**vue核心之虚拟DOM(vdom)**

**一、真实DOM和其解析流程？**

    浏览器渲染引擎工作流程都差不多，大致分为5步，**创建DOM树——创建StyleRules——创建Render树——布局Layout——绘制Painting**

    第一步，用HTML分析器，分析HTML元素，**构建一颗DOM树**(标记化和树构建)。

    第二步，用CSS分析器，分析CSS文件和元素上的inline样式，生成页面的样式表。

    第三步，将DOM树和样式表，关联起来，构建一颗Render树(这一过程又称为Attachment)。每个DOM节点都有**attach方法，接受样式信息**，返回一个render对象(又名renderer)。这些render对象最终会被构建成一颗Render树。

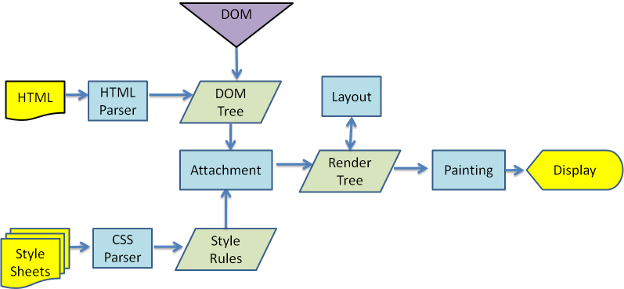
    第四步，有了Render树，浏览器开始布局，为每个Render树上的节点确定一个在显示屏上出现的精确坐标。

    第五步，Render树和节点显示坐标都有了，就调用每个节点**paint方法，把它们绘制**出来。

**DOM树的构建是文档加载完成开始的？**构建DOM数是一个渐进过程，为达到更好用户体验，渲染引擎会尽快将内容显示在屏幕上。**它不必**等到整个HTML文档解析完毕之后才开始构建render数和布局。

**Render树是DOM树和CSSOM树构建完毕才开始构建的吗？**这三个过程在实际进行的时候又不是完全独立，而是会有交叉。会造成一边加载，一遍解析，一遍渲染的工作现象。

**CSS的解析是从右往左逆向解析的**(从DOM树的下－上解析比上－下解析效率高)，**嵌套标签越多，解析越慢。**



webkit渲染引擎工作流程

**二、JS操作真实DOM的代价！**

        用我们传统的开发模式，原生JS或JQ操作DOM时，浏览器会从构建DOM树开始从头到尾执行一遍流程。在一次操作中，我需要更新10个DOM节点，浏览器收到第一个DOM请求后并不知道还有9次更新操作，因此会马上执行流程，最终执行10次。例如，第一次计算完，紧接着下一个DOM更新请求，这个节点的坐标值就变了，前一次计算为无用功。计算DOM节点坐标值等都是白白浪费的性能。即使计算机硬件一直在迭代更新，操作DOM的代价仍旧是昂贵的，频繁操作还是会出现页面卡顿，影响用户体验。

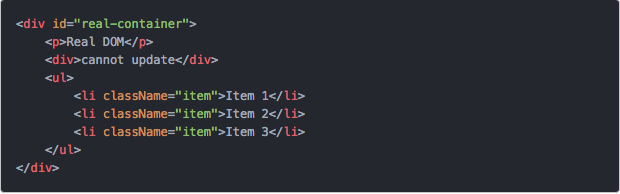
**三、为什么需要虚拟DOM，它有什么好处?**

        Web界面由DOM树(树的意思是数据结构)来构建，当其中一部分发生变化时，其实就是对应某个DOM节点发生了变化，

        虚拟DOM就是为了**解决浏览器性能问题**而被设计出来的。**如前**，若一次操作中有10次更新DOM的动作，虚拟DOM不会立即操作DOM，而是将这10次更新的diff内容保存到本地一个JS对象中，最终将这个JS对象一次性attch到DOM树上，再进行后续操作，避免大量无谓的计算量。**所以，**用JS对象模拟DOM节点的好处是，页面的更新可以先全部反映在JS对象(虚拟DOM)上，操作内存中的JS对象的速度显然要更快，等更新完成后，再将最终的JS对象映射成真实的DOM，交由浏览器去绘制。

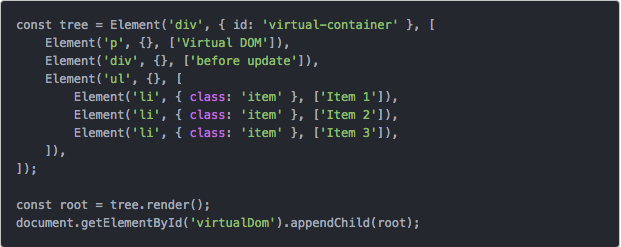
**四、实现虚拟DOM**

        例如一个真实的DOM节点。



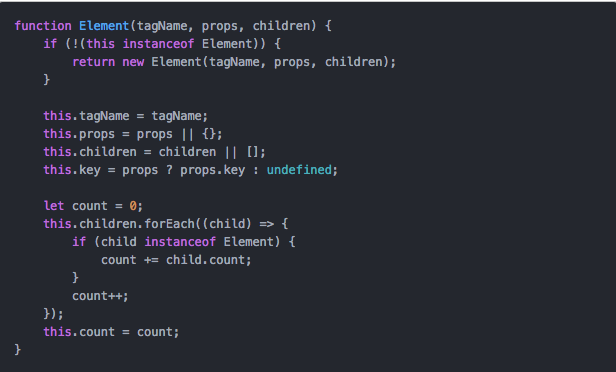
真实DOM

        我们用JS来模拟DOM节点实现虚拟DOM。



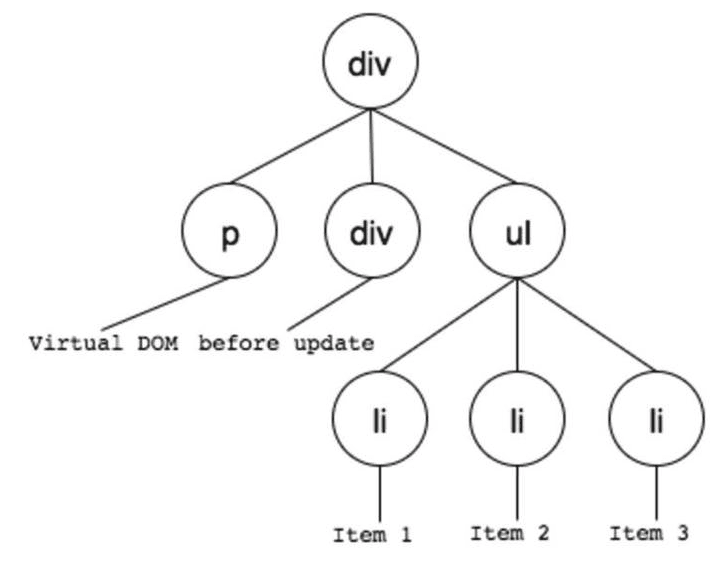
虚拟DOM

        其中的Element方法具体怎么实现的呢？



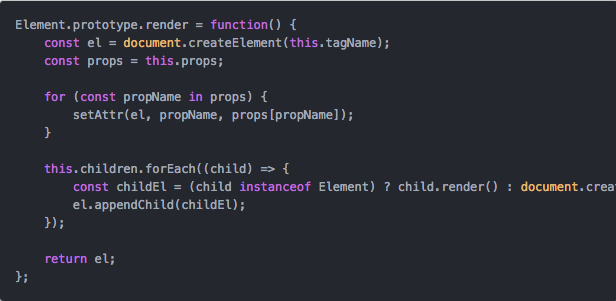
Element方法实现

        第一个参数是节点名（如div），第二个参数是节点的属性（如class），第三个参数是子节点（如ul的li）。除了这三个参数会被保存在对象上外，还保存了**key和count**。其相当于形成了虚拟DOM树。



虚拟DOM树

        有了JS对象后，最终还需要将其映射成真实DOM



虚拟DOM对象映射成真实DOM

        我们已经完成了创建虚拟DOM并将其映射成真实DOM，这样所有的更新都可以先反应到虚拟DOM上，如何反应？需要用到**Diff算法**。

        两棵树如果完全比较时间复杂度是O(n^3)，但参照《深入浅出React和Redux》一书中的介绍，React的Diff算法的时间复杂度是O(n)。要实现这么低的时间复杂度，意味着只能平层的比较两棵树的节点，放弃了深度遍历。这样做，似乎牺牲掉了一定的精确性来换取速度，但考虑到现实中前端页面通常也不会跨层移动DOM元素，这样做是最优的。

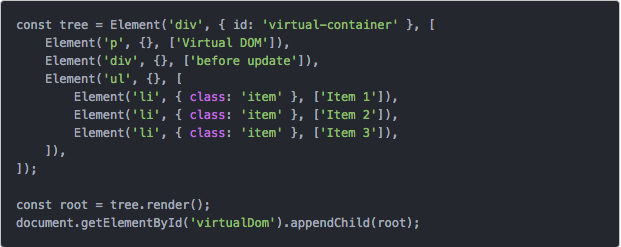
**深度优先遍历，记录差异**

        。。。。

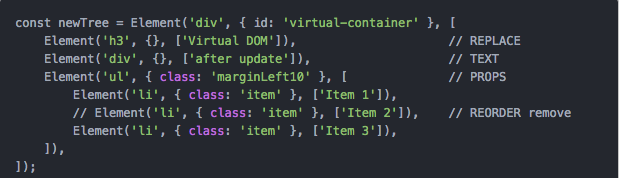
**Diff操作**

        在实际代码中，会对新旧两棵树进行一个深度的遍历，每个节点都会有一个标记。每遍历到一个节点就把该节点和新的树进行对比，如果有差异就记录到一个对象中。

        下面我们创建一棵新树，用于和之前的树进行比较，来看看Diff算法是怎么操作的。



old Tree



new Tree

        平层Diff，只有以下4种情况：

        1、**节点类型变了**，例如下图中的P变成了H3。我们将这个过程称之为**REPLACE**。直接将旧节点卸载并装载新节点。旧节点包括下面的子节点都将被卸载，如果新节点和旧节点仅仅是类型不同，但下面的所有子节点都一样时，这样做效率不高。但为了避免O(n^3)的时间复杂度，这样是值得的。这也提醒了开发者，应该避免无谓的节点类型的变化，例如运行时将div变成p没有意义。

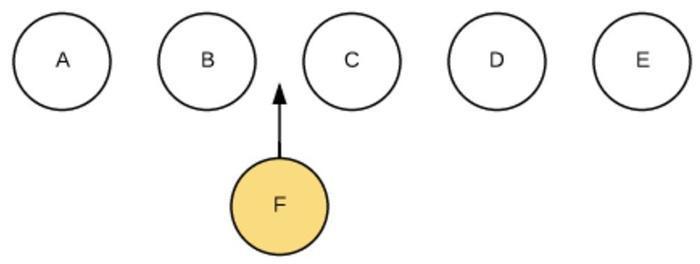
        2、**节点类型一样，仅仅属性或属性值变了。**我们将这个过程称之为**PROPS**。此时不会触发节点卸载和装载，而是节点更新。



查找不同属性方法

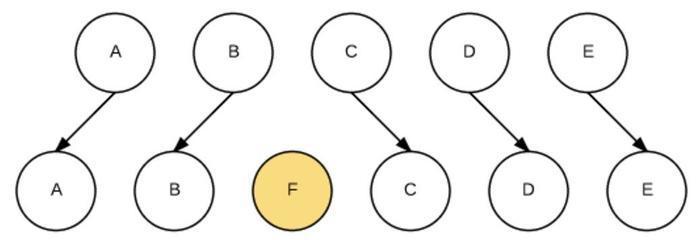
        3、**文本变了**，文本对也是一个Text Node，也比较简单，直接修改文字内容就行了，我们将这个过程称之为**TEXT**。

        4、移动／增加／删除 子节点，我们将这个过程称之为**REORDER**。看一个例子，在A、B、C、D、E五个节点的B和C中的BC两个节点中间加入一个F节点。



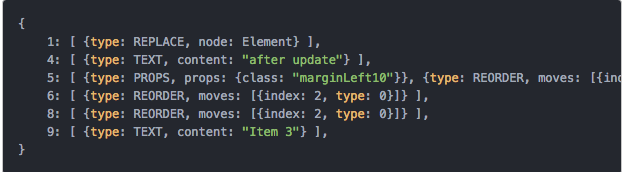
例子

        我们**简单粗暴的做法**是遍历每一个新虚拟DOM的节点，与旧虚拟DOM对比相应节点对比，在旧DOM中是否存在，不同就卸载原来的按上新的。这样会对F后边每一个节点进行操作。卸载C，装载F，卸载D，装载C，卸载E，装载D，装载E。**效率太低。**



粗暴做法

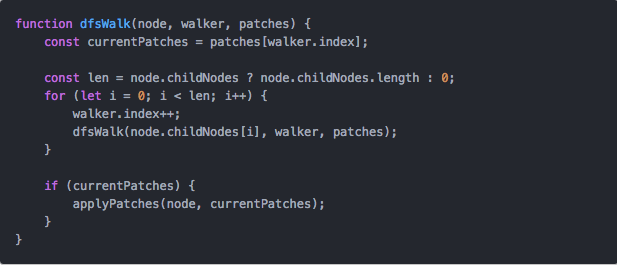
        如果我们在JSX里为数组或枚举型元素增加上key后，它能够根据key，直接找到具体位置进行操作，效率比较高。常见的**最小编辑距离问题**，可以用Levenshtein Distance算法来实现，时间复杂度是O(M\*N)，但通常我们只要一些简单的移动就能满足需要，降低精确性，将时间复杂度降低到O(max(M,N))即可。



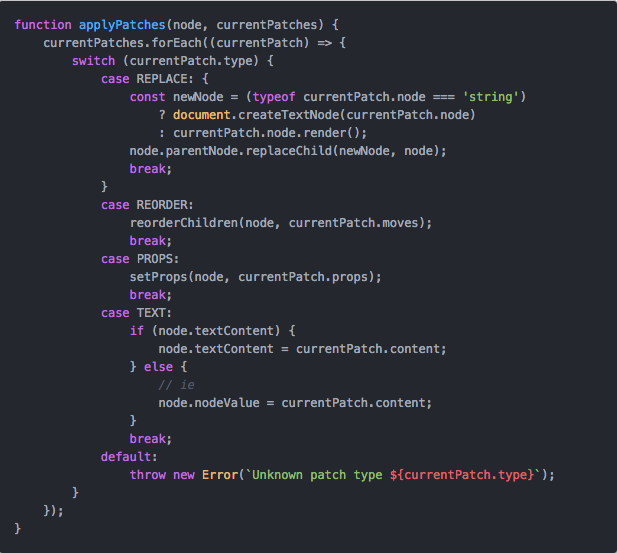
最终Diff出来的结果

**映射成真实DOM**

        虚拟DOM有了，Diff也有了，现在就可以将Diff应用到真实DOM上了。深度遍历DOM将Diff的内容更新进去。



根据Diff更新DOM



根据Diff更新DOM

我们会有两个虚拟DOM(js对象，new/old进行比较diff)，用户交互我们操作数据变化new虚拟DOM，old虚拟DOM会映射成**实际DOM(**js对象生成的DOM文档)通过**DOM fragment**操作给浏览器渲染。当修改new虚拟DOM，会把newDOM和oldDOM通过diff算法比较，得出diff结果数据表(用4种变换情况表示)。再把diff结果表通过**DOM** **fragment**更新到**浏览器DOM**中。

虚拟DOM的存在的意义？vdom 的真正意义是为了实现跨平台，服务端渲染，以及提供一个性能还算不错 Dom 更新策略。vdom 让整个 mvvm 框架灵活了起来

Diff算法只是为了虚拟DOM比较替换效率更高，通过Diff算法得到diff算法结果数据表(需要进行哪些操作记录表)。原本要操作的DOM在vue这边还是要操作的，只不过用到了js的**DOM** **fragment**来操作dom（统一计算出所有变化后统一更新一次DOM）进行浏览器DOM一次性更新。其实**DOM** **fragment**我们不用平时发开也能用，但是这样程序员写业务代码就用把DOM操作放到fragment里，这就是框架的价值，程序员才能专注于写业务代码**。**