## 为什么选择 webpack

webpack 的定义： webpack is a module bundler

简言之，在webpack的处理下，所有类型的文件都是模块，包括 js、CSS、图片、JSON，等等。既然是模块，那就可以通过import/require引入，而这一切离不开webpack的加载器loader

## React.js 项目

我在这里草拟了一个项目：一个[简单的单页面应用](https://blog.zfanw.com/webpack-tutorial/demo/index.html)，页面上有一张图片，点击图片图片会旋转，再次点击则重置。

### 初始化项目

首先，我们来初始化项目：

$ cd ~

$ mkdir webpack-demo && cd webpack-demo

$ npm init -y

项目初始化完成后，webpack-demo 目录下多出 package.json 文件，内容如下：

{

"name": "webpack-demo",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "ISC"

}

## 新建 HTML

在 webpack-demo 下新建 index.html：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>webpack 教程</title>

</head>

<body>

</body>

</html>

我们暂时还没有引用 JavaScript。

## 创建 JavaScript 文件

在 webpack-demo 目录下新建 index.js，内容暂时留空。

我们需要安装 react 及 react-dom：

$ npm i react react-dom

装好后，在 index.js 文件中 import 它们：

import React from 'react'

import ReactDOM from 'react-dom'

好了，现在在 index.html 中引用 index.js：

<body>

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

</body>

现在在浏览器中打开 index.html，不出意外，console 中报错了，只是不同浏览器下报的错不一样：

1. Firefox 59.0.2: SyntaxError: import declarations may only appear at top level of a module
2. Safari 11.0.3: SyntaxError: Unexpected identifier 'React'. import call expects exactly one argument.
3. Chrome 65.0.3325.181: Uncaught SyntaxError: Unexpected identifier

怎么办？浏览器的兼容性问题几乎是与生俱来的 - 要怎么抹平它们的差别，好让我们的代码能够运行在尽可能多的浏览器平台上？

欢迎 [babel.js](https://babeljs.io/) - babel 转换我们的 JavaScript，让它们在所有目标浏览器下正常运行。但我们不直接使用 babel，而是通过 [babel-loader](https://github.com/babel/babel-loader) 这个 webpack 加载器。

但在那之前，我们需要先安装好 webpack。

## 安装 webpack

在 webpack-demo 项目根目录中安装 webpack：

$ npm install -D webpack

安装完成后，我们就会看到当前安装的 webpack 版本号。

编译 JavaScript

webpack 是个打包工具，那么就需要一个输入（入口文件），一个输出（目标文件）。

我们试试在命令行下将 index.js 打包成 build.min.js：

$ npx webpack index.js -o build.min.js

The CLI moved into a separate package: webpack-cli

Would you like to install webpack-cli? (That will run npm install -D webpack-cli) (yes/NO)

没有输出，反而是提示我们是否安装 webpack-cli。这是因为 webpack 在 webpack 4 里将命令行相关的都迁移至 webpack-cli 包。

输入 yes 然后回车，稍等一会儿，webpack-cli 就安装好了。之后我们的命令继续执行，结果如下：

Hash: cbb2732def750315c477

Version: webpack 4.5.0

Time: 243ms

Built at: 2018-4-7 09:56:02

Asset Size Chunks Chunk Names

build.min.js 105 KiB 0 [emitted] main

Entrypoint main = build.min.js

[13] ./index.js 58 bytes {0} [built]

+ 13 hidden modules

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/

出现黄色警告。这是 webpack 4 引入的模式，包括 development、production、none 三个值，我们不传入值的话，会默认使用 production。

我们调整下命令：

$ npx webpack index.js -o build.min.js --mode development

你可能好奇，不指定输入、输出的话会怎样？

我们来试试：

$ npx webpack --mode development

Hash: 86d52d22c7b7982a7ebf

Version: webpack 4.5.0

Time: 41ms

Built at: 2018-4-7 10:00:39

ERROR in Entry module not found: Error: Can't resolve './src' in '/Users/sam/tmp/webpack-demo'

报错，说在 ./src 下找不到 index.js 文件 - 这也是 webpack 4 引入的特性，**默认输入文件位置在 src/index.js 下**。我们将项目根目录下的 index.js 移动到 src/index.js：

$ mkdir src

$ mv index.js src/index.js

然后再打包一遍：

$ npx webpack --mode development

Hash: 4726bc2e160759010056

Version: webpack 4.5.0

Time: 321ms

Built at: 2018-4-7 10:03:24

Asset Size Chunks Chunk Names

main.js 688 KiB main [emitted] main

Entrypoint main = main.js

[./src/index.js] 58 bytes {main} [built]

+ 21 hidden modules

**打包成功，并且目录下多出 dist/main.js - 这也正是 webpack 4 默认的输出文件**。换句话说，我们依照 webpack 的约定组织文件的话，就可以省去许多配置。

打包完成后，我们将 index.html 中对 JavaScript 引用调整为编译后的 dist/main.js：

<body>

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

+ <script src="dist/main.js"></script>

</body>

再刷新各浏览器看控制台，已经不再报错。

### 实时刷新

在 index.html 文件中引用 dist/main.js 文件后，我们有几个问题需要解决。

1. 入口文件 src/index.js 的变化，包括它所引用的其它模块的变化如何通知给 webpack，以便重新构建 dist/main.js 文件？
2. dist/main.js 文件更新后，浏览器中打开的页面该如何自动刷新？

#### 监控文件

第一个问题，webpack 有好几个解决办法，其中 watch 选项最直观，我们可以让 webpack 监控文件变化，一旦文件有变化，就重新构建。但默认情况下，watch 选项是禁用的。

我们可以在命令行下启用它：

$ npx webpack --mode development --watch

Webpack is watching the files…

Hash: 4726bc2e160759010056

Version: webpack 4.5.0

Time: 367ms

Built at: 2018-4-7 10:12:59

Asset Size Chunks Chunk Names

main.js 688 KiB main [emitted] main

Entrypoint main = main.js

[./src/index.js] 58 bytes {main} [built]

+ 21 hidden modules

我们试试在 index.js 文件中添加一行 console.log('hello webpack')，保存文件后就会看到命令行下的变化：

Hash: 0d285ea4d1c26ac17380

Version: webpack 4.5.0

Time: 19ms

Built at: 2018-4-7 10:13:35

Asset Size Chunks Chunk Names

main.js 688 KiB main [emitted] main

Entrypoint main = main.js

[./src/index.js] 87 bytes {main} [built]

+ 21 hidden modules

webpack 监控到 src/index.js 文件的变化，重新构建了 dist/main.js，耗时 19ms。

#### 刷新浏览器

至于**自动刷新浏览器的问题，webpack 提供**[**webpack-dev-server**](https://github.com/webpack/webpack-dev-server)**来解决**，它是一个基于 expressjs 的开发服务器，提供实时刷新浏览器页面的功能。不过目前 webpack-dev-server 已经进入维护模式，文档中推荐我们使用 [webpack-serve](https://github.com/webpack-contrib/webpack-serve)。

但这里暂时还是使用 webpack-dev-server 来说明。

### webpack-dev-server

首先在项目下安装 webpack-dev-server：

$ npm install -D webpack-dev-server

安装完成后在命令行下执行 webpack-dev-server：

$ npx webpack-dev-server --mode development

ℹ ｢wds｣: Project is running at http://localhost:8080/

ℹ ｢wds｣: webpack output is served from /

ℹ ｢wdm｣: Hash: 441c738298078bd7a80b

Version: webpack 4.5.0

Time: 504ms

Built at: 2018-4-7 10:20:52

Asset Size Chunks Chunk Names

main.js 1020 KiB main [emitted] main

Entrypoint main = main.js

[./node\_modules/ansi-html/index.js] 4.16 KiB {main} [built]

[./node\_modules/loglevel/lib/loglevel.js] 7.68 KiB{main} [built]

[./node\_modules/react-dom/index.js] 1.33 KiB {main} [built]

[./node\_modules/react/index.js] 190 bytes {main} [built]

[./node\_modules/sockjs-client/dist/sockjs.js] 176 KiB {main} [built]

[./node\_modules/url/url.js] 22.8 KiB {main} [built]

[./node\_modules/webpack-dev-server/client/index.js?http://localhost:8080] (webpack)-dev-server/client?http://localhost:8080 7.75 KiB {main} [built]

[./node\_modules/webpack-dev-server/client/overlay.js] (webpack)-dev-server/client/overlay.js 3.58 KiB{main} [built]

[./node\_modules/webpack-dev-server/client/socket.js] (webpack)-dev-server/client/socket.js 1.05 KiB {main} [built]

[./node\_modules/webpack-dev-server/node\_modules/strip-ansi/index.js] (webpack)-dev-server/node\_modules/strip-ansi/index.js 161 bytes {main} [built]

[./node\_modules/webpack/hot sync ^\.\/log$] (webpack)/hot sync nonrecursive ^\.\/log$ 170 bytes {main} [built]

[./node\_modules/webpack/hot/emitter.js] (webpack)/hot/emitter.js 77 bytes {main} [built]

[0] multi (webpack)-dev-server/client?http://localhost:8080 ./src 40 bytes {main} [built]

[./node\_modules/webpack/hot/log.js] (webpack)/hot/log.js 1.03 KiB {main} [optional] [built]

[./src/index.js] 87 bytes {main} [built]

+ 32 hidden modules

ℹ ｢wdm｣: Compiled successfully.

Ooops，令人晕眩的输出结果。

不过且忽视它们，我们现在可以在 http://localhost:8080/ 访问我们的项目。

在入口文件 src/index.js 里再添加一行代码验证下浏览器页面的实时刷新功能：

console.log('webpack live reload is working')

webpack 重新打包了 dist/main.js，但是浏览器并没有刷新。问题出在哪？

#### webpack-dev-server 未刷新页面

这个问题不注意的话很容易发生，所以单独说一下。

我们看前面 npx webpack-dev-server --mode development 的输出里有这么一行：

webpack output is served from /

是了，webpack-dev-server 构建的 main.js 其实是在 http://localhost:8080/main.js 的位置，而不是 http://localhost:8080/dist/main.js，而且，它存在于内存中，并不写入磁盘。而我们在 index.html页面中引用的是 dist/main.js。

我们可以在运行 webpack-dev-server 时指定 output.publicPath：

$ npx webpack-dev-server --mode development --output-public-path dist

webpack output is served from /dist

好了，解决问题。

现在再修改 src/index.js 文件，我们就可以在 chrome 浏览器的控制台看到如下内容：

[WDS] App updated. Reloading...

页面自动刷新了。实际上，我们看 webpack-dev-server 的命令行会发现，它同样监控入口文件的变化并重新编译 - 换句话说，webpack-dev-server 已经启用 --watch 效果。

### 编码

我们在 src/index.js 中先写个简单的 React 代码：

import React from 'react'

-import ReactDOM from 'react-dom'

+import ReactDOM from 'react-dom'

+ReactDOM.render(<div>hello webpack</div>, document.body)

注意，React 不推荐使用 body 上，这里只是图方便才这么写。

查看 webpack-dev-server 的状态：

[./src/index.js] 255 bytes {main} [built] [failed][1 error]

+ 25 hidden modules

ERROR in ./src/index.js

Module parse failed: Unexpected token (3:16)

You may need an appropriate loader to handle this file type.

| import React from 'react'

| import ReactDOM from 'react-dom'

| ReactDOM.render(<div>hello webpack</div>, document.body)

|

@ multi (webpack)-dev-server/client?http://localhost:8080 ./src

ℹ ｢wdm｣: Failed to compile.

报错。为什么？因为我们写了 JSX 语法，webpack 不认识。怎么办，找 babel-loader 来翻译。

$ npm install -D "babel-loader@^8.0.0-beta" @babel/core @babel/preset-react

接着重启 webpack-dev-server：

$ npx webpack-dev-server --mode development --output-public-path dist --module-bind 'js=babel-loader?presets[]=@babel/preset-react'

不再报错，且页面上已经显示 hello webpack。

#### webpack 配置文件

你可能已经开始厌恶冗长的命令行了，在你有这个想法时，webpack 配置文件就出现了。

我们来新建一个 webpack.config.js 文件：

const path = require('path')

module.exports = {

mode: 'development',

output: {

publicPath: '/dist/'

},

module: {

rules: [

{

test: /\.js$/,

include: [

path.resolve(\_\_dirname, 'src')

],

loader: 'babel-loader',

options: {

presets: ['@babel/preset-react']

}

}

]

}

}

这样，我们只要执行 npx webpack-dev-server 就行，不再需要输入一长串命令行。

## 图片加载器

我们需要一张图片，我从 unsplash 找来了一张玫瑰，放到 src/img/rose.jpg 位置。

我们在 src/index.js 中 import 它：

import ReactDOM from 'react-dom'

+import Rose from './img/rose.jpg'

ReactDOM.render(<div>hello webpack</div>, document.body)

不出意外，报错了：

[./src/img/rose.jpg] 177 bytes {main} [built] [failed] [1 error]

[./src/index.js] 178 bytes {main} [built]

+ 46 hidden modules

ERROR in ./src/img/rose.jpg

Module parse failed: Unexpected character '�' (1:0)

You may need an appropriate loader to handle this file type.

(Source code omitted for this binary file)

@ ./src/index.js 3:0-34

@ multi (webpack)-dev-server/client?http://localhost:8080 ./src

ℹ ｢wdm｣: Failed to compile.

和处理 JSX 一样的道理，我们需要一个翻译（即加载器）。

**在 webpack 里，负责图片翻译的是**[**file-loader**](https://github.com/webpack-contrib/file-loader)**：**

$ npm install -D file-loader

接着是配置 webpack.config.js：

+ },

+ {

+ test: /\.(png|jpg|gif)$/,

+ use: [

+ {

+ loader: 'file-loader',

+ options: {}

+ }

+ ]

}

]

重启 webpack-dev-server，发现 webpack 已经能正常编译了 - 图片摇身一变，也是一个模块。

而 webpack 在最终构建时，会自动将模块中引用的图片拷贝到相应目录 - 谢天谢地，再也不用手动拷贝。

另外，webpack 在打包图片时，还可以调整图片的名称 - 比如给它们加上一段 hash，这样 rose.jpg 图片在打包后就会变成 rose.23127e7474e4f5eee452931e4baf14d0.jpg，好处是我们在服务器上可以给图片设置一年甚至更长时间的缓存，而不必担心图片缓存造成数据陈旧的问题。

确保图片加载没问题后，我们来完善下代码：

-ReactDOM.render(<div>hello webpack</div>, document.body)

+class App extends React.Component {

+ render () {

+ return (

+ <div><img src={Rose} alt='玫瑰' /></div>

+ )

+ }

+}

+ReactDOM.render(<App />, document.body)

查看浏览器，发现图片太大。我们需要 CSS 来控制图片尺寸。

### 加载 CSS 文件

在 React.js 里，CSS 有很多种写法，比如我们可以直接写在 style 中：

<img src={Rose} alt='玫瑰' style={{maxWidth: 500}} />

因为这就是 JavaScript，我们也就不需要额外处理。

但我们也可以写在 css 文件中。

在 src 下新增 index.css：

.flower {

max-width: 500px

}

然后在 index.js 中引入并应用：

import React from 'react'

import ReactDOM from 'react-dom'

import Rose from './img/rose.jpg'

+import './index.css'

class App extends React.Component {

render () {

return (

- <div><img src={Rose} alt='玫瑰' /></div>

+ <div><img src={Rose} alt='玫瑰' className='flower' /></div>

)

}

}

又报错了：

[./src/index.css] 166 bytes {main} [built] [failed] [1 error]

[./src/index.js] 391 bytes {main} [built]

+ 47 hidden modules

ERROR in ./src/index.css

Module parse failed: Unexpected token (1:0)

You may need an appropriate loader to handle this file type.

| .flower {

| max-width: 500px;

| }

@ ./src/index.js 4:0-21

@ multi (webpack)-dev-server/client?http://localhost:8080 ./src

ℹ ｢wdm｣: Failed to compile.

同理，我们需要 CSS 加载器：

1. [**css-loader**](https://github.com/webpack-contrib/css-loader)**- 预处理 CSS 文件**
2. [**style-loader**](https://github.com/webpack-contrib/style-loader)**- 将 CSS 插入到 DOM 中的 style 标签**

注意，我们如果只使用了 css-loader，则 webpack 只是将 CSS 文件预处理成模块然后打包到构建文件中，并不会插入到页面 - 这是 style-loader 的作用。

我们先来安装它们：

$ npm install -D css-loader style-loader

然后修改 webpack.config.js 配置：

+ },

+ {

+ test: /\.css$/,

+ use: [{ loader: 'style-loader' }, { loader: 'css-loader' }]

}

]

}

请注意，loader 的顺序很重要，比如上面新增的这一节，如果把 style-loader 放到 css-loader 后面，我们就会撞见错误：

ERROR in ./index.css

Module build failed: Unknown word (5:1)

3 | // load the styles

4 | var content = require("!!./index.css");

> 5 | if(typeof content === 'string') content = [[module.id, content, '']];

| ^

6 | // Prepare cssTransformation

7 | var transform;

8 |

@ ./index.js 3:0-21

@ multi (webpack)-dev-server/client?http://localhost:8080 ./index.js

webpack: Failed to compile

因为 style-loader 无法理解 CSS 文件，需要先经 css-loader 预处理 - 是的，加载器的执行顺序是从后往前的。

重启 webpack-dev-server，编译正常，css 已生效。

只不过，这里的 CSS 虽然是 import 进来的，但仍是全局的，等效于我们平常使用 <link href="" /> 引用 CSS。webpack 还提供 [CSS Modules](https://github.com/webpack-contrib/css-loader#modules)，可以将样式真正意义上的模块化。

除了 CSS Modules 外，我们还有很多 [CSS in js](https://github.com/MicheleBertoli/css-in-js) 方案可选，它们允许我们将样式跟 HTML、JS 放到一起 - 如果你写过 Vue.js 的单文件模块，可能会很喜欢。

在成功配置图片与 CSS 后，我们可以继续完善代码：

src/index.js：

class App extends React.Component {

+ state = {

+ reset: 'yes'

+ }

+ onClick = () => {

+ this.setState({

+ reset: this.state.reset === 'yes' ? 'no' : 'yes'

+ })

+ }

render () {

return (

- <div><img src={Rose} alt='玫瑰' className='flower' /></div>

+ <div><img src={Rose} alt='玫瑰' className={this.state.reset === 'yes' ? 'flower' : 'flower flower--rotate'} onClick={this.onClick} /></div>

)

}

}

src/index.css：

+}

+.flower--rotate {

+ transform: rotate(30deg)

}

但随即发现 webpack-dev-server 又抛出错误：

[./src/index.js] 1.49 KiB {main} [built] [failed] [1 error]

+ 25 hidden modules

ERROR in ./src/index.js

Module build failed: SyntaxError: /Users/sam/tmp/webpack-demo/src/index.js: Support for the experimental syntax 'classProperties' isn't currently enabled (6:9):

4 | import './index.css'

5 | class App extends React.Component {

> 6 | state = {

| ^

7 | reset: 'yes'

8 | }

9 | onClick = () => {

Add @babel/plugin-proposal-class-properties (https://git.io/vb4SL) to the 'plugins' section of your Babel config to enable transformation.

at \_class.raise (/Users/sam/tmp/webpack-demo/node\_modules/babylon/lib/index.js:779:15)

at \_class.expectPlugin (/Users/sam/tmp/webpack-demo/node\_modules/babylon/lib/index.js:2087:18)

at \_class.parseClassProperty (/Users/sam/tmp/webpack-demo/node\_modules/babylon/lib/index.js:4904:12)

at \_class.pushClassProperty (/Users/sam/tmp/webpack-demo/node\_modules/babylon/lib/index.js:4871:30)

at \_class.parseClassMemberWithIsStatic (/Users/sam/tmp/webpack-demo/node\_modules/babylon/lib/index.js:4804:14)

at \_class.parseClassMember (/Users/sam/tmp/webpack-demo/node\_modules/babylon/lib/index.js:4741:10)

at \_class.parseClassBody (/Users/sam/tmp/webpack-demo/node\_modules/babylon/lib/index.js:4696:12)

at \_class.parseClass (/Users/sam/tmp/webpack-demo/node\_modules/babylon/lib/index.js:4646:10)

at \_class.parseStatementContent (/Users/sam/tmp/webpack-demo/node\_modules/babylon/lib/index.js:3987:21)

at \_class.parseStatement (/Users/sam/tmp/webpack-demo/node\_modules/babylon/lib/index.js:3959:17)

@ multi (webpack)-dev-server/client?http://localhost:8080 ./src

ℹ ｢wdm｣: Failed to compile.

这是因为我们用了 JavaScript 的新特性 - 需要 @babel/plugin-proposal-class-properties 插件的支持。

首先是安装该插件：

$ npm i -D @babel/plugin-proposal-class-properties

然后调整 webpack.config.js 配置：

options: {

- presets: ['@babel/preset-react']

+ presets: ['@babel/preset-react', '@babel/preset-env'],

+ plugins: ['@babel/plugin-proposal-class-properties']

}

},

重启 webpack-dev-server，编译正常。

查看浏览器，图片已经可以点击了。

## 打包

在完成项目开发后，我们需要输出文件给生产环境部署，只要执行：

$ npx webpack --mode production

就可以打包出所有静态资源。

## 部署

部署时，拷贝 dist 目录及 index.html 即可。

## 脚手架

前面的步骤里，我们几乎是一步、一步手动配置每个类型文件的加载器，一次添加一小节，然后重启 webpack-dev-server，恐怕没人喜欢这样干活。所以市面上有非常多的 boilerplates、presets 等，其中比较出名的有：

1. [create-react-app](https://github.com/facebookincubator/create-react-app/) react 官方出品的一套，只适用开发 react.js 项目；
2. [neutrino.js](https://neutrino.js.org/) 这是 Mozilla 出品的一套解决方案，Web、React、Node.js 等方案均有；
3. [Parcel](https://parceljs.org/) 最近新出的一套方案，零配置。