08 | 深入 React-Hooks 工作机制：“原则”的背后，是“原理”

2020/11/04 修言

React 团队面向开发者给出了两条 React-Hooks 的使用原则，原则的内容如下：

1. 只在 React 函数中调用 Hook；
2. 不要在循环、条件或嵌套函数中调用 Hook。

原则 1 无须多言，React-Hooks 本身就是 React 组件的“钩子”，在普通函数里引入意义不大。我相信更多的人在原则 2 上栽过跟头，或者说至今仍然对它半信半疑。其实，原则 2 中强调的所有“**不要**”，都是在指向同一个目的，那就是**要确保 Hooks 在每次渲染时都保持同样的执行顺序**。

为什么顺序如此重要？这就要从 Hooks 的实现机制说起了。这里我就以 useState 为例，带你从现象入手，深度探索一番 React-Hooks 的工作原理。

注：本讲 Demo 基于 React 16.8.x 版本进行演示。

从现象看问题：若不保证 Hooks 执行顺序，会带来什么麻烦？

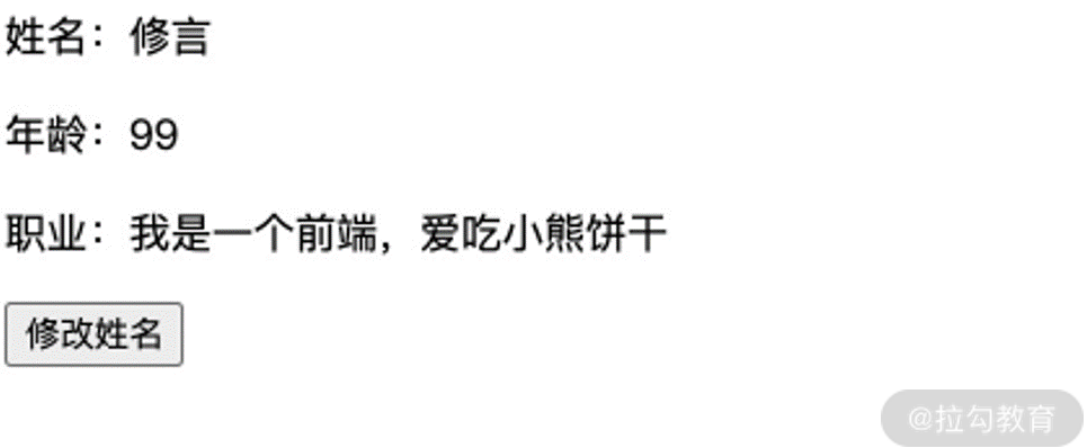
先来看一个小 Demo：

复制代码

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

function PersonalInfoComponent() {
  // 集中定义变量
  let name, age, career, setName, setCareer;

  // 获取姓名状态
  [name, setName] = useState("修言");

  // 获取年龄状态
  [age] = useState("99");

  // 获取职业状态
  [career, setCareer] = useState("我是一个前端，爱吃小熊饼干");

  // 输出职业信息
  console.log("career", career);
```

```
18.
19.   // 编写 UI 逻辑
20.   return (
21.     <div className="personalInfo">
22.       <p>姓名：{name}</p>
23.       <p>年龄：{age}</p>
24.       <p>职业：{career}</p>
25.       <button
26.         onClick={() => {
27.           setName("秀妍");
28.         }}
29.       >
30.         修改姓名
31.       </button>
32.     </div>
33.   );
34. }
35.
36. export default PersonalInfoComponent;
```

这个 PersonalInfoComponent 组件渲染出来的界面长这样：



PersonalInfoComponent 用于对个人信息进行展示，这里展示的内容包括姓名、年龄、职业。出于测试效果需要，PersonalInfoComponent 还允许你点击“修改姓名”按钮修改姓名信息。点击一次后，“修言”会被修改为“秀妍”，如下图所示：

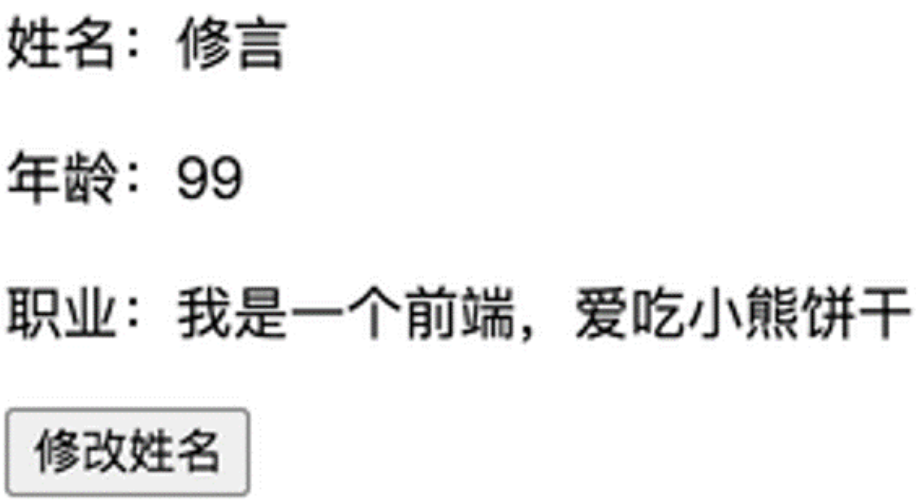
到目前为止，组件的行为都是符合我们的预期的，一切看上去都是那么的和谐。但倘若我对代码做一丝小小的改变，把一部分的 useState 操作放进 if 语句里，事情就会变得大不一样。改动后的代码如下：

复制代码

```
import React, { useState } from "react";
// isMounted 用于记录是否已挂载（是否是首次渲染）
let isMounted = false;
function PersonalInfoComponent() {
  // 定义变量的逻辑不变
  let name, age, career, setName, setCareer;

  // 这里追加对 isMounted 的输出，这是一个 debug 性质的操作
  console.log("isMounted is", isMounted);
  // 这里追加 if 逻辑：只有在首次渲染（组件还未挂载）时，才获取 name、age 两个状态
  if (!isMounted) {
    // eslint-disable-next-line
    [name, setName] = useState("修言");
    // eslint-disable-next-line
    [age] = useState("99");

    // if 内部的逻辑执行一次后，就将 isMounted 置为 true（说明已挂载，后续都不再是首次渲染
    isMounted = true;
  }

  // 对职业信息的获取逻辑不变
  [career, setCareer] = useState("我是一个前端，爱吃小熊饼干");
  // 这里追加对 career 的输出，这也是一个 debug 性质的操作
  console.log("career", career);
  // UI 逻辑的改动在于，name和age成了可选的展示项，若值为空，则不展示
  return (
    <div className="personalInfo">
      {name ? <p>姓名：{name}</p> : null}
      {age ? <p>年龄：{age}</p> : null}
      <p>职业：{career}</p>
      <button
        onClick={() => {
          setName("秀妍");
        }}
```

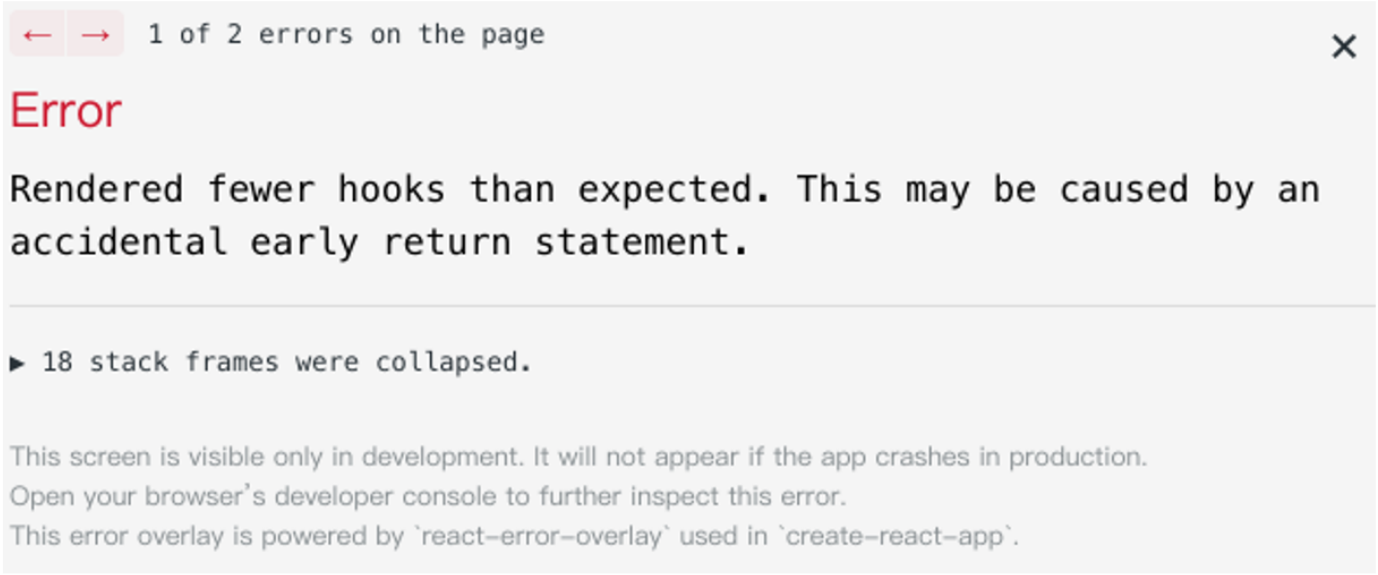
```
        >
          修改姓名
        </button>
      </div>
    );
}
export default PersonalInfoComponent;
```

修改后的组件在初始渲染的时候，界面与上个版本无异：



注意，你在自己电脑上模仿这段代码的时候，千万不要漏掉 if 语句里面`// eslint-disable-next-line`这个注释——因为目前大部分的 React 项目都在内部预置了对 React-Hooks-Rule（React-Hooks 使用规则）的强校验，而示例代码中把 Hooks 放进 if 语句的操作作为一种不合规操作，会被直接识别为 Error 级别的错误，进而导致程序报错。这里我们只有将相关代码的 eslint 校验给禁用掉，才能够避免校验性质的报错，从而更直观地看到错误的效果到底是什么样的，进而理解错误的原因。

修改后的组件在初始挂载的时候，实际执行的逻辑内容和上个版本是没有区别的，都涉及对 name、age、career 三个状态的获取和渲染。理论上来说，**变化应该发生在我单击“修改姓名”之后触发的二次渲染里**：二次渲染时，isMounted 已经被置为 true，if 内部的逻辑会被直接跳过。此时按照代码注释中给出的设计意图，这里我希望在二次渲染时，只获取并展示 career 这一个状态。那么事情是否会如我所愿呢？我们一起来看看单击“修改姓名”按钮后会发生什么：



组件不仅没有像预期中一样发生界面变化，甚至直接报错了。报错信息提醒我们，这是因为“**组件渲染的 Hooks 比期望中更少**”。

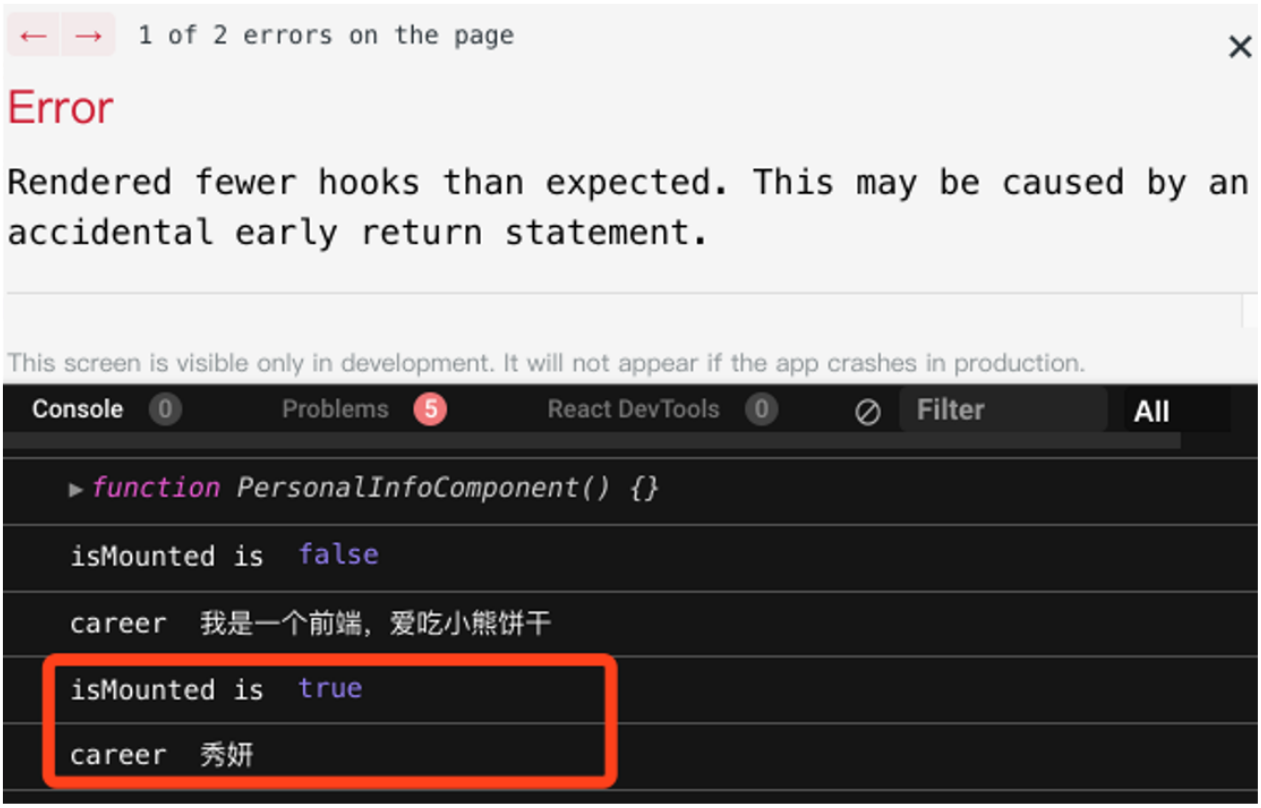
确实，按照现有的逻辑，初始渲染调用了三次 useState，而二次渲染时只会调用一次。但仅仅因为这个，就要报错吗?

按道理来说，二次渲染的时候，只要我获取到的 career 值没有问题，那么渲染就应该是没有问题的（因为二次渲染实际只会渲染 career 这一个状态），React 就没有理由阻止我的渲染动作。啊这……难道是 career 出问题了吗？还好我们预先留了一手 Debug 逻辑，每次渲染的时候都会尝试去输出一次 isMounted 和 career 这两个变量的值。现在我们就赶紧来看看，这两个变量到底是什么情况。

首先我将界面重置回初次挂载的状态，观察控制台的输出，如下图所示：

这里我把关键的 isMounted 和 career 两个变量用红色框框圈了出来：isMounted 值为 false，说明是初次渲染；career 值为“我是一个前端，爱吃小熊饼干”，这也是没有问题的。

接下来单击“修改姓名”按钮后，我们再来看一眼两个变量的内容，如下图所示：



二次渲染时，isMounted 为 true，这个没毛病。但是 career 竟然被修改为了“秀妍”，这也太诡异了？代码里面可不是这么写的。赶紧回头确认一下按钮单击事件的回调内容，代码如下所示：

```
<button
  onClick={() => {
    setName("秀妍");
  }}
  >
  修改姓名
</button>
```

确实，代码是没错的，我们调用的是 setName，那么它修改的状态也应该是 name，而不是 career。

那为什么最后发生变化的竟然是 career 呢？年轻人，不如我们一起来看一看 Hooks 的实现机制吧！

从源码调用流程看原理：Hooks 的正常运作，在底层依赖于顺序链表

这里强调“源码流程”而非“源码”，主要有两方面的考虑：

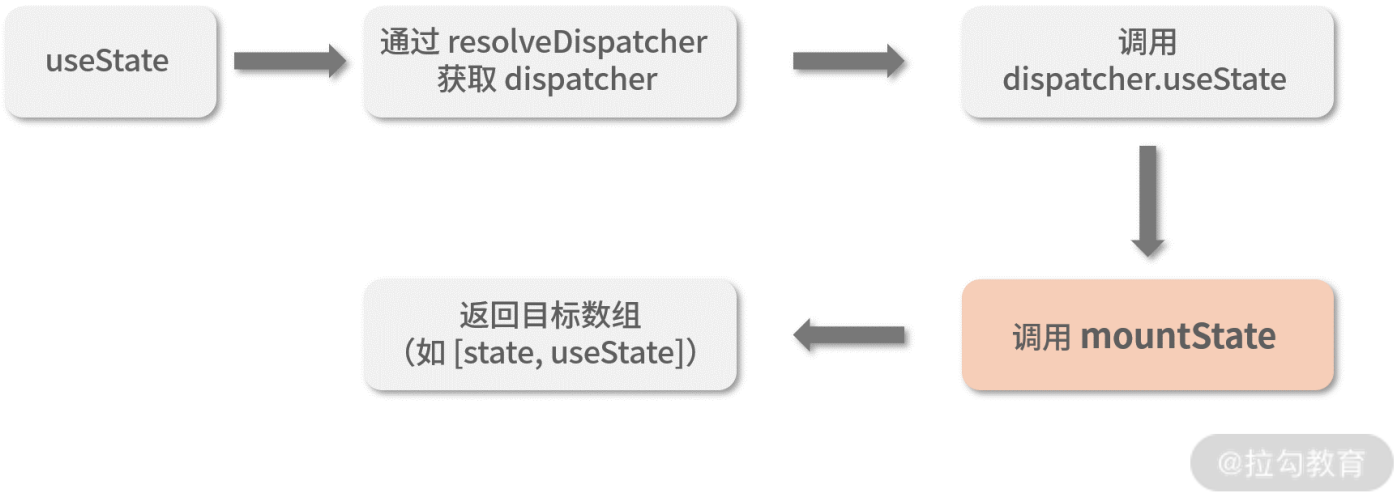
1. React-Hooks 在源码层面和 Fiber 关联十分密切，我们目前仍然处于基础夯实阶段，对 Fiber 机制相关的底层实现暂时没有讨论，盲目啃源码在这个阶段来说没有意义；

2. 原理 !== 源码，阅读源码只是掌握原理的一种手段，在某些场景下，阅读源码确实能够迅速帮我们定位到问题的本质（比如 React.createElement 的源码就可以快速帮我们理解 JSX 转换出来的到底是什么东西）；而 React-Hooks 的源码链路相对来说比较长，涉及的关键函数 renderWithHooks 中“脏逻辑”也比较多，整体来说，学习成本比较高，学习效果也难以保证。

综上所述，这里我不会精细地贴出每一行具体的源码，而是针对关键方法做重点分析。同时我也**不建议你在对 Fiber 底层实现没有认知的前提下去和 Hooks 源码死磕**。对于搞清楚“Hooks 的执行顺序为什么必须一样”这个问题来说，重要的并不是去细抠每一行代码到底都做了什么，而是要搞清楚整个**调用链路**是什么样的。如果我们能够理解 Hooks 在每个关键环节都做了哪些事情，同时也能理解这些关键环节是如何对最终的渲染结果产生影响的，那么理解 Hooks 的工作机制对于你来说就不在话下了。

以 useState 为例，分析 React-Hooks 的调用链路

首先要说明的是，React-Hooks 的调用链路在首次渲染和更新阶段是不同的，这里我将两个阶段的链路各总结进了两张大图里，我们依次来看。首先是首次渲染的过程，请看下图：



在这个流程中，useState 触发的一系列操作最后会落到 mountState 里面去，所以我们重点需要关注的就是 mountState 做了什么事情。以下我为你提取了 mountState 的源码：

复制代码

```
// 进入 mounState 逻辑
function mountState(initialState) {

  // 将新的 hook 对象追加进链表尾部
  var hook = mountWorkInProgressHook();

  // initialState 可以是一个回调，若是回调，则取回调执行后的值
  if (typeof initialState === 'function') {
    // $FlowFixMe: Flow doesn't like mixed types
    initialState = initialState();
```

```
  }

  // 创建当前 hook 对象的更新队列，这一步主要是为了能够依序保留 dispatch
  const queue = hook.queue = {
    last: null,
    dispatch: null,
    lastRenderedReducer: basicStateReducer,
    lastRenderedState: (initialState: any),
  };

  // 将 initialState 作为一个“记忆值”存下来
  hook.memoizedState = hook.baseState = initialState;

  // dispatch 是由上下文中一个叫 dispatchAction 的方法创建的，这里不必纠结这个方法具体做了
  var dispatch = queue.dispatch = dispatchAction.bind(null, currentlyRenderingF
  // 返回目标数组，dispatch 其实就是示例中常常见到的 setXXX 这个函数，想不到吧？哈哈
  return [hook.memoizedState, dispatch];
}
```

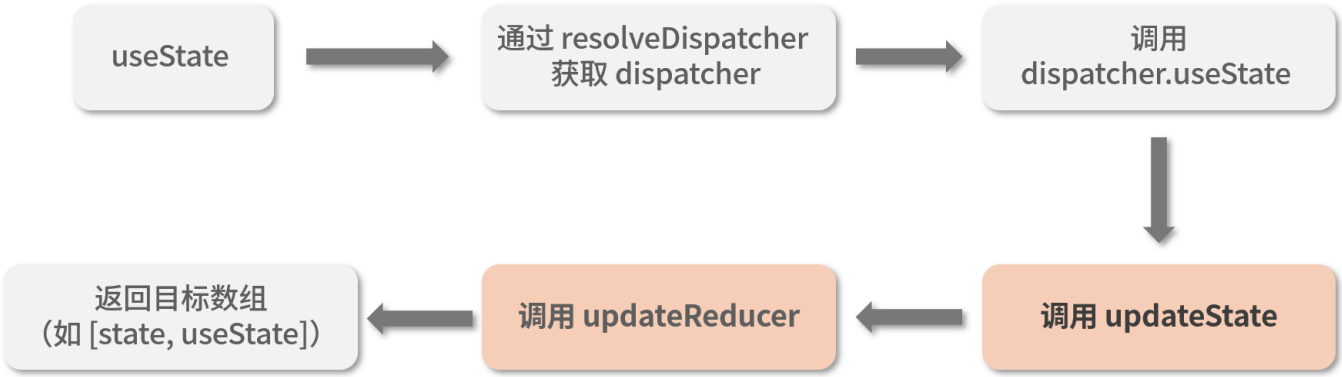
从这段源码中我们可以看出，**mounState 的主要工作是初始化 Hooks**。在整段源码中，最需要关注的是 mountWorkInProgressHook 方法，它为我们道出了 Hooks 背后的数据结构组织形式。以下是 mountWorkInProgressHook 方法的源码：

复制代码

```
function mountWorkInProgressHook() {
  // 注意，单个 hook 是以对象的形式存在的
  var hook = {
    memoizedState: null,
    baseState: null,
    baseQueue: null,
    queue: null,
    next: null
  };
  if (workInProgressHook === null) {
    // 这行代码每个 React 版本不太一样，但做的都是同一件事：将 hook 作为链表的头节点处理
    firstWorkInProgressHook = workInProgressHook = hook;
  } else {
    // 若链表不为空，则将 hook 追加到链表尾部
    workInProgressHook = workInProgressHook.next = hook;
  }
  // 返回当前的 hook
  return workInProgressHook;
}
```

到这里可以看出，**hook 相关的所有信息收敛在一个 hook 对象里，而 hook 对象之间以单向链表的形式相互串联**。

接下来我们再看更新过程的大图：



根据图中高亮部分的提示不难看出，首次渲染和更新渲染的区别，在于调用的是 mountState，还是 updateState。mountState 做了什么，你已经非常清楚了；而 updateState 之后的操作链路，虽然涉及的代码有很多，但其实做的事情很容易理解：**按顺序去遍历之前构建好的链表，取出对应的数据信息进行渲染**。

我们把 mountState 和 updateState 做的事情放在一起来看：mountState（首次渲染）构建链表并渲染；updateState 依次遍历链表并渲染。

看到这里，你是不是已经大概知道怎么回事儿了？没错，**hooks 的渲染是通过“依次遍历”来定位每个 hooks 内容的。如果前后两次读到的链表在顺序上出现差异，那么渲染的结果自然是不可控的。**

这个现象有点像我们构建了一个长度确定的数组，数组中的每个坑位都对应着一块确切的信息，后续每次从数组里取值的时候，只能够通过索引（也就是位置）来定位数据。也正因为如此，在许多文章里，都会直截了当地下这样的定义：Hooks 的本质就是数组。但读完这一课时的内容你就会知道，**Hooks 的本质其实是链表**。

接下来我们把这个已知的结论还原到 PersonalInfoComponent 里去，看看实际项目中，变量到底是怎么发生变化的。

站在底层视角，重现 PersonalInfoComponent 组件的执行过程

我们先来复习一下修改过后的 PersonalInfoComponent 组件代码：

复制代码

```
import React, { useState } from "react";
// isMounted 用于记录是否已挂载（是否是首次渲染）
let isMounted = false;
```

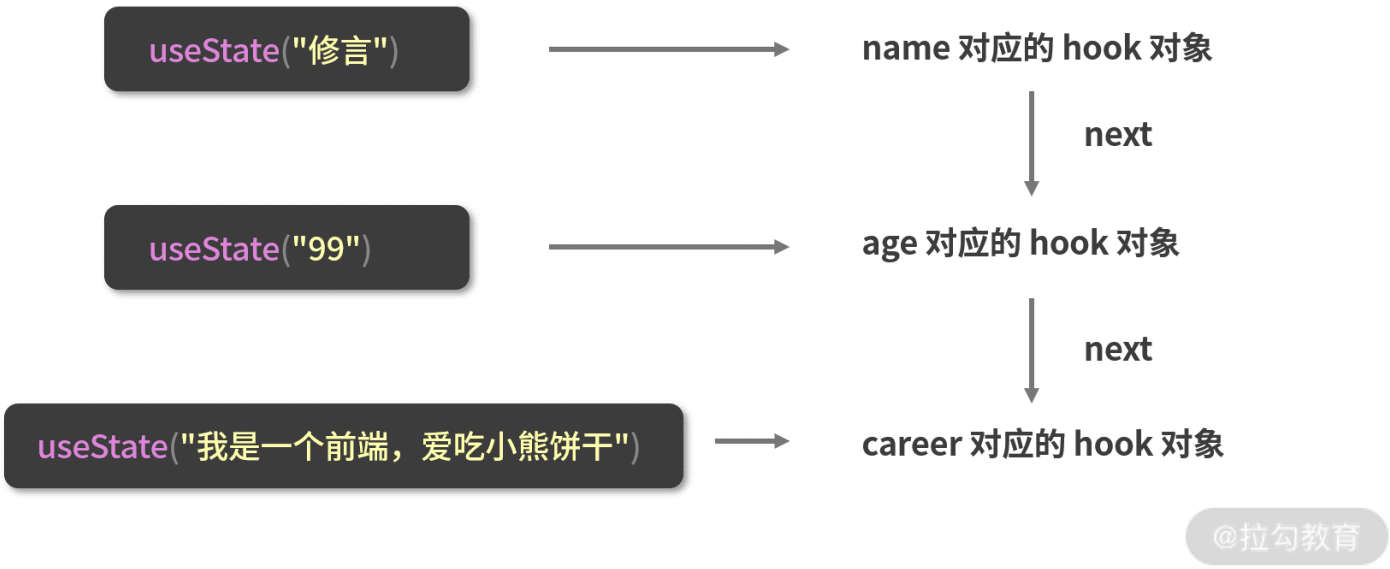
```
function PersonalInfoComponent() {
  // 定义变量的逻辑不变
  let name, age, career, setName, setCareer;

  // 这里追加对 isMounted 的输出，这是一个 debug 性质的操作
  console.log("isMounted is", isMounted);
  // 这里追加 if 逻辑：只有在首次渲染（组件还未挂载）时，才获取 name、age 两个状态
  if (!isMounted) {
    // eslint-disable-next-line
    [name, setName] = useState("修言");
    // eslint-disable-next-line
    [age] = useState("99");

    // if 内部的逻辑执行一次后，就将 isMounted 置为 true（说明已挂载，后续都不再是首次渲染
    isMounted = true;
  }

  // 对职业信息的获取逻辑不变
  [career, setCareer] = useState("我是一个前端，爱吃小熊饼干");
  // 这里追加对 career 的输出，这也是一个 debug 性质的操作
  console.log("career", career);
  // UI 逻辑的改动在于，name 和 age 成了可选的展示项，若值为空，则不展示
  return (
    <div className="personalInfo">
      {name ? <p>姓名：{name}</p> : null}
      {age ? <p>年龄：{age}</p> : null}
      <p>职业：{career}</p>
      <button
        onClick={() => {
          setName("秀妍");
        }}
      >
        修改姓名
      </button>
    </div>
  );
}
export default PersonalInfoComponent;
```

从代码里面，我们可以提取出来的 useState 调用有三个：

复制代码

```
[name, setName] = useState("修言");
[age] = useState("99");
[career, setCareer] = useState("我是一个前端，爱吃小熊饼干");
```

这三个调用在首次渲染的时候都会发生，伴随而来的链表结构如图所示：



当首次渲染结束，进行二次渲染的时候，实际发生的 useState 调用只有一个：

```
1. useState("我是一个前端，爱吃小熊饼干")
```

复制代码

而此时的链表情况如下图所示：

我们再复习一遍更新（二次渲染）的时候会发生什么事情：updateState 会依次遍历链表、读取数据并渲染。注意这个过程就像从数组中依次取值一样，是完全按照顺序（或者说索引）来的。因此 React 不会看你命名的变量名是 career 还是别的什么，它只认你这一次 useState 调用，于是它难免会认为：**喔，原来你想要的是第一个位置的 hook 啊**。

然后就会有下面这样的效果：

如此一来，career 就自然而然地取到了链表头节点 hook 对象中的“秀妍”这个值。

总结

三个课时学完了，到这里，我们对 React-Hooks 的学习，才终于算是告一段落。

在过去的三个课时里，我们摸排了“动机”，认知了“工作模式”，最后更是结合源码、深挖了一把 React-Hooks 的底层原理。我们所做的这所有的努力，都是为了能够真正吃透 React-Hooks，不仅要确保实践中不出错，还要做到面试时有底气。

接下来，我们就将进入整个专栏真正的“深水区”，逐步切入“虚拟 DOM → Diff 算法 → Fiber 架构”这个知识链路里来。在后续的学习中，我们将延续并且强化这种“刨根问底”的风格，紧贴源码、原理和面试题来向 React 最为核心的部分发起挑战。真正的战斗，才刚刚开始，大家加油~