理解了状态管理，就理解了前端开发的核心

原创 神说要有光 神光的编程秘籍 2022-05-03 00:00

收录于合集
#前端框架 15 #状态管理 2 #mobx 2 #react 12 #vue 2

状态管理是前端整天遇到的概念，但是大家是否思考过什么是状态，管理的又是什么呢？

我们知道，程序是处理数据的，数据是信息的载体，比如颜色是红色或蓝色这就是数据。

那为什么不叫数据管理呢？状态和数据是什么关系？

什么是状态

状态是数据的变化，比如颜色是红色或蓝色是数据，而颜色从红色变为蓝色这就是状态了。

状态的改变对应着视图的渲染或者某段逻辑的执行。比如颜色从红色变为蓝色可能就要重新渲染视图，并且执行发送请求到服务端的逻辑。

通过视图交互或者其他方式触发状态的变化，状态变化联动视图的渲染和逻辑的执行，这就是前端应用的核心。

为什么之前 jQuery 时代没咋听说状态管理的概念，而 Vue、React 时代经常听到呢？

jQuery 时代是手动把数据渲染到视图和执行数据变化之后的逻辑的，它可能没有明确的状态这一层，而是直接把数据渲染成 dom，下次需要数据也是从 dom 来取的。

而 Vue、React 前端框架的时代不需要手动操作 dom 和执行数据变化之后的逻辑，只要管理好状态，由前端框架负责状态变化之后的处理。

状态管理管理的是什么呢？

什么是状态管理

状态管理具体有两层含义：

- **状态变化之前的逻辑，一般是异步的。**
- **状态变化之后的联动处理，比如渲染视图或执行某段逻辑。**

比如 React 的 setState 不会马上修改状态，而是异步的批量的执行，把状态做一下合并。

比如 Redux 的 action 在修改全局 state 之前也是要经历中间件的处理的。

这些都是状态变化之前的异步过程的管理，是状态管理的第一层含义。

再比如 React setState 修改了状态之后要触发视图的渲染和生命周期函数的执行，hooks 在依赖数组的状态变化之后也会重新执行。（vue 的 data 修改之后会重新渲染视图、执行 computed 和 watch 逻辑）

Redux 修改了全局状态之后要通知组件做渲染或者做其他逻辑的处理，Vuex、Mobx 等都是。

这些是状态变化之后的联动处理的管理，是状态管理的第二层含义。

我们知道了什么是状态，什么是状态管理，那前端框架 Vue、React 和全局状态管理的库 Redux、Mobx、Vuex 都是怎么实现状态管理的呢？

状态管理的两种实现思路

状态不会是一个，多个状态的集合会用对象的 key、value 来表示，比如 React 的 state 对象，Vue 的 data 对象（虽然叫 data 也是指的状态）。

怎么监听一个对象的变化呢？

我们是不是可以提供一个 api 来修改，在这个 api 内做 state 变化之前的处理，并且在 state 变化之后做联动处理。

这样的方案只能通过 api 触发状态修改，直接修改 state 是触发不了状态管理逻辑的。

React 的 setState 就是这种思路，通过 setState 修改状态会做状态变化之前的批量异步的状态合并，会触发状态变化之后视图渲染和 hooks、生命周期的重新执行。但是直接修改 state 是没用的。

那怎么让直接修改状态也能监听到变化呢？

可以对状态对象做一层代理，代理它的 get、set，当执行状态的 get 的时候把依赖该状态的逻辑收集起来，当 set 修改状态的时候通知所有依赖它的逻辑（视图渲染、逻辑执行）做更新。

Vue 的 data 监听变化就是用的这种思路，在状态 get 的时候把依赖封装成 Watcher，当 set 的时候通知所有 Watcher 做更新。

这种思路叫做响应式（reactive），也就是状态变化之后自动响应变化做联动处理的意思。

代理 get、set 可以用 Object.defineProperty 的 api，但是它不能监听动态增删的对象属性，所以 Vue3 改为了用 Proxy 的 api 实现。

监听对象的变化就这两种方式：

- **提供 api 来修改，内部做联动处理。**
- **对对象做一层代理，set 的时候做联动处理，通知 get 时收集的所有依赖。**

前端框架状态变化的性能优化

但是频繁的修改 state 不是每次都要做联动处理，有一些可以合并的，比如两次都把颜色改为红色，那后续逻辑就没必要执行两次，需要再做些性能方面的优化。

所以 React 的 setState 是异步的，会做批量的 state 合并（注意，React 的 setState 传入的不是最终的 state，而是 state 的 diff，React 内部去把这些 diff state 更新到 state）。

```
export function batchedUpdates<A, R>(fn: A => R, a: A): R {
  const prevExecutionContext = executionContext;
  executionContext |= BatchedContext;
  try {
    return fn(a);
  } finally {
    executionContext = prevExecutionContext;
    // If there were legacy sync updates, flush them at the end
```


而 Vue 因为是直接修改的同一个对象，所以没必要做啥合并，它的 Watcher 执行是异步的，对多次放到队列里的 Watcher 做下去重就行了。

```
export function queueWatcher (watcher: Watcher) {
  const id = watcher.id
  if (has[id] == null) {
    has[id] = true
    if (!flushing) {
      queue.push(watcher)
    } else {
      // if already flushing, splice the watcher based on
```


组件间的状态管理

组件内的状态管理就是这样的，利用前端框架自带的 state 机制来管理。

那组件之间呢？一个组件的 state 变了如何联动其他组件变化？

props

通过 props，把当前组件的 state 作为 props 传入其他组件就行了，这样就能联动变化。

但是 props 只能一层层传递，如果组件和想联动变化的组件相隔很多层，传递 props 就很麻烦。

这种情况下前端框架也都提供了解决方案，React 提供了 Context、Vue 提供了 Event Bus。

Context、Event Bus

React 组件可以在 context 中存放 state，当 context 中的 state 变化的时候会直接触发关联组件的渲染。

Vue 可以在一个组件内 emit 一个事件，然后另一个组件 on 这个事件，然后更新自己的 data 来触发渲染。不过这两个 api 在 Vue3 都废弃了。

这种前端框架自带的任意层组件的状态联动方案只能处理简单的场景，复杂的场景还是得用全局状态管理库，比如 Redux、Vuex、Mobx 这些。

为什么这么说呢？

还记得状态管理的两层含义么？状态变化前的异步过程的管理，状态变化后的联动处理。

Context 和 Event Bus 都只做到了状态变化后的联动处理，但是没有对状态变化前的异步过程管理做支持。

比如多个组件都要修改 context 中的值（或者通过 event bus 修改全局状态），这个过程都要执行一段异步逻辑，要做 loading 的展示，那多个组件里怎么复用这段 loading 的逻辑呢？

还有，如果异步过程比较麻烦，需要用 rxjs 这样的库，用 context 和 event bus 的方案怎么和 rxjs 结合呢？

当然，是可以对 context 和 event bus 做一些逻辑复用的封装和一些结合 rxjs 方案之类的封装的，但是比较麻烦。

而且更重要的是如果你想做这些的时候，那也就没必要用 context 和 event bus 了，直接用全局状态管理库就行。

Redux、Mobx、Vuex

redux 就提供了中间件的机制，组件里发送 action 到 store（存放全局 state 的地方），之前会经历层层中间件的处理，在这里就可以做一些可复用的逻辑的封装，比如 loading 的处理，也可以结合 rxjs 这种异步过程处理方案。

redux 里最常用的中间件就是 redux-saga 和 redux-observable 了，这俩都是做异步过程的管理的。

redux-saga 是基于 generator 实现的，不管是同步还是异步，都只要声明式的描述要执行的逻辑就行，由 saga 内部的执行器会去做同步或异步的处理，描述异步逻辑就很简洁，而且 redux-saga 提供了很多内置的逻辑封装。

```
function* mainSaga(getState) {
  const results = yield [call(task1), call(task2), ...]
  yield put(showResults(results))
}
```


redux-observable 则是结合 rxjs 的方案了，把 action 变成数据源，经历层层 opreator 的处理，最后传递到 store。可以用 rxjs 生态大量的 oprator，做下组装就行，根本不用自己写异步逻辑的具体实现。

```
import { ajax } from 'rxjs/ajax';

const fetchUserEpic = (action$, state$) => action$.pipe(
  ofType('FETCH_USER'),
  mergeMap(({ payload }) => ajax.getJSON(`/api/users/${payload}`).pipe(
    map(response => ({
      type: 'FETCH_USER_FULFILLED',
      payload: response
    }))
  )
);
```

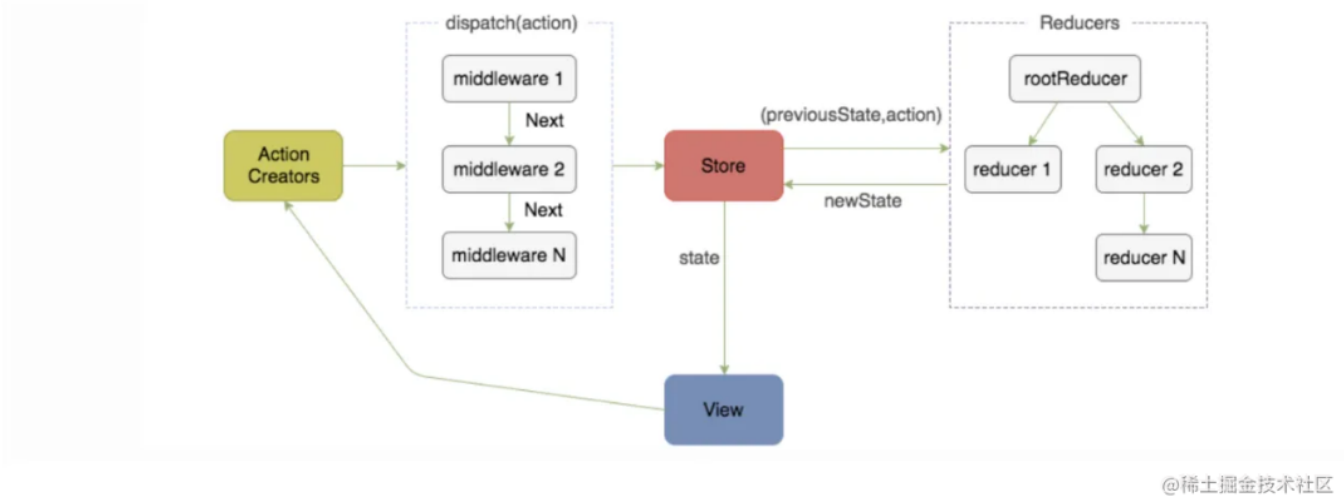

mobx 没有提供中间件机制，它的 action 是执行状态 class 的某个方法，可以用 class 的那套来做封装。

有的同学对这些状态管理库不太熟，简单来介绍下。

我们理清了状态管理的实现只有两种方案，一种是提供 api 做修改，一种是对 state 对象做响应式代理。

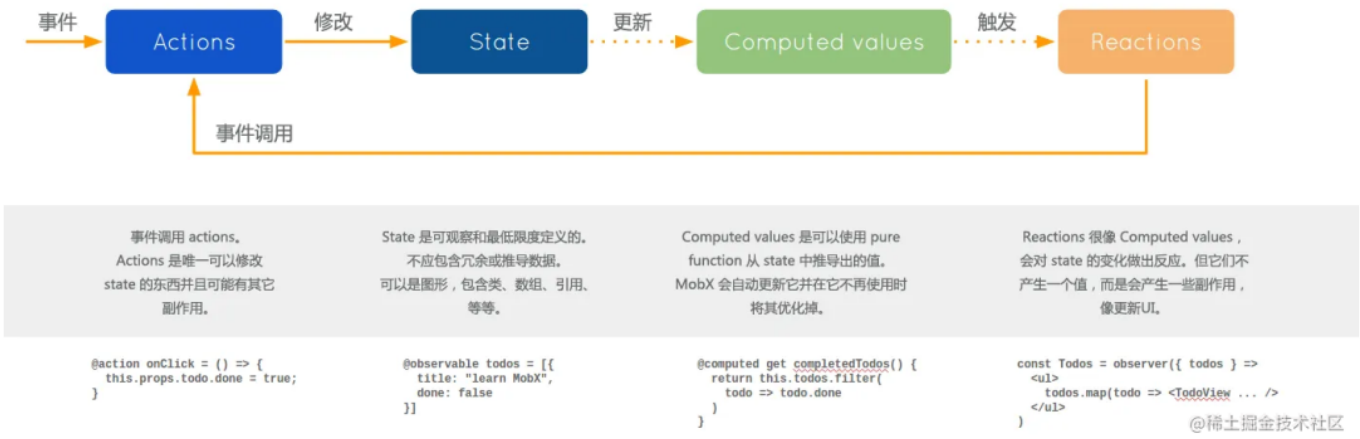
前端框架的状态管理是这样，独立的全局状态管理库也同样是这样。

redux 就是提供 api 来修改的方案，通过 reducer 函数来对传入的 action 做处理，返回新的 state。



而且 redux 这种思路是函数式的思想，每个 reducer 都是输入和输出一一对应的纯函数，返回的 state 都是全新的，为了方便创建新的 state，一般会搭配 immutable 库，只要修改属性就会返回新的 state 对象。

mobx 是响应式代理的方案，它对全局 state 做了一层代理（通过 Object.defineProperty），状态的 get 收集依赖，set 的时候触发依赖更新。



所以这种方案很自然的可以把全局 state 组织成一个个 class，是面向对象的思想，可以通过继承等方式实现逻辑复用。

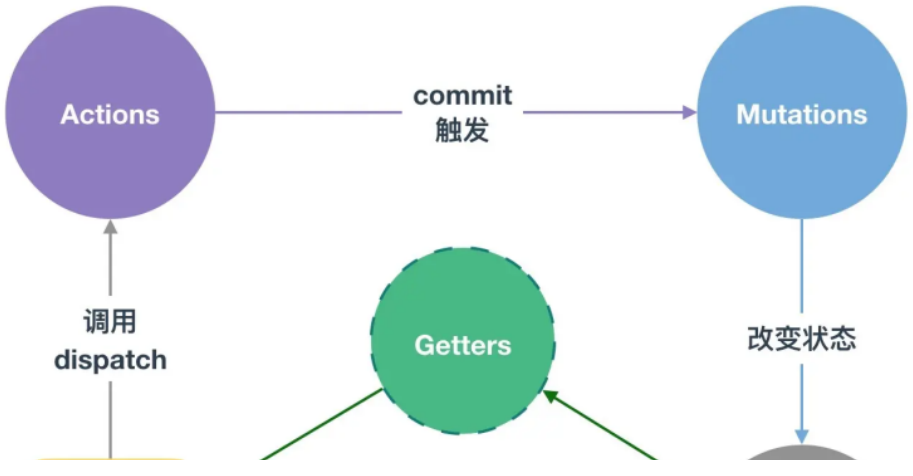
```
import React, {Component} from 'react';
import ReactDOM from 'react-dom';
import {observer} from 'mobx-react';

@observer
class TodoListView extends Component {
    render() {
        return <div>
            <ul>
                {this.props.todoList.todos.map(todo =>
                    <TodoView todo={todo} key={todo.id} />
                )}
            </ul>
            Tasks left: {this.props.todoList.unfinishedTodoCount}
        </div>
    }
}

const TodoView = observer(({todo}) =>
    <li>
        <input
            type="checkbox"
            checked={todo.finished}
            onClick={() => todo.finished = !todo.finished}
        />{todo.title}
    </li>
)

const store = new TodoList();
ReactDOM.render(<TodoListView todoList={store} />, document.getElementById('mount'));
```

vuex 则像是两种思路的结合，内部是用响应式代理来实现的变化监听，但是暴露出的 api 却是 redux 的 action 那一套。



和 React 搭配使用的话，需要把组件添加到状态的依赖中，这个不用自己调用 subscribe 之类的 api，直接用一些封装好的高阶组件（接受组件作为参数返回新的组件的组件）就行，比如 react-redux 的 connect，mobx-react 的 observer。

总结

讲了这么多，回过头来看一下就会发现：

不管是前端框架内置的组件内状态变化管理的方案（react 的 setState、vue 的直接修改 data），还是前端框架提供的组件间的状态管理方案（props、react 的 context、vue 的 event bus），或是第三方的全局状态管理方案（redux、vuex、mobx 等），都没有脱离那两种实现状态管理的方式：提供修改状态的 api 或者对状态对象做一层响应式代理。也没有脱离状态管理的两层含义：对状态变化前的异步过程做管理，状态变化后做联动处理。只不过它们用在了不同的地方（前端框架内、全局状态管理库），提供了不同的封装形式（对象、函数），基于不同的思想（函数式、面向对象）结合了不同的异步管理方案（rxjs、generator + 自定义执行器）。

所以，**状态就是数据的变化。前端应用的核心问题就是管理状态，管理状态变化之前的通过视图或者其他方式触发的异步过程，管理状态变化之后的联动渲染和联动的逻辑执行。**

虽然我们会用不同的前端框架，不同的全局状态管理库，结合不同的异步过程处理方案，但是思想都是一样的。

毫不夸张地说，理解了状态管理，就理解了前端开发的核心。

收录于合集 #react 12

< 上一篇
从根上理解 React Hooks 的闭包陷阱

下一篇 >
React Hooks 的实现必须依赖 Fiber 么?

文章已于2022-05-03修改