
React状态管理

什么是 "状态"?

状态就是 UI 中的动态数据

现代前端框架的意义，就是问题解决思路的革新，
把对 "过程" 的各种命令，变为了对 "状态" 的描述。

通过 js 修改视图

```
document.getElementById("city").innerHTML = "chongqing";
```

通过 react 修改视图

```
setCity("chongqing");
```

通过 vue 修改视图

```
this.city = "chongqing";
```

React 中的状态

State

状态如何传递?

场景	状态
父子	props
跨组件	Context

context的问题

Context 其实相当于 "状态提升"，并没有额外的性能优化，且写起来比较啰嗦。 为优化性能，一般会添加多个 Context，写起来就更啰嗦。在项目没那么复杂时，还不如层层传递简单。

```
const MyContext = React.createContext(defaultValue);
function App() {
  return (
    <Context1.Provider>
      <Context2.Provider>
        <Context3.Provider>
          <div className="App">
            <Header />
            <Main />
            <Footer />
          </div>
        </Context3.Provider>
      </Context2.Provider>
    </Context1.Provider>
  );
}
```

什么是 "状态管理"?

"状态管理" 就是为了解决组件间的 "跨级" 通信。

class时代的状态管理

Redux 与 MobX

redux

actionType.js

```
const CHANGENAME= 'CHANGENAME'
```

actions.js

```
const changeNameCreater =
(name) => ({type:CHANGENAME,data:name})
```

reducer.js

```
const initialState = { name:'tutu'}
function modifyName(state=initialState,action){
    switch(action.type){
        case CHANGENAME:
            return action.data
        default:
            return state
    }
}
```

store.js

```
const store = createStore(modifyName);
```

component

```
 connect(state => ({ lastname:modifyName.name}),
    {changeNameCreater}
)(Component)
```

redux的弊端

Redux 的利弊已讨论太多，简单来说，开发者关心的是 "使用"，而 Redux 关心的是 "哲学"。

之前开玩笑说，其实 Redux 用一行代码就可以表示，却写出了论文规格昏昏欲睡的文档：

```
createStore = (reducer, state) => ({ dispatch: (action) => (state = reducer(state, action)) });
```

mobx

store

```
class addressStore {
  @observable modalVisible = false;
  @computed get validationAddress() {
    return this.selectListValidationOption === 'originalAddress'
  }
  @action.bound
  setModalVisible(data) {
    this.modalVisible = data;
  }
}
```

component

```
@inject('loginStore')
@seoHoc('Account pet')
@observer
class Pet extends React.Component {}
```

mobx

"不够 React"，但用起来简单。

redux对比mobx

Mobx和Redux的目标都是管理好应用状态,但是最根本的区别在于对数据的处理方式不同。

Redux认为，数据的一致性很重要，为了保持数据的一致性，要求Store 中的数据尽量范式化，也就是减少-切不必要的冗余, 为了限制对数据的修改,要求Store中数据是不可改的(Immutable) 只能 通过action触发reducer来更新Store。

Mobx也认为数据的一致性很重要， 但是它认为解决问题的根本方法不是让数据范式化，而是不要给机 会让数据变得不一致。所以，Mobx鼓励数据干脆就“反范式化”， 有冗余没问题，只要所有数据之间 保持联动，改了一处，对应依赖这处的数据自动更新，那就不会发生数据不一致的问题。

我们真的需要redux、mobx吗?

新时代的状态管理库

zustand

```
const useStore = create(set => ({
  bears: 0,
  increasePopulation: () => set(state => ({ bears: state.bears + 1 })),
  removeAllBears: () => set({ bears: 0 })
}))
```

```
function BearCounter() {
  const bears = useStore(state => state.bears)
  return <h1>{bears} around here ...</h1>
}
```

新时代的状态管理.md

valtio

```
const state = proxy({ count: 0, text: 'hello' })
```

```
setInterval(() => {
  ++state.count
}, 1000)
```

```
function Counter() {
  const snap = useSnapshot(state)
  return (
    <div>
      {snap.count}
      <button onClick={() => ++state.count}>+1</button>
    </div>
  )
}
```


react组件设计

聪明组件

```
const RandomJoke = () => {
  const { data: joke } = useRequest();
  return <Joke value={joke} />;
};
```

傻瓜组件

```
import SmileFace from "./yaoming_ simile.png";
const Joke = ({ value }) => {
  return (
    <div>
      <img src={SmileFace} />
      {value | " loading..."}
    </div>
  );
};
```

高阶组件

```
const withDoNothing = (Component) => {
  const NewComponent = (props) => {
    return <Component {...props} />;
  };
  return NewComponent;
};
```

1.高阶组件不能去修改作为参数的组件，高阶组件必须是一个纯函数，不应该有任何副作用。
2.高阶组件返回的结果必须是一一个新的React组件,这个新的组件的JSX部分肯定会包含作为参数 的组件。
3.高阶组件一般需要把传给自己的props转手传递给作为参数的组件。

高阶组件的应用

实现一个只有在登录才显示的功能组件

```
const LogoutButton = () => {
  if (getUserId()) {
    return "退出登录";
  } else {
    return nu1l;
  }
};
```

```
const ShoppintCart = () => {
  if (getUserId()) {
    return "退出登录";
  } else {
    return nu1l;
  }
};
```

```
const withLogin = (Component) => {
  const NewComponent = (props) => {
    if (getUserId()) {
      return <Component {...props} />;
    } else {
      return null;
    }
  };
  return NewComponent;
};
```

```
const LogoutButton = withLogin((props) => {
  return "退出登录"; //显示“的JSX
});
const ShoppingCart = withLogin(() => {
  return "购物车";
});
```

高阶组件的高级用法

```
const withLoginAndLogout = (ComponentForLogin, ComponentForLogout) => {
  const NewComponent = (props) => {
    if (getUserId()) {
      return <ComponentForLogin {...props} />;
    } else {
      return <ComponentForLogout {...props} />;
    }
  };
  return NewComponent;
};
```

```
const TopButtons = withLoginAndLogout(LogoutButton, LoginButton);
```

链式调用高阶组件

```
const X1 = withOne(X);
const X2 = withTwo(X1);
const X3 = withThree(X2);
const SuperX = X3; //最终的SuperX具备三个高阶组件的超能力
```

renderProps

```
const Renderall = (props) => {
  return <>{props.children(props)}</>;
};
```

```
<RenderAll>
  {() => <h1>hel1o world</h1>}
</RenderAll>
```

renderProps的应用

```
const Login = (props) => {
  const userName = getUserName();
  if (userName) {
    const allProps = { userName, ...props };
    return <>{props.children(allProps)}</>;
  } else {
    return null;
  }
};
<Login>
{({userName}) => <h1>Hello {userName}</h1>}
</Login>
```

renderProps的应用

```
const Auth = (props) => {
  const userName = getUserName();
  if (userName) {
    const allProps = { userName, ...props };
    return <React.Fragment>{props.login(allProps)}</React.Fragment>;
  } else {
    <React.Fragment>{props.nologin(props)}</React.Fragment>;
  }
};
<Auth
  login={({ userName }) => <h1>Hello {userName}</h1>}
  nologin={() => <h1>Please login</h1>}
/>;
```

renderProps的优势

```
const withLogin = (Component) => {
  const NewComponent = (props) => {
    const userName = getUserName();
    if (userName) {
      return <Component {...props} userName={userName} />;
    } else {
      return null;
    }
  };
  return NewComponent;
};
```

用hoc实现的该组件要求组件必须由于userName才能正常显示

```
<Login>
  {(props) => {
    const { userName } = props;
    return <TheComponent {...props} name={userName} />;
  }}
</Login>;
```

组合组件

```
<Tab>
  <TabItem active={true} onClick={onClick}>One</TabItem>
  <TabItem active={false} onClick={onClick}>Two</TabItem>
  <TabItem active={false} onClick={onClick}>Three</TabItem>
</Tab>
```

如何让子组件自动获取到active?

组合组件

```
const Tabs = () => {
  const [activeIndex, setActiveIndex] = useState(0);
  return React.Children.map(this.props.children, (child, index) => {
    if (child.type) {
      return React.cloneElement(child, {
        active: activeIndex === index,
        onClick: () => setActiveIndex({ activeIndex: index }),
      });
    } else {
      return child;
    }
  });
};
```

antd大量使用这两个api来组合组件