

16 | 剖析 Fiber 架构下 Concurrent 模式的实现原理

2020/12/02 修言



69.03M

00:00/26:28



看视频

你好，欢迎来到第 16 讲，关于 Fiber 架构的实现原理和编码形态，其实我们已经洋洋洒洒地分析了 3 讲了。

在过去的 3 讲里，通过对整个 ReactDOM.render 所触发的渲染链路进行了分析和串联，我们已经把 Fiber 架构在实现层面的大部分要点都过了一遍。刚讲过的这部分知识，一方面相对来说复杂度比较高，需要一些耐心反复地理解和消化；另一方面，本讲接下来要讲解的内容，也和它存在着较强的依赖关系，因此对这些前置知识的把握就显得尤为重要。

下面我说几个函数，帮你检验一下自己的学习效果：

1. performSyncWorkOnRoot
2. workLoopSync
3. performUnitOfWork
4. beginWork
5. completeWork
6. completeUnitOfWork
7. reconcileChildFibers

如果你对这些函数的执行时机和工作内容仍然感到不那么熟悉，那么不妨回到前 3 个课时里，结合案例和源码，重新捋顺一遍自己的思路，再回来续上你的知识链路。在接下来的讲解中，若对以上方法及其相关逻辑有所涉及，我将不再重复赘述。

本讲我将带你去认识 Fiber 架构最迷人的那一面——Concurrent 模式（异步渲染）下的“时间切片”和“优先级”实现。

在切入正题之前，我首先会回答上一讲遗留下来的“两棵树”问题。“两棵树”之间的合作模式足以将挂载过程和更新过程联系起来，对于本讲来说，是一个不错的学习切入点。

current 树与 workInProgress 树：“双缓冲”模式在 Fiber 架构下的实现

什么是“双缓冲”模式

“双缓冲”模式其实是一种在游戏领域由来已久的经典设计模式。为了帮助你快速理解它，这里我先举一个生活中的例子：假如你去看一场总时长只有 1 个小时的话剧，这场话剧中场不休息，需要不间断地演出。

按照剧情的需求，半个小时处需要一次转场。所谓转场，就是说话剧舞台的灯光、布景、氛围等全部要切换到另一种风格里去。在不中断演出的情况下，想要实现转场，怎么办呢？场务工作做得再快，也要十几二十分钟，这对一场时长 1 小时的话剧来说，实在太漫长了。观众也无法接受这样的剧情“卡顿”体验。

有一种解法，那就是**准备两个舞台**来做这场戏，当第一个舞台处于使用中时，第二个舞台的布局已经完成。这样当第一个舞台的表演结束时，只需要把第一个舞台的灯光灭掉，第二个舞台的灯光亮起，就可以做到剧情的无缝衔接了。

事实上，在真实的话剧中，我们也确实常常看到这样的画面——演员从舞台的左侧走到了右侧，灯光一切换，就从卧室（左侧舞台）走到了公园（右侧舞台）；又从公园（右侧舞台）走到了办公室（左侧舞台）。左侧舞台的布景从卧室变成了办公室，这个过程正是在演员利用右侧舞台表演时完成的。

在这个过程中，我们可以认为，**左侧舞台和右侧舞台分别是两套缓冲数据，而呈现在观众眼前的连贯画面，就是不同的缓冲数据交替被读取后的结果。**

在计算机图形领域，通过让图形硬件交替读取两套缓冲数据，可以实现画面的无缝切换，减少视觉效果上的抖动甚至卡顿。而在 React 中，双缓冲模式的主要利好，则是**能够帮我们较大限度地实现 Fiber 节点的复用**，从而减少性能方面的开销。

current 树与 workInProgress 树之间是如何“相互利用”的

在 React 中，current 树与 workInProgress 树，两棵树可以对标“双缓冲”模式下的两套缓冲数据：当 current 树呈现在用户眼前时，所有的更新都会由 workInProgress 树来承接。workInProgress 树将会在用户看不到的地方（内存里）悄悄地完成所有改变，直到“灯光”打到它身上，也就是 current 指针指向它的时候，此时就意味着 commit 阶段已经执行完毕，workInProgress 树变成了那棵呈现在界面上的 current 树。

接下来我将用一个 Demo，带你切身感受一把 workInProgress 树和 current 树“相互利用”的过程。代码如下：

```
1. import { useState } from 'react';
2. function App() {
3.   const [state, setState] = useState(0)
4.   return (
5.     <div className="App">
6.       <div onClick={() => { setState(state + 1) }} className="container">
7.         <p style={{ width: 128, textAlign: 'center' }}>
8.           {state}
9.         </p>
10.      </div>
11.    </div>
12.  );
13. }
14.
15. export default App;
```

[复制代码](#)

这个组件挂载后呈现出的界面很简单，就是一个数字 0，如下图所示：

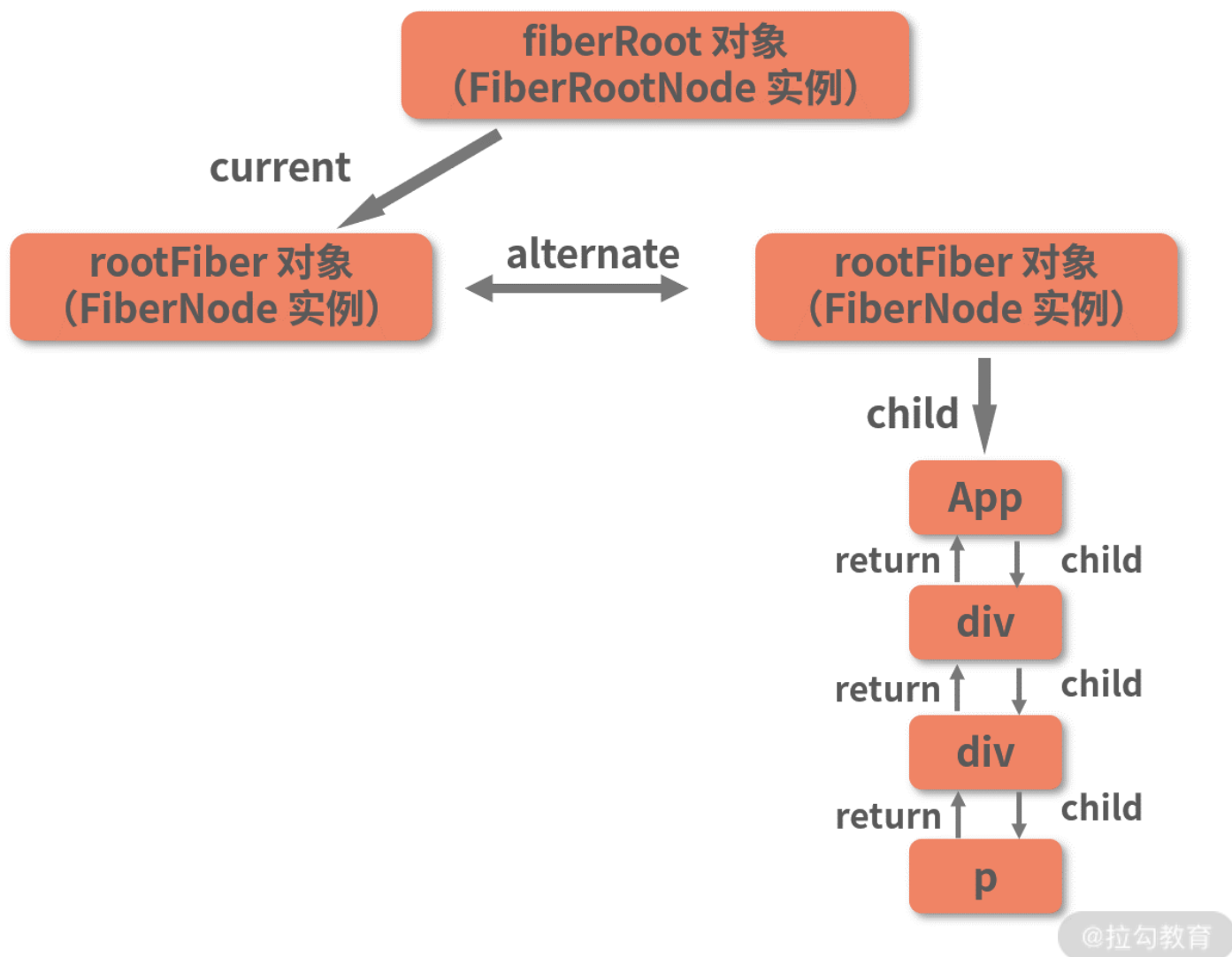
0

[@拉勾教育](#)

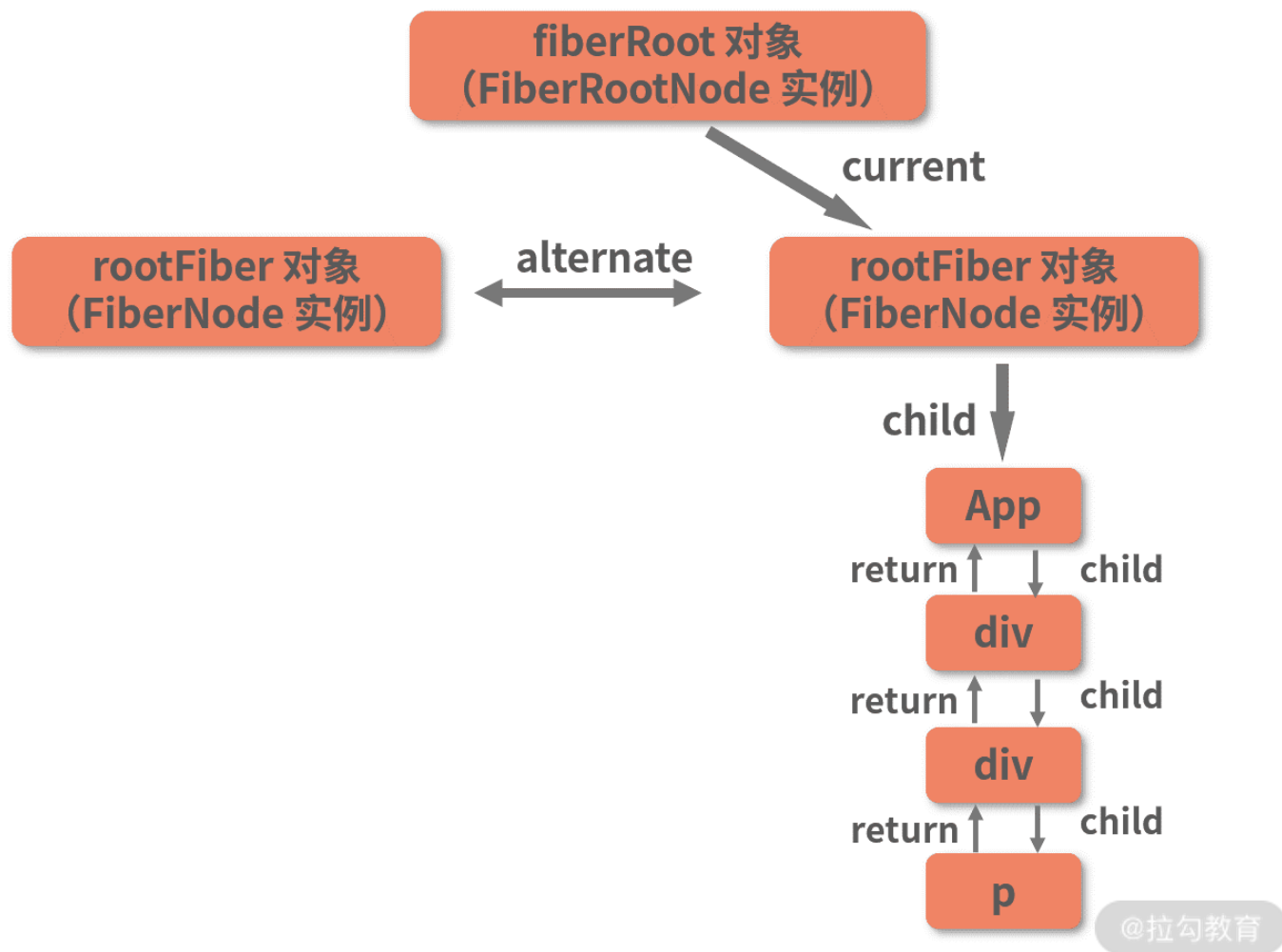
每点击数字 0 一下，它的值就会 +1，这就是我们的更新动作。

挂载后的 Fiber 树

关于 Fiber 树的构建过程，前面已经详细讲解过，这里不再重复。下面我直接为你展示挂载时的 render 阶段结束后，commit 执行前，两棵 Fiber 树的形态，如下图所示：



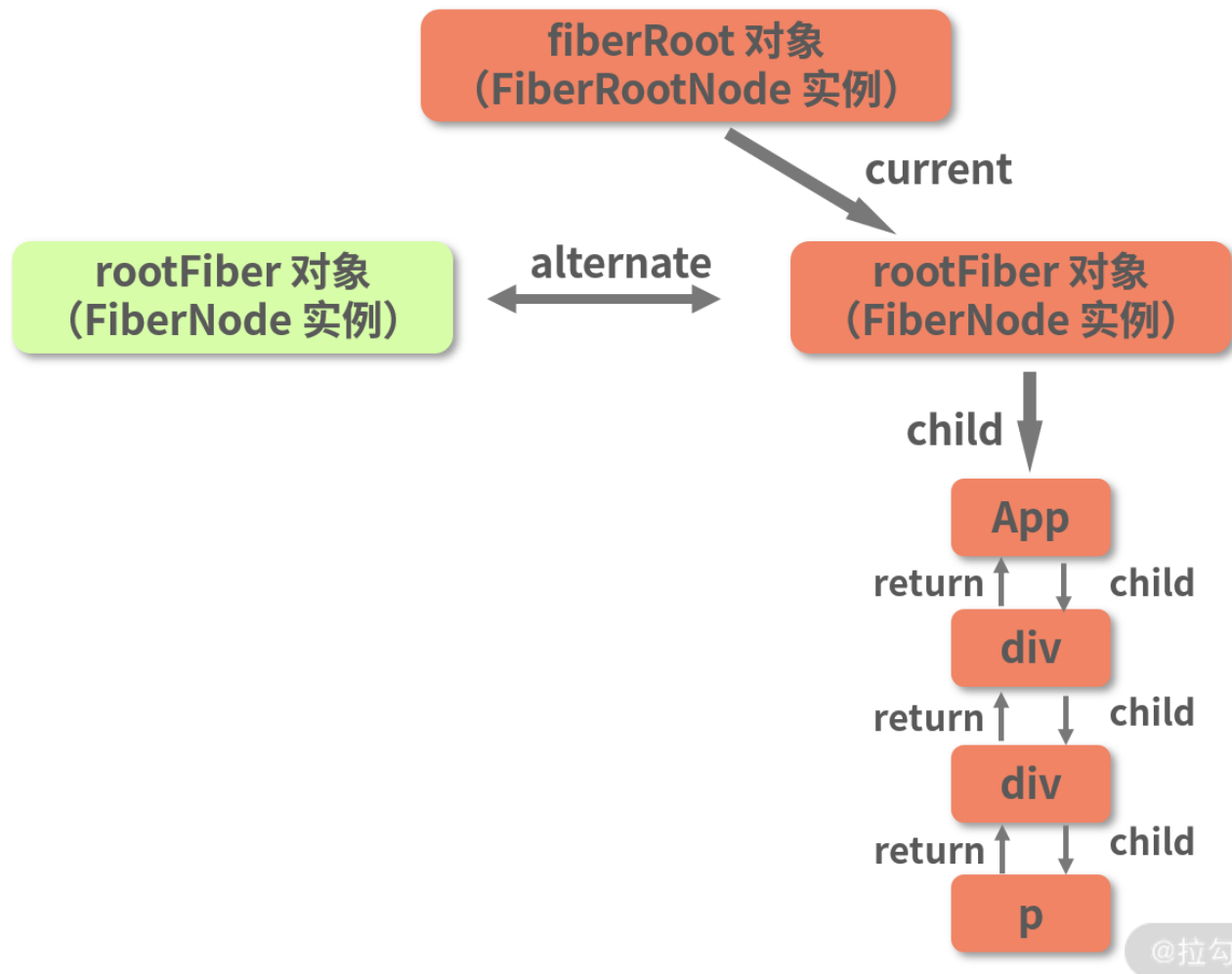
待 commit 阶段完成后，右侧的 workInProgress 树对应的 DOM 树就被真正渲染到了页面上，此时 current 指针会指向 workInProgress 树：



由于挂载是一个从无到有的过程，在这个过程中我们是在不断地创建新节点，因此还谈不上什么“节点复用”。节点复用要到更新过程中去看。

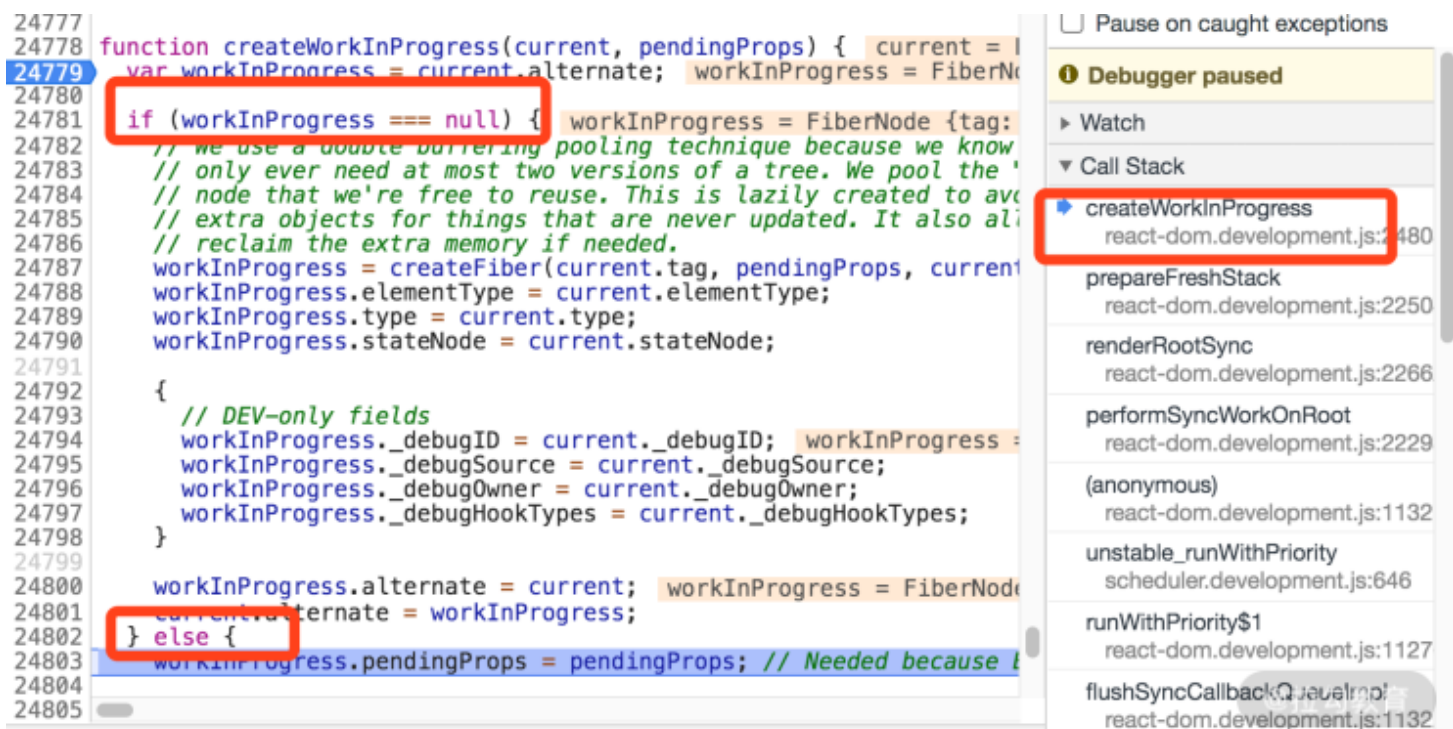
第一次更新

现在我点击数字 0，触发一次更新。这次更新中，下图高亮的 rootFiber 节点就会被复用：

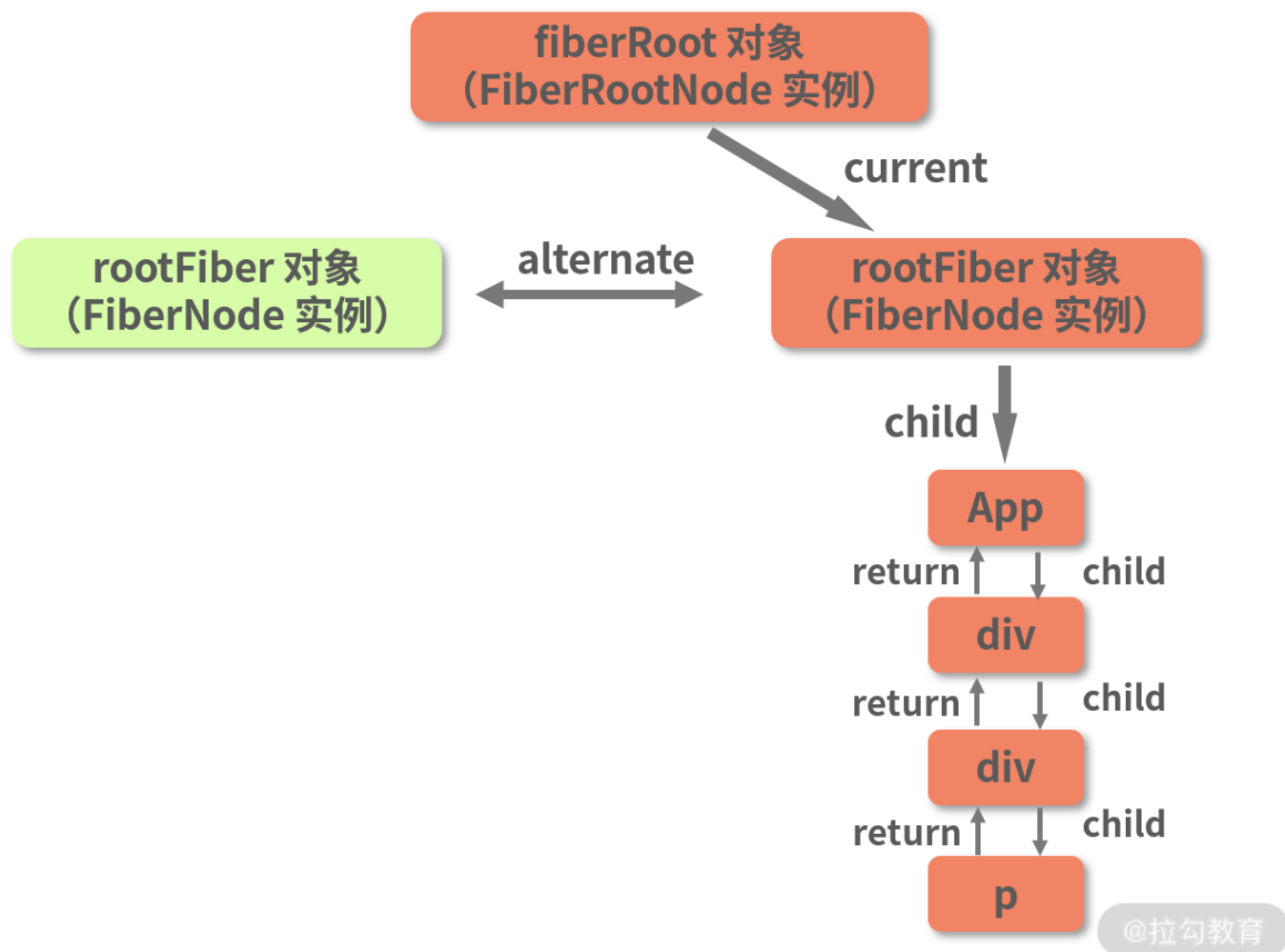


@拉勾教育

这段复用的逻辑在 `beginWork` 调用链路中的 `createWorkInProgress` 方法里。这里我为你截取了 `createWorkInProgress` 方法里面一段非常关键的逻辑，请看下图：



在 `createWorkInProgress` 方法中，会先取当前节点的 `alternate` 属性，将其记为 `workInProgress` 节点。对于 `rootFiber` 节点来说，它的 `alternate` 属性，其实就是上一棵 `current` 树的 `rootFiber`，如下图高亮部分所示：



当检查到上一棵 `current` 树的 `rootFiber` 存在时，`React` 会直接复用这个节点，让它作为下一棵 `workInProgress` 的节点存在下去，也就是说会走进 `createWorkInProgress` 的 `else` 逻辑里去。如果它和目标的 `workInProgress` 节点之间存在差异，直接在该节点上修改属性、使其与目标节点一致即可，而不必再创建新的 `Fiber` 节点。

至于剩下的 `App`、`div`、`p` 等节点，由于没有对应的 `alternate` 节点存在，因此它们的 `createWorkInProgress` 调用会走进下图高亮处的逻辑中：


```

777 function createWorkInProgress(current, pendingProps) {
778   var workInProgress = current.alternate;
779   if (workInProgress === null) {
780     // We use a double buffering/pooling technique because we know
781     // only ever need at most two versions of a tree. We pool the
782     // node that we're free to reuse. This is lazily created to avoid
783     // extra objects for things that are never updated. It also allows
784     // to reclaim the extra memory if needed.
785     workInProgress = createFiber(current.tag, pendingProps, current);
786     workInProgress.elementType = current.elementType;
787     workInProgress.type = current.type;
788     workInProgress.stateNode = current.stateNode;
789     {
790       // DEV-only fields
791       workInProgress._debugID = current._debugID;
792       workInProgress._debugSource = current._debugSource;
793       workInProgress._debugOwner = current._debugOwner;
794       workInProgress._debugHookTypes = current._debugHookTypes;
795     }
796     workInProgress.alternate = current;
797     current.alternate = workInProgress;
798   }
799   return workInProgress;
800 }
801
802 // ...
803

```

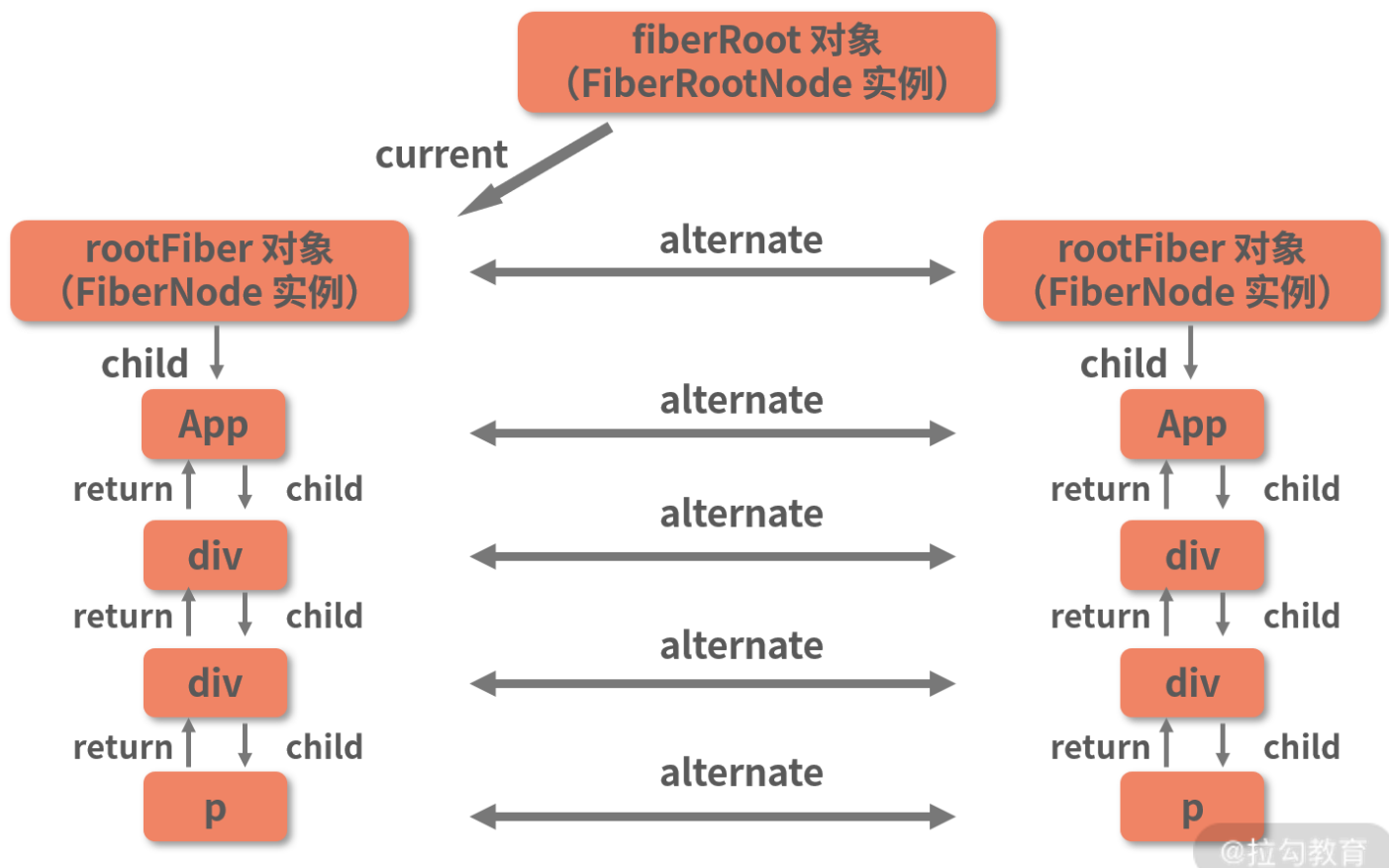
Line 24781, Column 1 (source mapped from 0.chunk.js) Coverage: n/a

Call Stack:

- createWorkInProgress (react-dom.development.js:2478)
- useFiber (react-dom.development.js:1332)
- reconcileSingleElement (react-dom.development.js:1402)
- reconcileChildFibers (react-dom.development.js:1411)
- reconcileChildren (react-dom.development.js:1699)
- updateFunctionComponent (react-dom.development.js:1737)
- beginWork (react-dom.development.js:1906)
- beginWork\$1 (react-dom.development.js:2594)
- performInitialWork

在这段逻辑里，将调用 createFiber 来新建一个 FiberNode。

第一次更新结束后，我们会得到一棵新的 workInProgress Fiber 树，current 指针最后将会指向这棵新的 workInProgress Fiber 树，如下图所示：



第二次更新

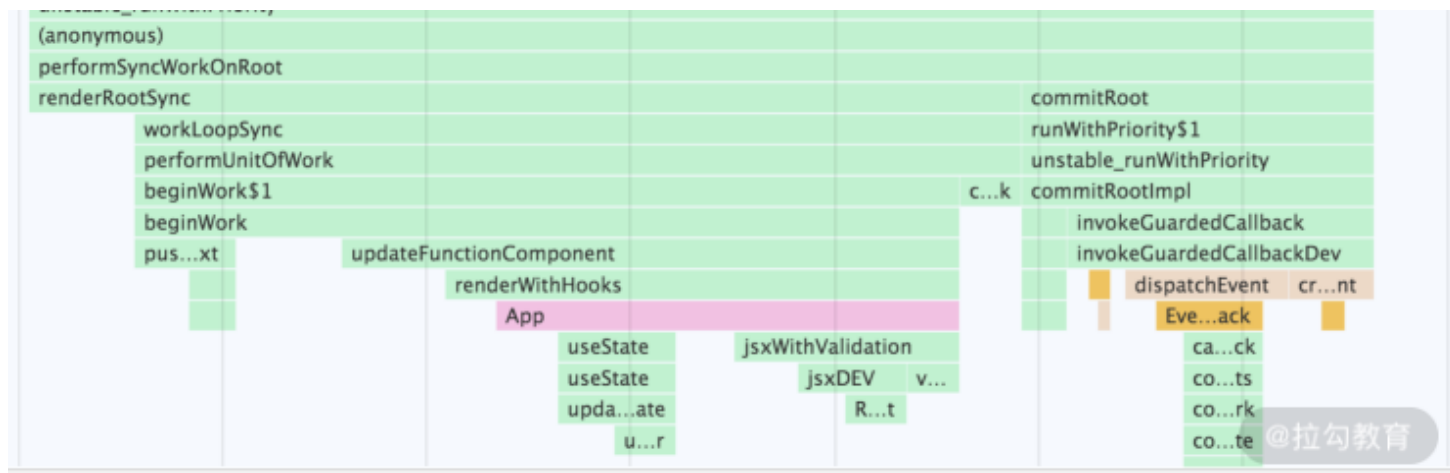
接下来我们再次点击数字 1，触发 state 的第二次更新。

在这次更新中，current 树中的每一个 alternate 属性都不为空（如上图所示）。因此每次通过 beginWork 触发 createWorkInProgress 调用时，都会一致地走入 else 里面的逻辑，也就是直接复用现成的节点。

以上便是 current 树和 work 树相互“打配合”，实现节点复用的过程。

更新链路要素拆解

在上一讲，我们已经学习了挂载阶段的渲染链路。同步模式下的更新链路与挂载链路的 render 阶段基本是一致的，都是通过 performSyncWorkOnRoot 来触发包括 beginWork、completeWork 在内的深度优先搜索过程。这里我为你展示一个更新过程的调用栈，请看下图：



你会发现还是熟悉的配方，还是原来的味道。其实，挂载可以理解作为一种特殊的更新，**ReactDOM.render** 和 **setState** 一样，也是一种触发更新的姿势。在 React 中，ReactDOM.render、setState、useState 等方法都是可以触发更新的，这些方法发起的调用链路很相似，是因为它们最后“殊途同归”，都会通过创建 **update** 对象来进入同一套更新工作流。

update 的创建

接下来我继续以开篇的 Demo 为例，为你拆解更新链路中的要素。在点击数字后，点击相关的回调被执行，它首先触发的是 **dispatchAction** 这个方法，如下图所示：

dispatchAction 中，会完成 update 对象的创建，如下图标红处所示：

```
function dispatchAction(fiber, queue, action) {
  {
    if (typeof arguments[3] === 'function') {
      error("State updates from the useState() and useReducer() Hooks don't");
    }
  }

  var eventTime = requestEventTime();
  var lane = requestUpdateLane(fiber);

  var update = {
    lane: lane,
    action: action,
    eagerReducer: null,
    eagerState: null,
    next: null
  }; // Append the update to the end of the list.

  var pending = queue.pending;

  if (pending === null) {
    // This is the first update. Create a circular list.
    update.next = update;
  } else {
    update.next = pending.next;
    pending.next = update;
  }

  queue.pending = update;
}
```

@拉勾教育

从 update 对象到 scheduleUpdateOnFiber

等等，这段逻辑你是否觉得似曾相识？如果你对 ReactDOM.render 系列的第一课时还有印象的话，我希望你能回忆起 updateContainer 这个方法。在 updateContainer 中，React 曾经有过性质一模一样的行为，这里我为你截取了 updateContainer 函数中的相关逻辑：

```
var update = createUpdate(eventTime, lane); // Caution: React
// being called "element".

update.payload = {
  element: element
};
callback = callback === undefined ? null : callback;

if (callback !== null) {
  {
    if (typeof callback !== 'function') {
      error('render(...): Expected the last optional `callback`');
    }
  }

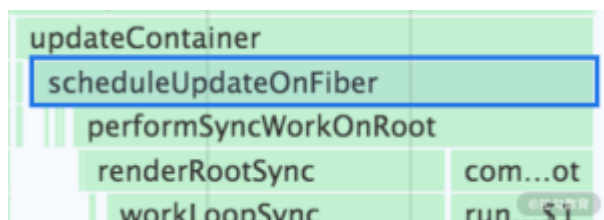
  update.callback = callback;
}

enqueueUpdate(current$1, update);
scheduleUpdateOnFiber(current$1, lane, eventTime);
```

@拉勾教育

图中这一段代码的逻辑是非常清晰的，以 `enqueueUpdate` 为界，它一共做了以下三件事。

1. `enqueueUpdate` 之前：**创建 update**。
2. `enqueueUpdate` 调用：**将 update 入队**。这里简单说下，每一个 Fiber 节点都会有一个属于它自己的 `updateQueue`，用于存储多个更新，这个 `updateQueue` 是以链表的形式存在的。在 `render` 阶段，**`updateQueue` 的内容会成为 `render` 阶段计算 Fiber 节点的新 `state` 的依据**。
3. `scheduleUpdateOnFiber`：**调度 update**。如果你对之前学过的知识还有印象，会记得同步挂载链表中，这个方法后面紧跟的就是 `performSyncWorkOnRoot` 所触发的 `render` 阶段，如下图所示：



现在再回过头来看 `dispatchAction` 的逻辑，你会发现 `dispatchAction` 里面同样有对这三个动作的处理。上面我对 `dispatchAction` 的局部截图，包含了对 `update` 对象的创建和入队处理。`dispatchAction` 的更新调度动作，在函数的末尾，如下图所示：

```
}  
  
scheduleUpdateOnFiber(fiber, lane, eventTime);  
}
```

这里有一个点需要提示一下：`dispatchAction` 中，调度的是当前触发更新的节点，这一点和挂载过程需要区分开来。在挂载过程中，`updateContainer` 会直接调度根节点。其实，对于更新这种场景来说，大部分的更新动作确实都不是由根节点触发的，而 `render` 阶段的起点则是根节点。因此在 `scheduleUpdateOnFiber` 中，有这样一个方法，见下图标红处：

```

function scheduleUpdateOnFiber(fiber, lane, eventTime) {
  checkForNestedUpdates();
  warnAboutRenderPhaseUpdatesInDEV(fiber);
  var root = markUpdateLaneFromFiberToRoot(fiber, lane);
  if (root === null) {
    warnAboutUpdateOnUnmountedFiberInDEV(fiber);
    return null;
  } // Mark that the root has a pending update.

  markRootUpdated(root, lane, eventTime);

  if (root === workInProgressRoot) {
    // Received an update to a tree that's in the middle of re-rendering.
    // that there was an interleaved update work on this root.
    // `deferRenderPhaseUpdateToNextBatch` flag is off and this is a
    // phase update. In that case, we don't treat render phase updates
    // they were interleaved, for backwards compat reasons.
    {
      workInProgressRootUpdatedLanes = mergeLanes(workInProgressRootUpdatedLanes, lane);
    }

    if (workInProgressRootExitStatus === RootSuspendedWithDelay) {
      // The root already suspended with a delay, which means
      // definitely won't finish. Since we have a new update,
      // suspended now, right before marking the incoming update,
      // effect of interrupting the current render and switching to
      // TODO: Make sure this doesn't override pings that happen
      // already started rendering.
      markRootSuspended$1(root, workInProgressRootRenderLanes);
    }
  }
}

```

`markUpdateLaneFromFiberToRoot` 将会从当前 Fiber 节点开始，向上遍历直至根节点，并将根节点返回。

`scheduleUpdateOnFiber` 如何区分同步还是异步？

如果你对之前学过的同步渲染链路分析还有印象，相信你对下面这段逻辑不会陌生：


```

if (lane === SyncLane) {
  if ( // Check if we're inside unbatchedUpdates
    (executionContext & LegacyUnbatchedContext) !== NoContext &
    (executionContext & (RenderContext | CommitContext)) === NoContext) {
    // Register pending interactions on the root to avoid losing them
    schedulePendingInteractions(root, lane); // This is a legacy behavior
    // root inside of batchedUpdates should be synchronous, i.e.
    // should be deferred until the end of the batch.
    performSyncWorkOnRoot(root);
  } else {
    ensureRootIsScheduled(root, eventTime);
    schedulePendingInteractions(root, lane);

    if (executionContext === NoContext) {
      // Flush the synchronous work now, unless we're already in
      // a batch. This is intentionally inside scheduleUpdateBecause
      // scheduleCallbackForFiber to preserve the ability to
      // without immediately flushing it. We only do this for
      // updates, to preserve historical behavior of legacy
      // updates.
      resetRenderTimer();
      flushSyncCallbackQueue();
    }
  }
} else {
  // Schedule a discrete update but only if it's not Sync

```

@拉勾教育

这是 `scheduleUpdateOnFiber` 中的一段逻辑。在同步的渲染链路中，`lane === SyncLane` 这个条件是成立的，因此会直接进入 `performSyncWorkOnRoot` 的逻辑，开启同步的 `render` 流程；而在异步渲染模式下，则将进入 `else` 的逻辑。

在 `else` 中，需要引起你注意的是 `ensureRootIsScheduled` 这个方法，该方法很关键，它将决定如何开启当前更新所对应的 `render` 阶段。在 `ensureRootIsScheduled` 中，有[这样一段核心逻辑](#)（解析在注释里）：

```

1. if (newCallbackPriority === SyncLanePriority) {
2.   // 同步更新的 render 入口
3.   newCallbackNode = scheduleSyncCallback(performanceSyncWorkOnRoot.bind(null, root));
4. } else {
5.   // 将当前任务的 lane 优先级转换为 scheduler 可理解的优先级
6.   var schedulerPriorityLevel = lanePriorityToSchedulerPriority(newCallbackPriority);
7.   // 异步更新的 render 入口
8.   newCallbackNode = scheduleCallback(schedulerPriorityLevel, performanceConcurrentWorkOnRoot);
9. }

```

■ 复制代码

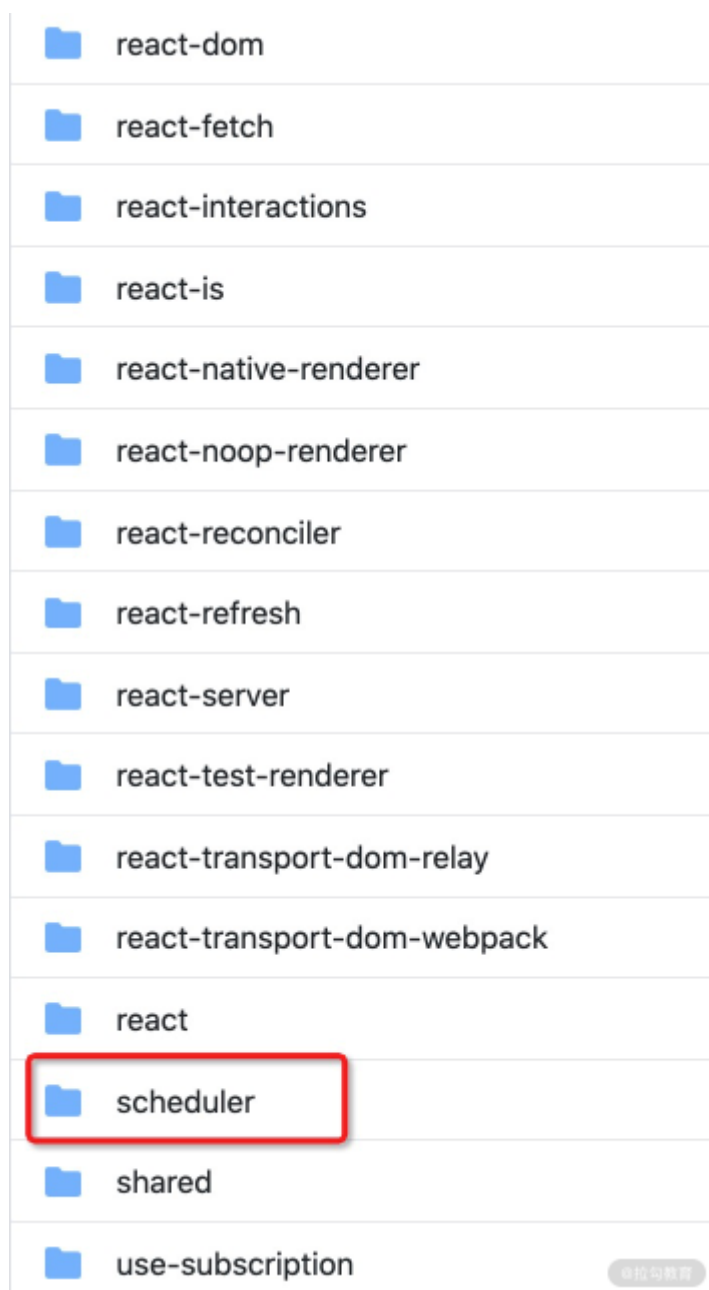
请你关注 `performSyncWorkOnRoot` 和 `performConcurrentWorkOnRoot` 这两个方法：前者是同步更新模式下的 `render` 阶段入口；而后者是异步模式下的 `render` 阶段入口。

从这段逻辑中我们可以看出，React 会以当前更新任务的优先级类型为依据，决定接下来是调度 `performSyncWorkOnRoot` 还是 `performConcurrentWorkOnRoot`。这里调度任务用到的函数分别是 `scheduleSyncCallback` 和 `scheduleCallback`，这两个函数在内部都是通过调用 **`unstable_scheduleCallback`** 方法来执行任务调度的。而 `unstable_scheduleCallback` 正是 Scheduler（调度器）中导出的一个核心方法，也是本讲的一个重点。

在解读 `unstable_scheduleCallback` 的工作原理之前，我们先来一起认识一下 Scheduler。

Scheduler——“时间切片”与“优先级”的幕后推手

Scheduler 从架构上来看，是 Fiber 架构分层中的“调度层”；从实现上来看，它并非一段内嵌的逻辑，而是一个与 `react-dom` 同级的文件夹，如下图所示，其中收敛了所有相对通用的调度逻辑：



通过前面的学习，我们已经知道 Fiber 架构下的异步渲染（即 Concurrent 模式）的核心特征分别是“时间切片”与“优先级调度”。而这两点，也正是 Scheduler 的核心能力。接下来，我们就以这两个特征为线索，解锁 Scheduler 的工作原理。

结合 React 调用栈，理解时间切片现象

在理解时间切片的实现原理之前，我们首先要搞清楚时间切片是一种什么样的现象。

在 ReactDOM.render 相关的课时中，我曾经强调过，同步渲染模式下的 render 阶段，是一个同步的、深度优先搜索的过程。同步的过程会带来什么样的麻烦呢？在第 12 讲中，大家已经从理论层面初步认识过这个问题。现在，我们直接通过调用栈来理解它，下面是一个渲染工作量相对比较大的 React Demo，代码如下：

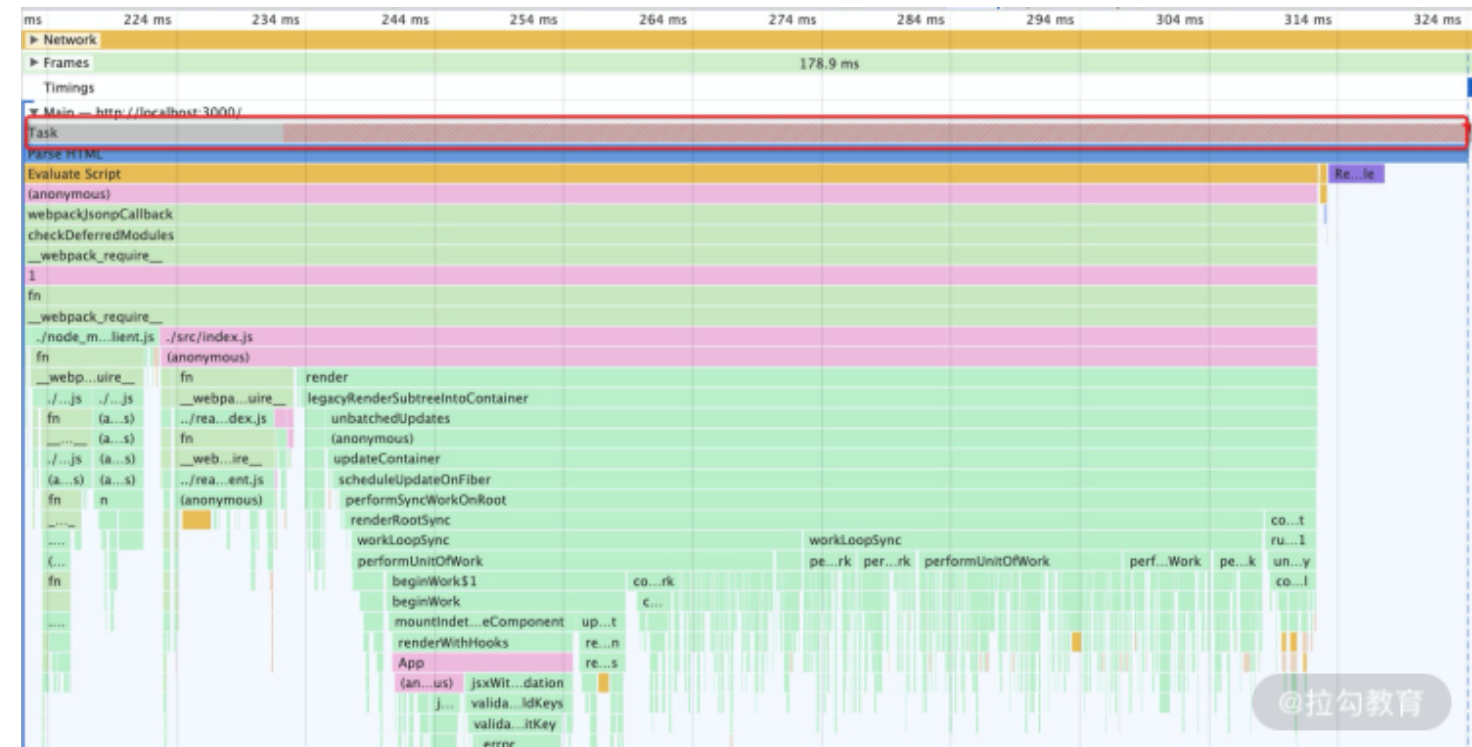
复制代码

```
1. import React from 'react';
2. function App() {
3.   const arr = new Array(1000).fill(0)
4.   const renderContent = arr.map(
5.     (i, index) => <p style={{ width: 128, textAlign: 'center' }}>`测试文本第${index}行`</p>
6.   )
7.   return (
8.     <div className="App">
9.       <div className="container">
10.        {
11.          renderContent
12.        }
13.      </div>
14.    </div>
15.  );
16. }
17. export default App;
```

这个 App 组件会在界面上渲染出 1000 行文本，局部效果如下图所示：

测试文本第0行
测试文本第1行
测试文本第2行
测试文本第3行
测试文本第4行
测试文本第5行
测试文本第6行
测试文本第7行
测试文本第8行
测试文本第9行
测试文本第10行
测试文本第11行
测试文本第12行
测试文本第13行
测试文本第14行
测试文本第15行
测试文本第16行
测试文本第17行
测试文本第18行
测试文本第19行
测试文本第20行
测试文本第21行

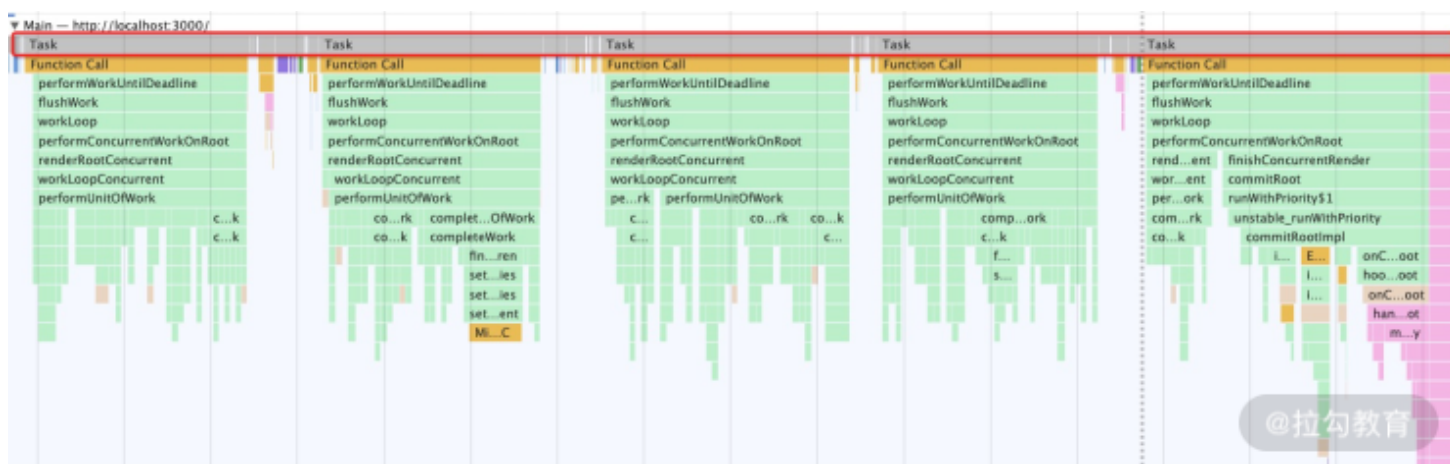
当我使用 ReactDOM.render 来渲染这个长列表时，它的调用栈如下图所示：



在这张图中，你就不必再重复去关注 beginWork、completeWork 之流了，请把目光放在调用栈的上层，也就是图中标红的地方——一个不间断的灰色“Task”长条，对浏览器来说就意味着是一个不可中断的任务。

在我的浏览器上，这个 Task 的执行时长在 130ms 以上（将鼠标悬浮在 Task 长条上就可以查看执行时长）。而浏览器的刷新频率为 60Hz，也就是说每 16.6ms 就会刷新一次。在这 16.6ms 里，除了 JS 线程外，渲染线程也是有工作要处理的，但超长的 Task 显然会挤占渲染线程的工作时间，引起“掉帧”，进而带来卡顿的风险，这也正是第 12 讲中所提到的“JS 对主线程的超时占用”问题。

若将 ReactDOM.render 调用改为 createRoot 调用（即开启 Concurrent 模式），调用栈就会变成下面这样：



请继续将你的注意力放在顶层的 Task 长条上。

你会发现那一个不间断的 Task 长条（大任务），如今像是被“切”过了一样，已经变成了多个断断续续的 Task “短条”（小任务），单个短 Task 的执行时长在我的浏览器中是 5ms 左右。这些短 Task 的工作量加起来，和之前长 Task 工作量是一样的。但短 Task 之间留出的时间缝隙，却给了浏览器喘息的机会，这就是所谓的“时间切片”效果。

时间切片是如何实现的？

在同步渲染中，循环创建 Fiber 节点、构建 Fiber 树的过程是由 **workLoopSync** 函数来触发的。这里我们来复习一下 workLoopSync 的源码，请看下图：

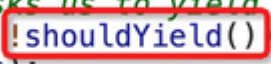
```
function workLoopSync() {  
  // Already timed out, so perform work without  
  while (workInProgress !== null) {  
    performUnitOfWork(workInProgress);  
  }  
}
```

@拉勾教育

在 workLoopSync 中，只要 workInProgress 不为空，while 循环就不会结束，它所触发的是一个同步的 performUnitOfWork 循环调用过程。

而在异步渲染模式下，这个循环是由 **workLoopConcurrent** 来开启的。workLoopConcurrent 的工作内容和 workLoopSync 非常相似，仅仅在循环判断上有一处不同，请注意下图源码中标红部分：

```
function workLoopConcurrent() {  
  // Perform work until Scheduler asks us to yield  
  while (workInProgress !== null && !shouldYield()) {  
    performUnitOfWork(workInProgress);  
  }  
}
```



@拉勾教育

shouldYield 直译过来的话是“需要让出”。顾名思义，当 **shouldYield()** 调用返回为 **true** 时，就说明当前需要对主线程进行让出了，此时 while 循环的判断条件整体为 **false**，while 循环将不再继续。

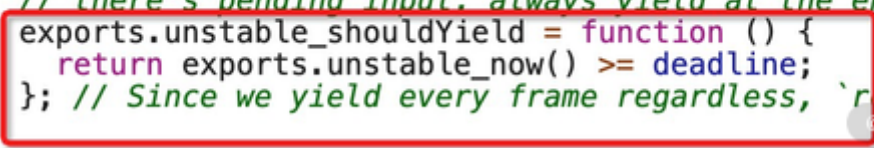
那么这个 shouldYield 又是何方神圣呢？在源码中，我们可以寻得这样两行赋值语句：

```
1. var Scheduler_shouldYield = Scheduler.unstable_shouldYield,  
2. ....  
3. var shouldYield = Scheduler_shouldYield;
```

 复制代码

从这两行代码中我们可以看出，shouldYield 的本体其实是 **Scheduler.unstable_shouldYield**，也就是 Scheduler 包中导出的 unstable_shouldYield 方法，该方法本身比较简单。其源码如下图标红处所示：

```
{  
  // `isInputPending` is not available. Since we have no  
  // there's pending input, always yield at the end of t  
  exports.unstable_shouldYield = function () {  
    return exports.unstable_now() >= deadline;  
  }; // Since we yield every frame regardless, `requestP
```



@拉勾教育

其中 unstable_now 这里实际取的就是 performance.now() 的值，即“当前时间”。那么 deadline 又是什么呢？它可以被理解为当前时间切片的到期时间，它的计算过程在 Scheduler 包中的 performWorkUntilDeadline 方法里可以找到，也就是下图的标红部分：

```
var performWorkUntilDeadline = function () {
  if (scheduledHostCallback !== null) {
    var currentTime = exports.unstable_now(); // Yield after `
    // cycle. This means there's always time remaining at the
    // the message event.

    deadline = currentTime + yieldInterval;
    var hasTimeRemaining = true;

    try {
      var hasMoreWork = scheduledHostCallback(hasTimeRemaining
        if (!hasMoreWork) {
          isMessageLoopRunning = false;
        }
      }
    }
  }
}
```

@拉勾教育

在这行算式里，currentTime 是当前时间，yieldInterval 是时间切片的长度。注意，时间切片的长度并不是一个常量，它是由 React 根据浏览器的帧率大小计算所得出来的，与浏览器的性能有关。

现在我们来总结一下时间切片的实现原理：React 会根据浏览器的帧率，计算出时间切片的大小，并结合当前时间计算出每一个切片的到期时间。在 workLoopConcurrent 中，while 循环每次执行前，会调用 shouldYield 函数来询问当前时间切片是否到期，若已到期，则结束循环、出让主线程的控制权。

优先级调度是如何实现的

在“更新链路要素拆解”这一小节的末尾，我们已经知道，无论是 scheduleSyncCallback 还是 scheduleCallback，最终都是通过调用 **unstable_scheduleCallback** 来发起调度的。unstable_scheduleCallback 是 Scheduler 导出的一个核心方法，它将结合任务的优先级信息为其执行不同的调度逻辑。

接下来我们就结合源码，一起来看看这个过程是如何实现的（解析在注释里）。

```
1. function unstable_scheduleCallback(priorityLevel, callback, options) { ■ 复制代码
2.   // 获取当前时间
3.   var currentTime = exports.unstable_now();
4.   // 声明 startTime, startTime 是任务的预期开始时间
5.   var startTime;
6.   // 以下是对 options 入参的处理
7.   if (typeof options === 'object' && options !== null) {
8.     var delay = options.delay;
9.
10.    // 若入参规定了延迟时间，则累加延迟时间
11.    if (typeof delay === 'number' && delay > 0) {
12.      startTime = currentTime + delay;
13.    } else {
14.      startTime = currentTime;
15.    }
16.  } else {
17.    startTime = currentTime;
18.  }
19.  // timeout 是 expirationTime 的计算依据
```

```
20. var timeout;
21. // 根据 priorityLevel, 确定 timeout 的值
22. switch (priorityLevel) {
23.   case ImmediatePriority:
24.     timeout = IMMEDIATE_PRIORITY_TIMEOUT;
25.     break;
26.   case UserBlockingPriority:
27.     timeout = USER_BLOCKING_PRIORITY_TIMEOUT;
28.     break;
29.   case IdlePriority:
30.     timeout = IDLE_PRIORITY_TIMEOUT;
31.     break;
32.   case LowPriority:
33.     timeout = LOW_PRIORITY_TIMEOUT;
34.     break;
35.   case NormalPriority:
36.   default:
37.     timeout = NORMAL_PRIORITY_TIMEOUT;
38.     break;
39. }
40. // 优先级越高, timeout 越小, expirationTime 越小
41. var expirationTime = startTime + timeout;
42.
43. // 创建 task 对象
44. var newTask = {
45.   id: taskIdCounter++,
46.   callback: callback,
47.   priorityLevel: priorityLevel,
48.   startTime: startTime,
49.   expirationTime: expirationTime,
50.   sortIndex: -1
51. };
52.
53. {
54.   newTask.isQueued = false;
55. }
56. // 若当前时间小于开始时间, 说明该任务可延时执行(未过期)
57. if (startTime > currentTime) {
58.   // 将未过期任务推入 "timerQueue"
59.   newTask.sortIndex = startTime;
60.   push(timerQueue, newTask);
61.
62.   // 若 taskQueue 中没有可执行的任务, 而当前任务又是 timerQueue 中的第一个任务
63.   if (peek(taskQueue) === null && newTask === peek(timerQueue)) {
64.     .....
65.     // 那么就派发一个延时任务, 这个延时任务用于检查当前任务是否过期
66.     requestHostTimeout(handleTimeout, startTime - currentTime);
67.   }
68. } else {
69.   // else 里处理的是当前时间大于 startTime 的情况, 说明这个任务已过期
70.   newTask.sortIndex = expirationTime;
71.   // 过期的任务会被推入 taskQueue
72.   push(taskQueue, newTask);
73.   .....
74.
75.   // 执行 taskQueue 中的任务
76.   requestHostCallback(flushWork);
77. }
```



```
78.   return newTask;  
79. }
```

从源码中我们可以看出，`unstable_scheduleCallback` 的主要工作是针对当前任务创建一个 `task`，然后结合 `startTime` 信息将这个 `task` 推入 `timerQueue` 或 `taskQueue`，最后根据 `timerQueue` 和 `taskQueue` 的情况，执行延时任务或即时任务。

要想理解这个过程，首先要搞清楚以下几个概念。

- **startTime**：任务的开始时间。
- **expirationTime**：这是一个和优先级相关的值，`expirationTime` 越小，任务的优先级就越高。
- **timerQueue**：一个以 `startTime` 为排序依据的小顶堆，它存储的是 `startTime` 大于当前时间（也就是待执行）的任务。
- **taskQueue**：一个以 `expirationTime` 为排序依据的小顶堆，它存储的是 `startTime` 小于当前时间（也就是已过期）的任务。

这里的“小顶堆”概念可能会触及一部分同学的知识盲区，我简单解释下：堆是一种特殊的[完全二叉树](#)。如果对一棵完全二叉树来说，它每个结点的结点值都不大于其左右孩子的结点值，这样的完全二叉树就叫“[小顶堆](#)”。小顶堆自身特有的插入和删除逻辑，决定了无论我们怎么增删小顶堆的元素，其根节点一定是所有元素中值最小的一个节点。这样的性质，使得小顶堆经常被用于实现[优先队列](#)。

结合小顶堆的特性，我们再来看源码中涉及 `timerQueue` 和 `taskQueue` 的操作，这段代码同时也是整个 `unstable_scheduleCallback` 方法中的核心逻辑：

```
1. // 若当前时间小于开始时间，说明该任务可延时执行(未过期)  
2. if (startTime > currentTime) {  
3.   // 将未过期任务推入 "timerQueue"  
4.   newTask.sortIndex = startTime;  
5.   push(timerQueue, newTask);  
6.  
7.   // 若 taskQueue 中没有可执行的任务，而当前任务又是 timerQueue 中的第一个任务  
8.   if (peek(taskQueue) === null && newTask === peek(timerQueue)) {  
9.     .....  
10.    // 那么就派发一个延时任务，这个延时任务用于将过期的 task 加入 taskQueue 队列  
11.    requestHostTimeout(handleTimeout, startTime - currentTime);  
12.  }  
13. } else {  
14.   // else 里处理的是当前时间大于 startTime 的情况，说明这个任务已过期  
15.   newTask.sortIndex = expirationTime;  
16.   // 过期的任务会被推入 taskQueue
```

[复制代码](#)


```
17.     push(taskQueue, newTask);
18.     .....
19.
20.     // 执行 taskQueue 中的任务
21.     requestHostCallback(flushWork);
22. }
```

若判断当前任务是待执行任务，那么该任务会在 `sortIndex` 属性被赋值为 `startTime` 后，被推入 `timerQueue`。随后，会进入这样的一段判断逻辑：

```
1. // 若 taskQueue 中没有可执行的任务，而当前任务又是 timerQueue 中的第一个任务
2. if (peek(taskQueue) === null && newTask === peek(timerQueue)) {
3.     .....
4.     // 那么就派发一个延时任务，这个延时任务用于将过期的 task 加入 taskQueue 队列
5.     requestHostTimeout(handleTimeout, startTime - currentTime);
6. }
```

[复制代码](#)

要理解这段逻辑，首先需要理解 `peek(xxx)` 做了什么：`peek()` 的入参是一个小顶堆，它将取出这个小顶堆的堆顶元素。

`taskQueue` 里存储的是已过期的任务，`peek(taskQueue)` 取出的任务若为空，则说明 `taskQueue` 为空、当前并没有已过期任务。在没有已过期任务的情况下，会进一步判断 `timerQueue`，也就是未过期任务队列里的情况。

而通过前面的科普，大家已经知道了小顶堆是一个相对有序的数据结构。`timerQueue` 作为一个小顶堆，它的排序依据其实正是 `sortIndex` 属性的大小。这里的 `sortIndex` 属性取值为 `startTime`，意味着小顶堆的堆顶任务一定是整个 `timerQueue` 堆结构里 `startTime` 最小的任务，也就是需要最早被执行的未过期任务。

若当前任务（`newTask`）就是 `timerQueue` 中需要最早被执行的未过期任务，那么 `unstable_scheduleCallback` 会通过调用 `requestHostTimeout`，为当前任务发起一个延时调用。

注意，这个延时调用（也就是 `handleTimeout`）并不会直接调度执行当前任务——它的作用是在当前任务到期后，将其从 `timerQueue` 中取出，加入 `taskQueue` 中，然后触发对 `flushWork` 的调用。真正的调度执行过程是在 `flushWork` 中进行的。`flushWork` 中将调用 `workLoop`，`workLoop` 会逐一执行 `taskQueue` 中的任务，直到调度过程被暂停（时间片用尽）或任务全部被清空。

以上便是针对未过期任务的处理。在这个基础上，我们不难理解 `else` 中，对过期任务的处理逻辑（也就是下面这段代码）：

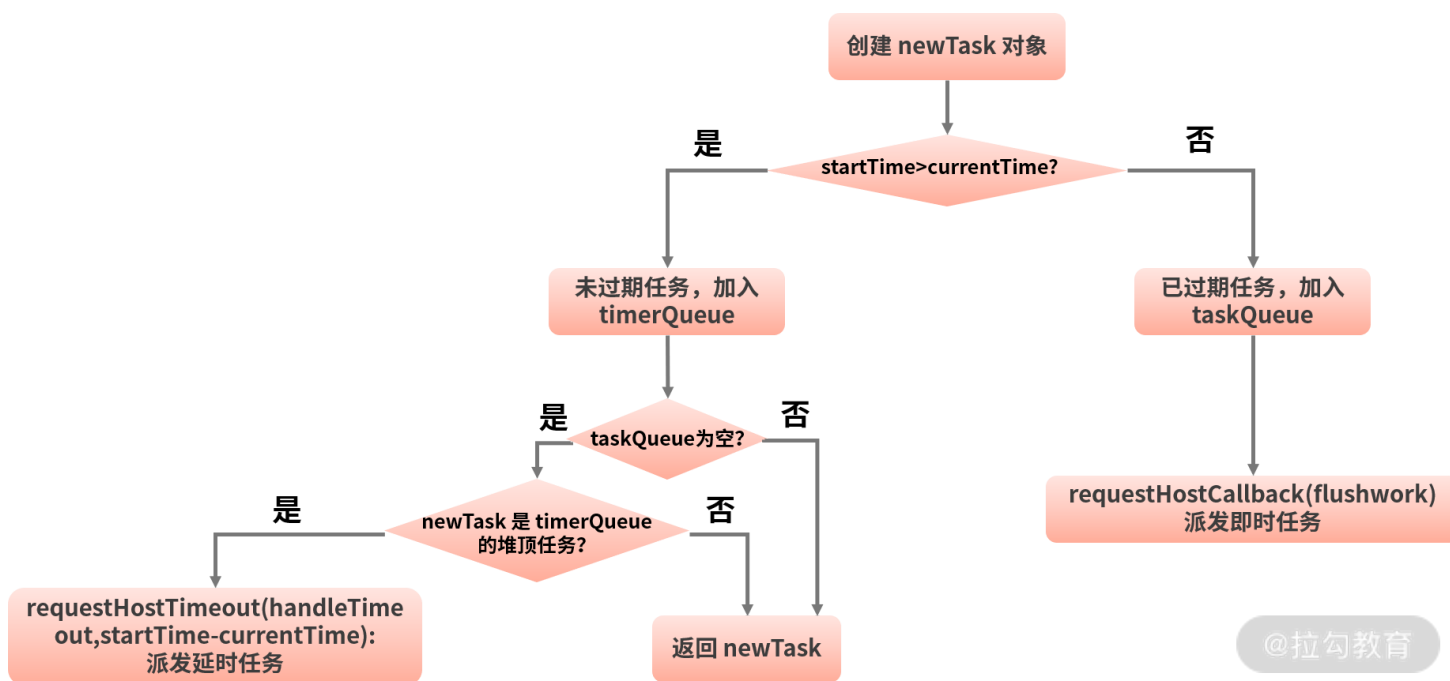
```
1. {  
2.   // else 里处理的是当前时间大于 startTime 的情况，说明这个任务已过期  
3.   newTask.sortIndex = expirationTime;  
4.   // 过期的任务会被推入 taskQueue  
5.   push(taskQueue, newTask);  
6.   .....  
7.   // 执行 taskQueue 中的任务  
8.   requestHostCallback(flushWork);  
9. }
```

与 timerQueue 不同的是，taskQueue 是一个以 expirationTime 为 sortIndex（排序依据）的小顶堆。对于已过期任务，React 在将其推入 taskQueue 后，会通过 requestHostCallback(flushWork) 发起一个针对 flushWork 的即时任务，而 flushWork 会执行 taskQueue 中过期的任务。

从 React 17.0.0 源码来看，当下 React 发起 Task 调度的姿势有两个：**setTimeout**、**MessageChannel**。在宿主环境不支持 MessageChannel 的情况下，会降级到 setTimeout。但不管是 setTimeout 还是 MessageChannel，它们发起的都是**异步任务**。

因此 requestHostCallback 发起的“即时任务”最早也要等到下一次事件循环才能够执行。“即时”仅仅意味它相对于“延时任务”来说，不需要等待指定的时间间隔，并不意味着同步调用。

这里为了方便大家理解，我将 unstable_scheduleCallback 方法的工作流总结进一张大图：



@拉勾教育

这张大图需要结合楼上的文字解析一起消化，如果你是跳读至此，还请回到文章中细嚼慢咽~^_^

总结

这一讲我们首先认识了“双缓存”模式在 Fiber 架构下的实现，接着对更新链路的种种要素进行了拆解，理解了挂载 / 更新等动作的本质。最后，我们结合源码对 Scheduler（调度器）的核心能力，也就是“时间切片”和“优先级调度”两个方面进行了剖析，最终揭开了 Fiber 架构异步渲染的神秘面纱，理解了 Concurrent 模式背后的实现逻辑。

到这里，关于 Fiber 架构的探讨，就要告一段落了。下一讲将讲解“特别的事件系统：React 事件与 DOM 事件有何不同”，到时见~