14 | ReactDOM.render 是如何串联渲染链路的？（中）

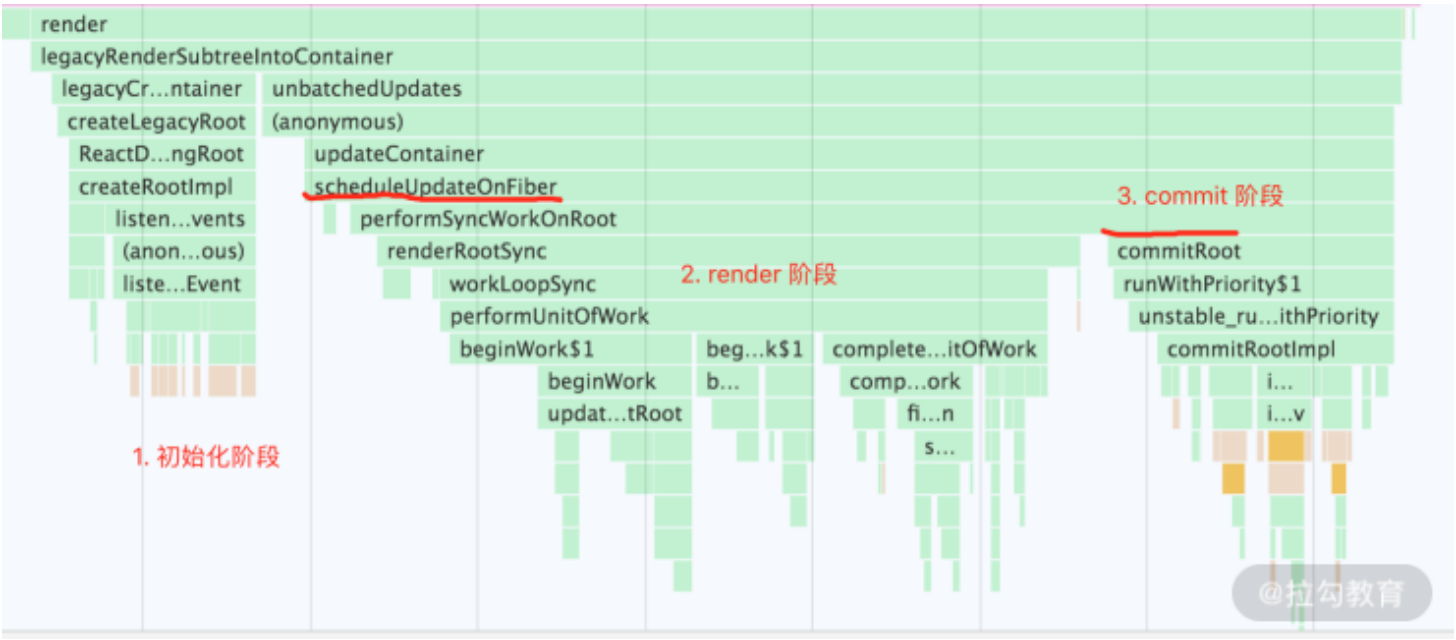
2020/11/25 修言

上一讲我们对 ReactDOM.render 的调用链路、包括其对应的初始化阶段的工作内容都有了学习和掌握。这一讲我们在此基础上，学习后续的 render 阶段和 commit 阶段。这其中，render 阶段可以认为是整个渲染链路中最为核心的一环，因为我们反复强调“找不同”的过程，恰恰就是在这个阶段发生的。

render 阶段做的事情有很多，这一讲我们将以 beginWork 为线索，着重探讨 Fiber 树的构建过程。

拆解 ReactDOM.render 调用栈——render 阶段

首先，我们复习一下 render 阶段在整个渲染链路中的定位，如下图所示。



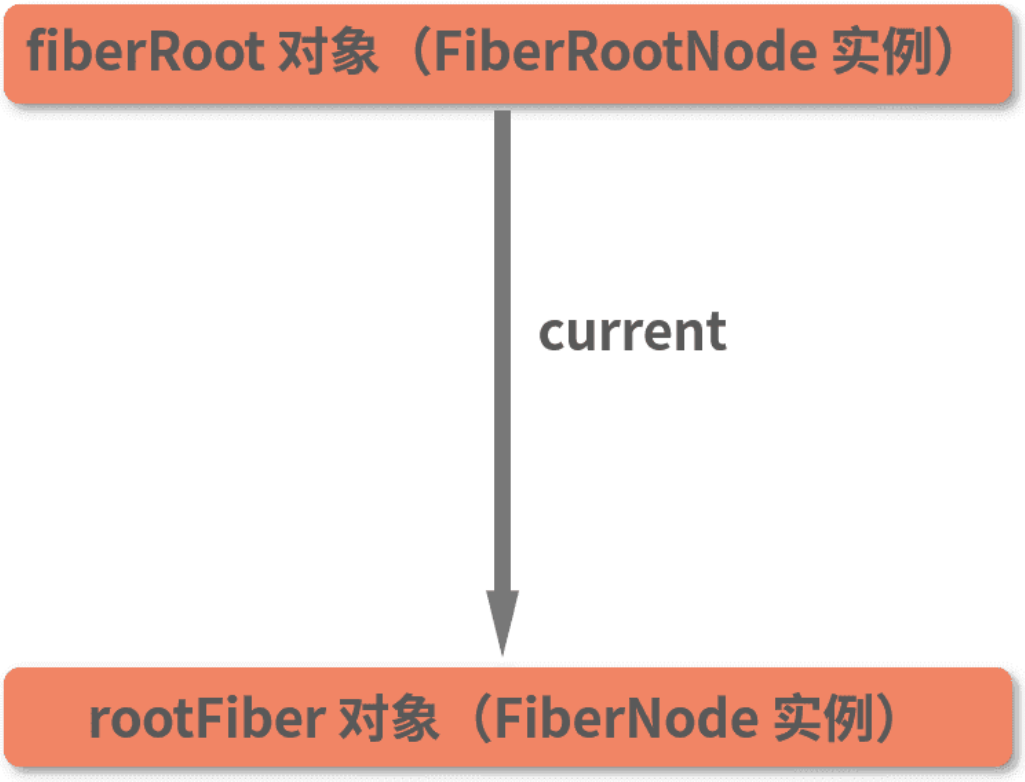
图中，performSyncWorkOnRoot 标志着 render 阶段的开始，finishSyncRender 标志着 render 阶段的结束。这中间包含了大量的 beginWork、completeWork 调用栈，正是 render 的工作内容。

> beginWork、completeWork 这两个方法需要注意，它们串联起的是一个“模拟递归”的过程。

在第 10 讲“栈调和”中强调过，React 15 下的调和过程**是一个递归的过程**。而 Fiber 架构下的调和过程，虽然并不是依赖递归来实现的，**但在 ReactDOM.render 触发的同步模式下，它仍然是一个深度优**

先搜索的过程。在这个过程中，**beginWork 将创建新的 Fiber 节点，而 completeWork 则负责将 Fiber 节点映射为 DOM 节点**。

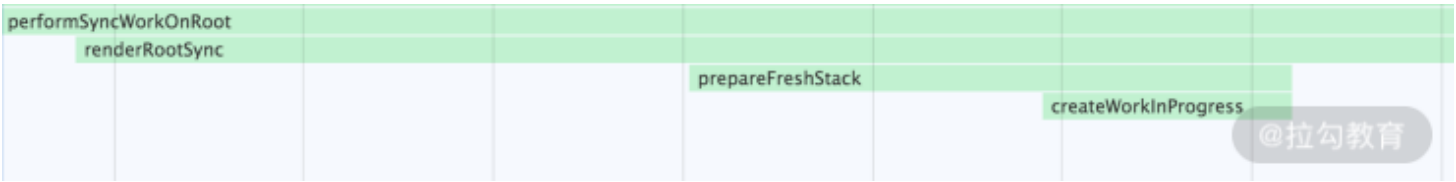
那么问题就来了：截止到上一讲，我们的 Fiber 树都还长这个样子：



就这么个样子，你遍历它，能遍历出来什么？到底怎么个遍历法？接下来我们就深入到源码里去一探究竟！

workInProgress 节点的创建

上一讲曾经提到，performSyncWorkOnRoot 是 render 阶段的起点，而这个函数最关键的地方在于它调用了 renderRootSync。下面我们放大 Performance 调用栈，来看看 renderRootSync 被调用后，紧接着发生了什么：



紧随其后的是 prepareFreshStack，这里不卖关子，prepareFreshStack 的作用是重置一个新的堆栈环境，其中最需要我们关注的步骤，就是对 **createWorkInProgress** 的调用。以下我对

createWorkInProgress 的主要逻辑进行了提取（解析在注释里）：

```
// 这里入参中的 current 传入的是现有树结构中的 rootFiber 对象
function createWorkInProgress(current, pendingProps) {
  var workInProgress = current.alternate;
  // ReactDOM.render 触发的首屏渲染将进入这个逻辑
  if (workInProgress === null) {
    // 这是需要你关注的第一个点，workInProgress 是 createFiber 方法的返回值
    workInProgress = createFiber(current.tag, pendingProps, current.key, current
    workInProgress.elementType = current.elementType;
    workInProgress.type = current.type;
    workInProgress.stateNode = current.stateNode;
    // 这是需要你关注的第二个点，workInProgress 的 alternate 将指向 current
    workInProgress.alternate = current;
    // 这是需要你关注的第三个点，current 的 alternate 将反过来指向 workInProgress
    current.alternate = workInProgress;
  } else {
    // else 的逻辑此处先不用关注
  }

  // 以下省略大量 workInProgress 对象的属性处理逻辑
  // 返回 workInProgress 节点
  return workInProgress;
}
```

复制代码

首先要声明的是，该函数中的 current 入参指的是现有树结构中的 rootFiber 对象，如下图所示：

```
> current
< ▾FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: FiberRootNode, …}
    actualDuration: 0
    actualStartTime: -1
    alternate: null
    child: null
    childLanes: 0
    dependencies: null
    elementType: null
    firstEffect: null
    flags: 0
    index: 0
    key: null
    lanes: 1
    lastEffect: null
    memoizedProps: null
    memoizedState: null
    mode: 8
    nextEffect: null
    pendingProps: null
    ref: null
    return: null
```


源码太长（其实经过处理已经不长了）不看版的重点如下：

- createWorkInProgress 将**调用 createFiber**，workInProgress**是 createFiber 方法的返回值**；
- workInProgress 的 **alternate 将指向 current**；

- **current 的 alternate 将反过来指向 workInProgress。**

理解了这三点，你就会自然而然地想知道 workInProgress 的本体到底是什么样的，也就是**createFiber 到底会返回什么**。下面我们就看看 createFiber 的逻辑：

```
1. var createFiber = function (tag, pendingProps, key, mode) {
2.
3.   return new FiberNode(tag, pendingProps, key, mode);
4. };
```

复制代码

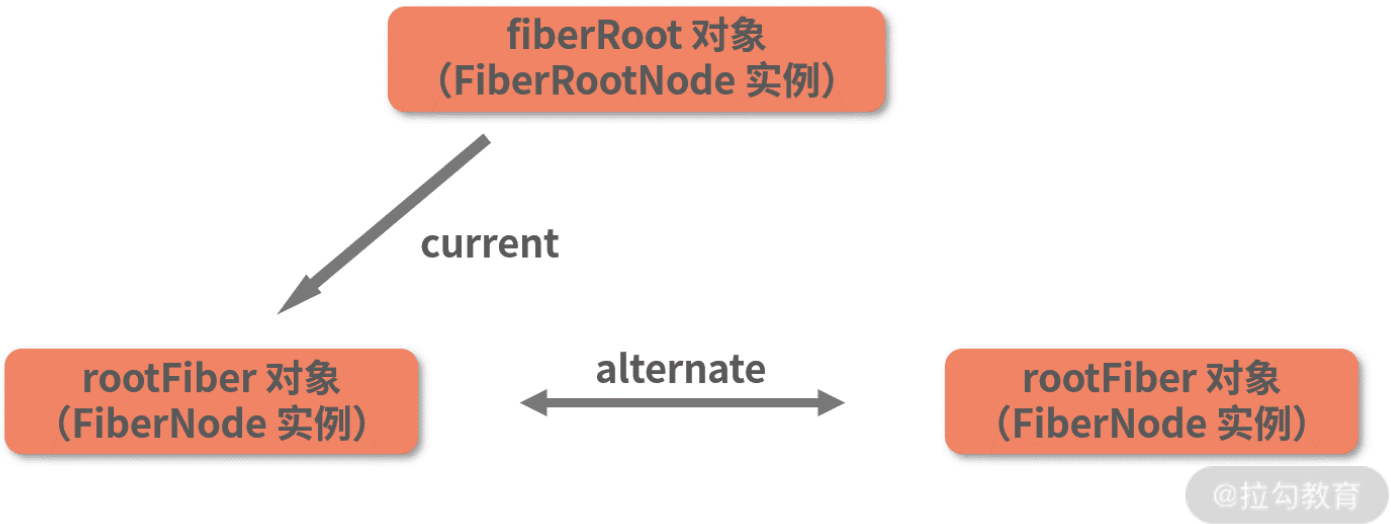
代码出奇的简单，但信息却给得很到位 —— createFiber 将创建一个 FiberNode 实例，而 FiberNode，上一讲已经讲过，它正是 Fiber 节点的类型。**因此 workInProgress 就是一个 Fiber 节点**。不仅如此，细心的你可能还会发现 workInProgress 的创建入参其实来源于 current，如下面代码所示：

```
1.  workInProgress = createFiber(current.tag, pendingProps, current.key, current.mo
```

复制代码

实锤了，workInProgress 节点其实就是 current 节点（即 rootFiber）的副本。

再结合 current 指向 rootFiber 对象（同样是 FiberNode 实例），以及 current 和 workInProgress 通过 alternate 互相连接这些信息，我们可以分析出这波操作执行完之后，整棵树的结构应该如下图所示：



完成了这个任务之后，就会进入 workLoopSync 的逻辑。这个 workLoopSync 函数也是个“人狠话不多”的主，它的逻辑同样是简洁明了的，如下所示（解析在注释里）：

```
function workLoopSync() {
  // 若 workInProgress 不为空
  while (workInProgress !== null) {
    // 针对它执行 performUnitOfWork 方法
    performUnitOfWork(workInProgress);
  }
}
```

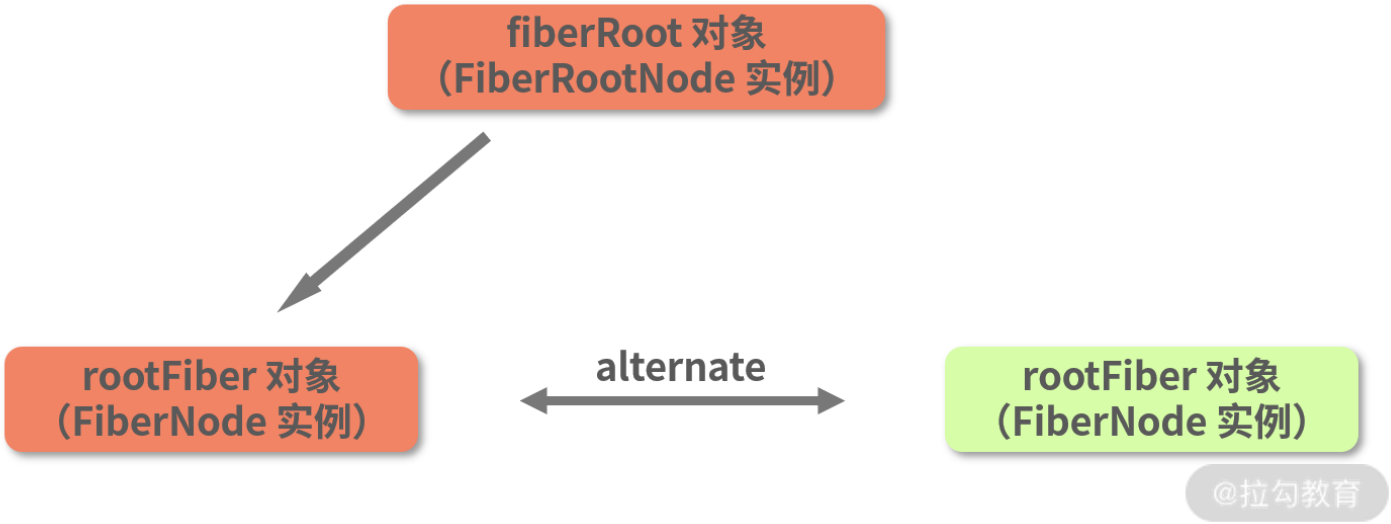
复制代码

workLoopSync 做的事情就是**通过 while 循环反复判断 workInProgress 是否为空，并在不为空的情况下针对它执行 performUnitOfWork 函数**。

而 performUnitOfWork 函数将**触发对 beginWork 的调用，进而实现对新 Fiber 节点的创建**。若 beginWork 所创建的 Fiber 节点不为空，则 performUniOfWork 会用这个新的 Fiber 节点来更新 workInProgress 的值，**为下一次循环做准备**。

通过循环调用 performUnitOfWork 来触发 beginWork，新的 Fiber 节点就会被不断地创建。当 workInProgress 终于为空时，说明没有新的节点可以创建了，也就意味着已经完成对整棵 Fiber 树的构建。

在这个过程中，**每一个被创建出来的新 Fiber 节点，都会一个一个挂载为最初那个 workInProgress 节点（如下图高亮处）的后代节点**。而上述过程中构建出的这棵 Fiber 树，也正是大名鼎鼎的 **workInProgress 树**。



相应地，图中 current 指针所指向的根节点所在的那棵树，我们叫它“**current 树**”。

这时候，相信一些同学心里已经开始犯嘀咕了：一棵 current 树，一棵 workInProgress 树，这名堂也太多了吧！况且这两棵 Fiber 树至少在现在看来，是完全没区别的（毕竟都还只有一个根节点，哈

哈）。React 这样设计的目的何在？或者换个问法——到底是什么样的事情一棵树做不到，非得搞两棵“一样”的树出来？

如果你想知道答案，就请好好把握住接下来的两讲内容吧！在一步一步理解 Fiber 树的构建和更新过程之后，我将带你去认识“两棵 Fiber 树”这一现象背后的动机。

接下来我们就深入到 beginWork 和 completeWork 的逻辑里去，一起看看 Fiber 树的构建过程及最终形态。

beginWork 开启 Fiber 节点创建过程

有一说一，beginWork 的源码实在是长到不科学。这里我们本着抓主要矛盾的原则，针对与树构建过程强相关的动作进行逻辑提取，代码如下（解析在注释里）：

复制代码

```
function beginWork(current, workInProgress, renderLanes) {
  ......

  //  current 节点不为空的情况下，会加一道辨识，看看是否有更新逻辑要处理
  if (current !== null) {
    // 获取新旧 props
    var oldProps = current.memoizedProps;
    var newProps = workInProgress.pendingProps;

    // 若 props 更新或者上下文改变，则认为需要"接受更新"
    if (oldProps !== newProps || hasContextChanged() || (
     workInProgress.type !== current.type )) {
      // 打个更新标
      didReceiveUpdate = true;
    } else if (xxx) {
      // 不需要更新的情况 A
      return A
    } else {
      if (需要更新的情况 B) {
        didReceiveUpdate = true;
      } else {
        // 不需要更新的其他情况，这里我们的首次渲染就将执行到这一行的逻辑
        didReceiveUpdate = false;
      }
    }
  } else {
    didReceiveUpdate = false;
  }
  ......
  // 这坨 switch 是 beginWork 中的核心逻辑，原有的代码量相当大
  switch (workInProgress.tag) {
    ......
    // 这里省略掉大量形如"case: xxx"的逻辑
    // 根节点将进入这个逻辑
    case HostRoot:
      return updateHostRoot(current, workInProgress, renderLanes)
```

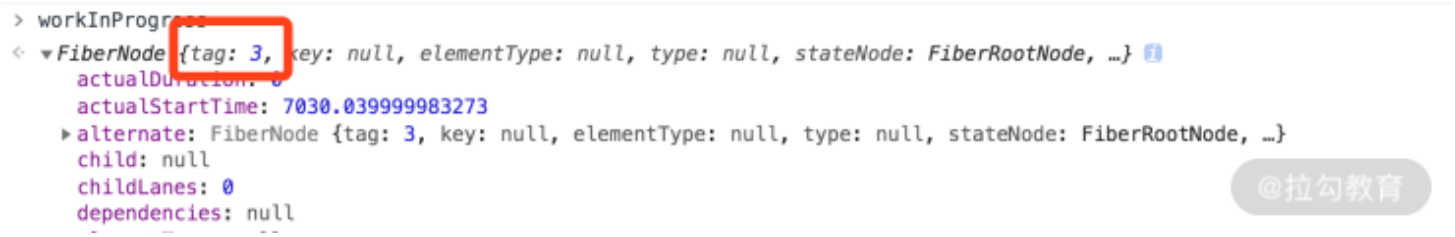
```
    // dom 标签对应的节点将进入这个逻辑
    case HostComponent:
      return updateHostComponent(current, workInProgress, renderLanes)

    // 文本节点将进入这个逻辑
    case HostText:
      return updateHostText(current, workInProgress)
    ......
    // 这里省略掉大量形如"case: xxx"的逻辑
  }
  // 这里是错误兜底，处理 switch 匹配不上的情况
  {
    {
      throw Error(
        "Unknown unit of work tag (" +
          workInProgress.tag +
          "). This error is likely caused by a bug in React. Please file an issu
      )
    }
  }
}
```

beginWork 源码太长不看版的重点总结：

1. beginWork 的入参是**一对用 alternate 连接起来的 workInProgress 和 current 节点**；
2. **beginWork 的核心逻辑是根据 fiber 节点（workInProgress）的 tag 属性的不同，调用不同的节点创建函数。**

当前的 current 节点是 rootFiber，而 workInProgress 则是 current 的副本，它们的 tag 都是 3，如下图所示：



而 3 正是 HostRoot 所对应的值，因此第一个 beginWork 将进入 updateHostRoot 的逻辑。

这里你先不必急于关注 updateHostRoot 的逻辑细节。事实上，在整段 switch 逻辑里，包含的形如“update+类型名”这样的函数是非常多的。在专栏示例的 Demo 中，就涉及了对 updateHostRoot、updateHostComponent 等的调用，十来种 updateXXX，我们不可能一个一个去扣每一个函数的逻辑。

幸运的是，这些函数之间不仅命名形式一致，工作内容也相似。就 render 链路来说，它们共同的特性，就是都会**通过调用 reconcileChildren 方法，生成当前节点的子节点**。

reconcileChildren 的源码如下：

```
function reconcileChildren(current, workInProgress, nextChildren, renderLanes) {
  // 判断 current 是否为 null
  if (current === null) {
    // 若 current 为 null，则进入 mountChildFibers 的逻辑
    workInProgress.child = mountChildFibers(workInProgress, null, nextChildren,
  } else {
    // 若 current 不为 null，则进入 reconcileChildFibers 的逻辑
    workInProgress.child = reconcileChildFibers(workInProgress, current.child, r
  }
}
```

从源码来看，reconcileChildren 也只是做逻辑的分发，具体的工作还要到 **mountChildFibers** 和 **reconcileChildFibers** 里去看。

ChildReconciler，处理 Fiber 节点的幕后“操盘手”

那么这两个函数又是何方神圣呢？在源码中，我们可以觅得这样两个赋值语句：

```
var reconcileChildFibers = ChildReconciler(true);
var mountChildFibers = ChildReconciler(false);
```

原来 reconcileChildFibers 和 mountChildFibers 不仅名字相似，出处也一致。**它们都是 ChildReconciler 这个函数的返回值，仅仅存在入参上的区别**。而 ChildReconciler，则是一个实打实的“庞然大物”，其内部的逻辑量堪比 N 个 beginWork。这里我将关键要素提取如下（解析在注释里）：

```
function ChildReconciler(shouldTrackSideEffects) {
  // 删除节点的逻辑
  function deleteChild(returnFiber, childToDelete) {
    if (!shouldTrackSideEffects) {
      // Noop.
      return;
    }
    // 以下执行删除逻辑
  }

  ......


  // 单个节点的插入逻辑
  function placeSingleChild(newFiber) {
    if (shouldTrackSideEffects && newFiber.alternate === null) {
```

```
      newFiber.flags = Placement;
    }
    return newFiber;
  }

  // 插入节点的逻辑
  function placeChild(newFiber, lastPlacedIndex, newIndex) {
    newFiber.index = newIndex;
    if (!shouldTrackSideEffects) {
      // Noop.
      return lastPlacedIndex;
    }
    // 以下执行插入逻辑
  }
  ......
  // 此处省略一系列 updateXXX 的函数，它们用于处理 Fiber 节点的更新

  // 处理不止一个子节点的情况
  function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren, l
    ......
  }
  // 此处省略一堆 reconcileXXXXX 形式的函数，它们负责处理具体的 reconcile 逻辑
  function reconcileChildFibers(returnFiber, currentFirstChild, newChild, lanes)
    // 这是一个逻辑分发器，它读取入参后，会经过一系列的条件判断，调用上方所定义的负责具体节点操
  }

  // 将总的 reconcileChildFibers 函数返回
  return reconcileChildFibers;
}
```

由于原本的代码量着实巨大，感兴趣的同学可以点开这个文件查看细节，此处我仅针对与主流程强相关的逻辑为你总结以下要点：

1. 关键的入参 shouldTrackSideEffects，意为“是否需要追踪副作用”，**因此 reconcileChildFibers 和 mountChildFibers 的不同，在于对副作用的处理不同**；
2. ChildReconciler 中定义了大量如 placeXXX、deleteXXX、updateXXX、reconcileXXX 等这样的函数，这些函数覆盖了对 Fiber 节点的创建、增加、删除、修改等动作，将直接或间接地被 reconcileChildFibers 所调用；
3. ChildReconciler 的返回值是一个名为 reconcileChildFibers 的函数，这个函数是一个逻辑分发器，**它将根据入参的不同，执行不同的 Fiber 节点操作，最终返回不同的目标 Fiber 节点**。

对于第 1 点，这里展开说说。对副作用的处理不同，到底是哪里不同？以 placeSingleChild 为例，以下是 placeSingleChild 的源码：

复制代码

```
function placeSingleChild(newFiber) {
  if (shouldTrackSideEffects && newFiber.alternate === null) {
    newFiber.flags = Placement;
  }
  return newFiber;
}
```

可以看出，一旦判断 shouldTrackSideEffects 为 false，那么下面所有的逻辑都不执行了，直接返回。那如果执行下去会发生什么呢？简而言之就是给 Fiber 节点打上一个叫“flags”的标记，像这样：

```
newFiber.flags = Placement;
```

复制代码

这个名为 flags 的标记有何作用呢？

小科普：flags 是什么

由于这里我引用的是 v17.0.0 版本的源码，属性名已经变更为 flags，但在更早一些的版本中，这个属性名叫“effectTag”。在时下的社区讨论中，effectTag 这个命名更常见，也更语义化，因此下文我将以“effectTag”代指“flags”。

Placement 这个 effectTag 的意义，是在渲染器执行时，也就是真实 DOM 渲染时，告诉渲染器：**我这里需要新增 DOM 节点**。 effectTag 记录的是**副作用的类型**，而**所谓“副作用”**，React 给出的定义是“**数据获取、订阅或者修改 DOM**”等动作。在这里，Placement 对应的显然是 DOM 相关的副作用操作。

像 Placement 这样的副作用标识，还有很多，它们均以二进制常量的形式存在，下图我为你截取了局部（你可以在这个文件里查看 effectTag 的类型）：

```
// You can change the rest (and add more).
export const Placement = /*                    */ 0b0000000000000000010;
export const Update = /*                       */ 0b0000000000000000100;
export const PlacementAndUpdate = /*           */ 0b0000000000000000110;
export const Deletion = /*                     */ 0b0000000000000001000;
```


回到我们的调用链路里来，由于 current 是 rootFiber，它不为 null，因此它将走入的是下图所高亮的这行逻辑。也就是说在 mountChildFibers 和 reconcileChildFibers 之间，它选择的是 **reconcileChildFibers**：

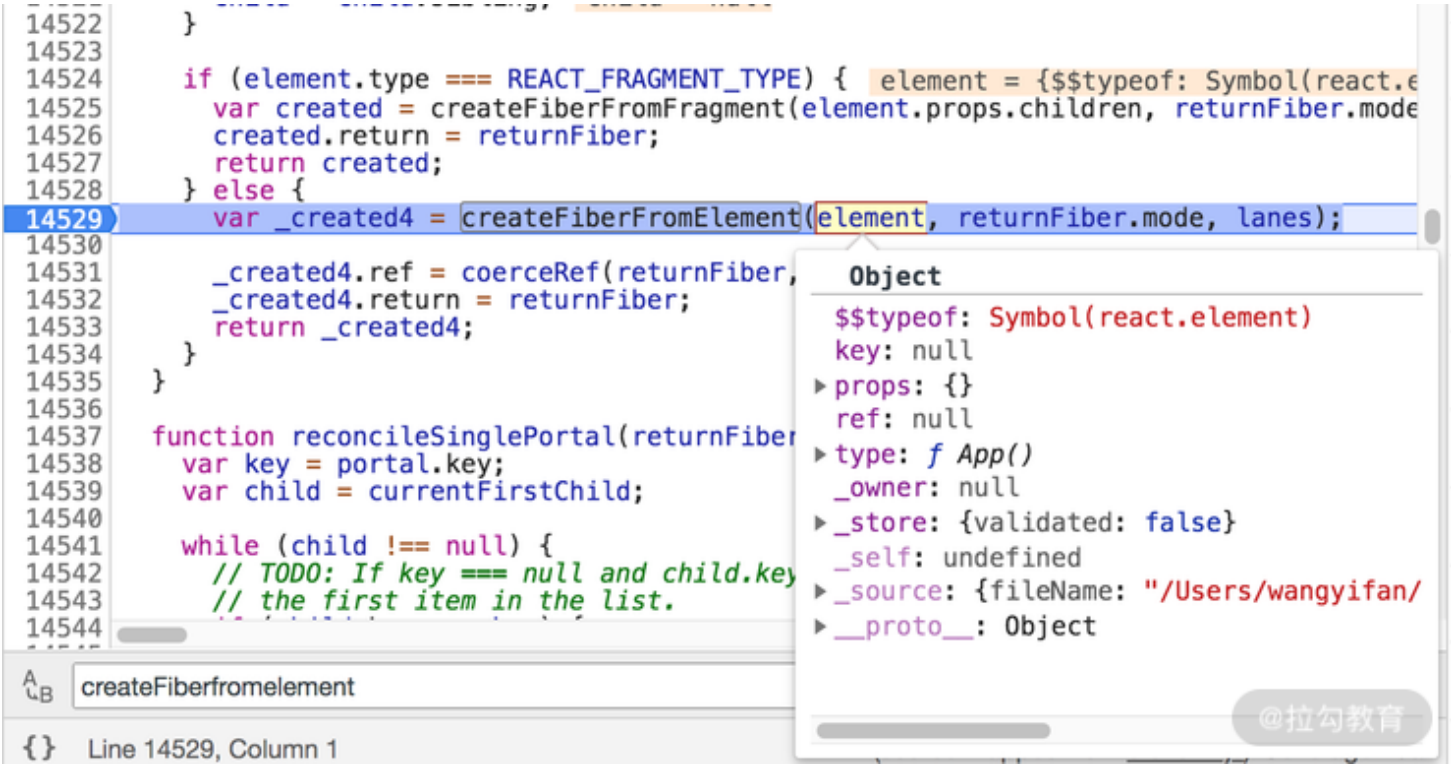
```
17521
17522 function reconcileChildren(current, workInProgress, nextChildren, renderL
17523   if (current === null) {
17524     // If this is a fresh new component that hasn't been rendered yet, we
17525     // won't update its child set by applying minimal side-effects. Inste
17526     // we will add them all to the child before it gets rendered. That me
17527     // we can optimize this reconciliation pass by not tracking side-effe
17528     workInProgress.child = mountChildFibers(workInProgress, null, nextChi
17529   } else {
17530     // If the current child is the same as the work in progress, it means
17531     // we haven't yet started any work on these children. Therefore, we u
17532     // the clone algorithm to create a copy of all the current children.
17533     // If we had any progressed work already, that is invalid at this poi
17534     // let's throw it out.
17535     workInProgress.child = reconcileChildFibers(workInProgress, current.c
17536   }
17537 }
```

结合前面的分析可知，reconcileChildFibers 是`ChildReconciler(true)`的返回值。入参为 true，意味着其内部逻辑是允许追踪副作用的，因此“打 effectTag”这个动作将会生效。

接下来进入 reconcileChildFibers 的逻辑，在 reconcileChildFibers 这个逻辑分发器中，会把 rootFiber 子节点的创建工作分发给 reconcileXXX 函数家族的一员——reconcileSingleElement 来处理，具体的调用形式如下图高亮处所示：

```
4     var isObject = typeof newChild === 'object' && newChild !== null;  isObject = true, n
5
6     if (isObject) {  isObject = true
7       switch (newChild.$$typeof) {  newChild = {$$typeof: Symbol(react.element), key: "2
8         case REACT_ELEMENT_TYPE:
9           return placeSingleChild(reconcileSingleElement(returnFiber, currentFirstChild,
0
1         case REACT_PORTAL_TYPE:
2           return placeSingleChild(reconcileSinglePortal(returnFiber, currentFirstChild, n
3
4         case REACT_LAZY_TYPE:
```

reconcileSingleElement 将基于 rootFiber 子节点的 ReactElement 对象信息，创建其对应的 FiberNode。这个过程中涉及的函数调用如下图高亮处所示：



这里需要说明的一点是：**rootFiber 作为 Fiber 树的根节点**，它并没有一个确切的 ReactElement 与之映射。结合 JSX 结构来看，**我们可以将其理解为是 JSX 中根组件的父节点**。课时所给出的 Demo 中，组件编码如下：

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);
```

可以看出，根组件是一个类型为 App 的函数组件，因此 **rootFiber 就是 App 的父节点**。

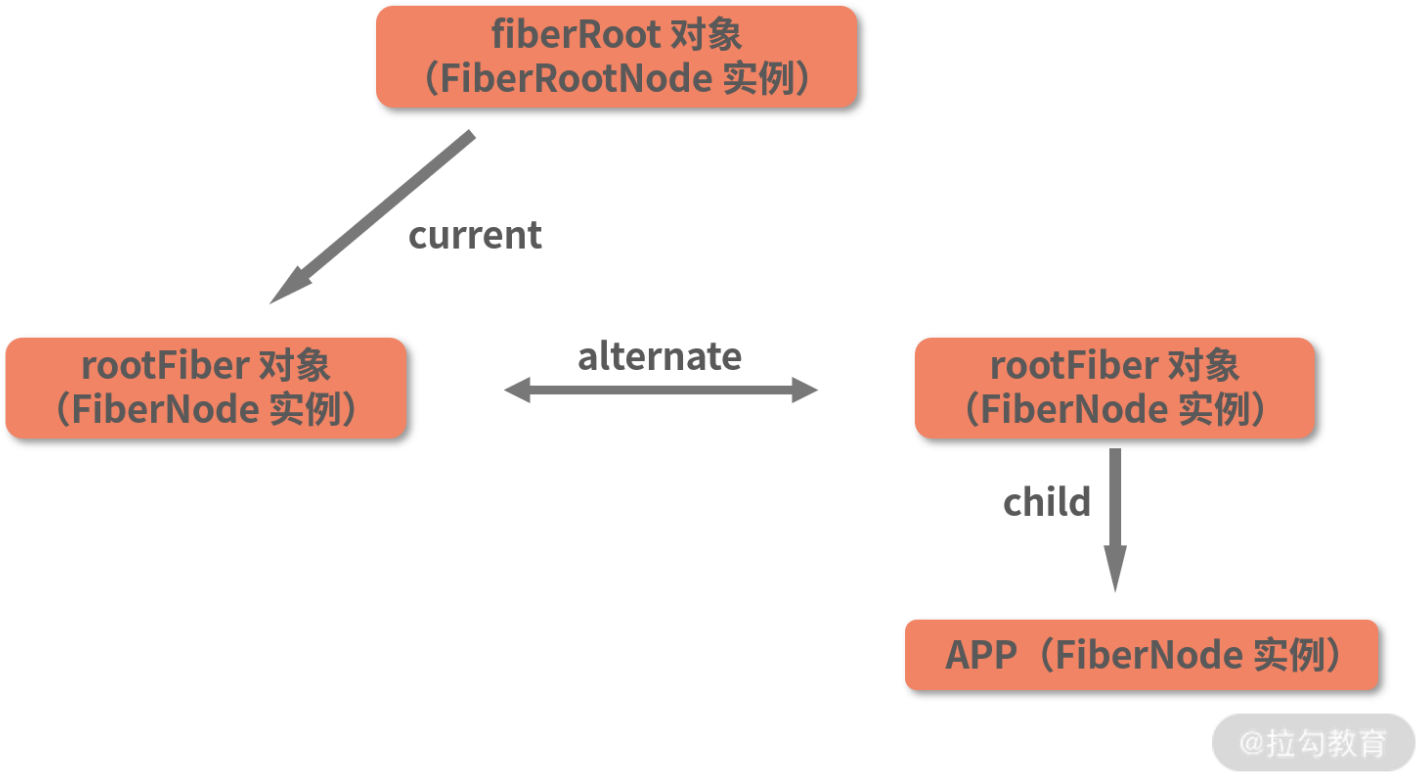
结合这个分析来看，图中的 _created4 是根据 rootFiber 的第一个子节点对应的 ReactElement 来创建的 Fiber 节点，那么它就是 **App 所对应的 Fiber 节点**。现在我为你打印出运行时的 _created4 值，会发现确实如此：

```
> _created4
FiberNode {tag: 2, key: null, stateNode: null, elementType: ƒ, type: ƒ, …}
    actualDuration: 0
    actualStartTime: -1
    alternate: null
    child: null
    childLanes: 0
    dependencies: null
  ▶ elementType: ƒ App()
    firstEffect: null
    flags: 0
    index: 0
    key: null
    lanes: 1
    lastEffect: null
    memoizedProps: null
    memoizedState: null
    mode: 8
    nextEffect: null
  ▶ pendingProps: {}
    ref: null
    return: null
    selfBaseDuration: 0
```


App 所对应的 Fiber 节点，将被 placeSingleChild 打上“Placement”（新增）的副作用标记，而后作为 reconcileChildFibers 函数的返回值，返回给下图中的 workInProgress.child：

```
function reconcileChildren(current, workInProgress, nextChildren, renderLanes) {
  if (current === null) {
    // If this is a fresh new component that hasn't been rendered yet, we
    // won't update its child set by applying minimal side-effects. Instead,
    // we will add them all to the child before it gets rendered. That means
    // we can optimize this reconciliation pass by not tracking side-effects.
    workInProgress.child = mountChildFibers(workInProgress, null, nextChildren, renderLa
  } else {
    // If the current child is the same as the work in progress, it means that
    // we haven't yet started any work on these children. Therefore, we use
    // the clone algorithm to create a copy of all the current children.
    // If we had any progressed work already, that is invalid at this point so
    // let's throw it out.
    workInProgress.child = reconcileChildFibers(workInProgress, current.child, nextChil
  }
}
```


reconcileChildren 函数上下文里的 workInProgress 就是 rootFiber 节点。那么此时，我们就将新创建的 App Fiber 节点和 rootFiber 关联了起来，整个 Fiber 树如下图所示：



Fiber 节点的创建过程梳理

分析完 App FiberNode 的创建过程，我们先不必急于继续往下走这个渲染链路。因为其实最关键的东西已经讲完了，剩余节点的创建只不过是对 performUnitOfWork、 beginWork 和 ChildReconciler 等相关逻辑的重复。

刚刚这一通分析所涉及的调用栈很长，相信不少人如果是初读的话，过程中肯定不可避免地要反复回看，确认自己现在到底在调用栈的哪一环。这里为了方便你把握逻辑脉络，我将本讲讲解的 beginWork 所触发的调用流程总结进一张大图：

Fiber 树的构建过程

理解了 Fiber 节点的创建过程，就不难理解 Fiber 树的构建过程了。

前面我们已经锲而不舍地研究了各路关键函数的源码逻辑，此时相信你已经能够将函数名与函数的工作内容做到对号入座。这里我们不必再纠结与源码的实现细节，可以直接从工作流程的角度来看后续节点

的创建。

循环创建新的 Fiber 节点

研究节点创建的工作流，我们的切入点是`workLoopSync`这个函数。

为什么选它？这里来复习一遍`workLoopSync`会做什么：

```
function workLoopSync() {
  // 若 workInProgress 不为空
  while (workInProgress !== null) {
    // 针对它执行 performUnitOfWork 方法
    performUnitOfWork(workInProgress);
  }
}
```

它会循环地调用 performUnitOfWork，而 performUnitOfWork，开篇我们已经点到过它，其主要工作是“通过调用 beginWork，来实现新 Fiber 节点的创建”；它还有一个次要工作，**就是把新创建的这个 Fiber 节点的值更新到 workInProgress 变量里去**。源码中的相关逻辑提取如下：

```
// 新建 Fiber 节点
next = beginWork$1(current, unitOfWork, subtreeRenderLanes);
// 将新的 Fiber 节点赋值给 workInProgress
if (next === null) {
  // If this doesn't spawn new work, complete the current work.
  completeUnitOfWork(unitOfWork);
} else {
  workInProgress = next;
}
```

如此便能够确保每次 performUnitOfWork 执行完毕后，当前的 **workInProgress 都存储着下一个需要被处理的节点，从而为下一次的 workLoopSync 循环做好准备**。

现在我在 workLoopSync 内部打个断点，尝试输出每一次获取到的 workInProgress 的值，workInProgress 值的变化过程如下图所示：

```
> workInProgress
◂ ▸FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: FiberRootNode, …}
> workInProgress
◂ ▸FiberNode {tag: 2, key: null, stateNode: null, elementType: f, type: f, …}
> workInProgress
◂ ▸FiberNode {tag: 5, key: null, elementType: "div", type: "div", stateNode: null, …}
> workInProgress
◂ ▸FiberNode {tag: 5, key: null, elementType: "div", type: "div", stateNode: null, …}
> workInProgress
◂ ▸FiberNode {tag: 5, key: null, elementType: "h1", type: "h1", stateNode: null, …}
> workInProgress
◂ ▸FiberNode {tag: 5, key: null, elementType: "p", type: "p", stateNode: null, …}
> workInProgress
◂ ▸FiberNode {tag: 5, key: null, elementType: "p", type: "p", stateNode: null, …}
```


共有 7 个节点，若你点击展开查看每个节点的内容，就会发现这 7 个节点其实分别是：

- rootFiber（当前 Fiber 树的根节点）
- App FiberNode（App 函数组件对应的节点）
- class 为 App 的 DOM 元素对应的节点，其内容如下图所示

```
> workInProgress
◂ ▾FiberNode {tag: 5, key: null, elementType: "div", type: "div", stateNode: nul
    actualDuration: 8581.27499994589
    actualStartTime: 849231.1149999732
    alternate: null
  ▸ child: FiberNode {tag: 5, key: null, elementType: "div", type: "div", stateN
    childLanes: 0
    dependencies: null
    elementType: "div"
    firstEffect: null
    flags: 0
    index: 0
    key: null
    lanes: 0
    lastEffect: null
  ▸ memoizedProps: {className: "App", children: {…}}
    memoizedState: null
```


- class 为 container 的 DOM 元素对应的节点，其内容如下图所示

- h1 标签对应的节点

- 第 1 个 p 标签对应的 FiberNode，内容为“我是第一段话”，如下图所示

- 第 2 个 p 标签对应的 FiberNode，内容为“我是第二段话”，如下图所示

```
> workInProgress
FiberNode {tag: 5, key: null, elementType: "p", type: "p", stateNode: null, …}
  actualDuration: 0.2049999893642962
  actualStartTime: 887997.8850000189
  alternate: null
  child: null
  childLanes: 0
  dependencies: null
  elementType: "p"
  firstEffect: null
  flags: 0
  index: 2
  key: null
  lanes: 0
  lastEffect: null
▶ memoizedProps: {children: "我是第二段话"}
  memoizedState: null
  mode: 8
  nextEffect: null
▶ pendingProps: {children: "我是第二段话"}
  ref: null
▶ return: FiberNode {tag: 5, key: null, elementType: "div", type: "div", stateNode: div.container,
  selfBaseDuration: 0.10000000474974513
  sibling: null
▶ stateNode: p
  tag: 5
```


结合这 7 个 FiberNode，再对照对照我们的 Demo：

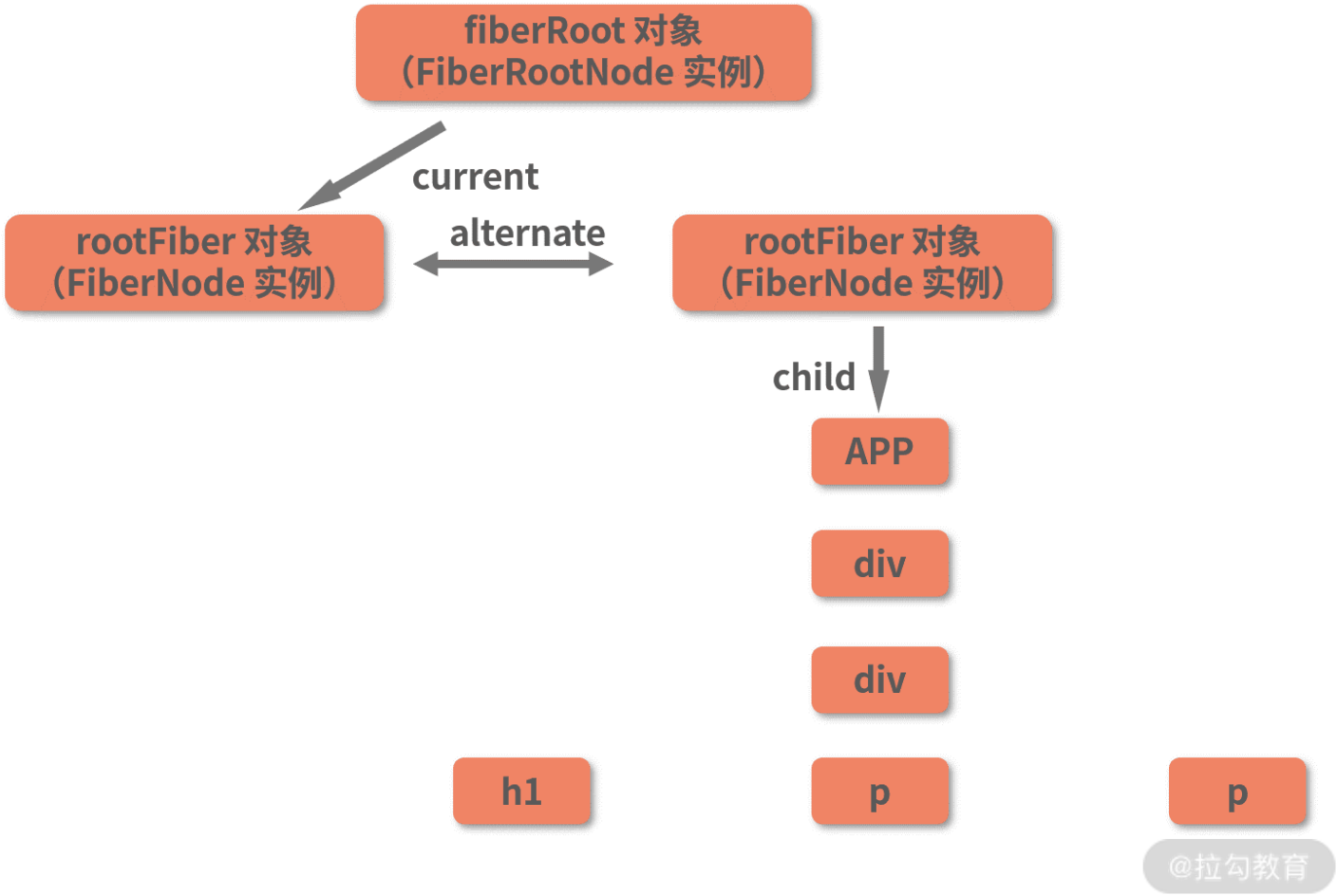
复制代码

```
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>
    );
}
```

你会发现组件自上而下，每一个非文本类型的 ReactElement 都有了它对应的 Fiber 节点。

> 注：React 并不会为所有的文本类型 ReactElement 创建对应的 FiberNode，这是一种优化策略。是否需要创建 FiberNode，在源码中是通过isDirectTextChild这个变量来区分的。

这样一来，我们构建的这棵树里，就多出了不少 FiberNode，如下图所示：



Fiber 节点有是有了，但这些 Fiber 节点之间又是如何相互连接的呢?

Fiber 节点间是如何连接的呢

不同的 Fiber 节点之间，将通过 child、return、sibling 这 3 个属性建立关系，其中 child、return 记录的是父子节点关系，而 sibling 记录的则是兄弟节点关系。

这里我以 h1 这个元素对应的 Fiber 节点为例，给你展示下它是如何与其他节点相连接的。展开这个 Fiber 节点，对它的 child、 return、sibling 3 个属性作截取，如下图所示：

child 属性为 null，说明 h1 节点没有子 Fiber 节点：

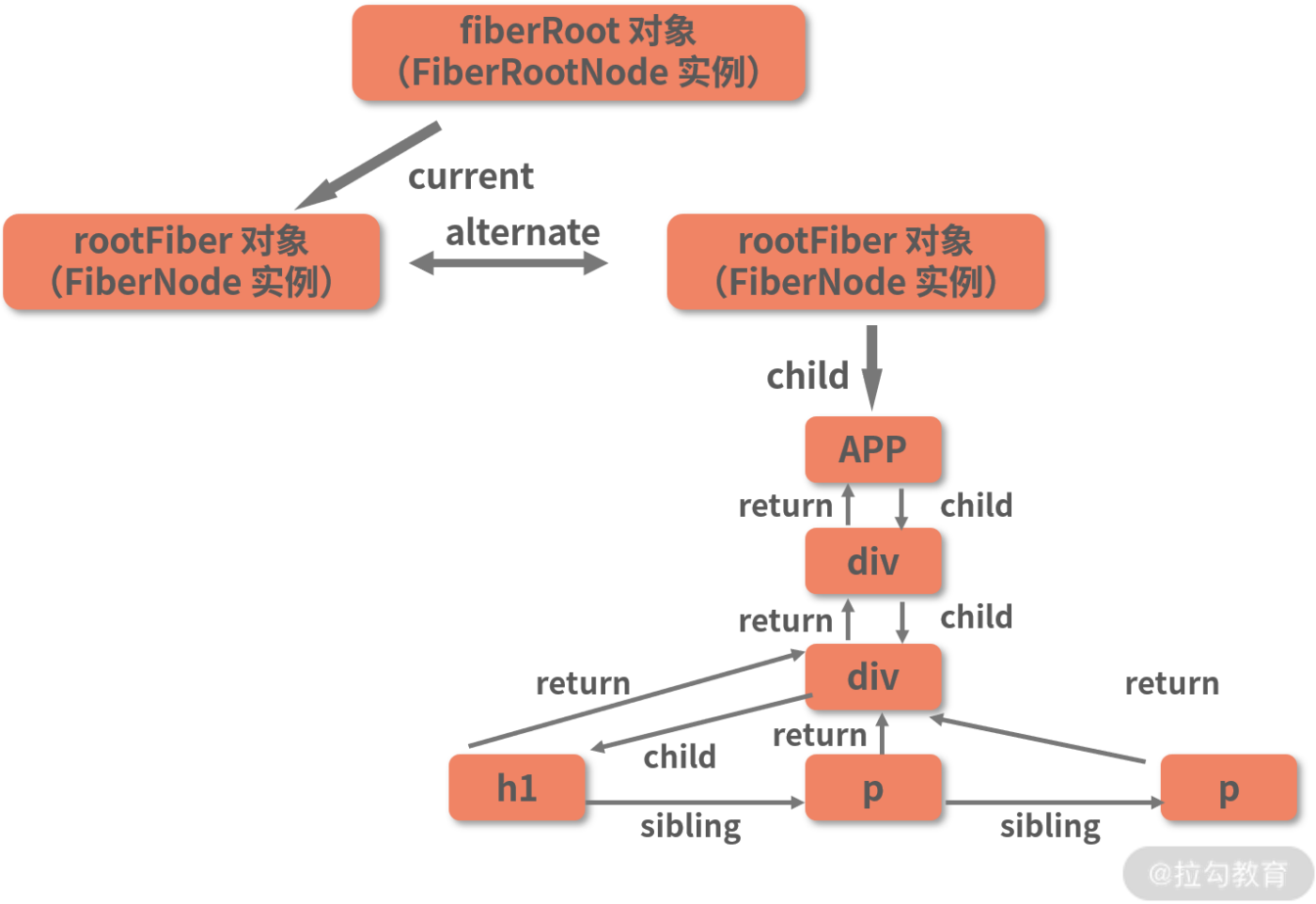
```
> workInProgress
<· ▼FiberNode {tag: 5, key: null, elementType: "h1", ty
    actualDuration: 1995.7000000285916
    actualStartTime: 872852.0649999846
    alternate: null
    child: null
```


return 属性局部截图：

sibling 属性局部截图：

可以看到，return 属性指向的是 class 为 container 的 div 节点，而 sibling 属性指向的是第 1 个 p 节点。结合 JSX 中的嵌套关系我们不难得知 ——**FiberNode 实例中，return 指向的是当前 Fiber 节点的父节点，而 sibling 指向的是当前节点的第 1 个兄弟节点。**

结合这 3 个属性所记录的节点间关系信息，我们可以轻松地将上面梳理出来的新 FiberNode 连接起来：



以上便是 workInProgress Fiber 树的最终形态了。从图中可以看出，虽然人们习惯上仍然将眼前的这个产物称为“Fiber 树”，但**它的数据结构本质其实已经从树变成了链表**。

注意，在分析 Fiber 树的构建过程时，我们选取了 **beginWork** 作为切入点，但整个 Fiber 树的构建过程中，并不是只有 beginWork 在工作。这其中，还穿插着 **completeWork** 的工作。只有将 completeWork 和 beginWork 放在一起来看，你才能够真正理解，Fiber 架构下的“深度优先遍历”到底是怎么一回事。

总结

通过本讲的学习，你掌握了 beginWork 的实现原理、理清了 Fiber 节点的创建链路，最终串联起了 Fiber 树的宏观构建过程。至此，你已经揽获了 render 阶段大半的知识，这一路道阻且难，胜在收获满满。

下一讲，我们一方面将乘胜追击，继续探索 completeWork 的工作内容，将整个 render 阶段讲透；另一方面，我会带你快速地过一遍 commit 阶段的工作流，并基于此去串联由初始化、render、commit

所组成的完整渲染工作流，力求对整个 ReactDOM.render 所触发的渲染链路形成一个系统、通透的理解。

此外，在本讲的开头，我还给你留下了一个悬念，也就是“为什么需要两棵 Fiber 树”的问题。这个问题的答案，也将会随着我们对 Fiber 探索的深入，逐渐浮出水面。