究竟什么是虚拟 DOM

DOM 怎么了

如我们所知，在浏览器渲染网页的过程中，加载到 HTML 文档后，会将文档解析并构建 DOM 树，然后将其与解析 CSS 生成的 CSSOM 树一起结合产生爱的结晶——RenderObject 树，然后将 RenderObject 树渲染成页面（当然中间可能会有一些优化，比如 RenderLayer 树）。这些过程都存在与渲染引擎之中，渲染引擎在浏览器中是于 JavaScript 引擎（JavaScriptCore 也好 V8 也好）分离开的，但为了方便 JS 操作 DOM 结构，渲染引擎会暴露一些接口供 JavaScript 调用。
由于这两块相互分离，通信是需要付出代价的，因此 JavaScript 调用 DOM 提供的接口性能不咋地。各种性能优化的最佳实践也都在尽可能的减少 DOM 操作次数。对于 DOM 元素来说是非常大的，随便一个 div 中的所有 key 值的字符串拼接的结果

```
▶ div#wrapper.wrapper_s                                                    VM407:1
undefined
var str = ""
undefined
for(var prop in div) {
    str += prop;
}
"sizzle-1548688996637aligntitlelangtranslatedirdatasethiddentabIndexaccessKeydraggablespellcheckautoc
apitalizecontentEditableisContentEditableinputModeoffsetParentoffsetTopoffsetLeftoffsetWidthoffsetHei
ghtstyleinnerTextouterTextonabortonbluroncanceloncanplayoncanplaythroughonchangeonclickoncloseonconte
xtmenuoncuechangeondblclickondragondragendondragenterondragleaveondragoverondragstartondroponduration
changeonemptiedonendedonerroronfocusoninputoninvalidonkeydownonkeypressonkeyuponloadonloadeddataonloa
dedmetadataonloadstartonmousedownonmouseenteronmouseleaveonmousemoveonmouseoutonmouseoveronmouseuponm
ousewheelonpauseonplayonplayingonprogressonratechangeonresetonresizeonscrollonseekedonseekingonselect
onstalledonsubmitonsuspendontimeupdateontoggleonvolumechangeonwaitingonwheelonauxclickongotpointercap
tureonlostpointercaptureonpointerdownonpointermoveonpointeruponpointercancelonpointeroveronpointerout
onpointerenteronpointerleaveonnoceclickfocusblurnamespaceURIprefixlocalNametagNameidclassNameclassList
slotattributesshadowRootassignedSlotinnerHTMLouterHTMLscrollTopscrollLeftscrollWidthscrollHeightclien
tTopclientLeftclientWidthclientHeightattributeStyleMaponbeforecopyonbeforecutonbeforepasteoncopyoncut
onpasteonsearchonselectstartpreviousElementSiblingnextElementSiblingchildrenfirstElementChildlastElem
entChildchildElementCountonwebkitfullscreenchangeonwebkitfullscreenerrorsetPointerCapturereleasePoint
erCapturehasPointerCapturehasAttributesgetAttributeNamesgetAttributegetAttributeNSsetAttributesetAttr
ibuteNSremoveAttributeremoveAttributeNShasAttributehasAttributeNStoggleAttributegetAttributeNodegetAt
tributeNodeNSsetAttributeNodesetAttributeNodeNSremoveAttributeNodeclosestmatcheswebkitMatchesSelector
attachShadowgetElementsByTagNamegetElementsByTagNameNSgetElementsByClassNameinsertAdjacentElementinse
rtAdjacentTextinsertAdjacentHTMLrequestPointerLockgetClientRectsgetBoundingClientRectscrollIntoViewsc
rollIntoViewIfNeededanimatecomputedStyleMapbeforeafterreplaceWithremoveprependappendquerySelectorquer
ySelectorAllwebkitRequestFullScreenwebkitRequestFullscreenscrollscrollToscrollBycreateShadowRootgetDe
stinationInsertionPointsELEMENT_NODEATTRIBUTE_NODETEXT_NODECDATA_SECTION_NODEENTITY_REFERENCE_NODEENT
ITY_NODEPROCESSING_INSTRUCTION_NODECOMMENT_NODEDOCUMENT_NODEDOCUMENT_TYPE_NODEDOCUMENT_FRAGMENT_NODEN
OTATION_NODEDOCUMENT_POSITION_DISCONNECTEDDOCUMENT_POSITION_PRECEDINGDOCUMENT_POSITION_FOLLOWINGDOCUM
ENT_POSITION_CONTAINSDOCUMENT_POSITION_CONTAINED_BYDOCUMENT_POSITION_IMPLEMENTATION_SPECIFICnodeTypen
odeNamebaseURIisConnectedownerDocumentparentNodeparentElementchildNodesfirstChildlastChildpreviousSib
lingnextSiblingnodeValuetextContenthasChildNodesgetRootNodenormalizecloneNodeisEqualNodeisSameNodecom
pareDocumentPositioncontainslookupPrefixlookupNamespaceURIisDefaultNamespaceinsertBeforeappendChildre
placeChildremoveChildaddEventListenerremoveEventListenerdispatchEvent"
```

可见我们任何页面中的 DOM 集合所占大小可见一斑。所以对于 DOM 操

作来说性能是很差的。

当初 MVC、MVP 的架构模式，希望能从代码组织方式来降低维护这种复杂应用程序的难度。但是 MVC 架构没办法减少你所维护的状态，也没有降低状态更新你需要对页面的更新操作（前端来说就是 DOM 操作），你需要操作的 DOM 还是需要操作，只是将数据、操作、视图分离了。
既然状态改变了要操作相应的 DOM 元素，为什么不做一个东西可以让视图和状态进行绑定，状态变更了视图自动变更，就不用手动更新页面了。这就是后来人们想出了 MVVM 模式，只要在模版中声明视图组件是和什么状态进行绑定的，双向绑定引擎就会在状态更新的时候自动更新视图。MVVM 可以很好的降低我们维护状态 -> 视图的复杂程度（大大减少代码中的视图更新逻辑）。但是这不是唯一的办法，还有一个非常直观的方法，可以大大降低视图更新的操作：一旦状态发生了变化，就用模版引擎重新渲染整个视图，然后用新的视图更换掉旧的视图。就像上面的表格，当用户点击的时候，还是在 JS 里面更新状态，但是页面更新就不用手动操作 DOM 了，直接把整个表格用模版引擎重新渲染一遍，然后设置一下 innerHTML 就完事了。
听到这样的做法，经验丰富的你一定第一时间意识这样的做法会导致很多的问题。最大的问题就是这样做会很慢，因为即使一个小小的状态变更都要重新构造整棵 DOM，性价比太低；而且这样做的话，input 和 textarea 的会失去原有的焦点。最后的结论会是：对于局部的小视图的更新，没有问题（Backbone 就是这么干的）；但是对于大型视图，如全局应用状态变更的时候，需要更新页面较多局部视图的时候，这样的做法不可取。
但是这里要明白和记住这种做法，因为后面你会发现，**其实 Virtual DOM 就是这么做的，只是加了一些特别的步骤来避免了整棵 DOM 树变更**。
另外一点需要注意的就是，上面提供的几种方法，其实都在解决同一个问题：**维护状态，更新视图**。在一般的应用当中，如果能够很好方案来应对这个问题，那么就几乎降低了大部分复杂性

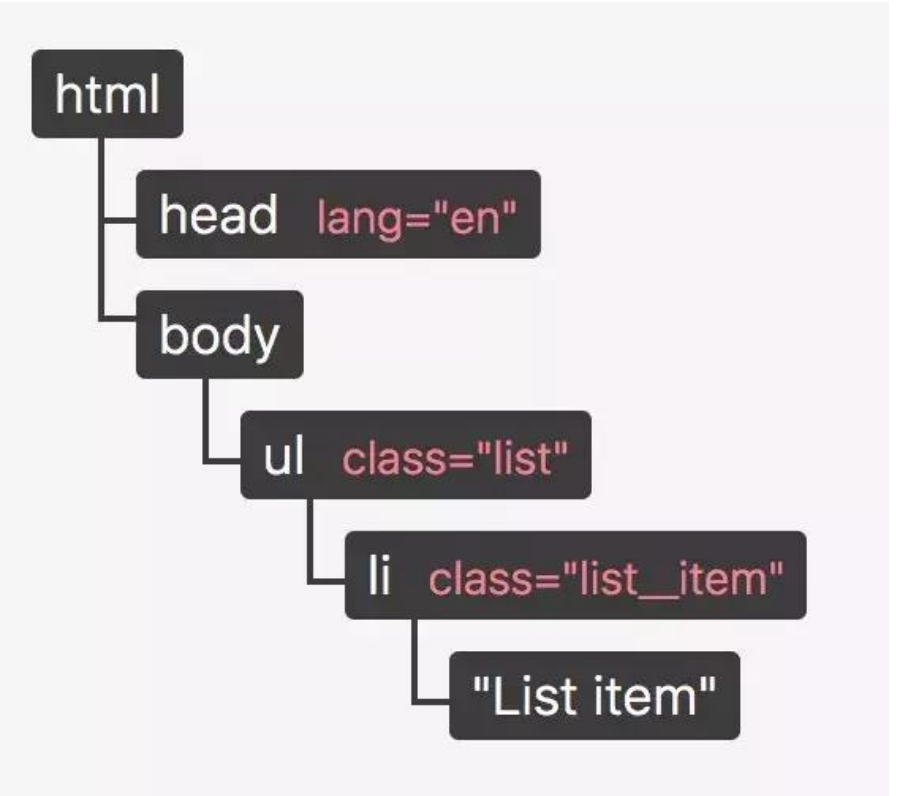
什么是虚拟 DOM

创建虚拟 DOM 是为了更高效、频繁地更新 DOM 。与 DOM 或 shadow DOM 不同，虚拟 DOM 不是官方规范，而是一种与 DOM 交互的新方法。虚拟 DOM 被认为是原始 DOM 的副本。此副本可被频繁地操作和更新，而无需使用 DOM API。一旦对虚拟 DOM 进行了所有更新，我们就可以查

看需要对原始 DOM 进行哪些特定更改，最后以目标化和最优化的方式进行更改。
“虚拟 DOM ”这个名称往往会增加这个概念实际上的神秘面纱。实际上，虚拟 DOM 只是一个常规的 Javascript 对象。

对于同样的一棵 DOM 树来说：

。



原生 DOM：

```
<html lang="en">
 <head></head>
 <body>
     <ul class="list">
         <li class="list__item">List item</li>
     </ul>
```

```
  </body>
</html>
```

虽说着这么几个简短的标签，但是事迹展开的内容量是非常巨大的。

用虚拟 DOM 来表示：

```
const vdom = {
    tagName: "html",
    children: [
        { tagName: "head" },
        {
            tagName: "body",
            children: [
                {
                    tagName: "ul",
                    attributes: { "class": "list" },
                    children: [
                        {
                            tagName: "li",
                            attributes: { "class": "list__item" },
                            textContent: "List item"
                        } // end li
                    ]
                } // end ul
            ]
        } // end body
    ]
} // end html
```

与原始 DOM 一样，它是我们的 HTML 文档基于对象的表示。因为它是一个简单的 Javascript 对象，我们可以随意并频繁地操作它，而无须触及真实的 DOM 。

不一定要使用整个对象，更常见是使用小部分的虚拟 DOM 。例如，我们可以处理列表组件，它将对无序列表元素进行相应的处理。

虚拟 DOM 原理

状态变更->重新渲染整个视图的方式用 JavaScript 对象表示 DOM 信息和结构，当状态变更的时候，重新渲染这个 JavaScript 的对象结构。当然这样做其实没什么卵用，因为真正的页面其实没有改变。

但是可以用新渲染的对象树去和旧的树进行对比，记录这两棵树差异。记录下来的不同就是我们需要对页面真正的 DOM 操作，然后把它们应用在真正的 DOM 树上，页面就变更了。这样就可以做到：视图的结构确实是整个全新渲染了，但是最后操作 DOM 的时候确实只变更有不同的地方。

这就是所谓的 Virtual DOM 算法。包括几个步骤：

用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中
当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异
把 2 所记录的差异应用到步骤 1 所构建的真正的 DOM 树上，视图就更新了
Virtual DOM 本质上就是在 JS 和 DOM 之间做了一个缓存。可以类比 CPU 和硬盘，既然硬盘这么慢，我们就在它们之间加个缓存：既然 DOM 这么慢，我们就在它们 JS 和 DOM 之间加个缓存。CPU（JS）只操作内存（Virtual DOM），最后的时候再把变更写入硬盘（DOM）。

简单实现一个虚拟 DOM

相对于 DOM 对象，原生的 JavaScript 对象处理得更快，而且简单。DOM 树上的结构，属性信息我们都能通过 JavaScript 进行表示出来，例如：

```
var element = {
    tagName: 'ul', //  节点标签名
    props: { // dom 的属性键值对
        id: 'list'
    },
    children: [
```

```
        {tagName: 'li', props: {class: 'item'}, children: ["Item 1"]},
        {tagName: 'li', props: {class: 'item'}, children: ["Item 2"]},
        {tagName: 'li', props: {class: 'item'}, children: ["Item 3"]}
    ]
}
```

那么在 html 渲染的结果就是：

```
<ul id="list">
    <li class="item">Item 1</li>
    <li class="item">Item 2</li>
    <li class="item">Item 3</li>
</ul>
```

既然能够通过 JavaScript 表示 DOM 树的信息，那么就可以通过使用 JavaScript 来构建 DOM 树。

然而光是构建 DOM 树，没什么卵用，我们需要将 JavaScript 构建的 DOM 树渲染到真实的 DOM 树上，用 JavaScript 表现一个 dom 一个节点非常简单，我们只需要记录他的节点类型，属性键值对，子节点:

```
function Element(tagName, props, children) {
    this.tagName = tagName
    this.props = props
    this.children = children
}
```

那么 ul 标签我们就可以使用这种方式来表示

```
var ul = new Element('ul', {id: 'list'}, [
    {tagName: 'li', props: {class: 'item'}, children: ["Item 1"]},
    {tagName: 'li', props: {class: 'item'}, children: ["Item 2"]},
    {tagName: 'li', props: {class: 'item'}, children: ["Item 3"]}
])
```

说了这么多，他只是用 JavaScript 表示的一个结构，那该如何将他渲染到真实的 DOM 结构中呢：

```
Element.prototype.render = function() {
    let el = document.createElement(this.tagName), //  节点名称
        props = this.props //  节点属性

    for (var propName in props) {
        propValue = props[propName]
        el.setAttribute(propName, propValue)
    }
```

```
        this.children.forEach((child) => {
            var childEl = (child instanceof Element)
                ? child.render()
                : document.createTextNode(child)
            el.appendChild(childEl)
        })
        return el
}
```

如果我们想将 ul 渲染到 DOM 结构中，就只需要

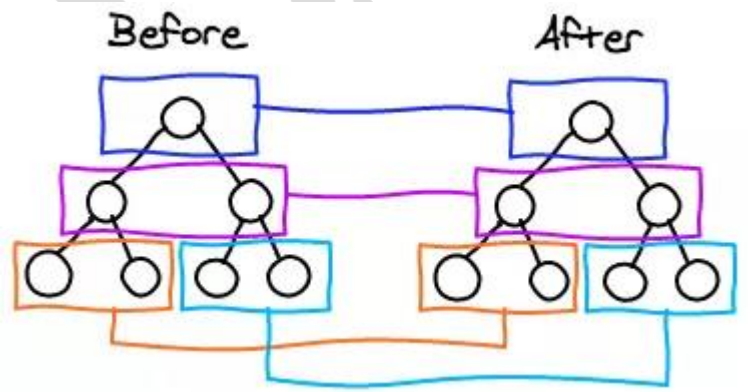
```
ulRoot = ul.render()
document.appendChild(ulRoot)
```

这样就完成了 ul 到 DOM 的渲染，也就有了真正的 DOM 结构

```
<ul id="list">
    <li class="item">Item 1</li>
    <li class="item">Item 2</li>
    <li class="item">Item 3</li>
</ul>
```

虚拟 DOM 的核心算法是 diff 算法(这里指的是优化后的算法)我们来看看 diff 算法是如何实现的：

diff 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。当发现节点不存在，则该节点和子节点会被完全删除，不会做进一步的比较。



在实际的代码中，会对新旧两棵树进行深度的遍历，给每一个节点进行标记。然后在新旧两棵树的对比中，将不同的地方记录下来。

```
// diff 算法，对比两棵树
```

```
function diff(oldTree, newTree) {
    var index = 0     // 当前节点的标志
    var patches = {} // 记录每个节点差异的地方
    dfsWalk(oldTree, newTree, index, patches)
    return patches
}
function dfsWalk(oldNode, newNode, index, patches) {
    // 对比 newNode 和 oldNode 的差异地方进行记录
    patches[index] = [...]

    diffChildren(oldNode.children, newNode.children, index, patches)
}
function diffChildren(oldChildren, newChildren, index, patches) {
    let leftNode = null
    var currentNodeIndex = index
    oldChildren.forEach((child, i) => {
        var newChild = newChildren[i]
        currentNodeIndex =    (leftNode && leftNode.count) // 计算节点的
标记
                ? currentNodeIndex + leftNode.count + 1
                : currentNodeIndex + 1
        dfsWalk(child, newChild, currentNodeIndex, patches) // 遍历子节
点
        leftNode = child
    })
}
```

同理，有 p 是 patches[1], ul 是 patches[3],以此类推

patches 指的是差异变化，这些差异包括：

1、节点类型的不同，

2、节点类型相同，但是属性值不同，文本内容不同。

所以有这么几种类型：

```
var REPLACE = 0,      // replace  替换
    REORDER = 1,      // reorder  父节点中子节点的操作
    PROPS   = 2,      // props  属性的变化
    TEXT    = 3       // text  文本内容的变化
```

如果节点类型不同，就说明是需要替换，例如将 div 替换成 section,就记录

下差异：

```
patches[0] = [{
    type: REPLACE,
    node: newNode // section
},{
    type: PROPS,
    props: {
        id: 'container'
    }
}]
```

在标题二中构建了真正的 DOM 树的信息，所以先对那一棵 DOM 树进行深度优先的遍历，遍历的时候同
patches 对象进行对比，找到其中的差异，然后应用到 DOM 操作中。

```
function patch(node, patches) {
    var walker = {index: 0} // 记录当前节点的标志
    dfsWalk(node, walker, patches)
}

function dfsWalk(node, walker, patches) {
    var currentPatches = patches[walker.index] // 这是当前节点的差异

    var len = node.childNodes
        ? node.childNodes.length
        : 0

    for (var i = 0; i < len; i++) { // 深度遍历子节点
        var child = node.childNodes[i]
        walker.index++
        dfsWalk(child, walker, patches)
    }

    if (currentPatches) {
        applyPatches(node, currentPatches) // 对当前节点进行 DOM 操作
    }
}
// 将差异的部分应用到 DOM 中
function applyPatches(node, currentPatches) {
```

```
    currentPatches.forEach((currentPatch) => {
        switch (currentPatch.type) {
            case REPLACE:
                var newNode = (typeof currentPatch.node === 'string')
                    ? document.createTextNode(currentPatch.node)
                    : currentPatch.node.render()
                node.parentNode.replaceChild(newNode, node)
                break;
            case REORDER:
                reorderChildren(node, currentPatch.moves)
                break
            case PROPS:
                setProps(node, currentPatch.props)
                break
            case TEXT:
                if (node.textContent) {
                    node.textContent = currentPatch.content
                } else {
                    node.nodeValue = currentPatch.content
                }
                break
            default:
                throw new Error('Unknown patch type ' + currentPatch.type)
        }
    })
}
```

这次的粗糙的 virtual-dom 基本已经实现了，具体的情况更加复杂。但这已经足够让我们理解 virtual-dom。