25 完成工作单元：创建 DOM 元素实例与收集副作用

更新时间：2020-09-24 09:33:02

> 才能一旦让懒惰支配，它就一无可为。——克雷洛夫

前言

上一节介绍了 React 如何解析不同类型的工作单元（Fiber 结点）以及什么时候完成工作单元。本节将会重点介绍 React 在完成工作单元时做的两件重要事情为`HostComponent`类型的 Fiber 结点**创建对应的 DOM 元素**和**收集副作用（Effect List）**。

首先，我们来看一下完成工作单元的入口函数`completeUnitOfWork`。

完成工作单元的入口 - completeUnitOfWork 函数

`completeUnitOfWork`函数是执行完成工作单元的入口，见代码示例 5.6.1。

```
// 源码位置：packages/react-reconciler/src/ReactFiberWorkLoop.js
function completeUnitOfWork(unitOfWork) {
  // unitOfWork为当前要完成的工作单元—Fiber结点
  workInProgress = unitOfWork;
  do {
    var current = workInProgress.alternate;
    // 执行完成工作单元的具体工作
    var next = completeWork(current, workInProgress, renderExpirationTime);
    if (next !== null) {
      // 如果检测到当前Fiber结点还有未完成的工作则返回该结点继续完成
      return next;
    }

    // 这里是收集副作用的模块 ...

    // 如果有兄弟结点则返回兄弟结点继续执行解析工作单元
    var siblingFiber = workInProgress.sibling;
    if (siblingFiber !== null) {
      return siblingFiber;
    }
    // 否则，返回父结点
    workInProgress = returnFiber;
  } while (workInProgress !== null);
  // ...
  return null;
}
```

代码示例 5.6.1 completeUnitOfWork 函数

在 completeUnitOfWork 函数中执行了一个遍历，这个遍历的方向是由子结点向父结点层层返回，程序的执行逻辑主要是下图 5.6.1 的右半部分。

图 5.6.1 解析与完成工作单元执行顺序示意图

在 completeUnitOfWork 函数中的每一次遍历中，通过执行 completeWork(current, workInProgress, renderExpirationTime) 来完成当前的工作单元，那么在 completeWork 函数中 React 做了哪些工作呢？

completeWork 函数 - 创建或更新 DOM 元素

completeWork 函数负责对不同类型的工作单元（Fiber 结点）分别处理，在这里我们重点介绍 React 对 HostComponent 和 HostText 类型 Fiber 结点的处理方式。见代码示例 5.6.2。

```
// 源码位置：packages/react-reconciler/src/ReactFiberCompleteWork.js
function completeWork(current, workInProgress, renderExpirationTime) {
  // workInProgress为当前完成的工作单元（Fiber结点）
  switch (workInProgress.tag) {
    case IndeterminateComponent:
    case FunctionComponent:
      break;
    case ClassComponent:
    // 处理context
    case HostComponent:
      {
        if (current !== null && workInProgress.stateNode != null) {
          // workInProgress.stateNode不为null，说明当前结点对应的DOM实例已经存在（一般为更新渲染），则执行更新DOM的逻辑
          updateHostComponent(current, workInProgress, type, newProps, rootContainerInstance);
        } else {
          // 如果当前结点对应的DOM实例不存在（一般为首次渲染），创建DOM实例并赋值到Fiber结点的stateNode属性上面
          var _instance5 = createInstance(type, newProps, rootContainerInstance, currentHostContext, workInProgress);
    // DOM实例append到其父DOM元素上
          appendAllChildren(_instance5, workInProgress, false, false);
          // 将DOM实例赋值到Fiber结点的stateNode属性上面
          workInProgress.stateNode = _instance5;
        }
        break;
      }
    case HostText:
      {
        var newText = newProps;
        if (current && workInProgress.stateNode != null) {
          // workInProgress.stateNode不为null，说明当前结点对应的DOM实例已经存在（一般为更新渲染），则执行更新DOM的逻辑
          var oldText = current.memoizedProps;
          updateHostText(current, workInProgress, oldText, newText);
        } else {
          // 如果当前结点对应的DOM实例不存在（一般为首次渲染），创建文本实例并赋值到Fiber结点的stateNode属性上面
          workInProgress.stateNode = createTextInstance(newText, _rootContainerInstance, _currentHostContext, workInProgress);
        }
        break;
      }
    default:
      throw ReactError(Error('Unknown unit of work tag. This error is likely caused by a bug in React'));
      break;
  }
  return null;
}
```

代码示例 5.6.2 completeWork 函数

前面提到过`HostComponent`和`HostText`类型的 Fiber 结点是有对应的真实 DOM 元素，因此在完成工作单元阶段 React 要为它们创建自己的 DOM 实例。

事实上，在完成工作单元的过程中分为了两种情况，分别是首次渲染和更新渲染。应用程序首次渲染时 React 为`HostComponent`和`HostText`类型的 Fiber 结点创建对应的 DOM 实例并将其赋值到结点上的`stateNode`属性上，而在应用程序更新渲染时 React 不需要为它们重新创建 DOM 实例，而是把新的值更新到 DOM 元素中。

`updateHostComponent`函数和`updateHostText`函数分别用于更新 DOM 元素。关于如何更新 DOM 元素请在下一章应用程序更新渲染流程中会有详细介绍。

在完成工作单元时 React 还有一项重要的工作就是收集副作用。那么，React 是如何收集副作用并生成副作用列表（Effect List）的呢？

完成工作单元时收集副作用

完成工作单元时收集副作用的逻辑依然是在 completeUnitOfWork 函数中执行，见代码示例 5.6.3。

```
function completeUnitOfWork(unitOfWork) {
  workInProgress = unitOfWork;
  do {
    var current = workInProgress.alternate;
    // ... 这里是执行完成工作单元的具体工作模块

    // 开始收集副作用，这里是将子树的副作用列表附加到父结点副作用列表里面
    if (returnFiber !== null && (returnFiber.effectTag & Incomplete) === NoEffect) {
      // workInProgress为当前Fiber结点
      // 将子树的所有副作用追加到父树的副作用列表中，子结点的完成顺序会影响副作用列表顺序
      if (returnFiber.firstEffect === null) {
        returnFiber.firstEffect = workInProgress.firstEffect;
      }
      if (workInProgress.lastEffect !== null) {
        if (returnFiber.lastEffect !== null) {
          returnFiber.lastEffect.nextEffect = workInProgress.firstEffect;
        }
        returnFiber.lastEffect = workInProgress.lastEffect;
      }

      // 下面会判断当前结点是否有副作用，如果有则将其收集到父结点的副作用列表中
      var effectTag = workInProgress.effectTag;
      // 在创建副作用列表时跳过NoWork和PerformedWork标识，因为他们只在开发环境被使用
      if (effectTag > PerformedWork) {
        if (returnFiber.lastEffect !== null) {
          returnFiber.lastEffect.nextEffect = workInProgress;
        } else {
          returnFiber.firstEffect = workInProgress;
        }
        returnFiber.lastEffect = workInProgress;
      }
    }

    // 如果有兄弟结点则返回兄弟结点继续执行解析工作单元（兄弟结点）
    // 否则，返回父结点
    workInProgress = returnFiber;
  } while (workInProgress !== null);
  // ...
  return null;
}
```

代码示例 5.6.3 completeUnitOfWork 函数收集副作用

还记得 React 中位运算的意义吗？比如下面这行代码：

```
// ps: NoEffect的值为0
// Incomplete为未完成标识
(returnFiber.effectTag & Incomplete) === NoEffect
```

如果上面这行代码返回值为 true，说明 returnFiber.effectTag & Incomplete 的运算结果为 0，也就是说在 returnFiber.effectTag 的二进制表示中 Incomplete 所在的那个位置为 0，也就是说当前的 effectTag 中不包含 Incomplete，即当前结点不存在未完成的工作。

在收集副作用的过程中主要有两种情况，第一种情况就是将子树上面的副作用列表连到父结点上面，第二种情况就是如果当前结点也有副作用标识，则将当前结点也连接到父结点的副作用列表中。

事实上，应用程序首次渲染时的副作用列表就是整个 workInProgress 树。React 收集完副作用列表后 workInProgress 树的结构见图 5.6.2。

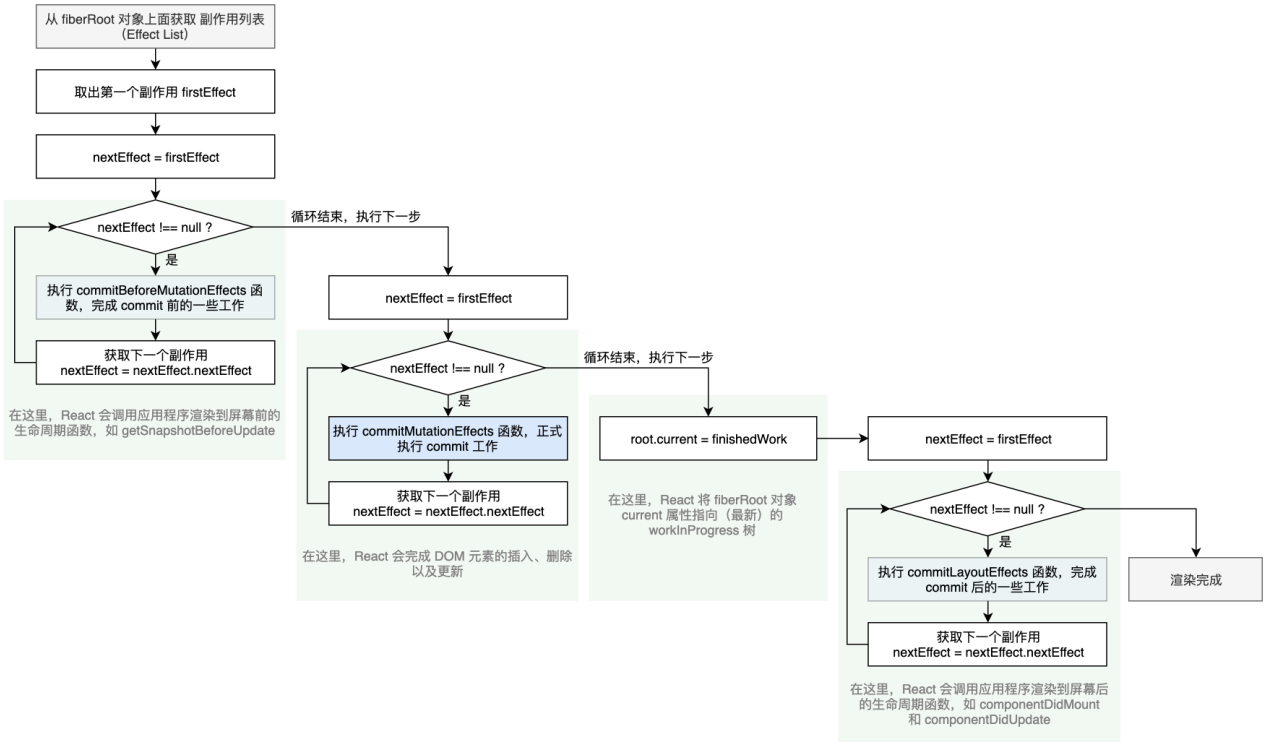

图 5.6.2 应用程序首次渲染完成工作循环后的 workInProgress 树

在图 5.6.2 中，workInProgress 树的根结点（也就是类型为 HostRoot 的结点）中的 firstEffect 和 lastEffect 属性指向的就是副作用列表。应用程序的首次渲染时副作用列表是整个 workInProgress 树，因为整个 workInProgress 树的结点中携带的内容都需要更新到屏幕，而在应用程序更新渲染时副作用列表将会是 workInProgress 树的子集。

小结

本节主要介绍了在完成工作单元时 React 重点做的两件事，分别是收集副作用以及为 HostComponent 类型的 Fiber 结点创建对应的 DOM 元素（或更新对应 DOM 元素中的属性）。在收集好的副作用列表中每个 HostComponent 类型 Fiber 结点的 stateNode 属性中存储了当前结点对应的 DOM 实例。那么，React 下一步要做的就是将副作用列表中的所有 DOM 实例更新到屏幕中。

下一节将会介绍 React 如何将副作用列表更新到屏幕。

}

24 React 如何解析工作单元以及什么时候完成工作单元

26 React 如何将副作用更新到屏幕