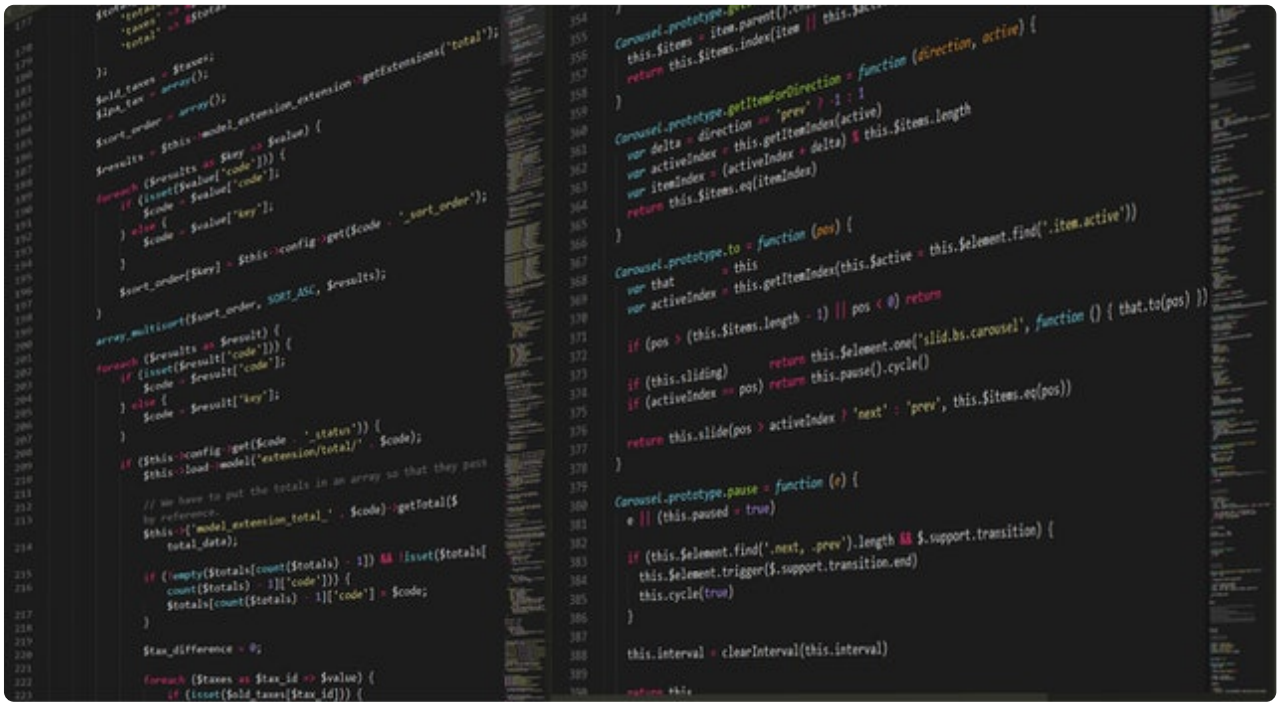
24 React 如何解析工作单元以及什么时候完成工作单元

更新时间：2020-09-22 09:52:11



> 知识犹如人体的血液一样宝贵。——高士其

前言

上一节我们从整体层面介绍了 React 通过工作循环构建 workInProgress 树，其中有两个关键环节就是解析工作单元和完成工作单元。本节将主要介绍 React 如何处理不同类型的工作单元（Fiber 结点）以及如何完成工作单元。

首先我们先看一下 React 元素的类型和 Fiber 结点的类型。

React 元素类型与 Fiber 结点类型

React 元素类型（`type`）主要包括宿主元素（`div`，`span`）和组件元素（`UpdateCounter`），这两种类型元素转换为 Fiber 结点后对应的类型分别是`HostComponent`和`ClassComponent`。当然，Fiber 结点还有更多的类型，见代码示例 5.4.3。

```
// 对应函数组件元素
var FunctionComponent = 0;
// 对应class组件元素
var ClassComponent = 1;
var IndeterminateComponent = 2;
// 根结点
var HostRoot = 3;
var HostPortal = 4;
// 对应宿主组件元素
var HostComponent = 5;
// 文本元素
var HostText = 6;
var Fragment = 7;
var Mode = 8;
var ContextConsumer = 9;
var ContextProvider = 10;
var ForwardRef = 11;
var Profiler = 12;
var SuspenseComponent = 13;
var MemoComponent = 14;
var SimpleMemoComponent = 15;
var LazyComponent = 16;
var IncompleteClassComponent = 17;
var DehydratedSuspenseComponent = 18;
var SuspenseListComponent = 19;
var FundamentalComponent = 20;
```

代码示例 5.5.1 React Fiber 结点类型

后面我们将主要讨论`HostRoot`，`ClassComponent`和`HostComponent`类型的 Fiber 结点。应用程序首次渲染时，将根组件元素转换为多个层级的 Fiber 结点过程中，第一步要做的就是解析 workInProgress 树的根（`HostRoot`）结点。

解析不同类型的 Fiber 结点

> 注意：下面文字描述中的「类型」两字有的是表示 Fiber 结点中的`tag`属性，有的是则表示`elementType`属性。

1. 解析 HostRoot 类型的结点

React 解析`HostRoot`类型结点的工作主要在`updateHostRoot`函数中执行，见代码示例 5.5.2。

```
// 源码位置：packages/react-reconciler/src/ReactFiberBeginWork.js
function updateHostRoot(current, workInProgress, renderExpirationTime) {
  // 形参workInProgress传入的是当前解析的Fiber结点，此处的Fiber结点类型为HostRoot
  // 获取updateQueue
  var updateQueue = workInProgress.updateQueue;
  // ...
  // 处理更新队列在第二章已经有过介绍哦
  processUpdateQueue(workInProgress, updateQueue, nextProps, null, renderExpirationTime);
  // 处理完更新队列后第一个更新对象被赋值到workInProgress.memoizedState.element
  var nextState = workInProgress.memoizedState;
  // 获取根组件元素
  var nextChildren = nextState.element;
  // 协调，用于获取下一个 Fiber 结点
  reconcileChildren(current, workInProgress, nextChildren, renderExpirationTime);
  // 返回下一个Fiber结点
  return workInProgress.child;
}
```

代码示例 5.5.2 解析 HostRoot 类型的结点

解析 HostRoot 类型结点的关键是处理更新队列（ processUpdateQueue ）获取根组件元素 element，element 就是要传入到协调算法中进行解析的 nextChildren。

2. 解析 ClassComponent 类型的结点

应用程序首次渲染时解析 HostRoot 类型（tag）结点完成后返回的一般是 ClassComponent 类型的结点（也就根组件元素对应的 Fiber 结点），这种类型的 Fiber 结点解析过程中是要进行组件实例化的，解析 ClassComponent 类型结点的逻辑见代码示例 5.5.3。

```
// 源码位置：packages/react-reconciler/src/ReactFiberBeginWork.js
function updateClassComponent(current, workInProgress, Component, nextProps, renderExpirationTime) {
  // 形参workInProgress传入的是当前解析的Fiber结点，此处的Fiber结点类型为ClassComponent
  // instance为组件实例
  var instance = workInProgress.stateNode;
  var shouldUpdate = void 0;
  if (instance === null) {
    // 如果instance为null，说明是首次渲染，此时组件还没有被实例化，然后进行组件实例化
    constructClassInstance(workInProgress, Component, nextProps, renderExpirationTime);
    // 组件实例化完成后可调用getDerivedStateFromProps生命周期函数
    mountClassInstance(workInProgress, Component, nextProps, renderExpirationTime);
    shouldUpdate = true;
  }
  // ...
  // nextUnitOfWork为要返回的下一个Fiber结点
  var nextUnitOfWork = finishClassComponent(current, workInProgress, Component, shouldUpdate, ...);
  // ...
  return nextUnitOfWork;
}

function constructClassInstance(workInProgress, ctor, props, renderExpirationTime) {
  // 形参workInProgress传入的是当前解析的Fiber结点，此处的Fiber结点类型为ClassComponent
  // 组件实例化
  var instance = new ctor(props, context);
  adoptClassInstance(workInProgress, instance);
  // ...
  return instance;
}

function adoptClassInstance(workInProgress, instance) {
  // 为组件实例添加更新器updater
  instance.updater = classComponentUpdater;
  // 将组件实例存储到当前Fiber结点的stateNode属性上
  workInProgress.stateNode = instance;
  // ...
}

function finishClassComponent(current, workInProgress, Component, shouldUpdate, hasContext, ...) {
  // ...
  var nextChildren = instance.render();
 reconcileChildren(current, workInProgress, nextChildren, renderExpirationTime);
  // 返回下一个Fiber结点
  return workInProgress.child;
}
```

代码示例 5.5.3 解析 ClassComponent 类型的结点

解析 ClassComponent 类型（tag）的结点时先从当前 Fiber 结点的 stateNode 属性上面取组件实例 instance，如果 instance 的值为 null 则说明是首次渲染，组件还没有被实例化。React 使用 constructClassInstance 函数完成组件实例化，同时为组件实例添加更新器（然后就可以调用 setState 方法，第六章会有介绍）。

组件实例生成后，它的核心作用之一就是在 finishClassComponent 函数内部执行 instance.render() 获取要传入到协调算法中进行解析的 nextChildren。

3. 解析 HostComponent 类型的结点

解析 ClassComponent 类型结点完成后返回的一般是 HostComponent 类型（tag）的结点，这种结点的具体类型（elementType）对应的是真实的 DOM 标签（如：div 和 span 等）。解析 HostComponent 类型（tag）结点的逻辑见代码示例 5.5.4。

```
function updateHostComponent(current, workInProgress, renderExpirationTime) {
  // 形参workInProgress传入的是当前解析的Fiber结点，此处的Fiber结点类型为HostComponent
  var type = workInProgress.type;
  var nextProps = workInProgress.pendingProps;
  var prevProps = current !== null ? current.memoizedProps : null;
 // HostComponent类型的Fiber结点，直接从其props上面获取nextChildren
  var nextChildren = nextProps.children;
  // ...
  reconcileChildren(current$$1, workInProgress, nextChildren, renderExpirationTime);
  return workInProgress.child;
}
```

代码示例 5.5.4 解析 HostComponent 类型的结点

还记得 React 元素的内部结构吗？现在来回顾一下，见代码示例 5.5.5。

```
element = {
  type: 'button',
  props: {
    className: 'button button-blue',
    children: {
      type: 'b',
      props: {
        children: 'OK!'
      }
    }
  }
}
```

代码示例 5.5.5 React 元素结构

解析 HostComponent 类型结点，可从当前结点对应的元素 props 中获取要传入到协调算法中进行解析的 nextChildren。

这里要强调一下，无论是解析 HostRoot 类型的结点，还是解析 HostComponent 类型的结点，这个过程中获取的 nextChildren 均为 React 元素。在执行协调算法的过程中，React 将该元素转换成对应的 Fiber 结点并返回。

4. 解析 HostText 类型的结点

当解析到元素的叶子元素时，比如代码示例 5.5.5 中的 b 元素的 children 为文本 OK!，此时对应的 Fiber 结点类型为 HostText。React 认为文本类型的 Fiber 结点已经是终点，不会再有孩子结点，直接返回 null，见代码示例 5.5.6。

```
function updateHostText(current, workInProgress) {
  // ....
  // 已经到了终点，需要立刻完成该工作单元
  return null;
}
```

代码示例 5.5.6 解析 HostText 类型的结点

当解析到 HostText 类型的 Fiber 结点时，React 认为当前结点已经是链路的终点了，需要完成当前链路。现在，我们来思考一个问题，上面解析结点的过程有没有遗漏的地方呢？

我们上面介绍的解析结点的过程都是按照 child 路线从上向下层层分析的，那如果遇到某个元素有多个 child，React 肯定不能只解析其中一个。那么，其他的 child （也就是被解析结点的 sibling）是什么时候解析的呢？答案就是在完成工作单元时。

完成工作单元的时机

还记得在何处执行 completeUnitOfWork 函数完成工作单元的吗？见代码示例 5.5.7。

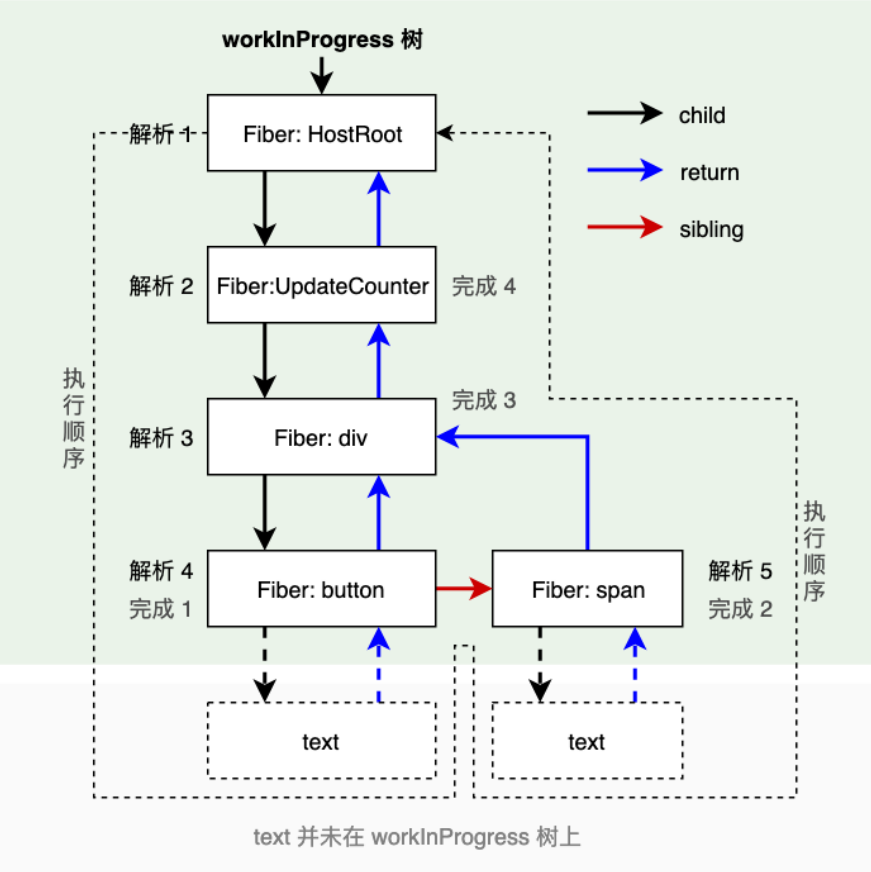
```
function performUnitOfWork(unitOfWork) {
  // 形参unitOfWork传入的是当前解析的Fiber结点
  // 解析工作单元，如果unitOfWork的tag值为HostText，则返回null
  next = beginWork(current, unitOfWork, renderExpirationTime);
  // ...
  if (next === null) {
    // 如果next为null，则完成当前工作单元
    next = completeUnitOfWork(unitOfWork);
  }
  // ...
  return next;
}
```

代码示例 5.5.7 完成工作单元的执行时机

completeUnitOfWork 函数用于完成工作单元，其实我们可以想到，**在完成工作单元过程中一件重要的事情就是检查当前结点是否有兄弟结点，如果没有兄弟结点就检查父结点有没有兄弟结点**。当然，在完成工作单元过程中还有一件重要的事就是为 HostComponent 类型的结点创建 DOM 元素实例。

小结

本节主要介绍了 React 解析工作单元中具体是怎么处理结点的以及何时开始完成工作单元。现在我们来总结一下解析工作单元与完成工作单元的整个执行顺序什么样的，见下图 5.5.1。



text 并未在 workInProgress 树上

在工作循环的过程中，React 总是先从外部根节点开始解析，当解析到叶子结点时就要开始完成工作单元。完成工作单元时如果有兄弟结点时则需要继续解析兄弟结点，然后完成兄弟结点后会逐层向上完成工作单元。

看到这里还有一个问题没有解答，就是在完成工作单元时 React 做了哪些事情呢？上面说了，React 会检查当前结点是否有兄弟结点，如果有则继续解析其兄弟结点。除此之外还有其他重要的事情，下一节将会详细介绍 React 完成工作单元时的工作内容。

}

23 React 如何构建 workInProgress 树

25 完成工作单元：创建 DOM 元素实例与收集副作用