27 React 应用程序更新渲染时内部运行流程概述

更新时间：2020-09-30 10:47:34

> 世界上最快乐的事，莫过于为理想而奋斗。——苏格拉底

前言

上一章主要介绍了应用程序首次渲染时 React 内部所做的一些工作，包括构建 fiberRoot 对象，构建 workInProgress 树，收集副作用列表以及将副作用渲染到屏幕等。事实上，应用程序首次渲染时的副作用列表就是整个 workInProgress 树，而在应用程序更新渲染时，副作用列表会是 workInProgress 树的子集。那么，在应用程序更新渲染时 React 又做了哪些不同的工作呢？

本章将会介绍应用程序在更新渲染时 React 的主要工作内容。现在，我们先来看看应用程序首次渲染结束后 React Fiber 架构在内存中的状态。

应用程序首次渲染完成后 React Fiber 架构的形态

前面提到过应用程序首次渲染完成后 fiberRoot 对象上面的`current`属性会指向 workInProgress 树，而原有的 workInProgress 树将会被制为`null`，新的 workInProgress 树只有一个`HostRoot`类型的根结点。此时 fiberRoot 对象的内存结构如下图 6.1.1。

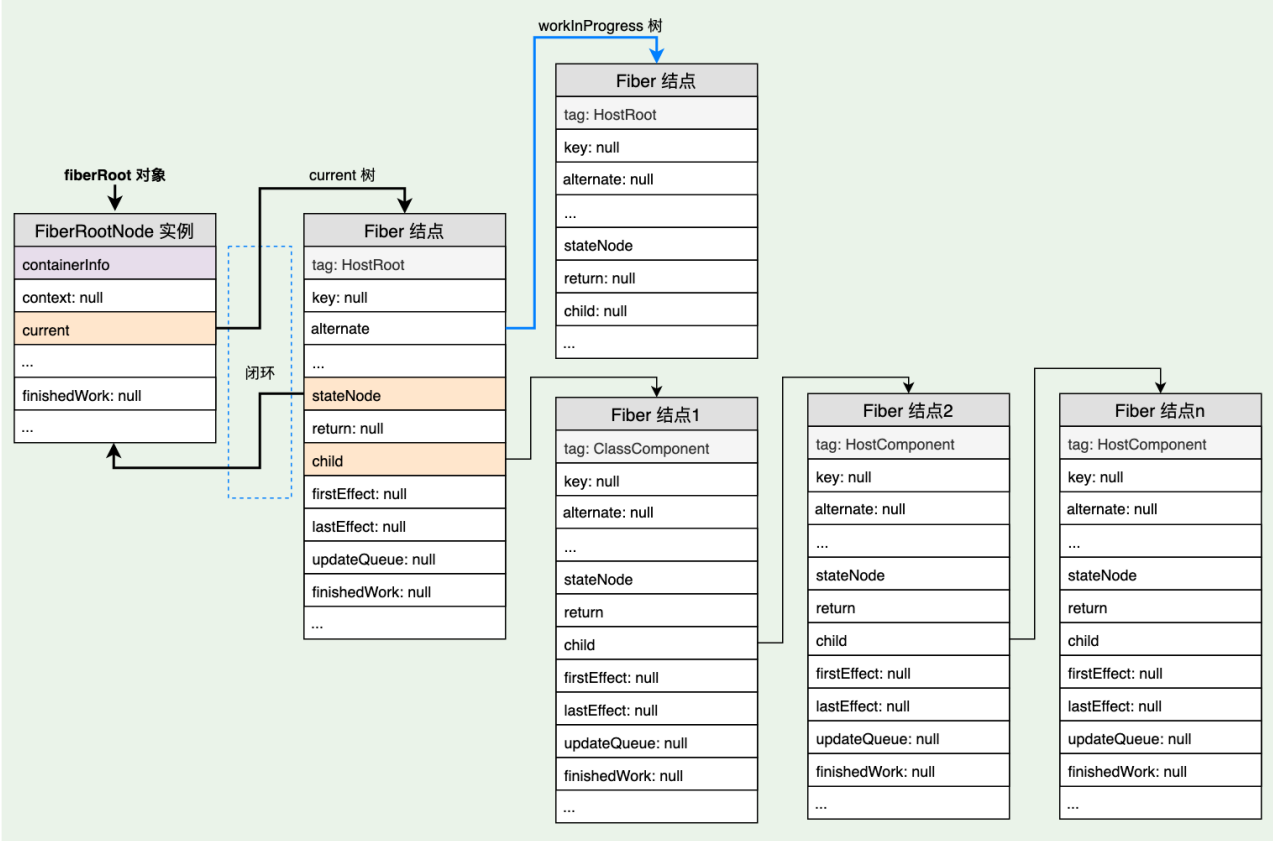

图 6.1.1 应用程序首次渲染完成后的 fiberRoot 对象内部结构

事实上，此时 fiberRoot 对象中的`current`属性指向的是上一次（首次）渲染结束后的 workInProgress 树。应用程序渲染到屏幕后，也就具备了交互能力，如果某个用户事件触发了更新逻辑，React 将会进入处理更新渲染的流程。那么，React 是如何申请与处理更新请求的呢？

React 如何申请与处理更新请求

我们要想更新页面中的信息，需要在程序中使用`setState`操作来进行更新请求，见代码示例 6.1.1。

```
class UpdateCounter extends React.Component{
  constructor(props) {
  super(props);
  this.state = {
   count: 0,
   text: '点击计数'
  };
 }

  handleClick() {
    // 通过setState申请更新
  this.setState({
   count: this.state.count + 1
  });
  }

  render() {
    return (
      <div className="wrap-box">
        <button key="1" onClick={() => {this.handleClick();}}>{this.state.text}</button>
      </div>
    )
  }
}
```

代码示例 6.1.1 程序向 React 发更新请求

根据代码示例 6.1.1 我们要思考一下几个问题？

1. 程序中的 this 具体指的是什么？
2. 为什么 this 可以调用 setState 函数？

在解答上面问题之前，我们回顾一下前面的一些知识点：

1. 只有 class 类型组件才能被实例化。
2. class 类型的组件继承于 React.Component 。

下面是 React 对 Component 构造函数的定义，见代码示例 6.1.2。

```
// 源码位置：packages/react/src/ReactBaseClasses.js
function Component(props, context, updater) {
  this.props = props;
  this.context = context;
  this.refs = emptyObject;
  // 更新器
  this.updater = updater || ReactNoopUpdateQueue;
}
// 在Component构造函数的原型对象上面定义了setState函数
Component.prototype.setState = function (partialState, callback) {
  // ...
  this.updater.enqueueSetState(this, partialState, callback, 'setState');
};
```

代码示例 6.1.2 React Component 构造函数的定义

前面文章中提到 class 类型的组件在解析工作单元的时候会被实例化，因此代码示例 6.1.1 中的 this 指的是当前组件的实例对象。由于 class 类型的组件继承于 React.Component ，而 React.Component 的原型对象上面定义了 setState 方法，因此组件实例可以访问到该方法。

在 setState 函数中调用了更新器（updater）的 enqueueSetState 函数，该函数的定义见代码示例 6.1.3。

```
// 源码位置：packages/react-reconciler/src/ReactFiberClassComponent.js
var classComponentUpdater = {
  isMounted: isMounted,
  enqueueSetState: function (inst, payload, callback) {
    // 形参inst就是开发者程序中的this
    var fiber = get(inst);
    var currentTime = requestCurrentTime();
    var suspenseConfig = requestCurrentSuspenseConfig();
    var expirationTime = computeExpirationForFiber(currentTime, fiber, suspenseConfig);
  // 根据过期时间创建更新对象
    var update = createUpdate(expirationTime, suspenseConfig);
    update.payload = payload;
    // 将更新对象加入到更新队列
    enqueueUpdate(fiber, update);
    // 调度更新任务
    scheduleWork(fiber, expirationTime);
  },
  // ...
}
```

代码示例 6.1.3 React 更新器的定义

React 更新器中会根据当前的组件实例从 current 树上面找到对应的 Fiber 结点，然后创建更新对象并将更新对象加入到**当前 Fiber 结点**的更新队列中。此时的 fiberRoot 对象内部结构如图 6.1.2。

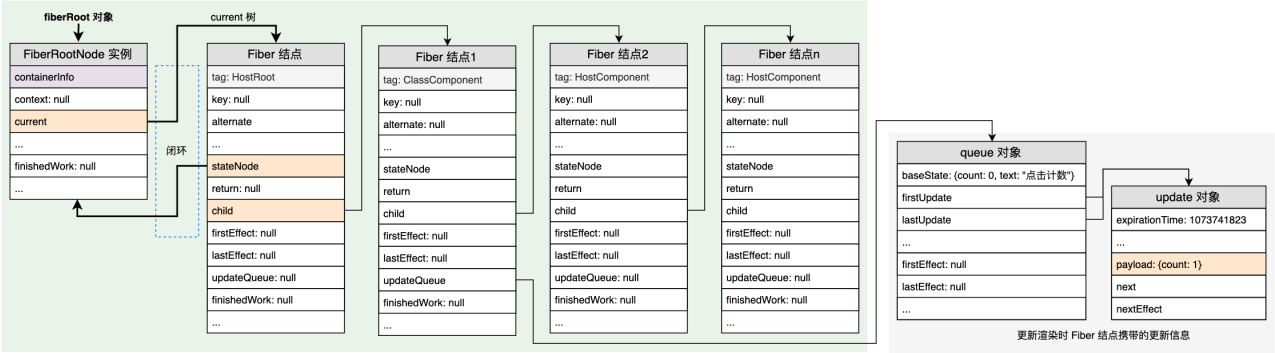

图 6.1.2 将更新加入对应结点的更新队列后 fiberRoot 对象结构

随后，React 向任务调度器申请更新任务执行权，获取任务执行后则任务整个渲染流程的 render 阶段。

应用程序的更新渲染流程没有 prerender 阶段

React 应用程序更新渲染时 React Fiber 架构的实体 fiberRoot 对象已经存在与内存中，因此不需要重新构建 fiberRoot 对象。当应用程序执行了 setState 操作后，React 内部执行流程将通过任务调度直接进入 render 阶段。在更新渲染过程中的 render 阶段，React 同样需要完成构建 workInProgress 树以及收集副作用列表工作。但是，相对于应用程序的首次渲染，更新渲染过程中 React 会重点分析结点的前后变化。

应用程序更新渲染时要做结点 diff

应用程序更新渲染时 Fiber 结点的前后变化 diff 不是几步就能完成的，这是一个很深的处理链路，这个链路也被称为「协调」过程。在应用程序首次渲染时也会经历「协调」过程，由于不存在上一次的结点状态，因此也就不涉及到前后结点对比环节。

小结

本节简单介绍了应用程序更新渲染时相对于首次渲染时的不同点以及 React 的重点工作方向。应用程序更新渲染时 React 如和构建 workInProgress 树，收集副作用以及「协调」过程的详细内容在后面文章中会陆续进行详细介绍。

}

26 React 如何将副作用更新到屏幕

28 React 应用程序更新渲染时如何构建 workInProgress 树？