文章转自:

<https://segmentfault.com/a/1190000005037309?from=groupmessage&isappinstalled=0#articleHeader6>

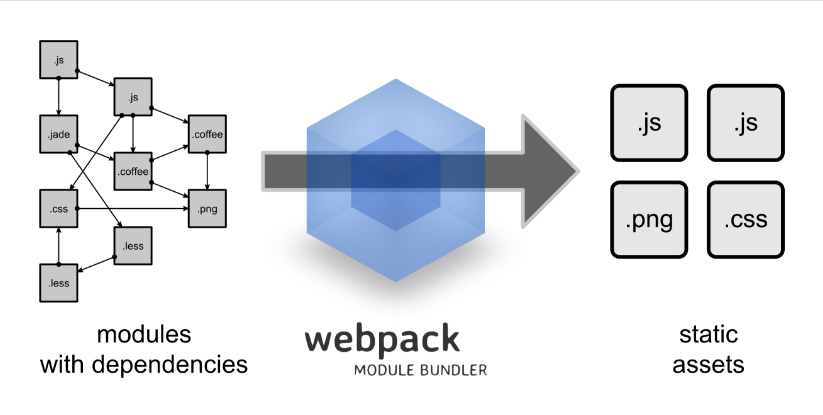
感谢作者的手把手教学.我按照流程走了一下.有些坑趟过了.

# webpack快速入门

## webpack介绍

Webpack是一个前端模块管理工具，有点类似browserify，出自Facebook的Instagram团队，但功能比browserify更为强大，可以说是目前最为强大的前端模块管理和打包工具。

Webpack将项目中的所有静态资源都当做模块，模块之间可以互相依赖，由webpack对它们进行统一的管理和打包发布，下图为官方网站说明：



webpack对React有着与生俱来的良好支持，随着React的流行，webpack也成了React项目中必不可少的一部分。特别是随着ES6的普及，使得webpack有了更广阔的用武之地。

## 安装配置webpack

### 安装nodejs和Npm(不会的各位请百度)

### Nodejs官网:http://nodejs.cn/

### Npm安装: http://jingyan.baidu.com/article/a17d528506d7f58098c8f2b0.html

### 初始化项目环境

$ mkdir react\_boilerplate

$ cd react\_boilerplate\

$ npm init -y

Wrote to .\react\_boilerplate\package.json:

{

"name": "react\_boilerplate",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

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

},

"keywords": [],

"author": "",

"license": "ISC"

}

npm init 加上一个-y选项会生成一个默认的package.json,关于这个文件，不是本文重点，在此不会详述，可以参考[官方文档](https://docs.npmjs.com/files/package.json)。可以简单的理解，这个文件是用于管理项目里面的依赖包的。

### 设置.gitignore(因为本人不用.所以这步没做)

### (如果是win机子.创建一个无名字的”.xxx”文件.首先建一个txt文件.重名 比如想定义成.gitignore这样的文件.就重名的时候” .gitignore.” 后边多加一个”.”就可以创建一个没有文件名只有文件类型的文件了)

如果我们使用git进行版本管理，一个.gitignore文件是必要的。这里我们可以先将项目需要安装的node包目录添加进去。

node\_modules

使用npm install安装的node包都会在node\_modules目录下，这个目录是不需要commit到git的。

### 安装webpack

安装webpack很简单，命令如下：(这里直接用-g全局安装就好)

npm i -g webpack --save-dev

其中--save-dev表示该包为开发环境依赖包。安装完后会生成一个node\_modules目录，并且在package.json文件中多出如下几行：

......

"devDependencies": {

"webpack": "^1.13.0"

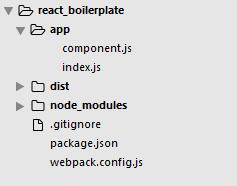
}

}

如果写为--save则表示该包为生产环境依赖包，在package.json文件中会新增或者修改dependencies 字段。

### 初始化项目结构和代码

安装完webpack后，我们可以给项目中增加一些内容了。项目的简单结构如下图所示：



app目录用于存放项目代码，dist目录为编译后的项目文件，webpack.config.js为webpack的配置文件。

我们给项目中的文件添加一些简单的代码，首先是组件代码：

##### app/component.js

module.exports = function () {

var element = document.createElement('h1');

element.innerHTML = 'Hello world';

return element;

};

然后需要一个入口文件，在入口文件中使用上面定义的组件：

##### app/index.js

var component = require('./component');

document.body.appendChild(component());

### 配置webpack

我们需要让webpack知道如何处理我们的项目目录结构，因此需要配置文件webpack.config.js。一个简单的配置文件如下所示：

var webpack = require('webpack');

var path = require('path'); //引入node的path库

var config = {

entry: ['./app/index.js'], //入口文件

output: {

path: path.resolve(\_\_dirname, 'dist'), // 指定编译后的代码位置为 dist/bundle.js

filename: 'bundle.js'

},

module: {

loaders: [

// 为webpack指定loaders

//{ test: /\.js$/, loaders: ['babel'], exclude: /node\_modules/ } ]

}}

module.exports = config;

到目前为止，我们已经可以让webpack工作了，在命令行执行

webpack

我们看到，会有一个新的文件/dist/bundle.js生成出来了。但是我们还需要一个html文件来加载编译后的代码，这就需要用到一个webpack插件：html-webpack-plugin。

### 安装html-webpack-plugin

使用如下命令安装：

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

然后在我们的webpack.config.js中增加下面几行：

plugins: [

new HtmlwebpackPlugin({

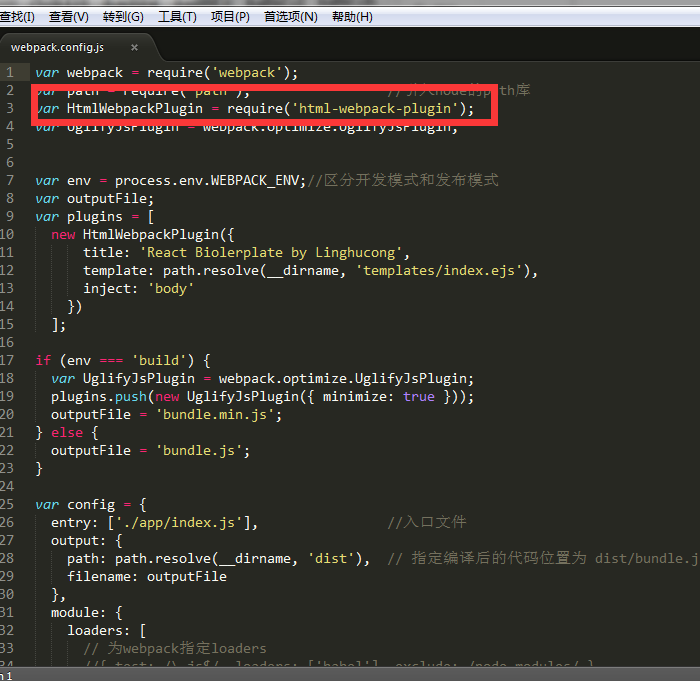
title: 'React Biolerplate by Linghucong'

})

]

(这个时候第一个坑.初学者不知道加在哪里.我来截图告诉你)

1.首先要引入.



2. 按照他的这一步.加在config文件里就行.记得和上一段要用”,”隔开

现在在命令行下再次执行webpack命令，会看到在dist目录下生成了两个文件：bundle.js和index.html。其中index.html内容如下：

##### dist/index.html

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>React Biolerplate by Linghucong</title>

</head>

<body>

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

</html>

有必要提一下，如果我们安装webpack的时候使用的是全局安装选项（npm install -g webpack），可以在命令行中直接执行webpack命令；如果没有使用-g，那么要用的webpack可执行文件位于：

./node\_modules/.bin/webpack

我们可以在package.json中为此命令增加一个快捷方式：

# package.json

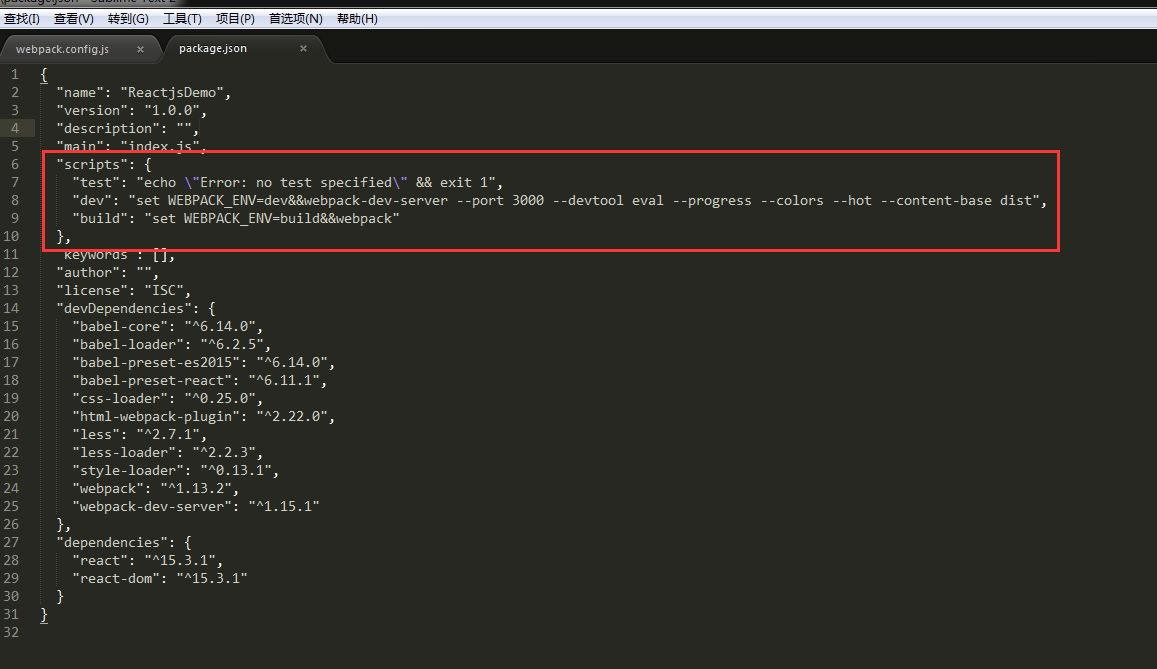
... other stuff

"scripts": {

"build": "./node\_modules/.bin/webpack"

}

(如果不知道这段代码加在哪里请看我的截图)



现在就可以直接使用命令npm run build来执行webpack了。

$ npm run build

> react\_boilerplate@1.0.0 build D:\node\react\_boilerplate

> webpack

### 安装webpack-dev-server

webpack-dev-server可以让我们在本地启动一个web服务器，使我们更方便的查看正在开发的项目。其安装也十分简单：

npm i webpack-dev-server --save-dev

然后在webpack.config.js文件中作如下修改：

# webpack.config.js

# ...

entry: [

'webpack/hot/dev-server',

'webpack-dev-server/client?http://localhost:3000',

'./app/index.js' //入口文件

],

# ...

(依然截图)



我们可以在package.json中增加webpack-dev-server的快捷方式：

# package.json

... other stuff

"scripts": {

"dev": "webpack-dev-server --port 3000 --devtool eval --progress --colors --hot --content-base dist",

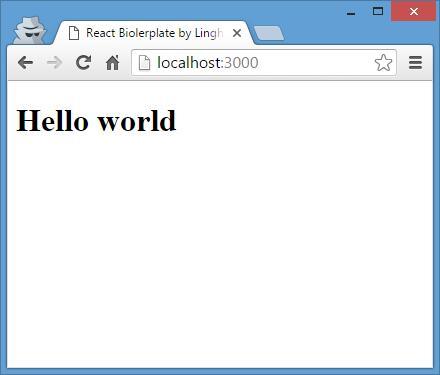
"build": "./node\_modules/.bin/webpack"

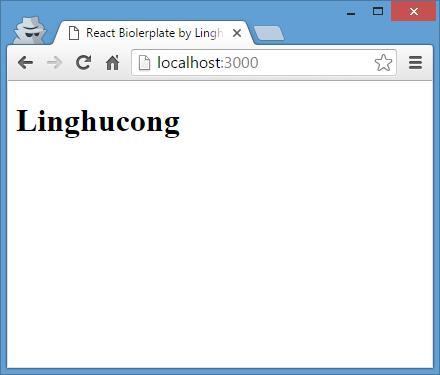
}

(这个上边已经有截图啦)

配置中指定web服务器端口号为3000，指定目录为dist。  
运行npm run dev：

$ npm run dev

web服务器启动完毕，此时访问 <http://localhost:3000/> 就可以看到我们的“Hello world”了。  


需要特别说明的是，webpack-dev-server是支持热加载的，也就是说我们对代码的改动，保存的时候会自动更新页面。比如我们在文件中将“Hello world”改为“Linghucong”，会看到页面实时更新了，无须再按F5刷新，爽吧？！  


webpack-dev-server的配置还可以放在webpack.config.js中，需要使用一个devServer属性，详细可以[参考官方文档](https://webpack.github.io/docs/webpack-dev-server.html)。

### 处理CSS样式

项目中使用CSS是必不可少的。webpack中使用  
loader的方式来处理各种各样的资源，根据设定的规则，会找到相应的文件路径，然后使用各自的loader来处理。CSS文件也需要特定的loader，一般需要使用两个：css-loader和 style-loader，如果使用LESS或者SASS还需要加载对应的loader。这里我们使用LESS，因此安装loaders:

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

##### 踩坑提醒

npm3.0以上需要单独安装less：npm install less --save-dev。(这个貌似是必须执行一下)

然后在文件webpack.config.js中配置：

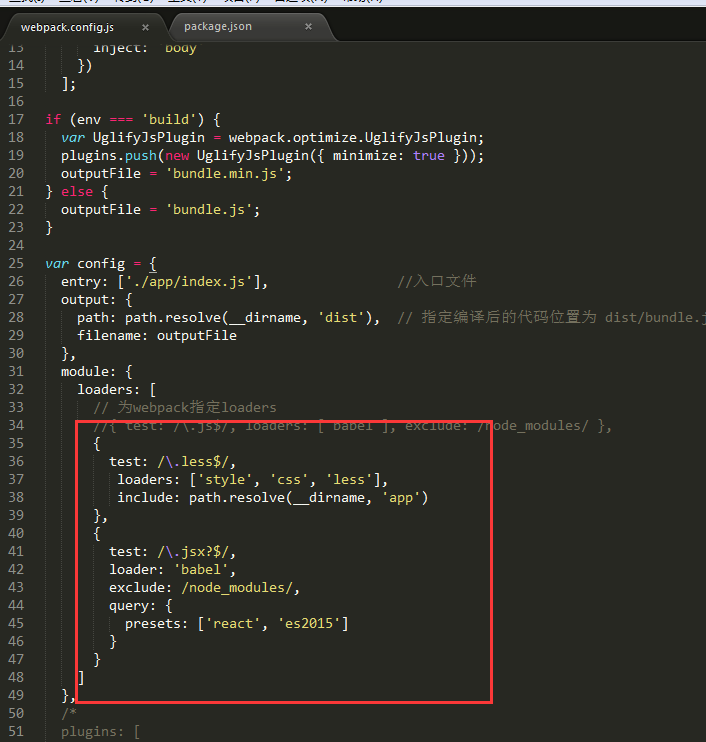
{

test: /\.less$/,

loaders: ['style', 'css', 'less'],

include: path.resolve(\_\_dirname, 'app')

}



可以看到，test里面包含一个正则，包含需要匹配的文件，loaders是一个数组，包含要处理这些文件的loaders，注意loaders的执行顺序是从右到左的。

新建一个LESS文件/app/index.less，其内容如下：

h1 {

color: green;

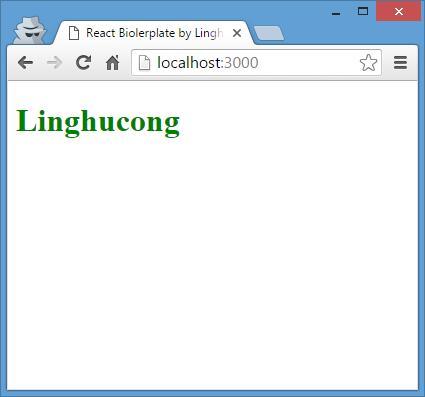
}

在入口文件index.js中引入这个文件：

require('./index.less');

然后运行webpack进行编译：npm run build:

$ npm run build

可以看到， <http://localhost:3000/> 页面上的文字已经变成绿色了。  


到目前为止的代码可以在[react\_boilerplate \_v1](https://github.com/jiji262/react_boilerplate/tree/master/_tutorial_/react_boilerplate_v1)中查看。

# webpack 支持ES6

## Javascript包管理格式

Javascript中的包管理比较常见的有几种方式：

#### CommonJS

//CommonJS 定义的是模块的同步加载，主要用于NodeJS

var MyModule = require('./MyModule');

// export at module root

module.exports = function() { ... };

// alternatively, export individual functions

exports.hello = function() {...};

#### AMD

//AMD 是异步加载，比如require.js使用这种规范

define(['./MyModule.js'], function (MyModule) {

// export at module root

return function() {};

});

// or

define(['./MyModule.js'], function (MyModule) {

// export as module function

return {

hello: function() {...}

};

});

#### ES6

//ES6 变得越来越主流了

import MyModule from './MyModule.js';

// export at module root

export default function () { ... };

// or export as module function,

// you can have multiple of these per module

export function hello() {...};

还有其他格式如UMD、CMD等，在此不再一一介绍。webpack对这些模块格式都可以很好的支持。在我们之前的项目中使用的是CommonJS格式的模块管理，但是随着ES6的普及和应用，同时得益于强大的[Babel](https://babeljs.io/)的存在，使我们可以方便的使用ES6的语法，而不必考虑浏览器支持的问题。

## webpack支持ES6语法

在webpack中支持ES6同样只需要安装配置相应的loader就可以了。

安装如下：

npm install babel-loader babel-core babel-preset-es2015 babel-preset-react --save-dev

在webpack.config.js中添加loader如下：

{

test: /\.jsx?$/,

loader: 'babel',

exclude: /node\_modules/,

query: {

presets: ['react', 'es2015']

}

}

由于后边需要支持React的jsx文件，所以我们在这里安装了babel-preset-react。

顺便提一下，我们可以在项目根目录下创建一个.babelrc文件，将loader中的presets放在文件.babelrc中：

# .babelrc

{

"presets": ["react", "es2015"]

}

此时我们运行npm run build，正常编译后，使用npm run dev，启动web服务器，打开 <http://localhost:3000/> 可以看到页面已经可以正常显示了。

##### 踩坑提醒

如果上面对于loader的配置写为（注意这里是loaders不是loader）：

{

test: /\.jsx?$/,

loaders: ['babel'],

exclude: /node\_modules/,

query: {

presets: ['es2015', 'react']

}

}

则可能会出现这样的错误：

$ npm run build

> react\_boilerplate@1.0.0 build D:\node\react\_boilerplate

> webpack

D:\node\react\_boilerplate\node\_modules\webpack-core\lib\LoadersList.js:54

if(!element.loader || element.loader.indexOf("!") >= 0) throw new Error("Cannot define 'query' and multiple loaders in loaders list");

^

Error: Cannot define 'query' and multiple loaders in loaders list

at getLoadersFromObject (D:\node\react\_boilerplate\node\_modules\webpack-core\lib\LoadersList.js:54:65)

at LoadersList. (D:\node\react\_boilerplate\node\_modules\webpack-core\lib\LoadersList.js:78:12)

at Array.map (native)

at LoadersList.match

...

原因是使用了多个loader，而query仅仅作用于babel-loader。如果非要使用loaders加载多个loader，可以做如下修改：

var babelPresets = {presets: ['react', 'es2015']};

......

loaders: ['other-loader', 'babel-loader?'+JSON.stringify(babelPresets)]

......

到目前为止的代码可以在[react\_boilerplate \_v2](https://github.com/jiji262/react_boilerplate/tree/master/_tutorial_/react_boilerplate_v2)中查看。

# 在项目中支持使用React

关于React的介绍和基本概念相信你已经有所了解，如果需要，可以参考本文最后的“参考阅读”中的链接，在此不再详述。

## 安装React

npm install react react-dom --save

这里我们使用的版本是15.0.1。

$ npm install react react-dom --save

react\_boilerplate@1.0.0 D:\node\react\_boilerplate

+-- react@15.0.1

| `-- fbjs@0.8.1

| +-- isomorphic-fetch@2.2.1

| | +-- node-fetch@1.5.1

| | | +-- encoding@0.1.12

| | | | `-- iconv-lite@0.4.13

| | | `-- is-stream@1.1.0

| | `-- whatwg-fetch@0.11.0

| `-- ua-parser-js@0.7.10

`-- react-dom@15.0.1

## 改造项目结构

在项目中我们使用了html-webpack-plugin插件来用webpack自动生成入口的index.html文件，但是里面的内容我们没法控制。html-webpack-plugin提供了一种模板的机制，可以让我们对生成的文件内容进行定制。

### 创建模板文件

我们使用一个新的目录templates用于存放模板文件，新建一个index.ejs文件：

##### templates/index.ejs

<%= htmlWebpackPlugin.options.title %>

Welcome to New Page

### 修改 html-webpack-plugin 设置

修改webpack.config.js文件如下：

plugins: [

new HtmlwebpackPlugin({

title: 'React Biolerplate by Linghucong',

template: path.resolve(\_\_dirname, 'templates/index.ejs'),

inject: 'body'

})

]

关于 html-webpack-plugin 更多高级用法可以[参考其项目主页](https://github.com/ampedandwired/html-webpack-plugin)。

### 支持sourcemap

sourcemap的作用各位自行Google吧。要生成编译出的js文件的sourcemap文件，只需要在webpack配置文件中加入如下一行配置即可：

devtool: 'source-map',

运行npm run build可以看到一个会在dist目录生成一个新的文件bundle.js.map，这就是sourcemap文件。

### Minification 代码压缩

要对生成的js文件进行压缩，需要使用一个新的插件：UglifyJsPlugin。  
修改webpack.config.js如下：

......

var UglifyJsPlugin = webpack.optimize.UglifyJsPlugin;

var config = {

......

plugins: [

......

new UglifyJsPlugin({ minimize: true })

]

}

module.exports = config;

运行npm run build可以看到生成的bundle.js文件已经被minify了。

在实际的项目开发中，我们在开发阶段一般不需要将代码minify，因为压缩之后很不方便调试。因此，我们有必要将开发模式和发布模式区分开。我们通过设置process.env.WEBPACK\_ENV来做区分。  
修改webpack.config.js如下：

......

var env = process.env.WEBPACK\_ENV;

var outputFile;

var plugins = [new HtmlwebpackPlugin({

title: 'React Biolerplate by Linghucong',

template: path.resolve(\_\_dirname, 'templates/index.ejs'),

inject: 'body'

})];

if (env === 'build') {

var UglifyJsPlugin = webpack.optimize.UglifyJsPlugin;

plugins.push(new UglifyJsPlugin({ minimize: true }));

outputFile = 'bundle.min.js';

} else {

outputFile = 'bundle.js';

}

var config = {

......

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: outputFile

},

......

plugins: plugins

}

module.exports = config;

同时需要修改npm run的快捷方式，在package.json文件中修改如下：

"scripts": {

"dev": "WEBPACK\_ENV=dev webpack-dev-server --port 3000 --devtool eval --progress --colors --hot --content-base dist",

"build": "WEBPACK\_ENV=build webpack"

},

##### 踩坑提醒

在Windows系统上不能像上述那样设置WEBPACK\_ENV，可以使用set来设置，如下：

"scripts": {

"test": "mocha --compilers js:babel-register --require ./test/test\_helper.js --recursive ./test",

"test:watch": "npm test -- --watch",

"dev": "set WEBPACK\_ENV=dev&&webpack-dev-server --port 3000 --devtool eval --progress --colors --hot --content-base dist",

"build": "set WEBPACK\_ENV=build&&webpack"

},

### 更新项目代码

对我们的组件稍作修改：

import './index.less';

import component from './component';

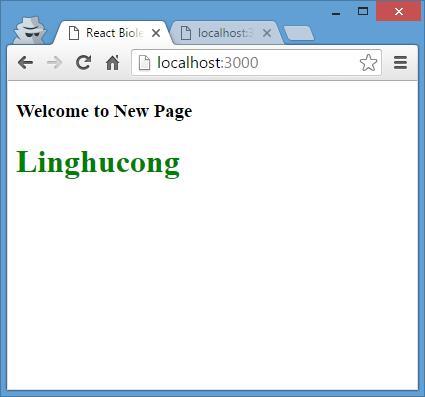
let content = document.getElementById("content");

content.appendChild(component());

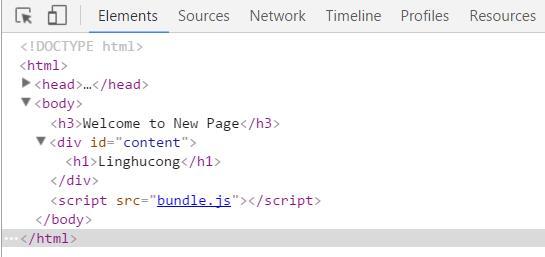
然后编译，运行：

$ npm run build

$ npm run dev

可以看到，目前访问 <http://localhost:3000/> 的页面显示已经发生了变化。  


通过查看源代码，可以看到我们页面正是应用了我们的模板文件。



到目前为止的代码可以在[react\_boilerplate \_v3](https://github.com/jiji262/react_boilerplate/tree/master/_tutorial_/react_boilerplate_v3)中查看。

## 创建React组件

我们将app/index.js修改一下，创建一个新的React组件。

##### app/index.js

import React from 'react';

import ReactDOM from 'react-dom';

class HelloReact extends React.Component {

render() {

return Hello React!

}

}

ReactDOM.render(, document.getElementById('content'));

代码十分简单，引入了react和react-dom，创建了一个叫做HelloReact的组件，并将其渲染到页面上id为content的DOM元素内。

## 编译React代码

我们在之前已经在webpack的配置中配置好了对React的支持，因此目前不需要做什么修改了。

npm run build之后就可以在页面上看到“Hello React!”了。

至此，我们基于ES6并使用webpack和Babel的React初始项目已经可以完美运行了。

(再次感谢作者的耐心文档.如果有问题.文档顶部是原文连接地址.附上我的环境配置文件.)