究竟为什么要学习 Vue——从性能角度说明

之前给大家说明了 MVC 模式以及变种模式 MVVM，相比大家对框架模式有了一定的了解。接下来我们来探究一下具有 MVVM 框架模式的 Vue。想必每个同学都应该知道到 Vue，哪怕就算你没有用过，最起码也是听过。在外边都说，**Vue 是一个很厉害的框架，很好用，各个大厂都在使用它，面试简历要求都要你会使用，**以上的这些原因真的是我们要学习 Vue 的原因么？**并不是**，我们要看到这些现象后面真正存在的本质。**为什么他很厉害，问什么它被很多人认可，为什么需要它等等这样的问题，这个才应该是我们关注的，**而非老大让用 Vue 就用，别人说他好你就学。

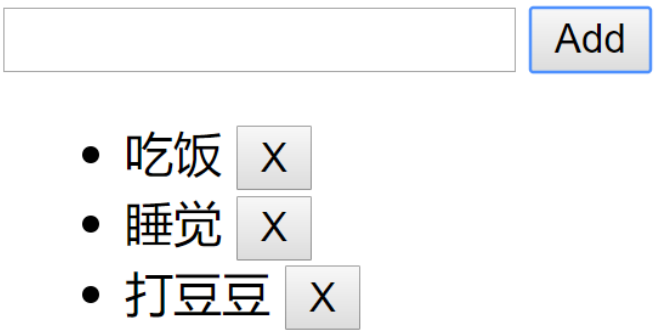
对于 Vue 来说，用过他的同学，发现与我们之前的思维逻辑不一样了。不知道的同学也没关系。我们从头开始。感受一下语言发展的历程。

为什么需要 Vue?——性能角度考虑

一个新的事物的产生必定是存在原因的，我们接下来探究他的原因。真正的原因来源——**来自代码方式。**代码方式怎么了并没有什么问题啊？以现在大家的编码量来说确实没有什么问题，包括之前 MVC 模式的提出，有些同学觉得可能会更难了，只能说在当前的这样的代码量上，没有什么大的问题，当项目，系统足够大的时候，我们的代码量上来的时候，问题就来了。接下来我们通过一个小的组件来以小见大，感受一下我们会出现的问题。

对于任何一个页面来说：（淘宝首屏了解一下）

包括在大家在写淘宝的时候，会把这样的页面会分成若干块，依次去完成每一个块。多个块的完成之后，整个页面也就出来。
接下来我们单纯拿一个页面中的一个小块来看我们的代码会发生什么样的问题以及该如何解决。
我们做一个 todoList



原生 JavaScript 代码如下

```
<div class="wrapper">
        <input type="text" id="inp">
        <button id="btn">Add</button>
        <ul id="list">

        </ul>
    </div>

    <script>
        var inp = document.getElementById('inp');
        var addBtn = document.getElementById('btn');
        var list = document.getElementById('list');
            addBtn.addEventListener('click', function () {
                var value = inp.value;
                var li = document.createElement('li');
                var delBtn = document.createElement('button');
                delBtn.addEventListener('click', function () {
                    list.removeChild(li);
                }, false);
                delBtn.innerHTML = 'X';
                li.innerHTML = value;
                li.appendChild(delBtn);
```

```
            list.appendChild(li);
            inp.value = '';
        }, false);


</script>
```

功能都能做到，但是我们联想一下，我们单纯只是实现了一个页面中的一个小块，基本上大的项目，一定有上百个甚至上千个这样的块。来实现这样的项目。

Js 的代码就上来了，就会暴露一些问题：

1.存在复杂的 Dom 操作

2.复杂的特效和动画的实现

3.请求网络数据存在跨域问题

最大的问题在于 API 沉重（浏览器大战，历史遗留性问题）。

我们希望有一个工具可以把这样的问题很好的解决，在这样的一个时期借助封装的思想出来了一个工具——Jquery。

还是上面的功能，用 Jquery 来实现：

```
<div class="wrapper">
        <input type="text" id="inp">
        <button id="btn">Add</button>
        <ul id="list">

        </ul>
</div>
<script>
    var inp = $('#inp');
    var addBtn = $("#btn");
    var list = $('#list');
    addBtn.on('click', function () {
        var value = inp.val();
        var li = $('<li/>').html(value).appendTo(list);
        $("<button/>").on('click', function () {
            li.remove();
        }).html('X').appendTo(li)
        inp.val('');
    })
```

</script>

相对比之前的原生的 JavaScript 来说，代码简洁，高级思想链式调用。

Jquery 本着最少的代码做最多的事情，这是 jQuery 的口号，而且名副其实。使用它的高级 selector，开发者只需编写几行代码就能实现令人惊奇的效果。开发者无需过于担忧浏览器差异，它除了还完全支持 Ajax，而且拥有许多提高开发者编程效率的其它抽象概念。jQuery 把 JavaScript 带到了一个更高的层次。以下是一个非常简单的示例：

```
$("p.item").addClass("item-list").show("slow");
```

通过以上简短的代码，开发者可以遍历“item”类中所有的<p>元素，然后向其增加“item-list”类，同时以动画效果缓缓显示每一个段落。开发者无需检查客户端浏览器类型，无需编写循环代码，无需编写复杂的动画函数，仅仅通过一行代码就能实现上述效果

。

虽说 Jquery 有诸多优点：**但是大家仍要切记原生 JavaScript 是重中之重，任何时间节点都要学好 JavaScript。**

我们通过 Jquery 确实可以从代码量上减少，而且用着更舒服。但是，无论 Jquery 还是原生的 JavaScript，我们对页面的处理逻辑一直是:

Data ➔ DOM 操作 ➔ Page

这个逻辑怎么了，又出现了什么问题？

这个流程能出现问题？

唯一能出现问题的只能是 DOM 操作了。

DOM 操作会有什么样的问题呢？

DOM 操作影响页面性能的核心问题

通过 js 操作 DOM 的代价很高，影响页面性能的主要问题有如下几点：

访问和修改 DOM 元素

修改 DOM 元素的样式，导致重绘或重排

通过对 DOM 元素的事件处理，完成与用户的交互功能

DOM 的修改会导致重绘和重排。

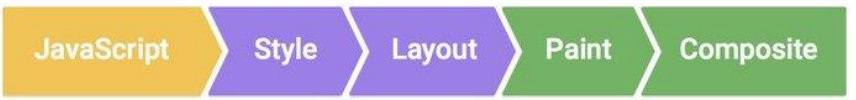
重绘是指一些样式的修改，元素的位置和大小都没有改变；

重排是指元素的位置或尺寸发生了变化，浏览器需要重新计算渲染树，而新的渲染树建立后，浏览器会重新绘制受影响的元素。

页面重绘的速度要比页面重排的速度快，在页面交互中要尽量避免页面的重排操作。浏览器不会在 js 执行的时候更新 DOM，而是会把这些 DOM 操

作存放在一个队列中，在 js 执行完之后按顺序一次性执行完毕，因此在 js 执行过程中用户一直在被阻塞。

1.页面渲染过程
一个页面更新时，渲染过程大致如下：



JavaScript: 通过 js 来制作动画效果或操作 DOM 实现交互效果
Style: 计算样式，如果元素的样式有改变，在这一步重新计算样式，并匹配到对应的 DOM 上
Layout: 根据上一步的 DOM 样式规则，重新进行布局（重排）
Paint: 在多个渲染层上，对新的布局重新绘制（重绘）
Composite: 将绘制好的多个渲染层合并，显示到屏幕上
在网页生成的时候，至少会进行一次布局和渲染，在后面用户的操作时，不断的进行重绘或重排，因此如果在 js 中存在很多 DOM 操作，就会不断地出发重绘或重排，影响页面性能。

2.DOM 操作对页面性能的影响
如前面所说，DOM 操作影响页面性能的核心问题主要在于 DOM 操作导致了页面的重绘或重排，为了减少由于重绘和重排对网页性能的影响，我们要知道都有哪些操作会导致页面的重绘或者重排。

2.1 导致页面重排的一些操作：
内容改变
文本改变或图片尺寸改变
DOM 元素的几何属性的变化
例如改变 DOM 元素的宽高值时，原渲染树中的相关节点会失效，浏览器会根据变化后的 DOM 重新排建渲染树中的相关节点。如果父节点的几何属性变化时，还会使其子节点及后续兄弟节点重新计算位置等，造成一系列的重排。
DOM 树的结构变化
添加 DOM 节点、修改 DOM 节点位置及删除某个节点都是对 DOM 树的更改，会造成页面的重排。浏览器布局是从上到下的过程，修改当前元素不会对其前边已经遍历过的元素造成影响，但是如果在所有的节点前添加一个新的元素，则后续的所有元素都要进行重排。

获取某些属性

除了渲染树的直接变化，当获取一些属性值时，浏览器为取得正确的值也会发生重排，这些属性包括：offsetTop、offsetLeft、 offsetWidth、offsetHeight、scrollTop、 scrollLeft、 scrollWidth、 scrollHeight、 clientTop、 clientLeft、clientWidth、clientHeight、getComputedStyle()。

浏览器窗口尺寸改变

窗口尺寸的改变会影响整个网页内元素的尺寸的改变，即 DOM 元素的集合属性变化，因此会造成重排。

2.2 导致页面重绘的操作

应用新的样式或者修改任何影响元素外观的属性

只改变了元素的样式，并未改变元素大小、位置，此时只涉及到重绘操作。

重排一定会导致重绘

一个元素的重排一定会影响到渲染树的变化，因此也一定会涉及到页面的重绘。

那既然 DOM 操作会带来很严重的性能问题，我们能不能创造一个性能又高，代码亲和性友好的工具呢！

真正的核心在于：

Data ➔ Page

在这里并没有 DOM 操作，那意味着是不是不要 DOM 操作了呢。不是这样，页面的构成和交互都是需要 DOM 操作的，在我们能避免的前提上希望利用更好的算法去优化我们操作 DOM 的过程使得性能最优。每次去想一个很高级的算法去处理逻辑，学习成本就会变得很大。我们想有一个工具或者框架能实现

在代码简洁，不增加过多的学习成本的前提下，我们需要一个**数据到页面（视图）的直接映射并且性能最优** 的工具

那么他就是我们说的 **Vue**

同样还是上面的代码我们利用 Vue 来实现一下：

```
<div id="app">
        <input type="text" v-model="val">
        <button @click="list.push(val);val='';">Add</button>
        <ul>
            <li v-for="(item, index) in list">{{item}} <button @click="list.splice(index,1)">X</button></li>
        </ul>
```

```
</div>
<script>
    new Vue({
        el: '#app',
        data: {
            val: '',
            list: []
        }
    })
</script>
```

这里的代码可能有些同学看不懂，没关系。大概让大家去做一些对比，先在视觉上感受一下 Vue 的魅力。在这里很显然 js 的代码少了很多，在 html 当中多了了不少的我们不认识的属性，这些东西都是 Vue 特性。

至此大家要明白，我们需要 Vue 目的之一是在逻辑（省去了高效率的 DOM 操作）上减少我们的代码量，并且达到性能最优。这个是我们需要 Vue 的目的之一。