深度剖析Virtual DOM算法

电子文娱事业群-生活旅行业务部-产品研发部-国际机票研发部

卢 文

2019-01-09

京东

目录

CONTENTS

1、前端应用状态管理的思考

2、Virtual DOM算法

3、构造一个简版ReactJs

京东

PART 1

前端应用状态管理的思考

京东

场景

随着应用的复杂会导致以下问题

1. 在JS中维护的字段越来越多。
2. 需要监听的事件越来越多。
3. 应用程序会变得非常难维护。

您肯定能想出好几种实现方式

最容易想到的可能是：

```
var sortKey="";//排序字段
var sortType="";//排序方式
var filter="";//过滤类型
var data=[{...},{...},...];//航班数据
```

监听事件，根据字段用JS操作DOM，达到更新页面的排序状态、筛选状态和列表数据的目的。

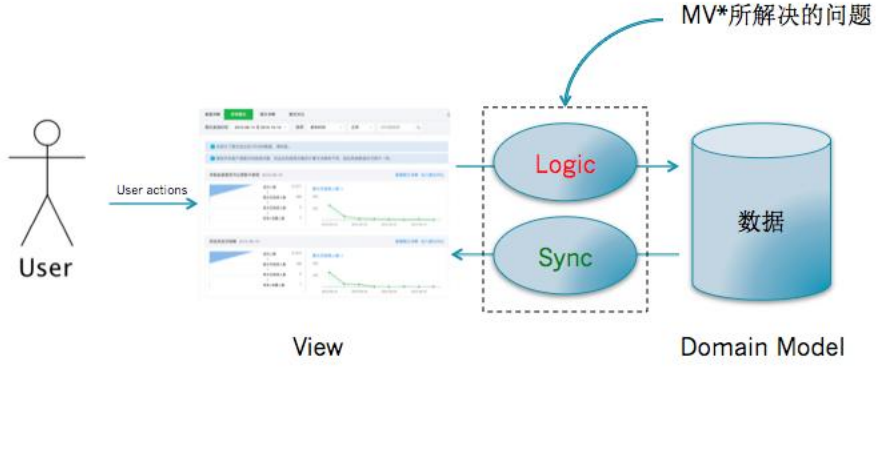
京东

架构模式

面对应用程序的复杂性管理问题，产生了应用架构模式

希望从*代码组织方式*上来降低维护复杂应用程序的难度。

1. MVC
2. MVP
3. MVVM



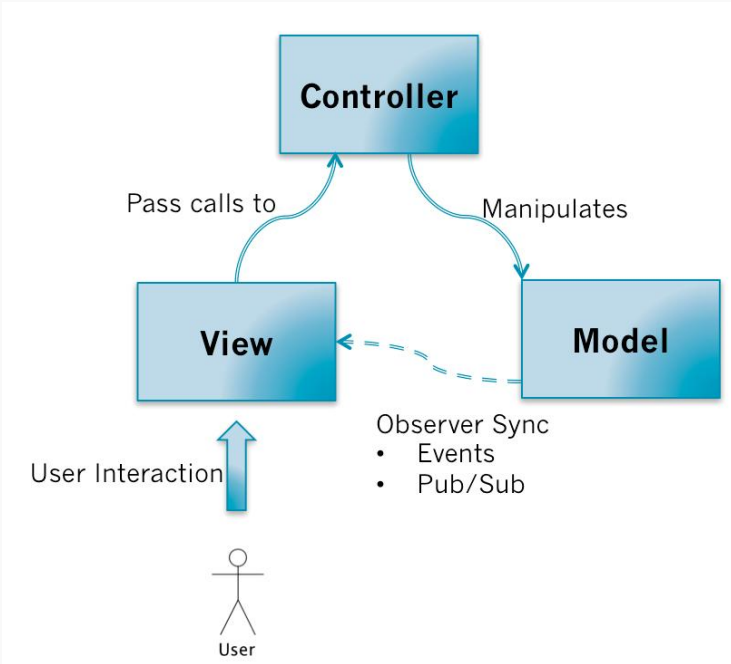
（一）MVC

Controller测试困难

视图同步操作是由View自己执行，而View只能在有UI的环境下运行。在**无UI环境的情况**，对Controller进行单元测试的时候，应用逻辑正确性是无法验证，**即**Model更新的时候，无法对View的更新操作进行断言。

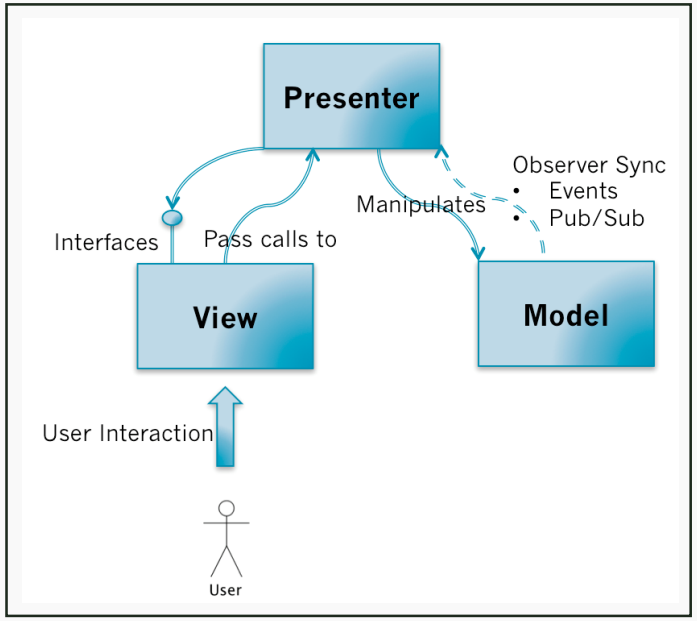
View无法组件化

View是强依赖特定Model的，如果需要把这个View抽出来作为可复用的组件就困难了。因为不同程序的Domain Model是不一样的。



（二）MVP

Presenter中除了应用逻辑以外，还有大量的View->Model，Model->View的手动同步逻辑，造成Presenter比较笨重，维护起来会比较困难。



思考

既然状态改变要操作相应的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。这就是后来的 MVVM 模式，只要在模版中声明视图组件是与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。

MVC、MVP没办法减少您所需要维护的状态，也没有降低状态更新时，您所需要对页面进行的更新操作，您需要操作的DOM还是需要操作，只是换了个地方。

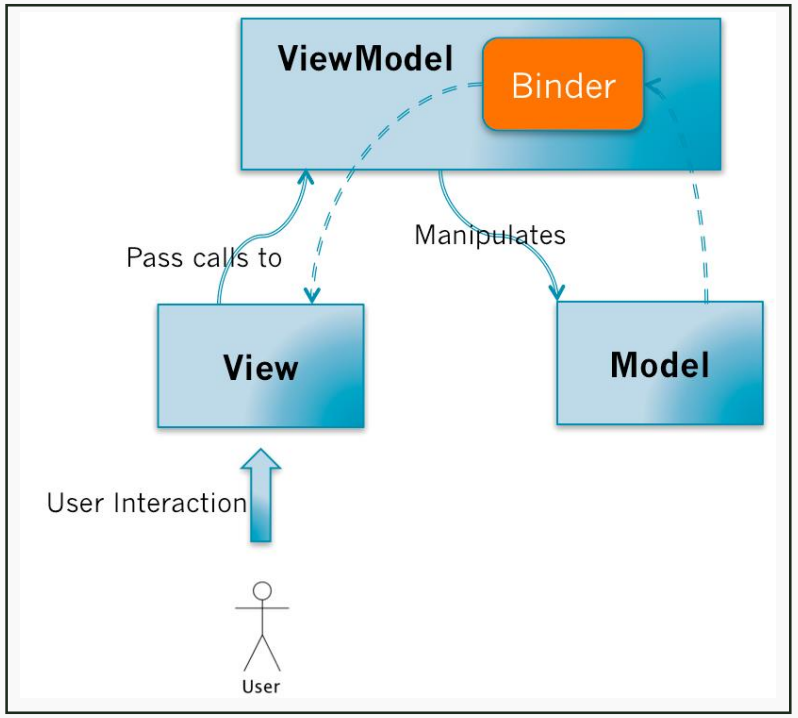
京东

（三）MVVM

MVVM代表的是Model-View-ViewModel，可以看作是一种特殊的MVP（Passive View）模式，或者说是对MVP模式的一种改良。

ViewModel的含义就是“Model of View”，视图的模型，包含了领域模型（Domain Model）和视图的状态（State）。

1. 过于简单的图形界面不适用。
2. 对于大型的图形应用程序，视图状态较多，ViewModel的构建和维护的成本都会比较高。
3. 数据绑定的声明是指令式地写在View的模版当中的，这些内容是没办法去打断点debug的。



总结

维护状态，更新视图。

MVC->MVP->MVVM，就是一个打怪升级的过程。后者解决了前者遗留的问题，把前者的缺点优化成了优点。

思考

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），但这是唯一的办法吗？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，也是能大大降低视图更新的操作的。

就像上面的航班列表，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。

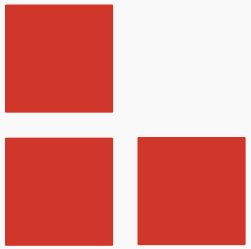
最大的问题就是这样做会很慢，因为即使一个小小的状态变化都需要重新构造整棵 DOM，性价比太低。

加一些特别的步骤来避免整棵 DOM 树的变更，可行吗？

京东

PART 2

Virtual DOM算法



京东

DOM

DOM是很慢的，一个简单的div的属性，如下：

```
var divEl = document.createElement('div');
var propsStr = '';
for(var key in divEl){
    propsStr += key + ',';
}
console.log(propsStr);
```

```
align,title,lang,translate,dir,dataset,hidden,tabIndex,accessKey,draggable,spellcheck,autocapitalize,contentEditable,isContentEditable,inputMode,offsetParent,offsetTop,offsetLeft,offsetWidth,offsetHeight,style,innerTex VM518:6
t,outerText,oncopy,oncut,onpaste,onabort,onblur,oncancel,oncanplay,oncanplaythrough,onchange,onclick,onclose,oncontextmenu,oncuechange,ondblclick,ondrag,ondragend,ondragenter,ondragleave,ondragover,ondragstart,ondrop,onduratio
nchange,onemptied,onended,onerror,onfocus,oninput,oninvalid,onkeydown,onkeypress,onkeyup,onload,onloadeddata,onloadedmetadata,onloadstart,onmousedown,onmouseenter,onmouseleave,onmousemove,onmouseout,onmouseover,onmouseup,onmou
sewheel,onpause,onplay,onplaying,onprogress,onratechange,onreset,onresize,onscroll,onseeked,onseeking,onselect,onstalled,onsubmit,onsuspend,ontimeupdate,ontoggle,onvolumechange,onwaiting,onwheel,onauxclick,ongotpointercapture,
onlostpointercapture,onpointerdown,onpointermove,onpointerup,onpointercancel,onpointerover,onpointerout,onpointerenter,onpointerleave,onselectstart,onselectionchange,nonce,click,focus,blur,ontouchcancel,ontouchend,ontouchmove,
ontouchstart,namespaceURI,prefix,localName,tagName,id,className,classList,slot,attributes,shadowRoot,assignedSlot,innerHTML,outerHTML,scrollTop,scrollLeft,scrollWidth,scrollHeight,clientTop,clientLeft,clientWidth,clientHeight,
attributeStyleMap,onbeforecopy,onbeforecut,onbeforepaste,onsearch,previousElementSibling,nextElementSibling,children,firstElementChild,lastElementChild,childElementCount,onwebkitfullscreenchange,onwebkitfullscreenerror,setPoin
terCapture,releasePointerCapture,hasPointerCapture,hasAttributes,getAttributeNames,getAttribute,getAttributeNS,setAttribute,setAttributeNS,removeAttribute,removeAttributeNS,hasAttribute,hasAttributeNS,toggleAttribute,getAttrib
uteNode,getAttributeNodeNS,setAttributeNode,setAttributeNodeNS,removeAttributeNode,closest,matches,webkitMatchesSelector,attachShadow,getElementsByTagName,getElementsByTagNameNS,getElementsByClassName,insertAdjacentElement,ins
ertAdjacentText,insertAdjacentHTML,requestPointerLock,getClientRects,getBoundingClientRect,scrollIntoView,scrollIntoViewIfNeeded,animate,computedStyleMap,before,after,replaceWith,remove,prepend,append,querySelector,querySelect
orAll,webkitRequestFullScreen,webkitRequestFullscreen,onfullscreenchange,onfullscreenerror,scroll,scrollTo,scrollBy,createShadowRoot,getDestinationInsertionPoints,requestFullscreen,ELEMENT_NODE,ATTRIBUTE_NODE,TEXT_NODE,CDATA_S
ECTION_NODE,ENTITY_REFERENCE_NODE,ENTITY_NODE,PROCESSING_INSTRUCTION_NODE,COMMENT_NODE,DOCUMENT_NODE,DOCUMENT_TYPE_NODE,DOCUMENT_FRAGMENT_NODE,NOTATION_NODE,DOCUMENT_POSITION_DISCONNECTED,DOCUMENT_POSITION_PRECEDING,DOCUMENT_P
OSITION_FOLLOWING,DOCUMENT_POSITION_CONTAINS,DOCUMENT_POSITION_CONTAINED_BY,DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC,nodeType,nodeName,baseURI,isConnected,ownerDocument,parentNode,parentElement,childNodes,firstChild,lastChild
,previousSibling,nextSibling,nodeValue,textContent,hasChildNodes,getRootNode,normalize,cloneNode,isEqualNode,isSameNode,compareDocumentPosition,contains,lookupPrefix,lookupNamespaceURI,isDefaultNamespace,insertBefore,appendChi
ld,replaceChild,removeChild,addEventListener,removeEventListener,dispatchEvent,
```

相对于 DOM 对象，原生的 JavaScript 对象处理起来更快，而且更简单。DOM 树的结构、属性都可以很容易地用 JavaScript 对象表示出来。

HTML写法

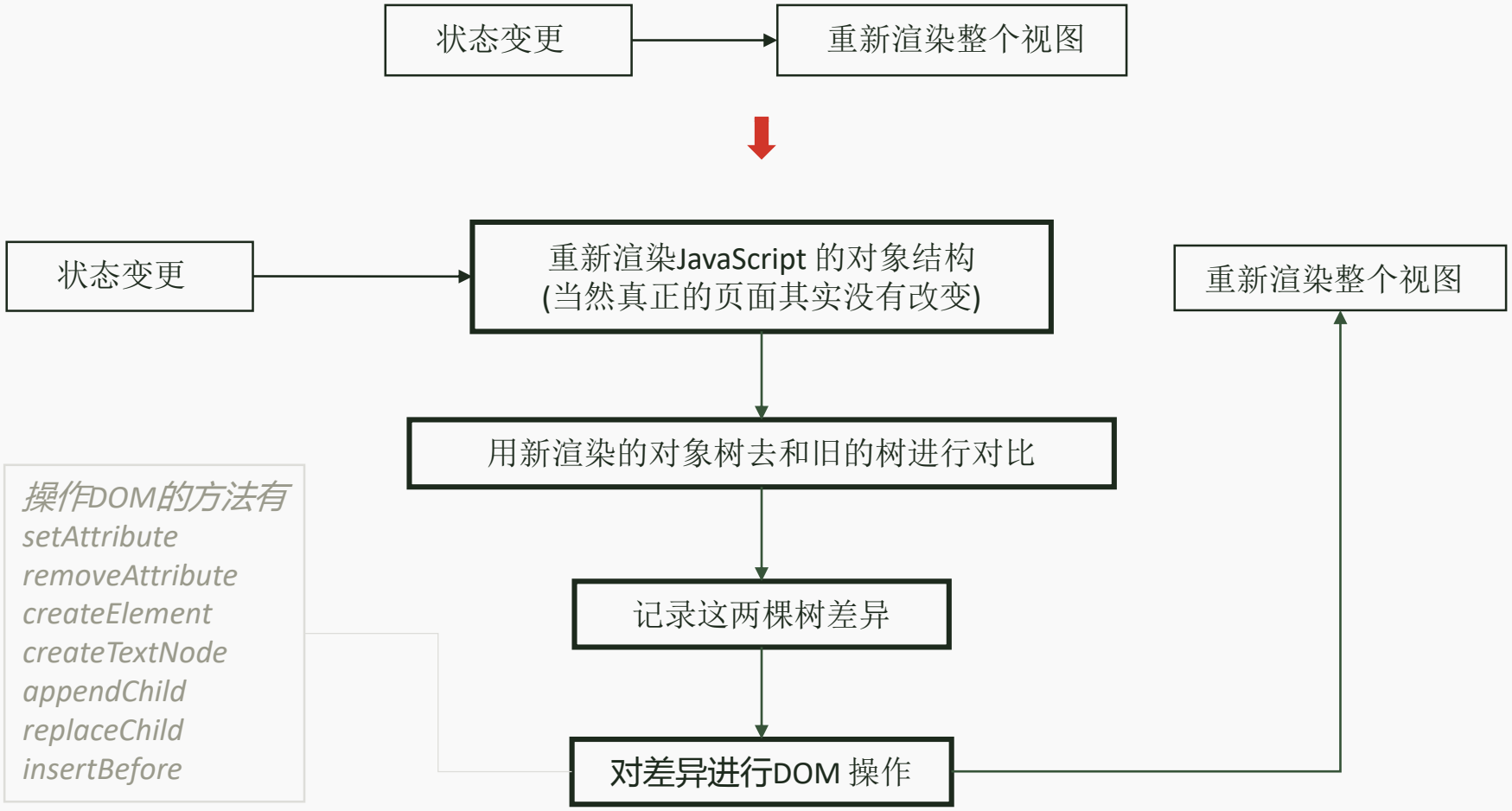
```
<ul>
  <li>
    <div class="time">
      <span>07:25</span>
      <span>-</span>
      <span>20:55</span>
    </div>
    <div class="airport">
      <span>浦东机场</span>
      <span>-</span>
      <span>东京羽田机场</span>
    </div>
    <div class="flight-price">
      <span class="tag">¥</span>
      <span class="price">1589</span>
    </div>
  </li>
</ul>
```

JavaScript对象

```
var element = {
  tagName: 'ul', //节点标签名
  props: {}, //DOM的属性，存储键值对的对象
  children: [ //该节点的子节点
    {
      tagName: 'li', props: {}, children: [
        {
          tagName: 'div', props: { class: 'time' }, children: [
            { tagName: 'span', props: {}, children: ['07:25'] },
            { tagName: 'span', props: {}, children: ['-'] },
            { tagName: 'span', props: {}, children: ['20:55'] }
          ]
        }, {
          tagName: 'div', props: { class: 'airport' }, children: [
            { tagName: 'span', props: {}, children: ['浦东机场'] },
            { tagName: 'span', props: {}, children: ['-'] },
            { tagName: 'span', props: {}, children: ['东京羽田机场'] }
          ]
        }, {
          tagName: 'div', props: { class: 'flight-price' }, children: [
            { tagName: 'span', props: { class: 'tag' }, children: ['¥'] },
            { tagName: 'span', props: { class: 'price' }, children: ['1589'] }
          ]
        }
      ]
    }
  ]
}
```

京东

渲染过程


状态变更
重新渲染整个视图
状态变更
重新渲染JavaScript 的对象结构
(当然真正的页面其实没有改变)
重新渲染整个视图
用新渲染的对象树去和旧的树进行对比
操作DOM的方法有
setAttribute
removeAttribute
createElement
createTextNode
appendChild
replaceChild
insertBefore
记录这两棵树差异
对差异进行DOM 操作

本质

Virtual DOM 本质上就是在 JS 和 DOM 之间做了一个缓存。

CPU（JS）只操作内存（Virtual DOM），最后再把变更写入硬盘（DOM）。

算法实现

1. 用JavaScript对象表示 DOM 树的结构。
2. 以这个表示DOM树结构的JavaScript对象构建一个真正的 DOM 树，插入到HTML文档当中。
3. 当状态变更的时候，重新构造一棵新的 JavaScript对象树，用新的JavaScript对象树和旧的JavaScript对象树进行比较，记录两棵树之间的差异。
4. 把步骤3所记录的差异应用到步骤1所构建的真正的DOM树上，视图也就更新了。

（一）用JavaScript对象表示DOM树

```
/**
 * 渲染 Element tree
 */
Element.prototype.render = function () {
    console.log('--render--',this.tagName);
    var el = document.createElement(this.tagName) //根据tagName构建
    var props = this.props

    for (var propName in props) { //设置节点的DOM属性
        var propValue = props[propName]
        _.setAttr(el, propName, propValue)
    }

    _.each(this.children, function (child) {
        var childEl = (child instanceof Element)
            ? child.render() //如果子节点也是虚拟DOM，递归构建DOM节点
            : document.createTextNode(child) //如果字符串，只构建文本节点
        el.appendChild(childEl)
    })

    return el
}
```

```
/**
 * 虚拟DOM元素
 * @param {String} tagName
 * @param {Object} props - {}
 * @param {Array<Element|String>} - 子元素
 */
function Element(tagName, props, children) {
    if (!(this instanceof Element)) {
        if (!_.isArray(children) && children != null) {
            children = _.slice(arguments, 2).filter(_.truthy)
        }
        return new Element(tagName, props, children)
    }

    //做一个参数兼容处理
    if (_.isArray(props)) {
        children = props
        props = {}
    }

    this.tagName = tagName
    this.props = props || {}
    this.children = children || []
    this.key = props
        ? props.key
        : void 0

    var count = 0

    _.each(this.children, function (child, i) {
        if (child instanceof Element) {
            count += child.count
        } else {
            children[i] = '' + child
        }
        count++
    })

    this.count = count
}
```

京东

（二）用JavaScript对象构建真正的DOM树

```
var tree = renderTree()
var root = tree.render()
document.body.appendChild(root)
```



```
for (var i = 0; i < flights.length; i++) {
  items.push(el('li', [
    el('div', { class: 'time' }, [
      el('span', [flights[i].timeDep]),
      el('span', ['-']),
      el('span', [flights[i].timeArr])
    ]),
    el('div', { class: 'airport' }, [
      el('span', [flights[i].airportDep]),
      el('span', ['-']),
      el('span', [flights[i].airportArr])
    ]),
    el('div', { class: 'flight-price' }, [
      el('span', { class: 'tag' }, '¥'),
      el('span', { class: 'price' }, flights[i].price)
    ])
  ]))
}

return el('div', { 'class': 'container' }, [
  el('h1', { style: 'color: ' + color }, data.trip),
  el('ul', items)
])
```



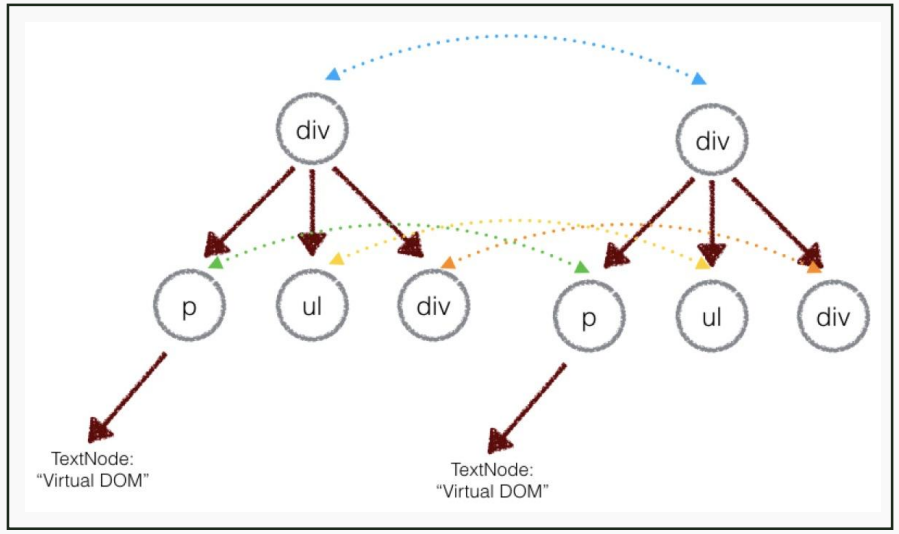
```
<div class="container">
  <ul>
    <li>
      <div class="time">
        <span>07:25</span>
        <span>-</span>
        <span>20:55</span>
      </div>
      <div class="airport">
        <span>浦东机场</span>
        <span>-</span>
        <span>东京羽田机场</span>
      </div>
      <div class="flight-price">
        <span class="tag">¥</span>
        <span class="price">1589</span>
      </div>
    </li>
  </ul>
</div>
```

京东

（三）比较两棵虚拟DOM树的差异

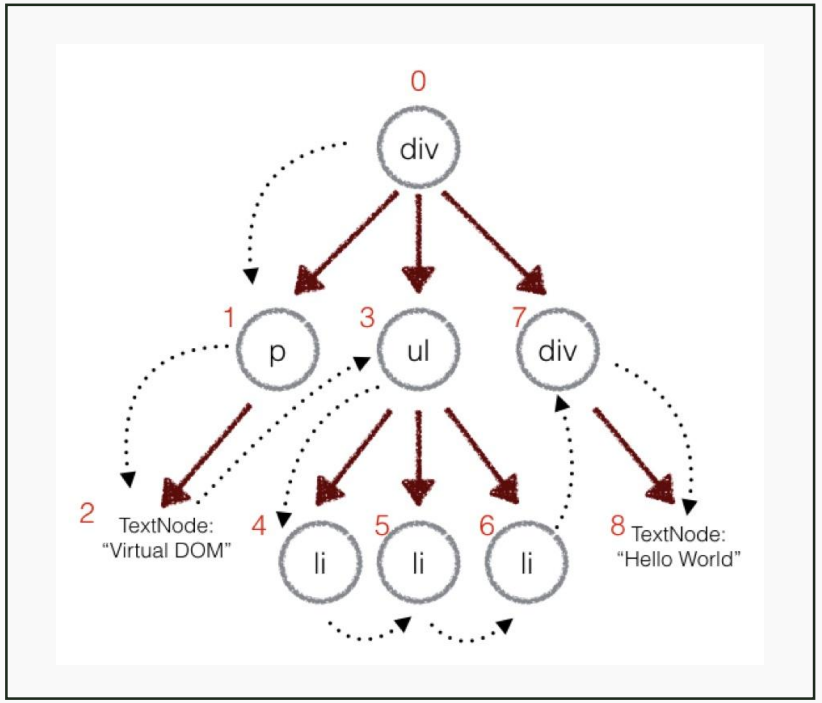
两颗完整树的diff算法复杂度O(n^3)。

同一层级的对比，算法复杂度O(n)。



深度优先遍历，记录差异

```
/**
 * 对两棵树进行深度优先遍历，Depth_Fisrst Search
 * @param {Element} oldTree
 * @param {Element} newTree
 * @param {Number} index - 当前节点的标志
 * @param {Array} patches - 用来记录每个节点差异的对象
 */
function dfsWalk(oldNode, newNode, index, patches) {
    var currentPatch = [] //数组用于存储新旧节点的不同

    //Node被移除
    if (newNode === null) {
        //当执行REORDER时，真实的DOM节点将别移除，这里无须做任何处理
        //TextNode内容替换
    } else if (_.isString(oldNode) && _.isString(newNode)) {
        if (newNode !== oldNode) {
            //记录文本节点的变化
            currentPatch.push({ type: patch.TEXT, content: newNode })
        }
        //节点名称相同，对比老节点的props和children
    } else if (
        oldNode.tagName === newNode.tagName &&
        oldNode.key === newNode.key
    ) {
        // Diff props
        var propsPatches = diffProps(oldNode, newNode)
        if (propsPatches) {
            //记录属于的变化
            currentPatch.push({ type: patch.PROPS, props: propsPatches })
        }
        //对比children，若节点包含ignore属性，则不进行对比
        if (!isIgnoreChildren(newNode)) {
            diffChildren(
                oldNode.children,
                newNode.children,
                index,
                patches,
                currentPatch
            )
        }
        //节点名称不同，用新节点替换老节点
    } else {
        //记录节点的变化
        currentPatch.push({ type: patch.REPLACE, node: newNode })
    }

    if (currentPatch.length) {
        patches[index] = currentPatch
    }
}
```



差异类型

```
//差异类型
var REPLACE = 0 //替换掉原来的节点
var REORDER = 1 //移动、删除、新增子节点
var PROPS = 2 //修改了节点的属性
var TEXT = 3 //对于文本节点，文本内容发生改变
```

```
//记录文本节点的变化
currentPatch.push({ type: patch.TEXT, content: newNode })
```

```
//记录节点的变化
currentPatch.push({ type: patch.REPLACE, node: newNode })
```

```
//记录属性的变化
currentPatch.push({ type: patch.PROPS, props: propsPatches })
```

```
//记录重新排序
var reorderPatch = { type: patch.REORDER, moves: diffs.moves }
```

重新排序

这个问题抽象出来其实是字符串的最小编辑距离（Edit Distance），又称Levenshtein距离，是指两个**字符**串之间，由一个转成另一个所需的最少编辑操作次数。许可的编辑操作包括将一个字符替换成另一个字符、插入一个字符、删除一个字符。一般来说，编辑距离越小，两个串的相似度越大。

$$edit[i][j]=\begin{cases}0 & i=0,j=0\\ j & i=0,j>0\\ i & i>0,j=0\\ \min(edit[i-1][j]+1,edit[i][j-1]+1,edit[i-1][j-1]+flag) & i>0,j>0\end{cases}$$

$$flag=\begin{cases}0 & A[i]=B[j]\\ 1 & A[i]\neq B[j]\end{cases}$$

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

通过动态规划求解，时间复杂度为O(M * N)。但是我们并不需要真的达到最小的操作，我们只需要优化一些比较常见的移动情况，牺牲一定DOM操作，让算法时间复杂度达到线性的（O(max(M, N))。

京东

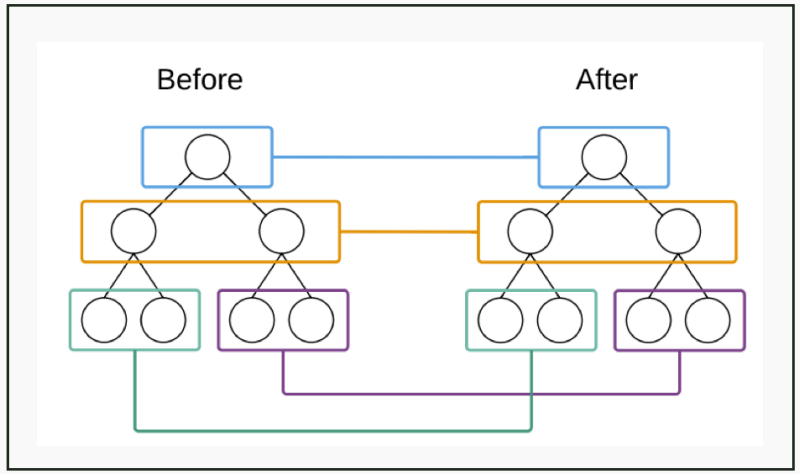
（四）把差异应用到真正的DOM树上

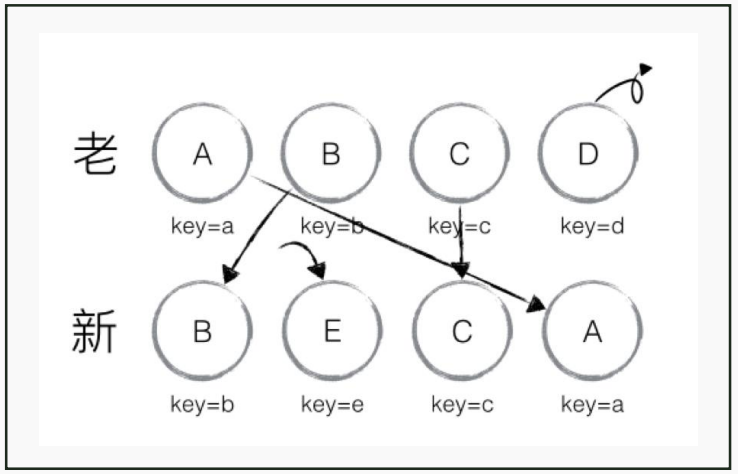
```
/**
 * 对DOM树进行深度优先的遍历，遍历的时候从patches对象中找出当前遍历的节点差异，然后进行DOM操作
 * @param {Object} node - DOM Object
 * @param {Object} walker
 * @param {Array} patches
 */
function dfsWalk(node, walker, patches) {
    var currentPatches = patches[walker.index]
    var len = node.childNodes
        ? node.childNodes.length
        : 0
    for (var i = 0; i < len; i++) {
        var child = node.childNodes[i]
        walker.index++
        dfsWalk(child, walker, patches)
    }

    if (currentPatches) {
        applyPatches(node, currentPatches)
    }
}
```

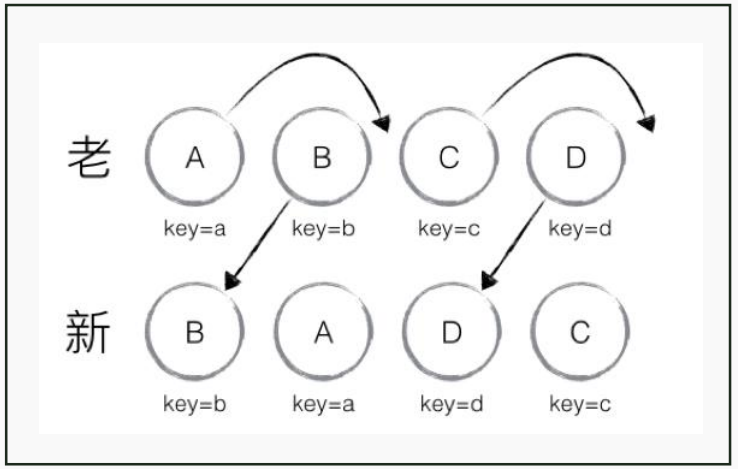
```
/**
 * 根据不同类型的差异对当前节点进行DOM操作
 * @param {Object} node - DOM Object
 * @param {Array} currentPatches
 */
function applyPatches(node, currentPatches) {
    _.each(currentPatches, function (currentPatch) {
        switch (currentPatch.type) {
            case REPLACE:
                var newNode = (typeof 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 {
                    //IE兼容性处理
                    node.nodeValue = currentPatch.content
                }
                break
            default:
                throw new Error('Unknown patch type ' + currentPatch.type)
        }
    })
}
```

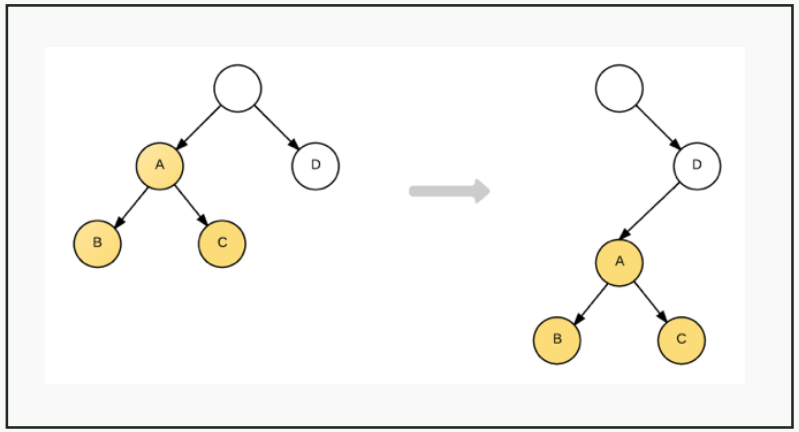
```
var newTree = renderTree()
var patches = diff(tree, newTree)
patch(root, patches)
```

京东


Before
After


老
A
B
C
D
key=a
key=b
key=c
key=d
新
B
E
C
A
key=b
key=e
key=c
key=a


老
A
B
C
D
key=a
key=b
key=c
key=d
新
B
A
D
C
key=b
key=a
key=d
key=c


A
D
B
C
D
A
B
C

思考

1. DOM节点跨层级的移动操作发生频率很低，是次要矛盾。
2. 拥有相同类的两个组件将会生成相似的树形结构，拥有不同类的两个组件将会生成不同的树形结构，这里也是抓前者放后者的思想。
3. 对于同一层级的一组子节点，通过唯一key进行区分。
4. 基于各自的策略，进行了算法优化，来保证整体界面构建的性能。

PART 3

构造一个简版ReactJs

京东

DEMO

京东

Thanks!

京东