# 29 React v16 版的协调算法是什么样的呢？

更新时间：2020-10-12 10:38:03

> 读书给人以快乐、给人以光彩、给人以才干。——培根

## 前言

无论是应用程序首次渲染还是更新渲染，在解析工作单元的时候 React 均需要执行「协调」算法以获取下一个 Fiber 结点。那么 React 中的「协调」算法到底是什么样的呢？本节将会主要介绍 React Fiber 架构下「协调」算法的核心逻辑。

在 React v16 版中，「协调」算法的代码在一个独立的包（react-reconciler）里面。React 官方将该算法独立成包的目的是为了能够让第三方可以直接使用该算法，并且提供了对应的 Api，见代码示例 6.3.1。

```
const Reconciler = require('react-reconciler');

const HostConfig = {
  // 需要在这里实现以下方法
};
const MyRenderer = Reconciler(HostConfig);
const RendererPublicAPI = {
  render(element, container, callback) {
    // 调用MyRenderer.updateContainer()执行根结点的更新。
    // 可以参考ReactDOM, React Native, 以及 React ART等示例。
  }
};

module.exports = RendererPublicAPI;
```

代码示例 6.3.1 独立使用 react-reconciler 包

下面，我们来看一下协调算法在 React 应用程序是如何工作的。

# 协调（reconcile）的核心逻辑

**什么是「协调」？**

在前面章节中已经提到了「协调」逻辑，应用程序执行到 render 阶段构建 workInProgress 树的过程中，每一次工作循环的目的—获取下一个 Fiber 结点的工作都是经过「协调」处理 React 元素来完成。**需要注意的是，工作循环的主要任务就是处理当前 Fiber 结点然后返回下一个 Fiber 结点，在这个过程中也会完成结点 diff 处理。**

个人认为 React 的协调过程不单单由一个算法完成，而是有众多判断逻辑以及独立的子算法共同完成。

**协调的前置工作 —对不同类型的 Fiber 结点分别解析以获得其子元素**

这个逻辑是开始工作单元解析时，在 `beginWork` 函数中根据当前 Fiber 结点的类型分别调用各自的解析函数，见代码示例 6.3.2。

```
// 源码位置：packages/react-reconciler/src/ReactFiberBeginWork.js
function beginWork(current, workInProgress, renderExpirationTime) {
  switch (workInProgress.tag) {
    case IndeterminateComponent:
     // ...
    case ClassComponent:
     // 解析ClassComponent类型的Fiber结点
     return updateClassComponent(current, workInProgress, ... );
    case HostRoot:
      // 解析HostRoot类型的Fiber结点
     return updateHostRoot(current, workInProgress, renderExpirationTime);
    case HostComponent:
     // 解析HostComponent类型的Fiber结点
     return updateHostComponent(current, workInProgress, renderExpirationTime);
   case HostText:
     // 解析HostComponent类型的Fiber结点
     return updateHostText(current, workInProgress);
    // ...
  }
}
```

代码示例 6.3.2 beginWork 函数开启工作单元解析

`updateClassComponent` 函数、`updateHostRoot` 函数、`updateHostComponent` 函数和 `updateHostText` 函数分别用于解析 `ClassComponent`、`HostRoot`、`HostComponent` 和 `HostText` 类型的 Fiber 结点。这个过程中首先获取当前 Fiber 结点的子元素，然后将子元素传入 `reconcileChildren` 函数获取下一个 Fiber 结点，逻辑流程见图 6.3.1。

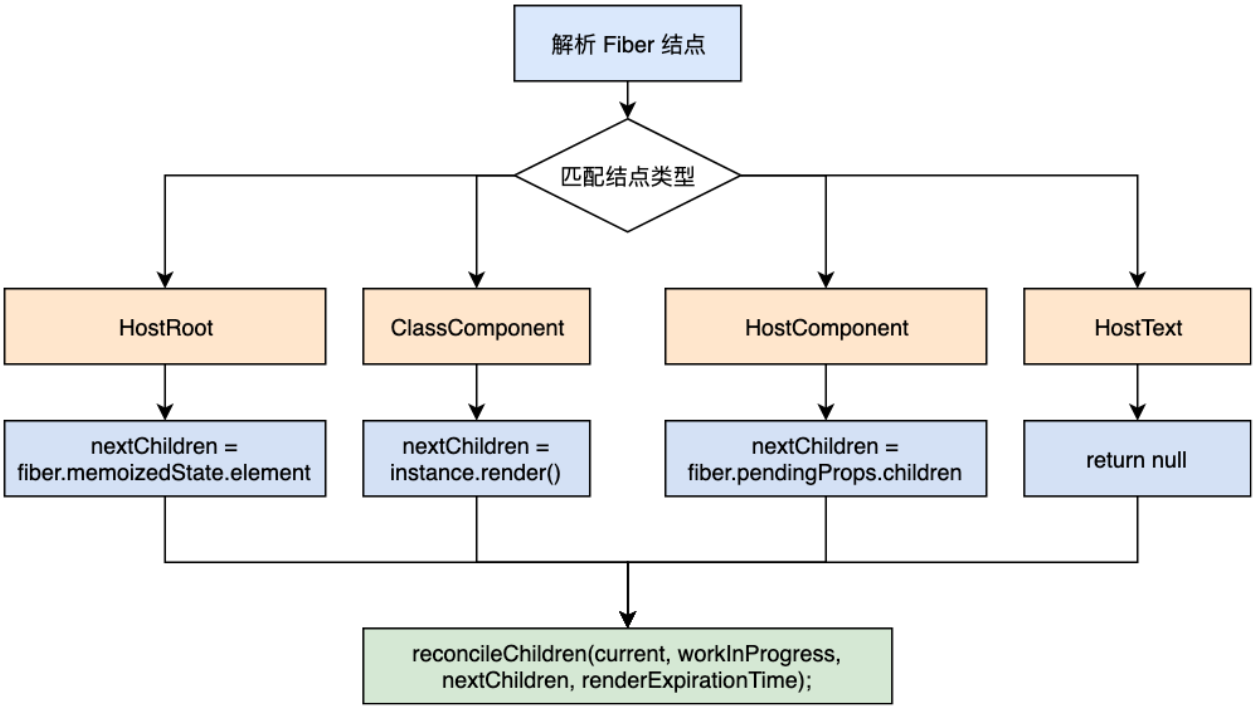


图 6.3.1 解析工作单元获取子元素

在解析工作单元时，React 需要获得当前 Fiber 结点的子元素并传入到 reconcileChildren 函数中，reconcileChildren 函数根据传入的元素返回下一个 Fiber 结点。如果将获取子元素作为整个「协调」算法执行的前置工作，那么获取下一个 Fiber 结点则是「协调」算法执行的最终目标。

**协调的最终目标 — 获得下一个 Fiber 结点**

reconcileChildren 函数实际上调用的是 reconcileChildFibers 函数，该函数是协调元素的入口函数，见代码示例 6.3.3。

```
// 源码位置：packages/react-reconciler/src/ReactChildFiber.js
function reconcileChildFibers(returnFiber, currentFirstChild, newChild, expirationTime) {
  ...
  // 获取当前传入（react）元素的类型
  var isObject = typeof newChild === 'object' && newChild !== null;
  // 如果当前元素是类型是对象，比如intance.render的返回值就是元素对象
  if (isObject) {
    switch (newChild.$$typeof) {
      case REACT_ELEMENT_TYPE:
        return placeSingleChild(reconcileSingleElement(returnFiber, currentFirstChild, newChild, ... ));
      case REACT_PORTAL_TYPE:
        return placeSingleChild(reconcileSinglePortal(returnFiber, currentFirstChild, newChild, ... ));
    }
  }
  // 如果当前元素的类型是字符串和数字，说明其值应该是文本
  if (typeof newChild === 'string' || typeof newChild === 'number') {
    return placeSingleChild(reconcileSingleTextNode(returnFiber, currentFirstChild, '' + newChild, ... ));
  }
  // 如果当前元素的类型是数组，比如一个div内部有多个子元素
  if (isArray(newChild)) {
    return reconcileChildrenArray(returnFiber, currentFirstChild, newChild, expirationTime);
  }
  ...
}
```

代码示例 6.3.3 reconcileChildFibers 函数根据元素类型分别处理

`reconcileChildFibers` 函数内部会根据传入的元素类型分别调用对应的处理函数，比如 `reconcileSingleElement` 函数、`reconcileSinglePortal` 函数和 `reconcileSingleTextNode` 函数分别用于 **协调** 单个 `element`、`portal` 和 `text` 数据类型的元素并 **返回新的 Fiber 结点**。那么，这里的 **协调（reconcile）具体做了哪些事情呢？**

> 注：Portal 提供了一种将子节点渲染到存在于父组件以外的 DOM 节点的优秀方案，有兴趣的同学可自行深入了解。

## 协调单个 element 类型的元素

`reconcileSingleElement` 函数用于协调单个 element 的结点，见代码示例 6.3.4。

```
// 源码位置：packages/react-reconciler/src/ReactChildFiber.js
function reconcileSingleElement(returnFiber, currentFirstChild, element, expirationTime) {
  var key = element.key;
  var child = currentFirstChild;
  ...
  if (child.key === key) {
    // 如果key相等，说明这个结点不需要直接删除并且可以复用
    var existing = useFiber(child, element.type === REACT_FRAGMENT_TYPE ? element.props.children : element.props, expirationTime);
    existing.ref = coerceRef(returnFiber, child, element);
    existing.return = returnFiber;
    // 返回复用的Fiber结点
    return existing;
  } else {
    // 如果key不相等则从父结点上把该结点删掉
    deleteChild(returnFiber, child);
  }
  ...
  // 如果key不相等
  if (element.type === REACT_FRAGMENT_TYPE) {
    var created = createFiberFromFragment(element.props.children, returnFiber.mode, expirationTime, element.key);
    created.return = returnFiber;
    // 返回新建的Fiber结点
    return created;
  } else {
    var _created4 = createFiberFromElement(element, returnFiber.mode, expirationTime);
    _created4.ref = coerceRef(returnFiber, currentFirstChild, element);
    _created4.return = returnFiber;
    // 返回新建的Fiber结点
    return _created4;
  }
}
```

代码示例 6.3.4 协调单个 element 类型的元素

`reconcileSingleElement` 函数中的参数解析如下：

- `returnFiber` 指向当前 Fiber 结点的父结点。
- `currentFirstChild` 指向的是当前的 Fiber 结点。
- `element` 指向的是当前 Fiber 结点经过 `setState( ... )` 操作后产生的的新的元素。

**协调过程中坚持的原则就是能复用尽量复用**，协调单个元素的主要依据就是先判断结点的 `key` 是否发生变化，如果没变则选择复用当前的 Fiber 结点并返回，否则创建新的的结点并返回。这里需要注意的是，复用结点并不代表不改变旧的结点，React 会将 `element.props` 更新到复用的结点中。

## 协调单个 text 类型的元素

基本原理同调和单个 element 类型的元素。

## 协调 array 类型的元素

协调 array 类型的元素的逻辑要比协调单个 element 和 text 类型的结点复杂一些，`reconcileChildrenArray`函数用于协调数组的结点，见代码示例 6.3.5。

> 注：代码示例 6.3.5 代码量较大，理解起来少有困难，建议结合下面的图片解释进行理解。

```
// 源码位置：packages/react-reconciler/src/ReactChildFiber.js
function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren, expirationTime) {
  // ...
  var resultingFirstChild = null;
  var previousNewFiber = null;
  var oldFiber = currentFirstChild;
  var newIdx = 0;
  var nextOldFiber = null;
  // case1：本次循环主要是检查更新渲染时，新的元素length相对于旧的Fiber链表长度是否相同或者增加
  for (; oldFiber !== null && newIdx < newChildren.length; newIdx++) {
    if (oldFiber.index > newIdx) {
      nextOldFiber = oldFiber;
      oldFiber = null;
    } else {
      // 将结点链表的指针指向下一个Fiber
      nextOldFiber = oldFiber.sibling;
    }
    // updateSlot函数用于复用旧的结点并将新的props更新进去
    var newFiber = updateSlot(returnFiber, oldFiber, newChildren[newIdx], expirationTime);
    // 如果新创建的Fiber结点为null并且旧结点也为null，则直接退出循环
    if (newFiber === null) {
      if (oldFiber === null) {
        oldFiber = nextOldFiber;
      }
      break;
    }
    // ...
    if (previousNewFiber === null) {
      // 设置用于返回的Fiber结点
      resultingFirstChild = newFiber;
    } else {
      // 兄弟结点通过sibling属性链接起来
      previousNewFiber.sibling = newFiber;
    }
    previousNewFiber = newFiber;
    oldFiber = nextOldFiber;
  }

  // 执行到这里说明链表长度和新的元素length相同，即双边完成 diff 处理
  if (newIdx === newChildren.length) {
    // 返回链表中的第一个Fiber结点，此时的Fiber结点已经更新了对应的props
    return resultingFirstChild;
  }

  // case2：oldFiber为null说明是首次渲染，那么我们需要将根据数组结构的元素生成链表结构的Fiber链
  if (oldFiber === null) {
    // 逐个遍历数组元素，然后根据每个元素创建Fiber结点
    for (; newIdx < newChildren.length; newIdx++) {
      var _newFiber = createChild(returnFiber, newChildren[newIdx], expirationTime);
      if (_newFiber === null) {
        continue;
      }
      lastPlacedIndex = placeChild(_newFiber, lastPlacedIndex, newIdx);
      if (previousNewFiber === null) {
```

```
      // 设置用于返回的Fiber结点
      resultingFirstChild = _newFiber;
    } else {
      // 兄弟结点通过sibling属性链接起来
      previousNewFiber.sibling = _newFiber;
    }
    previousNewFiber = _newFiber;
  }
  // 返回下一个Fiber结点
  return resultingFirstChild;
}

// 为了方便查找把所有的child重组为map结构
var existingChildren = mapRemainingChildren(returnFiber, oldFiber);

// case3：执行到这里说明前面两种case不成立，这是由于新的元素数组length发生了变化导致
for (; newIdx < newChildren.length; newIdx++) {
  // 创建Fiber结点
  var _newFiber2 = updateFromMap(existingChildren, returnFiber, newIdx, newChildren[newIdx], expirationTime);
  if (_newFiber2 !== null) {
    // ...
    if (previousNewFiber === null) {
      // 将刚创建的Fiber结点设置为返回结点
      resultingFirstChild = _newFiber2;
    } else {
      // 兄弟结点通过sibling属性链接起来
      previousNewFiber.sibling = _newFiber2;
    }
    previousNewFiber = _newFiber2;
  }
}
// 返回下一个Fiber结点
return resultingFirstChild;
}
```

代码示例 6.3.5 协调数组类型的元素

`reconcileChildrenArray`函数在协调数组元素时，要考虑的情况主要有三种，分别是：

1. 应用程序首次渲染时（对应代码注释中的 case2），这时`oldFiber`为`null`，React 需要为数组中的每一个元素创建一个 Fiber 结点，并将这些结点通过`sibling`链接起来。
2. 应用程序更新渲染时（对应代码注释中的 case1）新的元素数组的长度没有发生变化或者长度增加，这时 React 将链表中的 Fiber 结点与新的数组中的每一个元素按照协调单个元素的逻辑进行处理，见图 6.3.2。

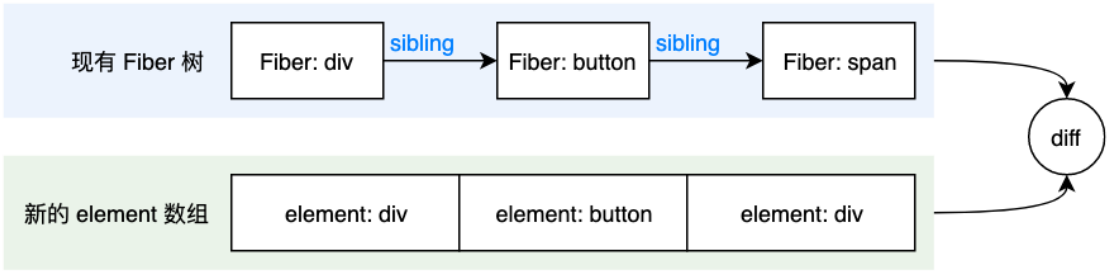


图 6.3.2 现有 Fiber 链表和新的 element 数组长度相同时两者的结构

如果新的 element 数组长度相对于旧的 Fiber 链表长度变大了怎么办呢？见代码示例 6.3.6。

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

  handleClick() {
    this.setState({
   count: this.state.count + 1
  });
  }

  render() {
  return (
   <div className="wrap-box">
    {
       // 根据count控制该元素显示或隐藏
     this.state.count > 0 && (
      <div className="title-text">
       <span>{this.state.text}</span>
      </div>
     )
    }
    <button key="1" onClick={this.handleClick}>{this.state.text}</button>
    <span key="2" id="spanText" className="span-text">{this.state.count}</span>
   </div>
  )
 }
}
```

代码示例 6.3.6 新的数组元素增多时的情况

代码示例 6.3.6 对应的 Fiber 链表结构与 新的 element 数组结构见图 6.3.3。

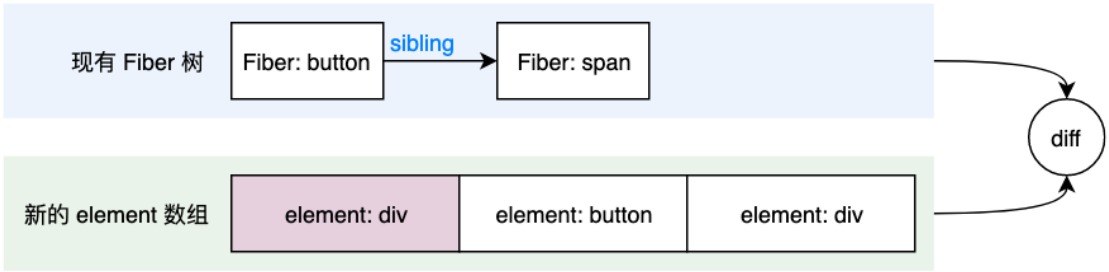


图 6.3.3 新的数组元素增多时 Fiber 和 element 数据结构

3. 应用程序更新渲染时（对应代码注释中的 case3）新的元素数组的长度相对于旧的 Fiber 链表长度变小，如将代码示例 6.3.6 中的控制元素显示的条件改为 `this.state.count === 0` 。这时 React 依然将链表中的 Fiber 结点与新的数组中的每一个元素按照协调单个元素的逻辑进行处理，但是数组元素的值却发生了变化，见图 6.3.4。

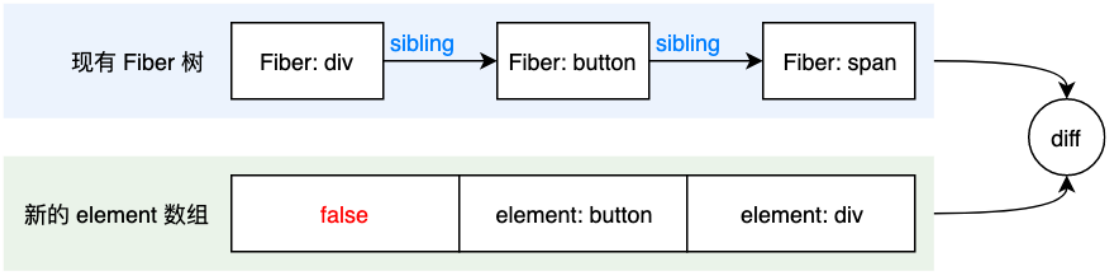


图 6.3.4 新的数组元素减少时 Fiber 和 element 数据结构

当新的 element 元素数组的长度相对于旧的 Fiber 链表长度变小，说明应用程序删除了某元素。此时 React 会找到元素之前在数组中的索引位置，然后将其值设置为`false`，然后将其对应的 Fiber 结点的 effectTag 标记为`Deletion`。

## 小结

本节主要介绍了 React Fiber 架构下「协调」算法的核心逻辑。现在我们对整个「协调」算法总结一下。

1. 协调的“对象”是 React 元素，得到的是下一个 Fiber 结点。
2. 我们常说的 React 中的 diff 算法是「协调」逻辑的一部分，被 diff 的双方分别是（新）元素和（已有）Fiber 结点。
3. 执行「协调」的过程对于单个元素和数组元素分别执行不同的方法进行处理。
4. 应用程序首次渲染时一般不涉及到 diff 过程，因为所有的元素均会转换为要插入的 Fiber 结点。
5. 应用程序更新渲染时通过进行（新）元素和（已有）Fiber 结点的 diff 处理，对结点标记对应的 effectTag。

在完成工作单元时 React 将所有标记了 effectTag 的结点收集起来，形成了 Effect List。下一节将会以具体实例说明更新渲染时的 Effect List 到底是什么样的。

}

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

30 通过实例分析应用程序更新渲染时的副作用列表