状态管理详解

近两年前端技术的发展如火如荼，大量的前端项目都在使用或转向 Vue 和 React 的阵营，由前端渲染页面的单页应用占比也越来越高，这就代表前端工作的复杂度也在直线上升，前端页面上展示的信息越来越多也越来越复杂。我们知道，任何状态都需要进行管理，那么今天我们来聊聊前端状态管理。

为什么需要状态管理

举个例子
图书馆的管理，原来是开放式的，所有人可以随意进出书库借书还书，如果人数不多，这种方式可以减少流程，增加效率，一旦人数变多就势必造成混乱。
Flux 思想就像是给这个图书馆加上了一个管理员（究竟什么是 flux 思想后面会说），所有借书还书的行为都需要委托管理员去做，管理员会规范对书库的操作行为，也会记录每个人的操作，减少混乱的现象。
一个比喻
我们寄一件东西的过程
没有快递时：

打包准备好要送出去的东西
直接到朋友家，把东西送给朋友
很直接很方便，很费时间

有了快递公司：
打包准备好要送出去的东西
到快递公司，填写物品，收件人等基本信息
快递公司替你送物品到你的朋友家，我们的工作结束了
多了快递公司，让快递公司给我们送快递。
当我们只寄送物品给一个朋友，次数较少，物品又较少的时候，我们直接去朋友家就挺好的。但当我们要频繁寄送给很多朋友很多商品的时候，问题就复杂了。
软件工程的本质即是管理复杂度。使用状态管理类框架会有一定的学习成本而且通常会把简单的事情做复杂，但如果我们想做复杂一点的事情（同

时寄很多物品到多个不同地址），对我们来说，快递会让复杂的事情变的简单。
这同时也解释了，是否需要添加状态管理框架，我们可以根据自己的业务实际情况和技术团队的偏好而添加，有些情况下，创建一个全局对象就能解决很多问题。

在工作中多个状态互相绑定的应用场景也是蛮多的，大家可以看看各大电商的购物车结算页面，包括大家定外卖的购物车页面。复杂之处主要在于**业务状态多**（比如商品状态、店铺优惠券和组合套餐等）、**跨层级或位置状态联动多**（勾选购物车商品，先请求获得被勾选商品最新价格、状态等，再计算价格等在上面和下面展示总价等）。随着不同操作，分布在不同地方的代码响应不同的逻辑和数据。

如何解决页面之间的多状态问题

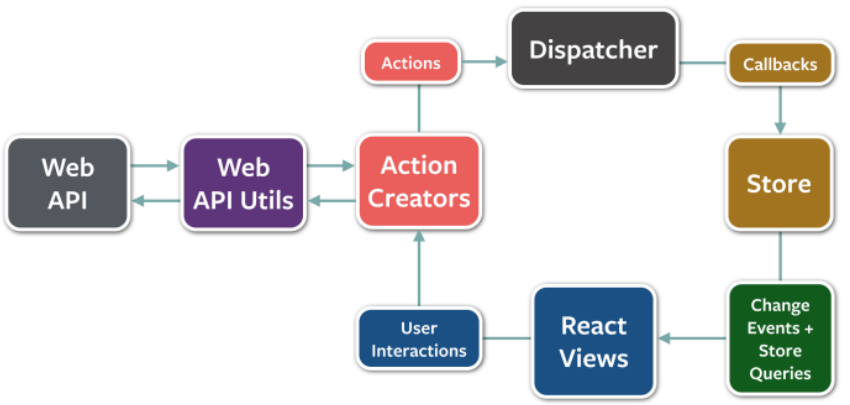
在 Web 应用开发中，AngularJS 扮演了重要角色。然而 AngularJS 数据和视图的双向绑定基于脏检测的机制，在性能上存在短板，任何数据的变更都会重绘整个视图。但是，由状态反应视图、自动更新页面的思想是先进的，为了解决性能上的问题，Facebook 的工程师们提出了 Virtual DOM 的思想。将 DOM 放到内存中，state 发生变化的时候，根据 state 生成新的 Virtual DOM，再将它和之前的 Virtual DOM 通过一个 diff 算法进行对比，将被改变的内容在浏览器中渲染，避免了 JS 引擎频繁调用渲染引擎的 DOM 操作接口，充分利用了 JS 引擎的性能。有了 Virtual DOM 的支持，React 也诞生了。
有了 React，「state => view」的思想也就有了很好的实践，但反过来呢，怎么在 view 中合理地修改 state 成为了一个新的问题，为此，Facebook 提出了 **Flux 思想。**

是的，Flux 不是某一个 JS 库的名称，而是一种架构思想，很多 JS 库则是这种思想的实现，例如 Alt、Fluxible 等，它用于构建客户端 Web 应用，规范数据在 Web 应用中的流动方式。

那么这个和状态管理有什么关系呢？我们知道，React 或是 Vue 只是一个视图层的库，并没有对数据层有任何的限制，换言之任何视图组件中都可能存在改变数据层的代码，而过度放权对于数据层的管理是不利的，另外

一旦数据层出现问题将会很难追溯，因为不知道变更是从哪些组件发起的。另外，如果数据是由父组件通过 props 的方式传给子组件的话，组件之间会产生耦合，违背了模块化的原则。

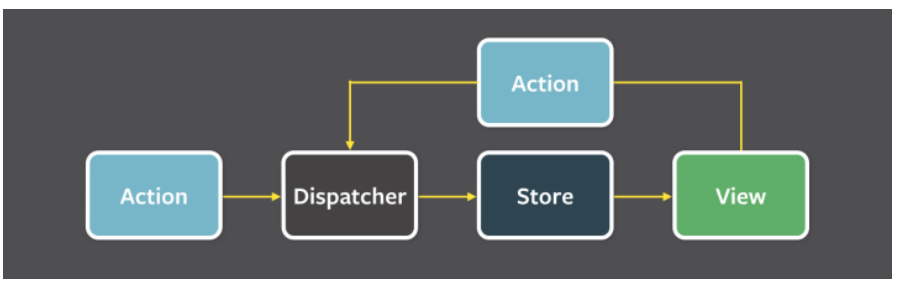
而 Flux 的思维方式是单向的，将之前放权到各个组件的修改数据层的 controller 代码收归一处，统一管理，组件需要修改数据层的话需要去触发特定的预先定义好的 dispatcher，然后 dispatcher 将 action 应用到 model 上，实现数据层的修改。然后数据层的修改会应用到视图上，形成一个单向的数据流。



Flux 思想的实现

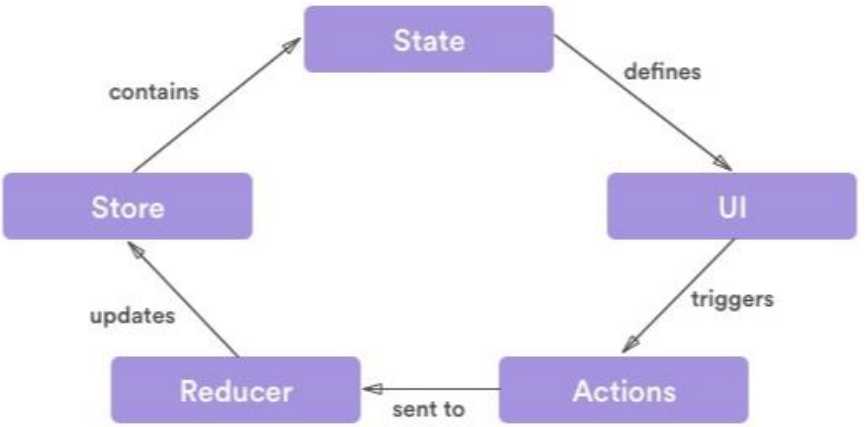
Flux 的实现有很多，不同的实现也各有亮点，下面介绍一些比较流行的 Flux 的实现。

Flux



这应该是 Flux 的一个比较官方”的实现，显得中规中矩，实现了 Flux 架构文档里的基本概念。它的核心是 Dispatcher，通过 Dispatcher，用户可以注册需要相应的 action 类型，对不同的 action 注册对应的回调，以及触发 action 并传递 payload 数据。

Redux



Redux 实际上相当于 Reduce + Flux，和 Flux 相同，Redux 也需要你维护一个数据层来表现应用的状态，而不同点在于 Redux 不允许对数据层进行修改，只允许你通过一个 Action 对象来描述需要做的变更。在 Redux 中，去掉了 Dispatcher，转而使用一个纯函数来代替，这个纯函数接收原 state tree 和 action 作为参数，并生成一个新的 state tree 代替原来的。而这个所谓的纯函数，就是 Redux 中的重要概念 —— Reducer。

在函数式编程中，Reduce 操作的意思是通过遍历一个集合中的元素并依次将前一次的运算结果代入下一次运算，并得到最终的产物，在 Redux 中，reducer 通过合并计算旧 state 和 action 并得到一个新 state 则反映了这样的过程。

因此，Redux 和 Flux 的第二个区别则是 Redux 不会修改任何一个 state，而是用新生成的 state 去代替旧的。这实际上是应用了不可变数据（Immutable Data），在 reducer 中直接修改原 state 是被禁止的，Facebook 的 Immutable 库可以帮助你使用不可变数据，例如构建一个可以在 Redux 中使用的 Store。

下面是一个用 Redux 构建应用的状态管理的示例：

```
const { List } = require('immutable')
const initialState = {
  books: List([])
}
import { createStore } from 'redux'

// action
const addBook = (book) => {
  return {
    type: ADD_BOOK,
    book
  }
}

// reducer
const books = (state = initialState, action) => {
  switch (action.type) {
    case ADD_BOOK:
      return Object.assign({}, state, {
        books: state.books.push(action.book)
      })
  }
  return state
}

// store
const bookStore = createStore(books, initialState)
// dispatching action
store.dispatch(addBook({/* new book */}))
```

Redux 的工作方式遵循了严格的单向数据流原则，从上面的代码示例中可以看出，整个生命周期分为：

在 store 中调用 dispatch，并传入 action 对象。action 对象是一个描述变化的普通对象，在示例中，它由一个 creator 函数生成。

接下来，store 会调用注册 store 时传入的 reducer 函数，并将当前的 state 和 action 作为参数传入，在 reducer 中，通过计算得到新的 state

并返回。

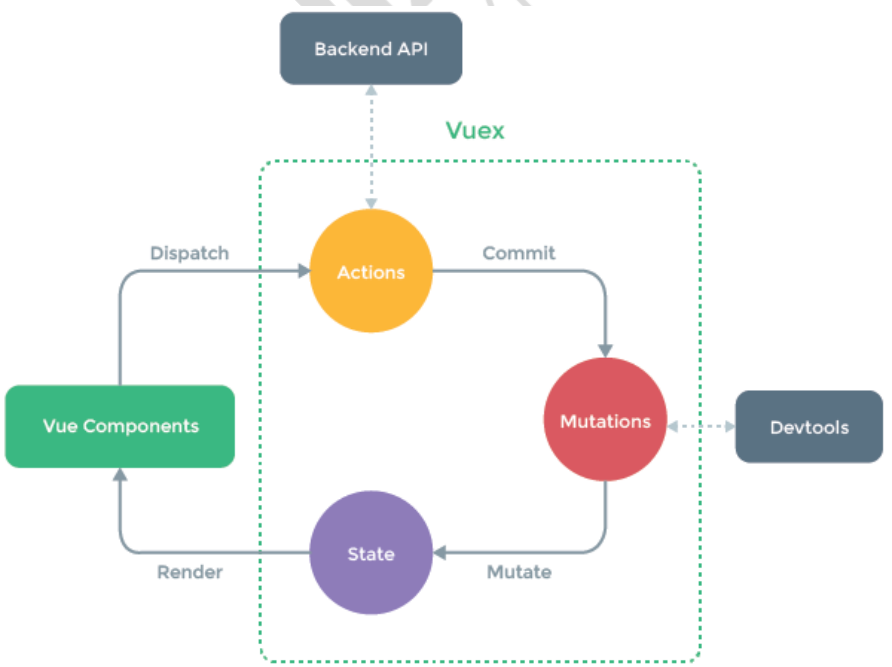
store 将 reducer 生成的新 state 树保存下来，然后就可以用新的 state 去生成新的视图，这一步可以借助一些库的帮助，例如官方推荐的 React Redux。

如果一个应用规模比较大的话，可能会面临 reducer 过大的问题。这时候我们可以对 reducer 进行拆分，例如使用 combineReducers，将多个 reducer 作为参数传入，生成新的 reducer。当触发一个 action 的时候，新 reducer 会触发原有的多个 reducer:

```
const book(state = [], action) => {
  // ...
  return newState
}
const author(state = {}, action) => {
  // ...
  return newState
}
const reducer = combineReducers({ book, author })
```

关于 Redux 的更多用法，可以仔细阅读文档，这里就不多介绍了。

Vuex



中国前端业务中使用 Vue 的比例是最高的，说到 Vue 中的状态管理就不得不提到 Vuex。Vuex 也是基于 Flux 思想的产品，所以在某种意义上它和 Redux 很像，但又有不同，下面通过 Vuex 和 Redux 的对比来看看 Vuex 有什么区别。

首先，和 Redux 中使用不可变数据来表示 state 不同，Vuex 中没有 reducer 来生成全新的 state 来替换旧的 state，Vuex 中的 state 是可以被修改的。这么做的原因和 Vue 的运行机制有关系，Vue 基于 ES5 中的 getter/setter 来实现视图和数据的双向绑定，因此 Vuex 中 state 的变更可以通过 setter 通知到视图中对应的指令来实现视图更新。

另外，在 Vuex 中也可以记录每次 state 改变的具体内容，state 的变更可被记录与追踪。例如 Vue 的官方调试工具中就集成了 Vuex 的调试工具，使用起来和 Redux 的调试工具很相似，都可以根据某次变更的 state 记录实现视图快照。

上面说到，Vuex 中的 state 是可修改的，而修改 state 的方式不是通过 actions，而是通过 mutations。一个 mutation 是由一个 type 和与其对应的 handler 构成的，type 是一个字符串类型用以作为 key 去识别具体的某个 mutation，handler 则是对 state 实际进行变更的函数。

```
// store
const store = {
  books: []
}

// mutations
const mutations = {
  [ADD_BOOKS](state, book) {
    state.books.push(book)
  }
}
```

那么 action 呢？Vuex 中的 action 也是 store 的组成部分，它可以被看成是连接视图与 state 的桥梁，它会被视图调用，并由它来调用 mutation handler，向 mutation 传入 payload。

这时问题来了，Vuex 中为什么要增加 action 这一层呢，是多此一举吗？Vuex 核心的概念——mutation 必须是同步函数，而 action 可以包含任

意的异步操作。
回到这个问题本身，如果在视图中不进行异步操作（例如调用后端 API）只是触发 action 的话，异步操作将会在 action 内部执行：

```
const actions = {
  addBook({ commit }) {
    request.get(BOOK_API).then(res => commit(ADD_BOOK, res.body.new_book))
  }
}
```

可以看出，这里的状态变更相当于是 action 产生的副作用，mutation 的作用是将这些副作用记录下来，这样就形成了一个完整数据流闭环，数据流的顺序如下：
在视图中触发 action，并根据实际情况传入需要的参数。
在 action 中触发所需的 mutation，在 mutation 函数中改变 state。
通过 getter/setter 实现的双向绑定会自动更新对应的视图。

MobX

MobX 是一个比较新的状态管理库，它的前身是 Mobservable，实际上 MobX 相当于是 Mobservable 的 2.0 版本。
Mobx 和 Redux 相比，差别就比较大了。如果说 Redux 吸收并发扬了很多函数式编程思想的话，Mobx 则更多体现了面向对象及的特点。MobX 的特点总结起来有以下几点：

Observable。它的 state 是可被观察的，无论是基本数据类型还是引用数据类型，都可以使用 MobX 的 (@)observable 来转变为 observable value。

Reactions。它包含不同的概念，基于被观察数据的更新导致某个计算值（computed values），或者是发送网络请求以及更新视图等，都属于响应

的范畴，这也是响应式编程（Reactive Programming）在 JavaScript 中的一个应用。

Actions。它相当于所有响应的源头，例如用户在视图上的操作，或是某个网络请求的响应导致的被观察数据的变更。
和 Redux 对单向数据流的严格规范不同，Mobx 只专注于从 store 到 view 的过程。在 Redux 中，数据的变更需要监听（可见上文 Redux 示例代码），而 Mobx 的数据依赖是基于运行时的，这点和 Vuex 更为接近。

总结

状态管理的研究并不是前端领域独有的问题，实际上前端状态管理的很多思想都是借鉴于成熟很多的软件开发体系。相对于软件开发，前端还是一个很新的领域，只有多学习其他领域的优秀经验前端界才能发展得更好。