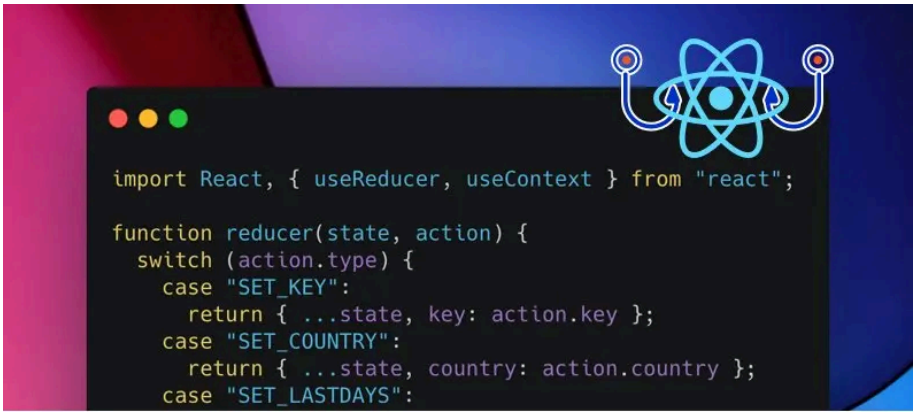
用动画和实战打开 React Hooks（三）：useReducer 和 useContext

一只图雀 2020-05-24 阅读 15 分钟



> 本文由图雀社区成员 **mRc** 写作而成，欢迎加入图雀社区，一起创作精彩的免费技术教程，予力编程行业发展。
> 如果您觉得我们写得还不错，记得 **点赞 + 关注 + 评论** 三连，鼓励我们写出更好的教程?

随着应用状态越来越复杂，我们迫切需要状态与数据流管理的解决方案。熟悉 React 开发的同学一定听说过 Redux，而在这篇文章中，我们将通过 useReducer + useContext 的组合实现一个简易版的 Redux。首先，我们将带你重新认识"老朋友"useState，并借此引出这篇文章的主角：Reducer 函数与 useReducer 钩子，并通过实战一步步带你理清数据流和状态管理的基本思想。

useState：柳暗花明

欢迎继续阅读《用动画和实战打开 React Hooks 系列》：

- 《用动画和实战打开 React Hooks（一）：useState 和 useEffect》
- 《用动画和实战打开 React Hooks（二）：自定义 Hook 和 useCallback》

如果你想要直接从这一篇开始学习，那么请克隆我们为你提供的源代码：

```
git clone -b third-part https://github.com/tuture-dev/covid-19-wi

# 如果你访问 GitHub 不流畅，我们还提供了 Gitee 地址
git clone -b third-part https://gitee.com/tuture/covid-19-with-ho
```

在这第三篇文章中，我们将首先来重温一下 `useState` 。在之前的两篇教程中，我们可以说和 `useState` 并肩作战了很久，是我们非常"熟悉"的老朋友了。但是回过头来，我们真的足够了解它吗？

一个未解决的问题

你很有可能在使用 `useState` 的时候遇到过一个问题：通过 Setter 修改状态的时候，怎么读取上一个状态值，并在此基础上修改呢？如果你看文档足够细致，应该会注意到 `useState` 有一个**函数式更新**（Functional Update）的用法，以下面这段计数器（代码来自 React 官网）为例：

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Count: {count}
      <button onClick={() => setCount(initialCount)}>Reset</butto
      <button onClick={() => setCount(prevCount => prevCount - 1)
      <button onClick={() => setCount(prevCount => prevCount + 1)
    </>
  );
}
```

可以看到，我们传入 `setCount` 的是一个函数，**它的参数是之前的状态，返回的是新的状态。**熟悉 Redux 的朋友马上就指出来了：这其实就是一个 Reducer 函数。

Reducer 函数的前生今世

Redux 文档里面已经详细地阐述了 Reducer 函数，但是我们这里将先回归最基础的概念，暂时忘掉框架相关的知识。在学习 JavaScript 基础时，你

应该接触过数组的 `reduce` 方法，它可以用一种相当炫酷的方式实现数组求和：

```
const nums = [1, 2, 3]
const value = nums.reduce((acc, next) => acc + next, 0)
```

其中 `reduce` 的第一个参数 `(acc, next) => acc + next` 就是一个 Reducer 函数。从表面上来看，这个函数接受一个状态的累积值 `acc` 和新的值 `next`，然后返回更新过后的累积值 `acc + next`。从更深层次来说，Reducer 函数有**两个必要规则**：

- 只返回一个值
- 不修改输入值，而是返回新的值

第一点很好判断，其中第二点则是很多新手踩过的坑，对比以下两个函数：

```
// 不是 Reducer 函数!
function buy(cart, thing) {
  cart.push(thing);
  return cart;
}

// 正宗的 Reducer 函数
function buy(cart, thing) {
  return cart.concat(thing);
}
```

上面的函数调用了数组的 `push` 方法，会**就地修改**输入的 `cart` 参数（是否 `return` 都无所谓了），违反了 Reducer 第二条规则，而下面的函数通过数组的 `concat` 方法返回了一个**全新的数组**，避免了直接修改 `cart` 。

我们回过头来看之前 `useState` 的函数式更新写法：

```
setCount(prevCount => prevCount + 1);
```

是不是一个很标准的 Reducer?

最熟悉的陌生人

我们在前两篇教程中大量地使用了 `useState`，你可能就此认为 `useState` 应该是最底层的**元素**了。但实际上在 React 的源码中，`useState` 的实现使

用了 `useReducer`（本文的主角，下面会讲到）。在 React 源码中有这么一个关键的函数 `basicStateReducer`（去掉了源码中的 Flow 类型定义）：

```
function basicStateReducer(state, action) {
  return typeof action === 'function' ? action(state) : action;
}
```

于是，当我们通过 `setCount(prevCount => prevCount + 1)` 改变状态时，传入的 `action` 就是一个 Reducer 函数，然后调用该函数并传入当前的 `state`，得到更新后的状态。而我们之前通过传入具体的值修改状态时（例如 `setCount(5)`），由于不是函数，所以直接取传入的值作为更新后的状态。

> **提示**
> 这里选取的是 React v16.13.1 的源码，但是整体的实现应该已经趋于稳定，原理上不会相差太多。

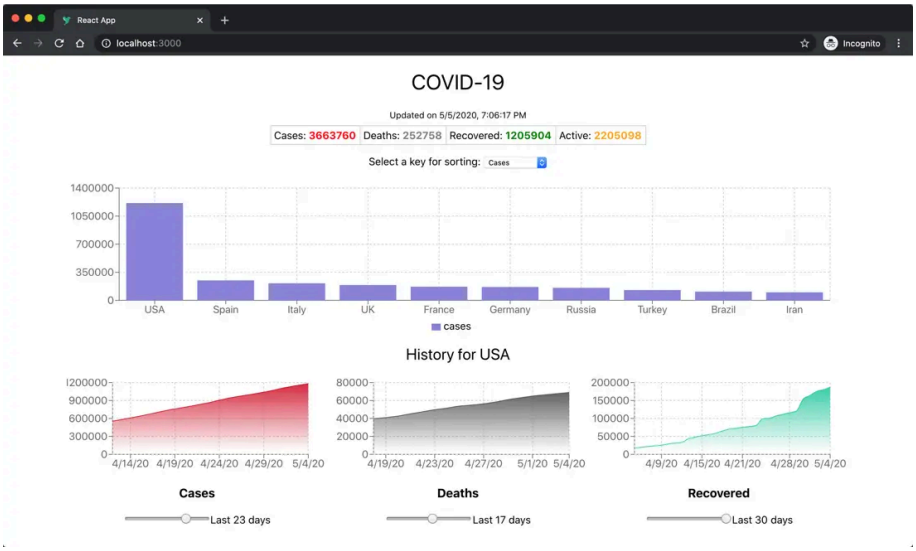
听上去还是有点迷迷糊糊？又到了我们的动画环节。首先，我们传入的 `action` 是一个具体的值：

当传入 Setter 的是一个 Reducer 函数的时候：

是不是一下子就豁然开朗了？

实战环节

这一步要写的代码比较多（可自行复制粘贴哈），我们要实现如下图所示的历史趋势图展示效果：



注意到我们展示了三个历史趋势（确诊病例 Cases、死亡病例 Deaths 和治愈病例 Recovered），并且每张历史趋势图可以调节过去的天数（从 0 到 30 天）。

实现历史趋势图

首先，让我们来实现历史曲线图 `HistoryChart` 组件。创建 `src/components/HistoryChart.js` 组件，代码如下：

```
// src/components/HistoryChart.js
import React from "react";
```

```
import {
  AreaChart,
  CartesianGrid,
  XAxis,
  YAxis,
  Tooltip,
  Area,
} from "recharts";

const TITLE2COLOR = {
  Cases: "#D0021B",
  Deaths: "#4A4A4A",
  Recovered: "#09C79C",
};

function HistoryChart({ title, data, lastDays, onLastDaysChange
  const colorKey = `color${title}`;
  const color = TITLE2COLOR[title];

  return (
    <div>
      <AreaChart
        width={400}
```

这里我们使用了 Recharts 的 AreaChart 组件来绘制历史趋势图，然后在表中分添加了一个范围拖动条，能够让用户选择查看过去 1 到 30 天的历史趋势。

`HistoryChart` 组件包含以下 Props：

- `title` 是图表标题
- `data` 就是绘制图表需要的历史数据
- `lastDays` 是显示过去 N 天的数据，可以通过 `data.slice(-lastDays)` 进行选择
- `onLastDaysChange` 是用户通过 `input` 修改处理过去 N 天时的事件处理函数

接着，我们需要一个辅助函数来对历史数据进行一些转换处理。
NovelCOVID 19 API 返回的历史数据是一个对象：

```
{
  "3/28/20": 81999,
  "3/29/20": 82122
}
```

为了能够适应 Recharts 的数据格式，我们希望转换成数组格式：

```
[
  {
    time: "3/28/20",
    number: 81999
  },
  {
    time: "3/29/20",
    number: 82122
  }
]
```

这个可以通过 `Object.entries` 很方便地进行转换。我们创建 `src/utils.js` 文件，实现 `transformHistory` 函数，代码如下：

```
// src/utils.js
export function transformHistory(timeline = {}) {
  return Object.entries(timeline).map((entry) => {
    const [time, number] = entry;
    return { time, number };
  });
}
```

接着我们来实现历史趋势图组 `HistoryChartGroup`，包含三个图表：确诊病例（Cases）、死亡人数（Deaths）和治愈病例（Recovered）。创建 `src/components/HistoryChartGroup.js`，代码如下：

```
// src/components/HistoryChartGroup.js
import React, { useState } from "react";

import HistoryChart from "./HistoryChart";
import { transformHistory } from "../utils";

function HistoryChartGroup({ history = {} }) {
  const [lastDays, setLastDays] = useState({
    cases: 30,
    deaths: 30,
    recovered: 30,
  });

  function handleLastDaysChange(e, key) {
    setLastDays((prev) => ({ ...prev, [key]: e.target.value }))
  }

  return (
    <div className='history-group'>
      <HistoryChart
        title='Cases'
        data={transformHistory(history.cases)}
```

```
        lastDays={lastDays.cases}
        onLastDaysChange={(e) => handleLastDaysChange(e, "cases
      />
```

调整 CountriesChart 组件

我们需要稍微调整一下 `CountriesChart` 组件，使得用户在点击一个国家的数据后，能够展示对应的历史趋势图。打开 `src/components/CountriesChart.js`，添加一个 `onClick` Prop，并传入 `BarChart` 中，如下面的代码所示：

```
// src/components/CountriesChart.js
// ...

function CountriesChart({ data, dataKey, onClick }) {
  return (
    <BarChart
      width={1200}
      height={250}
      style={{ margin: "auto" }}
      margin={{ top: 30, left: 20, right: 30 }}
      data={data}
      onClick={onClick}
    >
      // ...
    </BarChart>
  );
}

// ...
```

在根组件中集成

最后，我们调整根组件，把之前实现的历史趋势图和修改后的 `CountriesChart` 集成到应用中。打开 `src/App.js`，代码如下：

```
// src/App.js
// ...
import HistoryChartGroup from "./components/HistoryChartGroup";

function App() {
  // ...

  const [country, setCountry] = useState(null);
  const history = useCoronaAPI(`/historical/${country}`, {
    initialData: {},
    converter: (data) => data.timeline,
```

```
});

return (
  <div className='App'>
    <h1>COVID-19</h1>
    <GlobalStats stats={globalStats} />
    <SelectDataKey onChange={(e) => setKey(e.target.value)} /
    <CountriesChart
      data={countries}
      dataKey={key}
      onClick={(payload) => setCountry(payload.activeLabel)}
    />
```

> **成功**
> 写完之后开启项目，点击直方图中的任意一个国家，就会展示该国家历史趋势图（累计确诊、死亡病例、治愈病例），我们还可以随意调过去的天数。

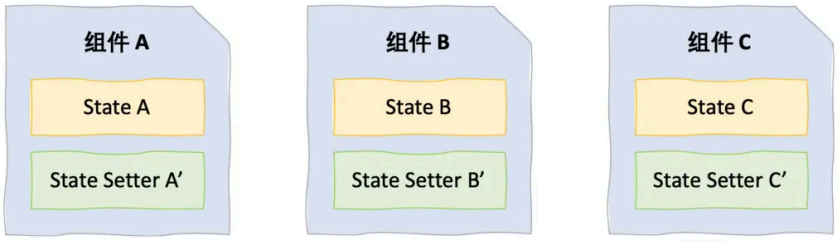
虽然现在我们的应用已经初步成型，但回过头来看代码，发现组件的状态和修改状态的逻辑散落在各个组件中，后面维护和实现新功能时无疑会遇到很大的困难，这时候就需要做专门的状态管理了。熟悉 React 开发的同学一定知道 Redux 或者 MobX 这样的库，不过借助 React Hooks，我们

useReducer + useContext：呼风唤雨

在之前我们说过，这篇文章将通过 React Hooks 来实现一个轻量级的、类似 Redux 的状态管理模型。不过在此之前，我们先简单地过一遍 Redux 的基本思想（熟悉的同学可以直接跳过哈）。

Redux 基本思想

之前，应用的状态（例如我们应用中当前国家、历史数据等等）散落在各个组件中，大概就像这样：



可以看到，每个组件都有自己的 State（状态）和 State Setter（状态修改函数），这意味着跨组件的状态读取和修改是相当麻烦的。而 Redux 的核心思想之一就是将状态放到**唯一的全局对象**（一般称为 Store）中，而修改状态则是调用对应的 Reducer 函数去更新 Store 中的状态，大概就像这样：

上面这个动画描述的是组件 A 改变 B 和 C 中状态的过程：

- 三个组件挂载时，从 Store 中**获取并订阅**相应的状态数据并展示（注意是**只读**的，不能直接修改）
- 用户点击组件 A，触发事件监听函数
- 监听函数中派发（Dispatch）对应的动作（Action），传入 Reducer 函数
- Reducer 函数返回更新后的状态，并以此更新 Store
- 由于组件 B 和 C 订阅了 Store 的状态，所以重新获取更新后的状态并调整 UI

> **提示**
> 这篇教程不会详细地讲解 Redux，想要深入学习的同学可以阅读我们的《Redux 包教包会》系列教程。

useReducer 使用浅析

首先，我们还是来看下官方介绍的 `useReducer` 使用方法：

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

首先我们来看下 `useReducer` 需要提供哪些参数：

1. 第一个参数 `reducer` 显然是必须的，它的形式跟 Redux 中的 Reducer 函数完全一致，即 `(state, action) => newState`。
2. 第二个参数 `initialArg` 就是状态的初始值。
3. 第三个参数 `init` 是一个可选的用于**懒初始化**（Lazy Initialization）的函数，这个函数返回初始化后的状态。

返回的 `state`（只读状态）和 `dispatch`（派发函数）则比较容易理解了。我们来结合一个简单的计数器例子讲解一下：

```
// Reducer 函数
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({ type: 'increment' })}>+</
    </>
  );
}
```

我们首先关注一下 Reducer 函数，它的两个参数 `state` 和 `action` 分别是当前状态和 `dispatch` 派发的动作。这里的动作就是普通的 JavaScript 对象，用来表示修改状态的操作，注意 `type` 是必须要有的属性，代表**动作的类型**。然后我们根据 `action` 的类型返回相应修改后的新状态。

然后在 `Counter` 组件中，我们通过 `useReducer` 钩子获取到了状态和 `dispatch` 函数，然后把这个状态渲染出来。在按钮 `button` 的 `onClick` 回调函数中，我们通过 `dispatch` 一个类型为 `increment` 的 Action 去更新状态。

天哪，为什么一个简单的计数器都搞得这么复杂！简简单单一个 `useState` 不就搞定了吗？

什么时候该用 useReducer

你也许发现，`useReducer` 和 `useState` 的使用目的几乎是一样的：**定义状态和修改状态的逻辑。**`useReducer` 使用起来较为繁琐，但如果你的状态管理出现了至少一个以下所列举的问题：

- 需要维护的状态本身比较复杂，多个状态之间相互依赖
- 修改状态的过程比较复杂

那么我们就强烈建议你使用 `useReducer` 了。我们来通过一个实际的案例讲解来感受一下 `useReducer` 的威力（这次不是无聊的计数器啦）。假设我们要做一个支持*撤销*和*重做*的编辑器，它的 `init` 函数和 Reducer 函数分别如下：

```
// 用于懒初始化的函数
function init(initialState) {
  return {
    past: [],
    present: initialState,
    future: [],
  };
}

// Reducer 函数
function reducer(state, action) {
  const { past, future, present } = state;
  switch (action.type) {
    case 'UNDO':
      return {
        past: past.slice(0, past.length - 1),
        present: past[past.length - 1],
        future: [present, ...future],
      };
    case 'REDO':
      return {
        past: [...past, present],
        present: future[0],
        future: future.slice(1),
      };
    default:
```

试试看用 `useState` 去写，会不会很复杂？

useContext 使用浅析

现在状态的获取和修改都已经通过 `useReducer` 搞定了，那么只差一个问题：怎么让所有组件都能获取到 `dispatch` 函数呢？

在 Hooks 诞生之前，React 已经有了在组件树中共享数据的解决方案：Context。在类组件中，我们可以通过 `Class.contextType` 属性获取到最近的

Context Provider，那么在函数式组件中，我们该怎么获取呢？答案就是 `useContext` 钩子。使用起来非常简单：

```
// 在某个文件中定义 MyContext
const MyContext = React.createContext('hello');

// 在函数式组件中获取 Context
function Component() {
  const value = useContext(MyContext);
  // ...
}
```

通过 `useContext`，我们就可以轻松地让所有组件都能获取到 `dispatch` 函数了！

实战环节

设计中心状态

好的，让我们开始用 useReducer + useContext 的组合来重构应用的状态管理。按照状态中心化的原则，我们把整个应用的状态提取到一个全局对象中。初步设计（TypeScript 类型定义）如下：

```
type AppState {
  // 数据指标类别
  key: "cases" | "deaths" | "recovered";

  // 当前国家
  country: string | null;

  // 过去天数
  lastDays: {
    cases: number;
    deaths: number;
    recovered: number;
  }
}
```

在根组件中定义 Reducer 和 Dispatch Context

这一次我们按照**自顶向下**的顺序，先在根组件 `App` 中配置好所有需要的 Reducer 以及 Dispatch 上下文。打开 `src/App.js`，修改代码如下：

```
// src/App.js
import React, { useReducer } from "react";
```

```
// ...

const initialState = {
  key: "cases",
  country: null,
  lastDays: {
    cases: 30,
    deaths: 30,
    recovered: 30,
  },
};

function reducer(state, action) {
  switch (action.type) {
    case "SET_KEY":
      return { ...state, key: action.key };
    case "SET_COUNTRY":
      return { ...state, country: action.country };
    case "SET_LASTDAYS":
      return {
        ...state,
        lastDays: { ...state.lastDays, [action.key]: action.day
```

我们来一一分析上面的代码变化：

1. 首先定义了整个应用的初始状态 `initialState`，这个是后面 `useReducer` 钩子所需要的
2. 然后我们定义了 Reducer 函数，主要响应三个 Action：`SET_KEY` 、`SET_COUNTRY` 和 `SET_LASTDAYS` ，分别用于修改数据指标、国家和过去天数这三个状态
3. 定义了 `AppDispatch` 这个 Context，用来向子组件传递 `dispatch`
4. 调用 `useReducer` 钩子，获取到状态 `state` 和分发函数 `dispatch`
5. 最后在渲染时用 `AppDispatch.Provider` 将整个应用包裹起来，传入 `dispatch` ，使子组件都能获取得到

在子组件中通过 Dispatch 修改状态

现在子组件的所有状态都已经提取到了根组件中，而子组件唯一要做的就是在响应用户事件时通过 `dispatch` 去修改中心状态。思路非常简单：

- 先通过 `useContext` 获取到 `App` 组件传下来的 `dispatch`
- 调用 `dispatch` ，发起相应的动作（Action）

OK，让我们开始动手吧。打开 `src/components/CountriesChart.js` ，修改代码如下：

```
// src/components/CountriesChart.js
import React, { useContext } from "react";
// ...
import { AppDispatch } from "../App";

function CountriesChart({ data, dataKey }) {
  const dispatch = useContext(AppDispatch);

  function onClick(payload = {}) {
    if (payload.activeLabel) {
      dispatch({ type: "SET_COUNTRY", country: payload.activeLabe
    }
  }

  return (
    // ...
  );
}

export default CountriesChart;
```

按照同样的思路，我们来修改 `src/components/HistoryChartGroup.js` 组件：

```
// src/components/HistoryChartGroup.js
import React, { useContext } from "react";

import HistoryChart from "./HistoryChart";
import { transformHistory } from "../utils";
import { AppDispatch } from "../App";

function HistoryChartGroup({ history = {}, lastDays = {} }) {
  const dispatch = useContext(AppDispatch);

  function handleLastDaysChange(e, key) {
    dispatch({ type: "SET_LASTDAYS", key, days: e.target.value })
  }

  return (
    // ...
  );
}

export default HistoryChartGroup;
```

最后一公里，修改 `src/components/SelectDataKey.js`：

```
// src/components/SelectDataKey.js
import React, { useContext } from "react";
import { AppDispatch } from "../App";

function SelectDataKey() {
  const dispatch = useContext(AppDispatch);

  function onChange(e) {
    dispatch({ type: "SET_KEY", key: e.target.value });
  }

  return (
    // ...
  );
}

export default SelectDataKey;
```

重构完成，把项目跑起来，应该会发现和上一步的功能分毫不差。

> **提示**
> 如果你熟悉 Redux，会发现我们的重构存在一个小小的遗憾：子组件只能通过传递 Props 的方式获取根组件 `App` 中的 `state` 。一个变通之计是通过把 `state` 也装进 Context 来解决，但如果遇到这种需求，笔者还是建议直接使用 Redux。

Redux 还有用吗：Control 与 Context 之争

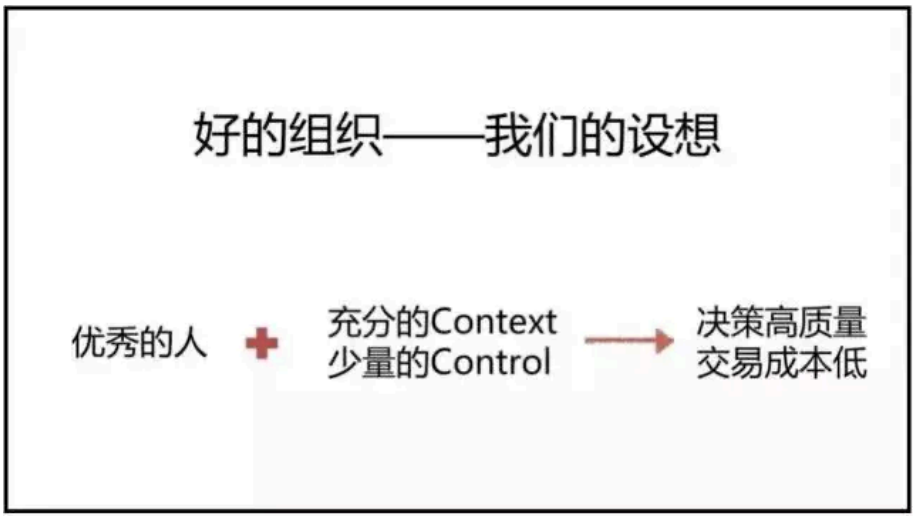
听到有些声音说有了 React Hooks，都不需要 Redux 了。那 Redux 到底还有用吗?

在回答这个问题之前，请允许我先胡思乱想一波。React Hooks 确实强大得可怕，特别是通过优秀的第三方自定义 Hooks 库，几乎能让每个组件都能游刃有余地处理复杂的业务逻辑。反观 Redux，它的核心思想就是将状态和修改状态的操作全部集中起来进行。

有没有发现，这其实刚好对应了两种管理学思想 Context 和 Control?

> 管理者需要 Context，not Control。—— 字节跳动创始人和 CEO 张一鸣

Control 就是将权力集中起来，员工们只需有条不紊地按照 CEO 的决策执行相应的任务，就像 Redux 中的全局 Store 是"唯一的真相来源"（Single Source of Truth），所有状态和数据流的更新必须经过 Store；而 Context 就是给予各部门、各层级足够的决策权，因为他们所拥有的**上下文**更充足，**专业度**也更好，就像 React 中响应特定逻辑的组件具有更充足的上下文，并且可以借助 Hooks "自给自足"地执行任务，而无需依赖全局的 Store。



聊到这里，我想你心里已经有自己的答案了。*如果你想要分享的话，记得在评论区留言哦~*

参考资料

- Sarah Drasner：Understanding the Almighty Reducer
- Kingsley Silas：Getting to Know the useReducer React Hook
- Kpax Qin：Redux状态管理之痛点、分析与改良
- 方应杭：尽量使用 useReducer，不要使用 useState（译文）
- 张一鸣：CEO 要避免"理性的自负", 这错误盖茨、乔布斯都犯过

> 想要学习更多精彩的实战技术教程？来图雀社区逛逛吧。

react.js hooks

赞 收藏 分享

阅读 2.1k • 发布于 2020-05-24

一只图雀
863 声望 • 1.2k 粉丝
我们图雀社区是一个供大家分享用 Tuture 写作工具撰写教程的一个平台。在这里，读者们可以尽情享受高质量的实战教...
关注作者

« 上一篇 下一篇 »
18 个 React 最佳实践技巧，助你在...从零到部署：用 Vue 和 Express 实...

引用和评论

推荐阅读

Taro 小程序开发大型实战（九）：使用 Authing 打造具有微信登录的企业级用户系统
一只图雀 • 赞 1 • 阅读 4.4k

React中的高优先级任务插队机制
nero • 赞 32 • 阅读 15.1k • 评论 14

文件导出
热饭班长 • 赞 8 • 阅读 2.9k

TS-react：react中常用的类型整理
wxp686 • 赞 1 • 阅读 6.9k

【从前端入门到全栈】Node.js之大文件分片上传
野生程序猿江辰 • 赞 3 • 阅读 268 • 评论 1

react组件解耦
热饭班长 • 赞 3 • 阅读 3.9k

react 踩坑
assassin_cike • 赞 1 • 阅读 3.4k

0 条评论

得票 | 最新

撰写评论 ...

提交评论

评论支持部分 Markdown 语法：`**粗体** _斜体_ [链接](http://example.com) \`代码\` - 列表 > 引用`。你还可以使用 @ 来通知其他用户。