader: 'eslint-loader',  
 // 把 eslint-loader 的执行顺序放到最前面，防止其它 Loader 把处理后的代码交给 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。

### 结合 TSLint

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

module.exports = {  
 module: {  
 rules: [  
 {  
 test: /\.js$/,  
 // node\_modules 目录的下的代码不用检查  
 include: /node\_modules/,  
 loader: 'tslint-loader',  
 // 把 tslint-loader 的执行顺序放到最前面，防止其它 Loader 把处理后的代码交给 tslint-loader 去检查  
 enforce: 'pre',  
 },  
 ],  
 },  
}

### 结合 stylelint

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

const StyleLintPlugin = require('stylelint-webpack-plugin');  
  
module.exports = {  
 // ...  
 plugins: [  
 new StyleLintPlugin(),  
 ],  
}

### 一些建议

把代码检查功能整合到 Webpack 中会导致以下问题：

* 由于执行检查步骤计算量大，整合到 Webpack 中会导致构建变慢；
* 整合代码检查到 Webpack 时输出的错误信息中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；
* 把代码检查步骤放到代码提交时，也就是说在代码提交前去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是通过了检查的。

如果你的项目是使用 Git 管理，Git 提供了 Hook 功能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，你需要做的只是在 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"  
 }  
}

precommit 和 prepush 你需要根据自己的情况选择一个，无需两个都设置。

# 3-17 通过 Node.js API 启动 Webpack

Webpack 除了提供可执行的命令行工具外，还提供可在 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加灵活，可用在一些特殊场景，下面将教你如何使用 Webpack 提供的 API。

Webpack 其实是一个 Node.js 应用程序，它全部通过 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。

## 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  
});

如果你的 Webpack 配置写在 webpack.config.js 文件中，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);

## 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，需要获取 Compiler 实例，方法如下：

// 如果不传 callback 回调函数，就会返回一个 Compiler 实例，用于让你去控制启动，而不是像上面那样立即启动  
const compiler = webpack(config);  
  
// 调用 compiler.watch 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 其它配置项* 中介绍过的 *Watch 和 WatchOptions*。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-17通过Node.jsAPI启动Webpack.zip)*

# 3-18 使用 Webpack Dev Middleware

在 *1-6 使用 DevServer* 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);  
// 给 app 注册 webpackMiddleware 中间件  
app.use(webpackMiddleware(compiler));  
// 启动 HTTP 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 通过 Node.js API 启动 Webpack* 中曾介绍过 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，但不会把文件输出到硬盘，而是保存在内存中；
* 往 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 集成到你现有的 HTTP 服务器中，让你现有的 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。

## Webpack Dev Middleware 支持的配置项

在 Node.js 中调用 webpack-dev-middleware 提供的 API 时，还可以给它传入一些配置项，方法如下：

// webpackMiddleware 函数的第二个参数为配置项  
app.use(webpackMiddleware(compiler, {  
 // webpack-dev-middleware 所有支持的配置项  
 // 只有 publicPath 属性为必填，其它都是选填项  
   
 // Webpack 输出资源绑定在 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变化，只会在请求到时再去编译对应的文件，  
 // 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'},  
   
 // 给特定文件后缀的文件添加 HTTP mimeTypes 文件类型头映射  
 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,  
}));

## Webpack Dev Middleware 与模块热替换

DevServer 提供了一个方便的功能，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，而是 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，需要额外的再接入 *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。

第一步：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',  
 // JS 执行入口文件  
 './src/main.js'  
 ],  
 output: {  
 // 把所有依赖的模块合并输出到一个 bundle.js 文件  
 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',  
};

该修改其实就是相当于完成了在 *4-6 开启模块热替换* 中提到的 webpack-dev-server --hot 的工作。

第二步：修改 HTTP 服务器代码 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);  
// 给 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));  
// 把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，监听在 3000 端口  
app.listen(3000, () => {  
 console.info('成功监听在 3000');  
});

第三步：修改网页执行入口文件 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();  
}

第四步：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 那样支持模块热替换的自定义 HTTP 服务了。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-18使用WebpackDevMiddleware.zip)*

# 3-19 加载图片

在网页中不可避免的会依赖图片资源，例如 PNG、JPG、GIF，下面来教你如何用 Webpack 加载图片资源。

## 使用 file-loader

*file-loader(https://github.com/webpack-contrib/file-loader)* 可以把 JavaScript 和 CSS 中导入图片的语句替换成正确的地址，并同时把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 会变成这样：

#app {  
 background-image: url(5556e1251a78c5afda9ee7dd06ad109b.png);  
}

并且在输出目录 dist 中也多出 ./imgs/a.png 对应的图片文件 5556e1251a78c5afda9ee7dd06ad109b.png， 输出的文件名是根据文件内容的计算出的 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 *提供项目完整代码(http://webpack.wuhaolin.cn/3-19加载图片file-loader.zip)*

## 使用 url-loader

*url-loader(https://github.com/webpack-contrib/url-loader)* 可以把文件的内容经过 base64 编码后注入到 JavaScript 或者 CSS 中去。

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 会变成这样：

#app {  
 background-image: url(data:image/png;base64,iVBORw01afer...); /\* 结尾省略了剩下的 base64 编码后的数据 \*/  
}

同理在 JavaScript 中效果也类似。

从上面的例子中可以看出 url-loader 会把根据图片内容计算出的 base64 编码的字符串直接注入到代码中，由于一般的图片数据量巨大， 这会导致 JavaScript、CSS 文件也跟着变大。 所以在使用 url-loader 时一定要注意图片体积不能太大，不然会导致 JavaScript、CSS 文件过大而带来的网页加载缓慢问题。

一般利用 url-loader 把网页需要用到的小图片资源注入内嵌到代码中去，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，该选项用于控制当文件大小小于 limit 时才使用 url-loader，否则使用 fallback 选项中配置的 loader。 相关 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，默认值就是 file-loader   
 fallback: 'file-loader',  
 }  
 }]  
 }  
 ]  
 },  
};

除此之外，你还可以做以下优化：

* 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它二进制类型的资源，例如 PDF、SWF 等等。

本实例 *提供项目完整代码(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，它也成为了 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：

1. SVG 相对于位图更清晰，在任意缩放的情况下后不会破坏图形的清晰度，SVG 能方便的解决适配不同的分辨率屏幕的问题；
2. 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，多数情况下 SVG 会更小；
3. 图形相同的 SVG 比对应的高清图有更好的渲染性能；
4. 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"/>

也就是说可以直接把 SVG 文件当成一张图片来使用，方法和使用图片时完全一样。 所以在 *3-19 加载图片* 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，只需要把 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，除了以上两种方法外还有其它方法，下面来一一说明。

## 使用 raw-loader

*raw-loader(https://github.com/webpack-contrib/raw-loader)* 可以把文本文件的内容读取出来，注入到 JavaScript 或 CSS 中去。

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 SVG 内容

也就是说 svgContent 的内容就等于字符串形式的 SVG，由于 SVG 本身就是 HTML 元素，在获取到 SVG 内容后，可以直接通过以下代码将 SVG 插入到网页中：

window.document.getElementById('app').innerHTML = svgContent;

使用 raw-loader 时相关的 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 去展示 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 中出现 background-image: url(./svgs/activity.svg) 这样的代码是不行的，因为 background-image: url(<svg>...</svg>) 是不合法的。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/3-20加载SVG-raw-loader.zip)*

## 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，以减少 SVG 的文件大小。

使用画图工具，例如 Adobe Illustrator、Sketch，制作的 SVG 图形在导出时有很多由工具自动生成的对于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 处理后会精简成如下：

<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*提供项目完整代码(http://webpack.wuhaolin.cn/3-20加载SVG-svg-inline-loader.zip)*

# 3-21 加载 Source Map

由于在开发过程中经常会使用新语言去开发项目，最后会把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，在调试代码的过程中你会发现生成的代码可读性非常差，这给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，以方便在浏览器中能通过源码调试。 控制 Source Map 输出的 Webpack 配置项是 devtool，其取值可以有很多种选项，下面来一一详细介绍。

|  |  |
| --- | --- |
| 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 Source Map 文件，并且会在 JavaScript 文件末尾追加 //# sourceMappingURL=bundle.js.map |
| hidden-source-map | 和 source-map 类似，但不会在 JavaScript 文件末尾追加 //# sourceMappingURL=bundle.js.map |
| inline-source-map | 和 source-map 类似，但不会额外生成一个单独 Source Map 文件，而是把 Source Map 转换成 base64 编码内嵌到 JavaScript 中 |
| eval-source-map | 和 eval 类似，但会把每个模块的 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 Source Map 文件没有列信息， |
| cheap-module-source-map | 和 cheap-source-map 类似，但会包含 Loader 生成的 Source Map |

其实以上表格只是列举了 devtool 可能取值的一部分， 它的取值其实可以由 source-map、eval、inline、hidden、cheap、module 这六个关键字随意组合而成。 这六个关键字每个都代表一种特性，它们的含义分别是：

* 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 文件所在，这样浏览器就不会自动加载 Source Map；
* inline：把生成的 Source Map 转换成 base64 格式内嵌在 JavaScript 文件中；
* cheap：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；同时来自 Loader 的 Source Map 不会使用；
* module：来自 Loader 的 Source Map 被简单处理成每行一个模块；

## 该如何选择

Devtool 配置项提供的这么多选项看似简单，但很多人搞不清楚它们之间的差别和应用场景。

如果你不关心细节和性能，只是想在不出任何差错的情况下调试源码，可以直接设置成 source-map，但这样会造成两个问题：

* source-map 模式下会输出质量最高最详细的 Source Map，这会造成构建速度缓慢，特别是在开发过程需要频繁修改的时候会增加等待时间；
* source-map 模式下会把 Source Map 暴露出去，如果构建发布到线上的代码的 Source Map 暴露出去就等于源码被泄露；

为了解决以上两个问题，可以这样做：

* 在开发环境下把 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，Source Map 中即使没有列信息也不会影响断点调试；
* 在生产环境下把 devtool 设置成 hidden-source-map，意思是生成最详细的 Source Map，但不会把 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把 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。

## 加载现有的 Source Map

有些从 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，以方便你在使用它们出问题的时候调试它们；

默认情况下 Webpack 是不会去加载这些附加的 Source Map 文件的，Webpack 只会在转换过程中生成 Source Map。 为了让 Webpack 加载这些附加的 Source Map 文件，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$/,  
 // 只加载你关心的目录下的 Source Map，以提升构建速度  
 include: [path.resolve(root, 'node\_modules/some-components/')],  
 use: ['source-map-loader'],  
 // 要把 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，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。 通常会采用 include 去命中只关心的文件。

再安装新引入的依赖：

npm i -D source-map-loader

重启 Webpack 后，你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，会遇到各种各样的需求，虽然前面的小节中已经给出了大部分场景需求的解决方案，但还是很难涵盖所有的可能性。 所以你自己需要有能力去分析遇到的问题，然后去寻找对应的解决方案，你可以按照以下思路去分析和解决问题：

1. 对所面临的问题本身要了解。例如在用 Webpack 去构建 React 应用时你需要先掌握 React 的基础知识。
2. 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，需要把源码转换成 ES5。
3. 找出从现实到目标的可能路径。例如把新语法转换成 ES5 可以使用 Babel 去转换源码。
4. 搜索社区中有没有现成的针对可能路径的 Webpack 集成方案。例如社区中已经有 babel-loader。
5. 如果找不到现成的方案说明你的需求非常特别，这时候你就需要编写自己的 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有以下2点能力很重要：

1. 从一个知识你需要尽可能多的联想到其相关连的知识，这有利于打通你的知识体系,从经验中更快地得出答案。
2. 善于使用搜索引擎去寻找你所面临的问题，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。

最重要的是你需要多实战，自己去解决问题，这有利于加深你的影响和理解，而不是只看不做。

# 第4章 优化

经过前面的学习你已经能用 Webpack 解决常见的问题，但还有很多可以优化的点你可能还不知道。 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，教你如何优化 Webpack 构建。

## 优化开发体验

优化开发体验的目的是为了提升开发时的效率，其中又可以分为以下几点：

1. **优化构建速度**。在项目庞大时构建耗时可能会变的很长，每次等待构建的耗时加起来也会是个大数目。

* *4-1 缩小文件搜索范围*
* *4-2 使用 DllPlugin*
* *4-3 使用 HappyPack*
* *4-4 使用 ParallelUglifyPlugin*

1. **优化使用体验**。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。

* *4-5 使用自动刷新*
* *4-6 开启模块热替换*

## 优化输出质量

优化输出质量的目的是为了给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，这可能关系到你的产品的生死。

优化输出质量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：

1. **减少用户能感知到的加载时间**，也就是首屏加载时间。

* *4-7 区分环境*
* *4-8 压缩代码*
* *4-9 CDN 加速*
* *4-10 使用 Tree Sharking*
* *4-11 提取公共代码*
* *4-12 按需加载*

1. **提升流畅度**，也就是提升代码性能。

* *4-13 使用 Prepack*
* *4-14 开启 Scope Hoisting*

优化的关键是找出问题所在，这样才能一针见血，*4-15 输出分析* 教你如何利用工具快速找出问题所在。

*4-16 优化总结* 对以上的优化方法做一个总结。

# 4-1 缩小文件搜索范围

Webpack 启动后会从配置的 Entry 出发，解析出文件中的导入语句，再递归的解析。 在遇到导入语句时 Webpack 会做两件事情：

1. 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。
2. 根据找到的要导入文件的后缀，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 去处理。

以上两件事情虽然对于处理一个文件非常快，但是当项目大了以后文件量会变的非常多，这时候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。

## 优化 loader 配置

由于 Loader 对文件的转换操作很耗时，需要让尽可能少的文件被 Loader 处理。

在*2-3 Resolve* 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。 为了尽可能少的让文件被 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?$/，提升正则表达式性能  
 test: /\.js$/,  
 // 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'),  
 },  
 ]  
 },  
};

你可以适当的调整项目的目录结构，以方便在配置 Loader 时通过 include 去缩小命中范围。

## 优化 resolve.modules 配置

在*2-4 Resolve* 中介绍过 resolve.modules 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下的 ./node\_modules 目录下去找想找的模块，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下的 ./node\_modules 目录下时，没有必要按照默认的方式去一层层的寻找，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')]  
 },  
};

## 优化 resolve.mainFields 配置

在*2-4 Resolve* 中介绍过 resolve.mainFields 用于配置第三方模块使用哪个入口文件。

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，其中有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因为有些模块可以同时用在多个环境中，准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。 它的 package.json 中就有2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，对应关系如下：

* 当 target 为 web 或者 webworker 时，值是 ["browser", "module", "main"]
* 当 target 为其它情况时，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，在你明确第三方模块的入口文件描述字段时，你可以把它设置的尽量少。 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，可以这样配置 Webpack：

module.exports = {  
 resolve: {  
 // 只采用 main 字段作为入口文件描述字段，以减少搜索步骤  
 mainFields: ['main'],  
 },  
};

使用本方法优化时，你需要考虑到所有运行时依赖的第三方模块的入口文件描述字段，就算有一个模块搞错了都可能会造成构建出的代码无法正常运行。

## 优化 resolve.alias 配置

在*2-4 Resolve* 中介绍过 resolve.alias 配置项通过别名来把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。
* 一套是把 React 所有相关的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 是用于开发环境，里面包含检查和警告的代码。dist/react.min.js 是用于线上环境，被最小化了。

默认情况下 Webpack 会从入口文件 ./node\_modules/react/react.js 开始递归的解析和处理依赖的几十个文件，这会时一个耗时的操作。 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 Webpack 配置如下：

module.exports = {  
 resolve: {  
 // 使用 alias 把导入 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 库外，大多数库发布到 Npm 仓库中时都会包含打包好的完整文件，对于这些库你也可以对它们配置 alias。

但是对于有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你的项目可能永远用不上。 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，例如 *lodash(https://github.com/lodash/lodash)*，你的项目可能只用到了其中几个工具函数，你就不能使用本方法去优化，因为这会导致你的输出代码中包含很多永远不会执行的代码。

## 优化 resolve.extensions 配置

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。 在*2-4 Resolve* 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，或者正确的后缀在越后面，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 时你需要遵守以下几点，以做到尽可能的优化构建性能：

* 后缀尝试列表要尽可能的小，不要把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，以做到尽快的退出寻找过程。
* 在源码中写导入语句时，要尽可能的带上后缀，从而可以避免寻找过程。例如在你确定的情况下把 require('./data') 写成 require('./data.json')。

相关 Webpack 配置如下：

module.exports = {  
 resolve: {  
 // 尽可能的减少后缀尝试的可能性  
 extensions: ['js'],  
 },  
};

## 优化 module.noParse 配置

在*2-3 Module* 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，例如 jQuery 、ChartJS， 它们庞大又没有采用模块化标准，让 Webpack 去解析这些文件耗时又没有意义。

在上面的 *优化 resolve.alias 配置* 中讲到单独完整的 react.min.js 文件就没有采用模块化，让我们来通过配置 module.noParse 忽略对 react.min.js 文件的递归解析处理， 相关 Webpack 配置如下：

const path = require('path');  
  
module.exports = {  
 module: {  
 // 独完整的 `react.min.js` 文件就没有采用模块化，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了，在根据自己项目的需要去按照以上方法改造后，你的构建速度一定会有所提升。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/4-1缩小文件搜索范围.zip)*

# 4-2 使用 DllPlugin

## 认识 DLL

在介绍 *DllPlugin(https://webpack.js.org/plugins/dll-plugin/)* 前先给大家介绍下 DLL。 用过 Windows 系统的人应该会经常看到以 .dll 为后缀的文件，这些文件称为**动态链接库**，在一个动态链接库中可以包含给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：

* 把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去。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被打包，而是去动态链接库中获取。
* 页面依赖的所有动态链接库需要被加载。

为什么给 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要编译一次，在之后的构建过程中被动态链接库包含的模块将不会在重新编译，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，只要不升级这些模块的版本，动态链接库就不用重新编译。

## 接入 Webpack

Webpack 已经内置了对动态链接库的支持，需要通过2个内置的插件接入，它们分别是：

* DllPlugin 插件：用于打包出一个个单独的动态链接库文件。
* DllReferencePlugin 插件：用于在主要配置文件中去引入 DllPlugin 插件打包出的动态链接库文件。

下面以基本的 React 项目为例，为其接入 DllPlugin，在开始前先来看下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，分别是：

* 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，也就是 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 函数代码  
}([  
 function(module, exports, \_\_webpack\_require\_\_) {  
 // 模块 ID 为 0 的模块对应的代码  
 },  
 function(module, exports, \_\_webpack\_require\_\_) {  
 // 模块 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，这些模块存放在一个数组里，用数组的索引号作为 ID。 并且还通过 \_dll\_react 变量把自己暴露在了全局中，也就是可以通过 window.\_dll\_react 可以访问到它里面包含的模块。

其中 polyfill.manifest.json 和 react.manifest.json 文件也是由 DllPlugin 生成出，用于描述动态链接库文件中包含哪些模块， 以 react.manifest.json 文件为例，其文件内容大致如下：

{  
 // 描述该动态链接库文件暴露在全局的变量名称  
 "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 文件中包含了哪些模块，以及每个模块的路径和 ID。

main.js 文件是编译出来的执行入口文件，当遇到其依赖的模块在 dll.js 文件中时，会直接通过 dll.js 文件暴露出的全局变量去获取打包在 dll.js 文件的模块。 所以在 index.html 文件中需要把依赖的两个 dll.js 文件给加载进去，index.html 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，接下来教你如何实现。

### 构建出动态链接库文件

构建输出的以下这四个文件

├── polyfill.dll.js  
├── polyfill.manifest.json  
├── react.dll.js  
└── react.manifest.json

和以下这一个文件

├── 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，用于给主构建使用。新建一个 Webpack 配置文件 webpack\_dll.config.js 专门用于构建它们，文件内容如下：

const path = require('path');  
const DllPlugin = require('webpack/lib/DllPlugin');  
  
module.exports = {  
 // JS 执行入口文件  
 entry: {  
 // 把 React 相关的放到一个单独的动态链接库  
 react: ['react', 'react-dom'],  
 // 把项目需要所有的 polyfill 放到一个单独的动态链接库  
 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',  
 // 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，例如对应 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 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'),  
 }),  
 ],  
};

### 使用动态链接库文件

构建出的动态链接库文件用于给其它地方使用，在这里也就是给执行入口使用。

用于输出 main.js 的主 Webpack 配置文件内容如下：

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',  
 // 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 文件读取 name 字段的值， 把值的内容作为在从全局变量中获取动态链接库中内容时的全局变量名。

### 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把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。

执行构建时流程如下：

1. 如果动态链接库相关的文件还没有编译出来，就需要先把它们编译出来。方法是执行 webpack --config webpack\_dll.config.js 命令。
2. 在确保动态链接库存在时，才能正常的编译出入口执行文件。方法是执行 webpack 命令。这时你会发现构建速度有了非常大的提升。

本实例*提供项目完整代码(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，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 需要处理的任务需要一件件挨着做，不能多个事情一起做。

文件读写和计算操作是无法避免的，那能不能让 Webpack 同一时刻处理多个任务，发挥多核 CPU 电脑的威力，以提升构建速度呢？

*HappyPack(https://github.com/amireh/happypack)* 就能让 Webpack 做到这点，它把任务分解给多个子进程去并发的执行，子进程处理完后再把结果发送给主进程。

由于 JavaScript 是单线程模型，要想发挥多核 CPU 的能力，只能通过多进程去实现，而无法通过多线程实现。

## 使用 HappyPack

分解任务和管理线程的事情 HappyPack 都会帮你做好，你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$/,  
 // 把对 .js 文件的处理转交给 id 为 babel 的 HappyPack 实例  
 use: ['happypack/loader?id=babel'],  
 // 排除 node\_modules 目录下的文件，node\_modules 目录下的文件都是采用的 ES5 语法，没必要再通过 Babel 去转换  
 exclude: path.resolve(\_\_dirname, 'node\_modules'),  
 },  
 {  
 // 把对 .css 文件的处理转交给 id 为 css 的 HappyPack 实例  
 test: /\.css$/,  
 use: ExtractTextPlugin.extract({  
 use: ['happypack/loader?id=css'],  
 }),  
 },  
 ]  
 },  
 plugins: [  
 new HappyPack({  
 // 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  
 id: 'babel',  
 // 如何处理 .js 文件，用法和 Loader 配置中一样  
 loaders: ['babel-loader?cacheDirectory'],  
 // ... 其它配置项  
 }),  
 new HappyPack({  
 id: 'css',  
 // 如何处理 .css 文件，用法和 Loader 配置中一样  
 loaders: ['css-loader'],  
 }),  
 new ExtractTextPlugin({  
 filename: `[name].css`,  
 }),  
 ],  
};

以上代码有两点重要的修改：

* 在 Loader 配置中，所有文件的处理都交给了 happypack/loader 去处理，使用紧跟其后的 querystring ?id=babel 去告诉 happypack/loader 去选择哪个 HappyPack 实例去处理文件。
* 在 Plugin 配置中，新增了两个 HappyPack 实例分别用于告诉 happypack/loader 去如何处理 .js 和 .css 文件。选项中的 id 属性的值和上面 querystring 中的 ?id=babel 相对应，选项中的 loaders 属性和 Loader 配置中一样。

在实例化 HappyPack 插件的时候，除了可以传入 id 和 loaders 两个参数外，HappyPack 还支持如下参数：

* 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，类型必须是整数。
* 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： ```js const HappyPack = require('happypack'); // 构造出共享进程池，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 是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，你需要给项目安装新的依赖：

npm i -D happypack

安装成功后重新执行构建你就会看到以下由 HappyPack 输出的日志：

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.

说明你的 HappyPack 配置生效了，并且可以得知 HappyPack 分别启动了3个子进程去并行的处理任务。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/4-3使用HappyPack.zip)*

## HappyPack 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巨多，而且这些转换操作都只能一个个挨着处理。 HappyPack 的核心原理就是把这部分任务分解到多个进程去并行处理，从而减少了总的构建时间。

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，收集到了这些文件的处理权后 HappyPack 就好统一分配了。

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，并且可以指定如何给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，核心调度器会把一个个任务分配给当前空闲的子进程，子进程处理完毕后把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。

用过 UglifyJS 的你一定会发现在构建用于开发环境的代码时很快就能完成，但在构建用于线上的代码时构建一直卡在一个时间点迟迟没有反应，其实卡住的这个时候就是在进行代码压缩。

由于压缩 JavaScript 代码需要先把代码解析成用 Object 抽象表示的 AST 语法树，再去应用各种规则分析和处理 AST，导致这个过程计算量巨大，耗时非常多。

为什么不把在*4-3 使用 HappyPack*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？

*ParallelUglifyPlugin(https://github.com/gdborton/webpack-parallel-uglify-plugin)* 就做了这个事情。 当 Webpack 有多个 JavaScript 文件需要输出和压缩时，原本会使用 UglifyJS 去一个个挨着压缩再输出， 但是 ParallelUglifyPlugin 则会开启多个子进程，把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 能更快的完成对多个文件的压缩工作。

使用 ParallelUglifyPlugin 也非常简单，把原来 Webpack 配置文件中内置的 UglifyJsPlugin 去掉后，再替换成 ParallelUglifyPlugin，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 JS 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,  
 // 删除所有的 `console` 语句，可以兼容ie浏览器  
 drop\_console: true,  
 // 内嵌定义了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，默认是 /.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返回。cacheDir 用于配置缓存存放的目录路径。默认不会缓存，想开启缓存请设置一个目录路径。
* workerCount：开启几个子进程去并发的执行压缩。默认是当前运行电脑的 CPU 核数减去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，Object 类型，直接透传给 UglifyJS 的参数。
* uglifyES：用于压缩 ES6 代码时的配置，Object 类型，直接透传给 UglifyES 的参数。

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，它们两都出自于同一个项目，并且它们两不能同时使用。

UglifyES 一般用于给比较新的 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，采用 UglifyES 压缩效果会更好。

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，重新执行构建你会发现速度变快了许多。如果设置 cacheDir 开启了缓存，在之后的构建中会变的更快。

本实例*提供项目完整代码(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，需要刷新浏览器让其重新运行最新的代码才行。 虽然这相比于开发原生 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，但我们可以把这个体验优化的更好。 借助自动化的手段，可以把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 把这些功能都内置了，并且还提供多种方案可选。

## 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 使用DevServer\*\* 扩展模块。 文件监听功能是 webpack 模块所带有的。

在*2-7 其它配置项* 中曾介绍过 Webpack 支持文件监听相关的配置项如下：

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，  
 // 防止文件更新太快导致重新编译频率太快。默认 300ms   
 aggregateTimeout: 300,  
 // 判断文件是否发生变化是通过不停的去询问系统指定文件有没有变化实现的  
 // 默认每秒问 1000 次  
 poll: 1000  
 }  
}

要让 Webpack 开启监听模式，有两种方式：

* 在配置文件 webpack.config.js 中设置 watch: true。
* 在执行启动 Webpack 命令时，带上 --watch 参数，完整命令是 webpack --watch。

### 文件监听工作原理

在 Webpack 中监听一个文件发生变化的原理是定时的去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 就是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 就是用于配置这个等待时间。 这样做的目的是因为我们在编辑代码的过程中可能会高频的输入文字导致文件变化的事件高频的发生，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，原理相似，只不过会对列表中的每一个文件都定时的执行检查。 但是这个需要监听的文件列表是怎么确定的呢？ 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，把这些依赖的文件都加入到监听列表中去。 可见 Webpack 这一点还是做的很智能的，不是粗暴的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编辑时间需要占用内存，定时检查周期检查需要占用 CPU 以及文件 IO，所以最好减少需要监听的文件数量和降低检查频率。

### 优化文件监听性能

在明白文件监听工作原理后，就好分析如何优化文件监听性能了。

开启监听模式时，默认情况下会监听配置的 Entry 文件和所有其递归依赖的文件。 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。 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。 所以一个很大的优化点就是忽略掉 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，你的 Webpack 消耗的内存和 CPU 将会大大降低。

有时你可能会觉得 node\_modules 目录下的第三方模块有 bug，想修改下第三方模块的文件在自己的项目中试试。 在这种情况下使用了忽略掉 node\_modules 目录下的文件优化后，你需要重启构建以看到最新效果。 但这种情况毕竟是非常少见的。

除了忽略掉部分文件的优化外，还有如下两种方法：

* watchOptions.aggregateTimeout 值越大性能越好，因为这能降低重新构建的频率。
* watchOptions.poll 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会让你感觉到监听模式的反应和灵敏度降低了。

## 自动刷新浏览器

监听到文件更新后的下一步是去刷新浏览器，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告诉 webpack-dev-server 模块。

### 自动刷新的原理

控制浏览器刷新有三种方法：

1. 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。
2. 往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。
3. 把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，你会看到如下日志：

> 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 中包含了以下七个模块：

[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]

这七个模块就是代理客户端的代码，它们被打包进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你会发现由代理客户端向 DevServer 发起的 WebSocket 连接：

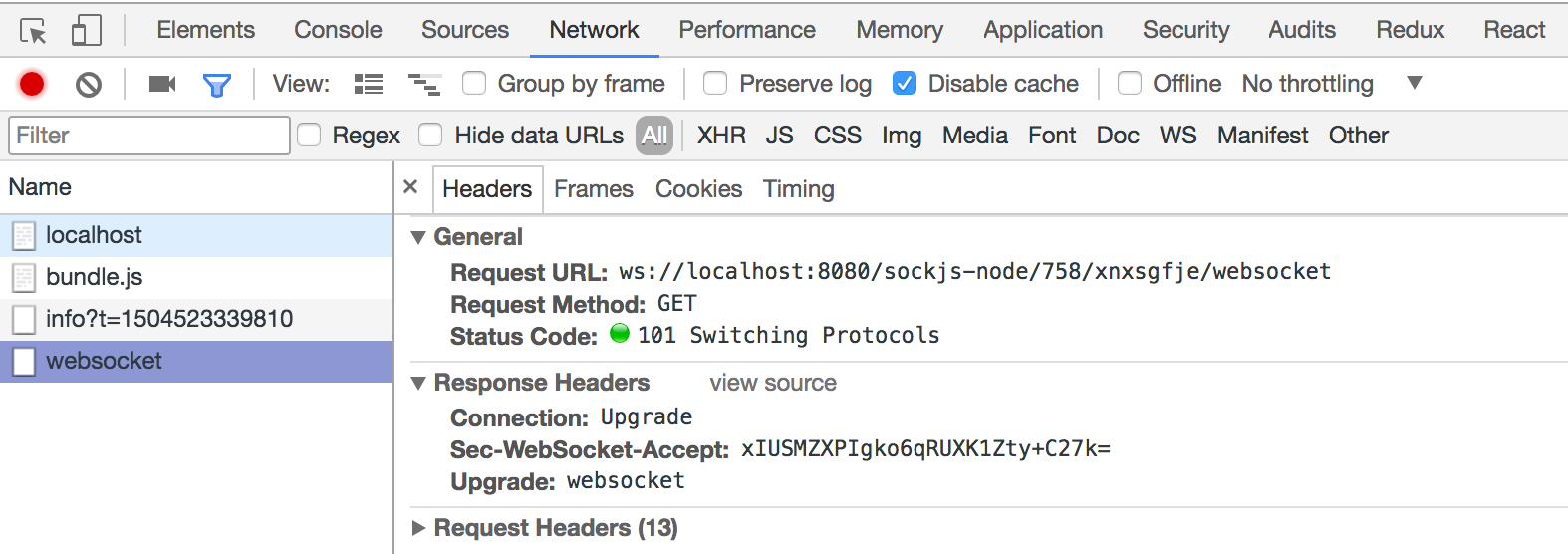


图4.5.1 代理客户端发起 WebSocket 连接

### 优化自动刷新的性能

在*2-6 DevServer*中曾介绍过 devServer.inline 配置项，它就是用来控制是否往 Chunk 中注入代理客户端的，**默认会注入**。 事实上，在开启 inline 时，DevServer 会为每个输出的 Chunk 中注入代理客户端的代码，当你的项目需要输出的 Chunk 有很多个时，这会导致你的构建缓慢。 其实要完成自动刷新，一个页面只需要一个代理客户端就行了，DevServer 之所以粗暴的为每个 Chunk 都注入，是因为它不知道某个网页依赖哪几个 Chunk，索性就全部都注入一个。 网页只要依赖了其中任何一个 Chunk，代理客户端就被注入到网页中去。

这里优化的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，可通过执行命令 webpack-dev-server --inline false（也可以在配置文件中设置），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了

在浏览器中打开网址 http://localhost:8080/webpack-dev-server/ 后，你会看到如下效果：

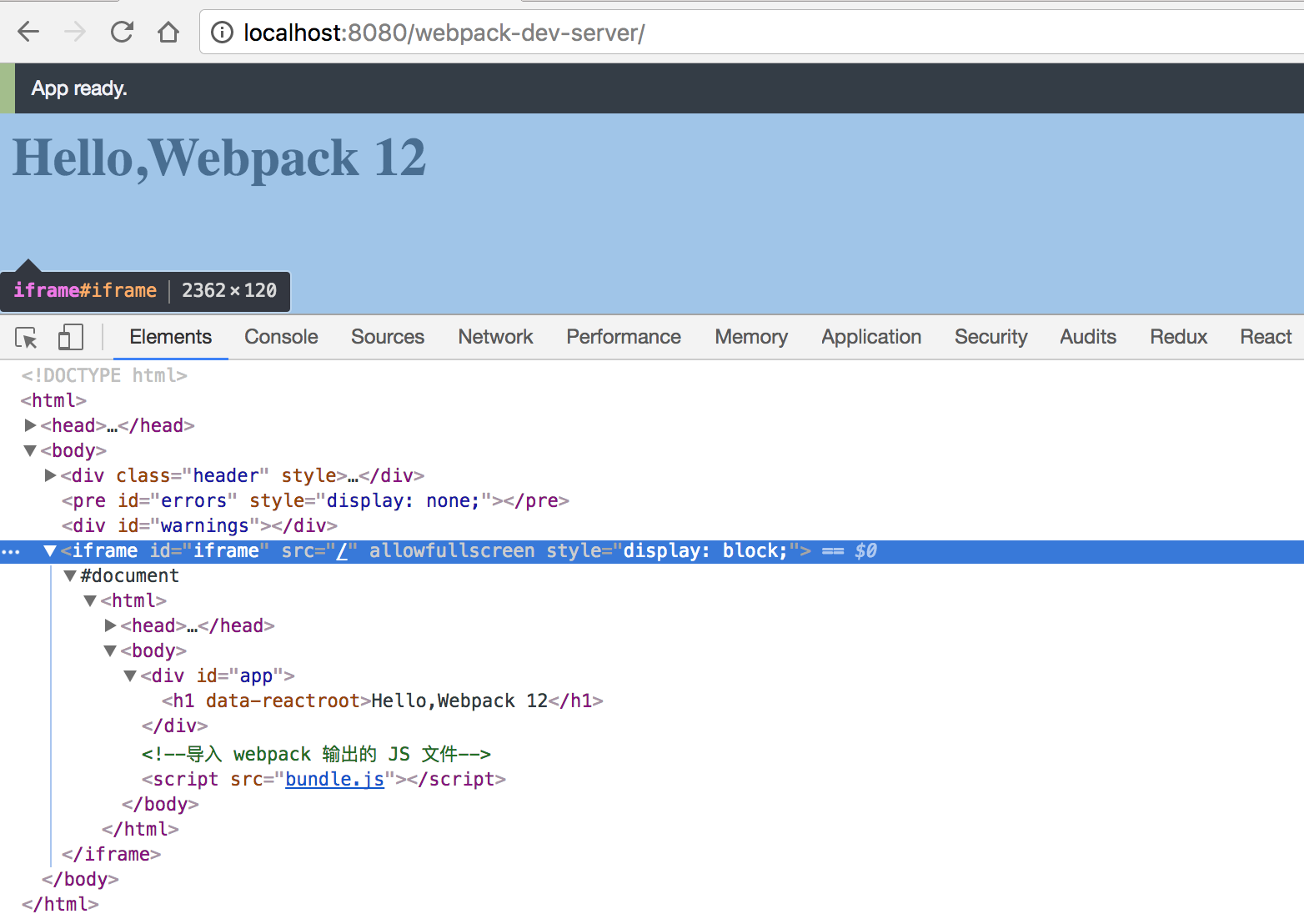


图4.5.2 通过 iframe 自动刷新

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。 同时你会发现构建时间从 1566ms 减少到了 1130ms，说明优化生效了。构建性能提升的效果在要输出的 Chunk 数量越多时会显得越突出。

在你关闭了 inline 后，DevServer 会自动地提示你通过新网址 http://localhost:8080/webpack-dev-server/ 去访问，这点是做的很人心化的。

如果你不想通过 iframe 的方式去访问，但同时又想让网页保持自动刷新功能，你需要手动往网页中注入代理客户端脚本，往 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>

给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记得删除掉这段用于开发环境的代码。

本实例*提供项目完整代码(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，除了在*4-5使用自动刷新*的技术可在不刷新整个网页的情况下做到超灵敏的实时预览。 原理是当一个源码发生变化时，只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有：

* 实时预览反应更快，等待时间更短。
* 不刷新浏览器能保留当前网页的状态，例如在使用 Redux 来管理数据的应用中搭配模块热替换能做到代码更新时 Redux 中的数据还保持不变。

总的来说模块热替换技术很大程度上的提高了开发效率和体验。

## 模块热替换的原理

模块热替换的原理和自动刷新原理类似，都需要往要开发的网页中注入一个代理客户端用于连接 DevServer 和网页， 不同在于模块热替换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，只需在启动时带上参数 --hot，完整命令是 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，实际上当启动时带上 `--hot` 参数，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 其实就是自动为你完成以上配置。

启动后日志如下：

> 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九个文件：

[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]

相比于自动刷新的代理客户端，多出了后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，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 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：

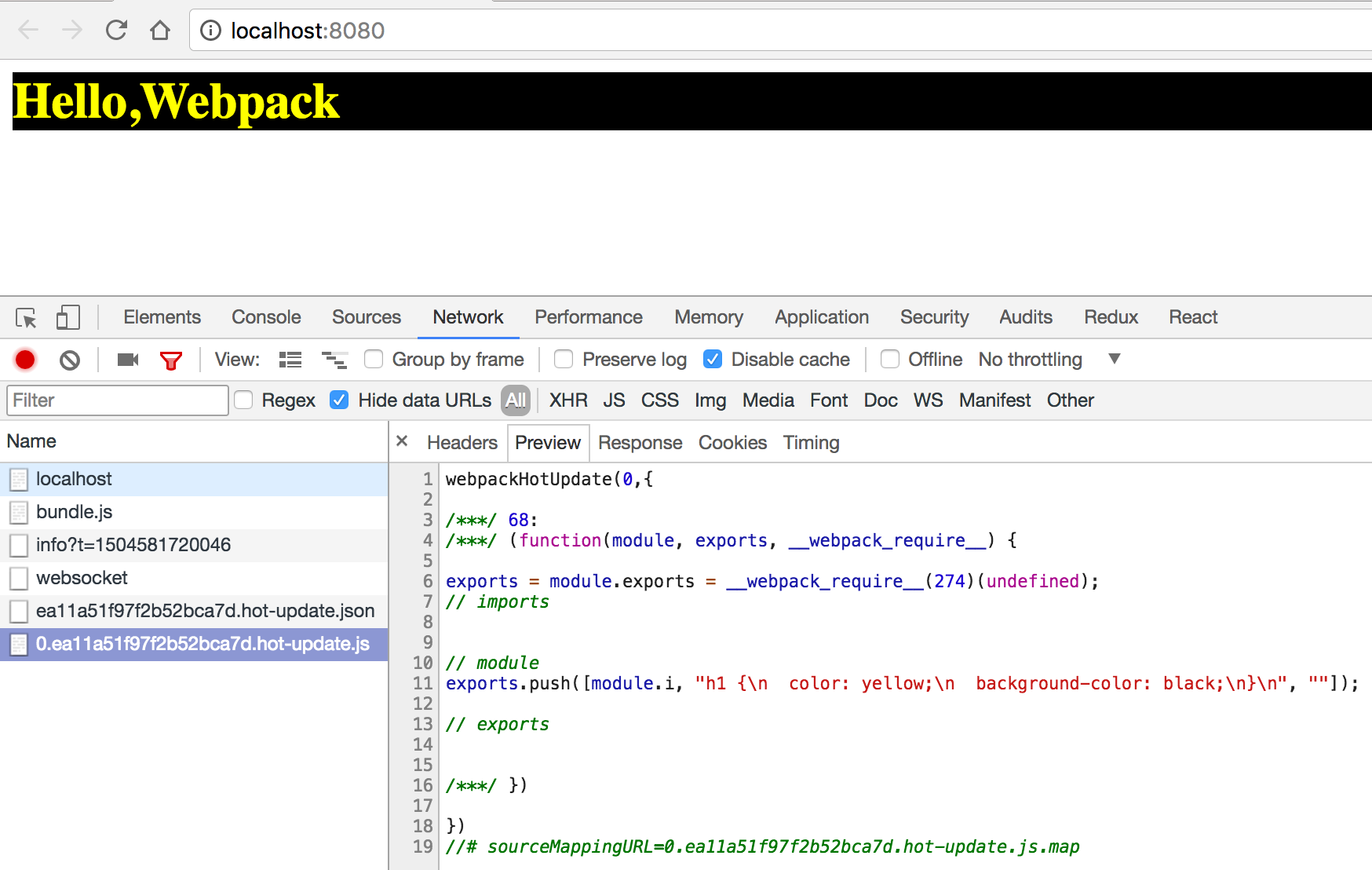


图4.5.3 模块热替换模式下的补丁

可见补丁中包含了 main.css 文件新编译出来 CSS 代码，网页中的样式也立刻变成了源码中描述的那样。

但当你修改 main.js 文件时，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 .js 文件会这样呢？

模块热替换功能为了让使用者灵活的控制老模块被替换时的逻辑，可以在源码中定义一些代码去做相应的处理。

把的 main.js 文件改为如下：

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 函数的第一个参数指出当前文件接受哪些子模块的替换，这里表示只接受 ./AppComponent 这个子模块  
 // 第二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'], () => {  
 // 新的 AppComponent 加载成功后重新执行下组建渲染逻辑  
 render(<AppComponent/>, window.document.getElementById('app'));  
 });  
}

其中的 module.hot 是当开启模块热替换后注入到全局的 API，用于控制模块热替换的逻辑。

现在修改 AppComponent.js 文件，把 Hello,Webpack 改成 Hello,World，你会发现模块热替换生效了。 但是当你编辑 main.js 时，你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？

当子模块发生更新时，更新事件会一层层往上传递，也就是从 AppComponent.js 文件传递到 main.js 文件， 直到有某层的文件接受了当前变化的模块，也就是 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受它，就会直接刷新网页。

那为什么没有地方接受过 .css 文件，但是修改所有的 .css 文件都会触发模块热替换呢？ 原因在于 style-loader 会注入用于接受 CSS 的代码。

请不要把模块热替换技术用于线上环境，它是专门为提升开发效率生的。

## 优化模块热替换

在发生模块热替换时，你会在浏览器的控制台中看到类似这样的日志：

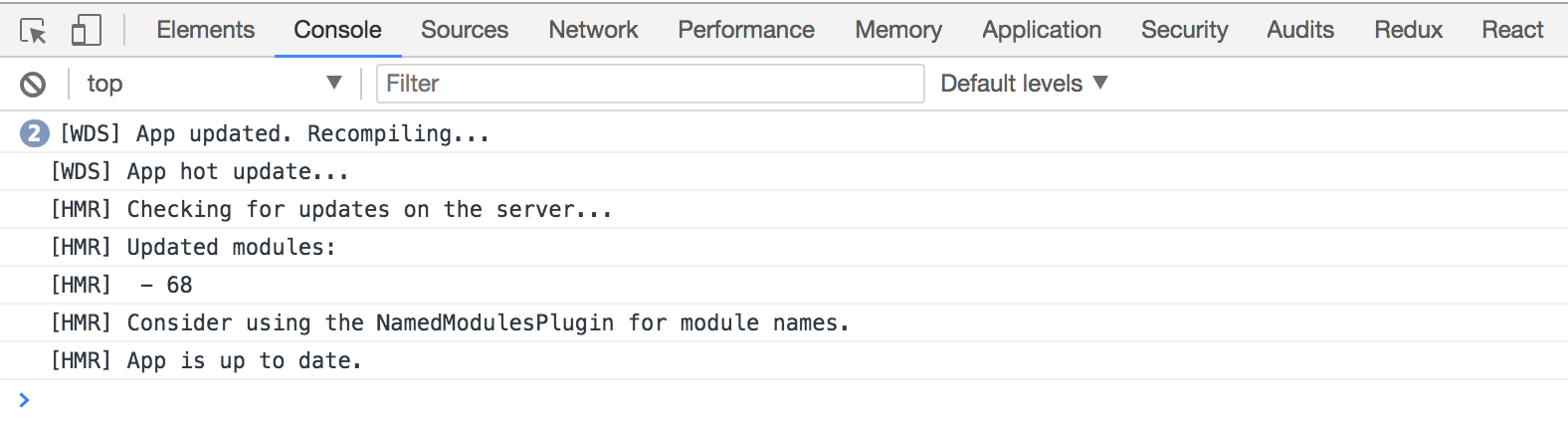


图4.5.4 模块热替换浏览器日志

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，最好是把替换了的模块的名称输出出来。 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(),  
 ],  
};

重启构建后你会发现浏览器中的日志更加友好了：

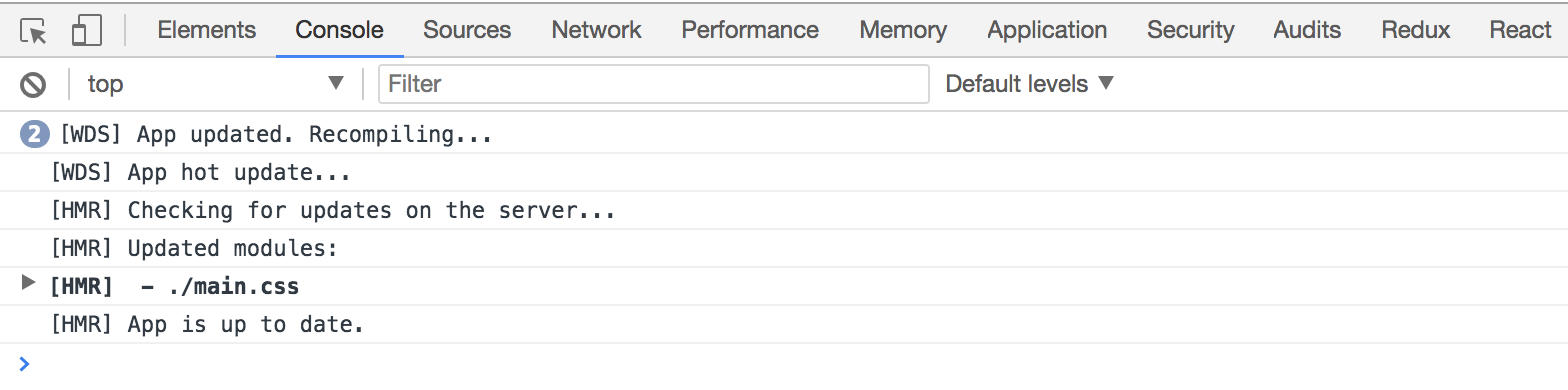


图4.5.4 现实出被替换模块的浏览器日志

除此之外，模块热替换还面临着和自动刷新一样的性能问题，因为它们都需要监听文件变化和注入客户端。 要优化模块热替换的构建性能，思路和在*4-5 使用自动刷新*中提到的很类似：监听更少的文件，忽略掉 node\_modules 目录下的文件。 但是其中提到的关闭默认的 inline 模式手动注入代理客户端的优化方法不能用于在使用模块热替换的情况下， 原因在于模块热替换的运行依赖每个 Chunk 中都必须包含代理客户端的代码去完成新模块替换老模块的工作。

# 4-7 区分环境

## 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：

1. 在开发过程中方便开发调试的环境。
2. 发布到线上给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，但是代码内容却不一样，差异包括：

* 线上代码被通过 *4-8 压缩代码* 中提到的方法压缩过。
* 开发用的代码包含一些用于提示开发者的提示日志，这些日志普通用户不可能去看它。
* 开发用的代码所连接的后端数据接口地址也可能和线上环境不同，因为要避免开发过程中造成对线上数据的影响。

为了尽可能的复用代码，在构建的过程中需要根据目标代码要运行的环境而输出不同的代码，我们需要一套机制在源码中去区分环境。 幸运的是 Webpack 已经为我们实现了这点。

## 如何区分环境

具体区分方法很简单，在源码中通过如下方式：

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');  
}

其大概原理是借助于环境变量的值去判断执行哪个分支。

当你的代码中出现了使用 *process(https://nodejs.org/api/process.html)* 模块的语句时，Webpack 就自动打包进 process 模块的代码以支持非 Node.js 的运行环境。 当你的代码中没有使用 process 时就不会打包进 process 模块的代码。这个注入的 process 模块作用是为了模拟 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，需要给当前运行环境设置环境变量 NODE\_ENV = 'production'，Webpack 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')==='"production"'。

执行构建后，你会在输出的文件中发现如下代码：

if (true) {  
 console.log('你正在线上环境');  
} else {  
 console.log('你正在使用开发环境');  
}

定义的环境变量的值被代入到了源码中，process.env.NODE\_ENV === 'production' 被直接替换成了 true。 并且由于此时访问 process 的语句被替换了而没有了，Webpack 也不会打包进 process 模块了。

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 脚本的方式去定义的环境变量，例如 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，没必要又通过 Shell 脚本的方式去定义一遍。

如果你想让 Webpack 使用通过 Shell 脚本的方式去定义的环境变量，你可以使用 EnvironmentPlugin，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),  
})

## 结合 UglifyJS

其实以上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其实应该直接是：

console.log('你正在线上环境');

Webpack 没有实现去除死代码功能，但是 UglifyJS 可以做这个事情，如何使用请阅读 *4-8 压缩代码* 中的压缩 JavaScript。

## 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外，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，分别是：

1. 开发环境：包含类型检查、HTML 元素检查等等针对开发者的警告日志代码。
2. 线上环境：去掉了所有针对开发者的代码，只保留让 React 能正常运行的部分，以优化大小和性能。

例如 React 源码中有大量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.... ')  
}

如果你不定义 NODE\_ENV=production 那么这些警告日志就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在区分开发环境和线上环境。

本实例 *提供项目完整代码(http://webpack.wuhaolin.cn/4-7区分环境.zip)*

# 4-8 压缩代码

浏览器从服务器访问网页时获取的 JavaScript、CSS 资源都是文本形式的，文件越大网页加载时间越长。 为了提升网页加速速度和减少网络传输流量，可以对这些资源进行压缩。 压缩的方法除了可以通过 GZIP 算法对文件压缩外，还可以对文本本身进行压缩。

对文本本身进行压缩的作用除了有提升网页加载速度的优势外，还具有混淆源码的作用。 由于压缩后的代码可读性非常差，就算别人下载到了网页的代码，也大大增加了代码分析和改造的难度。

下面来一一介绍如何在 Webpack 中压缩代码。

## 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，理解代码含义，从而能做到诸如去掉无效代码、去掉日志输出代码、缩短变量名等优化。

要在 Webpack 中接入 UglifyJS 需要通过插件的形式，目前有两个成熟的插件，分别是：

* UglifyJsPlugin：通过封装 UglifyJS 实现压缩。
* ParallelUglifyPlugin：多进程并行处理压缩，*4-4使用ParallelUglifyPlugin* 中有详细介绍。

由于 ParallelUglifyPlugin 在 *4-4使用ParallelUglifyPlugin* 中介绍过就不再复述， 这里重点介绍如何配置 UglifyJS 以达到最优化的压缩 JavaScript 代码。

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，所有的选型说明可以在 *其官方文档(https://github.com/mishoo/UglifyJS2#minify-options)* 上看到。 由于选项非常多，就挑出一些常用的拿出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。一般不会把压缩后的代码的 Source Map 发送给网站用户的浏览器，而是用于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，默认为是，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：是否剔除代码中所有的 console 语句，默认为不剔除。开启后不仅可以提升代码压缩效果，也可以兼容不支持 console 语句 IE 浏览器。
* collapse\_vars：是否内嵌定义了但是只用到一次的变量，例如把 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： 是否提取出出现多次但是没有定义成变量去引用的静态值，例如把 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为如下：

const UglifyJSPlugin = require('webpack/lib/optimize/UglifyJsPlugin');  
  
module.exports = {  
 plugins: [  
 // 压缩输出的 JS 代码  
 new UglifyJSPlugin({  
 compress: {  
 // 在UglifyJs删除没有用到的代码时不输出警告  
 warnings: false,  
 // 删除所有的 `console` 语句，可以兼容ie浏览器  
 drop\_console: true,  
 // 内嵌定义了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,  
 },  
 output: {  
 // 最紧凑的输出  
 beautify: false,  
 // 删除所有的注释  
 comments: false,  
 }  
 }),  
 ],  
};

从以上配置中可以看出 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 会自动为你注入一个带有默认配置的 UglifyJSPlugin。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/4-8压缩代码-ES5.zip)*

### 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 的代码相比于转换后的 ES5 代码有如下优点：

* 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，我们要尽可能的使用原生的 ES6 代码去运行，而不是转换后的 ES5 代码。

在你用上面所讲的压缩方法去压缩 ES6 代码时，你会发现 UglifyJS 会报错退出，原因是 UglifyJS 只认识 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。 在给 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,  
 // 删除所有的 `console` 语句，可以兼容ie浏览器  
 drop\_console: true,  
 // 内嵌定义了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,  
 },  
 output: {  
 // 最紧凑的输出  
 beautify: false,  
 // 删除所有的注释  
 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，但是其它的比如 babel-preset-react 还是要保留， 因为正是 babel-preset-env 负责把 ES6 代码转换为 ES5 代码。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/4-8压缩代码-ES6.zip)*

## 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，例如：

* margin: 10px 20px 10px 20px 被压缩成 margin: 10px 20px
* color: #ff0000 被压缩成 color:red

还有很多压缩规则可以去其官网查看，通常压缩率能达到 60%。

把 cssnano 接入到 Webpack 中也非常简单，因为 css-loader 已经将其内置了，要开启 cssnano 去压缩代码只需要开启 css-loader 的 minimize 选项。 相关 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  
 // 提取出 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 模版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`,// 给输出的 CSS 文件名称加上 hash 值  
 }),  
 ],  
};

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/4-8压缩代码-CSS.zip)*

# 4-9 CDN 加速

## 什么是 CDN

虽然前面通过了压缩代码的手段来减小网络传输大小，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。 导致这个问题的根本是网络传输过程耗时大，CDN 的作用就是加速网络传输。

CDN 又叫内容分发网络，通过把资源部署到世界各地，用户在访问时按照就近原则从离用户最近的服务器获取资源，从而加速资源的获取速度。 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，其大致原理可以如下：

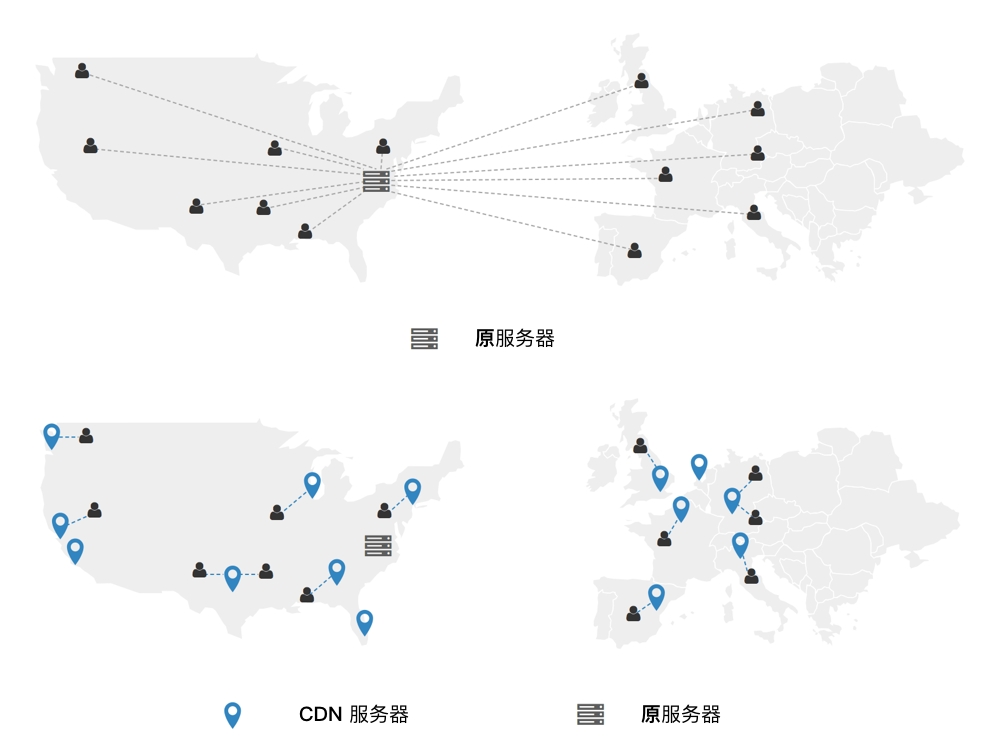


图4-9-1 CDN 原理

在本节中你不必理解 CDN 的具体运行流程和实现原理，你可以简单的把 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，就算你自己没有资源去搭建一套 CDN 服务，各大云服务提供商都提供了按量收费的 CDN 服务。

## 接入 CDN

要给网站接入 CDN，需要把网页的静态资源上传到 CDN 服务上去，在服务这些静态资源的时候需要通过 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，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 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内容如下：

body{background:url(arch\_ae805d49.png) repeat}h1{color:red}

可以看出到导入资源时都是通过相对路径去访问的，当把这些资源都放到同一个 CDN 服务上去时，网页是能正常使用的。 但需要注意的是由于 CDN 服务一般都会给资源开启很长时间的缓存，例如用户从 CDN 上获取到了 index.html 这个文件后， 就算在这之后 index.html 被重新发布给覆盖了，但是用户在很长一段时间内还是运行的之前的版本，这会新的导致发布不能立即生效。

要避免以上问题，业界比较成熟的做法是这样的：

* 针对 HTML 文件：不开启缓存，把 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 服务上去，同时给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内容而变化，只要文件发生变化其对应的 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，在 HTML 文件中的资源引入地址也需要换成 CDN 服务提供的地址，例如以上的 index.html 变为如下：

<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 的内容也应该变为如下：

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。

如果你对形如 //cdn.com/id/app\_a6976b6d.css 这样的 URL 感到陌生，你需要知道这种 URL 省掉了前面的 http: 或者 https: 前缀， 这样做的好处时在访问这些资源的时候会自动的根据当前 HTML 的 URL 是采用什么模式去决定是采用 HTTP 还是 HTTPS 模式。

除此之外，如果你还知道浏览器有一个规则是同一时刻针对同一个域名的资源并行请求是有限制的话（具体数字大概4个左右，不同浏览器可能不同）， 你会发现上面的做法有个很大的问题。由于所有静态资源都放到了同一个 CDN 服务的域名下，也就是上面的 cdn.com。 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，必须等其它资源加载完才能继续加载。 要解决这个问题，可以把这些静态资源分散到不同的 CDN 服务上去， 例如把 JavaScript 文件放到 js.cdn.com 域名下、把 CSS 文件放到 css.cdn.com 域名下、图片文件放到 img.cdn.com 域名下， 这样做之后 index.html 需要变成这样：

<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>

使用了多个域名后又会带来一个新问题：增加域名解析时间。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。 当然你可以通过在 HTML HEAD 标签中 加入 <link rel="dns-prefetch" href="//js.cdn.com"> 去预解析域名，以降低域名解析带来的延迟。

## 用 Webpack 实现 CDN 的接入

总结上面所说的，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。
* 静态资源的文件名称需要带上有文件内容算出来的 Hash 值，以防止被缓存。
* 不同类型的资源放到不同域名的 CDN 服务上去，以防止资源的并行加载被阻塞。

先来看下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: {  
 // 给输出的 JavaScript 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/',  
 },  
 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({  
 // 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/,  
 // 给输出的 PNG 文件名称加上 Hash 值  
 use: ['file-loader?name=[name]\_[hash:8].[ext]'],  
 },  
 // 省略其它 Loader 配置...  
 ]  
 },  
 plugins: [  
 // 使用 WebPlugin 自动生成 HTML  
 new WebPlugin({  
 // HTML 模版文件所在的文件路径  
 template: './template.html',  
 // 输出的 HTML 的文件名称  
 filename: 'index.html',  
 // 指定存放 CSS 文件的 CDN 目录 URL  
 stylePublicPath: '//css.cdn.com/id/',  
 }),  
 new ExtractTextPlugin({  
 // 给输出的 CSS 文件名称加上 Hash 值  
 filename: `[name]\_[contenthash:8].css`,  
 }),  
 // 省略代码压缩插件配置...  
 ],  
};

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，需要分别在：

* output.publicPath 中设置 JavaScript 的地址。
* css-loader.publicPath 中设置被 CSS 导入的资源的的地址。
* WebPlugin.stylePublicPath 中设置 CSS 文件的地址。

设置好 publicPath 后，WebPlugin 在生成 HTML 文件和 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*提供项目完整代码(http://webpack.wuhaolin.cn/4-9CDN加速.zip)*

# 4-10 使用 Tree Sharking

## 认识 Tree Sharking

Tree Sharking 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 Sharking 最先在 Rollup 中出现，Webpack 在 2.0 版本中将其引入。

为了更直观的理解它，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：

export function funcA() {  
}  
  
export function funB() {  
}

main.js 源码：

import {funcA} from './util.js';  
funcA();

Tree Sharking 后的 util.js：

export function funcA() {  
}

由于只用到了 util.js 中的 funcA，所以剩下的都被 Tree Sharking 当作死代码给剔除了。

需要注意的是要让 Tree Sharking 正常工作的前提是交给 Webpack 的 JavaScript 代码必须是采用 ES6 模块化语法的， 因为 ES6 模块化语法是静态的（导入导出语句中的路径必须是静态的字符串，而且不能放入其它代码块中），这让 Webpack 可以简单的分析出哪些 export 的被 import 过了。 如果你采用 ES5 中的模块化，例如 module.export={...}、require(x+y)、if(x){require('./util')}，Webpack 无法分析出哪些代码可以剔除。

## 接入 Tree Sharking

上面讲了 Tree Sharking 是做什么的，接下来一步步教你如何配置 Webpack 让 Tree Sharking 生效。

首先，为了把采用 ES6 模块化的代码交给 Webpack，需要配置 Babel 让其保留 ES6 模块化语句，修改 .babelrc 文件为如下：

{  
 "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 Sharking 的工作， 这时你会发现在控制台中输出了如下的日志：

> 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 确实正确的分析出了如何剔除死代码。

但当你打开 Webpack 输出的 bundle.js 文件看下时，你会发现用不上的代码还在里面，如下：

/\* 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用上了哪些没用上，要剔除用不上的代码还得经过 UglifyJS 去处理一遍。 要接入 UglifyJS 也很简单，不仅可以通过*4-8压缩代码*中介绍的加入 UglifyJSPlugin 去实现， 也可以简单的通过在启动 Webpack 时带上 --optimize-minimize 参数，为了快速验证 Tree Sharking 我们采用较简单的后者来实验下。

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 Sharking 确实做到了，用不上的代码都被剔除了。

当你的项目使用了大量第三方库时，你会发现 Tree Sharking 似乎不生效了，原因是大部分 Npm 中的代码都是采用的 CommonJS 语法， 这导致 Tree Sharking 无法正常工作而降级处理。 但幸运的时有些库考虑到了这点，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

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*2-4Resolve mainFields* 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 Sharking 对 redux 生效，需要配置 Webpack 的文件寻找规则为如下：

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 就采用 browser 或者 main 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，但对于提供了的不能放过，能优化的就优化。

目前越来越多的 Npm 中的第三方模块考虑到了 Tree Sharking，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，假如将来你要发布一个库到 Npm 时，希望你能支持 Tree Sharking， 以让 Tree Sharking 发挥更大的优化效果，让更多的人为此受益。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/4-10使用TreeSharking.zip)*

# 4-11 提取公共代码

## 为什么需要提取公共代码

大型网站通常会由多个页面组成，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，以及使用同一套样式代码，这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把这些公共的部分包含进去，会造成以下问题：

* 相同的资源被重复的加载，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。

如果把多个页面公共的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网站的其中一个网页，那么访问这个网站下的其它网页的概率将非常大。 在用户第一次访问后，这些页面公共代码的文件已经被浏览器缓存起来，在用户切换到其它页面时，存放公共代码的文件就不会再重新加载，而是直接从缓存中获取。 这样做后有如下好处：

* 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，但之后访问其它页面的速度将大大提升。

## 如何提取公共代码

你已经知道了提取公共代码会有什么好处，但是在实战中具体要怎么做，以达到效果最优呢？ 通常你可以采用以下原则去为你的网站提取公共代码：

* 根据你网站所使用的技术栈，找出网站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，把它们提取到一个单独的文件。 一般把这个文件叫做 base.js，因为它包含所有网页的基础运行环境；
* 在剔除了各个页面中被 base.js 包含的部分代码外，再找出所有页面都依赖的公共部分的代码提取出来放到 common.js 中去。
* 再为每个网页都生成一个单独的文件，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下：

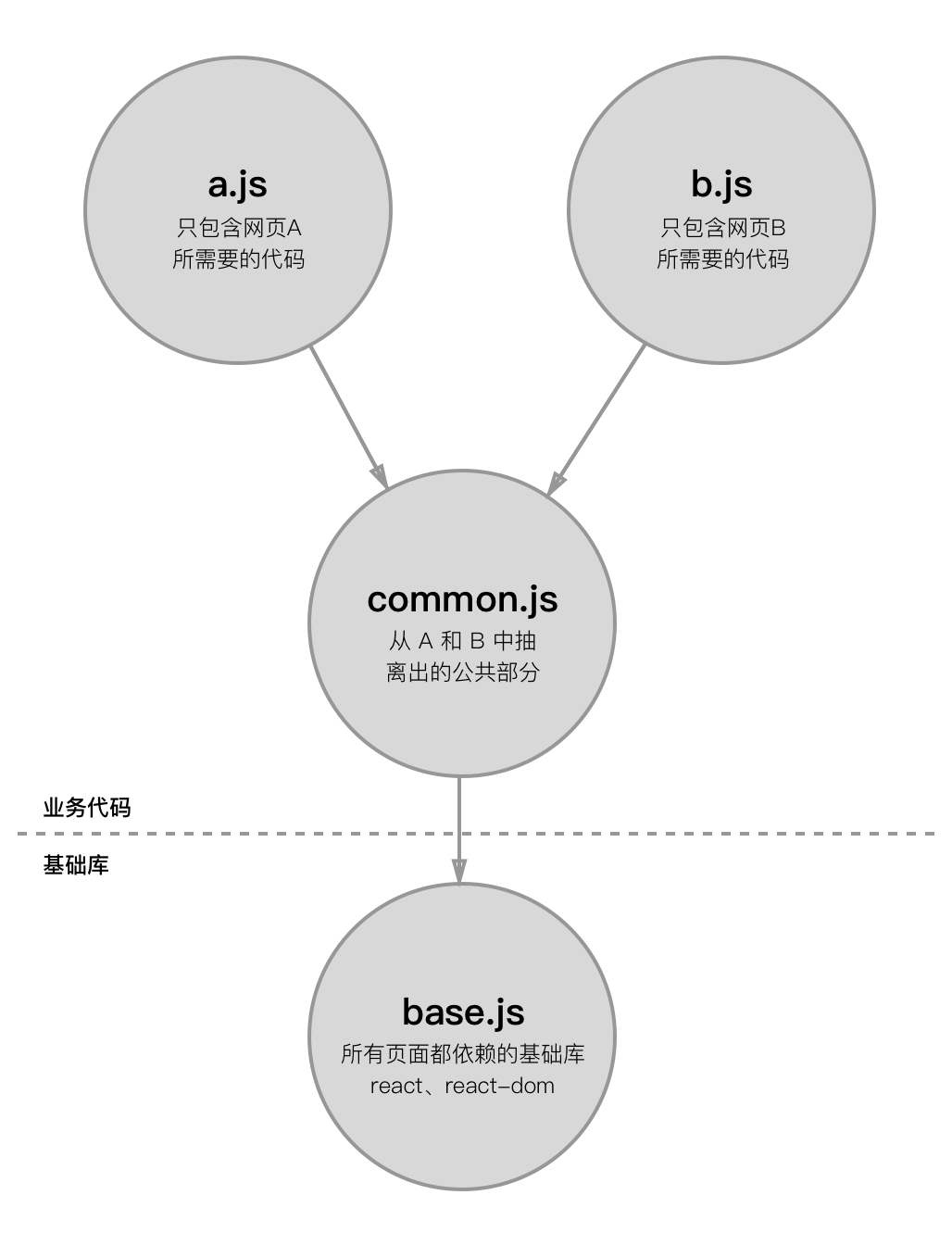


图4-11 提取公共代码文件结构图

读到这里你可以会有疑问：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 中去，为什么还需要再把网站所有页面都需要用到的基础库提取到 base.js 去呢？ 原因是为了长期的缓存 base.js 这个文件。

发布到线上的文件都会采用在*4-9CDN加速*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 Hash 值，也就是最终 base.js 的文件名会变成 base\_3b1682ac.js，以长期缓存文件。 网站通常会不断的更新发布，每次发布都会导致 common.js 和各个网页的 JavaScript 文件都会因为文件内容发生变化而导致其 Hash 值被更新，也就是缓存被更新。

把所有页面都需要用到的基础库提取到 base.js 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。 每次发布浏览器都会使用被缓存的 base.js 文件，而不用去重新下载 base.js 文件。 由于 base.js 通常会很大，这对提升网页加速速度能起到很大的效果。

## 如何通过 Webpack 提取公共代码

你已经知道如何提取公共代码，接下来教你如何用 Webpack 实现。

Webpack 内置了专门用于提取多个 Chunk 中公共部分的插件 CommonsChunkPlugin，CommonsChunkPlugin 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，这个新 Chunk 的名称  
 name: 'common'  
})

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，这个新 Chunk 中包含了被提取出的代码，在使用过程中必须指定 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，一个 Chunk 中会包含这个 Chunk 的入口文件和入口文件依赖的文件。

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，为了把基础运行库从 common 中抽离到 base 中去，还需要做一些处理。

首先需要先配置一个 Chunk，这个 Chunk 中只依赖所有页面都依赖的基础库以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';

接着再修改 Webpack 配置，在 entry 中加入 base，相关修改如下：

module.exports = {  
 entry: {  
 base: './base.js'  
 },  
};

以上就完成了对新 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 中提取公共的部分  
 chunks: ['common', 'base'],  
 // 把公共的部分放到 base 中  
 name: 'base'  
})

由于 common 和 base 公共的部分就是 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。

以上都配置好后重新执行构建，你将会得到四个文件，它们分别是：

* base.js：所有网页都依赖的基础库组成的代码；
* common.js：除了 base.js 中包含的部分外，网页 A、B 都需要的代码；
* 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，你需要在其 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>

以上就完成了提取公共代码需要的所有步骤。

针对 CSS 资源，以上理论和方法同样有效，也就是说你也可以对 CSS 文件做同样的优化。

以上方法可能会出现 common.js 中没有代码的情况，原因是除去基础运行库外很难再找到所有页面都会用上的模块。 在出现这种情况时，你可以采取以下做法之一：

* CommonsChunkPlugin 提供一个选项 minChunks，表示文件要被提取出来时需要在指定的 Chunks 中最小出现最小次数。 假如 minChunks=2、chunks=['a','b','c','d']，任何一个文件只要在 ['a','b','c','d'] 中任意两个以上的 Chunk 中都出现过，这个文件就会被提取出来。 你可以根据自己的需求去调整 minChunks 的值，minChunks 越小越多的文件会被提取到 common.js 中去，但这也会导致部分页面加载的不相关的资源越多； minChunks 越大越少的文件会被提取到 common.js 中去，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用 CommonsChunkPlugin 去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，你需要根据页面之间的关系去思考如何配置，该方法不通用。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/4-11提取公共代码.zip)*

# 4-12 分割代码按需加载

## 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。 对于采用单页应用前端架构的网站来说，会面临着一个网页需要加载的代码量很大的问题，因为许多功能都集中的做到了一个 HTML 里。 这会导致网页加载缓慢、交互卡顿，用户体验将非常糟糕。

导致这个问题的根本原因在于一次性的加载所有功能对应的代码，但其实用户每一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。

## 如何使用按需加载

在给单页应用做按需加载优化时，一般采用以下原则：

* 把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把它们分成几类。
* 把每一类合并为一个 Chunk，按需加载对应的 Chunk。
* 对于用户首次打开你的网站时需要看到的画面所对应的功能，不要对它们做按需加载，而是放到执行入口所在的 Chunk 中，以提升用户能感知的加载时间。
* 对应个别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，也就是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自己去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，你可以预言用户接下来可能会进行的操作，并提前加载好对应的代码，从而让用户感知不到网络加载时间。

## 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，main.js 文件中只包含监听按钮事件和加载按需加载的代码。
* 当按钮被点击时才去加载被分割出去的 show.js 文件，加载成功后再执行 show.js 里的函数。

其中 main.js 文件内容如下：

window.document.getElementById('btn').addEventListener('click', function () {  
 // 当按钮被点击后才去加载 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 文件内容如下：

module.exports = function (content) {  
 window.alert('Hello ' + content);  
};

代码中最关键的一句是 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 为入口新生成一个 Chunk；
* 当代码执行到 import 所在语句时才会去加载由 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 方法中获取到 show.js 导出的内容。

在使用 import() 分割代码后，你的浏览器并且要支持 *Promise API(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Promise)* 才能让代码正常运行， 因为 import() 返回一个 Promise，它依赖 Promise。对于不原生支持 Promise 的浏览器，你可以注入 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，默认名称将会是 [id].js。 /\* webpackChunkName: "show" \*/ 是在 Webpack3 中引入的新特性，在 Webpack3 之前是无法为动态生成的 Chunk 赋予名称的。

为了正确的输出在 /\* webpackChunkName: "show" \*/ 中配置的 ChunkName，还需要配置下 Webpack，配置如下：

module.exports = {  
 // JS 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。 如果没有这行，分割出的代码的文件名称将会是 [id].js。 chunkFilename 具体含义见*2-2 配置-Output*。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/4-12分割代码按需加载.zip)*

## 按需加载与 ReactRouter

在实战中，不可能会有上面那么简单的场景，接下来举一个实战中的例子：对采用了 *ReactRouter(https://reacttraining.com/react-router/web)* 的应用进行按需加载优化。 这个例子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，获取到了代码导出的值，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，异步地加载 PageAbout 组件  
 () => import(/\* webpackChunkName: 'page-about' \*/'./pages/about')  
 )}  
 />  
 <Route path='/login' component={getAsyncComponent(  
 // 异步加载函数，异步地加载 PageAbout 组件  
 () => import(/\* webpackChunkName: 'page-login' \*/'./pages/login')  
 )}  
 />  
 </div>  
 </HashRouter>  
 )  
}  
  
// 渲染根组件  
render(<App/>, window.document.getElementById('app'));

以上代码中最关键的部分是 getAsyncComponent 函数，它的作用是配合 ReactRouter 去按需加载组件，具体含义请看代码中的注释。

由于以上源码需要通过 Babel 去转换后才能在浏览器中正常运行，需要在 Webpack 中配置好对应的 babel-loader，源码先交给 babel-loader 处理后再交给 Webpack 去处理其中的 import(\*) 语句。 但这样做后你很快会发现一个问题：Babel 报出错误说不认识 import(\*) 语法。 导致这个问题的原因是 import(\*) 语法还没有被加入到在 *3-1使用ES6语言*中提到的 ECMAScript 标准中去， 为此我们需要安装一个 Babel 插件 babel-plugin-syntax-dynamic-import，并且将其加入到 .babelrc 中去：

{  
 "presets": [  
 "env",  
 "react"  
 ],  
 "plugins": [  
 "syntax-dynamic-import"  
 ]  
}

执行 Webpack 构建后，你会发现输出了三个文件：

* main.js：执行入口所在的代码块，同时还包括 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，以降低用户能感知到的加载时间；
* page-about.js：当用户访问 /about 时才会加载的代码块；
* page-login.js：当用户访问 /login 时才会加载的代码块。

同时你还会发现 page-about.js 和 page-login.js 这两个文件在首页是不会加载的，而是会当你切换到了对应的子页面后文件才会开始加载。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/4-12分割代码按需加载ReactRouter.zip)*

# 4-13 使用 Prepack

## 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。

Prepack 由 Facebook 开源，它采用较为激进的方法：在保持运行结果一致的情况下，改变源代码的运行逻辑，输出性能更高的 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如下：

console.log("hello webpack");

可以看出 Prepack 通过在编译阶段预先执行了源码得到执行结果，再直接把运行结果输出来以提升性能。

Prepack 的工作原理和流程大致如下：

1. 通过 Babel 把 JavaScript 源码解析成抽象语法树（AST），以方便更细粒度地分析源码；
2. Prepack 实现了一个 JavaScript 解释器，用于执行源码。借助这个解释器 Prepack 才能掌握源码具体是如何执行的，并把执行过程中的结果返回到输出中。

从表面上看去这似乎非常美好，但实际上 Prepack 还不够成熟与完善。Prepack 目前还处于初期的开发阶段，局限性也很大，例如：

* 不能识别 DOM API 和 部分 Node.js API，如果源码中有调用依赖运行环境的 API 就会导致 Prepack 报错；
* 存在优化后的代码性能反而更低的情况；
* 存在优化后的代码文件尺寸大大增加的情况。

总之，现在把 Prepack 用于线上环境还为时过早。

## 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()  
 ]  
};

重新执行构建你就会看到输出的被 Prepack 优化后的代码。

本实例*提供项目完整代码(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、运行的更快， 它又译作 "作用域提升"，是在 Webpack3 中新推出的功能。 单从名字上看不出 Scope Hoisting 到底做了什么，下面来详细介绍它。

## 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 打包后输出中的部分代码如下：

[  
 (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，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);  
 })  
]

从中可以看出开启 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到了 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代码；
* 代码在运行时因为创建的函数作用域更少了，内存开销也随之变小。

Scope Hoisting 的实现原理其实很简单：分析出模块之间的依赖关系，尽可能的把打散的模块合并到一个函数中去，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 需要分析出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。 原因和*4-10 使用 TreeSharking* 中介绍的类似。

## 使用 Scope Hoisting

要在 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 依赖源码需采用 ES6 模块化语法，还需要配置 mainFields。 原因在 *4-10 使用 TreeSharking* 中提到过：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，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 会降级处理不使用 Scope Hoisting 优化，为了知道 Webpack 对哪些代码做了降级处理， 你可以在启动 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 告诉了你哪个文件因为什么原因导致了降级处理。

也就是说要开启 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*提供项目完整代码(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，但这些方法也无法涵盖所有的场景，为此你需要对输出结果做分析，以决定下一步的优化方向。

最直接的分析方法就是去阅读 Webpack 输出的代码，但由于 Webpack 输出的代码可读性非常差而且文件非常大，这会让你非常头疼。 为了更简单直观的分析输出结果，社区中出现了许多可视化的分析工具。这些工具以图形的方式把结果更加直观的展示出来，让你快速看到问题所在。 接下来教你如何使用这些工具。

在启动 Webpack 时，支持两个参数，分别是：

* --profile：记录下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，你会发现项目中多出了一个 stats.json 文件。 这个 stats.json 文件是给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、含义是把 webpack --profile --json 输出的内容通过管道输出到 stats.json 文件中。

## 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，你就会看到一个弹窗提示你上传 JSON 文件，也就是需要上传上面讲到的 stats.json 文件，如图：

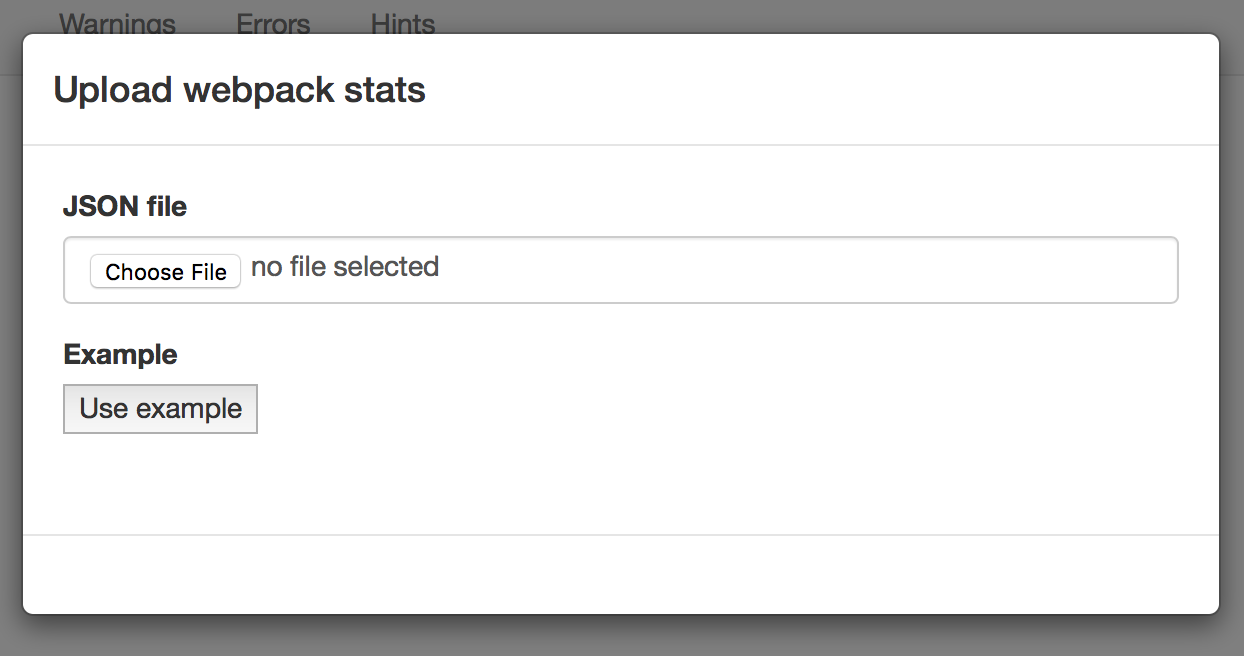


图 4-15-1 Webpack Analyse 上传文件弹窗

Webpack Analyse 不会把你选择的 stats.json 文件发达到服务器，而是在浏览器本地解析，你不用担心自己的代码为此而泄露。 选择文件后，你马上就能如下的效果图：

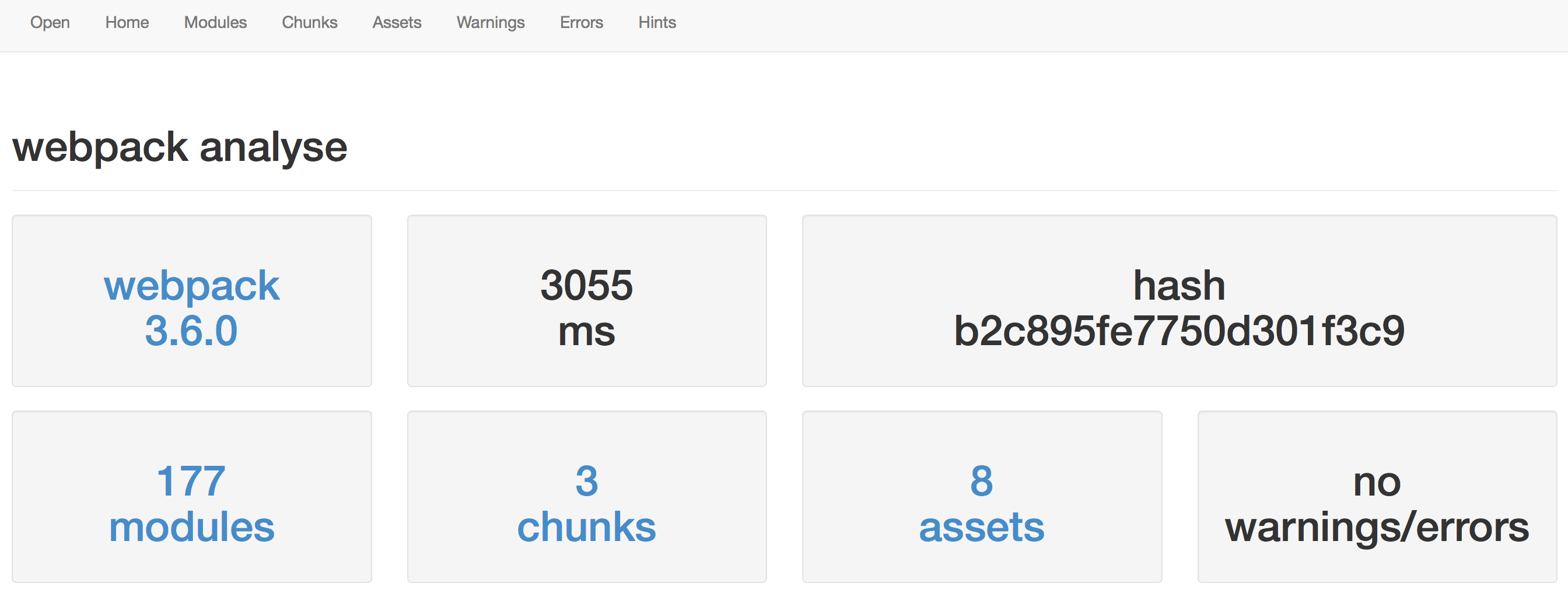


图 4-15-1 Webpack Analyse 主页

它分为了六大板块，分别是：

* **Modules**：展示所有的模块，每个模块对应一个文件。并且还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 Chunk、模块大小；
* **Chunks**：展示所有的代码块，一个代码块中包含多个模块。并且还包含代码块的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。并且还包括文件名称、大小、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**：展示处理每个模块的过程中的耗时。

下面以在 *3-10管理多个单页应用* 中使用的项目为例，来分析其 stats.json 文件。

点击 **Modules**，查看模块信息，效果图如下：

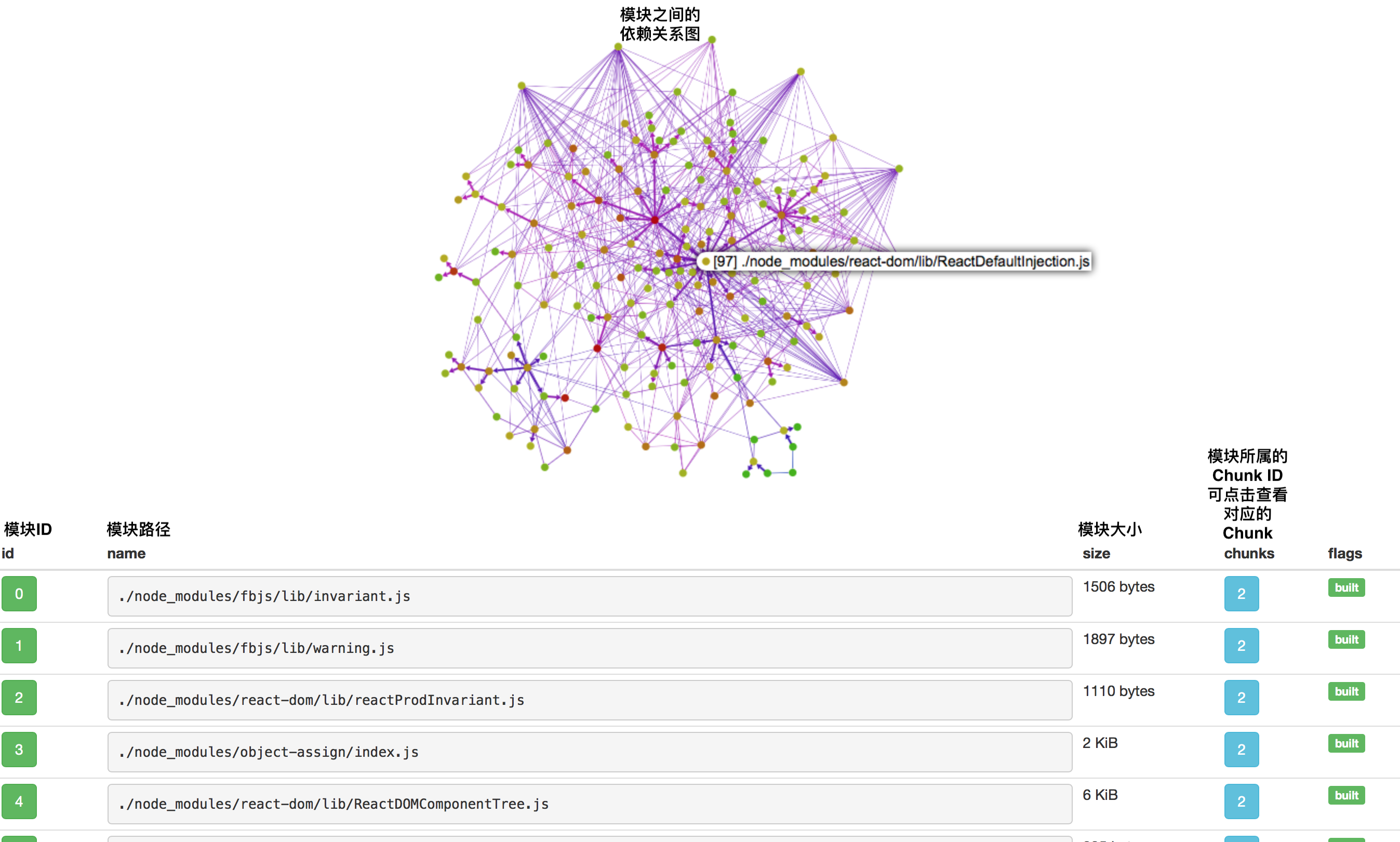


图 4-15-1 Webpack Analyse Modules

由于依赖了大量第三方模块，文件数量大，导致模块之间的依赖关系图太密集而无法看清，但你可以进一步放大查看。

点击 **Chunks**，查看代码块信息，效果图如下：

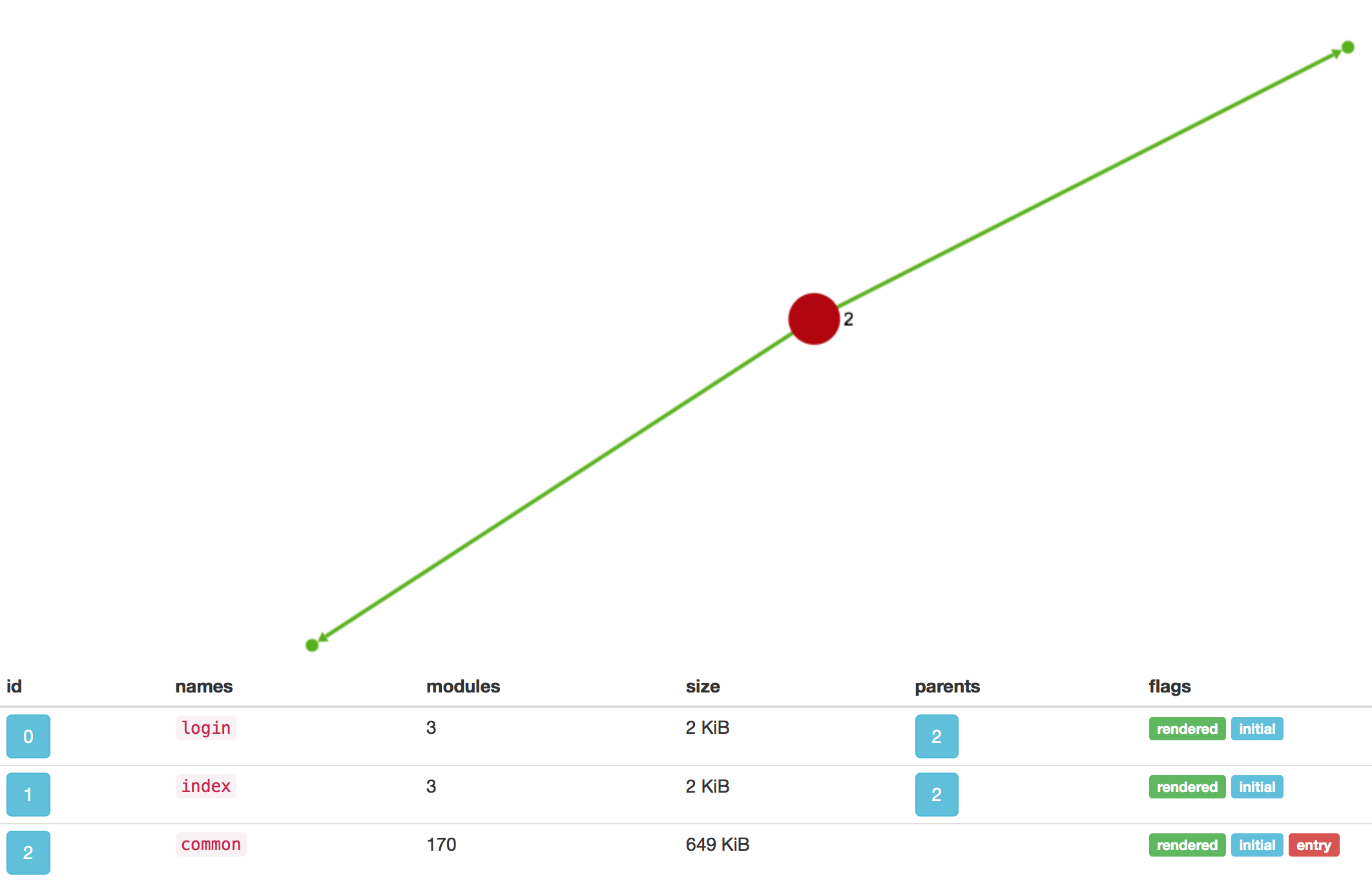


图 4-15-2 Webpack Analyse Chunks

由代码块之间的依赖关系图可以看出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。

点击 **Assets**，查看输出的文件资源，效果图如下：

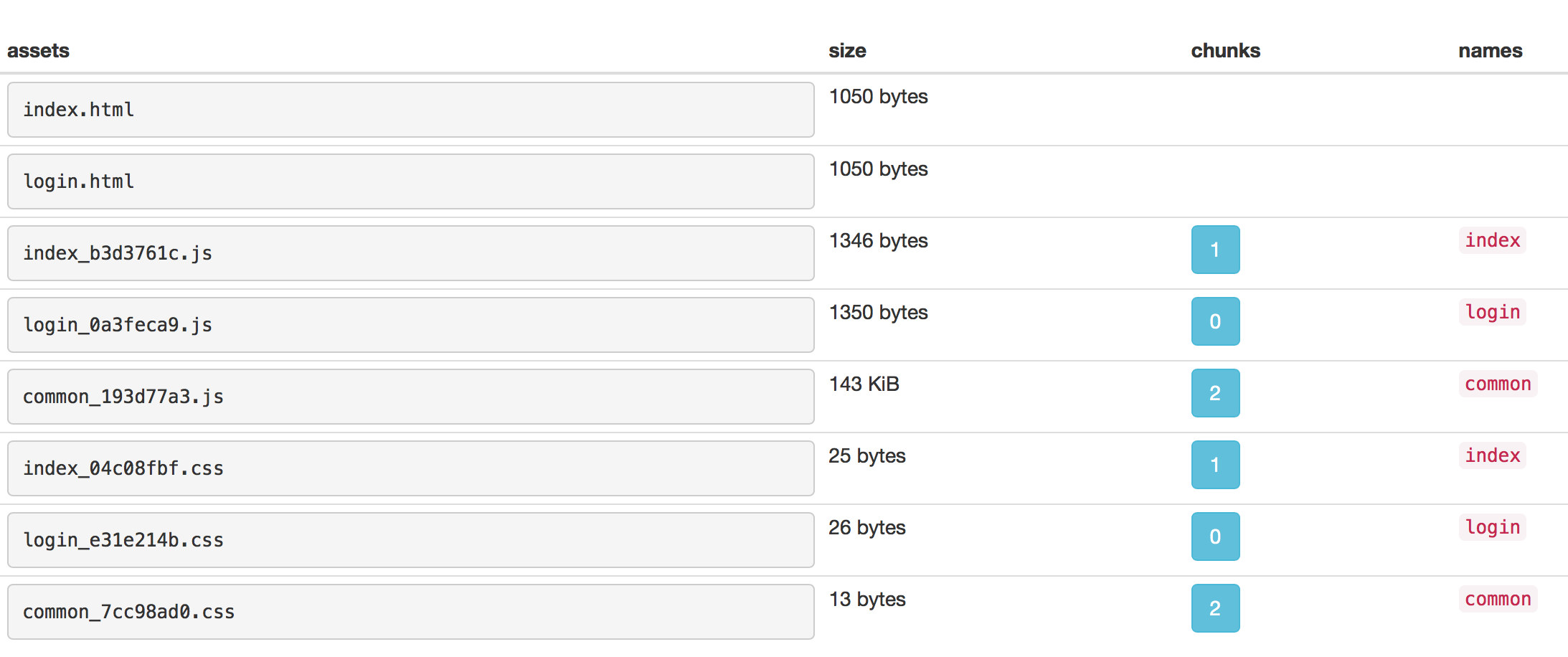


图 4-15-3 Webpack Analyse Assets

点击 **Hints**，查看输出过程中的耗时分布，效果图如下：

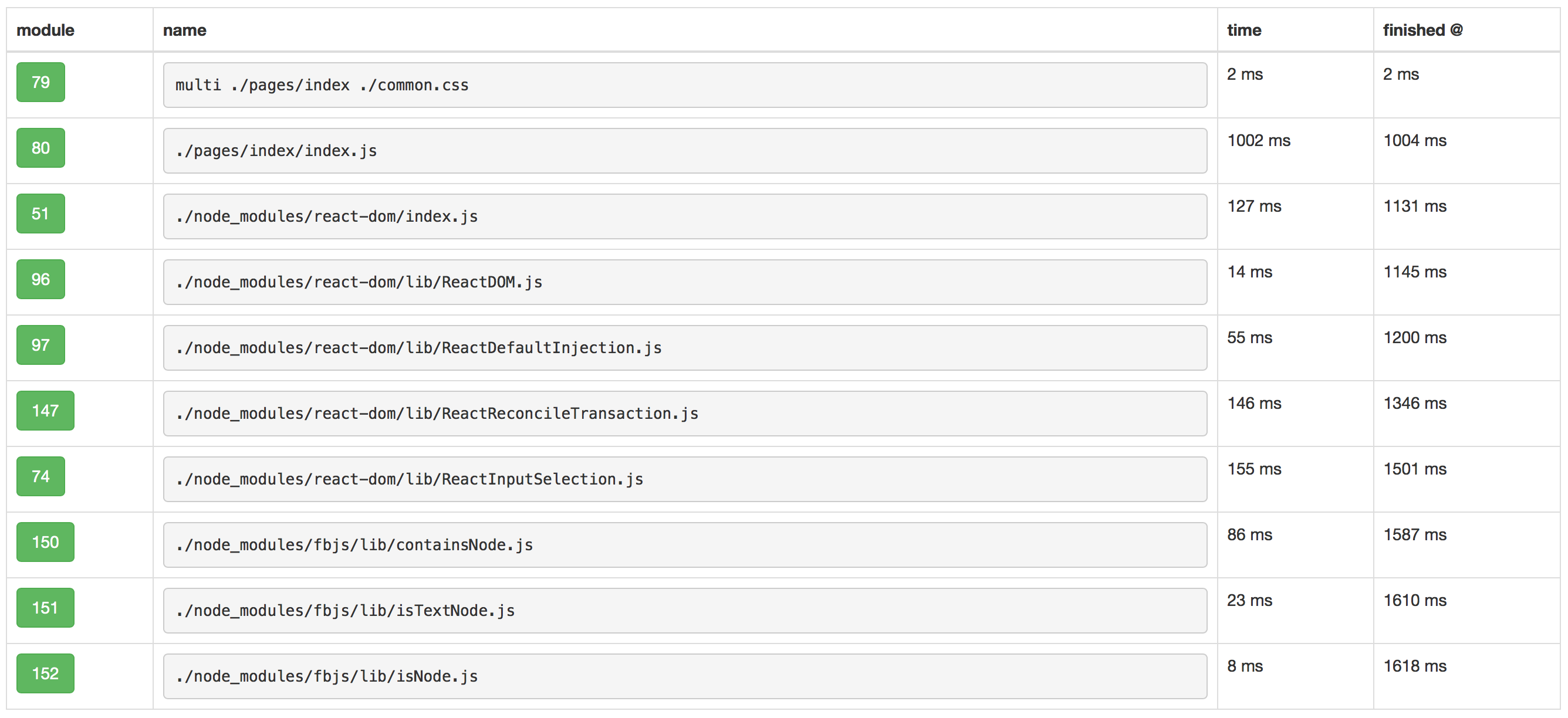


图 4-15-3 Webpack Analyse Hints

从 Hints 可以看出每个文件在处理过程的开始时间和结束时间，从而可以找出是哪个文件导致构建缓慢。

## 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， 它虽然没有官方那样有那么多功能，但比官方的要更加直观。

先来看下它的效果图：

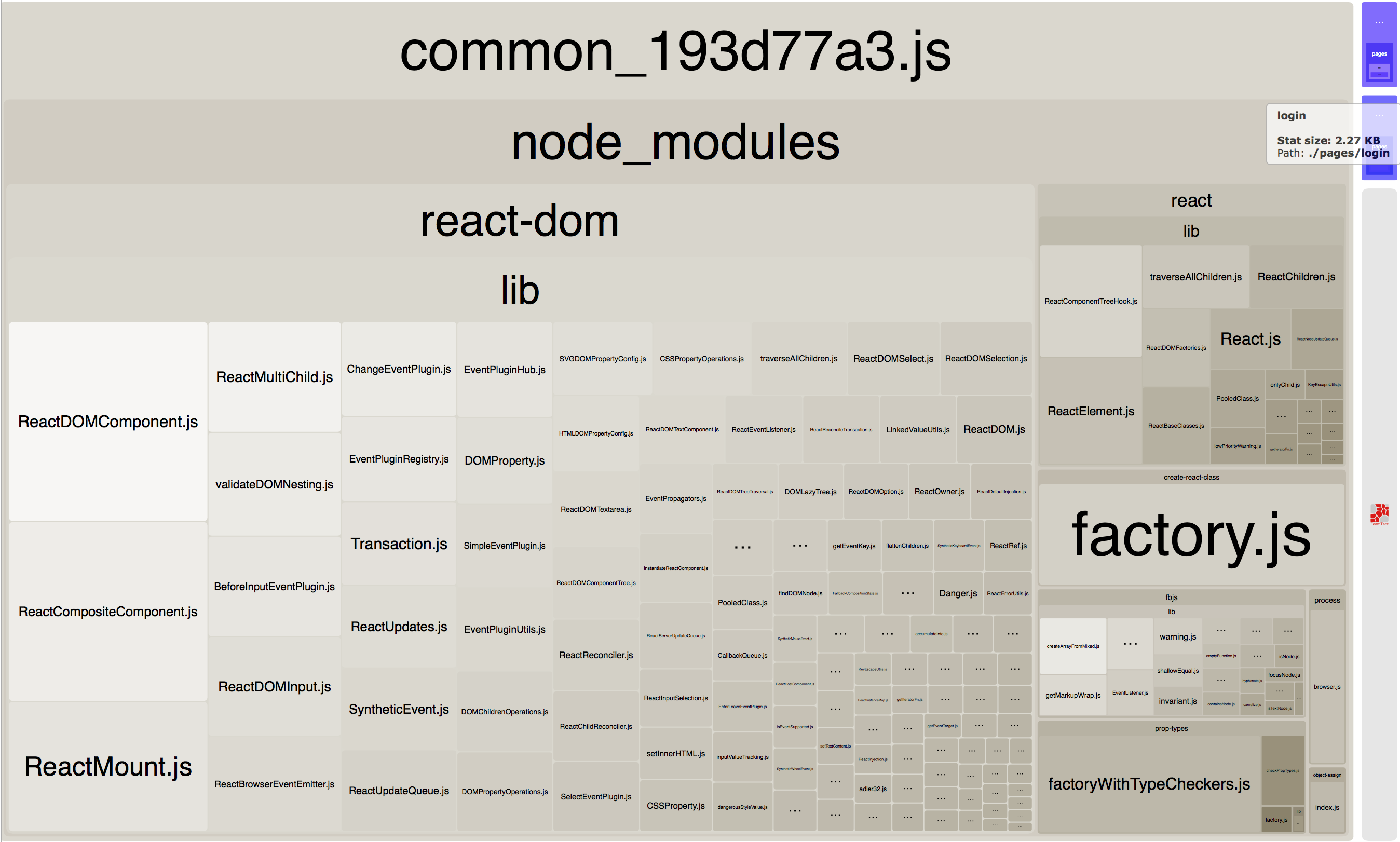


图 4-15-4 Webpack Analyse Assets

它能方便的让你知道：

* 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，一眼看出哪些文件尺寸大；
* 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：

1. 安装 webpack-bundle-analyzer 到全局，执行命令 npm i -g webpack-bundle-analyzer；
2. 按照上面提到的方法生成 stats.json 文件；
3. 在项目根目录中执行 webpack-bundle-analyzer 后，浏览器会打开对应网页看到以上效果。

本实例*提供项目完整代码(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，这些优化的方法都是来自项目实战中的经验积累。 虽然每一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可以相互组合，以达到最佳的效果。

以下将给出是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，按照开发环境和线上环境为该项目配置了两份文件，分别如下：

### 侧重优化开发体验的配置文件 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，把每一个目录看成一个单页应用  
const autoWebPlugin = new AutoWebPlugin('./src/pages', {  
 // HTML 模版文件所在的文件路径  
 template: './template.html',  
 // 提取出所有页面公共的代码  
 commonsChunk: {  
 // 提取出公共代码 Chunk 的名称  
 name: 'common',  
 },  
});  
  
module.exports = {  
 // AutoWebPlugin 会找为寻找到的所有单页应用，生成对应的入口配置，  
 // autoWebPlugin.entry 方法可以获取到生成入口配置  
 entry: autoWebPlugin.entry({  
 // 这里可以加入你额外需要的 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 Sharking 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?$/，提升正则表达式性能  
 test: /\.js$/,  
 // 使用 HappyPack 加速构建  
 use: ['happypack/loader?id=babel'],  
 // 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'),  
 },  
 {  
 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'],  
 // 把公共的部分放到 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/,  
 }  
};

### 侧重优化输出质量的配置文件 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，把每一个目录看成一个单页应用  
const autoWebPlugin = new AutoWebPlugin('./src/pages', {  
 // HTML 模版文件所在的文件路径  
 template: './template.html',  
 // 提取出所有页面公共的代码  
 commonsChunk: {  
 // 提取出公共代码 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 会找为寻找到的所有单页应用，生成对应的入口配置，  
 // autoWebPlugin.entry 方法可以获取到生成入口配置  
 entry: autoWebPlugin.entry({  
 // 这里可以加入你额外需要的 Chunk 入口  
 base: './src/base.js',  
 }),  
 output: {  
 // 给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?$/，提升正则表达式性能  
 test: /\.js$/,  
 // 使用 HappyPack 加速构建  
 use: ['happypack/loader?id=babel'],  
 // 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'),  
 },  
 {  
 test: /\.js$/,  
 use: ['happypack/loader?id=ui-component'],  
 include: path.resolve(\_\_dirname, 'src'),  
 },  
 {  
 // 增加对 CSS 文件的支持  
 test: /\.css/,  
 // 提取出 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 是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 配置中一样  
 // 通过 minimize 选项压缩 CSS 代码  
 loaders: ['css-loader?minimize'],  
 }),  
 new ExtractTextPlugin({  
 // 给输出的 CSS 文件名称加上 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 中提取公共的部分  
 chunks: ['common', 'base'],  
 // 把公共的部分放到 base 中  
 name: 'base'  
 }),  
 new DefinePlugin({  
 // 定义 NODE\_ENV 环境变量为 production 去除 react 代码中的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 JS 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,  
 // 删除所有的 `console` 语句，可以兼容ie浏览器  
 drop\_console: true,  
 // 内嵌定义了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,  
 }  
 },  
 }),  
 ]  
};

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/4-16优化总结.zip)*

本章介绍的优化方法虽然难以涵盖 Webpack 的方方面面，但足以解决实战中常见的场景。 对于本书没有介绍到的场景，你需要根据自己的需求按照以下思路去优化：

1. 找出问题的原因；
2. 找出解决问题的方法；
3. 寻找解决问题方法对应的 Webpack 集成方案。

同时你还需要跟紧社区的迭代，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。

# 第5章 原理

虽然通过前四章的学习你已经能用 Webpack 解决常见问题， 但当你在实战中遇到比较特殊的需求、在社区中找不到解决方案时，你需要编写自己的 Loader 或 Plugin， 要做到这点的前提是需要了解 Webpack 的工作原理。

了解 Webpack 的工作原理还能让你对 Webpack 有更深的认识，使用它时更加得心应手。

本章包含以下内容：

了解 Webpack 整体架构、工作流程，学会区分一个的功能的实现是通过 Loader 合适还是 Plugin 更合适： - *5-1 工作原理概括* - *5-2 输出文件分析*

如何开发、调试 Loader 和 Plugin： - *5-3 编写 Loader* - *5-4 编写 Plugin* - *5-5 调试 Webpack*

*5-6 原理总结* 对本章做一个总结。

# 5-1 工作原理概括

Webpack 以其使用简单著称，在使用它的过程中，使用者只需把它当作一个黑盒，需要关心的只有它暴露出来的配置。 本节将带你走进这个黑盒，看看 Webpack 是如何运行的。

## 基本概念

在了解 Webpack 原理前，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，一个模块对应着一个文件。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，用于把模块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出对应的事件，插件可以监听这些事件的发生，在特定时机做对应的事情。

## 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：

1. 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；
2. 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，执行对象的 run 方法开始执行编译；
3. 确定入口：根据配置中的 entry 找出所有的入口文件；
4. 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；
5. 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以及它们之间的依赖关系；
6. 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，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；
7. 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，把文件内容写入到文件系统。

在以上过程中，Webpack 会在特定的时间点广播出特定的事件，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。

## 流程细节

Webpack 的构建流程可以分为以下三大阶段：

1. 初始化：启动构建，读取与合并配置参数，加载 Plugin，实例化 Compiler。
2. 编译：从 Entry 发出，针对每个 Module 串行调用对应的 Loader 去翻译文件内容，再找到该 Module 依赖的 Module，递归地进行编译处理。
3. 输出：对编译后的 Module 组合成 Chunk，把 Chunk 转换成文件，输出到文件系统。

如果只执行一次构建，以上阶段将会按照顺序各执行一次。但在开启监听模式下，流程将变为如下：

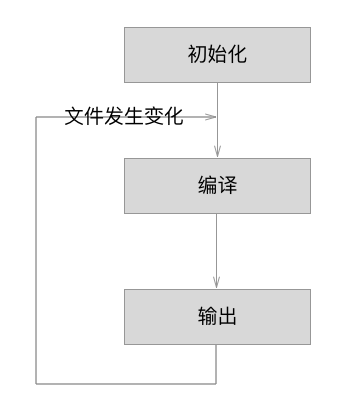


图5-1-1 监听模式的构建流程

在每个大阶段中又会发生很多事件，Webpack 会把这些事件广播出来供给 Plugin 使用，下面来一一介绍。

### 初始化阶段

|  |  |
| --- | --- |
| 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。 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。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。同时给插件传入 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 | 根据配置初始化完 resolver，resolver 负责在文件系统中寻找指定路径的文件。 |

### 编译阶段

|  |  |
| --- | --- |
| 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的编译，在这个事件中可以获取到是哪些文件发生了变化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，每当检测到文件变化，一次新的 Compilation 将被创建。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点回调供插件做扩展。 |
| 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，递归的编译和解析。 |
| 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，最重要的要数 compilation 事件了，因为在 compilation 阶段调用了 Loader 完成了每个模块的转换操作，在 compilation 阶段又包括很多小的事件，它们分别是：

|  |  |
| --- | --- |
| 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 对一个模块转换完后，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 后面对代码的分析。 |
| program | 从配置的入口模块开始，分析其 AST，当遇到 require 等导入其它模块语句时，便将其加入到依赖的模块列表，同时对新找出的依赖模块递归分析，最终搞清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 转换完成后，根据依赖关系开始生成 Chunk。 |

### 输出阶段

|  |  |
| --- | --- |
| 事件名 | 解释 |
| should-emit | 所有需要输出的文件已经生成好，询问插件哪些文件需要输出，哪些不需要。 |
| 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内容。 |
| after-emit | 文件输出完毕。 |
| done | 成功完成一次完成的编译和输出流程。 |
| failed | 如果在编译和输出流程中遇到异常导致 Webpack 退出时，就会直接跳转到本步骤，插件可以在本事件中获取到具体的错误原因。 |

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，并且把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，使用对应的模版生成最终要要输出的文件内容。

至于如何把 Chunk 输出为具体的文件，详情可以阅读 *5-2输出文件分析*。

# 5-2 输出文件分析

虽然在前面的章节中你学会了如何使用 Webpack ，也大致知道其工作原理，可是你想过 Webpack 输出的 bundle.js 是什么样子的吗？ 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？ 本节将解释清楚以上问题。

先来看看由 *1-3安装与使用* 中最简单的项目构建出的 bundle.js 文件内容，代码如下：

(  
 // webpackBootstrap 启动函数  
 // modules 即为存放所有模块的数组，数组中的每一个元素都是一个函数  
 function (modules) {  
 // 使用安装过的模块都存放在这里面  
 // 作用是把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 require 语句相似  
 function \_\_webpack\_require\_\_(moduleId) {  
 // 如果需要加载的模块已经被加载过，就直接从内存缓存中返回  
 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，并把它存在缓存中  
 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，同时把函数需要的参数传入  
 modules[moduleId].call(module.exports, module, module.exports, \_\_webpack\_require\_\_);  
 // 把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了一个数组里，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，可以简写为如下：

(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在于输出的文件中通过 \_\_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 的原因在于浏览器不能像 Node.js 那样快速地去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。 如果模块数量很多，加载时间会很长，因此把所有模块都存放在了数组中，执行一次网络加载。

如果仔细分析 \_\_webpack\_require\_\_ 函数的实现，你还有发现 Webpack 做了缓存优化： 执行加载过的模块不会再执行第二次，执行结果会缓存在内存中，当某个模块第二次被访问时会直接去内存中读取被缓存的返回值。

## 分割代码时的输出

在采用了 *4-12 按需加载* 中介绍过的优化方法时，Webpack 的输出文件会发生变化。

例如把源码中的 main.js 修改为如下：

// 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 内容如下：

// 加载在本文件(0.bundle.js)中包含的模块  
webpackJsonp(  
 // 在其它文件中存放着的模块的 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 内容如下：

(function (modules) {  
 /\*\*\*  
 \* webpackJsonp 用于从异步加载的文件中安装模块。  
 \* 把 webpackJsonp 挂载到全局是为了方便在其它文件中调用。  
 \*  
 \* @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) {  
 // 把 moreModules 添加到 modules 对象中  
 // 把所有 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，往 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;  
  
 // 文件的路径为配置的 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](/home/travis/build/gwuhaolin/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 去异步加载 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。

在使用了 CommonsChunkPlugin 去提取公共代码时输出的文件和使用了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本质上都是代码分割。

本实例*提供项目完整代码(http://webpack.wuhaolin.cn/5-2输出文件分析.zip)*

# 5-3 编写 Loader

Loader 就像是一个翻译员，能把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的经过多个翻译员翻译。

以处理 SCSS 文件为例：

1. SCSS 源代码会先交给 sass-loader 把 SCSS 转换成 CSS；
2. 把 sass-loader 输出的 CSS 交给 css-loader 处理，找出 CSS 中依赖的资源、压缩 CSS 等；
3. 把 css-loader 输出的 CSS 交给 style-loader 处理，转换成通过脚本加载的 JavaScript 代码；

可以看出以上的处理过程需要有顺序的链式执行，先 sass-loader 再 css-loader 再 style-loader。 以上处理的 Webpack 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',  
 // 给 css-loader 传入配置项  
 options:{  
 minimize:true,   
 }  
 },  
 'sass-loader'],  
 },  
 ]  
 },  
};

## 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 会链式的顺序执行， 第一个 Loader 将会拿到需处理的原内容，上一个 Loader 处理后的结果会传给下一个接着处理，最后的 Loader 将处理后的最终结果返回给 Webpack。

所以，在你开发一个 Loader 时，请保持其职责的单一性，你只需关心输入和输出。

## Loader 基础

由于 Webpack 是运行在 Node.js 之上的，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，这里简单起见，直接把原内容返回了，相当于该 Loader 没有做任何转换  
 return source;  
};

由于 Loader 运行在 Node.js 中，你可以调用任何 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);  
};

## Loader 进阶

以上只是个最简单的 Loader，Webpack 还提供一些 API 供 Loader 调用，下面来一一介绍。

### 获得 Loader 的 options

在最上面处理 SCSS 文件的 Webpack 配置中，给 css-loader 传了 options，以控制 css-loader。 如何在自己编写的 Loader 中获取到用户传入的 options 呢？需要这样做：

const loaderUtils = require('loader-utils');  
module.exports = function(source) {  
 // 获取到用户给当前 Loader 传入的 options  
 const options = loaderUtils.getOptions(this);  
 return source;  
};

### 返回其它结果

上面的 Loader 都只是返回了原内容转换后的内容，但有些场景下还需要返回除了内容之外的东西。

例如以用 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。 为了把 Source Map 也一起随着 ES5 代码返回给 Webpack，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);  
 // 当你使用 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 给 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，给 Webpack 返回一个 Error  
 err: Error | null,  
 // 原内容转换后的内容  
 content: string | Buffer,  
 // 用于把转换后的内容得出原内容的 Source Map，方便调试  
 sourceMap?: SourceMap,  
 // 如果本次转换为原内容生成了 AST 语法树，可以把这个 AST 返回，  
 // 以方便之后需要 AST 的 Loader 复用该 AST，以避免重复生成 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，其它环境下不用生成，以加速构建。 为此 Webpack 为 Loader 提供了 this.sourceMap API 去告诉 Loader 当前构建环境下用户是否需要 Source Map。 如果你编写的 Loader 会生成 Source Map，请考虑到这点。

### 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。 但在有些场景下转换的步骤只能是异步完成的，例如你需要通过网络请求才能得出结果，如果采用同步的方式网络请求就会阻塞整个构建，导致构建非常缓慢。

在转换步骤是异步时，你可以这样：

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);  
 });  
};

### 处理二进制数据

在默认的情况下，Webpack 传给 Loader 的原内容都是 UTF-8 格式编码的字符串。 但有些场景下 Loader 不是处理文本文件，而是处理二进制文件，例如 file-loader，就需要 Webpack 给 Loader 传入二进制格式的数据。 为此，你需要这样编写 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;

以上代码中最关键的代码是最后一行 module.exports.raw = true;，没有该行 Loader 只能拿到字符串。

### 缓存加速

在有些情况下，有些转换操作需要大量计算非常耗时，如果每次构建都重新执行重复的转换操作，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。

如果你想让 Webpack 不缓存该 Loader 的处理结果，可以这样：

module.exports = function(source) {  
 // 关闭该 Loader 的缓存功能  
 this.cacheable(false);  
 return source;  
};

## 其它 Loader API

除了以上提到的在 Loader 中能调用的 Webpack API 外，还存在以下常用 API：

* this.context：当前处理文件的所在目录，假如当前 Loader 处理的文件是 /src/main.js，则 this.context 就等于 /src。
* this.resource：当前处理文件的完整请求路径，包括 querystring，例如 /src/main.js?name=1。
* this.resourcePath：当前处理文件的路径，例如 /src/main.js。
* this.resourceQuery：当前处理文件的 querystring。
* this.target：等于 Webpack 配置中的 Target，详情见 *2-7其它配置项-Target*。
* this.loadModule：但 Loader 在处理一个文件时，如果依赖其它文件的处理结果才能得出当前文件的结果时， 就可以通过 this.loadModule(request: string, callback: function(err, source, sourceMap, module)) 去获得 request 对应文件的处理结果。
* 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：给当前处理文件添加其依赖的文件，以便再其依赖的文件发生变化时，会重新调用 Loader 处理该文件。使用方法为 addDependency(file: string)。
* this.addContextDependency：和 addDependency 类似，但 addContextDependency 是把整个目录加入到当前正在处理文件的依赖中。使用方法为 addContextDependency(directory: string)。
* this.clearDependencies：清除当前正在处理文件的所有依赖，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: {...})。

其它没有提到的 API 可以去 *Webpack 官网(https://webpack.js.org/api/loaders/)* 查看。

## 加载本地 Loader

在开发 Loader 的过程中，为了测试编写的 Loader 是否能正常工作，需要把它配置到 Webpack 中后，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，要使用 Loader 时会直接使用 Loader 的名称，代码如下：

module.exports = {  
 module: {  
 rules: [  
 {  
 test: /\.css/,  
 use: ['style-loader'],  
 },  
 ]  
 },  
};

如果还采取以上的方法去使用本地开发的 Loader 将会很麻烦，因为你需要确保编写的 Loader 的源码是在 node\_modules 目录下。 为此你需要先把编写的 Loader 发布到 Npm 仓库后再安装到本地项目使用。

解决以上问题的便捷方法有两种，分别如下：

#### Npm link

Npm link 专门用于开发和调试本地 Npm 模块，能做到在不发布模块的情况下，把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 模块代码，在项目中也能使用到编辑后的代码。

完成 Npm link 的步骤如下：

1. 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；
2. 在本地 Npm 模块根目录下执行 npm link，把正在本地模块注册到全局；
3. 在项目根目录下执行 npm link loader-name，把第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步中的 package.json 文件中配置的模块名称。

链接好 Loader 到项目后你就可以像使用一个真正的 Npm 模块一样使用本地的 Loader 了。

#### ResolveLoader

在 *2-7其它配置项* 中曾介绍过 ResolveLoader 用于配置 Webpack 如何寻找 Loader。 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。

假如本地的 Loader 在项目目录中的 ./loaders/loader-name 中，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，会再去 ./loaders/ 目录下寻找。

## 实战

上面讲了许多理论，接下来从实际出发，来编写一个解决实际问题的 Loader。

该 Loader 名叫 comment-require-loader，作用是把 JavaScript 代码中的注释语法

// @require '../style/index.css'

转换成

require('../style/index.css');

该 Loader 的使用场景是去正确加载针对 *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 去转换   
 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) {  
 // 使用正则把 // @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*提供项目完整代码(https://github.com/gwuhaolin/comment-require-loader)*

# 5-4 编写 Plugin

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

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给该插件传入的配置  
 constructor(options){  
 }  
   
 // Webpack 会调用 BasicPlugin 实例的 apply 方法给插件实例传入 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 获得其实例。 在初始化 compiler 对象后，再调用 basicPlugin.apply(compiler) 给插件实例传入 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 广播出来的事件。 并且可以通过 compiler 对象去操作 Webpack。

通过以上最简单的 Plugin 相信你大概明白了 Plugin 的工作原理，但实际开发中还有很多细节需要注意，下面来详细介绍。

## 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 环境所有的的配置信息，包含 options，loaders，plugins 这些信息，这个对象在 Webpack 启动时候被实例化，它是全局唯一的，可以简单地把它理解为 Webpack 实例；
* Compilation 对象包含了当前的模块资源、编译生成资源、变化的文件等。当 Webpack 以开发模式运行时，每当检测到一个文件变化，一次新的 Compilation 将被创建。Compilation 对象也提供了很多事件点回调供插件做扩展。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 只是代表了一次新的编译。

## 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，多个流程之间有存在依赖关系，只有完成当前处理后才能交给下一个流程去处理。 插件就像是一个插入到生产线中的一个功能，在特定的时机对生产线上的资源做处理。

Webpack 通过 *Tapable(https://github.com/webpack/tapable)* 来组织这条复杂的生产线。 Webpack 在运行过程中会广播事件，插件只需要监听它所关心的事件，就能加入到这条生产线中，去改变生产线的运作。 Webpack 的事件流机制保证了插件的有序性，使得整个系统扩展性很好。

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：

/\*\*  
\* 广播出事件  
\* 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 使用方法和上面一致。

在开发插件时，你可能会不知道该如何下手，因为你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，就能广播出新的事件，所以在新开发的插件中也能广播出事件，给其它插件监听使用。 - 传给每个插件的 Compiler 和 Compilation 对象都是同一个引用。也就是说在一个插件中修改了 Compiler 或 Compilation 对象上的属性，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，第二个参数为回调函数，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不往下执行   
 callback();  
});  
```

## 常用 API

插件可以用来修改输出文件、增加输出文件、甚至可以提升 Webpack 性能、等等，总之插件通过调用 Webpack 提供的 API 能完成很多事情。 由于 Webpack 提供的 API 非常多，有很多 API 很少用的上，又加上篇幅有限，下面来介绍一些常用的 API。

### 读取输出资源、代码块、模块及其依赖

有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。

在 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 能读取组成代码块的每个模块  
 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 去生成输出的文件资源，每个 Chunk 都对应一个及其以上的输出文件  
 // 例如在 Chunk 中包含了 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 存放当前所有即将输出的资源  
 // 调用一个输出资源的 source() 方法能获取到输出资源的内容  
 let source = compilation.assets[filename].source();  
 });  
 });  
  
 // 这是一个异步事件，要记得调用 callback 通知 Webpack 本次事件监听处理结束。  
 // 如果忘记了调用 callback，Webpack 将一直卡在这里而不会往后执行。  
 callback();  
 })  
 }  
}

### 监听文件变化

在*4-5使用自动刷新* 中介绍过 Webpack 会从配置的入口模块出发，依次找出所有的依赖模块，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();  
});

默认情况下 Webpack 只会监视入口和其依赖的模块是否发生变化，在有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 文件不会去导入 HTML 文件，Webpack 就不会监听 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，我们需要把 HTML 文件加入到依赖列表中，为此可以使用如下代码：

compiler.plugin('after-compile', (compilation, callback) => {  
 // 把 HTML 文件添加到文件依赖列表，好让 Webpack 去监听 HTML 模块文件，在 HTML 模版文件发生变化时重新启动一次编译  
 compilation.fileDependencies.push(filePath);  
 callback();  
});

### 修改输出资源

有些场景下插件需要修改、增加、删除输出的资源，要做到这点需要监听 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会存放在 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();  
});

### 判断 Webpack 使用了哪些插件

在开发一个插件时可能需要根据当前配置是否使用了其它某个插件而做下一步决定，因此需要读取 Webpack 当前的插件配置情况。 以判断当前是否使用了 ExtractTextPlugin 为例，可以使用如下代码：

// 判断当前配置使用使用了 ExtractTextPlugin，  
// compiler 参数即为 Webpack 在 apply(compiler) 中传入的参数  
function hasExtractTextPlugin(compiler) {  
 // 当前配置所有使用的插件列表  
 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;  
}

## 实战

下面我们举一个实际的例子，带你一步步去实现一个插件。

该插件的名称取名叫 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把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。使用该插件时方法如下：

module.exports = {  
 plugins:[  
 // 在初始化 EndWebpackPlugin 时传入了两个参数，分别是在成功时的回调函数和失败时的回调函数；  
 new EndWebpackPlugin(() => {  
 // Webpack 构建成功，并且文件输出了后会执行到这里，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，需要借助两个事件：

* **done**：在成功构建并且输出了文件后，Webpack 即将退出时发生；
* **failed**：在构建出现异常导致构建失败，Webpack 即将退出时发生；

实现该插件非常简单，完整代码如下：

class EndWebpackPlugin {  
  
 constructor(doneCallback, failCallback) {  
 // 存下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;

从开发这个插件可以看出，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工作原理概括* 中详细介绍过 Webpack 在运行过程中广播出常用事件，你可以从中找到你需要的事件。

本实例*提供项目完整代码(https://github.com/gwuhaolin/end-webpack-plugin)*

# 5-5 调试 Webpack

在编写 Webpack 的 Plugin 和 Loader 时，可能执行结果会和你预期的不一样，就和你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，本节将教你如何断点调试 *5-1工作原理概括(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 之上，调试 Webpack 就相对于调试 Node.js 程序。

## 在 Webstorm 中调试

Webstorm 集成了 Node.js 的调试工具，因此使用 Webstorm 调试 Webpack 非常简单。

### 1. 设置断点

在你认为可能出现问题的地方设下断点，点击编辑区代码左侧出现红点表示设置了断点，如图：



图5-5-1 设置断点

### 2. 配置执行入口

告诉 Webstorm 如何启动 Webpack，由于 Webpack 实际上就是一个 Node.js 应用，因此需要新建一个 Node.js 类型的执行入口。

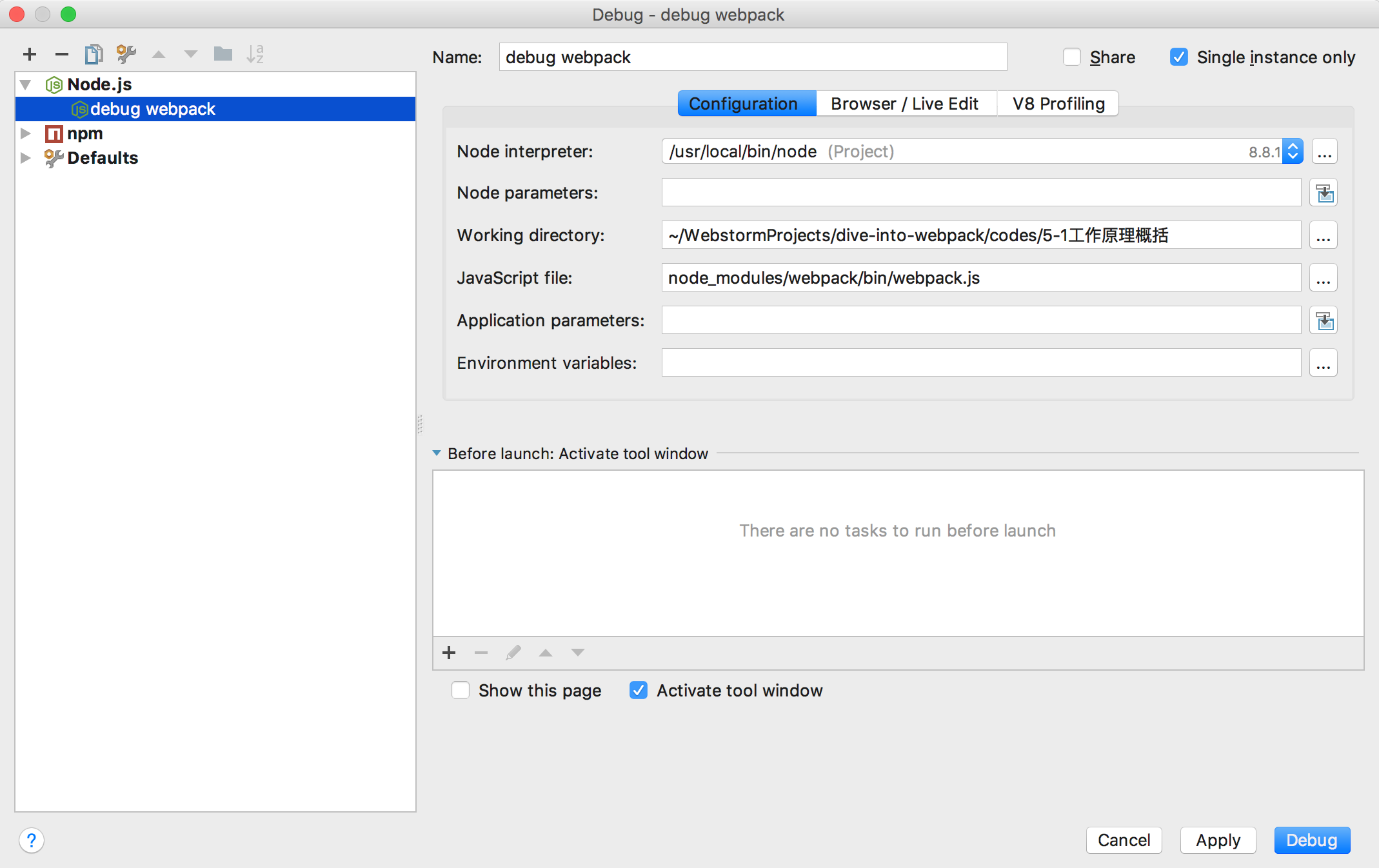


图5-5-2 配置执行入口

以上配置中有三点需要注意：

* Name 设置成了 debug webpack，就像设置了一个别名，方便记忆和区分；
* Working directory 设置为需要调试的插件所在的项目的根目录；
* JavaScript file 即 Node.js 的执行入口文件，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，下面启动调试，启动时选中前面设置的 debug webpack，方法如图

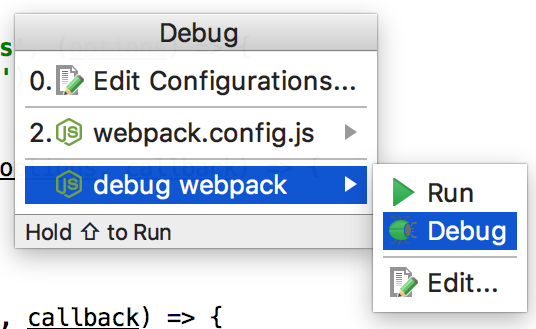


图5-5-3 启动 Webpack

### 4. 执行到断点

启动后程序就会停在断点所在的位置，在这里你可以方便的查看变量当前的状态，找出问题，如图：

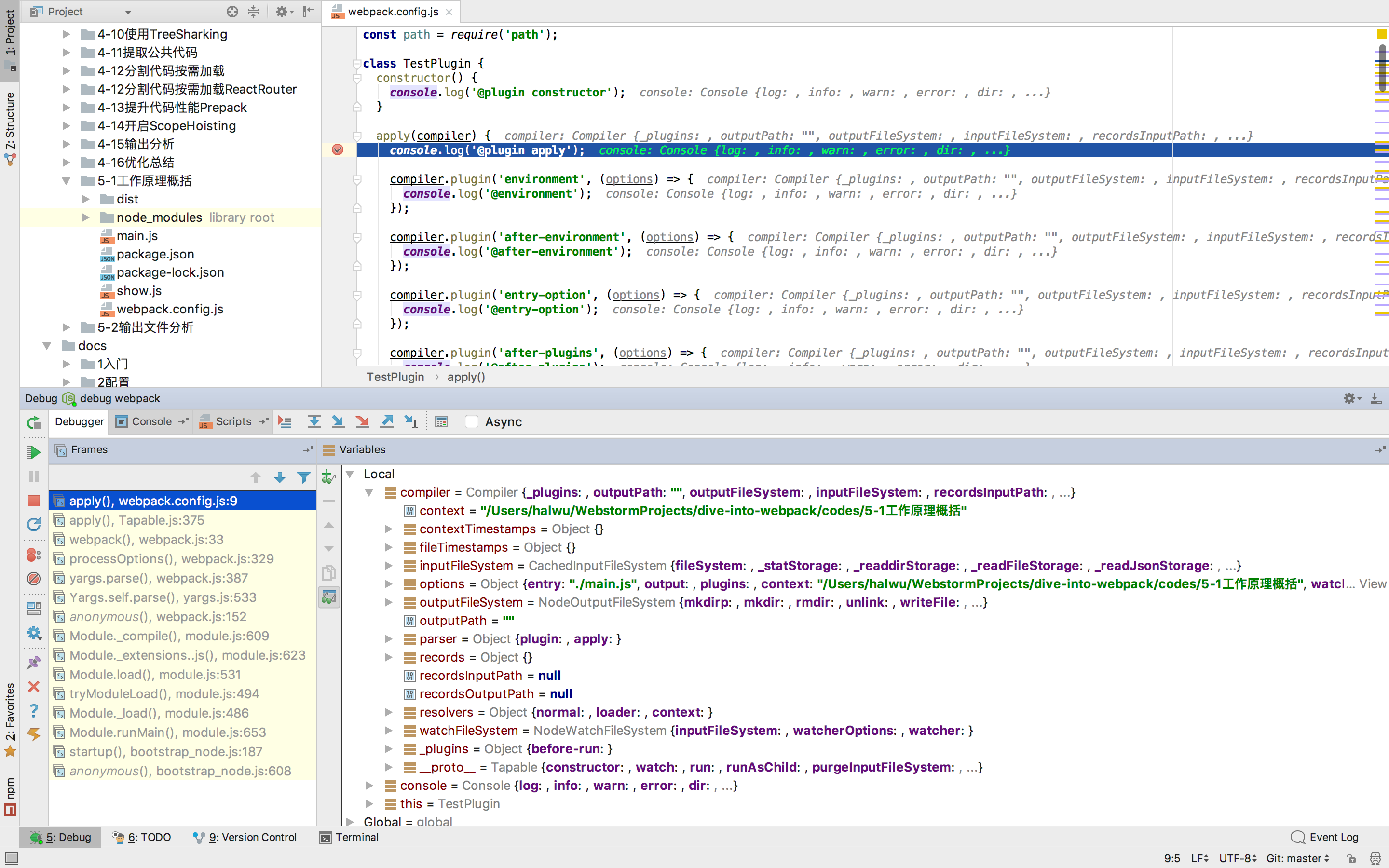


图5-5-4 执行到断点

VSCode 的断点调试方法和 Webstorm 很类似，这里就不重复介绍。

除了以上调试方法外，你还可以通过 Node.js 自带的 \* Debugger(https://nodejs.org/api/debugger.html)\* 断点调试。

# 5-6 原理总结

Webpack 是一个庞大的 Node.js 应用，如果你阅读过它的源码，你会发现实现一个完整的 Webpack 需要编写非常多的代码。 但你无需了解所有的细节，只需了解其整体架构和部分细节即可。

对 Webpack 的使用者来说，它是一个简单强大的工具； 对 Webpack 的开发者来说，它是一个扩展性的高系统。

Webpack 之所以能成功，在于它把复杂的实现隐藏了起来，给用户暴露出的只是一个简单的工具，让用户能快速达成目的。 同时整体架构设计合理，扩展性高，开发扩展难度不高，通过社区补足了大量缺失的功能，让 Webpack 几乎能胜任任何场景。

通过本章的学习，希望你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。