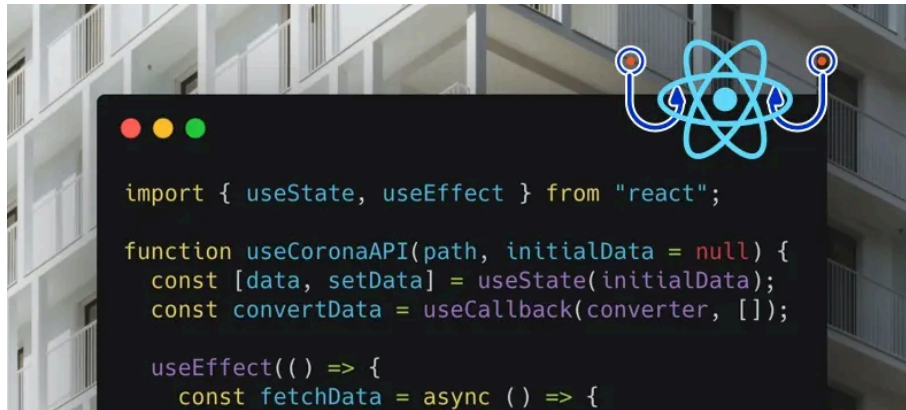


用动画和实战打开 React Hooks (二) : 自定义 Hook 和 useCallback



一只图雀

2020-05-06 阅读 9 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？

在第二篇教程中，我们将手把手带你用自定义 Hook 重构之前的组件代码，让它变得更清晰、并且可以实现逻辑复用。在重构完成之后，我们陷入了组件“不断获取数据并重新渲染”的无限循环，这时候，useCallback 站了出来，如同定海神针一般拯救了我们的应用.....

欢迎访问本项目的 [GitHub 仓库](#)和 [Gitee 仓库](#)。

自定义 Hook：量身定制

在[上一篇教程](#)中，我们通过动画的方式不断深入 `useState` 和 `useEffect`，基本上理清了 React Hooks 背后的实现机制——**链表**，同时也实现了

如果你想直接从这一篇教程开始阅读和实践，可下载本教程的源码：

```
git clone -b second-part https://github.com/tutture-dev/covid-19-w  
# 或者克隆 Gitee 的仓库  
git clone -b second-part https://gitee.com/tutture/covid-19-with-h
```

自定义 Hook 是 React Hooks 中最有趣的功能，或者说特色。简单来说，它用一种高度灵活的方式，能够让你在不同的函数组件之间共享某些特定的逻辑。我们先来通过一个非常简单的例子来看一下。

一个简单的自定义 Hook

先来看一个 Hook，名为 `useBodyScrollPosition`，用于获取当前浏览器的垂直滚动位置：

```
function useBodyScrollPosition() {  
  const [scrollPosition, setScrollPosition] = useState(null);  
  
  useEffect(() => {  
    const handleScroll = () => setScrollPosition(window.scrollY);  
    document.addEventListener('scroll', handleScroll);  
    return () => {  
      document.removeEventListener('scroll', handleScroll);  
    };  
  }, []);  
  
  return scrollPosition;  
}
```

通过观察，我们可以发现自定义 Hook 具有以下特点：

- 表面上：一个命名格式为 `useXXX` 的函数，但不是 React 函数式组件
- 本质上：内部通过使用 React 自带的一些 Hook（例如 `useState` 和 `useEffect`）来实现某些通用的逻辑

如果你发散一下思维，可以想到有很多地方可以去做自定义 Hook：DOM 副作用修改/监听、动画、请求、表单操作、数据存储等等。

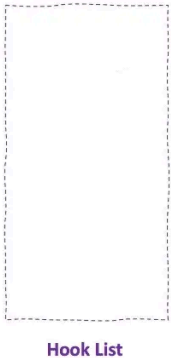
提示

这里推荐两个强大的 React Hooks 库：[React Use](#) 和 [Umi Hooks](#)。它们都实现了很多生产级别的自定义 Hook，非常值得学习。

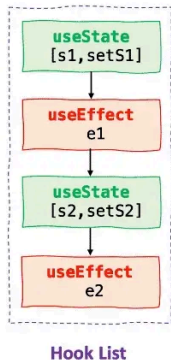
我想这便是 React Hooks 最大的魅力——通过几个内置的 Hook，你可以按照某些约定进行任意组合，“制造出”任何你真正需要的 Hook，或者调用他人写好的 Hook，从而轻松应对各种复杂的业务场景。就好像大千世界无奇不有，却不过是由一百多种元素组合而成。

管窥自定义 Hook 背后的原理

又到了动画时间。我们来看看在组件初次渲染时的情形：



我们在 App 组件中调用了 useCustomHook 钩子。可以看到，即便我们切换到了自定义 Hook 中，Hook 链表的生成依旧没有改变。再看看重渲染的情况：



同样地，即便代码的执行进入到自定义 Hook 中，我们依然可以从 Hook 链表中读取到相应的数据，这个“配对”的过程总能成功。

我们再次回味一下 Rules of Hook。它规定只有在两个地方能够使用 React Hook：

1. React 函数组件
2. 自定义 Hook

第一点我们早就清楚了，第二点通过刚才的两个动画相信你也明白了：**自定义 Hook 本质上只是把调用内置 Hook 的过程封装成一个个可以复用的函数，并不影响 Hook 链表的生成和读取。**

实战环节

让我们继续 COVID-19 数据应用的开发。接下来，我们打算实现历史数据的展示，包括确诊病例、死亡病例和治愈人数。

我们首先来实现一个自定义 Hook，名为 `useCoronaAPI`，用于共享从 NovelCOVID 19 API 获取数据的逻辑。创建 `src/hooks/useCoronaAPI.js`，填写代码如下：

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

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

export function useCoronaAPI(
  path,
  { initialData = null, converter = (data) => data, refetchInterval }
) {
  const [data, setData] = useState(initialData);

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(`${BASE_URL}${path}`);
      const data = await response.json();
      setData(converter(data));
    };
    fetchData();

    if (refetchInterval) {
      const intervalId = setInterval(fetchData, refetchInterval);
      return () => clearInterval(intervalId);
    }
  }, [converter, path, refetchInterval]);

  return data;
}
```

可以看到，定义的 `useCoronaAPI` 包含两个参数，第一个是 `path`，也就是 API 路径；第二是配置参数，包括以下参数：

- `initialData`：初始为空的默认数据
- `converter`：对原始数据的转换函数（默认是一个恒等函数）
- `refetchInterval`：重新获取数据的间隔（以毫秒为单位）

此外，我们还要注意 `useEffect` 所传入的 `deps` 数组，包括了三个元素（都是在 `Effect` 函数中用到的）：`converter`、`path` 和 `refetchInterval`，均来自 `useCoronaAPI` 传入的参数。

提示

在上一篇文章中，我们简单地提到过，不要对 `useEffect` 的依赖说谎，那么这里就是一个很好的案例：我们将 `Effect` 函数**所有用到的外部数据**（包括函数）全部加入到了依赖数组中。当然，由于 `BASE_URL` 属于模块级别的常量，因此不需要作为依赖。不过这里留了个坑，嘿嘿.....

然后在根组件 `src/App.js` 中使用刚刚创建的 `useCoronaAPI` 钩子，代码如下：

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

// ...
import { useCoronaAPI } from "../hooks/useCoronaAPI";

function App() {
  const globalStats = useCoronaAPI("/all", {
    initialData: {},
    refetchInterval: 5000,
  });

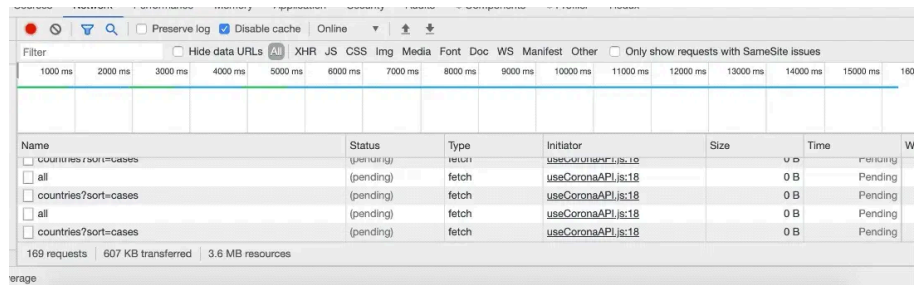
  const [key, setKey] = useState("cases");
  const countries = useCoronaAPI(`/countries?sort=${key}`, {
    initialData: [],
    converter: (data) => data.slice(0, 10),
  });

  return (
    // ...
  );
}

export default App;
```

整个 `App` 组件变得清晰了很多，不是吗？

但是当我们满怀期待地把应用跑起来，却发现整个应用陷入“无限请求”的怪圈中。打开 Chrome 开发者工具的 Network 选项卡，你会发现网络请求数量始终在飙升.....



吓得我们赶紧把网页关了。冷静下来之后，不禁沉思：这到底是为什么呢？

危险

NovelCOVID 19 API 属于公益性质的数据资源，我们应该尽快把页面关掉，避免给对方的服务器造成太大的请求压力。

useCallback：定海神针

如果你一字一句把[上一篇文章](#)看下来，其实可能已经发现了问题的线索：

依赖数组在判断元素是否发生改变时使用了 `Object.is` 进行比较，因此当 `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**，从而每次都会触发 Effect，失去了 `deps` 本身的意义。

OK，如果你没有印象也没关系，我们先来聊一聊初学 React Hooks 经常会遇到的一个问题：Effect 无限循环。

关于 Effect 无限循环

来看一下这段“永不停止”的计数器：

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

  useEffect(() => {
    setTimeout(() => setCount(count + 1), 1000);
  });

  return (
    <div className="App">
      <h1>{count}</h1>
    </div>
  );
}
```

如果你去运行这段代码，会发现数字永远在增长。我们来通过一段动画来演示一下这个“无限循环”到底是怎么回事：

我们的组件陷入了：**渲染 => 触发 Effect => 修改状态 => 触发重渲染**的无限循环。

想必你已经发现 `useEffect` 陷入无限循环的“罪魁祸首”了——因为没有提供正确的 `deps`！从而导致每次渲染后都会去执行 Effect 函数。事实上，在之前的 `useCoronaAPI` 中，也是因为传入的 `deps` 存在问题，导致每次渲染后都去执行 Effect 函数去获取数据，陷入了无限循环。那么，到底是哪个依赖出现了问题？

没错，就是那个 `converter` 函数！我们知道，在 JavaScript 中，原始类型和非原始类型在判断值是否相同的时候有巨大的差别：

```
// 原始类型
true === true // true
1 === 1 // true
'a' === 'a' // true

// 非原始类型
{} === {} // false
[] === [] // false
() => {} === () => {} // false
```

同样，每次传入的 `converter` 函数虽然形式上一样，但仍然是不同的函数（引用不相等），从而导致每次都会执行 Effect 函数。

关于记忆化缓存（Memoization）

Memoization，一般称为记忆化缓存（或者“记忆”），听上去是很高深的计算机专业术语，但是它背后的思想很简单：假如我们有一个**计算量很大的纯函数**（给定相同的输入，一定会得到相同的输出），那么我们在第一

次遇到特定输入的时候，把它的输出结果“记”（缓存）下来，那么下次碰到同样的输出，只需要从缓存里面拿出来直接返回就可以了，省去了计算的过程！

实际上，除了节省不必要的计算、从而提高程序性能之外，Memoization 还有一个用途：**用了保证返回值的引用相等。**

我们先通过一段简单的求平方根的函数，熟悉一下 Memoization 的原理。首先是一个没有缓存的版本：

```
function sqrt(arg) {  
  return { result: Math.sqrt(arg) };  
}
```

你也许注意到了我们特地返回了一个对象来记录结果，我们后面会和 Memoized 的版本进行对比分析。然后是加了缓存的版本：

```
function memoizedSqrt(arg) {  
  // 如果 cache 不存在，则初始化一个空对象  
  if (!memoizedSqrt.cache) {  
    memoizedSqrt.cache = {};  
  }  
  
  // 如果 cache 没有命中，则先计算，再存入 cache，然后返回结果  
  if (!memoizedSqrt.cache[arg]) {  
    return memoizedSqrt.cache[arg] = { result: Math.sqrt(arg) };  
  }  
  
  // 直接返回 cache 内的结果，无需计算  
  return memoizedSqrt.cache[arg];  
}
```



然后我们尝试调用这两个函数，就会发现一些明显的区别：

```
sqrt(9) // { result: 3 }  
sqrt(9) === sqrt(9) // false  
Object.is(sqrt(9), sqrt(9)) // false  
  
memoizedSqrt(9) // { result: 3 }  
memoizedSqrt(9) === memoizedSqrt(9) // true  
Object.is(memoizedSqrt(9), memoizedSqrt(9)) // true
```

普通的 `sqrt` 每次返回的结果的引用都不相同（或者说是一个全新的对象），而 `memoizedSqrt` 则能返回完全相同的对象。因此在 React 中，通

过 Memoization 可以确保多次渲染中的 Prop 或者状态的引用相等，从而能够避免不必要的重渲染或者副作用执行。

让我们来总结一下记忆化缓存 (Memoization) 的两个使用场景：

- 通过缓存计算结果，节省费时的计算
- 保证相同输入下返回值的引用相等

使用方法和原理解析

为了解决函数在多次渲染中的**引用相等** (Referential Equality) 问题，React 引入了一个重要的 Hook——`useCallback`。官方文档介绍的使用方法如下：

```
const memoizedCallback = useCallback(callback, deps);
```

第一个参数 `callback` 就是需要记忆的函数，第二个参数就是大家熟悉的 `deps` 参数，同样也是一个依赖数组（有时候也被称为输入 `inputs`）。在 Memoization 的上下文中，这个 `deps` 的作用相当于缓存中的键 (Key)，如果键没有改变，那么就直接返回缓存中的函数，并且确保是引用相同的函数。

在大多数情况下，我们都是传入空数组 `[]` 作为 `deps` 参数，这样 `useCallback` 返回的就**始终是同一个函数，永远不会更新**。

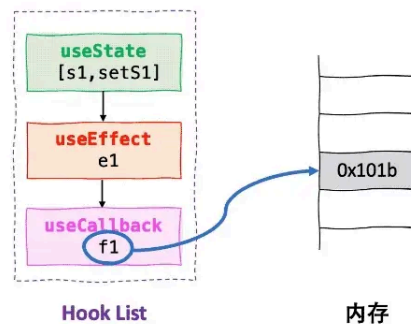
提示

你也许在刚开始学习 `useEffect` 的时候就发现：我们并不需要把 `useState` 返回的第二个 Setter 函数作为 Effect 的依赖。实际上，React 内部已经对 Setter 函数做了 Memoization 处理，因此每次渲染拿到的 Setter 函数都是完全一样的，`deps` 加不加都是没有影响的。

按照惯例，我们还是通过一段动画来了解一下 `useCallback` 的原理 (`deps` 为空数组的情况)，首先是初次渲染：

和之前一样，调用 `useCallback` 也是追加到 Hook 链表上，不过这里着重强调了这个函数 `f1` 所指向的内存位置（随便画了一个），从而明确告诉我们：**这个 `f1` 始终是指向同一个函数**。然后返回的 `onClick` 则是指向 Hook 中存储的 `f1`。

再来看看重渲染的情况：



重渲染的时候，再次调用 `useCallback` 同样返回给我们 `f1` 函数，并且这个函数还是指向同一块内存，从而使得 `onClick` 函数和上次渲染时真正做到了**引用相等**。


useCallback 和 useMemo 的关系

我们知道 `useCallback` 有个好基友叫 `useMemo`。还记得我们之前总结了 Memoization 的两大场景吗？`useCallback` 主要是为了解决函数的“**引用相等**”问题，而 `useMemo` 则是一个“**全能型选手**”，能够同时胜任引用相等和节约计算的任务。

实际上，`useMemo` 的功能是 `useCallback` 的**超集**。与 `useCallback` 只能缓存函数相比，`useMemo` 可以缓存任何类型的值（当然也包括函数）。

`useMemo` 的使用方法如下：

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b),
```



其中第一个参数是一个函数，这个函数返回值的返回值（也就是上面 `computeExpensiveValue` 的结果）将返回给 `memoizedValue`。因此以下两个钩子的使用是完全等价的：

```
useCallback(fn, deps);  
useMemo(() => fn, deps);
```

鉴于在前端开发中遇到的计算密集型任务是相当少的，而且浏览器引擎的性能也足够优秀，因此这一系列文章不会深入去讲解 `useMemo` 的使用。更何况，已经掌握 `useCallback` 的你，应该也已经知道怎么去使用 `useMemo` 了吧？

实战环节

熟悉了 `useCallback` 之后，我们开始修复 `useCoronaAPI` 钩子的问题。修改 `src/hooks/useCoronaAPI`，代码如下：

```
import { useState, useEffect, useCallback } from "react";  
  
// ...  
  
export function useCoronaAPI(  
  // ...  
) {  
  const [data, setData] = useState(initialData);  
  const convertData = useCallback(converter, []);  
  
  useEffect(() => {  
    const fetchData = async () => {  
      // ...  
      setData(convertData(data));  
    };  
    fetchData();  
  
    // ...  
  }, [convertData, path, refetchInterval]);  
  
  return data;  
}
```

可以看到，我们把 `converter` 函数用 `useCallback` 包裹了起来，把记忆化处理后的函数命名为 `convertData`，并且传入的 `deps` 参数为空数组 `[]`，

确保每次渲染都相同。然后把 `useEffect` 中所有的 `converter` 函数相应修改成 `convertData`。

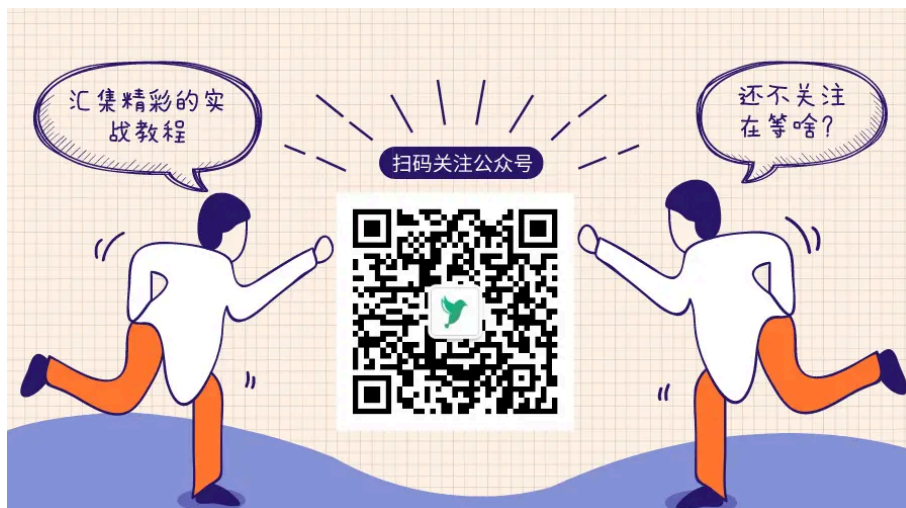
最后再次开启项目，一切又回归了正常，这次自定义 Hook 重构圆满完成！在下一篇教程中，我们将开始进一步推进 COVID-19 数据可视化项目的推进，通过曲线图的方式实现历史数据的展示（包括确诊、死亡和治愈）。数据状态变得越来越复杂，我们又该如何应对呢？敬请期待。

剧透提醒：用 `useReducer` + `useContext` 实现一个简单的 Redux！

参考资料

- [React 官方文档](#)
- DT-FE: [怎么用 React Hooks 造轮子](#)
- WellPaidGeed: [How to write custom hooks in React](#)
- Netlify Blog: [Deep dive: How do React hooks really work?](#)
- Andrew Myint: [How to Fix the Infinite Loop Inside "useEffect" \(React Hooks\)](#)
- Kent C. Dodds: [When to useMemo and useCallback](#)
- Sandro Dolidze: [React Hooks: Memoization](#)
- Chidume Nnamdi: [Understanding Memoization in JavaScript to Improve Performance](#)

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 1

收藏

分享



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

用动画和实战打开 React Hooks...

下一篇 »

一杯茶的时间，上手 Taro 京东小...

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。

©2024 图雀社区

除特别声明外，作品采用《署名-非商业性使用-禁止演绎 4.0 国际》进行许可

 使用 SegmentFault 发布

SegmentFault - 凝聚集体智慧，推动技术进步

服务协议 · 隐私政策 · 浙ICP备15005796号-2 · 浙公网安备33010602002000号