浏览器回流与重绘 与虚拟DOM的优势

Webkit 浏览器渲染过程

- 创建DOM Tree

> 用HTML分析器，分析HTML元素，构建一颗DOM树，HTML中的每个tag都是DOM树中的1个节点，根节点就是我们常用的document对象。DOM树里包含了所有HTML标签，包括display:none隐藏，还有用JS动态添加的元素等

- 创建Style Rules

> 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，在解析的过程中会去掉浏览器不能识别的样式，比如IE会去掉-moz开头的样式，而FF会去掉_开头的样式

- 构建Render Tree

> 将上面的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。这些render对象最终会被构建成一颗Render Tree。Render Tree 不包含隐藏的节点(比如display:none的节点，还有head节点)，因为这些节点不会用于呈现，而且不会影响呈现的，所以就不会包含到Render Tree中。注意visibility:hidden隐藏的元素还是会包含到Render Tree中的，因为visibility:hidden 会影响布局(layout)，会占有空间。根据CSS2的标准，Render Tree中的每个节点都称为Box (Box dimensions)，理解页面元素为一个具有填充、边距、边框和位置的盒子

- 布局Layout

> 有了Render Tree后，浏览器开始布局，会为每个Render Tree上的节点确定一个在显示屏上出现的精确坐标值

- 绘制Painting

> Render Tree有了，节点显示的位置坐标也有了，最后就是调用每个节点的paint方法，让它们显示出来

> 用传统的源生api或jQuery去操作DOM时，浏览器会从构建DOM树开始到绘制Painting从头到尾执行一遍流程。比如当你在一次操作时，需要更新10个DOM节点，理想状态是一次性构建完DOM树，再执行后续操作。但浏览器没这么智能，收到第一个更新DOM请求后，并不知道后续还有9次更新操作，因此会马上执行流程，最终执行10次流程。

回流（重排）与重绘

> 当Render Tree中的一部分(或全部)因为元素的规模尺寸，布局，隐藏等改变而需要重新构建。这就称为回流(reflow)。每个页面至少需要一次回流，就是在页面第一次加载的时候。在回流的时候，浏览器会使渲染树中受到影响的部分失效，并重新构造这部分渲染树，完成回流后，浏览器会重新绘制受影响的部分到屏幕中，该过程成为重绘。一般重绘是由于Render Tree中的一些元素需要更新属性，而这些属性只是影响元素的外观，风格,并没有影响页面布局，如background-color。回流必将引起重绘，而重绘不一定会引起回流。

回流何时发生

当页面布局和几何属性改变时就需要回流。下述情况会发生浏览器回流：

1.添加或者删除可见的DOM元素

2.元素位置改变

3.元素尺寸改变——边距、填充、边框、宽度和高度

4.内容改变——比如文本改变或者图片大小改变而引起的计算值宽度和高度改变

5.页面渲染初始化

6.浏览器窗口尺寸改变——resize事件发生时

```
var s = document.body.style;
s.padding = "2px"; // 回流+重绘
s.border = "1px solid red"; // 再一次 回流+重绘
s.color = "blue"; // 再一次重绘
s.backgroundColor = "#ccc"; // 再一次 重绘
s.fontSize = "14px"; // 再一次 回流+重绘
// 添加node，再一次 回流+重绘
document.body.appendChild(document.createTextNode('abc!'));
```

> 从上个实例代码中可以看到几行简单的JS代码就引起了6次左右的回流、重绘。而且我们也知道回流的花销也不小，如果每句JS操作都去回流重绘的话，浏览器可能就会受不了。所以很多浏览器都会优化这些操作，浏览器会维护1个队列，把所有会引起回流、重绘的操作放入这个队列，等队列中的操作到了一定的数量或者到了一定的时间间隔，浏览器就会flush队列，进行一个批处理。这样就会让多次的回流、重绘变成一次回流重绘。

> 同理，用传统的源生api或jQuery去操作DOM时，需要更新10个DOM节点时，浏览器会进行10次左右的回流,也就是说进行10次左右的DOM树构建和Render Tree构建，通俗来讲会将创建DOM Tree到绘制Painting这五个步骤进行10次，尽管浏览器具有队列批量处理的优化操作，但依旧影响性能

虚拟DOM

> 虚拟DOM就是为了解决这个浏览器性能问题而被设计出来的。例如前面的例子，假如一次操作中有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内容保存到本地的一个js对象中，最终将这个js对象一次性attach到DOM树上，通知浏览器去执行绘制工作（五个步骤只进行一次），这样可以避免大量的无谓的计算量。真实框架(vue react)会将diff结果放到DOM fragment 对象里一次性更新的。这就是框架的价值，程序员写业务代码不会都将DOM操作放到fragment 对象里，所以框架就设计出了虚拟DOM，自动完成这些工作。程序员才能专注于写业务代码

一般动态创建html元素都是创建好了直接appendChild()上去，但是如果要添加大量的元素还用这个方法的话就会导致大量的重绘以及回流，所以需要一个'缓存区'来保存创建的节点，然后再一次性添加到父节点中。这时候DocumentFragment对象就派上用场了。

w3c的官方说明：

DocumentFragment 节点不属于文档树，继承的 parentNode 属性总是 null。 不过它有一种特殊的行为，该行为使得它非常有用，即当请求把一个 DocumentFragment 节点插入文档树时，插入的不是 DocumentFragment 自身，而是它的所有子孙节点。这使得 DocumentFragment 成了有用的占位符，暂时存放那些一次插入文档的节点。它还有利于实现文档的剪切、复制和粘贴操作。

重点就在于DocumentFragment 节点不属于文档树。因此当把创建的节点添加到该对象时，并不会导致页面的回流，因此性能就自然上去了。

var fragment = document.createDocumentFragment();

你的知道浏览器的虚拟DOM与真实DOM的区别（注意：需不需要虚拟DOM，其实与框架的DOM操作机制有关）：

1.虚拟DOM不会进行排版与重绘操作

2.虚拟DOM进行频繁修改，然后一次性比较并修改真实DOM中需要改的部分（注意！），最后并在真实DOM中进行排版与重绘，减少过多DOM节点排版与重绘损耗

3.真实DOM频繁排版与重绘的效率是相当低的

4.虚拟DOM有效降低大面积（真实DOM节点）的重绘与排版，因为最终与真实DOM比较差异，可以只渲染局部（同2）

使用虚拟DOM的损耗计算：

总损耗 = 虚拟DOM增删改 +（与Diff算法效率有关）真实DOM差异增删改 +（较少的节点）排版与重绘 直接使用真实DOM的损耗计算：

总损耗 = 真实DOM完全增删改 +（可能较多的节点）排版与重绘 总之，一切为了减弱频繁的大面积重绘引发的性能问题，不同框架不一定需要虚拟DOM，关键看框架是否频繁会引发大面积的DOM操作