Redux的使用详解（二）

王红元 coderwhy

目录
content

认识Redux Toolkit

- **Redux Toolkit 是官方推荐的编写 Redux 逻辑的方法。**
  - 在前面我们学习Redux的时候应该已经发现，redux的编写逻辑过于的繁琐和麻烦。
  - 并且代码通常分拆在多个文件中（虽然也可以放到一个文件管理，但是代码量过多，不利于管理）；
  - Redux Toolkit包旨在成为编写Redux逻辑的标准方式，从而解决上面提到的问题；
  - 在很多地方为了称呼方便，也将之称为“RTK”；
- **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。它可以自动组合你的 slice reducer，添加你提供的任何 Redux 中间件，redux-thunk默认包含，并启用 Redux DevTools Extension。
  - createSlice：接受reducer函数的对象、切片名称和初始状态值，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

重构代码 – 创建counter的reducer

- **我们先对counter的reducer进行重构： 通过createSlice创建一个slice。**
- **createSlice主要包含如下几个参数：**
- name：用户标记slice的名词
  - 在之后的redux-devtool中会显示对应的名词；
- initialState：初始化值
  - 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：调用这个action时，传递的action参数；
- **createSlice返回值是一个对象，包含所有的actions；**

```
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
```

重构代码 – 创建home的reducer

```
const homeSlice = createSlice({
  name: "home",
  initialState: {
    banners: [],
    recommends: []
  },
  reducers: {
    changeBanners(state) {
      state.banners = []
    },
    changeRecommends(state) {
      state.recommends = []
    }
  }
})

export const { changeBanners, changeRecommends } = homeSlice.actions

export default homeSlice.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
```

Redux Toolkit的异步操作

- 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entraReducer中监听这些结果：
  - 见右图

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

  }
}
```

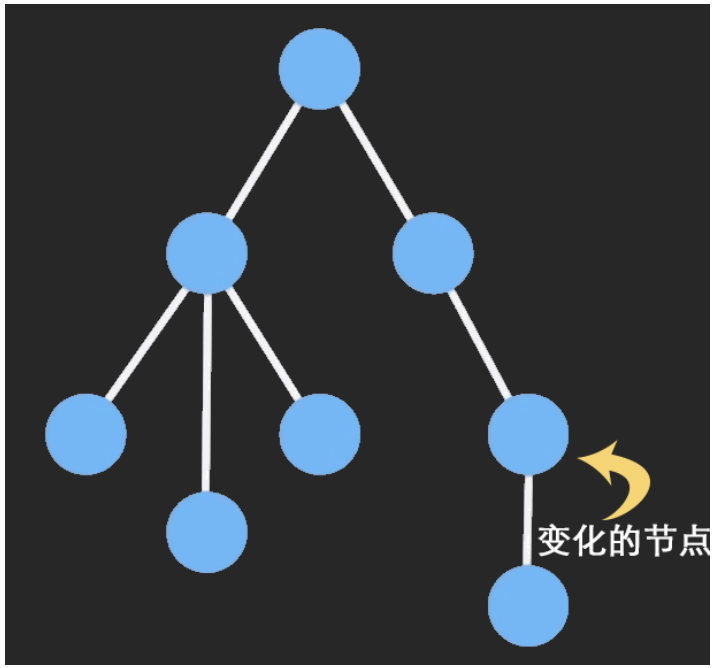
extraReducer的另外一种写法

- **extraReducer还可以传入一个函数，函数接受一个builder参数。**
  - 我们可以向builder中添加case来监听异步操作的结果：

```
extraReducers: (builder) => {
  builder.addCase(getHomeMultidataAction.pending, (state) => {
    console.log("getHomeMultidataAction pending")
  }).addCase(getHomeMultidataAction.fulfilled, (state, {payload}) =>
    state.banners = payload.banner.list
    state.recommends = payload.recommend.list
  })
}
```

Redux Toolkit的数据不可变性（了解）

- **在React开发中，我们总是会强调数据的不可变性：**
  - 无论是类组件中的state，还是redux中管理的state；
  - 事实上在整个JavaScript编码过程中，数据的不可变性都是非常重要的；
- **所以在前面我们经常会进行浅拷贝来完成某些操作，但是浅拷贝事实上也是存在问题的：**
  - 比如过大的对象，进行浅拷贝也会造成性能的浪费；
  - 比如浅拷贝后的对象，在深层改变时，依然会对之前的对象产生影响；
- **事实上Redux Toolkit底层使用了immerjs的一个库来保证数据的不可变性。**
- **在我们公众号的一片文章中也有专门讲解immutable-js库的底层原理和使用方法：**
  - https://mp.weixin.qq.com/s/hfeCDCcodBCGS5GpedxCGg
- **为了节约内存，又出现了一个新的算法：Persistent Data Structure（持久化数据结构或一致性数据结构）；**
  - 用一种数据结构来保存数据；
  - 当数据被修改时，会返回一个对象，但是新的对象会尽可能的利用之前的数据结构而不会对内存造成浪费；



自定义connect函数

```
export default function connect(mapStateToProps, mapDispatchToProps) {
  return function handleMapCpn(WrappedComponent) {
    return class extends PureComponent {
      constructor(props) {
        super(props);

        this.state = {
          storeState: mapStateToProps(store.getState())
        }
      }

      componentDidMount() {
        this.unsubscribe = store.subscribe(() => {
          this.setState({
            storeState: mapStateToProps(store.getState())
          })
        })
      }

      componentWillUnmount() {
        this.unsubscribe();
      }

      render() {
        return <WrappedComponent {...this.props}
                                 {...mapStateToProps(store.getState())}
                                 {...mapDispatchToProps(store.dispatch)}/>
      }
    }
  }
```

```
const mapStateToProps = state => {
  return {
    counter: state.counter
  }
}

const mapDispatchToProps = dispatch => {
  return {
    addNumber: function(number) {
      dispatch(addAction(number));
    }
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(Home);
```

context处理store

- **但是上面的connect函数有一个很大的缺陷：依赖导入的store**
  - 如果我们将其封装成一个独立的库，需要依赖用于创建的store，我们应该如何去获取呢?
  - 难道让用户来修改我们的源码吗？不太现实；
- **正确的做法是我们提供一个Provider，Provider来自于我们创建的Context，让用户将store传入到value中即可；**

```
import { StoreContext } from './utils/context';
import store from './store';

ReactDOM.render(
  <StoreContext.Provider value={store}>
    <App />
  </StoreContext.Provider>,
  document.getElementById('root')
);
```

```
import React, { PureComponent } from "react";

import { StoreContext } from './context';

export default function connect(mapStateToProps, mapDispatchToProps) {
  return function handleMapCpn(WrappedComponent) {
    class ConnectCpn extends PureComponent {
      constructor(props, context) {…
      }

      componentDidMount() {…
      }

      componentWillUnmount() {
        this.unsubscribe();
      }

      render() {
        return <WrappedComponent {...this.props}
          {...mapStateToProps(this.context.getState())}
          {...mapDispatchToProps(this.context.dispatch)} />
      }
    }

    ConnectCpn.contextType = StoreContext;

    return ConnectCpn;
  }
}
```

打印日志需求

- **前面我们已经提过，中间件的目的是在redux中插入一些自己的操作：**
  - 比如我们现在有一个需求，在dispatch之前，打印一下本次的action对象，dispatch完成之后可以打印一下最新的store state；
  - 也就是我们需要将对应的代码插入到redux的某部分，让之后所有的dispatch都可以包含这样的操作；
- **如果没有中间件，我们是否可以实现类似的代码呢？ 可以在派发的前后进行相关的打印。**
- **但是这种方式缺陷非常明显：**
  - 首先，每一次的dispatch操作，我们都需要在前面加上这样的逻辑代码；
  - 其次，存在大量重复的代码，会非常麻烦和臃肿；
- **是否有一种更优雅的方式来处理这样的相同逻辑呢？**
  - 我们可以将代码封装到一个独立的函数中
- **但是这样的代码有一个非常大的缺陷：**
  - 调用者（使用者）在使用我的dispatch时，必须使用我另外封装的一个函数dispatchAndLog；
  - 显然，对于调用者来说，很难记住这样的API，更加习惯的方式是直接调用dispatch；

修改dispatch

- **事实上，我们可以利用一个hack一点的技术：Monkey Patching，利用它可以修改原有的程序逻辑；**
- **我们对代码进行如下的修改：**
  - 这样就意味着我们已经直接修改了dispatch的调用过程；
  - 在调用dispatch的过程中，真正调用的函数其实是dispatchAndLog；
- **当然，我们可以将它封装到一个模块中，只要调用这个模块中的函数，就可以对store进行这样的处理：**

```
function patchLogging(store) {
  let next = store.dispatch;

  function dispatchAndLog(action) {
    console.log("dispatching:", action);
    next(addAction(5));
    console.log("新的state:", store.getState());
  }

  store.dispatch = dispatchAndLog;
}
```

thunk需求

- **redux-thunk的作用：**
  - 我们知道redux中利用一个中间件redux-thunk可以让我们的dispatch不再只是处理对象，并且可以处理函数；
  - 那么redux-thunk中的基本实现过程是怎么样的呢？事实上非常的简单。
- **我们来看下面的代码：**
  - 我们又对dispatch进行转换，这个dispatch会判断传入的

```
function patchThunk(store) {
  let next = store.dispatch;

  function dispatchAndThunk(action) {
    if (typeof action === "function") {
      action(store.dispatch, store.getState);
    } else {
      next(action);
    }
  }

  store.dispatch = dispatchAndThunk;
}
```

合并中间件

- 单个调用某个函数来合并中间件并不是特别的方便，我们可以封装一个函数来实现所有的中间件合并：

```
function applyMiddleware(store, middlewares) {
  middlewares = middlewares.slice();

  middlewares.forEach(middleware => {
    store.dispatch = middleware(store);
  })
}

applyMiddleware(store, [patchLogging, patchThunk]);
```

- 我们来理解一下上面操作之后，代码的流程：

- 当然，真实的中间件实现起来会更加的灵活，这里我们仅仅做一个抛砖引玉，有兴趣可以参考redux合并中间件的源码流程。

React中的state如何管理

- **我们学习了Redux用来管理我们的应用状态，并且非常好用（当然，你学会前提下，没有学会，好好回顾一下）。**
- **目前我们已经主要学习了三种状态管理方式：**
  - 方式一：组件中自己的state管理；
  - 方式二：Context数据的共享状态；
  - 方式三：Redux管理应用状态；

- **在开发中如何选择呢?**
  - 首先，这个没有一个标准的答案；
  - 某些用户，选择将所有的状态放到redux中进行管理，因为这样方便追踪和共享；
  - 有些用户，选择将某些组件自己的状态放到组件内部进行管理；
  - 有些用户，将类似于主题、用户信息等数据放到Context中进行共享和管理；
  - 做一个开发者，到底选择怎样的状态管理方式，是你的工作之一，可以一个最好的平衡方式（Find a balance that works for you, and go with it.）；

React中的state如何管理

- **Redux的作者有给出自己的建议：**

As Dan Abramov said:

> The way I classify it is when ever state needs to be shared by multiple components or multiple pages and we need to persist some data over route changes, all that data should go inside the redux store.

- **目前项目中我采用的state管理方案：**
  - UI相关的组件内部可以维护的状态，在组件内部自己来维护；
  - 大部分需要共享的状态，都交给redux来管理和维护；
  - 从服务器请求的数据（包括请求的操作），交给redux来维护；

- **当然，根据不同的情况会进行适当的调整，在后续学习项目实战时，我也会再次讲解以实战的角度来设计数据的管理方案。**