
目錄

0. 开始	1.1
1. 介绍	1.2
2. 安装	1.3
3. 实现 hello world	1.4
4. webpack 的配置文件 webpack.config.js	1.5
5. 使用第一个 webpack 插件 html-webpack-plugin	1.6
6. 使用 loader 处理 CSS 和 Sass	1.7
7. 初识 webpack-dev-server	1.8
8. 用 webpack 和 babel 配置 react 开发环境	1.9
9. 用 clean-webpack-plugin 来清除文件	1.10
10. 配置多个 HTML 文件	1.11
11. 如何使用 pug (jade) 作为 HTML 的模板	1.12

webpack 3 零基础入门教程

最详细，最简单的零基础 webpack 3 入门教程，人人都能学会。

原文发布于我的个人博客：<https://www.rails365.net>

源码位于：<https://github.com/yinsigan/webpack-tutorial>

电子版: [PDF](#) [Mobi](#) [ePbu](#)

联系我:

email: hfpp2012@gmail.com

qq: 903279182

现在我们要先把 webpack 用起来。

为了方便，我们先用 npm 创建一个空项目。

1. 用 npm init 初始化项目

打开终端，运行如下命令：

```
# 随便进一个目录
$ cd ~/codes
# 创建一个存放 webpack 项目的目录，名为 hello-webpack
$ mkdir hello-webpack
$ npm init
```

之后你会看到会提示你输入一些内容，你不用管，直接全部回车：

```
name: (hello-wepback)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
```

最后，你会发现 `hello-webpack` 目录下多出了一个名为 `package.json` 的文件。

它的内容如下：

```
{
  "name": "hello-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

整个 `json` 文件的内容很简单，主要是显示这个项目的名称、版本、作者、协议等信息，一看就能很清晰。

具体的信息这里我们先按下不表，以后我们会跟这个文件经常打交道的。

2. 集成 webpack

现在项目是空的，没有任何东西，我们现在需要把 `webpack` 集成进来，让这个项目可以用这个 `webpack`。

我们在终端上输入如下命令：

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

你会看到正在安装 `webpack` 的进度，稍等片刻，成功之后，我们再来看看 `package.json` 这个文件的内容。

```
{
  "name": "hello-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^3.8.1"
  }
}
```

多了下面这几行：

```
"devDependencies": {
  "webpack": "^3.8.1"
}
```

什么意思呢？就是说，我现在这个项目依赖于 **webpack** 这个工具，就算以后别人得到这个 `package.json` 文件，就会知道要安装这个 **webpack** 了。

同时你也会发现，多了一个目录，叫 `node_modules`，这个就是放刚才安装的 **webpack** 这个库的所有要用到的源码文件。

这点先了解这么多，其实知道大概的意思就行。

3. 创建 javascript 文件

现在项目创建了，**webpack** 也集成到项目中了，该是要把 **webpack** 玩起来的时候了。

首先创建一个目录叫 `src`，然后在该目录下创建 **javascript** 文件，叫 `app.js`。

这个文件的内容最简单，就输出 `hello world`，如下所示：

```
console.log('hello world');
```

4. 把 webpack 用起来

现在要把刚才的 js 文件用 webpack 编译一下。

编译后输出的文件，我们放到一个目录中，就叫 `dist` 好了。

先把这个目录创建好。

现在的目录结构是这样的：

```
.
├── dist
├── node_modules
├── package.json
└── src
    └── app.js
```

OK，现在开始转化，在终端上输入如下命令：

```
$ webpack ./src/app.js ./dist/app.bundle.js
```

意思就是说，把 `./src/app.js` 作为源文件，把转化后的结果放到 `./dist/app.bundle.js` 文件中。

下面是输出的结果：

```
→ hello-webpack webpack ./src/app.js ./dist/app.bundle.js
Hash: 6d1bc92c3f8f43db9268
Version: webpack 3.8.1
Time: 69ms

   Asset      Size  Chunks             Chunk Names
app.bundle.js  2.5 kB      0  [emitted]      main
   [0] ./src/app.js 28 bytes {0} [built]
```

成功了！

果然，在 `dist` 目录下生成了 `app.bundle.js` 文件。

它的内容如下：

```
46 /*****/
47 /*****/ // getDefaultExport function for compatibility with non-harmony modules
48 /*****/ __webpack_require__.n = function(module) {
49 /*****/   \var getter = module && module.__esModule ?
50 /*****/     \function getDefault() { return module['default']; } :
51 /*****/     \function getModuleExports() { return module; };
52 /*****/   \__webpack_require__.d(getter, 'a', getter);
53 /*****/   \return getter;
54 /*****/ }
55 /*****/
56 /*****/ // Object.prototype.hasOwnProperty.call
57 /*****/ __webpack_require__.o = function(object, property) { return Object.prototype.hasOwnProperty.call(object, property); };
58 /*****/
59 /*****/ // __webpack_public_path__
60 /*****/ __webpack_require__.p = "";
61 /*****/
62 /*****/ // Load entry module and return exports
63 /*****/ return __webpack_require__(__webpack_require__.s = 0);
64 /*****/ })
65 /*****/
66 /*****/ (function(module, exports) {
67 /*****/   * 0 */
68 /*****/   (function(module, exports) {
69
70 console.log("hello world");
71
72
73 /*****/ })
74 /*****/ })
```

我们的源码被包含进来了

大约总共有 74 行，大小约是 2.5 kb，可以看出多出了不少东西，但至少 `hello world` 那一行源码被包含进来了。

具体的内容，你打开转化后的文件看看就知道了。

5 webpack 的其他用法

上面介绍的只是 webpack 一个最简单的功能，它可不止有这个用法，还有其他的，我们来介绍一下。

5.1 --watch

首先，在开发环境中，总是要一边改，一边看转化效果吧，webpack 也能办到，多加一个参数就好。

```
$ webpack --watch ./src/app.js ./dist/app.bundle.js
```

输出效果：

```
hello-webpack webpack --watch ./src/app.js ./dist/app.bundle.js
Webpack is watching the files...
Hash: 6d1bc92c3f8f43db9268
Version: webpack 3.8.1
Time: 65ms

   Asset      Size  Chunks             Chunk Names
app.bundle.js  2.5 kB      0  [emitted]      main
[0] ./src/app.js 28 bytes {0} [built]
```

表示正在监听...

现在去改改 `src/app.js` 文件的内容，试试效果，看看 `dist/app.bundle.js` 是否实时变化了。

5.2 -p

之前转化的 `app.bundle.js` 文件大约有 74 行代码，差不多 2k 多的大小，好大啊，毕竟我们的代码只有一行而已。

在生产环境，或线上，我们肯定不希望这么大的体积，毕竟体积越大，带宽浪费就越多呀，下载也越慢。

如果要发布到线上环境，我们要把它压缩一下的。

而 `webpack` 本来就有这样的功能，也只是一个参数 `-p`

```
$ webpack -p ./src/app.js ./dist/app.bundle.js
```

输出如下：

```
hello-webpack webpack -p ./src/app.js ./dist/app.bundle.js
Hash: 6d1bc92c3f8f43db9268
Version: webpack 3.8.1
Time: 113ms

   Asset      Size  Chunks             Chunk Names
app.bundle.js  505 bytes      0  [emitted]      main
[0] ./src/app.js 28 bytes {0} [built]
```

可以看到，输出的文件相比以前的 2.5 kb 小了一些，大约 505 个字节。

我们打开来看下：

```
1 function(e){function n(t){if(r[t])return r[t].exports;var o=r[t]={i:t,l:1,exports:{}};return e[t].call(o.exports,o,o.exports,n),o.l=!0,o.exports=r,n.d=function(e,r,t){Object.defineProperty(e,r,{configurable:1,enumerable:1,get:t})},n.n=function(e){var r=e.__esModule?function(){return e};return n.d(r,"a",r),n.o=function(e,n){return Object.prototype.hasOwnProperty.call(e,n)},n.p="",n(n.s-0)}([function(e,n){}]);
```


好了，先说这么多吧。

1. 安装 nodejs

要使用 webpack，必须确保电脑上存在着 nodejs 这个运行环境，所以，如果没有 nodejs，要先安装它。

在浏览器输入下面的网址：

<https://nodejs.org/en/>

界面如下：

Node.js® is a JavaScript runtime built on **Chrome's V8 JavaScript engine**. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, **npm**, is the largest ecosystem of open source libraries in the world.

Important **security release for 8.x**, please update now!

Download for macOS (x64)

v6.11.3 LTS

Recommended For Most Users

v8.6.0 Current

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#)

[Other Downloads](#) | [Changelog](#) | [API Docs](#)

下载安装包后，点击可执行文件，不断地按下一步，就可以安装成功。

最后打开命令行终端，输入以下命令。

```
$ node -v
```

应该会显示所安装的 nodejs 的版本，比如我的：

```
v7.8.0
```

这样就表示电脑上有 nodejs 的环境了。

2. 安装 webpack

2. 安装

下一步，就可以安装 webpack 了。

在命令行终端上输入以下命令：

```
$ npm install -g webpack
```

npm 是 nodejs 管理插件用的工具，install 表示安装，-g 表示全局安装，这样会把可执行文件 webpack 放到 bin 目录下，以后就可以直接运行 webpack 目录了。

我们来检测一下是否把 webpack 安装成功了。

在终端上输入以下命令：

```
webpack -v
```

输出 webpack 的版本如下：

```
3.6.0
```

值得一提的是，如果用国内的网络安装 npm 包会很慢的话，可以考虑用一下淘宝的源，具体可以看 <https://npm.taobao.org/> 或 <https://github.com/Pana/nrm>

现在我们要先把 webpack 用起来。

为了方便，我们先用 npm 创建一个空项目。

1. 用 npm init 初始化项目

打开终端，运行如下命令：

```
# 随便进一个目录
$ cd ~/codes
# 创建一个存放 webpack 项目的目录，名为 hello-webpack
$ mkdir hello-webpack
$ npm init
```

之后你会看到会提示你输入一些内容，你不用管，直接全部回车：

```
name: (hello-wepback)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
```

最后，你会发现 `hello-webpack` 目录下多出了一个名为 `package.json` 的文件。

它的内容如下：

```
{
  "name": "hello-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

整个 `json` 文件的内容很简单，主要是显示这个项目的名称、版本、作者、协议等信息，一看就能很清晰。

具体的信息这里我们先按下不表，以后我们会跟这个文件经常打交道的。

2. 集成 webpack

现在项目是空的，没有任何东西，我们现在需要把 `webpack` 集成进来，让这个项目可以用这个 `webpack`。

我们在终端上输入如下命令：

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

你会看到正在安装 `webpack` 的进度，稍等片刻，成功之后，我们再来看看 `package.json` 这个文件的内容。

```
{
  "name": "hello-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^3.8.1"
  }
}
```

多了下面这几行：

```
"devDependencies": {
  "webpack": "^3.8.1"
}
```

什么意思呢？就是说，我现在这个项目依赖于 **webpack** 这个工具，就算以后别人得到这个 `package.json` 文件，就会知道要安装这个 **webpack** 了。

同时你也会发现，多了一个目录，叫 `node_modules`，这个就是放刚才安装的 **webpack** 这个库的所有要用到的源码文件。

这点先了解这么多，其实知道大概的意思就行。

3. 创建 javascript 文件

现在项目创建了，**webpack** 也集成到项目中了，该是要把 **webpack** 玩起来的时候了。

首先创建一个目录叫 `src`，然后在该目录下创建 **javascript** 文件，叫 `app.js`。

这个文件的内容最简单，就输出 `hello world`，如下所示：

```
console.log('hello world');
```

4. 把 webpack 用起来

现在要把刚才的 js 文件用 webpack 编译一下。

编译后输出的文件，我们放到一个目录中，就叫 `dist` 好了。

先把这个目录创建好。

现在的目录结构是这样的：

```
.
├── dist
├── node_modules
├── package.json
└── src
    └── app.js
```

OK，现在开始转化，在终端上输入如下命令：

```
$ webpack ./src/app.js ./dist/app.bundle.js
```

意思就是说，把 `./src/app.js` 作为源文件，把转化后的结果放到 `./dist/app.bundle.js` 文件中。

下面是输出的结果：

```
→ hello-webpack webpack ./src/app.js ./dist/app.bundle.js
Hash: 6d1bc92c3f8f43db9268
Version: webpack 3.8.1
Time: 69ms

   Asset      Size  Chunks             Chunk Names
app.bundle.js  2.5 kB      0  [emitted]      main
   [0] ./src/app.js 28 bytes {0} [built]
```

成功了！

果然，在 `dist` 目录下生成了 `app.bundle.js` 文件。

它的内容如下：

```
46 /*****/
47 /*****/ // getDefaultExport function for compatibility with non-harmony modules
48 /*****/ __webpack_require__.n = function(module) {
49 /*****/   \var getter = module && module.__esModule ?
50 /*****/     \function getDefault() { return module['default']; } :
51 /*****/     \function getModuleExports() { return module; };
52 /*****/   \__webpack_require__.d(getter, 'a', getter);
53 /*****/   \return getter;
54 /*****/ }
55 /*****/
56 /*****/ // Object.prototype.hasOwnProperty.call
57 /*****/ __webpack_require__.o = function(object, property) { return Object.prototype.hasOwnProperty.call(object, property); };
58 /*****/
59 /*****/ // __webpack_public_path__
60 /*****/ __webpack_require__.p = "";
61 /*****/
62 /*****/ // Load entry module and return exports
63 /*****/ return __webpack_require__(__webpack_require__.s = 0);
64 /*****/ })
65 /*****/
66 /*****/ ([
67 /*****/   * 0 */
68 /*****/   (function(module, exports) {
69
70     console.log("hello world");
71
72
73 /*****/   })
74 /*****/ ]);
```

我们的源码被包含进来了

大约总共有 74 行，大小约是 2.5 kb，可以看出多出了不少东西，但至少 `hello world` 那一行源码被包含进来了。

具体的内容，你打开转化后的文件看看就知道了。

5 webpack 的其他用法

上面介绍的只是 webpack 一个最简单的功能，它可不止有这个用法，还有其他的，我们来介绍一下。

5.1 --watch

首先，在开发环境中，总是要一边改，一边看转化效果吧，webpack 也能办到，多加一个参数就好。

```
$ webpack --watch ./src/app.js ./dist/app.bundle.js
```

输出效果：


```
→ hello-webpack webpack --watch ./src/app.js ./dist/app.bundle.js
Webpack is watching the files...
Hash: 6d1bc92c3f8f43db9268
Version: webpack 3.8.1
Time: 65ms
   Asset      Size  Chunks             Chunk Names
app.bundle.js  2.5 kB      0  [emitted]  main
[0] ./src/app.js 28 bytes {0} [built]
```

表示正在监听...

现在去改改 `src/app.js` 文件的内容，试试效果，看看 `dist/app.bundle.js` 是否实时变化了。

5.2 -d

之前转化的 `app.bundle.js` 文件大约有 74 行代码，差不多 2k 多的大小，好大啊，毕竟我们的代码只有一行而已。

在生产环境，或线上，我们肯定不希望这么大的体积，毕竟体积越大，带宽浪费就越多呀，下载也越慢。

如果要发布到线上环境，我们要把它压缩一下的。

而 `webpack` 本来就有这样的功能，也只是一个参数 `-p`

```
$ webpack -p ./src/app.js ./dist/app.bundle.js
```

输出如下：

```
→ hello-webpack webpack -p ./src/app.js ./dist/app.bundle.js
Hash: 6d1bc92c3f8f43db9268
Version: webpack 3.8.1
Time: 113ms
   Asset      Size  Chunks             Chunk Names
app.bundle.js  505 bytes      0  [emitted]  main
[0] ./src/app.js 28 bytes {0} [built]
```

可以看到，输出的文件相比以前的 2.5 kb 小了一些，大约 505 个字节。

我们打开来看下：

```
1 !function(e){function n(t){if(r[t])return r[t].exports;var o=r[t]={i:t,l:1,exports:{}};return e[t].call(o.exports,o,o.exports,n),o.l=!0,o.exports=n.d=function(e,r,t){n.o(e,r)||Object.defineProperty(e,r,{configurable:1,enumerable:1,get:t})},n.n=function(e){var r=e.__esModule?function(){return e};return n.d(r,"a",r),n.o=function(e,n){return Object.prototype.hasOwnProperty.call(e,n)},n.p="",n(n.s-0)}([function(e,n){}]);
```

好了，先说这么多吧。

在命令行中运行 `webpack` 命令确实可以实现 `webpack` 的功能，但是我们一般不这么做，我们要用配置文件来处理。

我们把之前学到的知识用 `webpack` 的配置文件来实现，配置文件的名字叫 `webpack.config.js` 位于项目根目录下。

1. 创建配置文件 `webpack.config.js`

它的内容如下：

```
module.exports = {  
  entry: './src/app.js',  
  output: {  
    filename: './dist/app.bundle.js'  
  }  
};
```

简单解释一下：`entry` 表示源文件，`output` 这边表示的是输出的目标文件。

很简单吧！

那怎么用呢？

直接在终端上输入 `webpack` 就可以了。`webpack` 命令会去找 `webpack.config.js` 文件，并读取它的内容（源文件和目标文件），最后进行相应的处理。

如下所示：

```
→ hello-webpack webpack
Hash: c9acd94ee8cce1cf12d2
Version: webpack 3.8.1
Time: 64ms

   Asset      Size  Chunks             Chunk Names
./dist/app.bundle.js  2.5 kB      0  [emitted]  main
  [0] ./src/app.js 28 bytes {0} [built]
→ hello-webpack webpack -p
Hash: c9acd94ee8cce1cf12d2
Version: webpack 3.8.1
Time: 113ms

   Asset      Size  Chunks             Chunk Names
./dist/app.bundle.js  505 bytes      0  [emitted]  main
  [0] ./src/app.js 28 bytes {0} [built]
→ hello-webpack webpack --watch

Webpack is watching the files...

Hash: c9acd94ee8cce1cf12d2
Version: webpack 3.8.1
Time: 68ms

   Asset      Size  Chunks             Chunk Names
./dist/app.bundle.js  2.5 kB      0  [emitted]  main
  [0] ./src/app.js 28 bytes {0} [built]
```

当然，`webpack.config.js` 的内容不止这么简单，可以更复杂些，我们以后再介绍。

2. 改造 package.json 的 scripts 部分

还记得上次说过的 `package.json` 这个文件吗？它主要放了一些项目的介绍信息，除此之外，它还要一个重要的功能。

就是可以放一些常用的命令行脚本，比如我们可以把我们经常要用的 `webpack` 命令放到这里来。

我把它改了一下，变成类似下面这样：

```
{
  "name": "hello-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev": "webpack -d --watch",
    "prod": "webpack -p"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^3.8.1"
  }
}
```

改动的内容主要是增加了下面几行：

```
"scripts": {
  "dev": "webpack -d --watch",
  "prod": "webpack -p"
},
```

怎么用呢？

很简单，分别是

```
$ npm run dev
```

和

```
$ npm run prod
```

你会发现 `npm run dev` 和 `webpack -d --watch` 的效果是一样的。

`-d` 这个参数之前没介绍过，它的意思就是说包含 `source maps`，这个有什么用呢，就是让你在用浏览器调试的时候，可以很方便地定位到源文件，知道这个意思就好了，不用深究太多。

你会想，为什么要用 `package.json` 的 `scripts` 功能呢？

我觉得主要有两个原因吧：

第一：简单维护，所有的命令都放一起了，也能方便地查看

第二：别人下载了你的源码，一查看 `package.json` 就能知道怎么运行这个项目。

先说这么多。

之前我们已经可以转化 js 文件了，但是一般来说，我们放在网页上的是 html 页面。

现在我们就把 html 和 js 还有 webpack 结合起来玩玩。

很简单，只要把 js 文件引入到 html 中就好。

1. 创建 index.html

首先在 `dist` 目录下创建 `index.html` 文件，其内容如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Project</title>
</head>
<body>
  <script src="app.bundle.js"></script>
</body>
</html>
```

这样，你在服务器上把这个 `index.html` 和 `app.bundle.js` 放到网站根目录中，用 `nginx` 托管起来，就可以用了。

前端的项目不就是这样处理的吗？

但是，我一般不会这么做，我会用更好的方式来处理这个问题。

为什么呢？

因为 `index.html` 文件太死了，连 js 文件都写死了，有时候引用的 js 文件是动态变化的呢？

打个比方，类似下面这种例子：

```
<script src="app.bundle1.js"></script>
<script src="app.bundle2.js"></script>
<script src="app.bundle3.js"></script>
```

而且还不确定有多少个。

还有一种情况，有时候为了更好的 `cache` 处理，文件名还带着 `hash`，例如下面这样：

```
main.9046fe2bf8166cbe16d7.js
```

这个 `hash` 是文件的 `md5` 值，随着文件的内容而变化，你总不能每变化一次，就改一下 `index.html` 文件吧？

效率太低！

下面我们要使用一个 `webpack` 的插件 `html-webpack-plugin` 来更好的处理这个问题。

2. html-webpack-plugin

`webpack` 吸引人的地方就是因为它有太多的插件，有时候一些需求，一个插件就搞定。

这么多插件，我们不可能全都学，全都用，要用也是找最好的，最常用的来玩，而且学了一个，其他的也差不多，掌握方法就好。

学习插件的第一步，是进入其主页，先把它的 `readme` 文档看看，至少知道它是干什么的，能解决什么问题，最后知道如何用就行了。

2.1 安装

先来安装，一条命令就好。

```
$ npm install html-webpack-plugin --save-dev
```

安装成功后，`package.json` 这个文件会多出一行 `"html-webpack-plugin": "^2.30.1"`，如下所示：


```
{
  "name": "hello-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev": "webpack -d --watch",
    "prod": "webpack -p"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "html-webpack-plugin": "^2.30.1",
    "webpack": "^3.8.1"
  }
}
```

2.2 使用

现在我们把之前的 `dist/index.html` 先删除掉，我们要用 `html-webpack-plugin` 这个插件来自动生成它。

把 `webpack.config.js` 文件改一下，如下所示：

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

module.exports = {
  entry: './src/app.js',
  output: {
    path: __dirname + '/dist',
    filename: 'app.bundle.js'
  },
  plugins: [new HtmlWebpackPlugin()]
};
```

最后，运行一下上文所说的 `npm run dev` 命令，你会发现现在 `dist` 目录生成了 `index.html` 文件，打开来看下。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Webpack App</title>
  </head>
  <body>
    <script type="text/javascript" src="app.bundle.js"></script></body>
</html>
```

连标题 `<title>Webpack App</title>` 都自动生成了，如果这是固定的话，就不太灵活，但是 `html-webpack-plugin` 有选项来处理这个问题。

3. 更好的使用 `html-webpack-plugin`

要改变 `title` 很简单，上文提到 `HtmlWebpackPlugin` 这个方法可以传入很多参数的，下面这样就可以解决这个问题。

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

module.exports = {
  entry: './src/app.js',
  output: {
    path: __dirname + '/dist',
    filename: 'app.bundle.js'
  },
  plugins: [new HtmlWebpackPlugin({
    title: "hello world"
  })]
};
```

再去看看新生成的 `index.html` 文件，是不是变化了。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>hello world</title>
  </head>
  <body>
    <script type="text/javascript" src="app.bundle.js"></script></body>
</html>
```

只是改变了一点点东西，其实也没多大用处，有时候我们要让 `index.html` 根据我们的意愿来生成。就是说它的内容是我们自己定的。

这个就不得不提到 `html-webpack-plugin` 的 `template` 功能。

把 `webpack.config.js` 更改如下：

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

module.exports = {
  entry: './src/app.js',
  output: {
    path: __dirname + '/dist',
    filename: 'app.bundle.js'
  },
  plugins: [new HtmlWebpackPlugin({
    template: './src/index.html',
  })]
};
```

接着新建 `src/index.html` 文件，内容如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Hello World</title>
</head>
<body>
</body>
</html>
```

我们再来看看新生成的 `dist/index.html` 文件，内容如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Hello World</title>
</head>
<body>
<script type="text/javascript" src="app.bundle.js"></script></bo
dy>
</html>
```

下面我再来介绍几个参数，以及它的结果。

`filename: 'index.html'` 默认情况下生成的 html 文件叫 `index.html`，但有时候你不想叫这个名字，可以改。

```
minify: {
  collapseWhitespace: true,
},
```

这个可以把生成的 `index.html` 文件的内容的没用空格去掉，减少空间。

效果如下：

```
1 <!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>Hello World</title></head><body><script type="text/javascript" src="app.bundle.js"></script></html>
```

`hash: true` 为了更好的 cache，可以在文件名后加个 hash。（这点不明白的先跳过）

效果如下：



最后的 `webpack.config.js` 内容大约是下面这样的：

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

module.exports = {
  entry: './src/app.js',
  output: {
    path: __dirname + '/dist',
    filename: 'app.bundle.js'
  },
  plugins: [new HtmlWebpackPlugin({
    template: './src/index.html',
    filename: 'index.html',
    minify: {
      collapseWhitespace: true,
    },
    hash: true,
  })]
};
```

`html-webpack-plugin` 肯定还有更多的用法和选项，这个只能根据需要慢慢探究了。

先说这么多。

1. 什么是 loader

官方的解释是这样的：

loader 用于对模块的源代码进行转换。loader 可以使你在 import 或"加载"模块时预处理文件。因此，loader 类似于其他构建工具中“任务(task)”，并提供了处理前端构建步骤的强大方法。loader 可以将文件从不同的语言（如 TypeScript）转换为 JavaScript，或将内联图像转换为 data URL。loader 甚至允许你直接在 JavaScript 模块中 import CSS 文件！

可能会一脸懵懂吧。

说白了，就是 loader 类似于 task，能够处理文件，比如把 Scss 转成 CSS，TypeScript 转成 JavaScript 等。

再不明白的话，还是用实例来说明吧。（其实它的概念并不重要，你会用就行）

2. 用 css-loader 和 style-loader 处理 CSS

现在我们来演示一下如何用 loader 来处理 CSS 文件。

先准备好内容。

src/app.css

```
body {  
  background: pink;  
}
```

src/app.js

```
import css from './app.css';  
  
console.log("hello world");
```

如果你现在运行 npm run dev 或 webpack 命令，就会出现类似下面的提示错误。

```
→ hello-webpack webpack
Hash: e5f147c724c0a042b7a5
Version: webpack 3.8.1
Time: 632ms

   Asset      Size  Chunks             Chunk Names
app.bundle.js  2.78 kB          0  [emitted]  main
index.html    193 bytes             [emitted]
[0] ./src/app.js 63 bytes {0} [built]
[1] ./src/app.css 163 bytes {0} [built] [failed] [1 error]

ERROR in ./src/app.css
Module parse failed: Unexpected token (1:5)
You may need an appropriate loader to handle this file type.
| body {
|   background: pink;
| }
|
@ ./src/app.js 1:12-32
Child HTML Webpack Plugin for "index.html":
  1 asset
    [0] ./node_modules/html-webpack-plugin/lib/loader.js!./src/index.html 490 bytes {0} [built]
    [2] (webpack)/buildin/global.js 488 bytes {0} [built]
    [3] (webpack)/buildin/module.js 495 bytes {0} [built]
    + 1 hidden module
```

这里有错误

意思就是说，默认情况下，webpack 处理不了 CSS 的东西。

我们来处理这个问题。

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

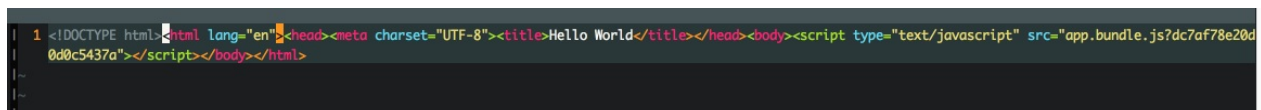
webpack.config.js

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

module.exports = {
  entry: './src/app.js',
  output: {
    path: __dirname + '/dist',
    filename: 'app.bundle.js'
  },
  plugins: [new HtmlWebpackPlugin({
    template: './src/index.html',
    filename: 'index.html',
    minify: {
      collapseWhitespace: true,
    },
    hash: true,
  })],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [ 'style-loader', 'css-loader' ]
      }
    ]
  }
};
```

我们来看下效果：

dist/index.html

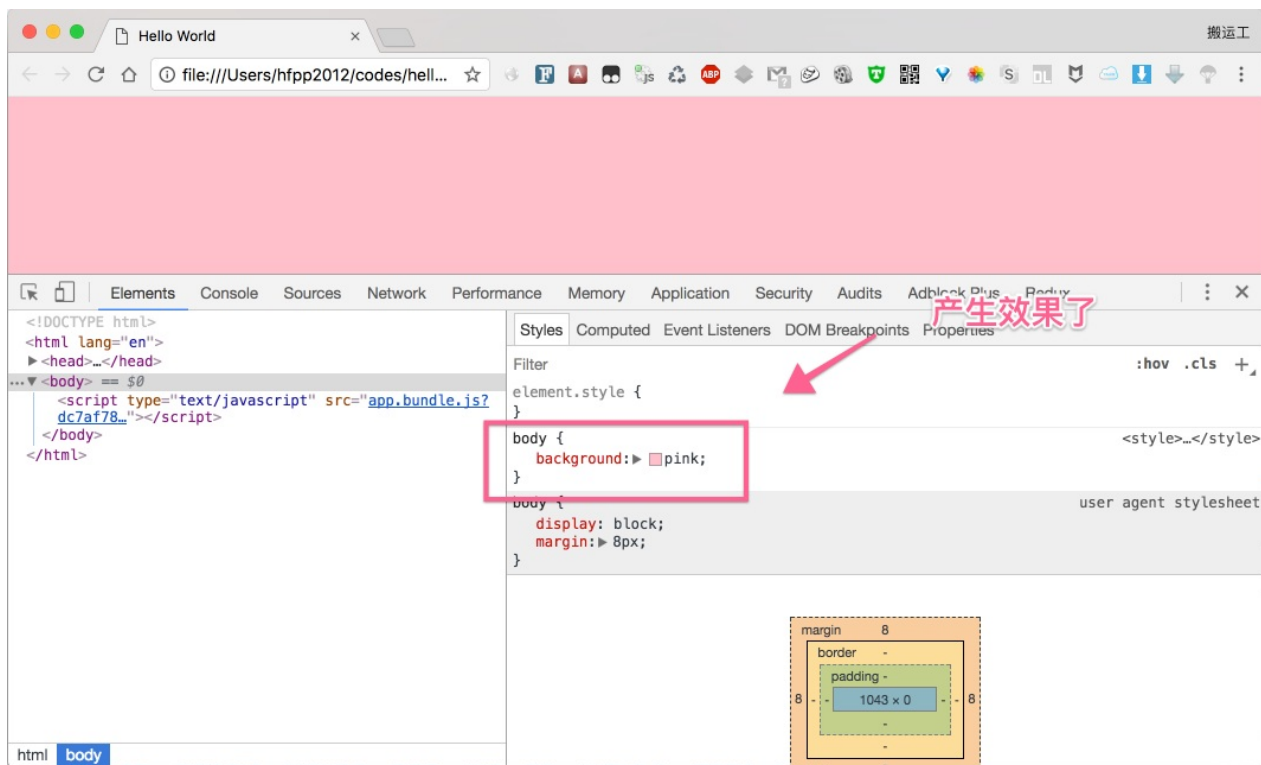


```
1 <!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>Hello World</title></head><body><script type="text/javascript" src="app.bundle.js?dc7af78e20d0d0c5437a"></script></body></html>
```

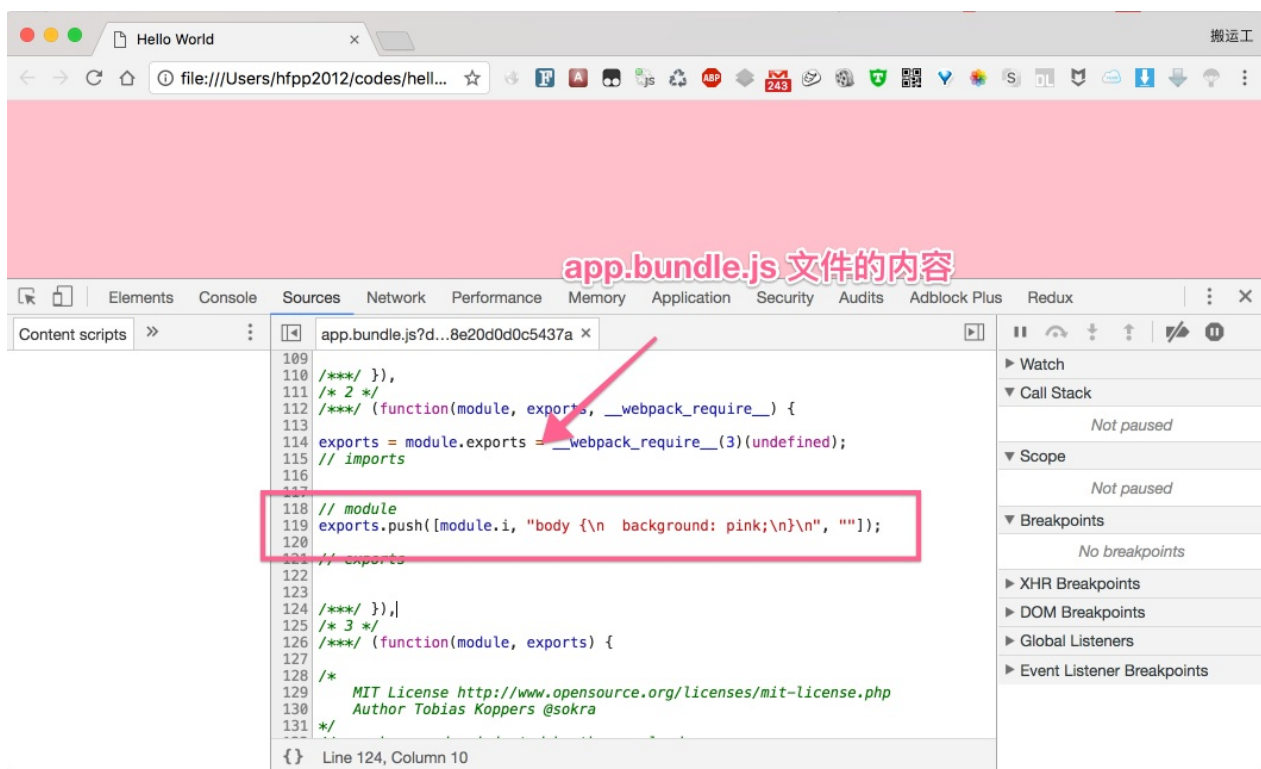
编译出的内容跟之前的差不多。

我们用浏览器打开 `dist/index.html` 文件。

6. 使用 loader 处理 CSS 和 Sass



编译出的 `app.bundle.js` 文件是有包含 CSS 的内容的。



3. 用 `sass-loader` 把 **SASS** 编译成 **CSS**

应该都知道 SASS 是什么吧，不懂的话可以查一下。

说白了，就是可以用更好的语法来写 CSS，比如用嵌套。看下面的例子应该就会理解的。

把 `src/app.css` 改名为 `src/app.scss`

src/app.scss

```
body {  
  background: pink;  
  p {  
    color: red;  
  }  
}
```

src/index.html

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <title>Hello World</title>  
</head>  
<body>  
  <p>hello world</p>  
</body>  
</html>
```

src/app.js

```
import css from './app.scss';  
  
console.log("hello world");
```

安装（中间可能要下载二进制包，要耐心等待）

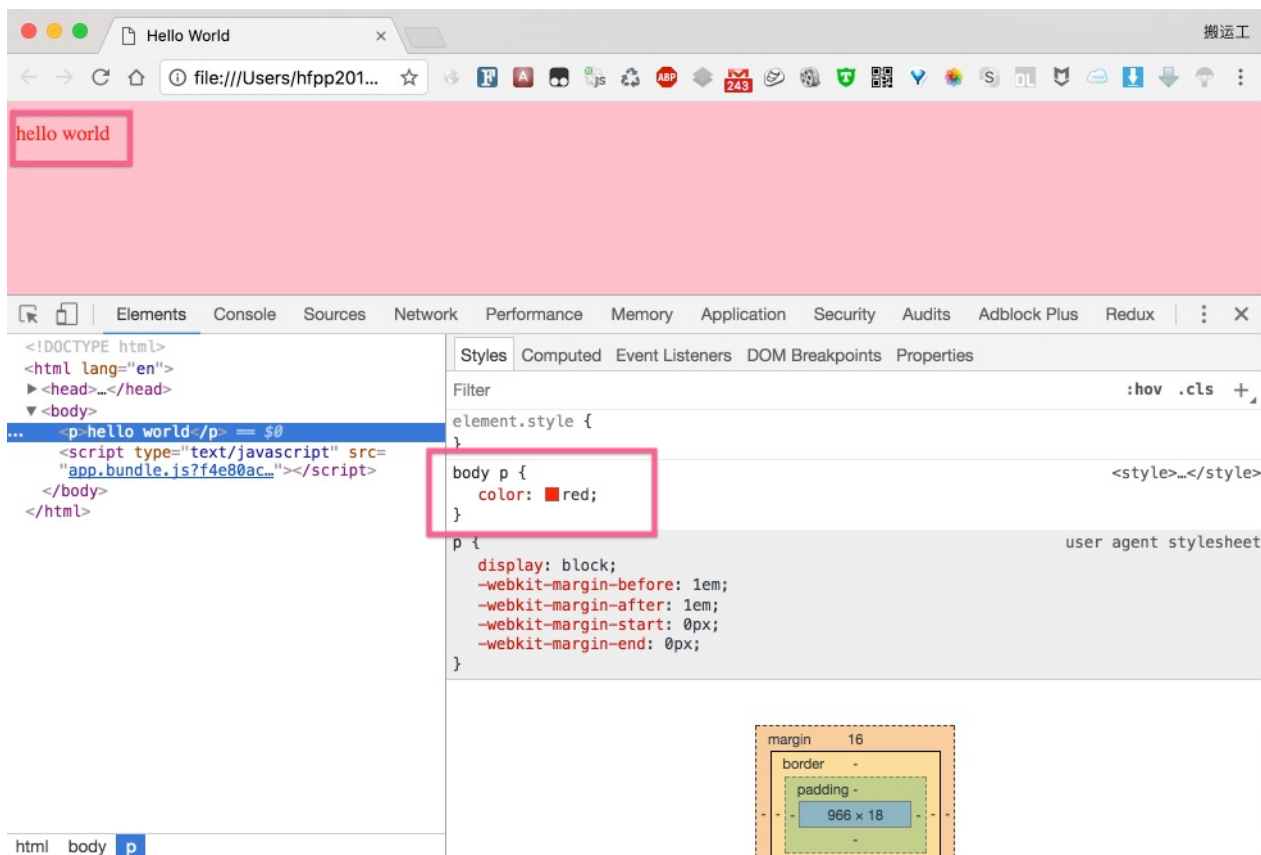
```
$ npm install sass-loader node-sass --save-dev
```

webpack.config.js

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

module.exports = {
  entry: './src/app.js',
  output: {
    path: __dirname + '/dist',
    filename: 'app.bundle.js'
  },
  plugins: [new HtmlWebpackPlugin({
    template: './src/index.html',
    filename: 'index.html',
    minify: {
      collapseWhitespace: true,
    },
    hash: true,
  })],
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: [ 'style-loader', 'css-loader', 'sass-loader' ]
      }
    ]
  }
};
```

效果如下：



4. 用 **extract-text-webpack-plugin** 把 CSS 分离成文件

有时候我们要把 SASS 或 CSS 处理好后，放到一个 CSS 文件中，用这个插件就可以实现。

```
$ npm install --save-dev extract-text-webpack-plugin
```

webpack.config.js

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

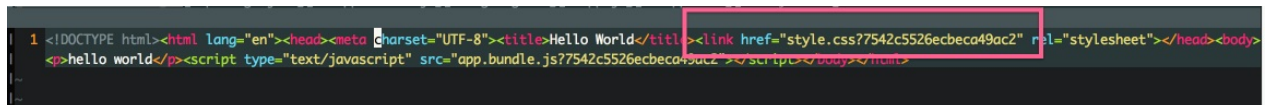
module.exports = {
  entry: './src/app.js',
  output: {
    path: __dirname + '/dist',
    filename: 'app.bundle.js'
  },
  plugins: [new HtmlWebpackPlugin({
    template: './src/index.html',
    filename: 'index.html',
    minify: {
      collapseWhitespace: true,
    },
    hash: true,
  }),
  new ExtractTextPlugin('style.css')
],
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: ExtractTextPlugin.extract({
          fallback: 'style-loader',
          //resolve-url-loader may be chained before sass-loader
          if necessary
            use: ['css-loader', 'sass-loader']
        })
      }
    ]
  }
};
```

在 `dist` 目录下生成了 `style.css` 文件。

dist/style.css

```
body {  
  background: pink; }  
body p {  
  color: red; }
```

dist/index.html



```
1 <!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>Hello World</title><link href="style.css?7542c5526ecbeca49ac2" rel="stylesheet"></head><body>  
<p>hello world</p><script type="text/javascript" src="app.bundle.js?7542c5526ecbeca49ac2"></script></body></html>
```

先说这么多吧。

现在我们来学习一个最常用的插件 `webpack-dev-server`，一般来说，这个插件，大家都会用，特别是开发环境下。

我们之前使用 `webpack -d --watch` 来在开发环境下编译静态文件，但是这个功能，完全可以用 `webpack-dev-server` 来代替。

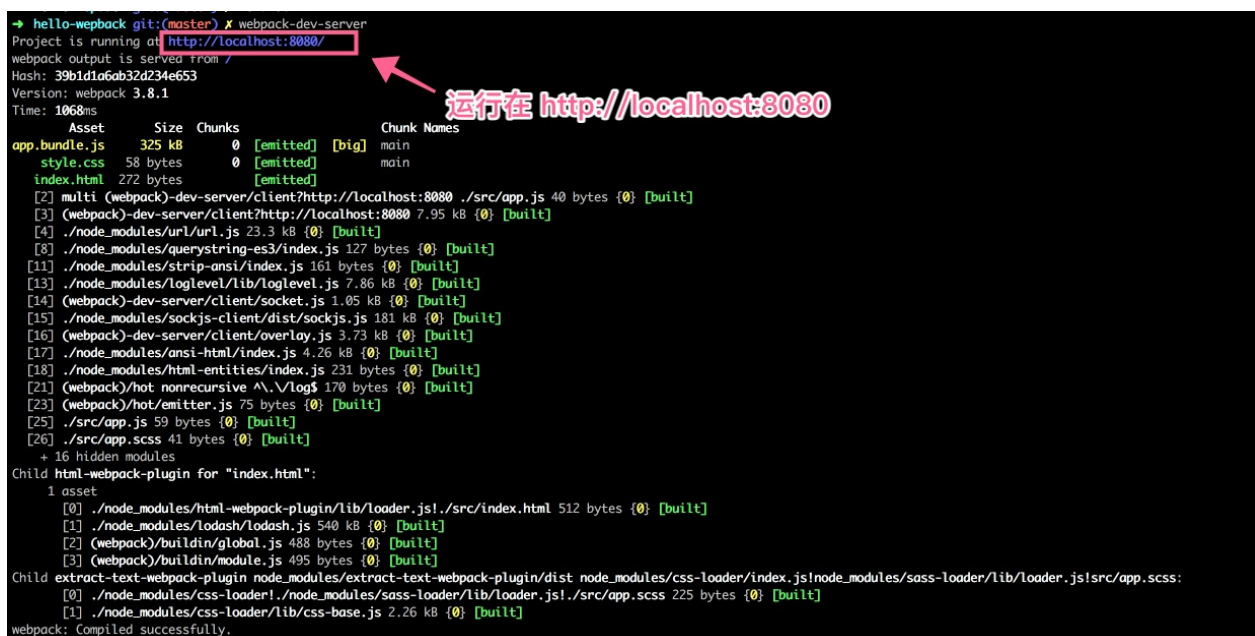
除此之外，`webpack-dev-server` 还有其他的功能，比如在本地上开启服务，打开浏览器等。

这节我们主要来简单体验一下 `webpack-dev-server` 的功能。

```
$ npm install --save webpack-dev-server
```

然后运行命令：

```
$ webpack-dev-server
```



```
hello-webkit git:(master) $ webpack-dev-server
Project is running at http://localhost:8080/
webpack output is served from /
Hash: 39b1d1a6ab32d234e653
Version: webpack 3.8.1
Time: 1068ms

   Asset      Size  Chunks             Chunk Names
app.bundle.js 325 kB      0  [emitted] [big]  main
style.css     58 bytes      0  [emitted]         main
index.html    272 bytes      0  [emitted]

[2] multi (webpack)-dev-server/client?http://localhost:8080 ./src/app.js 40 bytes {0} [built]
[3] (webpack)-dev-server/client?http://localhost:8080 7.95 kB {0} [built]
[4] ./node_modules/url/url.js 23.3 kB {0} [built]
[8] ./node_modules/QueryString-es3/index.js 127 bytes {0} [built]
[11] ./node_modules/strip-ansi/index.js 161 bytes {0} [built]
[13] ./node_modules/loglevel/lib/loglevel.js 7.86 kB {0} [built]
[14] (webpack)-dev-server/client/socket.js 1.05 kB {0} [built]
[15] ./node_modules/sockjs-client/dist/sockjs.js 181 kB {0} [built]
[16] (webpack)-dev-server/client/overlay.js 3.73 kB {0} [built]
[17] ./node_modules/ansi-html/index.js 4.26 kB {0} [built]
[18] ./node_modules/html-entities/index.js 231 bytes {0} [built]
[21] (webpack)/hot nonrecursive ^\.\/log$ 170 bytes {0} [built]
[23] (webpack)/hot/emitter.js 75 bytes {0} [built]
[25] ./src/app.js 59 bytes {0} [built]
[26] ./src/app.scss 41 bytes {0} [built]
+ 16 hidden modules
Child html-webpack-plugin for "index.html":
  1 asset
    [0] ./node_modules/html-webpack-plugin/lib/loader.js!./src/index.html 512 bytes {0} [built]
    [1] ./node_modules/lodash/lodash.js 540 kB {0} [built]
    [2] (webpack)/buildin/global.js 488 bytes {0} [built]
    [3] (webpack)/buildin/module.js 495 bytes {0} [built]
Child extract-text-webpack-plugin node_modules/extract-text-webpack-plugin/dist node_modules/css-loader/index.js!node_modules/sass-loader/lib/loader.js!src/app.scss:
    [0] ./node_modules/css-loader!./node_modules/sass-loader/lib/loader.js!./src/app.scss 225 bytes {0} [built]
    [1] ./node_modules/css-loader/lib/css-base.js 2.26 kB {0} [built]
webpack: Compiled successfully.
```

现在我们用浏览器打开 `localhost:8080` 也可以看到以前的效果。

下面是编译后的源码。



默认是运行在 `8080` 端口，这个我们可以改。

webpack.config.js

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

module.exports = {
  entry: './src/app.js',
  ...
  devServer: {
    port: 9000
  },
  ...
};
```

我们还可以配置一运行 `webpack-dev-server` 的时候就自动打开浏览器。

webpack.config.js

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

module.exports = {
  entry: './src/app.js',
  ...
  devServer: {
    port: 9000,
    open: true
  },
  ...
};
```

以后都会一直用 `webpack-dev-server` 的啦。

先这样吧。

相信这个话题很多人都感兴趣，这里就用几个简单的命令就可以搭建出 react 的开发环境。

1. 安装 react

```
$ npm install --save react react-dom
```

2. 建立 babel 和 react

```
$ npm install --save-dev babel-cli babel-preset-react babel-preset-env
```

创建 `.babelrc` 文件。

```
{  
  "presets": ["env", "react"]  
}
```

3. babel-loader

```
$ npm install --save-dev babel-loader
```

webpack.config.js

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

module.exports = {
  entry: './src/app.js',
  ...
  ...
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: ExtractTextPlugin.extract({
          fallback: 'style-loader',
          //resolve-url-loader may be chained before sass-loader
          if necessary
            use: ['css-loader', 'sass-loader']
        })
      },
      { test: /\.js$/, loader: 'babel-loader', exclude: /node_modules/ },
      { test: /\.jsx$/, loader: 'babel-loader', exclude: /node_modules/ }
    ]
  }
};
```

4. 写 react 组件

src/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Hello World</title>
</head>
<body>
  <div id="root"></div>
</body>
</html>
```

src/app.js

```
import css from './app.scss';

import React from 'react';
import ReactDOM from 'react-dom';
import Root from './Root';

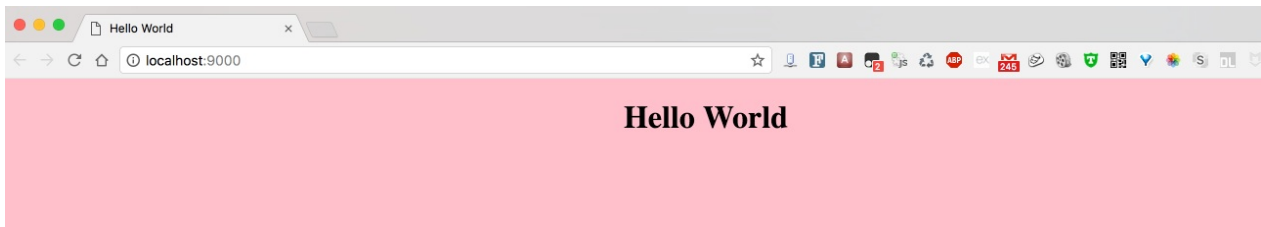
ReactDOM.render(
  <Root></Root>,
  document.getElementById('root')
);
```

src/Root.js

```
import React from 'react';

export default class Root extends React.Component {
  render() {
    return (
      <div style={{textAlign: 'center'}}>
        <h1>Hello World</h1>
      </div>);
  }
}
```

8. 用 webpack 和 babel 配置 react 开发环境



其实 **clean-webpack-plugin** 很容易知道它的作用，就是来清除文件的。

一般这个插件是配合 `webpack -p` 这条命令来使用，就是说在为生产环境编译文件的时候，先把 `build`或`dist` (就是放生产环境用的文件) 目录里的文件先清除干净，再生成新的。

1. 为什么要用 **clean-webpack-plugin**

如果还不理解为什么要用它，就看看下面的例子就可以知道的。

webpack.config.js

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

module.exports = {
  entry: {
    "app.bundle": './src/app.js'
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].[chunkhash].js'
  },
  ...
};
```

在终端上运行：

```
$ npm run prod
```

9. 用 clean-webpack-plugin 来清除文件

```
→ hello-webpack git:(master) ✖ npm run prod

> hello-webpack@1.0.0 prod /Users/hfpp2012/codes/hello-webpack
> webpack -p

Hash: 1e28f591a3e2c6f042dd
Version: webpack 3.8.1
Time: 4282ms

Asset      Size  Chunks  Chunk Names
app.bundle.e56abf8d6e5742c78c4b.js  101 kB    0  [emitted]  app.bundle
style.css   39 bytes    0  [emitted]  app.bundle
index.html  296 bytes    0  [emitted]

[4] ./src/app.js 550 bytes {0} [built]
[5] ./src/app.scss 41 bytes {0} [built]
[17] ./src/Root.js 2.35 kB {0} [built]
[18] ./node_modules/css-loader!./node_modules/sass-loader/lib/loader.js!./src/app.scss 202 bytes [built]
```

看看 `dist` 目录：

```
dist
├── app.bundle.e56abf8d6e5742c78c4b.js
├── index.html
└── style.css
```

你再把 `src/app.js` 改改内容，然后再执行 `npm run prod`。

```
→ hello-webpack git:(master) ✖ npm run prod

> hello-webpack@1.0.0 prod /Users/hfpp2012/codes/hello-webpack
> webpack -p

Hash: c74c38569bca37540159
Version: webpack 3.8.1
Time: 3590ms

Asset      Size  Chunks  Chunk Names
app.bundle.259c34c1603489ef3572.js  101 kB    0  [emitted]  app.bundle
style.css   39 bytes    0  [emitted]  app.bundle
index.html  296 bytes    0  [emitted]

[4] ./src/app.js 550 bytes {0} [built]
[5] ./src/app.scss 41 bytes {0} [built]
[17] ./src/Root.js 2.35 kB {0} [built]
[18] ./node_modules/css-loader!./node_modules/sass-loader/lib/loader.js!./src/app.scss 202 bytes [built]
```

当前的 hash 值

再多运行几次，生成的带 hash 的 `app.bundle.js` 文件就会很多。

```
dist
├── app.bundle.0e380cea371d050137cd.js
├── app.bundle.259c34c1603489ef3572.js
├── app.bundle.e56abf8d6e5742c78c4b.js
├── index.html
└── style.css
```

这些带 hash 的 `app.bundle.js` 只有最新的才有用，其他的都没用，我们要在 build 之前把它们全清空，这真是 `clean-webpack-plugin` 发挥的作用。

2. 使用 clean-webpack-plugin

首先来安装。

```
$ npm i clean-webpack-plugin --save-dev
```

webpack.config.js

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

let pathsToClean = [
  'dist',
]

module.exports = {
  entry: {
    "app.bundle": './src/app.js'
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].[chunkhash].js'
  },
  ...
  plugins: [
    new CleanWebpackPlugin(pathsToClean),
    ...
    new ExtractTextPlugin('style.css')
  ],
  ...
};
```

现在运行 `npm run prod` 试试，只有下面的文件：

```
dist
├─ app.bundle.0e380cea371d050137cd.js
├─ index.html
└─ style.css
```


先到这里。

之前我们只写了一个 html 文件，就是 `src/index.html`，但是有时候我们是需要多个的，这个时候，怎么办呢？

之前我们是这么做的，用了 `html-webpack-plugin` 这个插件来输出 html 文件。

webpack.config.js

```
...
new HtmlWebpackPlugin({
  template: './src/index.html',
  filename: 'index.html',
  minify: {
    collapseWhitespace: true,
  },
  hash: true,
})
...
```

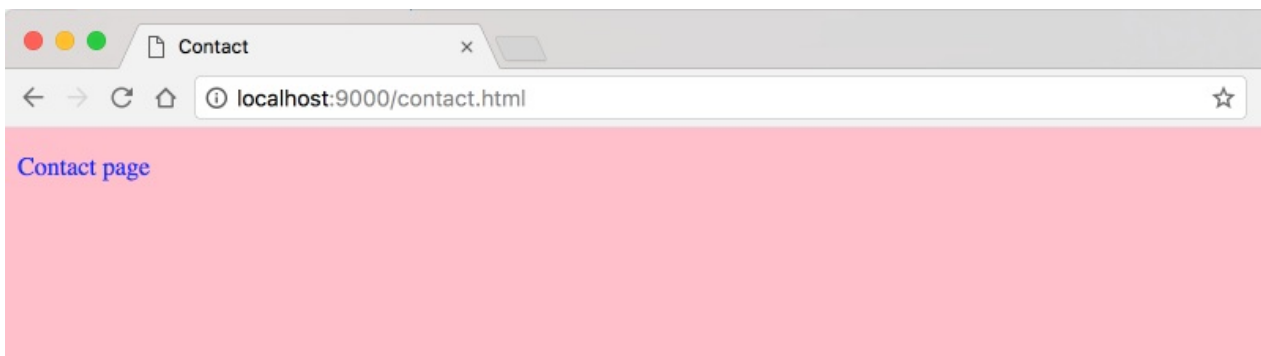
之前怎么做，现在还是怎么做，我们复制一下，改个名字不就好了吗？

webpack.config.js

```
new HtmlWebpackPlugin({
  template: './src/index.html',
  filename: 'index.html',
  minify: {
    collapseWhitespace: true,
  },
  hash: true,
}),
new HtmlWebpackPlugin({
  template: './src/contact.html',
  filename: 'contact.html',
  minify: {
    collapseWhitespace: true,
  },
  hash: true,
})
```

src/contact.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Contact</title>
</head>
<body>
  <p>Contact page</p>
</body>
</html>
```



有个问题， **contact.html** 使用的 **js** 和 **css** 跟 **index.html** 是一模一样的

如果我要让 **contact.html** 使用跟 **index.html** 不同的 **js**，如何做呢？（只要保证 **js** 不同，**css** 也会不同的，因为 **css** 也是由 **js** 里 **import** 的嘛）

那我们来改造一下：

```
...

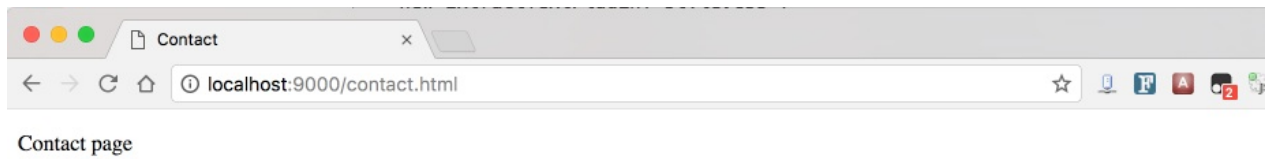
module.exports = {
  entry: {
    "app.bundle": './src/app.js',
    // 这行是新增的。
    "contact": './src/contact.js'
  },
  ...
  plugins: [
    new CleanWebpackPlugin(pathsToClean),
    new HtmlWebpackPlugin({
      template: './src/index.html',
      filename: 'index.html',
      minify: {
        collapseWhitespace: true,
      },
      hash: true,
      // 这行是新增的。
      excludeChunks: ['contact']
    }),
    new HtmlWebpackPlugin({
      template: './src/contact.html',
      filename: 'contact.html',
      minify: {
        collapseWhitespace: true,
      },
      hash: true,
      // 这行是新增的。
      chunks: ['contact']
    }),
    new ExtractTextPlugin('style.css')
  ],
  ...
};
```

上面的 `excludeChunks` 指的是不包含，`chunks` 代表的是包含。

src/contact.js

```
console.log('hi from contact js')
```

结果：



这样就 OK 了。

先说这么多。

首先肯定会问什么是 **pug**，如果是 **nodejs** 程序员的话，肯定听过 **jade** 吧，**pug** 就是从 **jade** 改名过来的。

说白了，它就是可以让你以更好的语法来写 **html**。

下面看看例子就会清楚的。

现在我们要代替之前的 `src/index.html` 改用 **pug** 语法来写。

首先把 `src/index.html` 改名叫 `src/index.pug`

```
$ rename src/index.html src/index.pug
```

src/index.pug

```
doctype html
html(lang="en")
  head
    title= pageTitle
    script(type='text/javascript').
      if (foo) bar(1 + 5)
  body
    h1 Pug - node template engine
    #root
    #container.col
      if youAreUsingPug
        p You are amazing
      else
        p Get on it!
    p.
      Pug is a terse and simple templating language with a
      strong focus on performance and powerful features.
```

上面的内容是从 **pug** 官方的示例中抄的，然后稍微改了一下。

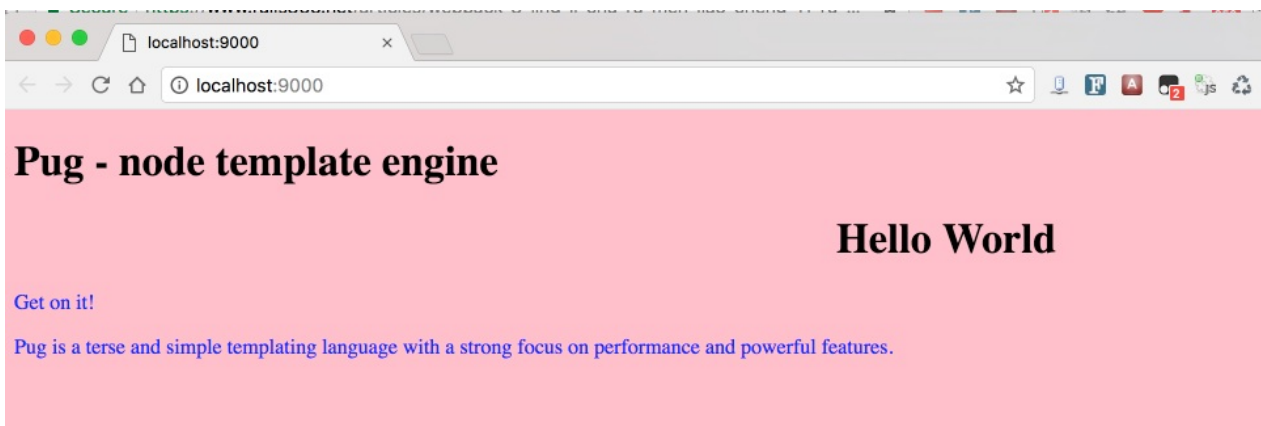
webpack.config.js

```
...

module.exports = {
  ...
  plugins: [
    ...
    new HtmlWebpackPlugin({
      template: './src/index.pug',
      ...
    }),
    ...
  ],
  module: {
    rules: [
      ...
      { test: /\.pug$/, loader: ['raw-loader', 'pug-html-loader']
    ] }
  ]
}
};
```

```
$ npm install --save-dev pug pug-html-loader raw-loader
```

这样基本没啥问题，来看下结果：



我们来试试 `pug` 的 `include` 功能，就是可以包含子模板。

src/index.pug

```
...  
  body  
    include includes/header.pug  
    ...
```

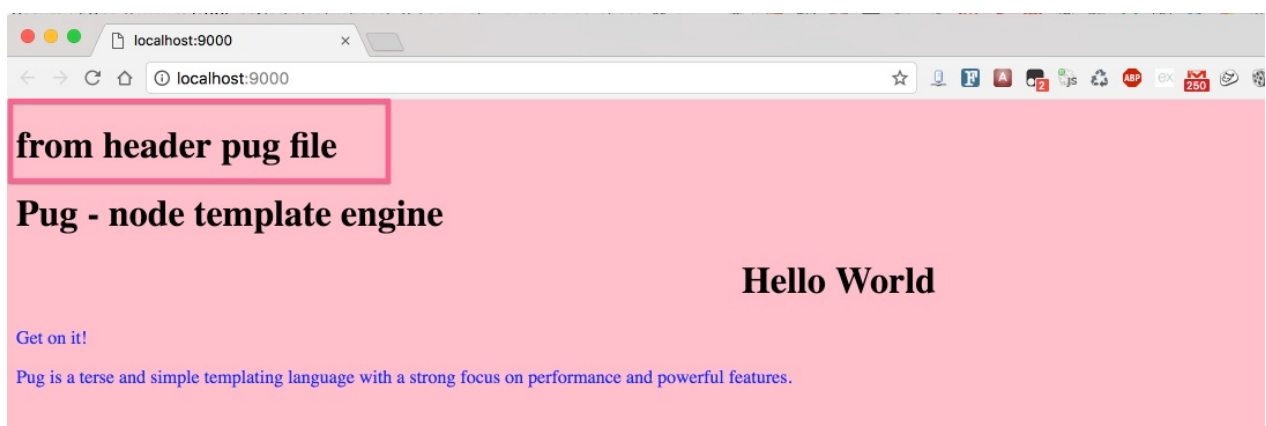
src/includes/header.pug

```
h1 from header pug file
```

目录结构是这样的：

```
src  
├─ Root.js  
├─ app.js  
├─ app.scss  
├─ contact.html  
├─ contact.js  
├─ includes  
│   └─ header.pug  
└─ index.pug
```

结果：



先这样吧。