

webpack 4.12.0

2018 年 6 月 19 日 15:06:49

目录

目录	1
序言	1
一、磨刀霍霍.....	1
1.1 入口(entry).....	1
1.2 出口(output)	2
1.3 解析器(loader).....	2
1.4 插件(plugins).....	3
1.5 配置(webpack.config.js)	4
1.6 接口(API).....	4
二、伐木搭船.....	4
2.1 知识储备.....	4
2.2 起步	5
2.2.1 安装基础包.....	5
2.2.2 章节目录.....	5
2.2.3 编写代码.....	6
2.2.4 注释.....	7
2.2.5 安装包.....	7
2.2.6 打包.....	7
2.2.7 查看输出.....	7
2.3 优化	9
2.3.1 编写代码.....	9
2.3.2 打包.....	9
2.3.3 查看输出.....	9
2.3.4 小吐槽.....	10
2.3.5 结束?	10
2.3.6 问题解答.....	11
2.3.7 完结?	11
2.3.8 生产模式.....	11
2.3.9 优化思考【可选】	12
2.3.10 完整代码【可选】	12
2.4 管理资源.....	14
2.4.1 目录结构.....	14
2.4.2 完整代码.....	15
2.4.3 讲解.....	19
2.4.4 准备.....	19
2.4.5 引入 CSS.....	22

2.4.6 引入图片.....	24
2.4.7 引入字体.....	26
2.4.8 引入数据文件.....	28
2.4.9 小结.....	30
2.5 管理输出.....	31
2.5.1 开篇点题.....	31
2.5.2 优雅拷贝.....	33
2.5.3 多 js 文件	34
2.5.4 动态更新 HTML.....	37
2.5.5 动态刷新 dist.....	40
2.6 开发	42
2.6.1 查找报错文件.....	42
2.6.2 实时重新加载.....	44
2.6.3 目录及代码整理.....	47
2.7 分离配置文件.....	52
2.8 调整	56
2.9 番外【可选】	Error! Bookmark not defined.
三、扬帆起航.....	56
3.1 loaders.....	56
3.1.1 css-loader	56
3.1.2 style-loader	57
3.1.3 less-loader	57
3.1.4 file-loader	57
3.1.5 url-loader.....	57
3.1.6 html-withimg-loader	57
3.2 plugins.....	57
3.2.1 html-webpack-plugin.....	57
3.2.2 clean-webpack-pugin	58
3.2.3 mini-css-extract-plugin.....	58
3.2.4 uglifyjs-webpack-plugin.....	58
3.2.5 optimize-css-assets-webpack-plugin	58
3.3 other npm	59
3.3.1 less.....	59
3.3.2 webpack	59
3.3.3 webpack-cli.....	59
3.3.4 webpack-dev-server	59
3.3.5 webpack-merge.....	59

序言

时光静静地飞，我悄悄地追，无论成功与否，我不曾后悔。——致我逝去的大学生活，2018/06/24 毕业快乐！

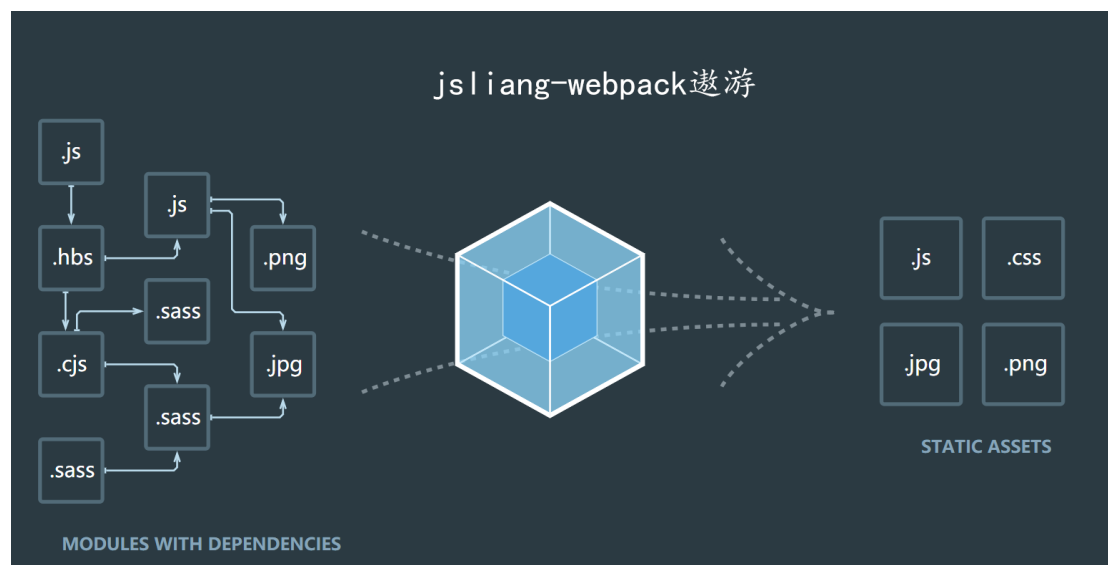
webpack 是我学习的第一个打包构建工具，没能跟着教学视频飞起来是个遗憾，毕竟市面上没有 webpack4 的教学视频，可能是技术更新太快了，毕竟听说 webpack5 又快出了，作为一枚从大一到现在初混社会都是自学过来的人，只能自己躺坑了，要不老了就跟不上了！

对于一个小萌新来说，webpack4 的中文文档是个坑，链接各种无效，浅入无出，是不是有很多小伙伴跟我一样一开始都是过到一点点基础就放弃了？“我靠！我还是等我的 webpack 构建师吧！”。然后 jsliang 到 GitHub 上也没找到想要的，最后还是自己一个一个 loader、插件去了解吧，webpack4 的周边也挺大的了~

然后顺带讲讲我的旅途吧：

- 一、磨刀霍霍：了解 webpack 基本概念；
- 二、伐木搭船：尝试 webpack 基本配置；
- 三、扬帆起航：使用 webpack 配置项目；

一、磨刀霍霍



超喜欢 webpack 大神的，loader 玩得贼溜，plugins 随手就用，分分钟教你做人。

啥？什么是 webpack？<https://www.webpackjs.com/>

啥？你不想皈依 webpack？<https://www.cnblogs.com/kasmine/p/6436131.html>

啥？loader、plugins 是什么？[https://.....](https://...)想多了！我给了链接你还会继续往下看么！

1.1 入口(entry)

构建 webpack 内部依赖图的开始，webpack 根据入口依序寻找其依赖的相关文件（直接或者间接）。在 webpack 中，根据你想做单页面应用程序还是多页面应用程序，可以配置

单个入口或多个入口。

1.2 出口(output)

输出打包后的文件到哪，自命名个文件名还是使用默认的./dist 文件。在 webpack 中，我们可以理解 webpack 为黑盒子，我们引导 webpack，写好入口配置，然后指定 webpack 打包输出的出口位置，webpack 就会根据我们的需求，使用黑魔法帮我们搞定。入口和出口的配置如下：

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

module.exports = {
  entry: './path/to/my/entry/file.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-first-webpack.bundle.js'
  }
};
```

解析：

output.path - 文件路径

output.filename - 文件名

1.3 解析器(loader)

说明：官方文档只表明什么时候使用 loader，但是 jsliang 使用 webpack 时觉得 loader 很像解析器，所以自己翻译成解析器，正式说法就是 loader，没有官方承认的“解析器”说法。

由于 webpack 自身只理解 JavaScript 的特性，所以为了帮助 webpack 处理非 JavaScript 文件，我们需要使用解析器(loader)。

```
const path = require('path');
const config = {
  output: {
    filename: 'my-first-webpack.bundle.js'
  },
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  }
}
```

```
};
```

```
module.exports = config;
```

解析:

rules.test - 应该被转换的文件范围

rules.use - 应该使用哪个 loader

上面我们配置了一种 txt 解析器 raw-loader，用来解析打包过程中出现的 txt 文件。其他的一些 loader 还有:

style-loader : 将模块的导出作为样式添加到 DOM 中

css-loader : 解析 CSS 文件后, 通过 import 加载, 并且返回 CSS 代码

vue-loader : 加载和转译 Vue 组件

1.4 插件(plugins)

插件可以用来处理各种任务, 就好比平时编写 JavaScript 时引入轮播图插件、弹窗插件一样。webpack 插件的范围包括: 从打包优化和压缩, 一直到重新定义环境都有其身影。

使用插件的方式: require()它, 然后添加到 plugins()数组中, 通过 new 调用它。

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); // 通过 npm 安
```

```
装 const webpack = require('webpack'); // 用于访问内置插件
```

```
const config = {
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin(),
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};
module.exports = config;
```

下面介绍几种常用插件:

ProvidePlugin : 不必通过 import/require 使用模块。使用\$配置的时候, 还能默认\$为 jQuery 的 exports。

HTMLWebpackPlugin : 简单创建 HTML 文件, 用于服务器访问。动态添加<link>和<script>, 在以 hash 命名的文件上非常有用, 因为每次编译都会改变。

IgnorePlugin : 从 bundle 中排除某些模块, 即打包时忽略正则匹配的模块。

1.5 配置(webpack.config.js)

在 webpack 中，我们可以通过配置我们需要打包的环境（入口、出口、loader、插件都在配置范围内），然后使用 npm 命令行，打包成我们需要的文件结构。

1.6 接口(API)

在 webpack 中，我们可以使用各种接口来定制化编译过程。在这过程中，我们可能使用 Node.js 的接口，也有可能使用 loader 的接口……这取决于我们想要做出怎样的成果。

二、伐木搭船

2.1 知识储备

工欲善其事，必先利其器。下面是伐木搭船章节所涉及的知识点，如有不适，请出门左转，感谢你的学习，《webpack 从入门到放弃》怀念你曾到来~

网址：

npm 官网：<https://www.npmjs.com.cn/>

cnpm 镜像：<https://npm.taobao.org/>

npx 工具：<https://www.npmjs.com/package/npx>

windows 命令行：<https://jingyan.baidu.com/article/c74d6000a78f050f6a595df4.html>

Visio Studio Code 下的终端控制台：

<https://jingyan.baidu.com/article/20095761ec9d69cb0721b4e9.html>

命令行解释：

--save-dev == -D （开发时依赖，生产环境下不依赖）

--save == -S （开发生产环境都依赖）

install == i

cnpm init -y == 默认所有配置为 yes

npx webpack == npx 是一个执行 npm 包的二进制文件

命令行：

安装 webpack: **cnpm i webpack --save-dev**

安装 webpack 指定版本: **cnpm i -D webpack@<version>**

安装 webpack-cli: **cnpm i webpack-cli -D**

全局安装 webpack: **cnpm i webpack -g**

初始化项目: **cnpm init -y**

安装 lodash: `cnpm i lodash -S`
webpack 打包: `npx webpack`
打包成生产环境所需: `cnpm run build`
安装 css 解析器: `cnpm i style-loader css-loader -D`
安装 file-loader 解析器: `cnpm i file-loader -D`
安装 url-loader 解析器: `cnpm i url-loader -D`
安装 data 解析器: `cnpm i csv-loader xml-loader -D`
安装 HtmlWebpackPlugin 插件: `cnpm i html-webpack-plugin -D`
安装 CleanWebpackPlugin 插件: `cnpm i clean-webpack-plugin -D`
安装实时重新加载插件: `cnpm i webpack-dev-server -D`
安装 webpack 配置分离工具: `cnpm i webpack-merge -D`

2.2 起步

2.2.1 安装基础包

话不多说，撸起~打开命令行：Windows 系统 Win+R->cmd 或者 Visio Studio Code 的 Ctrl+~，我们先安装基础的 webpack:

- `cd webpack-demo`
- `cnpm init -y`
- `cnpm i webpack webpack-cli -D`

知识储备:

什么是命令行:

<https://baike.baidu.com/item/%E5%91%BD%E4%BB%A4%E8%A1%8C%E7%95%8C%E9%9D%A2/9910197?fr=aladdin>

Windows 如何快速打开命令行:

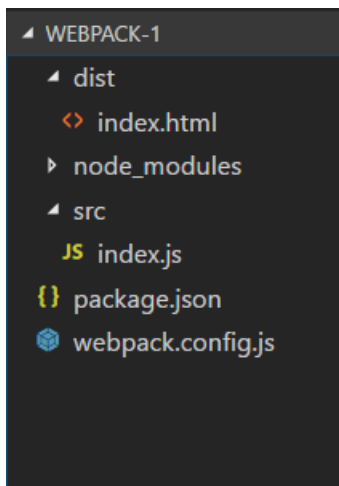
<https://jingyan.baidu.com/article/3aed632e79d388701180916d.html>

Visio Studio 如何打开命令行:

<https://jingyan.baidu.com/article/20095761ec9d69cb0721b4e9.html>

2.2.2 章节目录

然后创建该章节的目录（index.html、index.js、webpack.config.js 文件内容先为空）



2.2.3 编写代码

编写以下代码：

index.html

```
<!doctype html>
<html>

<head>
  <title>起步</title>
</head>

<body>
  <script src="main.js"></script>
</body>

</html>
```

index.js

```
import _ from 'lodash';

function component() {
  var element = document.createElement('div');

  // 调用 Lodash 的 _符号 全局变量
  element.innerHTML = _.join(['Hello', 'webpack'], ' ');

  return element;
}
```



```
document.body.appendChild(component());
```

2.2.4 注释

接下来对上面的内容解释一番

在这里，我们对 `index.html` 是毫无疑问的，单纯引入一个 `main.js` 而已。然后在 `index.js` 中，我们引入了 `lodash`，然后通过 `js` 往 `body` 里创建了个 `div` 元素，内容是 `Hello webpack`。所以毫无疑问，我们需要通过 `npm` 安装 `lodash`。

知识储备：

`lodash` 工具库：<https://www.lodashjs.com/>

2.2.5 安装包

安装 `lodash`

- `cnpm i lodash -S`

2.2.6 打包

开始打包

OK，一切准备就绪后，我们开始见证奇迹了！在命令行输入下面代码：

- `npx webpack`

2.2.7 查看输出

最后查看命令行输出结果：

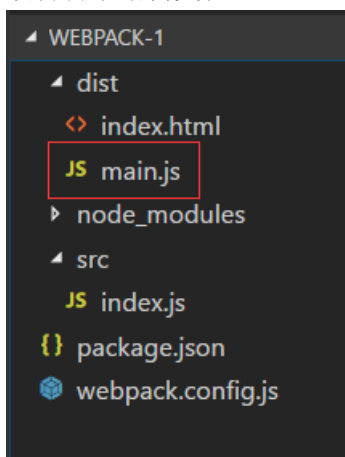
```

PS E:\MyWeb\jsliang-web\Webpack\webpack-1> npx webpack
npx: installed 1 in 1.407s
Path must be a string. Received undefined
E:\MyWeb\jsliang-web\Webpack\webpack-1\node_modules\_webpack@4.12.0\webpack\bin\webpack.js
Hash: e1723a9f0c81693d50d3
Version: webpack 4.12.0
Time: 263ms
Built at: 2018-06-20 11:13:40
    Asset      Size  Chunks             Chunk Names
  main.js  70.4 KiB       0  [emitted]  main
    [1] (webpack)/buildin/module.js 497 bytes {0} [built]
    [2] (webpack)/buildin/global.js 489 bytes {0} [built]
    [3] ./src/index.js 253 bytes {0} [built]
       + 1 hidden module

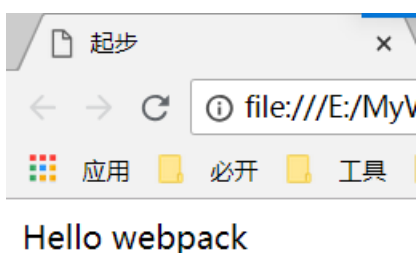
WARNING in configuration
The 'mode' option has not been set, webpack will fallback to 'production' for this value.
Set 'mode' option to 'development' or 'production' to enable defaults for each environment.
You can also set it to 'none' to disable any default behavior. Learn more: https://webpack.js.org/concepts/mode/

```

项目目录结构变化:



浏览器打开 index.html:



嗯，是的，webpack 打包基础版就这么完成了，好了，恭喜你们学会 webpack 了，祝大家在前端学习路上越走越远，webpack 学习就此结束，拜呐您~~

2.3 优化

咳，我知道优秀的你会继续往下看的，OK 咱们继续！

在上面的 webpack 打包基础版上，我们的确做到了“webpack 打包”，但是，如果后面我们需要修改文件入口、文件出口名字啥的，或者区分下开发模式和生产模式的代码，那改起来不是麻烦？所以，我们能不能创建一个配置文件，专门配置 webpack 所需呢？这时候我们要开始使用目录上的 `webpack.config.js` 文件了。

2.3.1 编写代码

编写代码如下：

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

2.3.2 打包

编写完代码后，我们再次打包。

在打包之前建议先删掉 `main.js`，以便我们观察新的打包是否成功（如果要用之前的方式再打包 `main.js` 出来，可以注释掉 `webpack.config.js` 的内容再运行 `npx webpack` 即可）。执行下面命令行：

- `npx webpack --config webpack.config.js`

2.3.3 查看输出

查看命令行输出结果：

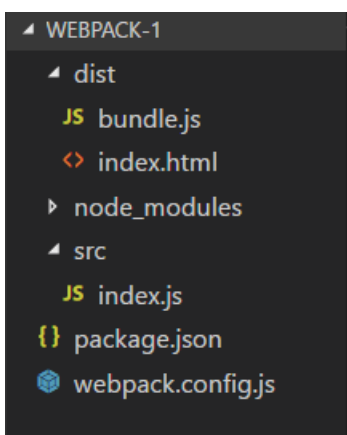
```

PS E:\MyWeb\jsliang-web\Webpack\webpack-1> npx webpack --config webpack.config.js
npx: installed 1 in 1.39s
Path must be a string. Received undefined
E:\MyWeb\jsliang-web\Webpack\webpack-1\node_modules\_webpack@4.12.0@webpack\bin\webpa
ck.js
Hash: 369b79a22a71184a948e
Version: webpack 4.12.0
Time: 268ms
Built at: 2018-06-20 11:39:49
    Asset      Size  Chunks             Chunk Names
bundle.js  70.4 KiB       0  [emitted]  main
[1] (webpack)/buildin/module.js 497 bytes {0} [built]
[2] (webpack)/buildin/global.js 489 bytes {0} [built]
[3] ./src/index.js 237 bytes {0} [built]
   + 1 hidden module

WARNING in configuration
The 'mode' option has not been set, webpack will fallback to 'production' for this va
lue. Set 'mode' option to 'development' or 'production' to enable defaults for each e
nvironment.
You can also set it to 'none' to disable any default behavior. Learn more: https://we
bpack.js.org/concepts/mode/

```

项目目录结构变化:



2.3.4 小吐槽

吐槽一番刚才结果。

OK, 如我们所愿, 我们根据 webpack.config.js 配置成功使用 webpack 打包了, 以后妈妈再也不用担心我的代码太乱了~

2.3.5 结束?

这样就结束了吗?

然而并没有结束, 看到上面那一串命令行了吗? 老长了好不好? 所以, 我们要调整下 npm 脚本, 缩减下工作量!

打开 package.json (jsliang 温馨提示：请记住 package.json 是 npm 的工作区，webpack.config.js 是 webpack 的工作区，有不懂请百度搜索对应的 npm、webpack 问题，别傻傻分不清哦~)，往"scripts":{ ... }里添加一行代码，新的"scripts"如下：

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "build": "webpack"  
},
```

OK，这句话的意思就是，咱用 "build" : "webpack" 代替 npx webpack -config webpack.config.js 的工作，以后，咱打包就用：

- **cnpm run build**

2.3.6 问题解答

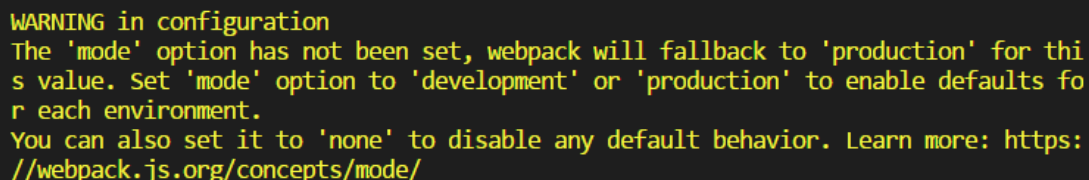
咦？！！！！

为什么我现在打开 index.html 没反应？是不是 jsliang 在骗我？！……亲，不是我的锅，注意 index.html 引入的文件名，是 main.js 哦~所以你要修改为 bundle.js，或者修改 webpack.config.js 的 bundle.js 为 main.js (为什么不一开始就修改？主要是因为区分之前的 main.js，让大家更好地了解添加 webpack.config.js 这个配置文件前后的不同哦~)

2.3.7 完结？

优化完结撒花？

噢，不，还没有！



```
WARNING in configuration  
The 'mode' option has not been set, webpack will fallback to 'production' for this value. Set 'mode' option to 'development' or 'production' to enable defaults for each environment.  
You can also set it to 'none' to disable any default behavior. Learn more: https://webpack.js.org/concepts/mode/
```

注意到这句话了吗？在 webpack 打包过程中，出现黄色意味着“警告”，出现红色意味着“报错”，所以，作为 0 容忍党，怎么能出现黄色警告呢！叽里呱啦翻译一下，大概意思就是提醒我们要设置 mode (模式)，然后 production 是生产模式，development 是开发模式，或者咱设置 none 告诉 webpack 我们默认了。OK，解释完毕，我们继续。

2.3.8 生产模式

设置生产模式

打开 webpack.config.js 这个配置文件，我们往里添加一句 mode:"production"，这样，就设置了 webpack 打包模式为生产模式啦~全文件如下：

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

module.exports = {
  mode: "production",
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

OK，我们删掉前面打包出来的 bundle.js，再运行 `cnpm run build`，可以发现，这时候的黄色警告就没有啦！

2.3.9 优化思考【可选】

再度思考

PS：可选意味着可以跳过

虽然我们可以在 `webpack.config.js` 这里配置，但是到时候我们开发的时候又要修改为 `development`，很麻烦的好吗？而且我们已经通过 `npm` 命令行来进行打包了，所以，能不能在 `npm` 那边配置而不用在 `webpack.config.js` 这里设置了？（毕竟之后开发模式的命令行是 `"cnpm run dev"`）

可以的，删掉 `mode:"production"`，我们打开 `package.json`，往 `"scripts"` 的 `"build"` 添加设置内容，使其变成：`"build": "webpack --mode production"`

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "webpack --mode production"
},
```

这样，我们就又可以偷懒啦~

2.3.10 完整代码【可选】

2.1 节起步 + 2.2 节优化 的代码如下

index.html

```
<!doctype html>
<html>

<head>
  <title>起步</title>
</head>
```

```
<body>
  <script src="bundle.js"></script>
</body>

</html>
```

index.js

```
import _ from 'lodash';

function component() {
  var element = document.createElement('div');

  // 调用 Lodash 的_全局变量
  element.innerHTML = _.join(['Hello', 'webpack'], ' ');

  return element;
}

document.body.appendChild(component());
```

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

package.json

```
{
  "name": "webpack-1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack --mode production"
  },
  "keywords": [],
```

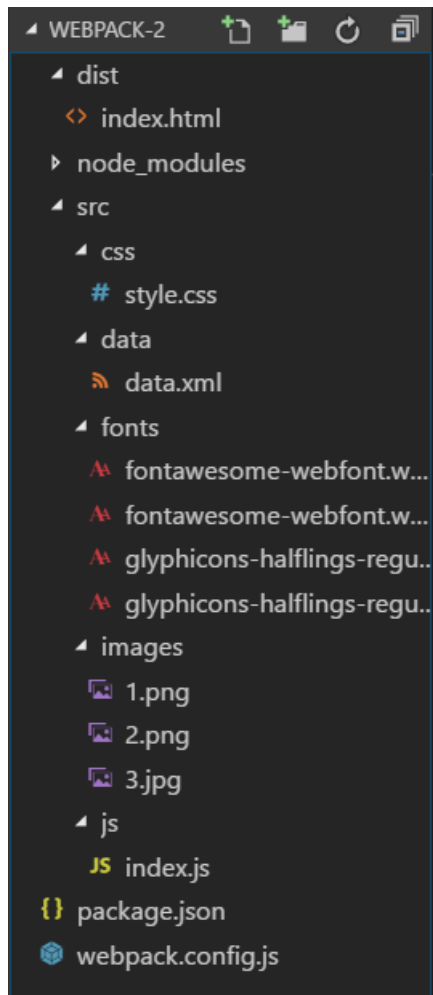
```
"author": "",
"license": "ISC",
"devDependencies": {
  "webpack": "^4.12.0",
  "webpack-cli": "^3.0.8"
}
```

2.4 管理资源

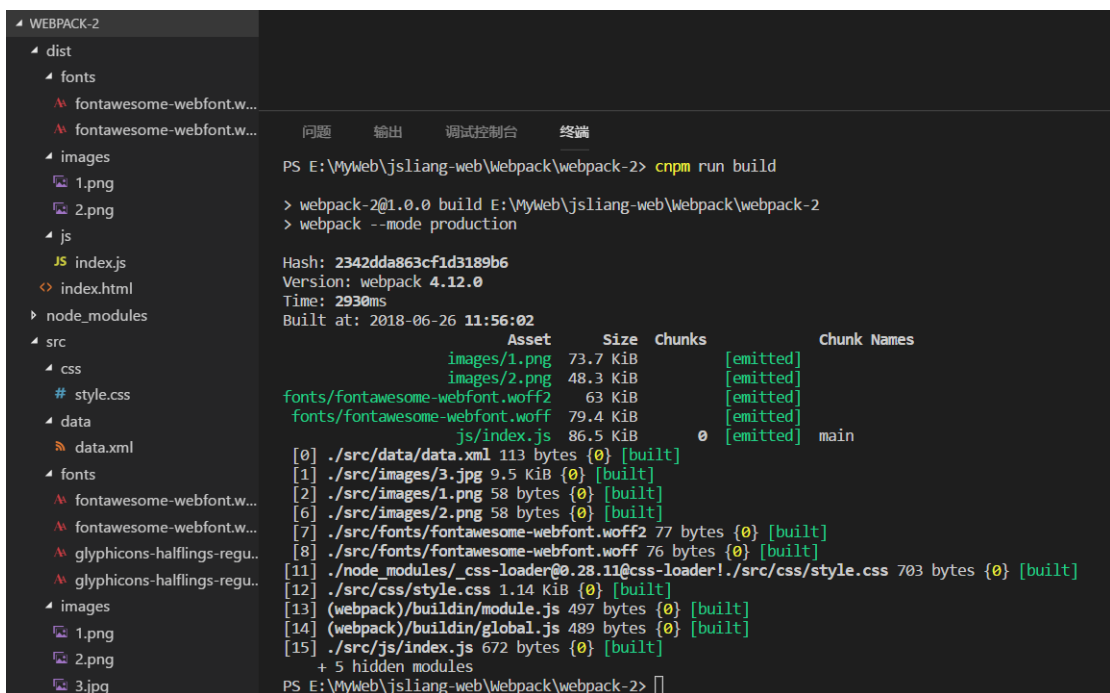
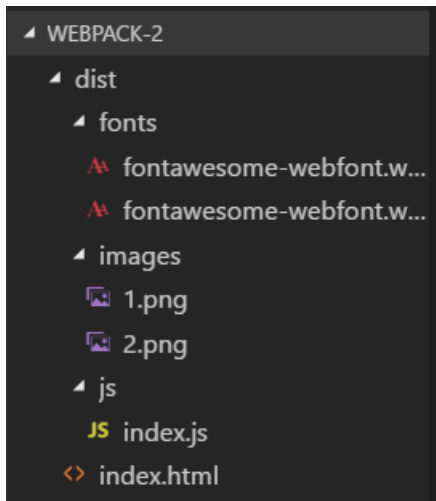
2.4.1 目录结构

项目目录结构：

（打包前）



（打包后）



2.4.2 完整代码

完整代码如下：

index.html

```
<!doctype html>
<html>

<head>

  <title>资源管理</title>

</head>
```

```

<body>
  <script src="./js/index.js"></script>
</body>

</html>

```

style.css

```

@font-face {
  font-family: 'MyFont';
  src: url('../fonts/fontawesome-webfont.woff') format('woff'),
  url('../fonts/fontawesome-webfont.woff2') format('woff2');
  font-weight: bold;
  font-style: normal;
}

.hello {
  color: red;
  background: url('../images/2.png');
  font-family: 'MyFont';
}

```

data.xml

```

<?xml version="1.0" encoding="UTF-8"?>
  <note>
    <to>Mary</to>
    <from>John</from>
    <heading>Reminder</heading>
    <body>Call Cindy on Tuesday</body>
  </note>

```

fonts

请自行查找 **woff/woff2** 文件进行替换，并更新引用字体的代码片段

images

请自行添加图片，并更新引用图片的代码片段

index.js

```

import _ from 'lodash';
import './css/style.css';
import Gif1 from '../images/1.png';
import Image1 from '../images/3.jpg';
import Data from '../data/data.xml';

function component() {

```

```

var element = document.createElement('div');

// 调用 Lodash 的_全局变量

element.innerHTML = _.join(['Hello', 'webpack', ', 我叫梁峻荣。'], ' ');
element.classList.add('hello');

// 将图片添加到 div 中

var myGif = new Image();
var myImage = new Image();
myGif.src = Gif1;
myImage.src = Image1;

element.appendChild(myGif);
element.appendChild(myImage);

console.log(Data);

return element;
}

document.body.appendChild(component());

```

package.json

```

{
  "name": "webpack-2",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack --mode production"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "css-loader": "^0.28.11",
    "csv-loader": "^2.1.1",
    "file-loader": "^1.1.11",
    "style-loader": "^0.21.0",
    "url-loader": "^1.0.1",
    "webpack": "^4.12.0",

```

```

    "webpack-cli": "^3.0.8",
    "xml-loader": "^1.2.1"
  },
  "dependencies": {
    "lodash": "^4.17.10"
  }
}

```

webpack.config.js

```

const path = require('path');

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/index.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\..(jpg|jpeg|png|gif|svg)$/,
        use: [
          'url-loader?limit=10240&name=images/[name].[ext]'
        ]
      },
      {
        test: /\..(woff|woff2|eot|ttf|otf)$/,
        loader: 'url-loader',
        options: {
          limit: 10240,
          name: 'fonts/[name].[ext]'
        }
      },
      {
        test: /\..(csv|tsv)$/,
        use: [
          'csv-loader'
        ]
      }
    ]
  }
}

```

```

    ]
  },
  {
    test: /\.xml$/,
    use: [
      'xml-loader'
    ]
  }
]
}
};

```

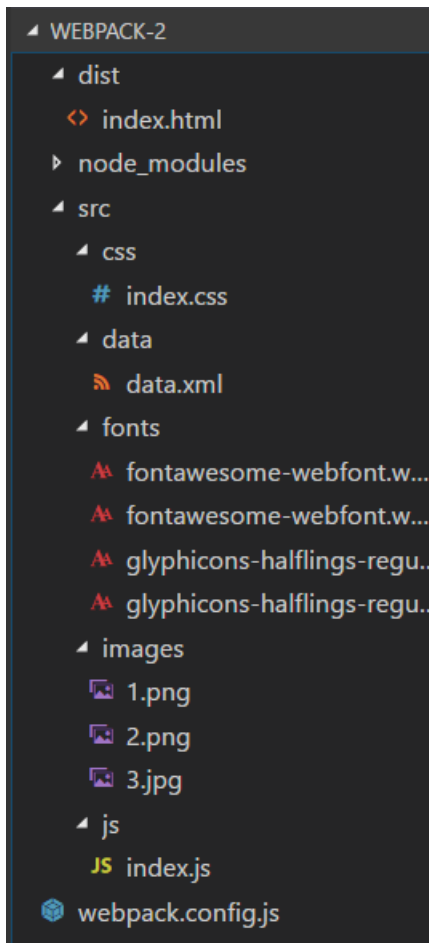
2.4.3 讲解

对上面目录进行详细讲解。

在这次试验中，我们要做的就是，将 `html/css/js/data/fonts` 文件分文别类，然后在打包到 `dist` 的时候，也让它根据我们的规则来打包输出，最终做成一个小小的 `demo`。

2.4.4 准备

首先需要根据上面的文件目录将文件准备好。（注意：`index.html`、`index.css`、`data.xml`、`index.js`、`webpack.config.js` 都是空文件，后面我们会添加它，而 `fonts` 的 4 个文件格式分别为 `woff`、`woff2`，至于 `images` 里面的图片，随便找自己喜欢的图片就 OK 了）



然后新建简单的 webpack 项目：

- `cd webpack-2`
- `cnpm init -y`
- `cnpm i webpack webpack-cli -D`
- `cnpm i lodash -S`

接着往 index.html、index.js、webpack.config.js 中添加内容：

index.html

```
<!doctype html>
<html>

<head>

  <title>资源管理</title>

</head>

<body>
  <script src="./js/index.js"></script>
</body>
```

```
</html>
```

index.js

```
import _ from 'lodash';

function component() {
  var element = document.createElement('div');

  // 调用 Lodash 的_全局变量

  element.innerHTML = _.join(['Hello', 'webpack', ', 我叫梁峻荣。'], ' ');
  element.classList.add('hello');

  return element;
}

document.body.appendChild(component());
```

webpack.config.js

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

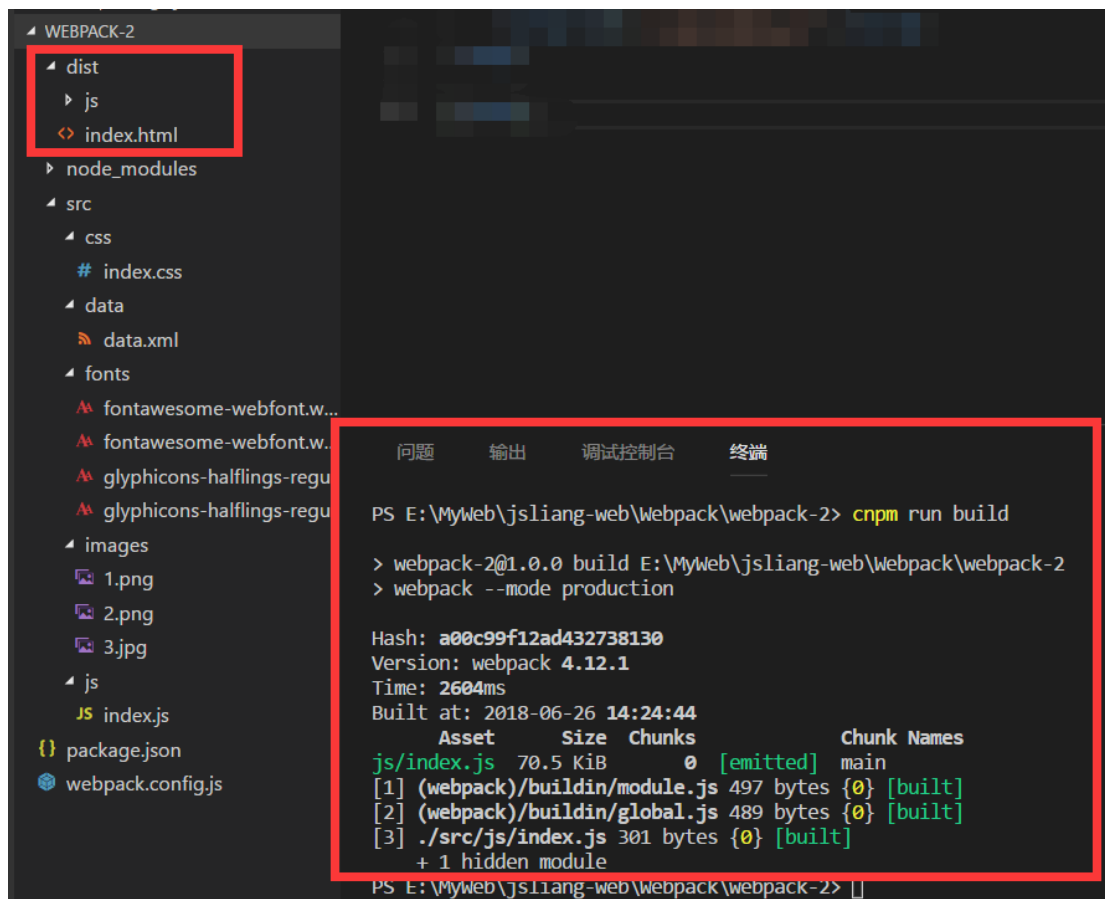
module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/index.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

以及新出来的 **package.json** 中添加"build"方法:

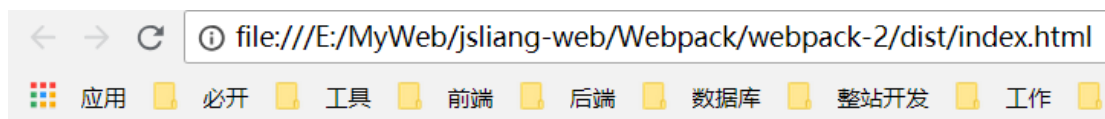
package.json

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "webpack --mode production"
},
```

OK, 准备就绪, 我们在控制台输出 **cnpm run build**, 你可以看到它神奇地能运行了, 并打包 **index.js** 到了 **js** 文件夹中:



打开 index.html，看到输出：



Hello webpack，我叫梁峻荣。

很好，聪明的小伙伴们可以回顾下刚才我们做了什么？

是的，我们在 dist 文件夹中写了个 index.html，引用了打包后的 index.js。在 index.js 中往网页添加了一行文字"Hello webpack，我叫梁峻荣"，在 webpack.config.js 中，我们配置了 js 入口以及打包输出的 js 路径是 js/index.js，在 package.json 中配置了打包快捷键"npm run build"。所以，当我们在控制台输出"npm run build"的时候，我们已经预料到会输出什么了！

2.4.5 引入 CSS

安装 css 的 loader：

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

接着，我们需要再 webpack.config.js 这个配置上新增配置规则，告诉 webpack，我们需要打包引入的 css 到 css/**/*.css 中：

webpack.config.js

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

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/index.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      }
    ]
  }
};
```

然后，往 index.css、index.js 中添加内容：

index.css

```
.hello {
  color: red;
}
```

index.js

```
import _ from 'lodash';
import '../css/index.css';

function component() {
  var element = document.createElement('div');

  // 调用 Lodash 的_全局变量

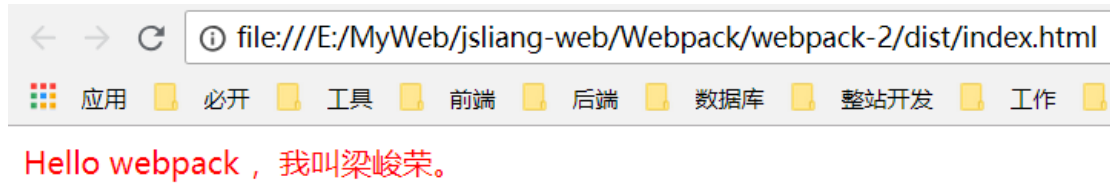
  element.innerHTML = _.join(['Hello', 'webpack', ', 我叫梁峻荣。'], ' ');
  element.classList.add('hello');

  return element;
}

document.body.appendChild(component());
```

请注意观察我们在这里做了什么？

是的，我们在 `index.js` 中引入了 `index.css` 样式，使我们字体变红了：



2.4.6 引入图片

说到 `css`，我们就应该想到，我们还需要对图片进行处理，将图片引用，并输出到 `dist` 的 `css` 文件夹中。

所以，想到就做！开始引入文件的 loader 吧：

- `cnpm i file-loader -D`
- `cnpm i url-loader -D`

【小插曲：在 `npm` 安装 `file-loader` 的时候，我以为不需要安装这个包（事实上 `file-loader` 与 `url-loader` 要相互依赖引用的，点击下面链接可以查看 `url-loader` 与 `file-loader` 的区别：<https://www.npmjs.com/package/url-loader>），所以我删除了这个依赖：• `cnpm uninstall file-loader -D` 删除完毕后产生一个 `package-lock.json` 的文件】

然后，修改 `webpack.config.js` 配置：

webpack.config.js

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

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/index.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\.?(jpg|jpeg|png|gif|svg)$/,
```

```

        use: [
          'url-loader?limit=10240&name=images/[name].[ext]'
        ]
      }
    ]
  }
};

```

接着，在 `css` 的 `background` 中引入图片背景，在 `js` 中往 `HTML` 内容添加图片：

index.css

```

.hello {
  color: red;
  background: url('../images/2.png');
}

```

index.js

```

import _ from 'lodash';
import '../css/index.css';
import Gif1 from '../images/1.png';
import Image1 from '../images/3.jpg';

function component() {
  var element = document.createElement('div');

  // 调用 Lodash 的_全局变量

  element.innerHTML = _.join(['Hello', 'webpack', '， 我叫梁峻荣。'], ' ');
  element.classList.add('hello');

  // 将图片添加到 div 中

  var myGif = new Image();
  var myImage = new Image();
  myGif.src = Gif1;
  myImage.src = Image1;

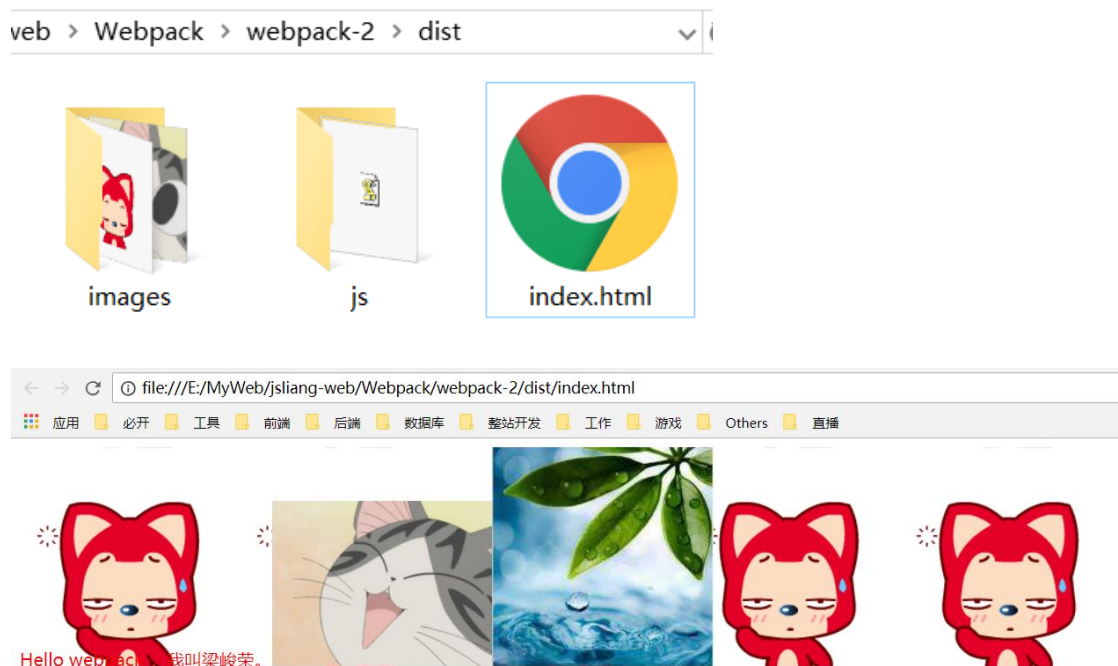
  element.appendChild(myGif);
  element.appendChild(myImage);

  return element;
}

document.body.appendChild(component());

```

最后，在打包前，我们先删除刚才打包的 js 文件夹，以防打包文件重复（虽然它会自动替换，但是在我们学习完美打包前，我们还是手动删除，以便更好进行观察），并在控制台输入命令："cnpm run build"，打开打包后的 index.html 查看：



值得注意的是：为什么我们在图片的引入上引入了三张图片：1.png、2.png、3.jpg，但是在 dist 打包后的 images 文件夹中，我们只剩下两张图片了呢？给我这么一说是不是想知道原因？那么，请自行百度下面的话，查看下 url-loader 的运行机制吧：

```
use: [  
  'url-loader?limit=10240&name=images/[name].[ext]'  
]
```

2.4.7 引入字体

装载完 css 和图片，我们还可以装载个字体 loader 啦~这里我们还用装载图片的 file-loader 和 url-loader，所以就不需要安装新的包。

然后修改 webpack.config.js 配置：

webpack.config.js

```
const path = require('path');  
  
module.exports = {  
  entry: './src/js/index.js',  
  output: {  
    filename: 'js/index.js',  
    path: path.resolve(__dirname, 'dist')  
  },  
  module: {
```

```

rules: [
  {
    test: /\.css$/,
    use: [
      'style-loader',
      'css-loader'
    ]
  },
  {
    test: /\. (jpg|jpeg|png|gif|svg)$/,
    use: [
      'url-loader?limit=10240&name=images/[name].[ext]'
    ]
  },
  {
    test: /\. (woff|woff2|eot|ttf|otf)$/,
    loader: 'url-loader',
    options: {
      limit: 10240,
      name: 'fonts/[name].[ext]'
    }
  }
]
};

```

并修改 index.css 内容:

index.css

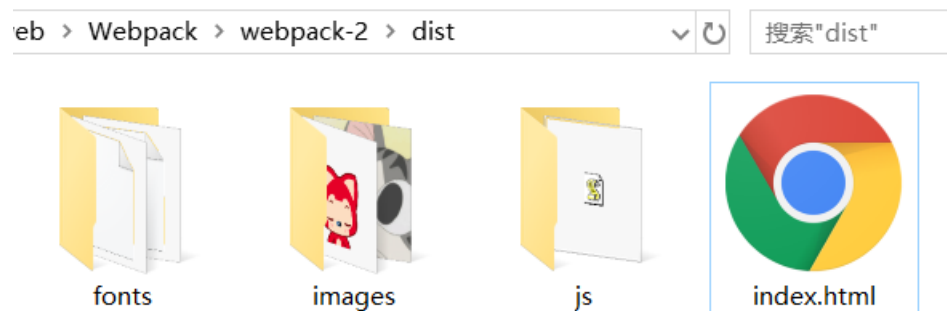
```

@font-face {
  font-family: 'MyFont';
  src: url('../fonts/fontawesome-webfont.woff') format('woff'),
  url('../fonts/fontawesome-webfont.woff2') format('woff2');
  font-weight: bold;
  font-style: normal;
}

.hello {
  color: red;
  background: url('../images/2.png');
  font-family: 'MyFont';
}

```

最后, 通过"cnpm run build"打包输出文件夹:



2.4.8 引入数据文件

其实到这步骤，小伙伴们应该都清楚啦，本章节就是引用各种 loader，扩展 webpack 的功能，因为 webpack 只能识别 javascript，所以使用 loader 能帮助它识别其他文件并进行打包构建：

- `cnpm i csv-loader xml-loader -D`

然后修改 webpack.config.js 配置：

webpack.config.js

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

```
module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/index.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\..(jpg|jpeg|png|gif|svg)$/,
        use: [
          'url-loader?limit=10240&name=images/[name].[ext]'
        ]
      }
    ]
  }
}
```

```

      test: /\.woff|woff2|eot|ttf|otf$/,
      loader: 'url-loader',
      options: {
        limit: 10240,
        name: 'fonts/[name].[ext]'
      }
    },
    {
      test: /\.csv|tsv$/,
      use: [
        'csv-loader'
      ]
    },
    {
      test: /\.xml$/,
      use: [
        'xml-loader'
      ]
    }
  ]
}
};

```

接着往空白的 **data.xml** 中新增数据，并在 **index.js** 中引用它并在控制台输出该数据：

data.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Mary</to>
  <from>John</from>
  <heading>Reminder</heading>
  <body>Call Cindy on Tuesday</body>
</note>

```

index.js

```

import _ from 'lodash';
import '../css/index.css';
import Gif1 from '../images/1.png';
import Image1 from '../images/3.jpg';
import Data from '../data/data.xml';

function component() {
  var element = document.createElement('div');

  // 调用 Lodash 的_全局变量

```

```

element.innerHTML = _.join(['Hello', 'webpack', ', 我叫梁峻荣。'], ' ');
element.classList.add('hello');

// 将图片添加到 div 中

var myGif = new Image();
var myImage = new Image();
myGif.src = Gif1;
myImage.src = Image1;

element.appendChild(myGif);
element.appendChild(myImage);

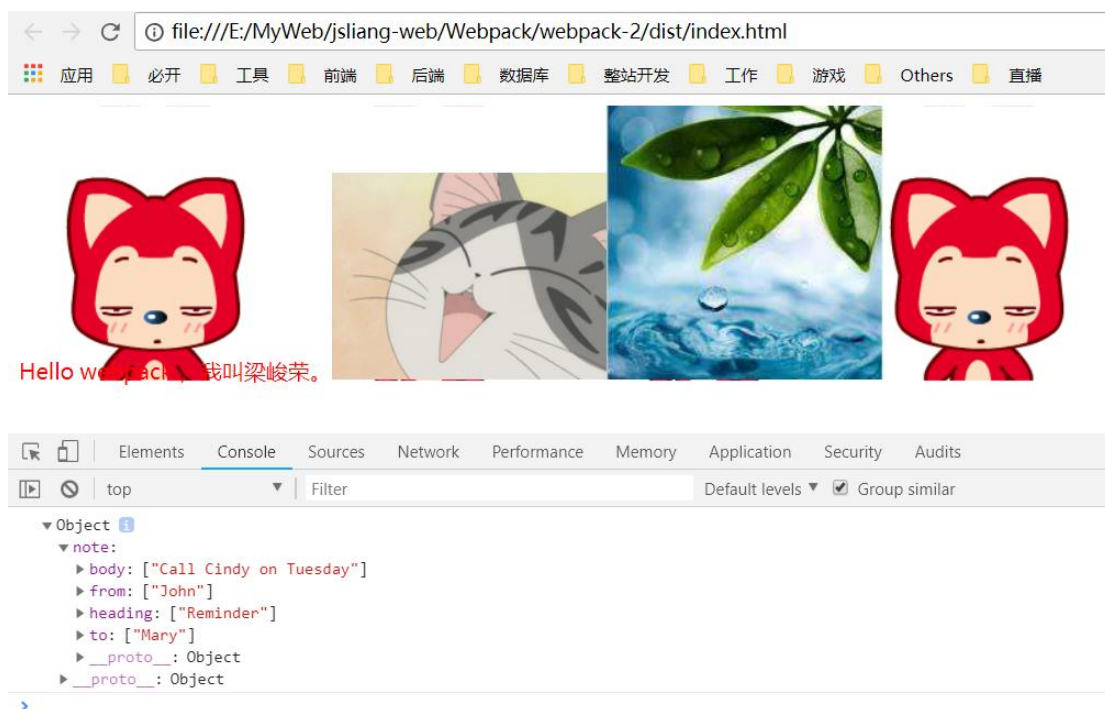
console.log(Data);

return element;
}

document.body.appendChild(component());

```

最后我们查看打包后 index.html 是否在控制台输出了内容：



2.4.9 小结

其实 jsliang 已经小结过了~至于在哪进行了小结，就看你有木有跟着实操一遍啦~记得

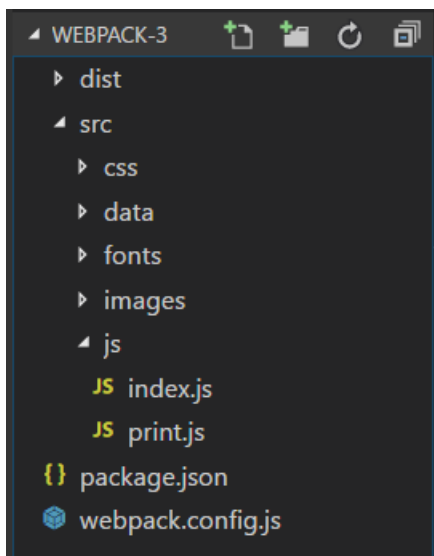
读万遍文档也要敲十万行代码哦~

2.5 管理输出

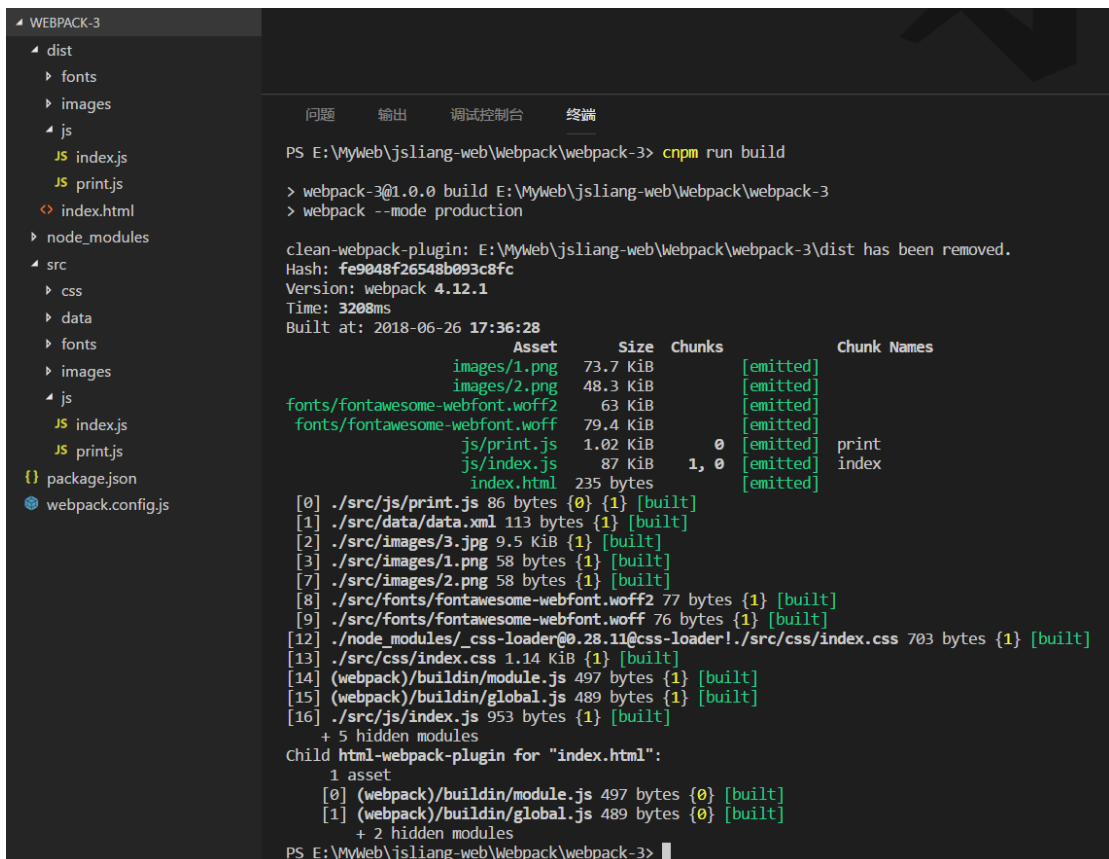
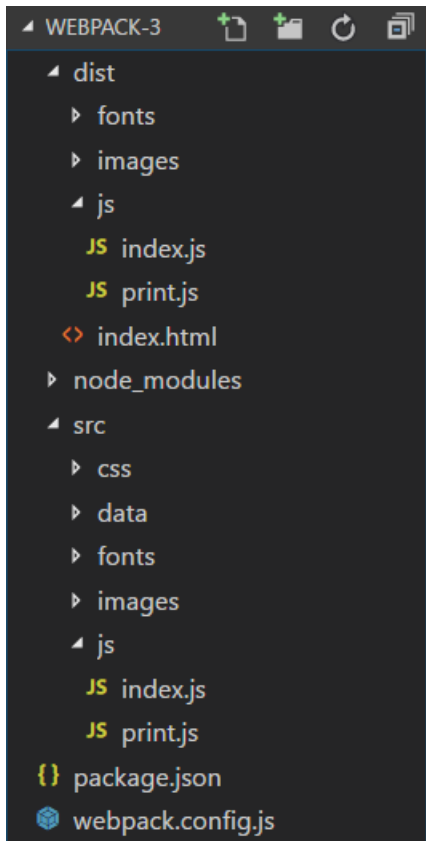
2.5.1 开篇点题

在本章节中，主要进行的三项工作是：1、进行多 js 文件管理；2、打包重构 index.html；3、打包前清理 dist 文件夹。

章节目录较前一章节发生的改变有：



打包后的目录如下：



2.5.2 优雅拷贝

在进行这章节项目之前， 我们需要拷贝下 **webpack-2** 文件夹（即上一章管理资源最终的代码）， 并进行小小的修改：

- 删除 **node_modules** 文件夹
- 删除 **dist** 文件夹下除 **index.html** 外其他文件
- 修改 **package.json**

```
{
  "name": "webpack-3",
  "version": "1.0.0",
  "description": "jsliang webpack study",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack --mode production"
  },
  "keywords": [
    "webpack-study"
  ],
  "author": "jsliang",
  "license": "ISC",
  "devDependencies": {
    "css-loader": "^0.28.11",
    "csv-loader": "^2.1.1",
    "file-loader": "^1.1.11",
    "style-loader": "^0.21.0",
    "url-loader": "^1.0.1",
    "webpack": "^4.12.1",
    "webpack-cli": "^3.0.8",
    "xml-loader": "^1.2.1"
  },
  "dependencies": {
    "lodash": "^4.17.10"
  }
}
```

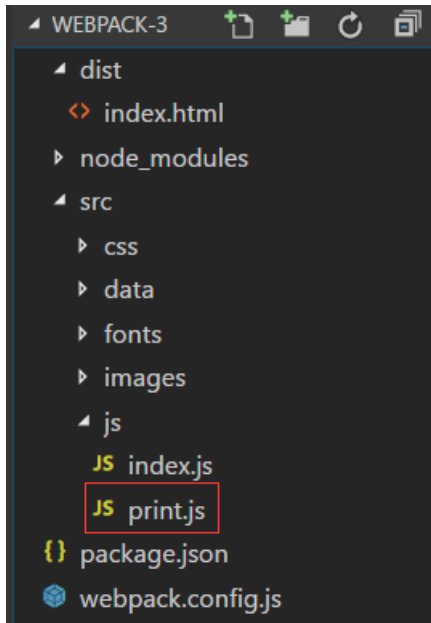
- 安装依赖： **cnpm i**
- 尝试是否能跟之前一样打包： **cnpm run build**

OK，最后发现，能完美继承 **webpack-2** 项目（PS：如果你经常使用 **vue-cli**、**react-cli**、**express** 等……你会发现大部分的项目的继承运行步骤跟上面类似，所以如果你在 **GitHub** 上 **download** 了一个项目，但是运行不了，那么，尝试了解下我上面的步骤，再试试是否能完美运行）

2.5.3 多 js 文件

很多情况下，我们将不止引用一个 js 文件，且该 js 文件命名可能随时发生变化，这时候就需要我们进行灵活配置，使其满足上面的要求。

本次目录改动为：



在这里，我们修改下配置文件，告诉 webpack 我们需要多 js 入口，并动态更改输出的 js 名称：

webpack.config.js

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

module.exports = {
  entry: {
    index: './src/js/index.js',
    print: './src/js/print.js'
  },
  output: {
    filename: 'js/[name].js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      }
    ]
  }
}
```

```

    ]
  },
  {
    test: /\. (jpg|jpeg|png|gif|svg)$/,
    use: [
      'url-loader?limit=10240&name=images/[name].[ext]'
    ]
  },
  {
    test: /\. (woff|woff2|eot|ttf|otf)$/,
    loader: 'url-loader',
    options: {
      limit: 10240,
      name: 'fonts/[name].[ext]'
    }
  },
  {
    test: /\. (csv|tsv)$/,
    use: [
      'csv-loader'
    ]
  },
  {
    test: /\.xml$/,
    use: [
      'xml-loader'
    ]
  }
]
}
};

```

并往 `js` 文件夹中新增一个 `print.js` 文件，并输入内容：

print.js

```

export default function printMe() {
  console.log('I get called from print.js!');
}

```

然后我们在 `index.js` 中引用该文件：

index.js

```

import _ from 'lodash';
import '../css/index.css';
import Gif1 from '../images/1.png';
import Image1 from '../images/3.jpg';

```

```

import Data from '../data/data.xml';
import printMe from './print.js';

function component() {
  var element = document.createElement('div');

  // 调用 Lodash 的_全局变量

  element.innerHTML = _.join(['Hello', 'webpack', ', 我叫梁峻荣。'], ' ');
  element.classList.add('hello');

  // 将图片添加到 div 中

  var myGif = new Image();
  var myImage = new Image();
  myGif.src = Gif1;
  myImage.src = Image1;
  element.appendChild(myGif);
  element.appendChild(myImage);

  // 输出数据

  console.log(Data);

  // 新增 print.js
  var br = document.createElement('br');
  element.appendChild(br);
  var btn = document.createElement('button');
  btn.innerHTML = 'Click me and check the console!';
  btn.onclick = printMe;
  element.appendChild(btn);

  return element;
}

document.body.appendChild(component());

```

并在 index.html 中引入新增的 print.js 文件:

index.html

```

<!doctype html>
<html>

<head>

  <title>资源管理</title>

```

```

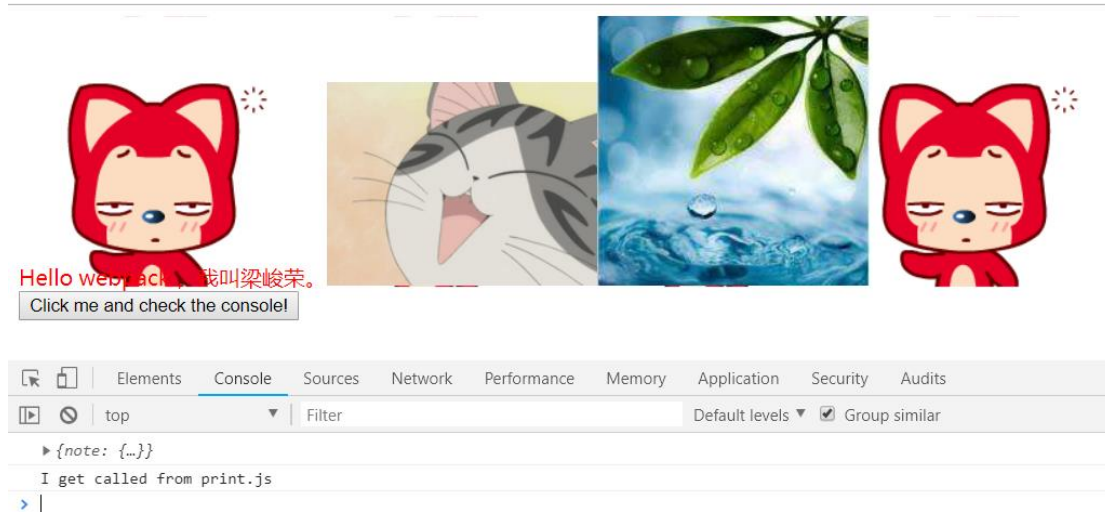
</head>

<body>
  <script src="./js/print.js"></script>
  <script src="./js/index.js"></script>
</body>

</html>

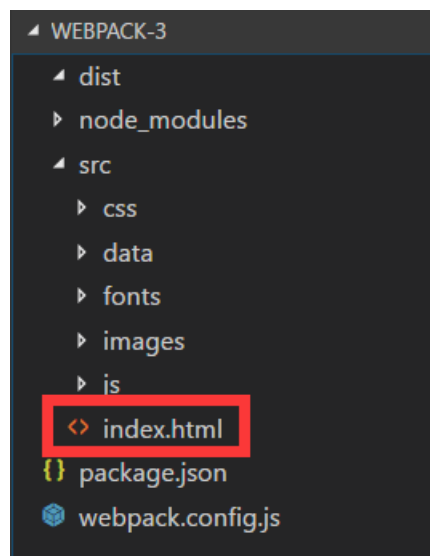
```

最后执行"cnpm run build", 查看输出内容:



2.5.4 动态更新 HTML

不知道你是否发现有发现, 在上面章节中, `index.html` 文件是放在 `dist` 文件中定死的, 这很不好, 不科学, 不符合我们的习惯, 就不能丢到 `src` 目录下么?



OK, 我们尝试引用一个“插件”，使其能按我们需求正常运行：

- `cnpm i html-webpack-plugin -D`

然后修改 `webpack.config.js` 配置：

webpack.config.js

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

module.exports = {
  entry: {
    index: './src/js/index.js',
    print: './src/js/print.js'
  },
  output: {
    filename: 'js/[name].js',
    path: path.resolve(__dirname, 'dist')
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: '管理输出'
    })
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\..(jpg|jpeg|png|gif|svg)$/,
        use: [
          'url-loader?limit=10240&name=images/[name].[ext]'
        ]
      },
      {
        test: /\..(woff|woff2|eot|ttf|otf)$/,
        loader: 'url-loader',
        options: {
          limit: 10240,
          name: 'fonts/[name].[ext]'
        }
      }
    ]
  }
}
```



```

    }
  },
  {
    test: /\. (csv|tsv) $/,
    use: [
      'csv-loader'
    ]
  },
  {
    test: /\.xml$/,
    use: [
      'xml-loader'
    ]
  }
]
}
};

```

最后执行"cnpm run build", 我们打开 index.html 源码, 惊奇发现, 这个 index.html 重构了!

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8">
5     <title>管理输出</title>
6   </head>
7   <body>
8     <script type="text/javascript" src="js/index.js"></script><script type="text/javascript" src="js/print.js"></script></body>
9 </html>

```

而我们一开始的代码是:

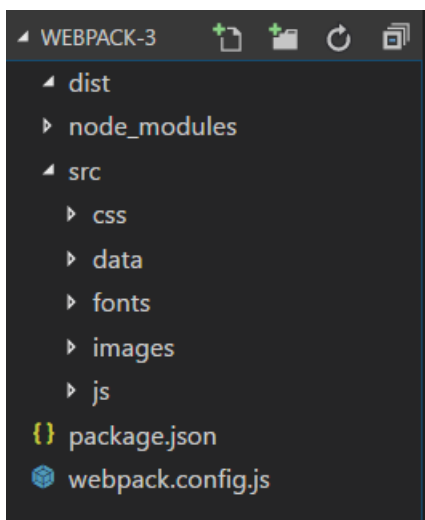


```

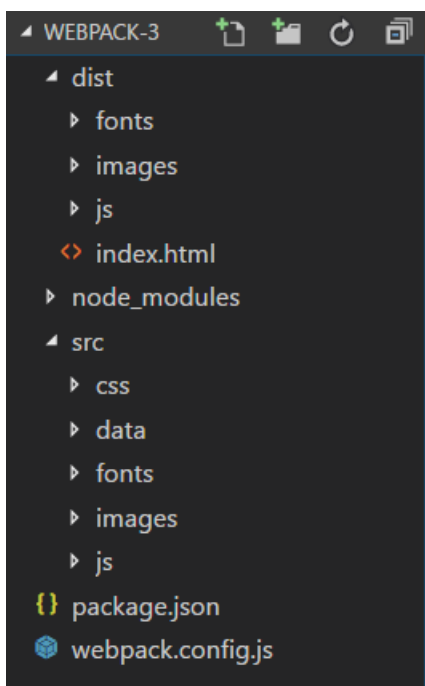
index.html x
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8">
5     <title>管理资源</title>
6   </head>
7   <body>
8     <script type="text/javascript" src="js/index.js"></script>
9     <script type="text/javascript" src="js/print.js"></script></body>
10 </html>

```

这时候, 我们是时候想一下, 如果它能重构, 是否说明我之前的 index.html 页面可以不需要了? 删掉 index.html, 再运行"cnpm run build", OK, 相信你们已经提前知道结果了:
(打包前)



(打包后)



可是，我们知道，这终不是我们想要的，部门希望能像往常一样能正常编写 HTML，而不是通过 js 生成操作 DOM，所以，探索，远远还没结束……

2.5.5 动态刷新 dist

是时候刷个小 bug 了，在前面，当我们要重新打包的时候，为了方便观察 dist 目录的生成，我们一般都会删掉里面的内容（index.html 除外），然后才执行"cnpm run build"进行打包。这是不科学的！我们更希望它能在我们打包前先自行清空，然后我们就可以“偷懒”了
~

是的，这小节就讲这个：

- `cnpm i clean-webpack-plugin -D`

然后修改下 webpack 配置，使其能加载这个插件：

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');
```

```
module.exports = {
  entry: {
    index: './src/js/index.js',
    print: './src/js/print.js'
  },
  output: {
    filename: 'js/[name].js',
    path: path.resolve(__dirname, 'dist')
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: '管理输出'
    }),
    new CleanWebpackPlugin(['dist'])
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\..(jpg|jpeg|png|gif|svg)$/,
        use: [
          'url-loader?limit=10240&name=images/[name].[ext]'
        ]
      },
      {
        test: /\..(woff|woff2|eot|ttf|otf)$/,
        loader: 'url-loader',
        options: {
          limit: 10240,

```

```

        name: 'fonts/[name].[ext]'
      }
    },
    {
      test: /\. (csv|tsv) $/,
      use: [
        'csv-loader'
      ]
    },
    {
      test: /\. xml $/,
      use: [
        'xml-loader'
      ]
    }
  ]
}
};

```

最后执行下"cnpm run build", 眼尖的同学可以发现, webpack 真的按我们的要求, 先删除 dist 文件夹, 再重新构建过了哦~

2.6 开发

在本章节中, 我们讲一些开发时候会用到的很实用的小技巧~

2.6.1 查找报错文件

有时候, 我们想查看下 js 或者 css 哪里写错了, 但是, webpack 打包输出的文件, 我们查看不出来啊, 毕竟打包的代码跟编写的代码不一致!

首先, 修改 webpack 配置:

webpack.config.js

```

const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');

module.exports = {
  entry: {
    index: './src/js/index.js',
    print: './src/js/print.js'
  }
};

```

```

},
output: {
  filename: 'js/[name].js',
  path: path.resolve(__dirname, 'dist')
},
devtool: 'inline-source-map',
plugins: [
  new HtmlWebpackPlugin({
    title: '管理输出'
  }),
  new CleanWebpackPlugin(['dist'])
],
module: {
  rules: [
    {
      test: /\.css$/,
      use: [
        'style-loader',
        'css-loader'
      ]
    },
    {
      test: /\..(jpg|jpeg|png|gif|svg)$/,
      use: [
        'url-loader?limit=10240&name=images/[name].[ext]'
      ]
    },
    {
      test: /\..(woff|woff2|eot|ttf|otf)$/,
      loader: 'url-loader',
      options: {
        limit: 10240,
        name: 'fonts/[name].[ext]'
      }
    },
    {
      test: /\..(csv|tsv)$/,
      use: [
        'csv-loader'
      ]
    },
    {
      test: /\.xml$/,

```

```

    use: [
      'xml-loader'
    ]
  }
]
}
};

```

然后，我们修改下 `print.js` 的打印输出：

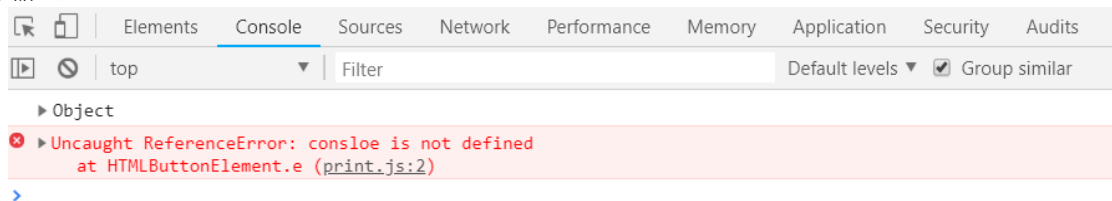
print.js

```

export default function printMe() {
  consloe.log('I get called from print.js!');
}

```

这里我们打错了个输出格式，将 `console` 写成 `consloe`，然后打包输出，查看点击后的控制台输出：



OK，这样我们就能发现错误是哪个文件哪行代码导致的了，但是这个方式记得是在开发的时候用的哦~

2.6.2 实时重新加载

学过 Vue 的小伙伴应该知道，Vue-Cli 有个很好用的命令行，叫 `cnpm run dev`，执行这个命令行后，我们就不用打包查看输出，可以直接在线观看它的输出了，如果搭配两个显示屏，那么就 happy 啦，一遍看输出一遍敲源码，更新了按下 `Ctrl+S` 就可以查看了。

所以，我们应该尝试下，在 `webpack` 中使用自动编译代码的方法，偷偷懒~

- `cnpm i webpack-dev-server -D`

然后，修改下 `webpack.config.js` 配置：

webpack.config.js

```

const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');

module.exports = {
  entry: {
    index: './src/js/index.js',
    print: './src/js/print.js'
  },

```

```

output: {
  filename: 'js/[name].js',
  path: path.resolve(__dirname, 'dist')
},
devServer: {
  contentBase: './dist'
},
plugins: [
  new HtmlWebpackPlugin({
    title: '管理输出'
  }),
  new CleanWebpackPlugin(['dist'])
],
module: {
  rules: [
    {
      test: /\.css$/,
      use: [
        'style-loader',
        'css-loader'
      ]
    },
    {
      test: /\..(jpg|jpeg|png|gif|svg)$/,
      use: [
        'url-loader?limit=10240&name=images/[name].[ext]'
      ]
    },
    {
      test: /\..(woff|woff2|eot|ttf|otf)$/,
      loader: 'url-loader',
      options: {
        limit: 10240,
        name: 'fonts/[name].[ext]'
      }
    },
    {
      test: /\..(csv|tsv)$/,
      use: [
        'csv-loader'
      ]
    }
  ]
}

```

```

        test: /\.xml$/,
        use: [
            'xml-loader'
        ]
    }
]
}
};

```

接着，往 `package.json` 添加"cnpm run dev"的命名行：

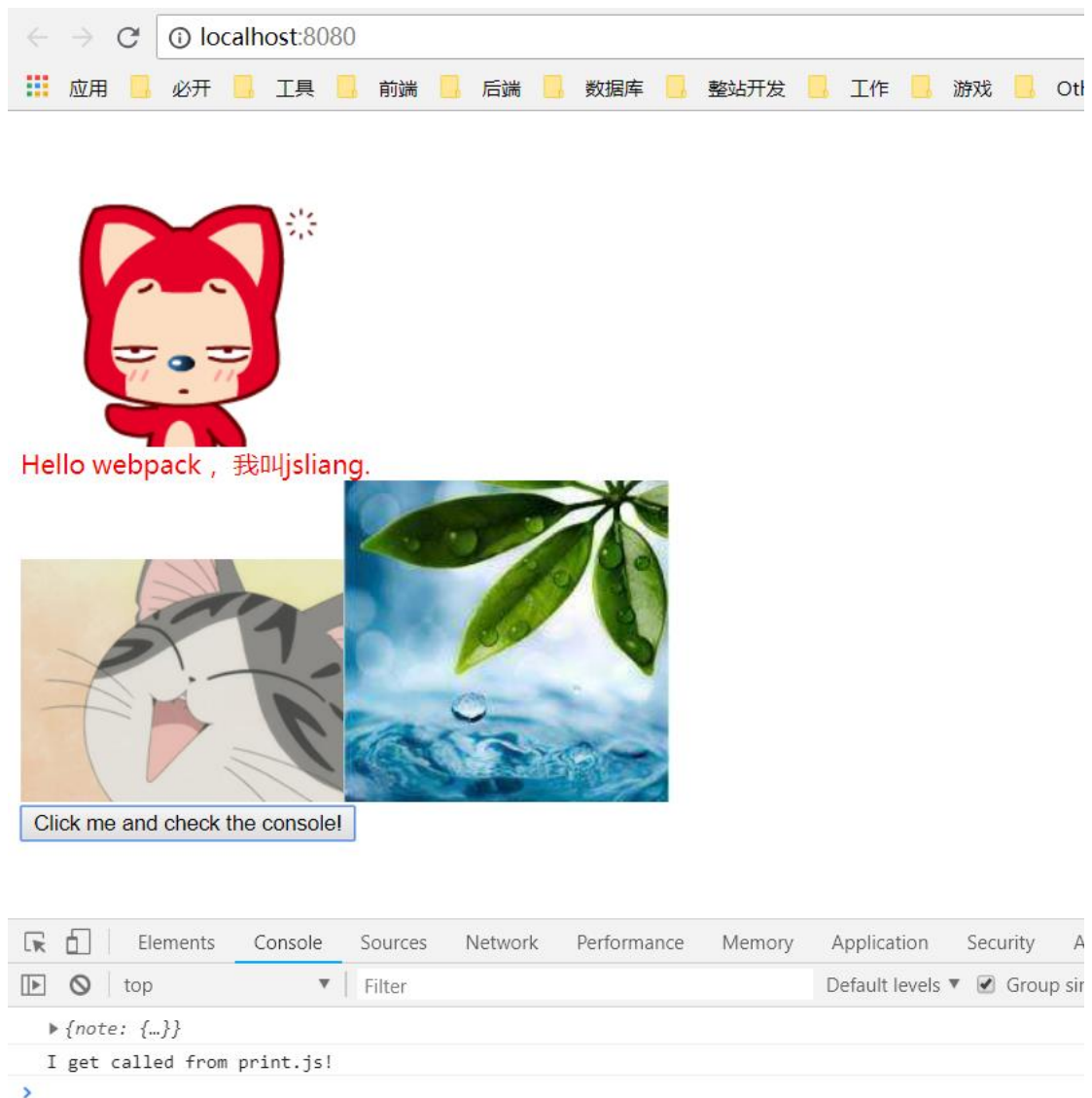
package.json

```

{
  "name": "webpack-4",
  "version": "1.0.0",
  "description": "jsliang webpack study",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack --mode production",
    "dev": "webpack-dev-server --open --mode development"
  },
  "keywords": [
    "webpack-study"
  ],
  "author": "jsliang",
  "license": "ISC",
  "devDependencies": {
    "clean-webpack-plugin": "^0.1.19",
    "css-loader": "^0.28.11",
    "csv-loader": "^2.1.1",
    "file-loader": "^1.1.11",
    "html-webpack-plugin": "^3.2.0",
    "style-loader": "^0.21.0",
    "url-loader": "^1.0.1",
    "webpack": "^4.12.1",
    "webpack-cli": "^3.0.8",
    "webpack-dev-server": "^3.1.4",
    "xml-loader": "^1.2.1"
  },
  "dependencies": {
    "lodash": "^4.17.10"
  }
}

```

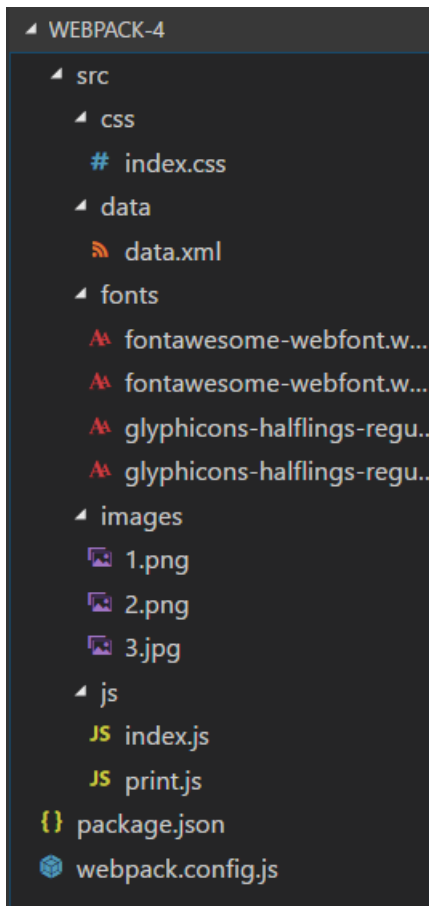

最后，我们执行下"`cnpm run dev`"，就可以看到自动新开了个 `localhost:8080` 页面，然后，我们修改下其他界面，它将自动刷新：



这下，我们开发的时候，就不用等打包了才能看到展示啦~

2.6.3 目录及代码整理

目录结构：



index.css

```
@font-face {
  font-family: 'MyFont';
  src: url('../fonts/fontawesome-webfont.woff') format('woff'),
       url('../fonts/fontawesome-webfont.woff2') format('woff2');
  font-weight: bold;
  font-style: normal;
}
```

```
.hello {
  color: red;
  background: url('../images/2.png') no-repeat;
  padding-top: 200px;
  font-family: 'MyFont';
}
```

data.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
<to>Mary</to>
<from>John</from>
```

```
<heading>Reminder</heading>
<body>Call Cindy on Tuesday</body>
</note>
```

fonts

fonts 目录下的字体文件请自行查找，格式为.woff 和.woff2，记得修改相应字体文件引用代码。

images

images 目录下的字体文件请自行查找，格式配置在 webpack.config.js 中，记得修改相应图片引用代码。

index.js

```
import _ from 'lodash';
import '../css/index.css';
import Gif1 from '../images/1.png';
import Image1 from '../images/3.jpg';
import Data from '../data/data.xml';
import printMe from './print.js';

function component() {
  var element = document.createElement('div');

  // 调用 Lodash 的_全局变量

  element.innerHTML = _.join(['Hello', 'webpack', ', 我叫 jsliang.<br>'], '
');

  element.classList.add('hello');

  // 将图片添加到 div 中

  var myGif = new Image();
  var myImage = new Image();
  myGif.src = Gif1;
  myImage.src = Image1;
  element.appendChild(myGif);
  element.appendChild(myImage);

  // 输出数据

  console.log(Data);

  // 新增 print.js
  var br = document.createElement('br');
```

```

    element.appendChild(br);
    var btn = document.createElement('button');
    btn.innerHTML = 'Click me and check the console!';
    btn.onclick = printMe;
    element.appendChild(btn);

    return element;
}

document.body.appendChild(component());

```

print.js

```

export default function printMe() {
  console.log('I get called from print.js!');
}

```

package.json

```

{
  "name": "webpack-4",
  "version": "1.0.0",
  "description": "jsliang webpack study",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack --mode production",
    "dev": "webpack-dev-server --open --mode development"
  },
  "keywords": [
    "webpack-study"
  ],
  "author": "jsliang",
  "license": "ISC",
  "devDependencies": {
    "clean-webpack-plugin": "^0.1.19",
    "css-loader": "^0.28.11",
    "csv-loader": "^2.1.1",
    "file-loader": "^1.1.11",
    "html-webpack-plugin": "^3.2.0",
    "style-loader": "^0.21.0",
    "url-loader": "^1.0.1",
    "webpack": "^4.12.1",
    "webpack-cli": "^3.0.8",
    "webpack-dev-server": "^3.1.4",
    "xml-loader": "^1.2.1"
  }
}

```

```

    },
    "dependencies": {
      "lodash": "^4.17.10"
    }
  }
}

```

webpack.config.js

```

const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');

```

```

module.exports = {
  entry: {
    index: './src/js/index.js',
    print: './src/js/print.js'
  },
  output: {
    filename: 'js/[name].js',
    path: path.resolve(__dirname, 'dist')
  },
  devServer: {
    contentBase: './dist'
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: '管理输出'
    }),
    new CleanWebpackPlugin(['dist'])
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\..(jpg|jpeg|png|gif|svg)$/,
        use: [
          'url-loader?limit=10240&name=images/[name].[ext]'
        ]
      }
    ]
  }
}

```

```

    },
    {
      test: /\.woff|woff2|eot|ttf|otf$/,
      loader: 'url-loader',
      options: {
        limit: 10240,
        name: 'fonts/[name].[ext]'
      }
    },
    {
      test: /\.csv|tsv$/,
      use: [
        'csv-loader'
      ]
    },
    {
      test: /\.xml$/,
      use: [
        'xml-loader'
      ]
    }
  ]
}
};

```

使用上面文件的方法：

- `cnpm i`
- `cnpm run build` - 打包
- `cnpm run dev` - 开发

看到这里，是不是对 GitHub 上 `download` 的项目的运行方式有个了解了呢~是的，就是这么出来的！

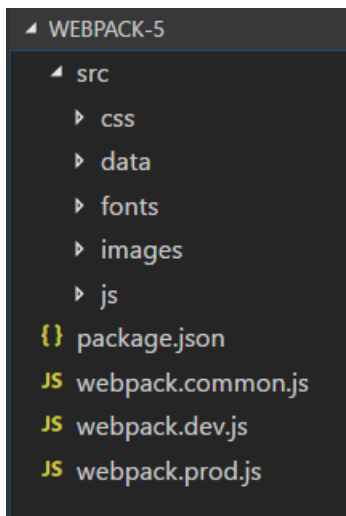
2.7 分离配置文件

经过上面一番折腾，是不是发觉，我们一开始觉得并不咋地的 `webpack.config.js` 在我们的不断更新下逐渐壮大了？这时候有木有一种冲动，将它切成小快快？！OK，那么我们接下来要做的，就是分离 `webpack.config.js`。

在上面的做法中，我们大概知道，`webpack.config.js` 有两种用途，一个是打包（生产）、一个是开发，所以，机智的我们，应该想到了，要把它切成三个部分：即共有、生产环境、开发环境，以方便到时候生产、开发的配置添加。

- `cnpm i webpack-merge -D`

安装完这个配置分离的工具后，我们将 `webpack.config.js` 分为 `webpack.common.js`、`webpack.dev.js`、`webpack.prod.js`，目录如下：



最后，将代码依序存放到不同位置：

webpack.common.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');
```

```
module.exports = {
  entry: {
    index: './src/js/index.js',
    print: './src/js/print.js'
  },
  output: {
    filename: 'js/[name].js',
    path: path.resolve(__dirname, 'dist')
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: '生产环境构建'
    }),
    new CleanWebpackPlugin(['dist'])
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
```

```

        'css-loader'
    ]
  },
  {
    test: /\. (jpg|jpeg|png|gif|svg)$/,
    use: [
      'url-loader?limit=10240&name=images/[name].[ext]'
    ]
  },
  {
    test: /\. (woff|woff2|eot|ttf|otf)$/,
    loader: 'url-loader',
    options: {
      limit: 10240,
      name: 'fonts/[name].[ext]'
    }
  },
  {
    test: /\. (csv|tsv)$/,
    use: [
      'csv-loader'
    ]
  },
  {
    test: /\.xml$/,
    use: [
      'xml-loader'
    ]
  }
]
}
};

```

webpack.dev.js

```

const merge = require('webpack-merge');
const common = require('./webpack.common.js');

module.exports = merge(common, {
  mode: 'development',
  devtool: 'inline-source-map',
  devServer: {
    contentBase: './dist'
  }
});

```


webpack.prod.js

```
const merge = require('webpack-merge');
const common = require('./webpack.common.js');

module.exports = merge(common, {
  mode: 'production'
});
```

同时，修改下 **package.json**，使命令行能启动对应的配置文件：

package.json

```
{
  "name": "webpack-5",
  "version": "1.0.0",
  "description": "jsliang webpack study",
  "main": "index.js",
  "scripts": {
    "build": "webpack --config webpack.prod.js",
    "dev": "webpack-dev-server --open --config webpack.dev.js"
  },
  "keywords": [
    "webpack-study"
  ],
  "author": "jsliang",
  "license": "ISC",
  "devDependencies": {
    "clean-webpack-plugin": "^0.1.19",
    "css-loader": "^0.28.11",
    "csv-loader": "^2.1.1",
    "file-loader": "^1.1.11",
    "html-webpack-plugin": "^3.2.0",
    "style-loader": "^0.21.0",
    "url-loader": "^1.0.1",
    "webpack": "^4.12.1",
    "webpack-cli": "^3.0.8",
    "webpack-dev-server": "^3.1.4",
    "webpack-merge": "^4.1.3",
    "xml-loader": "^1.2.1"
  },
  "dependencies": {
    "lodash": "^4.17.10"
  }
}
```

看到 package.json 的配置，结合 webpack.dev.js/webpack.prod.js 多出来的"mode"，聪明的小伙伴们是不是想到了什么？是的，我们把"mode"模式搬家了，这样我们就更清晰啦！

2.8 小结

经过上面的尝试与整理，估计很多小伙伴也是已经晕乎乎的啦：天呐！我看了那么长的文档，搞了那么多的 loaders 和 plugins，结果也只是入门？！jsliang 做回坏人告诉你：是的，这一切只是知识原始积累，还不到装逼的时刻！所以，整理下我们的代码，看看还有木有哪里不懂的，哪里需要修改的，接下来就该会会小怪兽 loaders 和 plugins 啦，打起架来 jsliang 可不会顾及你哦~

三、扬帆起航

大浪淘沙，千帆竞航，寻觅前辈足迹，开拓自己的 webpack4 之航道：

• Tiramisu_C

级别：☆

船只：<https://blog.csdn.net/nongweiyilady/article/details/79255746>

介绍：入门级 webpack 多页面配置

• Coffeer

级别：☆☆

船只：<https://github.com/Coffcer/Blog/issues/1>

介绍：正常级 webpack 多页面配置

下面主要介绍我再 webpack 多页面配置上使用到的 17 个 webpack 配置，仅供参考，有疑问的可以找 jsliang 提问

3.1 loaders

3.1.1 css-loader

作用：使用 import/require()形式解析@import 和 url()。在 jsliang 的 webpack4 小船中，该 loader 的作用是配合 mini-css-extract-plugin 将 css 提取打包到指定的 css 目录中。

直通车：<https://www.npmjs.com/package/css-loader>

3.1.2 style-loader

作用：官方解释：转换 JavaScript DOM 操作中出现的样式选择器为可用的 CSS 选择器；
个人解释：在 HTML 中以 style 的方式嵌入 css（即行内样式）。

注：因与 mini-css-extract-plugin 插件冲突，所以在配置上给 jsliang 关掉了，但是现在行内样式仍然可以正常使用。

直通车：<https://www.npmjs.com/package/style-loader>

3.1.3 less-loader

作用：能在项目中使用 less 来编写 css，将 less 转换为 css。在使用中，配合 mini-css-extract-plugin 将 less 转为 css 并打包到指定目录上。

直通车：<https://www.npmjs.com/package/less-loader>

3.1.4 file-loader

作用：将文件打包到指定 url 上。

直通车：<https://www.npmjs.com/package/file-loader>

3.1.5 url-loader

作用：像 file-loader 一样，不过可以将限制范围内的文件打包成 base64 插入到 JavaScript 中。但是在处理上，还是需要配合 file-loader 一起使用。

直通车：<https://www.npmjs.com/package/url-loader>

3.1.6 html-withimg-loader

作用：1、解决 HTML 上标签的 src 加载图片问题；2、可以引用子页面。

直通车：<https://www.npmjs.com/package/html-withimg-loader>

3.2 plugins

3.2.1 html-webpack-plugin

作用：根据 JavaScript 创建生成一个 HTML 文件，然后 JavaScript 会被<script>引用，如

果需要引用 css, 需要使用 ExtactTextWebpackPlugin 插件(Webpack4 使用 MiniCssExtractPlugin 插件)。

直通车: <https://www.npmjs.com/package/html-webpack-plugin>

3.2.2 clean-webpack-pugin

作用: 打包 webpack 前先清理指定文件夹。

直通车: <https://www.npmjs.com/package/clean-webpack-plugin>

3.2.3 mini-css-extract-plugin

作用:

将入口引用的*.css 文件, 移动到独立分离的 CSS 文件。

可以与 uglifyjs-webpack-plugin、optimize-css-assets-webpack-plugin 一起使用, 创建压缩 JS、CSS 的环境。

可以将所有*.css 文件提取到一个集合 css 文件 all.css 中。

注: 在 webpack3 及以下版本中为 extract-text-webpack-plugin。

直通车:

<https://www.npmjs.com/package/extract-text-webpack-plugin>

<https://www.npmjs.com/package/mini-css-extract-plugin>

解疑:

- filename 与 chunkFilename: <https://www.jianshu.com/p/d9ebab57bca1>

3.2.4 uglifyjs-webpack-plugin

作用: 压缩 JavaScript。cache: 启动文件缓存; parallel: 使用多进程运行; sourceMap: 检查报错(注: cheap-source-map 与该功能冲突)。

直通车: <https://www.npmjs.com/package/uglifyjs-webpack-plugin>

3.2.5 optimize-css-assets-webpack-plugin

作用: 优化、压缩 CSS。

直通车: <https://www.npmjs.com/package/optimize-css-assets-webpack-plugin>

3.3 other npm

3.3.1 less

作用：能在项目中使用 less，然后因为 less-loader 需要配合这个才能使用，所以需要安装 less。

直通车：<https://github.com/less/less.js>

3.3.2 webpack

作用：打包构建工具，本文主角，不多介绍。

直通车：<https://github.com/webpack/webpack>

3.3.3 webpack-cli

作用：webpack 脚手架，配合 webpack 一起使用，效果更佳~

直通车：<https://github.com/webpack/webpack-cli>

3.3.4 webpack-dev-server

作用：webpack 服务，能够在开发的时候起个服务器，并且设置后能即时重载代码，在浏览器直接看到改变。

直通车：<https://github.com/webpack/webpack-dev-server>

3.3.5 webpack-merge

作用：根据生产环境和开发环境分离 webpack.config.js 为 webpack.common.js、webpack.dev.js、webpack.prod.js，不会混淆开发和生产的配置。

直通车：<https://github.com/survivejs/webpack-merge>

3.3.6 glob

作用：通过该模块循环导出入口和插入 HtmlWebpackPlugin

直通车：<https://www.npmjs.com/package/glob>