

## 13 | ReactDOM.render 是如何串联渲染链路的？（上）

2020/11/23 修言



38.25M

00:00/14:40



看视频

由于 ReactDOM.render 的内容比较多，所以这里拆分了上中下三讲来讲解。

在上一讲，我们站在宏观角度对 Fiber 的架构分层和迭代动机有了充分的把握。从本讲开始，我们将以首次渲染为切入点，拆解 Fiber 架构下 ReactDOM.render 所触发的渲染链路，结合源码理解整个链路中所涉及的初始化、render 和 commit 等过程。

### ReactDOM.render 调用栈的逻辑分层

开篇先给到你一个简单的 React AppDemo：

```
1. import React from "react";
2. import ReactDOM from "react-dom";
3.
4. function App() {
5.   return (
6.     <div className="App">
7.       <div className="container">
8.         <h1>我是标题</h1>
9.         <p>我是第一段话</p>
10.        <p>我是第二段话</p>
11.      </div>
12.    </div>
13.  );
14. }
15.
16. const rootElement = document.getElementById("root");
17. ReactDOM.render(<App />, rootElement);
```

■ 复制代码

Demo 启动后，渲染出的界面如下图所示：

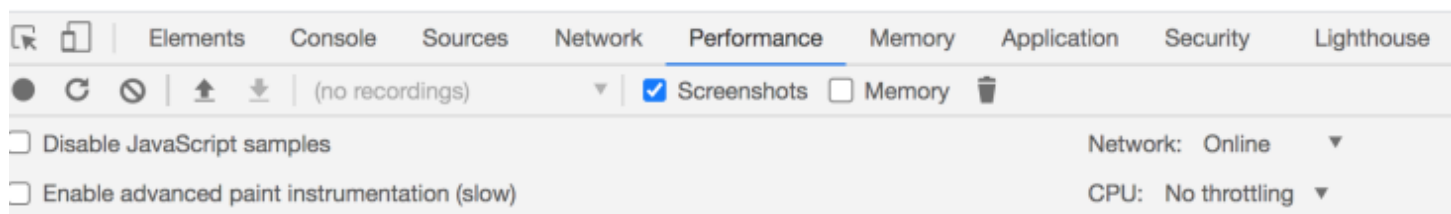
# 我是标题

我是第一段话


我是第二段话

@拉勾教育

现在请你打开 Chrome 的 Performance 面板，点击下图红色圈圈所圈住的这个“记录”按钮：



Click the record button  or hit **⌘ E** to start a new recording.

Click the reload button  or hit **⌘ ⇧ E** to record the page load.

After recording, select an area of interest in the overview by dragging. Then, zoom and pan the timeline with the mousewheel or **WASD** keys.

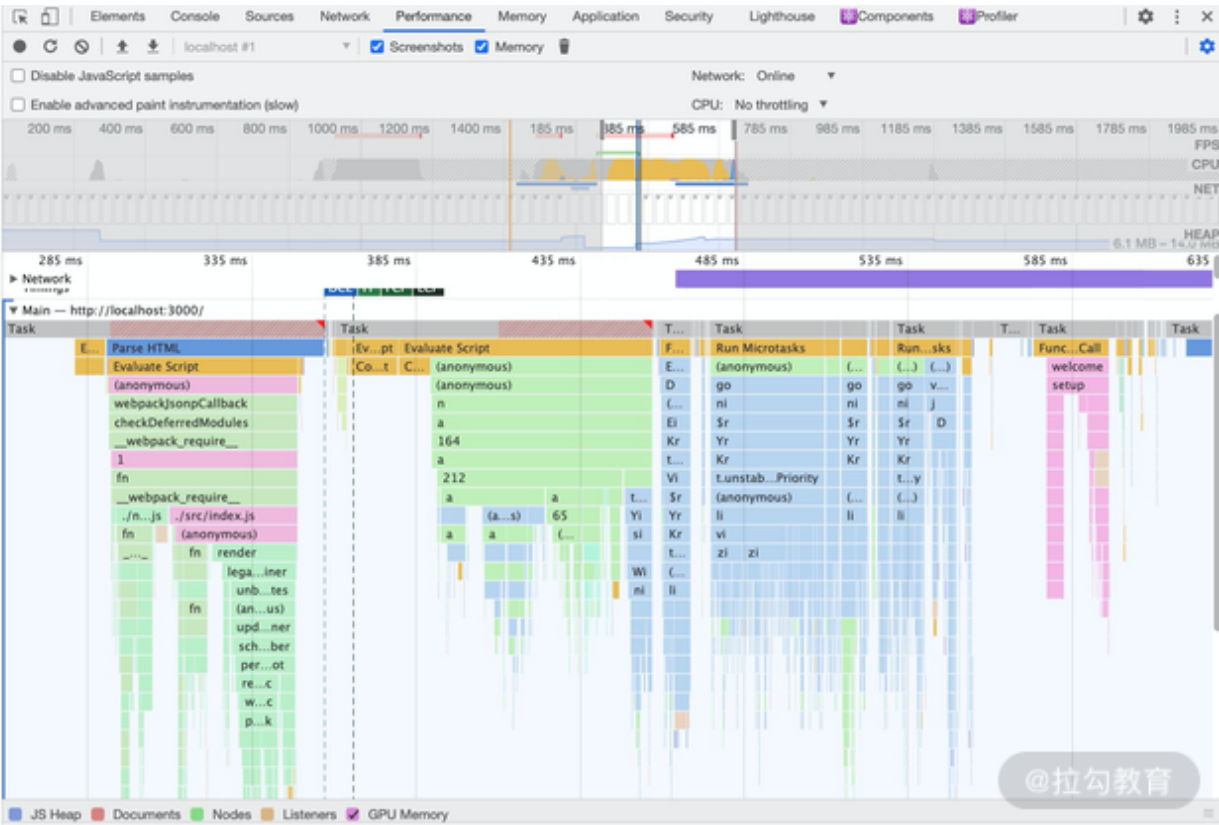
[Learn more](#)

@拉勾教育

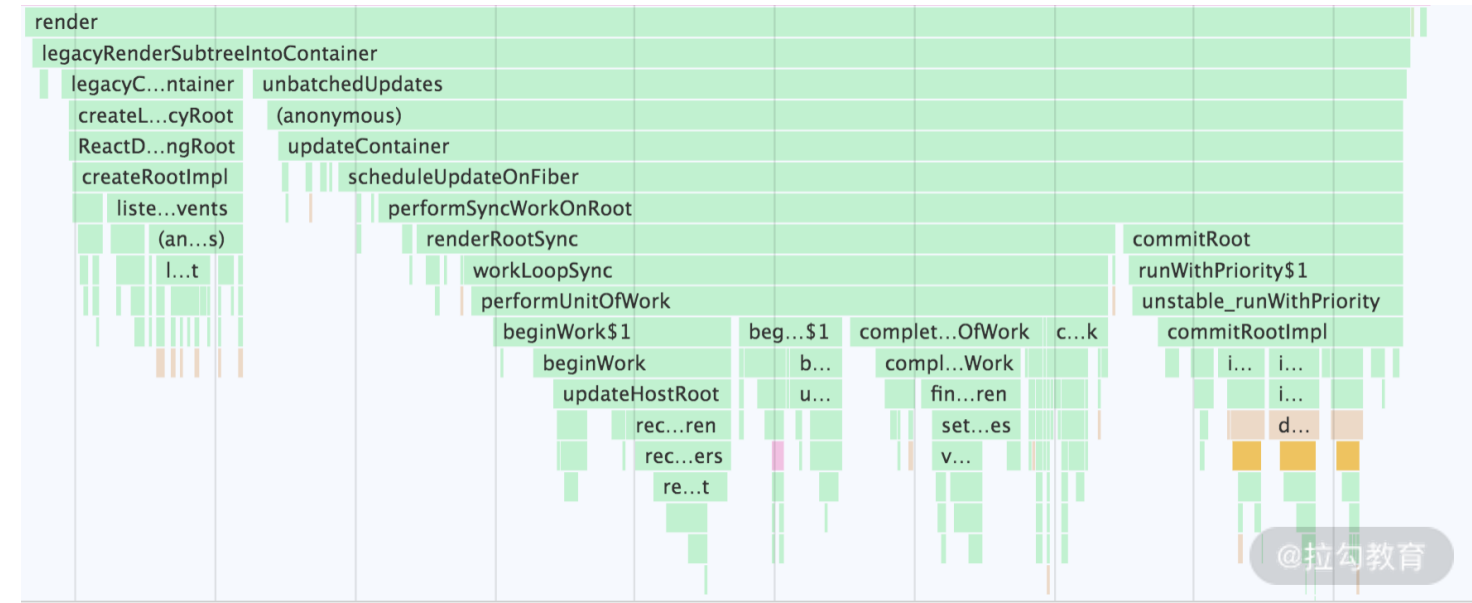
然后重新访问 Demo 页面对应的本地服务地址，待页面刷新后，终止记录，便能够得到如下图右下角所示的这样一个调用栈大图：

我是标题

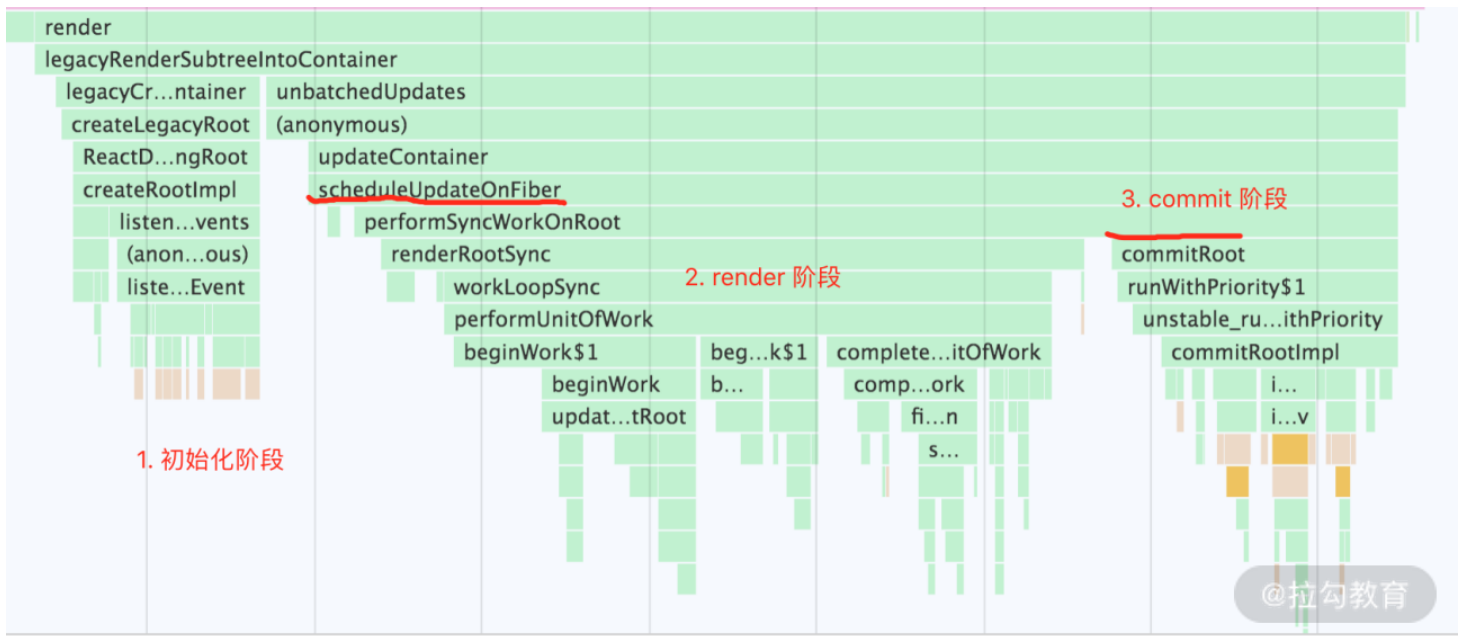
我是第一段话  
我是第二段话



放大该图，定位“src/index.js”这个文件路径，我们就可以找到 ReactDOM.render 方法对应的调用栈，如下图所示：



从图中你可以看到，ReactDOM.render 方法对应的调用栈非常深，中间涉及的函数数量也比较大。如果这张图使你心里发虚，请先不要急于撤退——分析调用栈只是我们理解渲染链路的一个手段，我们的目的是借此提取关键逻辑，而非理解调用栈中的每一个方法。就这张图来说，你首先需要把握的，就是整个调用链路中所包含的三个阶段：



图中 `scheduleUpdateOnFiber` 方法的作用是调度更新，在由 `ReactDOM.render` 发起的首屏渲染这个场景下，它触发的就是 `performSyncWorkOnRoot`。`performSyncWorkOnRoot` 开启的正是我们反复强调的 **render 阶段**；而 `commitRoot` 方法开启的则是真实 DOM 的渲染过程（**commit 阶段**）。因此以 `scheduleUpdateOnFiber` 和 `commitRoot` 两个方法为界，我们可以大致把 `ReactDOM.render` 的调用栈划分为三个阶段：

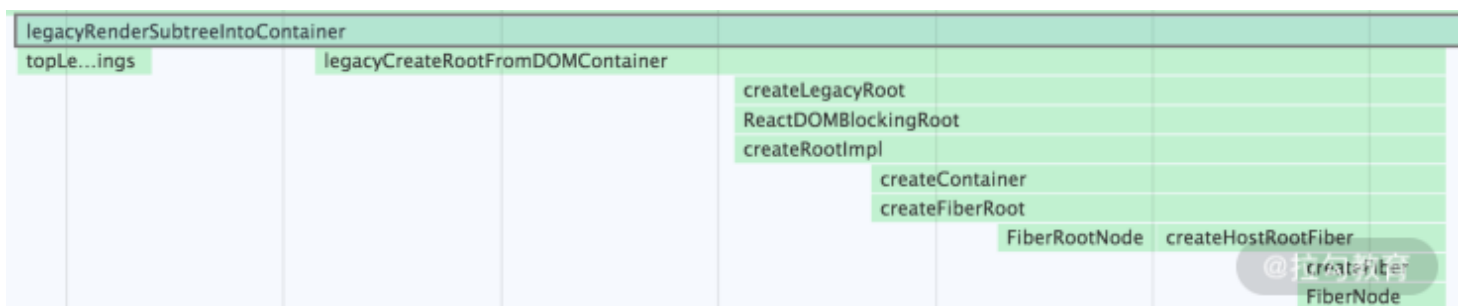
1. 初始化阶段
2. render 阶段
3. commit 阶段

接下来，我们就一起来看看这三个阶段分别做了哪些事情。

注：渲染链路串讲已被拆分为 3 个课时，本课时讲解的是初始化阶段。

#### 拆解 ReactDOM.render 调用栈——初始化阶段

首先我们提取出初始化过程中涉及的调用栈大图：



图中的方法虽然看上去又多又杂，但做的事情清清爽爽，那就是完成 **Fiber** 树中基本实体的创建。

什么是基本实体？基本实体有哪些？问题的答案藏在源码里，这里我为你提取了源码中的关键逻辑，首先是 `legacyRenderSubtreeIntoContainer` 方法。在 `ReactDOM.render` 函数体中，以下面代码所示的姿势调用了它：

```
1. return legacyRenderSubtreeIntoContainer(null, element, container, false, callback)
```

而 `legacyRenderSubtreeIntoContainer` 的关键逻辑如下（解析在注释里）：

```
1. function legacyRenderSubtreeIntoContainer(parentComponent, children, container,
2.    // container 对应的是我们传入的真实 DOM 对象
3.    var root = container._reactRootContainer;
4.    // 初始化 fiberRoot 对象
5.    var fiberRoot;
6.    // DOM 对象本身不存在 _reactRootContainer 属性，因此 root 为空
7.    if (!root) {
8.        // 若 root 为空，则初始化 _reactRootContainer，并将其值赋值给 root
9.        root = container._reactRootContainer = legacyCreateRootFromDOMContainer(cont
10.        // legacyCreateRootFromDOMContainer 创建出的对象会有一个 _internalRoot 属性，将其
11.        fiberRoot = root._internalRoot;
12.
13.        // 这里处理的是 ReactDOM.render 入参中的回调函数，你了解即可
14.        if (typeof callback === 'function') {
15.            var originalCallback = callback;
16.            callback = function () {
17.                var instance = getPublicRootInstance(fiberRoot);
18.                originalCallback.call(instance);
19.            };
20.        } // Initial mount should not be batched.
21.        // 进入 unbatchedUpdates 方法
22.        unbatchedUpdates(function () {
23.            updateContainer(children, fiberRoot, parentComponent, callback);
24.        });
25.    } else {
26.        // else 逻辑处理的是非首次渲染的情况（即更新），其逻辑除了跳过了初始化工作，与楼上基本一致
27.        fiberRoot = root._internalRoot;
28.        if (typeof callback === 'function') {
29.            var _originalCallback = callback;
30.            callback = function () {
31.                var instance = getPublicRootInstance(fiberRoot);
32.                _originalCallback.call(instance);
33.            };
34.        } // Update
35.
36.        updateContainer(children, fiberRoot, parentComponent, callback);
37.    }
38.    return getPublicRootInstance(fiberRoot);
39. }
```

这里我为你总结一下首次渲染过程中 `legacyRenderSubtreeIntoContainer` 方法的主要逻辑链路：

调用 `legacyCreateRootFromDOMContainer`  
创建 `container._reactRootContainer` 对象，并赋值给 `root`

将 `root` 上的 `_internalRoot` 属性赋值给 `fiberRoot`

将 `fiberRoot` 与方法入参一起，传入 `updateContainer` 方法，形成回调

将 `updateContainer` 回调作为参数传入，调用 `unbatchedUpdates`

@拉勾教育

在这个流程中，你需要关注到 `fiberRoot` 这个对象。`fiberRoot` 到底是什么呢？这里我将运行时的 `root` 和 `fiberRoot` 为你截取出来，其中 `root` 对象的结构如下图所示：

```
< ▼ ReactDOMBlockingRoot {_internalRoot: FiberRootNode} ⓘ
  ▼ _internalRoot: FiberRootNode
    callbackNode: null
    callbackPriority: 90
    ▶ containerInfo: div#root
    context: null
    ▶ current: FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: FiberRootNode, ...}
    finishedExpirationTime: 0
    finishedWork: null
    firstPendingTime: 0
    firstSuspendedTime: 0
    hydrate: false
    interactionThreadID: 1
    lastExpiredTime: 0
    lastPingedTime: 0
    lastSuspendedTime: 0
    ▶ memoizedInteractions: Set(0) {}
    nextKnownPendingLevel: 0
    pendingChildren: null
    pendingContext: null
    ▶ pendingInteractionMap: Map(0) {}
    pingCache: null
    tag: 0
    timeoutHandle: -1
    ▶ __proto__: Object
  ▶ __proto__: Object
```

@拉勾教育

可以看出，root 对象（container.\_reactRootContainer）上有一个 \_internalRoot 属性，这个 \_internalRoot 也就是 fiberRoot。fiberRoot 的本质是一个 FiberRootNode 对象，其中包含一个 current 属性，该属性同样需要划重点。这里我为你高亮出 current 属性的部分内容：

```
▼ current: FiberNode
  actualDuration: 0
  actualStartTime: -1
  alternate: null
  child: null
  childExpirationTime: 0
  dependencies: null
  effectTag: 0
  elementType: null
  expirationTime: 0
  firstEffect: null
  index: 0
  key: null
  lastEffect: null
  memoizedProps: null
  memoizedState: null
  mode: 8
  nextEffect: null
  pendingProps: null
  ref: null
  return: null
  selfBaseDuration: 0
  sibling: null
  ▶ stateNode: FiberRootNode {tag: 0, current: FiberNode, containerInfo: div#root, pendingChildren: null, pingCache: null, ...
    tag: 3
    treeBaseDuration: 0
    type: null
  ▶ updateQueue: {baseState: null, baseQueue: null, shared: {...}, effects: null}
  _debugHookTypes: null
  _debugID: 1
  _debugIsCurrentlyTiming: false
```

@拉勾教育

或许你会对 current 对象包含的海量属性感到陌生和头大，但这并不妨碍你 Get 到“current 对象是一个 FiberNode 实例”这一点，而 **FiberNode**，正是 **Fiber** 节点对应的对象类型。current 对象是一个 Fiber 节点，不仅如此，它还是当前 **Fiber** 树的头部节点。

考虑到 current 属性对应的 FiberNode 节点，在调用栈中实际是由 createHostRootFiber 方法创建的，React 源码中也有多处用 rootFiber 代指 current 对象，因此下文我们将以 rootFiber 指代 current 对象。

读到这里，你脑海中应该不难形成一个这样的指向关系：



## fiberRoot 对象 (FiberRootNode 实例)

current

## rootFiber 对象 (FiberNode 实例)

@拉勾教育

其中，fiberRoot 的关联对象是真实 DOM 的容器节点；而 rootFiber 则作为虚拟 DOM 的根节点存在。这两个节点，将是后续整棵 Fiber 树构建的起点。

接下来，fiberRoot 将和 ReactDOM.render 方法的其他入参一起，被传入 updateContainer 方法，从而形成一个回调。这个回调，正是接下来要调用的 unbatchedUpdates 方法的入参。我们一起来看看 unbatchedUpdates 做了什么，下面代码是对 unbatchedUpdates 主体逻辑的提取：

```
1. function unbatchedUpdates(fn, a) {
2.   // 这里是对上下文的处理，不必纠结
3.   var prevExecutionContext = executionContext;
4.   executionContext &= ~BatchedContext;
5.   executionContext |= LegacyUnbatchedContext;
6.   try {
7.     // 重点在这里，直接调用了传入的回调函数 fn，对应当前链路中的 updateContainer 方法
8.     return fn(a);
9.   } finally {
10.    // finally 逻辑里是对回调队列的处理，此处不用太关注
11.    executionContext = prevExecutionContext;
12.    if (executionContext === NoContext) {
13.      // Flush the immediate callbacks that were scheduled during this batch
14.      resetRenderTimer();
15.      flushSyncCallbackQueue();
16.    }
17.  }
18. }
```

■ 复制代码



在 `unbatchedUpdates` 函数体里，当下你只需要 Get 到一个信息：它直接调用了传入的回调 `fn`。而在当前链路中，`fn` 是什么呢？**`fn`** 是一个针对 `updateContainer` 的调用：

[复制代码](#)

```
1. unbatchedUpdates(function () {
2.   updateContainer(children, fiberRoot, parentComponent, callback);
3. });
```

接下来我们很有必要去看看 `updateContainer` 里面的逻辑。这里我将主体代码提取如下（解析在注释里，如果没有耐心读完可以直接看文字解读）：

[复制代码](#)

```
1. function updateContainer(element, container, parentComponent, callback) {
2.   .....
3.
4.   // 这是一个 event 相关的入参，此处不必关注
5.   var eventTime = requestEventTime();
6.
7.   .....
8.
9.   // 这是一个比较关键的入参，lane 表示优先级
10.  var lane = requestUpdateLane(current$1);
11.  // 结合 lane（优先级）信息，创建 update 对象，一个 update 对象意味着一个更新
12.  var update = createUpdate(eventTime, lane);
13.
14.  // update 的 payload 对应的是一个 React 元素
15.  update.payload = {
16.    element: element
17.  };
18.
19.  // 处理 callback，这个 callback 其实就是我们调用 ReactDOM.render 时传入的 callback
20.  callback = callback === undefined ? null : callback;
21.  if (callback !== null) {
22.    {
23.      if (typeof callback !== 'function') {
24.        error('render(...): Expected the last optional `callback` argument to be');
25.      }
26.    }
27.    update.callback = callback;
28.  }
29.
30.  // 将 update 入队
31.  enqueueUpdate(current$1, update);
32.  // 调度 fiberRoot
33.  scheduleUpdateOnFiber(current$1, lane, eventTime);
34.  // 返回当前节点（fiberRoot）的优先级
35.  return lane;
36. }
```

updateContainer 的逻辑相对来说丰富了点，但大部分逻辑也是在干杂活，它做的最关键的可以总结为三件：

1. 请求当前 Fiber 节点的 lane（优先级）；
2. 结合 lane（优先级），创建当前 Fiber 节点的 update 对象，并将其入队；
3. 调度当前节点（rootFiber）。

函数体中的 scheduleWork 其实就是 scheduleUpdateOnFiber，scheduleUpdateOnFiber 函数的任务是调度当前节点的更新。在这个函数中，会处理一系列与优先级、打断操作相关的逻辑。但是在 **ReactDOM.render** 发起的首次渲染链路中，这些意义都不大，因为这个渲染过程其实是同步的。我们可以尝试在 Source 面板中为该函数打上断点，逐行执行代码，会发现逻辑最终会走到下图的高亮处：

```
22706 function scheduleUpdateOnFiber(fiber, lane, eventTime) {
22707   checkForNestedUpdates();
22708   warnAboutRenderPhaseUpdatesInDEV(fiber);
22709   var root = markUpdateLaneFromFiberToRoot(fiber, lane);
22710
22711   if (root === null) {
22712     warnAboutUpdateOnUnmountedFiberInDEV(fiber);
22713     return null;
22714   } // Mark that the root has a pending update.
22715
22716   markRootUpdated(root, lane, eventTime);
22717
22718   if (root === workInProgressRoot) {
22719     // Received an update to a tree that's in the middle of rendering. Mark
22720     // that there was an interleaved update work on this root. Unless the
22721     // 'deferRenderPhaseUpdateToNextBatch' flag is off and this is a render
22722     // phase update. In that case, we don't treat render phase updates as if
22723     // they were interleaved, for backwards compat reasons.
22724     {
22725       workInProgressRootUpdatedLanes = mergeLanes(workInProgressRootUpdatedLanes, lane);
22726     }
22727   }
22728
22729   if (workInProgressRootExitStatus === RootSuspendedWithDelay) {
22730     // The root already suspended with a delay, which means this render
22731     // definitely won't finish. Since we have a new update, let's mark it as
22732     // suspended now, right before marking the incoming update. This has the
22733     // effect of interrupting the current render and switching to the update.
22734     // TODO: Make sure this doesn't override pings that happen while we've
22735     // already started rendering.
22736     markRootSuspended$1(root, workInProgressRootRenderLanes);
22737   }
22738   // TODO: requestUpdateLanePriority also reads the priority. Pass the
22739   // priority as an argument to that function and this one.
22740
22741   var priorityLevel = getCurrentPriorityLevel();
22742
22743   if (lane === SyncLane) {
22744     if ( // Check if we're inside unbatchedUpdates
22745       (executionContext & LegacyUnbatchedContext) !== NoContext && // Check if we're not already rendering
22746       (executionContext & (RenderContext | CommitContext)) === NoContext) {
22747       // Register pending interactions on the root to avoid losing traced interaction data.
22748       schedulePendingInteractions(root, lane); // This is a legacy edge case. The initial mount of a ReactDOM.render-ed
22749       // root inside of batchedUpdates should be synchronous, but layout updates
22750       // should be deferred until the end of the batch.
22751       performSyncWorkOnRoot(root);
22752     } else {
22753       ensureRootIsScheduled(root, eventTime);
22754       schedulePendingInteractions(root, lane);
22755       if (executionContext === NoContext) {
22756         // ...
22757       }
22758     }
  
```

@拉勾教育

performSyncWorkOnRoot直译过来就是“执行根节点的同步任务”，这里的“同步”二字需要注意，它明示了接下来即将开启的是一个同步的过程。这也正是为什么在整个渲染链路中，调度（Schedule）动作没有存在感的原因。

前面我们曾经提到过，performSyncWorkOnRoot 是 render 阶段的起点，render 阶段的任务就是完成 Fiber 树的构建，它是整个渲染链路中最核心的一环。在异步渲染的模式下，render 阶段应该是一个可

打断的异步过程（下一讲我们就将针对 render 过程作详细的逻辑拆解）。

而现在，我相信你心里更多的疑惑在于：都说 Fiber 架构带来的异步渲染是 React 16 的亮点，为什么分析到现在，竟然发现 ReactDOM.render 触发的首次渲染是个同步过程呢？

同步的 ReactDOM.render，异步的 ReactDOM.createRoot

其实在 React 16，包括近期发布的 React 17 小版本中，React 都有以下 3 种启动方式：

**legacy 模式：**

`ReactDOM.render(<App />, rootNode)`。这是当前 React App 使用的方式，当前没有计划删除本模式，但是这个模式可能不支持这些新功能。

**blocking 模式：**

`ReactDOM.createBlockingRoot(rootNode).render(<App />)`。目前正在实验中，作为迁移到 concurrent 模式的第一个步骤。

**concurrent 模式：**

`ReactDOM.createRoot(rootNode).render(<App />)`。目前在实验中，未来稳定之后，打算作为 React 的默认开发模式，这个模式开启了所有的新功能。

在这 3 种模式中，我们常用的 `ReactDOM.render` 对应的是 **legacy 模式**，它实际触发的仍然是同步的**渲染链路**。blocking 模式可以理解为 legacy 和 concurrent 之间的一个过渡形态，之所以会有这个模式，是因为 React 官方希望能够提供[渐进的迁移策略](#)，帮助我们更加顺滑地过渡到 Concurrent 模式。blocking 在实际应用中是比较低频的一个模式，了解即可。

按照官方的说法，“长远来看，模式的数量会收敛，不用考虑不同的模式，但就目前而言，模式是一项重要的迁移策略，让每个人都能决定自己什么时候迁移，并按照自己的速度进行迁移”。由此可以看出，Concurrent 模式确实是 React 的终极目标，也是其创作团队使用 Fiber 架构重写核心算法的动机所在。

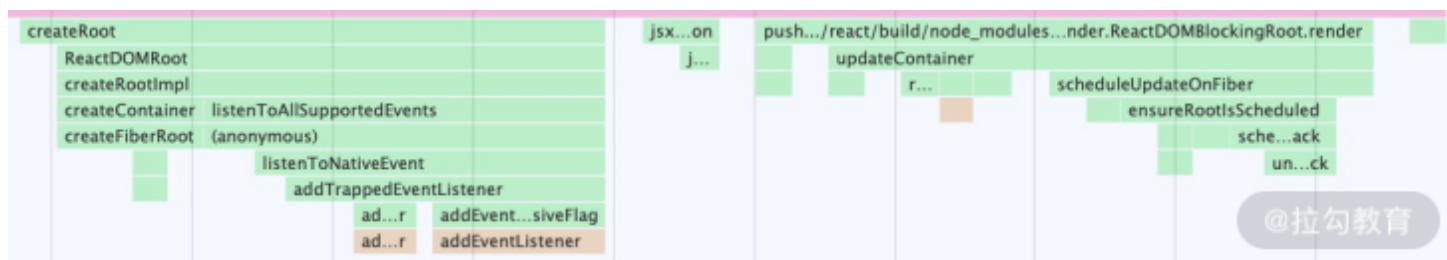
**拓展：关于异步模式下的首次渲染链路**

当下，如果想要开启异步渲染，我们需要调用 `ReactDOM.createRoot` 方法来启动应用，那 `ReactDOM.createRoot` 开启的渲染链路与 `ReactDOM.render` 有何不同呢？

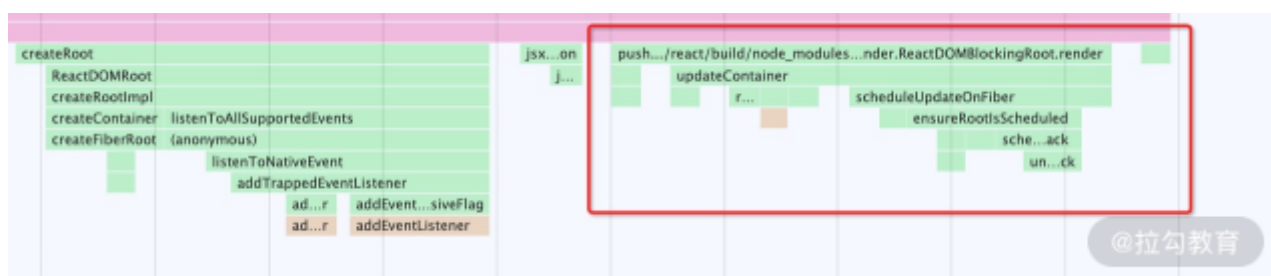
这里我修改一下调用方式，给你展示一下调用栈。由于本讲的源码取材于 React 17.0.0 版本，在这个版本中，createRoot 仍然是一个 unstable 的方法。因此实际调用的 API 应该是“unstable\_createRoot”：

```
1. ReactDOM.unstable_createRoot(rootElement).render(<App />);
```

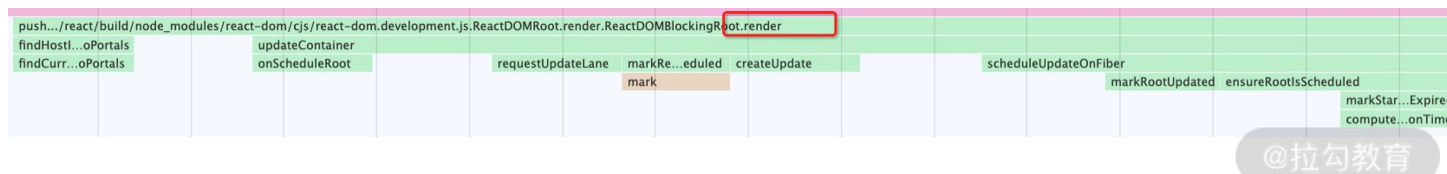
Concurrent 模式开启后，首次渲染的调用栈变成了如下图所示的样子：



乍一看，好像和 ReactDOM.render 差别很大，其实不然。图中 createRoot 所触发的逻辑仍然是一些准备性质的初始化工作，此处不必太纠结。关键在于下面我给你框出来的这部分，如下图所示：



我们拉近一点来看，如下图所示：



你会发现这地方也调用了一个 render。再顺着这个调用往下看，发现有大量的熟悉面孔：updateContainer、requestUpdateLane、createUpdate、scheduleUpdateOnFiber..... 这些函数在 ReactDOM.render 的调用栈中也出现过。

其实，当前你看到的这个 render 调用链路，和 ReactDOM.render 的调用链路是非常相似的，主要的区别在 scheduleUpdateOnFiber 的这个判断里：

```

3
4
5 if (lane === SyncLane) {
6   // Check if we're inside unbatchedUpdates
7   (executionContext & LegacyUnbatchedContext) !== NoContext && // Check if we're not already rendering
8   (executionContext & (RenderContext | CommitContext)) !== NoContext) {
9     // Register pending interactions on the root to avoid losing traced interaction data.
10    schedulePendingInteractions(root, lane); // This is a legacy edge case. The initial mount of a ReactDOM.render
11    // root inside of batchedUpdates should be synchronous, but layout updates
12    // should be deferred until the end of the batch.
13
14    performSyncWorkOnRoot(root);
15  } else {
16    ensureRootIsScheduled(root, eventTime);
17    schedulePendingInteractions(root, lane);
18
19    if (executionContext === NoContext) {
20      // Flush the synchronous work now, unless we're already working or inside
21      // a batch. This is intentionally inside scheduleUpdateOnFiber instead of
22      // scheduleCallbackForFiber to preserve the ability to schedule a callback
23      // without immediately flushing it. We only do this for user-initiated
24      // updates, to preserve historical behavior of legacy mode.
25      resetRenderTimer();
26      flushSyncCallbackQueue();
27    }
28  }
29 } else {
30   // Schedule a discrete update but only if it's not Sync.
31   if ((executionContext & DiscreteEventContext) !== NoContext && ( // Only updates at user-blocking priority or ;
32     // discrete, even inside a discrete event.
33     priorityLevel === UserBlockingPriority$2 || priorityLevel === ImmediatePriority$1)) {
34     // This is the result of a discrete event. Track the lowest priority
35     // discrete update per root so we can flush them early, if needed.
36     if (rootsWithPendingDiscreteUpdates === null) {
37       rootsWithPendingDiscreteUpdates = new Set([root]);
38     } else {
39       rootsWithPendingDiscreteUpdates.add(root);
40     }
41   } // Schedule other updates after in case the callback is sync.
42
43   ensureRootIsScheduled(root, eventTime);
44   schedulePendingInteractions(root, lane);
45 } // We use this when assigning a lane for a transition inside

```

@拉勾教育

在异步渲染模式下，由于请求到的 lane 不再是 SyncLane（同步优先级），故不会再走到 performSyncWorkOnRoot 这个调用，而是会转而执行 else 中调度相关的逻辑。

这里有个点要给你点出来——React 是如何知道当前处于哪个模式的呢？我们可以以 requestUpdateLane 函数为例，下面是它局部的代码：

```

1. function requestUpdateLane(fiber) {
2.   // 获取 mode 属性
3.   var mode = fiber.mode;
4.   // 结合 mode 属性判断当前的
5.   if ((mode & BlockingMode) === NoMode) {
6.     return SyncLane;
7.   } else if ((mode & ConcurrentMode) === NoMode) {
8.     return getCurrentPriorityLevel() === ImmediatePriority$1 ? SyncLane : SyncB
9.   }
10.   .....
11.   return lane;
12. }

```

■ 复制代码

上面代码中需要注意 fiber 节点上的 mode 属性：React 将会通过修改 mode 属性为不同的值，来标识当前处于哪个渲染模式；在执行过程中，也是通过判断这个属性，来区分不同的渲染模式。

因此不同的渲染模式在挂载阶段的差异，本质上来说并不是工作流的差异（其工作流涉及 初始化 → render → commit 这 3 个步骤），而是 mode 属性的差异。mode 属性决定着这个工作流是一气呵成（同步）的，还是分片执行（异步）的。



关于异步挂载/更新的实现细节，我们将在后续的第 16 讲“Fiber 架构实现原理与编码形态”中详细探讨。

### Fiber 架构一定是异步渲染吗？

之前我曾经被读者朋友问到过这样的问题：**React 16 如果没有开启 Concurrent 模式，那它还能叫 Fiber 架构吗？**

这个问题很有意思，从动机上来看，Fiber 架构的设计确实主要是为了 Concurrent 而存在。但经过了本讲紧贴源码的讲解，相信你也能够看出，在 React 16，包括已发布的 React 17 版本中，不管是否是 Concurrent，整个数据结构层面的设计、包括贯穿整个渲染链路的处理逻辑，已经完全用 Fiber 重构了一遍。站在这个角度来看，Fiber 架构在 React 中并不能够和异步渲染画严格的等号，它是一种**同时兼容了同步渲染与异步渲染的设计**。

### 总结

从本讲开始，我们以 ReactDOM.render 所触发的首次渲染为切入点，试图串联 React Fiber 架构下完整的工作链路，本讲为整个源码链路分析的前半部分。

正所谓“磨刀不误砍柴工”。虽然当前的进度条只推到了初始化这个位置，但在这部分的分析过程中，相信你已经对 Fiber 树的初始形态、Fiber 根节点的创建过程建立了感性的认知，同时把握住了 ReactDOM.render 同步渲染的过程特征，理解了 React 当下共存的3种渲染方式。在此基础上，我们再去理解 render 过程，就会轻松得多。

整个初始化的工作过程都是在为后续的 render 阶段做准备。现在，我们的 Fiber Tree 还处在只有根节点的起始状态。接下来，我们就要进入到最最关键的 render 阶段里去，一起去看看这棵树是怎么一点点丰满起来的，加油！