09 | 真正理解虚拟 DOM：React 选它，真的是为了性能吗?

2020/11/09 修言

在过去的十年里，前端技术日新月异。从最早的纯静态页面，到 jQuery 一统江湖，再到近几年大火的 MVVM 框架——研发模式升级这件事情对于前端来说，好像成了某种常态。其实**研发模式不断演进的背后，恰恰蕴含着前端人对 “DOM 操作” 这一核心动作的持续思考和改进**。而虚拟 DOM，正是先驱们在这个过程中孕育出的一颗明珠。

在 MVVM 框架这个领域分支，有一道至今仍然非常经典的面试题：“**为什么我们需要虚拟 DOM?** ”。

这个问题比较常见的回答思路是：“**DOM 操作是很慢的，而 JS 却可以很快**，直接操作 DOM 可能会导致频繁的回流与重绘，JS 不存在这些问题。因此虚拟 DOM 比原生 DOM 更快”。

但真的是这样吗？学完本课时，你心中自会有答案。

快速搞定虚拟 DOM 的两个“大问题”

温故而知新，在一切开始之前，我们先来复习一下虚拟 DOM是什么。

虚拟 DOM（Virtual DOM）本质上是**JS 和 DOM 之间的一个映射缓存**，它在形态上表现为一个能够描述 DOM 结构及其属性信息的 **JS 对象**。在第 01 课时，我们探讨了 JSX 和 DOM 之间的转换关系，其中就提到了虚拟 DOM 在 React 中的形态，如下图所示：

就这个示例来说，你需要把握住以下两点：

1. 虚拟 DOM 是 JS 对象
2. 虚拟 DOM 是对真实 DOM 的描述

这样就基本解决了虚拟 DOM“是什么”的问题，接下来我们看看 React 中的虚拟 DOM 大致是如何工作的。虚拟 DOM 在 React 组件的挂载阶段和更新阶段都会作为“关键人物”出镜，其参与的工作流程如下：

- **挂载阶段**，React 将结合 JSX 的描述，构建出虚拟 DOM 树，然后通过 ReactDOM.render 实现虚拟 DOM 到真实 DOM 的映射（触发渲染流水线）；
- **更新阶段**，页面的变化在作用于真实 DOM 之前，会先作用于虚拟 DOM，虚拟 DOM 将在 JS 层借助算法先对比出具体有哪些真实 DOM 需要被改变，然后再将这些改变作用于真实 DOM。

OK，现在我们用最短的时间迅速搞定了“What”和“How”两个大问题。或许过程有些粗糙，但这丝毫不影响你吃透本课时的核心内容，也就是虚拟 DOM 背后的“Why”。

“为什么需要虚拟 DOM？”“虚拟 DOM 的优势何在？”“虚拟 DOM 是否伴随更好的性能？”，要想回答好这无穷无尽的为什么，你千万**不要点对点**地去看待问题本身。虚拟 DOM 相对于过往的 DOM 操作解

决方案来说，是一个新生事物。要想理解一个新生事物存在、发展的合理性，我们必须**将其放在一个足够长的、合理的上下文中去讨论**。

接下来我要做的事情，就是帮你把这个上下文完全地铺开。当你清楚了虚拟 DOM 在历史长河中的位置后，将能迅速地理解它到底帮助前端开发解决掉了什么问题，彼时，所有的答案都会跃然纸上。

历史长河中的 DOM 操作解决方案

现在，让我们一起来回顾一下，那些没有虚拟 DOM 的苦逼日子。

1. 原生 JS 支配下的“人肉 DOM” 时期

在前端这个工种的萌芽阶段，前端页面“展示”的属性远远强于其“交互”的属性，这就导致 JS 的定位只能是“辅助”：在很长一段时间里，前端工程师们会花费大量的时间去实现静态的 DOM，待一切结束后，再补充少量 JS，实现一些类似于拖拽、隐藏、幻灯片之类的“特效”。

在这个阶段，作为前端开发者来说，虽然我们一无所有，但过得很快乐——简单的业务需求决定了我们不需要去做太多或太复杂的 DOM 操作，原生 JS，足矣。

2.解放生产力的先导阶段：jQuery 时期

时代的浪潮滚滚向前，人们很快就不再满足于简单到有些无聊的交互效果，开始追求更加丰富的用户体验，与之而来的就是大量 DOM 操作需求带来的前端开发工作量的激增。在这个过程中，早期前端们渐渐地明白了一个道理：原生 JS 提供的 DOM API，实在是太太太太太难用了。

为了能够实现高效的开发，jQuery 首先解决的就是“API 不好使”这个问题——它将 DOM API 封装为了相对简单和优雅的形式，同时一口气做掉了跨浏览器的兼容工作，并且提供了链式 API 调用、插件扩展等一系列能力用于进一步解放生产力。最终达到的效果正是我们喜闻乐见的“写得更少，做得更多”。

jQuery 使 DOM 操作变得简单、快速，并且始终确保其形式稳定、可用性稳定。虽然现在看来并不完美，但在当年能够一统江湖，确实当之无愧。

3.民智初启：早期模板引擎方案

jQuery 帮助我们能够以更舒服的姿势操作 DOM，但它并不能从根本上解决 DOM 操作量过大情况下前端侧的压力。

它就好比是一个**手持吸尘器**，虽然可以帮助我们更加方便快速地清洁某一处的灰尘，但是要想清洁多个位置的灰尘，你仍然需要拿着它四处奔走。这样虽说不必再弯腰擦地板，但还是避免不了跑断腿的结局。

既然“手持吸尘器”满足不了日益膨胀的 DOM 操作需求，那我们想要的到底是什么呢？是一个只需要接收命令，就能够自己跑来跑去、把活干得漂漂亮亮的“扫地机器人”。

而模板引擎方案，正是“扫地机器人”的雏形。

> 注：由于模板引擎更倾向于点对点解决烦琐 DOM 操作的问题，它在能力和定位上既不能够、也不打算替换掉 jQuery，两者是和谐共存的。因此这里不存在“模板引擎时期”，只有“模板引擎方案”。

怎么理解模板这个概念呢？我们来看一个例子。比如说我现在手里有一套员工数据，数据内容如下：

复制代码

```
const staff = [
  {
    name: '修言',
    career: '前端'
  },
  {
    name: '翠翠',
    career: '编辑'
  },
  {
    name: '花花',
    career: '运营'
  }
]
```

现在我想要在前端用表格展示这一堆数据，我就可以遵循模板的语法，把它塞进模板（template）里去。下面就是一个典型的模板语法使用示例：

复制代码

```
<table>
  {% staff.forEach(function(person){ %}
  <tr>
    <td>{% student.name %}</td>
    <td>{% student.age %}</td>
  </tr>
  {% }); %}
</table>
```

可以看出，模板语法其实就是把 JS 和 HTML 结合在一起的一种规则，而模板引擎做的事情也非常容易理解。

把 staff 这个数据源读进去，塞到预置好的 HTML 模板里，然后把两者融合在一起，吐出一段目标字符串给你。这段字符串的内容，其实就是一份标准的、可用于渲染的 HTML 代码，它将对应一个 DOM 元素。最后，将这个 DOM 元素挂载到页面中去，整个模板的渲染流程也就走完了。

这个过程可以用伪代码来表示，如下所示：

```
// 数据和模板融合出 HTML 代码
var targetDOM = template({data: students})
// 添加到页面中去
document.body.appendChild(targetDOM)
```

复制代码

当然，实际的过程会比我们描述的要复杂一些。这里我补充一下模板引擎的实现思路，供感兴趣的同学参考。模板引擎一般需要做下面几件事情：

1. 读取 HTML 模板并解析它，分离出其中的 JS 信息；
2. 将解析出的内容拼接成字符串，动态生成 JS 代码；
3. 运行动态生成的 JS 代码，吐出“目标 HTML”；
4. 将“目标 HTML”赋值给 innerHTML，触发渲染流水线，完成真实 DOM 的渲染。

使用模板引擎方案来渲染数据是非常爽的：每次数据发生变化时，我们都不用关心到底是哪里的数据变了，也不用手动去点对点完成 DOM 的修改。只**需要关注的仅仅是数据和数据变化本身**，DOM 层面的改变模板引擎会帮我们做掉。

如此看来，模板引擎像极了一个只需要接收命令，就能够把活干得漂漂亮亮的“扫地机器人”！可惜的是，模板引擎出现的契机虽然是为了使用户界面与业务数据相分离，但实际的应用场景基本局限在“实现高效的字符串拼接”这一个点上，因此不能指望它去做太复杂的事情。尤其令人无法接受的是，**它在性能上的表现并不尽如人意**：由于不够“智能”，它更新 DOM 的方式是将已经渲染出 DOM 整体注销后再整体重渲染，并且不存在更新缓冲这一说。在 DOM 操作频繁的场景下，模板引擎可能会直接导致页面卡死。

> 注：请注意小标题中“早期”这个限定词——本课时所讨论的“模板引擎”概念，指的是虚拟 DOM 思想推而广之以前，相对原始的一类模板引擎，这类模板引擎曾经主导了一个时代。但时下来看，越来越多的模板引擎正在引入虚拟 DOM，模板引擎最终也将走向现代化。

虽然指望模板引擎实现生产力解放有些天方夜谭，但模板引擎在思想上无疑具备高度的先进性：允许程序员只关心数据而不必关心 DOM 细节的这一操作，和 React 的“数据驱动视图”思想如出一辙，实在是高！

那该怎么办呢?

jQuery 救不了加班写 DOM 操作的前端，模板引擎也救不了，那该怎么办呢?

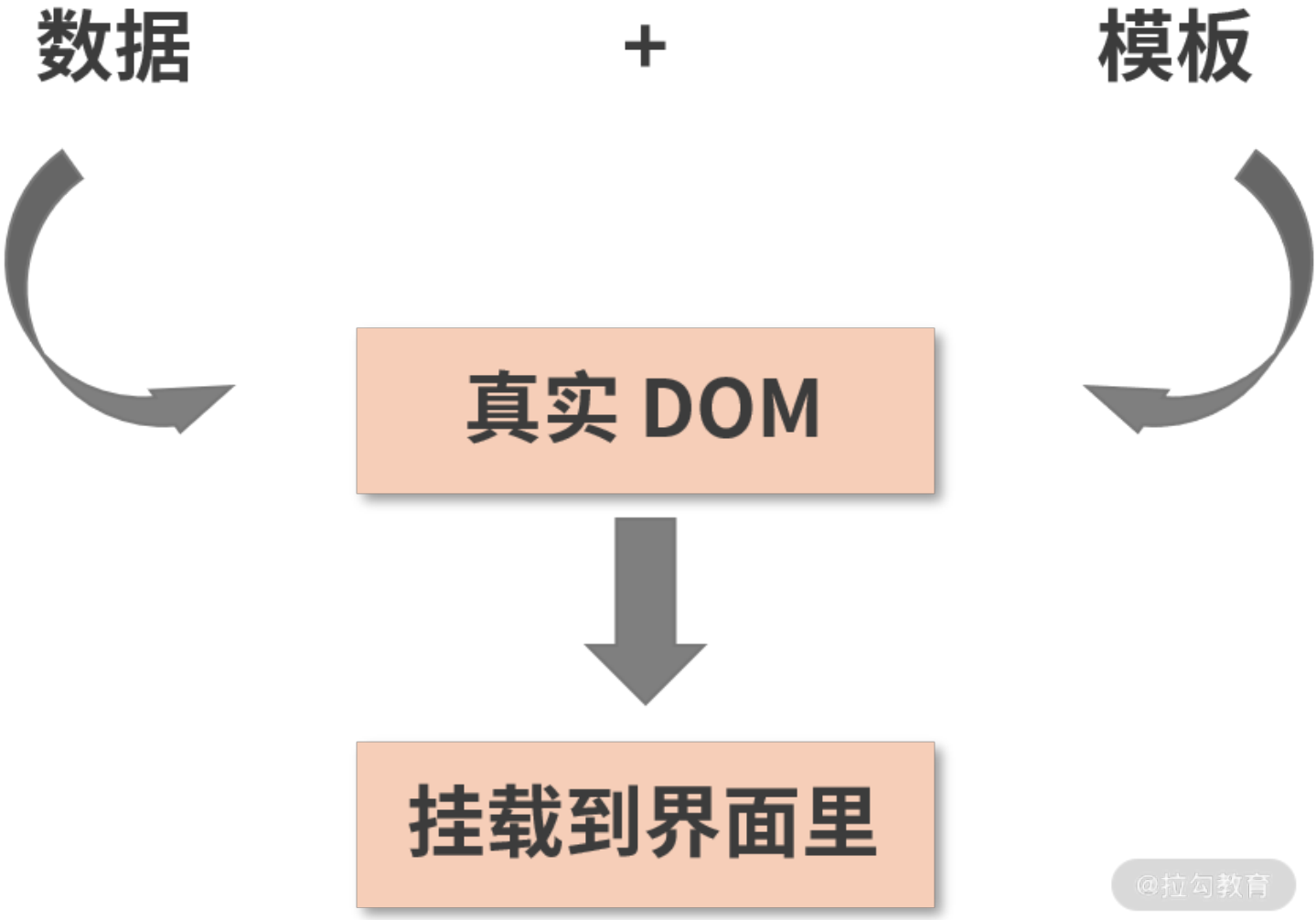
这时候有一批仁人志士，兴许是从模板引擎的设计思想上得到了启发，他们明确了要走“数据驱动视图”这条基本道路，于是便沿着这个思路往下摸索：模板引擎的数据驱动视图方案，核心问题在于对真实 DOM 的修改过于“大刀阔斧”，导致了 DOM 操作的范围过大、频率过高，进而可能会导致糟糕的性能。然后这帮人就想啊：既然操作真实 DOM 对性能损耗这么大，那我操作假的 DOM 不就行了?

沿着这个思路再往下走，就有了我们都爱的虚拟 DOM。

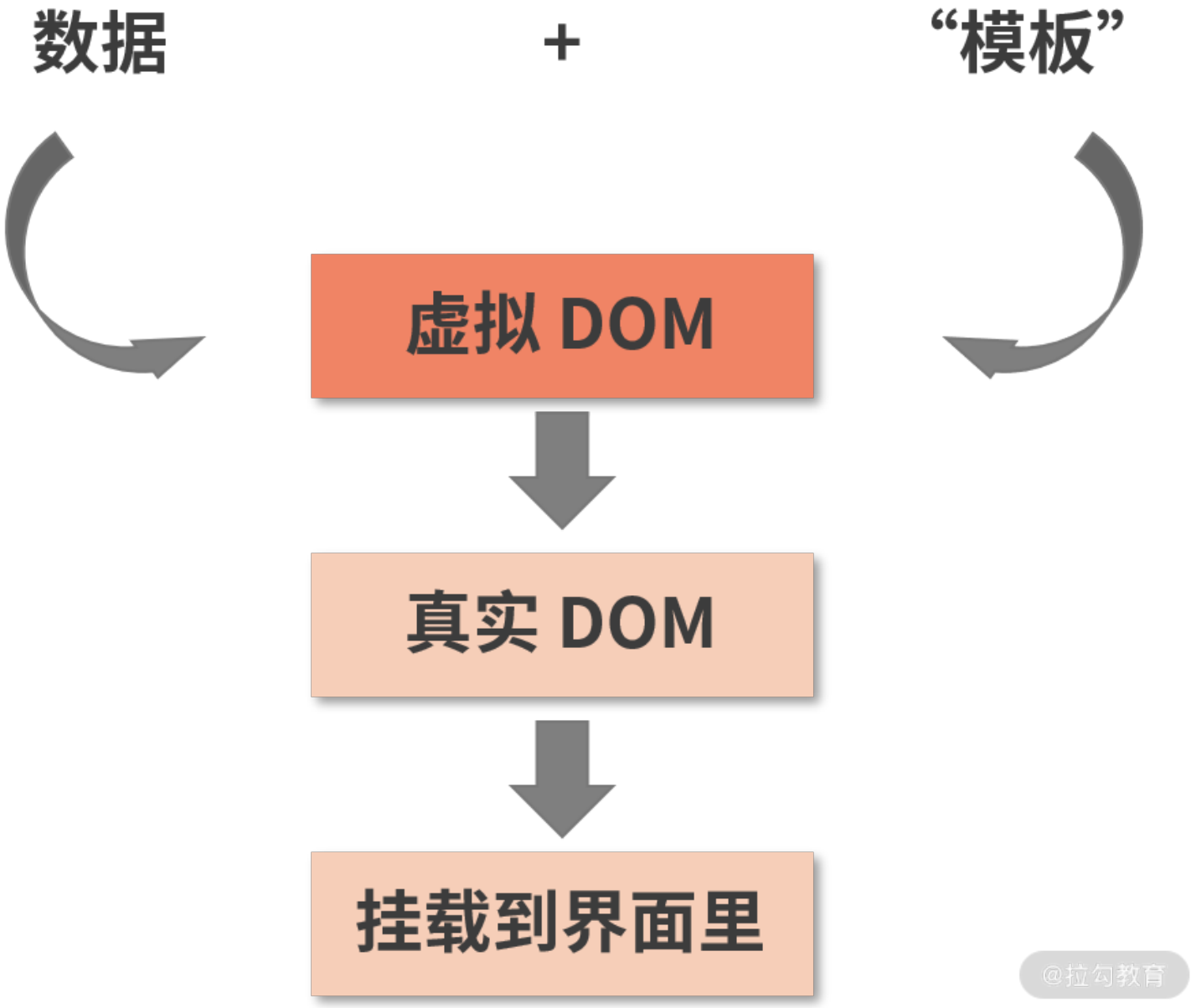
> 注：出于严谨，还是要解释下。真实历史中的虚拟 DOM 创作过程，到底有没有向模板引擎去学习，这个暂时无从考证。但是按照前端发展的过程来看，模板引擎和虚拟 DOM 确实在思想上存在递进关系，很多场景下，面试官也可能会问及两者的关系。因此在此处，我采取了这样一种表述方式，希望能够帮助你更好地把握住问题的关键所在。

虚拟 DOM 是如何解决问题的

读到这里，你可能对虚拟 DOM 已经有些感觉了。这里我来帮你总结下，同样是将用户界面与数据相分离，模板引擎是这样做的：

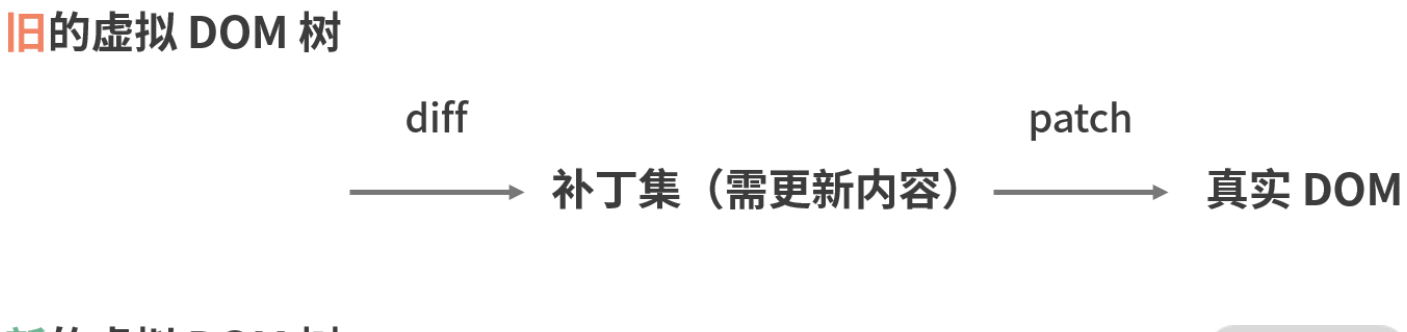


而在虚拟 DOM 的加持下，事情变成了这样：



注意图中的“模板”二字加了引号，这是因为虚拟 DOM 在实现上并不总是借助模板。比如 React 就使用了 JSX，前面咱们着重讲过，JSX 本质不是模板，而是一种使用体验和模板相似的 JS 语法糖。

区别就在于多出了一层虚拟 DOM 作为缓冲层。这个缓冲层带来的利好是：当 DOM 操作（渲染更新）比较频繁时，它会先将前后两次的虚拟 DOM 树进行对比，定位出具体需要更新的部分，生成一个“补丁集”，最后只把“补丁”打在需要更新的那部分真实 DOM 上，实现精准的“**差量更新**”。这个过程对应的虚拟 DOM 工作流如下图所示：



> 注：图中的 diff 和 patch 其实都是函数名，这些函数取材于一个独立的虚拟 DOM 库。之所以写明了具体流程对应的函数名，是因为我发现面试的时候，很多面试官习惯于用函数名指代过程，但不少人不清楚这个对应关系（尤其是 patch），会非常影响作答。这里提前帮你把这个坑给规避掉。

还需要说明的一点是， 虚拟 DOM 和 Redux 一样，不依附于任何具体的框架。学习虚拟 DOM，实际上可以完全不借助 React；但学习 React，就必须了解虚拟 DOM。如果你对虚拟 DOM 的具体实现过程感兴趣，可以在这个 GitHub 仓库里查看其源码细节。

回到主线剧情上来，差量更新可以确保虚拟 DOM 既能够提供高效的开发体验（开发者只需要关心数据），又能够保持过得去的性能（只更新发生了变化的那部分 DOM），实在是妙啊！

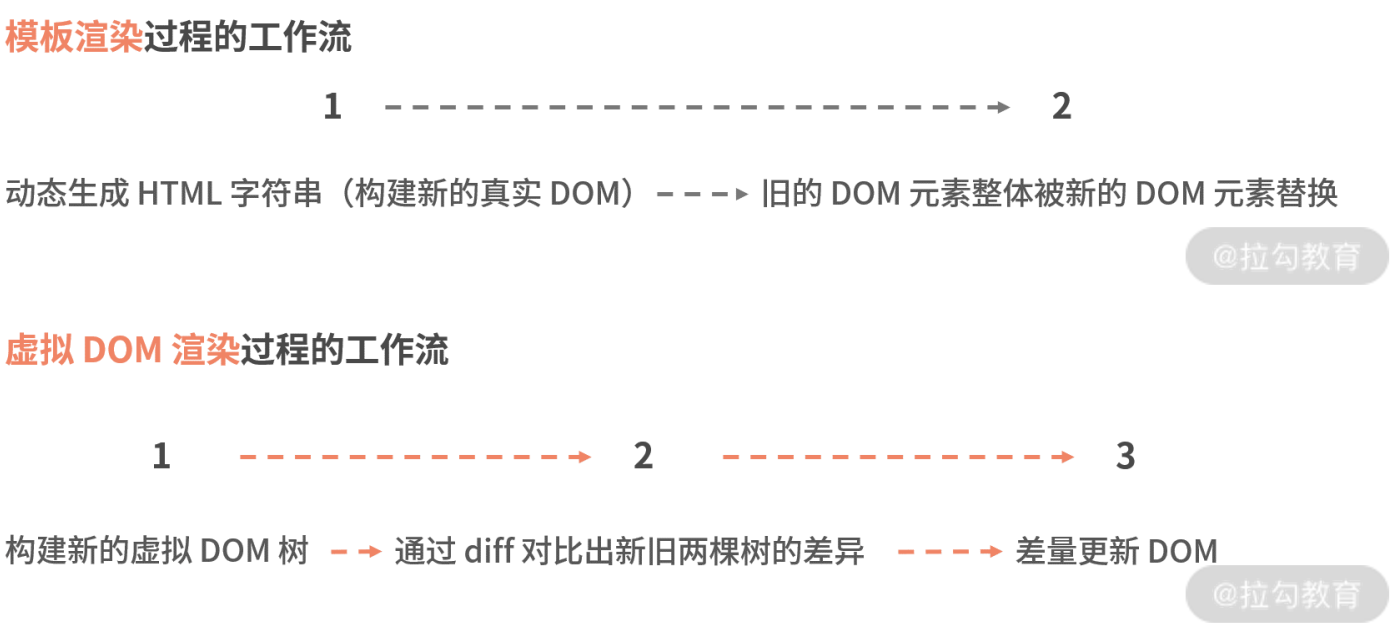
React 选用虚拟 DOM，真的是为了更好的性能吗？

读到这里，相信你至少已经 get 到了这样一个点：在整个 DOM 操作的演化过程中，主要矛盾并不在于性能，而在于开发者写得爽不爽，在于**研发体验/研发效率**。虚拟 DOM 不是别的，正是前端开发们为了追求更好的研发体验和研发效率而创造出来的高阶产物。

虚拟 DOM 并不一定会带来更好的性能，React 官方也从来没有把虚拟 DOM 作为性能层面的卖点对外输出过。**虚拟 DOM 的优越之处在于，它能够在提供更爽、更高效的研发模式（也就是函数式的 UI 编程方式）的同时，仍然保持一个还不错的性能。**

性能问题属于前端领域复杂度比较高的问题。当我们量化性能的时候，往往并不能只追求一个单一的数据，而是需要结合具体的参照物、渲染的阶段、数据的吞吐量等各种要素来作分情况的讨论。

拿前面讲过的模板渲染来举例，我们可以对比一下它和虚拟 DOM 在性能开销上的差异。两者的渲染工作流对比如下图所示：



从图中可以看出，模板渲染的步骤1，和虚拟 DOM 渲染的步骤1、2都属于 JS 范畴的行为，这两者是具备可比性的，我们放在一起来看：动态生成 HTML 字符串的过程本质是对字符串的拼接，对性能的消耗是有限的；而虚拟 DOM 的构建和 diff 过程逻辑则相对复杂，它不可避免地涉及递归、遍历等耗时操作。**因此在 JS 行为这个层面，模板渲染胜出。**

模板渲染的步骤3，和虚拟 DOM 的步骤3 都属于 DOM 范畴的行为，两者具备可比性，因此我们仍然可以愉快地对比下去：模板渲染是全量更新，而虚拟 DOM 是差量更新。

乍一看好像差量更新一定比全量更新高效，但你需要考虑这样一种情况：数据内容变化非常大（或者说整个发生了改变），促使差量更新计算出来的结果和全量更新极为接近（或者说完全一样）。

在这种情况下，DOM 更新的工作量基本一致，而虚拟 DOM 却伴随着开销更大的 JS 计算，此时会出现的一种现象就是模板渲染和虚拟 DOM 在整体性能上难分伯仲：若两者最终计算出的 DOM 更新内容完全一致，那么虚拟 DOM 大概率不敌模板渲染；但只要两者在最终 DOM 操作量上拉开那么一点点的差距，虚拟 DOM 就将具备战胜模板渲染的底气。**因为虚拟 DOM 的劣势主要在于 JS 计算的耗时，而 DOM 操作的能耗和 JS 计算的能耗根本不在一个量级**，极少量的 DOM 操作耗费的性能足以支撑大量的 JS 计算。

当然，上面讨论的这种情况相对来说比较极端。在实际的开发中，更加高频的场景是这样的：我每次 setState 的时候只修改少量的数据，比如一个对象中的某几个属性，再比如一个数组中的某几个元素。在这样的场景下，模板渲染和虚拟 DOM 之间 DOM 操作量级的差距就完全拉开了，虚拟 DOM 将在性能上具备绝对的优势。

> 注意，此处的结论是“在 XXX 场景下，虚拟 DOM 相对于 XXX 具备性能优势”，它是有严格限定条件的。有人不到黄河心不死，可能又要问“那虚拟 DOM 对比 jQuery 呢？”“那虚拟 DOM 对比原生 DOM 呢？”。
> 我想说的是，性能问题不能一概而论，而且咱都讲到这个份上了，就不要再钻性能这个牛角尖了。jQuery、原生 DOM 在思维模式上来说和虚拟 DOM 截然不同，强行比较意义不大。

前面又是分析又是举例地说了这么多，其实我最终希望你明白的事情只有一件：**虚拟 DOM 的价值不在性能，而在别处**。因此想要从性能角度来把握虚拟 DOM 的优势，无异于南辕北辙。偏偏在面试场景下，10 个人里面有 9 个都走这条歧路，最后9个人里面自然没有一个能自圆其说，实在让人惋惜。

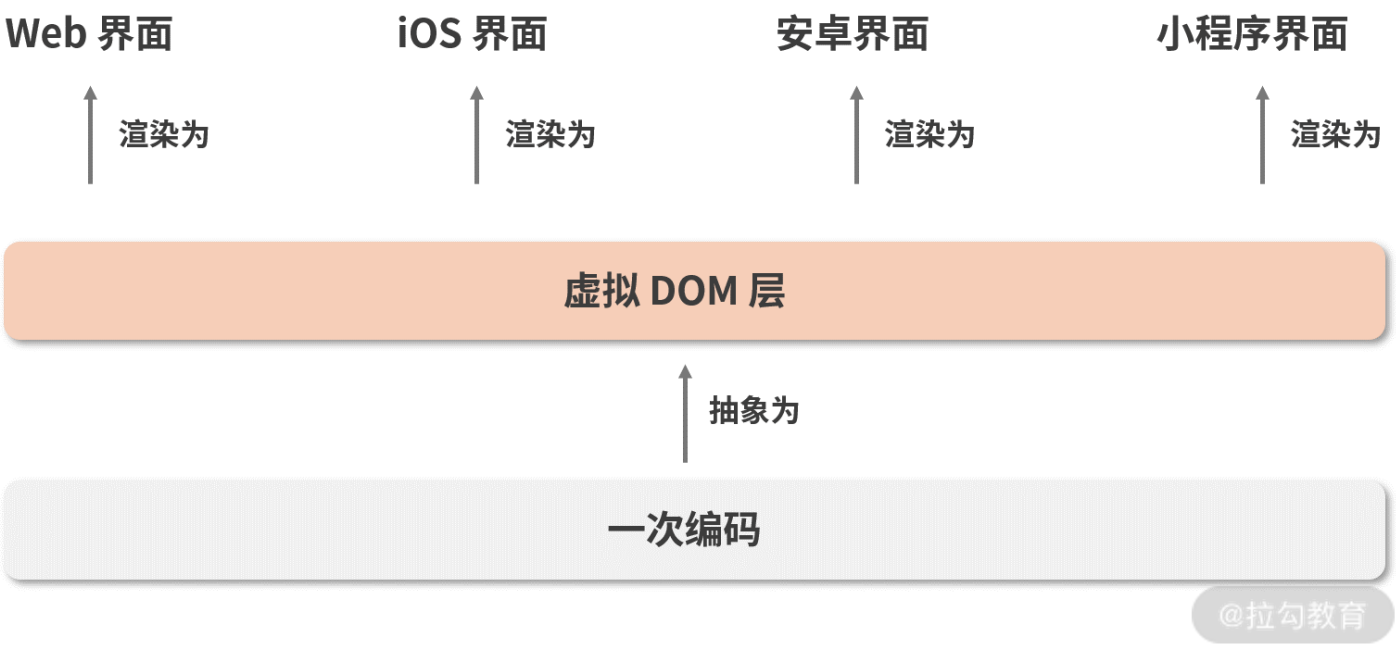
那么虚拟 DOM 的价值到底是什么呢?

最后我想和你聊聊虚拟 DOM 的价值，这又是一个宏大的、容易说错话的命题。当我们谈及某个事物的价值时，其实就像是在称赞一个美女，不同的人自然有着不同看待美女的视角。此处我无意于给你一个天衣无缝的标准答案（这样的答案想必也不存在），而是希望能够站在“虚拟 DOM 解决了哪些关键问题”这个视角，和你分享一些业内关于虚拟 DOM 的共识。

虚拟 DOM 解决的关键问题有以下两个。

1. 研发体验/研发效率的问题：这一点前面已经反复强调过，DOM 操作模式的每一次革新，背后都是前端对效率和体验的进一步追求。虚拟 DOM 的出现，为数据驱动视图这一思想提供了高度可用的载体，使得前端开发能够基于函数式 UI 的编程方式实现高效的声明式编程。

2. 跨平台的问题：虚拟 DOM 是对真实渲染内容的一层抽象。若没有这一层抽象，那么视图层将和渲染平台紧密耦合在一起，为了描述同样的视图内容，你可能要分别在 Web 端和 Native 端写完全不同的两套甚至多套代码。但现在中间多了一层描述性的虚拟 DOM，它描述的东西可以是真实 DOM，也可以是iOS 界面、安卓界面、小程序.....同一套虚拟 DOM，可以对接不同平台的渲染逻辑，从而实现“一次编码，多端运行”，如下图所示。其实说到底，跨平台也是研发提效的一种手段，它在思想上和1是高度呼应的。



在本课时的主线内容之外，虚拟 DOM 还有非常多的亮点值得我们去挖掘，这里我想着重拓展一下的是前面聊到的性能层面的优化。

除了差量更新以外，“**批量更新**”也是虚拟 DOM 在性能方面所做的一个重要努力：**“批量更新”在**通用虚拟 DOM 库**里是由 batch 函数来处理的**。在差量更新速度非常快的情况下（比如极短的时间里多次操作同一个 DOM），用户实际上只能看到最后一次更新的效果。这种场景下，前面几次的更新动作虽然意义不大，但都会触发重渲染流程，带来大量不必要的高耗能操作。

这时就需要请 batch 来帮忙了，**batch 的作用是缓冲每次生成的补丁集**，它会把收集到的多个补丁集暂存到队列中，再将最终的结果交给渲染函数，最终实现集中化的 DOM 批量更新。

总结

本课时我们首先一起回顾了 DOM 操作解决方案的发展史，从中明确了虚拟 DOM 定位和解决的主要问题，然后对虚拟 DOM 的通用工作流进行了分析。在这个工作流中，有一个过程值得我们格外去注意，那就是“diff”。

diff 指的是对比出两棵虚拟 DOM 树之间差异的过程，不同的框架对 diff 过程有着不同的实现思路。对于 React 框架来说，有特色的、与时俱进的 diff 算法正是它最迷人的地方，也是框架的核心所在。

在接下来的课时，我将从 React15 的 diff 过程切入，对经典的“栈调和”算法一探究竟。