React原理剖析

准备工作

创建一个全新项目：`create-react-app kreact`

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;
```

核心精简后：

```
const React = {
    createElement,
    Component
}
```

react-dom 主要是render逻辑

最核心的api：

React.createElement：创建虚拟DOM

React.Component：实现自定义组件

ReactDOM.render：渲染真实DOM

JSX

在线尝试

JSX预处理前：

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
)
```

JSX预处理后：

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)
```

使用自定义组件的情况：

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

const jsx = (
  <div id="demo">
    <span>hi</span>
    <Comp name="kaikeba" />
  </div>
);
console.log(jsx);

ReactDOM.render(jsx, document.querySelector("#root"));
```

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vdom用JS对象形式来描述dom树结构 一一对应

> 输出vdom观察其结构

实现三大接口：React.createElement, React.Component, ReactDom.render

CreateElement

- 创建./src/kreact.js，它需要包含createElement方法

```
function createElement() {
    console.log(arguments)
}
export default {createElement}
```

- 修改index.js实际引入kreact，测试

```
import React from "./kreact";
```

- 更新kreact.js：为createElement添加参数并返回结果对象

```
function createElement(type, props,...children){
    // 父元素需要子元素返回结果，这里可以通过JSX编译后的代码得出结论
    props.children = children;
    return {type,props}
}

export default {createElement}
```

render

- kreact-dom需要提供一个render函数，能够将vdom渲染出来，这里先打印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"
  ]
}
```

创建kvdom.js：将createElement返回的结果对象转换为vdom

- 传递给createElememnt的组件有三种组件类型，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

```
// kvdom.js
export function createVNode(vtype, type, props) {
    let vnode = {
        vtype: vtype, // 用于区分函数组件、类组件、原生组件
        type: type,
        props: props
    }
    return vnode
}
```

- 添加Component类，kreact.js

```
export 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 = {}
    }
}
```

> 浅层封装，setState现在只是一个占位符

- 添加类组件，index.js

```
import React, {Component} from "./kreact";
class Comp2 extends Component {
  render() {
    return (
      <div>
        <h2>hi {this.props.name}</h2>
      </div>
    )
  }
}
```

- 判断组件类型，kreact.js

```
import { createVNode } from "./kvdom";
function createElement(type, props, ...children) {
  //...

  //判断组件类型
  let vtype;
  if (typeof type === "string") {
    // 原生标签
    vtype = 1;
  }  else if (typeof type === "function") {
    if (type.isReactComponent) {
      // 类组件
      vtype = 2;
    } else {
      // 函数组件
      vtype = 3;
    }
  }

  return createVNode(vtype, type, props);
}
```

- 转换vdom为真实dom

```
export function initVNode(vnode) {
  let { vtype } = vnode;

  if (!vtype) {
    // 没有vtype，是一个文本节点
    return document.createTextNode(vnode);
  }

  if (vtype === 1) {
    // 1是原生元素
    return createElement(vnode);
  } else if (vtype === 2) {
    // 2是类组件
    return createClassComp(vnode);
  } else if (vtype === 3) {
```

```
    // 3是函数组件
    return createFuncComp(vnode);
  }
}

// 创建原生元素
function createElement(vnode) {
  const { type, props } = vnode;
  const node = document.createElement(type);

  // 过滤key，children等特殊props
  const { key, children, ...rest } = props;

  Object.keys(rest).forEach(k => {
    // 需要特殊处理的属性名：class和for
    if (k === "className") {
      node.setAttribute("class", rest[k]);
    } else if (k === "htmlFor") {
      node.setAttribute("for", rest[k]);
    } else {
      node.setAttribute(k, rest[k]);
    }
  });

  // 递归初始化子元素
  children.forEach(c => {
    // 子元素也是一个vnode，所以调用initVNode
    node.appendChild(initVNode(c));
  });
  return node;
}
// 创建函数组件
function createFuncComp(vnode) {
  const { type, props } = vnode;
  // type是函数，它本身即是渲染函数，返回vdom
  const newNode = type(props);
  return initVNode(newNode);
}
// 创建类组件
function createClassComp(vnode) {
  const { type } = vnode;
  // 创建类组件实例
  const component = new type(vnode.props);
  // 调用其render获得vdom
  const newNode = component.render();
  return initVNode(newNode);
}
```

- 执行渲染，kreact-dom.js

```
import { initVNode } from "./kvdom";
function render(vnode, container) {
  const node = initVNode(vnode);
  container.appendChild(node);
  // container.innerHTML = `<pre>${JSON.stringify(vnode,null,2)}</pre>`
}
```

- 渲染vdom数组, index.js

```
class Comp2 extends Component {
  render() {
    const users=[{id:1,name:'tom'},{id:2,name:'jerry'}]
    return (
      <div>
        <h2>hi {this.props.name}</h2>
        <ul>
          {users.map(user=>(<li key={user.id}>{user.name}</li>))}
        </ul>
      </div>
    )
  }
}
// 测试报错，因为kvdom中没有考虑到该情况
```

- 处理vdom数组，kvdom.js

```
children.forEach(c => {
    if (Array.isArray(c)) {// c是vdom数组的情况
      c.forEach(n => {
        node.appendChild(initVNode(n));
      });
    } else {
      node.appendChild(initVNode(c));
    }
  });
```

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组件的特点，就是拥有特殊状态并且可以通过setState更新状态，从而重新渲染视图，是学习React中最重要的api。

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，./kkreact/Component.js

```
export let updateQueue = {
    updaters: [],
    isPending: false,
    add(updater) {
        _.addItem(this.updaters, updater)
    },
    batchUpdate() {
        if (this.isPending) {
            return
        }
        this.isPending = true
        /*
         each updater.update may add new updater to updateQueue
         clear them with a loop
         event bubbles from bottom-level to top-level
         reverse the updater order can merge some props and state and reduce the
refresh times
         see Updater.update method below to know why
        */
        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 {
```

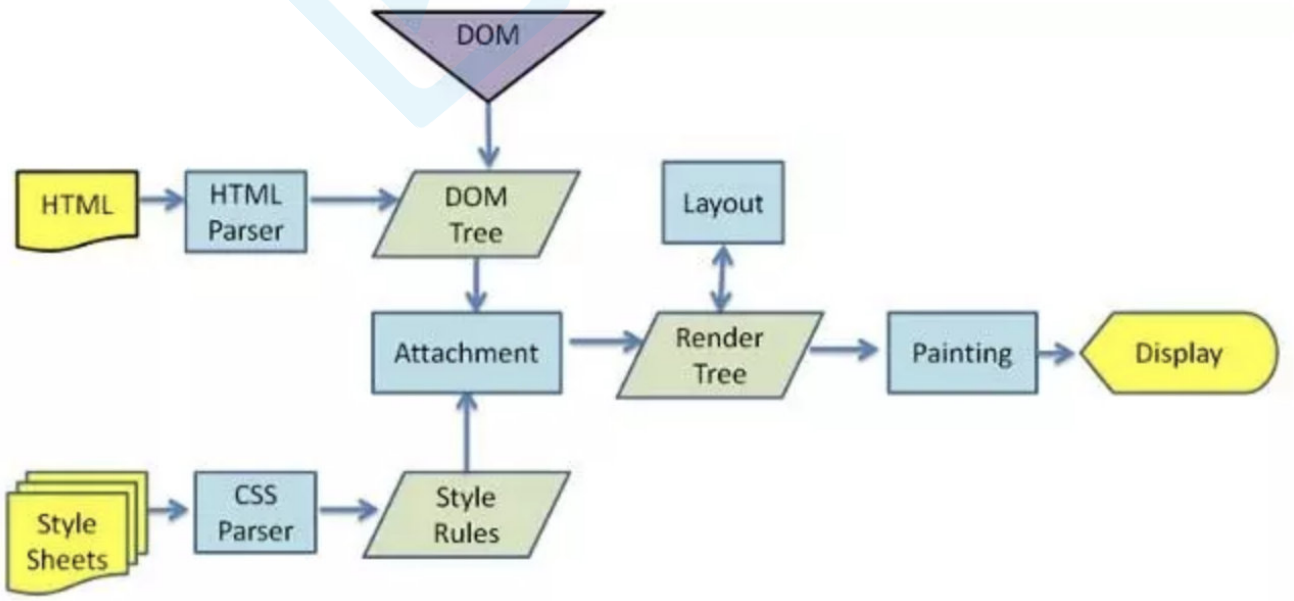
```
                    _.extend(state, nextState)
                }
            })
            pendingStates.length = 0
        }
        return state
    },
    clearCallbacks() {
        let { pendingCallbacks, instance } = this
        if (pendingCallbacks.length > 0) {
            this.pendingCallbacks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

虚拟dom

常见问题：react virtual dom是什么？说一下diff算法?

what? 用 JavaScript 对象表示 DOM 信息和结构，当状态变更的时候，重新渲染这个 JavaScript 的对象结构。这个 JavaScript 对象称为virtual dom；

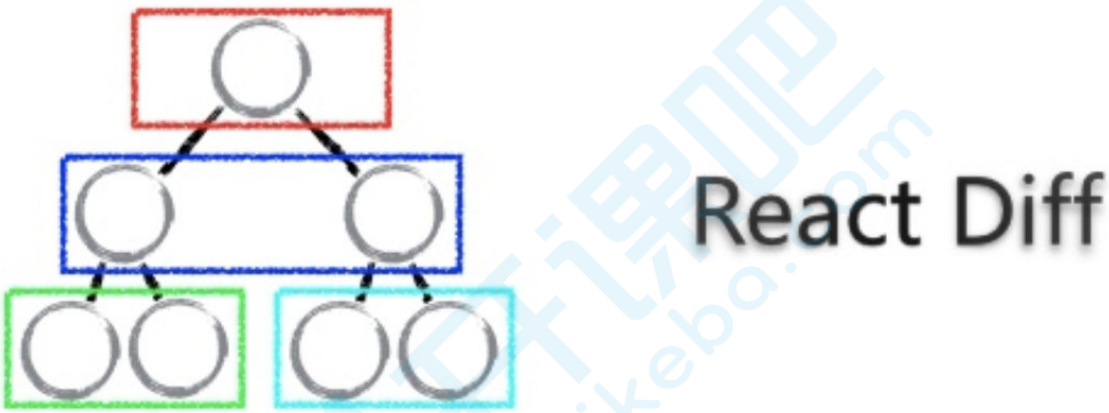
传统dom渲染流程



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

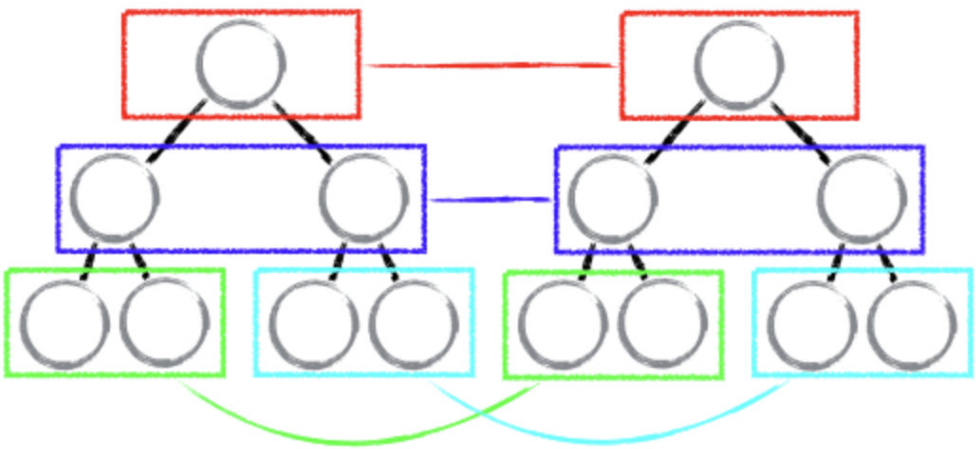
```
align title lang translate dir dataset hidden tabIndex accessKey draggable spellcheck contentEditable isContentEditable offsetParent offsetTop offsetLeft        VM1696:
offsetWidth offsetHeight style innerText outerText webkitdropzone onabort onblur oncancel oncanplay oncanplaythrough onchange onclick 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
```

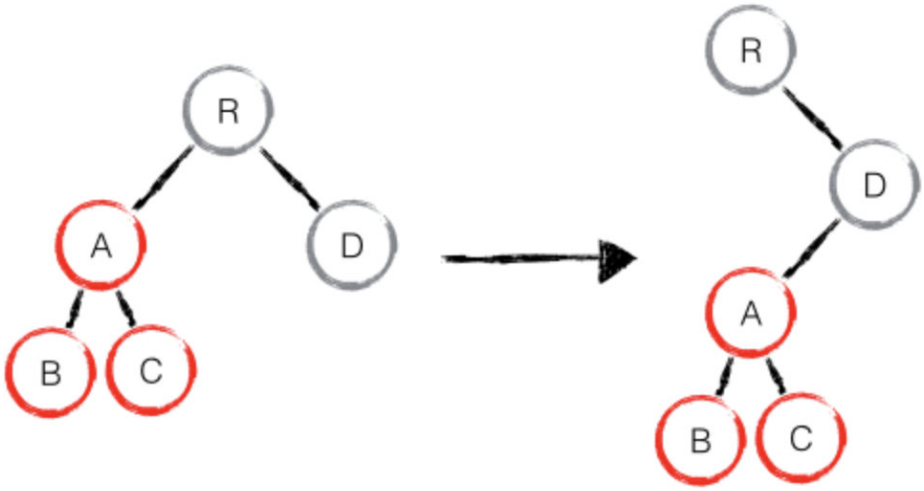
diff算法



diff 策略

1. 同级比较，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。

   例如：div->p, CompA->CompB

3. 对于同一层级的一组子节点，通过唯一的key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**

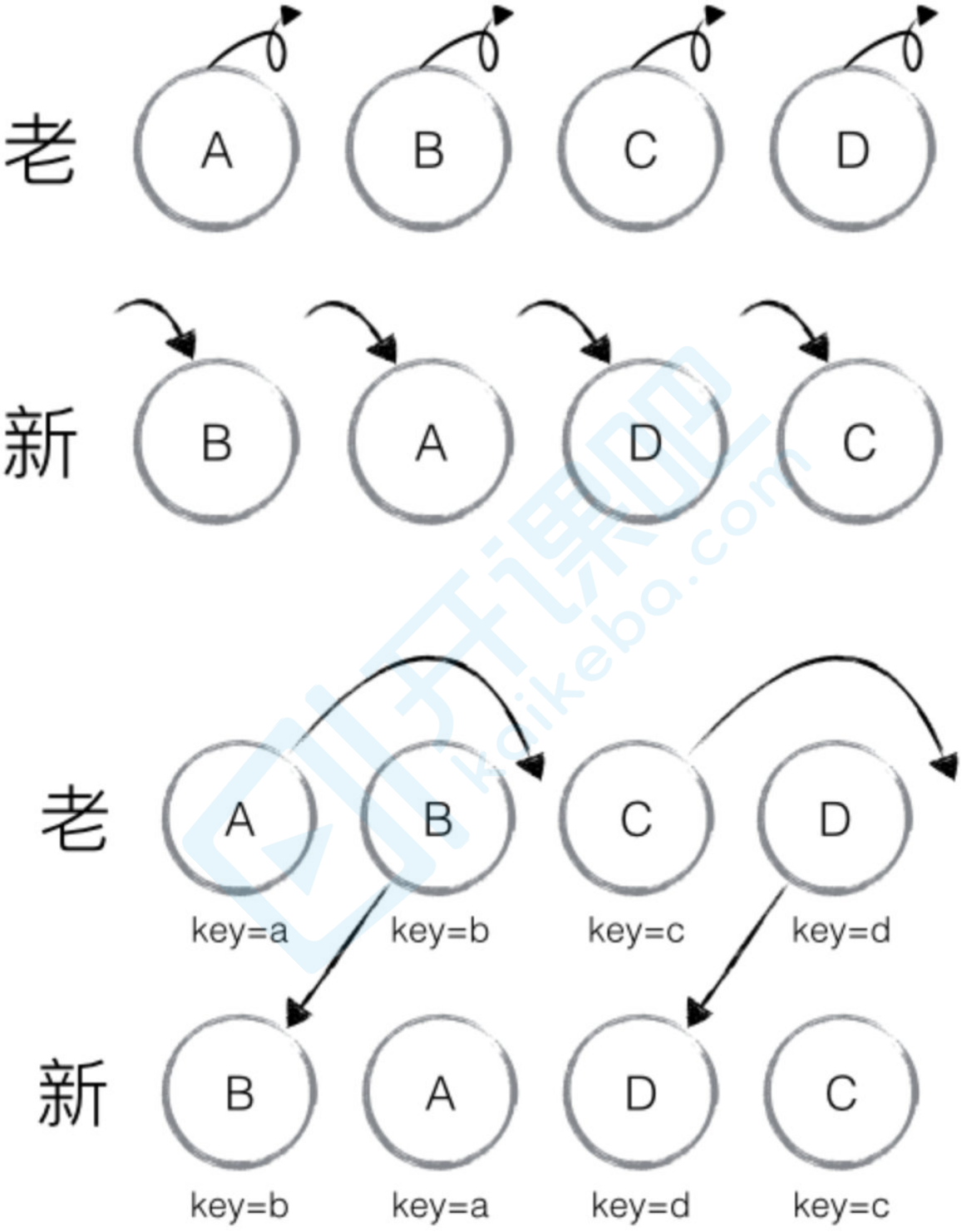
element diff

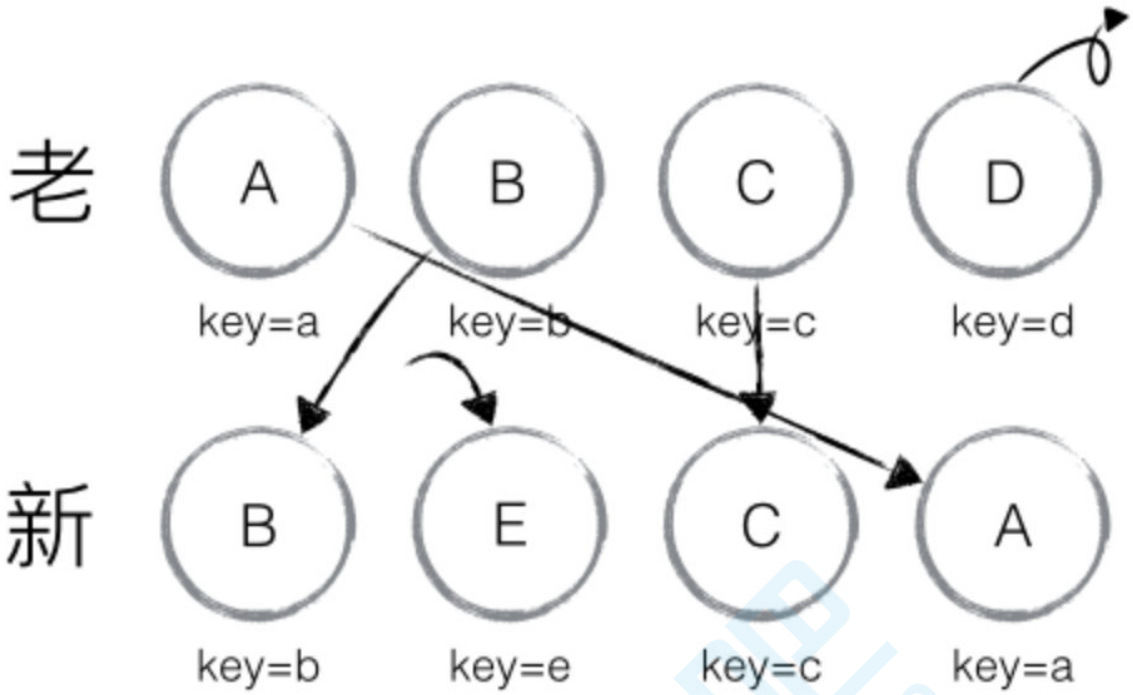
差异类型：

1. **替换原来的节点**，例如把div换成了p，Comp1换成Comp2
2. **移动、删除、新增子节点**， 例如ul中的多个子节点li中出现了顺序互换。
3. 修改了节点的**属性**，例如节点类名发生了变化。
4. 对于**文本节点**，文本内容可能会改变。

重排（reorder）操作：**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



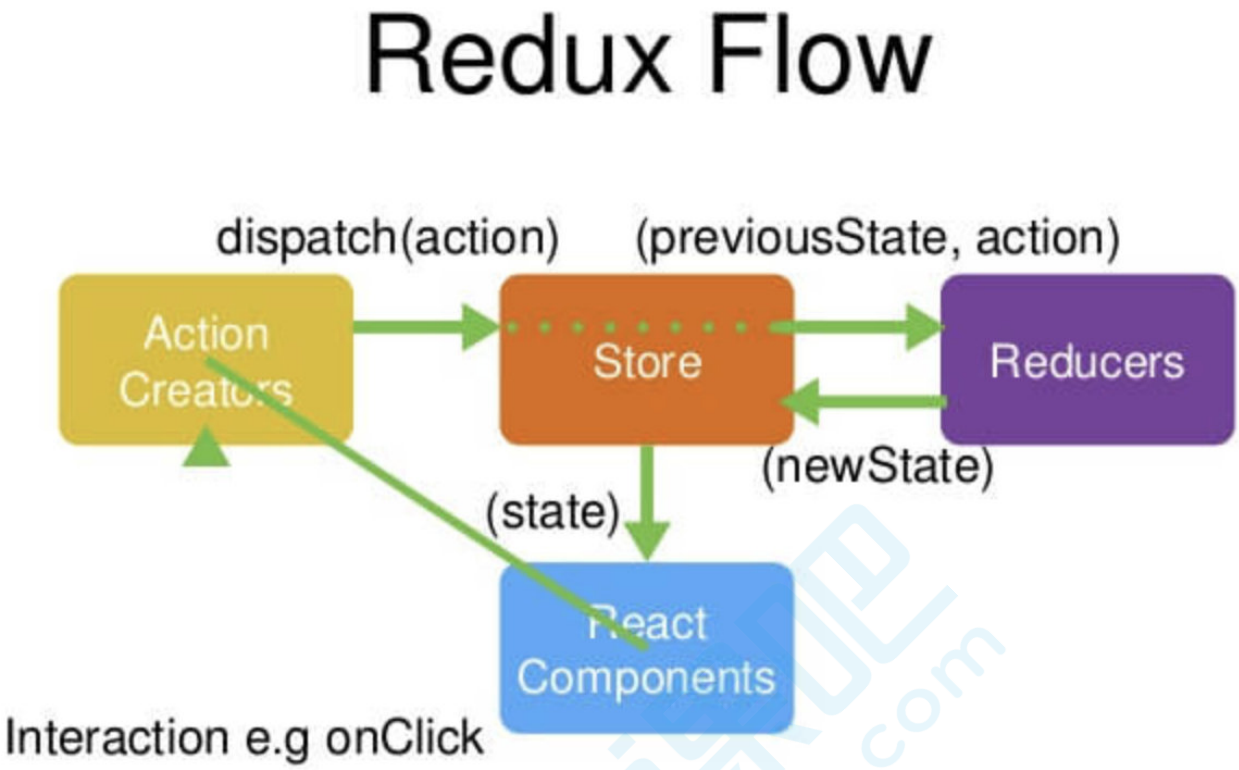
ReactDom.render

```
function renderTreeIntoContainer(vnode, container, callback, parentContext) {
    if (!vnode.vtype) {
        throw new Error(`cannot render ${ vnode } to container`)
    }
    if (!isValidContainer(container)) {
        throw new Error(`container ${container} is not a DOM element`)
    }
    let id = container[COMPONENT_ID] || (container[COMPONENT_ID] = _.getUid())
    let argsCache = pendingRendering[id]

    // component lify cycle method maybe call root rendering
    // should bundle them and render by only one time
    if (argsCache) {
        if (argsCache === true) {
            pendingRendering[id] = argsCache = { vnode, callback, parentContext }
        } else {
            argsCache.vnode = vnode
            argsCache.parentContext = parentContext
            argsCache.callback = argsCache.callback ? _.pipe(argsCache.callback,
callback) : callback
        }
        return
    }

    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()
    argsCache = pendingRendering[id]
    delete pendingRendering[id]

    let result = null
    if (typeof argsCache === 'object') {
        result = renderTreeIntoContainer(argsCache.vnode, container,
argsCache.callback, argsCache.parentContext)
    } else if (vnode.vtype === VELEMENT) {
        result = rootNode
    } else if (vnode.vtype === VCOMPONENT) {
        result = rootNode.cache[vnode.uid]
    }

    if (!isPending) {
        updateQueue.isPending = false
        updateQueue.batchUpdate()
    }

    if (callback) {
        callback.call(result)
    }

    return result
}
```

redux

1. 为什么需要redux 他是什么
2. 结局了什么问题
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
    }
}
```