**1. 当数据发生变化时，vue是怎么更新节点的？**

要知道渲染真实DOM的开销是很大的，比如有时候我们修改了某个数据，如果直接渲染到真实dom上会引起整个dom树的重绘和重排

我们先根据真实DOM生成一颗virtual DOM，当virtual DOM某个节点的数据改变后会生成一个新的Vnode，然后Vnode和oldVnode作对比，发现有不一样的地方就直接修改在真实的DOM上，然后使oldVnode的值为Vnode。

diff的过程就是调用名为patch的函数，比较新旧节点，一边比较一边给**真实的DOM**打补丁。

**2.virtual dom和真实dom 的区别**

virtual DOM是将真实的DOM的数据抽取出来，以对象的形式模拟树形结构。

<div>
<p>123</p>
</div>

对应的virtual DOM（伪代码）：

var Vnode = {
tag: 'div',
children: [
{ tag: 'p', text: '123' }
]
};

**当数据发生改变时，set方法会让调用Dep.notify通知所有订阅者Watcher，订阅者就会调用patch给真实的DOM打补丁，更新相应的视图。**

来看看 **patch**是怎么打补丁的（代码只保留核心部分）

function patch (oldVnode, vnode) {
// some code
if (**sameVnode(oldVnode, vnode)**) {//当前节点相等可以继续进行比较子节点
**patchVnode(oldVnode, vnode)**
} else {

//若不相等 直接全部替换掉
const oEl = oldVnode.el // 当前oldVnode对应的真实元素节点
let parentEle = api.parentNode(oEl) // 父元素
createEle(vnode) // 根据Vnode生成新元素
if (parentEle !== null) {
api.insertBefore(parentEle, vnode.el, api.nextSibling(oEl)) // 将新元素添加进父元素
api.removeChild(parentEle, oldVnode.el) // 移除以前的旧元素节点
oldVnode = null
}
}
// some code
return vnode
}

patch函数接收两个参数oldVnode和Vnode分别代表新的节点和之前的旧节点

**判断两节点是否值得比较，值得比较则执行patchVnode**

function **sameVnode** (a, b)

不值得比较则用Vnode替换oldVnode

如果两个节点都是一样的，那么就深入检查他们的子节点。如果两个节点不一样那就说明Vnode完全被改变了，就可以直接替换oldVnode。

**patchVnode**

当我们确定两个节点值得比较之后我们会对两个节点指定patchVnode方法。那么这个方法做了什么呢？

**patchVnode (oldVnode, vnode) {**
const el = vnode.el = oldVnode.el//找到对应的真实dom，称为el
let i, oldCh = oldVnode.children, ch = vnode.children
if (oldVnode === vnode) return//判断Vnode和oldVnode是否指向同一个对象，如果是，那么直接return
if (oldVnode.text !== null && vnode.text !== null && oldVnode.text !== vnode.text) {
api.setTextContent(el, vnode.text)//如果他们都有文本节点并且不相等，那么将el的文本节点设置为Vnode的文本节点。
}else {
updateEle(el, vnode, oldVnode)
if (oldCh && ch && oldCh !== ch) {
updateChildren(el, oldCh, ch)//如果不相等更新
}else if (ch){
createEle(vnode) //create el's children dom//如果oldVnode没有子节点而Vnode有，则将Vnode的子节点真实化之后添加到el
}else if (oldCh){
api.removeChildren(el)//如果oldVnode有子节点而Vnode没有，则删除el的子节点
}
}
}

**updateChildren**

代码量很大，不方便一行一行的讲解，所以下面结合一些示例图来描述一下。

**updateChildren (parentElm, oldCh, newCh) {**
let oldStartIdx = 0, newStartIdx = 0
let oldEndIdx = oldCh.length - 1
let oldStartVnode = oldCh[0]
let oldEndVnode = oldCh[oldEndIdx]
let newEndIdx = newCh.length - 1
let newStartVnode = newCh[0]
let newEndVnode = newCh[newEndIdx]
let oldKeyToIdx
let idxInOld
let elmToMove
let before
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
if (oldStartVnode == null) { // 对于vnode.key的比较，会把oldVnode = null
oldStartVnode = oldCh[++oldStartIdx]
}else if (oldEndVnode == null) {
oldEndVnode = oldCh[--oldEndIdx]
}else if (newStartVnode == null) {
newStartVnode = newCh[++newStartIdx]
}else if (newEndVnode == null) {
newEndVnode = newCh[--newEndIdx]
}else if (sameVnode(oldStartVnode, newStartVnode)) {
patchVnode(oldStartVnode, newStartVnode)
oldStartVnode = oldCh[++oldStartIdx]
newStartVnode = newCh[++newStartIdx]
}else if (sameVnode(oldEndVnode, newEndVnode)) {
patchVnode(oldEndVnode, newEndVnode)
oldEndVnode = oldCh[--oldEndIdx]
newEndVnode = newCh[--newEndIdx]
}else if (sameVnode(oldStartVnode, newEndVnode)) {
patchVnode(oldStartVnode, newEndVnode)
api.insertBefore(parentElm, oldStartVnode.el, api.nextSibling(oldEndVnode.el))
oldStartVnode = oldCh[++oldStartIdx]
newEndVnode = newCh[--newEndIdx]
}else if (sameVnode(oldEndVnode, newStartVnode)) {
patchVnode(oldEndVnode, newStartVnode)
api.insertBefore(parentElm, oldEndVnode.el, oldStartVnode.el)
oldEndVnode = oldCh[--oldEndIdx]
newStartVnode = newCh[++newStartIdx]
}else {
// 使用key时的比较
if (oldKeyToIdx === undefined) {
oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx, oldEndIdx) // 有key生成index表
}
idxInOld = oldKeyToIdx[newStartVnode.key]
if (!idxInOld) {
api.insertBefore(parentElm, createEle(newStartVnode).el, oldStartVnode.el)
newStartVnode = newCh[++newStartIdx]
}
else {
elmToMove = oldCh[idxInOld]
if (elmToMove.sel !== newStartVnode.sel) {
api.insertBefore(parentElm, createEle(newStartVnode).el, oldStartVnode.el)
}else {
patchVnode(elmToMove, newStartVnode)
oldCh[idxInOld] = null
api.insertBefore(parentElm, elmToMove.el, oldStartVnode.el)
}
newStartVnode = newCh[++newStartIdx]
}
}
}
if (oldStartIdx > oldEndIdx) {
before = newCh[newEndIdx + 1] == null ? null : newCh[newEndIdx + 1].el
addVnodes(parentElm, before, newCh, newStartIdx, newEndIdx)
}else if (newStartIdx > newEndIdx) {
removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx)
}
}

先说一下这个函数做了什么

将Vnode的子节点Vch和oldVnode的子节点oldCh提取出来

oldCh和vCh各有两个头尾的变量StartIdx和EndIdx，它们的2个变量相互比较，一共有4种比较方式。如果4种比较都没匹配，如果设置了key，就会用key进行比较，在比较的过程中，变量会往中间靠，一旦StartIdx>EndIdx表明oldCh和vCh至少有一个已经遍历完了，就会结束比较。

**图解updateChildren**

终于来到了这一部分，上面的总结相信很多人也看得一脸懵逼，下面我们好好说道说道。（这都是我自己画的，求推荐好用的画图工具...）

.

粉红色的部分为oldCh和vCh

我们将它们取出来并分别用s和e指针指向它们的头child和尾child

现在分别对oldS、oldE、S、E两两做sameVnode比较，有四种比较方式，当其中两个能匹配上那么真实dom中的相应节点会移到Vnode相应的位置，这句话有点绕，打个比方

如果是oldS和E匹配上了，那么真实dom中的第一个节点会移到最后

如果是oldE和S匹配上了，那么真实dom中的最后一个节点会移到最前，匹配上的两个指针向中间移动

如果四种匹配没有一对是成功的，分为两种情况

如果新旧子节点都存在key，那么会根据oldChild的key生成一张hash表，用S的key与hash表做匹配，匹配成功就判断S和匹配节点是否为sameNode，如果是，就在真实dom中将成功的节点移到最前面，否则，将S生成对应的节点插入到dom中对应的oldS位置，oldS和S指针向中间移动。

如果没有key,则直接将S生成新的节点插入真实DOM（ps：这下可以解释为什么v-for的时候需要设置key了，如果没有key那么就只会做四种匹配，就算指针中间有可复用的节点都不能被复用了）

再配个图（假设下图中的所有节点都是有key的，且key为自身的值）

第一步

oldS = a, oldE = d；
S = a, E = b;
复制代码

oldS和S匹配，则将dom中的a节点放到第一个，已经是第一个了就不管了，此时dom的位置为：a b d

第二步

oldS = b, oldE = d；
S = c, E = b;
复制代码

oldS和E匹配，就将原本的b节点移动到最后，因为E是最后一个节点，他们位置要一致，这就是上面说的：**当其中两个能匹配上那么真实dom中的相应节点会移到Vnode相应的位置**，此时dom的位置为：a d b

第三步

oldS = d, oldE = d；
S = c, E = d;

oldE和E匹配，位置不变此时dom的位置为：a d b

第四步

oldS++;
oldE--;
oldS > oldE;
复制代码

遍历结束，说明oldCh先遍历完。就将剩余的vCh节点根据自己的的index插入到真实dom中去，此时dom位置为：a c d b

一次模拟完成。

这个匹配过程的结束有两个条件：

**oldS > oldE表示oldCh先遍历完，那么就将多余的vCh根据index添加到dom中去（如上图）**

**S > E表示vCh先遍历完，那么就在真实dom中将区间为[oldS, oldE]的多余节点删掉**

}