Redux

Redux 介绍

> 本文主要是对 Redux 官方文档 的梳理以及自身对 Redux 的理解。

单页面应用的痛点

对于复杂的单页面应用，状态（state）管理非常重要。state 可能包括：服务端的响应数据、本地对响应数据的缓存、本地创建的数据（比如，表单数据）以及一些 UI 的状态信息（比如，路由、选中的 tab、是否显示下拉列表、页码控制等等）。如果 state 变化不可预测，就会难于调试（state 不易重现，很难复现一些 bug）和不易于扩展（比如，优化更新渲染、服务端渲染、路由切换时获取数据等等）。

Redux 就是用来确保 state 变化的可预测性，主要的约束有：

- state 以单一对象存储在 store 对象中
- state 只读
- 使用纯函数 reducer 执行 state 更新

> state 为单一对象，使得 Redux 只需要维护一棵状态树，服务端很容易初始化状态，易于服务器渲染。state 只能通过 dispatch(action) 来触发更新，更新逻辑由 reducer 来执行。

Actions、Reducers 和 Store

action 可以理解为应用向 store 传递的数据信息（一般为用户交互信息）。在实际应用中，传递的信息可以约定一个固定的数据格式，比如: Flux Standard Action。
为了便于测试和易于扩展，Redux 引入了 Action Creator:

```
functionaddTodo(text) {
  return {
    type: ADD_TODO,
    text,
  }
}
store.dispatch(addTodo(text))
```

> dispatch(action) 是一个同步的过程：执行 reducer 更新 state -> 调用 store 的监听处理函数。如果需要在 dispatch 时执行一些异步操作（fetch action data），可以通过引入 Middleware 解决。

reducer 实际上就是一个函数：`(previousState, action) => newState`。用来执行根据指定 action 来更新 state 的逻辑。通过 combineReducers(reducers) 可以把多个 reducer 合并成一个 root reducer。

> reducer 不存储 state, reducer 函数逻辑中不应该直接改变 state 对象, 而是返回新的 state 对象（可以考虑使用 immutable-js）。

store 是一个单一对象：

- 管理应用的 state
- 通过 `store.getState()` 可以获取 state
- 通过 `store.dispatch(action)` 来触发 state 更新
- 通过 `store.subscribe(listener)` 来注册 state 变化监听器
- 通过 `createStore(reducer, [initialState])` 创建

> 在 Redux 应用中，只允许有一个 store 对象，可以通过 combineReducers(reducers) 来实现对 state 管理的逻辑划分（多个 reducer）。

Middleware

middleware 其实就是高阶函数，作用于 dispatch 返回一个新的 dispatch（附加了该中间件功能）。可以形式化为：`newDispatch = middleware1(middleware2(...(dispatch)...))`。

```
// thunk-middleware
export defaultfunctionthunkMiddleware({ dispatch, getState }) {
    return next => action =>
        typeof action === 'function' ? action(dispatch, getState) : next(action)
}
```

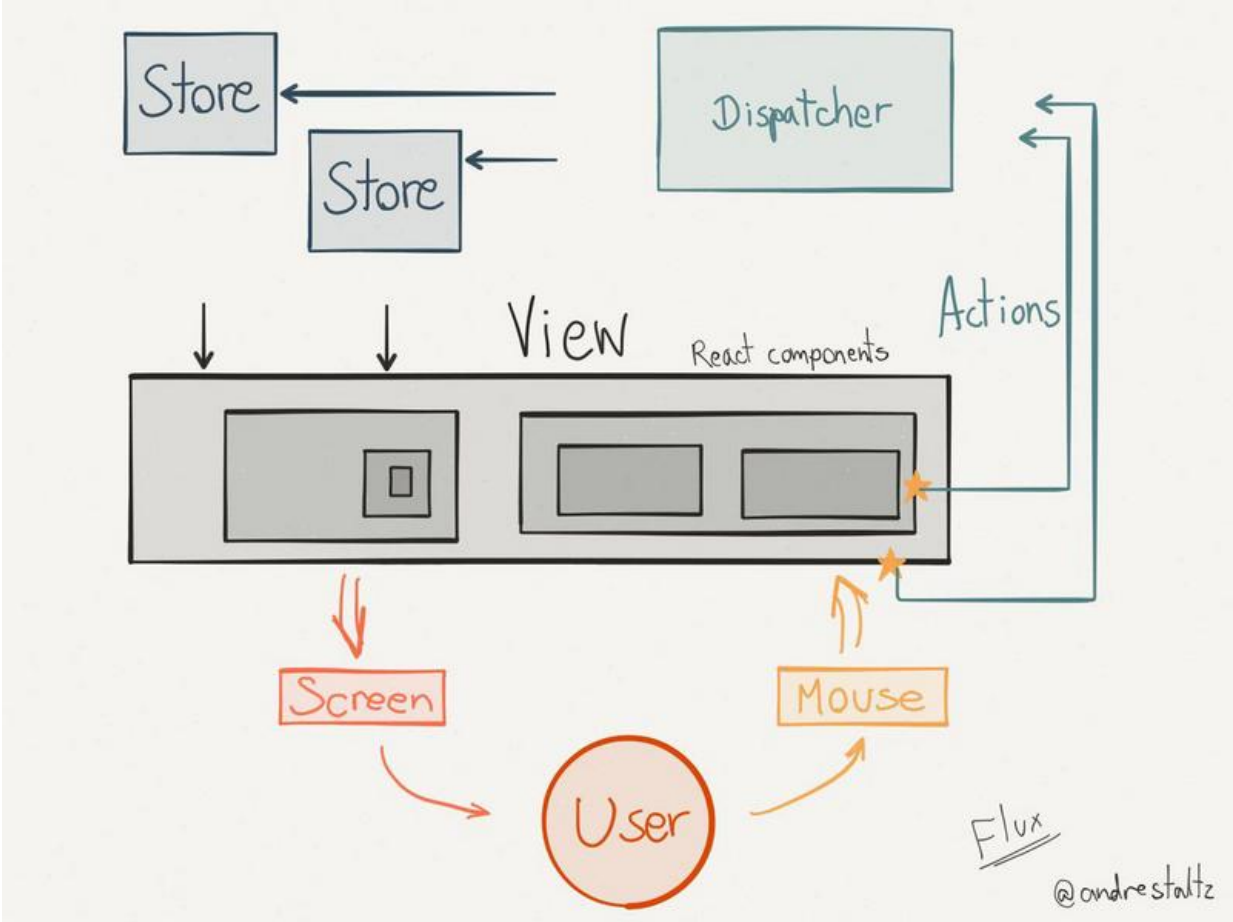
通过 thunk-middleware 我们可以看出中间件的一般形式：中间件函数接受两个参数参数： dispatch 和 getState（也就是说中间件可以获取 state 以及 `dispatch new action`）。中间件一般返回 `next(action)`（thunk-middleware 比较特殊，它用于 dispatch 执行异步回调的 action）。store 的创建过程如下：

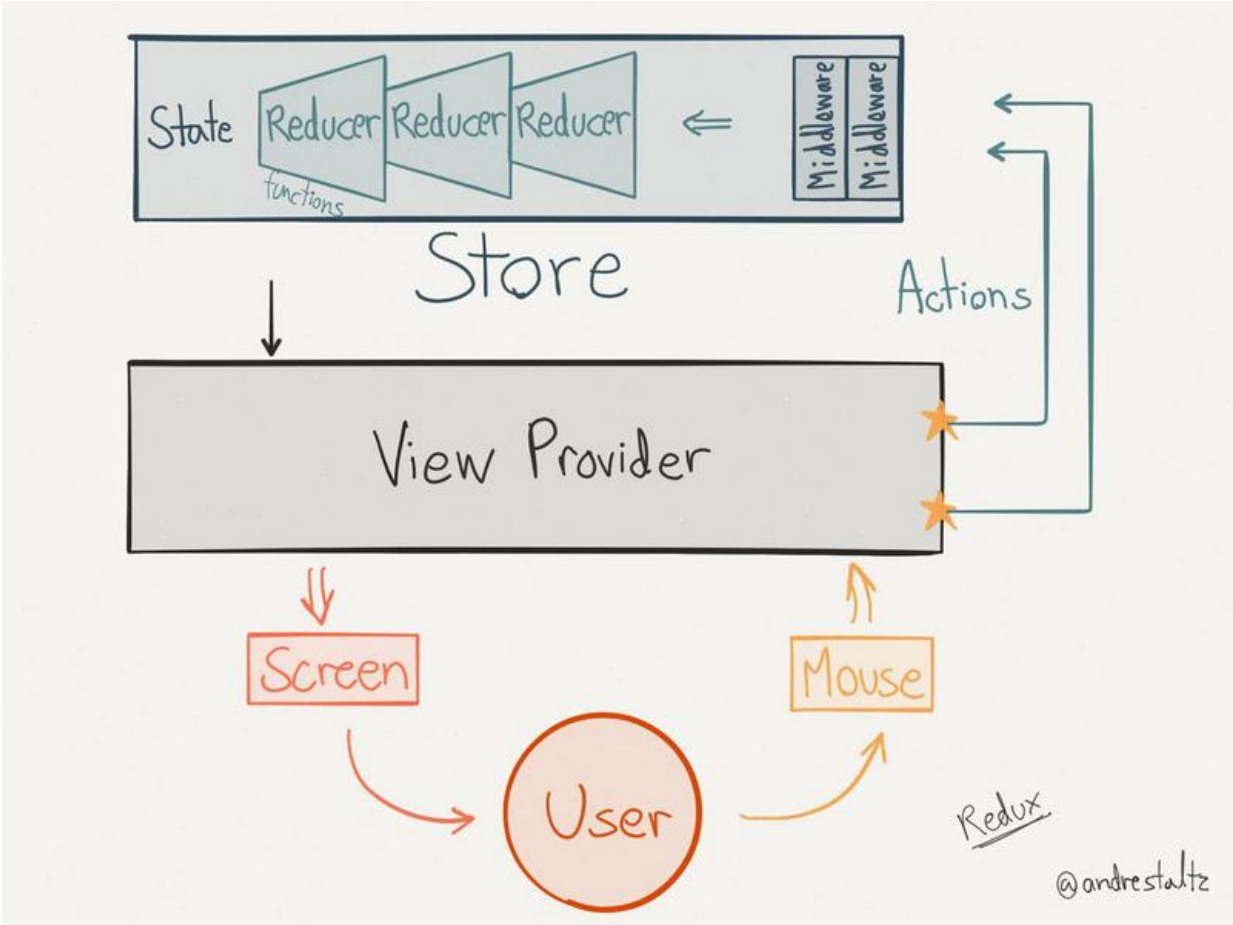
```
const reducer = combineReducers(reducers)
const finalCreateStore = applyMiddleware(promiseMiddleware, warningMiddleware,
    loggerMiddleWare)(createStore)
const store = finalCreateStore(reducer)
```

异步 Actions

单页面应用中充斥着大量的异步请求（ajax）。dispatch(action) 是同步的，如果要处理异步 action，需要使用一些中间件。redux-thunks 和 redux-promise 分别是使用异步回调和 Promise 来解决异步 action 问题的。

Redux 和传统 Flux 框架的比较





图来自 UNIDIRECTIONAL USER INTERFACE ARCHITECTURES

Redux 和 React

Redux 和 React 是没有必然关系的，Redux 用于管理 state，与具体的 View 框架无关。不过，Redux 特别适合那些 `state => UI` 的框架（比如：React, Deku）。

可以使用 react-redux 来绑定 React，`react-redux` 绑定的组件我们一般称之为 `smart components`，Smart and Dumb Components 在 `react-redux` 中区分如下：

Location	Use React-Redux	To read data, they	To change data, they	
“Smart” Components	Top level, route handlers	Yes	Subscribe to Redux state	Dispatch Redux actions
“Dumb” Components	Middle and leaf components	No	Read data from props	Invoke callbacks from props

> 简单来看：Smart component` 是连接 Redux 的组件（@connect），一般不可复用。Dumb component 是纯粹的组件，一般可复用。
> 两者的共同点是：无状态，或者说状态提取到上层，统一由 redux 的 store 来管理。redux state -> Smart component -> Dumb component -> Dumb component（通过 props 传递）。在实践中，少量 Dumb component 允许自带 UI 状态信息（组件 unmount 后，不需要保留 UI 状态）。
> 值得注意的是，Smart component 是应用更新状态的最小单元。实践中，可以将 route handlers 作为 Smart component，一个 Smart component 对应一个 reducer。

- 2015年08月27日发布