React 16 Diff策略简述

前言

React 16 将整体的数据结构从树改为了链表结构，相应的 Diff 算法也做了对应的调整，本篇主要总结一下 React 16 中的 Diff 算法究竟做了哪些改进。

对比

React 15 的 Diff 算法给出了三种 Diff 策略分析来优化树节点的传统循环比较，使得比较方式从 O(n^3)降级到 O(n)，保证了整体界面构建的性能，分别是 Tree diff、Component diff、Element diff。

1. Tree diff，进行同级比较，并非循环比较。这样比较次数就降为一层一次，时间复杂度直接降为 O(n)。如果同级相同位置节点不一样，则直接删除替换。同理，对于节点移动，也是删除移动。
2. Component diff，基本方案和 tree diff 一致，拥有相同类的两个组件将会生成相似的树形结构，拥有不同类的两个组件将会生成不同的树形结构。
3. Element diff，对于同一节点的元素，diff 算法提供了三种操作：插入、移动、删除。对于同一层级的一组子节点，它们可以通过唯一 key 进行区分，key 不相同，那么节点要删除重新构建，这里将会消耗大量性能。

React 16 的 diff 策略基于同层级节点从链表头部开始比较，节点的移动/插入/删除等操作都是在 fiber tree 的同一层级中进行。Diff 中 新老节点的对比，以新节点为标准，然后来构建整个 workInProgress，主要分为：

1. TextNode
2. React Element
3. Array

具体策略

Fiber

- 每个 Fiber的 child 指向其第一个孩子节点，没有孩子节点则为 null
- 每个 Fiber的sibling 指向其下一个兄弟节点，没有则为 null
- 每个 Fiber的return 指向其父节点
- 每个 Fiber有一个 index 属性表示其在兄弟节点中的排序
- 每个 Fiber的 stateNode 指向其原生节点
- 每个 Fiber有一个 key 属性

reconcileChildren 函数

React16 的 diff 策略主要依靠 reconcileChildren 函数来完成，reconcileChildren 对于刚创建的组件，会创建新的子 Fiber 节点，更新组件，将当前组件与该组件在上次更新时对应的 Fiber 节点比较，将比较的结果生成新 Fiber 节点。current 首次渲染只有 root 上有值，其他节点为 null，等状态更新时候所有

的 current 都会存在了，可以看到传入的参数还有一个 workInProgress，这个就是 react 中用到的双缓冲技术。

current 代表当前展示视图对应的 fiber 树，workInProgress 代表正在构建中的树，通过 alternate 连接。workInProgress 构建完成后根节点又通过改变 current 指向 workInProgress，所以 wip 又变回了 current 树，其中在构建 wip 树的时候会选择性的复用 current 树节点，见图 2-1。

```
  // 首次渲染时只有 root 节点存在 current，只有 root 会进入 reconcile 产生 effectTag
 export function reconcileChildren(
   current: Fiber | null,
   workInProgress: Fiber,
   nextChildren: any,
   renderExpirationTime: ExpirationTime,
  ) {
   if (current === null) {
     workInProgress.child = mountChildFibers(
     workInProgress,
     null,
     nextChildren,
     renderExpirationTime,
     );
   } else {
     workInProgress.child = reconcileChildFibers(
     workInProgress,
     current.child,
     nextChildren,
     renderExpirationTime,
   );
   }
  }
```

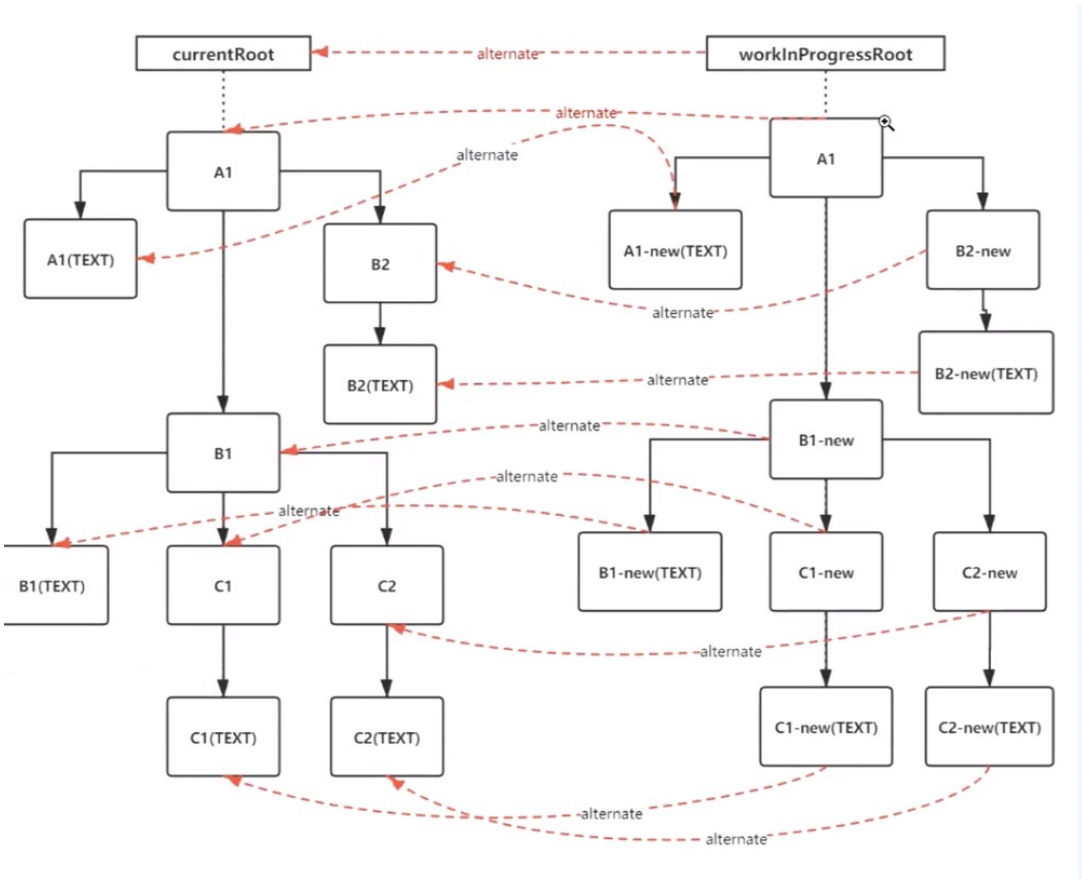

图 2-1

```
export const reconcileChildFibers = ChildReconciler(true)
export const mountChildFibers = ChildReconciler(false)
```

reconcileChildren 只是一个入口函数，初始加载和更新时，分别调用 `mountChildFibers` 和 `reconcileChildFibers`，`mountChildFibers` 和 `reconcileChildFibers` 方法其实是一样的，唯一的区别是生成这个方法的时候的一个参数（`shouldTrackSideEffects`，作用是判断是否要增加一些 `effectTag`，主要是用来优化初次渲染的 ）不同，但是最终都是执行 reconcileChildFibers（ChildReconciler 暴露出来的,并非指 `mountChildFibers` 和 `reconcileChildFibers` 中 `reconcileChildFibers` ）方法，如果不是首次渲染，就调用 reconcileChildFibers 去做 diff，然后得出 effect list。

`reconcileChildFibers` 如下：

```
function reconcileChildFibers(
  returnFiber: Fiber, // 即将 Diff 的这层的父节点
  currentFirstChild: Fiber | null, // 当前层的第一个 Fiber 节点
  newChild: any, // 即将更新的 vdom 节点，不是 Fiber 节点
  expirationTime: ExpirationTime, // 过期时间，跟调度有关
 ): Fiber | null {
  // 主要逻辑操作
 }
```

分类简述

了解了一些关于React 16 中reconcileChildren 函数流程知识，下面依据开头提到的几种情况简要介绍下React 16是怎么对不同的节点进行diff操作。

1. TextNode:

为了复用节点，对于 diff TextNode 会有两种情况。

- currentFirstChild 是一个 TextNode，那么则复用这个节点以提升性能，复用该节点之后，再删除之后的节点。
- currentFirstChild 不是 TextNode，节点不能复用，那么则删除全部老的节点，创建新的文字节点。ps: 删除节点不会真的从链表里面把节点删除，只是打一个 delete 的 tag，当 commit 的时候才会真正的去删除。

```
function reconcileSingleTextNode(...): Fiber {
  //旧的节点也是一个 text节点 则可以复用
  if (currentFirstChild !== null && currentFirstChild.tag === HostText) {
    //删除兄弟
    deleteRemainingChildren(returnFiber, currentFirstChild.sibling);
    //复用
    const existing = useFiber(currentFirstChild, textContent);
    existing.return = returnFiber;
    return existing;
  }
  // 否则创建新的fiber节点，将旧的节点和旧节点的兄弟都删除
  deleteRemainingChildren(returnFiber, currentFirstChild);
  const created = createFiberFromText(...);
  created.return = returnFiber;
  return created;
}
```

2. React Element

大体思路同上，先找有没有可以复用的节点，判断依据是相同的 key 和节点类型。如果没有就另外创建一个。与 textNode 区别的是在所有 `children` 节点中循环对比，如果key和节点类型相同，直接复用该节点，如果key不同，则把当前节点删除，然后再去找当前节点的兄弟节点，直到找到 key 相同，并且节点的类型相同，否则就删除所有的子节点，重新为新的 `children` 创建节点（ps.当 key 相同，节点类型不同的时候，React 会认为已经把这个节点重新覆盖，就不会再去找剩余的节点是否可以复用。只有在 key 不同的时候，才会去找兄弟节点是否可以复用。）

```
 function reconcileSingleElement(
  returnFiber: Fiber,
  currentFirstChild: Fiber | null,
  element: ReactElement,
  expirationTime: ExpirationTime,
): Fiber {
  const key = element.key
  let child = currentFirstChild
  // 找到相同的key，就会复用当前节点.
  while (child !== null) {
    // TODO: If key === null and child.key === null, then this only
applies to
    if (child.key === key) {
      if (
        child.tag === Fragment
          ? element.type === REACT_FRAGMENT_TYPE
```

```
          : child.elementType === element.type
      ) {
        deleteRemainingChildren(returnFiber, child.sibling)
        // 复用节点
        const existing = useFiber(
          child,
          element.type === REACT_FRAGMENT_TYPE
            ? element.props.children
            : element.props,
          expirationTime,
        )
        existing.ref = coerceRef(returnFiber, child, element)
        existing.return = returnFiber
        if (__DEV__) {
          existing._debugSource = element._source
          existing._debugOwner = element._owner
        }
        return existing
      } else {
        deleteRemainingChildren(returnFiber, child)
        break
      }
    } else {
      // 没有可复用节点，删除所有节点
      deleteChild(returnFiber, child)
    }
    child = child.sibling
  }
  // 创建节点
  //  Fragment是无意义的节点，需要创建 Fiber 的是它的 children，所以
createFiberFromFragment 传递的不是 element，而是element.props.children。
  if (element.type === REACT_FRAGMENT_TYPE) {
    const created = createFiberFromFragment(
      element.props.children,
      returnFiber.mode,
      expirationTime,
      element.key,
    )
    created.return = returnFiber
    return created
  } else {
    const created = createFiberFromElement(
      element,
      returnFiber.mode,
      expirationTime,
    )
    created.ref = coerceRef(returnFiber, currentFirstChild, element)
    created.return = returnFiber
    return created
```

```
  }
}
```

3. Array

reconcileChildFibers函数传入的newChild是数组时，将执行reconcileChildrenArray，可以根据reconcileChildrenArray的处理流程，把Array diff分为三个部分：

- 遍历新数组，新老数组进行对比，通过 updateSlot 方法对比key和type是否一致，找到可以直接复用的节点。
- 遍历完之后，删除剩余的老节点，添加剩余的新节点。遍历结束，若新节点已遍历完成，将剩余的老节点删除；若是老节点遍历完成，则将剩余新节点直接插入。
- 把所有老数组元素按 key 或 index 放 Map 里，通过新建的Map选取可复用的老节点，选取不到则插入新节点。

```
// 举例【a,b,c】变为【a,d,c】
// ab 保留，删除c，插入d
function reconcileChildrenArray(
  returnFiber: Fiber, // diff节点
  currentFirstChild: Fiber | null, // 老节点
  newChildren: Array<*>, // 新节点
  expirationTime: ExpirationTime, // 过期时间
): Fiber | null {
  if (__DEV__) {
    // First, validate keys.
    let knownKeys = null
    for (let i = 0; i < newChildren.length; i++) {
      const child = newChildren[i]
      knownKeys = warnOnInvalidKey(child, knownKeys)
    }
  }
  let resultingFirstChild: Fiber | null = null
  let previousNewFiber: Fiber | null = null
  let oldFiber = currentFirstChild
  let lastPlacedIndex = 0
  let newIdx = 0
  let nextOldFiber = null
  for (; oldFiber !== null && newIdx < newChildren.length; newIdx++) {
    if (oldFiber.index > newIdx) {
      nextOldFiber = oldFiber
      oldFiber = null
    } else {
      // 兄弟节点赋值给oldFiber
      nextOldFiber = oldFiber.sibling
    }
    // updateSlot 判断newChildren[newIdx]与oldFiber的 key 和type 是否一致。如果不一致，返回 null，如果一致，返回return指向returnFiber的newFiber
    const newFiber = updateSlot(
      returnFiber,
      oldFiber,
      newChildren[newIdx],
```

```
      expirationTime,
    )
    // 无法复用，跳出循环
    if (newFiber === null) {
      if (oldFiber === null) {
        oldFiber = nextOldFiber
      }
      break
    }
    // 更新标识
    if (shouldTrackSideEffects) {
      if (oldFiber && newFiber.alternate === null) {
        deleteChild(returnFiber, oldFiber)
      }
    }
    lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx)
    if (previousNewFiber === null) {
      resultingFirstChild = newFiber
    } else {
      previousNewFiber.sibling = newFiber
    }
    previousNewFiber = newFiber
    // 兄弟节点赋值给oldFiber
    oldFiber = nextOldFiber
  }
  // 遍历结束，新节点无剩余，删除老节点剩余节点
  if (newIdx === newChildren.length) {
    deleteRemainingChildren(returnFiber, oldFiber)
    return resultingFirstChild
  }
  // 新节点还有剩余，添加插入标记lastPlacedIndex
  if (oldFiber === null) {
    for (; newIdx < newChildren.length; newIdx++) {
      // 创建新fiber节点
      const newFiber = createChild(
        returnFiber,
        newChildren[newIdx],
        expirationTime,
      )
      if (!newFiber) {
        continue
      }
      lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx)
      if (previousNewFiber === null) {
        resultingFirstChild = newFiber
      } else {
        previousNewFiber.sibling = newFiber
      }
      previousNewFiber = newFiber
```

```
    }
    return resultingFirstChild
  }
  // Add all children to a key map for quick lookups.
  // 将所有老节点添加到map以便快速查找
  const existingChildren = mapRemainingChildren(returnFiber, oldFiber)
  // Keep scanning and use the map to restore deleted items as moves.
  for (; newIdx < newChildren.length; newIdx++) {
    // 通过新建的 map 对象中取旧节点，如果能取到意味着可以复用旧节点，反之创建新节点
    const newFiber = updateFromMap(
      existingChildren,
      returnFiber,
      newIdx,
      newChildren[newIdx],
      expirationTime,
    )
    if (newFiber) {
      if (shouldTrackSideEffects) {
        if (newFiber.alternate !== null) {
          // 复用老节点，从existingChildrenmap中删除旧节点值
          existingChildren.delete(newFiber.key === null ? newIdx : newFiber.key)
        }
      }
      lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx)
      if (previousNewFiber === null) {
        resultingFirstChild = newFiber
      } else {
        previousNewFiber.sibling = newFiber
      }
      previousNewFiber = newFiber
    }
  }
  if (shouldTrackSideEffects) {
    existingChildren.forEach(child => deleteChild(returnFiber, child))
  }
  // 得到结果
  return resultingFirstChild
}
function placeChild(
  newFiber: Fiber,
  lastPlacedIndex: number,
  newIndex: number,
): number {
  newFiber.index = newIndex;
  const current = newFiber.alternate;
  // 旧节点中的oldIndex， oldIndex < lastPlacedIndex，lastPlacedIndex不变，旧节点移动；反之旧节点位置保持不变，将lastPlacedIndex = oldIndex
```

```
  if (current !== null) {
    const oldIndex = current.index;
    if (oldIndex < lastPlacedIndex) {
      newFiber.effectTag = Placement;
      return lastPlacedIndex;
    } else {
      return oldIndex;
    }
  } else {
    // 旧节点中不存在，插入新节点
    newFiber.effectTag = Placement;
    return lastPlacedIndex;
  }
}
function mapRemainingChildren(
  returnFiber: Fiber,
  currentFirstChild: Fiber,
): Map<string | number, Fiber> {
  const existingChildren: Map<string | number, Fiber> = new Map();
  let existingChild = currentFirstChild;
  while (existingChild !== null) {
    if (existingChild.key !== null) {
      //如果 key 存在 则存储 key
      existingChildren.set(existingChild.key, existingChild);
    } else {
      // 否则就存 index 比如 一些纯文本、数字 节点没有key值
      existingChildren.set(existingChild.index, existingChild);
    }
    // 兄弟节点挨个遍历
    existingChild = existingChild.sibling;
  }
  return existingChildren;
}
```