

React18 SSR + Next.js

刘军 liujun

目录

content



1 邂逅React18 + SSR

2 Node Server 搭建

3 React18 + SSR 搭建

4 React18 SSR+Hydration

5 React18 SSR+Router

6 React18 SSR+Redux

邂逅React18 + SSR

- React和Vue一样，除了支持开发SPA应用之外，其实也是支持开发SSR应用的。
- 在React中创建SSR应用，需要调用 `ReactDOM.hydrateRoot` 函数，而不是`ReactDOM.createRoot`
 - `createRoot`：创建一个Root，接着调用其 `render` 函数将App直接过载到页面上
 - `hydrateRoot`：创建水合Root，是在激活的模式下渲染 App
- 服务端可用 `ReactDOM.renderToString` 来进行渲染。

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "../index.jsx";

// 在客户端激活应用，是应用可以进行交互（这个过程称为 hydration，水合）
// root 在server/index.js的首页中
ReactDOM.hydrateRoot(document.getElementById("root"), <App />);
```

Node Server 搭建

■ 需安装的依赖项:

- ❑ npm i express
- ❑ npm i -D nodemon
- ❑ npm i -D webpack webpack-cli webpack-node-externals

```
const path = require("path");

module.exports = {
  target: "node", // 打包目标为node服务
  mode: "development",
  entry: "./src/server/index.js", // 相对于根目录（执行nodemon命令所在的目录）
  output: {
    filename: "server_bundle.js",
    path: path.resolve(__dirname, "../build/server"), // 相对于当前目录
  },
};
```

```
01-node-server
✓ build/server
  JS server_bundle.js
✓ config
  JS server.config.js
> node_modules
✓ src/server
  JS index.js
package-lock.json
package.json
README.md
```

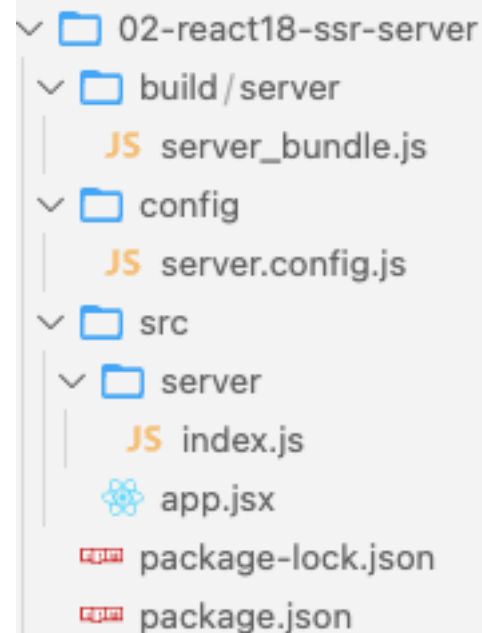
React18 + SSR 搭建

■ 需安装的依赖项:

- ❑ npm i express
- ❑ npm i react react-dom
- ❑ npm i -D nodemon
- ❑ npm i -D webpack webpack-cli webpack-node-externals webpack-merge
- ❑ npm i -D babel-loader @babel/preset-react @babel/preset-env

```
import ReactDOM from "react-dom/server";
import React from "react";
import App from "../index";
const express = require("express");

const app = express();
app.get("/", (req, res) => {
  const AppContent = ReactDOM.renderToString(<App />);
  res.send( ...
});
app.listen(3000, () => {
  console.log("ready 3000");
});
```



```
02-react18-ssr-server
├── build/server
│   └── JS server_bundle.js
├── config
│   └── JS server.config.js
├── src
│   └── server
│       ├── JS index.js
│       ├── app.jsx
│       ├── package-lock.json
│       └── package.json
```

React18 SSR+Hydration

■ 安装的依赖项同前面一样

```
server.get("/*", (req, res) => {  
  const appContent = ReactDOM.renderToString(<App />);  
  
  res.send(`  
    <!DOCTYPE html>  
    <html lang="en">  
      <head>  
        <meta charset="UTF-8" />  
        <meta http-equiv="X-UA-Compatible" content="IE=edge" />  
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
        <title>React18 + SSR</title>  
      </head>  
      <body>  
        <h1>React18 Server Side Render</h1>  
        <div id="root">${appContent}</div>  
        <script src="/client/client_bundle.js"></script>  
      </body>  
    </html>  
  `);  
});
```

```
03-react18-ssr-hydration  
├── build  
│   ├── client  
│   │   └── client_bundle.js  
│   └── server  
│       └── server_bundle.js  
├── config  
│   ├── client.config.js  
│   └── server.config.js  
├── src  
│   ├── client  
│   │   └── index.js  
│   └── server  
│       ├── index.js  
│       └── app.jsx  
├── package-lock.json  
└── package.json
```

React18 SSR+Router

■ 安装的依赖项同前面一样，还需增加

□ npm i react-router-dom (默认会自动安装 react-router)

```
// root 需在server/index.js中添加
ReactDOM.hydrateRoot(
  document.getElementById("root"),
  <BrowserRouter>
    <App />
  </BrowserRouter>
);

server.get("/*", (req, res) => {
  const appContent = ReactDOM.renderToString(
    <StaticRouter location={req.url}>
      <App />
    </StaticRouter>
  );
});
```

```
04-react18-ssr-router
├── build
│   ├── client
│   │   └── JS client_bundle.js
│   └── server
│       └── JS server_bundle.js
├── config
│   ├── JS base.config.js
│   ├── JS client.config.js
│   └── JS server.config.js
├── node_modules
├── src
│   ├── client
│   │   └── JS index.js
│   ├── pages
│   │   ├── about.jsx
│   │   └── home.jsx
│   ├── router
│   │   └── JS index.js
│   └── server
│       ├── JS index.js
│       └── app.jsx
├── package-lock.json
└── package.json
```

React18 SSR+Redux

■ 安装的依赖项同前面一样，还需增加

□ npm i react-redux @reduxjs/toolkit

□ npm i axios

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "../modules/counter";
import homeReducer from "../modules/home";
const store = configureStore({
  reducer: {
    counter: counterReducer,
    home: homeReducer,
  },
});
export default store;

<Provider store={store}>
  <Routes>
    <Route path="/" element={<Home />}></Route>
    <Route path="/about" element={<About />}></Route>
  </Routes>
</Provider>
```

```
05-react18-ssr-redux
├── build
│   ├── client
│   │   └── client_bundle.js
│   └── server
│       └── server_bundle.js
├── config
│   ├── base.config.js
│   ├── client.config.js
│   └── server.config.js
├── node_modules
├── src
│   ├── client
│   │   └── index.js
│   ├── pages
│   │   ├── about.jsx
│   │   └── home.jsx
│   ├── router
│   │   └── index.js
│   ├── server
│   │   └── index.js
│   ├── store
│   │   ├── home.js
│   │   ├── index.js
│   └── app.jsx
```


认识Redux Toolkit (RTK)

■ Redux Toolkit 是官方推荐的编写 Redux 逻辑的方法。

- ❑ 以前在使用redux时，通常会将redux代码拆分在多个模块中，每个模块需包含多个文件，如：constants、action、reducer、index 等
- ❑ 然后使用combineReducers对多个模块合并，这种代码组织方式过于繁琐和麻烦，导致代码量过多，也不利于后期管理
- ❑ Redux Toolkit 就是为了解决这种编码方式而诞生。
- ❑ 并且以前的 createStore 方式已标为过时，而 Redux Toolkit 已成为官方推荐；

■ 安装Redux Toolkit:

```
npm install @reduxjs/toolkit react-redux
```

■ Redux Toolkit的核心API主要是如下几个:

- ❑ **configureStore**: 包装createStore以提供简化的配置选项和良好的默认值。
 - ✓ 可自动组合 slice reducer
 - ✓ 可添加其它 Redux 中间件，redux-thunk默认包含，
 - ✓ 默认启用 Redux DevTools Extension
- ❑ **createSlice**: 接受切片名称、初始状态值和reducer函数的对象，并自动生成切片reducer，并带有相应的actions。
- ❑ **createAsyncThunk**: 接受一个动作类型字符串和一个返回承诺的函数，并生成一个pending/fulfilled/rejected基于该承诺分派动作类型的thunk。简单理解就是专门用来创建异步Action。

创建counter模块的reducer

■ 我们先创建counter模块的reducer: 通过createSlice创建一个slice。

■ createSlice主要包含如下几个参数:

□ name: 用来标记slice的名词

✓ redux-devtool中会显示对应的名词;

□ initialState: 第一次初始化时的值;

□ reducers: 相当于之前的reducer函数

✓ 对象类型, 并且可以添加很多的函数;

✓ 函数类似于redux原来reducer中的一个case语句;

✓ 函数的参数:

➤ 参数一: state

➤ 参数二: action

□ createSlice 返回值是一个对象

✓ 对象包含所有的 actions 和 reducer;

```
import { createSlice } from '@reduxjs/toolkit'

const counterSlice = createSlice({
  name: "counter",
  initialState: {
    counter: 0
  },
  reducers: {
    addNumber(state, { payload }) {
      state.counter += payload
    },
    subNumber(state, { payload }) {
      state.counter -= payload
    }
  }
})

export const { addNumber, subNumber } = counterSlice.actions

export default counterSlice.reducer
```

store的创建

■ `configureStore`用于创建store对象，常见参数如下：

- ❑ `reducer`，将slice中的reducer可以组成一个对象传入此处；
- ❑ `middleware`：可以使用参数，传入其他的中间件（自行了解）；
- ❑ `devTools`：是否配置devTools工具，默认为true；

```
import { configureStore } from "@reduxjs/toolkit"
import counterReducer from "./counter"
import homeReducer from "./home"

const store = configureStore({
  reducer: {
    counter: counterReducer,
    home: homeReducer
  }
})

export default store
```

■ 在app.js中将store接入应用:

- **Provider**, 内容提供者, 给所有的子或孙子组件提供store对象;
- **store**: 使用configureStore创建的store对象;

```
import { Provider } from 'react-redux'
import store from './store'
class App extends Component {
  render () {
    return (
      <Provider store={store}>
        {this.props.children}
      </Provider>
    )
  }
}
export default App
```

开始使用store

■ 在函数式组件中可以使用 react-redux 提供的 Hooks API 连接、操作 store。

□ **useSelector** 允许你使用 selector 函数从 store 中获取数据 (root state) 。

□ **useDispatch** 返回 redux store 的 dispatch 引用。你可以使用它来 dispatch actions。

□ **useStore** 返回一个 store 引用, 和 Provider 组件引用完全一致。

```
import { Component } from 'react'
import { useSelector } from 'react-redux'

export const CounterComponent = () => {
  const counter = useSelector(state => state.counter)
  return <View>{counter}</View>
}
```

```
import { Component } from 'react'
import { useStore } from 'react-redux'

export const CounterComponent = ({ value }) => {
  const store = useStore()
  return <div>{store.getState()}</div>
}
```

```
import { Component } from 'react'
import { useDispatch } from 'react-redux'

export const CounterComponent = ({ value }) => {
  const dispatch = useDispatch()

  return (
    <View>
      <Text>{value}</Text>
      <Button onClick={() => dispatch({ type: 'increment-counter' })}>
        Increment counter
      </Button>
    </View>
  )
}
```

Redux Toolkit异步Action操作

- 在之前的开发中，我们通过redux-thunk中间件让dispatch中可以进行异步操作。
- Redux Toolkit默认已经给我们继承了Thunk相关的功能：createAsyncThunk

```
export const getHomeMultidataAction = createAsyncThunk("home/multidata", async (payload, extraInfo) => {  
  const res = await axios.get("http://123.207.32.32:8000/home/multidata")  
  return res.data.data  
})
```

- 当createAsyncThunk创建出来的action被dispatch时，会存在三种状态：

- ❑ pending：action被发出，但是还没有最终的结果；
- ❑ fulfilled：获取到最终的结果（有返回值的结果）；
- ❑ rejected：执行过程中有错误或者抛出了异常；

- 我们可以在createSlice的extraReducer中监听这些结果：见右图

- ❑ extraReducers也支持函数，该函数会接收一个builder参数

✓ builder.addCase(getHomeMultidataAction.fulfilled, (state, action) => {})

```
extraReducers: {  
  [getHomeMultidataAction.pending](state, action) {  
    console.log("getHomeMultidataAction pending", action)  
  },  
  [getHomeMultidataAction.fulfilled](state, { payload }) {  
    console.log("getHomeMultidataAction fulfilled", payload)  
    state.banners = payload.banner.list  
    state.recommends = payload.recommend.list  
  },  
  [getHomeMultidataAction.rejected](state, action) {  
  
  }  
}
```