04-内置Hooks（2）：为什么要避免重复定义回调函数？

你好，我是王沛。这节课我们来继续学习内置 Hooks 的用法。

在上节课你已经看到了 useState 和 useEffect 这两个最为核心的 Hooks 的用法。理解了它们，你基本上就掌握了 React 函数组件的开发思路。

但是还有一些细节问题，例如事件处理函数会被重复定义、数据计算过程没有缓存等，还都需要一些机制来处理。所以在这节课，你会看到其它四个最为常用的内置 Hooks （包括useCallback、useMemo、useRef和useContext）的作用和用法，以及如何利用这些 Hooks 进行功能开发。

useCallback：缓存回调函数

在 React 函数组件中，每一次 UI 的变化，都是通过重新执行整个函数来完成的，这和传统的 Class 组件有很大区别：函数组件中并没有一个直接的方式在多次渲染之间维持一个状态。

比如下面的代码中，我们在加号按钮上定义了一个事件处理函数，用来让计数器加1。但是因为定义是在函数组件内部，因此在多次渲染之间，是无法重用 handleIncrement 这个函数的，而是每次都需要创建一个新的：

```
function Counter() {
  const [count, setCount] = useState(0);
  const handleIncrement = () => setCount(count + 1);
  // ...
  return <button onClick={handleIncrement}>+</button>
}
```

你不妨思考下这个过程。每次组件状态发生变化的时候，函数组件实际上都会重新执行一遍。在每次执行的时候，实际上都会创建一个新的事件处理函数 handleIncrement。这个事件处理函数中呢，包含了 count 这个变量的闭包，以确保每次能够得到正确的结果。

这也意味着，即使 count 没有发生变化，但是函数组件因为其它状态发生变化而重新渲染时，这种写法也会每次创建一个新的函数。创建一个新的事件处理函数，虽然不影响结果的正确性，但其实是没必要的。因为这样做不仅增加了系统的开销，更重要的是：**每次创建新函数的方式会让接收事件处理函数的组件，需要重新渲染**。

比如这个例子中的 button 组件，接收了 `handleIncrement` ，并作为一个属性。如果每次都是一个新的，那么这个 React 就会认为这个组件的 props 发生了变化，从而必须重新渲染。因此，我们需要做到的是：**只有当 count 发生变化时，我们才需要重新定一个回调函数。而**这正是 useCallback 这个 Hook 的作用。

它的 API 签名如下：

```
useCallback(fn, deps)
```

这里fn是定义的回调函数，deps是依赖的变量数组。只有当某个依赖变量发生变化时，才会重新声明 fn这个回调函数。那么对于上面的例子，我们可以把 handleIncrement这个事件处理函数通过 useCallback 来进行性能的优化：

```
import React, { useState, useCallback } from 'react';

function Counter() {
  const [count, setCount] = useState(0);
  const handleIncrement = useCallback(
    () => setCount(count + 1),
    [count], // 只有当 count 发生变化时，才会重新创建回调函数
  );
  // ...
  return <button onClick={handleIncrement}>+</button>
}
```

在这里，我们把 count 这个 state ，作为一个依赖传递给 useCallback。这样，只有 count 发生变化的时候，才需要重新创建一个回调函数，这样就保证了组件不会创建重复的回调函数。而接收这个回调函数作为属性的组件，也不会频繁地需要重新渲染。

除了useCallback，useMemo也是为了缓存而设计的。只不过，useCallback缓存的是一个函数，而useMemo缓存的是计算的结果。那么接下来，我们就一起学习下useMemo的用法吧。

useMemo：缓存计算的结果

useMemo 的 API 签名如下：

```
useMemo(fn, deps);
```

这里的fn是产生所需数据的一个计算函数。通常来说，fn会使用 deps 中声明的一些变量来生成一个结果，用来渲染出最终的 UI。

这个场景应该很容易理解：**如果某个数据是通过其它数据计算得到的，那么只有当用到的数据，也就是依赖的数据发生变化的时候，才应该需要重新计算**。

举个例子，对于一个显示用户信息的列表，现在需要对用户名进行搜索，且 UI 上需要根据搜索关键字显示过滤后的用户，那么这样一个功能需要有两个状态：

1. 用户列表数据本身：来自某个请求。
2. 搜索关键字：用户在搜索框输入的数据。

无论是两个数据中的哪一个发生变化，都需要过滤用户列表以获得需要展示的数据。那么如果不使用 useMemo 的话，就需要用这样的代码实现：

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

export default function SearchUserList() {
  const [users, setUsers] = useState(null);
  const [searchKey, setSearchKey] = useState("");

  useEffect(() => {
    const doFetch = async () => {
      // 组件首次加载时发请求获取用户数据
      const res = await fetch("https://reqres.in/api/users/");
      setUsers(await res.json());
    };
    doFetch();
  }, []);
  let usersToShow = null;

  if (users) {
    // 无论组件为何刷新，这里一定会对数组做一次过滤的操作
    usersToShow = users.data.filter((user) =>
      user.first_name.includes(searchKey),
    );
  }

  return (
    <div>
      <input
        type="text"
        value={searchKey}
        onChange={(evt) => setSearchKey(evt.target.value)}
      />
      <ul>
        {usersToShow &&
          usersToShow.length > 0 &&
          usersToShow.map((user) => {
            return <li key={user.id}>{user.first_name}</li>;
          })}
      </ul>
    </div>
  );
}
```

在这个例子中，无论组件为何要进行一次重新渲染，实际上都需要进行一次过滤的操作。但其实你只需要在 users 或者 searchKey 这两个状态中的某一个发生变化时，重新计算获得需要展示的数据就行了。那么，这个时候，我们就可以用 useMemo 这个 Hook 来实现这个逻辑，缓存计算的结果：

```
//...
// 使用 userMemo 缓存计算的结果
const usersToShow = useMemo(() => {
    if (!users) return null;
    return users.data.filter((user) => {
      return user.first_name.includes(searchKey));
    }
  }, [users, searchKey]);
//...
```

可以看到，通过 useMemo 这个 Hook，可以避免在用到的数据没发生变化时进行的重复计算。虽然例子展示的是一个很简单的场景，但如果是一个复杂的计算，那么对于提升性能会有很大的帮助。这也是userMemo的一大好处：避免重复计算。

除了避免重复计算之外，useMemo 还有一个很重要的好处：**避免子组件的重复渲染**。比如在例子中的 usersToShow 这个变量，如果每次都需要重新计算来得到，那么对于 UserList 这个组件而言，就会每次都需要刷新，因为它将 usersToShow 作为了一个属性。而一旦能够缓存上次的结果，就和 useCallback 的场景一样，可以避免很多不必要的组件刷新。

这个时候，如果我们结合 useMemo 和 useCallback 这两个 Hooks 一起看，会发现一个有趣的特性，那就是 **useCallback 的功能其实是可以用 useMemo 来实现的。**比如下面的代码就是利用 useMemo 实现了 useCallback 的功能：

```
const myEventHandler = useMemo(() => {
  // 返回一个函数作为缓存结果
  return () => {
    // 在这里进行事件处理
  }
}, [dep1, dep2]);
```

理解了这一点，相信你一下子会对这两个 Hooks 的机制有更进一步的认识，也就不用死记硬背两个 API 都是干嘛的了，因为从本质上来说，它们只是做了同一件事情：**建立了一个绑定某个结果到依赖数据的关系。只有当依赖变了，这个结果才需要被重新得到**。

useRef：在多次渲染之间共享数据

函数组件虽然非常直观，简化了思考 UI 实现的逻辑，但是比起 Class 组件，还缺少了一个很重要的能力：**在多次渲染之间共享数据。**

在类组件中，我们可以定义类的成员变量，以便能在对象上通过成员属性去保存一些数据。但是在函数组件中，是没有这样一个空间去保存数据的。因此，React 让useRef 这样一个 Hook 来提供这样的功能。

useRef 的API签名如下：

```
const myRefContainer = useRef(initialValue);
```

我们可以把useRef看作是在函数组件之外创建的一个容器空间。在这个容器上，我们可以通过唯一的 current 属设置一个值，从而在函数组件的多次渲染之间共享这个值。

你可能会有疑问，useRef 的这个功能具体有什么用呢？我们可以看一个例子。

假设你要去做一个计时器组件，这个组件有开始和暂停两个功能。很显然，你需要用 window.setInterval 来提供计时功能；而为了能够暂停，你就需要在某个地方保存这个 window.setInterval 返回的计数器的引

用，确保在点击暂停按钮的同时，也能用 window.clearInterval 停止计时器。那么，这个保存计数器引用的最合适的地方，就是 useRef，因为它可以存储跨渲染的数据。代码如下：

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

export default function Timer() {
  // 定义 time state 用于保存计时的累积时间
  const [time, setTime] = useState(0);

  // 定义 timer 这样一个容器用于在跨组件渲染之间保存一个变量
  const timer = useRef(null);

  // 开始计时的事件处理函数
  const handleStart = useCallback(() => {
    // 使用 current 属性设置 ref 的值
    timer.current = window.setInterval(() => {
      setTime((time) => time + 1);
    }, 100);
  }, []);

  // 暂停计时的事件处理函数
  const handlePause = useCallback(() => {
    // 使用 clearInterval 来停止计时
    window.clearInterval(timer.current);
    timer.current = null;
  }, []);

  return (
    <div>
      {time / 10} seconds.
      <br />
      <button onClick={handleStart}>Start</button>
      <button onClick={handlePause}>Pause</button>
    </div>
  );
}
```

这里可以看到，我们使用了 useRef 来创建了一个保存 window.setInterval 返回句柄的空间，从而能够在用户点击暂停按钮时清除定时器，达到暂停计时的目的。

同时你也可以看到，使用 useRef 保存的数据一般是和 UI 的渲染无关的，因此当 ref 的值发生变化时，是不会触发组件的重新渲染的，这也是 useRef 区别于 useState 的地方。

除了存储跨渲染的数据之外，useRef还有一个重要的功能，就是**保存某个 DOM 节点的引用**。我们知道，在 React 中，几乎不需要关心真实的 DOM 节点是如何渲染和修改的。但是在某些场景中，我们必须要获得真实 DOM 节点的引用，所以结合 React 的 ref属性和 useRef 这个 Hook，我们就可以获得真实的 DOM 节点，并对这个节点进行操作。

比如说，你需要在点击某个按钮时让某个输入框获得焦点，可以通过下面的代码来实现：

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
```

```
    // current 属性指向了真实的 input 这个 DOM 节点，从而可以调用 focus 方法
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

这段代码是 React 官方文档提供的一个例子，可以看到ref 这个属性提供了获得 DOM 节点的能力，并利用 useRef 保存了这个节点的应用。这样的话，一旦 input 节点被渲染到界面上，那我们通过 inputEl.current 就能访问到真实的 DOM 节点的实例了。

useContext：定义全局状态

在第2节课中你已经知道了，React 组件之间的状态传递只有一种方式，那就是通过 props。这就意味着这种传递关系只能在父子组件之间进行。

看到这里你肯定会问，如果要跨层次，或者同层的组件之间要进行数据的共享，那应该如何去实现呢？这其实就涉及到一个新的命题：**全局状态管理**。

为此，React 提供了 Context 这样一个机制，能够让所有在某个组件开始的组件树上创建一个 Context。这样这个组件树上的所有组件，就都能访问和修改这个 Context了。那么在函数组件里，我们就可以使用 useContext 这样一个 Hook 来管理 Context。

useContext 的API 签名如下：

```
const value = useContext(MyContext);
```

正如刚才提到的，一个 Context 是从某个组件为根组件的组件树上可用的，所以我们需要有 API 能够创建一个 Context，这就是 **React.createContext API，**如下：

```
const MyContext = React.createContext(initialValue);
```

这里的 MyContext 具有一个 Provider 的属性，一般是作为组件树的根组件。这里我仍然以 React 官方文档的例子来讲解，即：一个主题的切换机制。代码如下：

```
const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
```

```
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};
// 创建一个 Theme 的 Context

const ThemeContext = React.createContext(themes.light);
function App() {
  // 整个应用使用 ThemeContext.Provider 作为根组件
  return (
    // 使用 themes.dark 作为当前 Context
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

// 在 Toolbar 组件中使用一个会使用 Theme 的 Button
function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

// 在 Theme Button 中使用 useContext 来获取当前的主题
function ThemedButton() {
  const theme = useContext(ThemeContext);
  return (
    <button style={{
      background: theme.background,
      color: theme.foreground
    }}>
      I am styled by theme context!
    </button>
  );
}
```

看到这里你也许会有点好奇，Context 看上去就是一个全局的数据，为什么要设计这样一个复杂的机制，而不是直接用一个全局的变量去保存数据呢？

答案其实很简单，就是**为了能够进行数据的绑定**。当这个 Context 的数据发生变化时，使用这个数据的组件就能够自动刷新。但如果没有 Context，而是使用一个简单的全局变量，就很难去实现了。

不过刚才我们看到的其实是一个静态的使用 Context 的例子，直接用了 thems.dark 作为 Context 的值。那么如何让它变得动态呢？

比如说常见的切换黑暗或者明亮模式的按钮，用来切换整个页面的主题。事实上，动态 Context 并不需要我们学习任何新的 API，而是利用 React 本身的机制，通过这么一行代码就可以实现：

```
<ThemeContext.Provider value={themes.dark}>
```

可以看到，themes.dark 是作为一个属性值传给 Provider 这个组件的，如果要让它变得动态，其实只要用一个 state 来保存，通过修改 state，就能实现动态的切换Context 的值了。而且这么做，所有用到这个 Context 的地方都会自动刷新。比如这样的代码：

```
// ...

function App() {
  // 使用 state 来保存 theme 从而可以动态修改
  const [theme, setTheme] = useState("light");

  // 切换 theme 的回调函数
  const toggleTheme = useCallback(() => {
    setTheme((theme) => (theme === "light" ? "dark" : "light"));
  }, []);

  return (
    // 使用 theme state 作为当前 Context
    <ThemeContext.Provider value={themes[theme]}>
      <button onClick={toggleTheme}>Toggle Theme</button>
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

在这段代码中，我们使用 state 来保存 theme，从而达到可以动态调整的目的。

可以看到，Context 提供了一个方便在多个组件之间共享数据的机制。不过需要注意的是，它的灵活性也是一柄双刃剑。你或许已经发现，Context 相当于提供了一个定义 React 世界中全局变量的机制，而全局变量则意味着两点：

1. 会让调试变得困难，因为你很难跟踪某个 Context 的变化究竟是如何产生的。
2. 让组件的复用变得困难，因为一个组件如果使用了某个 Context，它就必须确保被用到的地方一定有这个 Context 的 Provider 在其父组件的路径上。

所以在 React 的开发中，除了像 Theme、Language 等一目了然的需要全局设置的变量外，我们很少会使用 Context 来做太多数据的共享。需要再三强调的是，Context 更多的是**提供了一个强大的机制，让 React 应用具备定义全局的响应式数据的能力**。

此外，很多状态管理框架，比如 Redux，正是利用了 Context 的机制来提供一种更加可控的组件之间的状态管理机制。因此，理解 Context 的机制，也可以让我们更好地去理解 Redux 这样的框架实现的原理。

小结

最后来总结一下今天的所学。在这节课，你看到了4个常用的 React内置 Hooks 的用法，包括：useCallback、useMemo、useRef 和 useContext。事实上，每一个 Hook 都是**为了解决函数组件中遇到的特定问题**。

因为函数组件首先定义了一个简单的模式来创建组件，但与此同时也暴露出了一定的问题。所以这些问题就要通过 Hooks 这样一个统一的机制去解决，可以称得上是一个非常完美的设计了。

有了这节课介绍的 4 个 Hooks，加上上节课我们学习的 useState 和 useEffect 这两个 核心Hooks，你几乎就能完成所有 React 功能的开发了。

当然，可能仍然会有一些边缘且复杂的特别场景，我们在这两节课中学习的 Hooks 并不能完全覆盖，那么我建议你可以去参考官方的 API 文档，先知道 React Hooks 还有哪些能力，以便在需要的时候能够查阅文档并使用。

思考题

useState 其实也是能够在组件的多次渲染之间共享数据的，那么在 useRef 的计时器例子中，我们能否用 state 去保存 window.setInterval() 返回的 timer 呢？

欢迎把你的想法和思考分享在留言区，我们一起交流讨论。下节课再见！

精选留言：

- 满月 2021-06-01 10:49:23
  我们能否用 state 去保存 window.setInterval() 返回的 timer 呢？
  我理解的是可以，只是没有 useRef 更优，因为在更新 state 值后会导致重新渲染，而 ref 值发生变化时，是不会触发组件的重新渲染的，这也是 useRef 区别于 useState 的地方。 [13赞]

  作者回复2021-06-01 22:46:55
  100分~

- 桃翁 2021-06-01 12:35:44
  useRef 如果只是用来 在多次渲染之间共享数据，是不是直接可以把变量定义到组件外面，这样也可以达到目的，感觉还更方便一点呢。 [5赞]

  作者回复2021-06-01 17:52:11
  useRef 可以保证这个变量只在当前组件的实例中使用。也就是说，如果一个组件页面上有多个实例，比如：
  <Timer /><Timer />
  那么组件外的普通变量是被 Timer 共享的，就会产生问题。

- cyh41 2021-06-01 10:52:04
  是任何场景 函数都用useCallback 包裹吗？那种轻量的函数是不是不需要？ [5赞]

  作者回复2021-06-01 22:48:54
  确实不是，useCallback 可以减少不必要的渲染，主要体现在将回调函数作为属性传给某个组件。如果每次都不一样就会造成组件的重新渲染。但是如果你确定子组件多次渲染也没有太大问题，特别是原生的组件，比如 button，那么不用 useCallback 也问题不大。所以这和子组件的实现相关，和函数是否轻量无关。但是比较好的实践是都 useCallback。

- 王雪 2021-06-01 23:50:06
  useCallBack依赖是空数组表示什么？ [2赞]

  作者回复2021-06-03 21:36:24
  没有意义，相当于每次都创建一个新的函数

- 琼斯基亚 2021-06-03 22:16:50
  老师，请问：

const handleIncrement = useCallback(() => setCount(count + 1), [count]);
const handleIncrement = useCallback(() => setCount(q => q + 1), []);
在性能方面是否后者优于前者？我的理解：
后者只创建了一次函数，但是又调用了多次在setCount的回调函数
前者只会在count变化的时候创建新的回调函数
这样分析下来我又觉得两者没什么差异
我不是太清楚这两者的优缺点，希望得到老师的解答。 [1赞]

- Geek_71adef 2021-06-01 09:08:47
1 useState 实现组件共享，考虑到组件之间的通信
2 state 去保存的话 会造成异步渲染 造成无限循环 [1赞]

作者回复2021-06-01 22:45:44
只有需要触发 UI 更新的状态才需要放到 state 里。这里的 timer 其实只是临时存放一个变量，无需用 state 保存。否则会造成不必要的渲染。

- 开开之之 2021-06-05 13:03:04
老师，下面是我写的例子。我发现当点击+按钮的时候，handleInputName也没有被重复定义，所以不明白useCallback的作用其实是？

```
import React, { useState, useCallback } from 'react'
import './style.css'

function Counter() {
const [count, setCount] = useState(0);
const [name, setName] = useState('')
const handleIncrement = useCallback(
() => {
console.log('define handleIncrement')
return setCount(count + 1)
},
[count]
)

const handleInputName = (event) => {
console.log('define handleInputName')
const value = event.target.value
return setName(value)
}


return <div className='container'>
<button onClick={handleIncrement}>+</button>
<input type="text" value={name} onChange={handleInputName} />
<p>{count}</p>
</div>
}

export default Counter
```

- 笨重的企鹅 2021-06-03 19:55:48
  useRef那个计时器的例子，不用useRef，直接声明一个普调变量：let timer = null（注意不能是const）也是OK的，所以多次渲染之间共享数据这个貌似不怎么实用

  作者回复2021-06-03 21:48:30
  能贴下完整的代码吗，理论上肯定要用 useRef 的~

- 七月有风 2021-06-03 17:25:22
  问下老师，useCallback、useMemo 和 useEffect的依赖机制一样吗？都是浅比较吗？

  作者回复2021-06-03 21:48:45
  是的，所以依赖比较都是浅比较

- Isaac 2021-06-03 14:20:21
  <button onClick={() => handleClick('hi')}></button>

  老师，上面这种写法，直接将箭头函数作为 props 传递给 button，是不是每次 render 的时候，也会生成一个新的箭头函数？

  如果是的话，怎么避免呢？

  作者回复2021-06-03 21:47:42
  是的，这种问题不大，因为 button 没有子节点，性能问题可以忽略。要避免的话就是用 useCallback。参数的话是可以在 useCallback 里处理的。

- 傻子来了快跑丶 2021-06-02 20:57:57
  老师讲的很棒，希望后面的课程质量更棒

  作者回复2021-06-03 21:39:08
  谢谢！

- D.W 2021-06-02 07:36:55
  老师，你好。
  1，如果用state保存timer，在清空times贵引起不必要的渲染吧

  2，请问像一些事件监听的函数，比如监听下拉框的变化，按钮的点击回调，应该没有必要用useCallback包裹吧？就像文中说的，使用useCallback主要是为了避免函数重新生成，接受函数作为参数的组件也重新渲染

  作者回复2021-06-03 21:36:58
  1. 正确~
  2. 没错，没有必要。

- aloha66 2021-06-01 12:25:15
  如果通过useMemo，依赖数组是容器组件的state,来优化展示组件的渲染，这个方案可行吗？
  const ItemRender = useMemo(() =><ComA {...props}/>},[state])
  ...
  {ItemRender}

  作者回复2021-06-01 22:50:54
  没有必要，因为虚拟 DOM 会帮你做这个优化。这种写法，其实虚拟 DOM 还是需要去整体进行 diff 计算

的。

- Sunny 2021-06-01 11:45:45

```
import React, {
useCallback,
useRef,
useReducer,
} from 'react';

const initialState = {time: 0};
function reducer(state, action) {
switch (action.type) {
case 'increment':
return {time: state.time + 1};
default:
throw new Error();
}
}

export default function Timer() {
console.log('--render--')

const [state, dispatch] = useReducer(reducer, initialState);

const timer = useRef(null);

const setIntervalCallback = useCallback( () => {
dispatch({type: 'increment'});
console.log('setinterval time:', state.time) //为什么这里的state.time不变?
}, [state.time]);//这里的state.time变化被监听到了

const handleStart = useCallback(() => {
console.log('handlestart')
timer.current = window.setInterval(setIntervalCallback, 1000);
}, [timer, setIntervalCallback]);

const handlePause = useCallback(() => {
console.log('handlePause')
window.clearInterval(timer.current);
timer.current = null;
}, [timer]);

return(
<div>
{state.time} seconds.
<MyStartBtn handleStart={handleStart}/>
<MyPauseBtn handlePause={handlePause}/>
</div>
);
}
```

```
function StartButton({handleStart}){
console.log('startButton render --')
return <button onClick={handleStart}>Start</button>;
}
const MyStartBtn = React.memo(StartButton, (prevProps, nextProps) => {
return prevProps.handleStart === nextProps.handleStart;
});

function PauseButton({handlePause}){
console.log('pauseButton render --')
return <button onClick={handlePause}>Pause</button>;
}
const MyPauseBtn = React.memo( PauseButton, (prev, next) => {
return prev.handlePause === next.handlePause;
})

/*
console.log打印结果：
--render--
startButton render --
setinterval time: 0
每秒循环打印上面3行...


疑问：
const setIntervalCallback = useCallback( () => {
dispatch({type: 'increment'});
console.log('setinterval time:', state.time) //为什么这里的state.time不变？
}, [state.time]);//这里的state.time变化被监听到了
*/
```

作者回复2021-06-01 22:54:45

因为点击 start 的时候，window.setInterval 用的 setIntervalCallback 这个函数已经是确定的了。之后 state 的变化并不会导致 setInterval 接收的函数换一个。所以一直是当时闭包里的 state.time = 0