前言

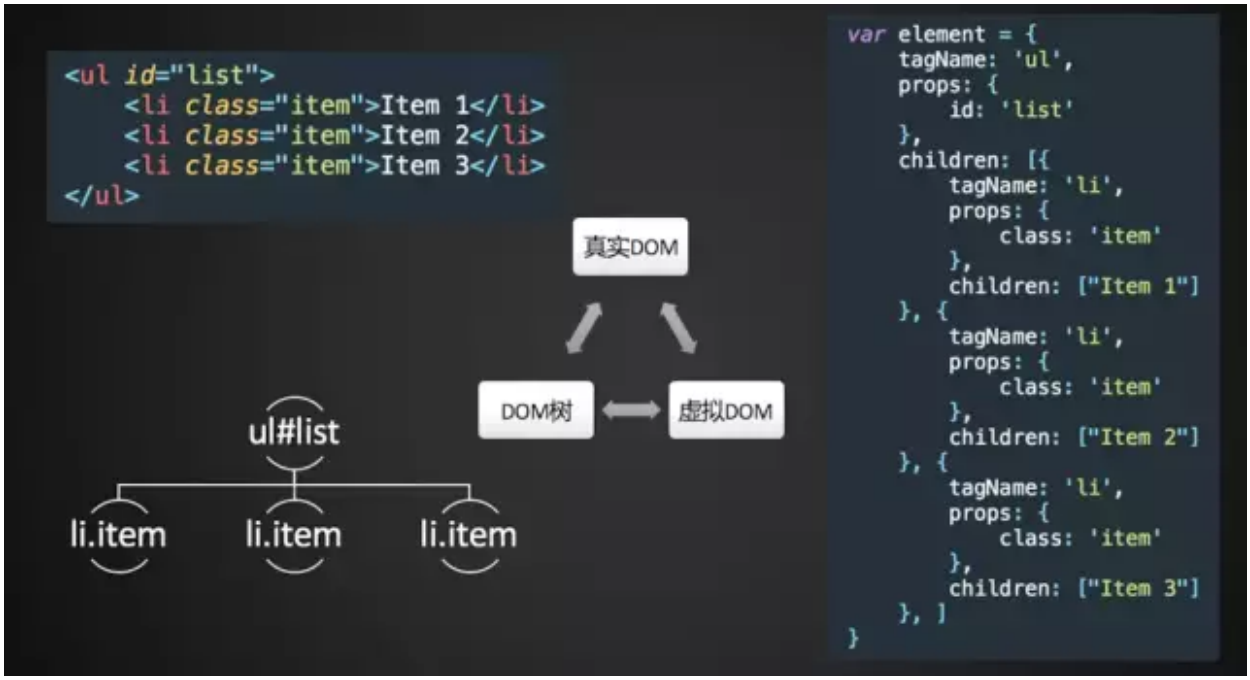
你有没有留意到？优秀的解决方案思想都是相通的：当你研究 Flutter 渲染原理时会发现 Flutter Rendering 层类似于 React 中的虚拟 DOM，当你去看 Weex 工作原理时，诶，又发现了虚拟 DOM 的身影，更别提 VUE 响应式视图的核心也是虚拟 DOM 了。

那这个虚拟 DOM 有什么用？为什么这么多框架都应用了它？本质上带来了什么优势？本文将结合前端和移动端来谈谈。

什么是 DOM？什么是虚拟 DOM?

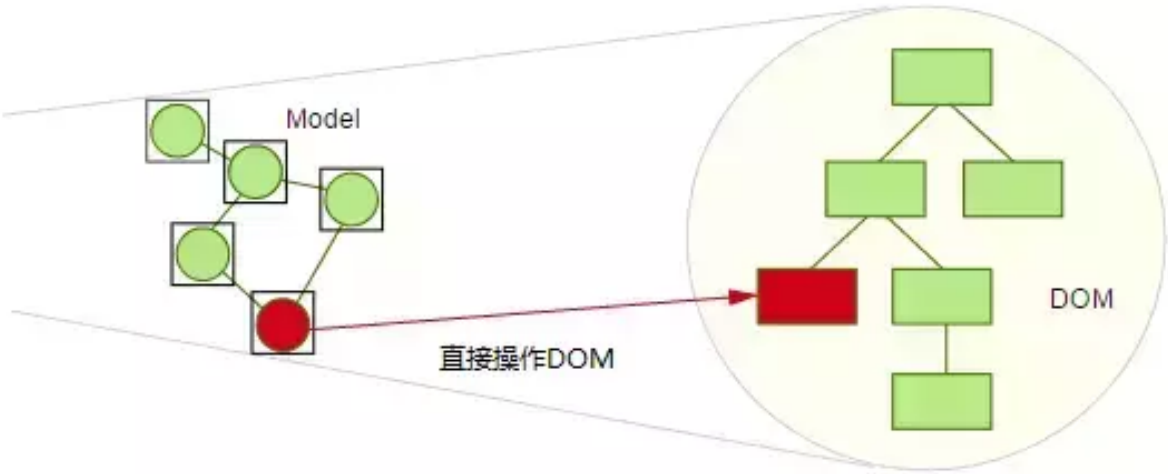
DOM 就是文档树，与用户界面控件树对应，在 web 开发中通常指 HTML 对应的渲染树，但广义的 DOM 也可以指 Android 中的 XML 布局对应的控件树，而 DOM 操作就是指直接操作渲染树（或控件树）。

虚拟 DOM 是一个用来表示真实的 DOM 结构的数据结构。



思考

想当年学前端的时候，还是 jQuery 的时代，想赋值？改个样式？取值？都是 document.getElementById()咔咔一顿操作。这样直接操作 DOM 会有什么问题？



直接操作 DOM 带来的问题

1. model 和 view 耦合

最直观的问题之一， 把用户请求的表现逻辑和控制层要实现的业务逻辑两者混合起来了，两部分的依赖非常强。

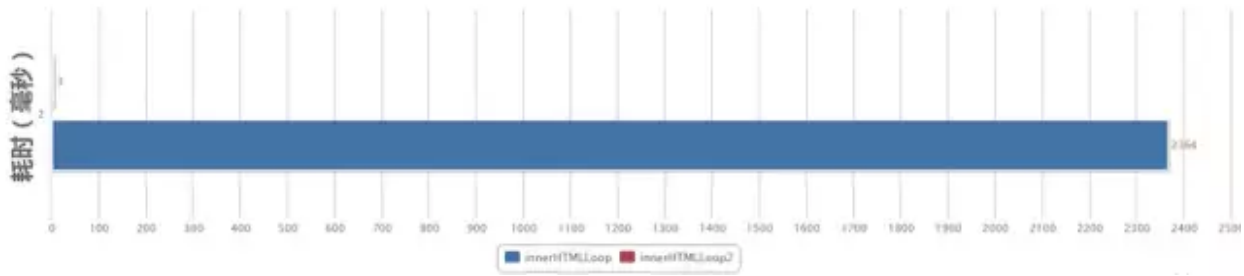
2. 高频操作引起性能损耗

写个简单Demo，我们看下效果。

```
<script language="JavaScript" type="text/javascript">
  function innerHTMLLoop() {
    for (var count = 0; count < 10000; count++) {
      document.getElementById('txt').innerHTML += "dom";
    }
  }

  function innerHTMLLoop2() {
    var content = '';
    for (var count = 0; count < 10000; count++) {
      content += 'dom';
    }
    document.getElementById('txt').innerHTML += content;
  }

  var start = new Date().getTime();
  innerHTMLLoop();
  var end = new Date().getTime();
  document.getElementById('txt').innerHTML += '运行时间：' + (end - start);
</script>
```



为什么会有性能损耗?

原因可以归结为 2 点:

1. 跨界交流损耗

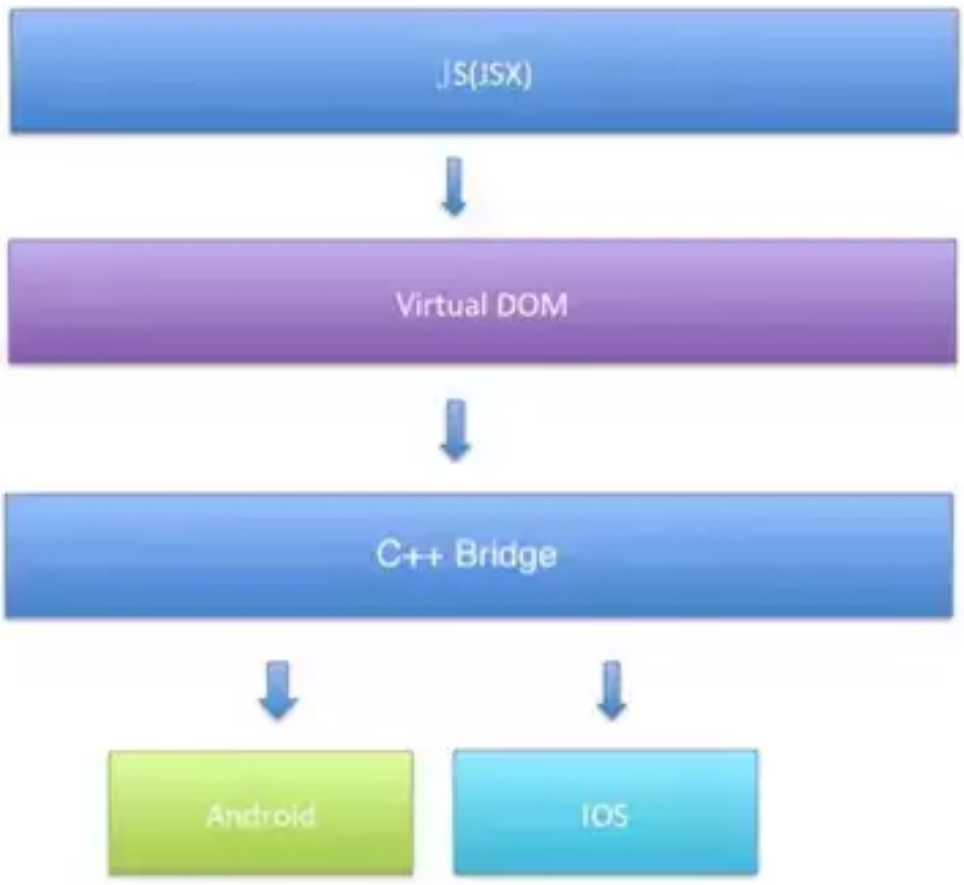
```
1 把 DOM 和 ECMAScript 各自想象成一个岛屿，它们之间用收费桥梁连接。
2                                  —《高性能JavaScript》
```

DOM 属于渲染引擎，而 JS 又是属于 JS 引擎，在浏览器内核中他们彼此独立。单独来看，两者都是很快的，但当我们用 JS 去操作 DOM 时，引擎之间进行了“跨界交流”。这个“跨界交流”的实现并不简单，它依赖了桥接接口作为“桥梁”，如下图:

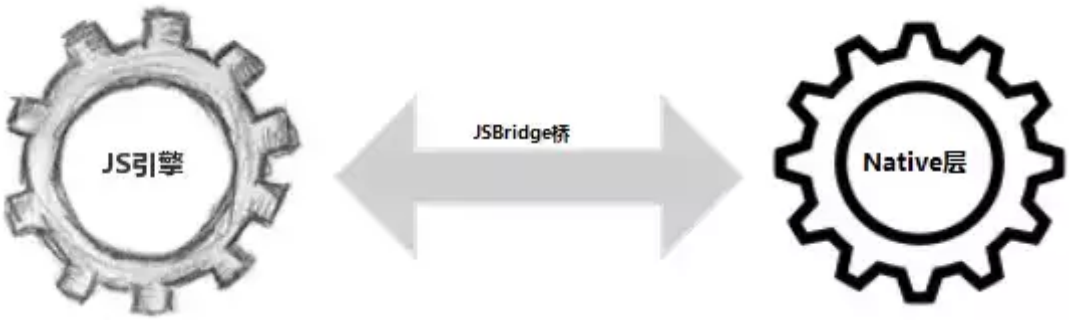
既然是收费桥梁，过“桥”就要收费。我们每操作一次 DOM（不管是为了修改还是仅仅为了访问其值），都要过一次“桥”。次数一多就会产生比较明显的性能问题。

那移动端混合开发的情况呢?

就拿 RectNative 举例，RectNative 是一套 UI 基于原生控件（非Web UI）业务逻辑基于 JS 的跨平台技术解决方案，JS 中所写控件标签不是真实控件，会在 Native 端解析为原生控件，如标签对应 Android 中的 TextView 控件。



在布局过程中 RN 需要在 JS 和 Native 之间通信，如果遇到滑动和拖动的情况，劣势就很明显了，这和在浏览器中要 JS 频繁操作 DOM 所带来的问题是相同的，都会带来比较可观的性能开销。



2. DOM 修改引起重绘或重排

修改 DOM 属性的代价更是昂贵，它会导致渲染引擎重新计算几何变化（重排和重绘）。我们来看下渲染步骤：

在页面生成时，至少会进行一次布局和渲染，后面用户操作时，如果修改了 DOM 节点，会触发渲染树（Render Tree）的变化，从而进行上图的步骤2、3、4、5，因此如果在 js 中存在很多 DOM 操作，就会不断地触发重绘或重排，影响页面性能。

在移动端，情况也好不到哪里去。

布局中的任何一个 View 一旦发生属性变化，都可能引起很大的连锁反应（如果所在的控件层级非常复杂的话）。例如某个 btn 的大小突然增加一倍，有可能会导致兄弟视图的位置变化，也有可能导致父视图的大小发生改变。当大量的 layout() 操作被频繁调用执行时，会引起整个 View 频繁地重渲，最终导致丢帧或 UI 卡顿。

解决办法

针对以上的问题，我们一一提出解决方案：

1. 减少跨界过桥次数，合并操作

> ECMAScript 每次访问 DOM，都要经过这座桥，并交纳“过桥费”，访问 DOM 的次数越多，费用也就越高。因此，推荐的做法是尽量减少过桥的次数，努力呆在 ECMAScript 岛上。——《高性能 JavaScript》

我们来分析下，怎么减少“过桥的次数”？过桥次数之所以频繁，和频繁的 DOM 操作有关。

比如我们给列表加数据，最差的方式就是这样：

```
for (var i = 0; i < N; i++) {
    var li = document.createElement("li");
           li.innerHTML = arr[i];
    ul.appendChild(li);
     }
```

这里会操作 N 次 DOM 触发 N 次重绘。重渲肯定是无法避免的，我们的目标是最小化重绘和重排次数。

那能不能不要立即去操作 DOM 呢?

将这 N 次更新的内容保存到一个 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。这样无论多么复杂的 DOM 操作，最终都只会触发一次渲染全流程，避免了大量的无谓计算量，这样不就可以了么！（欣喜若狂.jpg）

但优化 DOM 操作方式很多，不一定要依赖虚拟 DOM，所以这不是我们需要虚拟 DOM 的根本原因，根本的原因还是响应式需求。

2. 响应式

如果通过 JS 直接操作 DOM 的话，势必会造成视图数据和模型数据的不匹配，我们能不能让开发者只关心状态（数据）变化，而无需关心控件操作呢？当然可以！

React 中提出一个重要思想：状态改变则 UI 随之自动改变。

每次状态有变动就重构用户界面，重渲整个 view。如果没有虚拟 DOM，简单粗暴的做法就是直接重置 innerHTML，在大部分数据都变了的情况下，重置 innerHTML 还算合理，但如果只有一行数据变了，显然就有大量的浪费。

这是我们需要虚拟 DOM 的原因，用它来代替开发者的手工操作，确保只对真正有变化的部分进行实际的 DOM 操作（局部刷新）。

3. 总结

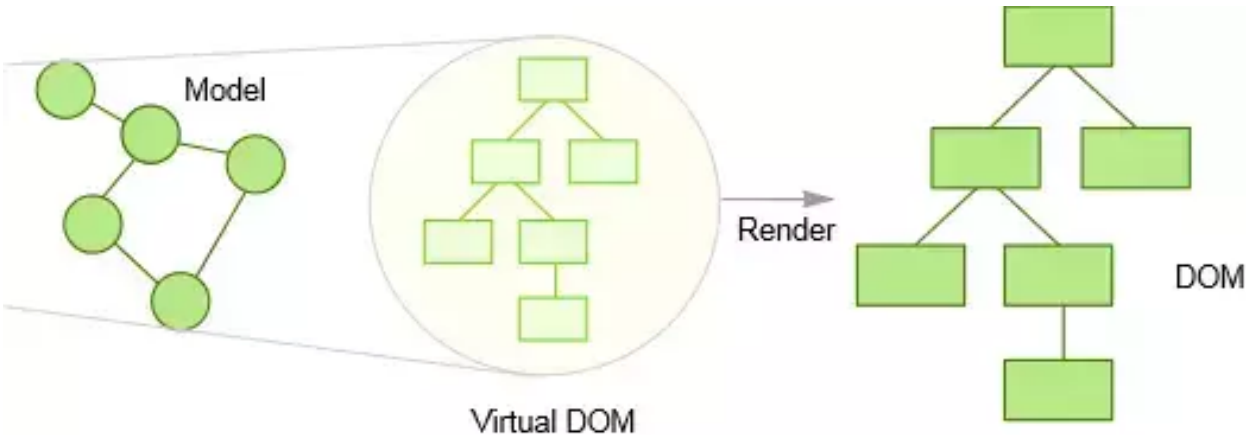
开发者对数据和状态所做的任何改动，都会被自动且高效的同步到虚拟 DOM（自动同步，体现响应式），最后再批量同步到真实 DOM 中，而不是每次改变都去操作一下 DOM（批量同步，体现合并操作）

1. 不需要直接操作控件，通过数据驱动视图
2. 最大程度降低对最终视图的修改，提高页面渲染效率

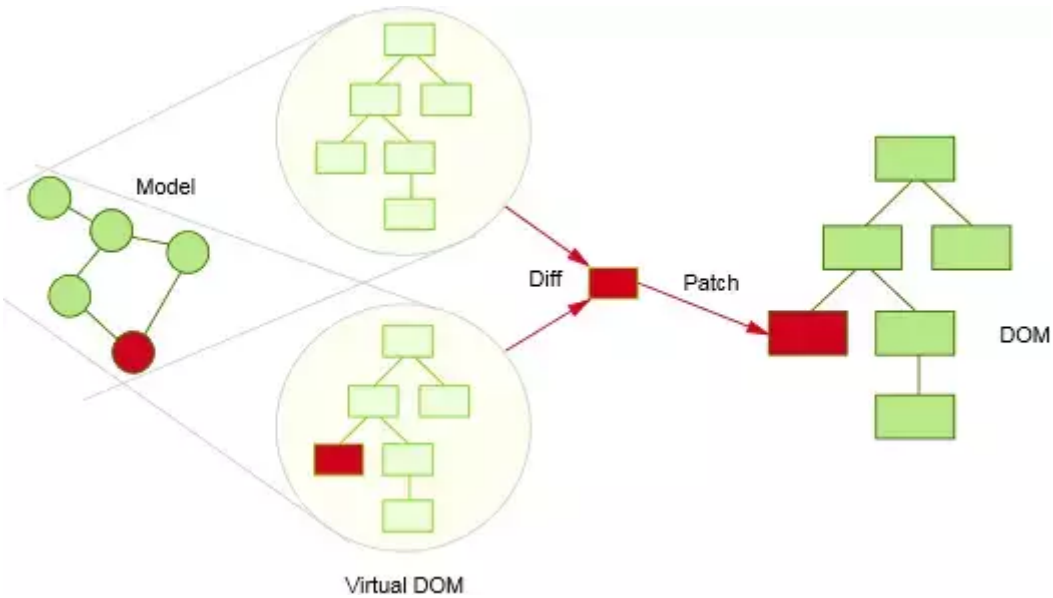
怎么利用虚拟 DOM?

1. React

当 React UI 渲染时，先渲染一个虚拟 DOM，这是一个轻量的纯 js 的对象结构，并没有完全实现 DOM，最主要的还是保留了节点之间的层次关系和一些基本属性，因为 DOM 实在是太复杂，实际在做最后绘制时，这些都是不需要关心的。所以虚拟 DOM 里每一个节点只有几个简单属性，哪怕是直接把虚拟 DOM 删了，根据新传进来的数据重新创建一个新的虚拟 DOM 都非常快。



当有变化时，生成一个新的虚拟 DOM。这个新的虚拟DOM反应了数据模型的新状态。现在我们有 2 个虚拟DOM：新的和老的。对比 DOM 树差异得到一个 Patch，把这个 Patch 打到真实的 DOM 上去，这有点像版本控制打patch的思路。



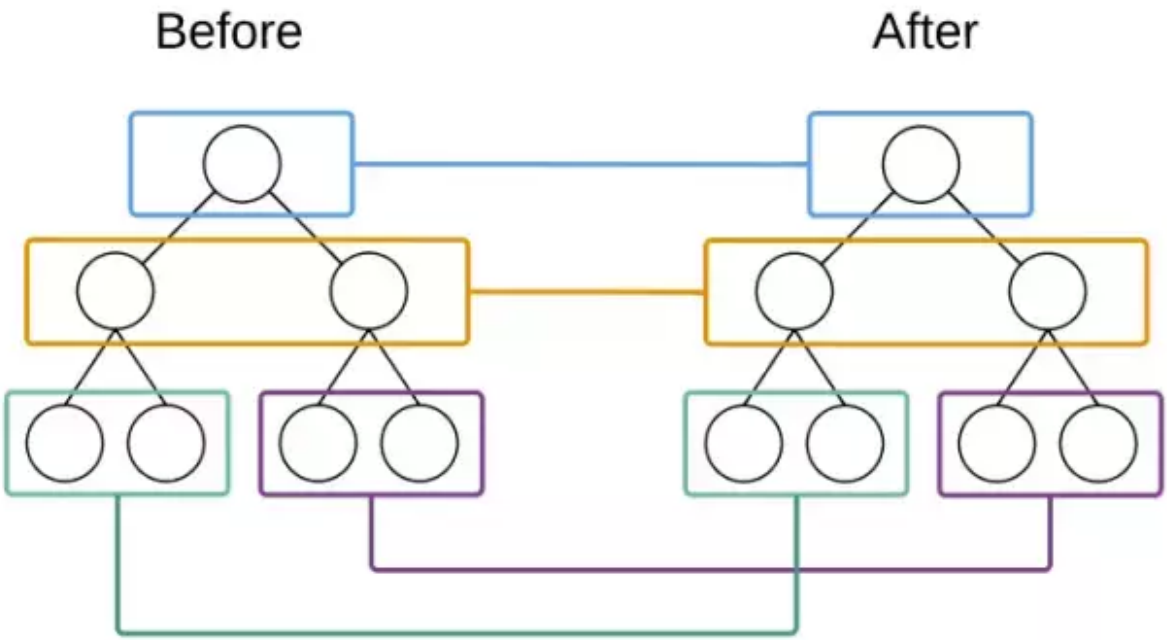
那我们怎么比较出两颗 DOM 树的差异呢？Diff 算法！

即给定任意两棵树，找到最少的转换步骤。但是标准的 Diff 算法复杂度需要 O(n^3)，这显然无法满足性能要求。Facebook 工程师结合 Web 界面的特点做出了两个简单的假设，使得 Diff 算法复杂度直接降低到 O(n)。

1. 两个相同组件产生类似的 DOM 结构，不同的组件产生不同的 DOM 结构；
2. 对于同一层次的一组子节点，它们可以通过唯一的 id 进行区分。

算法上的优化是 React 整个界面 Render 的基础，事实也证明这两个假设是合理而精确的，保证了整体界面构建的性能。

由这一对不同类型的节点的处理逻辑我们很容易得到推论，那就是 React 的 DOM Diff 算法实际上只会对树进行逐层比较，如下图：



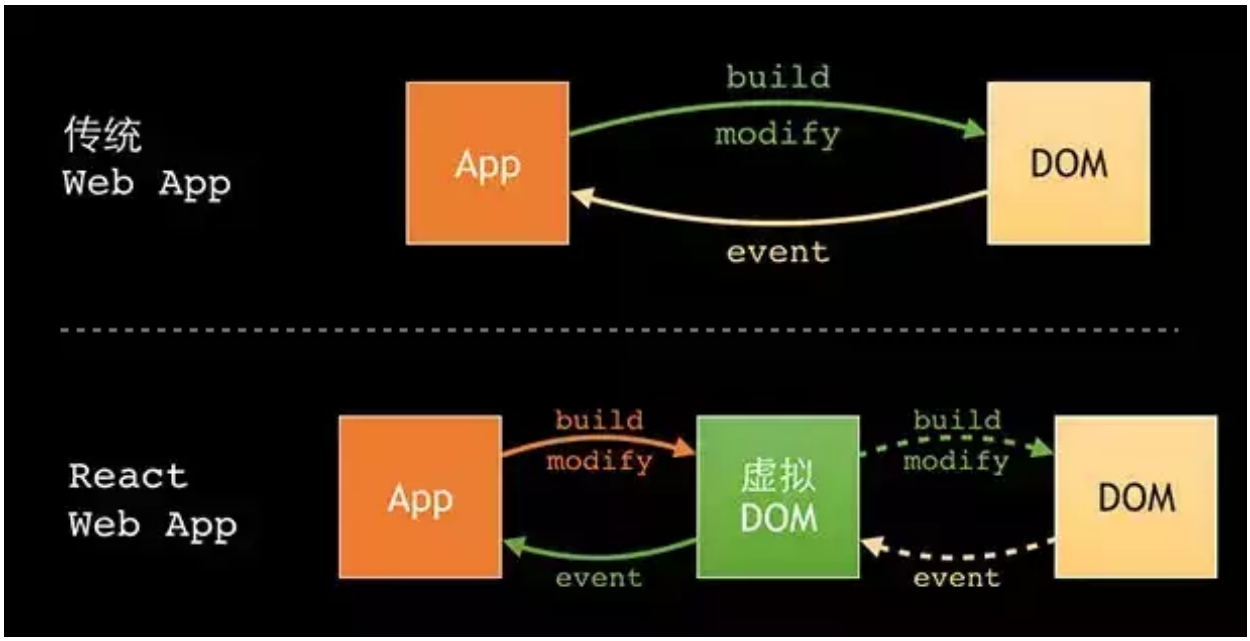
React 只会对相同颜色方框内的 DOM 节点进行比较，即同一个父节点下的所有子节点。当发现节点已经不存在，则该节点及其子节点会被完全删除掉，不会用于进一步的比较。这样只需要对树进行一次遍历，便能完成整个 DOM 树的比较。

实际实践起来，Diff 算法并没有这么简单，感兴趣的小伙伴可以在文末的推文去深入了解。

那跨平台方案的情况呢?

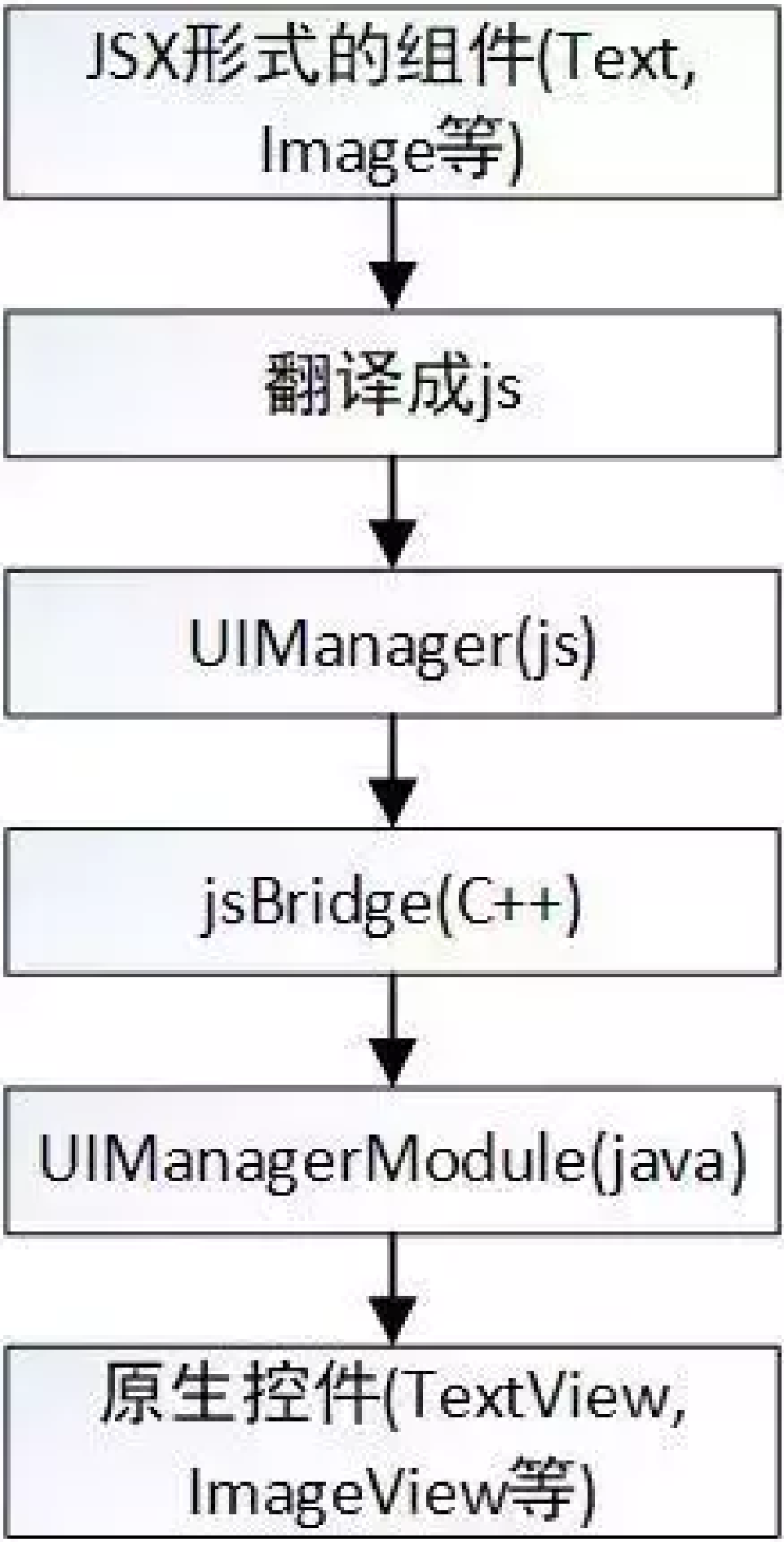
2. RN

上文已经提到 RN 是 React 在原生移动应用平台的衍生产物，那两者主要的区别是什么呢？主要的区别在于虚拟 DOM 映射的对象是什么。React 中虚拟 DOM 最终会映射为浏览器 DOM 树，而 RN 中虚拟 DOM 会通过 JavaScriptCore 映射为原生控件树。



步骤如下：

1. 布局消息传递：将虚拟 DOM 布局信息传递给原生；
2. 原生根据布局信息，映射成对应原生控件树，渲染控件树。

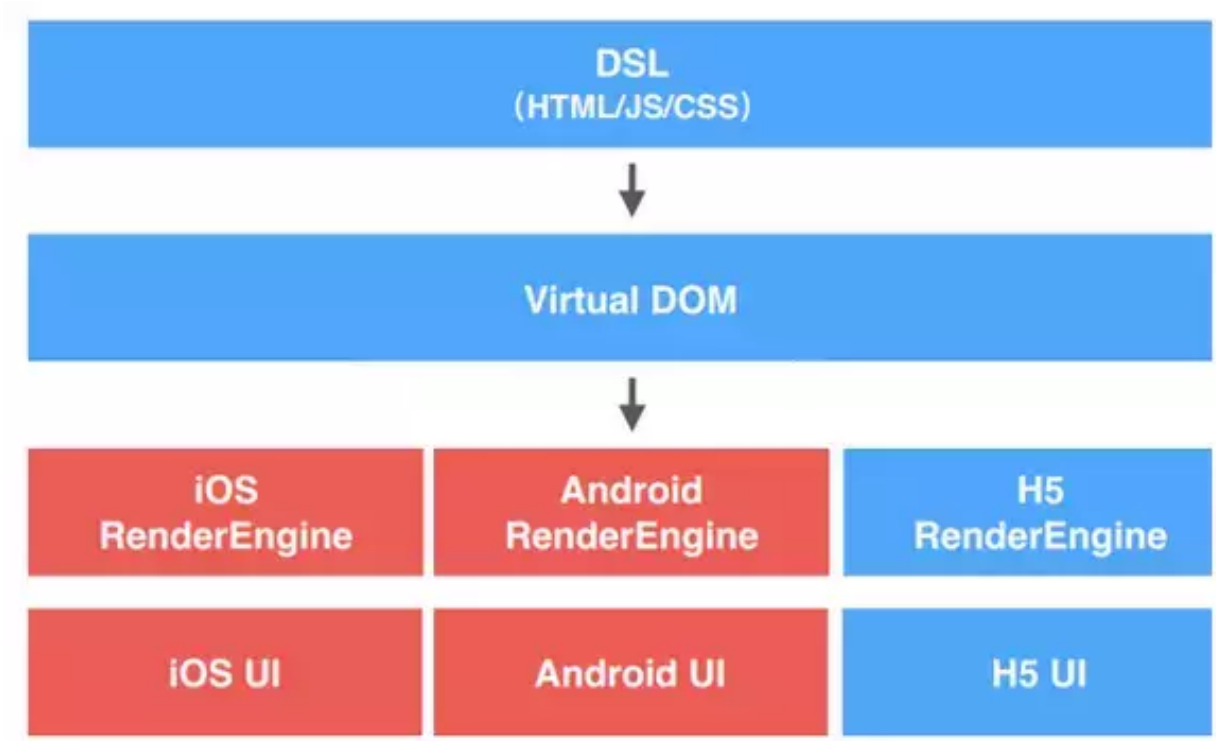


至此，RN 便实现了跨平台。

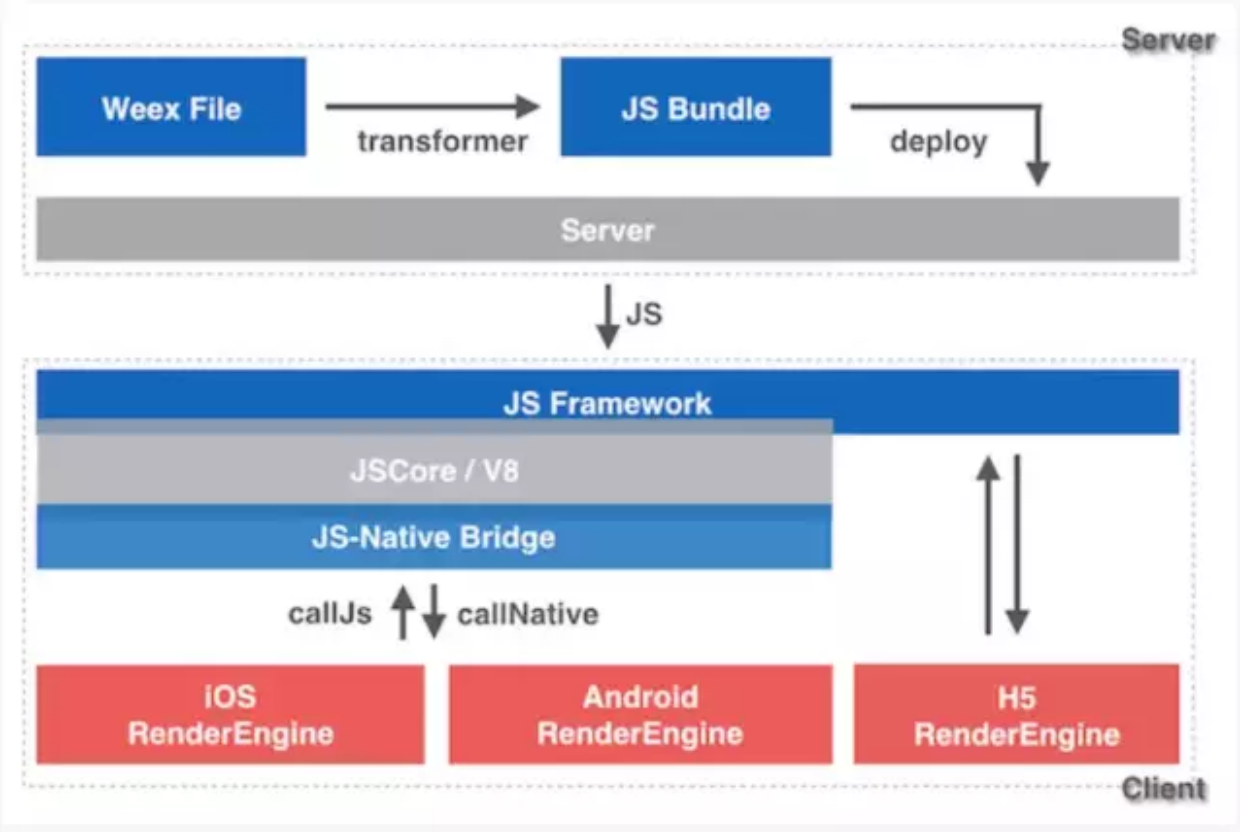
3. weex

weex 一定程度上用 JS 实现了 vue 一统天下的效果。

可以看到，weex 会编译构建虚拟 DOM，并发送渲染指令给 RenderEngine 层，这样，同样一份 JSON 数据，在不同平台的渲染引擎中能够渲染成不同版本的 UI，这是 Weex 可以实现动态化的原因。



那三端的语法都不一样，Weex是怎么统一的？重点在于 JS Framework！



weex 在 RN 的 JS V8 引擎基础上，多了 JS Framework 承当了重要的职责，它主要负责：管理 Weex 的生命周期；解析 JS Bundle，转为 Virtual DOM，再通过所在平台不同的 API 构建页面；进行双向的数据交互和响应。

这使得上层具备统一性，在开发过程中，代码模式、编译过程、模板组件、数据绑定、生命周期等上层语法是一致的。得益于上层的统一，只需要在 JS Framework 层的最后判断是由 Vue.js 生成真实的 DOM，还是通过 Native Api 渲染组件即可。

4. Flutter

RN 和 React 原理相通，那 Flutter 呢？Flutter Widget 的中心思想是用 Widget 构建你的UI（非原生控件）。那少了原生控件层和 js 层的通信损耗，不需要用虚拟 DOM 了吧?

非也！Flutter Widget 从 React 中获得了灵感，也是采用现代响应式框架构建。

先看看 Flutter 中三颗重要的树:

Widget 树：控件树，表示了我们在 dart 代码中所写的控件的结构，但这只是描述信息，渲染引擎是不认识的。

Widget 被开发人员配置了多个属性来定义它的展现形式，例如配置 Text 组件需要显示的字符串，配置输入框组件需要显示的内容……Element 树会记录这些配置信息。

Element 数：实际控件树

在手机屏幕上显示的控件并非我们在代码中所写的 Widget，Flutter 会根据 Widget 树信息生成控件对应的 Element 树，在 Flutter 中，一个 Widget 通过多次复用可以对应多个 Element 实例，Element 才是我们真正在屏幕上显示的元素。

Element 与 Widget 另一个区别在于，Widget 是不可变的，它的改变就意味着要重建，而其重建也非常频繁，如果我们将更多的任务交给它，将会对性能造成很大的损耗，因此我们把 Widget 树当作一个虚拟 DOM 树，真正被渲染在屏幕上的其实是 ElememtTree，它持有其对应 Widget 的引用，如果对应的 Widget 发生改变，它就会被标记为 dirty Element，下一次更新视图时根据这个状态只更新被修改的内容，这样就把可变状态与 Widget 关联起来，从而达到提升性能的效果。

RenderObject 树：渲染树，做组件布局渲染工作，包含渲染搭配、布局约束等信息。

简而言之，Flutter 引入虚拟 DOM 的目的是为了确定底层渲染树从一个状态转换到下一个状态所需的最小更改。

虚拟 DOM 对跨平台技术的意义

那分析完各种跨平台技术，你对虚拟 DOM 有了怎样的认识了呢?

为什么使用虚拟 DOM?

是因为快? （实际上不一定快）

是因为解耦?

是因为响应式?

对跨平台技术来说，更重要的意义在于：

虚拟 DOM 是 DOM 在内存中的一种轻量级表达方式，是一种统一约定！**可以通过不同的渲染引擎生成不同平台下的 UI！**

虚拟 DOM 的可移植性非常好，这意味着可以渲染到 DOM 以外的任何端，发挥你的想象力，可以做的事情很多。

再次审视虚拟 DOM

虚拟 DOM 真正的价值从来都不是性能，而是不管数据怎么变化，都可以用最小的代价来更新 DOM，而且掩盖了底层的 DOM 操作，让你用更声明式的方式来描述你的目的，从而让你的代码更容易维护。

虚拟 DOM 带来了很多好思路，打开了通向有趣架构的大门，例如将视图视为状态函数。它让我们编写代码，就像重新呈现整个场景一样。这不禁让我感慨，没有什么是加中间件不能解决的，如果有，那就再加多个中间件。

5 个词语概括下意义：

可维护性、最小的代价、效率、函数式UI、数据驱动

进一步思考

虚拟 DOM 的说明已经结束了，但是对于虚拟 DOM 的思考远没有结束。

Rect 的方式有两大缺点：

1、每次数据更改，哪怕改动很小，都会生成完整的虚拟 DOM，如果 DOM 很复杂，这个过程就会白白浪费很多计算资源；

2、比较虚拟 DOM 差异的过程，既慢又容易出错。因为 React 持有的新旧虚拟 DOM 相互独立，React 并不知道数据源发生了什么操作，只能根据两个虚拟 DOM 来猜测需要执行的操作。自动的猜测算法既不准又慢，必须要前端开发者手动提供 key 属性和一些额外的方法实现来帮助 React 猜对。

那么?

留个思考题，vue 是怎么利用虚拟 DOM 的？针对以上缺点怎么做改进？大家可以去了解一下。