

Virtual DOM 及 Diff 算法

1. JSX 到底是什么

使用 React 就一定会写 JSX，JSX 到底是什么呢？它是一种 JavaScript 语法的扩展，React 使用它来描述用户界面长成什么样子。虽然它看起来非常像 HTML，但它确实是 JavaScript。在 React 代码执行之前，Babel 会对将 JSX 编译为 React API。

```
<div className="container">
  <h3>Hello React</h3>
  <p>React is great </p>
</div>

React.createElement(
  "div",
  {
    className: "container"
  },
  React.createElement("h3", null, "Hello React"),
  React.createElement("p", null, "React is great")
);
```

从两种语法对比来看，JSX 语法的出现是为了让 React 开发人员编写用户界面代码更加轻松。

[Babel REPL](#)

2. DOM 操作问题

在现代 web 应用程序中使用 JavaScript 操作 DOM 是必不可少的，但遗憾的是它比其他大多数 JavaScript 操作要慢的多。

大多数 JavaScript 框架对于 DOM 的更新远远超过其必须进行的更新，从而使得这种缓慢操作变得更糟。

例如假设你有包含十个项目的列表，你仅仅更改了列表中的第一项，大多数 JavaScript 框架会重建整个列表，这比必要的工作要多十倍。

更新效率低下已经成为严重问题，为了解决这个问题，React 普及了一种叫做 Virtual DOM 的东西，Virtual DOM 出现的目的就是为了提高 JavaScript 操作 DOM 对象的效率。

3. 什么是 Virtual DOM

在 React 中，每个 DOM 对象都有一个对应的 Virtual DOM 对象，它是 DOM 对象的 JavaScript 对象表现形式，其实就是使用 JavaScript 对象来描述 DOM 对象信息，比如 DOM 对象的类型是什么，它身上有哪些属性，它拥有哪些子元素。

可以把 Virtual DOM 对象理解为 DOM 对象的副本，但是它不能直接显示在屏幕上。

```
<div className="container">
  <h3>Hello React</h3>
  <p>React is great </p>
</div>
```

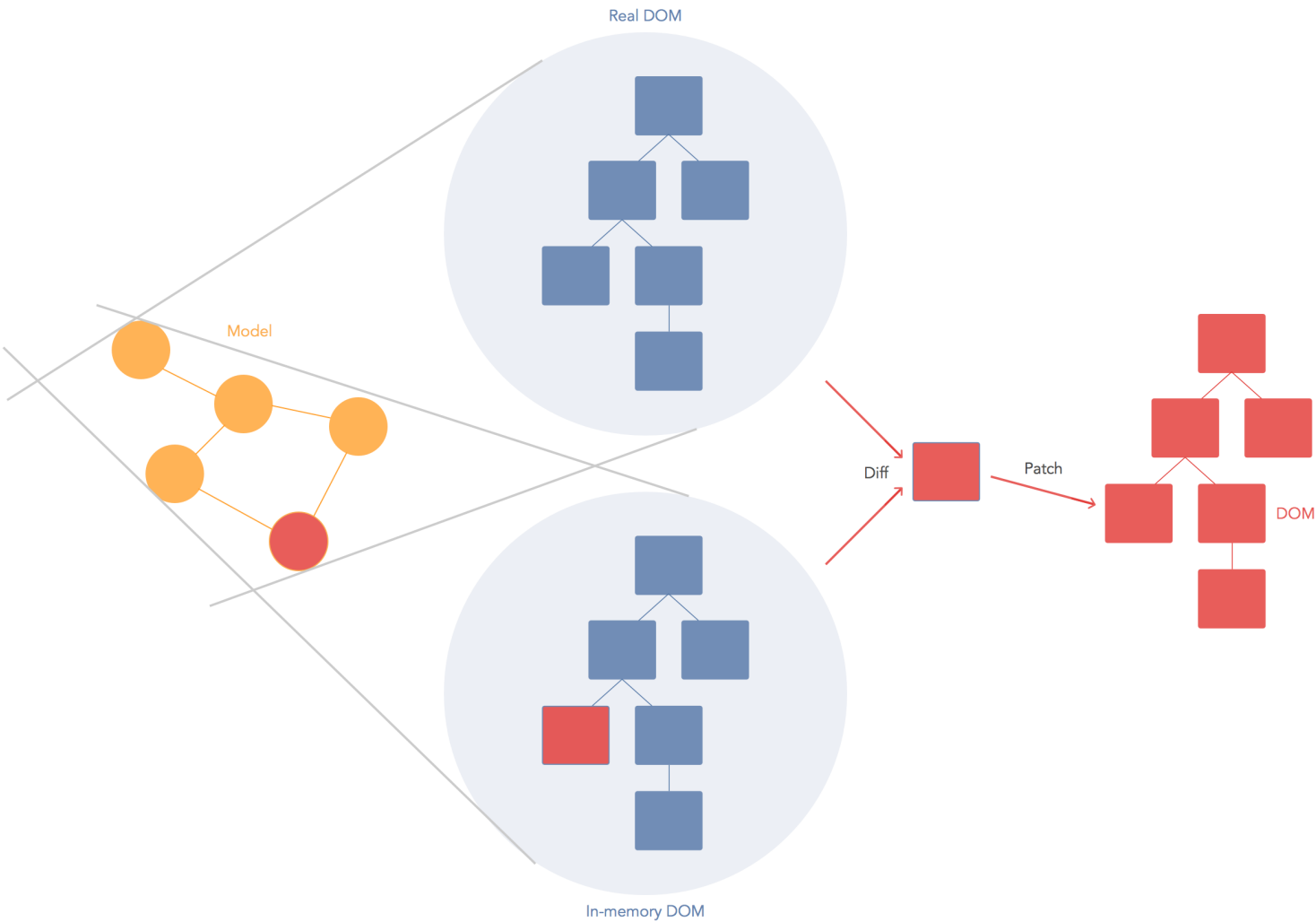
```
{
  type: "div",
  props: { className: "container" },
  children: [
    {
      type: "h3",
      props: null,
      children: [
        {
          type: "text",
          props: {
            textContent: "Hello React"
          }
        }
      ]
    }
  ],
},
{
  type: "p",
  props: null,
  children: [
    {
      type: "text",
      props: {
        textContent: "React is great"
      }
    }
  ]
}
]
```

4. Virtual DOM 如何提升效率

精准找出发生变化的 DOM 对象，只更新发生变化的部分。

在 React 第一次创建 DOM 对象后，会为每个 DOM 对象创建其对应的 Virtual DOM 对象，在 DOM 对象发生更新之前，React 会先更新所有的 Virtual DOM 对象，然后 React 会将更新后的 Virtual DOM 和 更新前的 Virtual DOM 进行比较，从而找出发生变化的部分，React 会将发生变化的部分更新到真实的 DOM 对象中，React 仅更新必要更新的部分。

Virtual DOM 对象的更新和比较仅发生在内存中，不会在视图中渲染任何内容，所以这一部分的性能损耗成本是微不足道的。



```
<div id="container">
  <p>Hello React</p>
</div>

<div id="container">
  <p>Hello Angular</p>
</div>

const before = {
  type: "div",
  props: { id: "container" },
  children: [
    {
      type: "p",
      props: null,
      children: [
        { type: "text", props: { textContent: "Hello React" } }
      ]
    }
  ]
}

const after = {
  type: "div",
  props: { id: "container" },
  children: [
    {
      type: "p",
      props: null,
      children: [
        { type: "text", props: { textContent: "Hello Angular" } }
      ]
    }
  ]
}
```

5. 创建 Virtual DOM

在 React 代码执行前，JSX 会被 Babel 转换为 React.createElement 方法的调用，在调用 createElement 方法时会传入元素的类型，元素的属性，以及元素的子元素，createElement 方法的返回值为构建好的 Virtual DOM 对象。

```
{
  type: "div",
  props: null,
  children: [{type: "text", props: {textContent: "Hello"}}]
}

/**
 * 创建 Virtual DOM
 * @param {string} type 类型
 * @param {object | null} props 属性
 * @param {createElement[]} children 子元素
 * @return {object} Virtual DOM
 */
function createElement (type, props, ...children) {
  return {
    type,
    props,
    children
  }
}
```

从 createElement 方法的第三个参数开始就都是子元素了，在定义 createElement 方法时，通过 ...children 将所有的子元素放置到 children 数组中。

```
const virtualDOM = (

# 嵌套1 <div>嵌套 1.1</div></div>{2 == 1 && <div>如果2和1相等渲染当前内容</div>}{2 == 2 && <div>2</div>}<span>这是一段内容</span><button onClick={() => alert("你好")}>点击我</button><h3>这个将会被删除</h3>2, 3</div>) console.log(virtualDOM)


```

通过以上代码测试，发现返回的 Virtual DOM 存在一些问题，第一个问题是文本节点被直接放入到了数组中

```
▼ Object ⓘ  
  ▼ children: Array(10)  
    ▶ 0: {type: "h1", props: null, children: Array(1)}  
    ▶ 1: {type: "h2", props: null, children: Array(1)}  
    ▶ 2: {type: "div", props: null, children: Array(2)}  
    ▼ 3:  
      ▶ children: ["(观察：这个将会被改变)"]  
      props: null  
      type: "h3"  
      ▶ __proto__: Object  
      4: false  
      ▶ 5: {type: "div", props: null, children: Array(1)}  
      ▶ 6: {type: "span", props: null, children: Array(1)}  
      ▶ 7: {type: "button", props: {...}, children: Array(1)}  
      ▶ 8: {type: "h3", props: null, children: Array(1)}  
      9: "2, 3"  
      length: 10  
      ▶ __proto__: Array(0)  
    ▶ props: {className: "container"}  
    type: "div"  
    ▶ __proto__: Object
```

而我们期望是文本节点应该是这样的

```
children: [  
  {  
    type: "text",  
    props: {  
      textContent: "React is great"  
    }  
  }  
]
```

通过以下代码对 Virtual DOM 进行改造，重新构建 Virtual DOM。

```
// 将原有 children 拷贝一份 不要在原有数组上进行操作  
const childElements = [].concat(...children).map(child => {  
  // 判断 child 是否是对象类型  
  if (child instanceof Object) {  
    // 如果是 什么都不需要做 直接返回即可  
    return child  
  } else {  
    // 如果不是对象就是文本 手动调用 createElement 方法将文本转换为 Virtual DOM  
    return createElement("text", { textContent: child })  
  }  
})  
return {  
  type,  
  props,  
  children: childElements  
}
```

```
▼ {type: "div", props: {...}, children: Array(10)} ⓘ
  ▼ children: Array(10)
    ▶ 0: {type: "h1", props: null, children: Array(1)}
    ▶ 1: {type: "h2", props: null, children: Array(1)}
    ▶ 2: {type: "div", props: null, children: Array(2)}
    ▼ 3:
      ▼ children: Array(1)
        ▼ 0:
          ▶ children: []
          ▶ props: {textContent: "(观察：这个将会被改变)"}
            type: "text"
          ▶ __proto__: Object
          length: 1
        ▶ __proto__: Array(0)
      props: null
      type: "h3"
      ▶ __proto__: Object
    ▼ 4:
      ▶ children: []
      ▼ props:
        textContent: false
        ▶ __proto__: Object
        type: "text"
        ▶ __proto__: Object
      ▶ 5: {type: "div", props: null, children: Array(1)}
      ▶ 6: {type: "span", props: null, children: Array(1)}
      ▶ 7: {type: "button", props: {...}, children: Array(1)}
      ▶ 8: {type: "h3", props: null, children: Array(1)}
      ▶ 9: {type: "text", props: {...}, children: Array(0)}
      length: 10
      ▶ __proto__: Array(0)
    ▶ props: {className: "container"}
      type: "div"
    ▶ __proto__: Object
```

通过观察返回的 Virtual DOM，文本节点已经被转化成了对象类型的 Virtual DOM，但是布尔值也被当做文本节点被转化了，在 JSX 中，如果 Virtual DOM 被转化为了布尔值或者null，是不应该被更新到真实 DOM 中的，所以接下来要做的事情就是清除 Virtual DOM 中的布尔值和null。

```
// 由于 map 方法无法从数据中刨除元素，所以此处将 map 方法更改为 reduce 方法
const childElements = [].concat(...children).reduce((result, child) => {
  // 判断子元素类型 刨除 null true false
  if (child !== null && child !== false && child !== true) {
    if (child instanceof Object) {
      result.push(child)
    } else {
      result.push(createElement("text", { textContent: child }))
    }
  }
}, [])
// 将需要保留的 Virtual DOM 放入 result 数组
return result
}, [])
```

在 React 组件中，可以通过 props.children 获取子元素，所以还需要将子元素存储在 props 对象中。

```
return {
  type,
  props: Object.assign({ children: childElements }, props),
  children: childElements
}
```

6. 渲染 Virtual DOM 对象为 DOM 对象

通过调用 render 方法可以将 Virtual DOM 对象更新为真实 DOM 对象。

在更新之前需要确定是否存在旧的 Virtual DOM，如果存在需要比对差异，如果不存在可以直接将 Virtual DOM 转换为 DOM 对象。

目前先只考虑不存在旧的 Virtual DOM 的情况，就是说先直接将 Virtual DOM 对象更新为真实 DOM 对象。

```
// render.js
export default function render(virtualDOM, container, oldDOM = container.firstChild) {
  // 在 diff 方法内部判断是否需要对比 对比也好 不对比也好 都在 diff 方法中进行操作
  diff(virtualDOM, container, oldDOM)
}
```

```
// diff.js
import mountElement from "./mountElement"

export default function diff(virtualDOM, container, oldDOM) {
  // 判断 oldDOM 是否存在
  if (!oldDOM) {
    // 如果不存在 不需要对比 直接将 Virtual DOM 转换为真实 DOM
    mountElement(virtualDOM, container)
  }
}
```

在进行 virtual DOM 转换之前还需要确定 Virtual DOM 的类 Component VS Native Element。

类型不同需要做不同的处理 如果是 Native Element 直接转换。

如果是组件 还需要得到组件实例对象 通过组件实例对象获取组件返回的 virtual DOM 然后再进行转换。

目前先只考虑 Native Element 的情况。

```
// mountElement.js
import mountNativeElement from "./mountNativeElement"

export default function mountElement(virtualDOM, container) {
  // 通过调用 mountNativeElement 方法转换 Native Element
  mountNativeElement(virtualDOM, container)
}

// mountNativeElement.js
import createDOMElement from "./createDOMElement"

export default function mountNativeElement(virtualDOM, container) {
  const newElement = createDOMElement(virtualDOM)
  container.appendChild(newElement)
}
```

```
// createDOMElement.js
import mountElement from "./mountElement"
import updateElementNode from "./updateElementNode"

export default function createDOMElement(virtualDOM) {
  let newElement = null
  if (virtualDOM.type === "text") {
    // 创建文本节点
    newElement = document.createTextNode(virtualDOM.props.textContent)
  } else {
    // 创建元素节点
    newElement = document.createElement(virtualDOM.type)
    // 更新元素属性
    updateElementNode(newElement, virtualDOM)
  }
  // 递归渲染子节点
  virtualDOM.children.forEach(child => {
    // 因为不确定子元素是 NativeElement 还是 Component 所以调用 mountElement 方法进行确定
    mountElement(child, newElement)
  })
  return newElement
}
```

7. 为元素节点添加属性

```
// createDOMElement.js
// 看看节点类型是文本类型还是元素类型
if (virtualDOM.type === "text") {
  // 创建文本节点 设置节点内容
  newElement = document.createTextNode(virtualDOM.props.textContent)
} else {
  // 根据 Virtual DOM type 属性值创建 DOM 元素
  newElement = document.createElement(virtualDOM.type)
  // 为元素设置属性
  updateElementNode(newElement, virtualDOM)
}
```

```
export default function updateElementNode(element, virtualDOM) {
  // 获取要解析的 VirtualDOM 对象中的属性对象
  const newProps = virtualDOM.props
  // 将属性对象中的属性名称放到一个数组中并循环数组
  Object.keys(newProps).forEach(propName => {
    const newPropsValue = newProps[propName]
    // 考虑属性名称是否以 on 开头 如果是就表示是个事件属性 onClick -> click
    if (propName.slice(0, 2) === "on") {
      const eventName = propName.toLowerCase().slice(2)
      element.addEventListener(eventName, newPropsValue)
      // 如果属性名称是 value 或者 checked 需要通过 [] 的形式添加
    } else if (propName === "value" || propName === "checked") {
      element[propName] = newPropsValue
      // 刨除 children 因为它是子元素 不是属性
    } else if (propName !== "children") {
      // className 属性单独处理 不直接在元素上添加 class 属性是因为 class 是 JavaScript 中的关键字
      if (propName === "className") {
        element.setAttribute("class", newPropsValue)
      } else {
        // 普通属性
        element.setAttribute(propName, newPropsValue)
      }
    }
  })
}
```

8. 渲染组件

8.1 函数组件

在渲染组件之前首先要明确的是，组件的 Virtual DOM 类型值为函数，函数组件和类组件都是这样的。

```
// 原始组件
const Heart = () => <span>&hearts;</span>
```

```
<Heart />
```

```
// 组件的 Virtual DOM
{
  type: f function() {},
  props: {}
  children: []
}
```

在渲染组件时，要先将 Component 与 Native Element 区分开，如果是 Native Element 可以直接开始渲染，如果是组件，特别处理。

```
// mountElement.js
export default function mountElement(virtualDOM, container) {
  // 无论是类组件还是函数组件 其实本质上都是函数
  // 如果 Virtual DOM 的 type 属性值为函数 就说明当前这个 Virtual DOM 为组件
  if (isFunction(virtualDOM)) {
    // 如果是组件 调用 mountComponent 方法进行组件渲染
    mountComponent(virtualDOM, container)
  } else {
    mountNativeElement(virtualDOM, container)
  }
}

// Virtual DOM 是否为函数类型
export function isFunction(virtualDOM) {
  return virtualDOM && typeof virtualDOM.type === "function"
}
```

在 mountComponent 方法中再进行函数组件和类型的区分，然后再分别进行处理。

```
// mountComponent.js
import mountNativeElement from './mountNativeElement'

export default function mountComponent(virtualDOM, container) {
  // 存放组件调用后返回的 Virtual DOM 的容器
  let nextVirtualDOM = null
  // 区分函数型组件和类组件
  if (isFunctionComponent(virtualDOM)) {
    // 函数组件 调用 buildFunctionalComponent 方法处理函数组件
    nextVirtualDOM = buildFunctionalComponent(virtualDOM)
  } else {
    // 类组件
  }
  // 判断得到的 Virtual Dom 是否是组件
  if (isFunction(nextVirtualDOM)) {
    // 如果是组件 继续调用 mountComponent 解剖组件
    mountComponent(nextVirtualDOM, container)
  } else {
    // 如果是 Navtive Element 就去渲染
    mountNativeElement(nextVirtualDOM, container)
  }
}

// Virtual DOM 是否为函数型组件
// 条件有两个：1. Virtual DOM 的 type 属性值为函数 2. 函数的原型对象中不能有render方法
// 只有类组件的原型对象中有render方法
export function isFunctionComponent(virtualDOM) {
  const type = virtualDOM && virtualDOM.type
  return (
    type && isFunction(virtualDOM) && !(type.prototype && type.prototype.render)
  )
}

// 函数组件处理
function buildFunctionalComponent(virtualDOM) {
  // 通过 Virtual DOM 中的 type 属性获取到组件函数并调用
  // 调用组件函数时将 Virtual DOM 对象中的 props 属性传递给组件函数 这样在组件中就可以通过 props 属性获取数据了
  // 组件返回要渲染的 Virtual DOM
  return virtualDOM && virtualDOM.type(virtualDOM.props || {})
}
```

8.2 类组件

类组件本身也是 Virtual DOM，可以通过 Virtual DOM 中的 type 属性值确定当前要渲染的组件是类组件还是函数组件。

在确定当前要渲染的组件为类组件以后，需要实例化类组件得到类组件实例对象，通过类组件实例对象调用类组件中的 render 方法，获取组件要渲染的 Virtual DOM。

类组件需要继承 Component 父类，子类需要通过 super 方法将自身的 props 属性传递给 Component 父类，父类会将 props 属性挂载为父类属性，子类继承了父类，自己本身也就自然拥有props属性了。这样做的好处是当 props 发生更新后，父类可以根据更新后的 props 帮助子类更新视图。

假设以下代码就是我们要渲染的类组件：

```
class Alert extends TinyReact.Component {
  constructor(props) {
    // 将 props 传递给父类 子类继承父类的 props 子类自然就有 props 数据了
    // 否则 props 仅仅是 constructor 函数的参数而已
    // 将 props 传递给父类的好处是 当 props 发生更改时 父类可以帮助更新 props 更新组件视图
    super(props)
    this.state = {
      title: "default title"
    }
  }
  render() {
    return (
      <div>
        <h2>{this.state.title}</h2>
        <p>{this.props.message}</p>
      </div>
    )
  }
}
```

TinyReact.render(<Alert message="Hello React" />, root)


```
// Component.js 父类 Component 实现
export default class Component {
  constructor(props) {
    this.props = props
  }
}
```

在 mountComponent 方法中通过调用 buildStatefulComponent 方法得到类组件要渲染的 Virtual DOM

```
// mountComponent.js
export default function mountComponent(virtualDOM, container) {
  let nextVirtualDOM = null
  // 区分函数型组件和类组件
  if (isFunctionComponent(virtualDOM)) {
    // 函数组件
    nextVirtualDOM = buildFunctionalComponent(virtualDOM)
  } else {
    // 类组件
    nextVirtualDOM = buildStatefulComponent(virtualDOM)
  }
  // 判断得到的 Virtual Dom 是否是组件
  if (isFunction(nextVirtualDOM)) {
    mountComponent(nextVirtualDOM, container)
  } else {
    mountNativeElement(nextVirtualDOM, container)
  }
}

// 处理类组件
function buildStatefulComponent(virtualDOM) {
  // 实例化类组件 得到类组件实例对象 并将 props 属性传递进类组件
  const component = new virtualDOM.type(virtualDOM.props)
  // 调用类组件中的render方法得到要渲染的 Virtual DOM
  const nextVirtualDOM = component.render()
  // 返回要渲染的 Virtual DOM
  return nextVirtualDOM
}
```

9. Virtual DOM 比对

在进行 Virtual DOM 比对时，需要用到更新后的 Virtual DOM 和更新前的 Virtual DOM，更新后的 Virtual DOM 目前我们可以通过 render 方法进行传递，现在的问题是更新前的 Virtual DOM 要如何获取呢？

对于更新前的 Virtual DOM，对应的其实就是已经在页面中显示的真实 DOM 对象。既然是这样，那么我们在创建真实DOM对象时，就可以将 Virtual DOM 添加到真实 DOM 对象的属性中。在进行 Virtual DOM 对比之前，就可以通过真实 DOM 对象获取其对应的 Virtual DOM 对象了，其实就是通过 render方法的第三个参数获取的，container.firstChild。

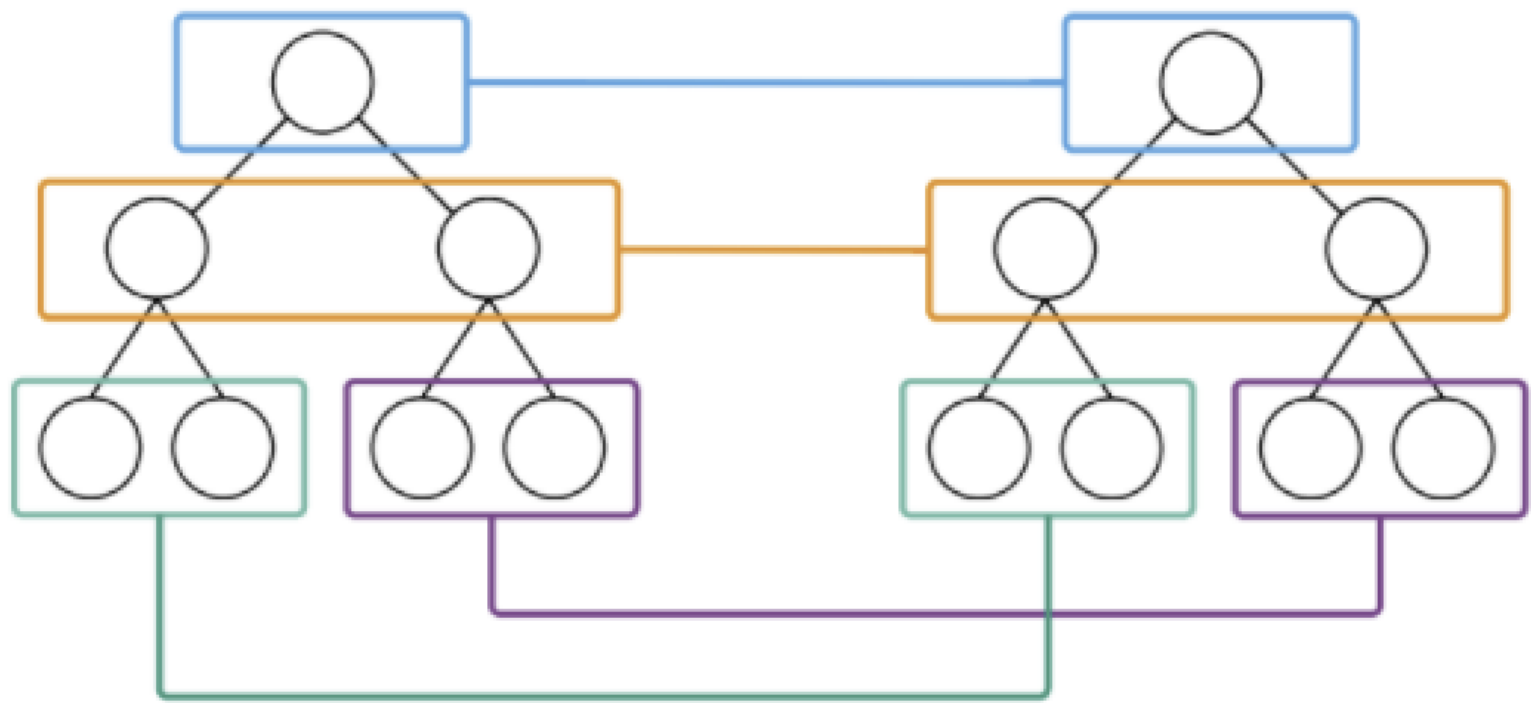
在创建真实 DOM 对象时为其添加对应的 Virtual DOM 对象

```
// mountElement.js
import mountElement from './mountElement'

export default function mountNativeElement(virtualDOM, container) {
  // 将 Virtual DOM 挂载到真实 DOM 对象的属性中 方便在对比时获取其 Virtual DOM
  newElement._virtualDOM = virtualDOM
}
```

Before

After



9.1 Virtual DOM 类型相同

Virtual DOM 类型相同，如果是元素节点，就对比元素节点属性是否发生变化，如果是文本节点就对比文本节点内容是否发生变化

要实现对比，需要先从已存在 DOM 对象中获取其对应的 Virtual DOM 对象。

```
// diff.js
// 获取未更新前的 Virtual DOM
const oldVirtualDOM = oldDOM && oldDOM._virtualDOM
```

判断 oldVirtualDOM 是否存在， 如果存在则继续判断要对比的 Virtual DOM 类型是否相同，如果类型相同判断节点类型是否是文本，如果是文本节点对比，就调用 updateTextNode 方法，如果是元素节点对比就调用 setAttributeForElement 方法

```
// diff.js
else if (oldVirtualDOM && virtualDOM.type === oldVirtualDOM.type) {
  if (virtualDOM.type === "text") {
    // 文本节点 对比文本内容是否发生变化
    updateTextNode(virtualDOM, oldVirtualDOM, oldDOM)
  } else {
    // 元素节点 对比元素属性是否发生变化
    setAttributeForElement(oldDOM, virtualDOM, oldVirtualDOM)
  }
}
```

updateTextNode 方法用于对比文本节点内容是否发生变化，如果发生变化则更新真实 DOM 对象中的内容，既然真实 DOM 对象发生了变化，还要将最新的 Virtual DOM 同步给真实 DOM 对象。

```
function updateTextNode(virtualDOM, oldVirtualDOM, oldDOM) {
  // 如果文本节点内容不同
  if (virtualDOM.props.textContent !== oldVirtualDOM.props.textContent) {
    // 更新真实 DOM 对象中的内容
    oldDOM.textContent = virtualDOM.props.textContent
  }
  // 同步真实 DOM 对应的 Virtual DOM
  oldDOM._virtualDOM = virtualDOM
}
```

setAttributeForElement 方法用于设置/更新元素节点属性

思路是先分别获取更新后的和更新前的 Virtual DOM 中的 props 属性，循环新 Virtual DOM 中的 props 属性，通过对比看一下新 Virtual DOM 中的属性值是否发生了变化，如果发生变化 需要将变化的值更新到真实 DOM 对象中

再循环未更新前的 Virtual DOM 对象，通过对比看看新的 Virtual DOM 中是否有被删除的属性，如果存在删除的属性 需要将 DOM 对象中对应的属性也删除掉

```
// updateNodeElement.js
export default function updateNodeElement(
  newElement,
  virtualDOM,
  oldVirtualDOM = {}
) {
  // 获取节点对应的属性对象
  const newProps = virtualDOM.props || {}
  const oldProps = oldVirtualDOM.props || {}
  Object.keys(newProps).forEach(propName => {
    // 获取属性值
    const newPropsValue = newProps[propName]
    const oldPropsValue = oldProps[propName]
    if (newPropsValue !== oldPropsValue) {
      // 判断属性是否是事件属性 onClick -> click
      if (propName.slice(0, 2) === "on") {
        // 事件名称
        const eventName = propName.toLowerCase().slice(2)
        // 为元素添加事件
        newElement.addEventListener(eventName, newPropsValue)
        // 删除原有的事件的事件处理函数
        if (oldPropsValue) {
          newElement.removeEventListener(eventName, oldPropsValue)
        }
      } else if (propName === "value" || propName === "checked") {
        newElement[propName] = newPropsValue
      } else if (propName !== "children") {
        if (propName === "className") {
          newElement.setAttribute("class", newPropsValue)
        } else {
          newElement.setAttribute(propName, newPropsValue)
        }
      }
    }
  })
}

// 判断属性被删除的情况
Object.keys(oldProps).forEach(propName => {
  const newPropsValue = newProps[propName]
  const oldPropsValue = oldProps[propName]
  if (!newPropsValue) {
    // 属性被删除了
    if (propName.slice(0, 2) === "on") {
      const eventName = propName.toLowerCase().slice(2)
      newElement.removeEventListener(eventName, oldPropsValue)
    } else if (propName !== "children") {
      newElement.removeAttribute(propName)
    }
  }
})
}
```

以上对比的仅仅是最上层元素，上层元素对比完成以后还需要递归对比子元素

```
else if (oldVirtualDOM && virtualDOM.type === oldVirtualDOM.type) {
  // 递归对比 Virtual DOM 的子元素
  virtualDOM.children.forEach((child, i) => {
    diff(child, oldDOM, oldDOM.childNodes[i])
  })
}
```

ul (1)

li (2)

li (5)

p (3)

p (4)

9.2 Virtual DOM 类型不同

当对比的元素节点类型不同时，就不需要继续对比了，直接使用新的 Virtual DOM 创建 DOM 对象，用新的 DOM 对象直接替换旧的 DOM 对象。当前这种情况要将组件刨除，组件要被单独处理。

```
// diff.js
else if (
  // 如果 Virtual DOM 类型不一样
  virtualDOM.type !== oldVirtualDOM.type &&
  // 并且 Virtual DOM 不是组件 因为组件要单独进行处理
  typeof virtualDOM.type !== "function"
) {
  // 根据 Virtual DOM 创建真实 DOM 元素
  const newDOMElement = createDOMElement(virtualDOM)
  // 用创建出来的真实 DOM 元素 替换旧的 DOM 元素
  oldDOM.parentNode.replaceChild(newDOMElement, oldDOM)
}
```

9.3 删除节点

删除节点发生在节点更新以后并且发生在同一个父节点下的所有子节点身上。

在节点更新完成以后，如果旧节点对象的数量多于新 VirtualDOM 节点的数量，就说明有节点需要被删除。

 old

1

23

34

4x

 new

1

3

4


```
// 获取就节点的数量
let oldChildNodes = oldDOM.childNodes
// 如果旧节点的数量多于要渲染的新节点的长度
if (oldChildNodes.length > virtualDOM.children.length) {
  for (
    let i = oldChildNodes.length - 1;
    i > virtualDOM.children.length - 1;
    i--
  ) {
    oldDOM.removeChild(oldChildNodes[i])
  }
}
```

9.4 类组件状态更新

以下代码是要更新状态的类组件，在类组件的 state 对象中有默认 title 状态，点击 change title 按钮调用 handleChange 方法，在 handleChange 方法中调用 this.setState 方法更改 title 的状态值。

```
class Alert extends TinyReact.Component {
  constructor(props) {
    super(props)
    this.state = {
      title: "default title"
    }
    // 更改 handleChange 方法中的 this 指向 让 this 指向类实例对象
    this.handleChange = this.handleChange.bind(this)
  }
  handleChange() {
    // 调用父类中的 setState 方法更改状态
    this.setState({
      title: "changed title"
    })
  }
  render() {
    return (
      <div>
        <h2>{this.state.title}</h2>
        <p>{this.props.message}</p>
        <button onClick={this.handleChange}>change title</button>
      </div>
    )
  }
}
```

setState 方法是定义在父类 Component 中的，该方法的作用是更改子类的 state，产生一个全新的 state 对象。

```
// Component.js
export default class Component {
  constructor(props) {
    this.props = props
  }
  setState (state) {
    // setState 方法被子类调用 此处this指向子类实例对象
    // 所以改变的是子类的 state 对象
    this.state = Object.assign({}, this.state, state)
  }
}
```

现在子类已经可以调用父类的 setState 方法更改状态值了，当组件的 state 对象发生更改时，要调用 render 方法更新组件视图。

在更新组件之前，要使用更新的 Virtual DOM 对象和未更新的 Virtual DOM 进行对比找出更新的部分，达到 DOM 最小化操作的目的。

在 setState 方法中可以通过调用 this.render 方法获取更新后的 Virtual DOM，由于 setState 方法被子类调用，this 指向子类，所以此处调用的是子类的 render 方法。

```
// Component.js
setState(state) {
  // setState 方法被子类调用 此处this指向子类
  // 所以改变的是子类的 state
  this.state = Object.assign({}, this.state, state)
  // 通过调用 render 方法获取最新的 Virtual DOM
  let virtualDOM = this.render()
}
```

要实现对比，还需要获取未更新前的 Virtual DOM，按照之前的经验，我们可以从 DOM 对象中获取其对应的 Virtual DOM 对象，未更新前的 DOM 对象实际上就是现在在页面中显示的 DOM 对象，我们只要能获取到这个 DOM 对象就可以获取到其对应的 Virtual DOM 对象了。

页面中的 DOM 对象要怎样获取呢？页面中的 DOM 对象是通过 mountNativeElement 方法挂载到页面中的，所以我们只需要在这个方法中调用 Component 类中的方法就可以将 DOM 对象保存在 Component 类中了。在子类调用 setState 方法的时候，在 setState 方法中再调用另一个获取 DOM 对象的方法就可以获取到之前保存的 DOM 对象了。

```
// Component.js
// 保存 DOM 对象的方法
setDOM(dom) {
  this._dom = dom
}
// 获取 DOM 对象的方法
getDOM() {
  return this._dom
}
```

接下来我们要研究一下在 mountNativeElement 方法中如何才能调用到 setDOM 方法，要调用 setDOM 方法，必须要得到类的实例对象，所以目前的问题就是如何在 mountNativeElement 方法中得到类的实例对象，这个类指的不是Component类，因为我们在代码中并不是直接实例化的Component类，而是实例化的它的子类，由于子类继承了父类，所以在子类的实例对象中也是可以调用到 setDOM 方法的。

mountNativeElement 方法接收最新的 Virtual DOM 对象，如果这个 Virtual DOM 对象是类组件产生的，在产生这个 Virtual DOM 对象时一定会先得到这个类的实例对象，然后再调用实例对象下面的 render 方法进行获取。我们可以在那个时候将类组件实例对象添加到 Virtual DOM 对象的属性中，而这个 Virtual DOM 对象最终会传递给 mountNativeElement 方法，这样我们就可以在 mountNativeElement 方法中获取到组件的实例对象了，既然类组件的实例对象获取到了，我们就可以调用 setDOM 方法了。

在 buildClassComponent 方法中为 Virtual DOM 对象添加 component 属性， 值为类组件的实例对象。

```
function buildClassComponent(virtualDOM) {
  const component = new virtualDOM.type(virtualDOM.props)
  const nextVirtualDOM = component.render()
  nextVirtualDOM.component = component
  return nextVirtualDOM
}
```

在 mountNativeElement 方法中获取组件实例对象，通过实例调用调用 setDOM 方法保存 DOM 对象，方便在对比时通过它获取它的 Virtual DOM 对象

```
export default function mountNativeElement(virtualDOM, container) {
  // 获取组件实例对象
  const component = virtualDOM.component
  // 如果组件实例对象存在
  if (component) {
    // 保存 DOM 对象
    component.setDOM(newElement)
  }
}
```

接下来在 `setState` 方法中就可以调用 `getDOM` 方法获取 DOM 对象了

```
setState(state) {
  this.state = Object.assign({}, this.state, state)
  let virtualDOM = this.render()
  // 获取页面中正在显示的 DOM 对象 通过它可以获取其对象的 Virtual DOM 对象
  let oldDOM = this.getDOM()
}
```

现在更新前的 Virtual DOM 对象和更新后的 Virtual DOM 对象就都已经获取到了，接下来还要获取到真实 DOM 对象父级容器对象，因为在调用 `diff` 方法进行对比的时候需要用到

```
setState(state) {
  this.state = Object.assign({}, this.state, state)
  let virtualDOM = this.render()
  let oldDOM = this.getDOM()
  // 获取真实 DOM 对象父级容器对象
  let container = oldDOM.parentNode
}
```

接下来就可以调用 `diff` 方法进行比对了，比对后会按照我们之前写好的逻辑进行 DOM 对象更新，我们就可以在页面中看到效果了

```
setState(state) {
  this.state = Object.assign({}, this.state, state)
  let virtualDOM = this.render()
  let oldDOM = this.getDOM()
  let container = oldDOM.parentNode
  // 比对
  diff(virtualDOM, container, oldDOM)
}
```

9.5 组件更新

在 `diff` 方法中判断要更新的 Virtual DOM 是否是组件。

如果是组件再判断要更新的组件和未更新前的组件是否是同一个组件，如果不是同一个组件就不需要做组件更新操作，直接调用 `mountElement` 方法将组件返回的 Virtual DOM 添加到页面中。

如果是同一个组件，就执行更新组件操作，其实就是将最新的 `props` 传递到组件中，再调用组件的`render`方法获取组件返回的最新的 Virtual DOM 对象，再将 Virtual DOM 对象传递给 `diff` 方法，让 `diff` 方法找出差异，从而将差异更新到真实 DOM 对象中。

在更新组件的过程中还要在不同阶段调用其不同的组件生命周期函数。

在 `diff` 方法中判断要更新的 Virtual DOM 是否是组件，如果是组件又分为多种情况，新增 `diffComponent` 方法进行处理

```
else if (typeof virtualDOM.type === "function") {
  // 要更新的是组件
  // 1) 组件本身的 virtualDOM 对象 通过它可以获取到组件最新的 props
  // 2) 要更新的组件的实例对象 通过它可以调用组件的生命周期函数 可以更新组件的 props 属性 可以获取到组件返回的最新的 Virtual DOM
  // 3) 要更新的 DOM 象 在更新组件时 需要在已有DOM对象的身上进行修改 实现DOM最小化操作 获取旧的 Virtual DOM 对象
  // 4) 如果要更新的组件和旧组件不是同一个组件 要直接将组件返回的 Virtual DOM 显示在页面中 此时需要 container 做为父级容器
  diffComponent(virtualDOM, oldComponent, oldDOM, container)
}
```

在 `diffComponent` 方法中判断要更新的组件是未更新前的组件是否是同一个组件

```
// diffComponent.js
export default function diffComponent(virtualDOM, oldComponent, oldDOM, container) {
  // 判断要更新的组件和未更新的组件是否是同一个组件 只需要确定两者使用的是否是同一个构造函数就可以了
  if (isSameComponent(virtualDOM, oldComponent)) {
    // 属同一个组件 做组件更新
  } else {
    // 不是同一个组件 直接将组件内容显示在页面中
  }
}
// virtualDOM.type 更新后的组件构造函数
// oldComponent.constructor 未更新前的组件构造函数
// 两者等价就表示是同一组件
function isSameComponent(virtualDOM, oldComponent) {
  return oldComponent && virtualDOM.type === oldComponent.constructor
}
```

如果不是同一个组件的话，就不需要执行更新组件的操作，直接将组件内容显示在页面中，替换原有内容

```
// diffComponent.js
else {
  // 不是同一个组件 直接将组件内容显示在页面中
  // 这里为 mountElement 方法新增了一个参数 oldDOM
  // 作用是在将 DOM 对象插入到页面前 将页面中已存在的 DOM 对象删除 否则无论是旧DOM对象还是新DOM对象都会显示在页面中
  mountElement(virtualDOM, container, oldDOM)
}
```

在 mountNativeElement 方法中删除原有的旧 DOM 对象

```
// mountNavtiveElement.js
export default function mountNativeElement(virtualDOM, container, oldDOM) {
  // 如果旧的DOM对象存在 删除
  if (oldDOM) {
    unmount(oldDOM)
  }
}
```

```
// unmount.js
export default function unmount(node) {
  node.remove()
}
```

如果是同一个组件的话，需要执行组件更新操作，需要调用组件生命周期函数

先在 Component 类中添加生命周期函数，子类要使用的话直接覆盖就可以

```
// Component.js
export default class Component {
  // 生命周期函数
  componentWillMount() {}
  componentDidMount() {}
  componentWillReceiveProps(nextProps) {}
  shouldComponentUpdate(nextProps, nextState) {
    return nextProps !== this.props || nextState !== this.state
  }
  componentWillUpdate(nextProps, nextState) {}
  componentDidUpdate(prevProps, prevState) {}
  componentWillUnmount() {}
}
```

新建 updateComponent 方法用于更新组件操作，并在 if 成立后调用

```
// diffComponent.js
if (isSameComponent(virtualDOM, oldComponent)) {
  // 属同一个组件 做组件更新
  updateComponent(virtualDOM, oldComponent, oldDOM, container)
}
```

在 updateComponent 方法中调用组件的生命周期函数，更新组件获取最新 Virtual DOM，最终调用 diff 方法进行更新

```
import diff from "./diff"

export default function updateComponent(
  virtualDOM,
  oldComponent,
  oldDOM,
  container
) {
  // 生命周期函数
  oldComponent.componentWillReceiveProps(virtualDOM.props)
  if (
    // 调用 shouldComponentUpdate 生命周期函数判断是否要执行更新操作
    oldComponent.shouldComponentUpdate(virtualDOM.props)
  ) {
    // 将未更新的 props 保存一份
    let prevProps = oldComponent.props
    // 生命周期函数
    oldComponent.componentWillUpdate(virtualDOM.props)
    // 更新组件的 props 属性 updateProps 方法定义在 Component 类型
    oldComponent.updateProps(virtualDOM.props)
    // 因为组件的 props 已经更新 所以调用 render 方法获取最新的 Virtual DOM
    const nextVirtualDOM = oldComponent.render()
    // 将组件实例对象挂载到 Virtual DOM 身上
    nextVirtualDOM.component = oldComponent
    // 调用diff方法更新视图
    diff(nextVirtualDOM, container, oldDOM)
    // 生命周期函数
    oldComponent.componentDidUpdate(prevProps)
  }
}

// Component.js
export default class Component {
  updateProps(props) {
    this.props = props
  }
}
```

10. ref 属性

为节点添加 ref 属性可以获取到这个节点的 DOM 对象，比如在 DemoRef 类中，为 input 元素添加了 ref 属性，目的是获取 input DOM 元素对象，在点击按钮时获取用户在文本框中输入的内容

```
class DemoRef extends TinyReact.Component {
  handle() {
    let value = this.input.value
    console.log(value)
  }
  render() {
    return (
      <div>
        <input type="text" ref={input => (this.input = input)} />
        <button onClick={this.handle.bind(this)}>按钮</button>
      </div>
    )
  }
}
```

实现思路是在创建节点时判断其 Virtual DOM 对象中是否有 ref 属性，如果有就调用 ref 属性中所存储的方法并且将创建出来的DOM对象作为参数传递给 ref 方法，这样在渲染组件节点的时候就可以拿到元素对象并将元素对象存储为组件属性了。

```
// createDOMElement.js
if (virtualDOM.props && virtualDOM.props.ref) {
  virtualDOM.props.ref(newElement)
}
```

在类组件的身上也可以添加 ref 属性，目的是获取组件的实例对象，比如下列代码中，在 DemoRef 组件中渲染了 Alert 组件，在 Alert 组件中添加了 ref 属性，目的是在 DemoRef 组件中获取 Alert 组件实例对象。


```
class DemoRef extends TinyReact.Component {
  handle() {
    let value = this.input.value
    console.log(value)
    console.log(this.alert)
  }
  componentDidMount() {
    console.log("componentDidMount")
  }
  render() {
    return (
      <div>
        <input type="text" ref={input => (this.input = input)} />
        <button onClick={this.handle.bind(this)}>按钮</button>
        <Alert ref={alert => (this.alert = alert)} />
      </div>
    )
  }
}
```

实现思路是在 mountComponent 方法中，如果判断了当前处理的是类组件，就通过类组件返回的 Virtual DOM 对象中获取组件实例对象，判断组件实例对象中的 props 属性中是否存在 ref 属性，如果存在就调用 ref 方法并且将组件实例对象传递给 ref 方法。

```
// mountComponent.js
let component = null
if (isFunctionalComponent(virtualDOM)) {}
else {
  // 类组件
  nextVirtualDOM = buildStatefulComponent(virtualDOM)
  // 获取组件实例对象
  component = nextVirtualDOM.component
}

// 如果组件实例对象存在的话
if (component) {
  // 判断组件实例对象身上是否有 props 属性 props 属性中是否有 ref 属性
  if (component.props && component.props.ref) {
    // 调用 ref 方法并传递组件实例对象
    component.props.ref(component)
  }
}
```

代码走到这，顺便处理一下组件挂载完成的生命周期函数

```
// 如果组件实例对象存在的话
if (component) {
  component.componentDidMount()
}
```

11. key 属性

在 React 中，渲染列表数据时通常会在被渲染的列表元素上添加 key 属性，key 属性就是数据的唯一标识，帮助 React 识别哪些数据被修改或者删除了，从而达到 DOM 最小化操作的目的。

key 属性不需要全局唯一，但是在同一个父节点下的兄弟节点之间必须是唯一的。

也就是说，在比对同一个父节点下类型相同的子节点时需要用到 key 属性。

11.1 节点对比

实现思路是在两个元素进行比对时，如果类型相同，就循环旧的 DOM 对象的子元素，查看其身上是否有key 属性，如果有就将这个子元素的 DOM 对象存储在一个 JavaScript 对象中，接着循环要渲染的 Virtual DOM 对象的子元素，在循环过程中获取到这个子元素的 key 属性，然后使用这个 key 属性到 JavaScript 对象中查找 DOM 对象，如果能够找到就说明这个元素是已经存在的，是不需要重新渲染的。如果通过key属性找不到这个元素，就说明这个元素是新增的是需要渲染的。

```
// diff.js
else if (oldVirtualDOM && virtualDOM.type === oldVirtualDOM.type) {
  // 将拥有key属性的元素放入 keyedElements 对象中
  let keyedElements = {}
  for (let i = 0, len = oldDOM.childNodes.length; i < len; i++) {
    let domElement = oldDOM.childNodes[i]
    if (domElement.nodeType === 1) {
      let key = domElement.getAttribute("key")
      if (key) {
        keyedElements[key] = domElement
      }
    }
  }
}

// diff.js
// 看一看是否有找到了拥有 key 属性的元素
let hasNoKey = Object.keys(keyedElements).length === 0

// 如果没有找到拥有 key 属性的元素 就按照索引进行比较
if (hasNoKey) {
  // 递归对比 Virtual DOM 的子元素
  virtualDOM.children.forEach((child, i) => {
    diff(child, oldDOM, oldDOM.childNodes[i])
  })
} else {
  // 使用key属性进行元素比较
  virtualDOM.children.forEach((child, i) => {
    // 获取要进行比对的元素的 key 属性
    let key = child.props.key
    // 如果 key 属性存在
    if (key) {
      // 到已存在的 DOM 元素对象中查找对应的 DOM 元素
      let domElement = keyedElements[key]
      // 如果找到元素就说明该元素已经存在 不需要重新渲染
      if (domElement) {
        // 虽然 DOM 元素不需要重新渲染 但是不能确定元素的位置就一定没有发生变化
        // 所以还要查看一下元素的位置
        // 看一下 oldDOM 对应的(i)子元素和 domElement 是否是同一个元素 如果不是就说明元素位置发生了变化
        if (oldDOM.childNodes[i] && oldDOM.childNodes[i] !== domElement) {
          // 元素位置发生了变化
          // 将 domElement 插入到当前元素位置的前面 oldDOM.childNodes[i] 就是当前位置
          // domElement 就被放入了当前位置
          oldDOM.insertBefore(domElement, oldDOM.childNodes[i])
        }
      } else {
        mountElement(child, oldDOM, oldDOM.childNodes[i])
      }
    }
  })
}

// mountNativeElement.js
if (oldDOM) {
  container.insertBefore(newElement, oldDOM)
} else {
  // 将转换之后的DOM对象放置在页面中
  container.appendChild(newElement)
}
```

11.2 节点卸载

在比对节点的过程中，如果旧节点的数量多于要渲染的新节点的数量就说明有节点被删除了，继续判断 keyedElements 对象中是否有元素，如果没有就使用索引方式删除，如果有就要使用 key 属性比对的方式进行删除。

实现思路是循环旧节点，在循环旧节点的过程中获取旧节点对应的 key 属性，然后根据 key 属性在新节点中查找这个旧节点，如果找到就说明这个节点没有被删除，如果没有找到，就说明节点被删除了，调用卸载节点的方法卸载节点即可。

```
// 获取就节点的数量
let oldChildNodes = oldDOM.childNodes
// 如果旧节点的数量多于要渲染的新节点的长度
if (oldChildNodes.length > virtualDOM.children.length) {
  if (hasNoKey) {
    for (
      let i = oldChildNodes.length - 1;
      i >= virtualDOM.children.length;
      i--
    ) {
      oldDOM.removeChild(oldChildNodes[i])
    }
  } else {
    for (let i = 0; i < oldChildNodes.length; i++) {
      let oldChild = oldChildNodes[i]
      let oldChildKey = oldChild._virtualDOM.props.key
      let found = false
      for (let n = 0; n < virtualDOM.children.length; n++) {
        if (oldChildKey === virtualDOM.children[n].props.key) {
          found = true
          break
        }
      }
      if (!found) {
        unmount(oldChild)
        i--
      }
    }
  }
}
```

卸载节点并不是说将节点直接删除就可以了，还需要考虑以下几种情况

1. 如果要删除的节点是文本节点的话可以直接删除
2. 如果要删除的节点由组件生成，需要调用组件卸载生命周期函数
3. 如果要删除的节点中包含了其他组件生成的节点，需要调用其他组件的卸载生命周期函数
4. 如果要删除的节点身上有 ref 属性，还需要删除通过 ref 属性传递给组件的 DOM 节点对象
5. 如果要删除的节点身上有事件，需要删除事件对应的事件处理函数


```
export default function unmount(dom) {
  // 获取节点对应的 virtualDOM 对象
  const virtualDOM = dom._virtualDOM
  // 如果要删除的节点时文本
  if (virtualDOM.type === "text") {
    // 直接删除节点
    dom.remove()
    // 阻止程序向下运行
    return
  }
  // 查看节点是否由组件生成
  let component = virtualDOM.component
  // 如果由组件生成
  if (component) {
    // 调用组件卸载生命周期函数
    component.componentWillUnmount()
  }

  // 如果节点具有 ref 属性 通过再次调用 ref 方法 将传递给组件的DOM对象删除
  if (virtualDOM.props && virtualDOM.props.ref) {
    virtualDOM.props.ref(null)
  }

  // 事件处理
  Object.keys(virtualDOM.props).forEach(propName => {
    if (propName.slice(0, 2) === "on") {
      const eventName = propName.toLowerCase().slice(2)
      const eventHandler = virtualDOM.props[propName]
      dom.removeEventListener(eventName, eventHandler)
    }
  })

  // 递归删除子节点
  if (dom.childNodes.length > 0) {
    for (let i = 0; i < dom.childNodes.length; i++) {
      unmount(dom.childNodes[i])
      i--
    }
  }

  dom.remove()
}
```

 联系方式