34 真正理解虚拟DOM（二）——Diff算法 & 面试考点解析

更新时间：2020-06-04 10:52:18

> 先相信你自己，然后别人才会相信你。——屠格涅夫

diff 算法

前面咱们说到，React 并不会蠢蠢地每次都给你重新生成一棵全新的 DOM 树，它会机智地对比当下 Virtual DOM 树和旧 Virtual DOM 树之间的差别，然后找出两者的不同之处，从而确保每次只做最必要的改动。

这个“找不同”的过程——调和过程（Reconciliation），是个重点（敲黑板）。

这里有坑：在一些面经中，作者会尝试把调和过程的考察和本节要讲的 diff 算法划等号——注意，这样的面经大概率写在 React16 发布之前。现在，如果面试官问你“调和过程是什么样的”，你首先要反问他“是 Fiber 调和过程，还是固有调和过程？”。由于 Fiber 架构目前很少有团队大范围地应用，所以大概率他仍然对我们本节描述的固有调和过程更感兴趣。下节，我们会为大家介绍 React16 开始采纳的 Fiber 架构。

按照传统算法的思路，比对两颗树形数据结构间的不同，需要递归逐个对比两棵树的节点，其复杂度是 O(n^3)。

这是个啥概念？比如说你做了个小项目，这个页面里有 100 个节点，走一次 `diff` 流程就要执行 100^3 = 1000000 次操作——这仅仅是一次的工作量！这样大规模的运算，浏览器是吃不消的——JS 虽快，也架不住你三次方哈。

很显然，机智的 React 团队没有采纳这种愚蠢的算法。他们采纳了一种复杂度仅为 O(n）的 `diff` 算法——现在，100个节点走一次 `diff` 只需要对比 100 次，对浏览器来说不费吹灰之力。实际上，这个算法也确实是 React 的一大亮点，React 团队给自家框架贴“高性能”标签那绝对是有理有据。

Diff 算法既是亮点，也是考点。下面我们就一起来看看这个 O(n）复杂度的算法是如何实现的:

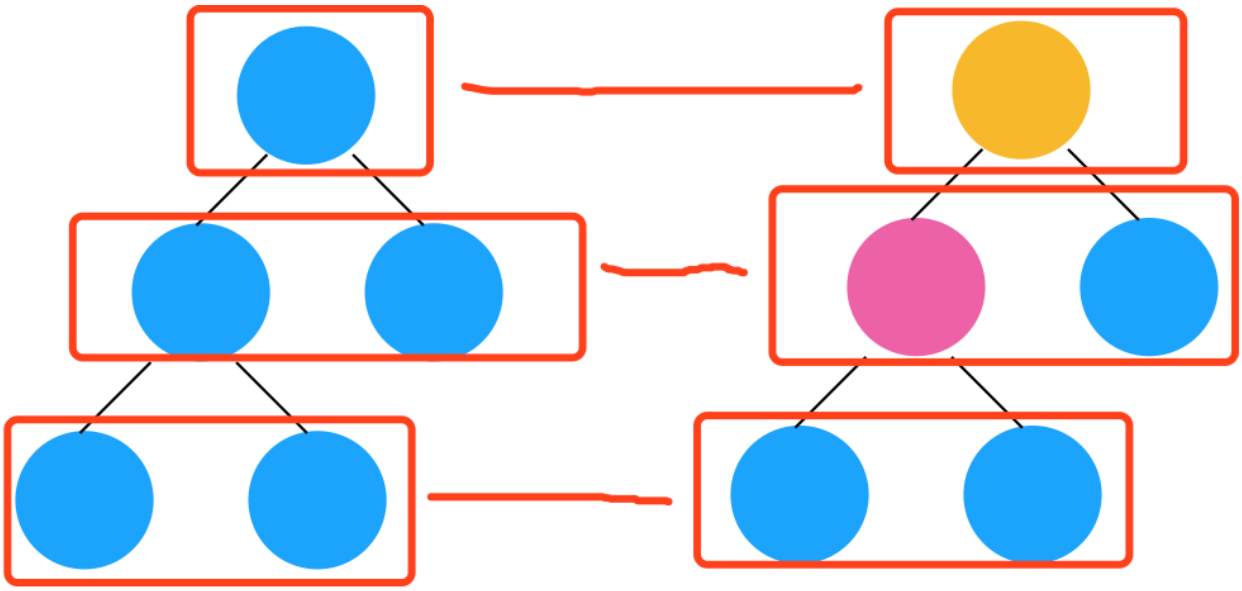
React 团队根据前端界面的特性，作了这样的假设：

相同的组件有着相同的 DOM 结构，不同的组件有着不同的 DOM 结构

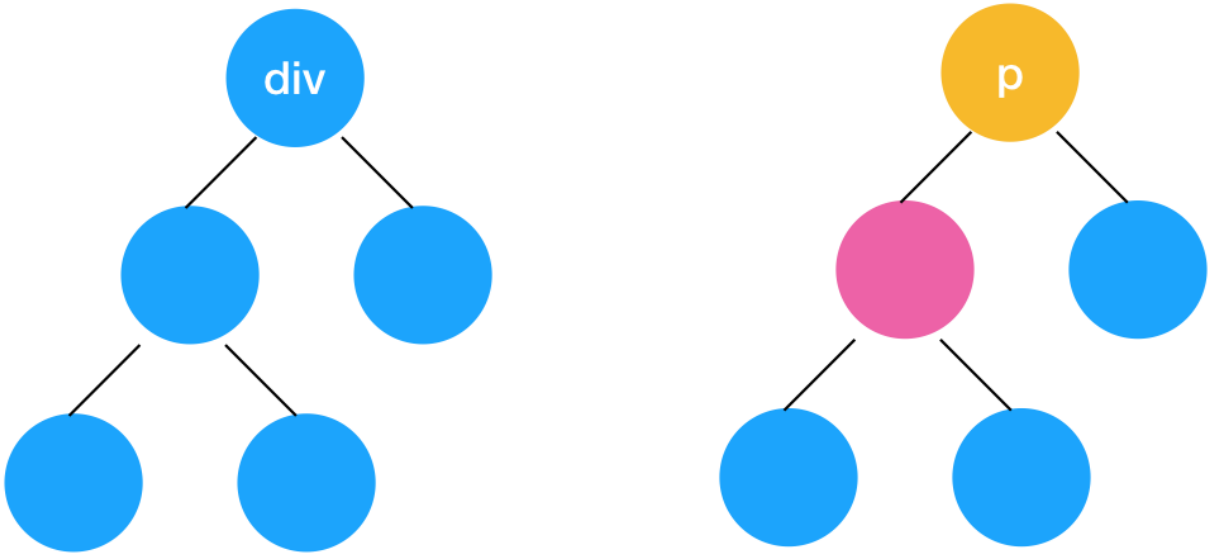
位于同一层次的一组子节点，它们之间可以通过唯一的 id 进行区分

DOM 结构中，跨层级的节点操作非常少，可以忽略不计

这样的假设蕴含着什么样的“天机”呢？简单说就是，首先，当我们考虑两棵树的“不同”时，可以一层一层来考虑，也就是“逐层对比”（如下图所示的关系）。



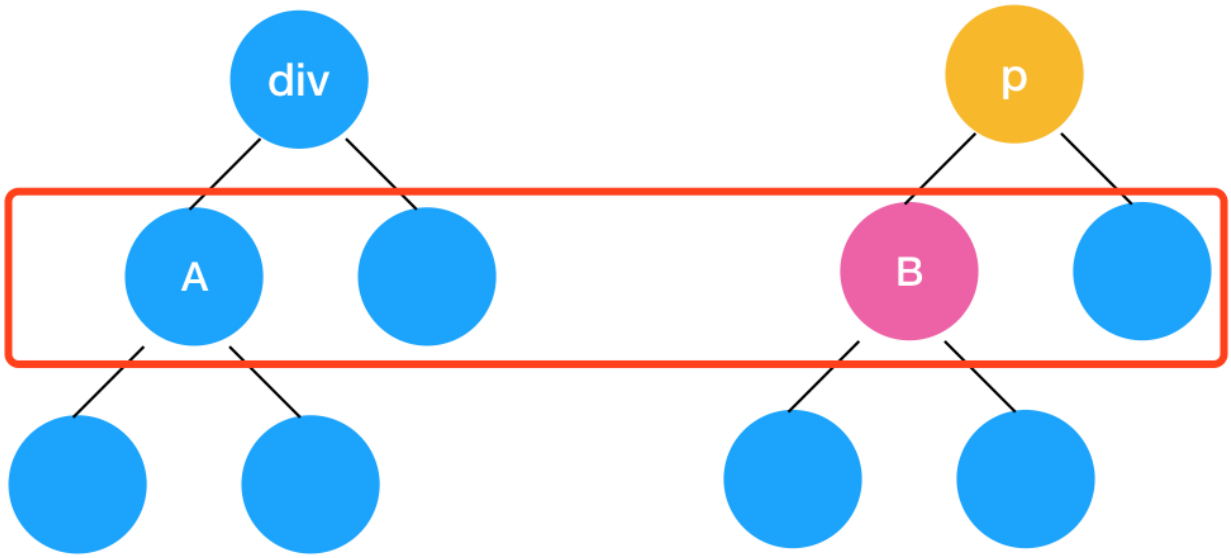
当对比两棵树时，`diff` 算法会优先比较两棵树的根节点，如果它们的类型不同，比如说之前是 div，现在变成 p 了：



那么就认为这两棵树完全不同，**这是两个完全不同的组件**。因此也没有必要再往下再比对子节点了，直接把 div 删掉，重建为 p。也就是卸载旧组件、挂载新组件。注意，这里是“重新挂载”，而非简单的“更新”。

若根节点类型相同，React 才会认为“你没变，你还是那个组件”。接下来，在保留这个组件的基础上，检查其属性的变化，然后根据属性变化的情况去更新组件。

处理完根节点这个层次的对比，React 会继续跳到下个层次去对比根节点的子节点们：



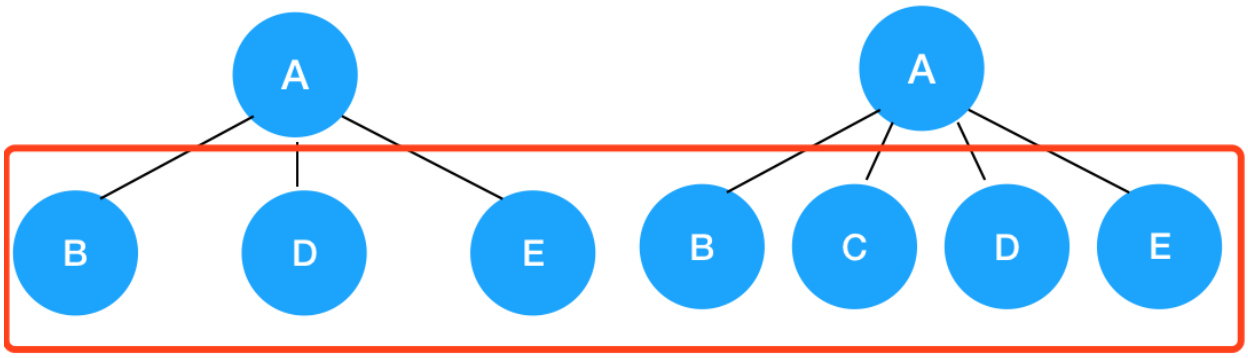
子节点的对比思路和根节点是一致的：比如说上面咱们看到 A 变成了 B，那么 React 会认为 A 和 B 的子节点都没有对比的必要了——爹都不是一个，儿子咋可能长一样呢？于是直接从 A 节点开始，把它和相关子节点一起删除重建为 B 及其子节点。

数组动态生成的组件，为什么一定要有“key”？

我们先来看看 React 是怎么介绍 key 的:

> Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a **stable** identity.
>
> key 是用来帮助 react 识别哪些内容被更改、添加或者删除。key 需要写在用数组渲染出来的元素内部，并且需要赋予其一个**稳定**的值。稳定在这里很重要，因为如果 key 值发生了变更，react 则会触发 UI 的重渲染。这是一个非常有用的特性。

结合咱们前面对 diff 的分析，我们来看这样的两个 Virtual DOM 树：



按照既有的思路，我们对第二层节点的对比过程是这样的：

对比 B 和 B，没变化，不动

对比 D 和 C，节点类型都不一样了，直接删掉 D 重建 C

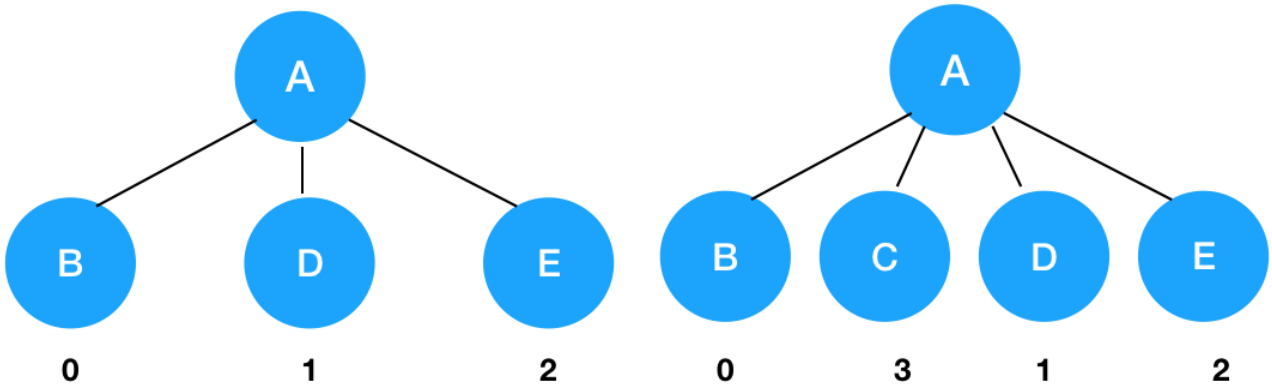
对比 E 和 D，节点类型都不一样了，直接删掉 E 重建 D

对比 空 和 E，发现 E 是一个新增节点，于是新增一个 E

你会发现，在这个过程里，C、D、E　三个子组件，明明都是现成可以用的东西，我们却大费周章地又把它们给重建了一遍。这很明显是不必要的。如何复用已经存在的组件？答案就是给它打上一个“记号”，让 React 知道，这是我们的老朋友，是没有发生变化的。这个“记号”就是“key”，举个例子：

```
const todoItems = todos.map((todo) =>
  • {todo.text}
)
```

当我们像这样尝试给每一个数组元素对应的节点都打上一个 id 作为“key”时，在 React 眼里，这些节点就是可追踪的。现在我们假设我们的 Virtual DOM 树中，B 的 id 是 0，D 的 id 是 1，E 的 id 是 2，　C 的 id 是 3：



当 C 被插入到 B 和 D 之间时，React 并不会再认为 C、D、E 这三个坑位都需要被重建——它会通过识别 id，意识到 D 和 E 都并没有发生变化，它们只是被调整了顺序而已。

tips：作为一个节点的唯一标识，在使用 key 之前，请务必确认 key 的**唯一**和**稳定**。

【发散题目】灵魂拷问：虚拟 DOM 真的性能更好吗？

这些年每次我问到候选人：你觉得 Virtual DOM 好在哪时，90% 的人都会脱口而出说“性能好”。

然后当我追问“为什么 Virtual DOM 性能好呢？”的时候，对方又往往闪烁其词，很少有能自圆其说的。

有人说“虚拟 DOM 比手动操作真实 DOM 要快，所以虚拟 DOM 性能更好”。

这种说法是错的。因为同样的 DOM 更改，你使用原生 JS 手动去操作，是不需要走 diff 流程的，实际上要快一些。

React 没有这么嚣张，它从来没有声称“我比直接操作 DOM 更快”。咱们讨论 React 的性能，要看跟谁比：

> 跟直接操作 DOM 比，那恐怕还是直接操作 DOM 快一些。不过你想啊，直接操作 DOM 可是个苦力活，咱们早就淘汰这种操作了，不然哪来的模板呢？
>
> 跟模板比，这就有的比了。模板每次数据变更时，它会直接重置 DOM——哪怕变更的只是一小段数据。相比之下，虚拟 DOM 方案每次只更新必要的 DOM，虽然它增加了 diff 过程。但增加的是 js 计算，换来的可是 DOM 开销，这可是杠杆撬地球一般的操作。。。所以说，这种情况下，我们会认为虚拟 DOM 从性能上来讲会更合适。

总结

React 并不承诺比直接操作 DOM 更快。但是别忘了，React 可是为我们提供了数据驱动、不用操心 DOM 细节的便利，在这个基础上，它仍然维持了一个过得去的性能。React 的强大之处不在于性能碾压了xx、碾压了xxx，而在于它在性能和开发体验&可维护性之间做到了一个很好的平衡。

}

33 真正理解虚拟DOM（一）——虚拟DOM思想

35 React16题眼：理解Fiber思想