react全家桶从0到1(最新)

本文从零开始，逐步讲解如何用react全家桶搭建一个完整的react项目。文中针对react、webpack、babel、react-route、redux、redux-saga的核心配置会加以讲解，希望通过这个项目，可以系统的了解react技术栈的主要知识，避免搭建一次后面就忘记的情况。

[代码库：https://github.com/teapot-py/react-demo](https://link.segmentfault.com/?url=https%3A%2F%2Fgithub.com%2Fteapot-py%2Freact-demo)

**首先关于主要的npm包版本列一下：**

1. react@16.7.0
2. webpack@4.28.4
3. babel@7+
4. react-router@4.3.1
5. redux@4+

从webpack开始

思考一下webpack到底做了什么事情？其实简单来说，就是从入口文件开始，不断寻找依赖，同时为了解析各种不同的文件加载相应的loader，最后生成我们希望的类型的目标文件。

这个过程就像是在一个迷宫里寻宝，我们从入口进入，同时我们也会不断的接收到下一处宝藏的提示信息，我们对信息进行解码，而解码的时候可能需要一些工具，比如说钥匙，而loader就像是这样的钥匙，然后得到我们可以识别的内容。

**回到我们的项目，首先进行项目的初始化，分别执行如下命令**

mkdir react-demo // 新建项目文件夹

cd react-demo // cd到项目目录下

npm init // npm初始化

**引入webpack**

npm i webpack --save

touch webpack.config.js

**对webpack进行简单配置，更新webpack.config.js**

const path = require('path');

module.exports = {

entry: './app.js', // 入口文件

output: {

path: path.resolve(\_\_dirname, 'dist'), // 定义输出目录

filename: 'my-first-webpack.bundle.js' // 定义输出文件名称

}

};

**更新package.json文件，在scripts中添加webpack执行命令**

"scripts": {

"dev": "./node\_modules/.bin/webpack --config webpack.config.js"

}

**如果有报错请按提示安装webpack-cli**

npm i webpack-cli

**执行webpack**

npm run dev

如果在项目文件夹下生成了dist文件，说明我们的配置是没有问题的。

接入react

**安装react相关包**

npm install react react-dom --save

**更新app.js入口文件**

import React from 'react

import ReactDom from 'react-dom';

import App from './src/views/App';

ReactDom.render(<App />, document.getElementById('root'));

**创建目录 src/views/App，在App目录下，新建index.js文件作为App组件，index.js文件内容如下：**

import React from 'react';

class App extends React.Component {

constructor(props) {

super(props);

}

render() {

return (<div>App Container</div>);

}

}

export default App;

**在根目录下创建模板文件index.html**

<!DOCTYPE html>

<html>

<head>

<title>index</title>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width,initial-scale=1, maximum-scale=1, minimum-scale=1, user-scalable=no">

</head>

<body>

<div id="root"></div>

</body>

</html>

到了这一步其实关于react的引入就OK了，不过目前还有很多问题没有解决

1. 如何解析JS文件的代码？
2. 如何将js文件加入模板文件中？

Babel解析js文件

Babel是一个工具链，主要用于在旧的浏览器或环境中将ECMAScript2015+的代码转换为向后兼容版本的JavaScript代码。

**安装babel-loader，@babel/core，@babel/preset-env，@babel/preset-react**

npm i babel-loader@8 @babel/core @babel/preset-env @babel/preset-react -D

1. babel-loader：使用Babel转换JavaScript依赖关系的Webpack加载器, 简单来讲就是webpack和babel中间层，允许webpack在遇到js文件时用bable来解析
2. @babel/core：即babel-core，将ES6代码转换为ES5。7.0之后，包名升级为@babel/core。@babel相当于一种官方标记，和以前大家随便起名形成区别。
3. @babel/preset-env：即babel-preset-env，根据您要支持的浏览器，决定使用哪些transformations / plugins 和 polyfills，例如为旧浏览器提供现代浏览器的新特性。
4. @babel/preset-react：即 babel-preset-react，针对所有React插件的Babel预设，例如将JSX转换为函数.

**更新webpack.config.js**

module: {

rules: [

{

test: /\.js$/, // 匹配.js文件

exclude: /node\_modules/,

use: {

loader: 'babel-loader'

}

}

]

}

**根目录下创建并配置.babelrc文件**

{

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

}

**配置HtmlWebPackPlugin**

这个插件最主要的作用是将js代码通过<script>标签注入到 HTML 文件中

npm i html-webpack-plugin -D

**webpack新增HtmlWebPackPlugin配置**

至此，我们看一下webpack.config.js文件的完整结构

const path = require('path');

const HtmlWebPackPlugin = require('html-webpack-plugin');

module.exports = {

entry: './app.js',

output: {

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

filename: 'my-first-webpack.bundle.js'

},

mode: 'development',

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader'

}

}

]

},

plugins: [

new HtmlWebPackPlugin({

template: './index.html',

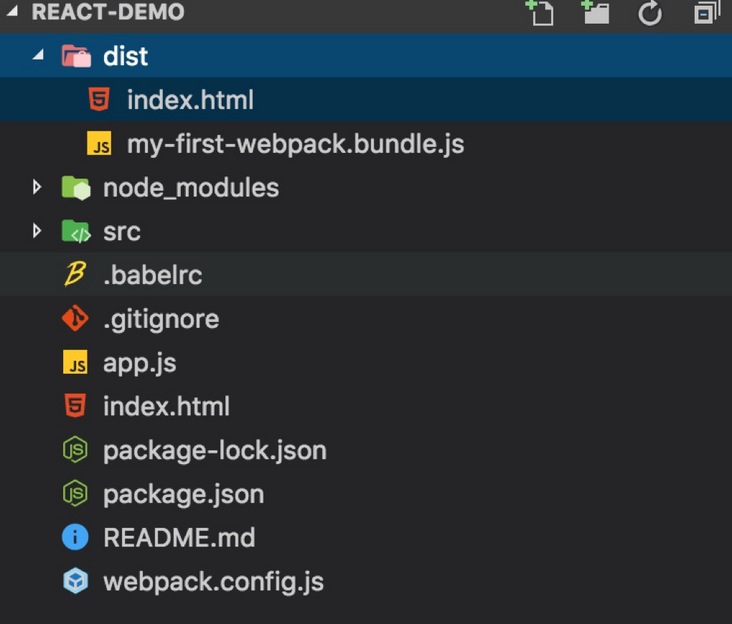
filename: path.resolve(\_\_dirname, 'dist/index.html')

})

]

};

**执行 npm run start，生成 dist文件夹**

当前目录结构如下  


可以看到在dist文件加下生成了index.html文件，我们在浏览器中打开文件即可看到App组件内容。

配置 webpack-dev-server

webpack-dev-server可以极大的提高我们的开发效率，通过监听文件变化，自动更新页面

**安装 webpack-dev-server 作为 dev 依赖项**

npm i webpack-dev-server -D

**更新package.json的启动脚本**

“dev": "webpack-dev-server --config webpack.config.js --open"

**webpack.config.js新增devServer配置**

devServer: {

hot: true, // 热替换

contentBase: path.join(\_\_dirname, 'dist'), // server文件的根目录

compress: true, // 开启gzip

port: 8080, // 端口

},

plugins: [

new webpack.HotModuleReplacementPlugin(), // HMR允许在运行时更新各种模块，而无需进行完全刷新

new HtmlWebPackPlugin({

template: './index.html',

filename: path.resolve(\_\_dirname, 'dist/index.html')

})

]

引入redux

redux是用于前端数据管理的包，避免因项目过大前端数据无法管理的问题，同时通过单项数据流管理前端的数据状态。

**创建多个目录**

1. 新建src/actions目录，用于创建action函数
2. 新建src/reducers目录，用于创建reducers
3. 新建src/store目录，用于创建store

下面我们来通过redux实现一个计数器的功能

**安装依赖**

npm i redux react-redux -D

**在actions文件夹下创建index.js文件**

export const increment = () => {

return {

type: 'INCREMENT',

};

};

**在reducers文件夹下创建index.js文件**

const initialState = {

number: 0

};

const incrementReducer = (state = initialState, action) => {

switch(action.type) {

case 'INCREMENT': {

state.number += 1

return { ...state }

break

};

default: return state;

}

};

export default incrementReducer;

**更新store.js**

import { createStore } from 'redux';

import incrementReducer from './reducers/index';

const store = createStore(incrementReducer);

export default store;

**更新入口文件app.js**

import App from './src/views/App';

import ReactDom from 'react-dom';

import React from 'react';

import store from './src/store';

import { Provider } from 'react-redux';

ReactDom.render(

<Provider store={store}>

<App />

</Provider>

, document.getElementById('root'));

**更新App组件**

import React from 'react';

import { connect } from 'react-redux';

import { increment } from '../../actions/index';

class App extends React.Component {

constructor(props) {

super(props);

}

onClick() {

this.props.dispatch(increment())

}

render() {

return (

<div>

<div>current number： {this.props.number} <button onClick={()=>this.onClick()}>点击+1</button></div>

</div>

);

}

}

export default connect(

state => ({

number: state.number

})

)(App);



点击旁边的数字会不断地+1

引入redux-saga

redux-saga通过监听action来执行有副作用的task，以保持action的简洁性。引入了sagas的机制和generator的特性，让redux-saga非常方便地处理复杂异步问题。  
redux-saga的原理其实说起来也很简单，通过劫持异步action，在redux-saga中进行异步操作，异步结束后将结果传给另外的action。

下面就接着我们计数器的例子，来实现一个异步的+1操作。

**安装依赖包**

npm i redux-saga -D

**新建src/sagas/index.js文件**

import { delay } from 'redux-saga'

import { put, takeEvery } from 'redux-saga/effects'

export function\* incrementAsync() {

yield delay(2000)

yield put({ type: 'INCREMENT' })

}

export function\* watchIncrementAsync() {

yield takeEvery('INCREMENT\_ASYNC', incrementAsync)

}

解释下所做的事情，将watchIncrementAsync理解为一个saga，在这个saga中监听了名为INCREMENT\_ASYNC的action，当INCREMENT\_ASYNC被dispatch时，会调用incrementAsync方法，在该方法中做了异步操作，然后将结果传给名为INCREMENT的action进而更新store。

**更新store.js**

在store中加入redux-saga中间件

import { createStore, applyMiddleware } from 'redux';

import incrementReducer from './reducers/index';

import createSagaMiddleware from 'redux-saga'

import { watchIncrementAsync } from './sagas/index'

const sagaMiddleware = createSagaMiddleware()

const store = createStore(incrementReducer, applyMiddleware(sagaMiddleware));

sagaMiddleware.run(watchIncrementAsync)

export default store;

**更新App组件**

在页面中新增异步提交按钮，观察异步结果

import React from 'react';

import { connect } from 'react-redux';

import { increment } from '../../actions/index';

class App extends React.Component {

constructor(props) {

super(props);

}

onClick() {

this.props.dispatch(increment())

}

onClick2() {

this.props.dispatch({ type: 'INCREMENT\_ASYNC' })

}

render() {

return (

<div>

<div>current number： {this.props.number} <button onClick={()=>this.onClick()}>点击+1</button></div>

<div>current number： {this.props.number} <button onClick={()=>this.onClick2()}>点击2秒后+1</button></div>

</div>

);

}

}

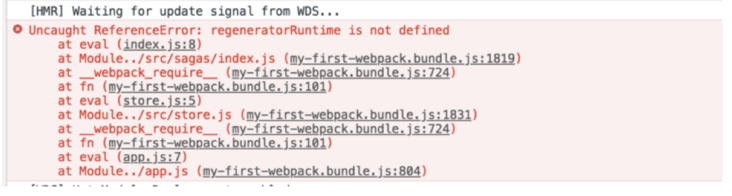
export default connect(

state => ({

number: state.number

})

)(App);

观察结果我们会发现如下报错：  


这是因为在redux-saga中用到了Generator函数，以我们目前的babel配置来说并不支持解析generator，需要安装@babel/plugin-transform-runtime

npm install --save-dev @babel/plugin-transform-runtime

这里关于babel-polyfill、和transfor-runtime做进一步解释

babel-polyfill

Babel默认只转换新的JavaScript语法，而不转换新的API。例如，Iterator、Generator、Set、Maps、Proxy、Reflect、Symbol、Promise等全局对象，以及一些定义在全局对象上的方法（比如Object.assign）都不会转译。如果想使用这些新的对象和方法，必须使用 babel-polyfill，为当前环境提供一个垫片。

babel-runtime

Babel转译后的代码要实现源代码同样的功能需要借助一些帮助函数，而这些帮助函数可能会重复出现在一些模块里，导致编译后的代码体积变大。  
Babel 为了解决这个问题，提供了单独的包babel-runtime供编译模块复用工具函数。  
在没有使用babel-runtime之前，库和工具包一般不会直接引入 polyfill。否则像Promise这样的全局对象会污染全局命名空间，这就要求库的使用者自己提供 polyfill。这些 polyfill一般在库和工具的使用说明中会提到，比如很多库都会有要求提供 es5的polyfill。  
在使用babel-runtime后，库和工具只要在 package.json中增加依赖babel-runtime，交给babel-runtime去引入 polyfill 就行了；  
[详细解释可以参考](https://segmentfault.com/q/1010000005596587?from=singlemessage&isappinstalled=1)

babel presets 和 plugins的区别

Babel插件一般尽可能拆成小的力度，开发者可以按需引进。比如对ES6转ES5的功能，Babel官方拆成了20+个插件。  
这样的好处显而易见，既提高了性能，也提高了扩展性。比如开发者想要体验ES6的箭头函数特性，那他只需要引入transform-es2015-arrow-functions插件就可以，而不是加载ES6全家桶。  
但很多时候，逐个插件引入的效率比较低下。比如在项目开发中，开发者想要将所有ES6的代码转成ES5，插件逐个引入的方式令人抓狂，不单费力，而且容易出错。  
这个时候，可以采用Babel Preset。  
可以简单的把Babel Preset视为Babel Plugin的集合。比如babel-preset-es2015就包含了所有跟ES6转换有关的插件。

更新.babelrc文件配置，支持genrator

{

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

"plugins": [

[

"@babel/plugin-transform-runtime",

{

"corejs": false,

"helpers": true,

"regenerator": true,

"useESModules": false

}

]

]

}

  
点击按钮会在2秒后执行+1操作。

引入react-router

在web应用开发中，路由系统是不可或缺的一部分。在浏览器当前的URL发生变化时，路由系统会做出一些响应，用来保证用户界面与URL的同步。随着单页应用时代的到来，为之服务的前端路由系统也相继出现了。而react-route则是与react相匹配的前端路由。

**引入react-router-dom**

npm install --save react-router-dom -D

更新app.js入口文件增加路由匹配规则

import App from './src/views/App';

import ReactDom from 'react-dom';

import React from 'react';

import store from './src/store';

import { Provider } from 'react-redux';

import { BrowserRouter as Router, Route, Switch } from "react-router-dom";

const About = () => <h2>页面一</h2>;

const Users = () => <h2>页面二</h2>;

ReactDom.render(

<Provider store={store}>

<Router>

<Switch>

<Route path="/" exact component={App} />

<Route path="/about/" component={About} />

<Route path="/users/" component={Users} />

</Switch>

</Router>

</Provider>

, document.getElementById('root'));

更新App组件，展示路由效果

import React from 'react';

import { connect } from 'react-redux';

import { increment } from '../../actions/index';

import { Link } from "react-router-dom";

class App extends React.Component {

constructor(props) {

super(props);

}

onClick() {

this.props.dispatch(increment())

}

onClick2() {

this.props.dispatch({ type: 'INCREMENT\_ASYNC' })

}

render() {

return (

<div>

<div>react-router 测试</div>

<nav>

<ul>

<li>

<Link to="/about/">页面一</Link>

</li>

<li>

<Link to="/users/">页面二</Link>

</li>

</ul>

</nav>

<br/>

<div>redux & redux-saga测试</div>

<div>current number： {this.props.number} <button onClick={()=>this.onClick()}>点击+1</button></div>

<div>current number： {this.props.number} <button onClick={()=>this.onClick2()}>点击2秒后+1</button></div>

</div>

);

}

}

export default connect(

state => ({

number: state.number

})

)(App);

  
点击列表可以跳转相关路由

总结

至此，我们已经一步步的，完成了一个简单但是功能齐全的react项目的搭建，下面回顾一下我们做的工作

1. 引入webpack
2. 引入react
3. 引入babel解析react
4. 接入webpack-dev-server提高前端开发效率
5. 引入redux实现一个increment功能
6. 引入redux-saga实现异步处理
7. 引入react-router实现前端路由

麻雀虽小，五脏俱全，希望通过最简单的代码快速的理解react工具链。其实这个小项目中还是很多不完善的地方，比如说样式的解析、Eslint检查、生产环境配置，虽然这几项是一个完整项目不可缺少的部分，但是就demo项目来说，对我们理解react工具链可能会有些干扰，所以就不在项目中加了。  
后面我会新建一个分支，把这些完整的功能都加上，同时也会对当前的目录结构进行优化。