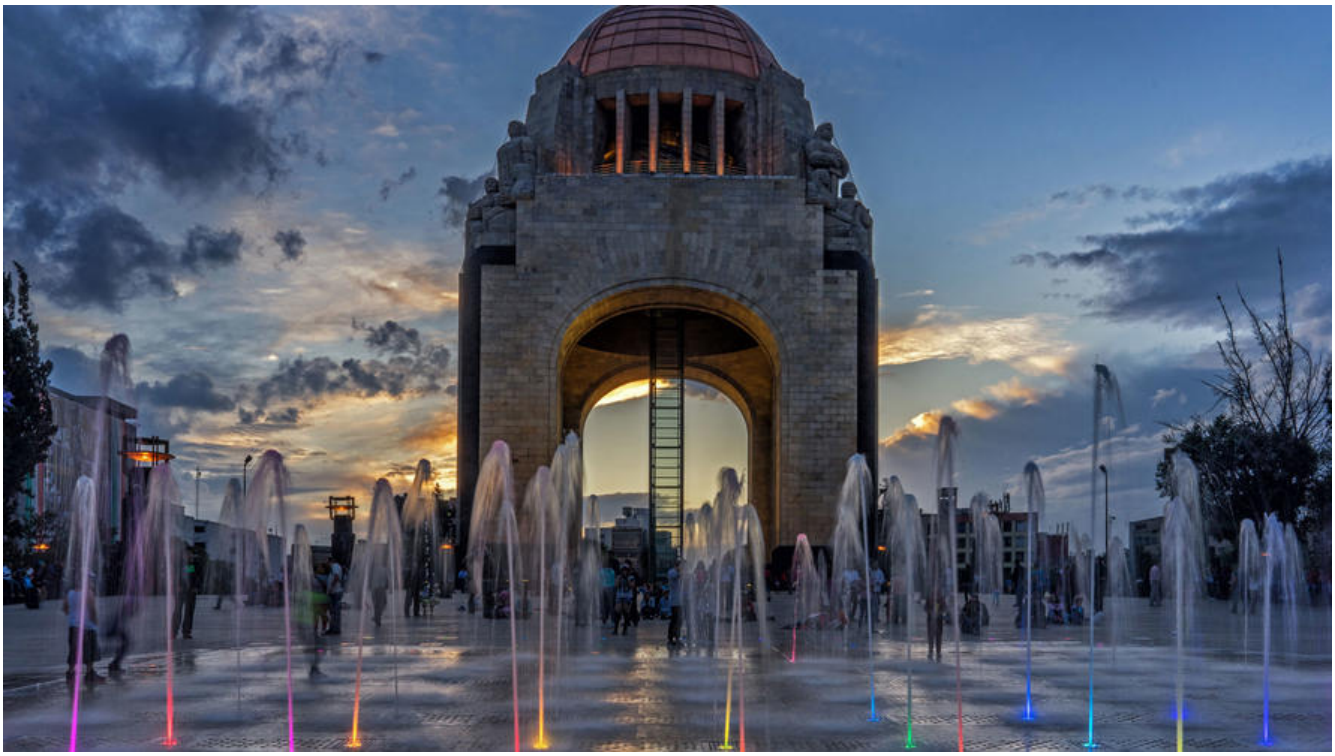
React 虚拟 DOM 浅析

作者：gmw-zjw
原文链接：https://ld246.com/article/1567510449986
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)



React中采用 Virtual DOM 与 diff 的完美结合，特别是其高效的 diff 算法，让用户可以无需顾忌性问题而”任性自由”的刷新页面，让开发者也可以无需关心 Virtual DOM 背后的运作原理，因为 React diff 会帮助我们计算出 Virtual DOM 中真正变化的部分，并只针对该部分进行实际 DOM 操作，而重新渲染整个页面，从而保证了每次操作更新后页面的高效渲染，因此 Virtual DOM 与 diff 是保证 React 性能口碑的幕后推手。

传统的Diff算法

传统 diff 算法通过循环递归对节点进行依次对比，效率低下，算法复杂度达到 O(n^3)，其中 n 是树节点的总数。

如果 React 只是单纯的引入 diff 算法而没有任何的优化改进，那么其效率是远远无法满足前端渲染要求的性能。

因此，想要将 diff 思想引入 Virtual DOM，就需要设计一种稳定高效的 diff 算法，而 React 做到了！

实现

在React 中，将 O(n^3) 复杂度的问题转换成 O(n) 复杂度的问题，这样的操作太大

diff 策略

- tree diff // Web UI 中 DOM 节点跨层级的移动操作特别少，可以忽略不计
- component dif // 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不同的树形结构
- element diff // 对于同一层级的一组子节点，它们可以通过唯一 id 进行区分

tree diff

React 对树的算法进行了简洁明了的优化，即对树进行分层比较，两棵树只会对同一层次的节点进行

较.

既然 DOM 节点跨层级的移动操作少到可以忽略不计，针对这一现象，React 通过 updateDepth 对 Virtual DOM 树进行层级控制，只会对相同颜色方框内的 DOM 节点进行比较，即同一个父节点下的有子节点。当发现节点已经不存在，则该节点及其子节点会被完全删除掉，不会用于进一步的比较。样只需要对树进行一次遍历，便能完成整个 DOM 树的比较.

```
updateChilern: function (nextNestChilderElemt, transaction, context) {
    updateDepath++
    var errThrown = true
    try {
        this._updateChilern(nextNestChilderElemt, transaction, context)
        errorThrown = false
    } catch (err) {
        updateDepath --
        if (!errorThrown) {
            clearQueue()
        } else {
            processQueue()
        }
    }
}
```

我们现在只是得到的同级下的节点，如果节点跨级，那应该怎么做呢?

答案是这样的，删除创建的，添加创建，因此 React 官方建议不要进行 DOM 节点跨层级的操作。

component diff

如果是同一类型的组件，按照原策略继续比较 virtual DOM tree。

如果不是，则将该组件判断为 dirty component，从而替换整个组件下的所有子节点。

对于同一类型的组件，有可能其 Virtual DOM 没有任何变化，如果能够确切的知道这点那可以节省量的 diff 运算时间，因此 React 允许用户通过 shouldComponentUpdate() 来判断该组件是否需要行 diff。

element diff

当节点处于同一层级时，React diff 提供了三种节点操作，分别为：INSERT_MARKUP（插入）、MOVE_EXISTING（移动）和 REMOVE_NODE（删除）。

INSERT_MARKUP，新的 component 类型不在老集合里， 即是全新的节点，需要对新节点执行插操作。

MOVE_EXISTING，在老集合有新 component 类型，且 element 是可更新的类型，generateComponentChildren 已调用 receiveComponent，这种情况下 prevChild=nextChild，就需要做移动操作可以复用以前的 DOM 节点。

REMOVE_NODE，老 component 类型，在新集合里也有，但对应的 element 不同则不能直接复用更新，需要执行删除操作，或者老 component 不在新集合里的，也需要执行删除操作。

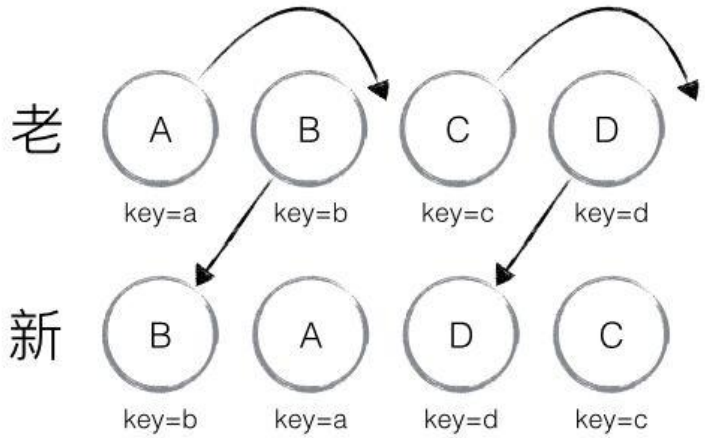
```
// inst
function equeueInerMarkup (parentInst, markup, toIndex) {
```

```
    updateQueue.push({
        parentInst: parentInst,
        parentNode: null,
        type: ReactMultiChildUpdateTypes.INSERT_MARKUP, // 插入节点
        markupIndex: markupQueue.push(markup) - 1,
        context: null,
        fromIndex: null,
        toIndex: toIndex
    })
}

// move
function enqueueMove (parentInst, markup, toIndex) {
    updateQueue.push({
        parentInst: parentInst,
        parentNode: null,
        type: ReactMultiChildUpdateTypes.MOVE_EXISTING, //移动一节点
        markupIndex: null,
        context: null,
        fromIndex: fromIndex,
        toIndex: toIndex
    })
}

//remove
function enqueueRmove (parentInst, fromIndex) {
    updateQueue.push({
        parentInst: parentInst,
        parentNode: null,
        type: ReactMultiChildUpdateTypes.REMOVE_NODE, // 移除一节点
        context: null,
        fromIndex: fromIndex,
        toIndex: null
    })
}
```

这样的做法会带来对性能的消耗，这时候react对此提出优化，在进行比较后会对结构相同的组件移位置，而不是删除/创建的方法，进而大大的提高了性能。



那么，如此高效的 diff 到底是如何运作的呢？让我们通过源码进行详细分析。

首先对新集合的节点进行循环遍历，for (name in nextChildren)，通过唯一 key 可以判断新老集合是否存在相同的节点，if (prevChild === nextChild)，如果存在相同节点，则进行移动操作，但在动前需要将当前节点在老集合中的位置与 lastIndex 进行比较，if (child._mountIndex < lastIndex) 则进行节点移动操作，否则不执行该操作。这是一种顺序优化手段，lastIndex 一直在更新，表示访过的节点在老集合中最右的位置（即最大的位置），如果新集合中当前访问的节点比 lastIndex 大，明当前访问节点在老集合中就比上一个节点位置靠后，则该节点不会影响其他节点的位置，因此不用加到差异队列中，即不执行移动操作，只有当访问的节点比 lastIndex 小时，才需要进行移动操作。

以上图为例，可以更为清晰直观的描述 diff 的差异对比过程：

从新集合中取得 B，判断老集合中存在相同节点 B，通过对比节点位置判断是否进行移动操作，B 在集合中的位置 B._mountIndex = 1，此时 lastIndex = 0，不满足 child._mountIndex < lastIndex 条件，因此不对 B 进行移动操作；更新 lastIndex = Math.max(prevChild._mountIndex, lastIndex ，其中 prevChild._mountIndex 表示 B 在老集合中的位置，则 lastIndex = 1，并将 B 的位置更新新集合中的位置prevChild._mountIndex = nextIndex，此时新集合中 B._mountIndex = 0，nextIn ex++ 进入下一个节点的判断。

从新集合中取得 A，判断老集合中存在相同节点 A，通过对比节点位置判断是否进行移动操作，A 在集合中的位置 A._mountIndex = 0，此时 lastIndex = 1，满足 child._mountIndex < lastIndex的件，因此对 A 进行移动操作enqueueMove(this, child._mountIndex, toIndex)，其中 toIndex 其实是 nextIndex，表示 A 需要移动到的位置；更新 lastIndex = Math.max(prevChild._mountIndex, la tIndex)，则 lastIndex = 1，并将 A 的位置更新为新集合中的位置 prevChild._mountIndex = nextI dex，此时新集合中A._mountIndex = 1，nextIndex++ 进入下一个节点的判断。

从新集合中取得 D，判断老集合中存在相同节点 D，通过对比节点位置判断是否进行移动操作，D 在集合中的位置 D._mountIndex = 3，此时 lastIndex = 1，不满足 child._mountIndex < lastIndex 条件，因此不对 D 进行移动操作；更新 lastIndex = Math.max(prevChild._mountIndex, lastIndex ，则 lastIndex = 3，并将 D 的位置更新为新集合中的位置 prevChild._mountIndex = nextIndex 此时新集合中D._mountIndex = 2，nextIndex++ 进入下一个节点的判断。

从新集合中取得 C，判断老集合中存在相同节点 C，通过对比节点位置判断是否进行移动操作，C 在集合中的位置 C._mountIndex = 2，此时 lastIndex = 3，满足 child._mountIndex < lastIndex 的件，因此对 C 进行移动操作 enqueueMove(this, child._mountIndex, toIndex)；更新 lastIndex = ath.max(prevChild._mountIndex, lastIndex)，则 lastIndex = 3，并将 C 的位置更新为新集合中的置 prevChild._mountIndex = nextIndex，此时新集合中 C._mountIndex = 3，nextIndex++ 进入一个节点的判断，由于 C 已经是最后一个节点，因此 diff 到此完成。

以上主要分析新老集合中存在相同节点但位置不同时，对节点进行位置移动的情况，如果新集合中有加入的节点且老集合存在需要删除的节点，那么 React diff 又是如何对比运作的呢？

以下图为例：

从新集合中取得 B，判断老集合中存在相同节点 B，由于 B 在老集合中的位置 B._mountIndex = 1 此时lastIndex = 0，因此不对 B 进行移动操作；更新 lastIndex = 1，并将 B 的位置更新为新集合的位置B._mountIndex = 0，nextIndex++进入下一个节点的判断。

从新集合中取得 E，判断老集合中不存在相同节点 E，则创建新节点 E；更新 lastIndex = 1，并将 E 的位置更新为新集合中的位置，nextIndex++进入下一个节点的判断。

从新集合中取得 C，判断老集合中存在相同节点 C，由于 C 在老集合中的位置C._mountIndex = 2，l stIndex = 1，此时 C._mountIndex > lastIndex，因此不对 C 进行移动操作；更新 lastIndex = 2 并将 C 的位置更新为新集合中的位置，nextIndex++ 进入下一个节点的判断。

从新集合中取得 A，判断老集合中存在相同节点 A，由于 A 在老集合中的位置A._mountIndex = 0，l stIndex = 2，此时 A._mountIndex < lastIndex，因此对 A 进行移动操作；更新 lastIndex = 2，

将 A 的位置更新为新集合中的位置，nextIndex++ 进入下一个节点的判断。

当完成新集合中所有节点 diff 时，最后还需要对老集合进行循环遍历，判断是否存在新集合中没有但集合中仍存在的节点，发现存在这样的节点 D，因此删除节点 D，到此 diff 全部完成。

```
_updateChildren: function(nextNestedChildrenElements, transaction, context) {
    var prevChildren = this._renderedChildren;
    var nextChildren = this._reconcilerUpdateChildren(
    prevChildren, nextNestedChildrenElements, transaction, context
);
    if (!nextChildren && !prevChildren) {
        return;
    }
    var name;
    var lastIndex = 0;
    var nextIndex = 0;
    for (name in nextChildren) {
        if (!nextChildren.hasOwnProperty(name)) {
         continue;
    }
        var prevChild = prevChildren && prevChildren[name];
        var nextChild = nextChildren[name];
        if (prevChild === nextChild) {
        // 移动节点
        this.moveChild(prevChild, nextIndex, lastIndex);
        lastIndex = Math.max(prevChild._mountIndex, lastIndex);
        prevChild._mountIndex = nextIndex;
        } else {
        if (prevChild) {
            lastIndex = Math.max(prevChild._mountIndex, lastIndex);
            // 删除节点
            this._unmountChild(prevChild);
        }
        // 初始化并创建节点
        this._mountChildAtIndex(
            nextChild, nextIndex, transaction, context
        );
        }
        nextIndex++;
    }
    for (name in prevChildren) {
        if (prevChildren.hasOwnProperty(name) &&
            !(nextChildren && nextChildren.hasOwnProperty(name))) {
        this._unmountChild(prevChildren[name]);
        }
    }
this._renderedChildren = nextChildren;
},
// 移动节点
moveChild: function(child, toIndex, lastIndex) {
    if (child._mountIndex < lastIndex) {
        this.prepareToManageChildren();
        enqueueMove(this, child._mountIndex, toIndex);
    }
```

```
},
// 创建节点
createChild: function(child, mountImage) {
    this.prepareToManageChildren();
    enqueueInsertMarkup(this, mountImage, child._mountIndex);
},
// 删除节点
removeChild: function(child) {
    this.prepareToManageChildren();
    enqueueRemove(this, child._mountIndex);
},

_unmountChild: function(child) {
    this.removeChild(child);
    child._mountIndex = null;
},

_mountChildAtIndex: function(
    child,
    index,
    transaction,
    context) {
    var mountImage = ReactReconciler.mountComponent(
        child,
        transaction,
        this,
        this._nativeContainerInfo,
        context
    );
    child._mountIndex = index;
    this.createChild(child, mountImage);
},
```

本文整理至知乎，如有侵权请联系作者删除！

原文链接：React 虚拟 DOM 浅析