Broadview®
www.broadview.com.cn

前端撷英馆

React全栈

Redux+Flux+webpack+Babel整合开发

张轩 杨寒星 著

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

内 容 简 介

本书从现代前端开发的标准、趋势和常用工具入手，由此引出了优秀的构建工具webpack 和JavaScript库React，之后用一系列的实例来阐述两者的特色、概念和基本使用方法。随着应用复杂度的增加，进而介绍了Flux 和Redux 两种架构思想，并且使用Redux 对现有程序进行改造，最后介绍了在开发过程中出现的反模式和性能优化方法。

本书适合有一定前端开发尤其是JavaScript 基础的读者阅读，如果您还没有接触过前端开发这个领域，请先阅读前端开发的入门书籍。

图书在版编目（CIP）数据

React 全栈：Redux+Flux+webpack+Babel 整合开发 / 张轩，杨寒星著. —北京：电子工业出版社，2016.10
（前端撷英馆）
ISBN 978-7-121-29899-8

Ⅰ. ①R… Ⅱ. ①张… ②杨… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP 数据核字(2016)第219048 号

策划编辑：张春雨
责任编辑：付 睿
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173 信箱　邮编100036
开　　本：787×980　1/16　印张：14　字数：251 千字
版　　次：2016 年10 月第1 版
印　　次：2016 年10 月第1 次印刷
定　　价：69.00 元

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

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

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

前　言

对一个前端工程师来说，这是最坏的时代，也是最好的时代。

在这样的领域里，每一年都不会风平浪静。如果说 2014 年是属于 MVVM，属于 Angular 的，那么 2015 年称为 React 元年并不为过。开发团队的不断完善以及 React 社区井喷式的发展让这个诞生于 2013 年的框架及其生态趋于成熟（就在不久前，React 官方宣布将在版本号 0.14.7 后直接使用版本号 15.0.0），大量团队在生产环境中的实践经验也让引入 React 不再是一件需要瞻前顾后反复调研的事情，如果 React 适合你，那么现在就可以放心地使用了。

可是对于很多还没有深入实践过 React 开发的工程师来说，React 到底做了什么？React 适合什么样的场景？又应该如何投入使用？在具体业务逻辑的实现上，怎样才是最佳的实践？这些都是需要去了解与思考的问题。

本书将从一个传统前端工程师的角度出发，介绍 React 产生的背景及其架构应用，并结合一些由浅入深的例子帮助读者掌握基于 React 的 Web 前端开发方法。

——杨寒星

前端开发是一个充满变化的领域，它的发展速度快得惊人。各种各样的新技术、新标准层出不穷，GitHub 上最火的语言是 JavaScript，最大的包管理器是 npm。新的流行框架日新月异，几年前的那些先驱者还是工程师口中津津乐道的宠儿，比如 YUI、Mootools、jQuery 等，今天已经不再那么流行，曾经名噪一时的 Backbone 框架，现

在也渐渐褪去热度，继往开来的 Angular、Vue.js、Ember 等 MVVM 框架竞相登场，再加上当红的新宠 **React.js** 大行其道，让好多工程师仿佛迷失在了大潮中。

前端开发是一个新兴的行业。几年前，被称作重构工程师的我们还都在对着 Photoshop 切图，把一些 jQuery 插件复制来复制去，完成一些炫酷的幻灯图特效，不断地处理着很多 IE 浏览器的怪异 Bug。这些功力其实到现在还能满足大部分的 Web 开发，完成大部分的项目。我们不妨把它称为“古典时代”，它影响深远，但是最终会慢慢远去。

在当前这个潮流下，很多工程师会抛出这样的言论：

学习一些新的工具、框架有什么用？业界发展得这么快，等我学会了这些，它也许已经“寿终正寝”了。天天跟风一样地追求各种框架，学会了也是迷茫，这些框架没有用武之地。旁门左道，天天布道没有用的东西，伪前端。

随着技术的进化、移动应用的飞速发展，一个前端工程师的职责不像原来那样只要把图转换成网页那么简单。如今产生了各种类型的新名词——Hybird 应用、全端工程师、SPA 等，各有其特定的应用场景。任何框架的发明和创造都有它们一定的历史原因，只有解决了需求的痛点，才能让工程师更快地解决难题。在我们学习的过程中，可以发现它背后的思想和解决方案，进而更好地充实自己。做技术的人最重要的就是保持开放的态度，有一颗好奇心，持续不断地学习。

在前端开发中占最重要部分的 JavaScript，也随着这些框架在慢慢进化着，原来令人不断诟病的缺点正在被标准制定者慢慢修补，新的特性不断浮出水面。前端工程师正处在发展最迅速的时代，这应该是一个让人兴奋的时代，犹如工业革命一样，每个工程师都见证着一场伟大的前端革命。

本书不仅讲述了怎样使用 React 和 webpack 开发一些应用，而且希望通过一系列的介绍让每个工程师都能站在前端技术的潮头，拥抱变化，学习新的标准和技术，成为新技术的弄潮儿。

——张轩

本书面向的读者

本书适合有一定前端开发经验尤其是有 JavaScript 基础的读者，如果你还没有接触过前端开发这个领域，请先阅读前端开发的入门书籍。

本书的代码示例

你可以在这里下载本书的代码示例：https://github.com/vikingmute/webpack-react-codes。

本书的代码执行环境

本书中默认的开发环境是 Node.js 5.0.0，书中介绍到的几个库的版本分别为 React@15.0.1、webpack@1.12.14 及 Redux@3.2.1，其他如未特别说明的则为最新版本。

目　录

第 2 章　webpack

如今，前端项目日益复杂，构建系统已经成为开发过程中不可或缺的一个部分，而模块打包（module bundler）正是前端构建系统的核心。

正如前面介绍到的，前端的模块系统经历了长久的演变，对应的模块打包方案也几经变迁。从最初简单的文件合并，到 AMD 的模块具名化并合并，再到 browserify 将 CommonJS 模块转换成为浏览器端可运行的代码，打包器做的事情越来越复杂，角色也越来越重要。

在这样一个竞争激烈的细分领域中，webpack 以极快的速度风靡全球，成为当下最流行的打包解决方案，并不是偶然。它功能强大、配置灵活，特有的 code spliting 方案正戳中了大规模复杂 Web 应用的痛点，简单的 loader/plugin 开发使它很快拥有了丰富的配套工具与生态。

对多种模块方案的支持与视一切资源为可管理模块的思路让它天然地适合 React 项目的开发，成为 React 官方推荐的打包工具。在本章中将着重介绍 webpack 这个工具的特点与使用，作为接下来使用 webpack 辅助开发 React 项目的准备。

2.1　webpack 的特点与优势

正如前面提到的，打包工具也有不同的开源方案，那么相比其他的流行打包工

具，webpack 有着怎样的特点与优势呢？本节将对 RequireJS、browserify 及 webpack 这三者做一个全面的比较。

2.1.1 webpack 与 RequireJS、browserify

首先对三者做一下简要的介绍。

RequireJS 是一个 JavaScript 模块加载器，基于 AMD 规范实现。它同时也提供了对模块进行打包与构建的工具 r.js，通过将开发时单独的匿名模块具名化并进行合并，实现线上页面资源加载的性能优化。这里拿来对比的是由 RequireJS 与 r.js 等一起提供的一个模块化构建方案。开发时的 RequireJS 模块往往是一个个单独的文件，RequireJS 从入口文件开始，递归地进行静态分析，找出所有直接或间接被依赖（require）的模块，然后进行转换与合并，结果大致如下（未压缩）。

```
// bundle.js
define('hello', [], function (require) {
   module.exports = 'hello!';
});
define('say', ['require', 'hello'], function (require) {
   var hello = require('./hello');
   alert(hello);
});
```

browserify 是一个以在浏览器中使用 Node.js 模块为出发点的工具。它最大的特点在于以下两点。

① 对 CommonJS 规范（Node.js 模块所采用的规范）的模块代码进行的转换与包装。

② 对很多 Node.js 的标准 package 进行了浏览器端的适配，只要是遵循 CommonJS 规范的 JavaScript 模块，即使是纯前端代码，也可以使用它进行打包。

webpack 则是一个为前端模块打包构建而生的工具。它既吸取了大量已有方案的优点与教训，也解决了很多前端开发过程中已存在的痛点，如代码的拆分与异步加载、对非 JavaScript 资源的支持等。强大的 loader 设计使得它更像是一个构建平台，而不只是一个打包工具。

2.1.2 模块规范

模块规范是模块打包的基础，我们首先对这三者所支持的模块化方案进行比较。

RequireJS 项目本身是最流行的 AMD 规范实现，格式如下。

```
// hello.js
define(function (require) {
   module.exports = 'hello!';
});
```

AMD 通过将模块的实现代码包在匿名函数（即 AMD 的工厂方法，factory）中实现作用域的隔离，通过文件路径作为天然的模块 ID 实现命名空间的控制，将模块的工厂方法作为参数传入全局的 define（由模块加载器事先定义），使得工厂方法的执行时机可控，也就变相模拟出了同步的局部 require，因而 AMD 的模块可以不经转换地直接在浏览器中执行。因此，在开发时，AMD 的模块可以直接以原文件的形式在浏览器中加载执行并调试，这也成为 RequireJS 方案不多的优点之一。

browserify 支持的则是符合 CommonJS 规范的 JavaScript 模块。不严格地说，CommonJS 可以看成去掉了 define 及工厂方法外壳的 AMD。上述 hello.js 对应的 CommonJS 版本是如下这样的。

```
// hello.js
module.exports = 'hello!';
```

正如我们在前面提到的 define 函数的作用，没有 define 函数的 CommonJS 模块是无法直接在浏览器中执行的——浏览器环境中无法实现同 Node.js 环境一样同步的 require 方法。同样也因为没有 define 与工厂方法，CommonJS 模块书写起来要更简单、干净。在这个显而易见的好处下，越来越多的前端项目开始采用 CommonJS 规范的模块书写方式。

考虑到 AMD 规范与 CommonJS 规范各有各的优点，且都有着可观的使用率，webpack 同时支持这两种模块格式，甚至支持二者混用。而且通过使用 loader，webpack 也可以支持 ES6 module（这一特性在即将到来的 webpack 2 中原生支持），可以说覆盖了现有的所有主流的 JavaScript 模块化方案。通过特定的插件实现 shim 后，在 webpack 中，甚至可以把以最传统全局变量形式暴露的库当作模块 require 进来。

2.1.3 非 javascript 模块支持

在现代的前端开发中，组件化开发成为越来越流行的趋势。将局部的逻辑进行封装，通过尽可能少的必要的接口与其他组件进行组装与交互，可以将大的项目逻辑拆成一个个小的相对独立的部分，减少开发与维护的负担。在传统的前端开发中，页面的局部组成所依赖的各种资源（JavaScript、CSS、图片等）是分开维护的，一个常见的目录组织方式（以 Less 为例对样式代码进行组织）如下。

```
- static/
   - javascript/
      - main.js
      - part-A/
      - ...
   - less/
      - main.less
      - part-A/
      - ...
   - ...
```

这意味着一个局部组件（如 part A）的引入至少需要：

- 在 main.js 中引入（require）part A 对应的 JavaScript 文件。
- 在 main.less 中引入（import）part A 对应的 Less 文件。

如果 part A 需要用到特定的模板，可能还需要在页面 HTML 文件中插入特定 ID 的 template 标签。而引入组件的入口越多，意味着组件内部与外部需要的约定越多，耦合度也越高。因此减少组件的入口文件数，尽可能将其所有依赖进行内部声明，可以提高组件的内聚度，便于开发与维护，这也是模块打包工具支持多种前端资源的意义所在。如上例中，在打包工具支持 Less 资源依赖的引入与合并的情况下，目录结构可以改成：

```
- app/
   - main/
      - index.js
      - index.less
   - part-A/
      - index.js
      - index.less
      - ...
```

```
    - ...
```

part A 的样式实现从 JavaScript 中直接引入。

```
// part-A/index.js
require('./index.less');
```

这样，仅需在 main/index.js 里声明对 part-A/index.js 的依赖，即可实现对组件 part A 的引入。说了这么多，我们来看一下这里提到的 3 个打包方案对非 JavaScript 模块资源的支持情况。

很多人不知道的是，RequireJS 是支持除 AMD 格式的 JavaScript 模块以外的其他类型的资源加载的，而且有着相当丰富的 plugin，从纯文本到模板，从 CSS 到字体等都有覆盖。然而基于 AMD 规范的非 JavaScript 资源加载有着本质的如下缺陷。

- 加载与构建的分离导致 plugin 需要分别实现两套逻辑。
- 浏览器的安全策略决定了绝大多数需要读取文本内容进行解析的静态资源无法被跨域加载（即使是 JavaScript 模块本身，也要依靠 define 方法包裹，类似于 JSONP 原理实现的跨域加载）。

因此在 RequireJS 的方案中，非 JavaScript 模块的资源虽然得到了支持，但支持得并不完善。

browserify 可以通过各种 transform 插件实现不同类型资源的引入与打包。

在 webpack 中，与 browserify 的 transform 相对应的是 loader，但是功能更加丰富。

2.1.4 构建产物

另外一个三者较大的区别在于构建产物。r.js 构建的结果是上述 define(function(){...})的集合。其结果文件的执行依赖页面上事先引入一个 AMD 模块加载器（如 RequireJS 自身），所以常见的 AMD 项目线上页面往往存在两个 JavaScript 文件：loader.js 及 bundle.js。而 browserify 与 webpack 的构建结果都是可以直接执行的 JavaScript 代码。它们也都支持通过配置生成符合特定格式的结果文件，如以 UMD 的形式暴露库的 exports，以便其他页面代码调用。后者的这种形式更加适用于

JavaScript 库（library）的构建。

2.1.5 使用

在使用上，三者也是有较大差异的。

作为 npm 包的 RequireJS 提供了一个可执行的 r.js 工具，通过命令行执行，使用方式如下。

```
npm install -g requirejs
r.js -o app.build.js
```

RequireJS 包也可以作为一个本地的 Node.js 依赖被安装，然后通过函数调用的形式执行打包。

```
var requirejs = require('requirejs');
requirejs.optimize({
   baseUrl: '../appDir/scripts',
   name: 'main',
   out: '../build/main-built.js'
}, function (buildResponse) {
   // success callback
}, function(err) {
   // err callback
});
```

显然，前者使用更简单，而后者更适合需要进行复杂配置的场合。不过 r.js 的可配置项相当有限，其功能也比较简单，仅仅是实现了 AMD 模块的合并，并输出为字符串。如果需要如监视等功能，则需要自己编码实现。

browserify 提供的命令行工具，用法与 r.js 很像，相当简洁。

```
npm install -g browserify
browserify main.js -o bundle.js
```

不过，它通过对大量配置项的支持，使得仅仅通过命令行工具也可以进行较复杂的任务。通过 browserify --help 及 browserify --help advanced 可以查看所有的配置项，覆盖了从输入/输出位置、格式到使用插件等各个方面。

browserify 同样支持直接调用其 Node.js 的 API。

```
var browserify = require('browserify');
var b = browserify();
b.add('./browser/main.js');
b.bundle().pipe(process.stdout);
```

通过调用 browserify 提供的方法，手工实现脚本构建，可以进行更为灵活的配置及精细的流程控制。

webpack 的使用与前两者大同小异，主要也支持命令行工具及 Node.js 的 API 两种使用方式，前者更常用一点，最简单的形式如下。

```
npm install webpack -g
webpack main.js bundle.js
```

不过它的特点是，虽然它会支持部分命令行参数形式的配置项，但是其主要配置信息需要通过额外的文件（默认是 webpack.config.js）进行配置。这个文件只需要是一个 Node.js 模块，且 export 一个 JavaScript 对象作为配置信息。相比命令行参数式配置，这种配置方式更为灵活强大，因为配置文件会在 Node.js 环境中运行，甚至可以在其中 require 其他模块，这样对复杂项目中不同任务的配置信息进行组织变得更容易。例如，可以实现一个 webpack.config.common.js，然后分别实现 webpack.config.dev.js 与 webpack.config.prod.js，用于开发环境与生产环境的构建（通过命令行参数指定配置文件），后两者可以直接通过 require 使用 webpack.config.common.js 中的公共配置信息，并在此基础上添加或修改以实现各自特有的部分。

得益于 webpack 众多的配置项、强大的配置方式以及丰富的插件体系，大多数时候，我们仅仅书写配置文件，然后通过命令行工具就可以完成项目的构建工作。不过，webpack 也提供了 Node.js 的 API，使用也很简单。

```
var webpack = require("webpack");

//返回一个 Compiler 实例
webpack({
   //webpack 配置
}, function(err, stats) {
   //……
```

```
});
```

2.1.6 webpack 的特色

在经过多方面的对比之后，我们能发现，在吸取了各前辈优点的基础上，webpack 几乎在每个方面都做到了优秀。不过除此之外，webpack 还有一些特色功能也是不得不提的。

1．代码拆分（code splitting）方案

对于较大规模的 Web 应用（特别是单页应用），把所有代码合并到单个文件是比较低效的做法，单个文件体积过大会导致应用初始加载缓慢。尤其如果其中很多逻辑只在特定情况下需要执行，每次都完整地加载所有模块就变得很浪费。webpack 提供了代码拆分的方案，可以将应用代码拆分为多个块（chunk），每个块包含一个或多个模块，块可以按需被异步加载。这一特性最早并不是由 webpack 提出的，但 webpack 直接使用模块规范中定义的异步加载语法作为拆分点，将这一特性实现得极为简单易用，下面以 CommonJS 规范为例。

```
require.ensure(["module-a"], function(require) {
   var a = require("module-a");
});
```

如上例，通过 require.ensure 声明依赖 module-a，module-a 的实现及其依赖会被合并为一个单独的块，对应一个结果文件。当执行到 require.ensure 时才去加载 module-a 所在的结果文件，并在 module-a 加载就绪后再执行传入的回调函数。其中的加载行为及回调函数的执行时机控制都由 webpack 实现，这对业务代码的侵入性极小。在真实使用中，需要被拆分出来的可能是某个体积较大的第三方库（延后加载并使用），也可能是一个点击触发浮层的内部逻辑（除非触发条件得到满足，否则不需要加载执行），将这些内容按需地异步加载可以让我们以较小的代价，来极大地提升大规模单页应用的初始加载速度。

2．智能的静态分析

熟悉 AMD 规范的都知道，在 AMD 模块中使用模块内的 require 方法声明依赖

的时候，传入的 moduleId 必须是字符串常量，而不可以是含变量的表达式。原因在于模块打包工具在打包前需要通过静态分析获取整个应用的依赖关系，如果传入 require 方法的 moduleId 是个含变量的表达式，其值需要在执行期才能确定，那么静态分析就无法确认依赖的到底是哪个模块，自然也就没办法把这个模块的代码事先打包进来。如果依赖模块没有被事先打包进来，在执行期再去加载，那么由于网络请求的时间不可忽视，请求时阻塞 JavaScript 的执行也不可行，模块内的同步 require 也就无从实现。

在 Node.js 中，模块文件都是直接从本地文件系统读取，其加载与执行是同步的，因此 require 一个表达式成为可能，在执行到 require 方法时再根据当前传入的 moduleId 进行实时查找、加载并执行依赖模块。然而当 CommonJS 规范被用于浏览器端，如通过 browserify 进行打包，出于与 AMD 模块构建类似的考虑，这一特性也无法被支持。

虽然未能从根本上解决这个问题，webpack 在这个问题上还是尽可能地为开发者提供了便利。首先，webpack 支持简单的不含变量的表达式，如下。

```
require(expr ? "a" : "b");
require("a" + "b");
require("not a".substr(4).replace("a", "b"));
```

其次，webpack 还支持含变量的简单表达式，如下。

```
require("./template/" + name + ".jade");
```

对于这种情况，webpack 会从表达式"./template/" + name + ".jade"中提取出以下信息。

- 目录./template 下。
- 相对路径符合正则表达式：/^.*\.jade$/。

然后将符合以上条件的所有模块都打包进来，在执行期，依据当前传入的实际值决定最终使用哪个模块。

这样的特性平时并不常用，但在一些特殊的情况下会让代码变得更简洁清晰，如下。

```
function render (tplName, data) {
   const render = require('./tpls/' + tplName);
   return render(data);
}
```

作为对比，如果不依赖这样的特性，可能要像下面这样实现。

```
const tpls = {
   'a.tpl': require('./tpls/a.tpl'),
   'b.tpl': require('./tpls/b.tpl'),
   'c.tpl': require('./tpls/c.tpl'),
};

function render (tplName, data) {
   const render = tpls[tplName];
   return render(data);
}
```

一方面，代码变得冗长了；另一方面，当添加新的 tpl 时，不仅需要向./tpls 目录添加新的模板文件，还需要手动维护这里的 tpls 表，这增加了编码时的心理负担。

3．模块热替换（Hot Module Replacement）

在传统的前端开发中，每次修改完代码都需要刷新页面才能让改动生效，并验证改动是否正确。虽然像 LiveReload 这样的功能可以帮助我们自动刷新页面，但当项目变大时，刷新页面往往要耗时好几秒，只有等待页面刷新完成才能验证改动。而且有些功能需要经过特定的操作、应用处于特定状态时才能验证，刷新完页面后还需要手动操作并恢复状态，较为烦琐。针对这一问题，webpack 提供了模块热替换的能力，它使得在修改完某一模块后无须刷新页面，即可动态将受影响的模块替换为新的模块，在后续的执行中使用新的模块逻辑。

这一功能需要配合修改 module 本身，但一些第三方工具已经帮我们做了这些工作。如配合 style-loader，样式模块可以被热替换；配合 react-hot-loader，可以对 React class 模块进行热替换。

配置 webpack 启用这一功能也相当简单，通过参数--hot 启动 webpack-dev-server 即可。

```
webpack-dev-server --hot
```

为了准确起见，需要说明的是，虽然这里说模块热替换是 webpack 的特色功能，但是有人借鉴 webpack 的方案，实现了插件 browserify-hmr，这让 browserify 也支持了模块热替换这一特性。

2.1.7 小结

除了上面介绍过的，业界还有一些其他的打包方案，如 rollup、jspm 提供的 bundle 工具等，不过它们或者还不够成熟，或者缺乏特点，所以没有在这里介绍。本节主要选取了 3 个相对成熟、主流的模块打包工具进行了比较，webpack 在功能、使用等方面均有一定的优势，且提供了一些很有用的特色功能，说它是目前最好的前端模块打包工具并不为过，这也正是越来越多的前端项目选择使用 webpack 的原因。此外，考虑到 React 官方也推荐使用 webpack，本书中介绍的 React 开发项目将全部使用 webpack 进行构建。

2.2 基于 webpack 进行开发

2.2.1 安装

webpack 是使用 Node.js 开发的工具，可以通过 npm 进行安装。npm 是 Node.js 的包管理工具，在这里我们首先需要确保 Node.js 的运行环境及已安装了 npm（安装过程可参考 Node.js 官网），然后通过 npm 进行 webpack 的安装。

```
npm install webpack -g
```

这个命令会默认安装 webpack 最新的稳定版本。本书示例中所使用的为 webpack@1.12.14。读者也可以在安装时指定版本为 1.12.14，以确保与书中保持一致的运行结果，如下。

```
npm install webpack@1.12.14 -g
```

大部分情况下需要以命令行工具的形式使用 webpack，所以我们这里将它安装在全局（-g），方便使用。有时候会希望编写自己的构建脚本，或是由项目指定需要依

赖的 webpack，在这种情况下将 webpack 安装到本地会更合适。对前端项目来说，webpack 扮演的是构建工具的角色，并不是代码依赖，应该被安装在 dev-dependencies 中，即：

```
npm install webpack --save-dev
```

在这里，采用第一种方式即可。

本章最终完成的示例代码都在 https://github.com/vikingmute/webpack-react-codes/tree/master/chapter2，读者阅读时可以进行参考。

2.2.2 Hello world

在这个示例中，将使用 webpack 构建一个简单的 Hello world 应用。应用包括两个 JavaScript 模块（完整代码见 chapter2/part1/）。

1．生成文本“Hello world! ”的 hello 模块（hello.js）。

```
module.exports = 'Hello world!';
```

2．打印文本的 index 模块（index.js）。

```
var text = require('./hello');
console.log(text);
```

页面内容（index.html）很简单。

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8">
   <title>hello</title>
</head>
<body>
   <script src="./bundle.js"></script>
</body>
</html>
```

需要注意的是，index.html 中引用的 bundle.js 并不存在，它就是我们使用 webpack 将会生成的结果文件。

现在我们的目录结构是如下这样的。

```
- index.html
- index.js
- hello.js
```

我们知道，如果在 index.html 中直接引用 index.js，代码是无法正常执行的，因为上面的代码是按照 CommonJS 的模块规范书写的，浏览器环境并不支持。那么基于 webpack 的做法是什么呢？其实很简单，一行命令就够了。

```
webpack ./index.js bundle.js
```

这个命令会告诉 webpack 将 index.js 作为项目的入口文件进行构建，并将结果输出为 bundle.js。然后就可以看到在当前目录下新增了一个文件 bundle.js，现在在浏览器中打开 index.html，bundle.js 会被加载进来并执行，控制台打印出“Hello world!”。

下面通过查看 bundle.js 的内容来分析一下 webpack 所施展的魔法到底是怎么一回事。

```
/******/ (function(modules) { // webpackBootstrap
/******/   // module 缓存对象
/******/   var installedModules = {};

/******/   // require 函数
/******/   function __webpack_require__(moduleId) {

/******/      // 检查 module 是否在 cache 中
/******/      if(installedModules[moduleId])
/******/         return installedModules[moduleId].exports;

/******/      // 新建一个 module 并且放入 cache 中
/******/      var module = installedModules[moduleId] = {
/******/         exports: {},
/******/         id: moduleId,
/******/         loaded: false
/******/      };

/******/      // 执行 module 函数
/******/      modules[moduleId].call(module.exports, module,
module.exports, __webpack_require__);

/******/      // 标记 module 已经加载
```

```
/******/      module.loaded = true;

/******/      // 返回 module 的导出模块
/******/      return module.exports;
/******/   }


/******/   // 暴露 modules 对象(__webpack_modules__)
/******/   __webpack_require__.m = modules;

/******/   // 暴露 modules 缓存
/******/   __webpack_require__.c = installedModules;

/******/   // 设置 webpack 公共路径__webpack_public_path__
/******/   __webpack_require__.p = "";

/******/   // 读取入口模块并且返回 exports 导出
/******/   return __webpack_require__(0);
/******/ })
/************************************************************************/
/******/   // webpackBootstrap 传入的参数是一个数组
/******/ ([
/* 0 */   // index.js 模板的工厂方法
/***/ function(module, exports, __webpack_require__) {

   var text = __webpack_require__(1);
   console.log(text);


/***/ },
/* 1 */   // hello.js 模块的工厂方法
/***/ function(module, exports) {

   module.exports = 'Hello world!';


/***/ }
/******/ ]);
```

整段代码的结构是一个立即执行函数表达式（IIFE），这是 JavaScript 中常见的独立作用域的方法。上段代码的匿名函数的定义旁有个注释 webpackBootstrap，这里我们就将这个函数称为 webpackBootstrap。

暂且不管 webpackBootstrap 的内部做了什么，先来看一下它的参数，webpackBootstrap 接收一个参数 modules，在函数最下面的注释中，我们看到实参是一个数组，数组的每一项都是一个匿名函数，分别定义在最后两个特定注释的地方。不难发现，这两个匿名函数的内容分别对应了刚才定义的两个模块 index 及 hello。

值得注意的是，在构建命令中只指定了 index 模块所对应的 JavaScript 文件，webpack 通过静态分析语法树，递归地检测到了所有依赖模块，以及依赖的依赖，并合并到最终的代码中。

这里的匿名函数称为工厂方法（factory），即运行就可以得到模块的方法，就像一个生产特定模块的工厂一样。如果你了解过 AMD 模块或 Node.js 中 CommonJS 模块运行的机制，你应该不会对这种将代码包装成工厂方法的做法感到陌生。模块代码被包装成函数之后，其运行时机变得可控，而且也拥有了独立的作用域，定义变量、声明函数都不会污染全局作用域。不过如果你细心的话，就不难发现工厂方法的内部代码与实现的模块源代码还是有区别的。require("./hello") 这个表达式被替换成了__webpack_require__(1)，对应地，工厂方法的参数列表中除了 CommonJS 规范所要求的 module 与 exports 外还包含了__webpack_require__，即用来替换 require 的方法。__webpack_require__提供的功能与 require 是一致的：声明对其他模块的依赖并获得该模块的 exports。不同之处在于__webpack_require__不需要提供模块的相对路径或其他形式的 ID，直接传入该模块在 modules 列表中的索引值即可。

那么这个替换有什么好处呢？首先，我们知道，CommonJS 中的 require 方法接收一个模块标识符（module identifier）作为参数，而模块标识符有以下两种形式。

- .或..开头的相对 ID（Relative ID），如./hello。
- 非.或..开头的顶级 ID（Top-Level ID），如 hello。

而不管是哪种形式，文件的“.js”后缀名都是可选省略的。也就是说，在指定了根目录（如这里指定 index.js 与 hello.js 所在的目录）的情况下，index 模块依赖 hello 模块有以下 4 种写法。

- require('./hello')
- require('./hello.js')
- require('hello')

- require('hello.js')

然而，这 4 种写法所指向的 hello 模块是同一个，从模块标识符到真实模块映射关系的实现被称为模块标识符的解析（resolve）过程。而使用__webpack_require__的好处在于，其接收的参数（数组 modules 中的索引值）与真实模块的实现是一一对应的，也就省掉了模块标识符的解析过程（准确地说，是把解析过程提前到了构建期），从而可以获得更好的运行性能。

然后，来看一下让我们的代码真正拥有了在浏览器环境中执行能力的函数 webpackBootstrap。这里的 Bootstrap 跟 UI 框架 Bootstrap 没什么关系，计算机领域中常用来表达引导程序的意思，如操作系统的启动过程。同样地，webpackBootstrap 函数是整个应用的启动程序。

首先，它通过参数 modules 获取到所有模块的工厂方法，接着在此基础上构造了__webpack_require__方法。__webpack_require__就是刚才提到的会传递给模块的工厂方法，用于加载指定模块的方法。加载模块的过程很简单，从 modules 数组中获得指定索引值所对应的项（即指定模块的工厂方法），构造一个空的 module，作为参数调用工厂方法。工厂方法的执行结果会体现在 module.exports 上，返回该内容即可。这边通过 installedModules 缓存了模块工厂方法的执行结果，确保了每个模块的实现代码只会执行一次，后续的调用会直接返回已缓存的结果。

构造完__webpack_require__之后，在之后直接使用这个方法执行了入口模块（webpack 构建时，会将入口模块放在数组 modules 的第 1 项）。至此，应用的引导启动便完成了。入口模块内部会继续通过传入的__webpack_require__方法执行其依赖的模块，整个应用便运行了起来。

总结一下的话，webpack 主要做了两部分工作，如下。

- 分析得到所有必需模块并合并。
- 提供了让这些模块有序、正常执行的环境。

2.2.3 使用 loader

通过最简单的 Hello world 应用，我们大概了解了 webpack 基本的使用与工作原

理。在这一点上，各模块打包工具基本都是一致的，下面将进一步了解 webpack 的一些特别而强大的功能。首先要介绍的就是 loader。下面将借助 webpack 的官方文档的来定义一下 loader。

```
    Loaders are transformations that are applied on a resource file of your app. They are functions (running in node.js) that take the source of a resource file as the parameter and return the new source.
```

翻译一下，“loader 是作用于应用中资源文件的转换行为。它们是函数（运行在 Node.js 环境中），接收资源文件的源代码作为参数，并返回新的代码。”举个例子，你可以通过 jsx-loader 将 React 的 JSX 代码转换为 JS 代码，从而可以被浏览器执行。

在本节中，将以前端开发的另一个主要开发内容 CSS 为例，介绍一下 loader 的功能与使用（完整代码见 chapter2/part2/）。在 webpack 中，每个 loader 往往表现为一个命名为 xxx-loader 的 npm 包，针对特定的资源类型（xxx）进行转换。而为了将 CSS 资源添加到项目中，下面要介绍两个 loader：style-loader 与 css-loader。前者将 CSS 代码以<style>标签的形式插入到页面上从而生效；后者通过检查 CSS 代码中的 import 语句找到依赖并合并。大部分情况下，我们将二者搭配使用。首先要安装这两个 loader 对应的 npm 包（你需要先在该目录下添加 package.json 文件或通过 npm init 自动生成）。

```
    npm install style-loader css-loader --save-dev
```

接着创建一个简单的 CSS 文件 index.css。

```
    div {
       width: 100px;
       height: 100px;
       background-color: red;
    }
```

我们在入口文件 index.js 中通过 require 方法引入 index.css。

```
    index.js:
    require('style!css!./index.css');
    document.body.appendChild(document.createElement('div'));
```

注意这里的 style!css!，类似 xxx!这样的写法是为了指定特定的 loader。这里是告诉 webpack 使用 style-loader 及 css-loader 这两个 loader 对 index.css 的内容进行处理。

然后在页面上创建一个 div 元素，以验证在 index.css 中编写的样式是否生效。

然后同样执行以下命令。

```
webpack ./index.js bundle.js
```

得到结果文件后，在页面中引入 bundle.js，在浏览器中打开页面即可看到效果。

与常规的前端开发不同的是，我们的页面上最终并没有插入<link>标签，结果文件中也没有 CSS 文件，却通过引入一个 JS 文件实现了样式的引入。这正是 webpack 的特点之一，任何类型的模块（资源文件），理论上都可以通过被转化为 JavaScript 代码实现与其他模块的合并与加载。webpack 官网的这张图（如图 2-1 所示）也很好地体现了这一点。

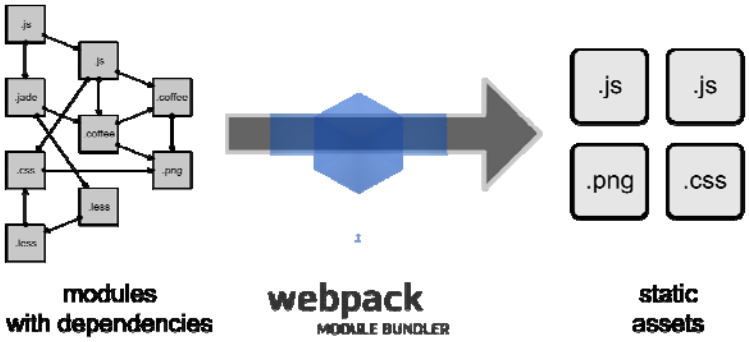

图 2-1　什么是 webpack

正如前面所述，这里通过 JavaScript 加载 CSS 是借助了 style-loader 的能力（将 CSS 代码以<style>标签的形式插入到页面，标签内容通过 JavaScript 生成）。与传统的页面直接插入标签相比，该方法也存在着不可忽视的缺陷：样式内容的生效时间被延后。

如果遵循常见的前端页面性能优化建议，一般会把<link>插入在页面的<head>中，而把<script>放在<body>的最后，这样在文档被解析到<head>的时候，样式文件就会被下载并解析，JavaScript 内容则会被延后到整个文档几乎被解析完成时才被加载与执行。在现在的做法中，样式内容其实是与 JavaScript 内容一起加载的，它的插入与解析甚至会被延后到 JavaScript 内容的执行期。相比前者，生效时间不可避免地

会晚很多，因而如果页面上本来就有内容，这部分内容会有一个短暂的无样式的瞬间，用户体验很不好。

当然，这个缺陷是可以避免的。借助 extract-text-webpack-plugin 这个插件，webpack 可以在打包时将样式内容抽取并输出到额外的 CSS 文件中，然后在页面中直接引入结果 CSS 文件即可。插件（plugin）在 webpack 的使用中是另一个很重要的概念，我们将在后面详细介绍 webpack 的插件及其使用。

2.2.4 配置文件

2.2.3 节介绍了使用 webpack 及其 loader 进行前端代码构建的方法，然而它还不够简单。

- 每次构建都需要指定项目的入口文件（./index.js）与构建输出文件（bundle.js）。
- 使用 loader 需要以 xxx!的形式指定，意味着每个有 require CSS 资源的地方，都需要写成如下形式。

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

作为天生厌倦重复劳动的程序员，我们有没有办法把这些事做得更优雅一点呢？答案就是本节的内容。

本节将介绍如何通过配置文件的形式对 webpack 的构建行为进行配置（完整代码见 chapter2/part3/），这也是 webpack 与 RequireJS、browserify 等相比一个很便利的特性。

webpack 支持 Node.js 模块格式的配置文件，默认会使用当前目录下的 webpack.config.js，配置文件只需要 export 的一个配置信息对象即可，形式如下。

```
module.exports = {
   // configuration
};
```

首先将以 2.2.3 节内容为例，介绍一些配置文件的编写及使用。一个最简单的配置信息对象包含以下信息。

- entry 项目的入口文件。
- output 构建的输出结果描述。本身是一个对象，包括很多字段，比较重要的如下。
  - path：输出目录。
  - filename：输出文件名。
  - publicPath：输出目录所对应的外部路径（从浏览器中访问）。

其中 publicPath 是一个很容易被忽略但是很重要的配置，它表示构建结果最终被真正访问时的路径。一个常见的前端构建上线过程是这样的：配置构建输出目录为 dist，构建完成后对 dist 目录进行打包，然后将其内容（结果文件往往会不止一个）发布到 CDN 上。比如其中的 dist/bundle.js，假设它最终发布的线上地址为 http://cdn.example.com/static/bundle.js，则这里的 publicPath 应当取输出目录（dist/）所对应的线上路径，即 http://cdn.example.com/static/。在我们的演示项目中，直接通过相对路径访问静态资源，不涉及打包上线 CDN 的过程，故不做配置。

所以对于先前的例子，我们的配置文件是以下这样的（webpack.config.js）。

```
var path = require('path');
module.exports = {
   entry: path.join(__dirname, 'index'),
   output: {
         path: __dirname,
         filename: 'bundle.js'
   },
   module: {
      loaders: [
         {
            test: /\.css$/,
            loaders: ['style', 'css']
         }
      ]
   }
};
```

其中 module 字段是上面没有介绍到的，module.loaders 是对于模块中的 loader 使用的配置，值为一个数组。数组的每一项指定一个规则，规则的 test 字段是正则表达式，若被依赖模块的 ID 符合该正则表达式，则对依赖模块依次使用规则中 loaders

字段所指定的 loader 进行转换。在这里，我们配置了对所有符合/\.css$/，即后缀名为.css 的资源使用 style-loader 与 css-loader，这样的话在 JavaScript 代码中 require CSS 模块的时候就不用每次都写一遍 style!css!了，只需要像依赖 JavaScript 模块一样写成：

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

这样每次构建的时候也不需要手动指定入口文件与输出文件了，直接在项目目录下执行：

```
webpack
```

webpack 会默认从 webpack.config.js 中获取配置信息，并执行构建过程，是不是方便很多呢？

2.2.5 使用 plugin

除了 loader 外，plugin（插件）是另一个扩展 webpack 能力的方式。与 loader 专注于处理资源内容的转换不同，plugin 的功能范围更广，也往往更为灵活强大。plugin 的存在可以看成是为了实现那些 loader 实现不了或不适合在 loader 中实现的功能，如自动生成项目的 HTML 页面（HtmlWebpackPlugin）、向构建过程中注入环境变量（EnvironmentPlugin）、向块（chunk）的结果文件中添加注释信息（BannerPlugin）等。

1．HtmlWebpackPlugin

webpack 内置了一些常用的 plugin，如上面提到的 EnvironmentPlugin 及 BannerPlugin，更多第三方的 plugin 可以通过安装 npm 包的形式引入，如 HtmlWebpackPlugin 对应的 npm 包是 html-webpack-plugin。这里就以 HtmlWebpackPlugin 为例介绍一下 webpack plugin 的使用。

在前面 Hello world 的示例中，我们看到，因为逻辑均实现在 JavaScript 中，页面（index.html）的实现中基本没有逻辑，除了提供一个几乎为空的 HTML 结构外，引入了将要被构建生成的结果文件 bundle.js。一方面，bundle.js 是在 webpack.config.js 中配置的 output.filename 的值，在这里直接取固定值不方便后续维护；另一方面，为

了充分利用浏览器缓存，提高页面的加载速度，在生产环境中常常会向静态文件的文件名添加 MD5 戳，即使用 bundle_[hash].js 而不是 bundle.js，这里的[hash]会在构建时被该 chunk 内容的 MD5 结果替换，以实现内容不变则文件名不变，内容改变导致文件名改变。在这样的情况下，在 HTML 页面中给定结果文件的路径就变得不太现实。而 HtmlWebpackPlugin 正是为了解决这一问题而生，它会自动生成一个几乎为空的 HTML 页面，并向其中注入构建的结果文件路径，即使路径中包含动态的内容，如 MD5 戳，也能够完美处理。

了解了 HtmlWebpackPlugin 的能力，下面来将它引入到先前的项目中（完整代码见 chapter2/part4/）。

2．安装 plugin

前面介绍到，webpack 会内置一部分 plugin，想要使用这些 plugin，不需要额外安装，直接使用即可。

```
var webpack = require('webpack');
webpack.BannerPlugin;  //这样就可以直接获取 BannerPlugin
```

但是，这里介绍的 HtmlWebpackPlugin 并不是内置 plugin，它在 npm 包 html-webpack-plugin 中实现，因此，首先需要安装这个包（这里使用的是 1.7.0 版本，注意对不同版本的包 html-webpack-plugin，其用法与配置格式可能会不一致）。

```
npm i html-webpack-plugin@1.7.0 --save-dev
```

安装完成后，在 webpack.config.js 中就可以获取这个插件了。

```
var HtmlWebpackPlugin = require('html-webpack-plugin');
```

3．配置 plugin

接下来是让 webpack 使用 HtmlWebpackPlugin，并对其行为进行配置。plugin 相关配置对应 webpack 配置信息中的 plugins 字段，它的值要求是一个数组，数组的每一项为一个 plugin 实例。

```
var path = require('path');
var HtmlWebpackPlugin = require('html-webpack-plugin');
```

```
module.exports = {
   entry: path.join(__dirname, 'index'),
   output: {
         path: __dirname,
         filename: 'bundle.js'
   },
   module: {
      loaders: [
         {
            test: /\.css$/,
            loaders: ['style', 'css']
         }
      ]
   },
   plugins: [
      new HtmlWebpackPlugin({
         title: 'use plugin'
      })
   ]
};
```

我们看到，我们创造了一个 HtmlWebpackPlugin 实例，并将其添加进了配置信息的 plugins 字段。在实例化时，传入了{title:'use plugin'}，这是传递给 HtmlWebpackPlugin 的配置信息，它告诉 HtmlWebpackPlugin 给生成的 HTML 页面设置<title>的内容为 use plugin。这样，原来的 index.html 就可以删除了。在构建完成后，这个插件会自动在 output 目录（在这里即当前目录）下生成文件 index.html。

再次执行构建命令 webpack，便可以看到效果。

2.2.6 实时构建

与 RequireJS 的小文件开发方式相比，基于 browserify 与 webpack 的开发方式多出了构建的步骤。如果每一次小的改动都要手动执行一遍构建才能看到效果，开发会变得非常烦琐。监听文件改动并实时构建的能力成为新一代打包工具的标配。在 webpack 中，通过添加--watch 选项即可开启监视功能，webpack 会首先进行一次构建，然后依据构建得到的依赖关系，对项目所依赖的所有文件进行监听，一旦发生改动则触发重新构建。命令也可以简写成如下形式。

```
webpack -w
```

除了 watch 模式外，webpack 还提供了 webpack-dev-server 来辅助开发与调试。webpack-dev-server 是一个基于 Express 框架的 Node.js 服务器。它还提供了一个客户端的运行环境，会被注入到页面代码中执行，并通过 Socket.IO 与服务器通信。这样，服务器端的每次改动与重新构建都会被通知到页面上，页面可以随之做出反应。除了最基本的自动刷新，还提供有如模块热替换（Hot Module Replacement）这样强大的功能。

使用 webpack-dev-server 需要额外安装 webpack-dev-server 包。

```
npm install webpack-dev-server -g
```

然后启动 webpack-dev-server 即可。

```
webpack-dev-server
```

webpack-dev-server 默认会监听 8080 端口，因此直接在浏览器里打开 http://localhost:8080，即可看到结果页面。

对于 webpack-dev-server 的配置，既可以通过命令行参数的形式传递，也可以通过在 webpack.config.js 的 export 中添加字段 devServer 实现。详细的使用可以参考 webpack 的官方文档，这里就不做赘述了。