23 React 如何构建 workInProgress 树

更新时间：2020-09-18 10:04:42

> 辛苦是获得一切的定律。——牛顿

前言

前面提到，React 应用程序首次渲染时在 prerender 阶段构建了 fiberRoot 对象，并将整个应用程序根组件元素作为更新内容加入到更新队列，然后附加到了 fiberRoot 对象上面。此时的 fiberRoot 对象将作为整个 Fiber 架构的根基进入到 render 阶段。

应用程序渲染过程进入到 render 阶段时，React 的一项重要工作就是构建 workInProgress 树，在这个过程中 React 也会完成结点 diff 的逻辑，最终会得到用于更新到屏幕的副作用列表。

构建 workInProgress 树的过程也是执行「协调算法」过程，通过循环解析工作单元获取下一个 Fiber 结点，同时父子结点和兄弟结点也会被串联起来。这个过程从整体上可以分为两步，分别是 **初始化 workInProgress 对象和完善 workInProgress 对象**。首先，我们先看一下应用程序执行过程刚进入 render 阶段时的 fiberRoot 对象。

进入 render 阶段时 fiberRoot 对象

应用程序首次渲染过程刚进入 render 阶段时 fiberRoot 对象的结构如图 5.4.1。

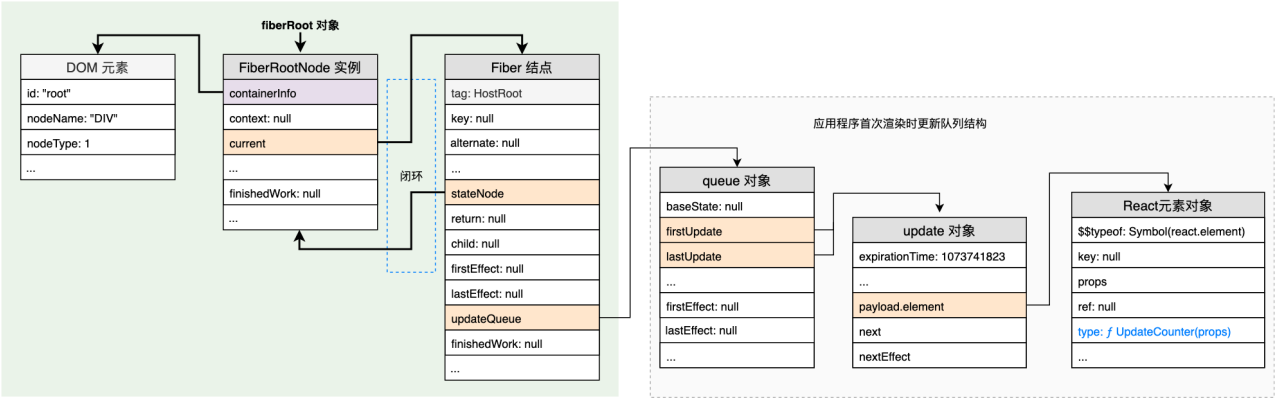

图 5.4.1 进入 render 阶段时 fiberRoot 对象结构

应用程序首次渲染过程刚进入 render 阶段时 fiberRoot 对象上面只有 current 树，此时的 current 树只有一个类型为 HostRoot 的 Fiber 结点，该 Fiber 结点的更新队列中只有一个更新对象，该更新对象的内容就是是应用程序的根组件元素。紧接着 React 要做的就是初始化 workInProgress 对象。

初始化 workInProgress 对象

render 阶段的主要逻辑在 renderRoot 函数内，在该函数内部会通过 prepareFreshStack() 函数初始化 workInProgress 对象，见代码示例 5.4.1。

```
// 源码位置：packages/react-reconciler/src/ReactFiberWorkLoop.js
function prepareFreshStack(root, expirationTime) {
  // 这里的形参root是fiberRoot对象
  // 重要参数重置
  root.finishedWork = null;
  root.finishedExpirationTime = NoWork;
  ...
  workInProgressRoot = root;
  // workInProgress变量会暴露到外层函数renderRoot作用域
  workInProgress = createWorkInProgress(root.current, null, expirationTime);
  renderExpirationTime = expirationTime;
  workInProgressRootExitStatus = RootIncomplete;
  ...
}
```

代码示例 5.4.1 初始化 workInProgress 对象的外部函数

真正为 workInProgress 对象赋初始值的函数是 createWorkInProgress，见代码示例 5.4.2。

```
// 源码位置：packages/react-reconciler/src/ReactFiber.js
function createWorkInProgress(current, pendingProps, expirationTime) {
  // 变量workInProgress和current.alternate指向了同一个内存，因此current.alternate就是workInProgress
  var workInProgress = current.alternate;
  // 首次渲染时current.alternate值为null，即workInProgress为null
  if (workInProgress === null) {
    // 开始为workInProgress树的第一个Fiber结点赋值，创建的是HostRoot类型的结点
    workInProgress = createFiber(current.tag, pendingProps, current.key, current.mode);
    workInProgress.elementType = current.elementType;
    workInProgress.type = current.type;
    workInProgress.stateNode = current.stateNode;
    ...
    // 这里的处理就形成了一个闭环
    workInProgress.alternate = current;
    current.alternate = workInProgress;
  } else {
    ...
  }
  workInProgress.childExpirationTime = current.childExpirationTime;
  workInProgress.expirationTime = current.expirationTime;

  workInProgress.child = current.child;
  workInProgress.memoizedProps = current.memoizedProps;
  workInProgress.memoizedState = current.memoizedState;
  workInProgress.updateQueue = current.updateQueue;
  ...
  workInProgress.sibling = current.sibling;
  workInProgress.index = current.index;
  workInProgress.ref = current.ref;
  ...
  return workInProgress;
}
```

代码示例 5.4.2 初始化 workInProgress 对象

事实上，workInProgress 树上面的第一个 Fiber 结点和 current 树上面的第一个 Fiber 结点中的属性值基本相同，workInProgress 对象初始化完成后 fiberRoot 对象的结构见图 5.4.2。

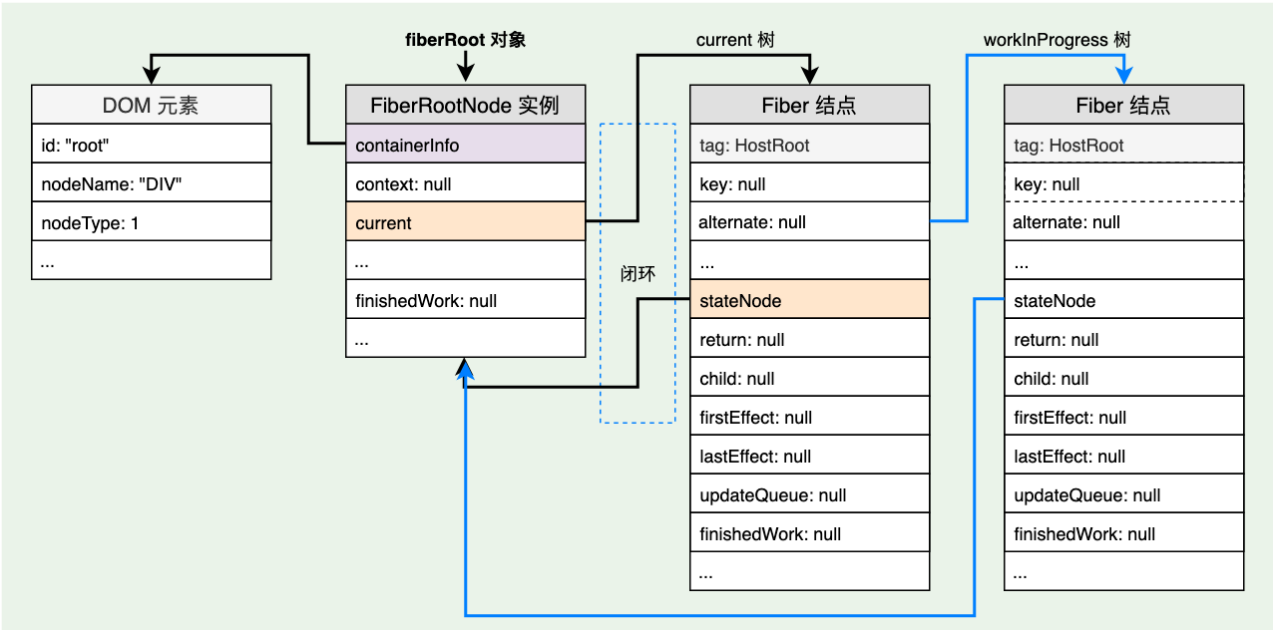

图 5.4.2 workInProgress 对象初始化后的fiberRoot结构

应用程序首次渲染时经过执行 createWorkInProgress 函数后，workInProgress 对象（树）上面有了第一个 Fiber 结点，该结点的 tag 属性为 HostRoot，它是整个 workInProgress 树的根结点。下面我们来看 React 如何来完善当前 workInProgress 对象，使之形成最终的workInProgress （对象）树。

完善 workInProgress 对象

workInProgress 对象被完善后就成了我们所说的 workInProgress 树，那么该树将会是什么样的结构呢？见下图 5.4.3。

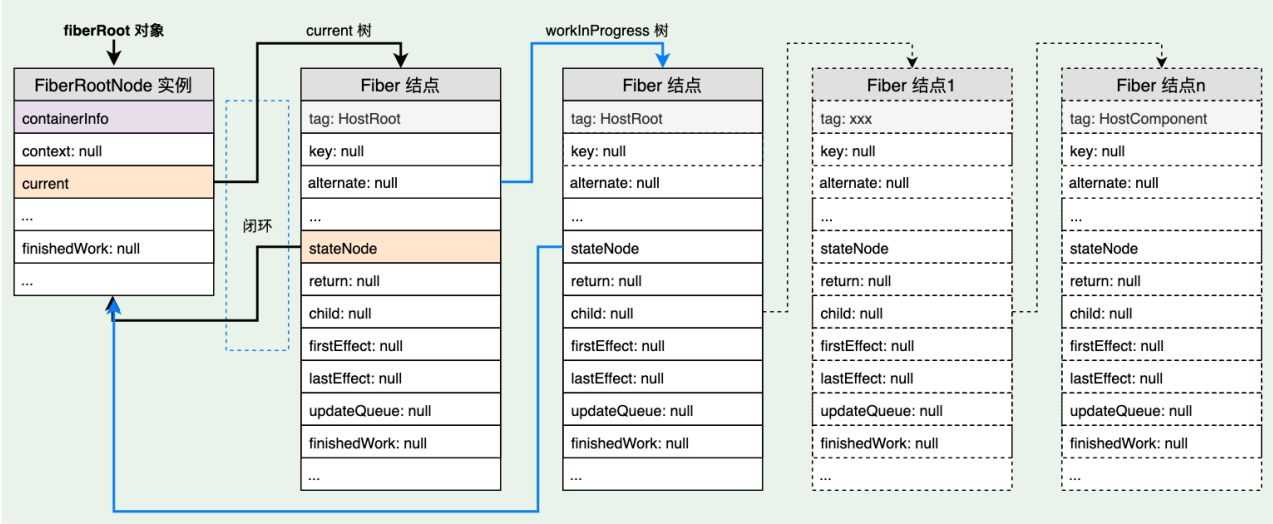

图 5.4.3 workInProgress 树结构示意图

前面提到过，应用程序首次渲染时更新队列中只有一个更新对象，该更新对象携带的更新内容就是应用程序的根组件元素。事实上，要想把组件元素描述的页面结构映射到屏幕上必须先将元素转换为 Fiber 结点。React 完善 workInProgress 对象的过程就是将元素转换为 Fiber 结点的过程。那么，这个过程是什么样的呢？

循环解析工作单元

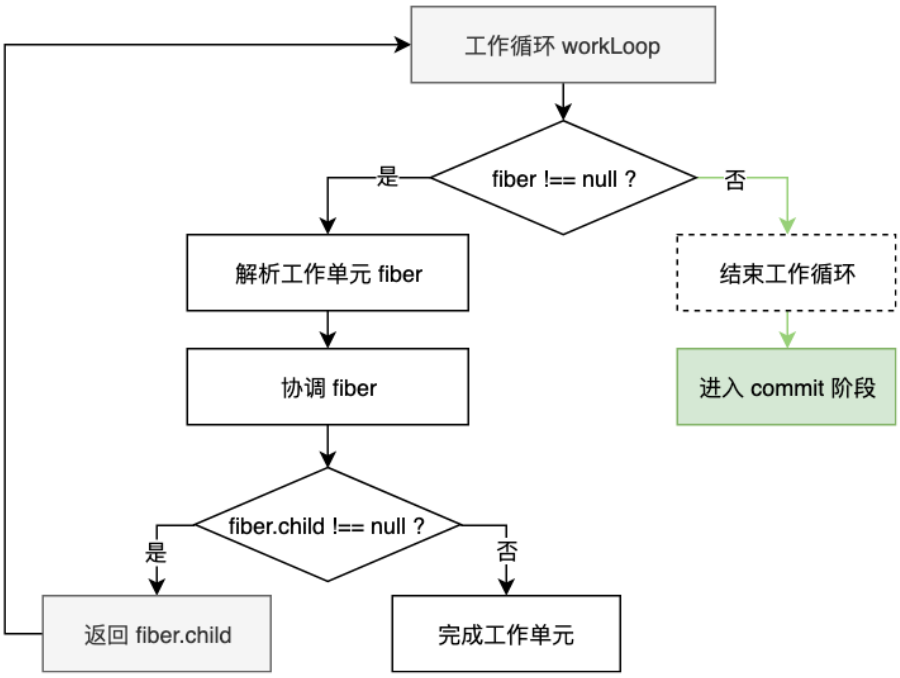

图 5.4.4 工作循环流程图

循环解析工作单元的过程主要是执行 workLoop 函数，这个过程也可以称为 **工作循环** ，每次循环解析的工作单元就是上一次 Fiber 结点的 child。第一次循环解析的是 HostRoot 类型的结点，该结点也就是所有结点的祖先。workLoop 函数负责执行工作循环，该函数在 renderRoot 函数里面被调用，见代码示例 5.4.3。

```
function renderRoot(root, expirationTime, isSync) {
  // ...
  do {
    try {
      if (isSync) {
        // 同步执行工作循环
        workLoopSync();
      } else {
        // 正常执行工作循环
        workLoop();
      }
      break;
    } catch (e) {
      // 处理异常
    }
  } while (true)
}

// 工作循环函数
function workLoop() {
  // 一直进行工作单元循环解析，直到任务调度器收回执行权
  while (workInProgress !== null && !shouldYield()) {
    workInProgress = performUnitOfWork(workInProgress);
  }
}
```

代码示例 5.4.3 工作循环的执行时机与逻辑

循环解析工作单元，那么工作单元具体是怎么解析的呢？

解析工作单元的逻辑

解析工作单元的逻辑主要在 performUnitOfWork 函数中进行，见代码示例 5.4.4。

```
function performUnitOfWork(unitOfWork) {
  // unitOfWork为当前需要解析的Fiber结点
  // 首次渲染时unitOfWork.alternate的值一般为null
  var current = unitOfWork.alternate;
  var next = void 0;
  // ...
  // next为beginWork执行结束返回的Fiber结点，该结点也就成了下一次要解析的工作单元
  next = beginWork(current, unitOfWork, renderExpirationTime);
  // ...
  if (next === null) {
    // 如果next值为null，说明当前的Fiber结点已经是叶子结点，接下来要执行完成工作单元解析工作
    next = completeUnitOfWork(unitOfWork);
  }
}
```

代码示例 5.4.4 performUnitOfWork 函数

在 beginWork 函数内部通过匹配 Fiber 结点的 tag 值调用 updateHostRoot， updateClassComponent， updateHostComponent 和 updateHostText 等函数分别解析 HostRoot， ClassComponent， HostComponent 和 HostText 等类型的 Fiber 结点。

> 注：工作单元解析的详细逻辑在下一节中进行介绍。

完成工作单元

工作单元解析执行到 workInProgress 树的叶子结点时，会完成当前工作单元。完成工作单元的主要工作是收集副作用，同时处理 HostComponent 类型的结点，创建对应的 DOM 元素并将他们 append 到父结点上。这部分内容在下一节进行详细介绍。

> 注：完成工作单元解析的详细逻辑在下一节中进行介绍。

小结

本节从整体层面介绍了 React 构建 workInProgress 树的时机与方式。从初始化 workInProgress 对象到形成 workInProgress 树，应用程序会执行多次循环，每一次循环都是在解析 Fiber 结点并返回下一个要解析的结点。这个过程中会使用重要的「协调」算法，同时也会进行结点的 diff 操作。

如果当前工作单元的解析返回值为 null，说明当前的 Fiber 已经是叶子结点，则需要完成当前解析工作。完成当前解析工作的过程要收集副作用，同时也要为 HostComponent 类型的结点创建对应的 DOM 元素并将他们 append 到父结点上。

下一节将会介绍 React 如何解析常用类型的工作单元（Fiber 结点）以及完成工作单元。关于 React 如何解析每一种类型的 Fiber 结点以及「协调」算法的整体逻辑，还有 diff 操作的过程等内容将会在后面文章中进行详细介绍。

}

22 什么是 current 树和 workInProgress 树

24 React 如何解析工作单元以及什么时候完成工作单元