链滴

react-server-side-render 最新学习与实践

作者：wuhongxu
原文链接：https://ld246.com/article/1521615160212
来源网站：链滴

写在前面

server side render(ssr)服务端渲染，亦即同构应用。主要有利于seo和首屏渲染，这是一篇比较新可运行的结构设计，基于比较新的react v16、react-router v5的同构设计。结合了 redux(Flux数据实现)。

项目地址:react-ssr-starter

不务正业（搞前端）差不多已经一年了，学习react也是一年前的事情了，但是一直以来对于react 服端渲染，兴趣缺缺，毕竟n久以前就是服务端渲染（通过模板引擎），随着ng、react、vue等框架的起，前端渲染越来越火热，没想到发展发展，又绕回了服务端渲染，而且居然还多加了一个node中层，把本来简单的结构一拆再拆，很多时候都是徒然增加复杂度。其实说起来可笑，我觉得服务端渲在国内的火热百度绝对占有相当大一部分因素（搜索引擎技术万年不更新，广告倒是多了一大堆,233。

好了，吐槽一段也该进入正题了。我等小菜没法改变现状，也只能适应现状了。在阅读了一大堆乱七糟的ssr相关博文之后，终于找到Server Side Rendering in React/Redux (JS)和React16+Redux+Ruter4+Koa+Webpack服务器端渲染（按需加载，热更新）让我暂时明白了其中一部分原理并开始进自己的构建。

ssr构想

传统的react为什么不利于seo？其实说白了，就是因为在路由请求到页面时，页面是一个没有任何数的html,它的数据必须要运行渲染dom/更改<head/>的js代码，而搜索引擎并不会执行这段代码，所搜索引擎拿不到任何东西(google已经提出了解决方案，百度仍然沉浸在十年前#雾)。

而ssr的结构无外乎就是把这一段需要运行的js放到node服务端去运行，然后直接向浏览器端输出htm。这样传统引擎就能够去爬取内容了。

为了实现ssr。我觉得需要考虑的问题应该主要集中在这几点上：

● 需要写那些方面的代码？

答：需要写两个方面的代码，一个是以前传统的客户端代码，一个是node服务端代码。如果你已经完了一个客户端渲染的代码，那么指需要小小修改一些东西(如果代码组织结构较好，甚至只需要改动个方法名，也就是说后面提到的ReactDOM.render改成ReactDOM.hydrate)，然后加入服务端代码基本上就能够完成服务端渲染啦。迁移也是非常方便的。

● node中间层干什么？怎么干？

答：node中间层主要任务是 获取用户请求，根据路由(更多是为了应对首屏渲染)准备初始数据（例：去api端请求数据），把初始数据填充到组件中，把整个填充好的组件输出到响应体中。这方面比好的实现是结合react-router匹配路由，结合redux的store填充数据。刚好与这两个库的思想吻合。尽量注意，其实在我以前的想法里，服务端渲染应该是每个单页/单页的部分组成组件每次页面跳转由服务端呈现的。但是在查看了这些博文，以及研究了相关代码之后，才了解，服务端渲染仅仅只针首屏渲染，首屏渲染完成后，后续的页面跳转，api请求等还是由前端自己管理，也就是说，其实nod中间层**只管刷新**，当然这是因为我们用react-router以及redux结合所希望达成的最好的效果，实际依靠react官方api，我们是可以完全实现所有页面都由服务端渲染完成，由客户端去请求的，但是那其实体验并不好（这也编程了纯服务端渲染，也就是说模板引擎干的事，显然，这不是我们想要的）服务端渲染主要解决的应该是首屏白屏和seo的问题。

● 浏览器端干什么？怎么干？

答：浏览器端主要是显示服务端渲染过来的Html（当然，这不用我们管，浏览器干的事）。我们主要要根据服务端提供的初始state和渲染的根节点，把每一层渲染的实际dom用react组件对应上（因为在html变成了已经渲染成功的样子，但是客户端还什么都没做，客户端的react表示一脸懵逼，还不道自己干了什么）。

而在react 16 以前react-dom只提供了render方法，去对应根结点，这个方法会删除掉原来根节点本已经由服务端渲染成功的子dom，然后根据初始状态重新渲染，也就出现了渲染两次的问题（服务端染一次，把dom结构加载到根节点中，客户端拿到html页面，再根据初始状态再渲染一次dom结构，当然，这其实某种程度已经满足了我们的需求，搜索引擎爬到了初始页面，不执行js，所有dom结还在，能够获取需要的信息，而用户看到页面中，因为react-dom的render方法执行效率也还是很可的，所以也没有什么问题（一般来说其实会有一点闪屏，因为dom擦除和重建）。

但是对于复杂的网页，或者追求用户体验的我们来说，这是真的不能忍，react 16 以前，大佬们使用种方式来避免第二次渲染，但是在react 16之后，react官方提供了一个新的方法来搞定这个问题啦，就是ReactDom.hydrate方法。这个方法和render使用是一样的，但是它不会擦除和重建dom，仅仅是把dom结构和我们的虚拟dom结构对应上，简直是大大的方便啊。所以客户端会该ReactDom.hydrte方法代替ReactDom.render方法，其他写法与以前的客户端渲染一样哦。

● 开发环境下怎么搞？

答：开发环境下，大致分为两种思路（其实也差不多）

第一种，使用Webpack的devServer做为开发服务器，当然这就无法完全重现服务端渲染的真实情况但是问题不大，因为本来差别也不大，只有一个首屏问题。

第二种，完全模拟服务端渲染，使用koa/express自行封装，使用babel的register方法添加node对于mport的支持（这种，去掉开发相关配置，其实完全可以用来直接做服务器）。

● 生成环境下怎么搞？

答：生产环境其实主要也就是两个方面，一个是客户端代码的编译。另一个是服务端代码，这个可以择两种，分别是使用webpack进行编译使其支持import/es6/es7/jsx等代码以及使用babel.register其支持import/es6/es7/jsx。各有优劣，我的选择是前者，也没什么特别的原因，任性！

- 架构的基本思想

答：其实基本思想就是，因为服务端需要渲染一部分组件（用于初始化），也就是说服务端需要包含部分react组件，而客户端也（当然）需要包含所有的组件。那么这一部分组件要想办法重用，这方其实没那么复杂，说白了就是服务端能够在目录中抽取(import/require)到所需的组件，没什么特别，主要是为了服务端代码，最大程度实现重用。另一方面是路由的匹配（服务端需要相关路由匹配以染对应组件），其实koa/express等有路由匹配相关的方法，但是同样是为了最大程度重用，我们要办法能够统一匹配路由，这方面我推荐采用react-router搭配react-router-config食用。

服务端没有history，所以需要模拟一个hisotry,正好，react-router提供了staticRouter静态路由可以拟，为了更好的食用，我使用history.js的memeryHistory，这样，服务端和客户端又能够重用路由息啦。

又想一想，还有什么能重用？对啦，是数据处理，包括数据请求，因为服务端需要初始化一部分的数啊。我们结合redux，也就成了相关的action和reducer，这一部分也能够重用。在服务端和客户端都要创建store，所以把创建store的代码提供出来给大家食用，就又重用了一部分代码啦。

经过了以上代码的重用，然后发现，服务端除了监听request和渲染html,其他什么都不用做，因为我在写客户端代码的时候，就无形中搞定了服务端渲染。所以基本上，来说，脚手架一旦搭建完成，用还是像以前那样开开心心的写客户端代码，而不用管服务端代码。想想还有点小激动呢～

核心代码

通过以上问题的抛出，其实我们心中已经能够有大体的思路，只是在实现上，我们就不得不各种找ap，各种想办法去对应上这些问题了，这是一个枯燥无聊的过程，如果你实在没有继续看下去的欲望，以直接食用我的脚手架react-ssr-starter，开箱即用哦。

代码分离方案

食用react-loadable组件，也是一个开箱即用的库，结合webpack的import()方法，分分钟实现代码离。示例

```
import React from 'react'
import Loadable from 'react-loadable'
import { homeInit } from './actions'

const Loading = () => {
  return <div>Loading...</div>
}

const routesConfig = [{
  path: '/',
  component: Loadable({
    loader: () => import(/* webpackChunkName: 'AppLayout'*/'./pages/AppLayout'),
    loading: Loading,
  }),
  routes: [{
    path: '/',
    exact:true,
    component: Loadable({
      loader: () => import(/* webpackChunkName: 'Home' */'./pages/Home'),
      loading: Loading,
    }),
```

```
  }, {
    path: '/user',
    component: Loadable({
      loader: () => import(/* webpackChunkName: 'User'*/'./pages/User'),
      loading: Loading,
    })
  }]
}]

export default routesConfig
```

路由重用

我们使用react-router+react-router-config的方案实现路由重用，首先时需要导入一个routesConfi。其实就是上面的代码。接下来我们需要能够在前后端都能加载这个routesConfig。那么就要分别由后端代码去读取,客户端需要解析出真正的Route节点，而服务端只需要匹配url即可

服务端代码:

```
import { matchRoutes } from 'react-router-config'
import Routes from './Routes'

let branch = matchRoutes(Routes, ctx.req.url)

let promises = branch.map(({ route }) => {
    return route.init ? (route.init(store)) : Promise.resolve(null)
  }).map(promise => {
    if (promise) {
      return new Promise((resolve) => {
        promise.then(resolve).catch(resolve)
      })
    }
  })
await Promise.all(promises).catch(err => console.error(err))
```

客户端代码:

```
import { hydrate, render, unmountComponentAtNode } from 'react-dom'
import { ConnectedRouter } from 'react-router-redux'
import { renderRoutes } from 'react-router-config'
const renderApp = (routes) => {
  const renderMethod = process.env.NODE_ENV === 'development' ? render : hydrate
  renderMethod(
    <Provider store={store}>
      <ConnectedRouter history={history}>
        {renderRoutes(routes)}
          </ConnectedRouter>
    </Provider>, Root)
}
renderApp(Routes)
```

store的重用设计

store的重用设计非常简单，说白了就是获取初始状态，有就加进去，没有就直接初始化一个store（务端没有，客户端需要读取服务端的初始状态，所有有）。

整个store初始化方法

```
import { createStore, applyMiddleware, compose } from 'redux'
import thunkMiddleware from 'redux-thunk'
import createHistory from 'history/createMemoryHistory'
import { routerMiddleware } from 'react-router-redux'
import rootReducer from '../reducers'

const routerReducers = routerMiddleware(createHistory())
const composeEnhancers = process.env.NODE_ENV == 'development' ? window.__REDUX_DE
TOOLS_EXTENSION_COMPOSE__ : compose
const middleware = [thunkMiddleware, routerReducers]
let configureStore = (initialState) => createStore(rootReducer, initialState, composeEnhancers
applyMiddleware(...middleware)))
export default configureStore
```

客户端使用(注意我们默认node服务端渲染的初始状态挂载到window.__
NITIAL_STATE__上面)：

```
const initialState = window && window.\_\_INITIAL_STATE\_\_
import { Provider } from 'react-redux'
import configuraStore from './store/configureStore'
let store = configuraStore(initialState)

//...
<Provider store={store}>
//...
```

服务端使用：

```
import configureStore from './store/configureStore'
let store = configureStore()
```

服务端渲染代码

这才是重中之重，服务端渲染代码，主要使用的时ReactDom/Server.renderToString方法。这样可把组件转换成string，接下来我们服务端需要做的工作就是继续拼接，把这个node装在到根节点下面然后把整个页面给渲染出去，这里我还使用了一个react-helmet库，这是用来做<head/>的元素，如 mata,title等字段的。我们在服务端要把这些字段进行替换。另外最重要的是，别忘了把初始状态载在html结点中。

```
import React from 'react'
import { renderToString } from 'react-dom/server'
import { StaticRouter, matchPath } from 'react-router-dom'
import { Provider } from 'react-redux'
import { renderRoutes } from 'react-router-config'
import { Helmet } from 'react-helmet'

import { getBundles } from 'react-loadable/webpack'
import Loadable from 'react-loadable'
```

```
const createTags = (modules, stats) => {
  let bundles = getBundles(stats, modules)
  let scriptfiles = bundles.filter(bundle => bundle.file.endsWith('.js'))
  let stylefiles = bundles.filter(bundle => bundle.file.endsWith('.css'))
  let scripts = scriptfiles.map(script => `<script src="/${script.file}"></script>`).join('\n')
  let styles = stylefiles.map(style => `<link href="/${style.file}" rel="stylesheet"/>`).join('\n')
  return { scripts, styles }
}

const prepHtml = (data, { html, head, rootString, scripts, styles, initState }) => {
  data = data.replace('<html', `<html ${html}`)
  data = data.replace('</head>', `${head} \n ${styles}</head>`)
  data = data.replace('<div id="root"></div>', `<div id="root">${rootString}</div>`)
  data = data.replace('<body>', `<body> \n <script>window.__INITIAL_STATE__ ={JSON.strin
ify(initState)}</script>`)
  data = data.replace('</body>', `${scripts}</body>`)
  return data
}
export const make = ({ ctx, store, context, template, Routes, stats }) => {
  let modules = []

  const rootString = renderToString(
    <Loadable.Capture report={moduleName => modules.push(moduleName)}>
      <Provider store={store}>
        <StaticRouter location={ctx.req.url} context={context}>
          {renderRoutes(Routes)}
        </StaticRouter>
      </Provider>
    </Loadable.Capture>
  )
  const initState = store.getState()
  const { scripts, styles } = createTags(modules, stats)

  const helmet = Helmet.renderStatic()
  return prepHtml(template, {
    html: helmet.htmlAttributes.toString(),
    head: helmet.title.toString() + helmet.meta.toString() + helmet.link.toString(),
    rootString,
    scripts,
    styles,
    initState
  })
}

export const getMatch = (routesArray, url) => {
  return routesArray.some(router => matchPath(url, {
    path: router.path,
    exact: router.exact,
  }))
}

import Routes from './Routes'
import Loadable from 'react-loadable'
import configureStore from './store/configureStore'
```

```
import { matchRoutes } from 'react-router-config'
import { getMatch, make } from './helpers/renderer'
import stats from '../dist/react-loadable.json'
import Koa from 'koa'
const server = new Koa()
const port = process.env.port || 3000,
  staticCache = require('koa-static-cache'),
  cors = require('koa2-cors')

var fs = require('fs')
var path = require('path')

server.use(cors())

const clientRouter = async (ctx, next) => {
  let html = fs.readFileSync(path.join(path.resolve(process.cwd(), 'dist'), 'index.html'), 'utf-8')
  let store = configureStore()

  let branch = matchRoutes(Routes, ctx.req.url)
  let promises = branch.map(({ route }) => {
    return route.init ? (route.init(store)) : Promise.resolve(null)
  }).map(promise => {
    if (promise) {
      return new Promise((resolve) => {
        promise.then(resolve).catch(resolve)
      })
    }
  })
  await Promise.all(promises).catch(err => console.error(err))

  let isMatch = getMatch(Routes, ctx.req.url)
  const context = {}
  if (isMatch) {
    let renderedHtml = await make({
      ctx,
      store,
      context,
      template: html,
      Routes,
      stats,
    })
    if (context.url) {
      ctx.status = 301
      ctx.redirect(context.url)
    } else {
      ctx.body = renderedHtml
    }
  } else {
    ctx.status = 404
    ctx.body = '未找到该页面'
  }
  await next()
}
```

```
server.use(clientRouter)
server.use(staticCache(path.resolve(process.cwd(), 'dist'), {
  maxAge: 365 * 24 * 60 * 60,
  gzip: true
}))

console.log(`\n==> :earth_americas:  Listening on port ${port}. Open up http://localhost:${por
}/ in your browser.\n`)

Loadable.preloadAll().then(() => {
  server.listen(port)
})
```

上面的代码，就基本把服务端的代码给写完了，没有想象中的那么长，但是也不算端。其中要注意几点。

- 我需要读取到dist目录的index.html目录，这里的目录读取方式有问题。但是大致意思差不多
- 每个路由都由初始方法，我默认挂载到了route的init字段中，会把store传入进去，可以执行store.dspatch方法来改变数据。

客户端代码

客户端代码跟以前的客户端渲染差不多，只是需要根据环境不同切换render方法或者hydrate方法

```
import React from 'react'
import { hydrate, render, unmountComponentAtNode } from 'react-dom'
import { ConnectedRouter } from 'react-router-redux'
import Loadable from 'react-loadable'
import { renderRoutes } from 'react-router-config'
import Routes from './Routes'
const initialState = window && window.__INITIAL_STATE__
import { Provider } from 'react-redux'
import configuraStore from './store/configureStore'
import createHistory from 'history/createBrowserHistory'
const history = createHistory()
let store = configuraStore(initialState)
const Root = document.getElementById('root')
const renderApp = (routes) => {
  const renderMethod = process.env.NODE_ENV === 'development' ? render : hydrate
  renderMethod(
    <Provider store={store}>
      <ConnectedRouter history={history}>
        {renderRoutes(routes)}
      </ConnectedRouter>
    </Provider>, Root)
}

Loadable.preloadReady().then(renderApp.bind(this, Routes))

if (process.env.NODE_ENV === 'development') {
  if (module.hot) {
    module.hot.accept('./reducers/index.js', () => {
```

```
    let newReducer = require('./reducers/index.js').default
    store.replaceReducer(newReducer)
  })
  module.hot.accept('./Routes.jsx', () => {
    unmountComponentAtNode(Root)
    var r = require('./Routes').default
    renderApp(r)
  })
 }
}
```

开发环境与生产环境

开发环境下还是才用客户端渲染的方式，所以与平常的客户端渲染配置没多大区别，也不再赘述。

主要讲讲生产环境，生产环境下，我们需要变量两个包，分别时server和client。client包中配置，一要加入ReactLoadablePlugin，以提供给服务端读取组件代码。服务端打包，一定要把target设置为node。就这亮点，配置为：

webpack.config.common.js

```
'use strict'
const path = require('path')
module.exports =  {
 output:{
  filename:'[name].[hash].js',
  path:path.resolve(__dirname,'dist'),
  publicPath:'/',
  chunkFilename:'[name].chunk.[hash:8].js',
 },
 context:path.resolve(__dirname,'src'),
 resolve: {
  extensions: ['.js', '.jsx','.json'],
  modules: [path.resolve(__dirname, 'src'), 'node_modules']
 },
}
```

webpack.config.prod.js:

```
const path = require('path')
const webpack = require('webpack')
const CleanWebpackPlugin = require('clean-webpack-plugin')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const ManifestPlugin = require('webpack-manifest-plugin')
const { ReactLoadablePlugin } = require('react-loadable/webpack')
const CopyWebpackPlugin = require('copy-webpack-plugin')
const ExtractTextPlugin = require('extract-text-webpack-plugin')
const common = require('./webpack.config.common')
const merge = require('webpack-merge')

module.exports = merge(common, {
 entry: {
  client: 'client.jsx',
```

```
},
module: {
  rules: [{
    test: /\.jsx?$/,
    exclude: /node_modules/,
    include: path.resolve(__dirname, 'src'),
    use: {
      loader: 'babel-loader',
      options: {
        cacheDirectory: true
      }
    }
  }, {
    test: /\.(css|scss|less)$/,
    exclude: /node_modules/,
    include: path.resolve(__dirname, 'src'),
    use: ExtractTextPlugin.extract({
      fallback: 'style-loader',//style-loader 将css插入到页面的style标签
      use: [{
        loader: 'css-loader',//css-loader 是处理css文件中的url(),require()等
        options: {
          sourceMap: true,
        }
      }, {
        loader: 'postcss-loader',
        options: {
          sourceMap: true,
        }
      }, {
        loader: 'sass-loader',
        options: {
          sourceMap: true,
        }
      }, {
        loader: 'less-loader',
        options: {
          sourceMap: true,
        }
      }]
    }),
  }, {
    test: /\.(svg|woff2?|ttf|eot|jpe?g|png|gif)(\?.*)?$/i,
    exclude: /node_modules/,
    use: {
      loader: 'url-loader',
      options: {
        limit: 1024,
        name: 'img/[sha512:hash:base64:7].[ext]'
      }
    }
  }],
},
plugins: [
  new ManifestPlugin(),
```

```
    new webpack.NoEmitOnErrorsPlugin(),
    new ExtractTextPlugin({
      filename: 'css/style.[hash].css',
      allChunks: true,
    }),
    new CopyWebpackPlugin([{ from: 'assets/z.png', to: 'favicon.ico' }]),
    new CleanWebpackPlugin(['./dist']),
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV)
    }),
    new webpack.optimize.OccurrenceOrderPlugin(),
    new HtmlWebpackPlugin({
      title: 'go-store-client',
      filename: 'index.html',
      template: './index.prod.html',
    }),
    new webpack.optimize.CommonsChunkPlugin({
      name: ['vendors', 'manifest'],
      minChunks: 2
    }),
    new ReactLoadablePlugin({
      filename: path.join('./dist/react-loadable.json'),
    }),
  ],
  externals: {
    'react': 'React',
    'react-dom': 'ReactDOM',
  }
})
```

webpack.config.server.js:

```
const path = require('path')
const webpack = require('webpack')
const CleanWebpackPlugin = require('clean-webpack-plugin')
const webpackNodeExternals = require('webpack-node-externals')
const ExtractTextPlugin = require('extract-text-webpack-plugin')
module.exports = {
  entry: './src/server.js',
  output: {
    filename: 'server.build.js',
    path: path.resolve(__dirname, 'build'),
  },
  resolve: {
    extensions: ['.js', '.jsx','.json'],
    modules: [path.resolve(__dirname, 'src'), 'node_modules']
  },
  target: 'node',
  externals: [webpackNodeExternals()],
  module: {
    rules: [{
      test: /\.jsx?$/,
      exclude: /node_modules/,
      include: path.resolve(__dirname, 'src'),
```

```
      use: {
        loader: 'babel-loader',
        options: {
          cacheDirectory: true
        }
      }
    }, {
      test: /\.(css|scss|less)$/,
      exclude: /node_modules/,
      include: path.resolve(__dirname, 'src'),
      use: ExtractTextPlugin.extract({
        fallback: 'style-loader',//style-loader 将css插入到页面的style标签
        use: [{
          loader: 'css-loader',//css-loader 是处理css文件中的url(),require()等
          options: {
            sourceMap: true,
          }
        }, {
          loader: 'postcss-loader',
          options: {
            sourceMap: true,
          }
        }, {
          loader: 'sass-loader',
          options: {
            sourceMap: true,
          }
        }, {
          loader: 'less-loader',
          options: {
            sourceMap: true,
          }
        }]
      }),
    }],
  },
  plugins: [
    new webpack.NoEmitOnErrorsPlugin(),
    new CleanWebpackPlugin(['./build']),
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV)
    }),
    new ExtractTextPlugin({
      filename: 'css/style.[hash].css',
      allChunks: true,
    }),
    new webpack.optimize.OccurrenceOrderPlugin(),
  ],
}
```

写在最后

我的最新代码提交于github，项目地址为react-ssr-starter，喜欢的可以拿去直接用～