15 | ReactDOM.render 是如何串联渲染链路的？（下）

2020/11/30 修言

47.84M 00:00/18:21 看视频

在上一讲我们从 beginWork 切入，摸索出了 Fiber 节点的创建链路与 Fiber 树的构建链路。本讲我们将以 completeWork 为线索，去寻觅 Fiber 树和 DOM 树之间的关联，将整个 render 阶段讲透。在此基础上，结合 commit 阶段工作流，你将会对 ReactDOM.render 所触发的渲染链路有一个完整、通透的理解。

本讲的实验 Demo 与前两讲保持一致，代码如下：

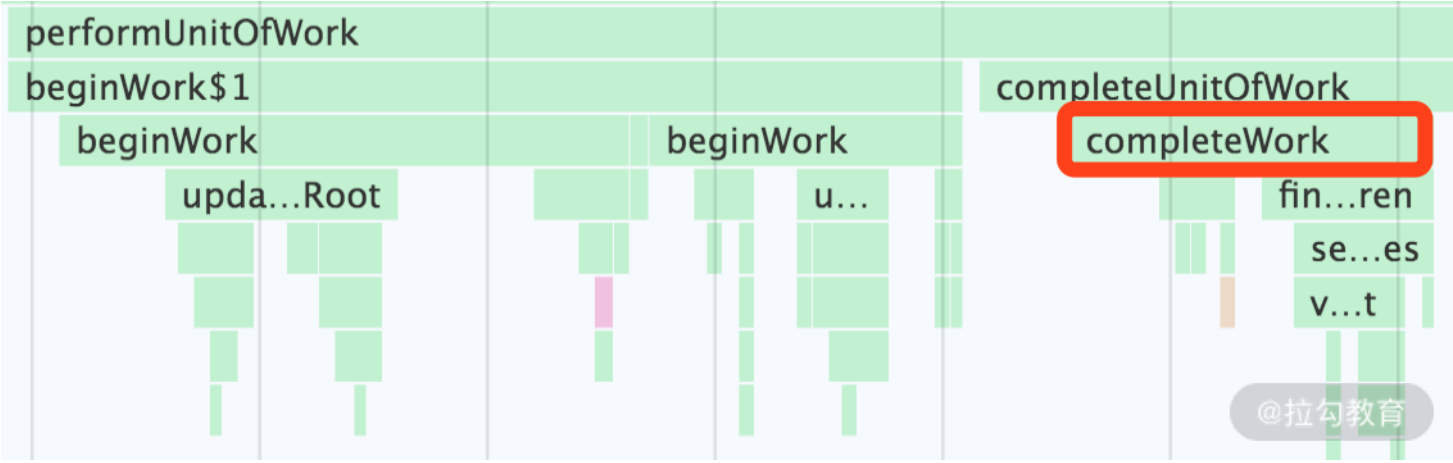
复制代码

```
import React from "react";
import ReactDOM from "react-dom";
function App() {
  return (
    <div className="App">
      <div className="container">
        <h1>我是标题</h1>
        <p>我是第一段话</p>
        <p>我是第二段话</p>
      </div>
    </div>
  );
}
const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

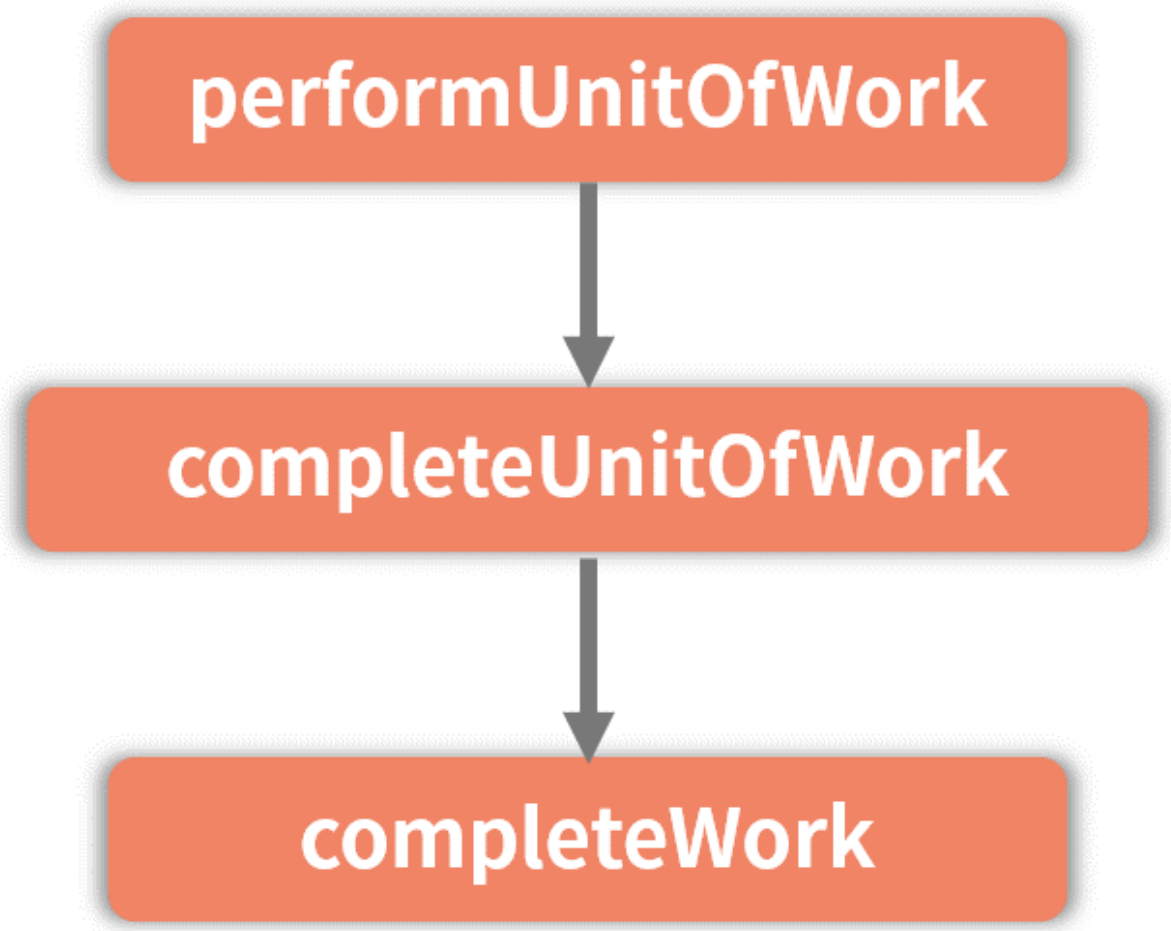
completeWork——将 Fiber 节点映射为 DOM 节点

completeWork 的调用时机

首先，我们先在调用栈中定位一下 completeWork。Demo 所对应的调用栈中，第一个 completeWork 出现在下图红框选中的位置：



从图上我们需要把握住的一个信息是，从 performUnitOfWork 到 completeWork，中间会经过一个这样的调用链路：



其中 completeUnitOfWork 的工作也非常关键，但眼下我们先拿 completeWork 开刀，你可以暂时将 completeUnitOfWork 简单理解为一个用于发起 completeWork 调用的“工具人”。completeUnitOfWork 是在 performUnitOfWork 中被调用的，那么 performUnitOfWork 是如何把握其调用时机的呢？我们直接来看相关源码（解析在注释里）：

复制代码

```
function performUnitOfWork(unitOfWork) {
  ......
  // 获取入参节点对应的 current 节点
  var current = unitOfWork.alternate;

  var next;
  if (xxx) {
    ...
    // 创建当前节点的子节点
    next = beginWork$1(current, unitOfWork, subtreeRenderLanes);
    ...
  } else {
    // 创建当前节点的子节点
    next = beginWork$1(current, unitOfWork, subtreeRenderLanes);
  }
  ......
  if (next === null) {
    // 调用 completeUnitOfWork
    completeUnitOfWork(unitOfWork);
  } else {
    // 将当前节点更新为新创建出的 Fiber 节点
    workInProgress = next;
  }
  ......
}
```

这段源码中你需要提取出的信息是：performUnitOfWork 每次会尝试调用 beginWork 来创建当前节点的子节点，若创建出的子节点为空（也就意味着当前节点不存在子 Fiber 节点），则说明当前节点是一个叶子节点。**按照深度优先遍历的原则，当遍历到叶子节点时，“递”阶段就结束了，随之而来的是“归”的过程**。因此这种情况下，就会调用 completeUnitOfWork，执行当前节点对应的 completeWork 逻辑。

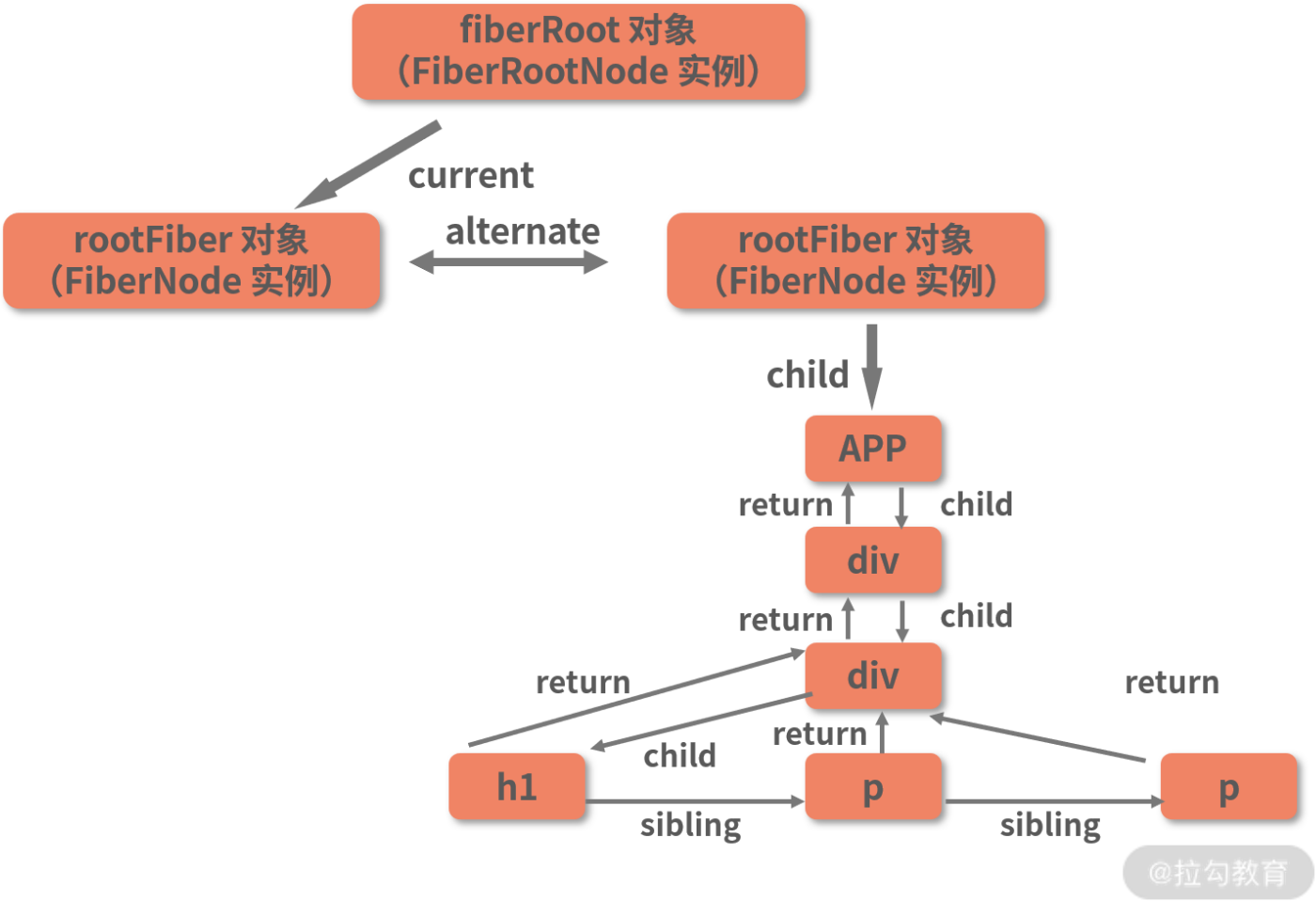
接下来我们在 Demo 代码的 completeWork 处打上断点，看看第一个走到 completeWork 的节点是哪个，结果如下图所示：

```
20168 function completeWork(current, workInProgress, renderLanes) {  curr
20169   var newProps = workInProgress.pendingProps;
20170
20171   switch (workIr
20172     case Indeter
20173     case LazyCom
20174     case SimpleM
20175     case Functic
20176     case Forwarc
20177     case Fragmer
20178     case Mode:
20179     case Profile
20180     case Context
20181     case MemoCom
20182       return nul
20183
20184     case ClassCo
20185       {
20186         var Comp
20187
```

FiberNode

```
actualDuration: 0.14999997802078724
actualStartTime: 5855.445000110194
alternate: null
child: null
childLanes: 0
dependencies: null
elementType: "h1"
firstEffect: null
flags: 0
index: 0
key: null
lanes: 0
lastEffect: null
```


显然，第一个进入 completeWork 的节点是 h1，这也符合我们上一讲所构建出来的 Fiber 树中的节点关系，如下图所示：



由图可知，按照深度优先遍历的原则，h1 确实将是第一个被遍历到的叶子节点。接下来我们就以 h1 为例，一起看看 completeWork 都围绕它做了哪些事情。

completeWork 的工作原理

这里仍然为你提取一下 completeWork 的源码结构和主体逻辑，代码如下（解析在注释里）：

复制代码

```
function completeWork(current, workInProgress, renderLanes) {
  // 取出 Fiber 节点的属性值，存储在 newProps 里
  var newProps = workInProgress.pendingProps;

  // 根据 workInProgress 节点的 tag 属性的不同，决定要进入哪段逻辑
  switch (workInProgress.tag) {
    case ......:
      return null;
    case ClassComponent:
      {
        .....
      }
    case HostRoot:
      {
        ......
      }
    // h1 节点的类型属于 HostComponent，因此这里为你讲解的是这段逻辑
    case HostComponent:
      {
        popHostContext(workInProgress);
        var rootContainerInstance = getRootHostContainer();
        var type = workInProgress.type;
        // 判断 current 节点是否存在，因为目前是挂载阶段，因此 current 节点是不存在的
        if (current !== null && workInProgress.stateNode != null) {
          updateHostComponent$1(current, workInProgress, type, newProps, rootCor
          if (current.ref !== workInProgress.ref) {
            markRef$1(workInProgress);
          }
        } else {
          // 这里首先是针对异常情况进行 return 处理
          if (!newProps) {
            if (!(workInProgress.stateNode !== null)) {
              {
                throw Error("We must have new props for new mounts. This error i
              }
            }

            return null;
          }

          // 接下来就为 DOM 节点的创建做准备了
          var currentHostContext = getHostContext();
          // _wasHydrated 是一个与服务端渲染有关的值，这里不用关注
          var _wasHydrated = popHydrationState(workInProgress);

          // 判断是否是服务端渲染
          if (_wasHydrated) {
            // 这里不用关注，请你关注 else 里面的逻辑
```

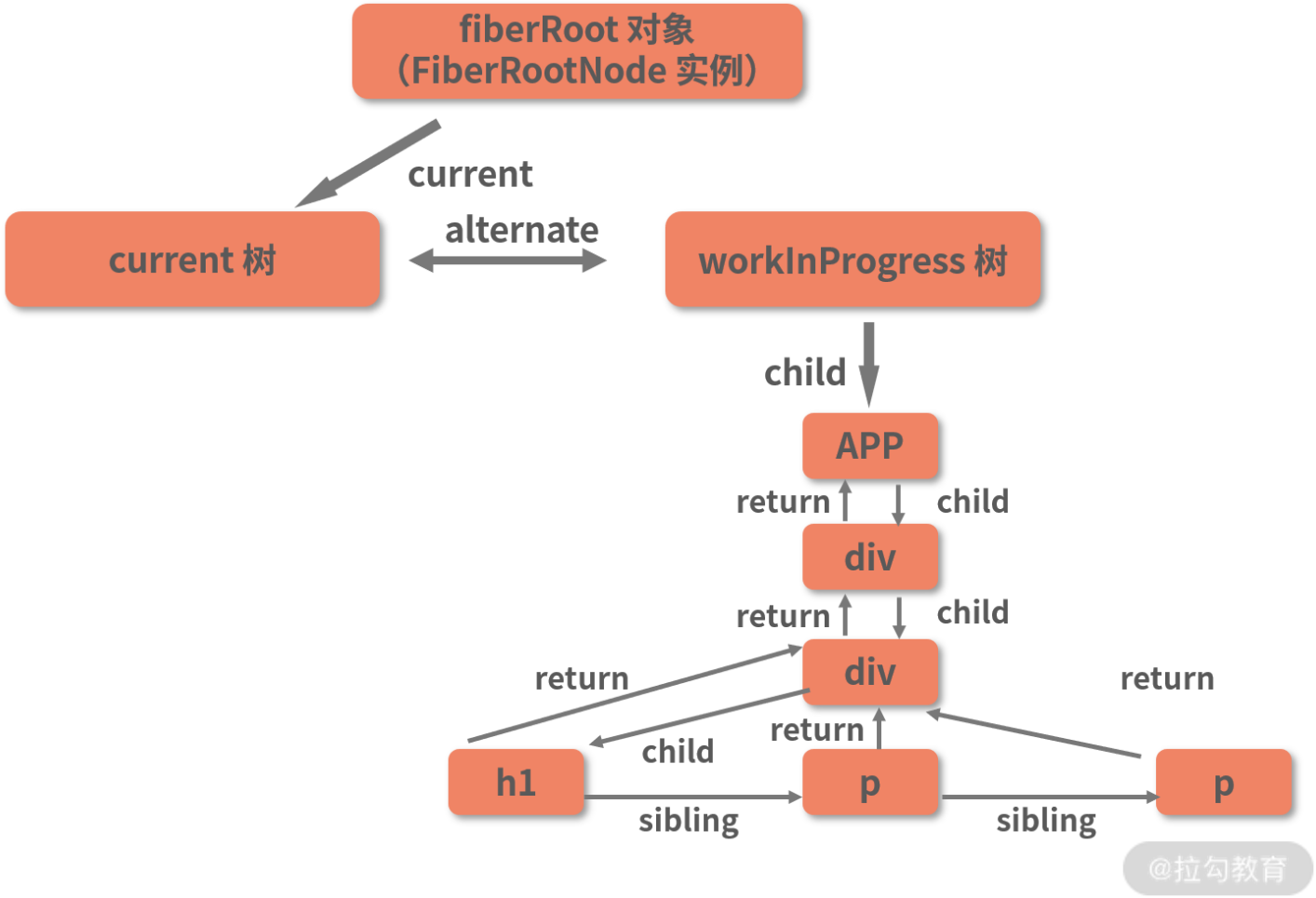
```
            if (prepareToHydrateHostInstance(workInProgress, rootContainerInstar
              markUpdate(workInProgress);
            }
          } else {
            // 这一步很关键， createInstance 的作用是创建 DOM 节点
            var instance = createInstance(type, newProps, rootContainerInstance,
            // appendAllChildren 会尝试把上一步创建好的 DOM 节点挂载到 DOM 树上去
            appendAllChildren(instance, workInProgress, false, false);
            // stateNode 用于存储当前 Fiber 节点对应的 DOM 节点
            workInProgress.stateNode = instance;

            // finalizeInitialChildren 用来为 DOM 节点设置属性
            if (finalizeInitialChildren(instance, type, newProps, rootContainer
              markUpdate(workInProgress);
            }
          }
          ......
        }
        return null;
      }
    case HostText:
      {
        ......
      }
    case SuspenseComponent:
      {
        ......
      }
    case HostPortal:
      ......
      return null;
    case ContextProvider:
      ......
      return null;
    ......
  }
  {
    {
      throw Error("Unknown unit of work tag (" + workInProgress.tag + "). This e
    }
  }
}
```

试图捋顺这段 completeWork 逻辑，你需要掌握以下几个要点。

1. completeWork 的核心逻辑是一段体量巨大的 switch 语句，在这段 switch 语句中，**completeWork 将根据 workInProgress 节点的 tag 属性的不同，进入不同的 DOM 节点的创建、处理逻辑**。

2. 在 Demo 示例中，h1 节点的 tag 属性对应的类型应该是 HostComponent，也就是“原生 DOM 元素类型”。

3. completeWork 中的 current、 workInProgress 分别对应的是下图中左右两棵 Fiber 树上的节点：



其中 workInProgress 树代表的是“当前正在 render 中的树”，而 current 树则代表“已经存在的树”。

workInProgress 节点和 current 节点之间用 alternate 属性相互连接。在组件的挂载阶段，current 树只有一个 rootFiber 节点，并没有其他内容。因此 h1 这个 workInProgress 节点对应的 current 节点是 null。

带着上面这些前提，再去结合注释读一遍上面提炼出来的源码，思路是不是就清晰多了？

捋顺思路后，我们直接来提取知识点。关于 completeWork，你需要明白以下几件事。

（1）用一句话来总结 completeWork 的工作内容：**负责处理 Fiber 节点到 DOM 节点的映射逻辑**。

（2）completeWork 内部有 3 个关键动作：

- **创建**DOM 节点（CreateInstance）
- 将 DOM 节点**插入**到 DOM 树中（AppendAllChildren）

- 为 DOM 节点**设置属性**（FinalizeInitialChildren）

（3）**创建好的 DOM 节点会被赋值给 workInProgress 节点的 stateNode 属性**。也就是说当我们想要定位一个 Fiber 对应的 DOM 节点时，访问它的 stateNode 属性就可以了。这里我们可以尝试访问运行时的 h1 节点的 stateNode 属性，结果如下图所示：

```
> workInProgress.stateNode
    <h1>我是标题</h1>
> workInProgress.stateNode.style
  ▸CSSStyleDeclaration {alignContent: "", alignItems: "", alignSelf: "", alignmentBaseline: "", all: "", …}
```


（4）将 DOM 节点插入到 DOM 树的操作是通过 appendAllChildren 函数来完成的。

说是将 DOM 节点插入到 DOM 树里去，实际上是将**子 Fiber 节点所对应的 DOM 节点**挂载到其**父 Fiber 节点所对应的 DOM 节点里去**。比如说在本讲 Demo 所构建出的 Fiber 树中，h1 节点的父结点是 div，那么 h1 对应的 DOM 节点就理应被挂载到 div 对应的 DOM 节点里去。

那么如果执行 appendAllChildren 时，父级的 DOM 节点还不存在怎么办?

比如 h1 节点作为第一个进入 completeWork 的节点，它的父节点 div 对应的 DOM 就尚不存在。其实不存在也没关系，反正 h1 DOM 节点被创建后，会作为 h1 Fiber 节点的 stateNode 属性存在，丢不掉的。当父节点 div 进入 appendAllChildren 逻辑后，会逐个向下查找并添加自己的后代节点，这时候，h1 就会被它的父级 DOM 节点“收入囊中”啦~

completeUnitOfWork —— 开启收集 EffectList 的“大循环”

completeUnitOfWork 的作用是开启一个大循环，在这个大循环中，将会重复地做下面三件事：

1. **针对传入的当前节点，调用 completeWork**，completeWork 的工作内容前面已经讲过，这一步应该是没有异议的；
2. 将**当前节点的副作用链**（EffectList）插入到其**父节点对应的副作用链**（EffectList）中；
3. 以当前节点为起点，循环遍历其兄弟节点及其父节点。当遍历到兄弟节点时，将 return 掉当前调用，触发兄弟节点对应的 performUnitOfWork 逻辑；而遍历到父节点时，则会直接进入下一轮循环，也就是重复 1、2 的逻辑。

步骤 1 无须多言，接下来我将为你解读步骤 2 和步骤 3 的含义。

completeUnitOfWork 开启下一轮循环的原则

在理解副作用链之前，首先要理解 completeUnitOfWork 开启下一轮循环的原则，也就是步骤 3。步骤 3 相关的源码如下所示（解析在注释里）：

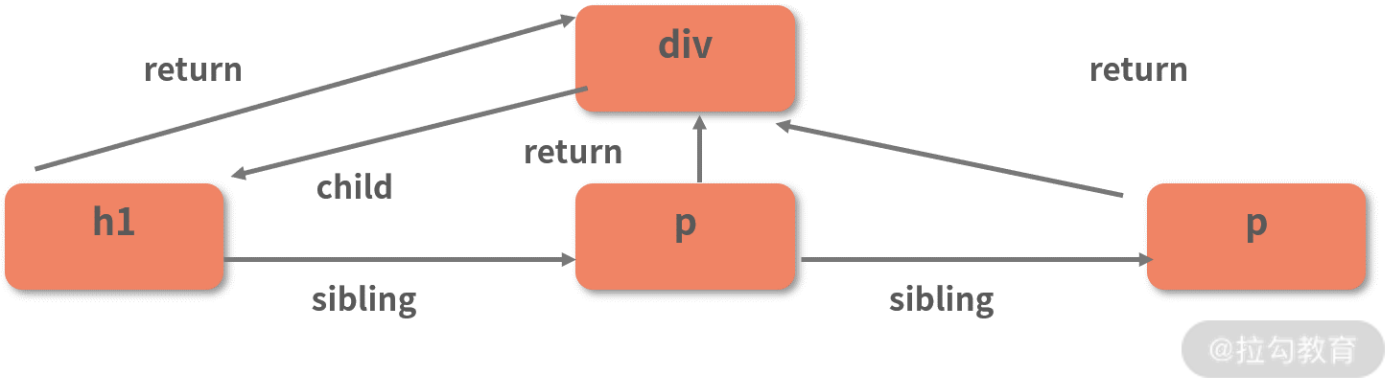
复制代码

```
do {
  ......
  // 这里省略步骤 1 和步骤 2 的逻辑

  // 获取当前节点的兄弟节点
  var siblingFiber = completedWork.sibling;

  // 若兄弟节点存在
  if (siblingFiber !== null) {
    // 将 workInProgress 赋值为当前节点的兄弟节点
    workInProgress = siblingFiber;
    // 将正在进行的 completeUnitOfWork 逻辑 return 掉
    return;
  }

  // 若兄弟节点不存在，completeWork 会被赋值为 returnFiber，也就是当前节点的父节点
  completedWork = returnFiber;
    // 这一步与上一步是相辅相成的，上下文中要求 workInProgress 与 completedWork 保持一致
  workInProgress = completedWork;
} while (completedWork !== null);
```

步骤 3 是整个循环体的收尾工作，它会在当前节点相关的各种工作都做完之后执行。

当前节点处理完了，自然是去寻找下一个可以处理的节点。我们知道，当前的 Fiber 节点之所以会进入 completeWork，是因为“递无可递”了，才会进入“归”的逻辑，这就意味着当前 Fiber 要么没有 child 节点、要么 child 节点的 completeWork 早就执行过了。因此 child 节点不会是下次循环需要考虑的对象，下次循环只需要考虑兄弟节点（siblingFiber）和父节点（returnFiber）。

那么为什么在源码中，遇到兄弟节点会 return，遇到父节点才会进入下次循环呢？这里我以 h1 节点的节点关系为例进行说明。请看下图：



结合前面的分析和图示可知，**h1 节点是递归过程中所触及的第一个叶子节点，也是其兄弟节点中被遍历到的第一个节点**；而剩下的两个 p 节点，此时都还没有被遍历到，也就是说连 beginWork 都没有执行过。

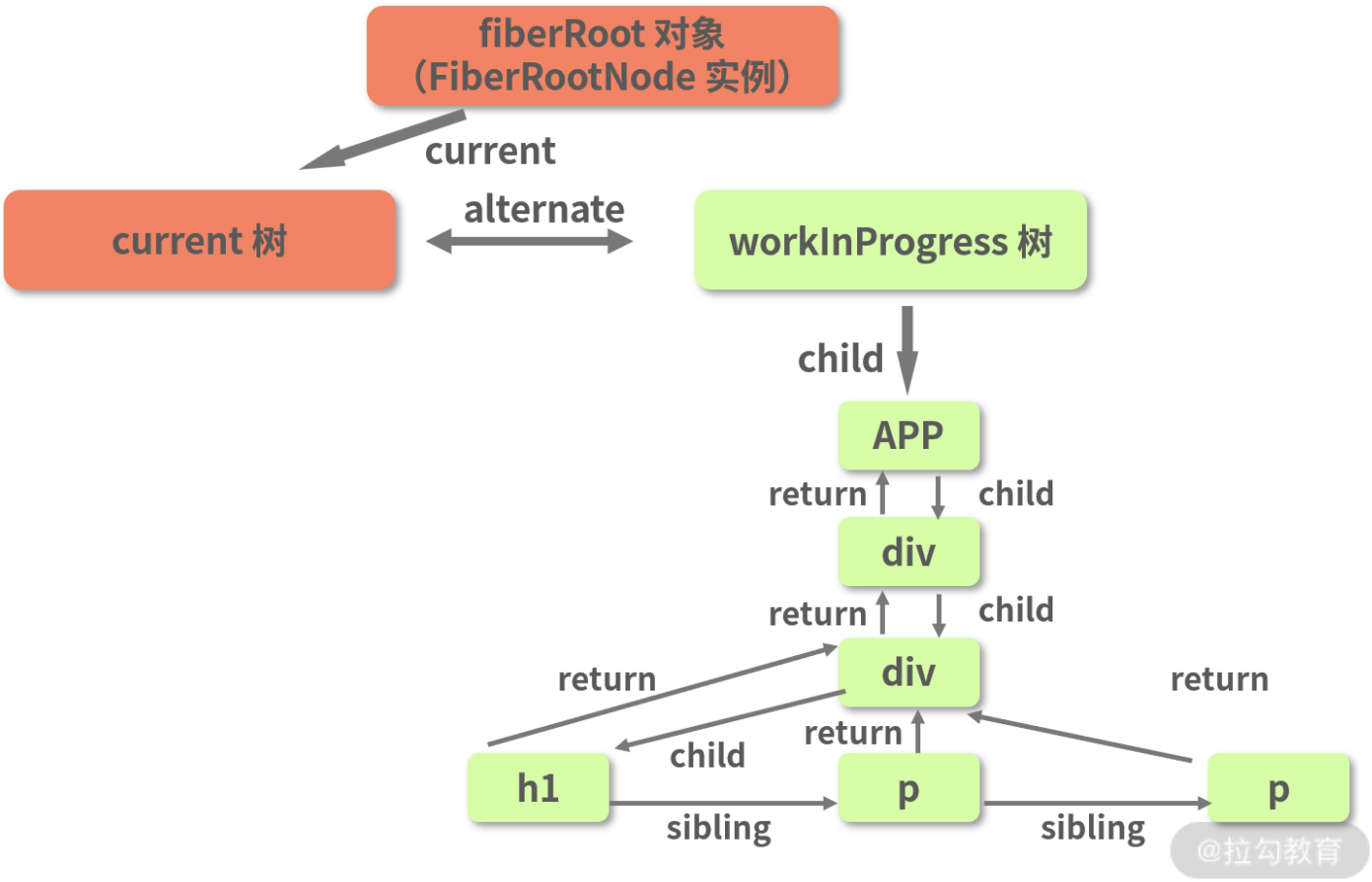
因此对于 h1 节点的兄弟节点来说，当下的第一要务是回去从 beginWork 开始走起，直到 beginWork “递无可递”时，才能够执行 completeWork 的逻辑。beginWork 的调用是在 performUnitOfWork 里发生的，因此 completeUnitOfWork 一旦识别到当前节点的兄弟节点不为空，就会终止后续的逻辑，退回到上一层的 performUnitOfWork 里去。

接下来我们再来看 h1 的父节点 div：在向下递归到 h1 的过程中，div 必定已经被遍历过了，也就是说 div 的“递”阶段（ beginWork） 已经执行完毕，只剩下“归”阶段的工作要处理了。因此，对于父节点，completeUnitOfWork 会毫不犹豫地把它推到下一次循环里去，让它进入 completeWork 的逻辑。

值得注意的是，completeUnitOfWork 中处理兄弟节点和父节点的顺序是：先检查兄弟节点是否存在，若存在则优先处理兄弟节点；确认没有待处理的兄弟节点后，才转而处理父节点。这也就意味着，**completeWork 的执行是严格自底向上的**，子节点的 completeWork 总会先于父节点执行。

副作用链（effectList）的设计与实现

无论是 beginWork 还是 completeWork，它们的应用对象都是 workInProgress 树上的节点。我们说 render 阶段是一个递归的过程，“递归”的对象，正是这棵 workInProgress 树（见下图右侧高亮部分）：

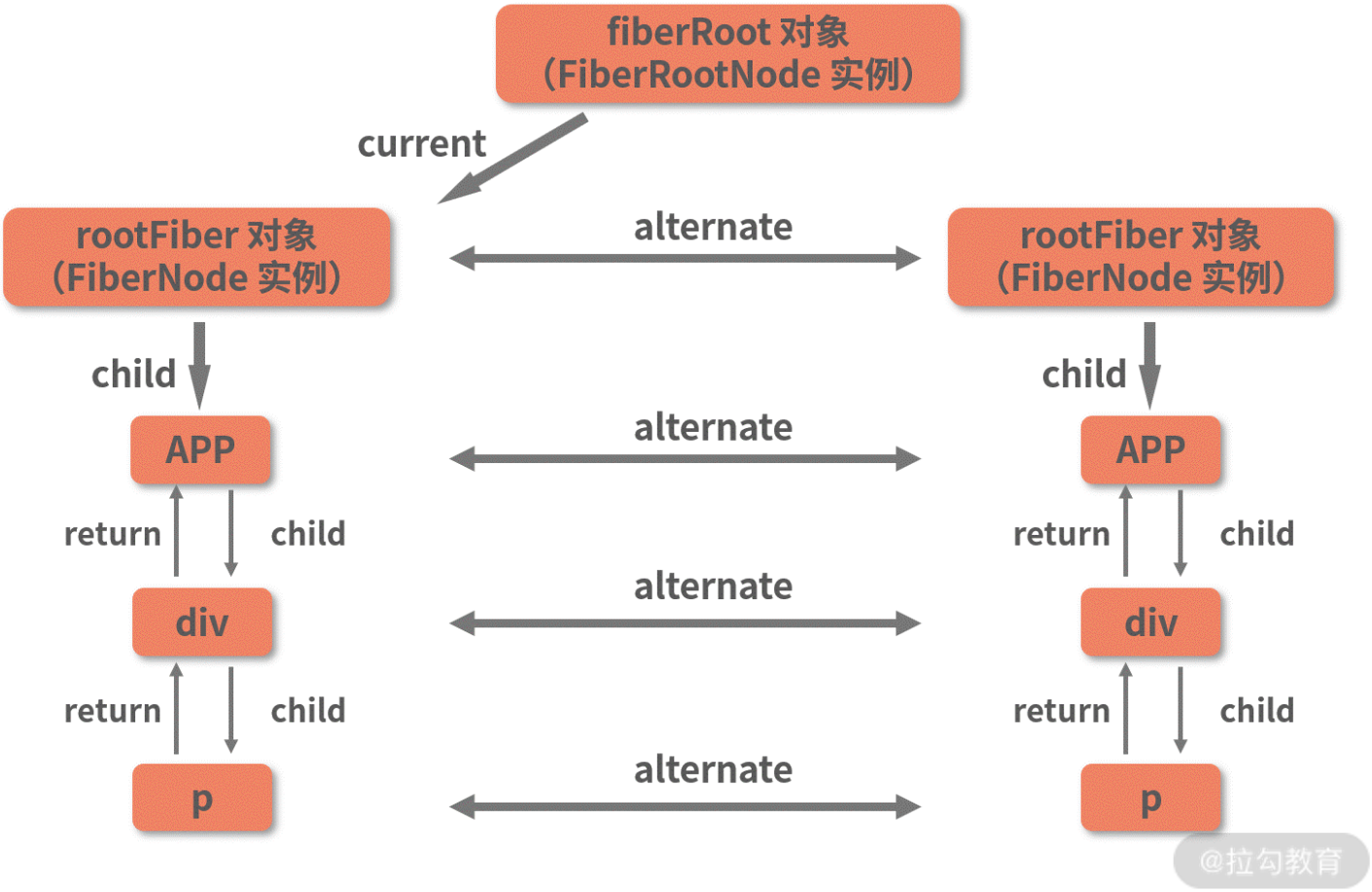


那么我们递归的目的是什么呢？或者说，render 阶段的工作目标是什么呢？

render 阶段的工作目标是找出界面中需要处理的更新。

在实际的操作中，并不是所有的节点上都会产生需要处理的更新。比如在挂载阶段，对图中的整棵 workInProgress 递归完毕后，React 会发现实际只需要对 App 节点执行一个挂载操作就可以了；而在更新阶段，这种现象更为明显。

更新阶段与挂载阶段的主要区别在于更新阶段的 current 树不为空，比如说情况可以是下图这样子的：



假如说我的某一次操作，仅仅对 p 节点产生了影响，那么对于渲染器来说，它理应只关注 p 节点这一处的更新。这时候问题就来了：**怎样做才能让渲染器又快又好地定位到那些真正需要更新的节点呢**?

在 render 阶段，我们通过艰难的递归过程来明确“p 节点这里有一处更新”这件事情。按照 React 的设计思路，render 阶段结束后，“找不同”这件事情其实也就告一段落了。**commit 只负责实现更新，而不负责寻找更新**，这就意味着我们必须找到一个办法能让 commit 阶段“坐享其成”，能直接拿到 render 阶段的工作成果。而这，正是**副作用链（effectList）**的价值所在。

副作用链（effectList） 可以理解为 render 阶段“工作成果”的一个集合：每个 Fiber 节点都维护着一个属于它自己的 effectList，effectList 在数据结构上以链表的形式存在，链表内的每一个元素都是一个 Fiber 节点。这些 Fiber 节点需要满足两个共性：

1. 都是当前 Fiber 节点的后代节点
2. 都有待处理的副作用

没错，Fiber 节点的 effectList 里记录的并非它自身的更新，而是其**需要更新的后代节点**。带着这个结论，我们再来品品小节开头 completeUnitOfWork 中的“步骤 2”：

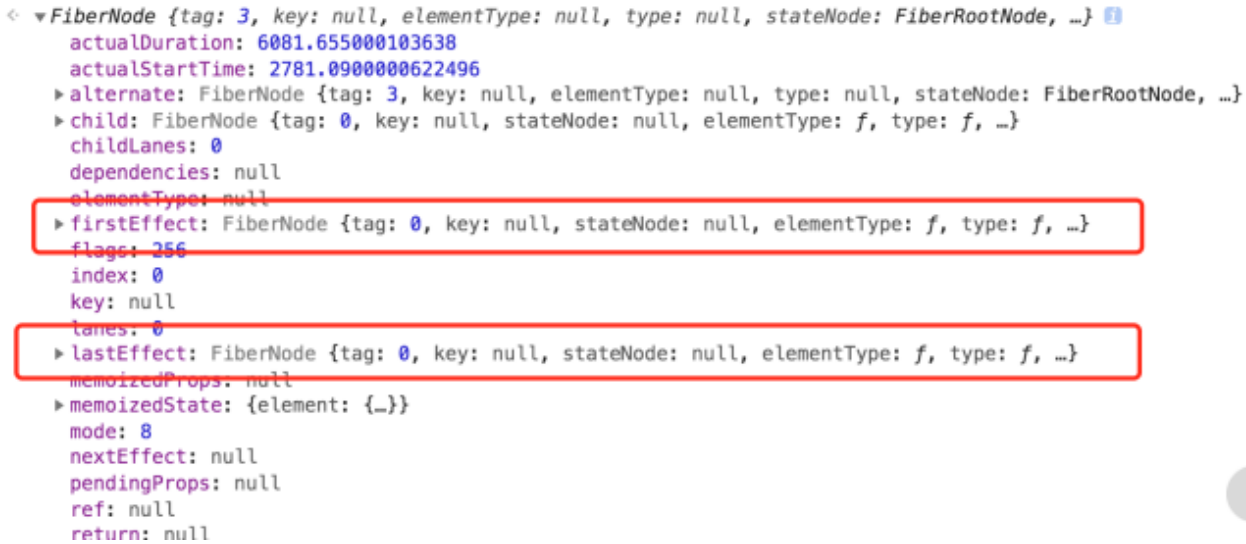
> 将**当前节点的副作用链**（effectList）插入到其**父节点对应的副作用链**（effectList）中。

咱们前面已经分析过，“**completeWork 是自底向上执行的**”，也就是说，子节点的 completeWork 总是比父节点先执行。试想，若每次处理到一个节点，都将当前节点的 effectList 插入到其父节点的 effectList 中。那么当所有节点的 completeWork 都执行完毕时，我是不是就可以从“终极父节点”，也就是 rootFiber 上，拿到一个**存储了当前 Fiber 树所有 effect Fiber**的“终极版”的 effectList 了？

把所有需要更新的 Fiber 节点单独串成一串链表，方便后续有针对性地对它们进行更新，这就是所谓的“收集副作用”的过程。

这里我以挂载过程为例，带你分析一下这个过程是如何实现的。

首先我们要知道的是，这个 effectList 链表在 Fiber 节点中是通过 firstEffect 和 lastEffect 来维护的，如下图所示：



其中 firstEffect 表示 effectList 的第一个节点，而 lastEffect 则记录最后一个节点。

对于挂载过程来说，我们唯一要做的就是把 App 组件挂载到界面上去，因此 App 后代节点们的 effectList 其实都是不存在的。effectList 只有在 App 的父节点（rootFiber）这才不为空。

那么 effectList 的创建逻辑又是怎样的呢？其实非常简单，只需要为 firstEffect 和 lastEffect 各赋值一个引用即可。以下是从 completeUnitOfWork 源码中提取出的相关逻辑（解析在注释里）：

复制代码

```
// 若副作用类型的值大于“PerformedWork”，则说明这里存在一个需要记录的副作用
if (flags > PerformedWork) {
  // returnFiber 是当前节点的父节点
  if (returnFiber.lastEffect !== null) {
    // 若父节点的 effectList 不为空，则将当前节点追加到 effectList 的末尾去
```

```
    returnFiber.lastEffect.nextEffect = completedWork;
  } else {
    // 若父节点的 effectList 为空，则当前节点就是 effectList 的 firstEffect
    returnFiber.firstEffect = completedWork;
  }

  // 将 effectList 的 lastEffect 指针后移一位
  returnFiber.lastEffect = completedWork;
}
```

代码中的 flags 咱们已经反复强调过了，它旧时的名字叫“effectTag”，是用来标识副作用类型的；而“completedWork”这个变量，在当前上下文中存储的就是“正在被执行 completeWork 相关逻辑”的节点；至于“PerformedWork”，它是一个值为 1 的常量，React 规定若 flags（又名 effectTag）的值小于等于 1，则不必提交到 commit 阶段。因此 completeUnitOfWork 只会对 flags 大于 PerformedWork 的 effect fiber 进行收集。

结合这些信息，再去读一遍源码片段，相信你的理解过程就会很流畅了。这里我以 App 节点为例，带你走一遍 effectList 的创建过程：

1. App FiberNode 的 flags 属性为 3，大于 PerformedWork，因此会进入 effectList 的创建逻辑；
2. 创建 effectList 时，并不是为当前 Fiber 节点创建，而是为它的父节点创建，App 节点的父节点是 rootFiber，rootFiber 的 effectList 此时为空；
3. rootFiber 的 firstEffect 和 lastEffect 指针都会指向 App 节点，App 节点由此成为 effectList 中的唯一一个 FiberNode，如下图所示。

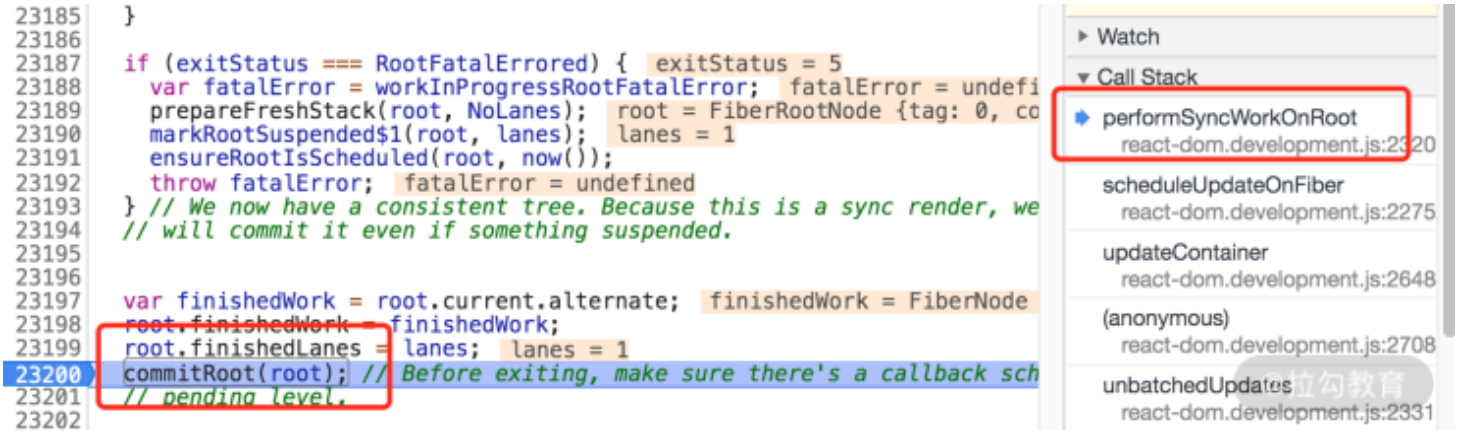
OK，读到这里，相信你已经对 effectList 的创建过程知根知底了。

现在，即便你对部分源码细节的消化可能没有那么快，也请你不要因为这些细节去中断自己串联整个渲染链路的思路。你只需要把握住“根节点（rootFiber）上的 effectList 信息，是 commit 阶段的更新线索”这个结论，就足以将 render 阶段和 commit 阶段串联起来。

commit 阶段工作流简析

在整个 ReactDOM.render 的渲染链路中，render 阶段是 Fiber 架构的核心体现，也是我们讲解的重点。对于 render 阶段，我对你的期望是“熟悉”，为了达成这个目标，我们对 render 阶段的学习还会再持续一个课时；而对于 commit 阶段，我只要求你做到“了解”。因此这里我会快速地带你过一遍 commit 阶段的重点知识，不占用你太多时间。

commit 会在 performSyncWorkOnRoot 中被调用，如下图所示：



这里的入参 root 并不是 rootFiber，而是 fiberRoot（FiberRootNode）实例。fiberRoot 的 current 节点指向 rootFiber，因此拿到 effectList 对后续的 commit 流程来说不是什么难事。

从流程上来说，commit 共分为 3 个阶段：**before mutation、mutation、layout。**

- before mutation 阶段，**这个阶段 DOM 节点还没有被渲染到界面上去**，过程中会触发 getSnapshotBeforeUpdate，也会处理 useEffect 钩子相关的调度逻辑。
- mutation，**这个阶段负责 DOM 节点的渲染**。在渲染过程中，会遍历 effectList，根据 flags（effectTag）的不同，执行不同的 DOM 操作。
- layout，**这个阶段处理 DOM 渲染完毕之后的收尾逻辑**。比如调用 componentDidMount/componentDidUpdate，调用 useLayoutEffect 钩子函数的回调等。除了这些之外，它还会**把 fiberRoot 的 current 指针指向 workInProgress Fiber 树**。

关于 commit 阶段的实现细节，感兴趣的同学课下可以参阅 commit 相关源码，这里不再展开讨论。对于 commit，如果你只能记住一个知识点，我希望你记住**它是一个绝对同步的过程**。render 阶段可以同步也可以异步，但 commit 一定是同步的。

总结

这一讲我们完成了对 ReactDOM.render 调用栈的分析。表面上剖析的是首次渲染的渲染链路，实际上将包括同步模式下的挂载、更新链路（与挂载链路的调用栈非常相似）都串联了一遍。

虽然还没有正式介入更新链路、包括异步更新模式的讲解，但你此时其实已经具备了理解这些知识的基础：Concurrent 模式（异步渲染）与 Legacy 模式（同步渲染）在数据结构设计、核心 API 调用等方面都是一致的。**这也就意味着我们这三讲所讲解的知识，都是可以在后续的学习中复用的**。

接下来，我们就将进入更新过程的学习，揭开 Concurrent 模式及 Scheduler 的神秘面纱。同时，针对上一讲遗留下来的“为什么需要两棵树”的问题，我也会在下一讲中为你解答。