React原理剖析

个人介绍

前百度前端架构师 8年前端经验，react/vue源码级，擅长nodejs

课堂主题

1. 讲解React原理
2. 讲解redux设计理念和源码
3. mvvm设计理念

课堂目标

1. 虚拟dom 能回答domdiff 的具体逻辑
2. react的render过程

知识点

React核心api

react

```
const React = {
  Children: {
    map,
    forEach,
    count,
    toArray,
    only,
  },

  createRef,
  Component,
  PureComponent,

  createContext,
  forwardRef,
  lazy,
  memo,

  useCallback,
  useContext,
  useEffect,
  useImperativeHandle,
  useDebugValue,
  useLayoutEffect,
  useMemo,
  useReducer,
  useRef,
  useState,

  Fragment: REACT_FRAGMENT_TYPE,
  StrictMode: REACT_STRICT_MODE_TYPE,
  Suspense: REACT_SUSPENSE_TYPE,

  createElement: __DEV__ ? createElementWithValidation : createElement,
  cloneElement: __DEV__ ? cloneElementWithValidation : cloneElement,
  createFactory: __DEV__ ? createFactoryWithValidation : createFactory,
  isValidElement: isValidElement,

  version: ReactVersion,

  unstable_ConcurrentMode: REACT_CONCURRENT_MODE_TYPE,
  unstable_Profiler: REACT_PROFILER_TYPE,

  __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED: ReactSharedInternals,
};
```

```
// Note: some APIs are added with feature flags.
// Make sure that stable builds for open source
// don't modify the React object to avoid deopts.
// Also let's not expose their names in stable builds.

if (enableStableConcurrentModeAPIs) {
  React.ConcurrentMode = REACT_CONCURRENT_MODE_TYPE;
  React.Profiler = REACT_PROFILER_TYPE;
  React.unstable_ConcurrentMode = undefined;
  React.unstable_Profiler = undefined;
}

export default React;
```

react-dom 主要是render逻辑

核心精简后

```
let React = {
    createElement,
    Component,
    PureComponent
}
```

最核心的api

JSX

在线尝试

1. 为什么需要jsx
2. 怎么用
3. 原理

REACT 编辑器 显示JSX

```
class App extends React.Component {
  render() {
    return (
      <div>
        Hello {this.props.name}, I am {2 + 2} years old
      </div>
    )
  }
}

ReactDOM.render(
  <App name="React" />,
  mountNode
)
```

REACT 编辑器 显示JSX

```
class App extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name,
      ", I am ",
      2 + 2,
      " years old"
    )
  }
}

ReactDOM.render(React.createElement(App, { name: "React" }),
mountNode)
```

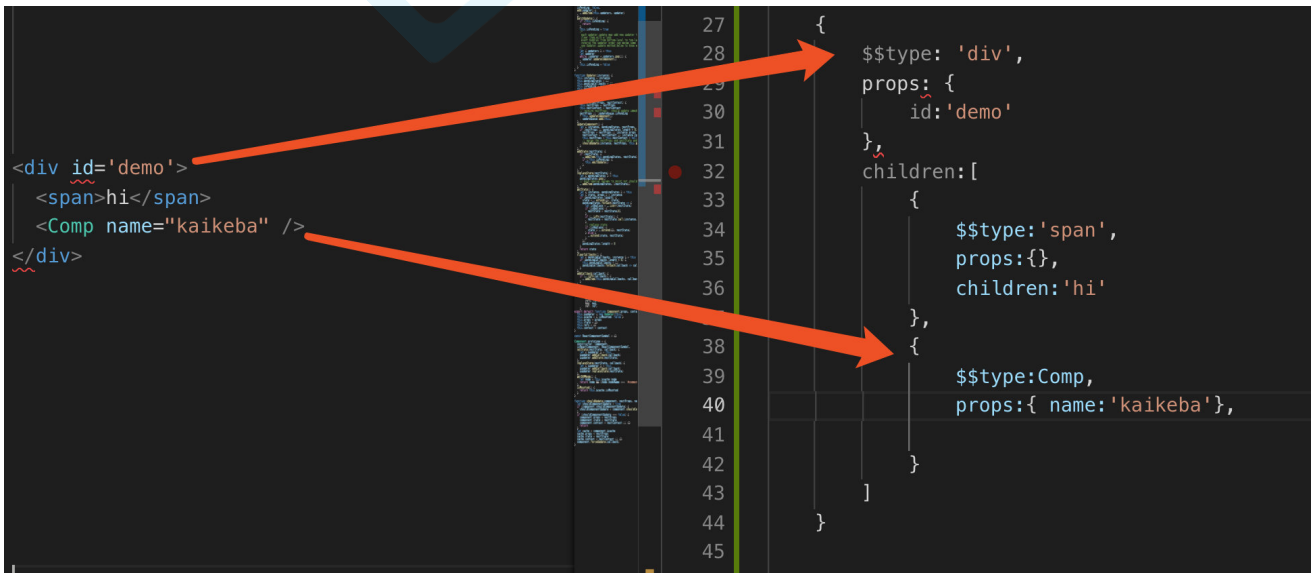
使用

```
function Comp(props){
    return <h2>hi {props.name}</h2>

}
ReactDOM.render(
  <div id='demo'>
    <span>hi</span>
    <Comp name="kaikeba" />
  </div>,
  mountNode
)
```

build后

```
function Comp(props) {
  return React.createElement(
    "h2",
    null,
    "hi ",
    props.name
  )
}
ReactDOM.render(React.createElement(
  "div",
  { id: "demo" },
  React.createElement(
    "span",
    null,
    "hi"
  ),
  React.createElement(Comp, { name: "kaikeba" })
), mountNode)
```

构建的dom 用js的对象，来描述dom树结构 一一对应q



三大接口，React.createElement, React.Component, ReactDom.render

```
create-react-app react08
```

删除src目录的内容，新建index.js

```
import React from './kreact'
import ReactDom from './kreact-dom'

ReactDom.render(
    <div id='demo'>
        嘿嘿
    </div>
    ,
    document.getElementById('root')
)
```

CreateElement

kreact.js

```
function createElement(type, props,...children){

    return {type,props,children}

}


export default {createElement}
```

kvdom是和虚拟dom相关的代码 所以虚拟dom 就是用js的对象描述一个dom树

```
export function createVnode(vtype, type, props) {
    let vnode = {
        vtype: vtype,
        type: type,
        props: props
    }
    return vnode
}
```

render

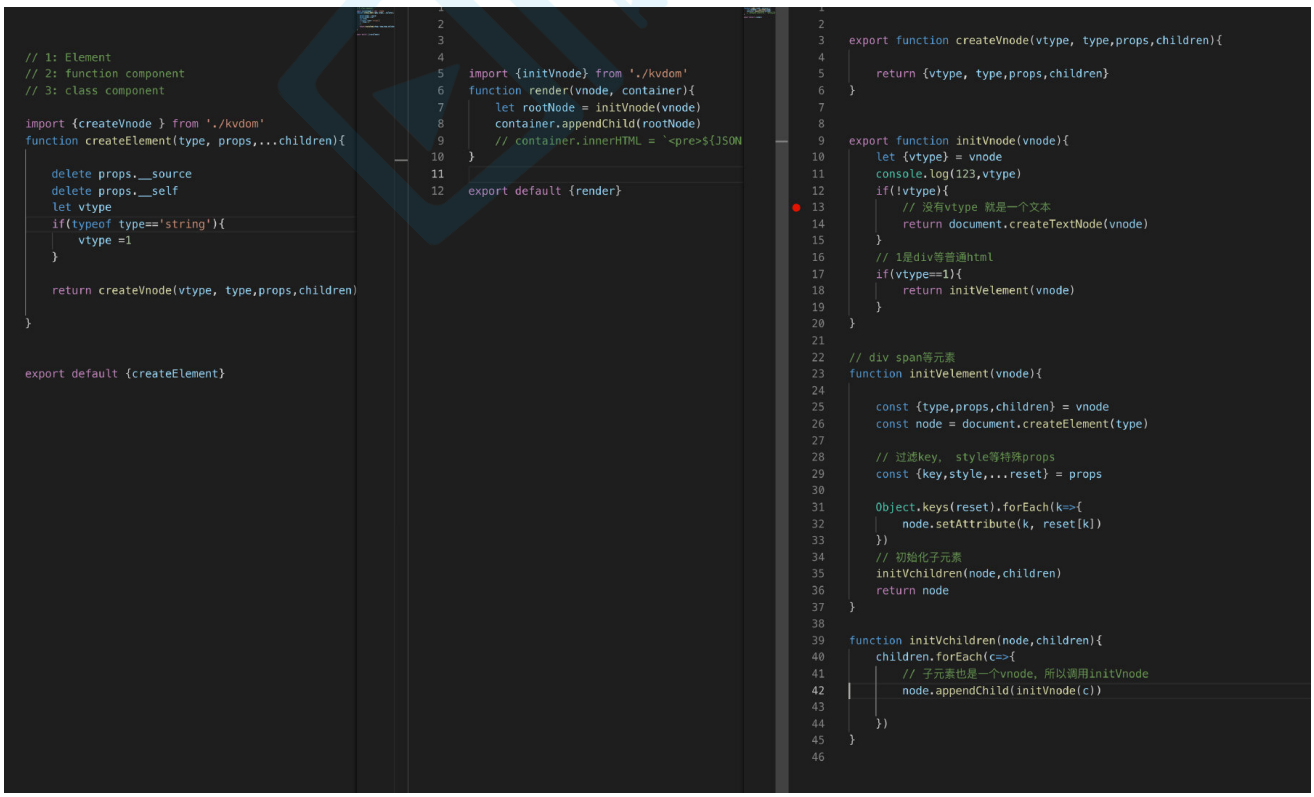
kreact-dom提供了render这一个函数，提供渲染 我们先渲染出vdom

```
function render(vnode, container){
    container.innerHTML = `<pre>${JSON.stringify(vnode,null,2)}</pre>`
}

export default {render}
```

页面效果

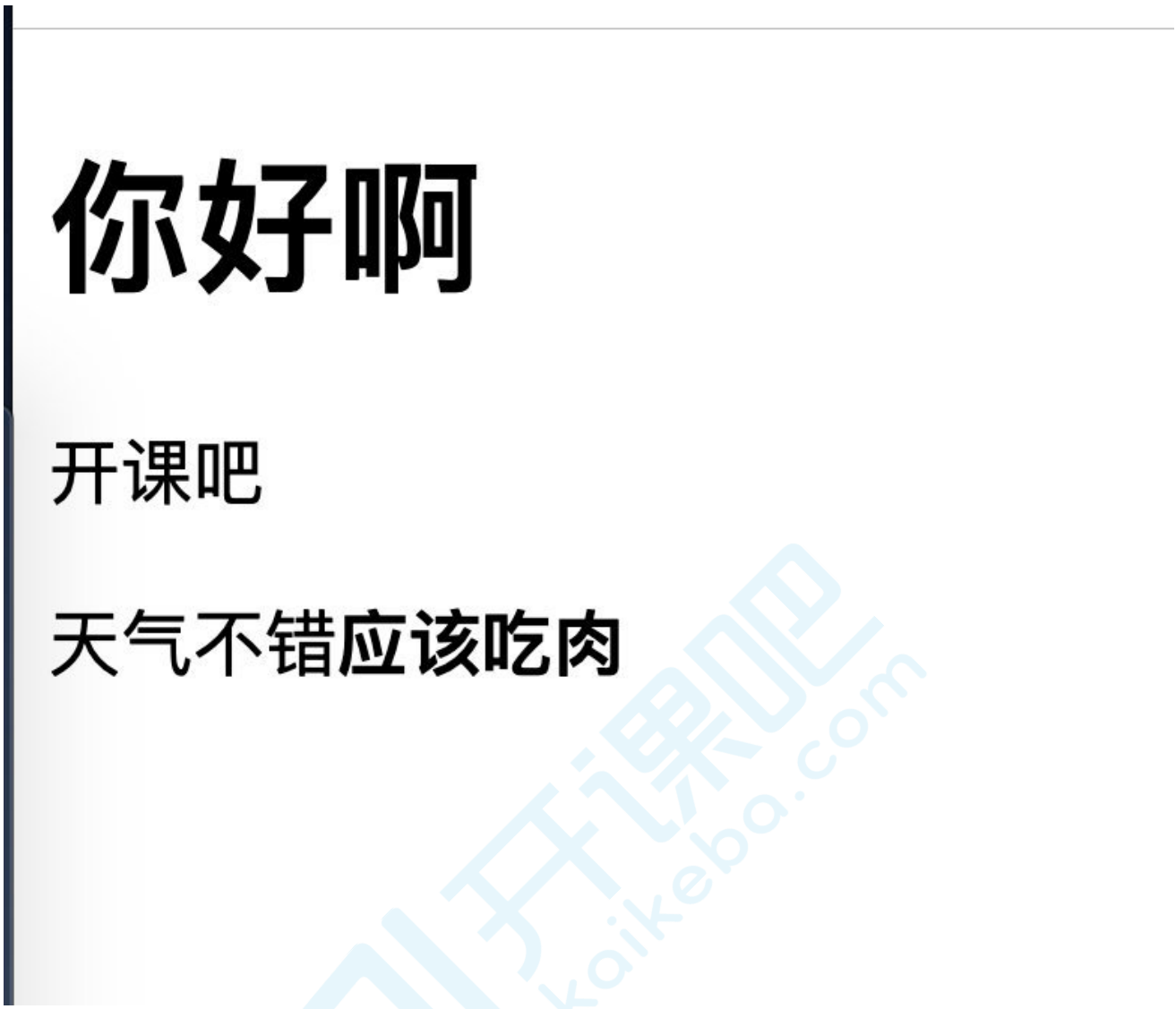

```
{
  "type": "div",
  "props": {
    "id": "demo",
    "__source": {
      "fileName": "/Users/woniuppp/work/react-lesson/react08/src/index.js",
      "lineNumber": 5
    }
  },
  "children": [
    "嘿嘿"
  ]
}
```


总共给createElememnt的type有三种组件类型，1：dom组件， 2. 函数式组件， 3. class组件，使用vtype属性标识， 并且抽离vdom相关代码到kvdom.js



尝试一下多个子元素递归

```
import React from './kreact'
import ReactDom from './kreact-dom'

ReactDom.render(
    <div id='demo'>
        <h1>你好啊</h1>
        <p>开课吧</p>
        <p>天气不错<strong>应该吃肉</strong></p>
    </div>
    ,
    document.getElementById('root')
)
```



dom搞定，开始搞组件，就是Component

Component

```
import React from './kreact'
import ReactDom from './kreact-dom'


function App1(props){
    return <h2>你好啊 {props.name}</h2>
}
class App extends React.Component{
    constructor(props){
        super(props)
    }

    render(){
        return <h2>你好啊 {this.props.name}</h2>
```

```
    }
}


ReactDom.render(
    <div id='demo'>
        <p>开课吧</p>
        <p>天气不错<strong>应该吃肉</strong></p>
        <App1 name='函数组件'/>
        <App name='class组件'/>
    </div>
    ,
    document.getElementById('root')
)
```

非常薄的一层封装，主要就是setState github 所以现在只是一个占位符

```
// 1: Element
// 2: function component
// 3: class component

import {createVnode } from './kvdom'
function createElement(type, props,...children){

    delete props.__source
    delete props.__self
    let vtype
    if(typeof type=='string'){
        vtype =1
    }else if(typeof type=='function'){
        // class组件
        if(type.isClassComponent){
            vtype= 3
        }else{
            // 函数组件
            vtype= 2
        }
    }
    return createVnode(vtype, type,props,children)
}

class Component {
    // 这个组件来区分是不是class组件
    static isClassComponent = true
    constructor(props){
        this.props = props
        this.state = {}
    }
}

export default {createElement, Component}
```

```
import React from './kreact'
import ReactDom from './kreact-dom'


function App1(props){
    return <h2>你好啊 {props.name}</h2>
}
class App extends React.Component{
    constructor(props){
        super(props)
    }

    render(){
        return <h2>你好啊 {this.props.name}</h2>
    }
}


ReactDom.render(
    <div id='demo'>
        <p>开课吧</p>
        <p>天气不错<strong>应该吃肉</strong></p>
        <App1 name='函数组件'/>
        <App name='class组件'/>
    </div>
    ,
    document.getElementById('root')
)
```

vdom

```
export function createVnode(vtype, type,props,children){

    return {vtype, type,props,children}
}


export function initVnode(vnode){
    let {vtype} = vnode
    console.log(123,vtype)
    if(!vtype){
        // 没有vtype 就是一个文本
        return document.createTextNode(vnode)
    }
    // 1是div等普通html
    if(vtype==1){
        return initVelement(vnode)
    }else if(vtype==2){
        return initFuncComp(vnode)
```

```
    }else if(vtype==3){
        return initClassComp(vnode)
    }
}

// div span等元素
function initVelement(vnode){

    const {type,props,children} = vnode
    const node = document.createElement(type)

    // 过滤key,  style等特殊props
    const {key,style,...reset} = props

    Object.keys(reset).forEach(k=>{
        node.setAttribute(k, reset[k])
    })
    // 初始化子元素
    initVchildren(node,children)
    return node
}

function initVchildren(node,children){
    children.forEach(c=>{
        // 子元素也是一个vnode，所以调用initVnode
        node.appendChild(initVnode(c))

    })
}

function initFuncComp(vnode){
    let {type,props} = vnode
    let newNode = type(props)
    return initVnode(newNode)

}


function initClassComp(vnode){
    const {type} = vnode
    let component = new type(vnode.props)
    let newNode = component.render()

    return initVnode(newNode)
}
```

开课吧

天气不错应该吃肉

你好啊 函数组件

你好啊 class组件

PureComponent

继承Component，主要是设置了shouldComponentUpdate生命周期

```
import shallowEqual from './shallowEqual'
import Component from './Component'

export default function PureComponent(props, context) {
    Component.call(this, props, context)
}

PureComponent.prototype = Object.create(Component.prototype)
PureComponent.prototype.constructor = PureComponent
PureComponent.prototype.isPureReactComponent = true
PureComponent.prototype.shouldComponentUpdate = shallowCompare

function shallowCompare(nextProps, nextState) {
    return !shallowEqual(this.props, nextProps) ||
            !shallowEqual(this.state, nextState)
}
```

setState

class的特点，就是可以setState，算是学习React中最重要的api

```
class App extends React.Component{
    constructor(props){
        super(props)
        this.state ={
            num:1
        }
```

```
    }
    componentDidMount(){
        setInterval(()=>{
            this.setState({
                num:this.state.num+1
            })
        })
    }
    render(){
        return <div>
            <h2>你好啊 {this.props.name} </h2>
            {this.state.num}
        </div>
    }
}
```

setState并没有直接操作去渲染，而是执行了一个异步的updater队列 我们使用一个类来专门管理

```
export let updateQueue = {
    updaters: [],
    isPending: false,
    add(updater) {
        _.addItem(this.updaters, updater)
    },
    batchUpdate() {
        if (this.isPending) {
            return
        }
        this.isPending = true
        let { updaters } = this
        let updater
        while (updater = updaters.pop()) {
            updater.updateComponent()
        }
        this.isPending = false
    }
}

function Updater(instance) {
    this.instance = instance
    this.pendingStates = []
    this.pendingCallbacks = []
    this.isPending = false
    this.nextProps = this.nextContext = null
    this.clearCallbacks = this.clearCallbacks.bind(this)
}

Updater.prototype = {
    emitUpdate(nextProps, nextContext) {
        this.nextProps = nextProps
        this.nextContext = nextContext
```

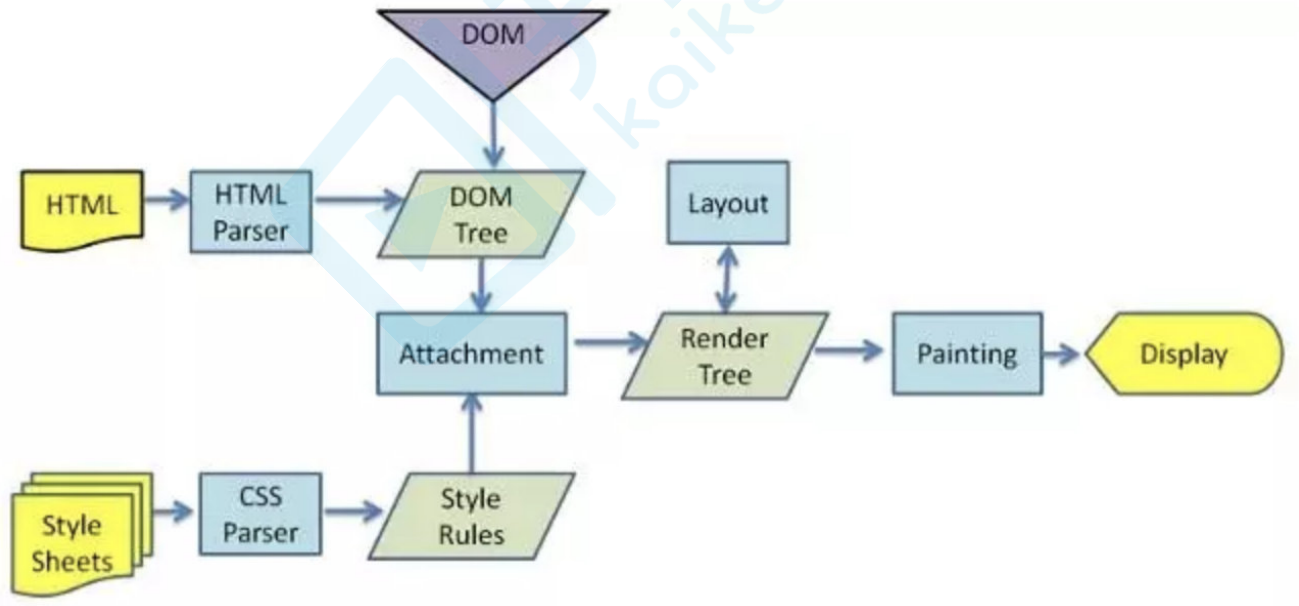
```
        // receive nextProps!! should update immediately
        nextProps || !updateQueue.isPending
        ? this.updateComponent()
        : updateQueue.add(this)
    },
    updateComponent() {
        let { instance, pendingStates, nextProps, nextContext } = this
        if (nextProps || pendingStates.length > 0) {
            nextProps = nextProps || instance.props
            nextContext = nextContext || instance.context
            this.nextProps = this.nextContext = null
            // merge the nextProps and nextState and update by one time
            shouldUpdate(instance, nextProps, this.getState(), nextContext,
this.clearCallbacks)
        }
    },
    addState(nextState) {
        if (nextState) {
            _.addItem(this.pendingStates, nextState)
            if (!this.isPending) {
                this.emitUpdate()
            }
        }
    },
    replaceState(nextState) {
        let { pendingStates } = this
        pendingStates.pop()
        // push special params to point out should replace state
        _.addItem(pendingStates, [nextState])
    },
    getState() {
        let { instance, pendingStates } = this
        let { state, props } = instance
        if (pendingStates.length) {
            state = _.extend({}, state)
            pendingStates.forEach(nextState => {
                let isReplace = _.isArr(nextState)
                if (isReplace) {
                    nextState = nextState[0]
                }
                if (_.isFn(nextState)) {
                    nextState = nextState.call(instance, state, props)
                }
                // replace state
                if (isReplace) {
                    state = _.extend({}, nextState)
                } else {
                    _.extend(state, nextState)
                }
            })
            pendingStates.length = 0
        }
        return state
```

```
    },
    clearCallbacks() {
        let { pendingCallbacks, instance } = this
        if (pendingCallbacks.length > 0) {
            this.pendingCallbacks = []
            pendingCallbacks.forEach(callback => callback.call(instance))
        }
    },
    addCallback(callback) {
        if (_.isFn(callback)) {
            _.addItem(this.pendingCallbacks, callback)
        }
    }
}
```

虚拟dom

1. 为什么需要虚拟dom
2. 传统的dom
3. 虚拟dom

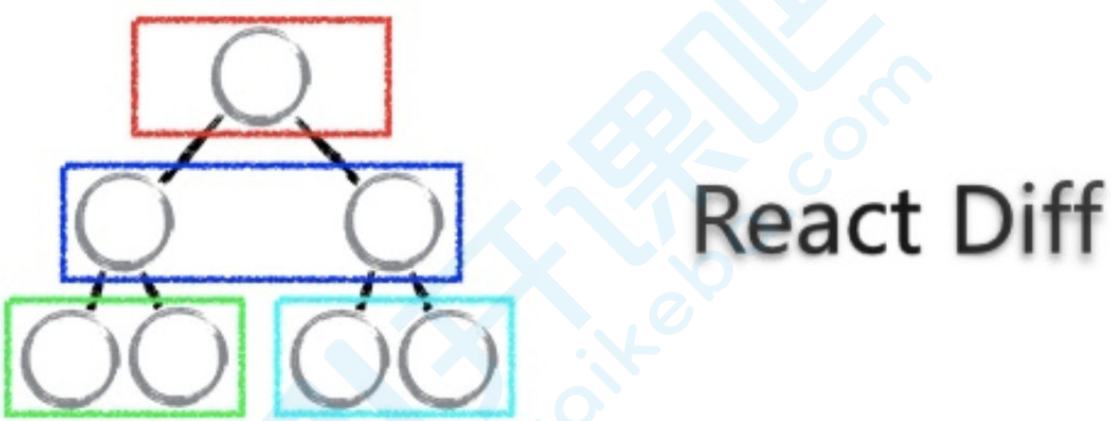
传统dom渲染逻辑



```
var div = document.createElement('div')
var str = ""
for (var key in div) {
  str = str + key + " "
}
console.log(str)
```

```
align title lang translate dir dataset hidden tabIndex accessKey draggable spellcheck contentEditable isContentEditable offsetParent offsetTop offsetLeft        VM1696:
offsetWidth offsetHeight style innerText outerText webkitdropzone onabort onblur oncancel oncanplay oncanplaythrough onchange onclick onclose oncontextmenu oncuechange
ondblclick ondrag ondragend ondragenter ondragleave ondragover ondragstart ondrop ondurationchange onemptied onended onerror onfocus oninput oninvalid onkeydown
onkeypress onkeyup onload onloadeddata onloadedmetadata onloadstart onmousedown onmouseenter onmouseleave onmousemove onmouseout onmouseover onmouseup onmousewheel
onpause onplay onplaying onprogress onratechange onreset onresize onscroll onseeked onseeking onselect onshow onstalled onsubmit onsuspend ontimeupdate ontoggle
onvolumechange onwaiting click focus blur onautocomplete onautocompleteerror namespaceURI prefix localName tagName id className classList attributes innerHTML outerHTML
shadowRoot scrollTop scrollLeft scrollWidth scrollHeight clientTop clientLeft clientWidth clientHeight onbeforecopy onbeforecut onbeforepaste oncopy oncut onpaste
onsearch onselectstart onwheel onwebkitfullscreenchange onwebkitfullscreenerror previousElementSibling nextElementSibling children firstElementChild lastElementChild
childElementCount hasAttributes getAttribute getAttributeNS setAttribute setAttributeNS removeAttribute removeAttributeNS hasAttribute hasAttributeNS getAttributeNode
getAttributeNodeNS setAttributeNode setAttributeNodeNS removeAttributeNode closest matches getElementsByTagName getElementsByTagNameNS getElementsByClassName
insertAdjacentHTML createShadowRoot getDestinationInsertionPoints requestPointerLock getClientRects getBoundingClientRect scrollIntoView insertAdjacentElement
insertAdjacentText scrollIntoViewIfNeeded webkitMatchesSelector animate remove webkitRequestFullScreen webkitRequestFullscreen querySelector querySelectorAll prepend
append before after replaceWith nodeType nodeName baseURI ownerDocument parentNode parentElement childNodes firstChild lastChild previousSibling nextSibling nodeValue
textContent hasChildNodes normalize cloneNode isEqualNode compareDocumentPosition contains lookupPrefix lookupNamespaceURI isDefaultNamespace insertBefore appendChild
replaceChild removeChild isSameNode ELEMENT_NODE ATTRIBUTE_NODE TEXT_NODE CDATA_SECTION_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_POSITION_FOLLOWING
DOCUMENT_POSITION_CONTAINS DOCUMENT_POSITION_CONTAINED_BY DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC addEventListener removeEventListener dispatchEvent
```

diff算法

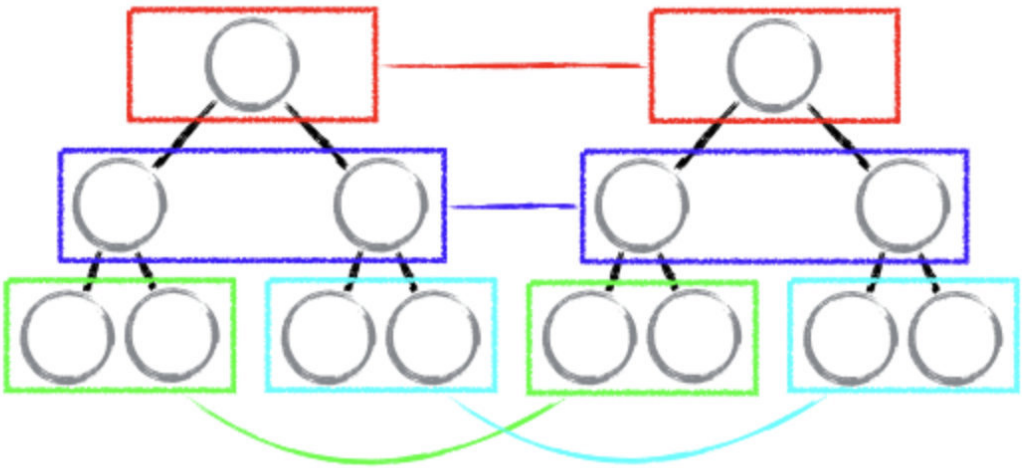


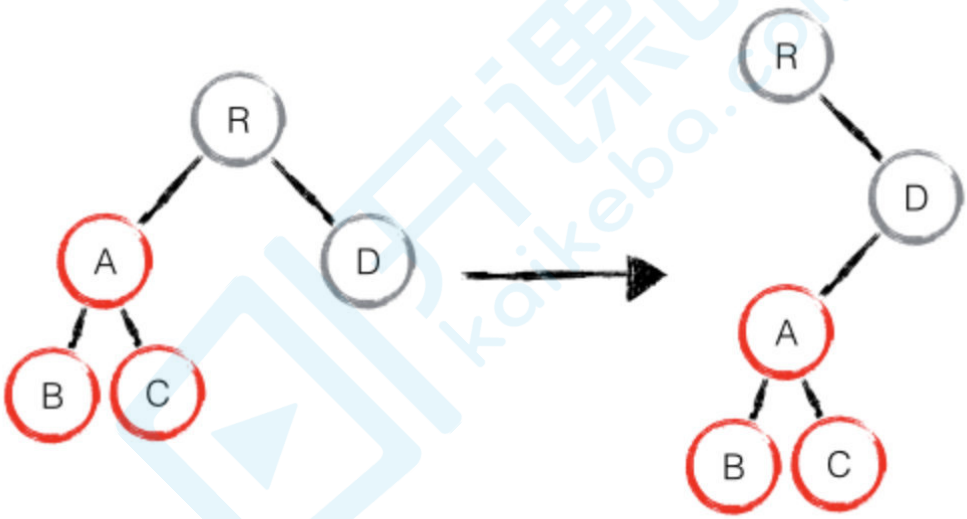
diff 策略

1. Web UI 中 DOM 节点跨层级的移动操作特别少，可以忽略不计。
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。
3. 对于同一层级的一组子节点，它们可以通过唯一 id 进行区分。

基于以上三个前提策略，React 分别对 tree diff、component diff 以及 element diff 进行算法优化，事实也证明这三个前提策略是合理且准确的，它保证了整体界面构建的性能。

- **tree diff**
- **component diff**
- **element diff** div span

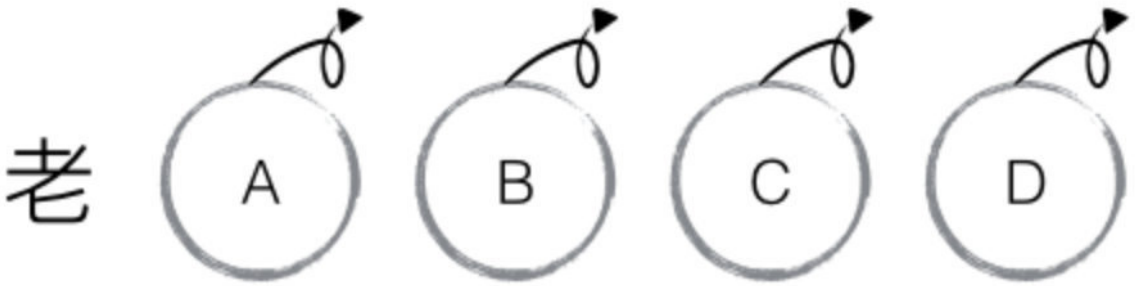


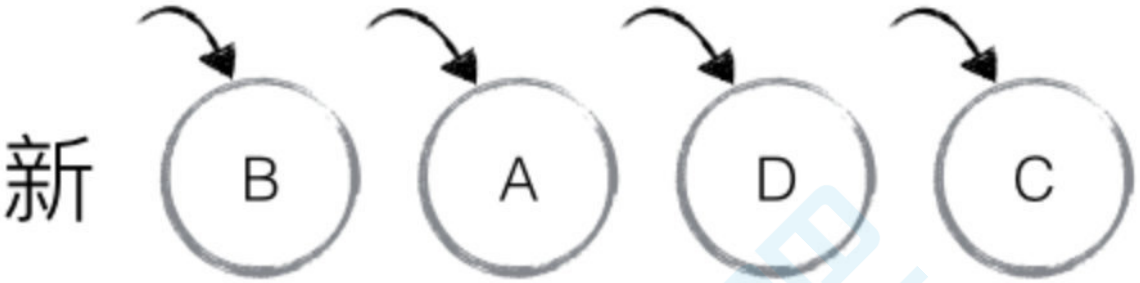


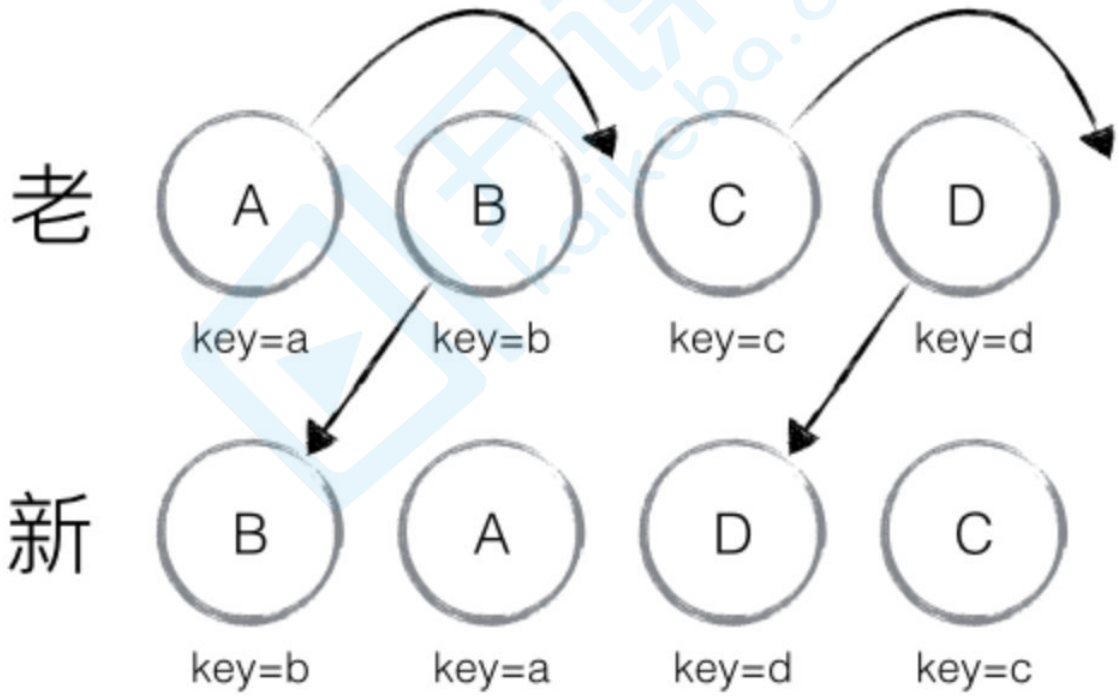
element diff

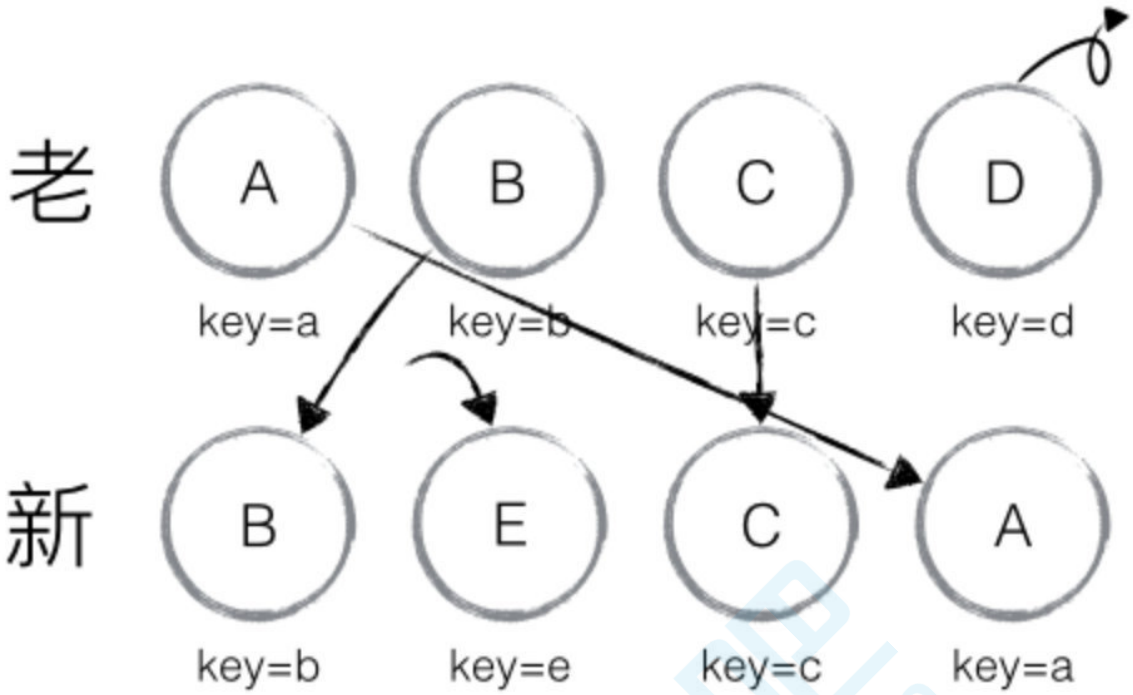
当节点处于同一层级时，React diff 提供了三种节点操作，分别为：**INSERT_MARKUP**（插入）、**MOVE_EXISTING**（移动）和 **REMOVE_NODE**（删除）。

- **INSERT_MARKUP**，新的 component 类型不在老集合里， 即是全新的节点，需要对新节点执行插入操作。
- **MOVE_EXISTING**，在老集合有新 component 类型，且 element 是可更新的类型，generateComponentChildren 已调用 receiveComponent，这种情况下 prevChild=nextChild，就需要做移动操作，可以复用以前的 DOM 节点。
- **REMOVE_NODE**，老 component 类型，在新集合里也有，但对应的 element 不同则不能直接复用和更新，需要执行删除操作，或者老 component 不在新集合里的，也需要执行删除操作。


老
A
B
C
D


新
B
A
D
C


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

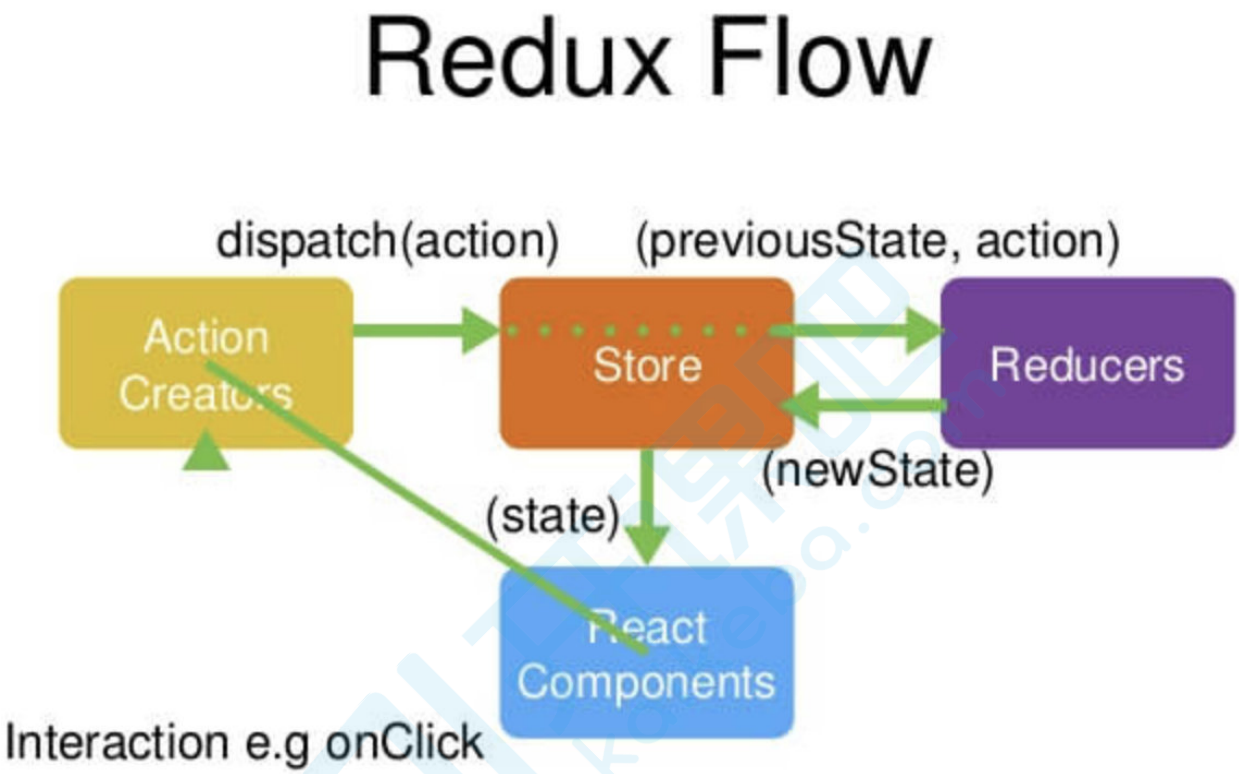


ReactDom.render

```
function renderTreeIntoContainer(vnode, container, callback, parentContext) {

    let id = container[COMPONENT_ID] || (container[COMPONENT_ID] = _.getUid())

    pendingRendering[id] = true
    let oldVnode = null
    let rootNode = null
    if (oldVnode = vnodeStore[id]) {
        rootNode = compareTwoVnodes(oldVnode, vnode, container.firstChild,
parentContext)
    } else {
        rootNode = initVnode(vnode, parentContext, container.namespaceURI)
        var childNode = null
        while (childNode = container.lastChild) {
            container.removeChild(childNode)
        }
        container.appendChild(rootNode)
    }
    vnodeStore[id] = vnode
    let isPending = updateQueue.isPending
    updateQueue.isPending = true
    clearPending()
    return result
}
```

redux

1. 为什么需要redux 他是什么
2. 解决了什么问题
3. 如何使用
4. 单向数据流




```
export function createStore(reducer, enhancer){
    if (enhancer) {
        return enhancer(createStore)(reducer)
    }
    let currentState = {}
    let currentListeners = []

    function getState(){
        return currentState
    }
    function subscribe(listener){
        currentListeners.push(listener)
    }
    function dispatch(action){
        currentState = reducer(currentState, action)
        currentListeners.forEach(v=>v())
        return action
    }
```

```
    dispatch({type:'@kaikeba/sheng'})
    return { getState, subscribe, dispatch}
}

export function applyMiddleware(...middlewares){
    return createStore=>(...args)=>{
        const store = createStore(...args)
        let dispatch = store.dispatch

        const midApi = {
            getState:store.getState,
            dispatch:(...args)=>dispatch(...args)
        }
        const middlewareChain = middlewares.map(middleware=>middleware(midApi))
        dispatch = compose(...middlewareChain)(store.dispatch)
        return {
            ...store,
            dispatch
        }

    }
}
export function compose(...funcs){
    if (funcs.length==0) {
        return arg=>arg
    }
    if (funcs.length==1) {
        return funcs[0]
    }
    return funcs.reduce((ret,item)=> (...args)=>ret(item(...args)))
}
function bindActionCreator(creator, dispatch){
    return (...args) => dispatch(creator(...args))
}
export function bindActionCreators(creators,dispatch){
    return Object.keys(creators).reduce((ret,item)=>{
        ret[item] = bindActionCreator(creators[item],dispatch)
        return ret
    },{})
}
```

react-redux

```
import React from 'react'
import PropTypes from 'prop-types'
import {bindActionCreators} from './woniu-redux'

export const connect = (mapStateToProps=state=>state,mapDispatchToProps={})=>
(WrapComponent)=>{
    return class ConnectComponent extends React.Component{
        static contextTypes = {
```

```
            store:PropTypes.object
        }
        constructor(props, context){
            super(props, context)
            this.state = {
                props:{}
            }
        }
        componentDidMount(){
            const {store} = this.context
            store.subscribe(()=>this.update())
            this.update()
        }
        update(){
            const {store} = this.context
            const stateProps = mapStateToProps(store.getState())
            const dispatchProps = bindActionCreators(mapDispatchToProps,
store.dispatch)
            this.setState({
                props:{
                    ...this.state.props,
                    ...stateProps,
                    ...dispatchProps
                }
            })
        }
        render(){
            return <WrapComponent {...this.state.props}></WrapComponent>
        }
    }
}

export class Provider extends React.Component{
    static childContextTypes = {
        store: PropTypes.object
    }
    getChildContext(){
        return {store:this.store}
    }
    constructor(props, context){
        super(props, context)
        this.store = props.store
    }
    render(){
        return this.props.children
    }
}
```

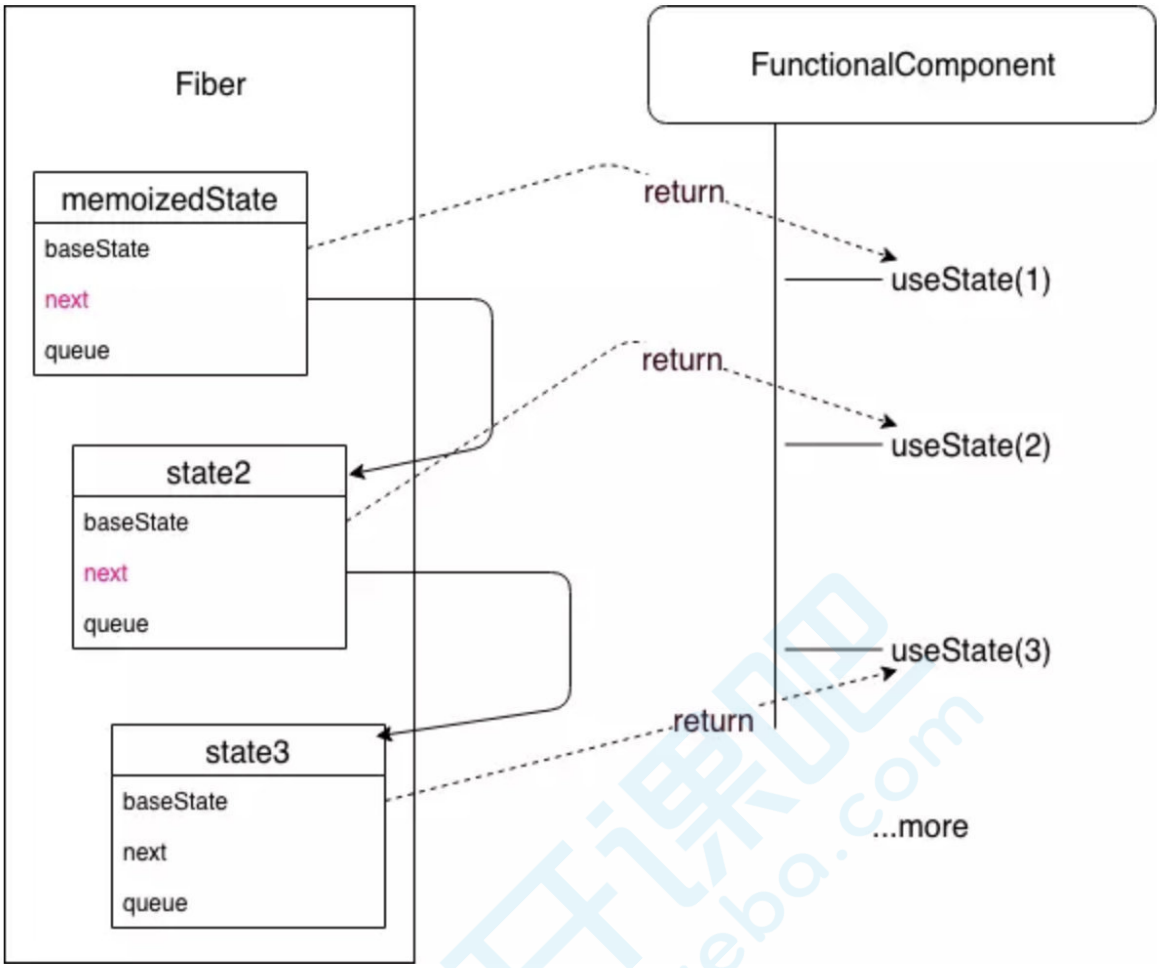
###

Hooks

1. Hooks是啥
    1. 为了拥抱正能量 函数式
2. Hooks带来的变革 让函数组件有了状态 可以完全替代class
3. 类似链表的实现原理

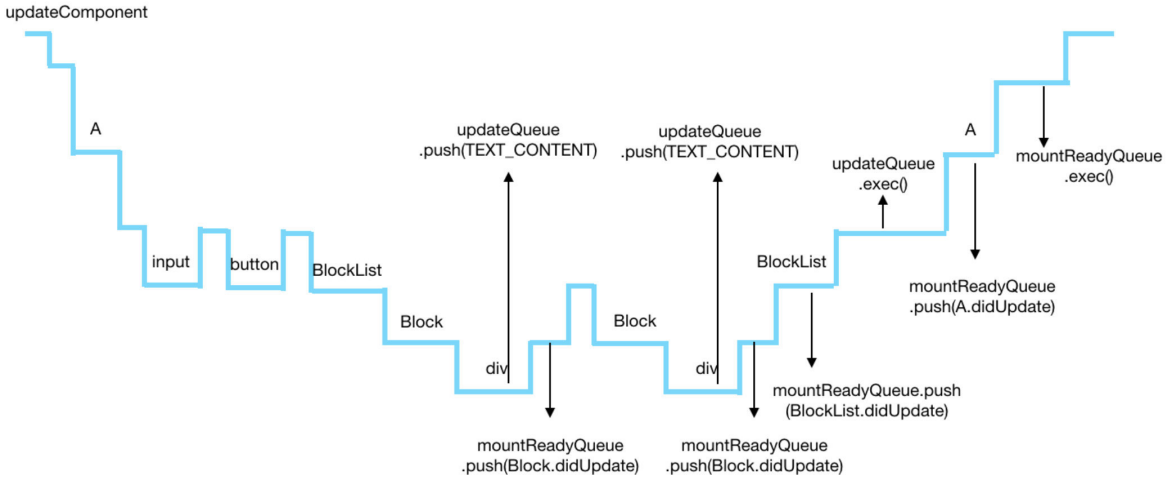
```
import React, { useState, useEffect } from 'react'

function FunComp(props) {
  const [data, setData] = useState('initialState')

  function handleChange(e) {
    setData(e.target.value)
  }

  useEffect(() => {
    subscribeToSomething()

    return () => {
      unSubscribeToSomething()
    }
  })

  return (
    <input value={data} onChange={handleChange} />
  )
}
```

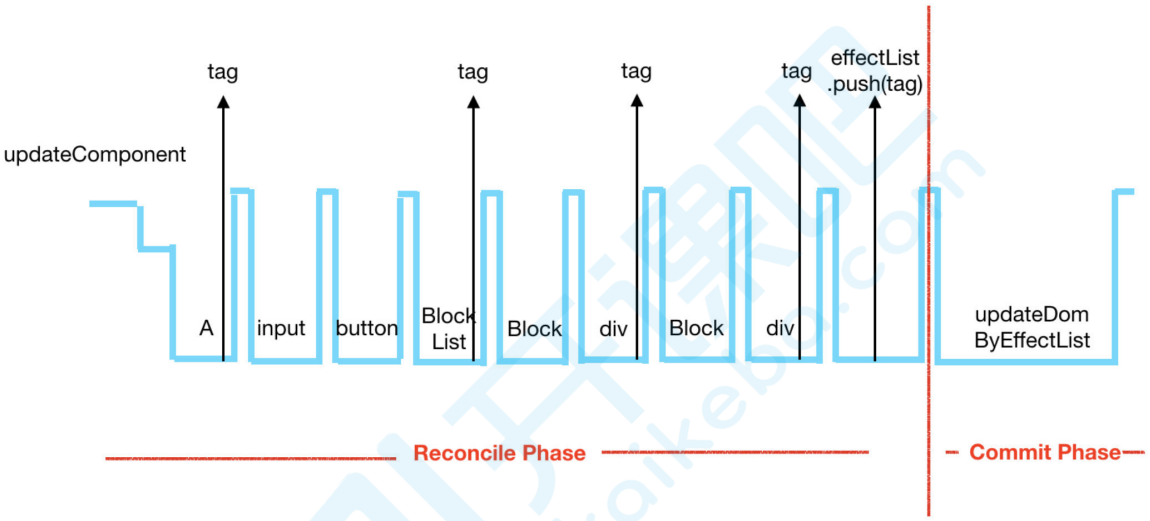
```
function FunctionalComponent () {
  const [state1, setState1] = useState(1)
  const [state2, setState2] = useState(2)
  const [state3, setState3] = useState(3)
}

hook1 => Fiber.memoizedState
state1 === hook1.memoizedState
hook1.next => hook2
state2 === hook2.memoizedState
hook2.next => hook3
state3 === hook2.memoizedState
```



Fibter

1. 为什么需要fiber
2. 任务分解的意义
3. 增量渲染（把渲染任务拆分成块，匀到多帧）
4. 更新时能够暂停，终止，复用渲染任务
5. 给不同类型的更新赋予优先级
6. 并发方面新的基础能力
7. 更流畅





扩展点

1. 扩展
    1. dva
    2. router
2. 和别的章节联系
    1. setState
    2. 虚拟dom
3. 知识体系
    1. React底层

总结

1. 回顾知识点
2. 提示学习方法

3. 提示本次课重点和必会知识

作业 && 答疑

1. setState处理成异步具体怎么操作的

   A setTimeout

   ```
   B 自己定义队列
   ```

   C 链表

2. fiber架构 如何利用浏览器空闲时间进行渲染的

A promise B setTimout C requestidleCallback

课堂目标

1. 讲明白dom diff 最好自己能写