# 老生常谈React的diff算法原理-面试版

> 第一次发文章 not only（虽然）版式可能有点烂 but also （但是）最后赋有手稿研究finally看完他你有收获

> diff算法：对于update的组件，他会将当前组件与该组件在上次更新是对应的Fiber节点比较，将比较的结果生成新的Fiber节点。

！为了防止概念混淆，强调

> 一个DOM节点，在某一时刻最多会有4个节点和他相关。

javascript 复制代码

```
- 1.current Fiber。如果该DOM节点已在页面中，current Fiber代表该DOM节点对应的Fiber节点。
- 2.workInProgress Fiber。如果该DOM节点将在本次更新中渲染到页面中，workInProgress Fiber代表该DOM节点对应
- 3.DOM节点本身。
- 4.JSX对象。即ClassComponent的render方法的返回结果，或者FunctionComponent的调用结果，JSX对象中包含描述DC
```

复制代码

diff算法的本质上是对比1和4，生成2。

## Diff的瓶颈以及React如何应对

javascript 复制代码

由于diff操作本身也会带来性能损耗，React文档中提到，即使在最前沿的算法中
将前后两棵树完全比对的算法的复杂程度为 O(n 3 )，其中 n 是树中元素的数量。

如果在React中使用了该算法，那么展示1000个元素所需要执行的计算量将在十亿的量级范围。
这个开销实在是太过高昂。

## 所以为了降低算法复杂度，React的diff会预设3个限制：

css 复制代码

```
1.同级元素进行Diff。如果一个DOM节点在前后两次更新中跨越了层级，那么React不会尝试复用他。
2.不同类型的元素会产生出不同的树。如果元素由div变为p，React会销毁div及其子孙节点，并新建p及其子孙节点。
3.者可以通过 key prop来暗示哪些子元素在不同的渲染下能保持稳定。
```

## 那么我们接下来看一下Diff是如何实现的

```js
// 根据newChild类型选择不同diff函数处理
function reconcileChildFibers(
  returnFiber: Fiber,
  currentFirstChild: Fiber | null,
  newChild: any,
): Fiber | null {

  const isObject = typeof newChild === 'object' && newChild !== null;

  if (isObject) {
    // object类型，可能是 REACT_ELEMENT_TYPE 或 REACT_PORTAL_TYPE
    switch (newChild.$$typeof) {
      case REACT_ELEMENT_TYPE:
        // 调用 reconcileSingleElement 处理
      // // ...省略其他case
    }
  }

  if (typeof newChild === 'string' || typeof newChild === 'number') {
    // 调用 reconcileSingleTextNode 处理
    // ...省略
  }

  if (isArray(newChild)) {
    // 调用 reconcileChildrenArray 处理
    // ...省略
  }

  // 一些其他情况调用处理函数
  // ...省略

  // 以上都没有命中，删除节点
  return deleteRemainingChildren(returnFiber, currentFirstChild);
}
```



## 我们可以从同级的节点数量将Diff分为两类：



1. 当newChild类型为object、number、string，代表同级只有一个节点
- 2. 当newChild类型为Array，同级有多个节点

> 单节点diff

css 复制代码

以类型**Object**为例，会进入这个函数reconcileSingleElement

参考 前端进阶面试题详细解答

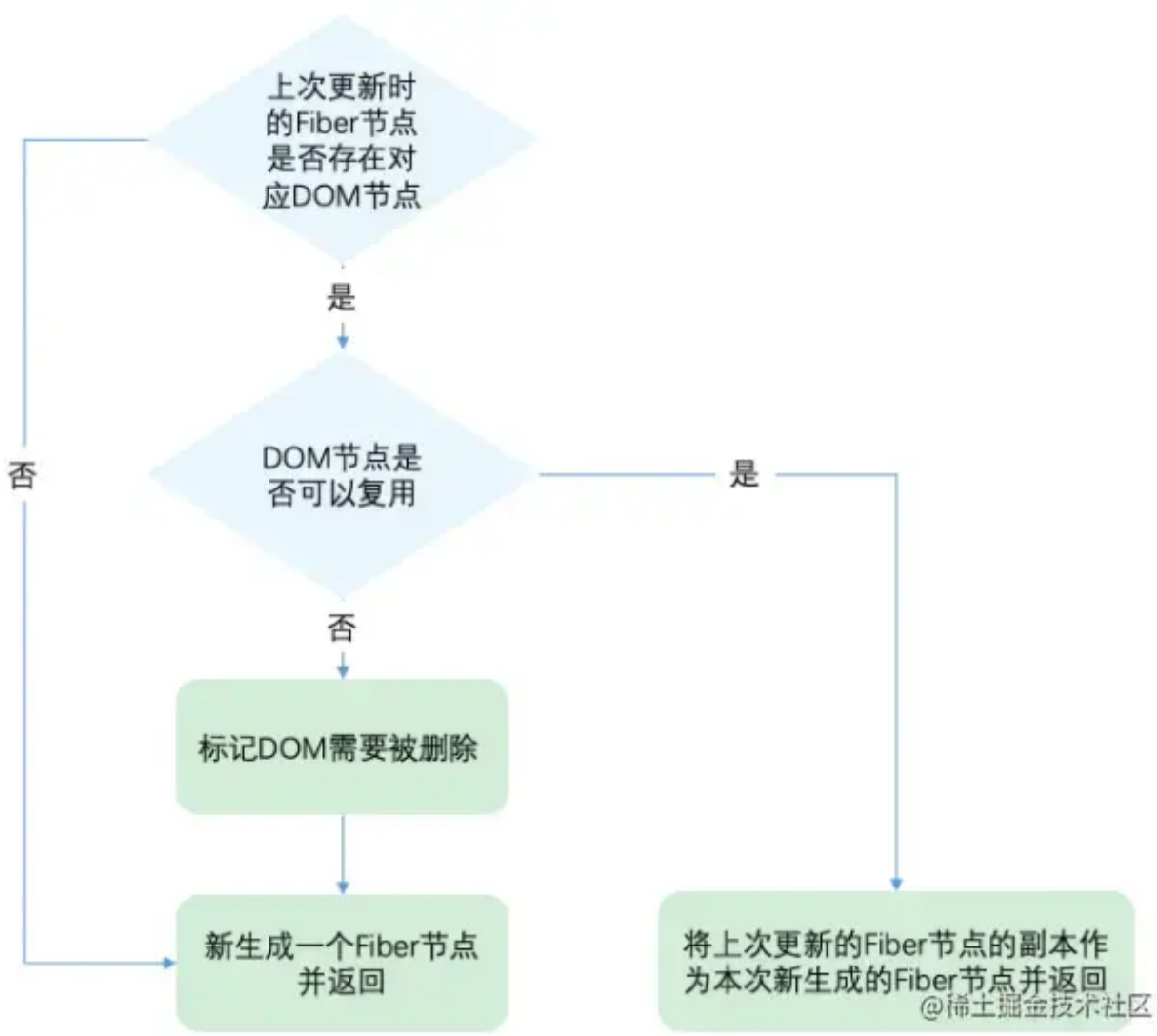
```js
const isObject = typeof newChild === 'object' && newChild !== null;

if (isObject) {
  // 对象类型，可能是 REACT_ELEMENT_TYPE 或 REACT_PORTAL_TYPE
  switch (newChild.$$typeof) {
    case REACT_ELEMENT_TYPE:
      // 调用 reconcileSingleElement 处理
    // ...其他case
  }
}
```



复制代码

这个函数会做如下事情：



复制代码

让我们看看第二步判断DOM节点是否可以复用是如何实现的。

```js
function reconcileSingleElement(
  returnFiber: Fiber,
  currentFirstChild: Fiber | null,
  element: ReactElement
): Fiber {
  const key = element.key;
  let child = currentFirstChild;

  // 首先判断是否存在对应DOM节点
  while (child !== null) {
    // 上一次更新存在DOM节点，接下来判断是否可复用

    // 首先比较key是否相同
    if (child.key === key) {

      // key相同，接下来比较type是否相同

      switch (child.tag) {
        // ...省略case

        default: {
          if (child.elementType === element.type) {
            // type相同则表示可以复用
            // 返回复用的fiber
            return existing;
          }

          // type不同则跳出循环
          break;
        }
      }
      // 代码执行到这里代表：key相同但是type不同
      // 将该fiber及其兄弟fiber标记为删除
      deleteRemainingChildren(returnFiber, child);
      break;
    } else {
      // key不同，将该fiber标记为删除
      deleteChild(returnFiber, child);
    }
    child = child.sibling;
  }

  // 创建新Fiber，并返回 ...省略
}
```



从代码可以看出，React通过先判断key是否相同，如果key相同则判断type是否相同，只有都相同时一个DOM节点才能复用。

这里有个细节需要关注下：

yaml 复制代码

1.当child !== null且key相同且type不同时执行deleteRemainingChildren将child及其兄弟fiber都标记删除。

2.当child !== null且key不同时仅将child标记删除。

> 例子：当前页面有3个li，我们要全部删除，再插入一个p。

```js
// 当前页面显示的
ul > li * 3

// 这次需要更新的
ul > p
```



由于本次更新时只有一个p，属于单一节点的Diff，会走上面介绍的代码逻辑。

> 解释：

css 复制代码

在reconcileSingleElement中遍历之前的3个fiber（对应的DOM为3个**li**），寻找本次更新的**p**是否可以复用之前的3个fibe

当key相同且type不同时，代表我们已经找到本次更新的**p**对应的上次的fiber，但是 **p** 与 **li** 的type不同，不能复用。
既然唯一的可能性已经不能复用，则剩下的fiber都没有机会了，所以都需要标记删除。

当key不同时只代表遍历到的该fiber不能被**p**复用，后面还有兄弟fiber还没有遍历到。所以仅仅标记该fiber删除。

> 练习题：


typescript 复制代码

习题1：未设置key prop默认 key = null;，所以更新前后key相同，都为null，但是更新前type为div，更新后为p，type

习题2：更新前后key改变，不需要再判断type，不能复用。

习题3：更新前后key没变，但是type改变，不能复用。

习题4：更新前后key与type都未改变，可以复用。children变化，DOM的子元素需要更新。

> 多节点diff

同级多个节点的Diff，一定属于下面3中情况的一种或多种。

- 情况1：节点更新

```
// 之前
<ul>
  <li key="0" className="before">0<li>
  <li key="1">1<li>
</ul>

// 之后 情况1 — 节点属性变化
<ul>
  <li key="0" className="after">0<li>
  <li key="1">1<li>
</ul>

// 之后 情况2 — 节点类型更新
<ul>
  <div key="0">0<li>
  <li key="1">1<li>
</ul>
```



- 情况2：节点新增或减少

```
// 之前
<ul>
  <li key="0">0<li>
  <li key="1">1<li>
</ul>

// 之后 情况1 — 新增节点
<ul>
  <li key="0">0<li>
  <li key="1">1<li>
  <li key="2">2<li>
</ul>

// 之后 情况2 — 删除节点
<ul>
  <li key="1">1<li>
</ul>
```



- 情况3：节点位置变化


复制代码

注意在这里diff算法无法使用双指针优化
在我们做数组相关的算法题时，经常使用双指针从数组头和尾同时遍历以提高效率，但是这里却不行。

css 复制代码

虽然本次更新的JSX对象 newChildren为数组形式，但是和newChildren中每个组件进行比较的是current fiber
同级的Fiber节点是由sibling指针链接形成的单链表。

即 newChildren[0]与fiber比较，newChildren[1]与fiber.sibling比较。

复制代码

所以无法使用双指针优化。

## 基于以上原因，Diff算法的整体逻辑会经历两轮遍历：

复制代码

1.第一轮遍历：处理更新的节点。
2.第二轮遍历：处理剩下的不属于更新的节点

## 第一轮遍历：

## 第一轮遍历步骤如下：

css 复制代码

let i = 0，遍历newChildren，将newChildren[i]与oldFiber比较，判断DOM节点是否可复用。

如果可复用，i++，继续比较newChildren[i]与oldFiber.sibling，可以复用则继续遍历。

如果不可复用，立即跳出整个遍历，第一轮遍历结束。

如果newChildren遍历完（即i === newChildren.length - 1）或者oldFiber遍历完（即oldFiber.sibling === null
跳出遍历，第一轮遍历结束。

上面3跳出的遍历

此时newChildren没有遍历完，oldFiber也没有遍历完。

> 上例子：

javascript 复制代码

前2个节点可复用，遍历到key === 2的节点发现type改变，不可复用，跳出遍历。

此时oldFiber剩下key === 2未遍历，newChildren剩下key === 2、key === 3未遍历。

上面4跳出的遍历

可能newChildren遍历完，或oldFiber遍历完，或他们同时遍历完。

> 上例子：

```
// 之前
<li key="0" className="a">0</li>
<li key="1" className="b">1</li>

// 之后 情况1 — newChildren与oldFiber都遍历完
<li key="0" className="aa">0</li>
<li key="1" className="bb">1</li>

// 之后 情况2 — newChildren没遍历完，oldFiber遍历完
// newChildren剩下 key==="2" 未遍历
<li key="0" className="aa">0</li>
<li key="1" className="bb">1</li>
<li key="2" className="cc">2</li>

// 之后 情况3 — newChildren遍历完，oldFiber没遍历完
// oldFiber剩下 key==="1" 未遍历
<li key="0" className="aa">0</li>
```



带着第一轮遍历的结果，我们开始第二轮遍历。

## 第一轮遍历：（4种情况）

diff 复制代码

- 1.newChildren与oldFiber同时遍历完

那就是最理想的情况：只有组件更新。此时Diff结束。

diff 复制代码

- 2.newChildren没遍历完，oldFiber遍历完

已有的DOM节点都复用了，这时还有新加入的节点，意味着本次更新有新节点插入
我们只需要遍历剩下的newChildren为生成的workInProgress fiber依次标记Placement。

diff 复制代码

- 3.newChildren遍历完，oldFiber没遍历完

意味着本次更新比之前的节点数量少，有节点被删除了。所以需要遍历剩下的oldFiber，依次标记Deletion。

vbnet 复制代码

- 4.newChildren与oldFiber都没遍历完

这意味着有节点在这次更新中改变了位置。

改变了位置就需要我们处理移动的节点

由于有节点改变了位置，所以不能再用位置索引i对比前后的节点，那么如何才能将同一个节点在两次更新中对应上呢？
我们需要使用**key**。

为了快速的找到**key**对应的oldFiber，我们将所有还未处理的oldFiber存入以**key**为**key**，oldFiber为value的Map中。

```js
const existingChildren = mapRemainingChildren(returnFiber, oldFiber);
```



vbnet 复制代码
接下来遍历剩余的newChildren，通过newChildren[i].key就能在existingChildren中找到**key**相同的oldFiber

标记节点是否移动

!既然我们的目标是寻找移动的节点，那么我们需要明确：节点是否移动是以什么为参照物？

复制代码
我们的参照物是：最后一个可复用的节点在oldFiber中的位置索引（用变量lastPlacedIndex表示）。

javascript 复制代码
由于本次更新中节点是按newChildren的顺序排列。
在遍历newChildren过程中，每个遍历到的可复用节点一定是当前遍历到的所有可复用节点中最靠右的那个
即一定在lastPlacedIndex对应的可复用的节点在本次更新中位置的后面。

那么我们只需要比较遍历到的可复用节点在上次更新时是否也在lastPlacedIndex对应的oldFiber后面
就能知道两次更新中这两个节点的相对位置改变没有。

我们用变量oldIndex表示遍历到的可复用节点在oldFiber中的位置索引。

如果oldIndex < lastPlacedIndex，代表本次更新该节点需要向右移动。

lastPlacedIndex初始为0，每遍历一个可复用的节点，如果oldFiber >= lastPlacedIndex，则lastPlacedIndex = ol

# 下面通过两个demo来看一下react团队的diff算法精髓

## demo1

```
// 之前 abcd  // 之后 acdb
```

> ===第一轮遍历开始===

```
a（之后）vs a（之前）
```

key不变，可复用

此时 a 对应的oldFiber（之前的a）在之前的数组（abcd）中索引为0

所以 lastPlacedIndex = 0;

> 继续第一轮遍历...

```
c（之后）vs b（之前）
```

key改变，不能复用，跳出第一轮遍历

此时 lastPlacedIndex === 0;

> ===第一轮遍历结束===

> ===第二轮遍历开始===

```
newChildren === cdb，没用完，不需要执行删除旧节点
```

```
oldFiber === bcd，没用完，不需要执行插入新节点
```

将剩余oldFiber（bcd）保存为map

```
// 当前oldFiber：bcd

// 当前newChildren：cdb
```

> 继续遍历剩余newChildren

javascript 复制代码

```
key === c 在 oldFiber中存在

const oldIndex = c（之前）.index;

此时 oldIndex === 2;  // 之前节点为 abcd，所以c.index === 2

比较 oldIndex 与 lastPlacedIndex;

如果 oldIndex >= lastPlacedIndex 代表该可复用节点不需要移动

并将 lastPlacedIndex = oldIndex;

如果 oldIndex < lastplacedIndex 该可复用节点之前插入的位置索引小于这次更新需要插入的位置索引，代表该节点需要

在例子中，oldIndex 2 > lastPlacedIndex 0，

则 lastPlacedIndex = 2;

c节点位置不变
```

> 继续遍历剩余newChildren

```
// 当前oldFiber：bd

// 当前newChildren：db
```

javascript 复制代码

```
key === d 在 oldFiber中存在

const oldIndex = d（之前）.index;

oldIndex 3 > lastPlacedIndex 2 // 之前节点为 abcd，所以d.index === 3
```

则 lastPlacedIndex = 3;

d节点位置不变

> 继续遍历剩余newChildren

```
// 当前oldFiber：b

// 当前newChildren：b
```

javascript 复制代码

```
key === b 在 oldFiber中存在

const oldIndex = b（之前）.index;

oldIndex 1 < lastPlacedIndex 3 // 之前节点为 abcd，所以b.index === 1

则 b节点需要向右移动
```

> ===第二轮遍历结束===

!最终acd 3个节点都没有移动，b节点被标记为移动

## demo2

```
// 之前 abcd

// 之后 dabc
```

> ===第一轮遍历开始===

css 复制代码

```
d（之后）vs a（之前）
key改变，不能复用，跳出遍历
```

> ===第一轮遍历结束===

> ===第二轮遍历开始===

javascript 复制代码

newChildren === dabc，没用完，不需要执行删除旧节点

oldFiber === abcd，没用完，不需要执行插入新节点

将剩余oldFiber（abcd）保存为map

继续遍历剩余newChildren

```
// 当前oldFiber：abcd

// 当前newChildren dabc
```

javascript 复制代码

key === d 在 oldFiber中存在

const oldIndex = d（之前）.index;

此时 oldIndex === 3; // 之前节点为 *abcd*，所以*d.index === 3*

比较 oldIndex 与 lastPlacedIndex;

oldIndex 3 > lastPlacedIndex 0

则 lastPlacedIndex = 3;

d节点位置不变

> 继续遍历剩余newChildren

```
// 当前oldFiber：abc

// 当前newChildren abc
```

javascript 复制代码

key === a 在 oldFiber中存在

const oldIndex = a（之前）.index; // 之前节点为 *abcd*，所以*a.index === 0*

此时 oldIndex === 0;

比较 oldIndex 与 lastPlacedIndex;

oldIndex 0 < lastPlacedIndex 3

则 a节点需要向右移动

> 继续遍历剩余newChildren

```
// 当前oldFiber：bc

// 当前newChildren bc
```

javascript 复制代码

```
key === b 在 oldFiber中存在

const oldIndex = b（之前）.index; // 之前节点为 abcd，所以b.index === 1

此时 oldIndex === 1;

比较 oldIndex 与 lastPlacedIndex;

oldIndex 1 < lastPlacedIndex 3

则 b节点需要向右移动
```

> 继续遍历剩余newChildren

```
// 当前oldFiber：c

// 当前newChildren c
```

javascript 复制代码

```
key === c 在 oldFiber中存在

const oldIndex = c（之前）.index; // 之前节点为 abcd，所以c.index === 2

此时 oldIndex === 2;

比较 oldIndex 与 lastPlacedIndex;

oldIndex 2 < lastPlacedIndex 3

则 c节点需要向右移动
```

> ===第二轮遍历结束===

!可以看到，我们以为从 abcd 变为 dabc，只需要将d移动到前面。 !但实际上React保持d不变，将abc分别移动到了d的后面。

> 用张老生常谈的图