用动画和实战打开 React Hooks（一）：useState 和 useEffect

一只图雀 2020-05-05 阅读 12 分钟

> 我们研发开源了一款基于 Git 进行技术实战教程写作的工具，我们图雀社区的所有教程都是用这款工具写作而成，欢迎 Star 哦
> 如果你想快速了解如何使用，欢迎阅读我们的 教程文档 哦

> 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?

自从 React 16.8 发布之后，它带来的 React Hooks 在前端圈引起了一场无法逆转的风暴。React Hooks 为函数式组件提供了无限的功能，解决了类组件很多的固有缺陷。这篇教程将带你快速熟悉并掌握最常用的两个 Hook：`useState` 和 `useEffect`。在了解如何使用的同时，还能管窥背后的原理，顺便实现一个 COVID-19（新冠肺炎）可视化应用。

> 欢迎访问本项目的 GitHub 仓库和 Gitee 仓库。

起步

前提条件

在阅读这篇教程之前，希望你已经做了如下准备：

- 掌握了 React 基础知识，例如组件、JSX、状态等等，如果你不了解的话，请先学习《一杯茶的时间，上手 React 框架》
- 配置好 Node 环境，可参考《一杯茶的时间，上手 Node.js》

为什么会有 Hooks?

在 Hooks 出现之前，类组件和函数组件的分工一般是这样的：

- **类组件**提供了完整的状态管理和生命周期控制，通常用来承接复杂的业务逻辑，被称为*"聪明组件"*
- **函数组件**则是纯粹的从数据到视图的映射，对状态毫无感知，因此通常被称为*"傻瓜组件"*

有些团队还制定了这样的 React 组件开发约定：

> 有状态的组件没有渲染，有渲染的组件没有状态。

那么 Hooks 的出现又是为了解决什么问题呢？我们可以试图总结一下类组件颇具代表性的**痛点**：

1. 令人头疼的 `this` 管理，容易引入难以追踪的 Bug
2. 生命周期的划分并不符合"内聚性"原则，例如 `setInterval` 和 `clearInterval` 这种具有强关联的逻辑被拆分在不同的生命周期方法中
3. 组件复用（数据共享或功能复用）的困局，从早期的 Mixin，到高阶组件（HOC），再到 Render Props，始终没有一个清晰直观又便于维护的组件复用方案

没错，随着 Hooks 的推出，这些痛点都成为了历史！

为什么要写这一系列 Hooks 教程?

如何快速学习并掌握 React Hooks 一直是困扰很多新手或者老玩家的一个问题，而笔者在日常的学习和开发中也发现了以下头疼之处：

- React 官方文档对 Hooks 的讲解偏应用，对原理的阐述一笔带过
- 讲 React Hooks 的优秀文章很多，但大多专注于讲解一两个 Hook，要想一网打尽有难度

- 看了很多使用方法甚至源码分析，但是没法和具体的使用场景对应起来，不了解怎么在实际开发中灵活运用

如果你也有同样的困惑，希望这一系列文章能帮助你拨开云雾，让 Hooks 成为你的称手兵器。我们将通过一个完整的 COVID-19 数据可视化项目，结合 Hooks 的动画原理讲解，让你真正地精通 React Hooks！

说实话，Hooks 的知识点相当分散，就像游乐园的游玩项目一样，选择一条完美的路线很难。但是不管怎么样，希望在接下来的旅程中，你能玩得开心?!

初始化项目

首先，通过 Create React App（以下简称 CRA） 初始化项目：

```
npx create-react-app covid-19-with-hooks
```

在少许等待之后，进入项目。

> **提示**
> 我们所有的数据源自 NovelCOVID 19 API，可以点击访问其全部的 API 文档。

一切就绪，让我们出发吧！

useState + useEffect：初来乍到

首先，让我们从最最最常用的两个 Hooks 说起：`useState` 和 `useEffect`。很有可能，你在平时的学习和开发中已经接触并使用过了（当然如果你刚开始学也没关系啦）。不过在此之前，我们先熟悉一下 React 函数式组件的运行过程。

理解函数式组件的运行过程

我们知道，Hooks **只能用于 React 函数式组件**。因此理解函数式组件的运行过程对掌握 Hooks 中许多重要的特性很关键，请看下图：

可以看到，函数式组件严格遵循 `UI = render(data)` 的模式。当我们第一次调用组件函数时，触发**初次渲染**；然后随着 `props` 的改变，便会重新调用该组件函数，触发**重渲染**。

你也许会纳闷，动画里面为啥要并排画三个一样的组件呢？因为我想通过这种方式直观地阐述函数式组件的一个重要思想：

> 每一次渲染都是完全独立的。

后面我们将沿用这样的风格，并一步步地介绍 Hook 在函数式组件中扮演怎样的角色。

useState 使用浅析

首先我们来简单地了解一下 `useState` 钩子的使用，官方文档介绍的使用方法如下：

```
const [state, setState] = useState(initialValue);
```

其中 `state` 就是一个状态变量，`setState` 是一个用于修改状态的 Setter 函数，而 `initialValue` 则是状态的初始值。

光看代码可能有点抽象，请看下面的动画：

<App />

与之前的纯函数式组件相比，我们引入了 `useState` 这个钩子，瞬间就打破了之前 `UI = render(data)` 的安静画面——函数组件居然可以**从组件之外把状态和修改状态的函数"钩"过来**！并且仔细看上面的动画，**通过调用 Setter 函数，居然还可以直接触发组件的重渲染**！

> **提示**
> 你也许注意到了所有的"钩子"都指向了一个绿色的问号，我们会在下面详细地分析那是什么，现在就暂时把它看作是组件之外可以访问的一个"神秘领域"。

结合上面的动画，我们可以得出一个重要的推论：**每次渲染具有独立的状态值**（毕竟每次渲染都是完全独立的嘛）。也就是说，每个函数中的 `state` 变量只是一个简单的**常量**，每次渲染时从钩子中获取到的常量，并没有附着数据绑定之类的神奇魔法。

这也就是老生常谈的 **Capture Value** 特性。可以看下面这段经典的计数器代码（来自 Dan 的这篇精彩的文章）：

```
function Counter() {
  const [count, setCount] = useState(0);

  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + count);
    }, 3000);
  }

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
      <button onClick={handleAlertClick}>
        Show alert
```

```
        </button>
      </div>
    );
  }
```

实现了上面这个计数器后（也可以直接通过这个 Sandbox 进行体验），按如下步骤操作：1）点击 Click me 按钮，把数字增加到 3；2）点击 Show alert 按钮；3）在 `setTimeout` 触发之前点击 Click me，把数字增加到 5。

结果是 Alert 显示 3！

如果你觉得这个结果很正常，恭喜你已经理解了 Capture Value 的思想！如果你觉得匪夷所思嘛……来简单解释一下：

- 每次渲染相互独立，因此每次渲染时组件中的状态、事件处理函数等等都是独立的，或者说**只属于**所在的那一次渲染
- 我们在 `count` 为 3 的时候触发了 `handleAlertClick` 函数，这个函数**所记住的** `count` 也为 3
- 三秒种后，刚才函数的 `setTimeout` 结束，输出**当时记住的**结果：3

这道理就像，你翻开十年前的日记本，虽然是现在翻开的，但记录的仍然是十年前的时光。或者说，日记本 Capture 了那一段美好的回忆。

useEffect 使用浅析

你可能已经听说 `useEffect` 类似类组件中的生命周期方法。但是在开始学习 `useEffect` 之前，建议你暂时忘记生命周期模型，毕竟函数组件和类组件是不同的世界。官方文档介绍 `useEffect` 的使用方法如下：

```
useEffect(effectFn, deps)
```

`effectFn` 是一个执行某些可能具有**副作用**的 Effect 函数（例如数据获取、设置/销毁定时器等），它可以返回一个**清理函数**（Cleanup），例如大家所熟悉的 `setInterval` 和 `clearInterval`：

```
useEffect(() => {
  const intervalId = setInterval(doSomething(), 1000);
  return () => clearInterval(intervalId);
});
```

可以看到，我们在 Effect 函数体内通过 `setInterval` 启动了一个定时器，随后又返回了一个 Cleanup 函数，用于销毁刚刚创建的定时器。

OK，听上去还是很抽象，再来看看下面的动画吧：

动画中有以下需要注意的点：

- 每个 Effect 必然在渲染之后执行，因此不会阻塞渲染，提高了性能
- 在运行每个 Effect 之前，运行前一次渲染的 Effect Cleanup 函数（如果有的话）
- 当组件销毁时，运行最后一次 Effect 的 Cleanup 函数

> **提示**
> 将 Effect 推迟到渲染完成之后执行是出于性能的考虑，如果你想在渲染之前执行某些逻辑（不惜牺牲渲染性能），那么可使用 `useLayoutEffect` 钩子，使用方法与 `useEffect` 完全一致，只是执行的时机不同。

再来看看 `useEffect` 的第二个参数：`deps` （依赖数组）。从上面的演示动画中可以看出，React 会在**每次渲染后都运行 Effect**。而依赖数组就是用来控制是否应该触发 Effect，从而能够减少不必要的计算，从而优化了性能。具体而言，只要依赖数组中的每一项与上一次渲染相比都没有改变，那么就跳过本次 Effect 的执行。

仔细一想，我们发现 `useEffect` 钩子与之前类组件的生命周期相比，有两个显著的特点：

- 将初次渲染（`componentDidMount`）、重渲染（`componentDidUpdate`）和销毁（`componentDidUnmount`）三个阶段的逻辑用一个统一的 API 去

解决

- 把相关的逻辑都放到一个 Effect 里面（例如 `setInterval` 和 `clearInterval`），更突出逻辑的内聚性

在最极端的情况下，我们可以指定 `deps` 为空数组 `[]`，这样可以确保 Effect **只会在组件初次渲染后执行。**实际效果动画如下：

可以看到，后面的所有重渲染都不会触发 Effect 的执行；在组件销毁时，运行 Effect Cleanup 函数。

> **注意**
> 如果你熟悉 React 的重渲染机制，那么应该可以猜到 `deps` 数组在判断元素是否发生改变时同样也使用了 `Object.is` 进行比较。因此一个隐患便是，当 `deps` 中某一元素为非原始类型时（例如函数、对象等），**每次渲染都会发生改变**，从而失去了 `deps` 本身的意义（条件式地触发 Effect）。我们会在接下来讲解如何规避这个困境。

实战环节

OK，到了实战环节，我们先实现获取全球数据概况（每 5 秒重新获取一次）。创建 `src/components/GlobalStats.js` 组件，用于展示全球数据概况，代码如下：

```
import React from "react";

function Stat({ number, color }) {
  return <span style={{ color: color, fontWeight: "bold" }}>{nu
}

function GlobalStats({ stats }) {
  const { cases, deaths, recovered, active, updated } = stats;

  return (
    <div className='global-stats'>
```

```
      <small>Updated on {new Date(updated).toLocaleString()}</s
      <table>
        <tr>
          <td>
            Cases: <Stat number={cases} color='red' />
          </td>
          <td>
            Deaths: <Stat number={deaths} color='gray' />
          </td>
          <td>
            Recovered: <Stat number={recovered} color='green' /
          </td>
          <td>
            Active: <Stat number={active} color='orange' />
```

可以看到，`GlobalStats` 就是一个简单的函数式组件，没有任何钩子。

然后修改 `src/App.js`，引入刚刚创建的 `GlobalStats` 组件，代码如下：

```
import React, { useState, useEffect } from "react";

import "./App.css";
import GlobalStats from "./components/GlobalStats";

const BASE_URL = "https://corona.lmao.ninja/v2";

function App() {
  const [globalStats, setGlobalStats] = useState({});

  useEffect(() => {
    const fetchGlobalStats = async () => {
      const response = await fetch(`${BASE_URL}/all`);
      const data = await response.json();
      setGlobalStats(data);
    };

    fetchGlobalStats();
    const intervalId = setInterval(fetchGlobalStats, 5000);

    return () => clearInterval(intervalId);
  }, []);

  return (
    <div className='App'>
      <h1>COVID-19</h1>
```

可以看到，我们在 `App` 组件中，首先通过 `useState` 钩子引入了 `globalStats` 状态变量，以及修改该状态的函数。然后通过 `useEffect` 钩子获取 API 数据，其中有以下需要注意的点：

1. 我们通过定义了一个 `fetchGlobalStats` 异步函数并进行调用从而获取数据，而不是直接把这个 async 函数作为 `useEffect` 的第一个参数；
2. 创建了一个 Interval，用于每 5 秒钟重新获取一次数据；
3. 返回一个函数，用于销毁之前创建的 Interval。

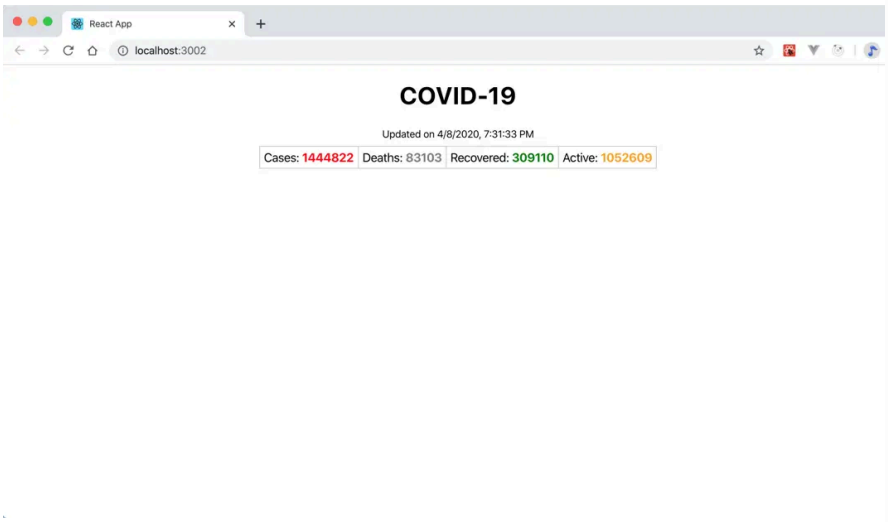
此外，第二个参数（依赖数组）为空数组，因此整个 Effect 函数只会运行一次。

> **注意**
>
> 有时候，你也许会不经意间把 Effect 写成一个 async 函数：
>
> ```
> useEffect(async () => {
>   const response = await fetch('...');
>   // ...
> }, []);
> ```
>
> 这样可以吗？**强烈建议你不要这样做。**`useEffect` 约定 Effect 函数要么没有返回值，要么返回一个 Cleanup 函数。而这里 async 函数会隐式地返回一个 Promise，直接违反了这一约定，会造成不可预测的结果。

最后附上应用的全局 CSS 文件，代码如下（直接复制粘贴即可）：

```
.App {
  width: 1200px;
  margin: auto;
  text-align: center;
}

.history-group {
  display: flex;
  justify-content: center;
  width: 1200px;
  margin: auto;
}

table,
th,
td {
  border: 1px solid #ccc;
  border-collapse: collapse;
}

th,
td {
  padding: 5px;
  text-align: left;
}
```

通过 `npm start` 开启项目：



此外，你可以检查一下控制台的 Network 选项卡，应该能看到我们的应用每五秒就会发起一次请求查询最新的数据。恭喜你，成功地用 Hooks 进行了一次数据获取！

useState + useEffect：渐入佳境

在上一步骤中，我们在 `App` 组件中定义了一个 State 和 Effect，但是实际应用不可能这么简单，一般都需要多个 State 和 Effect，这时候又该怎么去理解和使用呢？

深入 useState 的本质

在上一节的动画中，我们看到每一次渲染组件时，我们都能通过一个神奇的钩子把状态"钩"过来，不过这些钩子从何而来我们打了一个问号。现在，是时候解开谜团了。

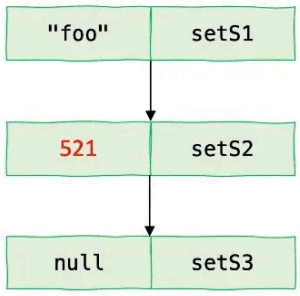
> **注意**
> 以下动画演示并不完全对应 React Hooks 的源码实现，但是它能很好地帮助你理解其工作原理。当然，也能帮助你去啃真正的源码。

我们先来看看当组件初次渲染（挂载）时，情况到底是什么样的：

注意以下要点：

1. 在初次渲染时，我们通过 `useState` 定义了多个状态；
2. 每调用一次 `useState` ，都会在组件之外生成一条 Hook 记录，同时包括状态值（用 `useState` 给定的初始值初始化）和修改状态的 Setter 函数；
3. 多次调用 `useState` 生成的 Hook 记录形成了一条**链表**；
4. 触发 `onClick` 回调函数，调用 `setS2` 函数修改 `s2` 的状态，不仅修改了 Hook 记录中的状态值，还即将**触发重渲染。**

OK，重渲染的时候到了，动画如下：



可以看到，在初次渲染结束之后、重渲染之前，Hook 记录链表依然存在。当我们逐个调用 `useState` 的时候，`useState` 便返回了 Hook 链表中存储的状态，以及修改状态的 Setter。

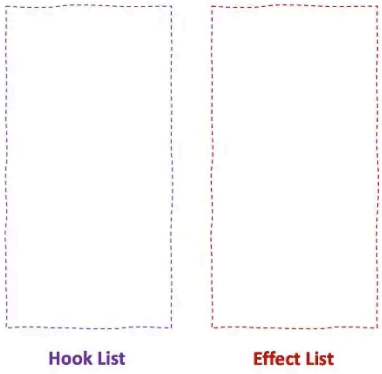
> **提示**
> 当你充分理解上面两个动画之后，其实就能理解为什么这个 Hook 叫 `useState` 而不是 `createState` 了——之所以叫 `use` ，是因为没有的时候才创建（初次渲染的时候），有的时候就直接读取（重渲染的时候）。

通过以上的分析，我们不难发现 `useState` 在设计方面的精巧（摘自张立理：对 React Hooks 的一些思考）：

- 状态和修改状态的 Setter 函数两两配对，并且后者一定影响前者，前者只被后者影响，作为一个整体它们完全不受外界的影响
- 鼓励细粒度和扁平化的状态定义和控制，对于代码行为的可预测性和可测试性大有帮助
- 除了 `useState` （和其他钩子），函数组件依然是实现渲染逻辑的"纯"组件，对状态的管理被 Hooks 所封装了起来

深入 useEffect 的本质

在对 `useState` 进行一波深挖之后，我们再来揭开 `useEffect` 神秘的面纱。实际上，你可能已经猜到了——同样是通过一个链表记录所有的 Hook，请看下面的演示：



注意其中一些细节：

1. `useState` 和 `useEffect` 在每次调用时都被添加到 Hook 链表中；
2. `useEffect` 还会额外地在一个队列中添加一个等待执行的 Effect 函数；
3. 在渲染完成后，依次调用 Effect 队列中的每一个 Effect 函数。

至此，上一节的动画中那两个"问号"的身世也就揭晓了——只不过是**链表**罢了！回过头来，我们想起来 React 官方文档 Rules of Hooks 中强调过一点：

> Only call hooks at the top level. 只在最顶层使用 Hook。

具体地说，不要在循环、嵌套、条件语句中使用 Hook——因为这些动态的语句很有可能会导致每次执行组件函数时调用 Hook 的顺序不能完全一致，导致 Hook 链表记录的数据失效。具体的场景就不画动画啦，自行脑补吧~

不要撒谎：关于 deps 的那些事

`useEffect` （包括其他类似的 `useCallback` 和 `useMemo` 等）都有个依赖数组（`deps`）参数，这个参数比较有趣的一点是：指定依赖的决定权完全在你手里。你当然可以选择"撒谎"，不管什么情况都给一个空的 `deps` 数组，仿佛在说"这个 Effect 函数什么依赖都没有，相信我"。

然而，这种有点偷懒的做法显然会引来各种 Bug。一般来说，所使用到的 `prop` 或者 `state` 都应该被添加到 `deps` 数组里面去。并且，React 官方还推出了一个专门的 ESLint 插件，可以帮你自动修复 `deps` 数组（*说实话，这个插件的自动修复有时候还是挺闹心的……*）。

实战环节

从这一步开始，我们将使用 Recharts 作为可视化应用的图表库，它提供了出色的 D3 和 React 的绑定层。通过如下命令添加 `recharts` 依赖：

```
npm install recharts
```

创建 `src/components/CountriesChart.js` ，用于展示多个国家的相关数据直方图，代码如下：

```
import React from "react";
import {
  BarChart,
  CartesianGrid,
  XAxis,
  YAxis,
  Tooltip,
  Legend,
  Bar,
} from "recharts";

function CountriesChart({ data, dataKey }) {
  return (
    <BarChart
      width={1200}
      height={250}
      style={{ margin: "auto" }}
      margin={{ top: 30, left: 20, right: 30 }}
      data={data}
    >
      <CartesianGrid strokeDasharray='3 3' />
      <XAxis dataKey='country' />
      <YAxis />
      <Tooltip />
      <Legend />
```

```
      <Bar dataKey={dataKey} fill='#8884d8' />
```

创建 `src/components/SelectDataKey.js`，用于选择需要展示的关键指标，代码如下：

```
import React from "react";

function SelectDataKey({ onChange }) {
  return (
    <>
      <label htmlFor='key-select'>Select a key for sorting: </lab
      <select id='key-select' onChange={onChange}>
        <option value='cases'>Cases</option>
        <option value='todayCases'>Today Cases</option>
        <option value='deaths'>Death</option>
        <option value='recovered'>Recovered</option>
        <option value='active'>Active</option>
      </select>
    </>
  );
}

export default SelectDataKey;
```

`SelectDataKey` 用于让用户选择以下关键指标：

- `cases`：累积确诊病例
- `todayCases`：今日确诊病例
- `deaths`：累积死亡病例
- `recovered`：治愈人数
- `active`：现存确诊人数

最后我们在根组件 `src/App.js` 中引入上面创建的两个组件，代码如下：

```
// ...
import GlobalStats from "./components/GlobalStats";
import CountriesChart from "./components/CountriesChart";
import SelectDataKey from "./components/SelectDataKey";

const BASE URL = "https://corona.lmao.ninja/v2";
```



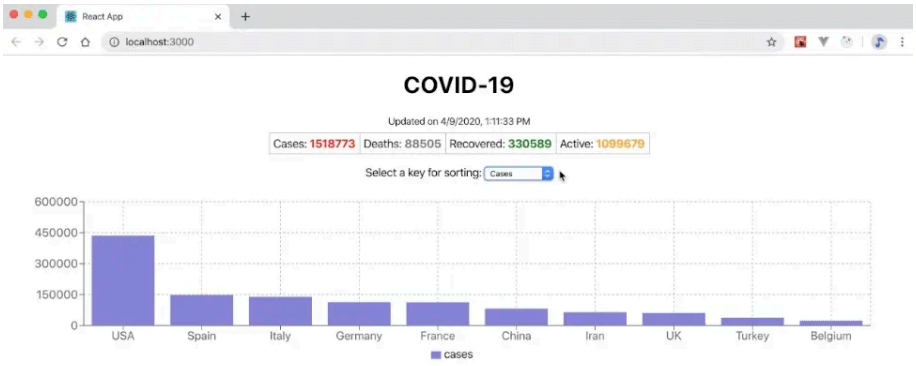
```
  useEffect(() => {
    // ...
```

```
  }, []);

  useEffect(() => {
    const fetchCountries = async () => {
      const response = await fetch(`${BASE_URL}/countries?sort=
      const data = await response.json();
      setCountries(data.slice(0, 10));
    };

    fetchCountries();
  }, [key]);
```

可以看到：

1. 我们创建了两个新的状态 `countries` （所有国家的数据）和 `key` （数据排序的指标，就是上面的五个）；
2. 我们又通过一个 `useEffect` 钩子进行数据获取，和之前获取全球数据类似，只不过注意我们这边第二个参数（依赖数组）是 `[key]` ，也就是只有当 `key` 状态改变的时候，才会调用 `useEffect` 里面的函数。
3. 最后使用之前创建的两个子组件，传入相应的数据和回调函数。

把项目跑起来，可以看到直方图显示了前十个国家的数据，并且可以修改排序的指标（比如可以从默认的累积确诊 `cases` 切换成死亡人数 `deaths`



看上去挺不错的！

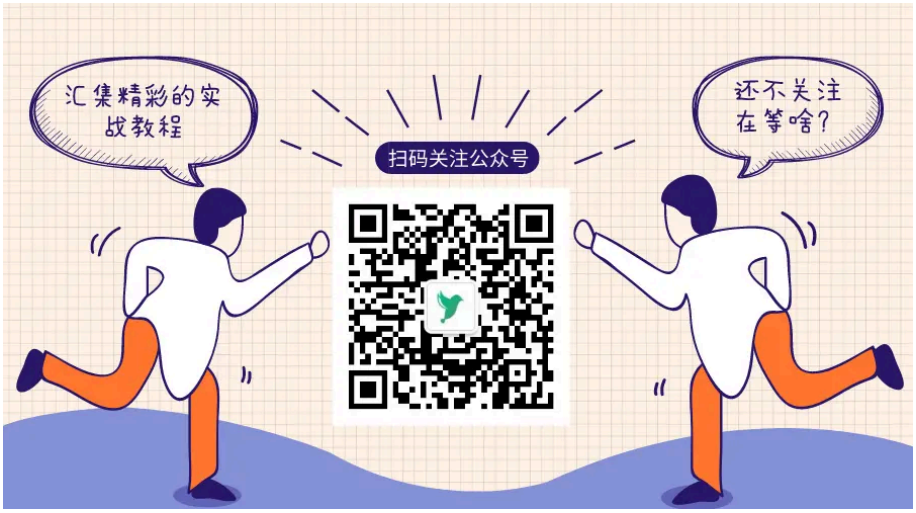
到这里，本系列第一篇也就讲完啦，希望你真正理解了 `useState` 和 `useEffect` ——最最最常用的两个 Hook。在下一篇教程中，我们将继续讲解自定义 Hook 和 `useCallback`。

参考资料

- React 官方文档
- Robin Wieruch：How to fetch data with React Hooks?
- Dan Abramov：A Complete Guide to useEffect

- Dan Abramov：How Are Function Components Different from Classes?
- Rudi Yardley：React hooks: not magic, just arrays
- Eytan Manor：Under the hood of React's hooks system
- 衍良：React Hooks 完全上手指南
- 张立理：对 React Hooks 的一些思考

> 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 6 收藏 3 分享

阅读 3.1k • 发布于 2020-05-05

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 »
一杯茶的时间，上手 Jest 测试框架 用动画和实战打开 React Hooks...

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。