Diff算法解析

diff算法主要在于新老dom的对比算法上面，可以通过尽可能多的dom元素的复用来避免大量重复dom的重新渲染，造成了资源的浪费；diff对比分为以下几方面：

1. 首先是对一些特殊的情况的对比：
    - 对比标签：对比标签是否一致，如果不一致直接进行替换，如果一致但是文本不一致则是替换文本即可，
    - 对比属性：用新的虚拟节点的属性和老得属性相对比，去更新节点，
        - 对比style样式有何区别，如果老的有新的没有直接置空
        - 如果老的有新的没有就删除老的多余的属性
        - 如果新的有，无论是style还是class 直接用新的去做更新就可以了
    - 对比子元素： 用新的虚拟节点的属性和老得虚拟节点做对比
        - 老的有儿子新的没有，清空即可
        - 新的有儿子老得没有，直接appendChild追加即可
        - 新老都有需要比对儿子
2. 新老都有需要比对儿子的过程就叫做diff算法，采用的是双指针的方式进行对比标记，给新老节点分别创建一个头指针和尾指针
    - 头指针和头指针对比：如果两个是同一个标签名，那么就需要对比儿子，更新属性以及再去递归更新子节点，新老节点的头指针都需要移动
    - 尾指针和尾指针相对比：如果两个是同一个标签名，那么就需要对比儿子，更新属性以及再去递归更新子节点，新老节点的尾指针都需要移动
    - 反转对比：新的开始和老的尾部相比较，相同的话对比儿子递归patch方法，老的尾指针插入到老的开始指针的前面，注意一直是新指针的后面，然后新得开始指针向后移动，老得结束指针向前移动，开始新得一轮
    - 反转对比：老得开始和新的尾部相比较，相同的话对比儿子递归patch方法，老得头指针通过insertBefore插入到老得尾指针的后面，注意一直是老指针的后面，然后老的头指针向后移动，新的尾指针向前移动，开始新得一轮
    - 暴力对比：如果儿子都不相同或者只有部分相同就需要进行暴力对比，无论头尾交叉都不相等，此时采用特殊的操作，将新节点p2作为循环条件，使用头尾指针，查找p2的元素是否在p1当中，如果不存在就放在p1头指针的前面，如果存在就将p1的这个元素移动到头指针的前面，为了避免移动元素操作数组会发生塌陷问题，所以使用空值进行占位，直至p1的头尾指针相遇，循环结束，然后将p1的头尾指针部分全部都删除，所以此时增加一项优化项：如果老得节点的头指针或者尾指针出现空值情况则直接跳过
    - 如果新节点的开始指针和结束指针如果重合了，将新的多余的插入进去即可，可能是向前添加还有可能是向后添加，统一使用insertBefore就可以实现

使用对比的方法

```
import {compileToFunction} from './compiler/index.js';
import { patch,createElm } from './vdom/patch';
// 1.创建第一个虚拟节点
let vm1 = new Vue({data:{name:'l1'}});
let render1 = compileToFunction('<div>{{name}}</div>')
let oldVnode = render1.call(vm1)
```

```
// 2.创建第二个虚拟节点
let vm2 = new Vue({data:{name:'l2'}});
let render2 = compileToFunction('<p>{{name}}</p>');
let newVnode = render2.call(vm2);
// 3.通过第一个虚拟节点做首次渲染
let el = createElm(oldVnode)
document.body.appendChild(el);
// 4.调用patch方法进行对比操作
patch(oldVnode,newVnode);
```

一.基本Diff算法

1. 比对标签: 在diff过程中会先比较标签是否一致，如果标签不一致用新的标签替换掉老的标签, 如果标签一致但是不存在则是文本节点那就比较文本的内容即可

```
function patch(oldVnode, vnode) {
    // 1. 如果标签不一致说明是两个不同元素，直接就替换掉就可以了
    if(oldVnode.tag !== vnode.tag){
        oldVnode.el.parentNode.replaceChild(createElm(vnode),oldVnode.el)
    }
    // 2. 如果标签一致但是不存在则是文本节点
    if(!oldVnode.tag){
        if(oldVnode.text !== vnode.text){
            oldVnode.el.textContent = vnode.text;
        }
    }
}
```

2. 标签一样比对属性: 当标签相同时，我们可以复用老的标签元素，并且进行属性的比对。

```
function patch(oldVnode, vnode) {
   let el = vnode.el = oldVnode.el; // 复用标签,并且更新属性
   // 更新属性，用新的虚拟节点的属性和老得属性相对比，去更新节点
   updateProperties(vnode,oldVnode.data);
}

function updateProperties(vnode,oldProps={}) {
    let newProps = vnode.data || {}; // 新的属性
    let el = vnode.el; // 真实的dom属性
    // 1. 比对样式 老得style={ color: red;} 新的style={bacground:red;}
    let newStyle = newProps.style || {};
    let oldStyle = oldProps.style || {};
    for(let key in oldStyle){
        if(!newStyle[key]){
            el.style[key] = ''
        }
    }
    // 2. 老得有 新的没有删除多余属性
    for(let key in oldProps){
```

```
            if(!newProps[key]){
                el.removeAttribute(key);
            }
        }
        // 3. 新的有，那就直接用新的去做更新就可以了
        for (let key in newProps) {
            if (key === 'style') {
                for (let styleName in newProps.style) {
                    el.style[styleName] = newProps.style[styleName];
                }
            } else if (key === 'class') {
                el.className = newProps.class;
            } else {
                el.setAttribute(key, newProps[key]);
            }
        }
    }
```

3. 比对子元素

```
// 比较孩子节点
let oldChildren = oldVnode.children || [];
let newChildren = vnode.children || [];

if(oldChildren.length > 0 && newChildren.length > 0){ // 新老都有需要比对儿子
- diff算法
    updateChildren(el, oldChildren, newChildren)

} else if(oldChildren.length > 0 ){ // 老的有儿子新的没有，清空即可
    el.innerHTML = '';

} else if(newChildren.length > 0){ // 新的有儿子老得没有
    for(let i = 0 ; i < newChildren.length ;i++){
        let child = newChildren[i];
        el.appendChild(createElm(child));
    }
}
```

二. Diff中的优化策略

vue当中diff做了很多优化，dom中操作会有很多常见的逻辑，把节点插入到当前儿子的头部、尾部、儿子正叙倒叙，vue2中采用的是双指针的方式

1. 在开头和结尾新增元素： 需要做一个循环，同时循环老得和新的，哪个先结束循环就先停止，将多余的删除或者添加进去 即 `a b c -> a b c d`或者`d a b c -> a b c`
    - 首先我们需要给新老节点分别创建一个头指针和尾指针

```
function isSameVnode(oldVnode,newVnode){
    // 如果两个人的标签和key 一样我认为是同一个节点 虚拟节点一样我就可以复用真实节点了
    return (oldVnode.tag === newVnode.tag) && (oldVnode.key === newVnode.key)
}
function updateChildren(parent, oldChildren, newChildren) {
    //  1. 给新老节点分别创建一个头指针和尾指针
    let oldStartIndex = 0; // 老得索引开始
    let oldStartVnode = oldChildren[0]; // 老得索引指向的节点
    let oldEndIndex = oldChildren.length - 1; // 老得索引结束
    let oldEndVnode = oldChildren[oldEndIndex]; // 老得索引指向的节点

    let newStartIndex = 0;
    let newStartVnode = newChildren[0];
    let newEndIndex = newChildren.length - 1;
    let newEndVnode = newChildren[newEndIndex];

    // oldStartIndex <= oldEndIndex  老节点头尾相碰了
    // newStartIndex <= newEndIndex 是因为我们不知道谁是长的谁是少的，比较谁先结束用谁的判断条件
    while (oldStartIndex <= oldEndIndex && newStartIndex <= newEndIndex) {
        // 优化向后追加逻辑
        if(isSameVnode(oldStartVnode,newStartVnode)){ // 头指针和头指针对比 如果两个是同一个标签名，那么就需要对比儿子
            patch(oldStartVnode,newStartVnode); // 更新属性以及再去递归更新子节点
            oldStartVnode = oldChildren[++oldStartIndex]; // 指针移动
            newStartVnode = newChildren[++newStartIndex]; // 指针移动
        }else if(isSameVnode(oldEndVnode,newEndVnode)){   // 优化向前追加逻辑，尾指针和尾指针相对比
            patch(oldEndVnode,newEndVnode); // 比较孩子
            oldEndVnode = oldChildren[--oldEndIndex];
            newEndVnode = newChildren[--newEndIndex];
        }
    }
    // while循环完事儿，新节点的开始指针和结束指针如果重合了
    if(newStartIndex <= newEndIndex){
        for(let i = newStartIndex ; i<=newEndIndex ;i++){
            // 将新的多余的插入进去即可，可能是向前添加还有可能是向后添加
            // 向后插入 ele = null
            // 向前插入 ele 就是当钱向谁前面插入
            let ele = newChildren[newEndIndex+1] == null? null:newChildren[newEndIndex+1].el;
            // 将新的多余的插入就可以了
            parent.insertBefore(createElm(newChildren[i]),ele);
        }
    }

}
```

2. 反转节点: 头部移动尾部 尾部移动到头部

a b c d -> d a b c

a b c d -> b c d a

p1 = a b c d -> p2 = d c b a

- 第一次：`pHead1 = a、pEnd1 = d、pHead2 = d、pEnd2 = a`；头和头对比发现不相同，尾和尾对比也不相同，所以进行交叉对比pHead1和pEnd2相对比发现相同，那就移动指针头指针pHead1向后移动 pHead1 = b 尾指针pEnd2向前移动 pEnd2 = b，并且将p1的节点查到最后一个节点的后面，此时 p1 = b c d a
- 第二次：`pHead1 = b、pEnd1 = d、pHead2 = d、pEnd2 = b`；头和头对比发现不相同，尾和尾对比也不相同，所以进行交叉对比pHead1和pEnd2相对比发现相同，那就移动指针头指针pHead1向后移动 pHead1 = c 尾指针pEnd2向前移动 pEnd2 = c，并且将p1的节点查到最后一个节点的后面，此时 p1 = c d b a
- 第三次：`pHead1 = c、pEnd1 = d、pHead2 = d、pEnd2 = c`；头和头对比发现不相同，尾和尾对比也不相同，所以进行交叉对比pHead1和pEnd2发现相同，将pHead1移动到最后一个节点的前面，此时 p1 = d c b a，循环结束了

```
// 头移动到尾部 老的尾部和新的头部相比较
else if(isSameVnode(oldStartVnode,newEndVnode)){
    patch(oldStartVnode,newEndVnode); // 对比儿子
    parent.insertBefore(oldStartVnode.el,oldEndVnode.el.nextSibling); // 移
动元素
    oldStartVnode = oldChildren[++oldStartIndex];
    newEndVnode = newChildren[--newEndIndex]
// 尾部移动到头部 老得尾和新的头
}else if(isSameVnode(oldEndVnode,newStartVnode)){
    patch(oldEndVnode,newStartVnode);
    parent.insertBefore(oldEndVnode.el,oldStartVnode.el);
    oldEndVnode = oldChildren[--oldEndIndex];
    newStartVnode = newChildren[++newStartIndex]
}
```

> 为什么要使用key，不能使用索引作为key值
>
> 如果是静态渲染的情况下是没有什么问题的，但是当页面元素要进行倒叙的话就会该复用的没有复用重新创建了儿子节点，这样就造成了浪费

3. 暴力比对：都不相等的情况下就需要进行暴力对比

p1 = c e d m

p2 = a d e q

无论头尾交叉都不相等，此时采用特殊的操作，将新节点p2作为循环条件，使用头尾指针，查找p2的元素是否在p1当中，如果不存在就放在p1头指针的前面，如果存在就将p1的这个元素移动到头指针的前面，为了避免移动元素操作数组会发生塌陷问题，所以使用空值进行占位，直至p1的头尾指针相遇，循环结束，然后将p1的头尾指针部分全部都删除

第一次：pH2 = a pH1 = c pE1 = m pE2 = q， 判断pH2在p1内部有没有，没有，插至p1前面 p1 = a c e d m，pH2向后移动

第二次：pH2 = d pH1 = c pE1 = m pE2 = q，判断pH2在p1内部有没有，有，移动p1内部元素，即 p1 = a d c e null m，pH2向后移动

第三次：pH2 = e pH1 = c pE1 = m pE2 = q，判断pH2在p1内部有没有，有，移动p1内部元素，即 p1 = a d e c null null m，pH2向后移动

第四次：pH2 = q pH1 = c pE1 = m pE2 = q，判断pH2在p1内部有没有，没有，执行插入操作，即 p1 = a d e q c null null m

第五次：循环结束，然后删除 c - m的元素，剩下的就是比对过后的

```
function makeIndexByKey(children) { // 搞一个索引的map表
    let map = {};
    children.forEach((item, index) => {
        map[item.key] = index // { 0: A, 1: B, 2: C}
    });
    return map;
}
let map = makeIndexByKey(oldChildren);

if(isSameVnode(oldEndVnode,newStartVnode)) {
   ....
} else { // 儿子都不相同或者一点点相同的
   // 对所有的孩子元素进行编号
   let moveIndex = map[newStartVnode.key]; // 拿到开头的虚拟节点的key 去老得中找
   if (moveIndex == undefined) { // 老的中没有将新元素插入
       parent.insertBefore(createElm(newStartVnode), oldStartVnode.el);
   } else { // 有的话做移动操作
       let moveVnode = oldChildren[moveIndex]; // 这个老节点需要移动
       oldChildren[moveIndex] = undefined; // 要移动的元素置空
       parent.insertBefore(moveVnode.el, oldStartVnode.el);
       patch(moveVnode, newStartVnode); // 对比儿子节点
   }
   newStartVnode = newChildren[++newStartIndex]
}
// 如果老得节点多于新的节点，就说明老得节点还没有处理的，说明这些节点都是不需要的节点，如果这里面有undefined说明这个节点已经是被处理过的，那就跳过就可以了，如果有值那么就需要将老得多的删除掉
if(oldStartIndex <= oldEndIndex){
    for(let i = oldStartIndex; i<=oldEndIndex;i++){
        let child = oldChildren[i];
        if(child != undefined){
            parent.removeChild(child.el)
        }
    }
}
```

4. 在比对过程中，可能出现空值情况则直接跳过

```
while (oldStartIndex <= oldEndIndex && newStartIndex <= newEndIndex) {
   if(!oldStartVnode){ // 如果老的头指针指向了null，跳过这次处理
       oldStartVnode = oldChildren[++oldStartIndex];
   }else if(!oldEndVnode){ // 如果老的尾指针指向了null，跳过这次处理
       oldEndVnode = oldChildren[--oldEndIndex]
   } else if(isSameVnode(oldStartVnode,newStartVnode)) {
       ...
   }
}
```

三 更新操作

```
Vue.prototype._update = function (vnode) {
    const vm  = this;
    const prevVnode = vm._vnode; // 保留上一次的vnode
    vm._vnode = vnode;
    if(!prevVnode){
        vm.$el = patch(vm.$el,vnode); // 需要用虚拟节点创建出真实节点 替换掉 真实的$el
        // 我要通过虚拟节点 渲染出真实的dom
    }else{
        vm.$el = patch(prevVnode,vnode); // 更新时做diff操作
    }
}
```