10 | React Hooks（下）：用Hooks处理函数组件的副作用

2022-09-13 宋一玮 来自北京

《现代React Web开发实战》

课程介绍 >

讲述：宋一玮

时长 20:03　大小 18.32M

你好，我是宋一玮，欢迎回到 React 应用开发的学习。

上节课我们讲了什么是 Hooks，React 18 里都有哪些 Hooks，然后深入学习了基础 Hooks 之一的 `useState`，在结束前也介绍了 `useRef`。

这节课我们紧接着来学习另一个基础 Hook：`useEffect`，以及用于组件性能优化的 Hooks：`useMemo` 和 `useCallback`。讲完这些 Hooks，我们回过头了解一下所有 React Hooks 共通的使用规则。最后回答上节课一开始提到的疑问：

- 函数组件加 Hooks 可以完全替代类组件吗？
- 还有必要学习类组件吗？

  好的，我们先从 `useEffect` 开始。

什么是副作用？

副作用（Side-effect，或简称 Effect）这个概念在上节课已经多次出现了，你可能还是觉得迷惑，到底什么是副作用？

计算机领域的副作用是指：

> 当调用函数时，除了返回可能的函数值之外，还对主调用函数产生附加的影响。例如修改全局变量，修改参数，向主调方的终端、管道输出字符或改变外部存储信息等。
> ——《副作用（计算机科学）- 维基百科》

总之，**副作用就是让一个函数不再是纯函数的各类操作**。注意，这个概念并不是贬义的，在 React 中，大量行为都可以被称作副作用，比如挂载、更新、卸载组件，事件处理，添加定时器，修改真实 DOM，请求远程数据，在 console 中打印调试信息，等等。

上节课提到 state，其实是绑定在组件函数之外的 `FiberNode` 上的。这让你想到了什么？对的，组件函数执行 state 更新函数从逻辑上讲也是一种副作用。

副作用 Hooks：useEffect

面对这么多副作用，React 大大方方地提供了 `useEffect` 这个执行副作用操作的 Hook。当你打算在函数组件加入副作用时，`useEffect` 基本上会成为你的首选。同时也建议你务必把副作用放在 `useEffect` 里执行，而不是直接放在组件的函数体中，这样可以避免很多难以调试的 Bug。

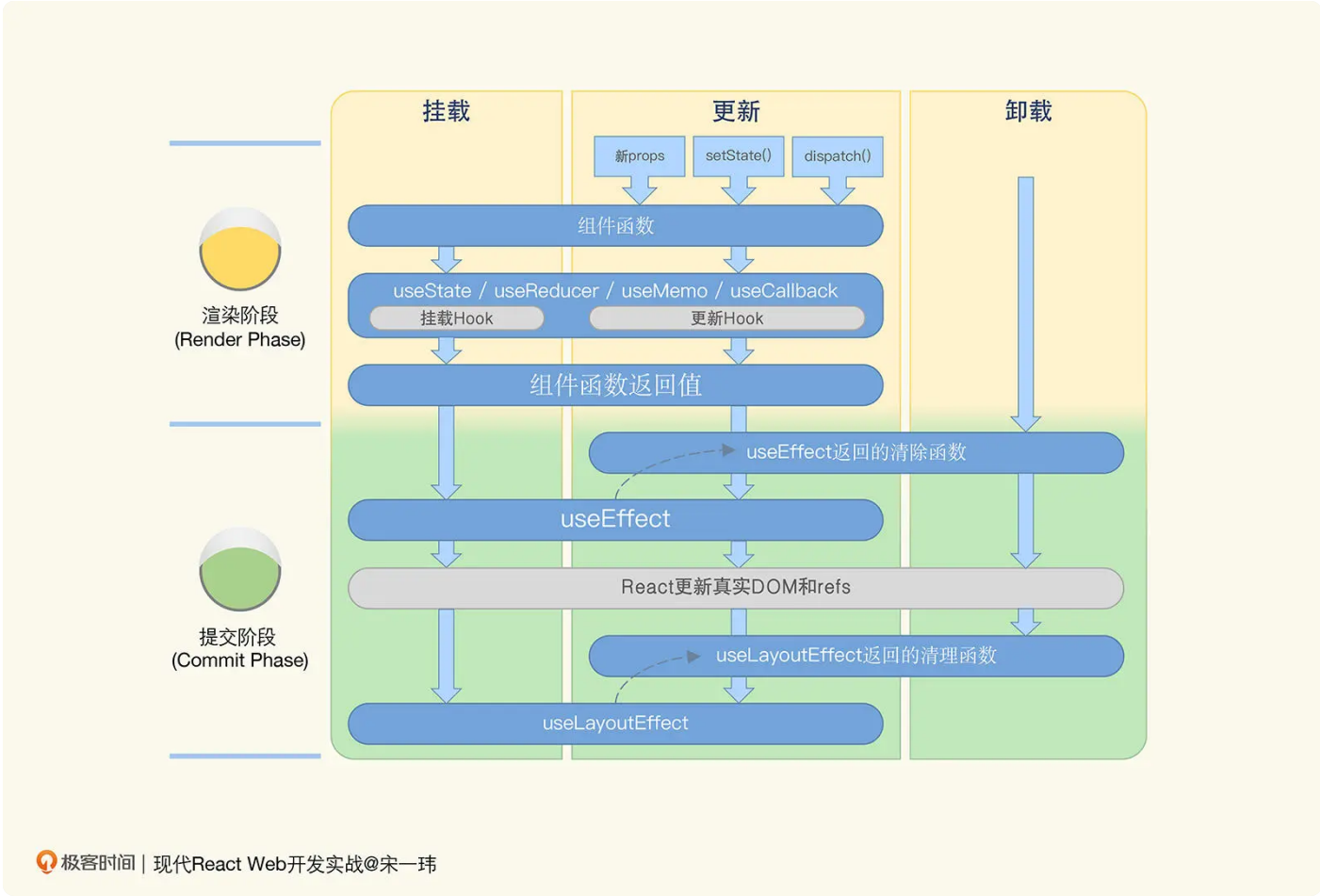
`useEffect` 这个 Hook 有几种用法。首先最简单的用法，只传入一个没有返回值的**副作用回调函数**（Effect Callback）：

复制代码

```
useEffect(() => {/* 省略 */});
//        -----------------
//                ^
//                |
//           副作用回调函数
```

虽然 useEffect 作为组件函数体的一部分，在每次组件渲染（包括挂载和更新阶段）时都会被调用，但作为参数的副作用回调函数是在**提交阶段**才会被调用的，这时**副作用回调函数可以访问到组件的真实 DOM**。

虽然这是最简单的用法，但现实中的用例反而比较少：毕竟每次渲染后都会被调用，如果使用不当，容易产生性能问题。这里提到了上节课讲到的渲染阶段和提交阶段，我把当时画的图贴过来，方便你参考。



接下来就是最常用的用法：**副作用的条件执行**。在上面用法的基础上，传入一个**依赖值数组**（Dependencies）作为第二个参数：

```
useEffect(() => {/* 省略 */}, [var1, var2]);
//        -----------------   -----------
//                ^                ^
//                |                |
//          副作用回调函数        依赖值数组
```

React 在渲染组件时，会记录下当时的依赖值数组，下次渲染时会把依赖值数组里的值依次与前一次记录下来的值做**浅对比**（Shallow Compare）。如果有不同，才会在提交阶段执行副作用回调函数，否则就跳过这次执行，下次渲染再继续对比依赖值数组。

依赖值数组里可以加入 props、state、context 值。一般来说，只要副作用回调函数中用到了自己范围之外的变量，都应该加入到这个数组里，这样 React 才能知道应用状态的变化和副作用间的因果关系。

下面来一个级联菜单的例子，当省份 state 值更新时，副作用回调函数会根据省份来更新城市列表，而城市列表也是一个 state，state 更新会使组件重新渲染（rerender），以达到刷新二级菜单选项的目的。

复制代码

```
//   ------------   --------------
//   | 省份... |v|   | 城市...  |v|
//   ------------   --------------

const [province, setProvince] = useState(null);
const [cities, setCities] = useState([]);
useEffect(() => {
  if (province === '山东') {
    // 这些数据可以是本地数据，也可以现从服务器端读取
    setCities(['济南', '青岛', '淄博']);
  }
}, [province]);
```

空数组`[]`也是一个有效的依赖值数组，由于在组件生命周期中依赖值不会有任何变化，所以副作用回调函数只会在组件挂载时执行一次，之后不论组件更新多少次，副作用都不会再执行。这个用法可以用来加载远程数据。

请你跟随我，立刻上手为 oh-my-kanban 项目加入远程数据的存取。为了简化实现，我们会使用浏览器内置的 `localStorage` 本地存储 API 代替远程服务。同样，为了简化逻辑，我们会利用 `JSON.stringify` 和 `JSON.parse` 序列化和反序列化看板列数据，直接读写 `localStorage` 中的单一 key。

在 `src/App.js` 的 App 组件代码中加入一个只在挂载时执行一次的 `useEffect`，在副作用回调函数中读取数据，为了模拟远程服务的耗时，我们加上一个 1 秒钟的计时器：

复制代码

```
const DATA_STORE_KEY = 'kanban-data-store';

function App() {
  const [showAdd, setShowAdd] = useState(false);
  const [todoList, setTodoList] = useState([/*...省略*/]);
  const [ongoingList, setOngoingList ] = useState([/*...省略*/]);
  const [doneList, setDoneList ] = useState([/*...省略*/]);
  useEffect(() => {
    const data = window.localStorage.getItem(DATA_STORE_KEY);
    setTimeout(() => {
      if (data) {
        const kanbanColumnData = JSON.parse(data);
        setTodoList(kanbanColumnData.todoList);
        setOngoingList(kanbanColumnData.ongoingList);
        setDoneList(kanbanColumnData.doneList);
      }
    }, 1000);
  },[]);
  // ...省略
}
```

有了读取，还需要有存储。在实际业务中，因为涉及到本地数据和远程数据的同步，这部分逻辑可能会非常复杂，而我们这里用一个偷懒的方法：加入一个“保存所有卡片”的按钮，由用户来决定什么时候存储。

复制代码

```
const DATA_STORE_KEY = 'kanban-data-store';

function App() {
  // ...省略
  const handleSaveAll = () => {
    const data = JSON.stringify({
      todoList,
      ongoingList,
      doneList
    });
    window.localStorage.setItem(DATA_STORE_KEY, data);
  };
  // ...省略
  return (
    <div className="App">
      <header className="App-header">
        <h1>我的看板 <button onClick={handleSaveAll}>保存所有卡片</button></h1>
        <img src={logo} className="App-logo" alt="logo" />
      </header>
      {/* ...省略 */}
    </div>
```

```
22     );
23   }
```

回到浏览器中，添加新卡片，再点击新加入的“保存所有卡片”按钮，你会在浏览器开发者工具的 Local Storage 中，找到一条新的数据。这时刷新浏览器，你会发现新添加的卡片还在，不像之前一刷就没了。

不过刚刷新浏览器后，1 秒时页面的突然变化还是有点突兀的。我们来加入一个读取状态提示：

```
 const COLUMN_BG_COLORS = {
+  loading: '#E3E3E3',
   todo: '#C9AF97',
   ongoing: '#FFE799',
   done: '#C0E8BA'
 };
 const DATA_STORE_KEY = 'kanban-data-store';
 function App() {
   // ...省略
+  const [isLoading, setIsLoading] = useState(true);
   useEffect(() => {
     const data = window.localStorage.getItem(DATA_STORE_KEY);
     setTimeout(() => {
       if (data) {
         const kanbanColumnData = JSON.parse(data);
         setTodoList(kanbanColumnData.todoList);
         setOngoingList(kanbanColumnData.ongoingList);
         setDoneList(kanbanColumnData.doneList);
       }
+       setIsLoading(false);
     }, 1000);
   },[]);
   // ...省略
   return (
     <div className="App">
       {/*...省略*/}
       <KanbanBoard>
+         {isLoading ? (
+           <KanbanColumn title="读取中..." bgColor={COLUMN_BG_COLORS.loading
+         ) : (<>
           <KanbanColumn bgColor={COLUMN_BG_COLORS.todo} title={/*待处理*/}>
             {/*...省略*/}
           </KanbanColumn>
+         </>)}
       </KanbanBoard>
     </div>
   );
 }
```

在浏览器中看下效果：

太棒了！你利用 `useEffect(effectCallback, [])` 完成了 App 挂载时读取“远程数据”的功能。

多提一句，依赖值数组并不是副作用 Hooks 专有的概念，`useCallback` 、 `useMemo` 也接受依赖值数组作为第二参数。后面的课程会详细讲解。

我们再来看一下第 8 节课在 oh-my-kanban 中加入的定时器功能：

复制代码

```
const KanbanCard = ({ title, status }) => {
  const [displayTime, setDisplayTime] = useState(status);
  useEffect(() => {
    const updateDisplayTime = () => {
      const timePassed = new Date() - new Date(status);
      let relativeTime = '刚刚';
      // ...省略
      setDisplayTime(relativeTime);
    };
    const intervalId = setInterval(updateDisplayTime, UPDATE_INTERVAL);
    updateDisplayTime();

    return function cleanup() {
      clearInterval(intervalId);
    };
  }, [status]);
```

可以看到，useEffect 接收了副作用回调函数和依赖值数组两个参数，其中副作用回调函数的返回值也是一个函数，这个返回的函数叫做**清除函数**。组件在下一次提交阶段执行同一个副作用回调函数之前，或者是组件即将被卸载之前，会调用这个清除函数。

同时定义副作用回调函数、清除函数和依赖值数组，这是 useEffect 最完整的一种用法。

```
复制代码
useEffect(() => {/* 省略 */; return () => {/* 省略 */};}, [status]);
//        ------------------------------------------     -------
//                          ^          -----------------        ^
//                          |                ^                  |
//                  副作用回调函数          清除函数          依赖值数组
```

回到上面定时器的例子中，可以看出，当组件挂载，以及传入组件的 status 属性发生变化时，会执行 setInterval、setDisplayTime 两个副作用操作。当组件的 status 属性再次变化时，以及组件被卸载时，会调用 cleanup 清除函数清理掉仍在运行的定时器。

在调用setDisplayTime 更新 state 后，组件会重新渲染，在页面上就能看到卡片显示了最新的相对时间。如果不清理定时器会怎样？如果是在更新阶段，组件就可能会有多个定时器在跑，会产生**竞争条件**；如果组件已被卸载，那么有可能导致**内存泄露**。

如果依赖值数组是一个**空数组**，那么清除函数只会在卸载组件时执行。

对比上节课讲到的类组件生命周期方法，useEffect 根据用法的不同，可以很容易地实现 componentDidMount 、 componentWillUnmount 的功能，而且还能根据 props、state 的变化有条件地执行副作用，比类组件生命周期方法灵活很多。

副作用 Hooks 除了useEffect，还有一个名字类似、用法也类似的useLayoutEffect。它的副作用执行时机晚于前者，是在真实 DOM 变更之后**同步**执行的，更接近类组件的 componentDidMount 、componentWillUnmount 。为保证性能，应尽量使用 useEffect 以避免阻塞。

性能优化 Hooks：useMemo 和 useCallback

接下来，趁着你对 useEffect 的参数形式印象深刻，我们占用一小部分篇幅，了解一下用于组件性能优化的 Hooks：useMemo 和 useCallback。

其实这两个 Hooks 与 useEffect 并不沾亲带故。且不说它们的用途完全不同，单从回调函数的执行阶段来看，前者是在渲染阶段执行，而后者是在提交阶段。看起来它们最大的相似点，在于 Hook 的**第二个参数都是依赖值数组**。

这里插入一个概念：**记忆化（Memoization），对于计算量大的函数，通过缓存它的返回值来节省计算时间，提升程序执行速度**。对于记忆化函数的调用者而言，存入缓存这件事本身就是一种副作用。useMemo 和 useCallback 做性能优化的原理就是记忆化，所以它们的本质和 useEffect 一样，都是在处理副作用。

先来看一下useMemo ，这个 Hook 接受两个参数，一个是**工厂函数**（Factory），另一个是依赖值数组，它的返回值就是执行工厂函数的返回值：

复制代码

```
const memoized = useMemo(() => createByHeavyComputing(a, b), [a, b]);
//    --------           ----------------------------------  ------
//       ^                              ^                      ^
//       |                              |                      |
//   工厂函数返回值                   工厂函数               依赖值数组
```

useMemo 的功能是**为工厂函数返回一个记忆化的计算值**，在两次渲染之间，**只有依赖值数组中的依赖值有变化时，该 Hook 才会调用工厂函数重新计算**，将新的返回值记忆化并返回给组件。

useMemo 最重要的使用场景，是将执行成本较高的计算结果存入缓存，通过减少重复计算来提升组件性能。我们依旧用上节课的斐波那契数列递归函数来举例，从 state 中获取num ，转换成整数n 后传递给函数 ，即计算第 n 个斐波那契数：

复制代码

```
const [num, setNum] = useState('0');
const sum = useMemo(() => {
  const n = parseInt(num, 10);
  return fibonacci(n);
}, [num]);
```

状态num 的初始值是字符串 '0' ，组件挂载时 useMemo 会执行一次 fibonacci(0) 计算并返回 0 。如果后续通过文本框输入的方式修改 num 的值，如 '40' ， '40' 与上次的 '0' 不

同，则 useMemo 再次计算 fibonacci(40)，返回 102334155，如果后续其他 state 发生了改变，但 num 的值保持 '40' 不变，则 useMemo 不会执行工厂函数，直接返回缓存中的 102334155，减少了组件性能损耗。

然后是 useCallback，它会把作为第一个参数的回调函数返回给组件，只要第二个参数依赖值数组的依赖项不改变，它就会保证一直返回同一个回调函数（引用），而不是新建一个函数，这也保证了回调函数的闭包也是不变的；相反，当依赖项改变时， useCallback 才会更新回调函数及其闭包。

复制代码

```
const memoizedFunc = useCallback(() => {/*省略*/}, [a, b]);
//    ------------                ---------------   -----
//         ^                           ^            ^
//         |                           |            |
//   记忆化的回调函数                 回调函数      依赖值数组
```

其实 useCallback 是 useMemo 的一个马甲，相当于：

复制代码

```
const memoizedFunc = useMemo(() => () => {/*省略*/}, [a, b]);
//    ------------           ----------------------   -----
//       ^                      ^  ----------------      ^
//       |                      |         ^              |
// 工厂函数返回的回调函数      工厂函数   回调函数        依赖值数组
```

你可能会有疑问，从马甲视图看来，“工厂函数直接返回另一个函数”这种操作一点也不重啊，为什么说useCallback 也能用来优化组件性能的呢？

如果你还记得，上节课讲什么是纯函数时，我们顺带提到了纯组件的特性：当组件的 props 和 state 没有变化时，将跳过这次渲染。而你在函数组件内频繁声明的事件处理函数，比如 handleSubmit，在每次渲染时都会创建一个新函数。

如果把这个函数随着 props 传递给作为子组件的纯组件，则会导致纯组件的优化无效，因为每次父组件重新渲染都会带着子组件一起重新渲染。这时就轮到useCallback 出马了，使用妥当的话，子组件不会盲目跟随父组件一起重新渲染，这样的话，反复渲染子组件的成本就节省下来了。

上面介绍了`useMemo` 和 `useCallback` 的完整概念和最典型的使用场景。我们还会在后续的《数据不可变性》和《大型项目》两节课中遇到这两个 Hooks，届时会结合实际项目再做进一步讲解。

Hooks 的使用规则

我们前面学习了基础的状态和副作用 Hooks，以及部分扩展 Hooks，相信你对这种函数式的 API 有了更进一步的了解。

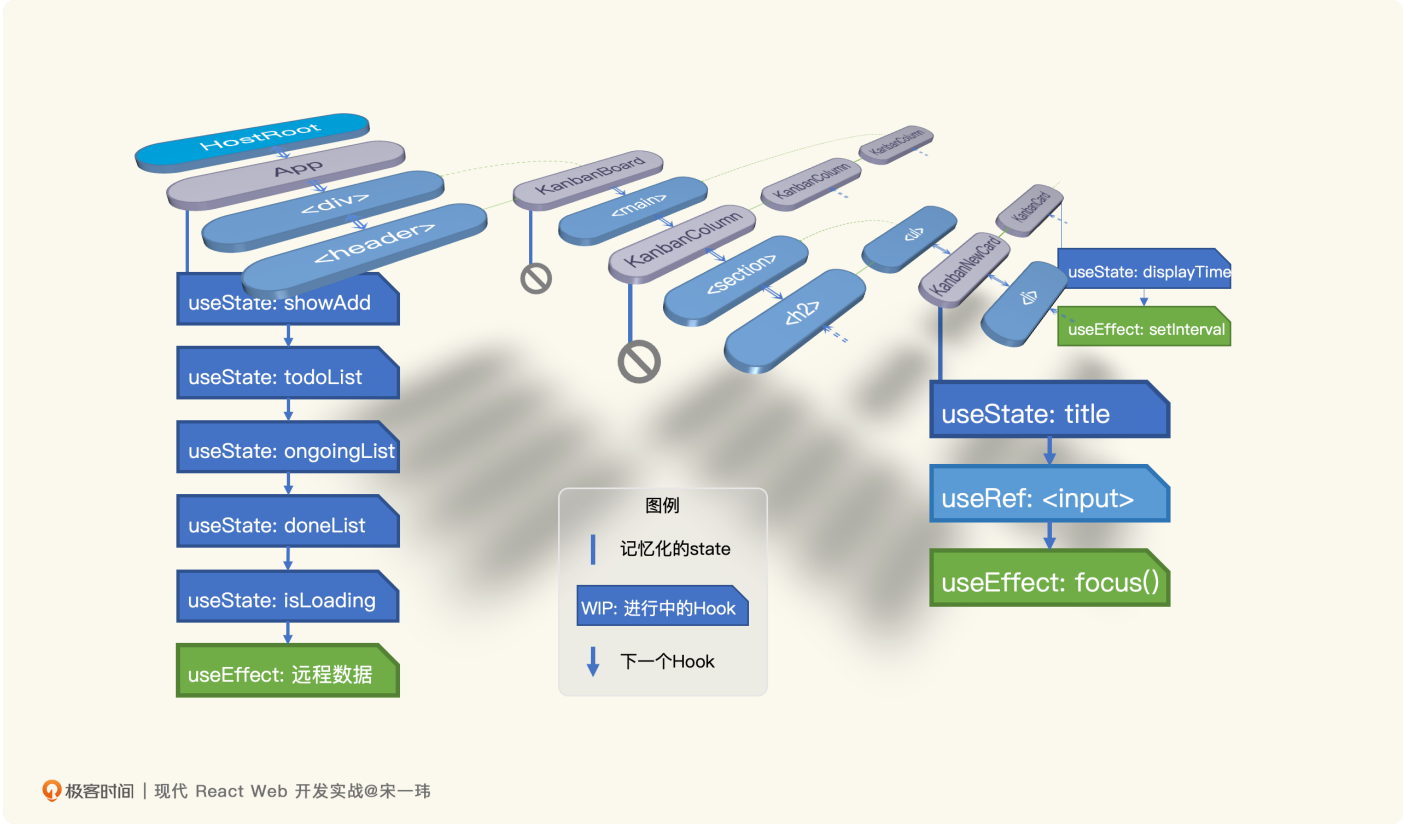
虽然借鉴了很多函数式编程的特性，Hooks 本身也都是 JavaScript 函数，但 Hooks 终归是一套 **React 特有的 API**，使用 Hooks 并不等于函数式编程，也不能把函数式编程的各种最佳实践完整地搬到 Hooks 身上。

比起传统的函数式编程，有两条限制，需要你在使用 Hooks 时务必注意。

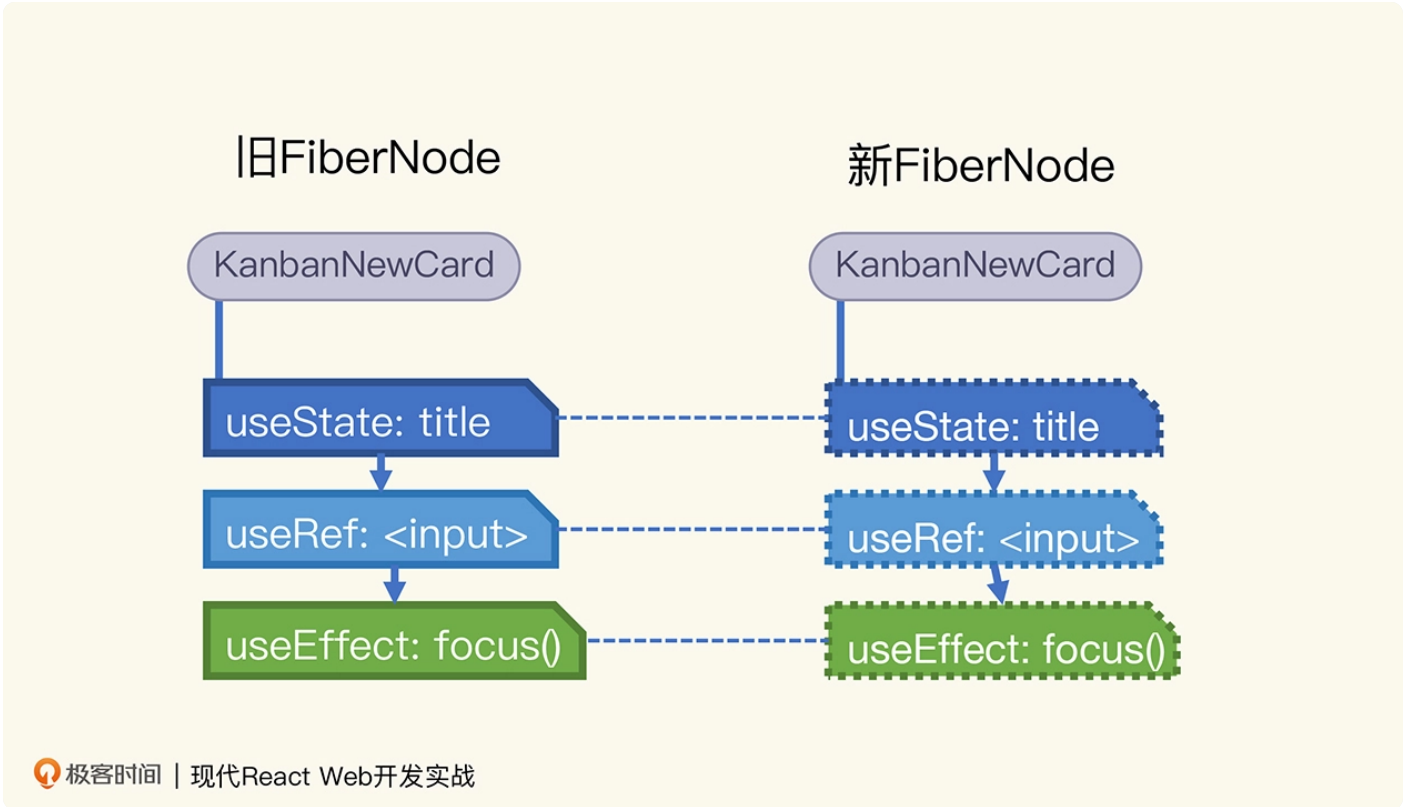
第一，只能在 React 的函数组件中调用 Hooks。这也包括了在自定义的 Hook 中调用其他 Hooks 这样间接的调用方式，目的是保证 Hooks 能“勾”到 React 的虚拟 DOM 中去，脱离 React 环境的 Hooks 是无法起作用的。

第二，只能在组件函数的最顶层调用 Hooks。无论组件函数运行多少遍，都要保证每个 Hook 的执行顺序，这样 React 才能识别每个 Hook，保持它们的状态。当然，这就要求开发者不能在循环、条件分支中或者任何 return 语句之后调用 Hooks。

其实从 Fiber 协调引擎的底层实现来看，也不难理解上面两个限制。函数组件首次渲染时会创建对应的 FiberNode，这个 FiberNode 上会保存一个记录 Hooks 状态的单向链表，链表的长度与执行组件函数时调用的 Hooks 个数相同：



当函数组件再次渲染时，每个 Hook 都会被再次调用，而这些 Hooks 会按顺序，去这个单向链表中一一认领自己上一次的状态，并根据需要沿用或者更新自己在链表中的状态：



这也说明了为什么一个 `useState` 每次渲染返回的 state 更新函数都是同一个函数（引用），`useEffect` 也是通过这个 Hook 状态来比对依赖值数组在两次渲染之间是否有更改，进而决

定是否再次执行副作用。

再回来看这两个限制。如果不在 React 的函数组件中调用 Hooks，React 就不会创建记录 Hooks 状态的单向链表；如果在循环、条件分支等不稳定的代码位置调用 Hooks，就有可能导致再次渲染时，执行 Hooks 的数量、种类和参数与上次的单向链表不一致，Hooks 内部的逻辑就乱掉了。

在满足这两个限制的前提下，Hooks 与其他 JS 函数无异，函数的组合、复用是非常灵活的。React 鼓励开发者自定义 Hooks，在这节课我们暂不展开，后面会专门有一节课讲 React 代码复用。

用类组件还是函数组件加 Hooks？

截止目前，我看到大部分 React 教程都是先学习类组件，再学习 Hooks，猜测主要有两方面的原因。一是类组件与以往的传统前端框架更相似；二是类组件的现存案例和文档更多，这两点都导致了教程制作的惯性。

但在这节课你会发现，我刻意地引导你**优先学习函数组件加 Hooks。**我猜想，你是不是有点担心自己因为少学了一部分内容而落后于其他人？我觉得你不用担心，原因有下面两点。

一是，React 官方文档已经推荐开发者在开发新应用时首选函数组件加 Hooks。从 2019 年初到 2022 年已经三年了，React 也已经从 v16.8.0 更新到 v18.2.0 了，实际情况又怎样了呢？

上数据，在 Github 上搜索包含 React "useState"的代码，返回的 JS、TS、TSX 文件总数为 17.9M，而 React "extends React.Component"加上 React "extends Component"两次搜索的结果为 10M。

当然这种统计并不严谨，但已经可以证明 Hooks 的受欢迎程度，可以认为**函数组件已经代替类组件成为主流组件形式**，学习好函数组件加 Hooks，基本就可以应对主流 React 应用开发了。

二是先入为主。类组件和函数组件代表了两种不同的编程方式，前者更面向对象，后者更接近函数式编程。先学习类组件，会让开发者倾向于用面向对象的思路理解 React 的各种概念，而

实际上，在 React v18.2.0 版本的源码中，面向对象的比重已经越来越低了。这时再去学习类组件以外的概念，开发者就不得不先修正之前的理解。

我有不少同事完整经历了从类组件到函数组件加 Hooks 的转换，我观察到，当他们在已经掌握类组件的基础上再学习 Hooks 时，会**不自觉地从前者中寻找参照物**，一旦发现在特定的功能上找不到参照物时，多少会走些弯路。

比如他们会用 `useEffect` 理解成类组件里的 `componentDidMount` 和 `componentWillUnmount`，但他们意外地发现 `useEffect` 在每次组件更新时都会被执行。学完前面内容的你，相信已经知道其中的原因了。

反过来优先学习函数组件加 Hooks，可以让开发者更直接地接触 React 元素、props、state、协调、渲染这些核心概念，提升学习效率和效果。

当然凡事也有例外。第 8 节课我们在介绍组件生命周期的错误处理阶段时，提到截止到 React v18.2.0，只有类组件才能成为错误边界，函数组件是不行的。像这样类组件独有的少数功能，我们在第三模块遇到时会详细介绍。

小结

这节课我们学习了 React 基本 Hooks 之一的副作用 Hook `useEffect`，同时顺带对比了副作用 Hooks 和类组件的生命周期函数。接着介绍了主要用于性能优化的 Hooks `useMemo` 和 `useCallback`。然后也强调了无论是哪种 Hooks，都只能在 React 函数组件中、函数的最顶层调用的限制。

在这节课末尾，也说明了为什么引导你优先学习函数组件加 Hooks，而不是传统的类组件。

下节课我们将学习交互性更强的内容，即 React 的事件处理。

思考题

1. 这节课我们学习了 `useState` 和 `useEffect`，在讲解 `useEffect` 时举了级联菜单的例子，不过这个例子限于篇幅没有写完，我想请你补全它。

需求很典型：第一级是省份的下拉列表，第二级是城市的下拉列表，当选中一个省份时城市的列表会相应改变。虽然我们下节课才会系统学习 onChange 这样的事件处理，但参考 oh-my-kanban 里的样例代码，我相信你很快就能写出来。

2. 我们在第 8 节课和这节课都提到了内存泄漏，你能列举出一些前端领域会导致内存泄漏的例子吗？

欢迎把你的思考和想法分享在留言区，我们下节课再见！

上一篇 09丨React Hooks（上）：为什么说在React中函数组件和Hooks是绝配？

下一篇 11丨事件处理：React合成事件是什么？为什么不用原生DOM事件？

精选留言 (4)

学习前端-react

2022-09-14 来自江苏

请问：use Effect 的执行是可以拿到真实dom的，那为啥在图中提交阶段却是在真实dom之前？

共 1 条评论 >　1

满眼星　辰

2022-09-16 来自宁夏

图例中，useLayoutEffect是同步更新dom，应该在useEffect之前执行，不是吗

都市夜归人

2022-09-16 来自北京

其实这两个 Hooks 与 useEffect 并不沾亲带故。且不说它们的用途完全不同，单从回调函数的执行阶段来看，前者是在渲染阶段执行，而后者是在提交阶段。

这句话与上面的生命周期图不太一致，求解惑

共 1 条评论 >

都市夜归人

2022-09-15 来自北京

```
const KanbanCard = ({ title, status }) => {
  const [displayTime, setDisplayTime] = useState(status);
  useEffect(() => {
    const updateDisplayTime = () => {
      const timePassed = new Date() - new Date(status);
      let relativeTime = '刚刚';
      // ...省略
      setDisplayTime(relativeTime);
    };
    const intervalId = setInterval(updateDisplayTime, UPDATE_INTERVAL);
    updateDisplayTime();

    return function cleanup() {
      clearInterval(intervalId);
    };
  }, [status]);
```

可以看到，useEffect 接收了副作用回调函数和依赖值数组两个参数，其中副作用回调函数的返回值也是一个函数，这个返回的函数叫做清除函数。组件在下一次提交阶段执行同一个副作用回调函数之前，或者是组件即将被卸载之前，会调用这个清除函数。

没有看懂，上面的哪有两个参数啊？

共 2 条评论 >