第三版：React 17 道

为什么选择使用框架而不是原生?

框架的好处:

1、 组件化: 其中以 React 的组件化最为彻底,甚至可以到函数级别的原子组件,高度的组件化可以是我们的工程易于维护、易于组合拓展

2、 天然分层: JQuery 时代的代码大部分情况下是面条代码,耦合严重,现代框架不管是 MVC、MVP 还是 MVVM 模式都能帮助我们进行分层，代码解耦更易于读写。

3、 生态: 现在主流前端框架都自带生态,不管是数据流管理架构还是 UI 库都有成熟的解决方案。

4、 开发效率: 现代前端框架都默认自动更新 DOM,而非我们手动操作,解放了开发者的手动 DOM 成本,提高开发效率,从根本上解决了 UI 与状态同步问题.

虚拟 DOM 的优劣如何?

优点:

1、 保证性能下限: 虚拟 DOM 可以经过 diff 找出最小差异,然后批量进行 patch,这种操作虽然比不上手动优化,但是比起粗暴的 DOM 操作性能要好很多,因此虚拟 DOM 可以保证性能下限

2、 无需手动操作 DOM：虚拟 DOM 的 diff 和 patch 都是在一次更新中自动进行的,我们无需手动操作 DOM,极大提高开发效率

3、 跨平台：虚拟 DOM 本质上是 JavaScript 对象,而 DOM 与平台强相关,相比之下虚拟 DOM 可以进行更方便地跨平台操作,例如服务器渲染、移动端开发等等

缺点:

无法进行极致优化：在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化,比如 VScode 采用直接手动操作 DOM 的方式进行极端的性能优化

虚拟 DOM 实现原理?

1、 虚拟 DOM 本质上是 JavaScript 对象,是对真实 DOM 的抽象

2、 状态变更时，记录新树和旧树的差异

3、 最后把差异更新到真正的 dom 中

虚拟 DOM 原理

React 最新的生命周期是怎样的?

React 16 之后有三个生命周期被废弃(但并未删除)

1、 componentWillMount

2、 componentWillReceiveProps

3、 componentWillUpdate

官方计划在 17 版本完全删除这三个函数，只保留 UNSAVE_前缀的三个函数，目的是为了向下兼容，但是对于开发者而言应该尽量避免使用他们，而是使用新增的生命周期函数替代它们

目前 React 16.8 +的生命周期分为三个阶段,分别是挂载阶段、更新阶段、卸载阶段

挂载阶段:

1、 constructor: 构造函数，最先被执行,我们通常在构造函数里初始化 state 对象或者给自定义方法绑定 this

2、 getDerivedStateFromProps: static getDerivedStateFromProps(nextProps, prevState),这是个静态方法,当我们接收到新的属性想去修改我们 state，可以使用 getDerivedStateFromProps

3、 render: render 函数是纯函数，只返回需要渲染的东西，不应该包含其它的业务逻辑,可以返回原生的 DOM、React 组件、Fragment、Portals、字符串和数字、Boolean 和 null 等内容

4、 componentDidMount: 组件装载之后调用，此时我们可以获取到 DOM 节点并操作，比如对 canvas，svg 的操作，服务器请求，订阅都可以写在这个里面，但是记得在 componentWillUnmount 中取消订阅

更新阶段:

1、 getDerivedStateFromProps: 此方法在更新个挂载阶段都可能会调用

2、 shouldComponentUpdate: shouldComponentUpdate(nextProps, nextState),有两个参数 nextProps 和 nextState，表示新的属性和变化之后的 state，返回一个布尔值，true 表示会触发重新渲染，false 表示不会触发重新渲染，默认返回 true,我们通常利用此生命周期来优化 React 程序性能

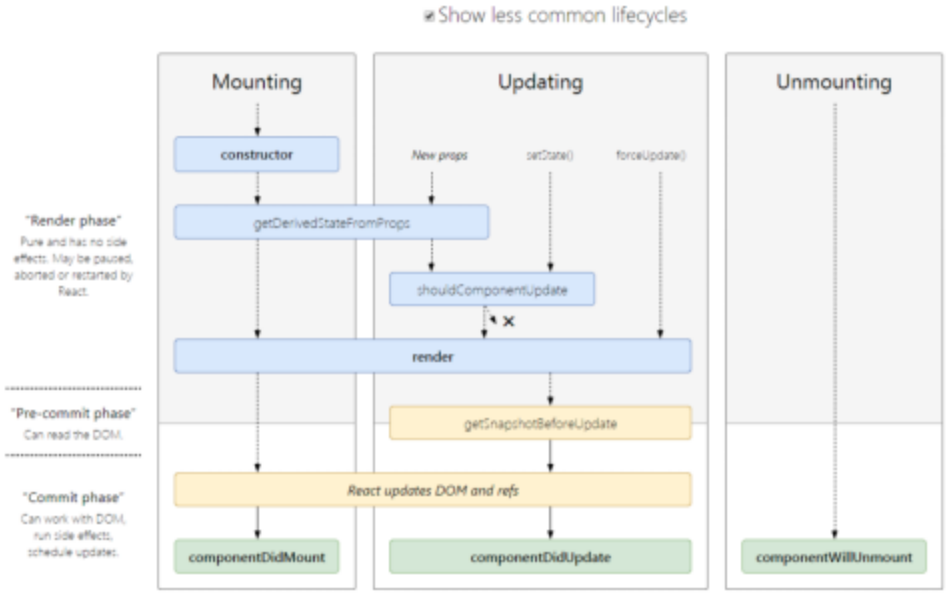
3、 render: 更新阶段也会触发此生命周期

4、 getSnapshotBeforeUpdate: getSnapshotBeforeUpdate(prevProps, prevState),这个方法在 render 之后，componentDidUpdate 之前调用，有两个参数 prevProps 和 prevState，表示之前的属性和之前的 state，这个函数有一个返回值，会作为第三个参数传给 componentDidUpdate，如果你不想要返回值，可以返回 null，此生命周期必须与 componentDidUpdate 搭配使用

5、 componentDidUpdate: componentDidUpdate(prevProps, prevState, snapshot),该方法在 getSnapshotBeforeUpdate 方法之后被调用，有三个参数 prevProps，prevState，snapshot，表示之前的 props，之前的 state，和 snapshot。第三个参数是 getSnapshotBeforeUpdate 返回的,如果触发某些回调函数时需要用到 DOM 元素的状态，则将对比或计算的过程迁移至 getSnapshotBeforeUpdate，然后在 componentDidUpdate 中统一触发回调或更新状态。

卸载阶段:

componentWillUnmount: 当我们的组件被卸载或者销毁了就会调用，我们可以在这个函数里去清除一些定时器，取消网络请求，清理无效的 DOM 元素等垃圾清理工作



React 的请求应该放在哪个生命周期中?

React 的异步请求到底应该放在哪个生命周期里,有人认为在 componentWillMount 中可以提前进行异步请求,避免白屏,其实这个观点是有问题的.

由于 JavaScript 中异步事件的性质，当您启动 API 调用时，浏览器会在此期间返回执行其他工作。当 React 渲染一个组件时，它不会等待 componentWillMount 它完成任何事情 - React 继续前进并继续 render,没有办法“暂停”渲染以等待数据到达。

而且在 componentWillMount 请求会有一系列潜在的问题,首先,在服务器渲染时,如果在 componentWillMount 里获取数据，fetch data 会执行两次，一次在服务端一次在客户端，这造成了多余的请求,其次,在 React 16 进行 React Fiber 重写后,componentWillMount 可能在一次渲染中多次调用.

目前官方推荐的异步请求是在 componentDidmount 中进行.

如果有特殊需求需要提前请求,也可以在特殊情况下在 constructor 中请求:

react 17 之后 componentWillMount 会被废弃,仅仅保留 UNSAFE_componentWillMount

setState 到底是异步还是同步?

先给出答案：有时表现出异步,有时表现出同步

1、 setState 只在合成事件和钩子函数中是“异步”的，在原生事件和 setTimeout 中都是同步的

2、 setState 的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形成了所谓的“异步”，当然可以通过第二个参数 setState(partialState, callback) 中的 callback 拿到更新后的结果

3、 setState 的批量更新优化也是建立在“异步”（合成事件、钩子函数）之上的，在原生事件和 setTimeout 中不会批量更新，在“异步”中如果对同一个值进行多次 setState，setState 的批量更新策略会对其进行覆盖，取最后一次的执行，如果是同时 setState 多个不同的值，在更新时会对其进行合并批量更新

React 组件通信如何实现?

React 组件间通信方式:

1、 父组件向子组件通讯：父组件可以向子组件通过传 props 的方式，向子组件进行通讯

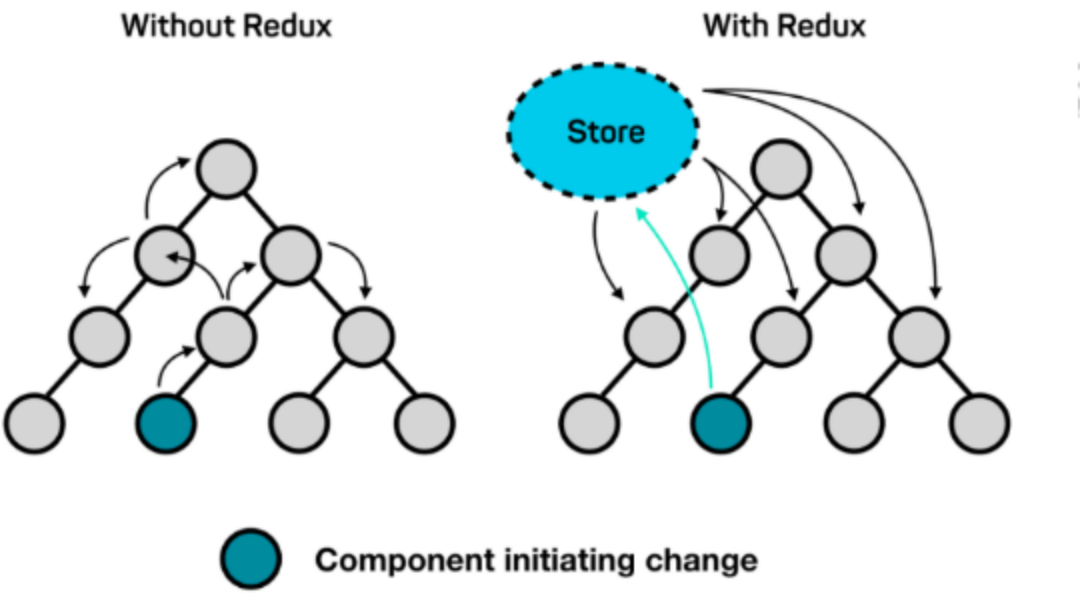
2、 子组件向父组件通讯: props+回调的方式,父组件向子组件传递 props 进行通讯，此 props 为作用域为父组件自身的函数，子组件调用该函数，将子组件想要传递的信息，作为参数，传递到父组件的作用域中

3、 兄弟组件通信：找到这两个兄弟节点共同的父节点,结合上面两种方式由父节点转发信息进行通信

4、 跨层级通信: Context 设计目的是为了共享那些对于一个组件树而言是“全局”的数据，例如当前认证的用户、主题或首选语言,对于跨越多层的全局数据通过 Context 通信再适合不过

5、 发布订阅模式：发布者发布事件，订阅者监听事件并做出反应,我们可以通过引入 event 模块进行通信

6、 全局状态管理工具：借助 Redux 或者 Mobx 等全局状态管理工具进行通信，这种工具会维护一个全局状态中心 Store,并根据不同的事件产生新的状态



React 有哪些优化性能是手段?

性能优化的手段很多时候是通用的详情见前端性能优化加载篇

React 如何进行组件/逻辑复用?

抛开已经被官方弃用的 Mixin,组件抽象的技术目前有三种比较主流:

1、 高阶组件

2、 属性代理

3、 反向继承

4、 渲染属性

5、 react-hooks

mixin、hoc、render props、react-hooks 的优劣如何?

Mixin 的缺陷:

1、 组件与 Mixin 之间存在隐式依赖（Mixin 经常依赖组件的特定方法，但在定义组件时并不知道这种依赖关系）

2、 多个 Mixin 之间可能产生冲突（比如定义了相同的 state 字段）

3、 Mixin 倾向于增加更多状态，这降低了应用的可预测性（The more state in your application, the harder it is to reason about it.），导致复杂度剧增

4、 隐式依赖导致依赖关系不透明，维护成本和理解成本迅速攀升：

5、 难以快速理解组件行为，需要全盘了解所有依赖 Mixin 的扩展行为，及其之间的相互影响

6、 组价自身的方法和 state 字段不敢轻易删改，因为难以确定有没有 Mixin 依赖它

7、 Mixin 也难以维护，因为 Mixin 逻辑最后会被打平合并到一起，很难搞清楚一个 Mixin 的输入输出

HOC 相比 Mixin 的优势:

HOC 通过外层组件通过 Props 影响内层组件的状态，而不是直接改变其 State 不存在冲突和互相干扰,这就降低了耦合度

不同于 Mixin 的打平+合并，HOC 具有天然的层级结构（组件树结构），这又降低了复杂度

HOC 的缺陷:

1、 扩展性限制: HOC 无法从外部访问子组件的 State 因此无法通过 shouldComponentUpdate 滤掉不必要的更新,React 在支持 ES6 Class 之后提供了 React.PureComponent 来解决这个问题

2、 Ref 传递问题: Ref 被隔断,后来的 React.forwardRef 来解决这个问题

3、 Wrapper Hell: HOC 可能出现多层包裹组件的情况,多层抽象同样增加了复杂度和理解成本

4、 命名冲突: 如果高阶组件多次嵌套,没有使用命名空间的话会产生冲突,然后覆盖老属性

5、 不可见性: HOC 相当于在原有组件外层再包装一个组件,你压根不知道外层的包装是啥,对于你是黑盒

Render Props 优点:

上述 HOC 的缺点 Render Props 都可以解决

Render Props 缺陷:

使用繁琐: HOC 使用只需要借助装饰器语法通常一行代码就可以进行复用,Render Props 无法做到如此简单

嵌套过深: Render Props 虽然摆脱了组件多层嵌套的问题,但是转化为了函数回调的嵌套

React Hooks 优点:

1、 简洁: React Hooks 解决了 HOC 和 Render Props 的嵌套问题,更加简洁

2、 解耦: React Hooks 可以更方便地把 UI 和状态分离,做到更彻底的解耦

3、 组合: Hooks 中可以引用另外的 Hooks 形成新的 Hooks,组合变化万千

4、 函数友好: React Hooks 为函数组件而生,从而解决了类组件的几大问题:

1、 this 指向容易错误

2、 分割在不同声明周期中的逻辑使得代码难以理解和维护

3、 代码复用成本高（高阶组件容易使代码量剧增）

React Hooks 缺陷:

1、 额外的学习成本（Functional Component 与 Class Component 之间的困惑）

2、 写法上有限制（不能出现在条件、循环中），并且写法限制增加了重构成本

3、 破坏了 PureComponent、React.memo 浅比较的性能优化效果（为了取最新的 props 和 state，每次 render()都要重新创建事件处函数）

4、 在闭包场景可能会引用到旧的 state、props 值

5、 内部实现上不直观（依赖一份可变的全局状态，不再那么“纯”）

6、React.memo 并不能完全替代 shouldComponentUpdate(因为拿不到 state change，只针对 props change)

你是如何理解 fiber 的?

React Fiber 是一种基于浏览器的**单线程调度算法**.

React 16 之前 ，reconcilation 算法实际上是递归，想要中断递归是很困难的，React 16 开始使用了循环来代替之前的递归.

Fiber：一种将 **recocilation （递归 diff），拆分成无数个小任务的算法；它随时能够停止，恢复。停止恢复的时机取决于当前的一帧（16ms）内，还有没有足够的时间允许计算。**

你对 Time Slice 的理解?

时间分片

1、 React 在渲染（render）的时候，不会阻塞现在的线程

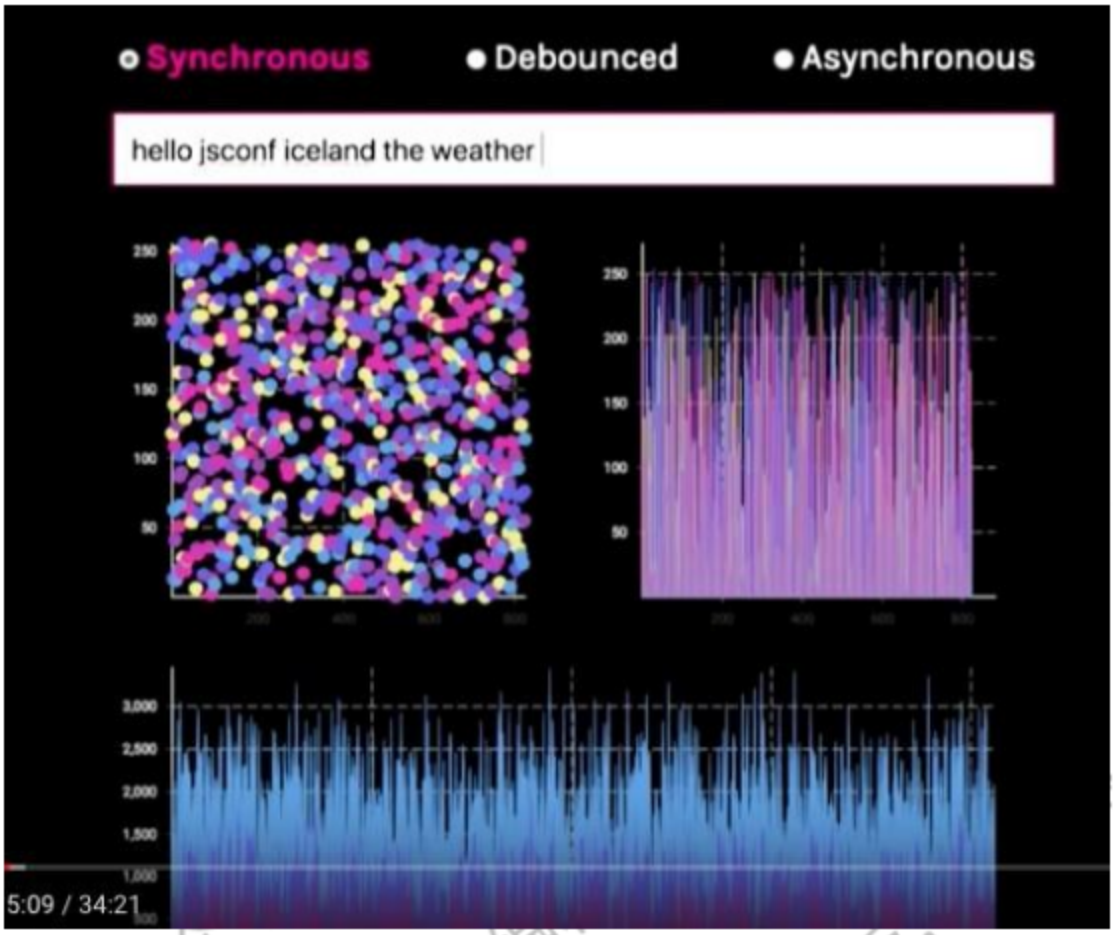
2、 如果你的设备足够快，你会感觉渲染是同步的

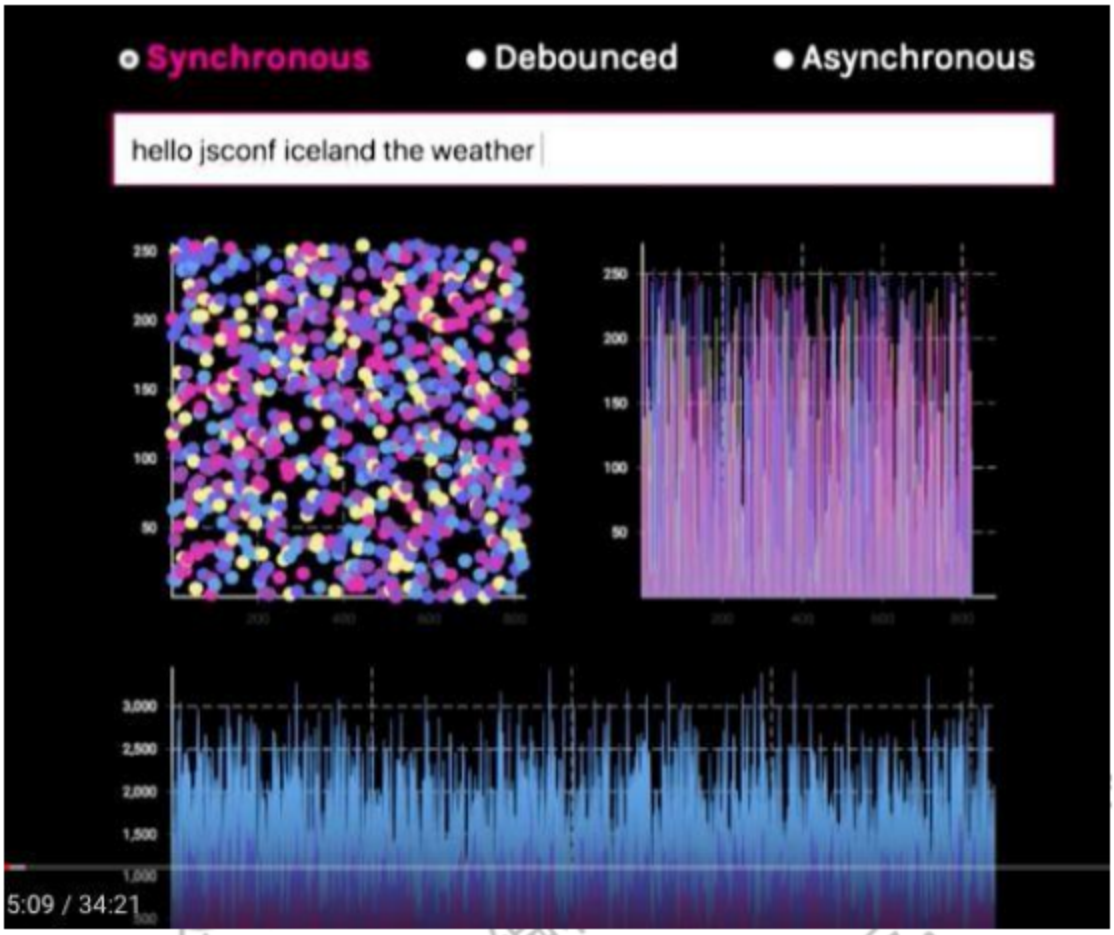
3、 如果你设备非常慢，你会感觉还算是灵敏的

4、 虽然是异步渲染，但是你将会看到完整的渲染，而不是一个组件一行行的渲染出来

5、 同样书写组件的方式

也就是说，这是 React 背后在做的事情，对于我们开发者来说，是透明的，具体是什么样的效果呢？

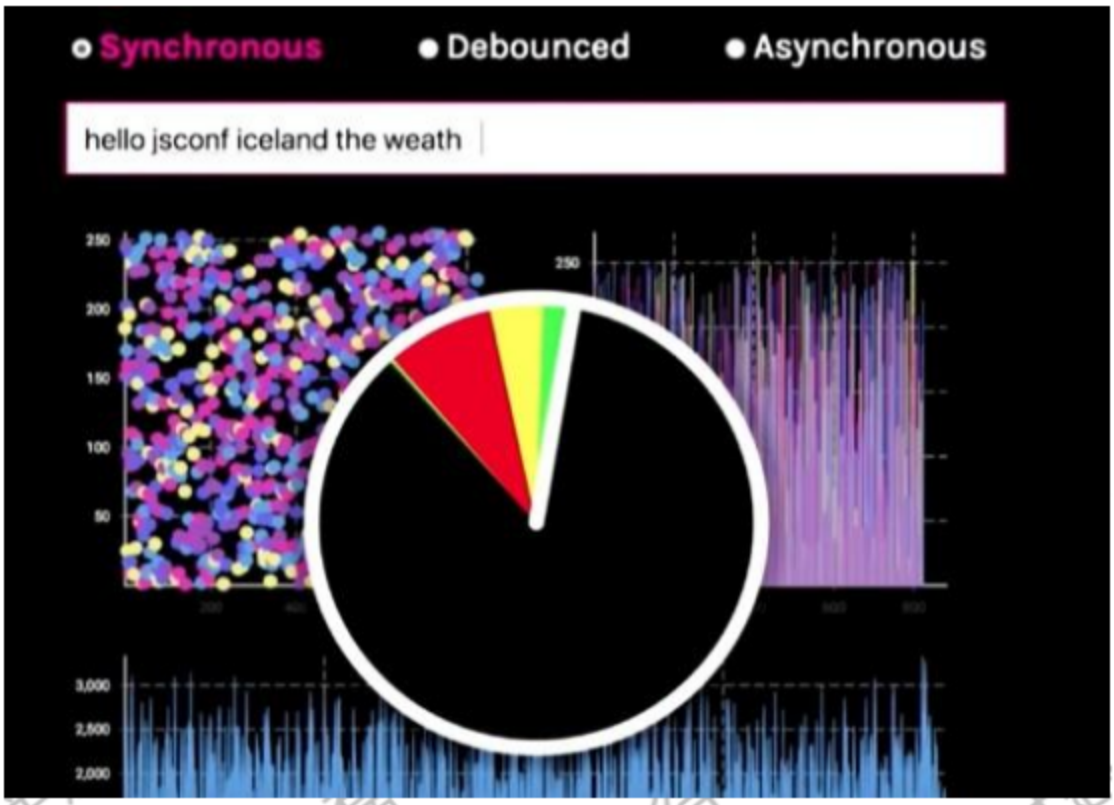


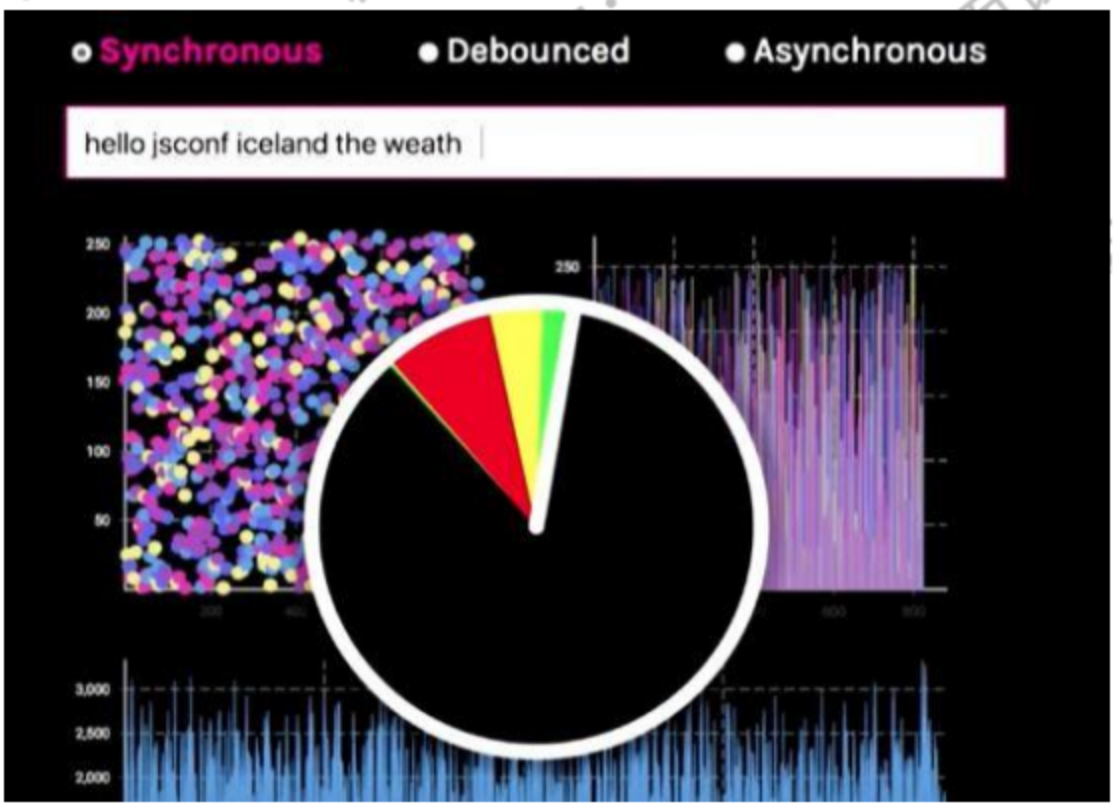


有图表三个图表，有一个输入框，以及上面的三种模式

这个组件非常的巨大，而且在输入框每次输入东西的时候，就会进去一直在渲染。**为了更好的看到渲染的性能，Dan 为我们做了一个表。

我们先看看，同步模式：

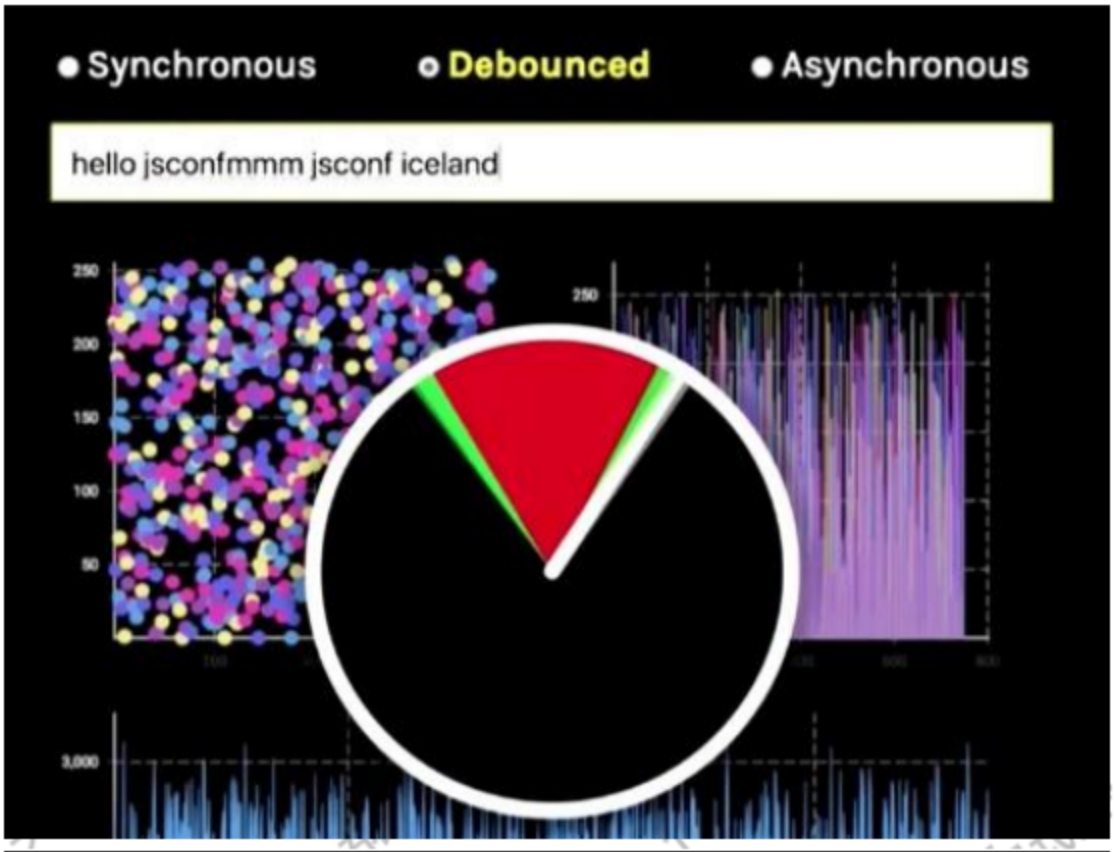


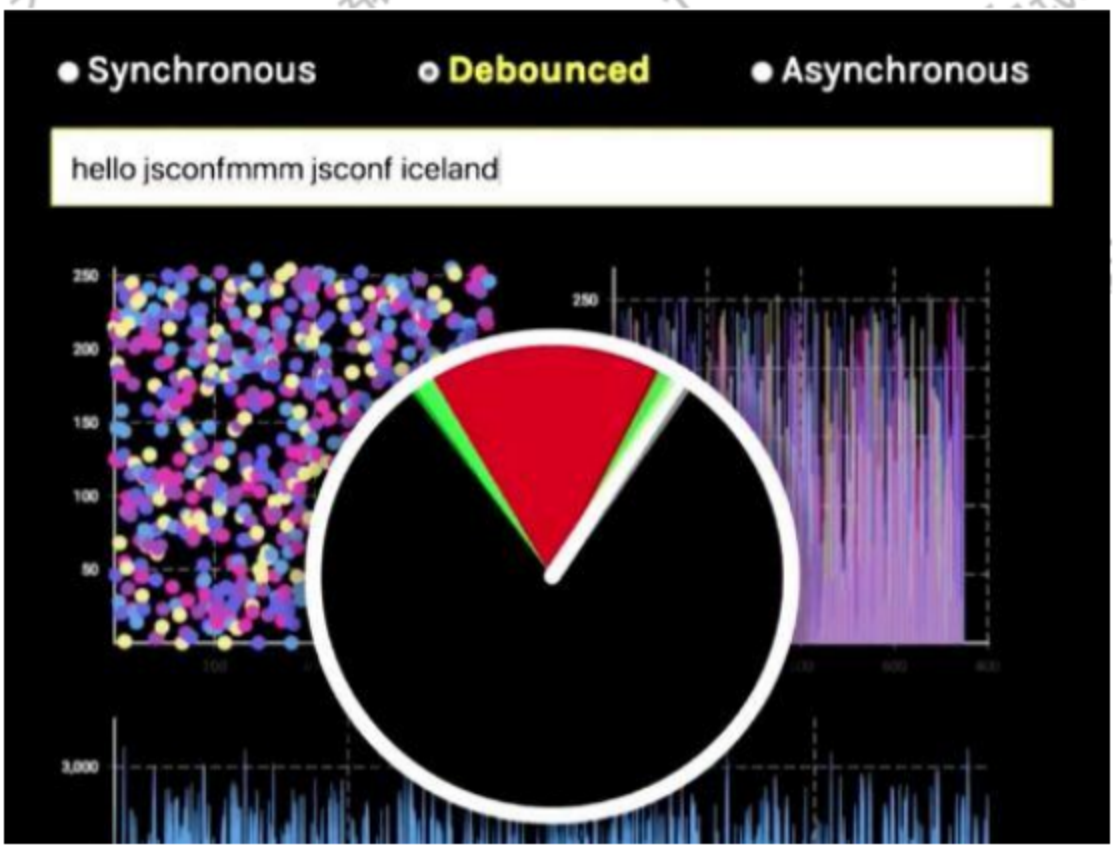


同步模式下，我们都知道，我们没输入一个字符，React 就开始渲染，当 React

渲染一颗巨大的树的时候，是非常卡的，所以才会有 shouldUpdate 的出现，在这里 Dan 也展示了，这种卡！

我们再来看看第二种（Debounced 模式）：



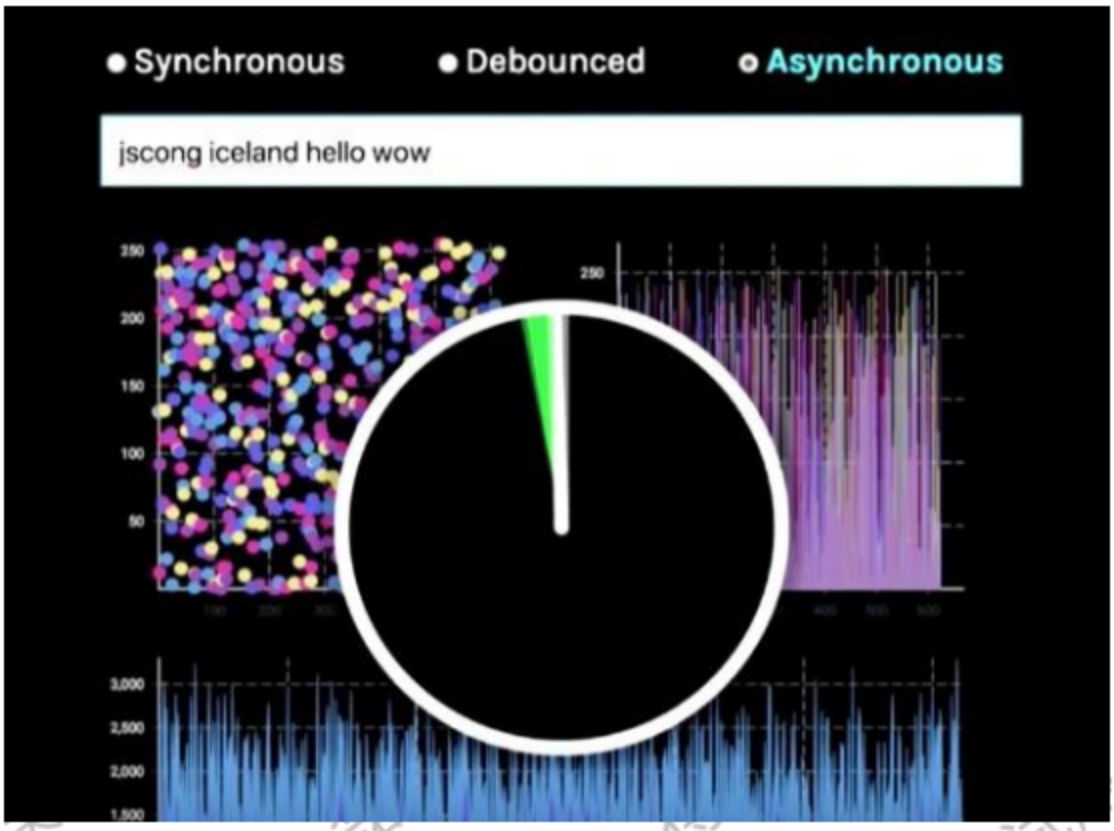


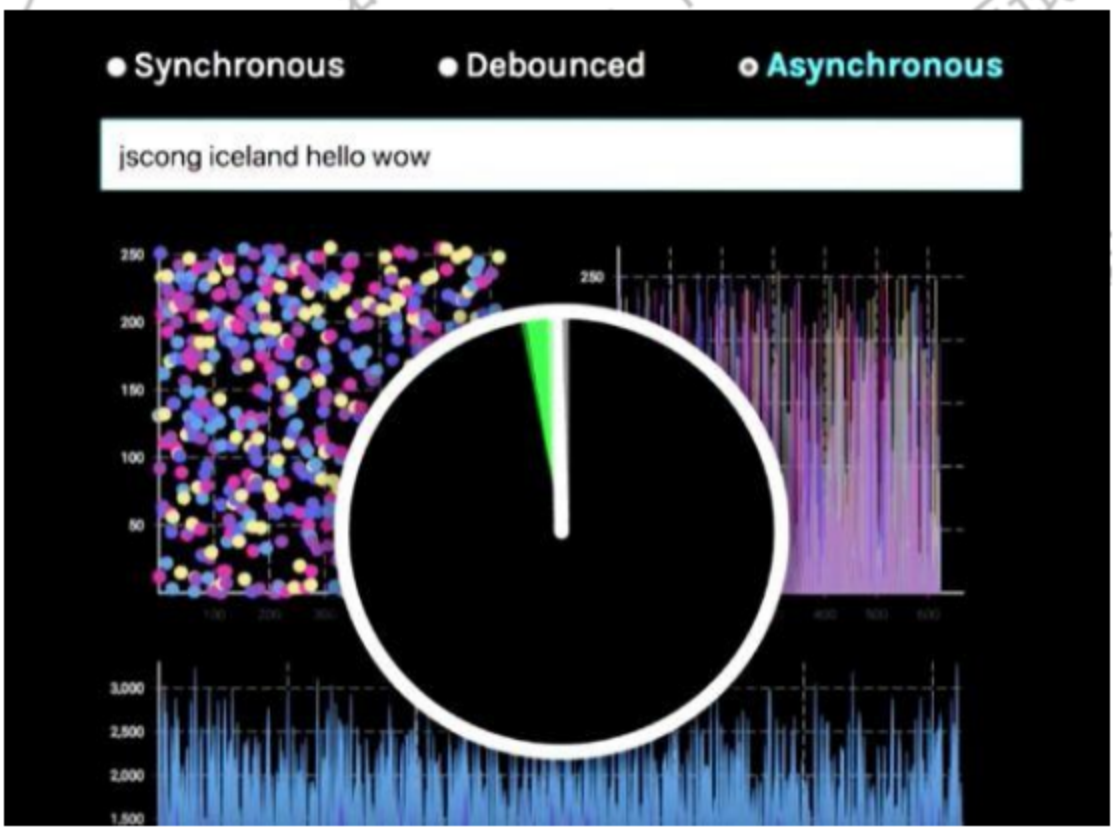
Debounced 模式简单的来说，就是延迟渲染，比如，当你输入完成以后，再开始

渲染所有的变化。

这么做的坏处就是，至少不会阻塞用户的输入了，但是依然有非常严重的卡顿。

切换到异步模式：





异步渲染模式就是不阻塞当前线程，继续跑。在视频里可以看到所有的输入，表上都会是原谅色的。

时间分片正是基于可随时打断、重启的 Fiber 架构,可打断当前任务,优先处理紧急且重要的任务,保证页面的流畅运行.

redux 的工作流程?

首先，我们看下几个核心概念：

1、 Store：保存数据的地方，你可以把它看成一个容器，整个应用只能有一个 Store。

2、 State：Store 对象包含所有数据，如果想得到某个时点的数据，就要对 Store 生成快照，这种时点的数据集合，就叫做 State。

3、 Action：State 的变化，会导致 View 的变化。但是，用户接触不到 State，只能接触到 View。所以，State 的变化必须是 View 导致的。Action 就是 View 发出的通知，表示 State 应该要发生变化了。

4、 Action Creator：View 要发送多少种消息，就会有多少种 Action。如果都手写，会很麻烦，所以我们定义一个函数来生成 Action，这个函数就叫 Action Creator。

5、 Reducer：Store 收到 Action 以后，必须给出一个新的 State，这样 View 才会发生变化。这种 State 的计算过程就叫做 Reducer。Reducer 是一个函数，它接受 Action 和当前 State 作为参数，返回一个新的 State。

6、 dispatch：是 View 发出 Action 的唯一方法。

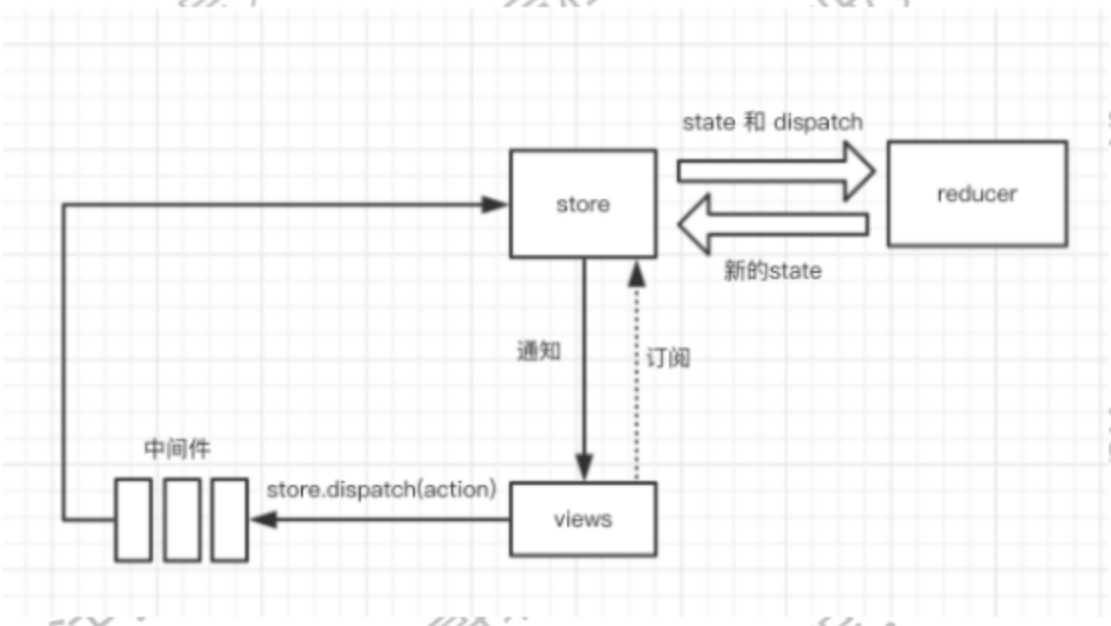
然后我们过下整个工作流程：

1、 首先，用户（通过 View）发出 Action，发出方式就用到了 dispatch 方法。

2、 然后，Store 自动调用 Reducer，并且传入两个参数：当前 State 和收到的 Action，Reducer 会返回新的 State

3、 State 一旦有变化，Store 就会调用监听函数，来更新 View。

到这儿为止，一次用户交互流程结束。可以看到，在整个流程中数据都是单向流动的，这种方式保证了流程的清晰。



react-redux 是如何工作的?

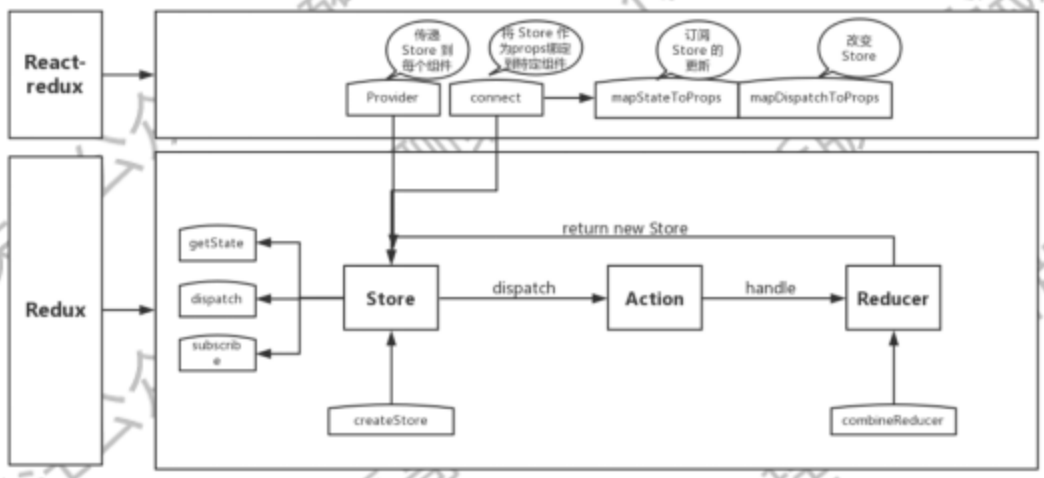
1、 Provider: Provider 的作用是从最外部封装了整个应用，并向 connect 模块传递 store

2、 connect: 负责连接 React 和 Redux

1、 获取 state: connect 通过 context 获取 Provider 中的 store，通过 store.getState()获取整个 store tree 上所有 state

2、 包装原组件：将 state 和 action 通过 props 的方式传入到原组件内部 wrapWithConnect 返回一个 ReactComponent 对象 Connect，Connect 重新 render 外部传入的原组件 WrappedComponent，并把 connect 中传入的 mapStateToProps, mapDispatchToProps 与组件上原有的 props 合并后，通过属性的方式传给 WrappedComponent

3、 监听 store tree 变化: connect 缓存了 store tree 中 state 的状态,通过当前 state 状态和变更前 state 状态进行比较,从而确定是否调用 this.setState()方法触发 Connect 及其子组件的重新渲染



redux 与 mobx 的区别?

两者对比:

1、 redux 将数据保存在单一的 store 中，mobx 将数据保存在分散的多个 store 中

2、 redux 使用 plain object 保存数据，需要手动处理变化后的操作；mobx 适用 observable 保存数据，数据变化后自动处理响应的操作

3、 redux 使用不可变状态，这意味着状态是只读的，不能直接去修改它，而是应该返回一个新的状态，同时使用纯函数；mobx 中的状态是可变的，可以直接对其进行修改

4、 mobx 相对来说比较简单，在其中有很多的抽象，mobx 更多的使用面向对象的编程思维；redux 会比较复杂，因为其中的函数式编程思想掌握起来不是那么容易，同时需要借助一系列的中间件来处理异步和副作用

5、 mobx 中有更多的抽象和封装，调试会比较困难，同时结果也难以预测；而 redux 提供能够进行时间回溯的开发工具，同时其纯函数以及更少的抽象，让调试变得更加的容易

场景辨析:

基于以上区别,我们可以简单得分析一下两者的不同使用场景.

mobx 更适合数据不复杂的应用: mobx 难以调试,很多状态无法回溯,面对复杂度高的应用时,往往力不从心.

redux 适合有回溯需求的应用：比如一个画板应用、一个表格应用，很多时候需要撤销、重做等操作，由于 redux 不可变的特性，天然支持这些操作.

mobx 适合短平快的项目: mobx 上手简单,样板代码少,可以很大程度上提高开发效率.

当然 mobx 和 redux 也并不一定是非此即彼的关系,你也可以在项目中用 redux 作为全局状态管理,用 mobx 作为组件局部状态管理器来用.

redux 中如何进行异步操作?

当然,我们可以在 componentDidmount 中直接进行请求无须借助 redux.

但是在一定规模的项目中,上述方法很难进行异步流的管理,通常情况下我们会借助 redux 的异步中间件进行异步处理.

redux 异步流中间件其实有很多,但是当下主流的异步中间件只有两种 redux-thunk、redux-saga，当然 redux-observable 可能也有资格占据一席之地,其余的异步中间件不管是社区活跃度还是 npm 下载量都比较差了.

redux 异步中间件之间的优劣?

redux-thunk 优点:

1、 体积小: redux-thunk 的实现方式很简单,只有不到 20 行代码

2、 使用简单: redux-thunk 没有引入像 redux-saga 或者 redux-observable 额外的范式,上手简单

redux-thunk 缺陷:

1、 样板代码过多：与 redux 本身一样,通常一个请求需要大量的代码,而且很多都是重复性质的

2、 耦合严重：异步操作与 redux 的 action 偶合在一起,不方便管理

3、 功能孱弱：有一些实际开发中常用的功能需要自己进行封装

redux-saga 优点：

1、异步解耦：异步操作被被转移到单独 saga.js 中，不再是掺杂在 action.js 或 component.js 中

2、 action 摆脱 thunk function：dispatch 的参数依然是一个纯粹的 action (FSA)，而不是充满 “黑魔法” thunk function

3、 异常处理：受益于 generator function 的 saga 实现，代码异常/请求失败都可以直接通过 try/catch 语法直接捕获处理

4、功能强大：redux-saga 提供了大量的 Saga 辅助函数和 Effect 创建器供开发者使用,开发者无须封装或者简单封装即可使用

5、 灵活：redux-saga 可以将多个 Saga 可以串行/并行组合起来,形成一个非常实用的异步 flow

6、 易测试，提供了各种 case 的测试方案，包括 mock task，分支覆盖等等

redux-saga 缺陷：

1、 额外的学习成本：redux-saga 不仅在使用难以理解的 generator function,而且有数十个 API,学习成本远超 redux-thunk,最重要的是你的额外学习成本是只服务于这个库的,与 redux-observable 不同,redux-observable 虽然也有额外学习成本但是背后是 rxjs 和一整套思想

2、 体积庞大：体积略大,代码近 2000 行，min 版 25KB 左右

3、 功能过剩：实际上并发控制等功能很难用到,但是我们依然需要引入这些代码

4、 ts 支持不友好：yield 无法返回 TS 类型

redux-observable 优点：

1、 功能最强：由于背靠 rxjs 这个强大的响应式编程的库,借助 rxjs 的操作符,你可以几乎做任何你能想到的异步处理

2、 背靠 rxjs：由于有 rxjs 的加持,如果你已经学习了 rxjs,redux-observable 的学习成本并不高,而且随着 rxjs 的升级 redux-observable 也会变得更强大

redux-observable 缺陷：

学习成本奇高：如果你不会 rxjs,则需要额外学习两个复杂的库

社区一般：redux-observable 的下载量只有 redux-saga 的 1/5,社区也不够活跃,在复杂异步流中间件这个层面 redux-saga 仍处于领导地位