



前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈

来自 Lucas ... · 盐选专栏

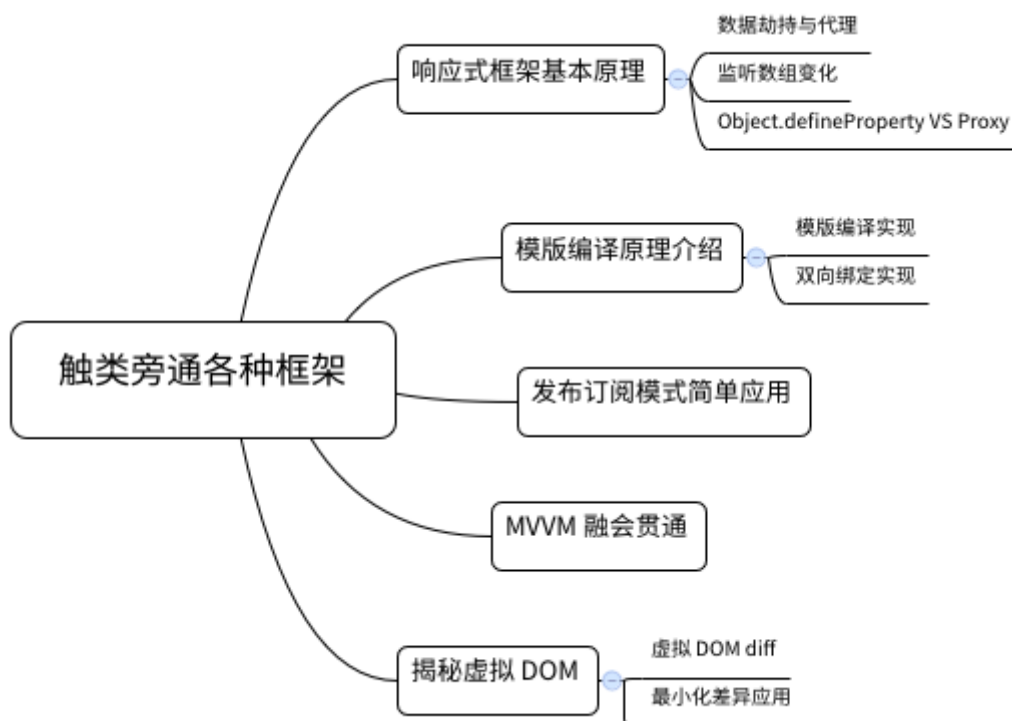
[查看详情 >](#)

触类旁通各种框架

框架在任何一种语言编程范畴中都扮演了举足轻重的地位，前端尤其是如此。目前流行的前端框架三驾马车：Angular、React 和 Vue，它们各有特点和受众，都值得开发者认真思考和学习。那么我们在精力有限的情况下，如何做到「触类旁通」、如何提取框架共性、提高学习和应用效率呢？

我们这一讲就来剖析这些框架的特点和本质，介绍如何学习并使用这些框架，进而了解前端框架的真谛。

相关知识点如下：



图片

我把现代框架的关键词进行提炼，掌握这些关键词，是我们学习的重要环节。这些关键词有：双向绑定、依赖收集、发布订阅模式、MVVM / MVC、虚拟 DOM、虚拟 DOM diff、模版编译等。

响应式框架基本原理

我们不再赘述响应式或数据双向绑定的基本概念，这里直接思考其行为：直观上，数据在变化时，不再需要开发者去手动更新视图，而视图会根据变化的数据「自动」进行更新。想完成这个过程，我们需要：

收集视图依赖了哪些数据

感知被依赖数据的变化

数据变化时，自动「通知」需要更新的视图部分，并进行更新

道理很简单，这个思考过程换成对应的技术概念就是：

依赖收集

数据劫持 / 数据代理

发布订阅模式

接下来，我们一步步拆解。

数据劫持与代理

感知数据变化的方法很直接，就是进行数据劫持或数据代理。我们往往通过 `Object.defineProperty` 实现。这个方法可以定义数据的 `getter` 和 `setter`，具体用法不再赘述。下面来看一个场景：

```
let data = {  
  stage: 'GitChat',
```

```
course: {
  title: '前端开发进阶',
  author: 'Lucas',
  publishTime: '2018 年 5 月'
}

Object.keys(data).forEach(key => {
  let currentValue = data[key]

  Object.defineProperty(data, key, {
    enumerable: true,
    configurable: false,
    get() {
      console.log(`getting ${key} value now, getting
value is`, currentValue)
      return currentValue
    },
    set(newValue) {
      currentValue = newValue
      console.log(`setting ${key} value now, setting
value is`, currentValue)
    }
  })
})
```

这段代码对 data 数据的 getter 和 setter 进行定义拦截，当我们读取或者改变 data 的值时：

```
data.course
```

```
// getting course value now, getting value is: {title:
"前端开发进阶", author: "Lucas", publishTime: "2018 年 5
月"}
```

```
data.course = '前端开发进阶 2'
```

```
// setting course value now, setting value is 前端开发进阶
```

但是这种实现有一个问题，例如：

```
data.course.title = '前端开发进阶 2'

// getting course value now, getting value is: {title:
"前端开发进阶", author: "Lucas", publishTime: "2018 年 5
月"}
```

只会有 getting course value now, getting value is: {title: "前端开发进阶", author: "Lucas", publishTime: "2018 年 5 月"} 的输出，这是因为我们尝试读取了 data.course 信息。但是修改 data.course.title 的信息并没有打印出来。

出现这个问题的原因是因为我们的实现代码只进行了一层 Object.defineProperty，或者说只对 data 的第一层属性进行了 Object.defineProperty，对于嵌套的引用类型数据结构：data.course，我们同样应该进行拦截。

为了达到深层拦截的目的，将 Object.defineProperty 的逻辑抽象为 observe 函数，并改用递归实现：

```
let data = {
  stage: 'GitChat',
  course: {
    title: '前端开发进阶',
    author: 'Lucas',
    publishTime: '2018 年 5 月'
  }
}

const observe = data => {
  if (!data || typeof data !== 'object') {
    return
  }
  Object.keys(data).forEach(key => {
```

```
let currentValue = data[key]

observe(currentValue)

Object.defineProperty(data, key, {
  enumerable: true,
  configurable: false,
  get() {
    console.log(`getting ${key} value now, getting
value is`, currentValue)
    return currentValue
  },
  set(newValue) {
    currentValue = newValue
    console.log(`setting ${key} value now, setting
value is`, currentValue)
  }
})
})
}

observe(data)
```

这样一来，就实现了深层数据拦截：

```
data.course.title = '前端开发进阶 2'

// getting course value now, getting value is: {// ...}
// setting title value now, setting value is 前端开发进阶 2
```

请注意，我们在 set 代理中，并没有对 newValue 再次递归进行 observe(newValue)。也就是说，如果赋值是一个引用类型：

```
data.course.title = {
  title: '前端开发进阶 2'
}
```

无法实现对 `data.course.title` 数据的观察。这里为了简化学习成本，默认修改的数值符合语义，都是基本类型。

在尝试对 `data.course.title` 赋值时，首先会读取 `data.course`，因此输出：
`getting course value now, getting value is: { // ... }`，赋值后，触发 `data.course.title` 的 setter，输出：`setting title value now, setting value is 前端开发进阶 2`。

因此我们总结出：对数据进行拦截并不复杂，这也是很多框架实现的第一步。

监听数组变化

如果上述数据中某一项变为数组：

```
let data = {
  stage: 'GitChat',
  course: {
    title: '前端开发进阶',
    author: ['Lucas', 'Ronaldo'],
    publishTime: '2018 年 5 月'
  }
}

const observe = data => {
  if (!data || typeof data !== 'object') {
    return
  }
  Object.keys(data).forEach(key => {
    let currentValue = data[key]

    observe(currentValue)

    Object.defineProperty(data, key, {
      enumerable: true,
      configurable: false,
      get() {
        console.log(`getting ${key} value now, getting`
```

```

    value is`, currentValue)
      return currentValue
    },
    set(newValue) {
      currentValue = newValue
      console.log(`setting ${key} value now, setting
value is`, currentValue)
    }
  })
})
}

observe(data)

```

```

data.course.author.push('Messi')
// getting course value now, getting value is: {//...}
// getting author value now, getting value is: (2)
[(...), (...)]

```

我们只监听到了 `data.course` 以及 `data.course.author` 的读取，而数组 `push` 行为并没有被拦截。这是因为 `Array.prototype` 上挂载的方法并不能触发 `data.course.author` 属性值的 setter，由于这并不属于做赋值操作，而是 `push` API 调用操作。然而对于框架实现来说，这显然是不满足要求的，当数组变化时我们应该也有所感知。

Vue 同样存在这样的问题，它的解决方法是：将数组的常用方法进行重写，进而覆盖掉原生的数组方法，重写之后的数组方法需要能够被拦截。

实现逻辑如下：

```

const arrExtend = Object.create(Array.prototype)
const arrMethods = [
  'push',
  'pop',
  'shift',

```

```
'unshift',
'splice',
'sort',
'reverse'
]

arrMethods.forEach(method => {
  const oldMethod = Array.prototype[method]
  const newMethod = function(...args) {
    oldMethod.apply(this, args)
    console.log(`${method} 方法被执行了`)
  }
  arrExtend[method] = newMethod
})
```

对于数组原生的 7 个方法：

push

pop

shift

unshift

splice

sort

reverse

进行重写，核心操作还是调用原生方法：oldMethod.apply(this, args)，除此之外可以在调用 oldMethod.apply(this, args) 前后加入我们需要的任何逻辑。示例代码中加入了一行 console.log。使用时：


```
Array.prototype = Object.assign(Array.prototype,
arrExtend)
```

```
let array = [1, 2, 3]
array.push(4)
// push 方法被执行了
```

对应我们的代码：

```
const arrExtend = Object.create(Array.prototype)
const arrMethods = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]

arrMethods.forEach(method => {
  const oldMethod = Array.prototype[method]
  const newMethod = function(...args) {
    oldMethod.apply(this, args)
    console.log(`${method} 方法被执行了`)
  }
  arrExtend[method] = newMethod
})

Array.prototype = Object.assign(Array.prototype,
arrExtend)
```

```
let data = {
  stage: 'GitChat',
  course: {
    title: '前端开发进阶',
```

```
    author: ['Lucas', 'Ronaldo'],
    publishTime: '2018 年 5 月'
  }
}

const observe = data => {
  if (!data || typeof data !== 'object') {
    return
  }
  Object.keys(data).forEach(key => {
    let currentValue = data[key]

    observe(currentValue)

    Object.defineProperty(data, key, {
      enumerable: true,
      configurable: false,
      get() {
        console.log(`getting ${key} value now, getting
value is`, currentValue)
        return currentValue
      },
      set(newValue) {
        currentValue = newValue
        console.log(`setting ${key} value now, setting
value is`, currentValue)
      }
    })
  })
}

observe(data)

data.course.author.push('Messi')
```

将会输出：

```
getting course value now, getting value is: {//...}  
  getting author value now, getting value is: (2) [(...),  
(...)]  
  // push 方法被执行了
```

这种 monkey patch 本质是重写原生方法，这天生不是很安全，也很不优雅，能有更好的实现吗？

答案是有的，使用 ES Next 的新特性——Proxy，之前也介绍过，它可以完成对数据的代理。

那么这两种方式有何区别呢？请继续阅读。

Object.defineProperty VS Proxy

我们首先尝试使用 Proxy 来完成代码重构：

```
let data = {  
  stage: 'GitChat',  
  course: {  
    title: '前端开发进阶',  
    author: ['Lucas'],  
    publishTime: '2018 年 5 月'  
  }  
}  
  
const observe = data => {  
  if (!data || Object.prototype.toString.call(data) !==  
    '[object Object]') {  
    return  
  }  
  
  Object.keys(data).forEach(key => {  
    let currentValue = data[key]  
    // 事实上 proxy 也可以对函数类型进行代理。这里只对承载数据类型  
    的 object 进行处理，读者了解即可。  
    if (typeof currentValue === 'object') {
```

```

    observe(currentValue)
    data[key] = new Proxy(currentValue, {
      set(target, property, value, receiver) {
        // 因为数组的 push 会引起 length 属性的变化，所以
        // push 之后会触发两次 set 操作，我们只需要保留一次即可，property 为
        // length 时，忽略
        if (property !== 'length') {
          console.log(`setting ${key} value now,
            setting value is`, currentValue)
        }
        return Reflect.set(target, property, value,
          receiver)
      }
    })
  }
}
else {
  Object.defineProperty(data, key, {
    enumerable: true,
    configurable: false,
    get() {
      console.log(`getting ${key} value now, getting
        value is:`, currentValue)
      return currentValue
    },
    set(newValue) {
      currentValue = newValue
      console.log(`setting ${key} value now, setting
        value is`, currentValue)
    }
  })
}
})
}

observe(data)

```

此时对数组进行操作：

```
data.course.author.push('messi')  
// setting author value now, setting value is ["Lucas"]
```

已经符合我们的需求了。注意这里在使用 Proxy 进行代理时，并没有对 getter 进行代理，因此上述代码的输出结果并不像之前使用 Object.defineProperty 那样也会有 getting value 输出。

整体实现并不难理解，需要读者了解最基本的 Proxy 知识。简单总结一下，对于数据键值为基本类型的情况，我们使用 Object.defineProperty；对于键值为对象类型的情况，继续递归调用 observe 方法，并通过 Proxy 返回的新对象对 data[key] 重新赋值，这个新值的 getter 和 setter 已经被添加了代理。

了解了 Proxy 实现之后，我们对 Proxy 实现数据代理和 Object.defineProperty 实现数据拦截进行对比，会发现：

Object.defineProperty 不能监听数组的变化，需要进行数组方法的重写

Object.defineProperty 必须遍历对象的每个属性，且对于嵌套结构需要深层遍历

Proxy 的代理是针对整个对象的，而不是对象的某个属性，因此不同于 Object.defineProperty 的必须遍历对象每个属性，Proxy 只需要做一层代理就可以监听同级结构下的所有属性变化，当然对于深层结构，递归还是需要进行的

Proxy 支持代理数组的变化

Proxy 的第二个参数除了 set 和 get 以外，可以有 13 种拦截方法，比起 Object.defineProperty() 更加强大，这里不再一一列举

Proxy 性能将会被底层持续优化，而 Object.defineProperty 已经不再是优化重点

模版编译原理介绍

到此，我们了解了如何监听数据的变化，那么下一步呢？以类 Vue 框架为例，我们看看一个典型的用法：

`{{stage}}` 平台课程: `{{course.title}}`

`{{course.title}}` 是 `{{course.author}}` 发布的课程

发布时间为 `{{course.publishTime}}`

```
let vue = new Vue({
  ele: '#app',
  data: {
    stage: 'GitChat',
    course: {
      title: '前端开发进阶',
      author: 'Lucas',
      publishTime: '2018 年 5 月'
    },
  },
})
```

其中模版变量使用了 `{{}}` 的表达方式输出模版变量。最终输出的 HTML 内容应该被合适的数据进行填充替换，因此还需要一步编译过程，该过程任何框架或类库中都是相通的，比如 React 中的 JSX，也是编译为 `React.createElement`，并在生成虚拟 DOM 时进行数据填充。

我们这里简化过程，将模版内容：

```
{{stage}} 平台课程: {{course.title}}
```

```
{{course.title}} 是 {{course.author}} 发布的课程
```

```
发布时间为 {{course.publishTime}}
```

输出为真实 HTML 即可。

模版编译实现

一提到这样的「模版编译」过程，很多开发者都会想到词法分析，也许都会感到头大。其实原理很简单，就是使用正则 + 遍历，有时也需要一些算法知识，我们来看现在的场景，只需要对 #app 节点下内容进行替换，通过正则识别出模版变量，获取对应的数据即可：

```
compile(document.querySelector('#app'), data)
```

```
function compile(el, data) {  
  let fragment = document.createDocumentFragment()  
  
  while (child = el.firstChild) {  
    fragment.appendChild(child)  
  }  
  
  // 对 el 里面的内容进行替换
```

```

function replace(fragment) {
  Array.from(fragment.childNodes).forEach(node => {
    let textContent = node.textContent
    let reg = /\{\{(.*)\}\}/g

    if (node.nodeType === 3 && reg.test(textContent)) {
      const nodeTextContent = node.textContent
      const replaceText = () => {
        node.textContent =
nodeTextContent.replace(reg, (matched, placeholder) => {
          return
placeholder.split('.').reduce((prev, key) => {
            return prev[key]
          }, data)
        })
      }

      replaceText()
    }

    // 如果还有子节点，继续递归 replace
    if (node.childNodes && node.childNodes.length) {
      replace(node)
    }
  })
}

replace(fragment)

el.appendChild(fragment)
return el
}

```

代码分析：我们使用 fragment 变量储存生成的真实 HTML 节点内容。通过 replace 方法对 {{变量}} 进行数据替换，同时 {{变量}} 的表达只会出现在 nodeType === 3 的文本类型节点中，因此对于符合 node.nodeType === 3

`&& reg.test(textContent)` 条件的情况，进行数据获取和填充。我们借助字符串 `replace` 方法第二个参数进行一次性替换，此时对于形如 `{{data.course.title}}` 的深层数据，通过 `reduce` 方法，获得正确的值。

因为 DOM 结构可能是多层的，所以对存在子节点的节点，依然使用递归进行 `replace` 替换。

这个编译过程比较简单，没有考虑到边界情况，只是单纯完成模版变量到真实 DOM 的转换，读者只需体会简单道理即可。

双向绑定实现

上述实现是单向的，数据变化引起了视图变化，那么如果页面中存在一个输入框，如何触发数据变化呢？比如：

```
<input v-model="inputData" />
```

我们需要在模版编译中，对于存在 `v-model` 属性的 `node` 进行事件监听，在输入框输入时，改变 `v-model` 属性值对应的数据即可（这里为 `inputData`），增加 `compile` 中的 `replace` 方法逻辑，对于 `node.nodeType === 1` 的 DOM 类型，伪代码如下：

```
function replace(el, data) {  
  // 省略...  
  if (node.nodeType === 1) {  
  
    let attributesArray = node.attributes  
  
    Array.from(attributesArray).forEach(attr => {  
      let attributeName = attr.name  
      let attributeValue = attr.value  
  
      if (name.includes('v-')) {  
        node.value = data[attributeValue]  
      }  
    })  
  }  
}
```

```
node.addEventListener('input', e => {
  let newVal = e.target.value
  data[attributeValue] = newVal
  // ...
  // 更改数据源，触发 setter
  // ...
})
})

}

if (node.childNodes && node.childNodes.length) {
  replace(node)
}
}
```

发布订阅模式简单应用

作为前端开发人员，我们对于所谓的「事件驱动」理念——即「事件发布订阅模式（Pub/Sub 模式）」一定再熟悉不过了。这种模式在 JavaScript 里面有与生俱来的基因：我们可以认为 JavaScript 本身就是事件驱动型语言，比如，应用中对一个 button 进行了事件绑定，用户点击之后就会触发按钮上面的 click 事件。这是因为此时有特定程序正在监听这个事件，随之触发了相关的处理程序。

这个模式的一个好处之一在于能够解耦，实现「高内聚、低耦合」的理念。这种模式对于我们框架的设计同样也不可或缺。请思考：通过前面内容的学习，我们了解了如何监听数据的变化。如果最终想实现响应式 MVVM，或所谓的双向绑定，那么还需要根据这个数据变化作出相应的视图更新。这个逻辑和我们在页面中对 button 绑定事件处理函数是多么相近。

那么这样一个「熟悉的」模式应该怎么实现呢，又该如何在框架中具体应用呢？看代码：

```
class Notify {
  constructor() {
    this.subscribers = []
  }
}
```

```
add(handler) {
  this.subscribers.push(handler)
}
emit() {
  this.subscribers.forEach(subscriber => subscriber())
}
}
```

使用：

```
let notify = new Notify()

notify.add(() => {
  console.log('emit here')
})

notify.emit()
// emit here
```

这就是一个简单实现的「事件发布订阅模式」，当然代码只是启发思路，真实应用还比较「粗糙」，没有进行事件名设置，APIs 也并不丰富，但完全能够说明问题了。其实读者翻看 Vue 源码，也能了解 Vue 中的发布订阅模式很简单。

MVVM 融会贯通

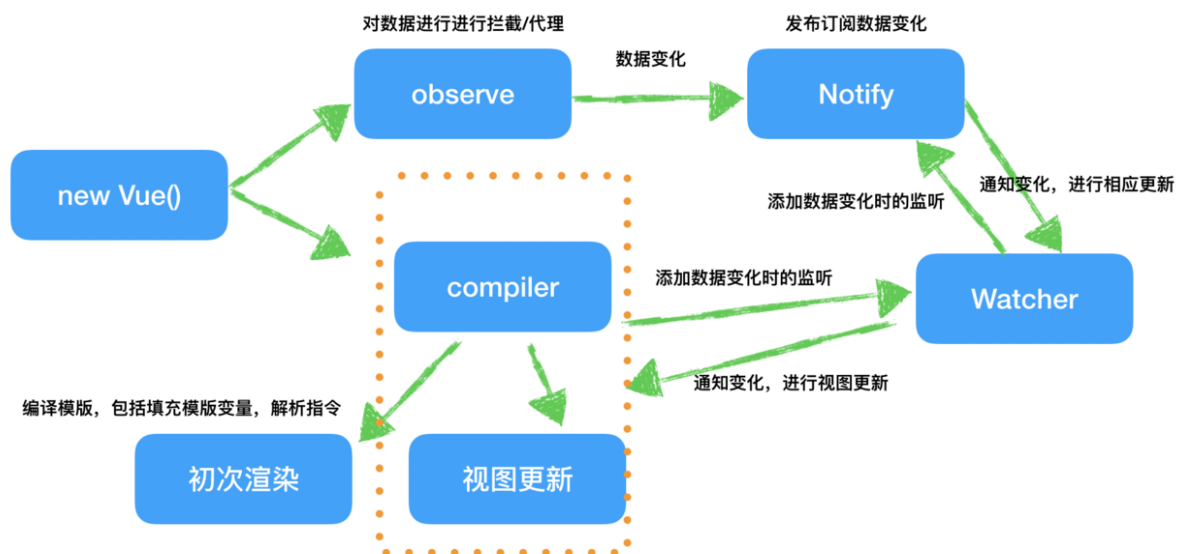
回顾一下前面的基本内容：数据拦截和代理、发布订阅模式、模版编译，那么如何根据这些概念实现一个 MVVM 框架呢？其实不管是 Vue 还是其他类库或框架，其解决思想都是建立在前文所述概念之上的。

我们来串联，整个过程是：首先对数据进行深度拦截或代理，对每一个属性的 getter 和 setter 进行「加工」，该「加工」具体做些什么后面马上会有说明。在模版初次编译时，解析指令（如 v-model），并进行依赖收集（{{变量}}），订阅数据的变化。

这里的依赖收集过程具体指：当调用 compiler 中的 replace 方法时，我们会读取数据进行模版变量的替换，这时候「读取数据时」需要做一个标记，用来表示

「我依赖这一项数据」，因此我要订阅这个属性值的变化。Vue 中定义一个 Watcher 类来表示观察订阅依赖。这就实现了整套流程，换个思路再复述一遍：我们知道模版编译过程中会读取数据，进而触发数据源属性值的 getter，因此上面所说的数据代理的「加工」就是在数据监听的 getter 中记录这个依赖，同时在 setter 触发数据变化时，执行依赖对应的相关操作，最终触发模版中数据的变化。

我们抽象成流程图来理解：



图片

这也是 Vue 框架（类库）的基本架构图。由此看出，Vue 的实现，或者大部分 MVVM 的实现，就是我们本节课程介绍的概念组合应用。

关于框架的对比剖析，更多话题我们留在《第 4-7 课：从框架和类库，我们该学到什么》一课中介绍。

揭秘虚拟 DOM

我们来看现代框架中另一个重头戏——虚拟 DOM。虚拟 DOM 这个概念其实并没有那么新，甚至在前端三大框架问世之前，虚拟 DOM 就已经存在了，只不过 React 创造性的应用了虚拟 DOM，为前端发展带来了变革。Vue 2.0 也很快跟

进，使得虚拟 DOM 彻底成为现代框架的重要基因。简单来说，虚拟 DOM 就是用数据结构表示 DOM 结构，它并没有真实 append 到 DOM 上，因此称之为「虚拟」。

应用虚拟 DOM 的收益也很直观：操作数据结构远比和浏览器交互去操作 DOM 快很多。请读者准确理解这句话：操作数据结构是指改变对象（虚拟 DOM），这个过程比修改真实 DOM 快很多。但虚拟 DOM 也最终是要挂载到浏览器上成为真实 DOM 节点，因此使用虚拟 DOM 并不能使得操作 DOM 的数量减少，但能够精确地获取最小的、最必要的操作 DOM 的集合。

这样一来，我们抽象表示 DOM，每次通过 DOM diff 计算出视图前后更新的最小差异，再去把最小差异应用到真实 DOM 上的做法，无疑更为可靠，性能更有保障。

那我们该如何表示虚拟 DOM 呢？又该如何产出虚拟 DOM 呢？

直观上我们看这样一段 DOM 结构：

chapter1

chapter2

chapter3

如果用 JavaScript 来表示，我们采用对象结构：

```
const chapterListVirtualDom = {
  tagName: 'ul',
  attributes: {
    id: 'chapterList'
  },
  children: [
    { tagName: 'li', attributes: { class: 'chapter' },
      children: ['chapter1'] },
    { tagName: 'li', attributes: { class: 'chapter' },
      children: ['chapter2'] },
    { tagName: 'li', attributes: { class: 'chapter' },
      children: ['chapter3'] },
  ]
}
```

很好理解：tagName 表示虚拟 DOM 对应的真实 DOM 标签类型；attributes 是一个对象，表示真实 DOM 节点上所有的属性；children 对应真实 DOM 的 childNodes，其中 childNodes 每一项又是类似的结构。

我们来实现一个虚拟 DOM 生成类，用于生产虚拟 DOM：

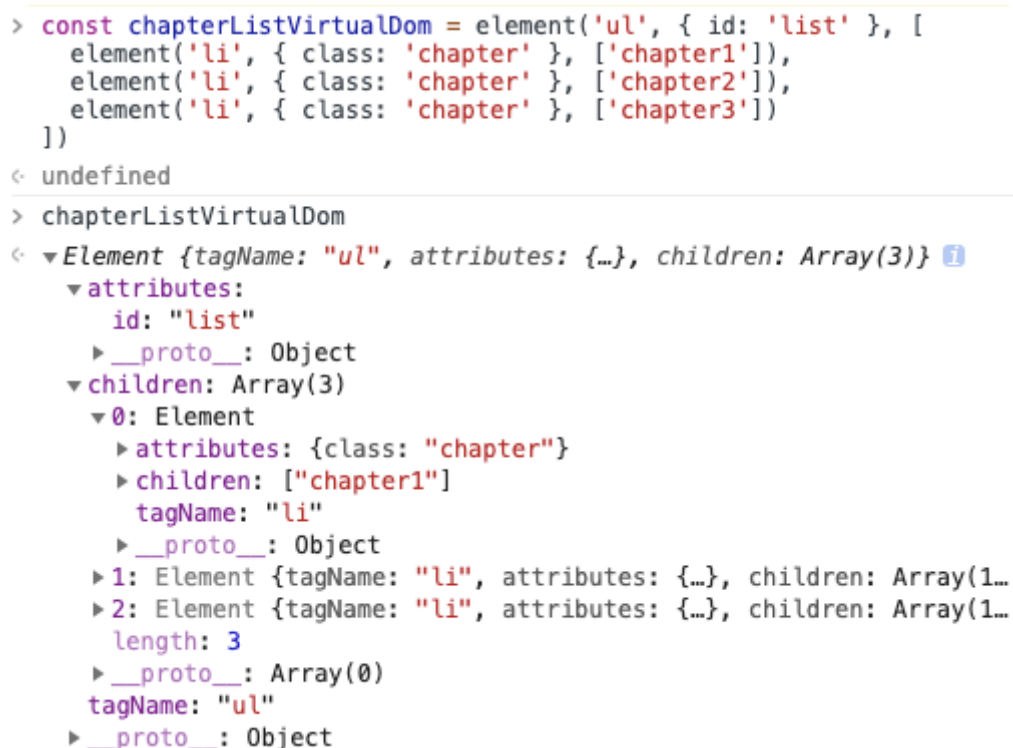
```
class Element {
  constructor(tagName, attributes = {}, children = []) {
    this.tagName = tagName
    this.attributes = attributes
    this.children = children
  }
}
```

```
function element(tagName, attributes, children) {  
  return new Element(tagName, attributes, children)  
}
```

上述虚拟 DOM 就可以这样生成：

```
const chapterListVirtualDom = element('ul', { id: 'list'  
, [  
  element('li', { class: 'chapter' }, ['chapter1']),  
  element('li', { class: 'chapter' }, ['chapter2']),  
  element('li', { class: 'chapter' }, ['chapter3'])  
])
```

如图：



```
> const chapterListVirtualDom = element('ul', { id: 'list' }, [  
  element('li', { class: 'chapter' }, ['chapter1']),  
  element('li', { class: 'chapter' }, ['chapter2']),  
  element('li', { class: 'chapter' }, ['chapter3'])  
])  
< undefined  
> chapterListVirtualDom  
< ▼ Element {tagName: "ul", attributes: {...}, children: Array(3)} ⓘ  
  ▼ attributes:  
    id: "list"  
    ▶ __proto__: Object  
  ▼ children: Array(3)  
    ▼ 0: Element  
      ▶ attributes: {class: "chapter"}  
      ▶ children: ["chapter1"]  
      tagName: "li"  
      ▶ __proto__: Object  
    ▶ 1: Element {tagName: "li", attributes: {...}, children: Array(1...  
    ▶ 2: Element {tagName: "li", attributes: {...}, children: Array(1...  
      length: 3  
      ▶ __proto__: Array(0)  
    tagName: "ul"  
    ▶ __proto__: Object
```

图片

是不是很简单？我们继续完成虚拟 DOM 向真实 DOM 节点的生成。首先实现一个 `setAttribute` 方法，后续的代码都将使用 `setAttribute` 方法来对 DOM 节点进行属性设置。

```
const setAttribute = (node, key, value) => {
  switch (key) {
    case 'style':
      node.style.cssText = value
      break
    case 'value':
      let tagName = node.tagName || ''
      tagName = tagName.toLowerCase()
      if (
        tagName === 'input' || tagName === 'textarea'
      ) {
        node.value = value
      } else {
        // 如果节点不是 input 或者 textarea，则使用
        setAttribute 去设置属性
        node.setAttribute(key, value)
      }
      break
    default:
      node.setAttribute(key, value)
      break
  }
}
```

Element 类中加入 `render` 原型方法，该方法的目的是根据虚拟 DOM 生成真实 DOM 片段：

```
class Element {
  constructor(tagName, attributes = {}, children = []) {
    this.tagName = tagName
    this.attributes = attributes
    this.children = children
  }
}
```



```
render () {
  let element = document.createElement(this.tagName)
  let attributes = this.attributes

  for (let key in attributes) {
    setAttribute(element, key, attributes[key])
  }

  let children = this.children

  children.forEach(child => {
    let childElement = child instanceof Element
      ? child.render() // 若 child 也是虚拟节点，递归进行
      : document.createTextNode(child) // 若是字符串，直接创建文本节点
    element.appendChild(childElement)
  })

  return element
}

function element (tagName, attributes, children) {
  return new Element(tagName, attributes, children)
}
```

实现也不困难，我们借助工具方法：setAttribute 进行属性的创建；对 children 每一项类型进行判断，如果是 Element 实例，进行递归调用 child 的 render 方法；直到遇见文本节点类型，进行内容渲染。

有了真实的 DOM 节点片段，我们趁热打铁，将真实的 DOM 节点渲染到浏览器上，实现 renderDOM 方法：

```
const renderDom = (element, target) => {
  target.appendChild(element)
}
```

执行代码：

```
const setAttribute = (node, key, value) => {
  switch (key) {
    case 'style':
      node.style.cssText = value
      break
    case 'value':
      let tagName = node.tagName || ''
      tagName = tagName.toLowerCase()
      if (
        tagName === 'input' || tagName === 'textarea'
      ) {
        node.value = value
      } else {
        // 如果节点不是 input 或者 textarea, 则使用
        setAttribute 去设置属性
        node.setAttribute(key, value)
      }
      break
    default:
      node.setAttribute(key, value)
      break
  }
}

class Element {
  constructor(tagName, attributes = {}, children = []) {
    this.tagName = tagName
    this.attributes = attributes
    this.children = children
  }

  render () {
    let element = document.createElement(this.tagName)
    let attributes = this.attributes
```

```
    for (let key in attributes) {
      setAttribute(element, key, attributes[key])
    }

    let children = this.children

    children.forEach(child => {
      let childElement = child instanceof Element
        ? child.render() // 若 child 也是虚拟节点，递归进行
        : document.createTextNode(child) // 若是字符串，直接创建文本节点
      element.appendChild(childElement)
    })

    return element
  }
}

function element (tagName, attributes, children) {
  return new Element(tagName, attributes, children)
}

const renderDom = (element, target) => {
  target.appendChild(element)
}

const chapterListVirtualDom = element('ul', { id: 'list' }, [
  element('li', { class: 'chapter' }, ['chapter1']),
  element('li', { class: 'chapter' }, ['chapter2']),
  element('li', { class: 'chapter' }, ['chapter3'])
])

const dom = chapterListVirtualDom.render()

renderDom(dom, document.body)
```

得到如图：

```
> const chapterListVirtualDom = element('ul', { id: 'list' }, [
  element('li', { class: 'chapter' }, ['chapter1']),
  element('li', { class: 'chapter' }, ['chapter2']),
  element('li', { class: 'chapter' }, ['chapter3'])
])
< undefined
> chapterListVirtualDom
< ▼ Element {tagName: "ul", attrs: {...}, children: Array(3)} ⓘ
  ▶ attrs: {id: "list"}
  ▶ children: (3) [Element, Element, Element]
    tagName: "ul"
  ▶ __proto__:
    ▶ constructor: class Element
    ▶ render: render () { let element = document.createElement(this...
    ▶ __proto__: Object
> chapterListVirtualDom.render()
< ▼ <ul>
  <li>chapter1</li>
  <li>chapter2</li>
  <li>chapter3</li>
</ul>
>
```

图片

虚拟 DOM diff

有了上述基础，我们可以产出一份虚拟 DOM，并渲染在浏览器中。当用户在特定操作后，会产出新的一份虚拟 DOM，如何得出前后两份虚拟 DOM 的差异，并交给浏览器需要更新的结果呢？这就涉及到 DOM diff 的过程。

直观上，因为虚拟 DOM 是个树形结构，所以我们需要对两份虚拟 DOM 进行递归比较，将变化存储在一个变量 patches 中：

```
const diff = (oldVirtualDom, newVirtualDom) => {
  let patches = {}

  // 递归树，比较后的结果放到 patches
  walkToDiff(oldVirtualDom, newVirtualDom, 0, patches)
```

```
    // 返回 diff 结果
    return patches
}
```

walkToDiff 前两个参数是两个需要比较的虚拟 DOM 对象；第三个参数记录 nodeIndex，在删除节点时使用，初始为 0；第四个参数是一个闭包变量，记录 diff 结果：

```
let initialIndex = 0

const walkToDiff = (oldVirtualDom, newVirtualDom, index,
patches) => {
  let diffResult = []

  // 如果 newVirtualDom 不存在，说明该节点被移除，我们将 type
  为 REMOVE 的对象推进 diffResult 变量，并记录 index
  if (!newVirtualDom) {
    diffResult.push({
      type: 'REMOVE',
      index
    })
  }
  // 如果新旧节点都是文本节点，是字符串
  else if (typeof oldVirtualDom === 'string' && typeof
newVirtualDom === 'string') {
    // 比较文本是否相同，如果不同则记录新的结果
    if (oldVirtualDom !== newVirtualDom) {
      diffResult.push({
        type: 'MODIFY_TEXT',
        data: newVirtualDom,
        index
      })
    }
  }
  // 如果新旧节点类型相同
  else if (oldVirtualDom.tagName ===
```

```
newVirtualDom.tagName) {
  // 比较属性是否相同
  let diffAttributeResult = {}

  for (let key in oldVirtualDom) {
    if (oldVirtualDom[key] !== newVirtualDom[key]) {
      diffAttributeResult[key] = newVirtualDom[key]
    }
  }

  for (let key in newVirtualDom) {
    // 旧节点不存在的新属性
    if (!oldVirtualDom.hasOwnProperty(key)) {
      diffAttributeResult[key] = newVirtualDom[key]
    }
  }

  if (Object.keys(diffAttributeResult).length > 0) {
    diffResult.push({
      type: 'MODIFY_ATTRIBUTES',
      diffAttributeResult
    })
  }

  // 如果有子节点，遍历子节点
  oldVirtualDom.children.forEach((child, index) => {
    walkToDiff(child, newVirtualDom.children[index],
    ++initialIndex, patches)
  })
}
// else 说明节点类型不同，被直接替换了，我们直接将新的结果 push
else {
  diffResult.push({
    type: 'REPLACE',
    newVirtualDom
  })
}
```

```
if (!oldVirtualDom) {
  diffResult.push({
    type: 'REPLACE',
    newVirtualDom
  })
}

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

我们最后将所有代码放在一起：

```
const setAttribute = (node, key, value) => {
  switch (key) {
    case 'style':
      node.style.cssText = value
      break
    case 'value':
      let tagName = node.tagName || ''
      tagName = tagName.toLowerCase()
      if (
        tagName === 'input' || tagName === 'textarea'
      ) {
        node.value = value
      } else {
        // 如果节点不是 input 或者 textarea, 则使用
        setAttribute 去设置属性
        node.setAttribute(key, value)
      }
      break
    default:
      node.setAttribute(key, value)
      break
  }
}
```

```
}

class Element {
  constructor(tagName, attributes = {}, children = []) {
    this.tagName = tagName
    this.attributes = attributes
    this.children = children
  }

  render () {
    let element = document.createElement(this.tagName)
    let attributes = this.attributes

    for (let key in attributes) {
      setAttribute(element, key, attributes[key])
    }

    let children = this.children

    children.forEach(child => {
      let childElement = child instanceof Element
        ? child.render() // 若 child 也是虚拟节点, 递归进行
        : document.createTextNode(child) // 若是字符串, 直接创建文本节点
      element.appendChild(childElement)
    })

    return element
  }
}

function element (tagName, attributes, children) {
  return new Element(tagName, attributes, children)
}

const renderDom = (element, target) => {
  target.appendChild(element)
}
```



```
}

const diff = (oldVirtualDom, newVirtualDom) => {
  let patches = {}

  // 递归树 比较后的结果放到 patches
  walkToDiff(oldVirtualDom, newVirtualDom, 0, patches)

  return patches
}

let initialIndex = 0

const walkToDiff = (oldVirtualDom, newVirtualDom, index,
patches) => {
  let diffResult = []

  // 如果 newVirtualDom 不存在, 说明该节点被移除, 我们将 type
  为 REMOVE 的对象推进 diffResult 变量, 并记录 index
  if (!newVirtualDom) {
    diffResult.push({
      type: 'REMOVE',
      index
    })
  }
  // 如果新旧节点都是文本节点, 是字符串
  else if (typeof oldVirtualDom === 'string' && typeof
newVirtualDom === 'string') {
    // 比较文本是否相同, 如果不同则记录新的结果
    if (oldVirtualDom !== newVirtualDom) {
      diffResult.push({
        type: 'MODIFY_TEXT',
        data: newVirtualDom,
        index
      })
    }
  }
}
```

```
// 如果新旧节点类型相同
else if (oldVirtualDom.tagName ===
newVirtualDom.tagName) {
  // 比较属性是否相同
  let diffAttributeResult = {}

  for (let key in oldVirtualDom) {
    if (oldVirtualDom[key] !== newVirtualDom[key]) {
      diffAttributeResult[key] = newVirtualDom[key]
    }
  }

  for (let key in newVirtualDom) {
    // 旧节点不存在的新属性
    if (!oldVirtualDom.hasOwnProperty(key)) {
      diffAttributeResult[key] = newVirtualDom[key]
    }
  }

  if (Object.keys(diffAttributeResult).length > 0) {
    diffResult.push({
      type: 'MODIFY_ATTRIBUTES',
      diffAttributeResult
    })
  }

  // 如果有子节点，遍历子节点
  oldVirtualDom.children.forEach((child, index) => {
    walkToDiff(child, newVirtualDom.children[index],
    ++initialIndex, patches)
  })
}
// else 说明节点类型不同，被直接替换了，我们直接将新的结果 push
else {
  diffResult.push({
    type: 'REPLACE',
    newVirtualDom
```

```
    })  
  }  
  
  if (!oldVirtualDom) {  
    diffResult.push({  
      type: 'REPLACE',  
      newVirtualDom  
    })  
  }  
  
  if (diffResult.length) {  
    patches[index] = diffResult  
  }  
}
```

我们对 diff 进行测试：

```
const chapterListVirtualDom = element('ul', { id: 'list'  
, [  
  element('li', { class: 'chapter' }, ['chapter1']),  
  element('li', { class: 'chapter' }, ['chapter2']),  
  element('li', { class: 'chapter' }, ['chapter3'])  
])  
  
const chapterListVirtualDom1 = element('ul', { id:  
'list2' }, [  
  element('li', { class: 'chapter2' }, ['chapter4']),  
  element('li', { class: 'chapter2' }, ['chapter5']),  
  element('li', { class: 'chapter2' }, ['chapter6'])  
])  
  
diff(chapterListVirtualDom, chapterListVirtualDom1)
```

得到如图 diff 数组：

```

> diff(chapterListVirtualDom, chapterListVirtualDom1)
< {0: Array(1), 1: Array(1), 2: Array(1), 3: Array(1), 4: Array(1),
  5: Array(1), 6: Array(1)}
  ▼ 0: Array(1)
    ▼ 0:
      ▼ diffAttributeResult:
        ▶ attributes: {id: "list2"}
        ▶ children: (3) [Element, Element, Element]
        ▶ __proto__: Object
        type: "MODIFY_ATTRIBUTES"
        ▶ __proto__: Object
        length: 1
      ▶ __proto__: Array(0)
    ▼ 1: Array(1)
      ▼ 0:
        ▼ diffAttributeResult:
          ▶ attributes: {class: "chapter2"}
          ▶ children: ["chapter4"]
          ▶ __proto__: Object
          type: "MODIFY_ATTRIBUTES"
          ▶ __proto__: Object
          length: 1
        ▶ __proto__: Array(0)
      ▶ 2: [{...}]
      ▶ 3: [{...}]
      ▶ 4: [{...}]
      ▶ 5: [{...}]
      ▶ 6: [{...}]
      ▶ __proto__: Object

```

图片

最小化差异应用

大功告成之前，我们来看看都做了哪些事情：通过 Element class 生成了虚拟 DOM，通过 diff 方法对任意两个虚拟 DOM 进行比对，得到差异。那么这个差异如何更新到现有的 DOM 节点中呢？看上去需要一个 patch 方法来完成：

```

const patch = (node, patches) => {
  let walker = { index: 0 }
  walk(node, walker, patches)
}

```

patch 方法接受一个真实的 DOM 节点，它是现有的浏览器中需要进行更新的 DOM 节点，同时接受一个最小化差异集合，该集合对接 diff 方法返回的结果。在 patch 方法内部，我们调用了 walk 函数：

```
const walk = (node, walker, patches) => {
  let currentPatch = patches[walker.index]

  let childNodes = node.childNodes

  childNodes.forEach(child => {
    walker.index++
    walk(child, walker, patches)
  })

  if (currentPatch) {
    doPatch(node, currentPatch)
  }
}
```

walk 进行自身递归，对于当前节点的差异调用 doPatch 方法进行更新：

```
const doPatch = (node, patches) => {
  patches.forEach(patch => {
    switch (patch.type) {
      case 'MODIFY_ATTRIBUTES':
        const attributes =
          patch.diffAttributeResult.attributes
        for (let key in attributes) {
          if (node.nodeType !== 1) return
          const value = attributes[key]
          if (value) {
            setAttribute(node, key, value)
          } else {
            node.removeAttribute(key)
          }
        }
        break
    }
  })
}
```

```

    case 'MODIFY_TEXT':
      node.textContent = patch.data
      break
    case 'REPLACE':
      let newNode = (patch.newNode instanceof Element)
      ? render(patch.newNode) :
      document.createTextNode(patch.newNode)
      node.parentNode.replaceChild(newNode, node)
      break
    case 'REMOVE':
      node.parentNode.removeChild(node)
      break
    default:
      break
  }
})
}

```

doPatch 对四种类型的 diff 进行处理，最终进行测试：

```

var element = chapterListVirtualDom.render()
  renderDom(element, document.body)

  const patches = diff(chapterListVirtualDom,
chapterListVirtualDom1)

  patch(element, patches)

```

全部代码放在一起：

```

const setAttribute = (node, key, value) => {
  switch (key) {
    case 'style':
      node.style.cssText = value
      break
    case 'value':

```

```
let tagName = node.tagName || ''
tagName = tagName.toLowerCase()
if (
  tagName === 'input' || tagName === 'textarea'
) {
  node.value = value
} else {
  // 如果节点不是 input 或者 textarea, 则使用
setAttribute 去设置属性
  node.setAttribute(key, value)
}
break
default:
  node.setAttribute(key, value)
  break
}
}

class Element {
  constructor(tagName, attributes = {}, children = []) {
    this.tagName = tagName
    this.attributes = attributes
    this.children = children
  }

  render () {
    let element = document.createElement(this.tagName)
    let attributes = this.attributes

    for (let key in attributes) {
      setAttribute(element, key, attributes[key])
    }

    let children = this.children

    children.forEach(child => {
      let childElement = child instanceof Element
```

```
    ? child.render() // 若 child 也是虚拟节点，递归进行
    : document.createTextNode(child) // 若是字符串，直接创建文本节点
```

```
    element.appendChild(childElement)
  })

  return element
}

function element (tagName, attributes, children) {
  return new Element(tagName, attributes, children)
}

const renderDom = (element, target) => {
  target.appendChild(element)
}

const diff = (oldVirtualDom, newVirtualDom) => {
  let patches = {}

  // 递归树 比较后的结果放到 patches
  walkToDiff(oldVirtualDom, newVirtualDom, 0, patches)

  return patches
}

let initialIndex = 0

const walkToDiff = (oldVirtualDom, newVirtualDom, index, patches) => {
  let diffResult = []

  // 如果 newVirtualDom 不存在，说明该节点被移除，我们将 type
  // 为 REMOVE 的对象推进 diffResult 变量，并记录 index
  if (!newVirtualDom) {
    diffResult.push({
```



```
        type: 'REMOVE',
        index
      })
    }
    // 如果新旧节点都是文本节点，是字符串
    else if (typeof oldVirtualDom === 'string' && typeof
newVirtualDom === 'string') {
      // 比较文本是否相同，如果不同则记录新的结果
      if (oldVirtualDom !== newVirtualDom) {
        diffResult.push({
          type: 'MODIFY_TEXT',
          data: newVirtualDom,
          index
        })
      }
    }
    // 如果新旧节点类型相同
    else if (oldVirtualDom.tagName ===
newVirtualDom.tagName) {
      // 比较属性是否相同
      let diffAttributeResult = {}

      for (let key in oldVirtualDom) {
        if (oldVirtualDom[key] !== newVirtualDom[key]) {
          diffAttributeResult[key] = newVirtualDom[key]
        }
      }

      for (let key in newVirtualDom) {
        // 旧节点不存在的新属性
        if (!oldVirtualDom.hasOwnProperty(key)) {
          diffAttributeResult[key] = newVirtualDom[key]
        }
      }

      if (Object.keys(diffAttributeResult).length > 0) {
        diffResult.push({
```

```
        type: 'MODIFY_ATTRIBUTES',
        diffAttributeResult
      })
    }

    // 如果有子节点，遍历子节点
    oldVirtualDom.children.forEach((child, index) => {
      walkToDiff(child, newVirtualDom.children[index],
        ++initialIndex, patches)
    })
  }
  // else 说明节点类型不同，被直接替换了，我们直接将新的结果 push
  else {
    diffResult.push({
      type: 'REPLACE',
      newVirtualDom
    })
  }

  if (!oldVirtualDom) {
    diffResult.push({
      type: 'REPLACE',
      newVirtualDom
    })
  }

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

const chapterListVirtualDom = element('ul', { id: 'list'
}, [
  element('li', { class: 'chapter' }, ['chapter1']),
  element('li', { class: 'chapter' }, ['chapter2']),
  element('li', { class: 'chapter' }, ['chapter3'])
])
```

```
const chapterListVirtualDom1 = element('ul', { id:
'list2' }, [
  element('li', { class: 'chapter2' }, ['chapter4']),
  element('li', { class: 'chapter2' }, ['chapter5']),
  element('li', { class: 'chapter2' }, ['chapter6'])
])
```

```
const patch = (node, patches) => {
  let walker = { index: 0 }
  walk(node, walker, patches)
}
```

```
const walk = (node, walker, patches) => {
  let currentPatch = patches[walker.index]
```

```
  let childNodes = node.childNodes
```

```
  childNodes.forEach(child => {
    walker.index++
    walk(child, walker, patches)
  })
```

```
  if (currentPatch) {
    doPatch(node, currentPatch)
  }
}
```

```
const doPatch = (node, patches) => {
  patches.forEach(patch => {
    switch (patch.type) {
      case 'MODIFY_ATTRIBUTES':
        const attributes =
patch.diffAttributeResult.attributes
        for (let key in attributes) {
          if (node.nodeType !== 1) return
          const value = attributes[key]
```

```

        if (value) {
            setAttribute(node, key, value)
        } else {
            node.removeAttribute(key)
        }
    }
    break
case 'MODIFY_TEXT':
    node.textContent = patch.data
    break
case 'REPLACE':
    let newNode = (patch.newNode instanceof Element)
? render(patch.newNode) :
document.createTextNode(patch.newNode)
    node.parentNode.replaceChild(newNode, node)
    break
case 'REMOVE':
    node.parentNode.removeChild(node)
    break
default:
    break
}
})
}

```

先执行：

```

var element = chapterListVirtualDom.render()
renderDom(element, document.body)

```

再执行：

```

const patches = diff(chapterListVirtualDom,
chapterListVirtualDom1)

patch(element, patches)

```

生成结果符合预期。

短短不到两百行代码，就实现了虚拟 DOM 思想的全部流程。当然其中还有一些优化手段，一些边界情况并没有进行特别处理，但是我们去翻看一些著名的虚拟 DOM 库：snabbdom、etch 等，其实现思想和上述教例完全一致。

总结

现代框架无疑极大程度上解放了前端生产力，其设计思想相互借鉴，存在非常多的共性。本讲我们通过分析前端框架中的共性，梳理概念原理，希望达到「任何一种框架变得不再神秘」的目的。掌握了这些基本思想，我们不仅能触类旁通，更快地上手框架，更能学习进阶，吸取优秀框架的精华。

点击查看下一节 ∨

你以为你懂 React 吗？