前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

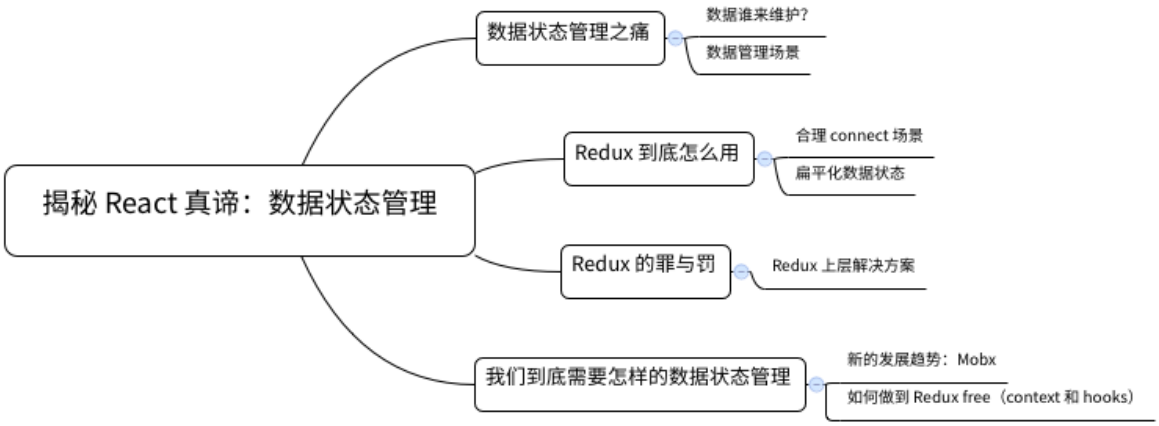
查看详情 >

揭秘 React 真谛：数据状态管理

如果说组件是 React 应用的骨骼，那么数据就是 React 应用的血液。单向数据流就像血液在应用体中穿梭。处理数据向来不是一件简单的事情，良好的数据状态管理不仅需要经验的积累，更是设计能力的反应。目前来看 Redux 无疑能够将数据状态理清，与此同时 Vue 阵营模仿 Redux 的 Vuex 也起到了相同的效果。这一讲我们就来谈谈数据状态管理，了解 Redux 的真谛，并分析其利弊和上层解决方案。

如果说组件是 React 应用的骨骼，那么数据就是 React 应用的血液。单向数据流就像血液在应用体中穿梭。处理数据向来不是一件简单的事情，良好的数据状态管理不仅需要经验的积累，更是设计能力的反应。目前来看 Redux 无疑能够将数据状态理清，与此同时 Vue 阵营模仿 Redux 的 Vuex 也起到了相同的效果。这一讲我们就来谈谈数据状态管理，了解 Redux 的真谛，并分析其利弊和上层解决方案。」

相关知识点如下：



数据状态管理之痛

我们先思考一个问题，为什么需要数据状态管理，数据状态管理到底在解决什么样的问题。这其实是框架、组件化带来的概念，让我们回到最初的起点，还是那个简单的案例：

点击页面中一处「收藏」之后，页面里其他「收藏」按钮也需要切换为「已收藏」状态：

如果没有数据状态，也许我们需要：

```
const btnEle1 = $('#btn1')
```

```
const btnEle2 = $('#btn2')


btnEle1.on('click', () => {
    if (btnEle.textContent === '已收藏') {
        return
    }
    btnEle1.textContent = '已收藏'
    btnEle2.textContent = '已收藏'
})

btnEle2.on('click', () => {
    if (btnEle2.textContent === '已收藏') {
        return
    }
    btnEle1.textContent = '已收藏'
    btnEle2.textContent = '已收藏'
})
```

这只是两个按钮的情况，处理起来就非常混乱难以维护了，这种情况非常容易滋生 bugs。

现代化的框架解决这个问题的思路是组件化，组件依赖数据，对应这个场景数据状态就是简单的：

```
hasMarked: false / true
```

根据这个数据，所有的收藏组件都可以响应正确的视图操作。我们把面条式的代码转换成可维护的代码，重中之重就成了数据的管理，这就是数据状态的雏形。但是数据一旦庞大起来，如何和组件形成良好的交互就是一门学问了。比如我们要思考：

一个组件需要和另一个组件共享状态

一个组件需要改变另一个组件的状态

以 React 为例，其他框架类似，如果 React 或者 Vue 自己来维护这些数据，数据状态就是一个对象，并且这个对象在组件之间要互相修改，及其混乱。

接着我们衍生出这样的问题：hasMarked 这类数据到底是应该放在 state 中维护，还是借助数据状态管理类库，比如在 Redux 中维护呢？至少这样一来，数据源是单一的，数据状态和组件是解耦的，也更加方便开发者进行调试和扩展数据。

数据谁来维护?

我们以 React state 和 Redux 为例，继续分析上面抛出的「数据谁来维护？」问题：

React 中 state 维护数据在组件内部，这样当某项 state 需要与其他组件共享时，我们可以通过 props 来完成组件间通讯。实践上来看，这就需要相对顶层的组件维护共享的 state 并提供修改此项 state 的方法，state 本身和修改方法都需要通过 props 传递给子孙组件。

使用 Redux 的时候，state 维护在 Redux store 当中。任何需要访问并更新 state 的组件都需要感知或订阅 Redux store，这通常借助容器组件来完成。Redux 对于数据采用集中管理的方式。

我尝试从数据持久度、数据消费范围上来回答这个问题。

首先，数据持久度上，不同状态数据在持久度上大体可以分为三类：

快速变更型

中等持续型

长远稳定型

快速变更型， 这类数据在应用中代表了某些原子级别的信息，且显著特点是变更频率最快。比如一个文本输入框数据值，可能随着用户输入在短时间内持续发生变化。这类数据显然更适合维护在 React 组件之内。

中等持续型数据， 在用户浏览或使用应用时，这类数据往往会在页面刷新前保持稳定。比如从异步请求接口通过 Ajax 方式得来的数据；又或者用户在个人中心页，编辑信息提交的数据。这类数据较为通用，也许会被不同组件所需求。在 Redux store 中维护，并通过 connect 方法进行连接，是不错的选择。

长远稳定型数据， 指在页面多次刷新或者多次访问期间都保持不变的数据。因为 Redux store 会在每次页面挂载后都重新生成一份，因此这种类型的数据显然应该存储在 Redux 以外其他地方，比如服务端数据库或者 local storage。

下面，我们从另一维度：数据消费范围来分析。数据特性体现在消费层面，即有多少组件需要使用。我们以此来区分 React 和 Redux 的不同分工。**广义上，越多组件需要消费同一种数据，那么这种数据维护在 Redux store 当中就越合理；反之，如果某种数据隔离于其他数据，只服务于应用中某单一部分，那么由 React 维护更加合理。**

具体来看，共享的数据在 React 当中，应该存在于高层组件，由此组件进行一层层传递。如果在 props 传递深度上，只需要一两个层级就能满足消费数据的组件需求，这样的跨度是可以接受的；反之，如果跨越层级很多，那么关联到的所有中间层级组件都需要进行接力赛式的传递，这样显然会增加很多乏味的传递代码，也破坏了中间组件的复用性。这个时候，使用 Redux 维护共享状态，合理设置容器组件，通过 connect 来打通数据，就是一种更好的方式。

一些完全不存在父子关系的组件，如果需要共享数据，比如前面提到过的一个页面需要多处展示用户头像。这往往会造成数据辐射分散的问题，对于 React 模式的状态管理十分不利。在这种场景下，使用 Redux 同样是更好的选择。

最后一点，如果你的应用有跟踪状态的功能，比如需要完成「重放」，「返回」或者「Redo/Undo」类似需求，那么 Redux 无疑是最佳选择。因为 Redux 天生擅长于此：每一个 action 都描述了数据状态的改变和更新，数据的集中管理非常方便进行记录。

最后，什么情况下该使用哪种数据管理方式，是 React 维护 state 还是 Redux 集中管理，这个讨论不会有唯一定论。这需要开发者对于 React、Redux 有深入理解，并结合场景需求完成选择。

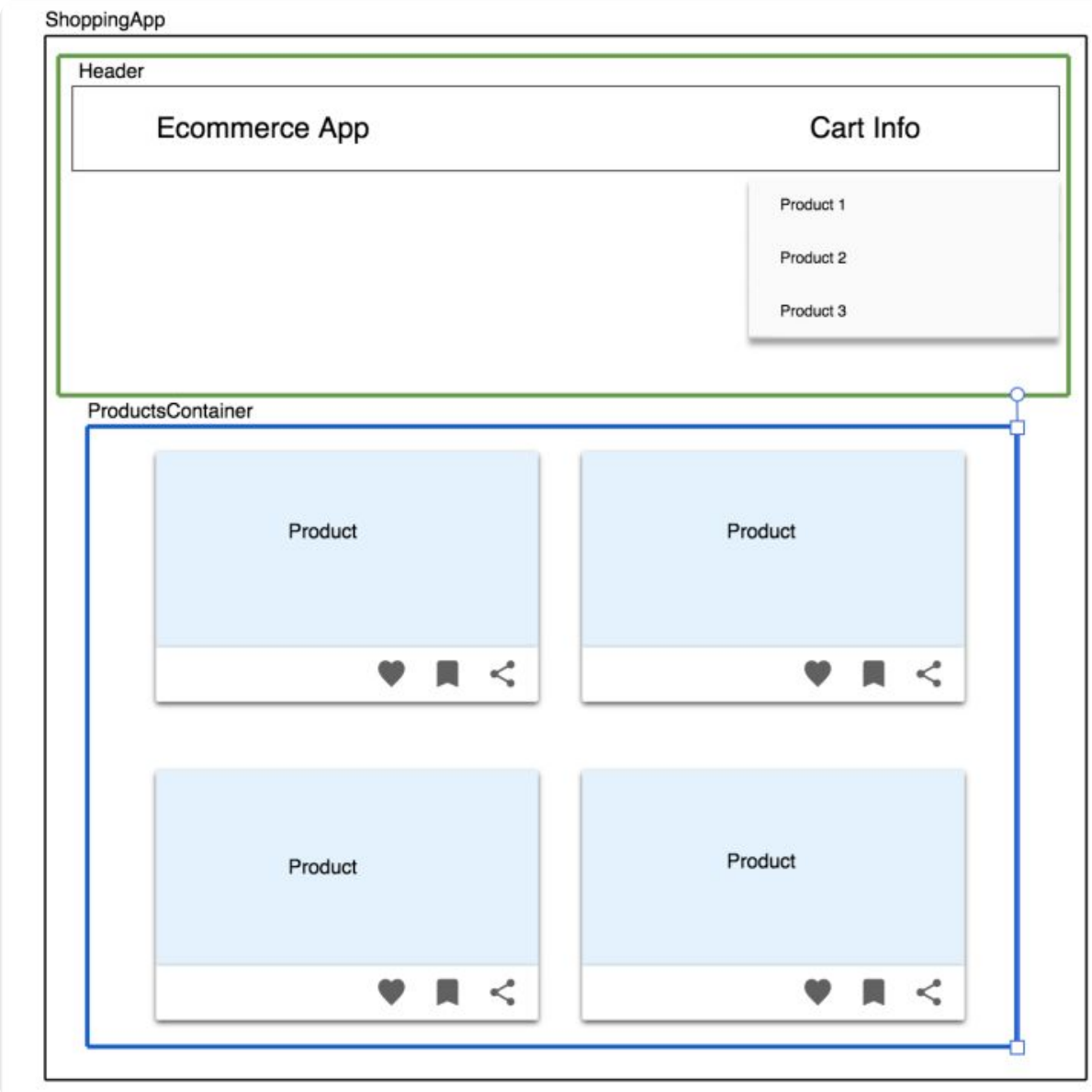
上面的 Redux 可以被任何一个数据管理类库所取代，也就是说，适合放在 Redux 中的数据，如果开发者没有使用 Redux，而使用了 Mobx，那么也应该放在 Mobx store 中。

数据管理场景

我们来看一个场景来加深理解。

Redux 到底怎么用

某电商网站，应用页面骨架如下：



对应代码：

```
        // 遍历渲染每一个商品
```

其中，ProductsContainer 组件负责渲染每一个商品条目：

```
import Product from './Product'
```

```
export default class ProductsContainer extends Component {
 constructor(props) {
   super(props);
   this.state = {
     products: [
       '商品 1',
       '商品 2',
       '商品 3'
     ]
   }
 }
 renderProducts() {
   return this.state.products.map((product) => {
     return
   })
 }
 render() {
   return (
```

```
        {this.renderProducts()}

    )
  }
}
```

Product 组件作为 UI 组件 / 展示组件，负责接受数据、展现数据，Product 即可以用函数式 / 无状态组件完成：

```
import React, { Component } from 'react'

export default class Product extends Component {
  render() {
    return (

        {this.props.name}

    )
  }
}
```

这样的设计，完全使用 React state 就可以完成，且合理高效。

但是，如果商品有「立即购买」按钮，点击购买之后加入商品到购物车（对应上面 Cart Info 部分）。这时候需要注意，购物车的商品信息会在更多页面被消费。比如：

当前页面右上角需要展示购物车里的商品数目

购物车页面本身

支付前 checkout 页面

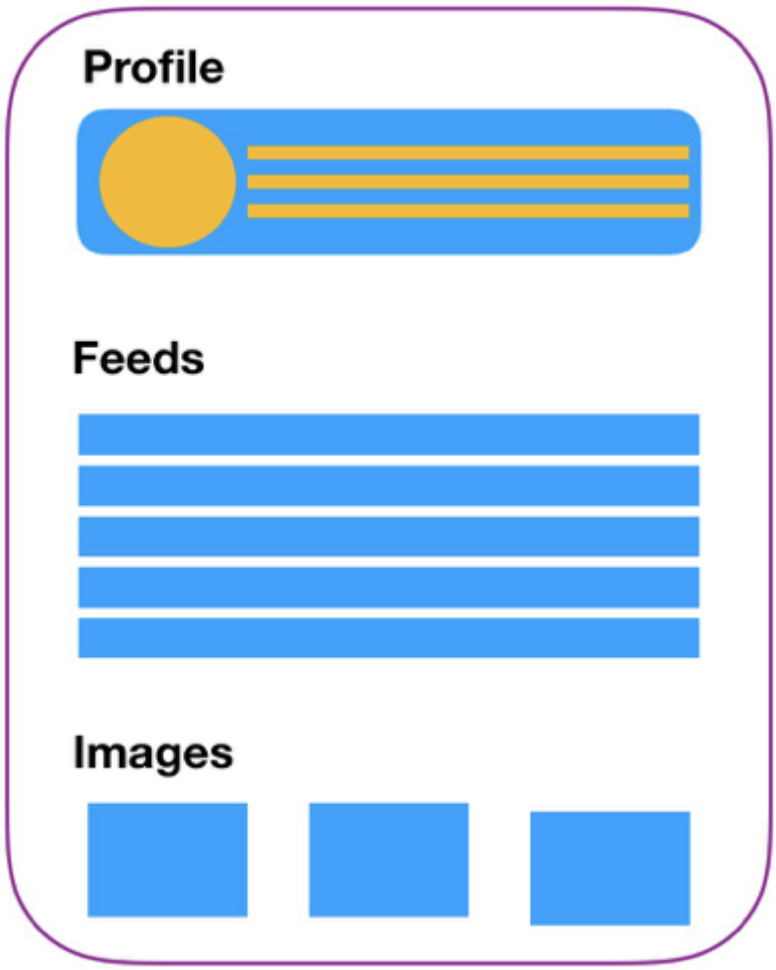
支付页面

这就是单页面应用需要对数据状态进行管理的信号：我们维护一个 cartList 数组，供应用消费使用，这个数组放在 Redux 或者 Mobx，或者 Vuex 当中都是可行的。

合理 connect 场景

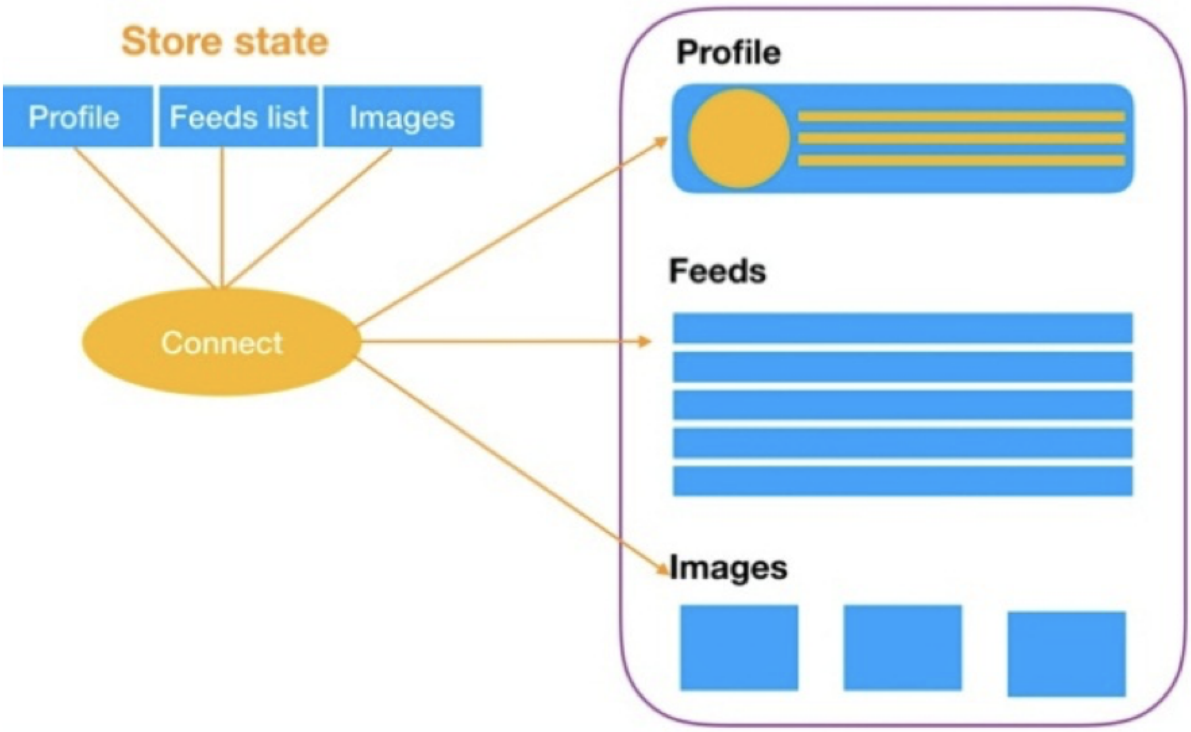
在使用 Redux 时，我们搭配 React-redux 来对组件和数据进行联通（connect），一个常陷入的误区就是滥用 connect，而没有进行更合理的设计分析。也可能只在顶层进行了 connect 设计，然后再一层层进行数据传递。

比如在一个页面中存在 Profile、Feeds（信息流）、Images（图片）区域，如图所示。



这些区域构成了页面的主体，它们分别对应于 Profile、Feeds、Images 组件，共同作为 Page 组件的子组件而存在。

如果只对 Page 这个顶层组件进行 connect 设计，其他组件的数据依靠 Page 组件进行分发，则设计如图所示：



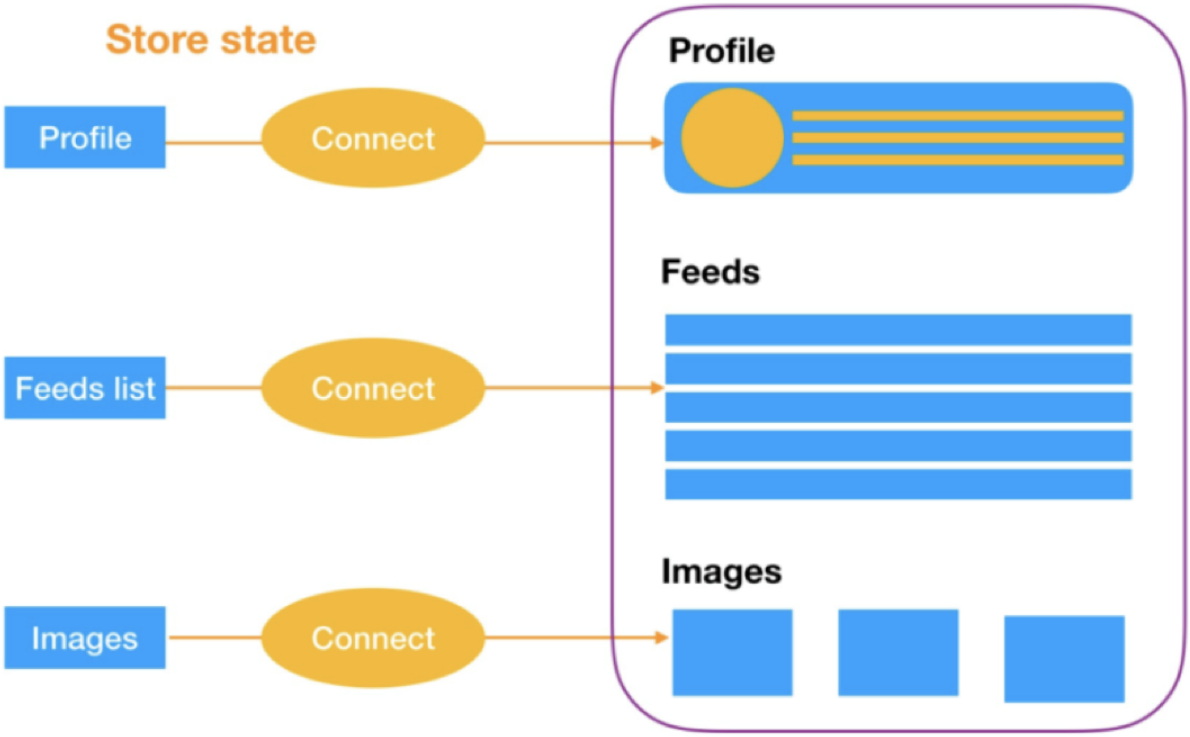
这样做存在的问题如下：

当改动 Profile 组件中的用户头像时，由于数据变动整个 Page 组件都会重新渲染；

当删除 Feeds 组件中的一条信息时，整个 Page 组件也都会重新渲染；

当在 Images 组件中添加一张图片时，整个 Page 组件同样都会重新渲染。

因此，更好的做法是对 Profile、Feeds、Images 这三个组件分别进行 connect 设计，在 connect 方法中使用 mapStateToProps 筛选出不同组件关心的 state 部分，如图所示：



这样做的好处很明显：

当改动 Profile 组件中的用户头像时，只有 Profile 组件重新渲染；

当删除 Feeds 组件中的一条信息时，只有 Feed 组件重新渲染；

当在 Images 组件中添加一张图片时，只有 Images 组件重新渲染。

扁平化数据状态

扁平化的数据结构是一个很有意义的概念，它不仅能够合理引导开发逻辑，同时也是性能优化的一种体现。请看这样的数据结构：

```
{
 articles: [{
   comments: [{
     authors: [{
     }]
```

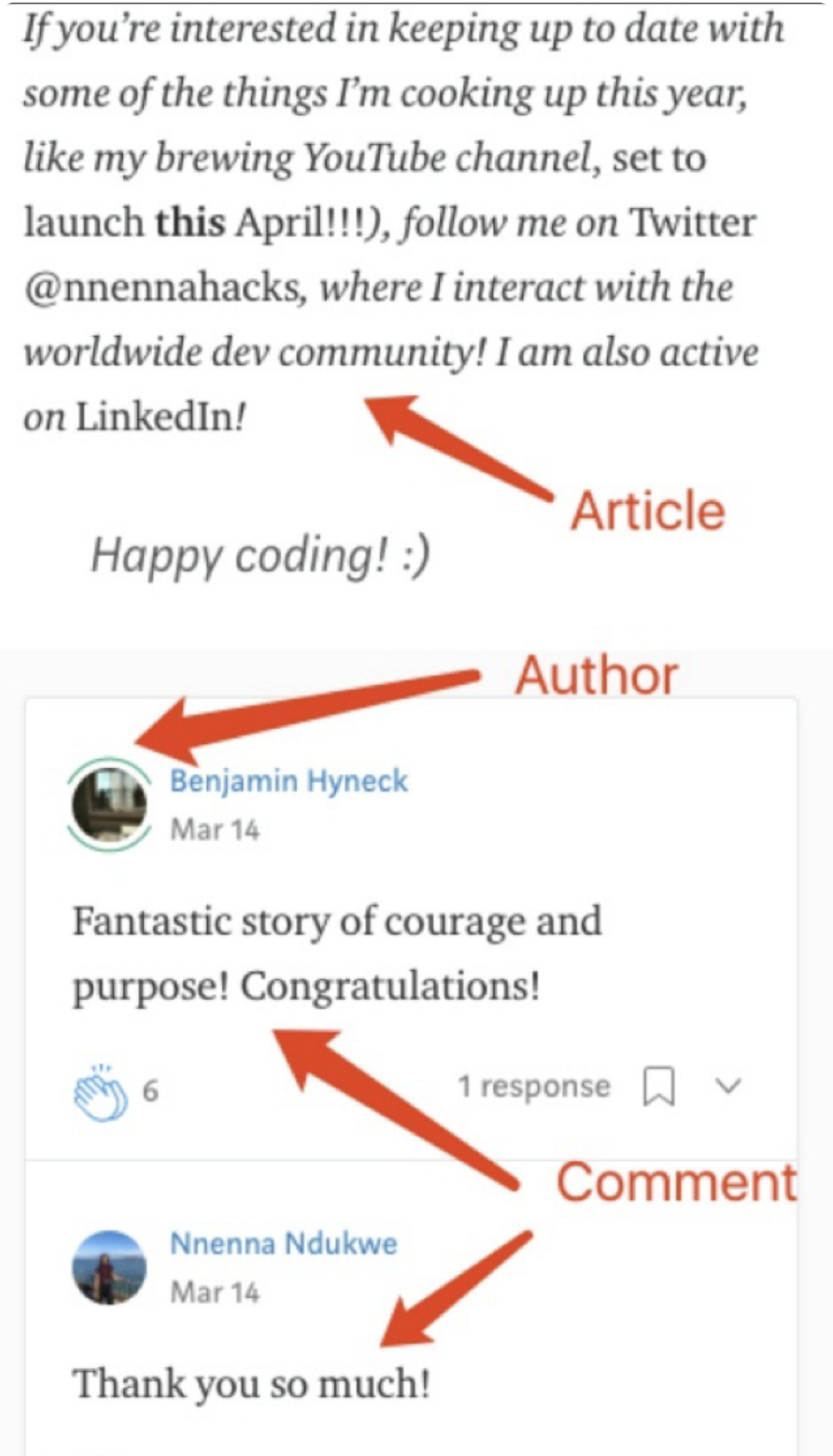
```
    }]

  }],

  ...

}
```

不难想象这是一个文章列表加文章评论互动的场景，其对应于三个组件：Article、Comment 和 Author。这样的页面设计比比皆是，如图所示：


If you're interested in keeping up to date with some of the things I'm cooking up this year, like my brewing YouTube channel, set to launch this April!!!), follow me on Twitter @nnennahacks, where I interact with the worldwide dev community! I am also active on LinkedIn!
Article
Happy coding! :)
Author
Benjamin Hyneck
Mar 14
Fantastic story of courage and purpose! Congratulations!
6
1 response
Comment
Nnenna Ndukwe
Mar 14
Thank you so much!

相关 reducer 的处理很棘手，如果 articles[2].comments[4].authors1 发生了变化，想要返回更新后的状态，并保证不可变性，操作起来不是那么简单的，我们需要对深层对象结构进行拷贝或递归。

因此，更好的数据结构设计一定是扁平化的，我们对 articles、comments、authors 进行扁平化处理。例如 comments 数组不再存储 authors 数据，而是记录 userId，需要时在 users 数组中进行提取即可：

```
{
 articles: [{
   ...
 }],
 comments: [{
   articleId: ..,
   userId: ...,
   ...
 }],
 users: [{
   ...
 }]
}
```

不同组件只需要关心不同的数据片段，比如 Comment 组件只关心 comments 数组；Author 组件只关心 users 数组。这样不仅操作更合理，而且有效减少了渲染压力。

Redux 的罪与罚

前文终点提到了 Redux，其实现原理较为简单，核心代码也不过几行，简要来说：Redux 是我们之前提到的发布订阅模式结合函数式编程的体现。这里不再过多赘述，我们主要来看看以 Redux 为首的数据状态管理类库的「缺陷」和发展点。

其实，Dan Abramov 很早就提到过 「You might not need Redux」，文中提到了 Redux 的限制。他也说过 「Try Mobx」 这种「打脸」行为。归纳一下，Redux 的限制主要体现在：

> Redux 带来了函数式编程、不可变性思想等，为了配合这些理念，开发者必须要写很多「模式代码（boilerplate）」，繁琐以及重复是开发者不愿意容忍的。当然也有很多 hack 旨在减少 boilerplate，但目前阶段，可以说 Redux 天生就附着繁琐。
>
> 使用 Redux，那么你的应用就要用 objects 或者 arrays 描述状态。
>
> 使用 Redux，那么你的应用就要使用 plain objects 即 actions 来描述变化。
>
> 使用 Redux，那么你的应用就要使用纯函数去处理变化。
>
> 应用中，状态很多都要抽象到 store，不能痛痛快快地写业务，一个变化就要对应编写 action（action creator）、reducer 等。

这些「缺点」和响应式结合函数式的 Mobx 相比，编程体验被「打了折扣」。

Redux 上层解决方案

为了弥补这些缺点，社区开启了一轮又一轮的尝试，其中一个努力方向是基于 Redux 封装一整套上层解决方案，这个方向以 Redux-sage、dva、rematch 类库或框架为主。

我总结一下这些解决方案的特点和思路：

简化初始化过程

传统的 Redux 初始化充满了 hack，过于函数式，且较为繁琐：

```
import { createStore, applyMiddleware, compose } from
'redux'

import thunk from 'redux-thunk'

import rootReducer from './reducers'



const initialState = {

   // ...

}


const store = initialState => createStore(

   rootReducer,

   initialState,

   compose(

       applyMiddleware(thunk),

       // ...

   )

)
```

这其中我们只应用了一个中间件，还没有涉及到 devtool 的配置。而不论是 Dva 还是其他方案，都采用面向对象式的配置化初始。

简化 reducers

传统的 reducers 可能需要写恼人的 switch...case 或很多样板代码，而更上层的解决方案进行封装后，类似：

```
const reducer = {

   ACTIONTYPE1: (state, action) => newState,

   ACTIONTYPE2: (state, action) => newState,

}
```

更加清爽。

带请求的副作用

处理网络请求，Redux 一般需要 thunk 中间件，它的原理是：首先 dispatch 一个 action，但是这个 action 不是 plain object 类型，而是一个函数；thunk 中间件发现 action type 为函数类型时，把 dispatch 和 getState 等方法作为参数，传递给函数进行副作用逻辑。

如果读者不是 React、Redux 开发者，也许很难看懂上一段描述，这也是 Redux 处理异步副作用的晦涩体现。更上层的解决方案 Redux-saga 采用 generator 的思想，或 async/await 处理副作用，无疑更加友好合理。

为了更好地配合生成器方案，上层方案将 action 分为普通 action 和副作用 action，开发者使用起来也更加清晰。

reducer 和 action 合并

为了进一步减少模版代码，一个通用的做法是在 Redux 之上，将 reducer 和 action 声明合并，类似：

```
const store = {
   state: {
       count: 0,
       state1: {}
   },
   reduers: {
       action1: (state, action) => newState,
       action2: (state, action) => newState,
   }
}
```

这样的声明一步到位，我们定义了两个 action：

- action1
- action2

它们出自于 store.reducers 的键名，而对应键值即为 reducer 逻辑。

这些都是基于 Redux 封装上层解决方案的基本思想，了解了这些，Dva、Redux-saga 原理已经对读者不再陌生！

当然，理清了数据状态管理的意义，简化了数据管理的操作，我们还要分析到底应该如何组织数据。

我们到底需要怎样的数据状态管理

关于 Redux，这里不再过多讨论。我们试图脱离开 Redux 本身，思考到底需要什么样的数据状态管理方案。整理我们的核心诉求就是：方便地修改数据，方便地获取数据。

新的发展趋势：Mobx

从核心诉求出发，我们有两种做法：修改数据，Redux 提倡函数式、提倡不可变性、提倡数据扁平化，获取数据说到底是依赖发布订阅模式。相对地，Mobx 是面向对象和响应式的结合，它的数据源是可变的，对数据的观察是响应式的：

```
const foo = observable({

   a: 1,

   b: 2

})


autoRun(() => {

   console.log(foo.a)

})


foo.b = 3 // 没有任何输出

foo.a = 2 // 输出：2
```

这像不像我们前面课程提到的数据拦截/代理？没错，它们的原理都是完全一致的。尝试对上面的代码改为：

```
const state = observable({

   state1: {}
```

```
})

autoRun(() => {
   return ()
})

state.state1 = {}
```

当我们改动 state.state1 时，autoRun 的回调将会触发，引起了组件的重新渲染。不同于 Redux，这就是另一种流派 Mobx 的核心理念。

不管是 Redux 还是 Mobx，它们都做到了：组件可以读取 state，修改 state；有新 state 时更新。这个 state 是单一数据源，只不过修改 state 方式不同。更近一步地说，Mobx 通过包装对象和数组为可观察对象，隐藏了大部分的样板代码，比 Redux 更加简洁，也更加「魔幻」，更像是「双向绑定」。

对此我的建议是：在数据状态不太复杂的情况下，Mobx 也许更加简洁高效；如果数据状态非常复杂，或者你是函数式编程的粉丝，可以考虑 Redux，但是在 Redux 层上进行封装，使用类似 Dva 方案，是一个明智的选择。

如何做到 Redux free（context 和 hooks）

做到 Redux free，有两种选择：一个是拥抱 Mobx 或者 GraphQL，但还是没有脱离框架或者类库；另一个选择就是选择原生 React 方案，其中之一就是 context API，React 16.3 介绍了稳定版的 context 特性，它从某种程度上可以更方便地实现组件间通信，尤其是对于跨越多层父子组件的情况，更加高效。我们知道 Redux-react 就是基于 context 实现的，那么在一些简单的情况下，完全可以使用稳定的 context，而抛弃 Redux。

在 ReactConf 2018 会议中，React 团队发布了 React hooks。简单来看，hooks 给予了函数式组件像类组件工作的能力，函数式组件可以使用 state，并且在一些副作用后进行 update。useReducer hooks 搭配 context API 以及 useContext hook，完全可以模仿一个简单的 Redux。useReducer hooks 使我

们可以像 reducer 的方式一样更新 state，useContext 可以隔层级传递数据，原生 React 似乎有了内置 Redux 的能力。当然这种能力是不全面的，比如对网络请求副作用的管理、时间旅行和调试等。

这不是一篇讲解 React 的课程，具体代码细节我们不再展开，感兴趣的读者可以参考：

react hooks VS redux

from-redux-to-hooks-case-study

总结

其实数据状态管理没有永恒的「最佳实践」。随着应用业务的发展，数据的复杂程度是不断扩张的，数据和组件是绑定在一起的概念，我们如何梳理好数据，如何对于特定的行为修改特定的数据，给予特定组件特定的数据，是一个非常有趣的话题，也是进阶路上的「必修课」。

点击查看下一节

React 的现状与未来