从源码层面理解 React 是如何做 diff 的

reconcileChildFibers

React 的节点对比逻辑是在 reconcileChildFibers 方法中实现的。

reconcileChildFibers 是 ChildReconciler 方法内部定义的方法，通过调用 ChildReconciler 方法，并传入一个 shouldTrackSideEffects 参数返回。这样做是为了根据不同使用场景 ，产生不同的效果。

因为一个组件的更新和挂载的流程不同的。比如挂载会执行挂载的生命周期函数，更新则不会。

```js 复制代码
// reconcileChildFibers，和内部方法同名
export const reconcileChildFibers = ChildReconciler(true);

// mountChildFibers 是在一个节点从无到有的情况下调用
export const mountChildFibers = ChildReconciler(false);
```

reconcileChildFibers 的核心实现：

```js 复制代码
function reconcileChildFibers(
  returnFiber,
  currentFirstChild,
  newChild,
  lanes,
) {
  // newChild 可能是数组或对象
  // 如果是数组，那它的 $$typeof 就是 undefined
  switch (newChild.$$typeof) {
    case REACT_ELEMENT_TYPE:
      // 单节点 diff
      return placeSingleChild(
        reconcileSingleElement(
          returnFiber,
          currentFirstChild,
          newChild,
          lanes,
        ),
```

```
      );
    // ...
  }

  // 多节点 diff
  if (isArray(newChild)) {
    return reconcileChildrenArray(
      returnFiber,
      currentFirstChild,
      newChild,
      lanes,
    );
  }
}
```

newChild 是在组件 render 时得到 ReactElement，通过访问组件的 props.children 得到。

如果 newChild 是对象（非数组），会 调用 reconcileSingleElement（普通元素的情况），做单个节点的对比。

如果是数组时，就会 调用 reconcileChildrenArray，进行多节点的 diff。

更新和挂载的逻辑有点不同，后面都会用 “更新” 的场景进行讲解。

单节点 diff

先看看 **单节点 diff**。

需要注意的是，**这里的 “单节点” 指的是新生成的 ReactElement 是单个的。**只要新节点是数组就不算单节点，即使数组长度只为 1。 此外旧节点可能是有兄弟节点的（sibling 不为 null）。

> fiber 对象是通过链表来表示节点之间的关系的，它的 sibling 指向它的下一个兄弟节点，index 表示在兄弟节点中的位置。
>
> ReactElement 则是对象或数组的形式，通过 React.createElement() 生成。

单节点 diff 对应 reconcileSingleElement 方法，其核心实现为：


```
function reconcileSingleElement(
  returnFiber, // 父 fiber
```

```
  currentFirstChild, // 更新前的 fiber
  element, // 新的 ReactElement
) {
  const key = element.key;
  let child = currentFirstChild;

  while (child !== null) {

    if (child.key === key) {
      const elementType = element.type;
      // key 相同，且类型相同（比如新旧都是 div 类型）
      // 则走 “更新” 逻辑
      if (child.elementType === elementType) {
        // 【分支 1】
        // 将旧节点后所有的 sibling 打上删除 tag
        deleteRemainingChildren(returnFiber, child.sibling);

        // 创建 WorkInProgress，也就是原来 fiber 的替身啦
        const existing = useFiber(child, element.props.children);
        existing.return = returnFiber;
        return existing;
      } else {
        //【分支 2】
        deleteRemainingChildren(returnFiber, child);
        break;
      }
    }
    // 当前节点 key 不匹配，将它标记为待删除
    else {
      // 【分支 3】
      deleteChild(returnFiber, child);
    }
    // 取下一个兄弟节点，继续做对比
    child = child.sibling;
  }

  // 执行到这里说明没发现可复用节点，需要创建一个 fiber 出来
  const created = createFiberFromElement(element, returnFiber.mode, lanes);
  created.return = returnFiber;
  return created;
}
```

currentFirstChild 是更新前的节点，它是以链表的保存的，它的 sibling 指向它的下一个兄弟节点。

分支很多，下面我们进行详细地分析。

分支 1：key 相同且 type 相同

当发现 key 相同时，React 会尝试复用组件。新旧节点的 key 都没有设置的话，会设置为 null，如果新旧节点的 key 都为 null，会认为相等。

此外还要判断新旧类型是否相同（比如都是 div），因为类型都不同了，是无法复用的。

如果都满足，就会将旧 fiber 的后面的兄弟节点都标记为待删除，具体是调用 `deleteRemainingChildren()` 方法，它会在父 fiber 的 deletions 数组上，添加指定的子 fiber 和它之后的所有兄弟节点，作为删除标记。

之后的 commit 阶段会再进行正式的删除，再执行一些调用生命周期函数等逻辑。

`useFiber()` 会创建旧的 fiber 的替身，更新到 fiber 的 alternate 属性上，最后这个 useFiber 返回这个 alternate。然后直接 return，结束这个方法。

分支 2：key 相同但 type 不同

type 不同是无法复用的，如果 type 不同但 key 却相同，React 会认为没有匹配的可复用节点了。直接就将剩下的兄弟节点标记为删除，然后结束循环。

分支 3：key 不匹配

key 不同，用 `deleteChild()` 方法将当前的 fiber 节点标记为待删除，取出下一个兄弟节点再和新节点再比较，不断循环，直到匹配到其中一种分支为止。

以上就是三个分支。

如果能走到循环结束，说明没能找到能复用的 fiber，就会根据 ReactElement 调用 `createFiberFromElement()` 方法创建一个新的 fiber，然后返回它。

外部会拿到这个 fiber，调用 `placeSingleChild()` 将其 打上待更新 tag。

reconcileChildrenArray

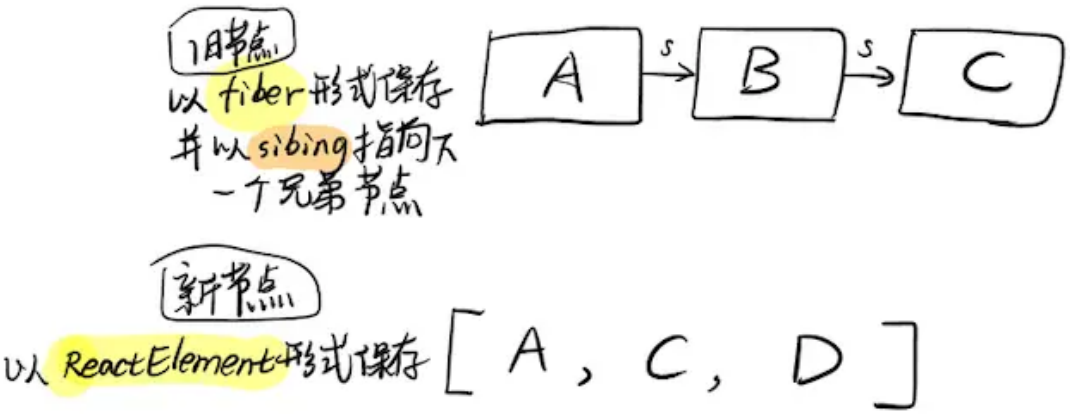
然后是 **多节点 diff**。

对应 ReactElement 为数组的场景，这种场景的算法实现要复杂的多。

多节点 diff 对应 reconcileChildrenArray 方法，因为算法比较复杂，先不直接贴比较完整的代码，而是分成几个阶段去一点点讲解。

多节点的 diff 分 4 个阶段，下面细说。

阶段1：同时从左往右遍历



旧 fiber 和 element 各自的指针一起从左往右走。指针分别为 nextFiber 和 newIdx，从左往右不断遍历。

遍历中发生的逻辑有：

1. 有一个指针走完，即 nextFiber 变成 null 或 newIdx 大于 newChildren.length，循环结束；
2. 如果 key 不同，就会结束遍历（在源码中的体现是 `updateSlot()` 返回 null 赋值给 newFiber，然后就 break 跳出循环）；
3. 如果 key 相同，但 type 不同，说明这个旧节点是不能用的了，给它 打上 “删除” 标记，然后继续遍历；
4. key 相同，type 也相同，复用节点。对于普通元素类型，最终会调用 updateElement 方法。

updateElement 方法会判断 fiber 和 element 的类型是否相同，如果相同，会给 fiber 的 alternate 生成一个 workInProcess（替身） fiber 返回，否则 创建一个新的 fiber 返回。它们

会带上新的 pendingProps 属性。

```js
function reconcileChildrenArray(
  returnFiber,
  currentFirstChild, // 旧的 fiber
  newChildren, // 新节点数组
  lanes,
) {
  let oldFiber = currentFirstChild;
  let lastPlacedIndex = 0;
  let newIdx = 0;
  let nextOldFiber = null;

  // 【1】分别从左往右遍历对比更新
  for (; oldFiber !== null && newIdx < newChildren.length; newIdx++) {
    if (oldFiber.index > newIdx) { // 旧 fiber 比新 element 多
      nextOldFiber = oldFiber;
      oldFiber = null;
    } else {
      nextOldFiber = oldFiber.sibling;
    }
    // 更新节点（或生成新的待插入节点）
    // 方法内部会判断 key 是否相等，不相等会返回 null。
    const newFiber = updateSlot(
      returnFiber,
      oldFiber,
      newChildren[newIdx],
      lanes,
    );

    // 如果当前新旧节点不匹配，就跳出循环
    if (newFiber === null) {
      if (oldFiber === null) {
        oldFiber = nextOldFiber;
      }
      break;
    }

    if (shouldTrackSideEffects) {
      if (oldFiber && newFiber.alternate === null) {
        // newFiber 不是基于 oldFiber 的 alternate 创建的
        // 说明 oldFiber 要销毁掉，要打上 “删除” 标记
        deleteChild(returnFiber, oldFiber);
      }
    }

    // 打 “place” 标记
    lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
```

```
  }
}
```

阶段 2：新节点遍历完的情况

跳出循环后，我们先看 新节点数组是否遍历完（newIdx 是否等于 newChildren.length）。

是的话，就将旧节点中剩余的所有节点编辑为 “删除”，然后直接结束整个函数。

```js 复制代码
function reconcileChildrenArray(
  returnFiber,
  currentFirstChild, // 旧的 fiber
  newChildren, // 新节点数组
  lanes,
) {
  // 【1】分别从左往右遍历对比更新
  // ...

  // 【2】如果新节点遍历完，将旧节点剩余节点全都标记为删除
  if (newIdx === newChildren.length) {
    // We've reached the end of the new children. We can delete the rest.
    deleteRemainingChildren(returnFiber, oldFiber);
    return resultingFirstChild;
  }
}
```

阶段 3：旧节点遍历完，新节点没遍历完的情况

如果是旧节点遍历完了，但新节点没有遍历完，就将新节点中的剩余节点，根据 element 构建为 fiber。

```js 复制代码
function reconcileChildrenArray(
  returnFiber,
  currentFirstChild, // 旧的 fiber
  newChildren, // 新节点数组
  lanes,
) {
  // 【1】分别从左往右遍历对比更新
  // ...

  // 【2】如果新节点遍历完，将旧节点剩余节点全都标记为删除
  // ...
```

```
  // 【3】如果旧节点遍历完了，但新节点没有遍历完，根据剩余新节点生成新 fiber
  if (oldFiber === null) {
    for (; newIdx < newChildren.length; newIdx++) {
      // 通过 element 创建 fiber
      const newFiber = createChild(returnFiber, newChildren[newIdx], lanes);
      if (newFiber === null) {
        continue;
      }
      // fiber 设置 index，并打上 “placement” 标签
      lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
      if (previousNewFiber === null) {
        resultingFirstChild = newFiber;
      } else {
        // 新建的 fiber 彼此连起来
        previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
    }
    // 返回新建 fiber 中的第一个
    return resultingFirstChild;
  }
}
```

阶段 4：使用 map 高效匹配新旧节点进行更新

【4】如果新旧节点都没遍历完，那我们会调用 mapRemainingChildren 方法，先将剩余的旧节点，放到 Map 映射中，以便快速访问。

map 中会优先使用 fiber.key（保证会转换为字符串）作为键；如果 fiber.key 是 null，则使用 fiber.index（数值类型），key 和 index 的值是不会冲突的。值自然就是 fiber 对象本身。

然后就是遍历剩余的新节点，调用 updateFromMap 方法，从映射表中找到对应的旧节点，和新节点进行对比更新。

遍历完后就是收尾工作了，map 中剩下的就是没能匹配的旧节点，给它们打上 “删除” 标记。