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

更新时间：2020-10-09 10:19:29

> 对自己不满是任何真正有才能的人的根本特征之一。——契诃夫

前言

上一节提到当应用程序触发了 `setState( ... )` 操作后，React 接收的更新请求并获得更新任务执行权后就进入了整个渲染流程的 render 阶段。在更新渲染过程的 render 阶段 React 也要构建 workInProgress 树，此时的构建流程和首次渲染时基本相同。但是，在应用程序的更新渲染时，React 会更加关心结点前后 `state` 和 `props` 的变化。本节将会介绍应用程序更新渲染时 React 如何构建 workInProgress 树。

从 current 树上面获取 workInProgress 对象

```
workInProgress = current.alternate
// 重置相关属性
workInProgress.pendingProps = pendingProps;
workInProgress.effectTag = NoEffect;
workInProgress.nextEffect = null;
workInProgress.firstEffect = null;
workInProgress.lastEffect = null;
...
```

代码示例 6.2.1 获取 workInProgress 对象

前面提到，React 应用程序首次渲染完成后 fiberRoot 对象上面会存在 current 树。其中 current.alternate 指向的是下一次更新渲染时的 workInProgress 树，渲染开始执行前它是只有一个 HostRoot 类型的 Fiber 结点的对象。

有了 workInProgress 对象后，下一步就是通过工作循环完善该对象，这个过程依然是要进行解析工作单元以及完成工作单元。

解析工作单元

在应用程序更新渲染时，React 解析工作单元（Fiber 结点）的主要关注点会是 ClassComponent 类型的 Fiber 结点。解析 ClassComponent 类型的结点时需要完成以下几方面的工作。

处理结点的更新队列（updateQueue）

上一节提到，当某个 class 类型的组件内部触发了 setState(...) 操作后，React 会创建对应的更新对象并将其加入到该组件对应的 Fiber 结点的更新队列中。此时的 fiberRoot 结构对象如下图 6.2.1。

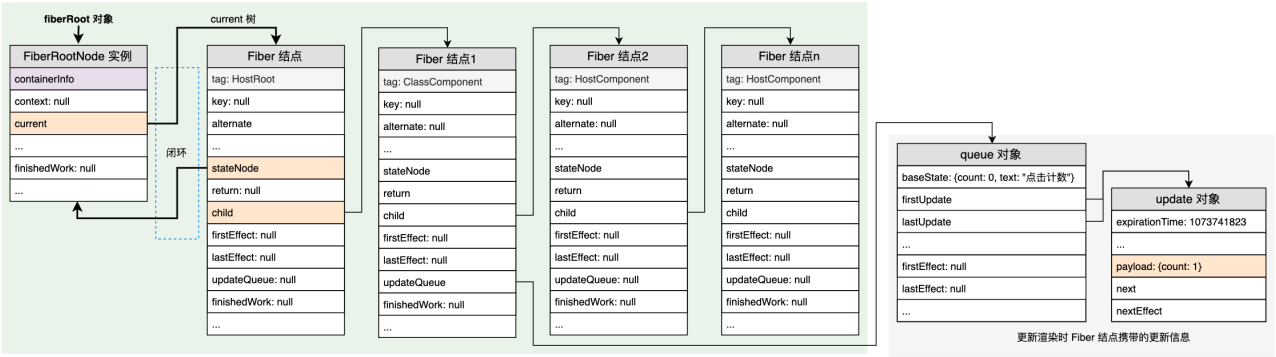

图 6.2.1 将更新加入对应结点的更新队列后 fiberRoot 对象结构

应用程序执行到 render 阶段，在解析工作单元时 React 会处理该结点的更新队列，目的就是获取更新对象中最新的 payload 信息，见代码示例 6.2.2。

```
function updateClassInstance(current, workInProgress, ctor, newProps, renderExpirationTime) {
  // 形成workInProgress为当前解析的Fiber结点
  var instance = workInProgress.stateNode;
  var oldProps = workInProgress.memoizedProps;
  var oldState = workInProgress.memoizedState;
  var newState = instance.state = oldState;
  var updateQueue = workInProgress.updateQueue;
  // 如果当前Fiber结点中的updateQueue不为null，则处理该updateQueue
  if (updateQueue !== null) {
    // 处理updateQueue
    processUpdateQueue(workInProgress, updateQueue, newProps, instance, renderExpirationTime);
    // 处理updateQueue后，将会获得新的state
    newState = workInProgress.memoizedState;
  }

  // ...
}
```

代码示例 6.2.2 处理该结点的 updateQueue

在 processUpdateQueue 函数中会处理当前 Fiber 结点的更新队列并将最终的 state 赋值到当前 Fiber 结点的 memoizedState 属性中。

检查结点是否需要更新

处理完该结点的 updateQueue 后就可以得到最新的 state，然后 React 需要对比前后 state 以决定当前结点是否需要更新，见代码示例 6.2.2。

```
function updateClassInstance(current, workInProgress, ctor, newProps, renderExpirationTime) {
  // 形成workInProgress为当前解析的Fiber结点
  // 处理更新队列模块 ...

  // 检查当前结点是否需要更新
  var shouldUpdate = checkHasForceUpdateAfterProcessing() || checkShouldComponentUpdate(workInProgress, ctor, oldProps, newProps, oldState, n
ewState, nextContext);
}

function checkShouldComponentUpdate(workInProgress, ctor, oldProps, newProps, oldState, newState, nextContext) {
  var instance = workInProgress.stateNode;
  if (typeof instance.shouldComponentUpdate === 'function') {
    // 如果当前组件中有shouldComponentUpdate生命周期函数，则优先执行该函数
    var shouldUpdate = instance.shouldComponentUpdate(newProps, newState, nextContext);
    return shouldUpdate;
  }
  // 如果当前组件中没有shouldComponentUpdate生命周期函数，则进行前后props和state对比
  if (ctor.prototype && ctor.prototype.isPureReactComponent) {
    return !shallowEqual(oldProps, newProps) || !shallowEqual(oldState, newState);
  }

  return true;
}
```

代码示例 6.2.3 检查该结点是否需要更新

检查工作单元是否需要更新首先根据该结点是否有 shouldComponentUpdate 生命周期函数，有则调用该函数并取其执行结果，没有则进行结点的 props 和 state 前后对比。这里在进行对象是否相等比较时，React 定义了 shallowEqual 方法，那么 **React 是怎么比较引用类型数据是否相等**的呢？见代码示例 6.2.4。

```
// 源码位置：packages/shared/shallowEqual.js
function shallowEqual(objA, objB) {
  if (is(objA, objB)) {
    return true;
  }

  if (typeof objA !== 'object' || objA === null || typeof objB !== 'object' || objB === null) {
    return false;
  }

  var keysA = Object.keys(objA);
  var keysB = Object.keys(objB);

  if (keysA.length !== keysB.length) {
    return false;
  }

  // Test for A's keys different from B.
  for (var i = 0; i < keysA.length; i++) {
    if (!Object.prototype.hasOwnProperty.call(objB, keysA[i]) || !is(objA[keysA[i]], objB[keysA[i]])) {
      return false;
    }
  }

  return true;
}

function is(x, y) {
  return x === y && (x !== 0 || 1 / x === 1 / y) || x !== x && y !== y;
}
```

代码示例 6.2.4 React 比较引用类型数据是否相等

shallowEqual 函数的逻辑并不复杂，可以看做是教科书级别的引用类型数据相等检测方法，我们应该熟记该方法。

协调（reconcile）与结点 diff 过程

```
// 执行「协调算法」，获取下一个Fiber结点
reconcileChildren(current, workInProgress, nextChildren, renderExpirationTime);
```

当检测到 class 组件对应的 Fiber 结点需要更新后，React 会通过 instance.render() 获取组件元素，将该组件元素传入到「协调算法」中，然后获得下一个 Fiber 结点。获得下一个 Fiber 结点的过程也是 React 进行（新的）元素与现有 Fiber 结点进行 diff 处理的过程。下节将会对这部分内容进行详细介绍。

在应用程序更新渲染过程中，render 阶段还有一项重要工作就是为结点标记 effectTag。

为结点标记 effectTag

我们知道，React 中的更新类型包括了插入（ Placement ），更新（ Update ），插入并更新（ PlacementAndUpdate ）以及删除（ Deletion ）等多种类型。那么，这些类型的 effectTag 是怎么以及何时被标记到 Fiber 结点中的呢？下面列举了一下主要的标记方式。

标记插入（ Placement ）

在「协调」获取下一个 Fiber 结点的过程中调用 placeChild 函数为下一个结点标记 Placement，见代码示例6.2.5。

```
function placeChild(newFiber, lastPlacedIndex, newIndex) {
 // ...
  newFiber.effectTag = Placement;
  // ...
  return lastPlacedIndex;
}

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

代码示例 6.2.5 为Fiber结点标记 Placement

placeChild 函数一般在「协调」处理数组元素时被调用并返回要插入的位置索引，数组元素相对于现有 Fiber 结点结构有了新的元素时，需要将新元素对应的 Fiber 结点标记为 Placement 。

而 placeSingleChild 函数一般是在有新的结点生成时会被调用，比如程序中控制元素显示的条件值变为 true 。

标记更新（ Update ）

```
function markUpdate(workInProgress) {
 // 该函数为结点标记更新，也会会将Placement转化为PlacementAndUpdate.
 workInProgress.effectTag |= Update;
}

updateHostComponent = function (current, workInProgress, type, newProps, rootContainerInstance) {
 var instance = workInProgress.stateNode;
 var currentHostContext = getHostContext();
 // 为HostComponent类型结点对应的DOM实例计算更新内容，如何计算见下文
 var updatePayload = prepareUpdate(instance, type, oldProps, newProps, rootContainerInstance, currentHostContext);
  workInProgress.updateQueue = updatePayload;
  // 如果DOM实例的更新内容不为空，则向当前结点标记Update
  if (updatePayload) {
    markUpdate(workInProgress);
  }
}
```

代码示例 6.2.6 为Fiber结点标记 Update

在完成工作单元时，使用 updateHostComponent 函数完成 HostComponent 类型的工作单元（Fiber 结点）。这时 React 使用 prepareUpdate 方法对比结点对应 DOM 实例的属性及内容是否发生变化，如果有变化则为该结点标记 Update 。

标记插入并更新（ PlacementAndUpdate ）

如果一个结点既被标记了 Placement 也被标记了 Update ，那么该结点的 effectTag 的二进制对应的就是 Placement AndUpdate 。

标记删除（ Deletion ）

```
function deleteChild(returnFiber, childToDelete) {
 // ...
 childToDelete.nextEffect = null;
 childToDelete.effectTag = Deletion;
}
```

代码示例 6.2.7 为Fiber结点标记 Deletion

在「协调」数组元素时，如果新的数组元素比此处 Fiber 树的结点数量少时，那么丢下的那个结点将会被调用 deletChild 函数标记为 Deletion 。还有一种情况就是程序中控制元素显示的条件值变为 false 也会导致当前结点被标记为 Deletion 。

完成工作单元—计算现有 DOM 实例与新的结点之间的属性变化

应用程序更新渲染时，在完成工作单元阶段相对于首次渲染 React 需要重点计算现有的 DOM 实例与新的结点之间的属性变化，此处逻辑主要在 diffProperties 函数中执行，见代码示例 6.2.8。

```
// 源码位置：packages/react-dom/src/client/ReactDOMComponent.js
function diffProperties(domElement, tag, lastRawProps, nextRawProps, rootContainerElement) {
 // 存储diff后最终的更新信息
 var updatePayload = null;
 // 现有的（上一个）props
 var lastProps = void 0;
 // 新的props
 var nextProps = void 0;

 // 第一步：收集现有的props和新的props

 // 如果是表单元素的结点，则分别计算器更新前后其value属性
```

```
  switch (tag) {
    case 'input':
      lastProps = getHostProps(domElement, lastRawProps);
      nextProps = getHostProps(domElement, nextRawProps);
      updatePayload = [];
      break;
    case 'option':
    case 'select':
      // ...
    default:
      // 非表单元素则从形参中获取更新前后的属性值
      lastProps = lastRawProps;
      nextProps = nextRawProps;
      break;
  }

  // 第二步：收集当前结点所有的style属性

  var propKey = void 0;
  var styleName = void 0;
  var styleUpdates = null;
  for (propKey in lastProps) {
    // 处理style的属性
    if (propKey === 'style') {
      var lastStyle = lastProps[propKey];
      for (styleName in lastStyle) {
        if (lastStyle.hasOwnProperty(styleName)) {
          // 为style对象添加对应的样式名属性，默认值为空
          styleUpdates[styleName] = '';
        }
      }
    }
    // ...
  }

  // 第三步：计算当前结点所有新的的style属性并将各个的值更新到对应styleUpdates[styleName]中

  for (propKey in nextProps) {
    var nextProp = nextProps[propKey];
    var lastProp = lastProps != null ? lastProps[propKey] : undefined;
    // ...
    if (propKey === 'style') {
      for (styleName in nextProp) {
        if (nextProp.hasOwnProperty(styleName) && lastProp[styleName] !== nextProp[styleName]) {
          // 将需要更新的样式添加到样式更新对象
          styleUpdates[styleName] = nextProp[styleName];
        }
      }
    }
    // ...
  }
  // 将styleUpdates添加到updatePayload中
  if (styleUpdates) {
    (updatePayload = updatePayload || []).push('style', styleUpdates);
  }
  // 返回updatePayload
  return updatePayload;
}
```

代码示例 6.2.8 diffProperties 函数用于计算更新前后的DOM属性变化

`diffProperties` 函数主要用于计算（`HostComponet` 类型的）结点更新前后所有的 style 属性是否发生变化，并将所有发生变化的属性以及对应的值收集起来。这些属性变化的数据在 commit 结点被统一更新到 DOM。

小结

应用程序更新渲染构建 workInProgress 树的整体流程与首次渲染时基本相同。不同的地方就是对于 ClassComponent 类型的结点，React 会先处理该结点的更新队列并获取最新的 state，然后判断该结点是否需要执行后续的更新逻辑。如果 ClassComponent 类型的结点需要执行更新，则通过执行「协调」逻辑获取下一个 Fiber 结点，这个过程中也会进行新元素与当前 Fiber 结点 diff 操作。结点 diff 处理的过程中 React 会为结点标记对应的 effectTag，最常用的几种 effectTag 包括插入（ Placement ），更新（ Update ），插入并更新（ PlacementAndUpdate ）以及删除（ Deletion ）等。

下一节将会介绍应用程序更新渲染时 React「协调」与结点 diff 的详细过程。

}

27 React 应用程序更新渲染时内部运行流程概述

29 React v16 版的协调算法是什么样的呢？