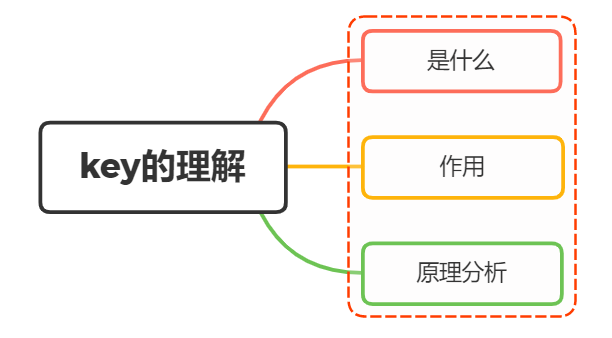
# 面试官：你知道vue中key的原理吗？说说你对它的理解



## 一、Key是什么

开始之前，我们先还原两个实际工作场景

1. 当我们在使用v-for时，需要给单元加上key

<ul>  
 <li v-for="item in items" :key="item.id">...</li>  
</ul>

1. 用+new Date()生成的时间戳作为key，手动强制触发重新渲染

* <Comp :key="+new Date()" />

那么这背后的逻辑是什么，key的作用又是什么？

一句话来讲

key是给每一个vnode的唯一id，也是diff的一种优化策略，可以根据key，更准确， 更快的找到对应的vnode节点

### 场景背后的逻辑

当我们在使用v-for时，需要给单元加上key

* 如果不用key，Vue会采用就地复地原则：最小化element的移动，并且会尝试尽最大程度在同适当的地方对相同类型的element，做patch或者reuse。
* 如果使用了key，Vue会根据keys的顺序记录element，曾经拥有了key的element如果不再出现的话，会被直接remove或者destoryed

用+new Date()生成的时间戳作为key，手动强制触发重新渲染

* 当拥有新值的rerender作为key时，拥有了新key的Comp出现了，那么旧key Comp会被移除，新key Comp触发渲染

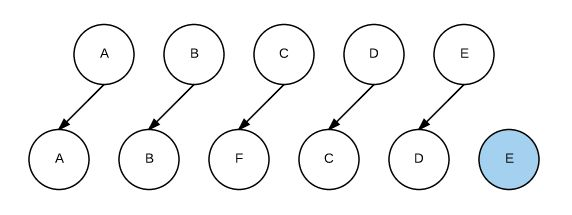
## 二、设置key与不设置key区别

举个例子：

创建一个实例，2秒后往items数组插入数据

<body>  
 <div id="demo">  
 <p v-for="item in items" :key="item">{{item}}</p>  
 </div>  
 <script src="../../dist/vue.js"></script>  
 <script>  
 // 创建实例  
 const app = new Vue({  
 el: '#demo',  
 data: { items: ['a', 'b', 'c', 'd', 'e'] },  
 mounted () {  
 setTimeout(() => {   
 this.items.splice(2, 0, 'f') //   
 }, 2000);  
 },  
 });  
 </script>  
</body>

在不使用key的情况，vue会进行这样的操作：



分析下整体流程：

* 比较A，A，相同类型的节点，进行patch，但数据相同，不发生dom操作
* 比较B，B，相同类型的节点，进行patch，但数据相同，不发生dom操作
* 比较C，F，相同类型的节点，进行patch，数据不同，发生dom操作
* 比较D，C，相同类型的节点，进行patch，数据不同，发生dom操作
* 比较E，D，相同类型的节点，进行patch，数据不同，发生dom操作
* 循环结束，将E插入到DOM中

一共发生了3次更新，1次插入操作

在使用key的情况：vue会进行这样的操作：

* 比较A，A，相同类型的节点，进行patch，但数据相同，不发生dom操作
* 比较B，B，相同类型的节点，进行patch，但数据相同，不发生dom操作
* 比较C，F，不相同类型的节点
* 比较E、E，相同类型的节点，进行patch，但数据相同，不发生dom操作
* 比较D、D，相同类型的节点，进行patch，但数据相同，不发生dom操作
* 比较C、C，相同类型的节点，进行patch，但数据相同，不发生dom操作
* 循环结束，将F插入到C之前

一共发生了0次更新，1次插入操作

通过上面两个小例子，可见设置key能够大大减少对页面的DOM操作，提高了diff效率

### 设置key值一定能提高diff效率吗？

其实不然，文档中也明确表示

当 Vue.js 用 v-for 正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序， 而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素

这个默认的模式是高效的，但是只适用于不依赖子组件状态或临时 DOM 状态 (例如：表单输入值) 的列表渲染输出

建议尽可能在使用 v-for 时提供 key，除非遍历输出的 DOM 内容非常简单，或者是刻意依赖默认行为以获取性能上的提升

## 三、原理分析

源码位置：core/vdom/patch.js

这里判断是否为同一个key，首先判断的是key值是否相等如果没有设置key，那么key为undefined，这时候undefined是恒等于undefined

function sameVnode (a, b) {  
 return (  
 a.key === b.key && (  
 (  
 a.tag === b.tag &&  
 a.isComment === b.isComment &&  
 isDef(a.data) === isDef(b.data) &&  
 sameInputType(a, b)  
 ) || (  
 isTrue(a.isAsyncPlaceholder) &&  
 a.asyncFactory === b.asyncFactory &&  
 isUndef(b.asyncFactory.error)  
 )  
 )  
 )  
}

updateChildren方法中会对新旧vnode进行diff，然后将比对出的结果用来更新真实的DOM

function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue, removeOnly) {  
 ...  
 while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {  
 if (isUndef(oldStartVnode)) {  
 ...  
 } else if (isUndef(oldEndVnode)) {  
 ...  
 } else if (sameVnode(oldStartVnode, newStartVnode)) {  
 ...  
 } else if (sameVnode(oldEndVnode, newEndVnode)) {  
 ...  
 } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved right  
 ...  
 } else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode moved left  
 ...  
 } else {  
 if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx, oldEndIdx)  
 idxInOld = isDef(newStartVnode.key)  
 ? oldKeyToIdx[newStartVnode.key]  
 : findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx)  
 if (isUndef(idxInOld)) { // New element  
 createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm, false, newCh, newStartIdx)  
 } else {  
 vnodeToMove = oldCh[idxInOld]  
 if (sameVnode(vnodeToMove, newStartVnode)) {  
 patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue, newCh, newStartIdx)  
 oldCh[idxInOld] = undefined  
 canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm, oldStartVnode.elm)  
 } else {  
 // same key but different element. treat as new element  
 createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm, false, newCh, newStartIdx)  
 }  
 }  
 newStartVnode = newCh[++newStartIdx]  
 }  
 }  
 ...  
}

## 参考文献

* https://juejin.cn/post/6844903826693029895
* https://juejin.cn/post/6844903985397104648
* https://vue3js.cn/docs/zh