使用 Redux 管理你的 React 应用

文桥　2017/08/28

1 为什么要用 React？

`React` 是一个专注于视图层的 `JavaScript` 框架。`React` 维护了状态到视图的映射关系，开发者只需关心状态即可，由 `React` 来操控视图。在小型应用中，单独使用 `React` 是没什么问题的。但在复杂应用中，容易碰到一些状态管理方面的问题。

1.1 React 的优势

- 专注视图层与 `Javascript` 编程
- 组件化编程，促进代码重用
- `virtual DOM`，摆脱 `DOM` 操作的噩耗
- 鼓励函数式编程，拥抱 `OOP`
- 支持服务端渲染（`SSR`）

1.2 React 的不足

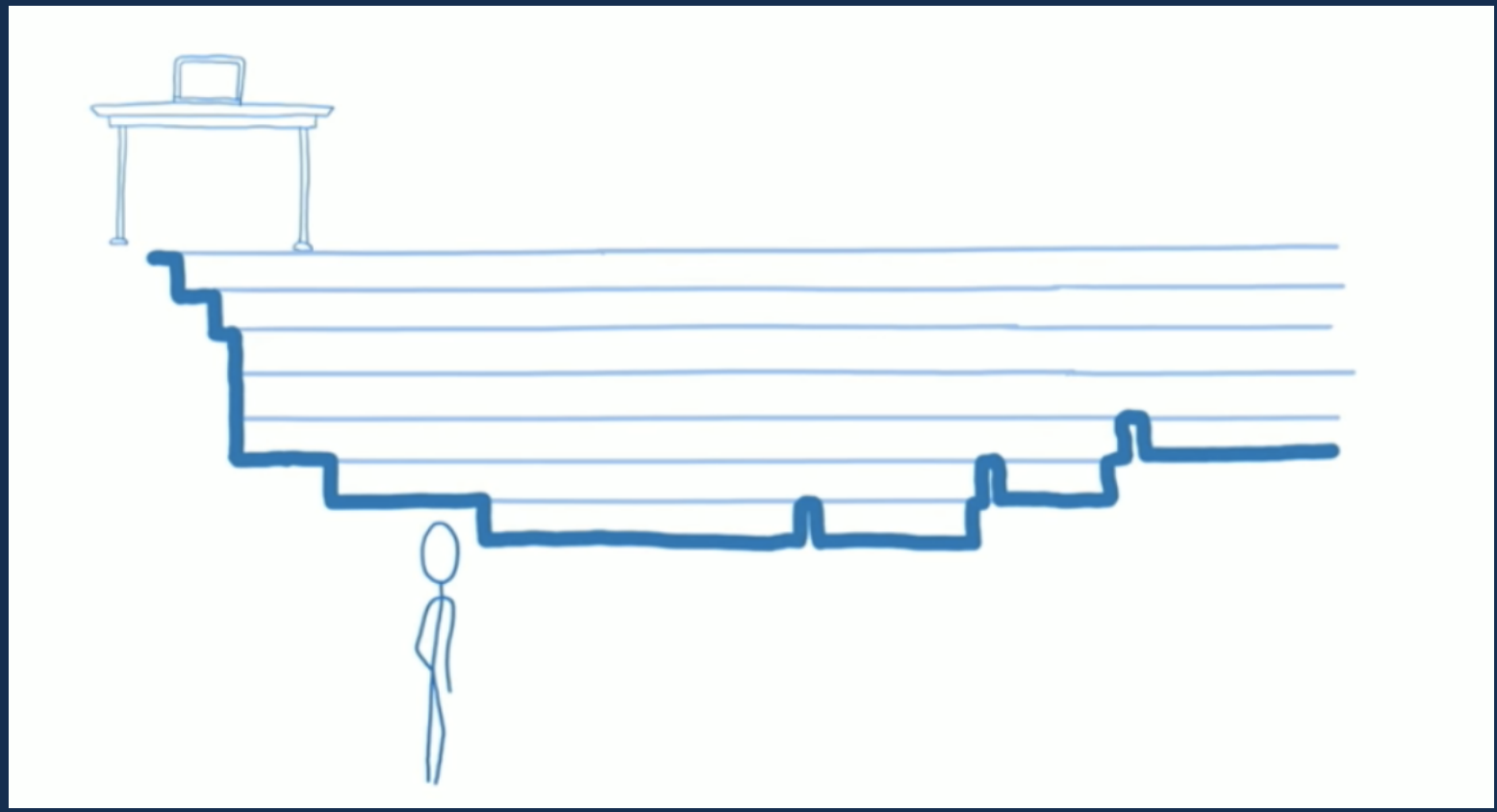
- 专注视图层，难以进行有效的状态管理（尤其是组件件通信）

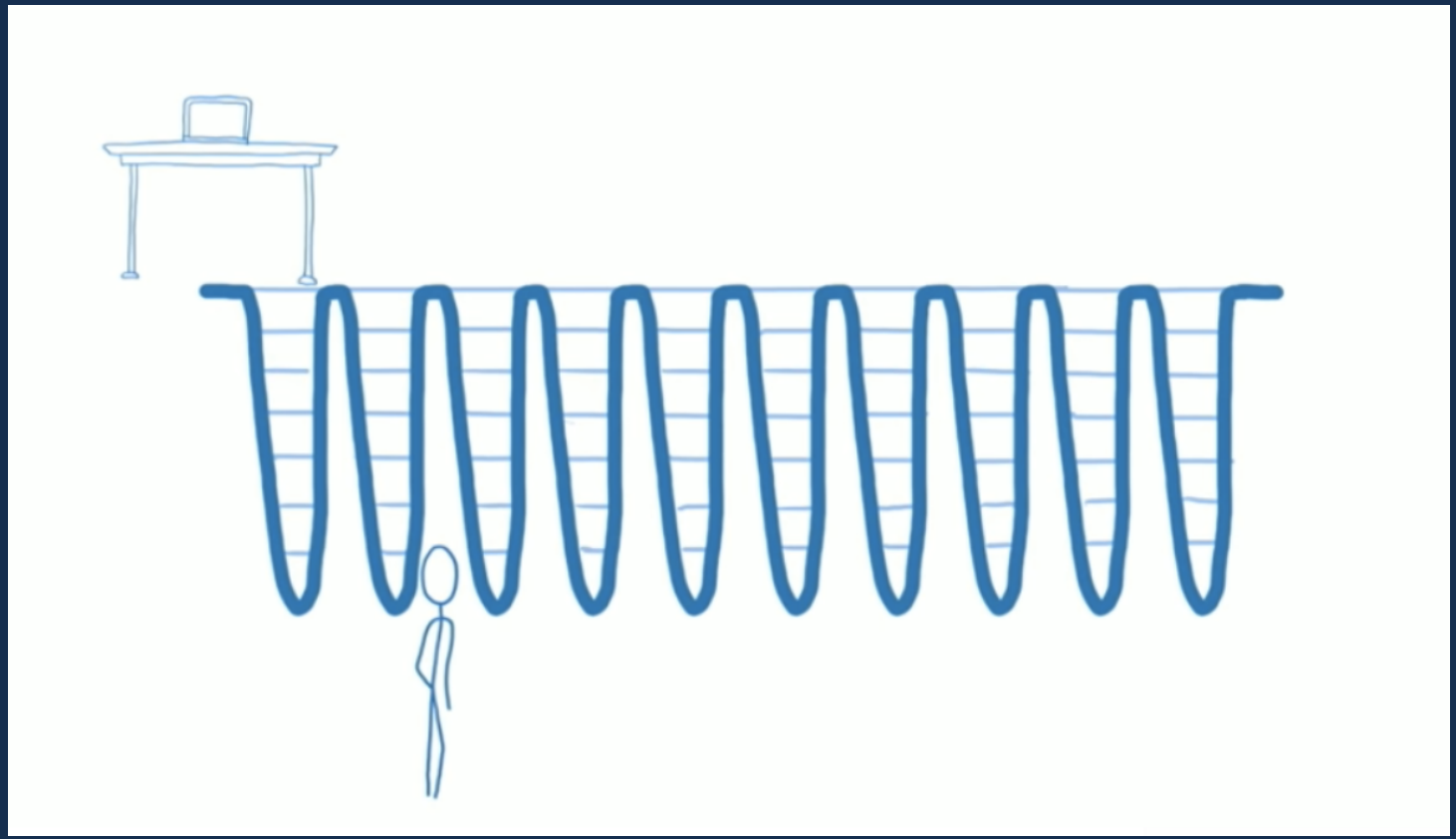
> React 只提供了在内部组件修改状态的接口 setState。导致数据、业务逻辑和视图层耦合在组件内部，不利于扩展和维护。React 应用即一颗组件树。兄弟节点，或者不在同一树杈的节点之间的状态同步是非常麻烦。（解决方案：Flux 架构）

- Diff 型 Virtual DOM 性能缺陷

> 每次有 state 的变化 React 重新计算，如果计算量过大，浏览器主线程来不及做其他的事情，比如 rerender 或者 layout，那例如动画就会出现卡顿现象。（解决方案：React Fiber -React 制定了一种名为 Fiber 的数据结构，加上新的算法，使得大量的计算可以被拆解，异步化，浏览器主线程得以释放，保证了渲染的帧率。从而提高响应性。）

详情请参考： 如何理解 React Fiber 架构？





2 模型驱动视图（Model-Driven-View）

- 给定一个数据模型，可以得到对应的的视图

```
V = f(M)
```

- 当数据模型产生变化的时候，其对应的视图也会随之变化

```
V + ΔV = f(M + ΔM)
```

- 如果从变更的角度去解读 Model，数据模型不是无缘无故变化的，它是由某个操作引起的

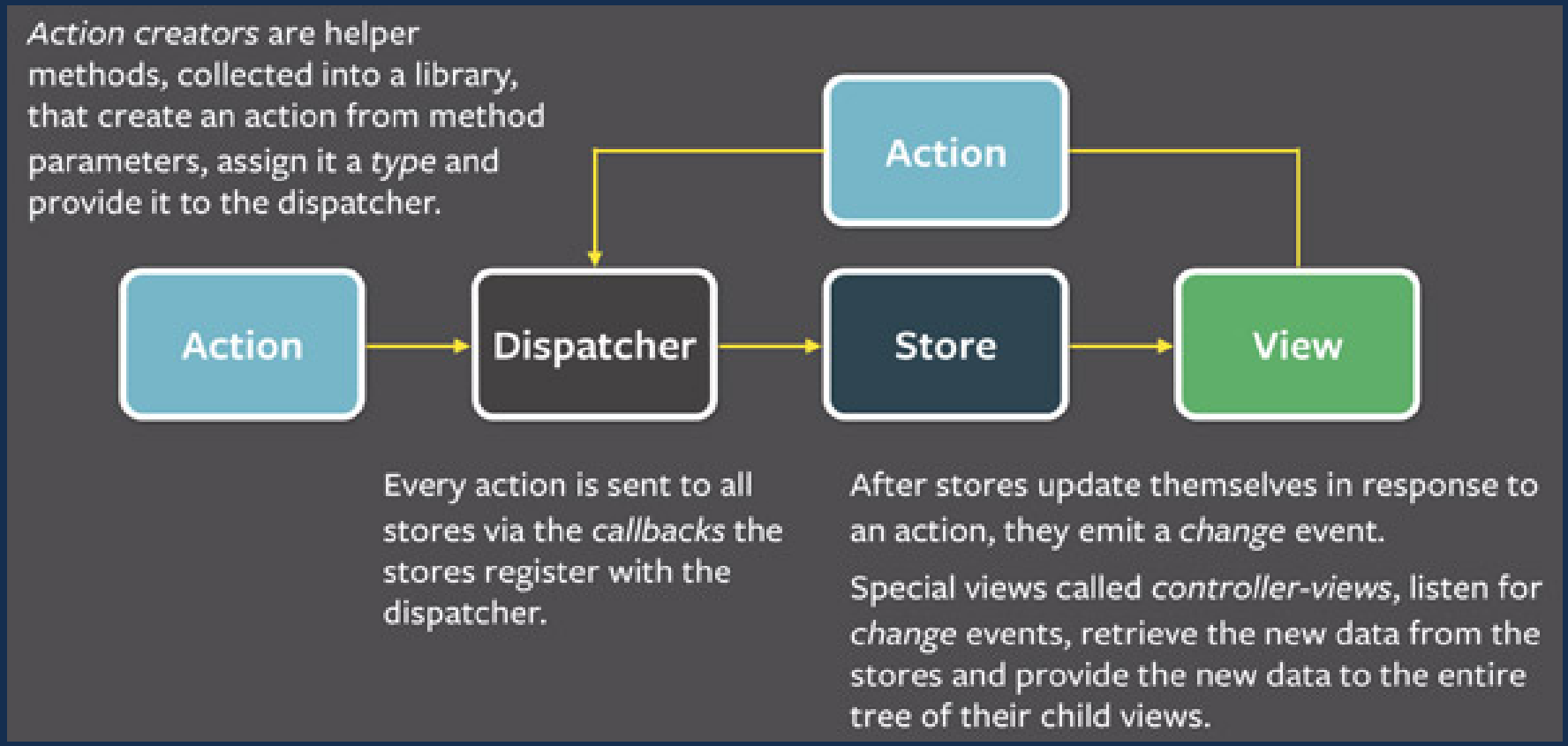
```
ΔM = perform(action)
```

- 当我们把把每次的变更综合起来，可以得到对整个应用状态的表达

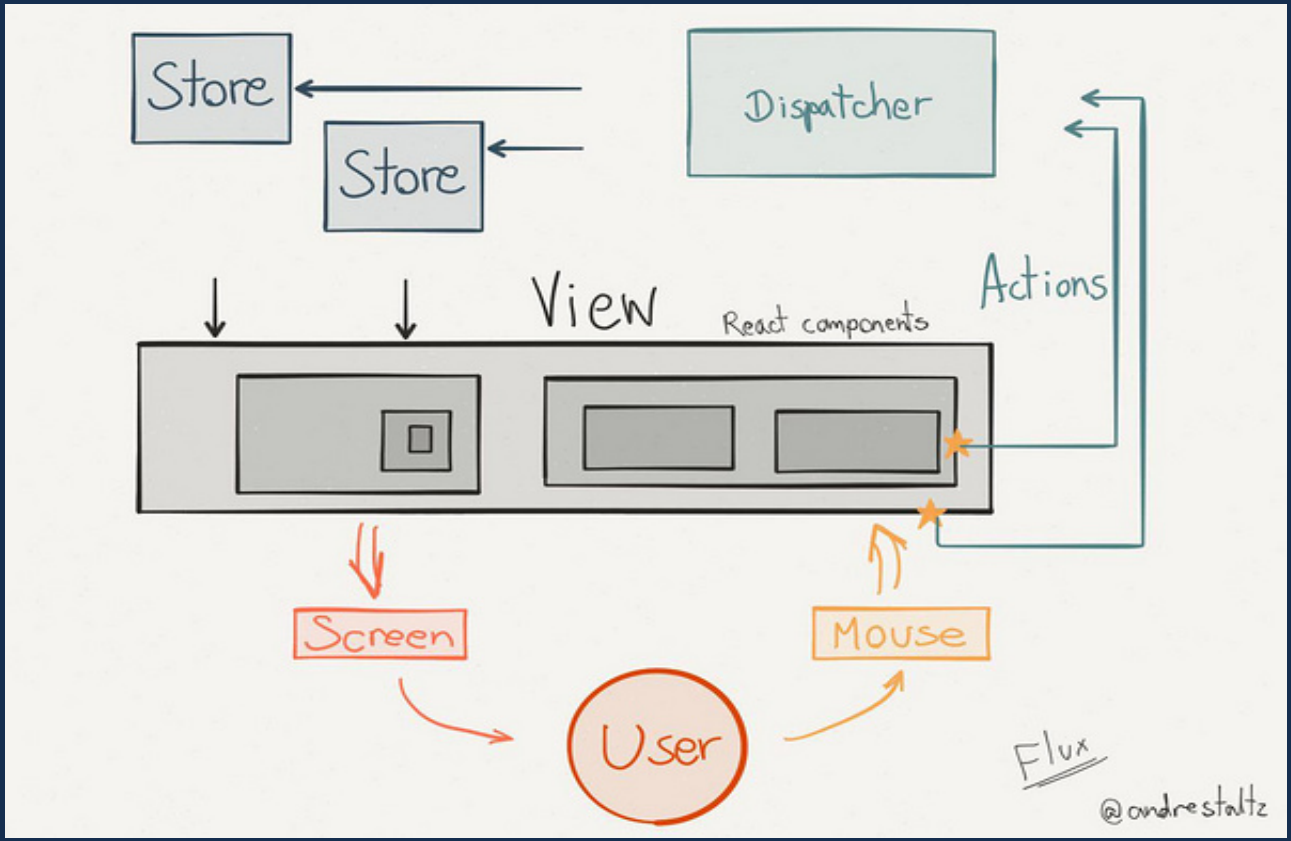
```
state := actions.reduce(reducer, initState)
```

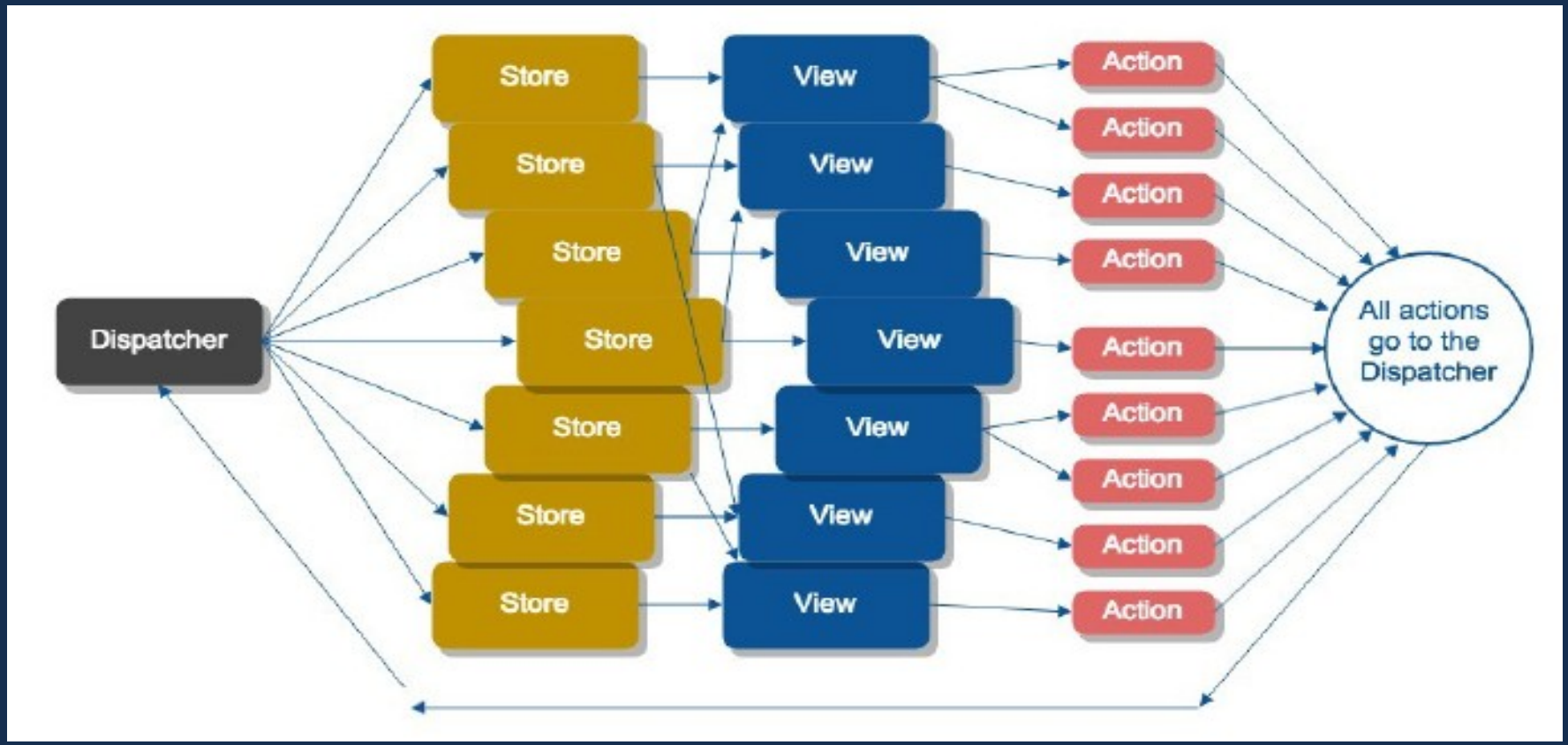
> 在初始状态上，依次叠加后续的变更，所得的就是当前状态。这就是当前最流行的数据流方案Redux的核心理念。（从整体来说，使用Redux，相当于把整个应用都实现为命令模式，一切变动都由命令驱动。）

3 Flux 架构

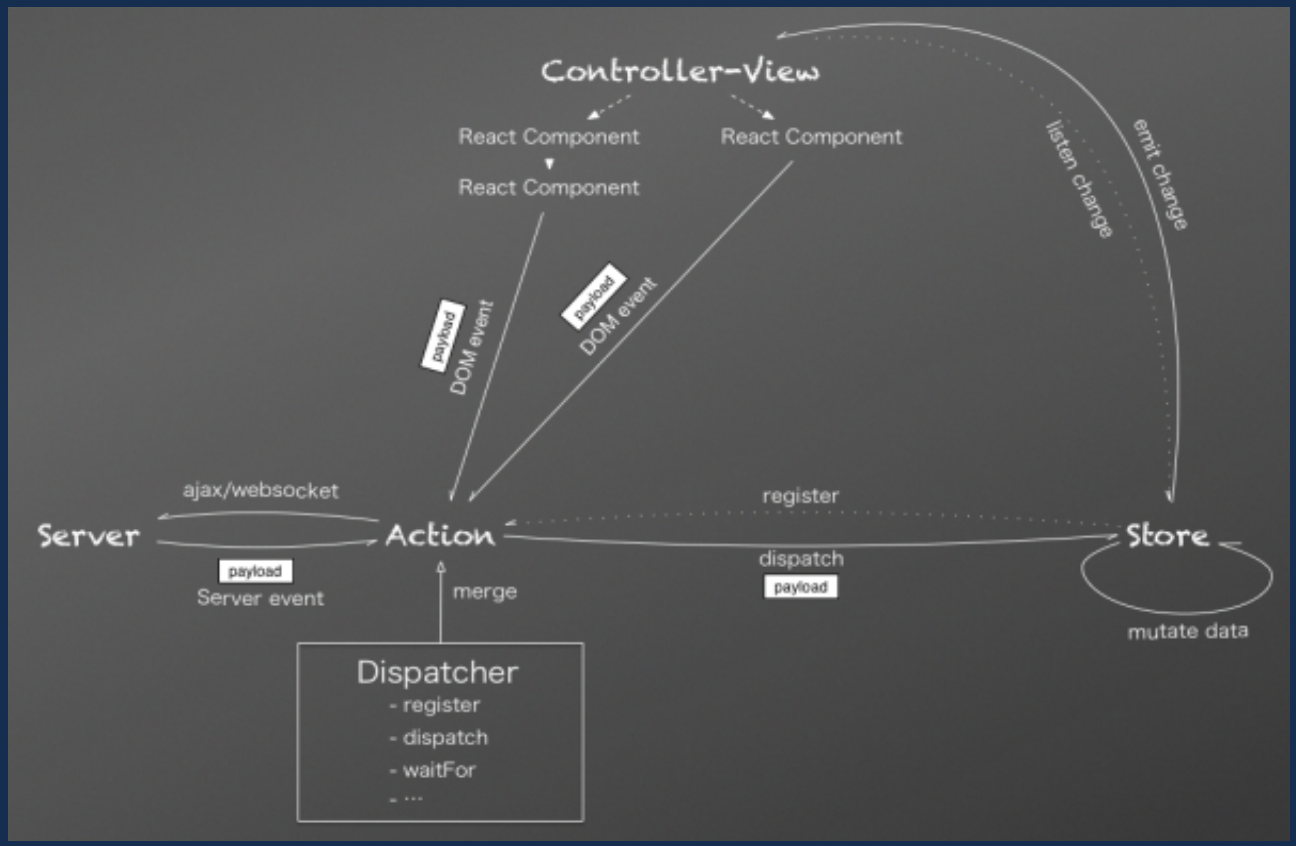


详情请参考： 如何理解 Facebook 的 flux 应用架构？

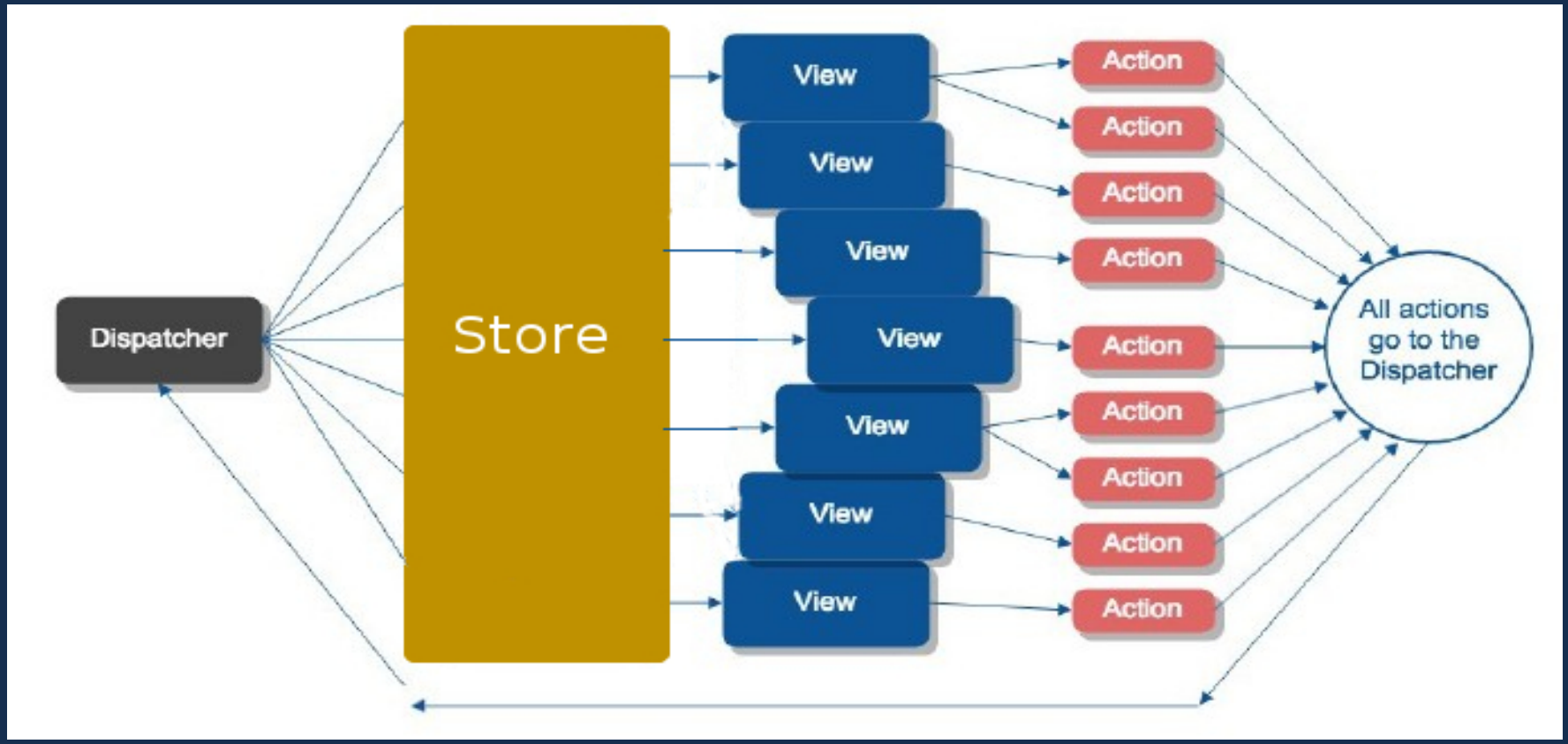

Store
Store
Dispatcher
Actions
View
React components
Screen
Mouse
User
Flux
@andrestaltz


Dispatcher
Store
Store
Store
Store
Store
Store
Store
View
View
View
View
View
View
View
Action
Action
Action
Action
Action
Action
Action
Action
All actions go to the Dispatcher

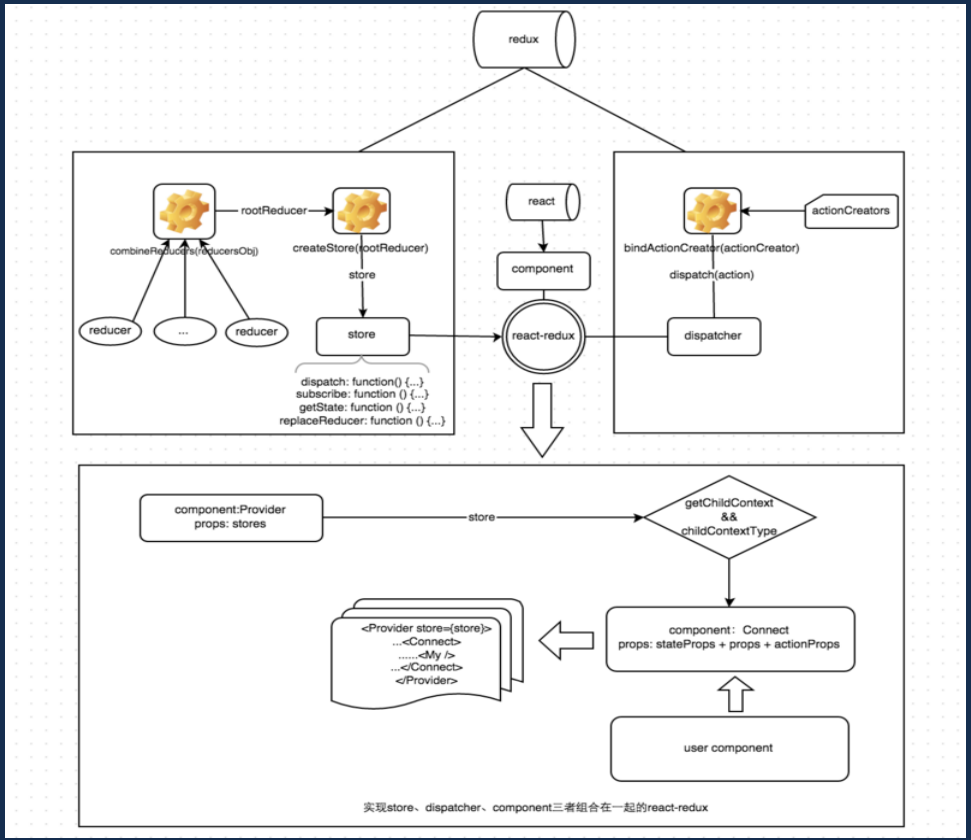
4 React 中的 Flux 模式



5 Redux - JavaScript 状态容器，提供可预测化的状态管理



6 Redux and React



7 Redux 示例

- 源码地址

https://github.com/AidanDai/reddit-thunk

- 本地运行

```
git clone git@github.com:AidanDai/reddit-thunk.git

cd reddit-thunk

npm install

npm run start
```

http://127.0.0.1:8000

8 Redux 的问题 - 组件频繁的重新渲染

> `React Redux` 采取了很多的优化手段，保证组件直到必要时才执行重新渲染。一种是对 `mapStateToProps` 和 `mapDispatchToProps` 生成后传入 `connect` 的 `props` 对象进行浅层的判等检查。遗憾的是，如果当 `mapStateToProps` 调用时都生成新的数组或对象实例的话，此种情况下的浅层判等不会起任何作用。一个典型的示例就是通过 `ID` 数组返回映射的对象引用，如下所示：

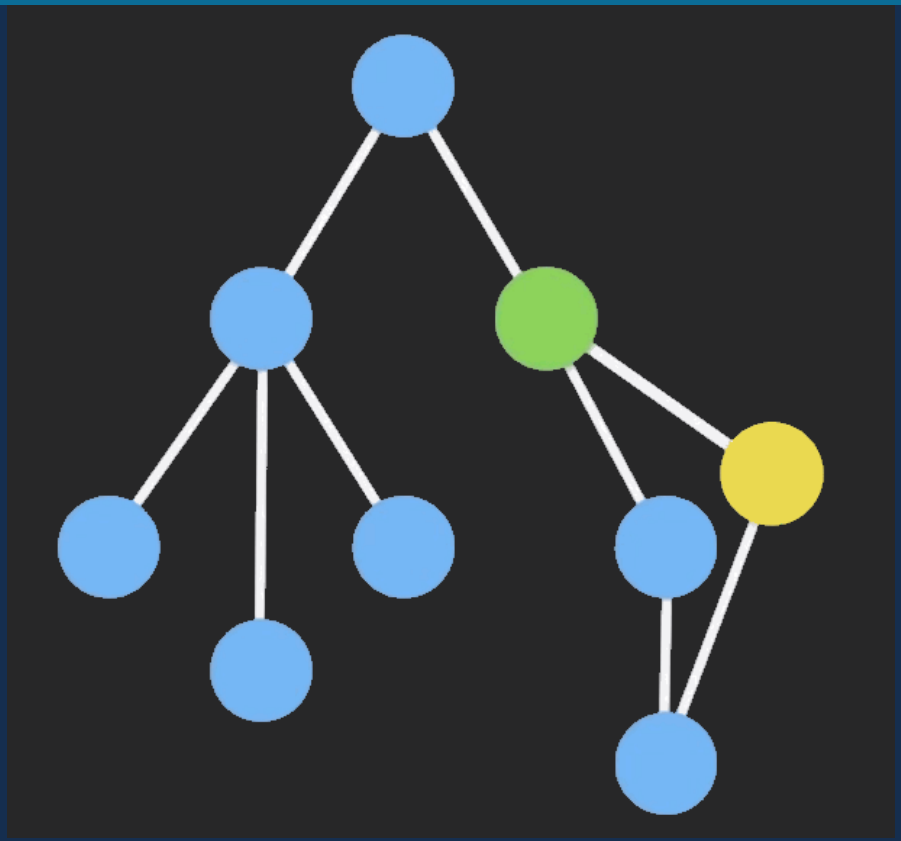
```
const mapStateToProps = (state) => {
  return {
    objects: state.objectIds.map(id => state.objects[id])
  }
}
```

尽管每次数组内都包含了同样的对象引用，数组本身却指向不同的引用，所以浅层判等的检查结果会导致 `React Redux` 重新渲染包装的组件。

- 这种额外的重新渲染也可以避免，使用 `reducer` 将对象数组保存到 `state`，利用 `Reselect` 缓存映射的数组
- 在组件的 `shouldComponentUpdate` 方法中，采用 `_.isEqual` 等对 `props` 进行更深层次的比较（注意在自定义的 `shouldComponentUpdate()` 方法中不要采用了比重新渲染本身更为昂贵的实现）

9 Immutable Data

> `Immutable Data` 就是一旦创建，就不能再被更改的数据。对 `Immutable` 对象的任何修改或添加删除操作都会返回一个新的 `Immutable` 对象。`Immutable` 实现的原理是 `Persistent Data Structure`（持久化数据结构），也就是使用旧数据创建新数据时，要保证旧数据同时可用且不变。同时为了避免 `deepCopy` 把所有节点都复制一遍带来的性能损耗，`Immutable` 使用了 `Structural Sharing`（结构共享），即如果对象树中一个节点发生变化，只修改这个节点和受它影响的父节点，其它节点则进行共享。请看下面动画：



FAQ

- 如何组织 State ？（State 范式化）
- 如何组织代码结构，提高可复用性(划分组件、Reducer 逻辑复用等)？
- 如何理解与使用高阶 reducer ？（Reducer 逻辑复用）

Thanks

参考资料

单页应用的数据流方案探索

如何理解 React Fiber 架构？

如何理解 Facebook 的 flux 应用架构？

React Flux架构简介

redux - 入门实例 TodoList

理解 React，但不理解 Redux，该如何通俗易懂的理解 Redux？

解读redux工作原理

为何组件频繁的重新渲染？

Redux 常见问题

Immutable 详解及 React 中实践