

# The Differences Of Redux and MobX

Min Zhou  
NetID: mz22

## Introduction

Both Redux and MobX are state management libraries. Most of the time we use them to help us develop web application with React. You can also substitute React with other frameworks like Angular, Vue etc. But in this paper I will only talk about how they work with React. Since we have learned a lot about Redux, I will focus more on the usage of MobX and his differences with Redux.

## Main things you need to do by using Redux and MobX

Using Redux:

- (1) make a pure reducer function taking the state and action as input, it can reduce a new state based on the action's type.
- (2) create a store object using the reducer as input, the object has three methods—`getState()`, `dispatch()` and `subscribe()`;
- (3) subscribe an action to the store, usually the action is `render()` function when you use react with redux.
- (4) Where you need to change the state, dispatch an action with a specific type to store, the dispatch function would call the reducer and create a new state. Then it would call the function you subscribe, if it is `render()` function, then the component would re-render.

Using MobX:

- (1) make an observable state, using the `@observable` decorator or observable functions to make the state trackable for MobX.
- (2) Using `@computed` decorator to create functions which can derive data from state automatically
- (3) Using `autorun` function, whenever the observable state changes, this function would run automatically.
- (4) When you use MobX with React, Use `@observer` decorator from the `mobx-react` package to wrap the render function into `autorun`, Son when the state is changed, the component would re-render[1].

## One store VS Multiple Stores

Although we can choose to use multiple stores in Redux, most of the time we still only use one global store to keep all state. We use multiple reduces to store different state for different components. However in MobX, you can choose how many stores you want. You can create multiple stores for multiple states or just create one store and put all state in it. The way to create store is different. For Redux, you need to use `createStore` build-in method and use reducer as its arguments. For MobX, you just declare it by your self.

```
// Store for MobX
class ObservableTodoStore {
  @observable todos = [];
}
const observableTodoStore = new ObservableTodoStore();

// Store for Redux
const store = createStore(Reducer, {});
```

## State

Redux and MobX are both state management libraries. State are really important for them. For Redux, states are Read-only. Because redux can only detect a change when the position of state in memory changes. So when the state changes, we do not change it directly, we make a brand-new object based on the old object. So the reducer is pure function. However in MobX, states can be read and wrote. We change the state directly, It is impure.

```
// State Change for MobX
class UserStore {
  @observable users = [];

  @action addUser = (user) => {
    this.users.push(user);
  }
}

//State Change for Redux
(users, action) => return [...users, action.addUser]
```

Secondly, In Redux, the state is plain javascript objects. In MobX, the data in state is Observable data. This give your objects observable capabilities which enables you to track changes inside the objects while you do not need to create a new object. This is smarter than Redux. The MobX got this ability by recording any observable property that is dereferenced. In Redux, we track the state all by ourself while in MobX, it is more implicit and magical.

```
// State data of MobX
let article = observable({
  id: 123,
  text: "article"
})
class store = {
  @observable article;
}

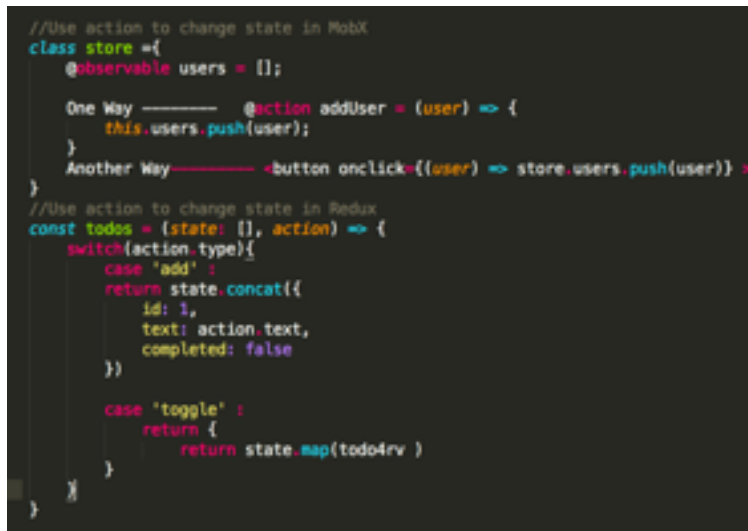
//State data of Redux
const state = {
  id: 123,
  text: "article"
}
```

Thirdly, In Redux, we are recommended to use normalized state instead of nested state. One important reason is that the redux detect the position of object in memory to decide if it is changed. So if you use nested state, an update to deeply nested data object can cause totally unrelated UI component to re-render even if the data they are displaying hasn't actually changed[2]. In MobX, you can use nested state and keep data denormalized. We can do that

because of the concept of computed value and we just keep track of the value we care about and do not need to make a new object to re-render the whole UI-component.

## Action

When you use Redux, you have to use actions which contains a type. That is the only way redux change the state. When there is an event, we use the `store.dispatch` to dispatch an action to store. Then we call the reducer and get the new state based on the type of action and value in the action object. Then we can re-render the UI component based on the changes. In MobX, we can choose to use an action to change state or just simply do a reassignment of the state value and it will also be fine.



```
//Use action to change state in MobX
class store = {
  @observable users = [];

  One Way ----- @action addUser = (user) => {
    this.users.push(user);
  }
  Another Way ----- <button onclick={() => store.users.push(user)} >
}

//Use action to change state in Redux
const todos = (state: [], action) => {
  switch(action.type){
    case 'add':
      return state.concat([
        id: 1,
        text: action.text,
        completed: false
      ])

    case 'toggle':
      return {
        return state.map(todo => {

```

## dispatch and subscribe VS autorun

In Redux, we need a reducer which is a function take state and action as argument. It returns a new state based on the action's type. When we need to create a store, we need to set the reducer as function `createStore`'s function. So how redux change the state? First we subscribe an event listener to the store. This event listener function would call `getState` to get the newest state in the store. When we dispatch an action to the store, the store would call the reducer and get the new state. Then the store call the event listener we subscribed before and get the new state. In MobX, first we got an observable state, and MobX would keep track of every references of the state. Then we set an action function which would change the state directly, once the observable state is changed, the `autorun` function would run automatically. This can combine perfectly with react re-render.

## Combine with React

How dose redux combine with react? The most important part is the `subscribe` function of store we created. As shown above, the `subscribe` function is to subscribe an event listener to the store. Then every time you dispatch an action, the event listener will be called as well. So for react, we can treat the render function of every component as event listener. So every time the state is changed, the specific component would re-render at the same time.

In MobX, it is easier to connect with react. We just use @observer decorator from mobx-react package and it will wrap the whole React component render method in autorun, automatically keeping your components in sync with the state[3]. As we talk about above, when the observable state is changed, the MobX would run automatically. This is the main reason why MobX can combine perfectly with react.

```
// How Redux work with React

const Counter = ({value,onIncrement,onDecrement}) => {
  return (
    <div>{value}</div>
    <button onClick={onIncrement}>+</button>
    <button onClick={onDecrement}>-</button>
  )
}

const render = () => {
  ReactDOM.render(
    <Counter value={store.getState(),onIncrement=
      () => store.dispatch({type: 'add'})} />,
    document.getElementById('root')
  )
}

const store = createStore(Reducer);
store.subscribe(render);
```

```
// How MobX works with React
import {observer} from 'mobx-react';
import {observable} from 'mobx';

var appState = observable({
  timer: 0,
});

@observer
class TimerView extends React.Component {
  render() {
    return (<button onClick={this.onReset.bind(this)}>
      Seconds passed: {this.props.appState.timer}
    </button>);
  }

  onReset () {
    this.props.appState.timer = 1;
  }
}

React.render(<TimerView appState={appState} />, document.body);
```

## Who is better?

MobX: It is easier for me to understand the important concept when I learn it for the first time. Because I am more familiar with java, MobX is more object-oriented. In addition, MobX has less hard word to remember and programmer did less things when use it. We can change the state directly with or without using action. So there is less things we need to do when we need to manipulate the state. We do not need a reducer, we do not need to dispatch the action in order to change the state. It is just done magically.

Redux: redux is more explicit. Programmer know what they are doing for most of the time. So it is easier to debug. The reducer is pure function and anytime we need to change the state, we need to do make a action with specific type. This would stop us from doing stupid things especially in large project and make it easier for us to test our code.

## References

- [1] <http://cn.mobx.js.org/refguide/action.html>
- [2] <http://redux.js.org/docs/recipes/reducers/NormalizingStateShape.html>
- [3] <https://www.robinwieruch.de/redux-mobx-confusion/>

