

# 课件 1218 redux、react-redux、mobx、mobx-react 原理详解

## 核心概念

redux 和 mobx 都是为状态管理而生，与 react 本身并无关联的两个库，为了结合 react，分别借助了 react-redux、mobx-react 作为桥梁，因此能在 react 环境中完美运行。状态管理本身是一个抽象的概念，对一个 react 组件而言，state 是在父组件维护，还是在本组件声明，抑或是在组件外部借助 createContext 传递，都是一种状态管理方式。这种框架自带的管理方式，会因不同开发者的风格而迥乎不同——对于大型项目的维护与迭代，显然很不利。但不管使用哪种管理状态的方式，有一条主线是一致的，我们把这条主线提取出来：

- 创建 state
- 注入 state
- state 的变化触发 UI 的更新 (rerender)

下文，我们将紧紧沿着这条主线学习本文的主题。本文涉及到的主要方法：

### ◦ redux + react-redux

```
1 // redux
2 store = createStore(reducer)
3 store.dispatch(action)
4 store.getState()
5 store.subscribe(listener)
6 combineReducers
7 bindActionCreators(actionCreator, dispatch)
8 异步中间件 redux-thunk
9
10 // react-redux
11 connect(mapStateToProps, mapDispatchToProps)(App)
12 Provider + context
13 // hook 版本:
14 useSelector
15 useDispatch
```

### ◦ mobx + mobx-react

mobx-react 依赖于 mobx-react-lite。后者是应用于函数式组件的 mobx 库，也可以单独使用。mobx-react 既能支持 class 组件，又能支持函数组件。如果只用 hook 版本的接口，可以考虑只引入 mobx-react-lite。

```
1 // mobx
2 observable
3 reaction
4 autorun
5 computed
```

```
6 action
7 flow
8 ...
9 // mobx-react:
10 observer
11 inject
12 Provider + context
13 // hook 版本:
14 // 关注废弃中的接口 https://www.npmjs.com/package/mobx-react-lite
15 useLocalObservable // useLocalStore 已经宣告废弃中, 直接学习 useLocalObservable
16 Observer
```

## 课程目标

熟悉状态管理的基本流程, 掌握 `redux + react-redux`、`mobx + mobx-react` 的使用和基本原理, 熟练运用上面列出的方法或 api, 了解 `mobx` 性能突出的本质并能基于最佳实践进行代码实现。基本要求: 两个框架 hook api 的掌握运用。

## 准备工作

```
1 // 在文件夹内打开终端, 确保全局安装过 yarn
2
3 yarn create react-app mobx-redux // 创建 mobx-redux 项目
4 cd mobx-redux
5 // 进入项目目录安装这 4 个依赖
6 yarn add redux react-redux mobx mobx-react -S
7 // 就绪后启动项目
8 yarn start
```

## 原理讲解

### redux

redux 推崇单项数据流, 状态的 immutable。即状态只能由 store 派发, UI 层展示, 不能反过来, 状态的更新只能通过 `action => reducer` 触发, 并替换原先的状态, 而非修改原先的状态。

回顾状态管理的主线 (创建 state、注入 state、UI 与 state 的同步), 我们使用 redux 实践这一过程。

#### 1. 创建 state

```
1 // store.js
2 import { createStore } from 'redux'
3 // store 是状态的容器, 包含了状态、修改状态的方法, 监听状态改变的方法。
4 const initialState = { count: 0 };
5 const reducer = function (state = initialState, action) {
```

```

6   switch (action.type) {
7     case 'ADD': {
8       return { count: state.count + 1 };
9     }
10    default:
11      return state;
12  }
13 }
14
15 const store = createStore(reducer); // 先忽略其他参数
16 // 此处可以断点查看 store
17 export default store;

```

## 2. 注入 state

```

1 // App.js
2 import React, { Component } from 'react';
3 import store from './store';
4
5 export default class App extends Component {
6   onAdd = () => {
7     store.dispatch({
8       type: 'ADD'
9     });
10  };
11  render() {
12    return (
13      <div className="App">
14        App-{store.getState().count}
15        <button onClick={this.onAdd}>增加</button>
16      </div>
17    );
18  }
19 }

```

前两步可以展示状态了，但是点击按钮，可以得知状态发生了变化了，但是界面不更新。所以还得有一步：

## 3. UI 与 state 的同步

```

1 // App.js
2 import React, { Component } from 'react';
3 import store from './store';
4

```

```

5 export default class App extends Component {
6   constructor() {
7     super();
8     store.subscribe(() => {
9       this.forceUpdate(); // 组件的强制刷新方法
10    });
11  }
12  ...
13 }

```

现在点击按钮，状态的变更与 UI 的展示便同步了！其实这个过程，通过分析 redux 的源码就能一清二楚了，核心原理等价于下面的逻辑：

```

1 function createStore(reducer) {
2   let state;
3   const listeners = []; // 订阅事件的数组
4   const getState = () => {
5     return state;
6   };
7
8   const dispatch = action => {
9     state = reducer(state, action);
10    listeners.forEach(listener => listener());
11  };
12
13   const subscribe = listener => {
14     if (!listeners.includes(listener)) {
15       listeners.push(listener);
16     }
17     return function unsubscribe() {
18       listeners = listeners.filter(l => l !== listener);
19     };
20  };
21   dispatch({ type: '@@redux-init@@' });
22   // 执行一次业务中不存在的 type，目的是初始化 state
23   return { // 关注我们前面用到的 3 个接口
24     getState, dispatch, subscribe,
25   };
26 }

```

使用这里的 createStore 替换 redux 中导入的 createStore，也能正常运行~

真实的 `redux` 源码涉及到 `currentListeners`, `nextListeners` `replaceReducer`, `[$$observable]: observable` 等内容，感兴趣的可以自行了解。

实践中，我们不会像上面的例子那样，直接把 `store` 引到业务组件，而是借助 `react-redux` 的 `connect` 方法，将组件需要的状态注入到 `props` 中，像这样：

```
1 import { connect } from 'react-redux';
2 // 这两个映射用来生成注入组件的 props
3 const mapStateToProps = state => ({
4   count: state.count
5 });
6 const mapDispatchToProps = dispatch => ({
7   dispatch,
8   add: () => dispatch({ type: 'ADD' })
9 });
10
11 export default connect(mapStateToProps, mapDispatchToProps)(App);
12
```

那么 `connect` 又是怎么与 `store` 关联起来的呢？不得不先来了解一下 `react context`（熟悉的可以跳过）：

`context` 的使用方式很灵活，可以通过 `Provider/Consumer`，也可以通过 `contextTypes/getChildContext`，`useContext` 等方式，具体哪种可以根据便利性决定。如

```
1 import { createContext, useContext, useState } from 'react';
2
3 const Context = createContext();
4 // 子组件通过 context 拿到上层数据
5 function Sub() {
6   const ctx = useContext(Context); // 也可以使用 Consumer 来获取 context 对象
7   return (
8     <div>
9       { ctx.count }
10      <button onClick={ctx.increment}>增加</button>
11      <button onClick={ctx.decrement}>减少</button>
12    </div>
13  );
14 }
15 // 父组件通过 Context.Provider 传递数据到下级、后代组件
16 export default function Parent() {
17   const [count, setCount] = useState(0);
18   const value = {
```

```

19     count,
20     increment() {
21         setCount(c => c + 1);
22     },
23     decrement() {
24         setCount(c => c - 1);
25     }
26 };
27
28 return (
29     <Context.Provider value={value}>
30         <Sub />
31     </Context.Provider>
32 );
33 }
34

```

基于这个思路，我们来模拟 connect（这是一个高阶组件）：

```

1  import React, { Component } from 'react';
2  import PropTypes from 'prop-types';
3
4  export function connect(mapStateToProps, mapDispatchToProps) {
5      return function (WrappedComponent) {
6          class Connect extends React.Component {
7              static contextTypes = { // 这里使用函数式组件的话，也可以通过 useContext 来获取 context
8                  store: PropTypes.object
9              }
10             componentDidMount() {
11                 //从 context 获取 store 并订阅更新
12                 this.context.subscribe(this.forceUpdate.bind(this));
13             }
14             render() {
15                 return (
16                     <WrappedComponent
17                         // 传入该组件的 props, 需要由 connect 这个高阶组件原样传回原组件
18                         { ...this.props }
19                         // 根据 mapStateToProps 把 state 挂到 this.props 上
20                         { ...mapStateToProps(this.context.store.getState()) }
21                         // 根据 mapDispatchToProps 把 dispatch(action) 挂到 this.props 上
22                         { ...mapDispatchToProps(this.context.store.dispatch) }
23
24                     />

```



```

24     );
25   }
26 }
27
28   return Connect;
29 };
30 }

```

connect 使用 contextTypes 定义的方式获取 store，即 store 应该还在上一层组件。在大型项目中，我们一定能在入口组件（一般是 main.js 或 index.js）找到这样的代码：

```

1  import React from 'react';
2  import { Provider } from 'react-redux';
3  import store from './store';
4  import ReactDOM from 'react-dom';
5  // 忽略其他部分逻辑
6  ReactDOM.render(
7    <Provider store={store}>
8      <App />
9    </Provider>
10    document.querySelector('#app')
11  );

```

而 connect 导出的组件乃至整个项目的组件，都是 Provider 的后代组件，因此能够取到这里传递的 store。于是整条链路就打通了：

初始化：

- createStore 生成全局的 store =>
- Provider 注入 store 到项目根节点 =>
- connect 通过 context API 拿到 store 并映射为被包装组件的 props，同时把 Connect 组件的更新方法注册到 store listeners

更新：

- 被包装的组件调用 props 中的 action，即 dispatch => reducer
- reducer 生成新的 state，同时遍历执行上一步采集的 listeners
- Connect 基于变化的状态强制更新，包括子组件都会受到影响

最后的代码引入了 react-redux，能很容易想到，这里的 Provider 基于 createContext 实现，如果与 connect 合起来看的话更好理解：

```

1  // react-redux 同时导出 Provider 与 connect
2  import { createContext, useContext, useState } from 'react';
3  const ReduxContext = createContext();
4
5  const Provider = ReduxContext.Provider;

```

```

6
7 const connect = (mapStateToProps, mapDispatchToProps) => WrappedComponent => prop => {
8   const [, forceUpdate] = useState([]);
9   const { store } = useContext(ReduxContext); // 这里的参数一定与 Provider 是同一个对象
10  store.subscribe(() => forceUpdate([])); // 用于强制更新组件
11  const props = {
12    ...mapStateToProps(store.getState()),
13    ...mapDispatchToProps(store.dispatch),
14    {...prop}
15  };
16
17  return <WrappedComponent {...props} />;
18 }
19 export { Provider, connect };

```

## 从复杂到简单

我们知道 mapDispatchToProps 的结构通常是这样的（也可以是一个对象或缺省）：

```

1 const mapDispatchToProps = (dispatch, ownProps) => ({
2   onDecrement: (payload) => dispatch({ type: 'DECREMENT', payload }),
3   onAdd: () => dispatch({ type: 'ADD' }),
4   ...
5 });
6 // dispatch 的参数是一个 action

```

第一次变形：

```

1 function onDecrement(payload) {
2   return { type: 'DECREMENT', payload };
3 }
4 function onAdd() {
5   return { type: 'ADD' };
6 }
7 // 上面叫做 actionCreator
8 const mapDispatchToProps = (dispatch, ownProps) => ({
9   onDecrement: (...args) => dispatch(onDecrement(...args)),
10  onAdd: (...args) => dispatch(onAdd(...args)),
11  ...
12 });

```

我们发现 mapDispatchToProps 内部的代码相似度出奇得高，于是继续变形：



```

1 // action.js
2 export function onDecrement(payload) {
3   return { type: 'DECREMENT', payload };
4 }
5 export function onAdd() {
6   return { type: 'ADD' };
7 }
8
9 // connect 文件内
10 import * as actions from './action';
11 import { bindActionCreators } from 'redux';
12
13 const mapDispatchToProps = (dispatch, ownProps) => ({
14   ...bindActionCreators(actions, dispatch),
15   dispatch
16 });

```

这样，connect 包装过的组件，就能在 props 直接拿到与 action.js 导出的函数同名的方法了！

## 从同步到异步

通常，一个 action 会对应地调起一个 reducer。但是当发出请求的时候，往往需要在一个动作里执行多个 reducer，比如请求前的 loading，请求后的 loaded 或 error。虽然可以在组件内通过多次调用实现，但非常繁琐。这时候的 action，我们更希望能支持以函数的形式呈现，例如：

```

1 export const onDecrement = payload => (dispatch, getState) => {
2   dispatch({ type: 'LOADING' });
3   fetch(...).then(data => {
4     dispatch({ type: 'LOADED' });
5     dispatch({ type: 'DECREMENT', payload: data });
6   }).catch(err => {
7     dispatch({ type: 'ERROR', payload: err });
8   });
9 }

```

此时，dispatch 的参数不再只是 action 对象，还可以是一个函数。我们就需要借助中间件，以 redux-thunk 为例：

```

1 // store.js
2 import { createStore, applyMiddleware, combineReducers } from 'redux';
3 import thunk from 'redux-thunk';
4 // 如果只有一个 reducer

```

```

5  const store = createStore(reducer, applyMiddleware(thunk));
6  ...
7
8  // 当有多个 reducer 时，我们会在不同模块维护，然后集中到这里合并为一个 reducer:
9  // 借助 combineReducers。同时下文 mapStateToProps 及 useSelector
10 // 的第一个参数，也将相应地变成一个 { home, about } 对象
11 import home from '@home/reducer';
12 import about from '@about/reducer';
13 const reducer = combineReducers({
14   home,
15   aboutState: about // 可以重命名，保持下文使用的键名与此处一致即可
16 });
17 const store = createStore(reducer, applyMiddleware(thunk));
18 ...

```

当然了，有兴趣的也可以了解一下 redux-promise 中间件，不再赘述。

## 由高阶到 hook

使用 connect + mapStateToProps + mapDispatchToProps 来获取状态与方法的形式可以说是 react hooks 早先结合 redux 唯一的方式，但是在 react-redux 支持 hooks 之后，这一切越来越简单了：

```

1  // App.js
2  import React from 'react';
3  import { useSelector, useDispatch } from 'react-redux';
4
5  export default function App() {
6    const number = useSelector(state => state.number);
7    // 如果上文使用了 combineReducers，这里应该写成：
8    // const number = useSelector(state => state.home.number);
9    // 或 const number = useSelector(({ home }) => home.number);
10   const dispatch = useDispatch();
11   const onDecrement = () => {
12     dispatch({
13       type: 'DECREMENT',
14       payload: 2
15     });
16   };
17   return <button onClick={onDecrement}>{number}</button>;
18 }
19

```

注意，这里不需要使用高阶函数 connect 注入 props 。虽然二者混用能正常运转，但强烈不推荐！

## mobx

mobx 推崇数据响应式，状态的 mutable 。即状态创建后，后续都是在修改这个状态，基于代理拦截数据的 setter，从而触发副作用（在 react 中，包括 render 也可认为是 mobx 的副作用回调）。

1. 在 react 中使用 mobx 的时候，你应该忘记 react 自带的组件更新方式，时刻牢记这句话，否则使用 mobx 将失去价值，甚至引入 bug！
2. 不要随意解构或使用基本类型的变量代替代理对象的属性，下文将演示这样导致的问题。
3. 出于优化的目的，列表的每一项尽量封装为组件，这样 mobx 将会尽情体现它的速度！

mobx 最简单的示例：

```
1 import { observable, reaction } from 'mobx';
2
3 const data = observable({ value: 0 });
4 const dispose = reaction(
5   () => data.value,
6   (cur, prev) => {
7     console.log(cur, prev);
8     if (cur > 3) {
9       dispose();
10      alert('不再追踪 data');
11    }
12  });
13
14 export default function Mobx() {
15   const onChange = () => {
16     data.value++;
17   };
18   return (
19     <button onClick={onChange}>改变 data { data.value }</button>
20   );
21 }
```

可以看到，点击按钮时，控制台打印了 data.value 的值，但是组件并没有同步显示。回顾状态管理的主线（创建 state、注入 state、UI 与 state 的同步），我们分析这一过程：

```
1 // 1. 创建 state
2 const data = observable({ value: 0 });
3
4 // 2. 注入 state
5 <button onClick={onChange}>改变 data { data.value }</button>
```

```
6
7 // 3. UI 与 state 的同步
8 ?? onChange 是更新状态的, 那么 state => UI 这一步这里是没有的
```

补上 UI 与 state 的同步的逻辑:

```
1 import { observer } from 'mobx-react';
2 ...
3 export default observer(function Mobx() {
4   ...
5 });
```

这里的 observer 与 react-redux 中的 subscribe 订阅更新方法起到了一样的作用: 当状态变更时, subscribe listeners 通知组件更新, 而 observer 将重新执行我们传给它的函数 (这里是 Mobx 组件)。不同的是, 我们将完全抛弃诸如 setState, useState 等原生方法更新组件。

**错误示例:**

```
1 const value = data.value;
2 export default observer(function Mobx() {
3   return (
4     <button onClick={onChange}>改变 data{ value }</button>
5   );
6 });
```

**正常运行的示例:**

```
1
2 export default observer(function Mobx() {
3   const value = data.value;
4   return (
5     <button onClick={onChange}>改变 data{ value }</button>
6   );
7 });
```

上面两个示例的唯一区别就是, 将 value 属性的访问置于 observer 内或外。如果是外部, observer 内本质是使用了基本类型的值 value, 它不具有响应能力; 而 data.value 在 observer 内被访问, 将会在初次渲染时触发 observer 对 Mobx 函数的追踪, 便于后续 value 属性发生变化时, 重新执行 Mobx。这里我们可以重新审视一下代理的本质:

```
1 const data = { value: 0 };
2
3 const proxyData = new Proxy(data, {
4   get(target, key) {
```

```

5     const result = target[key];
6     // 此时进行 track
7     console.log('取值属性名', key, ', 值: ', result);
8     return result;
9 },
10 set(target, key, value) {
11     // 此时进行 trigger
12     console.log('修改属性名', key, ', 值: ', value);
13     return Reflect.set(target, key, value);
14 }
15 });
16 proxyData.value; // 取值
17 proxyData.value = 3; // 改值

```

针对对象属性的访问进行拦截，因此在拦截步骤中，应该出现形如 `obj[key]` 或 `obj.key` 形式的访问，这样 `track` 过程才能正确地将类组件的 `render` 方法或者 `observer` 的参数（函数组件）添加到副作用列表。所以基本类型的值，必须经历一次对象的包装，才能被代理：

```

1 const primitive = observable.box(0);
2 // 如果你了解过 Vue3 的 ref 方法，你也就理解这种做法和实现原理了。
3 export default observer(function Mobx() {
4     const value = primitive.get(); // 取值方式
5     const onChange = () => {
6         primitive.set(value + 1); // 更新方式
7     };
8     return (
9         <button onClick={onChange}>改变 data{ value }</button>
10     );
11 });

```

实践中，`react` 组件的状态应当在加载时创建，卸载后销毁，因此推荐的写法是外部定义 `State` 类，组件内再实例化状态：

```

1 import React, { Component } from 'react';
2 import { makeAutoObservable } from 'mobx';
3 import { observer } from 'mobx-react';
4
5 // 创建 state
6 class State {
7     constructor() {
8         makeAutoObservable(this); // 这是 mobx 6.x 的方式，可以代替下面的所有声明：
9         // this.value = observable.box(0);

```

```

10     // this.data = observable({});
11 }
12 value = 0
13 onChange = () => {
14     this.value++;
15 }
16 }
17
18 class Mobx extends Component {
19     state = new State() // 注入 state
20     render() {
21         return (
22             <button onClick={this.state.onChange}>改变 data{ this.state.value }</button>
23         );
24     }
25 }
26
27 export default observer(Mobx); // state 同步

```

使用装饰器,

```

yarn add @babel/plugin-proposal-decorators -D
// babel配置处
"plugins": [
    ["@babel/plugin-proposal-decorators", { "legacy": true }]
]

```

为了规范性, 开启严格模式下, 修改状态的方法应当使用 action 来修饰, 明确这是一个触发组件更新的方法, 绑定 this 可以使用 action.bound。

```

1 import { configure } from 'mobx';
2 // 开启后, action 外部修改状态将会抛出错误
3 configure({ enforceActions: true });
4

```

- class + mobx

```

1 import React, { Component } from 'react';
2 import { makeObservable, action, observable } from 'mobx';
3 import { observer } from 'mobx-react';
4 // 创建 state
5 class State {
6     constructor() {
7         // 这里不是自动处理, 而是人为控制哪些数据具有响应能力, 哪些方法是修改状态的

```



```

8     makeObservable(this, {
9         value: observable,
10        // 注意 onChangeValue 不是属性声明的箭头函数，所以应当绑定 state 实例
11        onChangeValue: action.bound,
12    });
13 }
14 count = 0
15 value = 0
16 onChangeValue() {
17     this.value++;
18 }
19 onChangeCount = () => {
20     this.count++;
21 }
22 }
23
24 @observer // state 同步
25 class Mobx extends Component {
26     state = new State() // 注入 state
27     render() {
28         return (<>
29             <button onClick={this.state.onChangeValue}>
30                 改变 value{ this.state.value }
31             </button>
32             &nbsp;
33             <button onClick={this.state.onChangeCount}>
34                 改变 count{ this.state.count }
35             </button>
36         </>
37     );
38 }
39 }
40
41 export default Mobx;

```

在上面的例子中，makeObservable 定义了响应能力的属性和方法，count 是普通属性，所以改变 count 按钮并不会看到页面有任何变化。相比于粗暴地使用 makeAutoObservable，能够使开发者更清晰地知道自己在干什么，从而达到节约性能的目的。那么，如果我们要实现请求的控制该怎么做呢？

```

1 import React, { Component } from 'react';
2 import { makeObservable, action, observable, flow, computed } from 'mobx';
3 import { observer } from 'mobx-react';

```

```

4
5 // 假装这是一个请求接口
6 const api = (name = '') => new Promise(resolve => {
7   setTimeout(() => {
8     const length = Math.ceil(Math.random() * 10);
9     resolve({
10       code: 200,
11       data: Array(length).fill(0).map((_, index) => ({ name: name + Math.random(), id:
12     }));
13   }, 1000);
14 });
15
16 class State {
17   constructor() {
18     // 这里不是自动处理，而是人为控制哪些数据具有响应能力，哪些方法是修改状态的
19     makeObservable(this, {
20       list: observable,
21       loading: observable,
22       assign: action.bound, // 如果你比较懒，这么干也不是不可以，该方法不是必须的
23       overflow: computed // 顺带把衍生值学了，衍生的属性只能访问，不要直接修改
24     });
25   }
26   list = []
27   loading = false
28   get overflow() {
29     return this.list.length > 30; // list 长度变化就会重新计算
30   },
31   // flow 是优雅地处理异步 action 的方法，所以这里相当于内部的代码也运行在 action 的作用内。
32   onFetch = flow(function *(string) {
33     this.loading = true;
34     const { data } = yield api(string); // 异常处理自己做~
35     this.list = data;
36     this.loading = false;
37     // 或者 this.assign({ list: data, loading: false });
38   })
39   // 这样定义一个方法，会失去语义性，即你都不知道自己在做啥，只知道是在修改一个值
40   assign(obj) {
41     Object.assign(this, obj);
42   }
43 }

```

```

45 export default @observer class Mobx extends Component {
46   state = new State()
47   render() {
48     if (this.state.loading) {
49       return <p>loading...</p>;
50     }
51     return (
52       <>
53         <div>
54           {
55             this.state.list.map(item => (
56               <p key={item.id}>{ item.name }</p>
57             ))
58           }
59         </div>
60         <button onClick={() => this.state.onFetch('响应')}>获取数据</button>
61       </>
62     );
63   }
64 }
65

```

有人是否会提出这样的问题，组件加载时就自动请求数据，是不是还得用上 react 生命周期？其实，基于 mobx 的状态管理，大多数情况是不需要使用生命周期的（不访问节点或全局注册事件的话），初始请求可以在 State 的构造函数中发起，参数可以来自 new State(参数)。那么 componentDidMount 中的逻辑，该怎么处理？

举例：

```

1  componentDidMount(preProps, preState) {
2    if (this.state.list.length > 5) {
3      alert('exceeded'); // 这是你自定义的逻辑
4    }
5  }

```

交给 mobx 来做，并且我们可以随时调用 this.dispose() 来取消这个监听。

```

1  import { reaction, autorun } from 'mobx';
2
3  state = ...
4  dispose = reaction(
5    () => this.state.list.length, // 每当表达式的结果有变，就会执行下一个方法
6    (newLen, oldLen, { dispose }) => {

```

```

7     if (newLen > 5) {
8         alert('exceeded');
9     }
10 }
11 )
12 render() {...}
13 // 如果想在创建时就执行，可以传入第三个参数 { fireImmediately: true },
14 // 不过在你不关心上一个值时，还有更简单的方法 autorun
15
16 // autorun 会直接响应 this.state.list 的长度变化，自动运行回调
17 dispose = autorun(() => {
18     if (this.state.list.length > 5) {
19         alert('exceeded');
20     }
21 })

```

在组件卸载时，执行 `dispose()` 取消订阅即可。

- functional + mobx

我们知道函数式组件的更新相当于重新调用组件自身（只有 `render` 部分），那么我们最先想到的是，怎么在实例化后的状态不会因为更新而被重置。最直接的处理方式：

```

1 import React, { useState } from 'react';
2 import State from './State';
3 import { observer } from 'mobx-react';
4
5 export default observer(function FunctionalMobx() {
6     const [state] = useState(() => new State());
7
8     if (state.loading) {
9         return <p>loading...</p>;
10    }
11    return (
12        <>
13            <div>
14                {
15                    state.list.map(item => (
16                        <p key={item.id}>{ item.name }</p>
17                    ))
18                }
19            </div>
20            <button onClick={() => state.onFetch('响应')}>获取数据</button>
21        </>

```

```
22   );  
23 });
```

这么跑起来确实没毛病，至于 reaction/autorun，也可以搬到 State 中去做。不过官方基于 hooks 实现了一套创建响应式数据的方法：

```
1  import React from 'react';  
2  import { useLocalObservable, observer } from 'mobx-react';  
3  
4  export default observer(function FunctionalMobx() {  
5    const state = useLocalObservable(() => ({  
6      list: [],  
7      get overflow() {  
8        return this.list.length > 30;  
9      },  
10     loading: false,  
11     async onFetch(string) { // 内部的 this 已经默认绑定到了 state  
12       this.loading = true;  
13       const { data } = await api(string);  
14       this.list = data;  
15       this.loading = false;  
16     }  
17   }));  
18  
19   if (state.loading) {  
20     return <p>loading...</p>;  
21   }  
22   return (  
23     <>  
24     <div>  
25       {  
26         state.list.map(item => (  
27           <p key={item.id}>{ item.name }</p>  
28         ))  
29       }  
30     </div>  
31     <button onClick={() => state.onFetch('响应')}>获取数据</button>  
32   </>  
33   );  
34 });
```

既然手机都从翻盖换触屏了，那手机壳也顺带换了吧：

```

1 import React from 'react';
2 import { useLocalObservable, Observer } from 'mobx-react';
3 // 注意, 导入的 observer 函数变成 Observer 组件了~
4 // 函数组件不再被 observer 包裹
5 export default function FunctionalMobx() {
6   const state = useLocalObservable(() => ({
7     // ...这里一样, 略
8   }));
9
10  return ( // 使用 Observer 包裹 [使用了响应式数据] 的 jsx, 并且是个函数
11    <Observer>
12      {() => {
13        if (state.loading) {
14          return <p>loading...</p>;
15        }
16        return (<>
17          <div>
18            {
19              state.list.map(item => (
20                <p key={item.id}>{ item.name }</p>
21              ))
18            }
22          </div>
23          <button onClick={() => state.onFetch('响应')}>获取数据</button>
24        </>);
25      }}
26    </Observer>
27  );
28 }

```

再次回顾状态管理的三步骤：创建 state、注入 state、state 与 UI 的更新，这次更好理解这个过程了，Observer 就是用于追踪页面更新与 state 使用情况的部分。而且跟踪组件的更新情况可以发现，未来 list 或 loading 的变化，都不再执行 Observer 标签之外的逻辑了！这意味着，使用 useCallback 优化函数引用的逻辑，也不再需要，下面的逻辑也不会再在更新阶段执行：

```

1 useEffect(() => {
2   // 除了 mount, 这里的逻辑不会因为 state.loading 的变化而执行, 因为组件更新只限定在
3   // 了 Observer 内。
4 }, [state.loading]);

```

上述特性并不意味着 useEffect 失去了用武之地。当 observer 监控整个组件或父组件更新引起的子组件 rerender，以及获取 DOM、注册全局事件、数据追踪时，useEffect 必不可少。例如：



### 1. 父组件的数据同步

```
function Sub ({ syncCount }) {  
  const const state = useLocalObservable(() => ({  
    syncCount  
  }));  
  useEffect(() => {  
    state.syncCount = syncCount;  
  }, [syncCount]);  
}
```

### 2. 添加全局事件、获取 DOM

```
const box = useRef();  
useEffect(() => {  
  function onResize() {  
    const style = getComputedStyle(box.current.style);  
  }  
  window.addEventListener('resize', onResize, false);  
  return () => window.removeEventListener('resize', onResize, false);  
}, []);
```

### 3. 数据追踪，一般不需要依赖项，想想为什么？

```
import { autorun, reaction } from 'mobx';  
useEffect(() => autorun(() => {  
  if (store.loaded) {  
    alert('下载完成');  
  }  
}), []);  
useEffect(() => reaction(  
  () => state.count > 5,  
  disabled => {  
    // 执行一些自定义逻辑  
  }), []);
```

下面是一个列表优化演示：

```
1  const state = useLocalObservable(() => ({  
2    ...  
3    onChange(item) {  
4      // 点击某个条目时，修改该条目的 name，还记得 mobx 使用的是同一份数据吗，这里的更改能影响到  
5      item.name = '我被修改了';  
6    }  
7  });  
8  return (  
9    <Observer>  
10     {() => {  
11       ...  
12     }  
13   )  
14 );
```

```

12     return (<>
13         <div>
14             {
15                 state.list.map(item => {
16                     console.log('render', item.name); // 这里记录组件的渲染情况
17                     return <p key={item.id} onClick={() => state.onChange(item)}>
18                         { item.name }
19                     </p>;
20                 })
21             }
22         </div>
23         <button onClick={() => state.onFetch('响应')}>获取数据</button>
24     </>);
25 }}
26 </Observer>
27 );

```

点击任意一项，所有项都有重新渲染：

App-0 增加

响应0.7353333890472233

我被修改了

响应0.6840688130745818

响应0.45235154864484173

响应0.06440505975998034

响应0.9598179562145133

响应0.2276092616640697

响应0.9995498881050864

响应0.3055674487837914

响应0.442129821029255

获取数据

元素
控制台
来源
网络
应用
内存
性能

top
过滤

render 响应0.7353333890472233  
render 我被修改了  
render 响应0.6840688130745818  
render 响应0.45235154864484173  
render 响应0.06440505975998034  
render 响应0.9598179562145133  
render 响应0.2276092616640697  
render 响应0.9995498881050864  
render 响应0.3055674487837914  
render 响应0.442129821029255

假如每一项都是一个复杂的组件呢？这是不是性能的浪费？let's fix it!

```

1  const Item = ({ item }) => ( // 这里是要使用 Observer 的，想想为什么？
2      <Observer>
3          {() => {
4              console.log('render', item.name);
5              return <p onClick={() => state.onChange(item)}>{ item.name }</p>;
6          }}
7      </Observer>
8  );

```

```

9
10 return ( // 这里的 Observer 追踪的是 [state.loading] 和 [state.list 整体] 的变化
11   <Observer>
12     {() => {
13       if (state.loading) {
14         return <p>loading...</p>;
15       }
16       return (<>
17         <div>
18           {
19             state.list.map(item => <Item key={item.id} item={item} />)
20           }
21         </div>
22         <button onClick={() => state.onFetch('响应')}>获取数据</button>
23       </>);
24     }}
25   </Observer>
26 );

```

现在点击某一项，其他项将不发生任何变化，因为 `item.name = '我被修改了'` 这行逻辑触发的变更，将直接跨过没有使用到 `item.name` 的 19 行，转而直接渲染子组件中使用了这个被修改了 `name` 的项，这样的渲染粒度十分精准，性能可以达到极致！

举一反三，我希望上面的渲染不要涉及静态的节点，因为 `item.name` 是内容，节点本身是不需要更新的，那么：

```

1 const Item = ({ item }) => (
2   <p onClick={() => state.onChange(item)}>
3     <Observer>{() => item.name}</Observer>
4   </p>
5 );

```

因为 `state.onChange` 是一个方法，它并不需要随着更新而改变，现在的组件更新，局限在了 `innerHTML` 内，是不是把传统的 `react function` 组件全量执行渲染给颠覆了呢？如果再次点击修改后的元素，你会发现即使触发了方法，但是值没变化，`Observer` 内的函数都不会执行。`mobx` 用得好好对于性能的提升是巨大的，但有的同学 `observer` 一把梭，直接将整个组件作为 `track` 的内容，这与直接调用 `react` 原生的 `setState` 何异呢？所以选择了 `mobx`，不要再掺入 `react` 的更新方式，以上面的组件为例，如果父组件触发了 `setState`，意味着本组件声明的方法会重新生成，`useEffect` 也可能受到影响，可能会难以预测的问题（虽然并不一定产生错误）。

针对 `mobx` 与 `react` 结合的大型项目，其实还有 `mobx-state-tree` 这个库，`mobx-state-tree` 之于 `react`，应该同 `redux` 之于 `react` 的关系更近似一些，本文不再对其展开。最后，我们类比 `react-redux` 的 `Provider + connect` 的方式，学习如何在全局状态共享中使用 `mobx`。

```

1 import React from 'react';
2 import { render } from 'react-dom';

```

```

3 import { Provider } from 'mobx-react';
4 import App from './App';
5 import store from './store';
6
7 render(
8   <Provider {...store}>
9     <App />
10  </Provider>,
11  document.getElementById('root')
12 )

```

store.js

```

1 class Home {
2   constructor() {
3     makeObservable(this, {
4       data: observable,
5       onChange: action.bound,
6     });
7   }
8   data = 'home'
9   onChange(data) {
10     this.data = data;
11   }
12 }
13
14 class About {
15   constructor() {
16     makeObservable(this, {
17       value: observable,
18       onChange: action.bound,
19     });
20   }
21   value = 'About'
22   onChange(value) {
23     this.value = value;
24   }
25 }
26 // 上面的模块应该分散在各自的业务中，这里就不使用导入的方式了
27 export default {
28   home: new Home(),

```

```
29   about: new About();
30 }
```

下级组件使用时（也可以使用类组件，装饰器的形式注入）：

```
1  import React from 'react';
2  import { inject, Observer } from 'mobx-react';
3
4  function MobxReact({ home, about }) {
5    return (
6      <h3>
7        <h2>mobx-react</h2>
8        <button onClick={() => home.onChange(Math.random())}>
9          home:
10         <Observer>{ () => home.data }</Observer>
11       </button>
12       <br />
13       <button onClick={() => about.onChange(Math.random())}>
14         about:
15         <Observer>{ () => about.value }</Observer>
16       </button>
17     </h3>
18   );
19 }
20
21 export default inject('home', 'about')(MobxReact);
22 // 以下的两种形式也可以
23 // 1.
24 export default inject(({ home, about }) => ({ home, about }))(MobxReact);
25 // 2.
26 @inject('home', 'about')
27 class MobxReact extends Component {}
28 export default MobxReact;
```

至此，本文的内容就满足常规开发的需要了，如有进阶需求请移步官网。