内容

手写createElement、Component、render三个api

1. 创建kreact：实现createElement并返回vdom

```
function createElement(type, props, ...children) {
    // console.log(arguments);
    // 返回虚拟DOM
    props.children = children;
    delete props.__self;
    delete props.__source;

    // 能够区分组件类型：
    // vtype: 1-原生标签；2-函数组件；3-类组件
    let vtype;
    if (typeof type === 'string') {
        // 原生标签
        vtype = 1;
    } else {
        // console.log(typeof type);
        if (type.isReactComponent) {
            vtype = 3;
        } else {
            vtype = 2;
        }
    }


    return {vtype, type, props}
}


export class Component {
    static isReactComponent = true;
    constructor(props) {
        this.props = props;
        this.state = {};
    }
    setState() {}
    forceUpdate() {}
}

export default {createElement}
```

2. 创建kreact-dom：实现render，能够将kvdom返回的dom追加至container

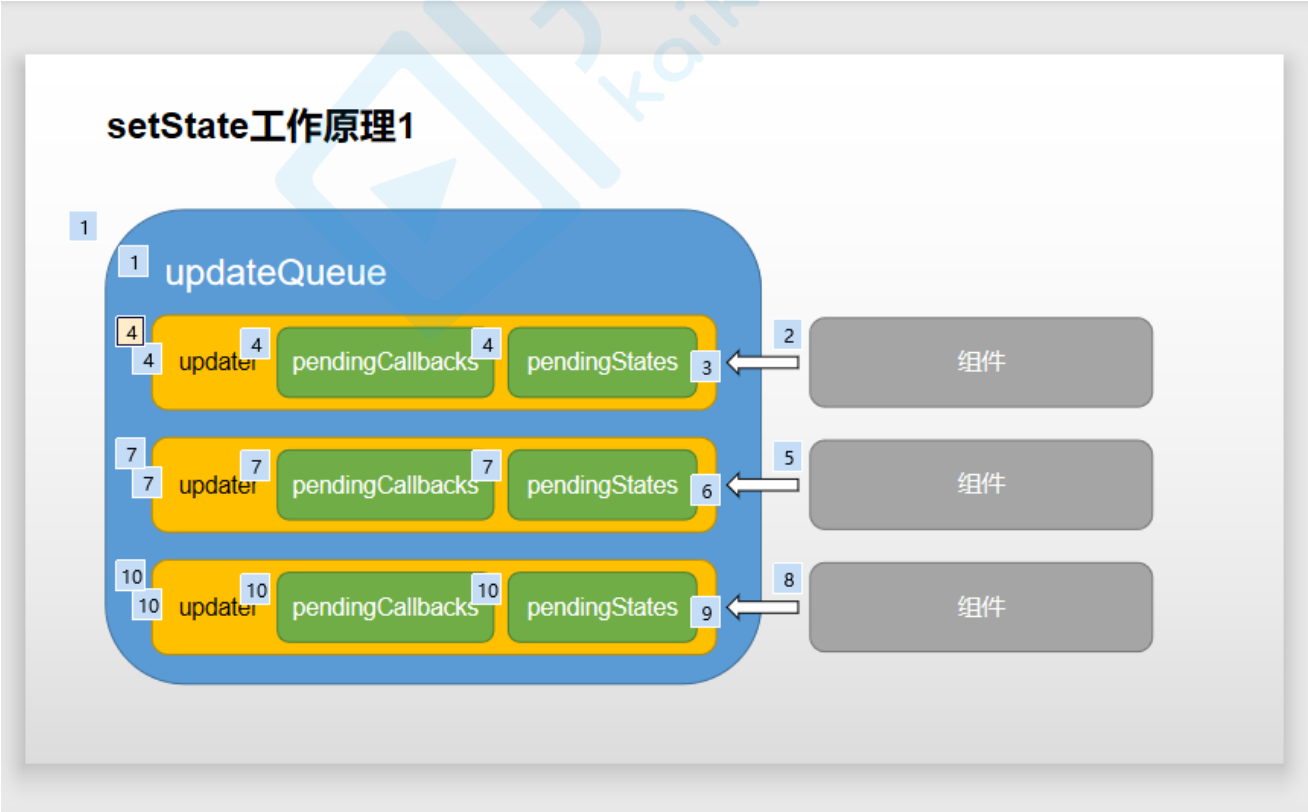
```
import { initVNode } from "./kvdom";

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

export default { render };
```

3. 创建kvdom：实现initVNode，能够将vdom转换为dom

```
// 执行和vdom相关的操作
export function initVNode(vnode) {
  let { vtype } = vnode;

  if (!vtype) {
    // 文本节点
    return document.createTextNode(vnode);
  }

  if (vtype === 1) {
    // 原生标签:div
    return createNativeElement(vnode);
  } else if (vtype === 2) {
    return createFuncComp(vnode);
  } else {
    return createClassComp(vnode);
  }
}

function createNativeElement(vnode) {
  // 1.vnode[type]
  const { type, props } = vnode;
  // 创建DOM
  const node = document.createElement(type);
  // 过滤特殊属性
  const { key, children, ...rest } = props;
  Object.keys(rest).forEach(k => {
    // 需特殊处理的htmlFor，className
    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
  children.forEach(c => {
```

```
    if (Array.isArray(c)) {
      c.forEach(n => node.appendChild(initVNode(n)));
    } else {
      node.appendChild(initVNode(c));
    }
  });

  return node;
}
function createFuncComp(vnode) {
  // 此处type是一个函数
  const { type, props } = vnode;
  const vdom = type(props);
  return initVNode(vdom);
}
function createClassComp(vnode) {
  // 此处type是一个class
  const { type, props } = vnode;
  const component = new type(props);
  const vdom = component.render();
  return initVNode(vdom);
}
```

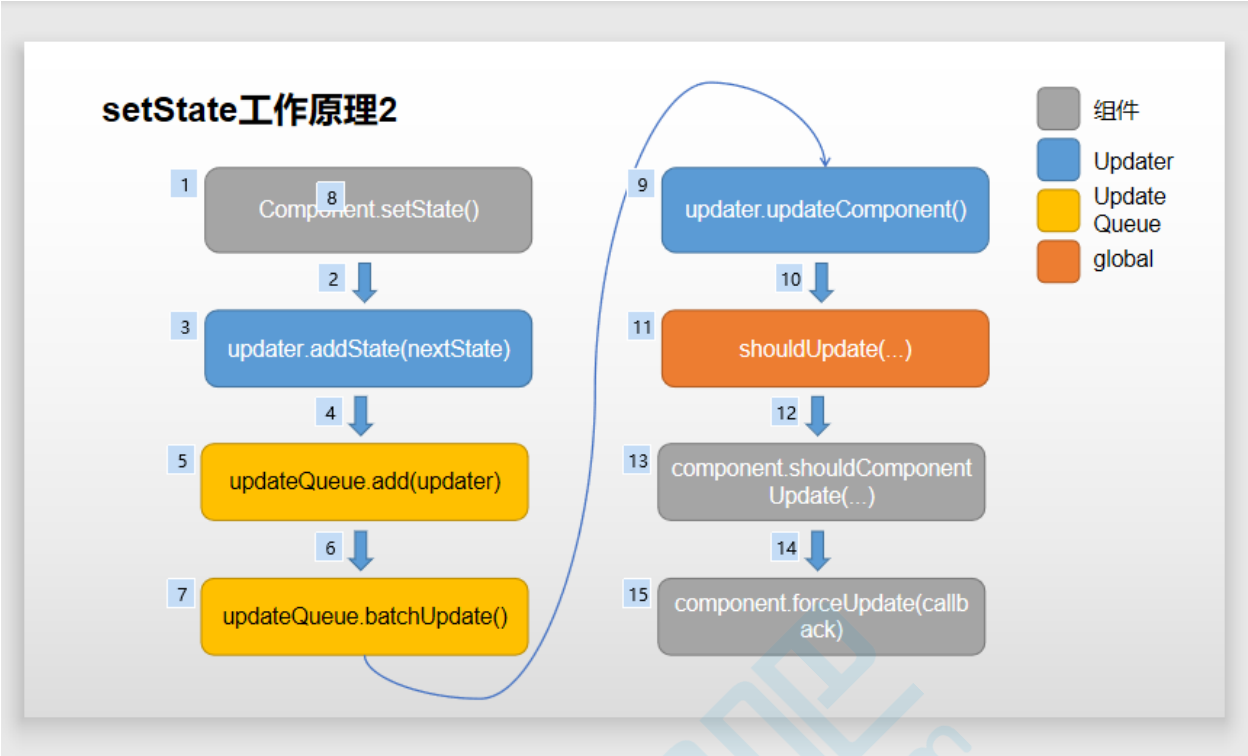
setState原理剖析

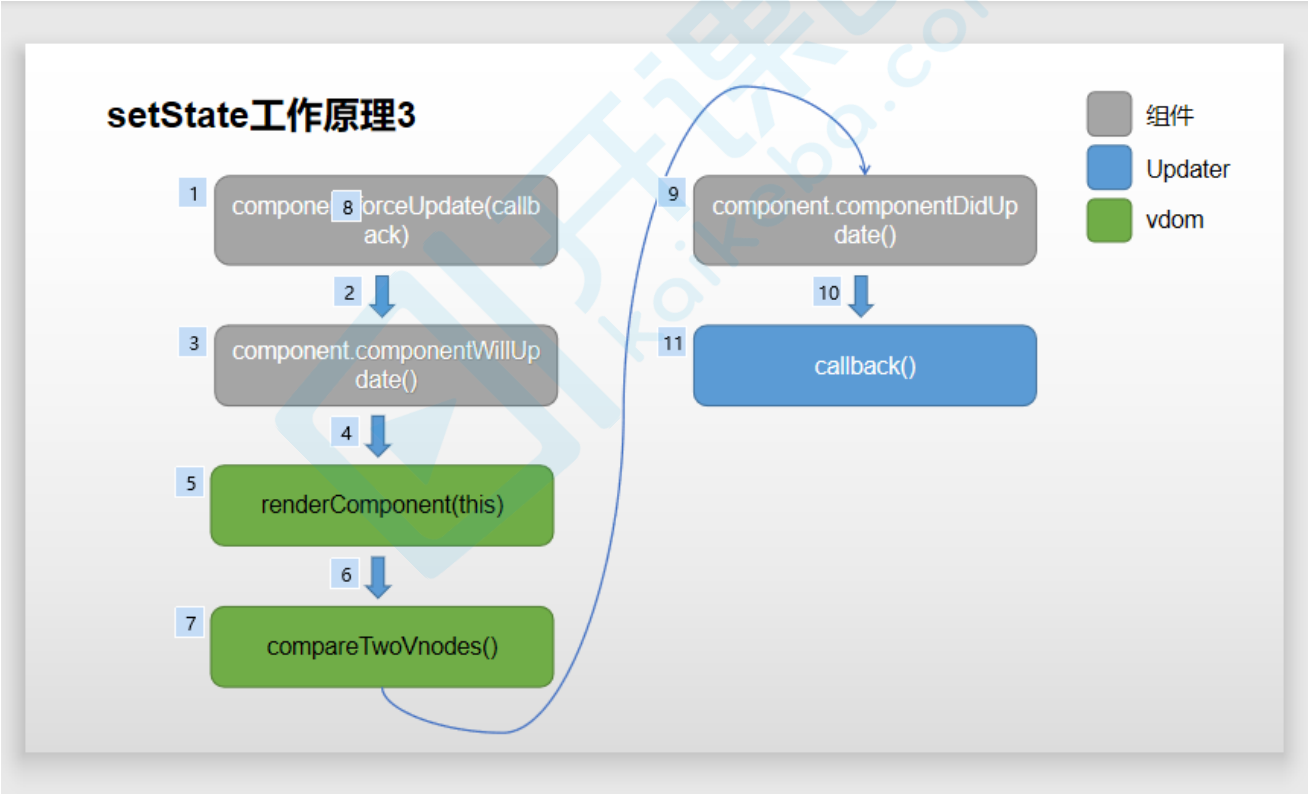


1. setState批量行为：React会合并多次setState操作为一次执行，关键代码如下

```
// Updater
getState() {
        let { instance, pendingStates } = this
        let { state, props } = instance
        // 合并待处理状态数组
        if (pendingStates.length) {
            state = {...state}
            pendingStates.forEach(nextState => {
                let isReplace = _.isArr(nextState)
                if (isReplace) {
                    nextState = nextState[0]
                }
                if (_.isFn(nextState)) {
                    // 函数方式将立即执行，它可以获得之前合并的状态结果
                    nextState = nextState.call(instance, state, props)
                }
                // replace state
                if (isReplace) {
                    state = {...nextState}
                } else {
                    // 合并新旧状态
                    state = {...state, ...nextState}
                }
            })
            pendingStates.length = 0
        }
        return state
    }
```

2. 异步：setState调用后，会调用其updater.addState，最终调用updateQueue.add将任务添加到队列等待系统批量更新batchUpdate。具体执行流程如下：





关键代码：

```
// Component
setState(nextState, callback) {
        // 添加异步队列   不是每次都更新
        this.$updater.addCallback(callback)
        this.$updater.addState(nextState)
    }
```

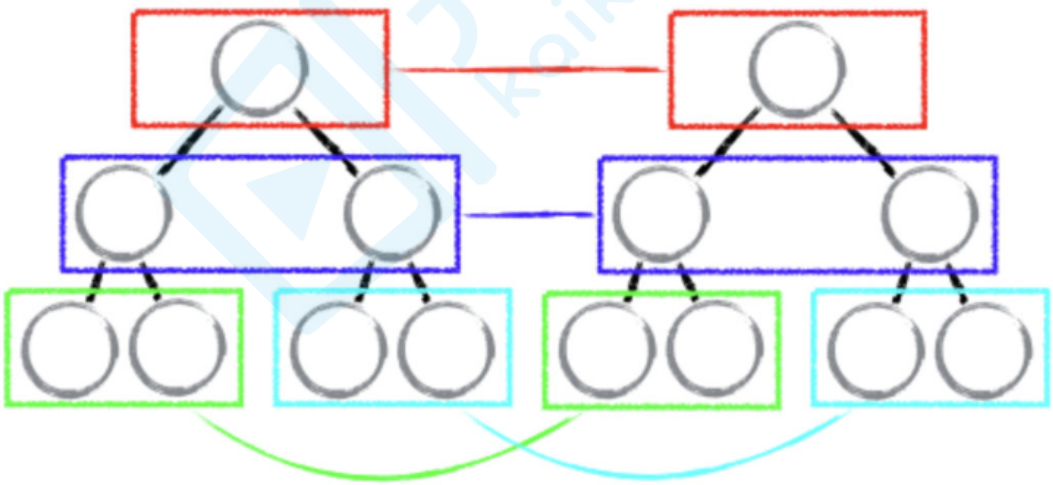
```
// updater
addState(nextState) {
        if (nextState) {
            // 放入更新队列
            this.pendingStates.push(nextState)
            // 如果当前队列没有工作则直接更新
            if (!this.isPending) {
                this.emitUpdate()
            }
        }
    }

    emitUpdate(nextProps, nextContext) {
        this.nextProps = nextProps
        this.nextContext = nextContext
        // receive nextProps!! should update immediately
        nextProps || !updateQueue.isPending
        ? this.updateComponent()
        : updateQueue.add(this)
    }

// updateQueue
add(updater) {
        this.updaters.push(updater)
    }

batchUpdate() {
        if (this.isPending) {
            return
        }
        this.isPending = true
        let { updaters } = this
        let updater
        while (updater = updaters.pop()) {
            updater.updateComponent()
        }
        this.isPending = false
    }

//updater
updateComponent() {
        let { instance, pendingStates, nextProps, nextContext } = this
        if (nextProps || pendingStates.length > 0) {
            // ...
            // getState 合并所有的state的数据，一次更新
            shouldUpdate(instance, nextProps, this.getState(), nextContext,
this.clearCallbacks)
        }
    }

function shouldUpdate(component, nextProps, nextState, nextContext, callback) {
    // 是否应该更新 判断shouldComponentUpdate生命周期
    // ...
```

```
    component.forceUpdate(callback)
}


// Component
// 跳过所有生命周期执行强制更新
    forceUpdate(callback) {
        // 实际更新组件的函数
        let { $updater, $cache, props, state, context } = this
        //...
        // 下面才是重点 diff
        let newVnode = renderComponent(this)
        let newNode = compareTwoVnodes(vnode, newVnode, node, getChildContext(this,
parentContext))

        // ...
    }
```

虚拟DOM原理剖析

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。

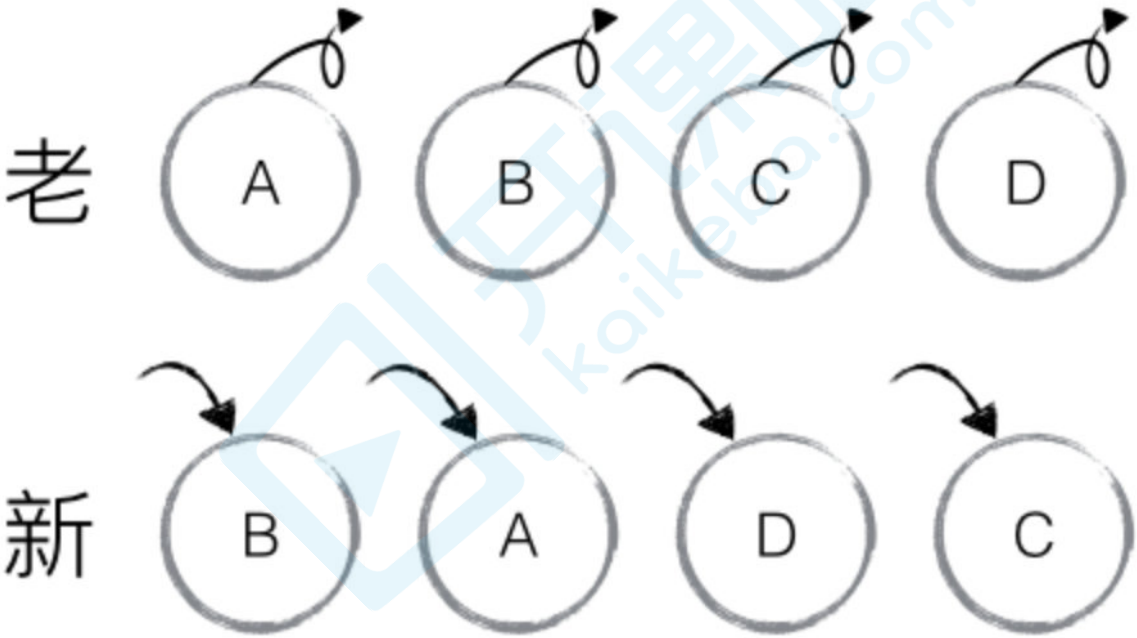
element diff

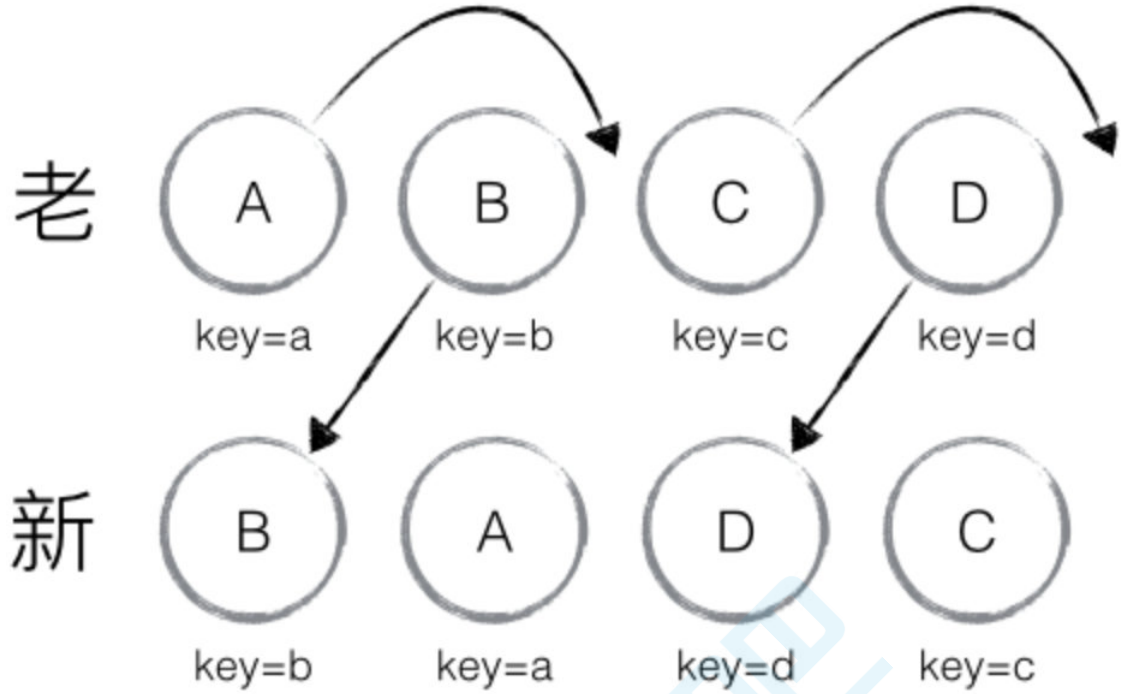
差异类型：

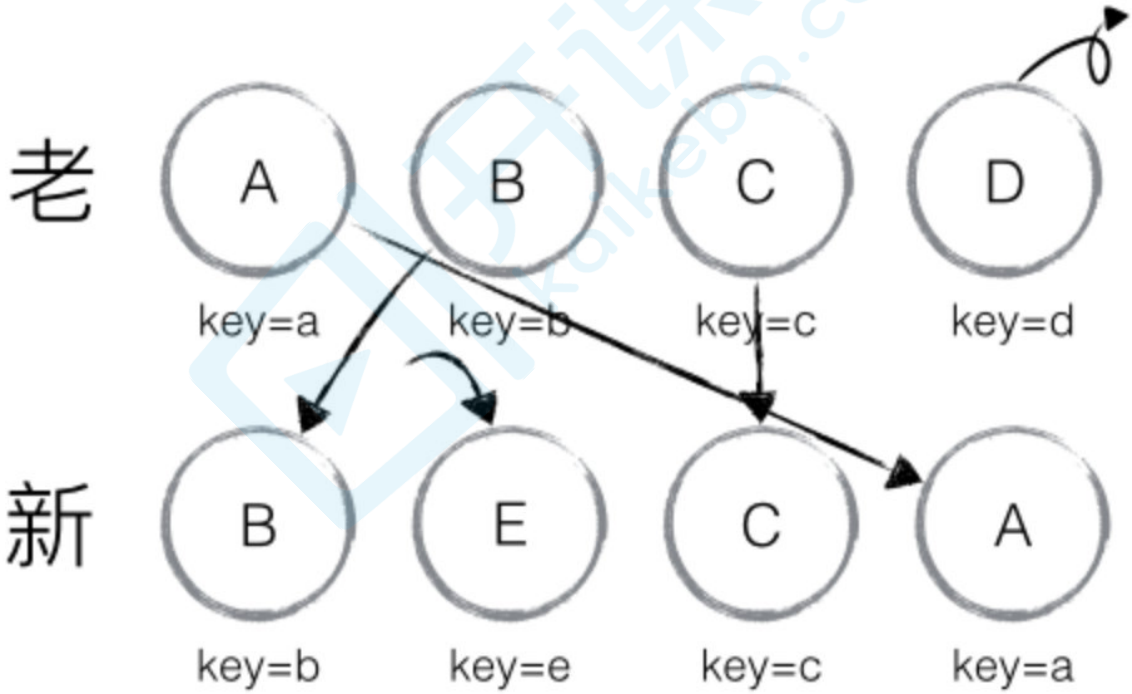
1. **替换原来的节点**，例如把div换成了p，Comp1换成Comp2
2. **移动、删除、新增子节点**， 例如ul中的多个子节点li中出现了顺序互换。
3. 修改了节点的**属性**，例如节点类名发生了变化。
4. 对于**文本节点**，文本内容可能会改变。

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。



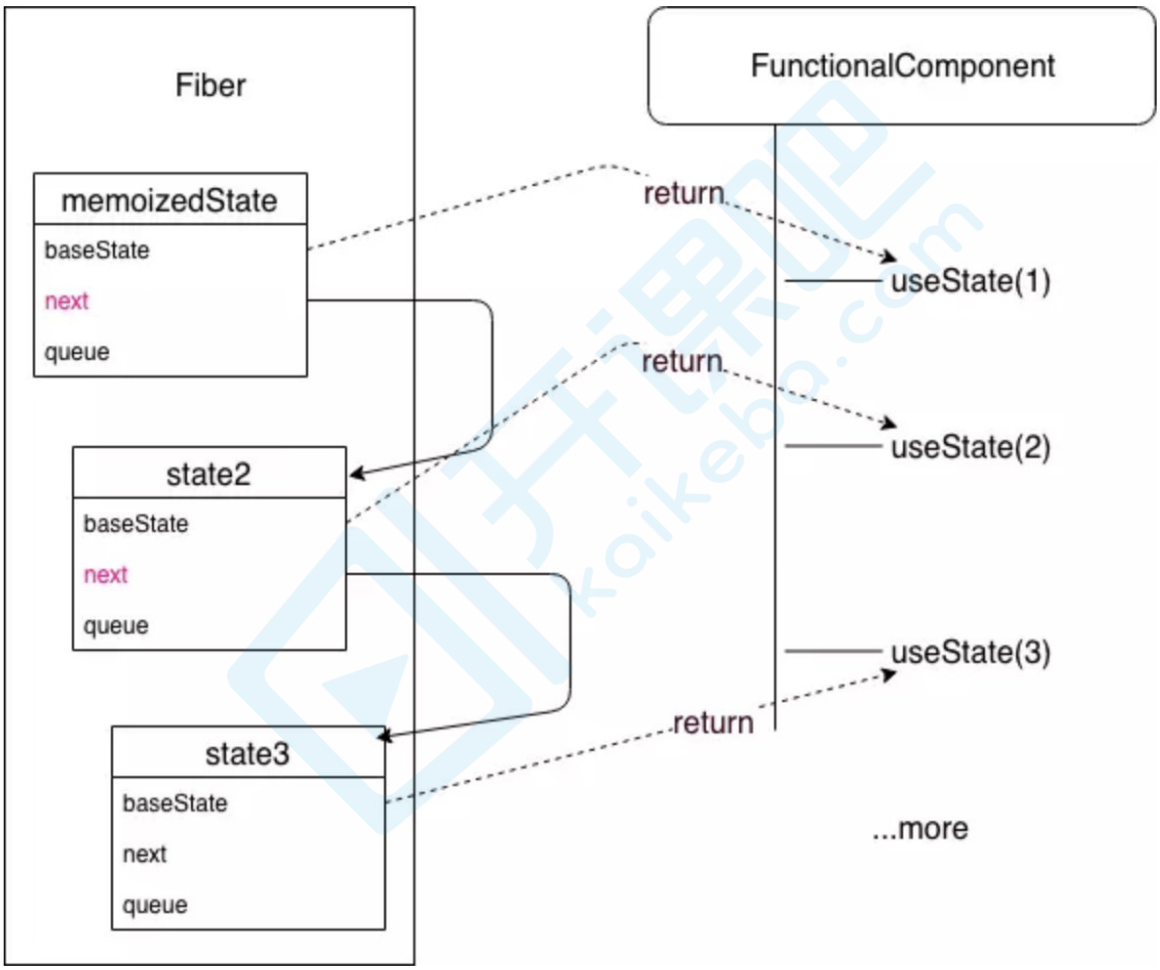




Hooks原理

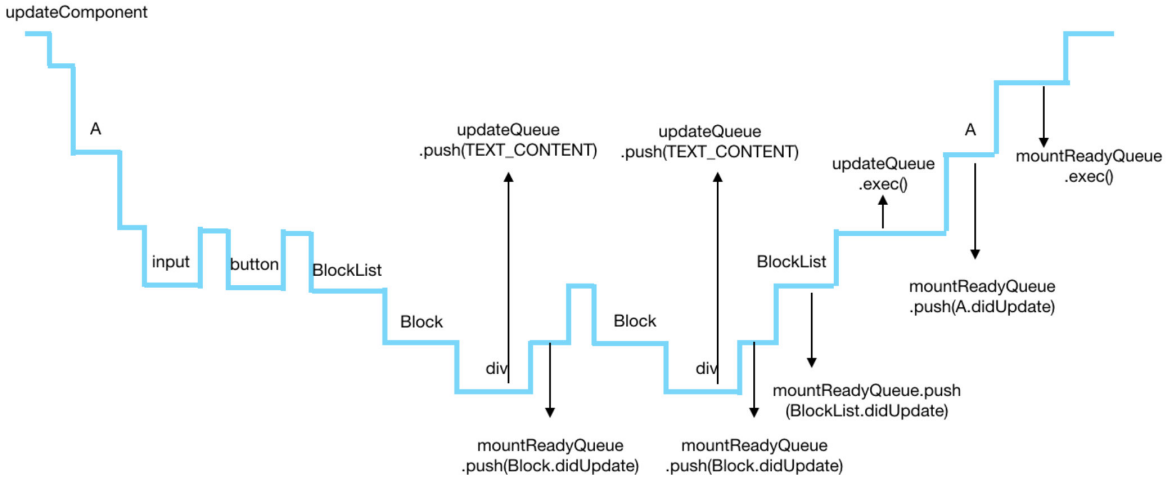
将函数组件状态保存在外部作用域，类似链表的实现原理，由于有严格的顺序关系，所以函数组件中useState这些api不能出现条件、循环语句中

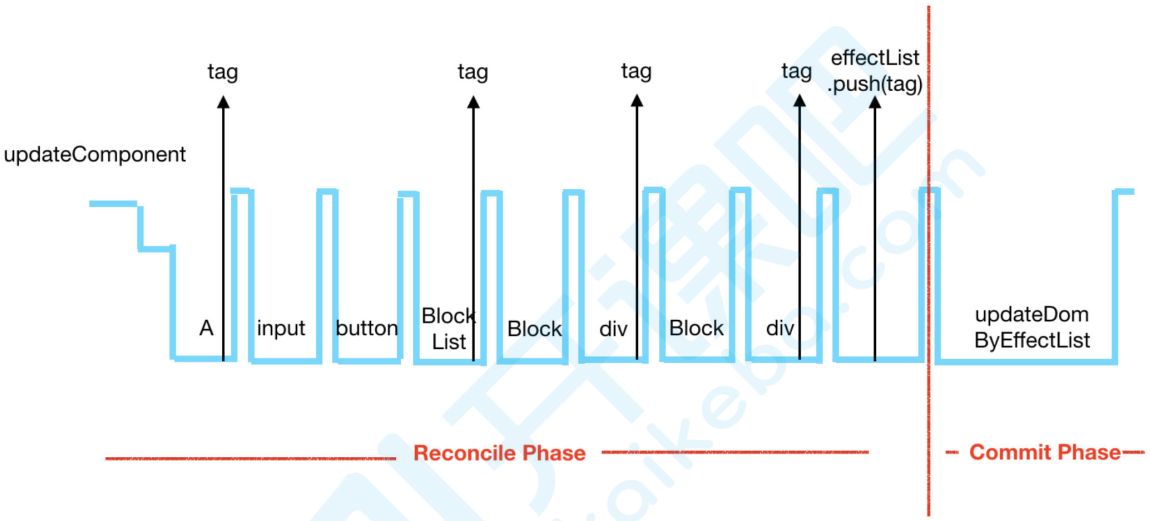
```
function FunctionalComponent () {
  const [state1, setState1] = useState(1)
  const [state2, setState2] = useState(2)
  const [state3, setState3] = useState(3)
}

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


updateComponent
A
input
button
BlockList
Block
div
updateQueue
.push(TEXT_CONTENT)
mountReadyQueue
.push(Block.didUpdate)
Block
div
updateQueue
.push(TEXT_CONTENT)
mountReadyQueue
.push(Block.didUpdate)
BlockList
mountReadyQueue.push
(BlockList.didUpdate)
updateQueue
.exec()
A
mountReadyQueue
.push(A.didUpdate)
mountReadyQueue
.exec()


updateComponent
tag
tag
tag
tag
effectList
.push(tag)
A
input
button
Block
List
Block
div
Block
div
updateDom
ByEffectList
Reconcile Phase
Commit Phase