链滴

react-starter 脚手架搭建过程

作者：wuhongxu

原文链接：https://ld246.com/article/1489956628548

来源网站：链滴

本文献给一些react新手，不对，应该说针对想要通过react进入前端世界的程序猿们~

背景

公司的项目需要使用react，两三个月前还是一个基本不怎么懂前端，当时的我除了jquery什么都不道，但是公司要求使用react开发新的项目，完全不懂前端的我只能去试着学习使用react进行开发，后什么nodejs/webpack/es5/es6/babel/jsx一堆堆全部涌过来，讲真，彻底被前端的复杂世界搞懵了，从来没想过前端是这么的复杂。然后匆匆忙忙的开始学习，循环往复十来天。接着公司要求写项了，我擦，我还什么都没学会，没办法，硬着头皮上，去全球最大同性交友网站（#雾）瞅了瞅，找个star很多的脚手架，开始了开发之旅，其中的痛苦真的是不忍赘述，经常为了目标熬夜赶工，结果是各种拖拖踏踏，一直没能完成。还好一个月后被叫去做其他项目，不然真的是崩崩崩。接下来的时开始了真正的探索react路途...一两个月的工作之余熬夜苦学之后，尝试了很多不同的脚手架之后，开有了自己对于开发、编译的需求，处于对于想要将一个技术完全征服的欲望，于是开始准备搭建自己react脚手架

准备工作

知识准备

react自不必说，首先是去了解nodejs、webpack、babel、es6等等，多学习源生js，彻底抛弃jquer才能真正感受到源生js带来的魅力，了解一些js规范，比如commonjs之类的。一定要用心去感悟（）。

需求准备

对于个人的一路走来的感觉来说，前端开发经验少的人，应该还是先使用别人的脚手架，多读读人家readme的说明，了解了解别人对于本项目的一些规范或者需求，这里推荐一个react starter kit，感确实很全面，不过里面没有使用jsx语法定义路由，而是使用的webpack的api来实现的按需加载，开可能会让人有点昏，不过多去了解了解也就没什么障碍，官方也给出了很好的解释。多写写，多动手就会明白自己的一些需求了。这里列出我自己的一些想要的需求。

开发模式下：

1. 热加载（没它开发会累哭）
2. 能与后台交互时的解决跨域问题

生产模式下：

1. 能把react模块分开
2. 能自动将生成js文件绑定到html页面
3. 压缩js、css代码
4. 想要使用ant design开发，但是antd样式文件较大，能够按需加载

通用的：

1. 使用es6、jsx语法
2. 能生成map文件
3. 能加载各种文件：json，图片，字体，css，scss，less

4. 找路径不要那么麻烦的使用 '../../'之类的，能直接从项目跟路径找。
5. 命令能够兼容linux或者windows
6. 使用eslint进行代码检查
7. 使用esformatter进行代码格式化
8. 能自动处理css样式兼容性问题

开始搭建

虽然我搭建环境有时候在linux有时候在windows，不过写此文的时候处于windows环境下，懒得切统，所以统一使用windows环境，一般来说linux没什么不同，想起来有不同的时候我会说。

工具推荐

强烈推荐vscode，支持得真的好=-=，虽然我还不是很用的来。

初始化

安装nodejs，我使用nvm来管理不同的nodejs版本，而且通过nvm安装nodejs也超级简单，所以推用这种方式吧，下载地址。下载安装，配置环境变量，搞定，具体教程自行百度。然后使用nvm安装新的nodejs稳定版，目前是6.10.0。 nvm install 6.10.0，然后等安装完毕，新建一个react-start文夹，cd进去， 使用命令 nvm use 6.10.0 ，强烈建议安装cnpm 配置淘宝源，这样安装依赖会快得多否则等秃（#雾，使用命令 npm install -g cnpm --registry=https://registry.npm.taobao.org,使用npm init初始化项目，基本上可以一路回车下去。完成初始化，安装全局webpack依赖cnpm install ebpack -g

搭建骨架

跟路径新建webpack.config.js，webpack会默认读取这个文件，先编写个大概，然后慢慢去补

```
'use strict'

const path = require( 'path' )
const webpack = require( 'webpack' )
const rootPath = path.resolve( __dirname )
const srcPath = path.join( rootPath, 'src' )
const common = {
  rootPath: rootPath,
  srcPath: srcPath,
  dist: path.join( rootPath, 'dist' ),
  indexHtml: path.join( srcPath, 'index.html' ),
  staticDir: path.join( rootPath, 'static' ),
  entry:path.join( common.srcPath, 'index.js' )
}
var webpackConfig = {
    entry: common.entry,
    output: {
      filename: 'bundle.js',
      path: common.dist,
    },
```

```
}

module.exports = webpackConfig
```

新建src文件夹，我们编写的js源代码文件就写在这里面了

进入src文件夹，新建index.js

```
    var render = function(){
        console.log('hello world');
    }
```

根目录新建dist文件夹，这是我们生成编译后的代码文件夹，新建index.html

```
    <!DOCTYPE html>
<html>
  <head>
  <title>react-starter</title>
  </head>
  <body>
    <script src="bundle.js"></script>
  </body>
</html>
```

根目录运行webpack，等待编译完成，浏览器打开index.html，打开控制台就能够看到hello world
。

按照需求安装依赖编写webpack.config.js

为了能够将项目迁移到其他电脑，本地安装webpackcnpm install webpack --save-dev

加入环境变量

为了实现linux和windows都能设置环境变量，安装cross-env，cnpm install cross-env --save-dev

修改config，加入

```
const env = process
  .env
  .NODE_ENV
  .trim()
const isDev = (env === 'development')
```

实现热加载

安装webpack-dev-server,cnpm install webpack-dev-server，编写config，

```
'use strict'

const path = require( 'path' )
const webpack = require( 'webpack' )
const rootPath = path.resolve( __dirname )
const srcPath = path.join( rootPath, 'src' )
```

```
const env = process
  .env
  .NODE_ENV
  .trim()
const isDev = (env === 'development')

const common = {
  rootPath: rootPath,
  srcPath: srcPath,
  dist: path.join( rootPath, 'dist' ),
  indexHtml: path.join( srcPath, 'index.html' ),
  staticDir: path.join( rootPath, 'static' ),
  entry:path.join( common.srcPath, 'index.js' )
}
//开发模式下修改入口文件
if ( isDev ) {
  common.entry = [
    'react-hot-loader/patch',
    // activate HMR for React

    'webpack-dev-server/client?http://localhost:3000',
    // bundle the client for webpack-dev-server and connect to the provided endpoint

    'webpack/hot/only-dev-server',
    // bundle the client for hot reloading only- means to only hot reload for
    // successful updates

    path.join( common.srcPath, 'index.js' )
  ]
}
var webpackConfig = {
    entry: common.entry,
    output: {
      filename: 'bundle.js',
      path: common.dist,
      publicPath: '/', //让HMR知道在哪里加载热更新块所必需的
    },
}
//开发模式下添加devServer字段
//devServer候选字段参考https://webpack.js.org/configuration/dev-server/
if ( isDev ) {
  webpackConfig.devServer = {
    historyApiFallback:true,
    hot: true,
    contentBase: path.resolve( __dirname, 'dist' ),
    publicPath: '/',
    clientLogLevel: 'none', //日志
    compress: true, //压缩
    port:3000,
    stats: {
      colors: true
    },
    proxy:{
```

```
      '/api/*':{
        target: 'http://localhost',
        changeOrigin: true,
        secure: false,
      }
    }
  }
}

module.exports = webpackConfig
```

添加es6语法支持

webpack2.0好像默认支持es6语法，不过稳妥起见，还是使用Babel进行es2016转换

安装babel系列，cnpm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-stage-0 支持到js标准的stage 0阶段,

webpackConfig中添加module字段

```
module: {
    //webpack1.0中可以省略 '-loader'，但是官方说法为了有明确的区分，在webpack2.0中，不能省略
    rules: [
      {
        test: /\.(js)$/,
        loader: 'babel-loader',
        include: [
          path.join( common.rootPath, 'src' ), //转换src路径下的代码
        ],
        exclude: /node_modules/, //忽略node_modules路径代码
      }
    ]
}
```

添加react

安装react，cnpm install --save react react-dom react-router ，cnpm install babel-preset-react react-hot-loader@^3.0.0-beta.6 babel-plugin-transform-runtime babel-preset-react babel-preset-react-optimize --save-dev,继续修改webpackConfig,在上一个修改的代码中的test字段中，修为：test: /\.(js|jsx)$/,

根目录新建.babelrc

```
{
  "presets": ["es2015","stage-0", "react"],
  "plugins": ["transform-runtime","react-hot-loader/babel"],
  "env": {
    "production": {
      "presets": ["react-optimize"]
    }
  }
```

```
}
```

编写config

```
'use strict'

const path = require( 'path' )
const webpack = require( 'webpack' )

const rootPath = path.resolve( __dirname )
const srcPath = path.join( rootPath, 'src' )

const env = process
  .env
  .NODE_ENV
  .trim()
const isDev = (env === 'development')

const common = {
  rootPath: rootPath,
  srcPath: srcPath,
  dist: path.join( rootPath, 'dist' ),
  indexHtml: path.join( srcPath, 'index.html' ),
  staticDir: path.join( rootPath, 'static' )
}
if ( isDev ) {
  common.entry = [
    'react-hot-loader/patch',
    // activate HMR for React

    'webpack-dev-server/client?http://localhost:3000',
    // bundle the client for webpack-dev-server and connect to the provided endpoint

    'webpack/hot/only-dev-server',
    // bundle the client for hot reloading only- means to only hot reload for
    // successful updates

    path.join( common.srcPath, 'index.js' )
  ]
}
if ( isDev ) {
    new webpack.HotModuleReplacementPlugin(), // HMR全局启用
    new webpack.NamedModulesPlugin(), // 在HMR更新的浏览器控制台中打印更易读的模块名称
  ]
}
const webpackConfig = {
  entry: common.entry,
  output: {
    filename: 'bundle.js',
    path: common.dist,
    publicPath: '/', //让HMR知道在哪里加载热更新块所必需的
  },
  context: path.resolve( __dirname, 'src' ),
  module: {
```

```
    //webpack1.0中可以省略 '-loader'，但是官方说法为了有明确的区分，在webpack2.0中，不能省略
    rules:[
        {
        test: /\.(js|jsx)$/,
        loader: 'babel-loader',
        include: [
          path.join( common.rootPath, 'src' ), //转换src路径下的代码
        ],
        exclude: /node_modules/, //忽略node_modules路径代码
      }
    ]
  }
}
if ( isDev ) {
  webpackConfig.devServer = {
    historyApiFallback:true,
    hot: true,
    contentBase: path.resolve( __dirname, 'dist' ),
    publicPath: '/',
    clientLogLevel: 'none', //日志
    compress: true, //压缩
    port:3000,
    stats: {
      colors: true
    },
    proxy:{
      '/api/*':{
        target: 'http://localhost',
        changeOrigin: true,
        secure: false,
      }
    }
  }
}

module.exports = webpackConfig
```

在html文件中加入

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

新建/src/components文件夹

新建/src/components/index.js

```
    import React,{Component} from 'react'
    export const App = ()=>(
        <div>hello world!<div>
    )
```

编写/src/index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
```

```
import { AppContainer } from 'react-hot-loader'
// AppContainer is a necessary wrapper component for HMR

const Root = document.getElementById( 'root' )

const isDev = !(process.env.NODE_ENV === 'development')

let render = () => {
  const Routes = require( './routes/index' ).default
  ReactDOM.render( <AppContainer><Routes/></AppContainer>, Root )
}

if ( isDev ) {
  if ( window.devToolsExtension ) {
    window.devToolsExtension.open()
  }
}

if ( isDev ) {
  if ( module.hot ) {
    // Development render functions
    const renderApp = render

    // Wrap render in try/catch
    render = () => {
        renderApp()
    }

    // Setup hot module replacement
    module.hot.accept( './components', () => setImmediate( () => {
      ReactDOM.unmountComponentAtNode( Root )
      render()
    } )
    )
  }
}

// Hot Module Replacement API
// if (module.hot) {
//   module
//     .hot
//     .accept('./', () => {
//       render(Routers)
//     })
// }

render()
```

运行命令 cross-env NODE_ENV=development webpack-dev-server 打开浏览器，输入localhost 3000看看是不是已经有了hell world啦？在源文件中将hello world修改成其他字样，看看是不是已能够自动刷新网页了？可喜可贺~

支持各种文件加载

在webpack概念中，各种文件都可以通过loader达到all in js的效果，从而让js能够做各种操作

所以接下来要安装各种loader，

cnpm install --save css-loader file-loader html-loader json-loader less less-loader node-sass sass-loader style-loader url-loader

简单说明（一下省略 “-loader”）：

- style:将css样式转换成 css in js，也就是说没有单独的css文件，全部都在js文件中夹杂着，这里引主要是为了热加载
- css、file、html、json、less、sass：顾名思义就是将这些类型文件加载到js中
- less、node-sass：因为less和sass是预处理语言，要使其发挥作用，必须加入这些支持
- url-loader:file-loader的加强版，可以使用limit参数，主要用来加载图片

修改config

```
/**
 * webpack2.0 配置文件
 * @authors wuhongxu (wuhongxu1208@gmail.com)
 * @date    2017-03-10 01:52:00
 * @version 1.0.0
 * @link <link>https://zido.site/</link>
 *
 */

'use strict'

const path = require( 'path' )
const webpack = require( 'webpack' )

const rootPath = path.resolve( __dirname )
const srcPath = path.join( rootPath, 'src' )

const env = process
  .env
  .NODE_ENV
  .trim()
const isDev = (env === 'development')

const common = {
  rootPath: rootPath,
  srcPath: srcPath,
  dist: path.join( rootPath, 'dist' ),
  indexHtml: path.join( srcPath, 'index.html' ),
  staticDir: path.join( rootPath, 'static' )
}
if ( isDev ) {
  common.entry = [
    'react-hot-loader/patch',
    // activate HMR for React

    'webpack-dev-server/client?http://localhost:3000',
    // bundle the client for webpack-dev-server and connect to the provided endpoint
```

```
    'webpack/hot/only-dev-server',
    // bundle the client for hot reloading only- means to only hot reload for
    // successful updates

    path.join( common.srcPath, 'index.js' )
  ]
} else {
  common.entry = {
    app: path.join( common.srcPath, 'index.js' )
}

const styleLoaders = {
  style: {
    loader: 'style-loader'
  },
  css: {
    loader: 'css-loader',
    options: {
      //将css进行hash编码，保证模块性，保证单独使用，而不会污染全局
      // modules: true
    }
  }
}

const webpackConfig = {
  entry: common.entry,
  output: {
    filename: 'bundle.js',
    path: common.dist,
    publicPath: '/', //让HMR知道在哪里加载热更新块所必需的
  },
  context: path.resolve( __dirname, 'src' ),
  module: {
    //webpack1.0中可以省略 '-loader'，但是官方说法为了有明确的区分，在webpack2.0中，不能
省略
    rules: [
      {
        test: /\.(js|jsx)$/,
        loader: 'babel-loader',
        include: [
          path.join( common.rootPath, 'src' ), //转换src路径下的代码
        ],
        exclude: /node_modules/, //忽略node_modules路径代码
      },
      {
        test: /\.json$/,
        use: 'json-loader'
      },
      {
        test: /.html$/,
        use: 'html-loader'
      },
      {
```

```
      test: /\.(woff2?|eot|ttf|otf)$/,
      use: {
        loader: 'url-loader',
        options: {
          limit: 10240,
          name: '[name]-[hash:6].[ext]'
        }
      }
    },
    {
      test: /\.(png|jpg|gif|svg)$/,
      use: {
        loader: 'url-loader',
        options: {
          limit: 10240,
          name: '[name]-[hash:6].[ext]'
        }
      }
    },
    {
      test: /\.css$/,
      use: ( handleStyle( extractCss, [
        styleLoaders.style,
        styleLoaders.css,
        styleLoaders.postcss
      ] ))
    },
    {
      test: /\.scss$/,
      use: ( handleStyle( extractScss, [
        styleLoaders.style,
        styleLoaders.css,
        styleLoaders.postcss,
        {
          loader: 'sass-loader'
        }
      ] ))
    },
    {
      test: /\.less$/,
      use: ( handleStyle( extractLess, [
        styleLoaders.style,
        styleLoaders.css,
        styleLoaders.postcss,
        {
          loader: 'less-loader'
        }
      ] ))
    }
  ]
 }
}
if ( isDev ) {
  webpackConfig.devServer = {
```

```
    historyApiFallback:true,
    hot: true,
    contentBase: path.resolve( __dirname, 'dist' ),
    publicPath: '/',
    clientLogLevel: 'none', //日志
    compress: true, //压缩
    port:3000,
    stats: {
      colors: true
    },
    proxy:{
      '/api/*':{
        target: 'http://localhost',
        changeOrigin: true,
        secure: false,
      }
    }
  }
}

module.exports = webpackConfig
```

处理css样式兼容性，开发模式下将css单独出来

此时需要引入css后处理器postcss，然后使用autoprefixer来进行前缀配置，css文件单独出来需要使webpack的插件extractPlugin ，安装依赖，cnpm install autoprefixer-loader extract-text-webpack-plugin postcss-loader --save-dev

编写postcss.config.js

```
//处理css前缀，用来更好的兼容各种浏览器
//在 package.json中 使用 browserslist 字段已经定义好了浏览器适配（官方推荐）
module.exports = {
  plugins: [
    require('autoprefixer')
  ]
}
```

在package.json中添加字段

```
  "browserslist": [
    "> 1%",
    "last 2 versions"
  ],
```

编写webpack.config.js

```
'use strict'

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

```
//使用插件
const HtmlWebpackPlugin = require( 'html-webpack-plugin' )
const ExtractTextPlugin = require( 'extract-text-webpack-plugin' )
const extractCss = new ExtractTextPlugin( 'style/[name]-css-[hash:6].css' )
const extractScss = new ExtractTextPlugin( 'style/[name]-scss-[hash:6].css' )
const extractLess = new ExtractTextPlugin( 'style/[name]-less-[hash:6].css' )

const rootPath = path.resolve( __dirname )
const srcPath = path.join( rootPath, 'src' )

const env = process
  .env
  .NODE_ENV
  .trim()
const isDev = (env === 'development')

const common = {
  rootPath: rootPath,
  srcPath: srcPath,
  dist: path.join( rootPath, 'dist' ),
  indexHtml: path.join( srcPath, 'index.html' ),
  staticDir: path.join( rootPath, 'static' )
}
if ( isDev ) {
  common.entry = [
    'react-hot-loader/patch',
    // activate HMR for React

    'webpack-dev-server/client?http://localhost:3000',
    // bundle the client for webpack-dev-server and connect to the provided endpoint

    'webpack/hot/only-dev-server',
    // bundle the client for hot reloading only- means to only hot reload for
    // successful updates

    path.join( common.srcPath, 'index.js' )
  ]
} else {
  common.entry = {
    app: path.join( common.srcPath, 'index.js' ),
    vendor: [
      'react'
    ]
  }
}
if ( isDev ) {
  common.plugins = [
    new HtmlWebpackPlugin( {
      template: common.indexHtml,
      inject: 'body'
    } ),
    new webpack.HotModuleReplacementPlugin(), // HMR全局启用
    new webpack.NamedModulesPlugin(), // 在HMR更新的浏览器控制台中打印更易读的模块名称
  ]
```

```
} else {
  common.plugins = [
    new webpack.optimize.UglifyJsPlugin(),
    new HtmlWebpackPlugin( {
      template: common.indexHtml,
      inject: 'body'
    } ),
    new webpack.NoEmitOnErrorsPlugin(),
    // stataic目录下静态资源的复制
    new CopyWebpackPlugin( [
      {
        context: common.rootPath,
        from: 'static/*',
        ignore: [
          '*.md'
        ]
      }
    ] ),
    new webpack.optimize.CommonsChunkPlugin( {
      name: 'vendor',
      filename: 'vendor.bundle.js'
    } )
  ]
}

common.plugins.push( new webpack.DefinePlugin( {
  'process.env': {
    NODE_ENV: JSON.stringify(env)
  }
} ) )
const styleLoaders = {
  style: {
    loader: 'style-loader'
  },
  css: {
    loader: 'css-loader',
    options: {
      //将css进行hash编码，保证模块性，保证单独使用，而不会污染全局
      // modules: true
    }
  },
  postcss: {
    //css后处理器，这里主要是为了加载 autoprefixer 用来处理css前缀
    loader: 'postcss-loader'
  }
}
function handleStyle( plugin, list ) {
  //如果不是开发模式，删除数组中的第一个元素，并使用extrat-plugin将样式额外打包
  if ( !isDev ) {
    list.splice( 0, 1 )
    return plugin.extract( list )
  }
  return list
}
```

```
const webpackConfig = {
  entry: common.entry,
  output: {
    filename: 'bundle.js',
    path: common.dist,
    publicPath: '/', //让HMR知道在哪里加载热更新块所必需的
  },
  context: path.resolve( __dirname, 'src' ),
  devtool: isDev
    ? 'cheap-module-eval-source-map'
    : 'cheap-module-source-map',
  module: {
    //webpack1.0中可以省略 '-loader'，但是官方说法为了有明确的区分，在webpack2.0中，不能
省略
    rules: [
      {
        test: /\.(js|jsx)$/,
        enforce: 'pre',
        loader: 'eslint-loader',
        exclude: /node_modules/,
        options: {
          emitWarning: true,
          emitError: true,
          //failOnWarning: false, failOnError: true,
          useEslintrc: false,
          // configFile: path.join(__dirname, "eslint_conf.js")
          configFile: path.join( __dirname, '.eslintrc.json' )
        }
      },
      {
        test: /\.(js|jsx)$/,
        loader: 'babel-loader',
        include: [
          path.join( common.rootPath, 'src' ), //转换src路径下的代码
        ],
        exclude: /node_modules/, //忽略node_modules路径代码
        options:{
          plugins: [
            ['import', [{ libraryName: 'antd', style: 'css' }]],//按需加载antd 样式，有效小包大小
          ]
        }
      },
      {
        test: /\.json$/,
        use: 'json-loader'
      },
      {
        test: /.html$/,
        use: 'html-loader'
      },
      {
        test: /\.(woff2?|eot|ttf|otf)$/,
        use: {
```

```
        loader: 'url-loader',
        options: {
          limit: 10240,
          name: '[name]-[hash:6].[ext]'
        }
      }
    },
    {
      test: /\.(png|jpg|gif|svg)$/,
      use: {
        loader: 'url-loader',
        options: {
          limit: 10240,
          name: '[name]-[hash:6].[ext]'
        }
      }
    },
    {
      test: /\.css$/,
      use: ( handleStyle( extractCss, [
        styleLoaders.style,
        styleLoaders.css,
        styleLoaders.postcss
      ] ))
    },
    {
      test: /\.scss$/,
      use: ( handleStyle( extractScss, [
        styleLoaders.style,
        styleLoaders.css,
        styleLoaders.postcss,
        {
          loader: 'sass-loader'
        }
      ] ))
    },
    {
      test: /\.less$/,
      use: ( handleStyle( extractLess, [
        styleLoaders.style,
        styleLoaders.css,
        styleLoaders.postcss,
        {
          loader: 'less-loader'
        }
      ] ))
    }
  ]
},
resolve: {
  extensions: [
    '.js',
    '.jsx',
    '.json'
```

```
    ],
    alias: {
      Root: path.resolve( __dirname, 'src' ),
      Components: path.resolve( __dirname, 'src/components' ),
      Layouts: path.resolve( __dirname, 'src/layouts' ),
      Routes: path.resolve( __dirname, 'src/routes' ),
    } //为某些路径设置别名
  },
  plugins: (function () {
    //如果是开发模式不将样式文件进行分离。tip:为了实现热加载
    if ( isDev )
      return common.plugins
    common
      .plugins
      .push( extractCss )
    common
      .plugins
      .push( extractLess )
    common
      .plugins
      .push( extractScss )
    //返回组装完成后的plugins
    return common.plugins
  })()
}
if ( isDev ) {
  webpackConfig.devServer = {
    historyApiFallback:true,
    hot: true,
    contentBase: path.resolve( __dirname, 'dist' ),
    publicPath: '/',
    clientLogLevel: 'none', //日志
    compress: true, //压缩
    port:3000,
    stats: {
      colors: true
    },
    proxy:{
      '/api/*':{
        target: 'http://localhost',
        changeOrigin: true,
        secure: false,
      }
    }
  }
}

module.exports = webpackConfig
```

使用eslint和esformatter

这一步的作用其实不是很大，更多的是自定义一个js代码的规范，其实不标准照样可以运行

安装eslint和esformatter的各项依赖，cnpm install --save-dev eslint esformatter es-strip-semico

ons eslint-config-airbnb eslint-loader eslint-plugin-import eslint-plugin-jsx-a11y eslint-plugin react

根目录新建.esformatter,这里的代码太长且不重要，这里不贴了，github地址,请自行查看

根目录新建 .eslintrc.json 同上，github地址

编写webpack.config.js

在module.rules里面新增

```
{
        test: /\.(js|jsx)$/,
        enforce: 'pre',
        loader: 'eslint-loader',
        exclude: /node_modules/,
        options: {
          emitWarning: true,
          emitError: true,
          //failOnWarning: false, failOnError: true,
          useEslintrc: false,
          // configFile: path.join(__dirname, "eslint_conf.js")
          configFile: path.join( __dirname, '.eslintrc.json' )
        }
      },
```

这样会强制在运行前检查js代码，非常严格，一旦有错将不能运行，如果觉得没必要，可以去掉这一代码

其他细节

其他小的细节，是一看就懂的，这里就不再赘述步骤。

项目的github地址

聊聊其中的曲折和收获

其实本来这个项目开始是使用webpack1.x搭建的，结果搭建好之后，才知道，喵蛋的，2.x都出来了于是果断放弃之前的，将项目速度改进到2.x版本。

很多东西都是根据其他的项目而来，但是那些版本都太老旧了，很多的依赖都是我看了名字之后，转去github或者npm查看新版本的用法，这段是最费时间，最费眼睛的了（因为我的英文实在是差劲QQ，要不是chrome的翻译插件，我估计我已经崩了），对于我这种特别崇尚最新的人来说，用老旧东西，是我非常非常不能忍受的，所以我用的依赖基本上都是最新的，其中有些除了github上面，其地方基本都么有相关资料。

说实话，做出来之后，还能很好的运行，我真的是长长的出了一口大气，很舒心的感觉，终于特么的走通一回了，好气啊~~~

最大的收获，就是这一份走通的感觉了吧，个人前端后端都在弄，一直依赖都是两边奔波却基本上都怎么摸透过。感觉暂时前端终于能稍微放一下，继续搞后端，目标分布式和并发，走起！

写完这篇都特么快五点了，必须得睡觉了。顺便也请各位纠纠错，其实写的时候也没多仔细，难免会所遗漏与错误。

最后

最新最新的react脚手架react-starter，算是提供个思路和参考吧，有用请star~ 也请大佬们能够帮我改修改 smile