

19 | Redux：大型应用应该如何管理状态？

2022-05-11 蒋宏伟

《React Native 新架构实战课》

课程介绍 >



讲述：蒋宏伟

时长 24:02 大小 22.02M



你好，我是蒋宏伟。

今天这一讲，我要和你聊的是，如何使用 **Redux** 来管理复杂的、大型的应用状态。

有人认为，在 **React** 出了 **hooks** 之后，官方提供的 **useReducer** 和 **useContext** 的组合，已近似能够代替 **Redux** 了，所以，现在是时候抛弃 **Redux**，直接用 **useReducer** 和 **useContext** 对大型应用进行状态管理了。

也有人认为，虽然 **Redux** 解决了状态管理的问题，但是 **Redux** 模板代码太多，应该抛弃 **Redux**，改用 **Mobx** 或 **Zustand** 这类写起来更简单的工具。

但我认为，从目前来看，**Redux** 依旧是我们开发大型项目时，应该最优先考虑的状态管理工具。为什么呢？

一方面，大型项目的状态管理复杂度很高，`useContext` 并不是状态管理工具，它只是一个提供了跨层级传递状态的工具而已。真要拿 `useReducer` 和 `useContext` 来写大型项目，你需要写更多的模板代码，而且更难维护。

另一方面，开发大型项目需要考虑团队成员的协作成本，目前来看，无论是 [npm trends 上的下载量](#)，还是我对 `React Native` 开发者的 [调研报告](#) 都显示，`Redux` 的流行程度远超于其他状态管理工具。团队招一个新人，新人熟悉 `Redux` 概率远比熟悉 `Mobx`、`Zustand` 的概率更高，学习成本、协作成本也是最低的。

至于以前被大家吐槽最多的，`Redux` 模板代码多的问题，现在也可以使用 `Redux Toolkit` 来解决一部分了。`Redux` 官方是这么说的：

We want *all* Redux users to write their Redux code with Redux Toolkit, because it simplifies your code *and* eliminates many common Redux mistakes and bugs!

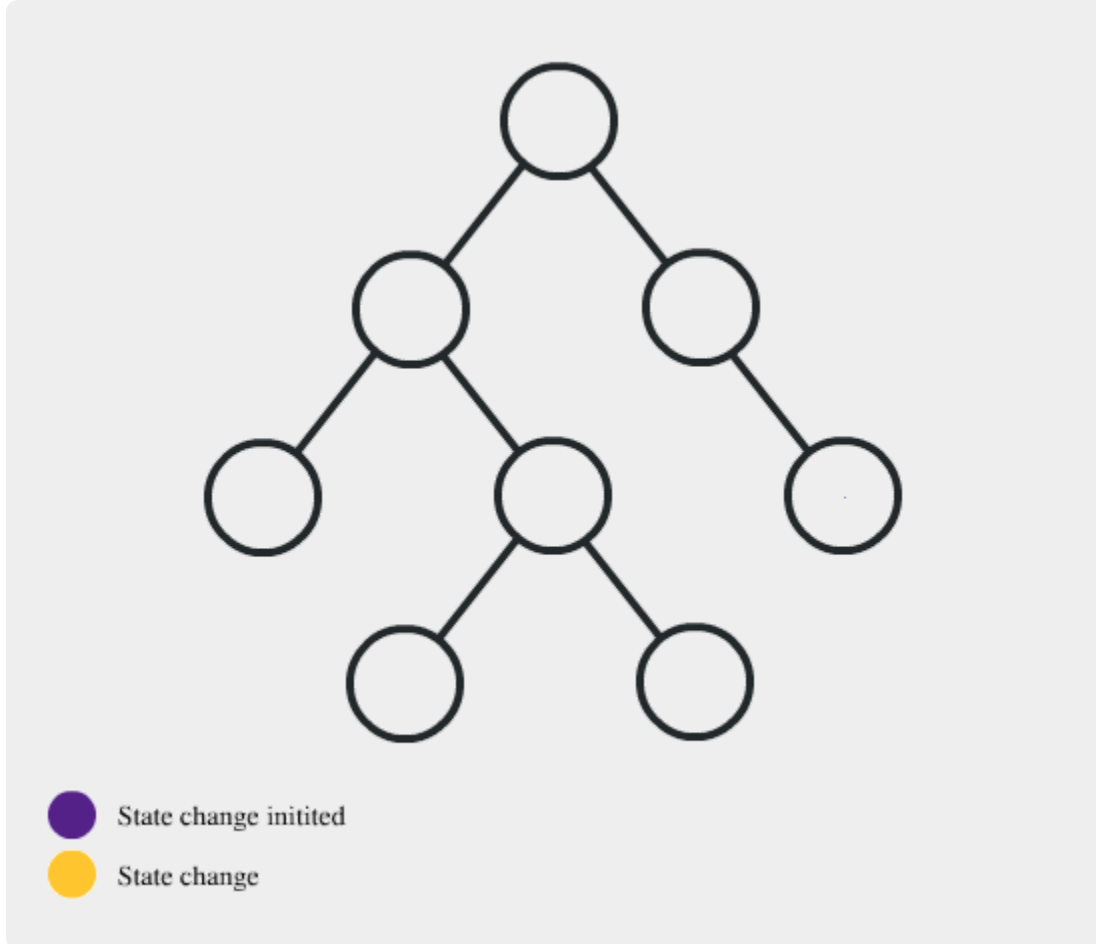
我们希望所有的 `Redux` 开发者都能用 `Redux Toolkit` 来写代码，因为它能简化代码，减少 BUG 和消除常见的对 `Redux` 的误解。

所以，今天这节课，我们就一起探究 `Redux/Redux Toolkit` 到底何时用、怎么用。

Redux 什么时候用？

首先，我们需要明确的是：一个应用的状态管理要复杂到什么程度才需要引入 `Redux`？一个最好的判断标准是，**当你觉得状态管理是你研发痛点的时候，你才需要开始着手解决。**

举个例子，在一个大型项目中，我们经常遇到的问题就是，全局状态管理的问题，示意图如下：



这张图来自 [《When do I know I'm ready for Redux?》](#)，图中描述的就是一个全局状态更新的难题。在一个 App 中，它有一个全局状态被多个组件使用了，因此它将该全局状态放在了 Root 组件中。当其中一个组件，触发了该全局的状态更新操作后，Root 的新状态会逐层传递到整个应用上。

虽然将全局状态挂在 Root 组件上，并通过将状态和状态更新函数逐层传递到各个组件上，能解决跨组件更新的难题。但是这个方案却带来了性能损耗和维护成本高等问题。

所以当你在使用 useState、useReducer 管理状态时，如果会遇到性能问题和维护性问题时，你就可以考虑使用 Redux 了。

你看，Redux 通过 Store 存储了全局状态，它不仅解决了状态需要逐层传递的问题，还避免了整个应用的 re-render，不容易出现相关的性能问题。

这时候，我们再来看什么时候需要 Redux，什么时候不需要 Redux 这个问题，相信你已经有了答案。

当你有大量的全局状态需要管理时，当应用状态频繁更新遇到性能瓶颈时，当管理状态的逻辑复杂到需要代码分治时，当多人协作开发需要遵守同一套最佳实践时，就是你考虑使用 **Redux** 的时候了。

Redux 的工作原理

没有接触过 **Redux** 的同学可能会问，“听说 **Redux** 特别复杂，我以前没有学过，能用一句话介绍一下 **Redux** 的工作原理吗？”

Redux 的核心原理，我们用一句话概括就是：**State 驱动 View 更新，用户操作 View 触发 Action，再通过 Action 来更新 State。**

如何理解这句话中的 **State**、**View** 和 **Action** 呢？

我用一个最简单的计数组件来给你举例子，示例代码如下：

 复制代码

```
1 function Counter() {
2   // State 状态
3   const [counter, setCounter] = useState(0)
4
5   // “Action ” 指令
6   const increment = () => {
7     setCounter(prevCounter => prevCounter + 1)
8   }
9
10  // View 视图
11  return (
12    <View>
13      <Text>Value: {counter} </Text>
14      <Text onPress={increment}>+1</Text>
15    </View>
16  )
17 }
```

使用 **useState** 管理应用状态时，有三个部分：**State**、**View** 和所谓的 “**Action**”。**State**、**View** 部分你已经非常熟悉了，**State** 是状态，**View** 是 **JSX** 创建的视图，但所谓的 “**Action**” 是什么呢？

“Action”并不难理解，在计数应用中，`increment` 函数可以理解为“函数形式的”状态更新指令。`increment` 函数内部调用的是状态更新函数 `setCounter`，用于更新视图 `View`。

使用 **Redux** 管理应用状态时，也有三部分：**State**、**View** 和 **Action**。State 是状态，View 是视图，Action 是更新指令。

不同的是，`useState` 管理的是组件状态，`Redux` 管理的是全局状态。

在 **Redux** 中，**State** 是一个存储在全局中的对象，用于描述整个应用的状态：

```
1 {  
2   counter: 0  
3 }
```

 复制代码

是的，没错。`Redux` 状态只是一个普通对象。如果你用 `Redux` 来实现计数应用，那么它的状态就是 `counter`，状态的默认值是 `0`。

`Redux` 的全局状态可以只有一个，也可以有多个，状态的值既可以是原始数据类型 `number`、`string` 等，也可以是更复杂的 `object`、`array` 数据类型。

我们再来看一个 `Todo` 应用的状态例子，它就有两个状态，并且状态的数据类型也更复杂：

```
1 {  
2   todos: [{  
3     text: 'Eat food',  
4     completed: true  
5   }, {  
6     text: 'Exercise',  
7     completed: false  
8   }],  
9   visibilityFilter: 'SHOW_COMPLETED'  
10 }
```

 复制代码

`Todo` 应用的两个全局状态分别是 `todos` 和 `visibilityFilter`。`todos` 状态是 `array` 嵌套 `object` 的复合数据类型，而 `visibilityFilter` 是 `string` 的基本数据类型。

Redux 中的 View 也就是 JSX 视图，我就不给你举例了。接着我们再来了解 Redux 中 Action 究竟是什么。

在 Redux 中，Action 是一个包含 type 字段的对象，用来描述“发生了什么事情”。我们以计数应用状态的 Action 为例：

```
1 { type: 'COUNT_INCREMENT' }
```

 复制代码

这里，计数应用“+1”的操作，被抽象成了一个 type 等于 COUNT_INCREMENT 的字符串指令。

但如果是一个更复杂的 Todo 应用，Todo 应用就要增加一个待办事项。待办事项的内容是用户输入的，这要怎么实现呢？Redux 是如何知道用户输入是“10 点买菜”，还是“12 点做饭”的待办事项呢？

这就需要有一个另外的字段来承载待办事项的信息了，一个 TODO 应用的 Action 示例如下：

```
1 { type: 'ADD_TODO', text: '10点买菜' }
2 { type: 'ADD_TODO', text: '12点做饭' }
3 { type: 'TOGGLE_TODO', index: 1 }
```

 复制代码

你看，这个示例中有 3 个 Action，前两个 Action 的 type 是 ADD_TODO，就是增加待办事项，text 是你自定义的参数名，其内容分别是用户输入的内容“10 点买菜”和“12 点做饭”。第三个 Action 就是标记待办事项是否完成的 Action，它的 type 是 TOGGLE_TODO，它的自定义参数名是 index，自定义参数值是数字 1。

像 text、index 这样字段，开发者可以自己定义。但有一些工具，比如后面要介绍的 Redux Toolkit，会统一将其定义为 payload 字段，省去了字段名不确定的麻烦事。

那当 Redux 收到 Action 通知后，如何更新全局状态呢？这就要用到 Reducer 了。

Reducer 就是一个普通的状态更新函数。Redux 会将当前应用的状态 State 和指令 Action 作为参数，传给 Reducer 函数，并接收一个 Reducer 返回的 newState 作为新全局状态。函数示意如下：

```
1 const reducer = (state, action) => newState
```

[复制代码](#)

在一个简单的计数应用中，Reducer 可以这样写：

```
1 function increment(state = 0, action) {
2   return state+1
3 }
4
5 function counterApp(state = {}, action) {
6   return {
7     counter: increment(state.counter, action),
8   }
9 }
```

[复制代码](#)

在上述代码中，counterApp 就是 Redux **管理所有全局状态的函数**，increment 函数就是你**处理分片状态的 Reducer 函数**。counterApp 函数接收当前状态 state 和发出的指令 action 作为参数，并且将这些参数传给 increment，由 increment 更新函数来更新状态。

看到这儿，你可能会想，为什么 Redux 处理状态时，既要有管理所有全局状态的函数。又有处理分片状态的 Reducer 函数，搞这么麻烦干嘛，直接用一个函数来处理不行吗？

不行，因为在大型应用中，用一个函数处理不过来。我们来看一个较为复杂的 TODO 应用的示例，你就明白了：

```
1 function filter(state = 'SHOW_ALL', action) {
2   if (action.type === 'SET_VISIBILITY_FILTER') {
3     return action.filter
4   } else {
5     return state
6   }
7 }
8
```

[复制代码](#)

```

9  function todos(state = [], action) {
10    switch (action.type) {
11      case 'ADD_TODO':
12        return state.concat([{ text: action.text, completed: false }])
13      case 'TOGGLE_TODO':
14        return state.map((todo, index) =>
15          action.index === index
16            ? { text: todo.text, completed: !todo.completed }
17            : todo
18        )
19      default:
20        return state
21    }
22  }
23
24  function todoApp(state = {}, action) {
25    return {
26      todosState: todos(state.todos, action),
27      filterState: filter(state.visibilityFilter, action)
28    }
29  }

```

这部分代码，包含三个函数，其中 `todoApp` 是 Redux 管理所有全局状态的函数，它管理了 `todosState` 和 `filterState` 两个全局状态。另外两个处理分片状态的 **Reducer 函数** `filter`、`todos` 分别处理了 `todosState` 和 `filterState` 全局状态的操作逻辑。

你看，在计数应用中全局状态只有一个，用一个函数来管理所有全局状态，用另一个 `reducer` 来管理分片状态，显得有点多余。但是在 `TodoApp` 中，有两个状态，而且它的状态处理逻辑比较复杂。

一个应用越复杂，管理状态的逻辑也就越复杂，代码量可不止几行、十几行，上百行、上千行也是可能的，用一个函数来处理所有状态是不现实的。所以我们既需要一个管理所有全局状态的函数，又需要若干个能够处理分片状态的 **Reducer 函数**。

这就是 Redux 既有管理所有全局状态的函数，又有处理不同分片状态的 **Reducer 函数** 的原因。

完整的流程是，在初始化时，**Redux 通过 Reducer 来初始化 State，State 驱动 View 渲染。**在更新状态时，用户操作 **View 触发 Action，Action 和当前 State 会被分发给处理分片状态的 Reducer 函数，由 Reducer 函数来执行更新逻辑和返回新的 State，并最终刷新 View。**这些就是 Redux 的核心原理。

Redux 的最佳实践

在了解完 Redux 的核心原理后，我们再学习 Redux 的最佳实践，就会轻松很多。

目前，Redux 社区和官方团队，经历多年的探索，慢慢摸索了一套最佳实践，并且把这套最佳实践封装到了 Redux Toolkit 工具集中。使用 Redux Toolkit 这套最佳实践，不仅能够避免常见的 Redux 使用误区，还能少写很多代码。

比如，在使用 [🔗 Redux Toolkit](#) 之后，Redux 官方提供的 Todo App 的代码中的 todosSlice.js 文件的代码行数，就由 [🔗 196 行](#)减少到 [🔗 131 行](#)，代码量减少了 33%。

Redux Toolkit 的本质是提供了一些**工具函数**，简化纯手写 Redux 代码的冗余逻辑，其中最重要的两个工具函数是：

- **configureStore**：管理所有全局状态的函数，它的返回值是一个 Store 对象；
- **createSlice**：管理分片全局状态的函数，其返回值是一个分片对象，该对象上最重要的两个属性是：
 - **actions**：创建分片 action 的函数集合；
 - **reducer**：已经创建好的分片 reducer。

另外，Redux 还有一个专门的 [🔗 React Redux 库](#)，这个库主要为 React/React Native 应用提供了 1 个组件和 2 个常用的钩子函数：

- **Provider**：Provider 是一个组件，该组件接收存储所有全局状态的 Store 对象作为参数。Provider 组件底层用的是 useContext，它为整个应用的其他组件提供获取 Store 对象的能力；
- **useSelector**：从 Store 中获取当前组件需要用到的状态；
- **useDispatch**：用于发送指令的钩子函数，其返回值是 dispatch 函数，而 dispatch 函数的入参是 action。

目前，使用 Redux 开发 React Native 应用的最佳实践，就是同时使用 Redux Toolkit 和 React Redux，实现全局状态管理。

接下来，我们还是以计数应用和 Todo 应用为例，学习如何使用 Redux Toolkit 和 React Redux。

首先，我们要引入相关的工具函数，示例代码如下：

 复制代码

```
1 import {configureStore, createSlice } from '@reduxjs/toolkit';
2 import {Provider, useSelector, useDispatch } from 'react-redux';
```

这里，我们分别从 '@reduxjs/toolkit' 和 'react-redux' 中引入了 configureStore、createSlice、Provider、useSelector、useDispatch。

那如何使用它们来管理应用状态呢？整个过程大致分为 5 步：

1. 使用 Provider 组件向应用的其他组件提供获取 Store 的能力；
2. 使用 configureStore 函数创建 Store；
3. 使用 createSlice 函数创建分片；
4. 使用 useSelector 获取分片 State；
5. 使用 useDispatch 生成的 dispatch 来发送 action。

首先我们来看如何使用 Provider 组件向其他组件提供获取 Store 的能力。

一般而言，Provider 组件通常是最顶层的组件，它包裹住了整个应用。以计数应用为例，示例代码如下：

 复制代码

```
1 const store = ...
2
3 // 计数应用
4 function CounterApp() {}
5
6 // 根组件
7 export default function Root() {
8   return (
9     <Provider store={store}>
10       <CounterApp />
11     </Provider>

```

```
12     );  
13 }  
14  
15 AppRegistry.registerComponent('App', () => Root);
```

你看，Root 是整个应用的根组件，这个组件使用了 Provider 包裹住代表计数应用的 CounterApp 组件，并且 Provider 接收了 store 作为参数，因此 CounterApp 组件及其所有的子组件都能从 store 中获取任意的全局状态。

那存储全局状态的对象 store 是如何创造的呢？这就到了第二步，创建 store 用到的函数是 **configureStore**。

使用 configureStore 函数创建 store 的方法如下：

 复制代码

```
1 const counterSlice = ...  
2  
3 const store = configureStore({  
4   reducer: {  
5     counter: counterSlice.reducer,  
6   }  
7 });  
8  
9 console.log(store.getState()) // { counter }
```

在这段代码中，我使用了 configureStore 函数来创建 store 对象。configureStore 函数的入参中最关键的对象属性是 reducer 属性。在计数应用中，configureStore 入参的 reducer 的结构是 { counter }，那么 configureStore 返回值 store 对象的结构也是 { counter }。

也就是说，计数应用存在一个全局状态 counter。

你也可以根据不同的情况，创建不同的 store 对象。比如 Todo 应用你可以这么创建：

 复制代码

```
1 const todosSlice = ...  
2 const filtersSlice = ...  
3  
4 const store = configureStore({  
5   reducer: {
```

```

6   todosState: todosSlice.reducer,
7   filterState: filtersSlice.reducer,
8 },
9 })
10
11 console.log(store.getState()) // { todos: filters }

```

在 Todo 应用中，configureStore 入参的 reducer 的结构是 { todos, filters }，那么 store 对象的结构也是 { todos, filters }，也就是说，Todo 应用存在两个全局状态，分别是 todos 和 filters。

在 Redux 中，分片 state 是通过分片 reducer 生成的。在计数应用中，分片是 counterSlice，分片状态 counter 是通过 counterSlice.reducer 创建的；在 Todo 应用中，分片是 todosSlice 和 filtersSlice，分片状态 todosState 和 filterState 是通过 todosSlice.reducer 和 filtersSlice.reducer 创建的。

因此，接下来你需要关注的是如何创建分片。那么如何创建分片呢？**这就是第三步的任务，创建分片用到的函数是 createSlice。**

使用 createSlice 函数创建分片的示例代码如下：

 复制代码

```

1  const initialState = {
2    value: 0,
3  };
4
5  const counterSlice = createSlice({
6    name: 'counter',
7    initialState,
8    reducers: {
9      increment: (state, action) => {
10        // 可以直接修改的“状态”
11        state.value += action.payload;
12      },
13    },
14  });
15
16  console.log(counterSlice.actions.increment(1))
17  // {"payload": 1, "type": "counter/increment"}
18
19  console.log(counterSlice.reducer)
20  /*
21  function reducer(state, action) {
22    if (!_reducer) _reducer = buildReducer();
23    return _reducer(state, action);

```

```
24 }  
25 */
```

创建分片需要三个参数：

- **name**：分片的名字，字符串类型。示例中是“counter”字符串；
- **initialState**：分片的初始化状态。示例中是 { value: 0 } 的对象；
- **reducers**：对象类型，用于创建分片 action 和分片 reducer。

createSlice 函数会自动生成分片 action creators 和分片 reducer，并将这两个自动生成的值挂在其返回对象 counterSlice 的 actions 属性和 reducer 属性上。

那怎样验证 createSlice 函数确实帮你创建了分片 action creators 和分片 reducer 呢？

你可以打印 counterSlice.actions.increment(1) 来检验。actions.increment 是创建 increment action 的函数，打印的结果是 { “type”: “counter/increment”, “payload”: 1 }。其中，“type” 字段是分片 action 的必传字段，“type” 的值由分片名字 “counter” 和 reducers 对象的键名 “increment” 共同组成。“payload” 字段是专门用于传参的字段，其参数来源于 actions.increment 创建函数的第一个入参，也就是数字 1。

你也可以打印 counterSlice.reducer，它内部是一个由 Redux 帮我们生成的 _reducer 函数。

在完成以上三步后，应用中的组件就能够使用分片 State 和分片 Action 来展示和改变 UI 视图了。

那接着新的问题就来了：组件该如何获取和使用分片 State 和分片 Action 呢？

这就是最后第四步和第五步的任务，这两步都是在具体组件中进行的，我们放在一起看就好，组件的示例代码如下：

 复制代码

```
1 function CounterApp() {  
2   const counter = useSelector((state) => state.counter.value);  
3  
4   const dispatch = useDispatch();  
5
```

```

6     const handlePress = () => {
7         dispatch(counterSlice.actions.increment(10));
8     };
9
10    return (
11        <View style={styles.box}>
12            <Text>{counter}</Text>
13            <Text onPress={handlePress}>+10</Text>
14        </View>
15    );
16 }

```

在 CounterApp 组件中，我们首先使用 `useSelector` 获取状态，然后再使用 `useDispatch` 获取 `dispatch` 函数，并且在对应时机调用 `dispatch(action)` 发送指令。

其中，**`useSelector`** 的主要作用是**按需获取状态**。虽然，你也可以直接使用 `store.getState()` 获取所有状态，但获取所有全局状态会有一个弊端，那就是只要任何一个全局状态发生了改变，该组件就会 **re-render**，这容易导致应用性能变差。因此，你需要 `useSelector` 帮你从 `store` 中按需获取状态。

`useSelector` 的入参是一个函数，你可以通过这个函数从所有 `state` 中选择该组件中用到的 `state`。在这个计数组件中，只用到了 `state.counter.value`，因此这个函数只用返回 `state.counter.value` 作为状态就可以了。

因此，在上述计数应用的初始化时，默认的计数值是 0，每当你点击一次“+10”按钮时，就会触发 `action` 指令，将计数值“+10”。

我把完整的代码放在了这里，你可以仔细看下：

 复制代码

```

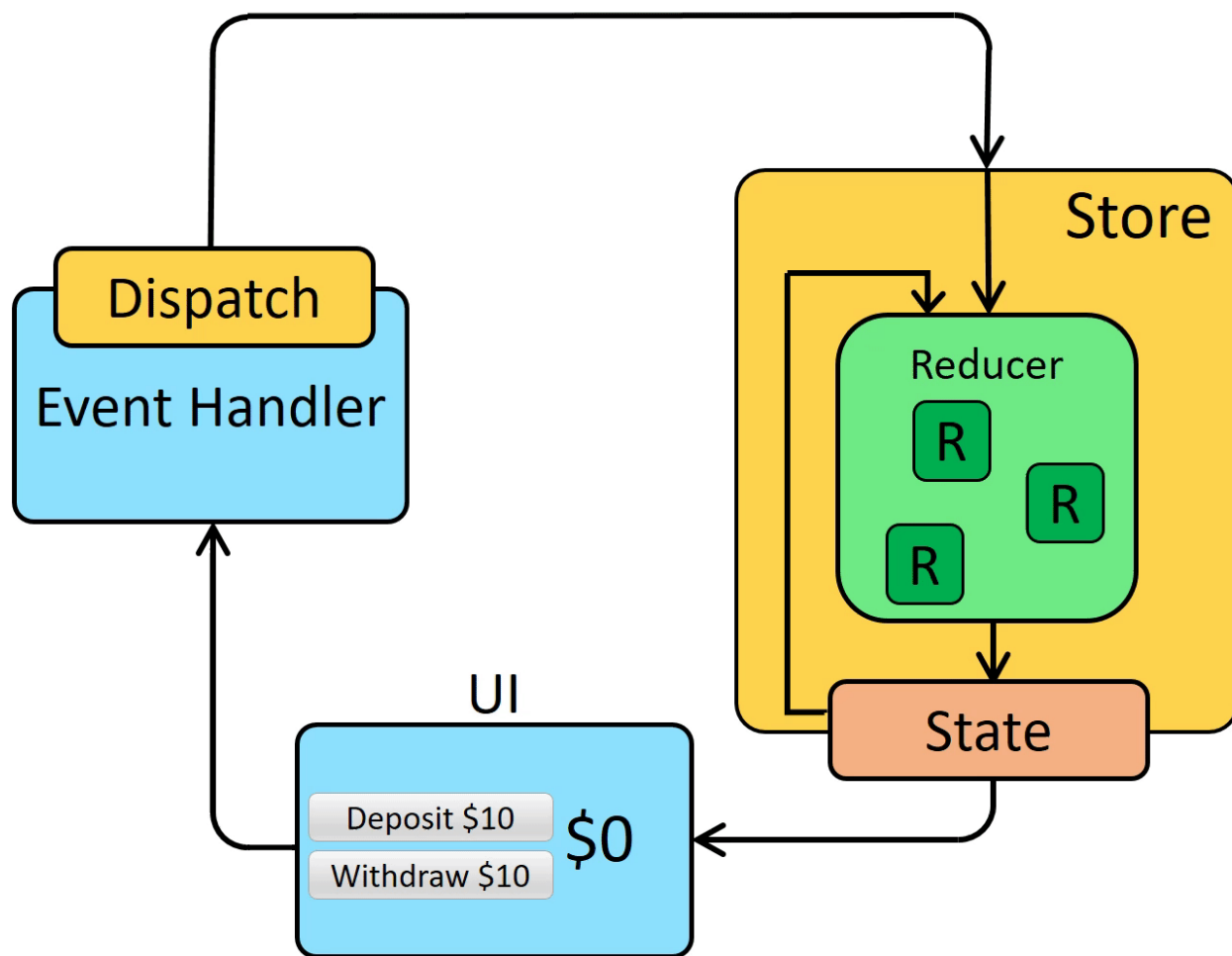
1  import React from 'react';
2  import {View, Text, StyleSheet} from 'react-native';
3  import {
4      useSelector,
5      useDispatch,
6      Provider,
7      TypedUseSelectorHook,
8  } from 'react-redux';
9  import {configureStore, createSlice, PayloadAction} from '@reduxjs/toolkit';
10
11  interface CounterState {
12      value: number;

```

```
13 }
14
15 const initialState: CounterState = {
16   value: 0,
17 };
18
19 const counterSlice = createSlice({
20   name: 'counter',
21   initialState,
22   // Reducer
23   reducers: {
24     increment: (state, action) => {
25       state.value += action.payload;
26     },
27   },
28 });
29
30 // Store
31 const store = configureStore({
32   reducer: {
33     counter: counterSlice.reducer,
34   },
35 });
36
37 function CounterApp() {
38   // State
39   const counter = useSelector((state) => state.counter.value);
40
41   const dispatch = useDispatch();
42
43   // Event Handler
44   const handlePress = () => {
45     // Action
46     const action = counterSlice.actions.increment(10);
47     // dispatch
48     dispatch(action);
49   };
50
51   // View/UI
52   return (
53     <View >
54       <Text>{counter}</Text>
55       {/* click event: deposit */}
56       <Text onPress={handlePress}>+10</Text>
57     </View>
58   );
59 }
60
61 export default function Root() {
62   return (
63     <Provider store={store}>
64       <CounterApp />
```

```
65     </Provider>
66   );
67
```

在这段完整代码中，我帮你把 Redux 中的关键概念都标记出来了。Redux 中的概念比较多，包括 Store、Reducer、State、UI、Event Handle、Dispatch、Action。在看计数应用的实现代码的同时，你也可以结合 Redux 官方提供的原理图，把使用 Redux 的最佳实现和原理结合在一起理解：



附加材料

- 三个官网：[Redux](#)、[Redux Toolkit](#)、[React Redux](#)。
- 这一讲提供的是简版的最佳实践，官网完整的最佳实践见 [《Redux Style Guide》](#)。
- 简版最佳实现的计数应用 [Demo](#)。
- 完整最佳实践的计数应用 [Demo](#)。
- 完整最佳实践的 Todo 应用 [Demo](#)。

总结

这一讲，我和你介绍了三个重点，分别是 **Redux** 什么时候用、**Redux** 的工作原理和 **Redux** 的最佳实践。

不过，**Redux** 并不是万能药，它只适合复杂或大型应用的全局状态管理，在简单或小型应用中使用 **Redux**，就像大炮打苍蝇大材小用。

虽然使用 **Redux** 来管理状态需要 5 个步骤，比使用 **useState** 的 3 个步骤多了两个步骤。但使用 **Redux** 来管理全局状态有很多好处，比如它能提高你应用的可测试性和可维护性，而这些特性正是复杂项目、大型项目所需要的。


但对于一个大型项目而言，**Redux** 并不是唯一状态管理方案，通常我还会将 **Redux** 和 **useState**、**ReactQuery** 搭配起来使用。我更喜欢使用 **useState** 来管理组件状态，使用 **Redux** 来管理全局状态，使用 **ReactQuery** 来管理异步状态。

作业

React/React Native 生态中，状态管理工具可谓是百家争鸣，那你选择的方案是什么呢？欢迎在评论区和我分享你的答案。我们下一讲见。

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | Navigation：页面之间怎么跳转？

下一篇 20 | Sentry：线上错误与性能监控怎么处理？

精选留言 (1)

写留言



python4

2022-05-14

函数命名上加一点redux概念是否更好, 方便新手牢记概念, 比如: filter -> filterReducer

