导读

本电子书为 webpack 实战手册，最早由网友 Array-Huang 发布于网络，由汇智网编目整理。

毫无疑问，对于开发者而言，看文档是不可避免的学习环节，但往往也最枯燥的，从文档开始使开发者的学习效率大打折扣。为了弥补这一遗憾，汇智网推出了适合初学者快速上手的在线互动式 Vue.js 工程化实践课程，读者可以通过以下链接访问在线教程：

http://xc.hubwiz.com/course/598bad66c7fd1d49453979c9?affid=webpack7878

教程预置了开发环境。进入教程后，可以在每一个知识点立刻进行同步实践，而不必在开发环境的搭建上浪费时间：

汇智网带来的是一种全新的交互式学习方式，可以极大提高学习编程的效率和学习效果：

汇智网课程内容已经覆盖以下的编程技术：

Node.js、MongoDB、JavaScript、C、C#、PHP、Python、Angularjs、Ionic、React、UML、redis、mySQL、Nginx、CSS、HTML、Flask、Gulp、Mocha、Git、Meteor、Canvas、zebra、Typescript、Material Design Lite、ECMAScript、Elasticsearch、Mongoose、jQuery、d3.js、django、cheerio、SVG、phoneGap、Bootstrap、jQueryMobile、Saas、YAML、Vue.js、webpack、Firebird，jQuery EasyUI，ruby，asp.net，c++，Express ，Spark.....

第一章：一套配置吃天下

1.1 webpack 配置常用部分有哪些？

1.1.1 前言

webpack 的配置文件是一个 node.js 的 module，用 CommonJS 风格来书写，形如：

```
module.exports = {
    entry: "./entry",
    output: {
        path: __dirname + "/dist",
        filename: "bundle.js"
    }
}
```

webpack 的配置文件并没有固定的命名，也没有固定的路径要求，如果你直接用 `webpack` 来执行编译，那么 webpack 默认读取的将是当前目录下的 `webpack.config.js`

```
$ pwd
/d/xampp/htdocs/webpack-seed
$ webpack # webpack 此时读取的实际上是
/d/xampp/htdocs/webpack-seed/webpack.config.js
```

如果你有其它命名的需要或是你有多份配置文件，可以使用`--config`参数传入路径：

```
$ webpack --config ./webpackConfig/dev.config.js
```

另外，在 CLI 执行 `webpack` 指令时可传入的参数（当然除了`--config`）实际上都可以在配置文件里面直接声明，我强烈建议可以的话尽量都在配置文件里写好，有需要的话写两份配置也好三份也好（反正配置文件间也是可以互相引用的，相同的部分就拆成一个 module 出来以供读取，最后拼成各种情况下需要的配置就好了）。

1.1.2 入口文件配置：entry 参数

entry 可以是字符串（单入口），可以是数组（多入口），但为了后续发展，请务必使用 object，因为 object 中的 key 在 webpack 里相当于此入口的 name，既可以后续用来拼生成文件的路径，也可以用来作为此入口的唯一标识。 我推荐的形式是这样的：

```
entry: { // pagesDir 是前面准备好的入口文件集合目录的路径
  'alert/index': path.resolve(pagesDir, `./alert/index/page`),
  'index/login': path.resolve(pagesDir, `./index/login/page`),
```

```
  'index/index': path.resolve(pagesDir, `./index/index/page`),
},
```

对照我的脚手架项目[webpack-seed][3]的文件目录结构，就很清楚了：

```
├─src # 当前项目的源码
    ├─pages # 各个页面独有的部分，如入口文件、只有该页面使用到的 css、模板文件等
    │  ├─alert # 业务模块
    │  │  └─index # 具体页面
    │  ├─index # 业务模块
    │  │  ├─index # 具体页面
    │  │  └─login # 具体页面
```

由于每一个入口文件都相当于 entry 里的一项，因此这样一项一项地来写实在是有点繁琐，我就稍微写了点代码来拼接这 entry：

```
var pageArr = [
  'index/login',
  'index/index',
  'alert/index',
];
var configEntry = {};
pageArr.forEach((page) => {
  configEntry[page] = path.resolve(pagesDir, page + '/page');
});
```

1.1.3 输出文件：output 参数

output 参数告诉 webpack 以什么方式来生成/输出文件，值得注意的是，与 entry 不同，output 相当于一套规则，所有的入口都必须使用这一套规则，不能针对某一个特定的入口来制定 output 规则。output 参数里有这几个子参数是比较常用的：path、publicPath、filename、chunkFilename，这里先给个[webpack-seed][4]中的示例：

```
    output: {
      path: buildDir, // var buildDir = path.resolve(__dirname, './build');
      publicPath: '../../../../build/',
      filename: '[name]/entry.js',    // [name]表示 entry 每一项中的 key，用以批量指定生成后文件的名称
      chunkFilename: '[id].bundle.js',
    },
```

1.1.3.1path

path 参数表示生成文件的根目录，需要传入一个**绝对路径**。path 参数和后面的 filename 参数共同组成入口文件的完整路径。

1.1.3.2publicPath

publicPath 参数表示的是一个 URL 路径（指向生成文件的根目录），用于生成 css/js/图片/字体文件等资源的路径，以确保网页能正确地加载到这些资源。publicPath 参数跟 path 参数的区别是：path 参数其实是针对本地文件系统的，而 publicPath 则针对的是浏览器；因此，publicPath 既可以是一个相对路径，如示例中的`'../../../../build/'`，也可以是一个绝对路径如`http://www.xxxxx.com/`。一般来说，我还是更推荐相对路径的写法，这样的话整体迁移起来非常方便。那什么时候用绝对路径呢？其实也很简单，当你的 html 文件跟其它资源放在不同的域名下的时候，就应该用绝对路径了，这种情况非常多见于后端渲染模板的场景。

1.1.3.3filename

filename 属性表示的是如何命名生成出来的入口文件，规则有以下三种：

- [name]，指代入口文件的 name，也就是上面提到的 entry 参数的 key，因此，我们可以在 name 里利用`/`，即可达到控制文件目录结构的效果。
- [hash]，指代本次编译的一个 hash 版本，值得注意的是，只要是在同一次编译过程中生成的文件，这个[hash]的值就是一样的；在缓存的层面来说，相当于一次全量的替换。
- [chunkhash]，指代的是当前 chunk 的一个 hash 版本，也就是说，在同一次编译中，每一个 chunk 的 hash 都是不一样的；而在两次编译中，如果某个 chunk 根本没有发生变化，那么该 chunk 的 hash 也就不会发生变化。这在缓存的层面上来说，就是把缓存的粒度精细到具体某个 chunk，只要 chunk 不变，该 chunk 的浏览器缓存就可以继续使用。

下面来说说如何利用 filename 参数和 path 参数来设计入口文件的目录结构，如示例中的`path: buildDir, // var buildDir = path.resolve(__dirname, './build');`和`filename: '[name]/entry.js'`，那么对于 key 为'index/login'的入口文件，生成出来的路径就是`build/index/login/entry.js`了，怎么样，是不是很简单呢？

1.1.3.4chunkFilename

chunkFilename 参数与 filename 参数类似，都是用来定义生成文件的命名方式的，只不过，chunkFilename 参数指定的是除入口文件外的 chunk（这些 chunk 通常是由于 webpack 对代码的优化所形成的，比如因应实际运行的情况来异步加载）的命名。

1.1.4 各种 Loader 配置：module 参数

webpack 的核心实际上也只能针对 js 进行打包，那 webpack 一直号称能够打包任何资源是怎么一回事呢？原来，webpack 拥有一个类似于插件的机制，名为 `Loader`，通过 Loader，webpack 能够针对每一种特定的资源做出相应的处理。Loader 的种类相当多，有些比较基础的是官方自己开发，而其它则是由 webpack 社区开源贡献出来的，这里是 Loader 的 List：[list of loaders][5]。 而 module 正是配置什么资源使用哪个 Loader 的参数（因为就算是同一种资源，也可能有不同的 Loader 可以使用，当然不同 Loader 处理的手段不一样，最后结果也自然就不一样了）。module 参数有几个子参数，但是最常用的自然还是 `loaders` 子参数，这里也仅对 loaders 子参数进行介绍。

1.1.4.1loaders 参数

loaders 参数又有几个子参数，先给出一个官方示例：

```
module.loaders: [
  {
    // "test" is commonly used to match the file extension
    test: /\.jsx$/,

    // "include" is commonly used to match the directories
    include: [
      path.resolve(__dirname, "app/src"),
      path.resolve(__dirname, "app/test")
    ],

    // "exclude" should be used to exclude exceptions
    // try to prefer "include" when possible

    // the "loader"
    loader: "babel-loader"
  }
]
```

下面一一对这些子参数进行说明：

- `test` 参数用来指示当前配置项针对哪些资源，该值应是一个条件值(condition)。
- `exclude` 参数用来剔除掉需要忽略的资源，该值应是一个条件值(condition)。
- `include` 参数用来表示本 loader 配置仅针对哪些目录/文件，该值应是一个条件值(condition)。这个参数跟 `test` 参数的效果是一样的（官方文档也是这么写的），我也不明白为嘛有俩同样规则的参数，不过我们姑且可以自己来划分这两者的用途：`test` 参数用来指示文件名（包括文件后缀），而 `include` 参数则用来指示目录；注意同时使用这两者的时候，实际上是 `and` 的关系。

- `loader`/`loaders` 参数，用来指示用哪个/哪些 loader 来处理目标资源，这俩货表达的其实是一个意思，只是写法不一样，我个人推荐用 `loader` 写成一行，多个 loader 间使用`!`分割，这种形式类似于管道的概念，又或者说是函数式编程。形如 `loader: 'css?!postcss!less'`，可以很明显地看出，目标资源先经 less-loader 处理过后将结果交给 postcss-loader 作进一步处理，然后最后再交给 css-loader。

  条件值(condition)可以是一个字符串（某个资源的文件系统绝对路径），可以是一个函数（官方文档里是有这么写，但既没有示例也没有说明，我也是醉了），可以是一个正则表达式（用来匹配资源的路径，最常用，强烈推荐！），最后，还可以是一个数组，数组的元素可以为上述三种类型，元素之间为与关系（既必须同时满足数组里的所有条件）。需要注意的是，loader 是可以接受参数的，方式类似于 URL 参数，形如'css?minimize&-autoprefixer'，具体每个 loader 接受什么参数请参考 loader 本身的文档（一般也就只能在 github 里看了）。

1.1.5 添加额外功能：plugins 参数

这 plugins 参数相当于一个插槽位（类型是数组），你可以先按某个 plugin 要求的方式初始化好了以后，把初始化后的实例丢到这里来。

1.2 听说 webpack 连 less/css 也能打包？

1.2.1 前言

过去讲前端模块化、组件化，更多还是停留在 js 层面，毕竟 js 作为一种更典型的程序语言，在这方面的想象和操作空间都更大一些。但近年来，组件化要求得更多了，HTML/CSS/JS 这三件套一件可都不能少（甚至包括其它类型的资源，比如说图片），而这样的组件，无疑是高内聚的。

1.2.2 文章简介

本文将介绍如何使用 webpack 来打包 less/css（没用过 sass，但毕竟也是通过 loader 来加载的，相信与 less 无异），首先是介绍相关的 webpack plugin&loader，然后将介绍如何加载不同应用层次的 less/css。

1.2.3 用到什么 loader 了？

在[《webpack 多页应用架构系列（二）：webpack 配置常用部分有哪些？》][2]里我就说过，webpack 的核心只能打包 js 文件，而 js 以外的资源都是靠 loader 进行转换或做出相应的处理的。下面我就来介绍打包 less/css 所需要的 loader。

1.2.3.1less-loader

针对 less 文件，我们首先需要使用 less-loader 来加载。less-loader 会调用所依赖的 `less` 模块对 less 文件进行编译(包括`@import` 语法)。至于说 less-loader 所接受的参数，实质上大部分是传递给 `less` 模块使用的参数，由于我本人应用 less 的程度不深，因此没有传任何参数、直接就使用了。如果你之前对 `less` 模块就已经有了一套配置的话，请参考[less-loader 的文档][3]进行配置。
另外，less-loader 并不会针对 `url()`语法做特别的转换，因此，如果你想把 `url()` 语句里涉及到的文件（比如图片、字体文件等）也一并用 webpack 打包的话，就必须利用管道交给 css-loader 做进一步的处理。

1.2.3.2css-loader

针对 css 文件，我们需要使用 css-loader 来加载.css-loader 的功能比较强大，一些新颖的特性比如 `Local Scope` 或是 `CSS Modules` 都是支持的。
我目前只用到了 css-loader 的[压缩功能(Minification)][4]，对于这个功能，有一点是需要注意的，那就是如果你的代码里也和我一样，有许多为了浏览器兼容性的**废弃** CSS 代码的话，请务必关闭 `autoprefixer` 已避免你的**废弃** CSS 代码被 css-loader 删除了，形如 `css?minimize&-autoprefixer`。
上面提到 css-loader 会对 `url()`语句做处理，这里稍微再说两句。在 less/css 里的这 `url()`语句，在 css-loader 看来，就跟 `require()`语句是一样的，只要在 webpack 配置文件里定义好加载各类型资源的 loader，那这 `url()`语句实际上什么资源都能处理。一般我在 `url()`语句都会以相对路径的方式（相对于此语句所在的 less/css 文件）来指定文件路径；请不要使用以`/`开头（即相对于网站根目录，因为对于文件系统来说，这明显是令人混淆的）的路径，尽管 css-loader 也可以通过设置 `root` 参数来适配。

1.2.3.3postcss-loader

习惯用 postcss 的童鞋们有福啦，webpack 可以通过 postcss-loader 来兼容 postcss。由于 postcss 只算是一个加分项，因此这里也不作过多介绍，只介绍一下如何把 postcss 搭进 webpack，不明白的童鞋麻烦先把 postcss 搞懂了再看。

放上我的脚手架项目的代码：

```
var precss       = require('precss');
var autoprefixer = require('autoprefixer');

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

```
            {
                test:   /\.css$/,
                exclude: /node_modules|bootstrap/,
                loader: 'css?minimize&-autoprefixer!postcss',
            }
        ]
    },
    postcss: function () {
        return [precss, autoprefixer({
            remove: false,
            browsers: ['ie >= 8', '> 1% in CN'],
        })];
    }
}
```

从 loader 的配置`'css?minimize&-autoprefixer!postcss'`上看，实际上就是先让 postcss-loader 处理完了再传递给 css-loader。而 `postcss` 项则是 postcss-loader 所接受的参数，实际上就是返回一个包含你所需要的 postcss's plugins 的数组啦，这些 plugin 有各自的初始化参数，不过这些都是 postcss 的内容了，这里就不做介绍了。

1.2.4 用到什么 Plugin 了？

加载 less/css 这一块主要用到的是 `extract-text-webpack-plugin`（下文简称为 `ExtractTextPlugin` 吧），而且由于我用的是 `webpack 1`，因此用的也是相对应 `webpack 1` 的版本（[1 的文档在这里不要搞错了哈][5]）。
ExtractTextPlugin 的作用是把各个 chunk 加载的 css 代码（可能是由 less-loader 转换过来的）合并成一个 css 文件并在页面加载的时候以`<link>`的形式进行加载。
相对于使用 style-loader 直接把 css 代码段跟 js 打包在一起并在页面加载时以 inline 的形式插入 DOM，我还是更喜欢 ExtractTextPlugin 生成并加载 CSS 文件的形式；倒不是看不惯 inline 的 css，只是用文件形式来加载的话会快很多，尤其后面介绍用 webpack 来生成 HTML 的时候，这`<link>`会直接生成在`<head>`里，那么在 CSS 的加载上就跟传统的前端页面没有差别了，体验非常棒。
ExtractTextPlugin 的初始化参数不多，唯一的必填项是 `filename` 参数，也就是如何来命名生成的 CSS 文件。跟 webpack 配置里的 output.filename 参数类似，这 ExtractTextPlugin 的 filename 参数也允许使用变量，包括[id]、[name]和[contenthash]；理论上来说如果只有一个 chunk，那么不用这些变量，写死一个文件名也是可以的，但由于我们要做的是多页应用，必然存在多个 chunk（至少每个 entry 都对应一个 chunk 啦）。这里我是这么设置的：

```
new ExtractTextPlugin('[name]/styles.css'), [name]对应的是 chunk 的 name，我在 webpack 配置中是这样
```

[name]对应的是 chunk 的 name，我在 webpack 配置中把各个 entry 的 name 都按 `index/index`、`index/login` 这样的形式来设置了，那么最后 css 的路径就

会像这样：`build/index/index/styles.css`，也就是跟 chunk 的 js 文件放一块了（js 文件的路径形如 `build/index/index/entry.js`）。
除了要把这初始化后的 ExtractTextPlugin 放到 webpack 配置中的 `plugins` 参数里，我们还要在 loader 配置里做相应的修改：

```
module.exports = {
    module: {
        loaders: [
            {
                test:   /\.css$/,
                exclude: /node_modules|bootstrap/,
                loader:
ExtractTextPlugin.extract('css?minimize&-autoprefixer!postcss'),
            }
        ]
    },
}
```

如此一来，ExtractTextPlugin 就算是配置好了。

1.2.5 如何加载不同应用层次的 less/css

在我的设计中，有三种应用层次的 less/css 代码段：

- 基础的、公用的代码段，包括 CSS 框架、在 CSS 框架上进行定制的 CSS theme，基本上每个页面都会应用到这些 CSS 代码段。
- 组件的代码段，这里的组件指的是你自己写的组件，而且组件本身含有 js，并负责加载 css 以及其它逻辑。
- 每个页面独有的 CSS 代码段，很可能只是对某些细节进行微调。

首先来回顾一下我设计的文件目录结构：

```
├─src # 当前项目的源码
    ├─pages # 各个页面独有的部分，如入口文件、只有该页面使用到的 css、模板文件等
    │  ├─alert # 业务模块
    │  │  └─index # 具体页面
    │  ├─index # 业务模块
    │  │  ├─index # 具体页面
    │  │  └─login # 具体页面
    │  │      └─templates # 如果一个页面的 HTML 比较复杂，可以分成多块再拼在一起
    │  └─user # 业务模块
    │      ├─edit-password # 具体页面
    │      └─modify-info # 具体页面
    └─public-resource # 各个页面使用到的公共资源
        ├─components # 组件，可以是纯 HTML，也可以包含 js/css/image 等，看自己
需要
```

```
        │  ├─footer # 页尾
        │  ├─header # 页头
        │  ├─side-menu # 侧边栏
        │  └─top-nav # 顶部菜单
        ├─config # 各种配置文件
        ├─iconfont # iconfont 的字体文件
        ├─imgs # 公用的图片资源
        ├─layout # UI 布局，组织各个组件拼起来，因应需要可以有不同的布局套路
        │  ├─layout # 具体的布局套路
        │  └─layout-without-nav # 具体的布局套路
        ├─less # less 文件，用 sass 的也可以，又或者是纯 css
        │  ├─base-dir
        │  ├─components-dir # 如果组件本身不需要 js 的，那么要加载组件的 css 比较
困难，我建议可以直接用 less 来加载
        │  └─base.less # 组织所有的 less 文件
        ├─libs # 与业务逻辑无关的库都可以放到这里
        └─logic # 业务逻辑
```

1.2.5.1 基础代码段

基础的 CSS 代码（实际上我的项目中用的都是 less）我统一都放到 `src/public-resource/less` 目录里。我使用一个抽象的文件 `base.less` 将所有的 less 文件组织起来（利用`@import`），这样的话我用 js 加载起来就方便多了。在我的脚手架项目（[Array-Huang/webpack-seed][6]）里，CSS 框架我用的是 bootstrap，并且使用了 bootstrap-loader 进行加载，因此就没有把 bootstrap 的 CSS 文件放到 `src/public-resource/less/base-dir` 目录里，这个目录里放的都是我定制的 theme 了。
`src/public-resource/less/components-dir` 目录放的是某些第三方组件所用到的 css，又或是不含 js 的组件所用到的 css。其实这部分 CSS 是否应该归在下一类，我也考虑良久，只是由于归到下一类的话加载起来不方便，不方便原因如下：

- 某些第三方库是要你自己加载 CSS 的，如果你打算写适配器来封装这些第三方库，那自然可以直接在适配器来加载 CSS，这就属于下一类了；然而，有一些使用起来很简单的库，你还写适配器那就有点画蛇添足了。
- 某些自己写的组件可能仅包含 HTML 和 CSS，那么谁来加载 CSS？

所以干脆还是交由 `base.less` 一并加载了算了。
我设计了一个 `common.page.js`，并在每一个页面的入口文件里都首先加载这 `common.page.js`，那么，只要我在这 `common.page.js` 里加载 `base.less`，所有的页面都能享受到这份基础 CSS 代码段。

1.2.5.2 组件代码段

组件的代码我都放在了 `src/public-resource/components`，每一个组件统一放在一个独立的目录，并由该组件的 js 负责加载其 CSS。

1.2.5.3 页面代码段

页面独有的 CSS 我自然是放在该页面自己的目录里，利用该页面的入口文件进行加载。

1.2.5.4 最终生成的 CSS 代码都在哪？

由于我使用了 ExtractTextPlugin，因此这些 CSS 代码最终都会生成到所属 chunk 的目录里成为一个 CSS 文件。

- 基础代码段肯定是保存在 CommonsChunkPlugin 所生成的公共代码 chunk 所在的目录里了，在我的脚手架项目（Array-Huang/webpack-seed）里就是 `build/commons` 了（我的公共代码 chunk 的 name 是'commons'）。
- 组件代码段看情况，该组件用的页面多的话（大于 CommonsChunkPlugin 的 minChunks 参数）就会被归到跟基础代码段一起咯；反之，则哪个页面用到它，就放到哪个页面 chunk 的目录里咯。
- 页面代码段就不用想了，肯定是在那个页面 chunk 的目录里了，毕竟才用了 1 次。

1.3 听说 webpack 连图片和字体也能打包？

1.3.1 前言

上一篇《听说 webpack 连 less/css 也能打包？》说到使用 loader 来加载 CSS，这一篇来讲讲如何**笼统**地加载其它类型的资源。为什么强调是**“笼统”**呢？这是因为本文介绍的方法并不针对任何类型的资源，这意味着，什么类型的资源都能用，但效果也都只是有限的。

1.3.2 用的什么 loader 呢？

本文介绍俩 loader：file-loader 和 url-loader。

1.3.2.1file-loader

file-loader 的主要功能是：把源文件迁移到指定的目录（可以简单理解为从源文件目录迁移到 `build` 目录），并返回新文件的路径（简单拼接而成）。
file-loader 需要传入 `name` 参数，该参数接受以下变量（以下讨论的前提是：源文件 `src/public-resource/imgs/login-bg.jpg`;在根目录内执行 `webpack` 命令，也就是当前的上下文环境与 `src` 目录同级）：

- [ext]：文件的后缀名，示例为'jpg'。
- [name]：文件名本身，示例为'login-bg'。
- [path]：相对于当前执行 webpack 命令的目录的相对路径（不含文件名本身），示例为'src/public-resource/imgs/'。这个参数我感觉用处不大，除非你想把迁移后的文件放回源文件的目录或其子目录里。
- [hash]：源文件内容的 hash，用于**缓存解决方案**。

我的做法是，
`require('!file-loader?name=static/images/[name].[ext]!../imgs/login-bg.jpg')`，这样 `login-bg.jpg` 的路径就变成 `static/images/login-bg.jpg` 了，注意这还不是完整的路径，最终还是要拼上 webpack 配置中的 `output.publicPath` 参数的；比如说我的 `output.publicPath` 参数是`../../../../build/`，那么最终从 `require()`里获得的完整路径就会是`../../../../build/static/images/login-bg.jpg` 了。

1.3.2.2url-loader

url-loader 的主要功能是：将源文件转换成 DataUrl(声明文件 mimetype 的 base64 编码)。据我所知，在前端范畴里，图片和字体文件的 DataUrl 都是可以被浏览器所识别的，因此可以把图片和字体都转化成 DataUrl 收纳在 HTML/CSS/JS 文件里，以减少 HTTP 连接数。

url-loader 主要接受以下参数：

- limit 参数，数据类型为整型，表示*目标文件的体积*大于多少**字节**就换用 file-loader 来处理了，不填则永远不会交给 file-loader 处理。例如 `require("url?limit=10000!./file.png");`，表示如果目标文件大于 10000 字节，就交给 file-loader 处理了。
- mimetype 参数，前面说了，DataUrl 是需要声明文件的 mimetype 的，因此我们可以通过这个参数来强行设置 mimetype，不填写的话则默认从目标文件的后缀名进行判断。例如 `require("url?mimetype=image/png!./file.jpg");`，强行把 jpg 当 png 使哈。
- 一切 file-loader 的参数，这些参数会在启用 file-loader 时传参给 file-loader，比如最重要的 name 参数。

1.3.3 实操演示

接下来还是用我的脚手架项目[webpack-seed][3]来介绍如何利用 url-loader 和 file-loader 来加载各类资源。

1.3.3.1 图片

这一块我是直接在 webpack 配置文件里设置的：

```
        {
          // 图片加载器，雷同 file-loader，更适合图片，可以将较小的图片转成 base64，减少 http 请求
          // 如下配置，将小于 8192byte 的图片转成 base64 码
          test: /\.(png|jpg|gif)$/,
          loader: 'url-loader?limit=8192&name=./static/img/[hash].[ext]',
        },
```

由于使用了[hash]，因此即便是不同页面引用了相同名字但实际内容不同的图片，也不会造成“覆盖”的情况出现；进一步讲，如果不同页面引用了在不同位置但实际内容相同的图片，这还可以归并成一张图片，方便浏览器缓存呢。

1.3.3.2 字体文件

这一块我也还是直接在 webpack 配置里配置的：

```
        {
          // 专供 iconfont 方案使用的，后面会带一串时间戳，需要特别匹配到
          test: /\.(woff|woff2|svg|eot|ttf)\??.*$/,
          loader: 'file?name=./static/fonts/[name].[ext]',
        },
```

需要声明的是，由于我使用的是阿里妈妈的 iconfont 方案，此方案加载字体文件的方式有一点点特殊，所以正则匹配的时候要注意一点，iconfont 的 CSS 是这样的，你们看看就明白了：

```
@font-face {font-family: "iconfont";
  src: url('iconfont.eot?t=1473142795'); /* IE9*/
  src: url('iconfont.eot?t=1473142795#iefix') format('embedded-opentype'), /* IE6-IE8 */
  url('iconfont.woff?t=1473142795') format('woff'), /* chrome, firefox */
  url('iconfont.ttf?t=1473142795') format('truetype'), /* chrome, firefox, opera, Safari, Android, iOS 4.2+*/
  url('iconfont.svg?t=1473142795#iconfont') format('svg'); /* iOS 4.1- */
}
```

1.3.3.3 其它资源

也许你会问，我们为什么还需要转移其它资源呢？直接引用不就可以了吗？

我之前也是这么做的，直接引用源文件目录 `src` 里的资源，比如说 `webuploader` 用到的 swf 文件，比如说用来兼容 IE 而又不需要打包的 js 文件。但是后来我发现，这样做的话，就导致部署上线的时候要把 `build` 目录和 `src` 目录同时放上去了；而且由于 `build` 目录和 `src` 目录同级，我就只能用 `build` 目录和 `src` 目录的上一级目录作为网站的根目录了（因为如果把 `build` 目录设为网站，用户就读取不到 `src` 目录了），反正就是各种的不方便。
那么，我是怎么做的呢？

我建了一个 config 文件，名为 `build-file.config.js`，内容如下：

```
module.exports = {
  js: {
    xdomain:
require('!file-loader?name=static/js/[name].[ext]!../../../vendor/ie-fix/
xdomain.all.js'),
    html5shiv:
require('!file-loader?name=static/js/[name].[ext]!../../../vendor/ie-fix/
html5shiv.min.js'),
    respond:
require('!file-loader?name=static/js/[name].[ext]!../../../vendor/ie-fix/
respond.min.js'),
  },
  images: {
    'login-bg':
require('!file-loader?name=static/images/[name].[ext]!../imgs/login-bg.jp
g'),
  },
};
```

这个 config 文件起到两个作用：

1. 每次加载到这个 config 文件的时候，会执行那些 `require()`语句，对目标文件进行转移（从 `src` 目录到 `build` 目录）。
2. 调用目标文件的代码段，可以从这个 config 文件取出目标文件转移后的完整路径，例如我在 `src/public-resource/components/header/html.ejs` 里是这么用的：

```
<!DOCTYPE html>
<html lang="zh-cmn-Hans">
<head>
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <title><% if (pageTitle) { %> <%= pageTitle %> - <% } %> XXXX 后台</title>
```

```
  <meta name="viewport"
content="width=device-width,initial-scale=1,maximum-scale=1" />
  <meta name="renderer" content="webkit" />

  <!--[if lt IE 10]>
    <script src="<%= BUILD_FILE.js.xdomain %>" slave="<%=
SERVER_API_URL %>cors-proxy.html"></script>
    <script src="<%= BUILD_FILE.js.html5shiv %>"></script>
  <![endif]-->
</head>
<body>
  <!--[if lt IE 9]>
    <script src="<%= BUILD_FILE.js.respond %>"></script>
  <![endif]-->
```

恩，你可能会好奇这 HTML 里怎么能直接引用 js 的值，哈哈哈，超纲了超纲了，这是我后面要讲到的内容了。

+

第二章：实战 webpack

2.1 怎么打包公共代码才能避免重复？

2.1.1 前言

与单页应用相比，多页应用存在多个入口（每个页面即一个入口），每一个入口（页面）都意味着一套完整的 js 代码（包括业务逻辑和加载的第三方库/框架等）。 在文章《webpack 配置常用部分有哪些？》中，我介绍了如何配置多页应用的入口(entry)，然而，如果仅仅如此操作，带来的后果就是，打包生成出来的每一个入口文件都会完整包含所有代码。 你也许会说："咱们以前写页面不也是每个页面都会加载所有的代码吗？浏览器会缓存，没事的啦"。其实问题在于，以前写代码都是单个单个 js 来加载的，一个页面加载下来的确所有页面都能共享到缓存；而到了 webpack 这场景，由于对于每一个页面来说，所有的 js 代码都打包成唯一一个 js 文件了，而浏览器是无法分辨出该文件内的公共代码并加以缓存的，所以，浏览器就没办法实现公共代码在页面间的缓存了（当前页面的缓存还是 OK 的，也就是说刷新不需要重新加载）。

2.1.2 想智能判断并打包公共代码？

CommonsChunkPlugin 的效果是：在你的多个页面（入口）所引用的代码中，找出其中满足条件（被多少个页面引用过）的代码段，判定为公共代码并打包

成一个独立的 js 文件。至此，你只需要在每个页面都加载这个公共代码的 js 文件，就可以既保持代码的完整性，又不会重复下载公共代码了（多个页面间会共享此文件的缓存）。

2.1.2.1 再提一下使用 Plugin 的方法

大部分 Plugin 的使用方法都有一个固定的套路：

1. 利用 Plugin 的初始方法并传入 Plugin 预设的参数进行初始化，生成一个实例。
2. 将此实例插入到 webpack 配置文件中的 `plugins` 参数（数组类型）里即可。

2.1.2.2CommonsChunkPlugin 的初始化常用参数有哪些？

- `name`，给这个包含公共代码的 chunk 命个名（唯一标识）。
- `filename`，如何命名打包后生产的 js 文件，也是可以用上`[name]`、`[hash]`、`[chunkhash]`这些变量的啦（具体是什么意思，请看我上一篇文章中关于 filename 的那一节）。
- `minChunks`，公共代码的判断标准：某个 js 模块被多少个 chunk 加载了才算是公共代码。
- `chunks`，表示需要在哪些 chunk（也可以理解为 webpack 配置中 entry 的每一项）里寻找公共代码进行打包。不设置此参数则默认提取范围为所有的 chunk。

2.1.2.3 实例分析

实例来自于我的脚手架项目 `webpack-seed`，我是这样初始化一个 CommonsChunkPlugin 的实例：

```
  var commonsChunkPlugin = new webpack.optimize.CommonsChunkPlugin({
    name: 'commons', // 这公共代码的 chunk 名为'commons'
    filename: '[name].bundle.js', // 生成后的文件名，虽说用了[name]，但实际上就是'commons.bundle.js'了
    minChunks: 4, // 设定要有 4 个 chunk（即 4 个页面）加载的 js 模块才会被纳入公共代码。这数目自己考虑吧，我认为 3-5 比较合适。
  });
```

最终生成文件的路径是根据 webpack 配置中的 ouput.path 和上面 CommonsChunkPlugin 的 filename 参数来拼的，因此想控制目录结构的，直接在 filename 参数里动手脚即可，例如：`filename: 'commons/[name].bundle.js'`

2.1.3 总结

整体来说，这套方案还是相当简单的，而从效果上说，也算是比较均衡的，比较适合项目初期使用。

2.2 老式 jQuery 插件还不能丢，怎么兼容？

2.2.1 前言

目前前端虽处于百花齐放阶段，angular/react/vue 竞相角逐，但毕竟尚未完全成熟，有些需求还是得依靠我们的老大哥 jQuery 的。

我个人对 jQuery 并不反感，但我对 jQuery 生态的停滞不前相当无奈，比如说赫赫有名的 bootstrap（特指 3 代），在 webpack 上打包还得靠个 loader 的，太跟不上时势了。况且，bootstrap 还算好的，有些 jquery 插件都有一两年没更新了，连 NPM 都没上架呢，可偏偏就是找不到它们的替代品，项目又急着要上，这可咋办呐？

别急，今天就教你适配兼容老式 jQuery 插件。

2.2.2 老式 jQuery 插件为和不能直接用 webpack 打包？

如果你把 jQuery 看做是一个普通的 js 模块来加载（要用到 jQuery 的模块统统先 require 后再使用），那么，当你加载老式 jQuery 插件时，往往会提示找不到 jQuery 实例（有时候是提示找不到`$`），这是为啥呢？
要解释这个问题，就必须先稍微解释一下 jQuery 插件的机制：jQuery 插件是通过 jQuery 提供的 `jQuery.fn.extend(object)`和 `jQuery.extend(object)`这俩方法，来把插件本身实现的方法挂载到 `jQuery`（也即`$`）这个对象上的。传统引用 jQuery 及其插件的方式是先用`<script>`加载 jQuery 本身，然后再用同样的方法来加载其插件；jQuery 会把 `jQuery` 对象设置为全局变量（当然也包括了`$`），既然是全局变量，那么插件们很容易就能找到 `jQuery` 对象并挂载自身的方法了。而 webpack 作为一个遵从模块化原则的构建工具，自然是要把各模块的上下文环境给分隔开以减少相互间的影响；而 jQuery 也早已适配了 AMD/CMD 等加载方式，换句话说，我们在 require `jQuery` 的时候，实际上并不会把 `jQuery` 对象设置为全局变量。说到这里，问题也很明显了，jQuery 插件们找不到 `jQuery` 对象了，因为在它们各自的上下文环境里，既没有局部变量 `jQuery`（因为没有适配 AMD/CMD，所以就没有相应的 require 语句了），也没有全局变量 `jQuery`。

2.2.3 怎么来兼容老式 jQuery 插件呢？

方法有不少，下面一个一个来看。

`2.2.3.1ProvidePlugin` + `expose-loader`

首先来介绍我最为推荐的方法：`ProvidePlugin` + `expose-loader`，在我公司的项目，以及我个人的脚手架开源项目 `webpack-seed` 里使用的都是这一种方法。

ProvidePlugin 的配置是这样的：

```
  var providePlugin = new webpack.ProvidePlugin({
    $: 'jquery',
    jQuery: 'jquery',
    'window.jQuery': 'jquery',
    'window.$': 'jquery',
  });
```

ProvidePlugin 的机制是：当 webpack 加载到某个 js 模块里，出现了未定义且名称符合(字符串完全匹配)配置中 key 的变量时，会自动 require 配置中 value 所指定的 js 模块。

如上述例子，当某个老式插件使用了 `jQuery.fn.extend(object)`，那么 webpack 就会自动引入 `jquery`(此处我是用 NPM 的版本，我也推荐使用 NPM 的版本)。另外，使用 ProvidePlugin 还有个好处，就是，你自己写的代码里，再！也！不！用！require！jQuery！啦！毕竟少写一句是一句嘛哈哈哈。

接下来介绍 expose-loader，这个 loader 的作用是，将指定 js 模块 export 的变量声明为全局变量。下面来看下 expose-loader 的配置：

```
/*
    很明显这是一个 loader 的配置项，篇幅有限也只能截取相关部分了
    看不明白的麻烦去看本系列的另一篇文章《webpack 多页应用架构系列（二）：webpack 配置常用部分有哪些？》：https://segmentfault.com/a/1190000006863968
 */
{
  test: require.resolve('jquery'),  // 此 loader 配置项的目标是 NPM 中的 jquery
  loader: 'expose?$!expose?jQuery', // 先把 jQuery 对象声明成为全局变量 `jQuery`，再通过管道进一步又声明成为全局变量`$`
},
```

你或许会问，有了 ProvidePlugin 为嘛还需要 expose-loader？问得好，如果你所有的 jQuery 插件都是用 webpack 来加载的话，的确用 ProvidePlugin 就足够了；但理想是丰满的，现实却是骨感的，总有那么些需求是只能用`<script>`来加载的。

2.2.3.2externals

externals 是 webpack 配置中的一项，用来将某个全局变量**“伪装”**成某个 js 模块的 exports，如下面这个配置：

```
    externals: {
      'jquery': 'window.jQuery',
    },
```

那么，当某个 js 模块显式地调用 `var $ = require('jquery')`的时候，就会把 `window.jQuery` 返回给它。

与上述 `ProvidePlugin + expose-loader` 的方案相反，此方案是先用`<script>`加载的 jQuery 满足老式 jQuery 插件的需要，再通过 externals 将其转换成符合模块化要求的 exports。
我个人并不太看好这种做法，毕竟这就意味着 jQuery 脱离 NPM 的管理了，不过某些童鞋有其它的考虑，例如为了加快每次打包的时间而把 jQuery 这些比较大的第三方库给分离出去（直接调用公共 CDN 的第三方库？），也算是有一定的价值。

2.2.3.3imports-loader

这个方案就相当于手动版的 ProvidePlugin，以前我用 requireJS 的时候也是用的类似的手段，所以我一开始从 requireJS 迁移到 webpack 的时候用的也是这种方法，后来知道有 ProvidePlugin 就马上换了哈。

这里就不详细说明了，放个例子大家看看就懂：

```
// ./webpack.config.js

module.exports = {
    ...
    module: {
        loaders: [
            {
                test: require.resolve("some-module"),
                loader: "imports?$=jquery&jQuery=jquery", // 相当于`var $ =
require("jquery");var jQuery = require("jquery");`
            }
        ]
    }
};
```

2.2.4 总结

以上的方案其实都属于 shimming，并不特别针对 jQuery，请举一反三使用。另外，上述方案并不仅用于 shimming，比如用上 `ProvidePlugin` 来写少几个 require，自己多多挖掘，很有乐趣的哈~~

2.2.5 补充：误用 externals（2016-10-17 更新）

有童鞋私信我，说用了我文章的方案依然提示`$ is not a function`，在我仔细分析后，发现：

1. 他用的是我推荐的 `ProvidePlugin` + `expose-loader` 方案，也就是说，他已经把 jquery 打包进来了。
2. 但是他又不明就里得配了 externals：

```
externals: {
  jquery: 'window.jQuery',
},
```

1. 然而实际上他并没有直接用`<script>`来引用 jQuery，因此 window.jQuery 是个 null。
2. 结果，他的 jquery 插件获得的`$`就是个 null 了。

这里面我们可以看出，externals 是会覆盖掉 `ProvidePlugin` 的。
但这里有个问题，`expose-loader` 的作用就是设置好 window.jQuery 和 window.$，那 window.jQuery 怎么会是 null 呢？我的猜想是：externals 在 `expose-loader` 设置好 `window.jQuery` 前就已经取了 `window.jQuery` 的值(`null`)了。
说了这么多，其实关键意思就是，不要手贱不要手贱不要手贱（重要的事情说三遍）！

2.3 开发环境、生产环境傻傻分不清楚？

2.3.1 前言

开发环境与生产环境分离的原因如下：

- 在开发时，不可避免会产生大量 debug 又或是测试的代码，这些代码不应出现在生产环境中（也即不应提供给用户）。
- 在把页面部署到服务器时，为了追求极致的技术指标，我们会对代码进行各种各样的优化，比如说混淆、压缩，这些手段往往会彻底破坏代码本身的可读性，不利于我们进行 debug 等工作。
- 数据源的差异化，比如说在本地开发时，读取的往往是本地 mock 出来的数据，而正式上线后读取的自然是 API 提供的数据了。

如果硬是要在开发环境和生产环境用完全一样的代码，那么必然会付出沉重的代价，这点想必也不用多说了。

下面主要针对两点来介绍如何分离开发环境和生产环境：一是如何以不同的方式进行编译，也即如何分别形成开发环境及生产环境的 webpack 配置文件；二是在业务代码中如何根据环境的不同而做出不同的处理。

2.3.2 如何分离开发环境和生产环境的 webpack 配置文件

如果同时把一份完整的开发环境配置文件和一份完整的生产环境配置文件列在一起进行比较，那么会出现以下三种情况：

- 开发环境有的配置，生产环境不一定有，比如说开发时需要生成 sourcemap 来帮助 debug，又或是热更新时使用到的 `HotModuleReplacementPlugin`。
- 生产环境有的配置，开发环境不一定有，比如说用来混淆压缩 js 用的 `UglifyJsPlugin`。
- 开发环境和生产环境都拥有的配置，但在细节上有所不同，比如说 `output.publicPath`，又比如说 `css-loader` 中的 `minimize` 和 `autoprefixer` 参数。

更重要的是，实际上开发环境和生产环境的配置文件的绝大部分都是一致的，对于这一致的部分来说，我们坚决要消除冗余，否则后续维护起来不仅麻烦，而且还容易出错。

2.3.2.1 怎么做呢？

答案很简单：分拆 webpack 配置文件成 N 个小 module。原先我们是一个完整的配置文件，有好几百行，从头看到尾都头大了，更别说分离不分离的了。下面来看看我分离的结果：

```
├─webpack.dev.config.js # 开发环境的 webpack 配置文件(无实质内容,仅为组织整理)
├─webpack.config.js # 生产环境的 webpack 配置文件（无实质内容，仅为组织整理）
├─webpack-config # 存放分拆后的 webpack 配置文件
  ├─entry.config.js # webpack 配置中的各个大项，这一级目录里的文件都是
  ├─module.config.js
  ├─output.config.js
  ├─plugins.dev.config.js # 俩环境配置中不一致的部分，此文件由开发环境配置文件 webpack.dev.config.js 来加载
  ├─plugins.product.config.js # 俩环境配置中不一致的部分，此文件由生产环境配置文件 webpack.config.js 来加载
  ├─resolve.config.js
  │
  ├─base # 主要是存放一些变量
  │   ├─dir-vars.config.js
  │   ├─page-entries.config.js
  │
  ├─inherit # 存放生产环境和开发环境相同的部分，以供继承
  │   ├─plugins.config.js
  │
  └─vendor # 存放 webpack 兼容第三方库所需的配置文件
      ├─eslint.config.js
      ├─postcss.config.js
```

文件目录结构看过了，接下来看一下我是如何组织整理最后的配置文件的：

```
/* 开发环境 webpack 配置文件 webpack.dev.config.js */
module.exports = {
  entry: require('./webpack-config/entry.config.js'),
```

```
  output: require('./webpack-config/output.config.js'),

  module: require('./webpack-config/module.config.js'),

  resolve: require('./webpack-config/resolve.config.js'),

  plugins: require('./webpack-config/plugins.dev.config.js'),

  eslint: require('./webpack-config/vendor/eslint.config.js'),

  postcss: require('./webpack-config/vendor/postcss.config.js'),
};
```

这样，你就可以很轻松地处理开发/生产环境配置文件中相同与不同的部分了。

2.3.2.2 如何分别调用开发/生产环境的配置文件呢？

还记得我在《[webpack 多页应用架构系列（二）：webpack 配置常用部分有哪些？][3]》里讲过，我们在控制台调用 webpack 命令来启动打包时，可以添加上`--config`参数来指定 webpack 配置文件的路径吗？我们可以配合上`npm scripts`来使用，在 package.json 里定义：

```
  "scripts": {
    "build": "node build-script.js && webpack --progress --colors",
    "dev": "node build-script.js && webpack --progress --colors
--config ./webpack.dev.config.js",
    "watch": "webpack --progress --colors --watch
--config ./webpack.dev.config.js"
  },
```

这样一来，当我们开发的时候就可以使用`npm run dev`或`npm run watch`，而到要上线打包的时候就运行`npm run build`。

2.3.3 业务代码如何判断生产/开发环境

在业务代码里要判断生产/开发环境其实很简单，只需一个变量即可：

```
if (IS_PRODUCTION) {
    // 做生产环境该做的事情
} else {
    // 做开发环境该做的事情
}
```

这么一来，关键就在于这变量`IS_PRODUCTION`是怎么来的了。

在我还没分离开发和生产环境时，我用的办法是，开发时在业务代码所使用的配置文件中把这变量设为`false`，而在最后打包上线时就手动改为`true`。这种

方法我用过一段时间，非常繁琐，而且经常上线后发现，我嘞个去怎么 ajax 读的是我本地的 mock 服务器。
我参考了许多文章，先粗略讲讲我没有采用的方法：

- 用 `EnvironmentPlugin` 引入 process.env，这样就可以在业务代码中靠 `process.env.NODE_ENV` 来判断了。
- 用 `ProvidePlugin` 来控制在不同环境里加载不同的配置文件（业务代码用的）。

那我用的是什么方法呢？我最后选用的是[`DefinePlugin`][4]。
举个官方例子，其大概用法是这样的：

```
new webpack.DefinePlugin({
    PRODUCTION: JSON.stringify(true),
    VERSION: JSON.stringify("5fa3b9"),
    BROWSER_SUPPORTS_HTML5: true,
    TWO: "1+1",
    "typeof window": JSON.stringify("object")
})
```

`DefinePlugin` 可能会被误认为其作用是在 webpack 配置文件中为编译后的代码上下文环境设置全局变量，但其实不然。它真正的机制是：`DefinePlugin` 的参数是一个 object，那么其中会有一些 `key-value` 对。在 webpack 编译的时候，会把业务代码中没有定义（使用 var/const/let 来预定义的）而变量名又与 `key` 相同的变量（直接读代码的话的确像是全局变量）替换成 `value`。例如上面的官方例子，`PRODUCTION` 就会被替换为 `true`；`VERSION` 就会被替换为`'5fa3b9'`（注意单引号）；`BROWSER_SUPPORTS_HTML5` 也是会被替换为 `true`；`TWO` 会被替换为 `1+1`（相当于是一个数学表达式）；`typeof window` 就被替换为`'object'`了。
再举个例子，比如你在代码里是这么写的：

```
if (!PRODUCTION)
    console.log('Debug info')
if (PRODUCTION)
    console.log('Production log')
```

那么在编译生成的代码里就会是这样了：

```
if (!true)
    console.log('Debug info')
if (true)
    console.log('Production log')
```

而如果你用了 `UglifyJsPlugin`，则会变成这样：

```
console.log('Production log')
```

如此一来，只要在俩环境的配置文件里用 `DefinePlugin` 分别定义好 `IS_PRODUCTION` 的值，我们就可以在业务代码里进行判断了：

```
  /* global IS_PRODUCTION:true */
  if (!IS_PRODUCTION) {
    console.log('如果你看到这个 Log，那么这个版本实际上是开发用的版本');
  }
```

需要注意的是，如果你在 webpack 里整合了 ESLint，那么，由于 ESLint 会检测没有定义的变量（ESLint 要求使用全局变量时要用 `window.xxxxx` 的写法），因此需要一个 `global` 注释声明（`/* global IS_PRODUCTION:true */`）IS_PRODUCTION 是一个全局变量(当然在本例中并不是)来规避 warning。

第三章：整合第三方工具

3.1 教练我要写 ES6！webpack 怎么整合 Babel？

3.1.1 前言

一直以来，我对 ES6 都不甚感兴趣，一是因为在生产环境中使用 ES5 已是处处碰壁，其次则是只当这 ES6 是语法糖不曾重视。只是最近学习 react 生态，用起 babel 来转换 jsx 之余，也不免碰到诸多用上 ES6 的教程、案例，因此便稍作学习。这一学习，便觉得这语法糖实在是甜，忍不住尝鲜，于是记录部分自觉对自己有用的方法在此。

这是我数月前的一篇文章《ES6 部分方法点评（一）》中的一段，如今再看我自己的代码，触目皆是 ES6 的语法。在当前的浏览器市场下，想在生产环境用上 ES6，Babel 是必不可少的。

由于我本身只用了 ES6 的语法而未使用 ES6 的其它特性，因此本文只介绍如何利用 webpack 整合 Babel 来编译 ES6 的语法，而实际上若要使用 ES6 的其它属性甚至是 ES7（ES2016），其实只需要引入 Babel 其它的 preset/plugin 即可，在用法上并无多大变化。

3.1.2 用到哪些 npm 包？

首先要说到的是 `babel-loader`，这是 webpack 整合 Babel 的关键，我们需要配置好 babel-loader 来加载那些使用了 ES6 语法的 js 文件；换句话说，那些本来就是 ES5 语法的文件，其实是不需要用 babel-loader 来加载的，用了也只会浪费我们编译的时间。
然后就是 babel 相关的 npm 包，其中包括：

- `babel-core`，babel 的核心，没啥好说的。
- `babel-preset-es2015-loose`，babel 的 preset（相当于是一整套 plugin）。babel 是有许多 preset 的，看自己需要来选用，比如说我只管 ES6（ES2016）语法的就可以用 `babel-preset-es2015` 或 `babel-preset-es2015-loose`。这俩 preset 其实用法一样，差别就在于：

  许多 Babel 的插件有两种模式：

  尽可能符合 ECMAScript6 语义的 normal 模式和提供更简单 ES5 代码的 loose 模式。

  优点：生成的代码可能更快，对老的引擎有更好的兼容性，代码通常更简洁，更加的“ES5 化”。

缺点：你是在冒险——随后从转译的 ES6 到原生的 ES6 时你会遇到问题。

我自己的考虑是，肯定要更好的兼容性和更好的性能啦这还用想的吗？（敲黑板）

- `babel-plugin-transform-runtime` 和 `babel-runtime`，这属于优化项，不用也没啥问题，下文会细说。

3.1.3 如何配置 babel-loader

babel-loader 的配置并不复杂，与其它 loader 并无二致：

```
    {
      test: /\.js$/,
      exclude: /node_modules|vendor|bootstrap/,
      loader:
'babel-loader?presets[]=es2015-loose&cacheDirectory&plugins[]=transform-r
untime',
    },
```

下面来详细解释此配置：

- `test: /\.js$/`表明我只用 babel-loader 来加载 js 文件，如果你只是小部分 js 文件应用了 ES6，那么也可以给这些文件换个`.es6`的后缀名并把此处改为 `test: /\.es6$/`。
- `exclude: /node_modules|vendor|bootstrap/`，上文已经说到了，不需要用 babel 来加载的文件还是剔除掉，否则会大量增加编译的时间，一般我们只用 babel 编译我们自己写的应用代码。
- `loader: 'babel-loader?presets[]=es2015-loose&cacheDirectory&plugins[]=transform-runtime'`，这一行是指定使用 babel-loader 并传入所需参数，这些参数其实也是可以通过 babel 配置文件.babelrc，不过我还是推荐在这里以参数的方式传入。下面来介绍这些参数：

3.1.3.1preset 参数：`babel-preset-es2015-loose`

上文已经解释过 preset 是什么以及为啥要使用 `babel-preset-es2015-loose` 了，这里不再累述。

3.1.3.2cacheDirectory 参数

cacheDirectory 参数默认为 false，若你设置为一个文件目录路径（表示把 cache 存到哪），或是保留为空（表示操作系统默认的缓存目录），则相当于开启 cache。这里的 cache 指的是 babel 在编译过程中某些可以缓存的步骤，具体是什么我也不太清楚，反正是只要开启了 cache 就可以加快 webpack 整

体编译速度。我测试了一下，未开启 cache 的时候我的脚手架项目(Array-Huang/webpack-seed)需要 15 秒半来编译；而开启 cache 后的第一次编译时间并没有减少，第二次编译则变为 14 秒了，足足减少了 1 秒半了棒棒哒。

3.1.3.3plugins 参数

虽说一个 preset 已经包括 N 个 plugin 了，但总有一些漏网之鱼是要专门加载的。这里我只用到了 `transform-runtime`，这个 plugin 的效果是：不用这 plugin 的话，babel 会为每一个转换后的文件（在 webpack 这就是每一个 chunk 了）都添加一些辅助的方法（仅在需要的情况下）；而如果用了这个 plugin，babel 会把这些辅助的方法都集中到一个文件里统一加载统一管理，算是一个减少冗余，增强性能的优化项吧，用不用也看自己需要了；如果不用的话，前面也不需要安装 `babel-plugin-transform-runtime` 和 `babel-runtime` 了。

3.2 总有刁民想害朕！ESLint 为你阻击垃圾代码

3.2.1 前言

3.2.1.1 刁民，还不退下？啊.....来人啊快救驾！

你所在的团队里有没有“老鼠屎”？就是专门写各种看起来溜得飞起但实际上晦涩难懂的代码？又或是缩进换行乱成一团？

你写代码是不是特粗心？经常落下些语法错误，debug 起来想死？

如果你有以上问题，ESLint 帮到你！[手动滑稽]

3.2.1.2ESLint 的用途是？

从上面两个应用场景，你大概已经猜到 ESLint 是用来干什么的了：

- 审查代码是否符合编码规范和统一的代码风格；
- 审查代码是否存在语法错误；

语法错误好说，编码规范和代码风格如何审查呢？ESLint 定义好了一大堆规则作为可配置项;同时，一些大公司会开源出来他们使用的配置（比如说 `airbnb`），你可以在某套现成配置的基础上进行修改，修改成适合你们团队使用的编码规范和代码风格。

3.1.2.3 本文主要讲什么？

本文着重介绍如何在 webpack 里整合进 ESLint，而并不介绍 ESLint 本身，因此，对于没有使用过 ESLint 的小伙伴，请先去自己入门一下啦。

3.2.2webpack 如何整合 ESLint？

这次我们需要使用到 `eslint-loader`，先放出配置的代码：

```
/* 这是 webpack 配置文件的内容，省略无关部分 */
{
  module: {
    preLoaders: [{
      test: /\.js$/, // 只针对 js 文件
      loader: 'eslint', // 指定启用 eslint-loader
      include: dirVars.srcRootDir, // 指定审查范围仅为自己团队写的业务代码
      exclude: [/bootstrap/], // 剔除掉不需要利用 eslint 审查的文件
    }],
  },
  eslint: {
    configFile: path.resolve(dirVars.staticRootDir, './.eslintrc'), // 指定
eslint 的配置文件在哪里
    failOnWarning: true, // eslint 报 warning 了就终止 webpack 编译
    failOnError: true, // eslint 报 error 了就终止 webpack 编译
    cache: true, // 开启 eslint 的 cache，cache 存在 node_modules/.cache 目录里
  }
}
```

接下来解释一下这份 eslint-loader 的配置。

3.2.2.1 为嘛把 eslint-loader 放在 `preLoaders` 而不是 `loaders` 里？

理论上来说放 loaders 里也无伤大雅，但放 preLoaders 里有以下好处：

- 放在 preLoader 是先于 loader 的，因此当 ESLint 审查到问题报了 warning/error 的时候就会停掉，可以稍微省那么一点点时间吧大概[手动滑稽]。
- 如果你使用了 babel，或类似的 loader，那么，通过 webpack 编译前后的代码相差就很大了，这会造成两个问题（以 babel 为例）：
  - babel 把你的代码转成什么样你自己是无法控制的，这往往导致无法通过 ESLint 的审查。
  - 我们实际上并不关心编译后生成的代码，我们只需要管好我们自己手写的代码即可，反正谁也不会没事去读读编译后的代码吧？

3.2.2.2 如何传参给 eslint-loader？

从 eslint-loader 官方文档可以看出，eslint-loader 的配置还是比较多也比较复杂的，因此采用了独立的一个配置项 `eslint`（跟 `module` 同级哈）。

3.2.3 总结

只要你能在自己团队里成功推行 ESLint，那么最起码，你可以放心不用再看到那些奇奇怪怪的代码了，因为，它们都编译不通过呐哈哈哈哈哈……

3.2.4 后话

通过 webpack 整合 ESLint，我们可以保证编译生成的代码都是没有语法错误且符合编码规范的；但在开发过程中，等到编译的时候才察觉到问题可能也是太慢了点儿。

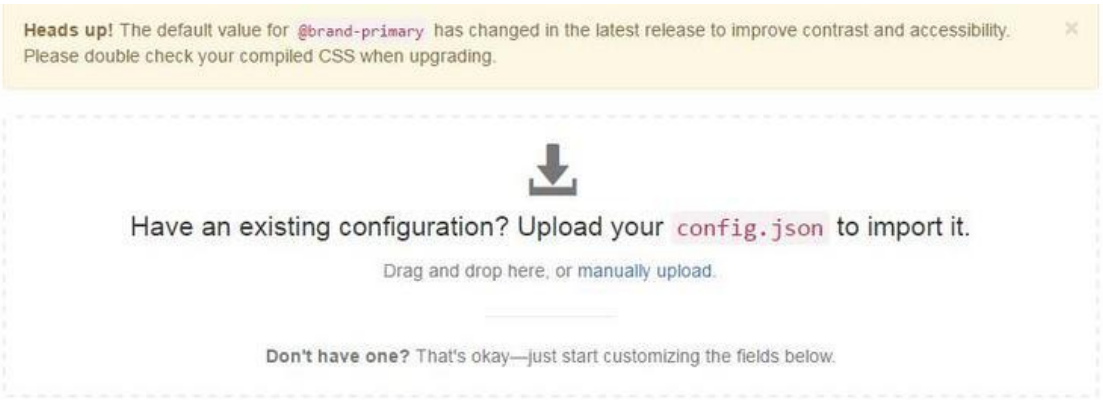
因此我建议可以把 ESLint 整合进编辑器或 IDE 里，像我本人在用 `Sublime Text 3` 的，就可以使用一个名为 `SublimeLinter` 的插件，一写了有**问题**的代码，就马上会标识出来，如下图所示：

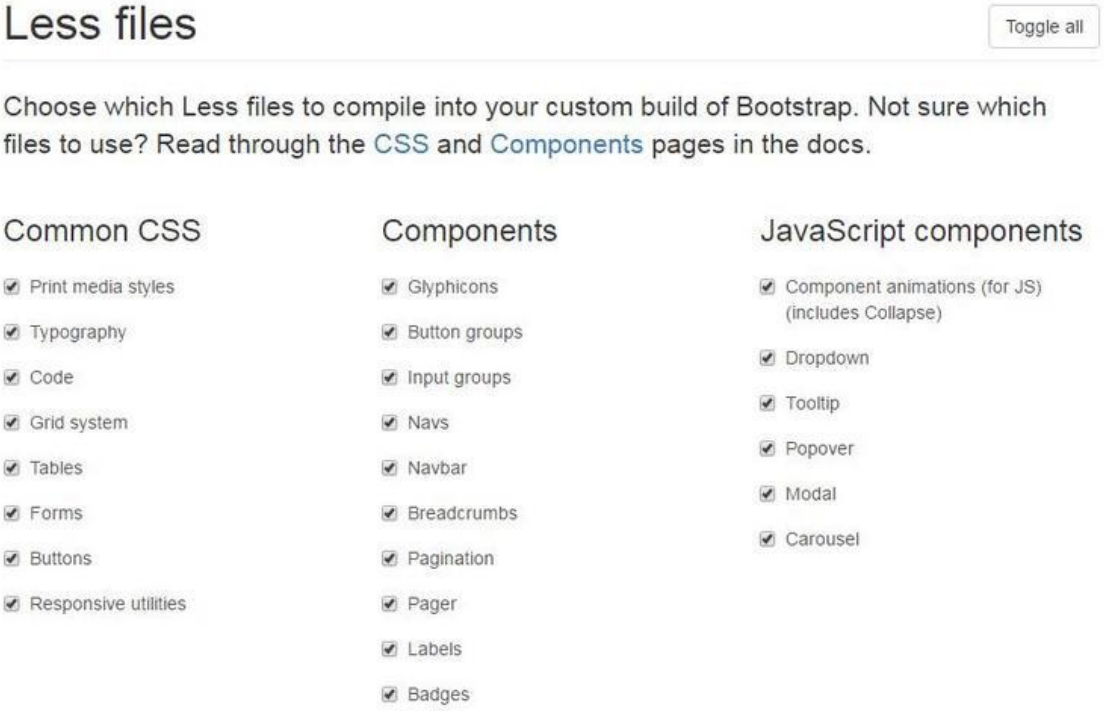
第四章：webpack 的进阶应用

4.1 如何打造一个自定义的 bootstrap？

4.1.1 前言

一般我们用 bootstrap 呐，都是用的从官网或 github 下载下来 build 好了的版本，千人一脸呐多没意思。当然，官网也给我们提供了自定义的工具，如下图所示，但每次要改些什么就要重新在官网上打包一份，而且还是个国外的网站，甭提有多烦躁了。





那么，有没有办法让我们随时随地都能根据业务的需要来自定义 bootstrap 呢？答案自然是肯定的，webpack 有啥干不了的呀（大误）[手动滑稽]

sass/less 的两套方案

bootstrap 主要由两部分组成：样式和 jQuery 插件。这里要说的是样式，bootstrap 有 less 的方案，也有 sass 的方案，因此，也存在两个 loader 分别对应这两套方案：less <=> bootstrap-webpack 和 sass <=> bootstrap-loader 。

我个人惯用的是 less，因此本文以 `bootstrap-webpack` 为例来介绍如何打造一个自定义的 bootstrap。

4.1.2 开工了！

4.1.2.1 先引入全局的 jQuery

众所周知，bootstrap 这货指明是要全局的 jQuery 的，甭以为现在用 webpack 打包的就有什么突破了。引入全局 jQuery 的方法请看这篇文章《老式 jQuery 插件还不能丢，怎么兼容？》（`ProvidePlugin` + `expose-loader`），我的脚手架项目 Array-Huang/webpack-seed 也是使用的这套方案。

4.1.2.2 如何加载 bootstrap 配置？

bootstrap-webpack 提供一个默认选配下的 bootstrap，不过默认的我要你何用（摔

好，言归正题，我们首先需要新建两个配置文件 `bootstrap.config.js` 和 `bootstrap.config.less`，并将这俩文件放在同一级目录下（像我就把业务代码里用到的 config 全部丢到同一个目录里了哈哈哈）。
因为每个页面都需要，也只需要引用一次，因此我们可以找个每个页面都会加载的公共模块(用 Array-Huang/webpack-seed 来举例就是 `src/public-resource/logic/common.page.js`，我每个页面都会加载这个 js 模块)来加载 bootstrap：

```
require('!!bootstrap-webpack!bootstrapConfig'); // bootstrapConfig 是我在 webpack 配置文件中设好的 alias，不设的话这里就填实际的路径就好了
```

上文已经说到，bootstrap-webpack 其实就是一个 webpack 的 loader，所以这里是用 loader 的语法。需要注意的是，如果你在 webpack 配置文件中针对 js 文件设置了 loader（比如说 babel），那么在加载 bootstrap-webpack 的时候请在最前面加个`!!`，表示这个 `require` 语句忽略 webpack 配置文件中所有 loader 的配置，还有其它的用法，看自己需要哈：
adding ! to a request will disable configured preLoaders adding !! to a request will disable all loaders specified in the configuration adding -! to a request will disable configured preLoaders and loaders but not the postLoaders

4.1.2.3 如何配置 bootstrap?

上文提到有两个配置文件，`bootstrap.config.js` 和 `bootstrap.config.less`，显然，它们的作用是不一样的。

`bootstrap.config.js`

`bootstrap.config.js` 的作用就是配置需要加载哪些组件的样式和哪些 jQuery 插件，可配置的内容跟官网是一致的，官方给出这样的例子：

```
module.exports = {
  scripts: {
    // add every bootstrap script you need
    'transition': true
  },
  styles: {
    // add every bootstrap style you need
    "mixins": true,

    "normalize": true,
    "print": true,

    "scaffolding": true,
    "type": true,
  }
};
```

当时我是一下子懵逼了，就这么几个？完整的例子/文档在哪里？后来终于被我找到默认的配置了，直接拿过来在上面改改就能用了：

```
var ExtractTextPlugin = require('extract-text-webpack-plugin');
module.exports = {
  styleLoader:
ExtractTextPlugin.extract('css?minimize&-autoprefixer!postcss!less'),
  scripts: {
    transition: true,
    alert: true,
    button: true,
    carousel: true,
    collapse: true,
    dropdown: true,
    modal: true,
    tooltip: true,
    popover: true,
    scrollspy: true,
```

```
    tab: true,
    affix: true,
  },
  styles: {
    mixins: true,

    normalize: true,
    print: true,

    scaffolding: true,
    type: true,
    code: true,
    grid: true,
    tables: true,
    forms: true,
    buttons: true,

    'component-animations': true,
    glyphicons: false,
    dropdowns: true,
    'button-groups': true,
    'input-groups': true,
    navs: true,
    navbar: true,
    breadcrumbs: true,
    pagination: true,
    pager: true,
    labels: true,
    badges: true,
    jumbotron: true,
    thumbnails: true,
    alerts: true,
    'progress-bars': true,
    media: true,
    'list-group': true,
    panels: true,
    wells: true,
    close: true,

    modals: true,
    tooltip: true,
    popovers: true,
    carousel: true,
```

```
    utilities: true,
    'responsive-utilities': true,
  },
};
```

这里的 `scripts` 项就是 jQuery 插件了，而 `styles` 项则是样式，可以分别对照着 bootstrap 英文版文档来查看。

需要解释的是 `styleLoader` 项，这表示用什么 loader 来加载 bootstrap 的样式，相当于 webpack 配置文件中针对 `.less` 文件的 loader 配置项吧，这里我也是直接从 webpack 配置文件里抄过来的。

另外，由于我使用了 iconfont 作为图标的解决方案，因此就去掉了 `glyphicons`；如果你要使用 `glyphicons` 的话，请务必在 webpack 配置中设置好针对各类字体文件的 loader 配置，否则可是会报错的哦。

`bootstrap.config.less`

`bootstrap.config.less` 配置的是 less 变量，bootstarp 官网上也有相同的配置，这里就不多做解释了，直接放个官方例子：

```
@font-size-base: 24px;
@btn-default-color: #444;
@btn-default-bg: #eee;
```

需要注意的是，我一开始只用了 `bootstrap.config.js` 而没建 `bootstrap.config.less`，结果发现报错了，还来建了个空的 `bootstrap.config.less` 就编译成功了，因此，无论你有没有配置 less 变量的需要，都请新建一个 `bootstrap.config.less`。

4.1.3 总结

至此，一个可自定义的 bootstrap 就出炉了，你想怎么折腾都行了，什么不用的插件不用的样式，统统给它去掉，把体积减到最小，哈哈哈。

4.1.4 后话

此方案有个缺点：此方案相当于每次编译项目时都把整个 bootstrap 编译一遍，而 bootstrap 是一个庞大的库，每次编译都会耗费不少的时间，如果只是编译一次也就算了，每次都要耗这时间那可真恶心呢。所以，我打算折腾一下看能不能有所改进，在这里先记录下原始的方案，后面如果真能改进会继续写文的了哈。

4.2 预打包 Dll，实现 webpack 音速编译

4.2.1 前言

书承上文《如何打造一个自定义的 bootstrap》。

上文说到我们利用 webpack 来打包一个可配置的 bootstrap，但文末留下一个问题：由于 bootstrap 十分庞大，因此每次编译都要耗费大部分的时间在打包 bootstrap 这一块，而换来的仅仅是配置的便利，十分不划算。

我也并非是故意卖关子，这的确是我自己开发中碰到的问题，而在撰写完该文后，我立即着手探索解决之道。终于，发现了 webpack 这一大杀器：`DllPlugin`&`DllReferencePlugin`，打包时间过长的问题得到完美解决。

4.2.2 解决方案的机制和原理

`DllPlugin`&`DllReferencePlugin` 这一方案，实际上也是属于代码分割的范畴，但与 CommonsChunkPlugin 不一样的是，它不仅仅是把公用代码提取出来放到一个独立的文件供不同的页面来使用，它更重要的一点是：把公用代码和它的使用者（业务代码）从编译这一步就分离出来，换句话说，我们可以分别来编译公用代码和业务代码了。这有什么好处呢？很简单，业务代码常改，而公用代码不常改，那么，我们在日常修改业务代码的过程中，就可以省出编译公用代码那一部分所耗费的时间了（是不是马上就联想到坑爹的 bootstrap 了呢）。整个过程大概是这样的：

1. 利用 `DllPlugin` 把公用代码打包成一个**“Dll 文件”**（其实本质上还是 js，只是套用概念而已）；除了 Dll 文件外，`DllPlugin` 还会生成一个 manifest.json 文件作为公用代码的索引供 `DllReferencePlugin` 使用。
2. 在业务代码的 webpack 配置文件中配置好 `DllReferencePlugin` 并进行编译，达到利用 `DllReferencePlugin` 让业务代码和 Dll 文件实现关联的目的。
3. 在各个页面

   中，先加载 Dll 文件，再加载业务代码文件。

4.2.2.1 适用范围

Dll 文件里只适合放置不常改动的代码，比如说第三方库（谁也不会有事无事就升级一下第三方库吧），尤其是本身就庞大或者依赖众多的库。如果你自己整理了一套成熟的框架，开发项目时只需要在上面添砖加瓦的，那么也可以把这套框架也打包进 Dll 文件里，甚至可以做到多个项目共用这一份 Dll 文件。

4.2.2.2 如何配置哪些代码需要打包进 Dll 文件？

我们需要专门为 Dll 文件建一份 webpack 配置文件，不能与业务代码共用同一份配置：

```
const webpack = require('webpack');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
const dirVars = require('./webpack-config/base/dir-vars.config.js'); // 与业务代码共用同一份路径的配置表

module.exports = {
  output: {
    path: dirVars.dllDir,
    filename: '[name].js',
    library: '[name]', // 当前 Dll 的所有内容都会存放在这个参数指定变量名的一个全局变量下，注意与 DllPlugin 的 name 参数保持一致
  },
  entry: {
    /*
      指定需要打包的 js 模块
      或是 css/less/图片/字体文件等资源，但注意要在 module 参数配置好相应的 loader
    */
    dll: [
      'jquery', '!!bootstrap-webpack!bootstrapConfig',
      'metisMenu/metisMenu.min', 'metisMenu/metisMenu.min.css',
    ],
  },
  plugins: [
    new webpack.DllPlugin({
      path: 'manifest.json', // 本 Dll 文件中各模块的索引，供 DllReferencePlugin 读取使用
      name: '[name]',  // 当前 Dll 的所有内容都会存放在这个参数指定变量名的一个全局变量下，注意与参数 output.library 保持一致
      context: dirVars.staticRootDir, // 指定一个路径作为上下文环境，需要与 DllReferencePlugin 的 context 参数保持一致，建议统一设置为项目根目录
    }),
    /* 跟业务代码一样，该兼容的还是得兼容 */
    new webpack.ProvidePlugin({
      $: 'jquery',
      jQuery: 'jquery',
      'window.jQuery': 'jquery',
      'window.$': 'jquery',
    }),
```

```
    new ExtractTextPlugin('[name].css'), // 打包 css/less 的时候会用到 ExtractTextPlugin
  ],
  module: require('./webpack-config/module.config.js'), // 沿用业务代码的 module 配置
  resolve: require('./webpack-config/resolve.config.js'), // 沿用业务代码的 resolve 配置
};
```

4.2.2.3 如何编译 Dll 文件？

编译 Dll 文件的代码实际上跟编译业务代码是一样的，记得利用`--config`指定上述专供 Dll 使用的 webpack 配置文件就好了：

```
$ webpack --progress --colors --config ./webpack-dll.config.js
```

另外，建议可以把该语句写到`npm scripts`里，好记一点哈。

4.2.2.4 如何让业务代码关联 Dll 文件？

我们需要在供编译业务代码的 webpack 配置文件里设好`DllReferencePlugin`的配置项：

```
new webpack.DllReferencePlugin({
  context: dirVars.staticRootDir, // 指定一个路径作为上下文环境，需要与 DllPlugin 的 context 参数保持一致，建议统一设置为项目根目录
  manifest: require('../../manifest.json'), // 指定 manifest.json
  name: 'dll',  // 当前 Dll 的所有内容都会存放在这个参数指定变量名的一个全局变量下，注意与 DllPlugin 的 name 参数保持一致
});
```

配置好`DllReferencePlugin`了以后，正常编译业务代码即可。不过要注意，必须要先编译 Dll 并生成 manifest.json 后再编译业务代码；而以后每次修改 Dll 并重新编译后，也要重新编译一下业务代码。

4.2.2.5 如何在业务代码里使用 Dll 文件打包的 module/资源？

不需要刻意做些什么，该怎么 require 就怎么 require，webpack 都会帮你处理好的了。

4.2.2.6 如何整合 Dll？

在每个页面里，都要按这个顺序来加载 js 文件：Dll 文件 => `CommonsChunkPlugin`生成的公用 chunk 文件（如果没用`CommonsChunkPlugin`那就忽略啦） => 页面本身的入口文件。

有两个注意事项：

- 如果你是像我一样利用 `HtmlWebpackPlugin` 来生成 HTML 并自动加载 chunk 的话，请务必在`<head>`里手写`<script>`来加载 Dll 文件。
- 为了完全分离源文件和编译后生成的文件，也为了方便在编译前可以清空 build 目录，不应直接把 Dll 文件编译生成到 build 目录里，我建议可以先生成到源文件 src 目录里，再用 `file-loader` 给原封不动搬运过去。

4.2.2.7 光说不练假把式，来个跑分啊大兄弟！

下面以我的脚手架项目 Array-Huang/webpack-seed 为例，测试一下（使用开发环境的 webpack 配置文件 `webpack.dev.config.js`）使用这套 Dll 方案前后的 webpack 编译时间：

- 使用 Dll 方案前的编译时间为：10 秒 17
- 使用 Dll 方案后的编译时间为：4 秒 29

由于该项目只是一个脚手架，涉及到的第三方库并不多，我只把 jQuery、bootstrap、metisMenu 给打包进 Dll 文件里了，尽管如此，还是差了将近 6 秒了，相信在实际项目中，这套 `DllPlugin`&`DllReferencePlugin` 的方案能为你省下更多的时间来找女朋友（大误）。

4.3 利用 webpack 生成 HTML 普通网页&页面模板

4.3.1 为什么要用 webpack 来生成 HTML 页面

按照我们前面的十一篇的内容来看，自己写一个 HTML 页面，然后在上面加载 webpack 打包的 js 或其它类型的资源，感觉不也用得好好的么？

是的没错，不用 webpack 用 requireJs 其实也可以啊，甚至于，传统那种人工管理模块依赖的做法也没有什么问题嘛。

但既然你都已经看到这一篇了，想必早已和我一样，追求着以下这几点吧：

- 更懒，能自动化的事情绝不做第二遍。
- 更放心，调通的代码比人靠谱，更不容易出错。
- 代码洁癖，什么东西该放哪，一点都不能含糊，混在一起我就要死了。

那么，废话不多说，下面就来说说使用 webpack 生成 HTML 页面有哪些好处吧。

4.3.1.1 对多个页面共有的部分实现复用

在实际项目的开发过程中，我们会发现，虽然一个项目里会有很多个页面，但这些页面总有那么几个部分是相同或相似的，尤其是页头页尾，基本上是完全一致的。那我们要怎么处理这些共有的部分呢？

复制粘贴流

不就是复制粘贴的事嘛？写好一份完整的 HTML 页面，做下个页面的时候，直接 copy 一份文件，然后直接在 copy 的文件上进行修改不就好了吗？

谁是这么想这么做的，放学留下来，我保证不打死你！我曾经接受过这么一套系统，顶部栏菜单想加点东西，就要每个页面都改一遍，可维护性烂到爆啊。

Iframe 流

Iframe 流常见于管理后台类项目，可维护性 OK，就是缺陷比较多，比如说：

- 点击某个菜单，页面是加载出来了但是浏览器地址栏上的 URL 没变，刷新的话又回到首页了。
- 搜索引擎收录完蛋，前台项目一般不能用 Iframe 来布局。
- 没有逼格，Low 爆了，这是最重要的一点（大误）。

SPA 流

最近这几年，随着移动互联网的兴起，SPA 也变得非常常见了。不过 SPA 的局限性也非常大，比如搜索引擎无法收录，但我个人最在意的，是它太复杂了，尤其是一些本来业务逻辑就多的系统，很容易懵圈。

后端模板渲染

这倒真是一个办法，只是，需要后端的配合，利用后端代码把页面的各个部分给拼合在一起，所以这方法对前端起家的程序员还是有点门槛的。

利用前端模板引擎生成 HTML 页面

所谓“用 webpack 生成 HTML 页面”，其实也并不是 webpack 起的核心作用，实际上靠的还是前端的模板引擎将页面的各个部分给拼合在一起来达到公共区域的复用。webpack 更多的是组织统筹整个生成 HTML 页面的过程，并提供更大的控制力。最终，webpack 生成的到底是完整的页面，还是供后端渲染

的模板，就全看你自己把控了，非常灵活，外人甚至察觉不出来这到底是你自己写的还是代码统一生成的。

4.3.1.2 处理资源的动态路径

如果你想用**在文件名上加 hash** 的方法作为缓存方案的话，那么用 webpack 生成 HTML 页面就成为你唯一的选择了，因为随着文件的变动，它的 hash 也会变化，那么整个文件名都会改变，你总不能在每次编译后都手动修改加载路径吧？还是放心交给 webpack 吧。

4.3.1.3 自动加载 webpack 生成的 css、less

如果你使用 webpack 来生成 HTML 页面，那么，你可以配置好每个页面加载的 chunk（webpack 打包后生成的 js 文件），生成出来的页面会自动用`<script>`来加载这些 chunk，路径什么的你都不用管了哈（当然前提是你配置好了 output.publicPath）。另外，用`extract-text-webpack-plugin`打包好的 css 文件，webpack 也会帮你自动添加到`<link>`里，相当方便。

4.3.1.4 彻底分离源文件目录和生成文件目录

使用 webpack 生成出来的 HTML 页面可以很安心地跟 webpack 打包好的其它资源放到一起，相对于另起一个目录专门存放 HTML 页面文件来说，整个文件目录结构更加合理：

```
build
  - index
    - index
      - entry.js
      - page.html
    - login
      - entry.js
      - page.html
      - styles.css
```

4.3.2 如何利用 webpack 生成 HTML 页面

webpack 生成 HTML 页面主要是通过`html-webpack-plugin`来实现的，下面来介绍如何实现。

4.3.2.1html-webpack-plugin 的配置项

每一个 html-webpack-plugin 的对象实例都只针对/生成一个页面，因此，我们做多页应用的话，就要配置多个 html-webpack-plugin 的对象实例：

```
pageArr.forEach((page) => {
  const htmlPlugin = new HtmlWebpackPlugin({
    filename: `${page}/page.html`,
    template: path.resolve(dirVars.pagesDir, `./${page}/html.js`),
    chunks: [page, 'commons'],
    hash: true, // 为静态资源生成 hash 值
    minify: true,
    xhtml: true,
  });
  configPlugins.push(htmlPlugin);
});
```

pageArr 实际上是各个 chunk 的 name，由于我在 output.filename 设置的是 '[name]/entry.js'，因此也起到构建文件目录结构的效果（具体请看这里），附上 pageArr 的定义：

```
module.exports = [
  'index/login',
  'index/index',
  'alert/index',
  'user/edit-password', 'user/modify-info',
];
```

html-webpack-plugin 的配置项真不少，这里仅列出多页应用常用到的配置：

- filename，生成的网页 HTML 文件的文件名，注意可以利用/来控制文件目录结构的，其最终生成的路径，是基于 webpack 配置中的 output.path 的。
- template，指定一个基于某种模板引擎语法的模板文件，html-webpack-plugin 默认支持 **ejs** 格式的模板文件，如果你想使用其它格式的模板文件，那么需要在 webpack 配置里设置好相应的 loader，比如 handlebars-loader 啊 html-loader 啊之类的。如果不指定这个参数，html-webpack-plugin 会使用一份默认的 ejs 模板进行渲染。如果你做的是简单的 SPA 应用，那么这个参数不指定也行，但对于多页应用来说，我们就依赖模板引擎给我们拼装页面了，所以这个参数非常重要。
- inject，指示把加载 js 文件用的<script>插入到哪里，默认是插到<body>的末端，如果设置为'head'，则把<script>插入到<head>里。
- minify，生成压缩后的 HTML 代码。
- hash，在**由 html-webpack-plugin 负责加载的 js/css 文件**的网址末尾加个 URL 参数，此 URL 参数的值是代表本次编译的一个 hash 值，每次编译后该 hash 值都会变化，属于缓存解决方案。
- chunks，以数组的形式指定由 html-webpack-plugin 负责加载的 chunk 文件（打包后生成的 js 文件），不指定的话就会加载所有的 chunk。

4.3.2.2 生成一个简单的页面

下面提供一份供生成简单页面（之所以说简单，是因为不指定页面模板，仅用默认模板）的配置：

```
var HtmlWebpackPlugin = require('html-webpack-plugin');
var webpackConfig = {
  entry: 'index.js',
  output: {
    path: 'dist',
    filename: 'index_bundle.js'
  },
  plugins: [new HtmlWebpackPlugin({
    title: '简单页面',
    filename: 'index.html',
  })],
};
```

使用这份配置编译后，会在 dist 目录下生成一个 index.html，内容如下所示：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>简单页面</title>
  </head>
  <body>
    <script src="index_bundle.js"></script>
  </body>
</html>
```

由于没有指定模板文件，因此生成出来的 HTML 文件仅有最基本的 HTML 结构，并不带实质内容。可以看出，这更适合 React 这种把 HTML 藏 js 里的方案。

4.3.2.3 利用模板引擎获取更大的控制力

接下来，我们演示如何通过制定模板文件来生成 HTML 的内容，由于 html-webpack-plugin 原生支持 ejs 模板，因此这里也以 ejs 作为演示对象：

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8"/>
```

```
    <meta name="viewport"
content="width=device-width,initial-scale=1,maximum-scale=1" />
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <h1>这是一个用<b>html-webpack-plugin</b>生成的 HTML 页面</h1>
    <p>大家仔细瞧好了</p>
  </body>
</html>
```

'html-webpack-plugin'的配置里也要指定 **template** 参数：

```
var HtmlWebpackPlugin = require('html-webpack-plugin');
var webpackConfig = {
  entry: 'index.js',
  output: {
    path: 'dist',
    filename: 'index_bundle.js'
  },
  plugins: [new HtmlWebpackPlugin({
    title: '按照 ejs 模板生成出来的页面',
    filename: 'index.html',
    template: 'index.ejs',
  })],
};
```

那么，最后生成出来的 HTML 文件会是这样的：

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8"/>
    <meta name="viewport"
content="width=device-width,initial-scale=1,maximum-scale=1" />
    <title>按照 ejs 模板生成出来的页面</title>
  </head>
  <body>
    <h1>这是一个用<b>html-webpack-plugin</b>生成的 HTML 页面</h1>
    <p>大家仔细瞧好了</p>
    <script src="index_bundle.js"></script>
  </body>
</html>
```

到这里，我们已经可以控制整个 HTML 文件的内容了，那么生成后端渲染所需的模板也就不是什么难事了，以 PHP 的模板引擎 smarty 为例：

```
<!DOCTYPE html>
```

```
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8"/>
    <meta name="viewport"
content="width=device-width,initial-scale=1,maximum-scale=1" />
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <h1>这是一个用<b>html-webpack-plugin</b>生成的 HTML 页面</h1>
    <p>大家仔细瞧好了</p>
    <p>这是用 smarty 生成的内容：<b>{$articleContent}</b></p>
  </body>
</html>
```

4.3.2.4 处理资源的动态路径

接下来在上面例子的基础上，我们演示如何处理资源的动态路径：

```
var HtmlWebpackPlugin = require('html-webpack-plugin');
var webpackConfig = {
  entry: 'index.js',
  output: {
    path: 'dist',
    filename: 'index_bundle.[chunkhash].js'
  },
  plugins: [new HtmlWebpackPlugin({
    title: '按照 ejs 模板生成出来的页面',
    filename: 'index.html',
    template: 'index.ejs',
  })],
  module: {
    loaders: {
      // 图片加载器，雷同 file-loader，更适合图片，可以将较小的图片转成 base64，
减少 http 请求
      // 如下配置，将小于 8192byte 的图片转成 base64 码
      test: /\.(png|jpg|gif)$/,
      loader: 'url?limit=8192&name=./static/img/[hash].[ext]',
    },
  },
};
<!DOCTYPE html>
<html>
  <head>
```

```
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8"/>
    <meta name="viewport"
content="width=device-width,initial-scale=1,maximum-scale=1" />
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <h1>这是一个用<b>html-webpack-plugin</b>生成的 HTML 页面</h1>
    <p>大家仔细瞧好了</p>
    <img src="<%= require('./imgs/login-bg.jpg')  %>" />
  </body>
</html>
```

我们改动了什么呢？

1. 参数 `output.filename` 里，我们添了个变量[chunkhash]，这个变量的值会随 chunk 内容的变化而变化，那么，这个 chunk 文件最终的路径就会是一个动态路径了。
2. 我们在页面上添加了一个`<img>`，它的 src 是 require 一张图片，相应地，我们配置了针对图片的 loader 配置，如果图片比较小，`require()`就会返回 DataUrl，而如果图片比较大，则会拷贝到 `dist/static/img/`目录下，并返回新图片的路径。

下面来看看，到底 `html-webpack-plugin` 能不能处理好这些动态的路径。

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8"/>
    <meta name="viewport"
content="width=device-width,initial-scale=1,maximum-scale=1" />
    <title>按照 ejs 模板生成出来的页面</title>
  </head>
  <body>
    <h1>这是一个用<b>html-webpack-plugin</b>生成的 HTML 页面</h1>
    <p>大家仔细瞧好了</p>
    <img
src="
wcJCQgKDBQNDAsLDBkSEw8UHRofHh0aHBwgJC4nICIsIxwcKDcpLDAxNDQ0Hyc5PTgyPC4zND
L/2wBDAQkJCQwLDBgNDRgyIRwhMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjI
yMjIyMjIyMjIyMjL/wAARCAAaAFADASIAAhEBAxEB/8QAHwAAAQUBAQEBAQEAAAAAAAAA
AAECAwQFBgcICQoL/8QAtRAAAgEDAwIEAwUFBAQAAAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBk
aEII0KxwRVS0fAkM2JyggkKFhcYGRolJicoKSo0NTY3ODk6Q0RFRkdISUpTVFVWV1hZWmNkZW
ZnaGlqc3R1dnd4eXqDhIWGh4iJipKTlJWWl5iZmqKjpKWmp6ipqrKztLW2t7i5usLDxMXGx8j
JytLT1NXW19jZ2uHi4+Tl5ufo6erx8vP09fb3+Pn6/8QAHwEAAwEBAQEBAQEBAQAAAAAAAAEC
AwQFBgcICQoL/8QAtREAAgECBAQDBAcFBAQAAQJ3AAECAxEEBSExBhJBUQdhcRMiMoEIFEKRo
bHBCSMzUvAVYnLRChYkNOEl8RcYGRomJygpKjU2Nzg5OkNERUZHSElKU1RVVldYWVpjZGVmZ2
hpanN0dXZ3eHl6goOEhYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMn
```

```
K0tPU1dbX2Nna4uPk5ebn6Onq8vP09fb3+Pn6/9oADAMBAAIRAxEAPwD29Y4fJkm1BxJLCkYu
HZHSHdH8+9EYkDls5BJ4ALEpxxjfETw1c3McU9tdSaW0ssZubq0MkZkbuHZiQu1nBXbnDgfKB
g7niC3uJfCmqyQQ3dvLLBKsiTT+YwVTIwKqC6/MTjAwdjAZBRVHiFxNZnwFZxgxPfR6jIQox5
iRsiZPqFJAz67e5FcVfEShPTok/W7tY4sViJ0pJQ03/D7j6J+0XEtzHDCsSr5SyysHZeWGFT
gK2QJOd2V+QlSGrB1LxFqh1+40/SvD76ibDY8ki34gAZ1OAQR8wwenPODjIFafhvY2jWUnlwC
Q2duA6Nlnj2Aru44+Yvgcjvnkgc9c+HbaefW9ZbxTqFtEJXeYafKEEOxcEPt3FiAoOOCM9ATX
VU3tH+rHqYZxtzVFbRd+vprsdHpd9dyqovNKubW7mbfPF53mxwghgp3khSD5YyqZILgkDcWqy
LpbHTZrnUJFght/MZ5JDgLGpOGJ3N/CAck5PUgHgYvhLU7m80m8kupWvb2zlktmaNgvniNm2s
FJCqWyRnjOOTxxz/jXWLifxLYaXPoGtahoCIt3K2nWhmS6lzlI2OQuwYDEZOTtzgZzrSjGrJW
7X+RNWEoSaa2f9f0zY8G+Im8T6PqGpW9ggubW8ure2+0M6F1LB1DFgzRg5QEDIG3gYAUa2u6/
p3hTTGurx5GBcssKuGlfc43bQzDIG7OM4A4HYVwvwl1xDDr9odC1GwRL+4uiBat5MQ+UeSuAC
XX+4FBwBx2q/8AGJHbwvaMqkqt4NxA6fI1XjoqhN8q7foPCU1WrKE31aNnQ/iDo2u6p/ZsaXV
rdlQyR3UYUvldwxgn+Eg84yCMZrS/tJdOura1nH2ayjs7mWSW7l3MqwtGodnLH5SrFiWOcYzg
5FeTQXlvd/EXwuYLiKbbbWqMUbOGCjI+ozyO1eo6syzeIYrZHTzv7OuIgjXLW5Z5SpjVXX5gS
IJTlASoQn0zjGTcbef5G+IoQpzVuqv6GrfwXs0Z+xXv2eTy3X5ow4JI+VuehBx6jBYEHII8lv
PBXizVoE0660iwikF2ZH1OIwxq64xyiAMecnJGeegr1Pw3LJP4X0iaaRpJZLKFndzlmJQEknu
a06xqUI1XeX9df6sebiKCq+7J2tfYpW+mRW8Onp5k5NlGI4yszKrfLt+ZQdrcdNwODyMVhaj4
FtL/AFS6vItS1LTlugpmj0+4MQlcE5Z+oORgYwO+c7q6SSztpYriKS3heO5BE6MgIlyoU7h/F
8oA57DFTV0SjzJSl5m9NujpDQzdNtLfRLOHS7aAhUSR4lhiYLtDDgsSRu+YdWBY7iBgHFmeGN
poy9u84kZQ2WBSPZl1cqxwPmwMqCclewyLNYXhj/iZ+BNG+3/6X9q0yDz/AD/n83dEN27P3s5
Oc9c1UYX1+X5icmSWGgafo0N6Astwl7qBvXWVBJtmZ1IIAXgKwUgn7uMk8ZqDxfp+sX+gzQ6N
ND9oLhmhuIo3SVAOUw6kdcHnv3Aq5qsskeo6IqSOqyXrK4VsBh9nmOD6jIB+oFalZzbqaSY6d
TknzJapnlmg+DtbuvFWm6tqOlwaTDZRR7o43iPmuoOSFj+Vcnk/XjNdlDCqeL727S3YyQaTbo
kCLHuIMkx2gnofkAxuC+vQEdDRUqCW39XNqmJlUfvLpY//2Q==" />
    <script src="index_bundle.c3a064486c8318e5e11a.js"></script>
  </body>
</html>
```

显然，`html-webpack-plugin` 成功地将 chunk 加载了，又处理好了转化为 DataUrl 格式的图片，这一切，都是我们手工难以完成的事情。

4.3.3 还未结束

至此，我们实现了使用 webpack 生成 HTML 页面并尝到了它所带来的甜头，但我们尚未实现**对多个页面共有的部分实现复用**，下一篇《构建一个简单的模板布局系统》我们就来介绍这部分的内容。

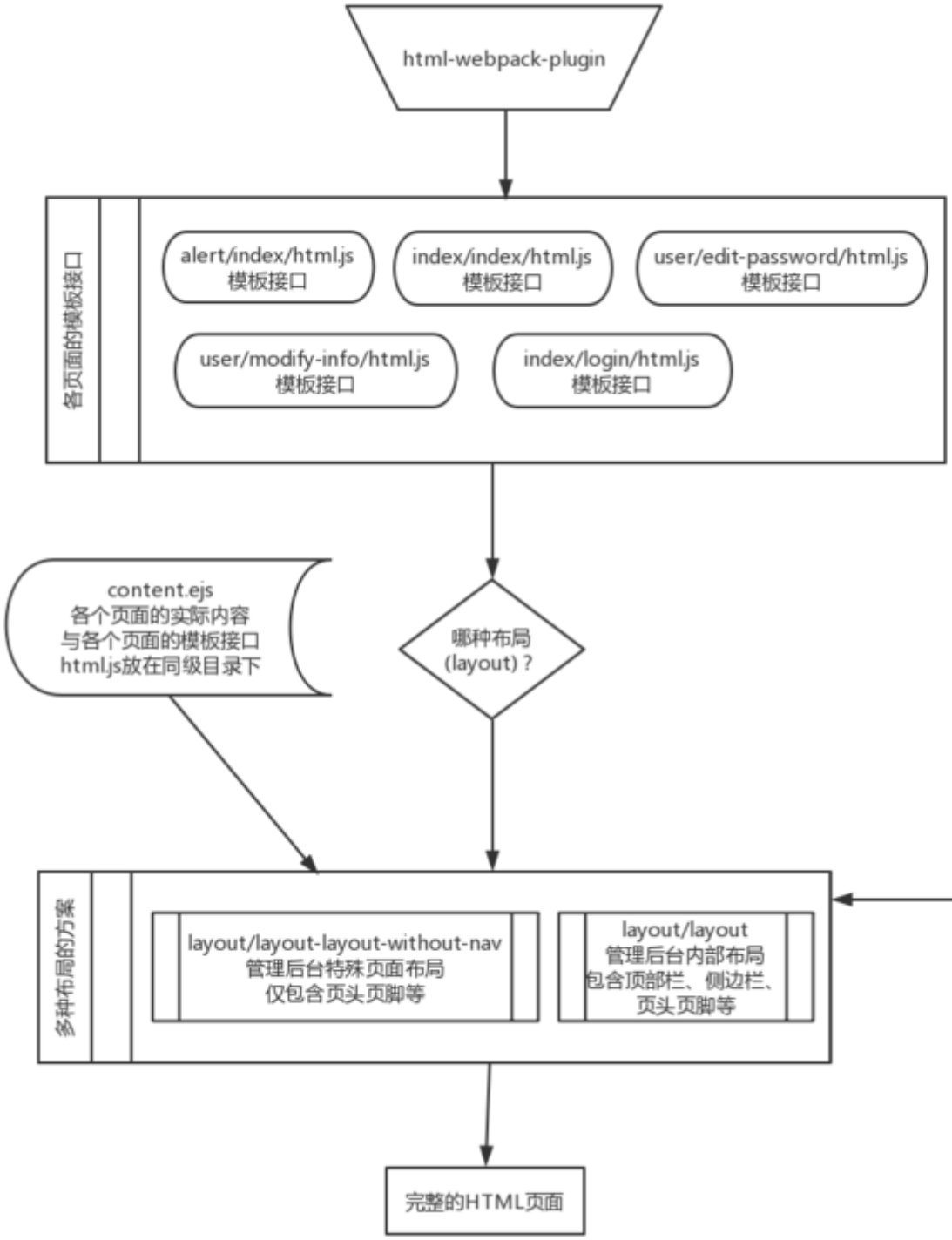
4.4 构建一个简单的模板布局系统

4.4.1 前言

上文《利用 webpack 生成 HTML 普通网页&页面模板》我们基本上已经搞清楚如何利用 `html-webpack-plugin` 来生成 HTML 普通网页&页面模板，本文将以

我的脚手架项目 `Array-Huang/webpack-seed` 介绍如何在这基础上搭建一套简单的模板布局系统。

4.4.2 模板布局系统架构图



4.4.3 模板布局系统各部分详解

上文我们说到，利用模板引擎&模板文件，我们可以控制 HTML 的内容，但这种控制总体来说还是比较有限的，而且很大程度受限于你对该模板引擎的熟悉程度，那么，有没有更简单的方法呢？

有！我们可以就用我们最熟悉的 **js** 来肆意组装、拼接出我们想要的 HTML！

首先来看一个上文提到的例子：

```
var HtmlWebpackPlugin = require('html-webpack-plugin');
var webpackConfig = {
  entry: 'index.js',
  output: {
    path: 'dist',
    filename: 'index_bundle.js'
  },
  plugins: [new HtmlWebpackPlugin(
    title: '按照 ejs 模板生成出来的页面',
    filename: 'index.html',
    template: 'index.ejs',
  )],
};
```

这个例子是给 `html-webpack-plugin` 指定一个名为 `index.ejs` 的 ejs 模板文件，来达到生成 HTML 页面文件的目的，从 `html-webpack-plugin` 的文档我们可以看出，除了默认支持的 ejs 外，其实还可以使用其它模板引擎（例如 `jade`、`handlebars`、`underscore`）,支持的方法是在 webpack 配置文件中配置好相应的 loader 即可。

因此，我们可以推理出，`html-webpack-plugin` 其实并不关心你用的是什么模板引擎，只要你的模板最后 export 出来的是一份完整的 HTML 代码（字符串）就可以了。于是，我做了一个大胆的尝试，给 `html-webpack-plugin` 的 `template` 参数指定一个 js 文件，然后在此 js 文件末尾 export 出一份完整的 HTML 代码来。这个 js 文件我命名为**“模板接口”**（上面架构图上有标识），意思是，不是光靠这一个 js 文件就能形成一份模板，“接口”之后是一套完整的**模板布局体系**。下面以 webpack-seed 项目里的 `src/pages/alert/index`（“消息通知”页）作为例子进行说明。

4.4.3.1html-webpack-plugin 配置

先来看看我是如何给 `html-webpack-plugin` 指定一个 js 作为模板的：

```
/*
  这是用来生成 alert/index 页的 HtmlWebpackPlugin 配置
  在原项目中是循环批量 new HtmlWebpackPlugin 的，此处为了更容易理解，特别针对
alert/index 页做了修改
```

```
*/
new HtmlWebpackPlugin({
    filename: `alert/index/page.html`,
    template: path.resolve(dirVars.pagesDir, `./alert/index/html.js`), // 指定为一个 js 文件而非普通的模板文件
    chunks: ['alert/index', 'commons'], // 自动加载上 index/login 的入口文件以及公共 chunk
    hash: true, // 为静态资源生成 hash 值
    xhtml: true,  // 需要符合 xhtml 的标准
});
```

4.4.3.2 模板接口

下面来介绍这个作为模板接口的 js 文件：

```
/* 选自 webpack-seed/pages/alert/index/html.js  */
const content = require('./content.ejs');  // 调取存放本页面实际内容的模板文件
const layout = require('layout');  // 调用管理后台内部所使用的布局方案，我在 webpack 配置里定义其别名为'layout'
const pageTitle = '消息通知'; // 页面名称

// 给 layout 传入“页面名称”这一参数（当然有需要的话也可以传入其它参数），同时也传入页面实际内容的 HTML 字符串。content({ pageTitle })的意思就是把 pageTitle 作为模板变量传给 ejs 模板引擎并返回最终生成的 HTML 字符串。
module.exports = layout.init({ pageTitle }).run(content({ pageTitle }));
```

从代码里我们可以看出，模板接口的作用实际上就是整理好当前页面独有的内容，然后交与 layout 作进一步的渲染；另一方面，模板接口直接把 layout 最终返回的结果（完整的 HTML 文档）给 export 出来，供 `html-webpack-plugin` 生成 HTML 文件使用。

4.4.3.3 页面实际内容长啥样？

```
<!-- 选自 webpack-seed/pages/alert/index/content.ejs -->
<div id="page-wrapper">
  <div class="container-fluid" >
    <h2 class="page-header"><%= pageTitle %></h2>
    <!-- ...... -->
  </div>
</div>
```

4.4.3.4layout

接着我们来看看整套模板布局系统的核心——layout。layout 的主要功能就是接收各个页面独有的参数（比如说页面名称），并将这些参数传入各个公共组件生成各组件的 HTML，然后根据 layout 本身的模板文件将各组件的 HTML

以及页面实际内容的 HTML 拼接在一起，最终形成一个完整的 HTML 页面文档。

```
/* 选自 webpack-seed/src/public-resource/layout/layout/html.js */
const config = require('configModule');
const noJquery = require('withoutJqueryModule');
const layout = require('./html.ejs'); // 整个页面布局的模板文件，主要是用来统筹各个公共组件的结构
const header = require('../../components/header/html.ejs'); // 页头的模板
const footer = require('../../components/footer/html.ejs'); // 页脚的模板
const topNav = require('../../components/top-nav/html.ejs'); // 顶部栏的模板
const sideMenu = require('../../components/side-menu/html.ejs'); // 侧边栏的模板
const dirsConfig = config.DIRS;

/* 整理渲染公共部分所用到的模板变量 */
const pf = {
  pageTitle: '',
  constructInsideUrl: noJquery.constructInsideUrl,
};

const moduleExports = {
  /* 处理各个页面传入而又需要在公共区域用到的参数 */
  init({ pageTitle }) {
    pf.pageTitle = pageTitle; // 比如说页面名称，会在<title>或面包屑里用到
    return this;
  },

  /* 整合各公共组件和页面实际内容，最后生成完整的 HTML 文档 */
  run(content) {
    const headerRenderData = Object.assign(dirsConfig, pf); // 页头组件需要加载 css/js 等，因此需要比较多的变量
    const renderData = {
      header: header(headerRenderData),
      footer: footer(),
      topNav: topNav(pf),
      sideMenu: sideMenu(pf),
      content,
    };
    return layout(renderData);
  },
};

module.exports = moduleExports;
```

接下来看看 layout 本身的模板文件长啥样吧：

```
<!-- 选自 webpack-seed/src/public-resource/layout/layout/html.ejs -->
<%= header %>
<div id="wrapper">
  <%= topNav %>
  <%= sideMenu %>
  <%= content %>
</div>
<%= footer %>
```

4.4.3.5 组件

整个页面的公共部分，被我以区域的形式切分成一个一个的组件，下面以页头组件作为例子进行解释：

```
<!DOCTYPE html>
<html lang="zh-cmn-Hans">
<head>
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <title><% if (pageTitle) { %> <%= pageTitle %> - <% } %> XXXX 后台</title>
  <meta name="viewport"
content="width=device-width,initial-scale=1,maximum-scale=1" />
  <meta name="renderer" content="webkit" />

  <link rel="stylesheet" type="text/css" href="<%= BUILD_FILE.dll.css %>">
  <script type="text/javascript" src="<%= BUILD_FILE.dll.js %>"></script>
  <!--[if lt IE 10]>
    <script src="<%= BUILD_FILE.js.xdomain %>" slave="<%=
SERVER_API_URL %>cors-proxy.html"></script>
    <script src="<%= BUILD_FILE.js.html5shiv %>"></script>
  <![endif]-->
</head>
<body>
  <!--[if lt IE 9]>
    <script src="<%= BUILD_FILE.js.respond %>"></script>
  <![endif]-->
```

页头组件控制的范围基本上就是整个`<head>`以及`<body>`的头部。
不要小看这`<body>`的头部，由于 webpack 在使用 `extract-text-webpack-plugin` 生成 CSS 文件并自动加载时，会把`<link>`放在`<head>`的最后，而众所周知，实现 IE8 下 **Media Queries** 特性的 `respond.js` 是需要放在 css 后面来加载的，因此，我们就只能把 `respond.js` 放到`<body>`的头部来加载了。

由于我的脚手架项目还是比较简单的，所以这些公共组件的 HTML 都是直接根据模板文件来输出的；如果组件本身要处理的逻辑比较多，可以使用跟**模板接口**一样的思路，利用 js 文件来拼接。

至于组件本身行为的逻辑(js)，可以一并放到各组件的目录里，在公共 chunk 里调用便是了。本文实际上只关注于如何生成 HTML，这里提到这个只是提示一下组件的文件目录结构。

这里稍微再解释一下 `BUILD_FILE.js.*`和 `BUILD_FILE.dll.*`是什么，这些其实都是没有用 webpack 打包起来的 js/css，我用 file-loader 把这些文件从 src 目录搬到 build 目录了，这里模板变量输出的都是搬运后的路径，具体请看《听说 webpack 连图片和字体也能打包？》。启动搬运的代码放在 `webpack-seed/src/public-resource/config/build-file.config.js`。

4.4.4 总结

有了这套模板布局系统，我们就可以轻松地生成具有相同布局的多个静态页面了，**如何管理页面布局公共部分**这一多页应用的痛点也就顺利解决了。

4.5No 复制粘贴！多项目共用基础设施

4.5.1 前言

本文介绍如何在多项目间共用同一套**基础设施**，又或是某种层次的**框架**。

4.5.1.1 基础设施是什么？

一个完整的网站，不可能只包含一个 jQuery，或是某个 MVVM 框架，其中必定包含了许多解决方案，例如：如何上传？如何兼容 IE？如何跨域？如何使用本地存储？如何做用户信息反馈？又或者具体到如何选择日期？等等等等……这里面必定包含了 UI 框架、JS 框架、各种小工具库，不论是第三方的还是自己团队研发的。而以上所述的种种，就构成了一套完整的解决方案，也称**基础设施**。

基础设施有个重要的特征，那就是与业务逻辑无关，不论是 OA 还是 CMS 又或是 CRM，只要整体产品形态类似，我们就可以使用同一套基础设施。

4.5.1.2 框架

框架这个概念很泛，泛得让人心生困惑，但抽象出来说，框架就是一套定义代码在哪里写、怎么写的规则。不能说我们要怎么去**用**框架，反倒是框架**控制**我们怎么去**填**代码。

本系列前面的十来篇文章，分开来看是不同的**点**，但如果所有文章合起来，并连同示例项目（Array-Huang/webpack-seed），实际上阐述的就是一套完整的多页应用框架（或称架构）。这套框架规定了整个应用的方方面面，举几个例子：

- 每个页面的文件放在哪个目录？
- 页面的 HTML、入口文件、css、图片等等应该怎么放？
- 编码规范（由 ESLint 来保证）。

当然，这只是我的框架，我希望你们可以看懂了，然后根据自己的需求来调整，变成**你们的框架**。甚至说，我自己在做不同类型的项目时，整体架构也都会有不少的变化。

4.5.2 为什么要共用基础设施/框架/架构？

4.5.2.1 缘起

数月前，我找同事要了一个他自己写的地区选择器，拉回来一看遍地都是 ESLint 的报错（他负责的项目没有用 ESLint，比较随意），我这人有强迫症的怎么看得过眼，卷起袖子就开始改，改好也就正常使用了。过了一段时间，来了新需求，同事在他那改好了地区选择器又发了一份给我，我一看头都大了，又是满地报错，这不是又要我再改一遍吗？当时我就懵了，只好按着他的思路，对我的版本做了修改。从此，也确立了我们公司会有两份外观功能都一致，但是实现却不一样的地区选择器。

很坑爹是吧？

4.5.2.2 多项目共享架构变动

上面说的是组件级的，下面我们来说架构级别的。

我在公司主要负责的项目有两个，在我的不懈努力下，已经做到跟我的脚手架项目 Array-Huang/webpack-seed 大体上同构了。但维持同构显然是要付出代价的，我在脚手架项目试验过的改进，小至改个目录路径，大至引入个 plugin 啊 loader 啊什么的，都要分别在公司的两个项目里各做一遍，超烦哒（嫌弃脸

试想只是两个项目就已经这样了，如果是三个、四个，甚至六个、七个呢？堪忧啊堪忧啊！

4.5.2.3 快速创建新项目

不知道你们有没有这样子的经验：接到新项目时，灵机一动“这不就是我的 XX 项目吗？”，然后赶紧搬出 XX 项目的源码，然后删掉业务逻辑，保留可复用的基础设施。

也许你会说，这不已经比从零开始要好多了吗？总体上来说，是吧，但还不够好：

- 你需要花时间重温整个项目的架构，搞清楚哪些要删、哪些要留。
- 毕竟是快刀斩乱麻，清理好的架构比不上原先的思路那么清晰。
- 清理完代码想着跑跑看，结果一大堆报错，一个一个来调烦的要命，而且还很可能是删错了什么了不得的东西，还要去原先额项目里搬回来。

以上这些问题，你每创建一个新项目都要经历一遍，我问你怕了没有。

4.5.2.3.1 脚手架不是可以帮助快速创建新项目吗？

是的没错，脚手架本身就算是一整套基础设施了，但依然有下列问题：

- 维护一套脚手架你知道有多麻烦吗？公司项目一忙起来，加班都做不完，哪顾得上脚手架啊。最后新建项目的时候发现脚手架已经落后 N 多了，你到底是用呢还是不用呢？
- 甭跟我提 Github 上开源的脚手架，像我这么有个性的人，会直接用那些妖艳贱货吗？
- 不同类型的项目技术选型不一样，比如说：需不需要兼容低版本 IE；是 web 版的还是 Hybrid App 的；是前台还是后台。每一套技术选型就是一套脚手架，难道你要维护这么多套脚手架吗？

4.5.2.4 上述问题，通过共用基础设施，都能解决

- 既然共用了基础设施，要怎么改肯定都是所有项目一起共享的了，不论是组件层面的还是架构本身。
- 假设你每个不同类型的项目都已经准备好了与其它项目共用基础设施，那么，你根本不需要花费多余的维护成本，创建新项目的时候看准了跟之前哪个项目是属于同一类型的，凑一脚就行了呗，轻松。

4.5.3 怎么实现多项目共用一套基础设施呢？

4.5.3.1 示例项目

在之前的文章里，我使用的一直都是 Array-Huang/webpack-seed 这个脚手架项目作为示例，而为了实践**多项目共用基础设施**，我对该项目的架构做了较大幅度的调整，升级为 **2.0.0** 版本。为免大家看前面的文章时发现示例项目货不对板，感到困惑，我新开了一个 repo 来存放调整后的脚手架：Array-Huang/webpack-seed-v2，并且，我在两个项目的 README 里我都注明了相应的内容，大家可不要混淆了哈。

下面就以从 Array-Huang/webpack-seed 到 Array-Huang/webpack-seed-v2 的改造过程来介绍如何实现多项目共用基础设施。

4.5.3.2 改造思路

改造思路其实很简单，就是**把预想中多个项目都能用得上的部分从现有项目里抽离出来**。

4.5.3.3 如何抽离

抽离的说法是针对原项目的，如果单纯从文件系统的角度来说，只不过是**移动**了某些文件和目录。

移动到哪里了呢？自然是移动到与项目目录同级的地方，这样就方便多个项目引用这个**核心**了。

如果你跟我一样，在原项目中定义了大量路径和 alias 的话，移动这些文件/目录就只是个改变量的活了：

选自 `webpack-seed/webpack-config/base/dir-vars.config.js`:

```
var path = require('path');
var moduleExports = {};

// 源文件目录
moduleExports.staticRootDir = path.resolve(__dirname, '../../'); // 项目根目录
moduleExports.srcRootDir = path.resolve(moduleExports.staticRootDir, './src'); // 项目业务代码根目录
moduleExports.vendorDir = path.resolve(moduleExports.staticRootDir, './vendor'); // 存放所有不能用 npm 管理的第三方库
```

```
moduleExports.dllDir = path.resolve(moduleExports.srcRootDir, './dll'); //
存放由各种不常改变的 js/css 打包而来的 dll
moduleExports.pagesDir = path.resolve(moduleExports.srcRootDir, './pages');
// 存放各个页面独有的部分，如入口文件、只有该页面使用到的 css、模板文件等
moduleExports.publicDir = path.resolve(moduleExports.srcRootDir,
'./public-resource'); // 存放各个页面使用到的公共资源
moduleExports.logicDir = path.resolve(moduleExports.publicDir, './logic');
// 存放公用的业务逻辑
moduleExports.libsDir = path.resolve(moduleExports.publicDir, './libs');
// 与业务逻辑无关的库都可以放到这里
moduleExports.configDir = path.resolve(moduleExports.publicDir, './config');
// 存放各种配置文件
moduleExports.componentsDir = path.resolve(moduleExports.publicDir,
'./components'); // 存放组件，可以是纯 HTML，也可以包含 js/css/image 等，看自己
需要
moduleExports.layoutDir = path.resolve(moduleExports.publicDir, './layout');
// 存放 UI 布局，组织各个组件拼起来，因应需要可以有不同的布局套路

// 生成文件目录
moduleExports.buildDir = path.resolve(moduleExports.staticRootDir,
'./build'); // 存放编译后生成的所有代码、资源（图片、字体等，虽然只是简单的从源
目录迁移过来）

module.exports = moduleExports;
```

选自 webpack-seed/webpack-config/resolve.config.js:

```
var path = require('path');
var dirVars = require('./base/dir-vars.config.js');
module.exports = {
  // 模块别名的配置，为了使用方便，一般来说所有模块都是要配置一下别名的
  alias: {
    /* 各种目录 */
    iconfontDir: path.resolve(dirVars.publicDir, 'iconfont/'),
    configDir: dirVars.configDir,

    /* vendor */
    /* bootstrap 相关 */
    metisMenu: path.resolve(dirVars.vendorDir, 'metisMenu/'),

    /* libs */
    withoutJqueryModule: path.resolve(dirVars.libsDir,
'without-jquery.module'),
    routerModule: path.resolve(dirVars.libsDir, 'router.module'),

    libs: path.resolve(dirVars.libsDir, 'libs.module'),
```

```
    /* less */
    lessDir: path.resolve(dirVars.publicDir, 'less'),

    /* components */

    /* layout */
    layout: path.resolve(dirVars.layoutDir, 'layout/html'),
    'layout-without-nav': path.resolve(dirVars.layoutDir,
'layout-without-nav/html'),

    /* logic */
    cm: path.resolve(dirVars.logicDir, 'common.module'),
    cp: path.resolve(dirVars.logicDir, 'common.page'),

    /* config */
    configModule: path.resolve(dirVars.configDir, 'common.config'),
    bootstrapConfig: path.resolve(dirVars.configDir, 'bootstrap.config'),
  },

  // 当 require 的模块找不到时，尝试添加这些后缀后进行寻找
  extentions: ['', 'js'],
};
```

4.5.3.4 抽离对象

抽离的方法很简单，那么关键就看到底是哪些部分可以抽离、需要抽离了，这一点看我抽离后的成果就比较清晰了：

先来看根目录：

```
├─ core # 抽离出来的基础设施，或称“核心”
├─ example-admin-1 # 示例项目 1，被抽离后剩下的
├─ example-admin-2 # 示例项目 2，嗯，简单起见，直接复制了 example-admin-1，不过
还是要做一点调整的，比如说配置
├─ npm-scripts # 没想到 npm-scripts 也能公用吧？
├─ vendor # 无法在 npm 上找到的第三方库
├─ .eslintrc # ESLint 的配置文件
├─ package.json # 所有的 npm 库依赖建议都写到这里，不建议写到具体项目的
package.json 里
```

再来看看 `core` 目录

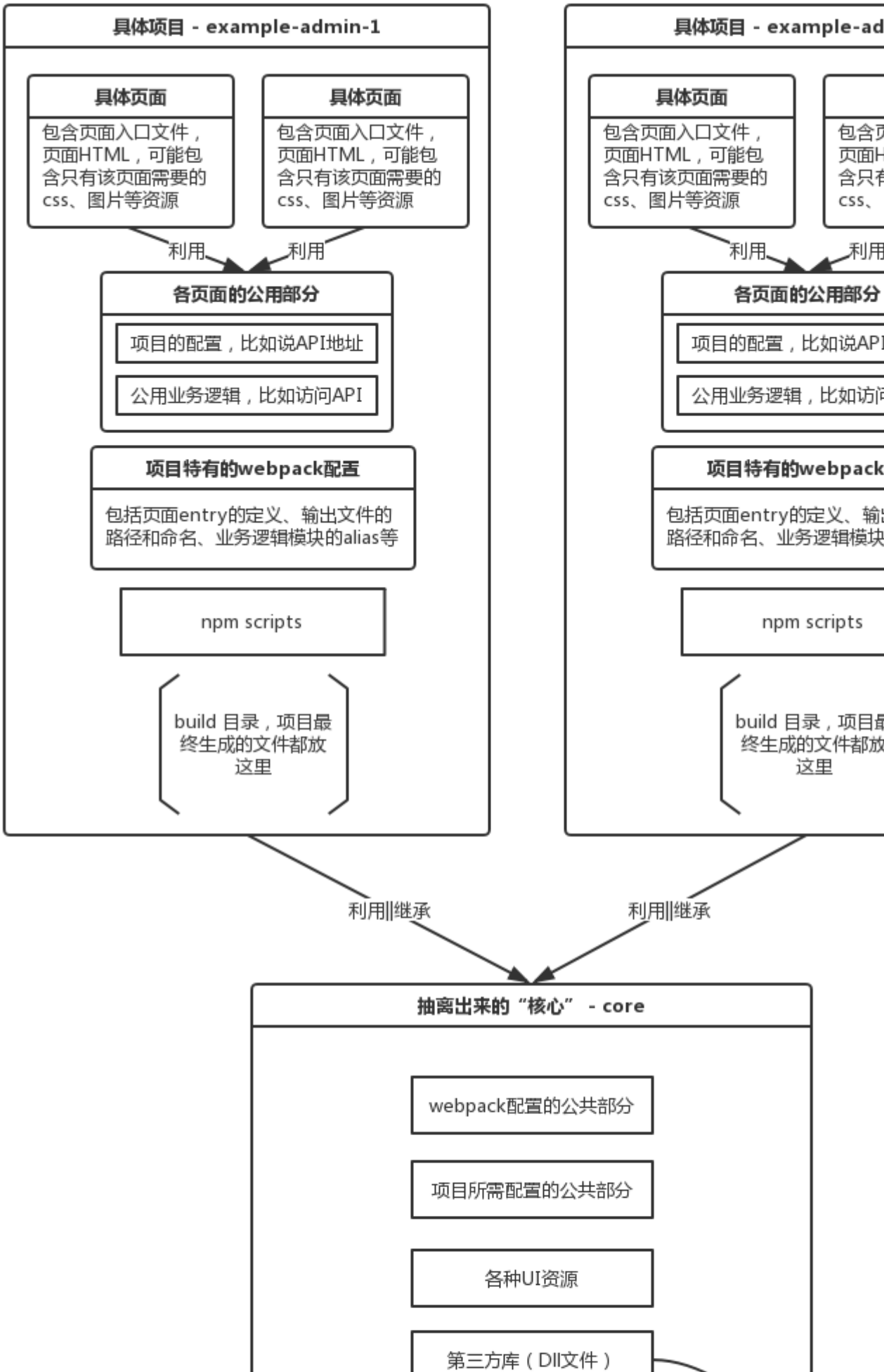
```
├─ _webpack.dev.config.js # 整理好公用的开发环境 webpack 配置，以备继承
├─ _webpack.product.config.js # 整理好公用的生产环境 webpack 配置，以备继承
├─ webpack-dll.config.js # 用来编译 Dll 文件用的 webpack 配置文件
```

```
├─ manifest.json # Dll 文件的资源目录
├─ package.json # 没有什么实质内容，我这里就放了个编译 Dll 用的 npm script
├─components # 各种 UI 组件
│  ├─footer
│  ├─header
│  ├─side-menu
│  └─top-nav
├─config # 公共配置，有些是提供给具体项目的配置来继承的，有些本身就有用（比如说“核心”部分本身需要的配置）
├─dll # 之前的文章里就说过，我建议把各种第三方库（包括 npm 库也包括非 npm 库）都打包成 Dll 来加速 webpack 编译过程，这部分明显就属于基础设施了
├─iconfont # 字体图标能不能公用，这点我也是比较犹豫的，看项目实际需要吧，不折腾的话还是推荐公用
├─layout # 布局，既然是同类型项目，布局肯定是基本一样的
│  ├─layout
│  └─layout-without-nav
├─less # 样式基础，在我这项目里就是针对 bootstrap 的 SB-Admin 主题做了修改
│  ├─base-dir
│  └─components-dir
├─libs # 自己团队研发的一些公共的方法/库，又或是针对第三方库的适配器（比如说对 alert 库封装一层，后面要更换库的时候就方便了）
├─npm-scripts # 与根目录下的 npm-scripts 目录不一样，这里的不是用来公用的，而是“核心”使用到的 script，比如我在这里就放了编译 dll 的 npm script
└─webpack-config # 公用的 webpack 配置，尤其是关系到“核心”部分的配置，比如说各第三方库的 alias。这里的配置是用来给具体项目来继承的，老实说我现在继承的方法也比较复杂，回头看看有没有更简单的方法。
    ├─base
    ├─inherit
    └─vendor
```

最后总结一下，是哪些资源被抽离出来了：

- webpack 配置中属于架构的部分，比如说各种 loader、plugin、“核心”部分的 alias。
- “核心”部分所需的配置，比如我这项目里为了定制 bootstrap 而建的配置。
- 各种与 UI 相关的资源，比如 UI 框架/样式、UI 组件、字体图标。
- 第三方库，以 Dll 文件的形式存在。
- 自研库/适配器。

4.5.4 结构图


具体项目 - example-admin-1
具体页面
包含页面入口文件，页面HTML，可能包含只有该页面需要的css、图片等资源
具体页面
包含页面入口文件，页面HTML，可能包含只有该页面需要的css、图片等资源
利用
利用
各页面的公用部分
项目的配置，比如说API地址
公用业务逻辑，比如访问API
项目特有的webpack配置
包括页面entry的定义、输出文件的路径和命名、业务逻辑模块的alias等
npm scripts
build 目录，项目最终生成的文件都放这里
具体项目 - example-ad
具体页面
包含页面入口文件，页面HTML，可能包含只有该页面需要的css、图片等资源
利用
利用
各页面的公用部分
项目特有的webpack
npm scripts
build 目录，项目最终生成的文件都放这里
利用||继承
利用||继承
抽离出来的“核心” - core
webpack配置的公共部分
项目所需配置的公共部分
各种UI资源
第三方库（Dll文件）

4.6 论前端如何在后端渲染开发模式下夹缝生存

4.6.1 前言

近年来前端领域发展迅猛，**前后端分离**早已成为业界共识，各类管控系统(to B)上个 SPA 什么的也不值一提，但唯独**偏展示类的项目**，为了 SEO，始终还是需要依赖于服务器端渲染 html。

我过往也曾尝试为 SPA 弥补 SEO，但现在看来，效果虽然达到了，但工作量也大大地增加（因为后端用的 PHP，不能做到前后同构）。

虽然无法改变*依赖服务器端渲染*这一现实，但我们可以去勇敢地拥抱它，用前端的坚船利炮(aka webpack)，把服务器端模板层给啃下来！

4.6.2 前导知识

- webpack 的基本使用
- 利用 webpack 生成 HTML 文件，及，构建模板布局系统
- 你所在项目后端模板引擎（如我目前使用的 Slim 框架，模板引擎仅为原生 PHP）

4.6.3 两个阶段

整个前端项目，以本文主题的视角来看，可以分为两个阶段：

4.6.3.1 纯静态页面开发阶段

在这个阶段里，一切开发都跟静态网站无二致，按 UI 稿切好页面搞好交互，要用到 ajax 请求 API 的也尽管写，跟后端的协作点仅在于 API 文档。

传统前端的工作也就到这里为止了，但对我们来说，目前的成果并不是我们最终的交付；因此，注意了，在这个阶段我们是可以**“偷懒”**的，比如说，一些明显应该由服务器端循环生成的部分（商品列表、文章列表等），我们写一遍就 OK 了。

4.6.3.2 动态页面改造阶段

这就是所谓的**“套页面”**，传统来说是由后端来做的，实际上后端也是苦不堪言，毕竟模板不是自己写的，有时还是需要改造一番，而这正是我们前端要大力争取的活。

在这个阶段里，我们的主要工作是按照后端模板引擎的规则来撰写模板变量占位符，当然这里面也不会少了循环输出和逻辑判断，另外也可能需要用到后端定义的一些函数，视项目需求而定。

4.6.3.3 在两个阶段里来回往返

这两个阶段不一定是完全独立的，有需要的话也是可以做到来回往返的。

那什么时候才叫做*“有需要”*呢？举个例子，当你把原先的静态页面都改造成需要后端渲染的页面模板后，却发现后端此时并未准备好相应的模板变量，而你此时又需要对页面的 UI 部分进行修改，那么你就很被动了，因为改好的这些页面模板根本跑不起来了。有两种解决方案：

- 参考 `API mock` 的思路，来个`模板变量 mock`，这就相当于一直留在**动态页面改造**阶段了。
- 回到**纯静态页面开发**阶段，让页面不需要后端渲染也能跑起来。具体怎么做呢？

1. 区分开两个阶段，使用不同的 webpack 配置。
2. 在我们构建生成页面的**前端模板**（注意分清与后端模板的区别），判断（判断依据看这里）本次执行 webpack 打包是在哪个“阶段”，继而选择是生成静态（且完整）的 element，还是带有模板变量占位符的 element。这样一来，我们就可以随时选择在不同的阶段（或称环境）里进行开发了。

4.6.4 改造开始

本文着重介绍如何将静态页面改造成后端渲染需要的模板。

4.6.4.1 配合后端模板命名规则生成相应模板文件

不同项目因应本身所使用的后端框架或是其它需求，对模板放置的目录结构也会有所不一样，那么，如何构建后端所需要的目录结构呢？

在静态网页阶段，我习惯把 html/css/js 都按照所属页面归到各自的目录中（公用的 css/js 也当然是放到公用目录中），看 HtmlWebpackPlugin 配置：

```
pageArr.forEach((page) => {
  const htmlPlugin = new HtmlWebpackPlugin({
    filename: `${page}/index.html`, // page 变量形如'product/index'、
'product/detail'
    template: path.resolve(dirVars.pagesDir, `./${page}/html.js`),
    chunks: [page, 'commons/commons'],
    hash: true,
    xhtml: true,
  });
  pluginsConfig.push(htmlPlugin);
```

```
});
```

而在改造阶段，则放到后端指定位置：

```
pageArr.forEach((page) => {
  const htmlPlugin = new HtmlWebpackPlugin({
    filename: `../../view/frontend/${page}.php`, // 通过控制相对路径来确定模板的根目录
    template: path.resolve(dirVars.pagesDir, `./${page}/html.js`),
    chunks: [page, 'commons/commons'],
    hash: true,
    xhtml: true,
  });
  pluginsConfig.push(htmlPlugin);
});
```

此时我模板目录结构是这样的：

```
│
├─alert
│      index.php
│
├─article
│      detail.php
│      index.php
│
├─index
│      index.php
│
├─product
│      detail.php
│      index.php
│
└─user
        edit-password.php
        modify-info.php
```

这里需要注意的是，我的前端项目目录实际上是作为后端目录里的一个子目录来存放的，这样才能依靠相对路径来确定模板文件存放的根目录位置。

4.6.4.2 处理站内链接

对于站内链接，我建议在前端模板里使用一个函数来适配两个阶段：

```
{
  /* 拼接系统内部的 URL */
  constructInsideUrl(url, request, urlTail) {
    urlTail = urlTail || '';
```

```
    let finalUrl = config.PAGE_ROOT_PATH + url;
    if (!config.IS_PRODUCTION_MODE) {
      finalUrl += '/index.html' + urlTail;
      return finalUrl;
    }
    return `<?php echo cf::constructInsideUrl(array('module' => '${url}'),
$isStaticize)?>`;
  },
};
```

在前端模板里这么用：

```
<a href="<%= constructInsideUrl('index/index') %>">
  <img src="<%= require('./logo.png') %>">
</a>
```

这样做，就能分别在静态页面阶段和后端渲染阶段生成相应的超链接。再者，在后端渲染阶段，我们生成出来的也不一定是一个完整的 url，可以像我上述代码一样，生成调用后端函数的模板代码，从而灵活满足后端的一些需求（比如说，我的项目有静态化的需求，那么，静态化后的站内链接跟动态渲染的又会有所不同了）。

4.6.4.3 处理模板变量

这一块其实我要说的不多，无非就是按照后端模板引擎的规则，输出变量、循环输出变量、判断条件输出变量、调用后端（模板引擎）函数调整输出变量。

关键是，我们需要拿到一份**模板变量文档**，跟 API 文档类似，它实际上也是一份前后端的数据协议。有了这份文档，我们才能在后端未完工的情况下，进入**动态页面改造阶段**，并根据其中内容实现`模板变量 mock`。

4.6.4.4 争讨模板布局渲染权

关于利用模板布局系统对多个页面共有的部分实现复用，在之前的文章里已经提及了，我设计该系统的思路恰恰是来自于后端模板渲染。那么，在前后端均可以实现**模板布局系统**的前提下，我们应如何抉择呢？我的答案是，前端一定要吃下来！

从前端的角度来看：

- 我们在**纯静态页面开发阶段**的产物就已经是一个个完整的页面了，再要拆开并不现实。
- 由于在 webpack 的辅助下这套**模板布局系统**功能相当强大，因此并没有给整个项目添加额外的成本。

从后端的角度来看：

- 服务器拼接多个 HTML 代码段本身也是有成本（比如磁盘 IO 成本）的，倒不如渲染一个完整的页面。
- 在公共组件的分治管理上不会有很大变化，只不过以前是一个一个组件渲染好后再拼在一起，而现在是把各个组件的数据整合在一起来统一渲染罢了。

4.6.5 总结

在后端渲染的项目里使用 webpack 多页应用架构是绝对可行的，可不要给老顽固们吓唬得又回到传统前端架构了。

4.7 善用浏览器缓存，该去则去，该留则留

4.7.1 前言

一个成熟的项目，自然离不开迭代更新；那么在部署前端这一块，我们免不了总是要顾及到浏览器缓存的，本文将介绍如何在 webpack (架构)的帮助下，妥善处理好浏览器缓存。

实际上，我很早以前就想写这一 part 了，只是苦于当时我所掌握的方案不如人意，便不敢献丑了；而自从 webpack 升级到 v2 版本后，以及第三方 plugin 的日益丰富，我们也有了更多的手段来处理 cache。

4.7.2 浏览器缓存简单介绍

下面来简单介绍一下浏览器缓存，以及为何我要在标题中强调“该去则去，该留则留”。

4.7.2.1 浏览器缓存是啥？

浏览器缓存(Browser Cache)，是浏览器为了节省网络带宽、加快网站访问速度而推出的一项功能。浏览器缓存的运行机制是这样的：

1. 用户使用浏览器第一次访问某网站页面，该页面上引入了各种各样的静态资源（js/css/图片/字体……），浏览器会把这些静态资源，甚至是页面本身(html 文件)，都一一储存到本地。
2. 用户在后续的访问中，如果需要再次请求同样的静态资源（根据 url 进行匹配），且静态资源没有过期（服务器端有一系列判别资源是否过期的策略，比如 `Cache-Control`、`Pragma`、`ETag`、`Expires`、`Last-Modified`），则直接使用前面本地储存的资源，而不需要重复请求。

由于 webpack 只负责构建生成网站前端的静态资源，不涉及服务器，因此本文不讨论以 *HTTP Header* 为基础的缓存控制策略；那我们讨论什么呢？

很简单，由于浏览器是根据静态资源的 **url** 来判断该静态资源是否已有缓存，而静态资源的文件目录又是相对固定的，那么重点明显就在于静态资源的**文件名**了；我们就通过操控静态资源的文件名，来决定静态资源的“去留”。

4.7.2.2 浏览器缓存，该留不留会怎么样？

每次部署上线新版本，静态资源的文件名若有变化，则浏览器判断是第一次读取这个静态资源；那么，即便这个静态资源的内容跟上一版的完全一致，浏览器也要重新下载这个静态资源，浪费网络带宽、拖慢页面加载速度。

4.7.2.3 浏览器缓存，该去不去会怎么样？

每次部署上线新版本，静态资源的文件名若没有变化，则浏览器判断可加载之前缓存下来的静态资源；那么，即便这个静态资源的内容跟上一版的有所变化，浏览器也察觉不到，使用了老版本的静态资源。那这会造成什么样的影响呢？可大可小，小至用户看到的依然是老版的资源，达不到上线更新版本的目的；大至造成网站运行报错、布局错位等问题。

4.7.3 如何通过操控静态资源的文件名达到控制浏览器缓存的目的呢？

在 webpack 关于文件名命名的配置中，存在一系列的变量（或者理解成命名规则也可），通过这些变量，我们可以根据所要生成的文件的具体情况来进行命名，而不必预设好一个固定的名称。在缓存处理这一块，我们主要用到`[hash]`和`[chunkhash]`这两个变量。关于这两个变量的介绍，我在之前的文章——《webpack 配置常用部分有哪些？》就已经解释过是什么意思了，这里就不再累述。
这里总结下`[hash]`和`[chunkhash]`这两个变量的用法：

- 用`[hash]`的话，由于每次使用 webpack 构建代码的时候，此 hash 字符串都会更新，因此相当于**强制刷新浏览器缓存**。
- 用`[chunkhash]`的话，则会根据具体 chunk 的内容来形成一个 hash 字符串来插入到文件名上；换句说， chunk 的内容不变，该 chunk 所对应生成出来的文件的文件名也不会变，由此，**浏览器缓存便能得以继续利用**。

4.7.4 有哪些资源是需要兼顾浏览器缓存的？

理论上来说，除了 HTML 文件外（HTML 文件的路径需要保持相对固定，只能从服务器端入手），webpack 生成的所有文件都需要处理好浏览器缓存的问题。

js

在 webpack 架构下，js 文件也有不同类型，因此也需要不同的配置：

1. 入口文件(Entry)：在 webpack 配置中的 `output.filename` 参数中，让生成的文件名中带上`[chunkhash]`即可。
2. 异步加载的 chunk：`output.chunkFilename` 参数，操作同上。
3. 通过 `CommonsChunkPlugin` 生成的文件：在 `CommonsChunkPlugin` 的配置参数中有 `filename` 这一项，操作同上。但需要注意的是，如果你使用`[chunkhash]`的话，webpack 构建的时候可是会报错的哦；那可咋办呢，用`[hash]`的话，这 `common chunk` 不就每次上线新版本都强制刷新了吗？这其实是因为，webpack 的 runtime && manifest 会统一保存在你的 `common chunk` 里，解决的方法，就请看下面关于“webpack 的 runtime && manifest”的部分了。

css

对于 css 来说，如果你是用 `style-loader` 直接把 css 内联到`<head>`里的，那么，你管好引入该 css 的 js 文件的浏览器缓存就好了。
而如果你是使用 `extract-text-webpack-plugin` 把 css 独立打包成 css 文件的，那么在文件名的配置上，~~同样加上`[chunkhash]`即可~~加上`[contenthash]`即可(感谢@FLYiNg_hbt 提醒)。这个`[contenthash]`是什么东西呢？其实就是 `extract-text-webpack-plugin` 为了与`[chunkhash]`区分开，而自定义的一个命名规则，其实际含义跟`[chunkhash]`可以说是一致的，只是`[chunkhash]`已被占用作为 chunk 的内容 hash 字符串了，继续用`[chunkhash]`会造成“文件改动监测失败”的问题。

图片、字体文件等静态资源

如《听说 webpack 连图片和字体也能打包？》里介绍的，处理这类静态资源一般使用 `url-loader` 或 `file-loader`。
对于 `url-loader` 来说，就不需要关心浏览器缓存了，因为它是把静态资源转化成 dataurl 了，而并非独立的文件。
而对于 `file-loader` 来说，同样是在文件名的配置上加上`[chunkhash]`即可。另外需要注意的是，`url-loader` 一般搭配有降级到 `file-loader` 的配置（使用 loader 加载的文件大于一个你设定的值就降级到使用 `file-loader` 来加载），同样需要在文件名的配置上加上`[chunkhash]`。

webpack 的 runtime && manifest

所谓的 runtime，就是帮助 webpack 编译构建后的打包文件在浏览器运行的一些辅助代码段，换句话说，打包后的文件，除了你自己的源码和 npm 库外，还有 webpack 提供的一点辅助代码段。

而 manifest，则是 webpack 用以查找 chunk 真实路径所使用的一份关系表，简单来说，就是 **chunk 名**对应 **chunk 路径**的关系表。manifest 一般来说会被藏到 runtime 里，因此我们查看 runtime 的时候，虽然能找得到 manifest，但一般都不那么直观，形如下面这一段（仅 common chunk 部分）：

```
u.type = "text/javascript", u.charset = "utf-8", u.async = !0, u.timeout =
12e4, n.nc && u.setAttribute("nonce", n.nc), u.src = n.p + "" + e + "." + {
    0: "e6d1dff43f64d01297d3",
    1: "7ad996b8cbd7556a3e56",
    2: "c55991cf244b3d833c32",
    3: "ecbcdaa771c68c97ac38",
    4: "6565e12e7bad74df24c3",
    5: "9f2774b4601839780fc6"
}[e] + ".bundle.js";
```

runtime && manifest 被打包到哪里去了？

那么，这 runtime && manifest 的代码段，会被放到哪里呢？一般来说，如果没有使用 CommonsChunkPlugin 生成 common chunk，runtime && manifest 会被放在以入口文件为首的 chunk（俗称“大包”）里，如果是我们这种多页（又称多入口）应用，则会每个大包一份 runtime && manifest；这夸张的冗余我们自然是不能忍的，那么 用上 CommonsChunkPlugin 后，runtime && manifest 就会统一迁到 common chunk 了。

runtime && manifest 给 common chunk 带来的缓存危机

虽说把 runtime && manifest 迁到 common chunk 后，代码冗余的问题算是解决了，但却造成另一问题：由于我们在上述的静态资源的文件名命名上都采用了[chunkhash]的方案，因此也使得只要我们稍一改动源代码，就会有起码一个 chunk 的命名会产生变化，这就会导致我们的 runtime && manifest 也产生变化，从而导致我们的 common chunk 也发生变化，这或许就是 webpack 规定含有 runtime && manifest 的 common chunk 不能使用[chunkhash]的原因吧（反正 chunkhash 肯定会变的，还不如不用呢是不是）。
要解决上述问题（这问题很严重啊我摔，common chunk 怎么能用不上缓存啊，这可是最大的 chunk 啊），我们就需要把 runtime && manifest 给独立出去。方法也很简单，在用来打包 common chunk 的 CommonsChunkPlugin 后，再加一 CommonsChunkPlugin：

```
  /* 抽取出所有通用的部分 */
  new webpack.optimize.CommonsChunkPlugin({
    name: 'commons/commons',      // 需要注意的是，chunk 的 name 不能相同！！！
    filename: '[name]/bundle.[chunkhash].js', // 由于 runtime 独立出去了，这里便可以使用[chunkhash]了
    minChunks: 4,
  }),
  /* 抽取出 webpack 的 runtime 代码，避免稍微修改一下入口文件就会改动 commonChunk，导致原本有效的浏览器缓存失效 */
  new webpack.optimize.CommonsChunkPlugin({
    name: 'webpack-runtime',
    filename: 'commons/commons/webpack-runtime.[hash].js', // 注意 runtime 只能用[hash]
  }),
```

这样一来，`runtime && manifest` 代码段就会被打包到这个名为 `webpack-runtime` 的 chunk 里了。这是什么原理呢？据说是在使用 `CommonsChunkPlugin` 的情况下， webpack 会把 `runtime && manifest` 打包到最后面的一个 `CommonsChunkPlugin` 生成的 chunk 里，而如果这个 chunk 没有其它代码，那么自然就达到了把 `runtime && manifest` 独立出去的目的了。
需要注意的是，如果你用了 `html-webpack-plugin` 来生成 html 页面，记得要把这 `runtime && manifest` 的 chunk 插入到 html 页面上，不然页面报错了可不怪我哦。
至此，由于 `runtime && manifest` 独立出去成一个 chunk 了，于是 `common chunk` 的命名便可以使用`[chunkhash]`了，也就是说，`common chunk` 现在也能做到公共模块内容有更新了，才更新文件名；另一方面，这个独立出去的 `runtime && manifest` chunk，是每次 webpack 打包构建的时候都会更新了。

有必要把 manifest 从 `runtime && manifest` chunk 中独立出去吗？

是的，不用惊讶，的确是有这么一个骚操作。

把 manifest 独立出去的理由是这样的：manifest 独立出去后，runtime 的部分基本上就不会有变动了；到这里，我们就知道，`runtime && manifest` 里实际上就是 manifest 在变；因此把 manifest 独立出去，也是进一步地利用浏览器缓存（可以把 runtime 的缓存保留下来）。
具体是怎么做的呢？主流有俩方案：

- 利用 chunk-manifest-webpack-plugin 把 manifest 生成一个 json 文件，然后由 webpack 异步加载。
- 如果你是用 `html-webpack-plugin` 来生成 html 页面的话，还可以利用 inline-chunk-manifest-html-webpack-plugin（`html-webpack-plugin` 作者推荐）来把 manifest 直接输出到 html 页面上，这样就能省一个 Http 请求了。

我试用过第二种方案，好使，但最终还是放弃了，为什么呢？

把 manifest 独立出去后，只剩下 runtime 的 chunk 的命名还是只能用 `[hash]`，而不能利用`[chunkhash]`，这就导致我们根本没法利用浏览器缓存。后来，我又想出一个折衷的办法，连`[hash]`也不要了，直接写死一个文件名；这样的话，的确浏览器缓存就能保存下来了。但后来我还是反转了自己，这种方法虽然能留下浏览器缓存，却做不到“该去则去”。或许大家会有疑问，你不是说 runtime 不会变的吗，那留下缓存有什么关系呀？是的，在同一 webpack 环境下 runtime 的确不会变，但难保 webpack 环境改变后，这 runtime 会怎么样呀。比如说 webpack 的版本升级了、 webpack 的配置改了、loader & plugin 的版本升级了，在这些情况下，谁敢保证 runtime 永远不会变啊？这 runtime 一用错了过期的缓存，那很可能整个系统都会崩溃的啊，这个险我实在是冒不起，所以只能作罢。
不过我看了下 Array-Huang/webpack-seed 的 `runtime && manifest` chunk，也才 2kb 而已嘛，你们管好自己的强迫症和代码洁癖好吗？！

4.7.5 缓存问题杂项

4.7.5.1 模块 id 带来的缓存问题

webpack 处理模块(module)间依赖关系时，需要给各个模块定一个 id 以作标识。webpack 默认的 id 命名规则是根据模块引入的顺序，赋予一个整数(1、2、3……)。当你在源码中任意增添或删减一个模块的依赖，都会对整个 id 序列造成极大的影响，可谓是“牵一发而动全身”了。那么这对我们的浏览器缓存会有什么样直接的影响呢？影响就是会造成，各个 chunk 中都不一定有实质的变化，但引用的依赖模块 id 却都变了，这明显就会造成 chunk 的文件名的变动，从而影响浏览器缓存。

webpack 官方文档里推荐我们使用一个已内置进 webpack2 里的 plugin：`HashedModuleIdsPlugin`，这个 plugin 的官方文档在这里。
webpack1 时代便有一个 `NamedModulesPlugin`，它的原理是直接使用模块的相对路径作为模块的 id，这样只要模块的相对路径，模块 id 也就不会变了。那么这个 `HashedModuleIdsPlugin` 对比起 `NamedModulesPlugin` 来说又有什么进步呢？
是这样的，由于模块的相对路径有可能会很长，那么就会占用大量的空间，这一点是一直为社区所诟病的；但这个 `HashedModuleIdsPlugin` 是根据模块的相对路径生成(默认使用 md5 算法)一个长度可配置（默认截取 4 位）的字符串作为模块的 id，那么它占用的空间就很小了，大家也就可以安心服用了。
To generate identifiers that are preserved over builds, webpack supplies the NamedModulesPlugin (recommended for development) and HashedModuleIdsPlugin (recommended for production).

从上可知，官方是推荐开发环境用 `NamedModulesPlugin`，而生产环境用 `HashedModuleIdsPlugin` 的，原因似乎是与热更新(hmr)有关；不过就我看来，仅

在生产环境用 `HashedModuleIdsPlugin` 就行了，开发环境还管啥浏览器缓存啊，俺开 chrome dev-tool 设置了不用任何浏览器缓存的。
用法也挺简单的，直接加到 `plugin` 参数就成了：

```
plugins: {
  // 其它 plugin
  new webpack.HashedModuleIdsPlugin(),
}
```

4.7.5.2 由某些 plugin 造成的文件改动监测失败

有些 plugin 会生成独立的 chunk 文件，比如 `CommonsChunkPlugin` 或 `ExtractTextPlugin`（从 js 中提取出 css 代码段并生成独立的 css 文件） 。
这些 plugin 在生成 chunk 的文件名时，可能没料想到后续还会有其它 plugin （比如用来混淆代码的 `UglifyJsPlugin`）会对代码进行修改，因此，由此生成的 chunk 文件名，并不能完全反映文件内容的变化。
另外，`ExtractTextPlugin` 有个比较严重的问题，那就是它生成文件名所用的 `[chunkhash]`是直接取自于引用该 css 代码段的 js chunk ；换句话说，如果我只是修改 css 代码段，而不动 js 代码，那么最后生成出来的 css 文件名依然没有变化，这可算是非常严重的浏览器缓存“该去不去”问题了。 2017-07-26 改动：改用`[contenthash]`便不会出现此问题，上见 ***css 部分***。
有一款 plugin 能解决以上问题：webpack-plugin-hash-output。

There are other webpack plugins for hashing out there. But when they run, they don't "see" the final form of the code, because they run before plugins like webpack.optimize.UglifyJsPlugin. In other words, if you change webpack.optimize.UglifyJsPlugin config, your hashes won't change, creating potential conflicts with cached resources.

The main difference is that webpack-plugin-hash-output runs in the last compilation step. So any change in webpack or any other plugin that actually changes the output, will be "seen" by this plugin, and therefore that change will be reflected in the hash.

简单来说，就是这个 `webpack-plugin-hash-output` 会在 webpack 编译的最后阶段，重新对所有的文件取文件内容的 md5 值，这就保证了文件内容的变化一定会反映在文件名上了。
用法也比较简单：

```
plugins: {
  // 其它 plugin
  new HashOutput({
    manifestFiles: 'webpack-runtime', // 指定包含 manifest 在内的 chunk
  }),
}
```

4.7.6 总结

浏览器缓存很重要，很重要，很重要，出问题了怕不是要给领导追着打。另外，这一块的细节特别多，必须方方面面都顾到，不然哪一方面出了纰漏就全局泡汤。