12 | 如何理解 Fiber 架构的迭代动机与设计思想?

2020/11/18 修言

在理解 Fiber 架构之前，我们先来看看 React 团队在“React 哲学”中对 React 的定位：

> 我们认为，React 是用 JavaScript 构建快速响应的大型 Web 应用程序的首选方式。它在 Facebook 和 Instagram 上表现优秀。

这段话里有 4 个字值得我们细细品味，那就是“**快速响应**”，这 4 个字可以说是 React 团队在用户体验方面最为要紧的一个追求。关于这点，在 React 15 时代已经可见一斑：正是出于对“快速响应”的执着，React 才会想方设法把原本 $O(n^3)$ 的 Diff 时间复杂度优化到了前无古人的 O(n)。

然而，随着时间的推移和业务复杂度的提升，React 曾经被人们津津乐道的 Stack Reconciler 也渐渐在体验方面显出疲态。为了更进一步贯彻“快速响应”的原则，React 团队“壮士断腕”，在 16.x 版本中将其最为核心的 Diff 算法整个重写，使其以“Fiber Reconciler”的全新面貌示人。

那么 Stack Reconciler 到底有着怎样根深蒂固的局限性，使得 React 不得不从架构层面做出改变？而 Fiber 架构又是何方神圣，基于它来实现的调和过程又有什么不同呢？本讲我们就围绕这两个大问题展开讨论。

前置知识：单线程的 JavaScript 与多线程的浏览器

大家在入门前端的时候，想必都听说过这样一个结论：JavaScript 是单线程的，浏览器是多线程的。

对于多线程的浏览器来说，它除了要处理 JavaScript 线程以外，还需要处理包括事件系统、定时器/延时器、网络请求等各种各样的任务线程，这其中，自然也包括负责处理 DOM 的**UI 渲染**线程。而 **JavaScript 线程是可以操作 DOM 的**。

这意味着什么呢？试想如果渲染线程和 JavaScript 线程同时在工作，那么渲染结果必然是难以预测的：比如渲染线程刚绘制好的画面，可能转头就会被一段 JavaScript 给改得面目全非。这就决定了**JavaScript 线程和渲染线程必须是互斥的**：这两个线程不能够穿插执行，必须串行。**当其中一个线程执行时，另一个线程只能挂起等待。**

具有相似特征的还有事件线程，浏览器的 Event-Loop 机制决定了事件任务是由一个异步队列来维持的。当事件被触发时，对应的任务不会立刻被执行，而是由事件线程把它添加到任务队列的末尾，**等待 JavaScript** 的同步代码执行完毕后，在空闲的时间里执行出队。

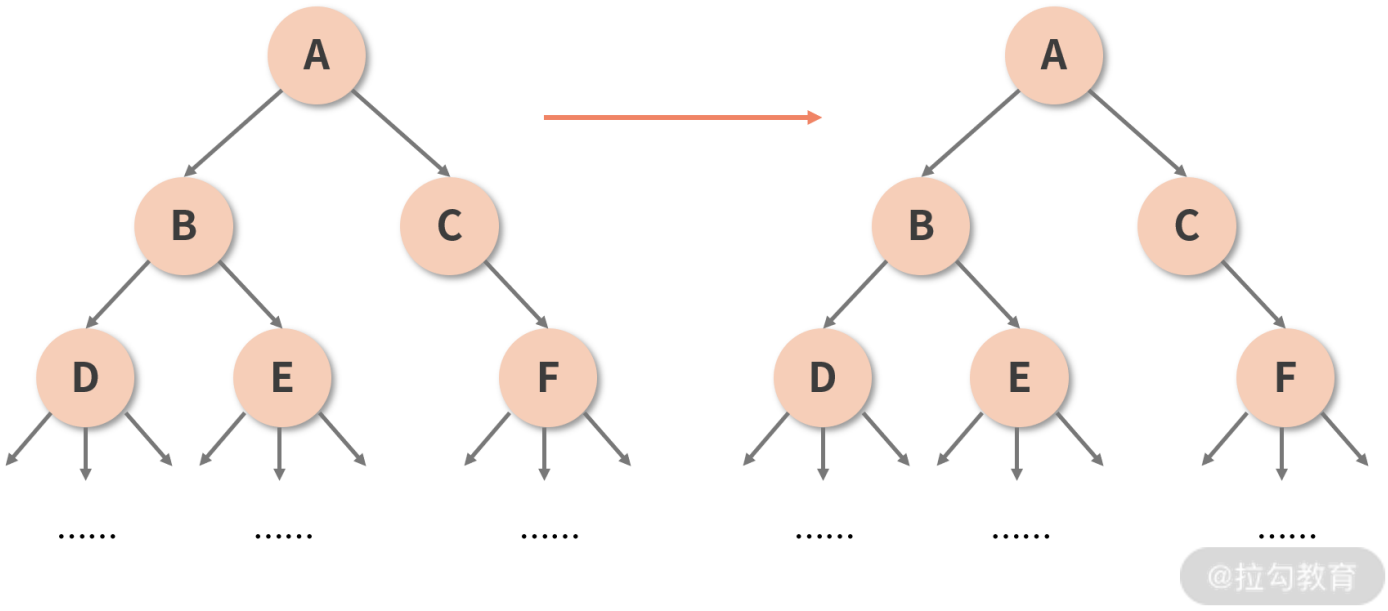
在这样的机制下，若 JavaScript 线程长时间地占用了主线程，那么**渲染层面的更新就不得不长时间地等待，界面长时间不更新，带给用户的体验就是所谓的“卡顿”**。一般页面卡顿的时候，你会做什么呢？我个人的习惯是更加频繁地在页面上点来点去，期望页面能够给我哪怕一点点的响应。遗憾的是，**事件线程也在等待 JavaScript，这就导致你触发的事件也将是难以被响应的**。

试想一下界面不更新、交互无反应的这种感觉，是不是非常令人抓狂？这其实正是 Stack Reconciler 后期所面临的困局。

为什么会产生“卡顿”这样的困局？

Stack Reconciler 所带来的一个无解的问题，正是**JavaScript 对主线程的超时占用问题**。为什么会出现这个问题？这就对应上了我们“**第 10 讲**”中所强调的一个关键知识点——**Stack Reconciler 是一个同步的递归过程**。

同步的递归过程，意味着不撞南墙不回头，意味着一旦更新开始，就像吃了炫迈一样，根本停不下来。这里我用一个案例来帮你复习一下这个过程，请先看下面这张图：



在 React 15 及之前的版本中，虚拟 DOM 树的数据结构载体是计算机科学中的“树”，其 Diff 算法的遍历思路，也是沿袭了传统计算机科学中“对比两棵树”的算法，在此基础上优化得来。因此从本质上来

说，栈调和机制下的 Diff 算法，其实是**树的深度优先遍历的过程**。而树的深度优先遍历，总是和递归脱不了关系。

拿这棵树来举例，若 A 组件发生了更新，那么栈调和的工作过程是这样的：对比第 1 层的两个 A，确认节点可复用，继续 Diff 其子组件。当 Diff 到 B 的时候，对比前后的两个 B 节点，发现可复用，于是继续 Diff 其子节点 D、E。待 B 树最深层的 Diff 完成、逐层回溯后，再进入 C 节点的 Diff 逻辑……调和器会重复“父组件调用子组件”的过程，直到最深的一层节点更新完毕，才慢慢向上返回。

这个过程的致命性在于它是同步的，不可以被打断。当处理结构相对复杂、体量相对庞大的虚拟 DOM 树时，**Stack Reconciler 需要的调和时间会很长，这就意味着 JavaScript 线程将长时间地霸占主线程，进而导致我们上文中所描述的渲染卡顿/卡死、交互长时间无响应等问题**。

设计思想：Fiber 是如何解决问题的

什么是 Fiber？从字面上来理解，Fiber 这个单词翻译过来是“丝、纤维”的意思，是比线还要细的东西。在计算机科学里，我们有进程、线程之分，而 **Fiber 就是比线程还要纤细的一个过程，也就是所谓的“纤程”**。纤程的出现，**意在对渲染过程实现更加精细的控制**。

Fiber 是一个多义词。从架构角度来看，Fiber 是对 React 核心算法（即调和过程）的重写；从编码角度来看，Fiber 是 React 内部所定义的一种数据结构，它是 Fiber 树结构的节点单位，也就是 React 16 新架构下的“虚拟 DOM”；从工作流的角度来看，Fiber 节点保存了组件需要更新的状态和副作用，一个 Fiber 同时也对应着一个工作单元。

本讲我们将站在架构角度来理解 Fiber。

Fiber 架构的应用目的，按照 React 官方的说法，是实现“**增量渲染**”。所谓“增量渲染”，通俗来说就是把一个渲染任务分解为多个渲染任务，而后将其分散到多个帧里面。不过严格来说，增量渲染其实也只是一种手段，**实现增量渲染的目的，是为了实现任务的可中断、可恢复，并给不同的任务赋予不同的优先级，最终达成更加顺滑的用户体验**。

Fiber 架构核心：“可中断”“可恢复”与“优先级”

在 React 16 之前，React 的渲染和更新阶段依赖的是如下图所示的两层架构：

正如上文所分析的那样，Reconciler 这一层负责对比出新老虚拟 DOM 之间的变化，Renderer 这一层负责将变化的部分应用到视图上，从 Reconciler 到 Renderer 这个过程是严格同步的。

而在 React 16 中，为了实现“可中断”和“优先级”，两层架构变成了如下图所示的三层架构：

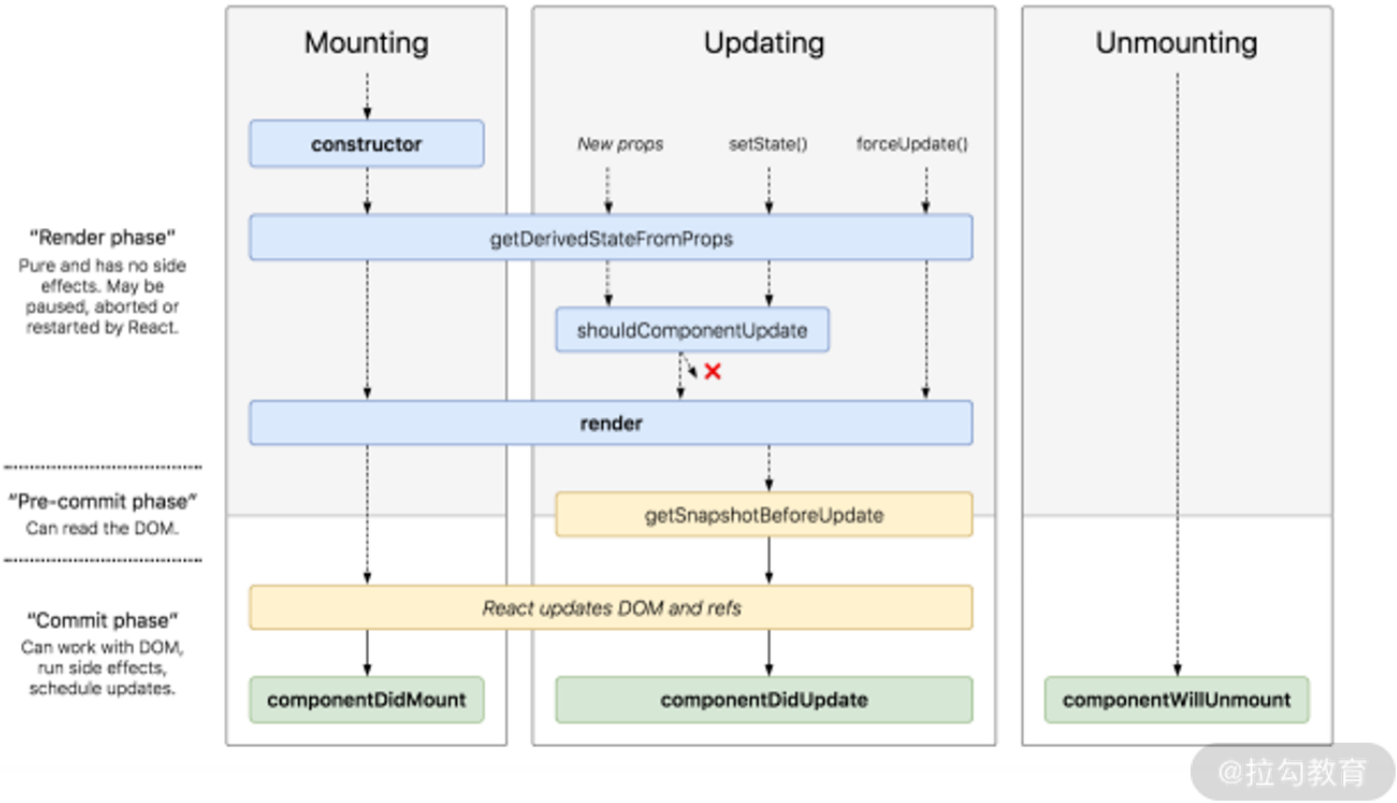
多出来的这层架构，叫作“Scheduler（调度器）”，调度器的作用是调度更新的优先级。

在这套架构模式下，更新的处理工作流变成了这样：首先，**每个更新任务都会被赋予一个优先级**。当更新任务抵达调度器时，高优先级的更新任务（记为 A）会更快地被调度进 Reconciler 层；此时若有新的更新任务（记为 B）抵达调度器，调度器会检查它的优先级，若发现 B 的优先级高于当前任务 A，那么当前处于 Reconciler 层的 A 任务就会被**中断**，调度器会将 B 任务推入 Reconciler 层。当 B 任务完成渲染后，新一轮的调度开始，之前被中断的 **A 任务将会被重新推入 Reconciler 层，继续它的渲染之旅，这便是所谓“可恢复”**。

以上，便是架构层面对“可中断”“可恢复”与“优先级”三个核心概念的处理。

Fiber 架构对生命周期的影响

在基础篇我们曾经探讨过，React 16 的生命周期分为这样三个阶段，如下图所示：

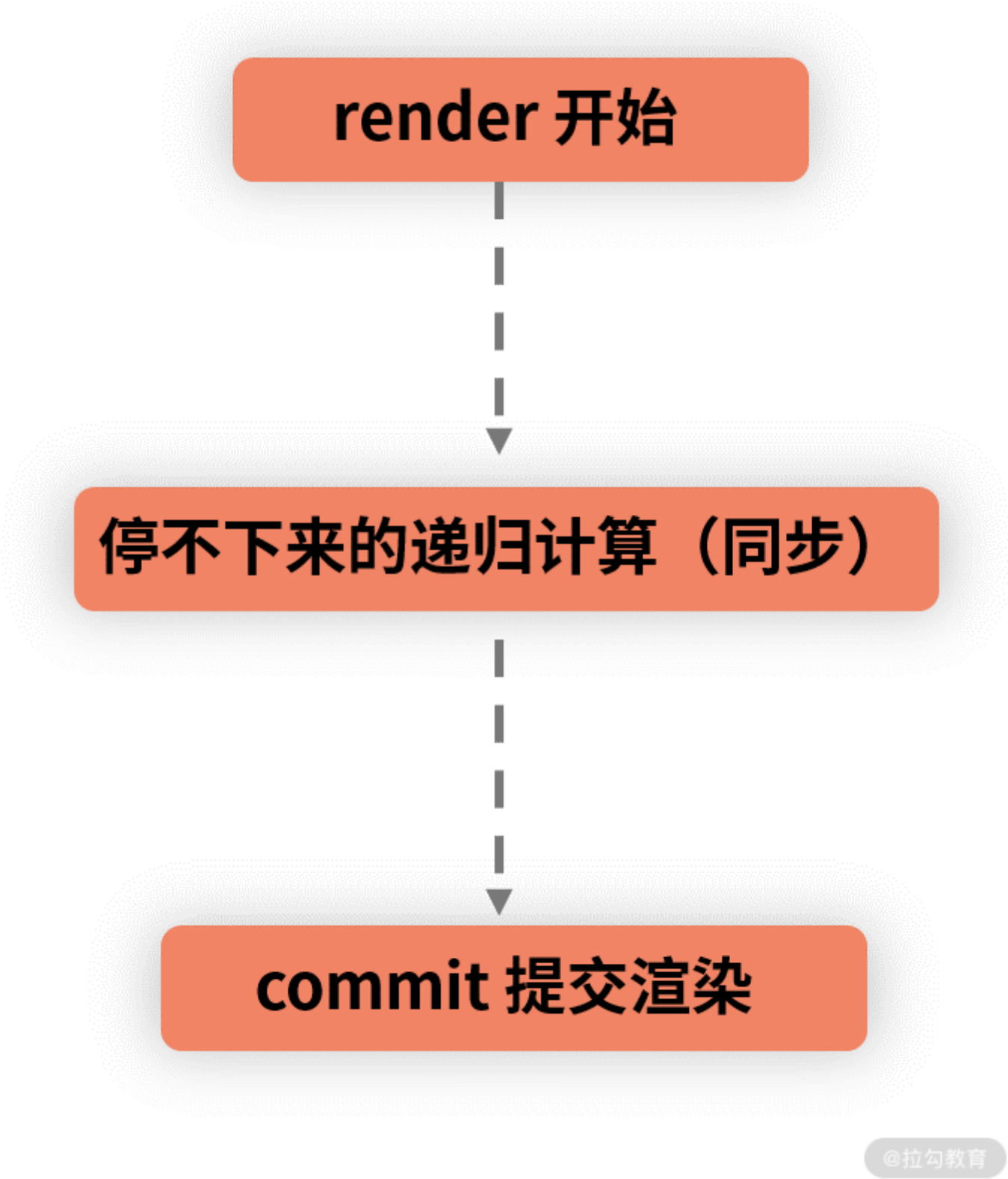


1. render 阶段：纯净且没有副作用，可能会被 React 暂停、终止或重新启动。
2. pre-commit 阶段：可以读取 DOM。
3. commit 阶段：可以使用 DOM，运行副作用，安排更新。

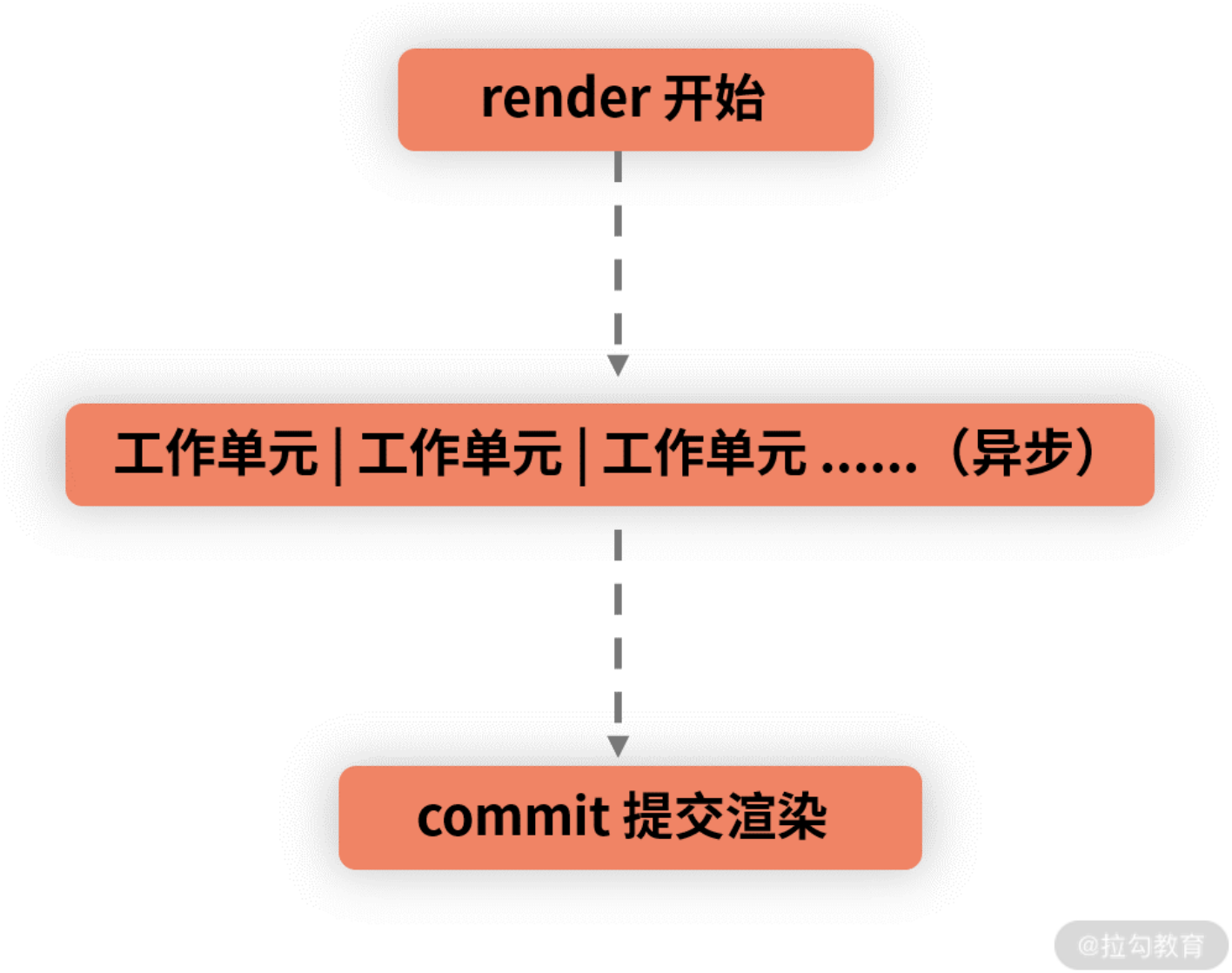
其中 pre-commit 和 commit 从大阶段上来看都属于 commit 阶段。

在 render 阶段，React 主要是在内存中做计算，明确 DOM 树的更新点；而 commit 阶段，则负责把 render 阶段生成的更新真正地执行掉。这两个阶段做的事情，非常适合和本讲刚刚讲过的 React 架构分层结合起来理解。

首先我们来看 React 15 中从 render 到 commit 的过程：



而在 React 16 中，render 到 commit 的过程变成了这样，如下图所示：



可以看出，新老两种架构对 React 生命周期的影响主要在 render 这个阶段，这个影响是通过增加 Scheduler 层和改写 Reconciler 层来实现的。

在 render 阶段，一个庞大的更新任务被分解为了一个个的工作单元，这些工作单元有着不同的优先级，React 可以根据优先级的高低去实现工作单元的打断和恢复。由于 render 阶段的操作对用户来说其实是“不可见”的，所以就算打断再重启，对用户来说也是 0 感知。但是，工作单元（也就是任务）的重启将会伴随着对部分生命周期的重复执行，这些生命周期是：

- componentWillMount
- componentWillUpdate
- shouldComponentUpdate

- componentWillReceiveProps

其中 shouldComponentUpdate 的作用是通过返回 true 或者 false，来帮助我们判断更新的必要性，一般在这个函数中不会进行副作用操作，因此风险不大。

而 “componentWill” 开头的三个生命周期，则常年被开发者以各种各样的姿势滥用，是副作用的“重灾区”。关于这点，我在第 03 讲“为什么 React 16 要更改组件的生命周期？（下）”中已经有过非常细致的讲解，此处不再赘述。你在这里需要做的，是把 React 架构分层的变化与生命周期的变化建立联系，从而对两者的设计动机都形成更加深刻的理解。

总结

通过本讲的学习，你已经知道了 React 16 中 Fiber 架构的架构分层和宏观视角下的工作流。但这一切，都还只是我们学习 Fiber Reconciler 的一个起点。Fiber Reconciler 目前对于你来说仍然是一个黑盒，关于它，还有太多的谜题需要我们一一去探索，这些谜题包括但不限于：

- React 16 在所有情况下都是异步渲染的吗？
- Fiber 架构中的“可中断”“可恢复”到底是如何实现的？
- Fiber 树和传统虚拟 DOM 树有何不同？
- 优先级调度又是如何实现的？
- ……

所有这些问题的答案，我们都需要从 Fiber 架构下的 React 源码中去寻找。

下一讲我们就将以 ReactDOM.render 串联起的渲染链路作为引子，切入对 Fiber 相关源码的探讨。

ReactDOM.render 之后到底发生了什么？this.setState 之后又发生了什么？我想，当你对这两个问题形成概念之后，上面罗列出的所有小问题都将迎刃而解。