07.17 【课件】redux & react-redux 和 mobx & mobx-react 原理详解

本节课的主题

通过核心原理讲解 redux, react-redux 和 mobx, mobx-react 的实现，针对两个状态管理库的特点，对使用场景、核心实现进行分析。

核心概念

一、Redux 是基于状态管理与共享而生的一套单向数据流方案。独立于 react 的 JS 库，可通过 react-redux 桥梁应用于 react。（推崇思想：immutable，便于时间旅行，MVC）

主要涉及到的接口概念：

redux：

```
store = createStore(reducer)
store.dispatch(action)
store.getState()
store.subscribe(listener)
combineReducers
bindActionCreators(actionCreator, dispatch)
```

reat-redux：

```
connect(mapStateToProps, mapDispatchToProps)(ViewComponent)
Provider + context
```

ViewComponent 是开发者定义的容器组件。

二、Mobx 是基于 defineProperty（v4 及之前）或 Proxy（v4 之后）来实现对数据的劫持并响应动作的状态管理方案，可通过 mobx-react 桥梁应用于 react。（推崇思想：mutable，MVVM，时间旅行可借助 mobx-state-tree 库，提示：快照）

mobx 的一些常用接口：

```
configure({ enforceActions: true })
observable
reaction
autorun
runInAction
computed
action
flows
...
```

mobx-react:

```
observer
inject
Provider + context
makeAutoObservable
```

从用法到原理

> 凡是涉及数据共享的方案，我们首先要想到的就是：

（1）组件之外声明要共享的状态（包括安全合法修改状态的方法）；

（2）想方设法将该状态注入 UI 组件；

（3）让组件的更新与状态的变化同步。

记住这 3 条规则，便于我们理解现有的状态管理库逻辑，以及为实现我们自己的小型状态管理库提供思路。关于（1），并不是不能在组件内部声明要共享的状态，因为状态管理库与 UI 库是独立的，状态与组件解耦是必然的。

Redux

步骤（1）

```
// store.js
import { createStore } from 'redux';

const initialState = { list: [] };
// reducers 会在独立的文件
function reducers(state = initialState, action) {
   switch(action.type) {
     case 'ADD': {
       return {...state, list: [...state.list, action.payload]};
     }
     default: return state;
   }
}
const store = createStore(reducers);

export default store;
```

步骤（2.1）：注入项目根容器

```
import { render } from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';
```

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

想必学习上节课的同学已经知道怎么实现 `Provider` 了，没错，就是 `context API`，这样，App 以及所有后代组件均能通过 `context.Consumer` 访问到 store 的值，只不过，`Consumer` 并没有被 `react-redux` 显式暴露而已。这里先埋下伏笔。

步骤（2.2）：分发到模块组件（并不限定，任意组件均可以）

我们直觉是，导入产生上述 `Provider` 的 context 对象，在下文中使用 context.Consumer 标签包裹模块组件，从 Consumer 标签的 children 方法参数中获取 store。或者使用 useContext(context)返回的 store 来应用到下文。但这么实现的前提是，`react-redux` 向我们暴露了这个 context，然而并没有。

实践中的注入过程如下：

react-redux

```
import { connect } from 'react-redux';
import Container from './Container'; // 某模块的根组件

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

这里不要忽略一个问题，即上面的 `Container` 组件均将作为 `Provider` 组件的后代元素，所以才能使用 context。connect 是一个高阶函数，连续两次调用后，返回的仍是一个组件，只不过向最初的 `Container` 组件多注入了一些 props。这时候我们的思路是，connect 方法一定与上述 Provider 组件共享一个 context 对象！

假设在 react-redux 的源码中，导出的 Provider 如下：

```
const { Provider, Consumer } = createContext();
// Provider 被应用在上面的 App 组件外面
```

那么基于这同一个 `Consumer` 我们实现 connect 可以如下：

```
import React from 'react';

export const connect = (mapStateToProps, mapDispatchToProps) =>
  Component => function Connect(props) {
     return (
```

```
      <Consumer>
       {
         store => <Component
         // 传入该组件的 props，需要由 connect 这个高阶组件原样传回原组件
         { ...props }
         { ...mapStateToProps(store.getState()) }
         { ...mapDispatchToProps(store.dispatch) }
        />
       }
     </Consumer>
   );
 }
```

步骤（3）

根据步骤（2），我们已经可以在组件 `Container` 中访问到 store 中的数据和更新数据的方法了，形参 `Component` 即上文提到的 `Container`。最后一个问题，当 UI 组件触发了 action 时，store 虽然能够变化，但是组件并不会渲染更新。React 组件更新的前提是调用自身的 setState 方法，或者其父组件更新导致的 props 变化。对于状态的赋值（这里指全局 store），并不能触发更新。因此，我们完善 Connect 组件如下：

```
import React, { useContext, useEffect, useState } from 'react';
// const { Provider, Consumer } = createContext();
const reduxContext = createContext();

export const connect = (mapStateToProps, mapDispatchToProps) =>
  Component => function Connect(props) {
    const store = useContext(reduxContext);

    const [, setCount] = useState(true);
    const forceUpdate = () => setCount(value => !value);
    useEffect(() => store.subscribe(forceUpdate), []);
    // 每当 store 有更新，都将执行这里注册的 forceUpdate 方
    // 法，用来更新 Connect 组件，后代组件也会随之更新。

    return (
      <Component
        // 传入该组件的 props，需要由 connect 这个高阶组件原样传回原组件
        { ...props }
        { ...mapStateToProps(store.getState()) }
        { ...mapDispatchToProps(store.dispatch) }
       />
    );
}
```

至此，redux + react-redux 的整个流程已跑通！我们的焦点集中在了 store 对象的方法上。

注意标红的方法，而 store 是 createStore 方法的返回值。于是我们根据已知信息，来反推 createStore 的实现：

```
function createStore(reducers) {
  // 其实完备的 createStore 还会有其他参数，这里只介绍核心逻辑
  let state, listeners = [];
  const store = {
    dispatch(action) {
      state = reducers(state, action);
      // 每当发起一个 action 的时候，即 store 发生变化，
      // 就执行一遍之前订阅过的事件
      listeners.forEach(listener => listener());
      return action; // 可以不返回，有返回值的话允许我们多次调用
    },
    getState() {
      // 闭包的形式保证 state 必须通过暴露的方法来更新
      return state;
    },
    subscribe(listener) {
      listeners.push(listener);
      // 每当订阅一个事件的时候，随即返回注销该事件的方法
      return function unsubscribe() {
        listeners = listeners.filter(cur => cur !== listener);
      };
    }
  };
  store.dispatch({ type: '@@redux-init@@' });
  return store;
}
```

createStore 的源码比较繁多，在理解其原理的前提下，极简的实现大抵如此。倒数第二行的主动调用，目的是初始化 state，type 是 reducers 中不存在的值，因此在首次执行会将 initialState 赋值给 state。

mapStateToProps， mapDispatchToProps 定义了 UI 组件能获得哪些 props，同学们可以先思考一下。

mobx

回顾 3 个步骤：创建状态，状态注入，状态与更新同步，我们依然按照这个过程学习 mobx 与 mobx-react。

步骤（1）

由于 mobx 的状态基于对普通对象的封装（代理），所以状态的声明借助方法 observable：

```
import { observable } from 'mobx';
const object = observable({ value: 0 });
console.log(object.value);
// 基本类型的值包装
const count = observable.box(1);
console.log(count.get());
```

也可以将响应式数据包装到类的属性中：

```
class State {
  name = observable.box('张三')
  something = observable({ money: 123, age: 24 })
}
const state = new State();
state.name.get();
state.name.set('李四');
state.something.money = 0;
```

通常借助装饰器，直接修饰类的属性或方法（需要 babel 支持），注意与上述方式的差异。

```
class State {
  @observable name = '张三'
  @observable something = { money: 123, age: 24 }
}
const state = new State();
state.name = '李四';
state.something = { text: '文本', score: 120 };
```

状态的声明形式多种多样，但本质就是为了创建具有【响应能力】的数据源。

步骤（2）

对于局部状态（非全局共享），可直接将上述的数据源挂载到 react 组件上：

```
class App extends React.Component {
  state = new State()
  render() {
    return <span>{ this.state.name }</span>;
  }
}

// 也可以直接在组件中声明状态！
class App extends React.Component {
```

```
  @observable name = '王麻子'
  render() {
    return <span>{ this.name }</span>;
  }
}

import { useState } from 'react';
function App() {
  const [name] = useState(() => observable.box('奥利奥'));
  return <span>{ name.get() }</span>;
}
```

上面的状态声明和注入组件的方式及其灵活，而且还有不同的修改方式，强烈建议以老师课堂的讲解使用方式(下文)为准，否则及易引发各种各样的 bug!

步骤（3）

走到第二步，组件能够正常显示状态中定义的数据，也可以通过用户行为修改状态，如下所示：

```
class App extends React.Component {
  @observable name = '王麻子'

  onChange = (e) => {
    this.name = e.target.value;
  }

  render() {
    return <>
      <span>{ this.name }</span>
      <input onChange={this.onChange} />
    </>;
  }
}
```

但是这样页面并不会更新(span 标签的值不变),站在 react 的角度来说,没有触发 setState 或 forceUpdate 方法。所以，需要一个桥梁 mobx-react。

mobx-react

```
import { observer } from 'mobx-react';
// 装饰 App 组件
@observer
class App extends React.Component {
```

```
  @observable name = '王麻子'
  age = 24

  onChange = (e) => {
    this.name = e.target.value;
  }

  render() {
    return <>
      <span>{ this.name }</span>
      <input onChange={this.onChange} />
    </>;
  }
}
// 或者 export default observer(App);
```

为了规范状态的修改，区别于响应式属性 name 和普通属性 age，对 name 的操作应当在 action 中进行，尤其是严格模式时，configure({ enforceActions: true })，在 action 之外修改 name 将会报错。因此上面的 onChange 方法要这样写：

```
  import { action } from 'mobx'
  ...
  onChange = action(e => {
    this.name = e.target.value;
  })

  // 或者
  @action
  onChange(e) {
    this.name = e.target.value;
  }
```

对于异步修改状态：

```
  import { runInAction } from 'mobx'

  @data = {}

  componentDidMount() {
    fetch('/api').then(runInAction(data => {
      this.data = data;
    }));
  }
```

mobx 也提供了较为优雅的形式：

```
  import { flow } from 'mobx'
```

```
...

@data = {}

fetch = flow(function *(){
  const data = yield fetch('/api');
  this.data = data;
})

componentDidMount() {
   this.fetch();
}
```

对于局部状态，以上的用例是比较常见的，而对于全局状态，我们借助 context API 来实现一下：

```
import { createContext, useContext } from 'react';
import { render } from 'react-dom';
import { observable, action, computed } from 'mobx';

class User {
  @observable name = ''
  @observable age = 18
  @observable school = 'Qinghua'

  @computed get detail() {
    return this.name + this.age + this.school;
  }

  @action onChange(obj) {
     Object.assign(this, obj);
  }
}

class Status {
  @observable running = false
  @observable eating = true
  @observable sleeping = false

  @action onChange(obj) {
    Object.assign(this, obj);
  }
}
```

```
class Message {
  constructor() {
    makeAutoObservable(this); // mobx 5+ 版本的使用方式
  }
  count = 1
  onChange(value) {
    this.count += value;
  }
}
// 将不同模块的状态集中为一个 store
const store = {
  user: new User(),
  status: new Status()
};

const mobxContext = createContext();

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

// 为便于后代组件消费 store，直接将 Consumer 封装出来

function inject(Component) {
    const StateComponent = props => {
       const store = useContext(mobxContext);
       return <Component { ...props } { ...store } />
    };

    return StateComponent;
}
// 下文使用 inject(Home)，或 @inject class Home extends ...
// 均能访问到所有的模块
```

其实，我们真正从 mobx-react 中引入的 inject 方法，通常作为装饰器使用，需要定义将注入到被装饰组件的 props，例如：

```
@inject(['user', 'status'])
export default class App extends React.Component {
    render() {
      console.log(this.props); // { user, status }
```

```
    }
  }
  // 或这种使用方式
  @inject(({ user }) => ({ user }))
  export default class App extends React.Component {
      render() {
        console.log(this.props); // { user }
      }
  }
```

*装饰器的概念同学们先行了解

可以看到，mobx-react 中的 inject 有入参，执行一次后才去装饰组件，这样可以人为控制装饰的功能。迅速实现一下？

Mobx 原理篇

每次到了原理环节，就显得过于【敷衍】。当我们去面试的时候，被问道 Vue/mobx 的响应式原理，每个人都能说出下面这两个“兄弟”。但是似乎很多同学自己也说服不了自己——这么几行代码，就把一个库的底层说完了，如果再深层探讨一下如何自己实现一下，那不就打脸了嘛。

1. Object.defineProperty

```
const data = { value: 1 };

Object.defineProperty(data, '_value', {
  get() {
    // 此处依赖收集
     console.log('访问了', '_value');
     return this.value;
  },
  set(v) {
     // 此处执行回调更新
     console.log('访问了', '_value');
     this.value = v;
  }
});
```

2. Proxy

```
const data = { value: 1 };

const proxy = new Proxy(data, {
  get(target, key) {
    // 此处依赖收集
     console.log('访问了', key);
     return target[key];
  },
  set(target, key, value) {
    // 此处执行回调更新
    console.log('修改了', key);
    return Reflect.set(target, key, value);
  }
});
```

以 Proxy 为例，我们现场实现一个极简的响应库，仅作为响应式核心原理的学习使用。