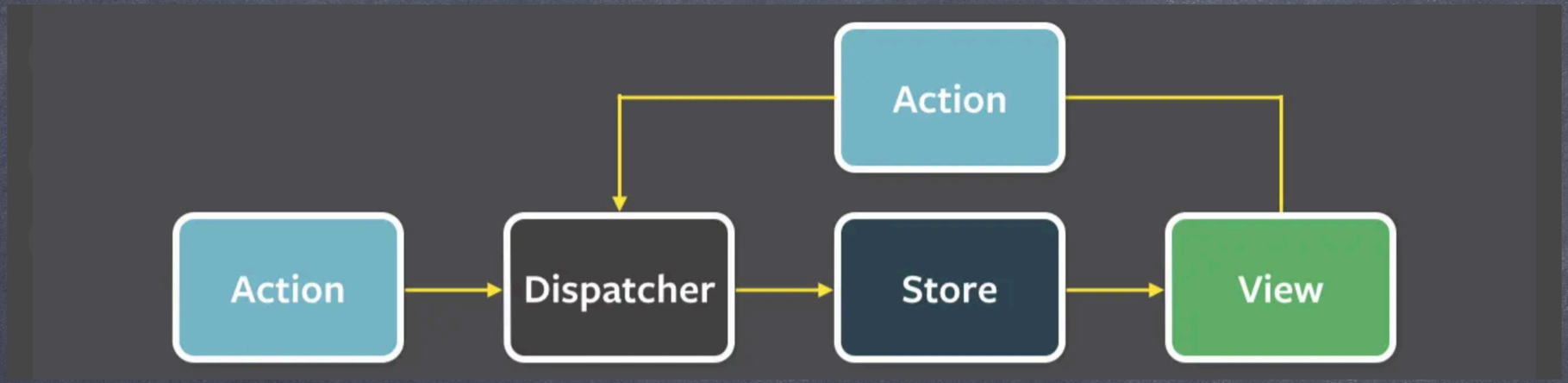
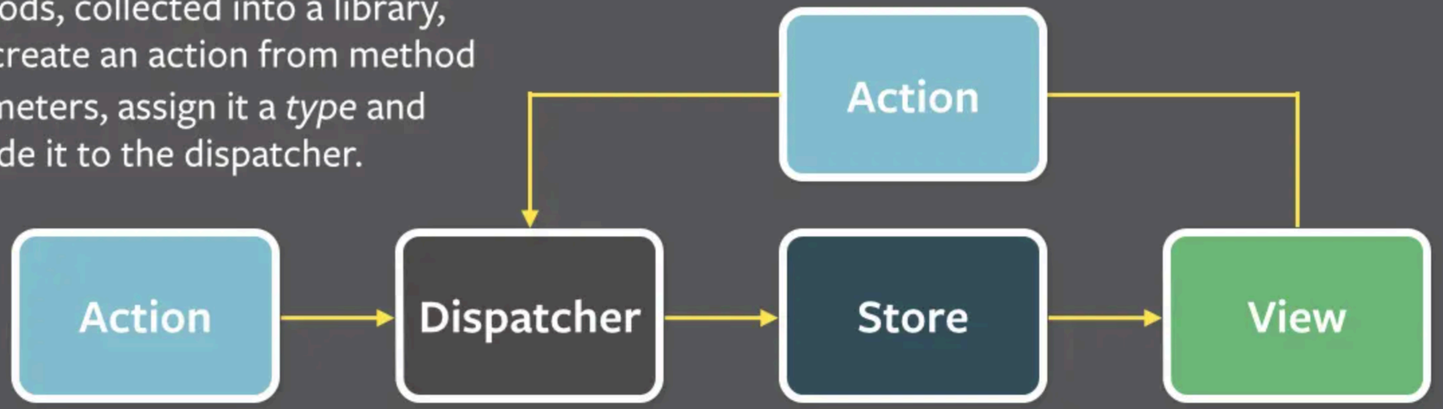
React

flux

- UI产生动作消息，将动作传递给分发器
- 分发器广播给所有store
- 订阅的store做出反应，传递新的state给UI


Action
Action
Dispatcher
Store
View

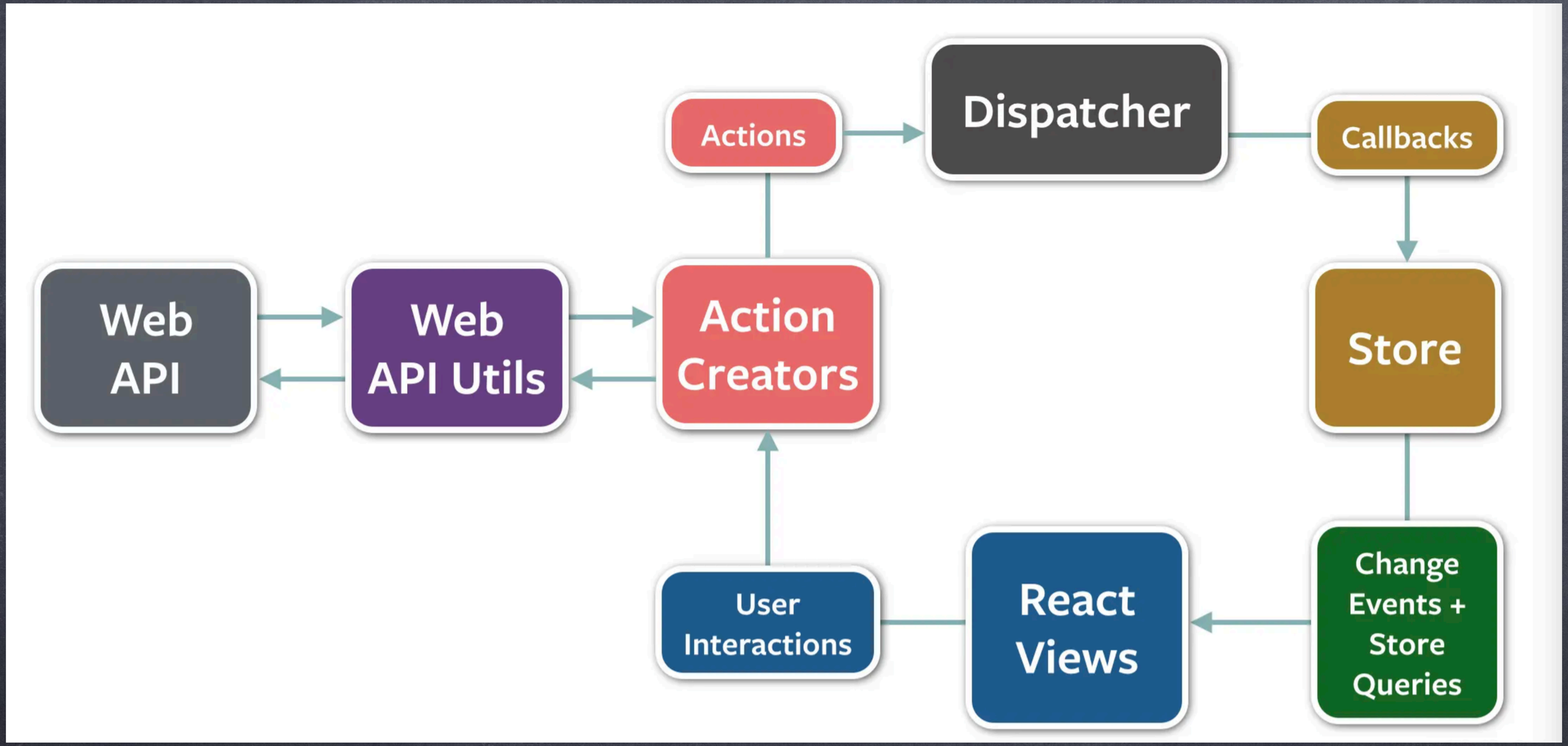
Action creators are helper methods, collected into a library, that create an action from method parameters, assign it a *type* and provide it to the dispatcher.



Every action is sent to all stores via the *callbacks* the stores register with the dispatcher.

After stores update themselves in response to an action, they emit a *change* event.

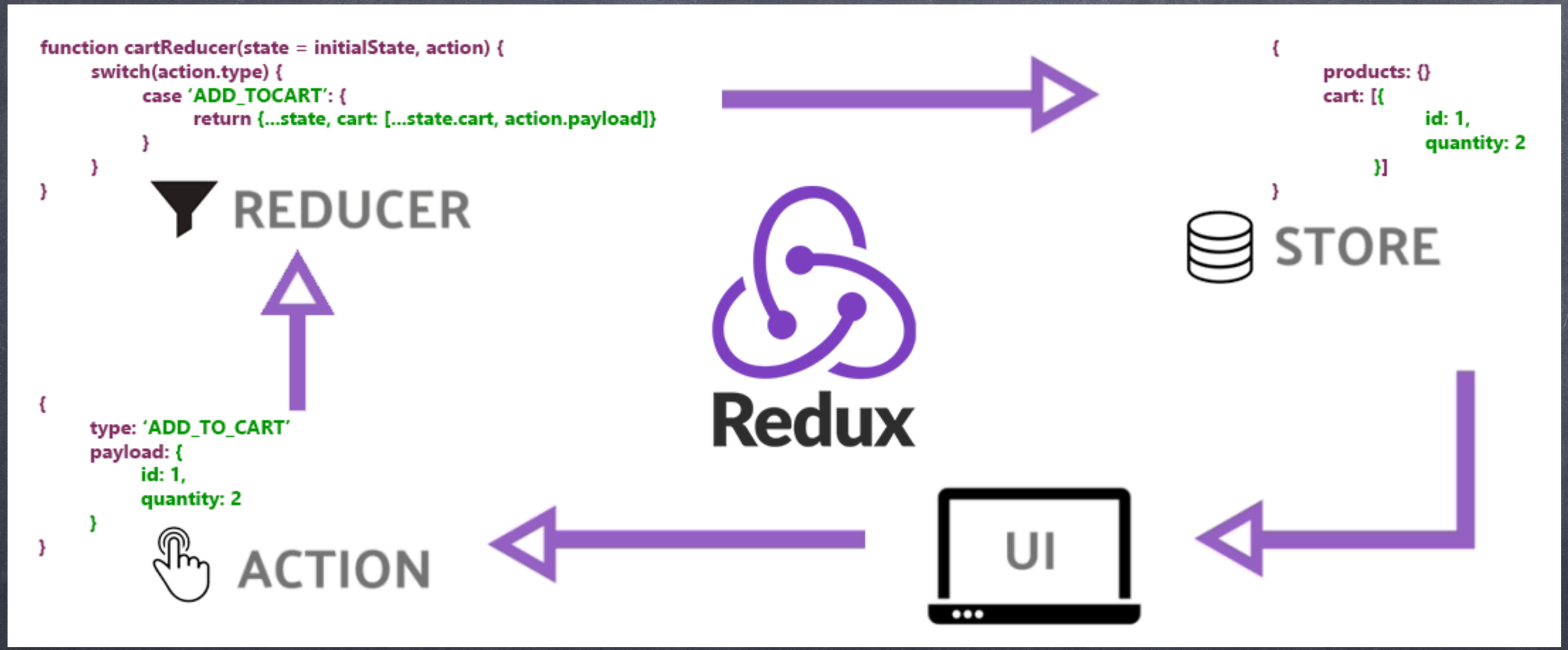
Special views called *controller-views*, listen for *change* events, retrieve the new data from the stores and provide the new data to the entire tree of their child views.

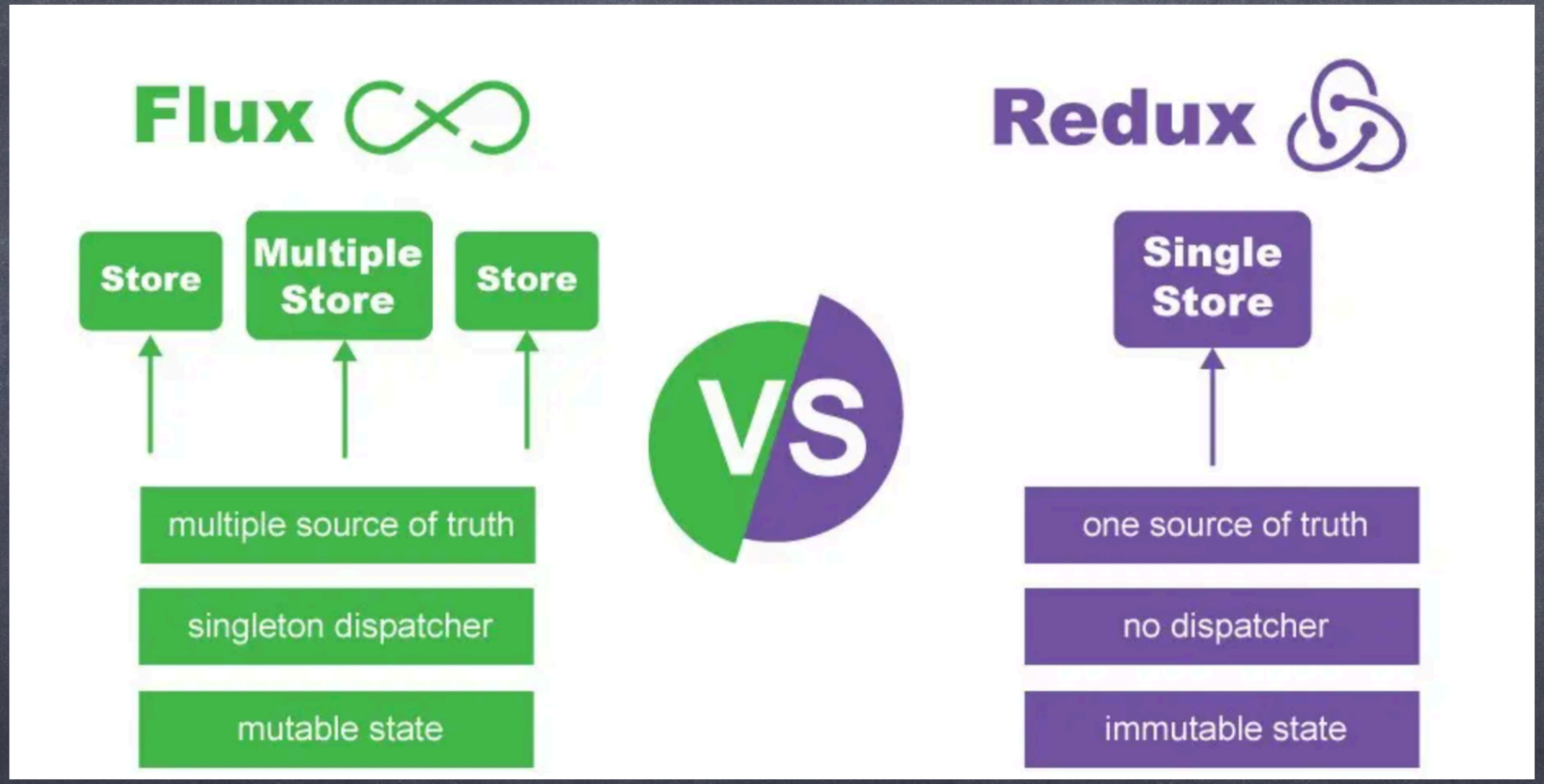

Actions
Dispatcher
Callbacks
Web API
Web API Utils
Action Creators
Store
User Interactions
React Views
Change Events + Store Queries

redux

- 单一数据源，整个应用state存储在一个单一store树中
- State状态为只读，不应该直接修改state，而是通过action触发state修改
- 使用纯函数进行状态修改，需要开发者书写reducers纯函数进行处理，reducer通过当前状态树和action进行计算，返回一个新的state

- Redux并没有 dispatcher。它依赖纯函数来替代事件处理器，也不需要额外的实体来管理它们。Flux尝尝被表述为：`(state, action) => state`，而纯函数也是实现了这一思想。
- Redux为不可变数据集。在每次Action请求触发以后，Redux都会生成一个新的对象来更新State，而不是在当前状态上进行更改。
- Redux有且只有一个Store对象。它的Store储存了整个应用程序的State。


function cartReducer(state = initialState, action) {
switch(action.type) {
case 'ADD_TOCART': {
return {...state, cart: [...state.cart, action.payload]}
}
}
}
REDUCER
{
products: {}
cart: [{
id: 1,
quantity: 2
}]
}
STORE
Redux
{
type: 'ADD_TO_CART'
payload: {
id: 1,
quantity: 2
}
}
ACTION
UI


Flux
Store
Multiple Store
Store
multiple source of truth
singleton dispatcher
mutable state
VS
Redux
Single Store
one source of truth
no dispatcher
immutable state

index.js

Copy

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
import store from './app/store'
import { Provider } from 'react-redux'

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

features/counter/Counter.js

```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { decrement, increment } from './counterSlice'
import styles from './Counter.module.css'

export function Counter() {
  const count = useSelector((state) => state.counter.value)
  const dispatch = useDispatch()

  return (
    <div>
      <div>
        <button
          aria-label="Increment value"
          onClick={() => dispatch(increment())}
        >
          Increment
        </button>
        <span>{count}</span>
        <button
          aria-label="Decrement value"
          onClick={() => dispatch(decrement())}
        >
          Decrement
        </button>
      </div>
    </div>
  )
}
```

features/counter/counterSlice.js

Copy

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

export const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0,
  },
  reducers: {
    increment: (state) => {
      // Redux Toolkit allows us to write "mutating" logic in reducers. It
      // doesn't actually mutate the state because it uses the Immer library,
      // which detects changes to a "draft state" and produces a brand new
      // immutable state based off those changes
      state.value += 1
    },
    decrement: (state) => {
      state.value -= 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload
    },
  },
})

// Action creators are generated for each case reducer function
export const { increment, decrement, incrementByAmount } = counterSlice.actions

export default counterSlice.reducer
```

mobx

- 定义状态并使其可观察
- 创建视图以响应状态变化
- 更改状态（自动响应UI变化）

MobX 支持单向数据流，也就是**动作**改变**状态**，而状态的改变会更新所有受影响的**视图**。

当**状态**改变时，所有**衍生**都会进行**原子级的自动**更新。因此永远不可能观察到中间值。

所有**衍生**默认都是**同步**更新。这意味着例如**动作**可以在改变**状态**之后直接可以安全地检查计算值。

计算值 是**延迟**更新的。任何不在使用状态的计算值将不会更新，直到需要它进行副作用（I / O）操作时。 如果视图不再使用，那么它会自动被垃圾回收。

所有的**计算值**都应该是**纯净**的。它们不应该用来改变**状态**。

redux: https://react-redux.js.org/

mobx: https://cn.mobx.js.org/

感谢

「狗哥」