link: null
title: 珠峰架构师成长计划
description: src\react\index.js
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=250 sentences=1354, words=10675

1. 渲染原生组件

1.1 src\index.js

```
import React from './react';
import ReactDOM from './react-dom';

let onClick = () => { alert('hello'); }
let element = React.createElement('button',
  { id: 'sayHello', onClick },
  "say", React.createElement('span', { style: { color: 'red' } }, 'Hello'));

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

1.2 react\index.js

src\react\index.js

```
import { TEXT, ELEMENT } from './constants';
import { ReactElement } from './vdom';

function createElement(type, config = {}, ...children) {
    delete config.__source;
    delete config.__self;
    let { key, ref, ...props } = config;
    let $typeof = null;
    if (typeof type === 'string') {
        $typeof = ELEMENT;
    }
    props.children = children.map(item => typeof item === 'object' || typeof item === 'function' ? item
        : { $typeof: TEXT, type: TEXT, content: item });
    return ReactElement($typeof, type, key, ref, props);
}

const React = {
    createElement
}
export default React;
```

1.3 react\constants.js

src\react\constants.js

```
export const TEXT = Symbol.for('TEXT');
export const ELEMENT = Symbol.for('ELEMENT');
```

1.4 react\vdom.js

src\react\vdom.js

```
import { TEXT, ELEMENT } from './constants';
import { setProps ,onlyOne,flatten} from './utils';
export function createDOM(element) {
     element = onlyOne(element);
    let { $typeof } = element;
    let dom = null;
    if (!$typeof) {
        dom = document.createTextNode(element);
    }else if ($typeof === TEXT) {
        dom = document.createTextNode(element.content);
    }else if ($typeof === ELEMENT) {
        dom = createNativeDOM(element);
    }
    element.dom = dom;
    return dom;
}

function createNativeDOM(element) {
    let { type, props } = element;
    let dom = document.createElement(type);
    createChildren(element, dom);
    setProps(dom, props);
    if (props.ref)
        props.ref.current = dom;
    return dom;
}

function createChildren(element, parentNode) {
    element.props.children && flatten(element.props.children).forEach((childElement, index) => {
        childElement._mountIndex = index;
        let childDOM = createDOM(childElement);
        parentNode.appendChild(childDOM);
    });
}

export function ReactElement($typeof, type, key, ref, props) {
    let element = {
        $typeof,
        type,
        props,
        key,
        ref
    };
    return element;
}
```

1.5 react-dom\index.js

src\react-dom\index.js

```
import { createDOM } from '../react/vdom';
export function render(element, container) {
    let dom = createDOM(element);
    container.appendChild(dom);
}
export default {render}
```

1.6 react\utils.js

src\react\utils.js

```
import { addEvent } from './event';
export function setProps(elem, props) {
    for (let key in props) {
        if (key !== 'children') {
            let value = props[key];
            setProp(elem, key, value);
        }
    }
}
function setProp(elem, key, value) {
    if (/^on/.test(key)) {
        addEvent(elem, key, value);
    } else if (key === 'style') {
        if (value) {
            for (let styleName in value) {
                if (value.hasOwnProperty(styleName)) {
                    elem.style[styleName] = value[styleName];
                }
            }
        }
    } else {
        elem.setAttribute(key, value);
    }
    return elem;
}
export function onlyOne(obj) {
    return Array.isArray(obj) ? obj[0] : obj;
}
export function isFunction(obj) {
    return typeof obj === 'function';
}
export function flatten(array) {
    var flattend = [];
    (function flat(array) {
        array.forEach(function (el) {
            if (Array.isArray(el)) flat(el);
            else flattend.push(el);
        });
    })(array);
    return flattend;
}
```

1.7 react\event.js

src\react\event.js

```
export function addEvent(dom, eventType, listener) {
    eventType = eventType.toLowerCase();
    let eventStore = dom.eventStore || (dom.eventStore = {});
    eventStore[eventType] = listener;
    document.addEventListener(eventType.slice(2), dispatchEvent, false);
}
let syntheticEvent;

function dispatchEvent(event) {
    let { type, target } = event;
    syntheticEvent = getSyntheticEvent(event);
    while (target) {
        let { eventStore } = target;
        let listener = eventStore && eventStore[eventType];
        if (listener) {
            listener.call(target, syntheticEvent);
        }
        target = target.parentNode;
    }
    for (let key in syntheticEvent) {
        if (syntheticEvent.hasOwnProperty(key))
            delete syntheticEvent[key];
    }
}
function persist() {
    syntheticEvent = {};
    Object.setPrototypeOf(syntheticEvent, {
        persist
    });
}
function getSyntheticEvent(nativeEvent) {
    if (!syntheticEvent) {
        persist();
    }
    syntheticEvent.nativeEvent = nativeEvent;
    syntheticEvent.currentTarget = nativeEvent.target;
    for (let key in nativeEvent) {
        if (typeof nativeEvent[key] == 'function') {
            syntheticEvent[key] = nativeEvent[key].bind(nativeEvent);
        } else {
            syntheticEvent[key] = nativeEvent[key];
        }
    }
    return syntheticEvent;
}
```

2. 渲染类组件和函数组件

2.1 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
+class ClassCounter extends React.Component {
+    constructor(props) {
+        super(props);
+    }
+    render() {
+        return React.createElement('div', { id: 'counter' }, 'hello');
+    }
+}
+function FunctionCounter(props) {
+    return React.createElement('div', { id: 'counter' }, 'hello');
+}

+let element1 = React.createElement('div', { id: 'counter' }, 'hello');
+let element2 = React.createElement(ClassCounter);
+let element3 = React.createElement(FunctionCounter);
ReactDOM.render(
+    element1,
    document.getElementById('root')
);
```

2.2 react\constants.js

src\react\constants.js

```
export const TEXT = Symbol.for('TEXT');
+export const ELEMENT = Symbol.for('ELEMENT');
+export const FUNCTION_COMPONENT = Symbol.for('FUNCTION_COMPONENT');
+export const CLASS_COMPONENT = Symbol.for('CLASS_COMPONENT');
```

2.3 react\index.js

src\react\index.js

```
+import { TEXT, ELEMENT, FUNCTION_COMPONENT, CLASS_COMPONENT } from './constants';
import { ReactElement } from './vdom';
+import { Component } from './component';
function createElement(type, props = {}, ...children) {
    let $typeof = null;
    if (typeof type
        $typeof = ELEMENT;
+    } else if (typeof type === 'function' && type.prototype.isReactComponent) {
+        $typeof = CLASS_COMPONENT;
+    } else if (typeof type === 'function') {
+        $typeof = FUNCTION_COMPONENT;
+    }
    props.children = children.map(item => typeof item
        : { $typeof: TEXT, type: TEXT, content: item });
    return ReactElement($typeof, type, props);
}
+export {
+    Component
+}
+const React = {
    createElement,
+    Component
+}
export default React;
```

2.4 react\vdom.js

src\react\vdom.js

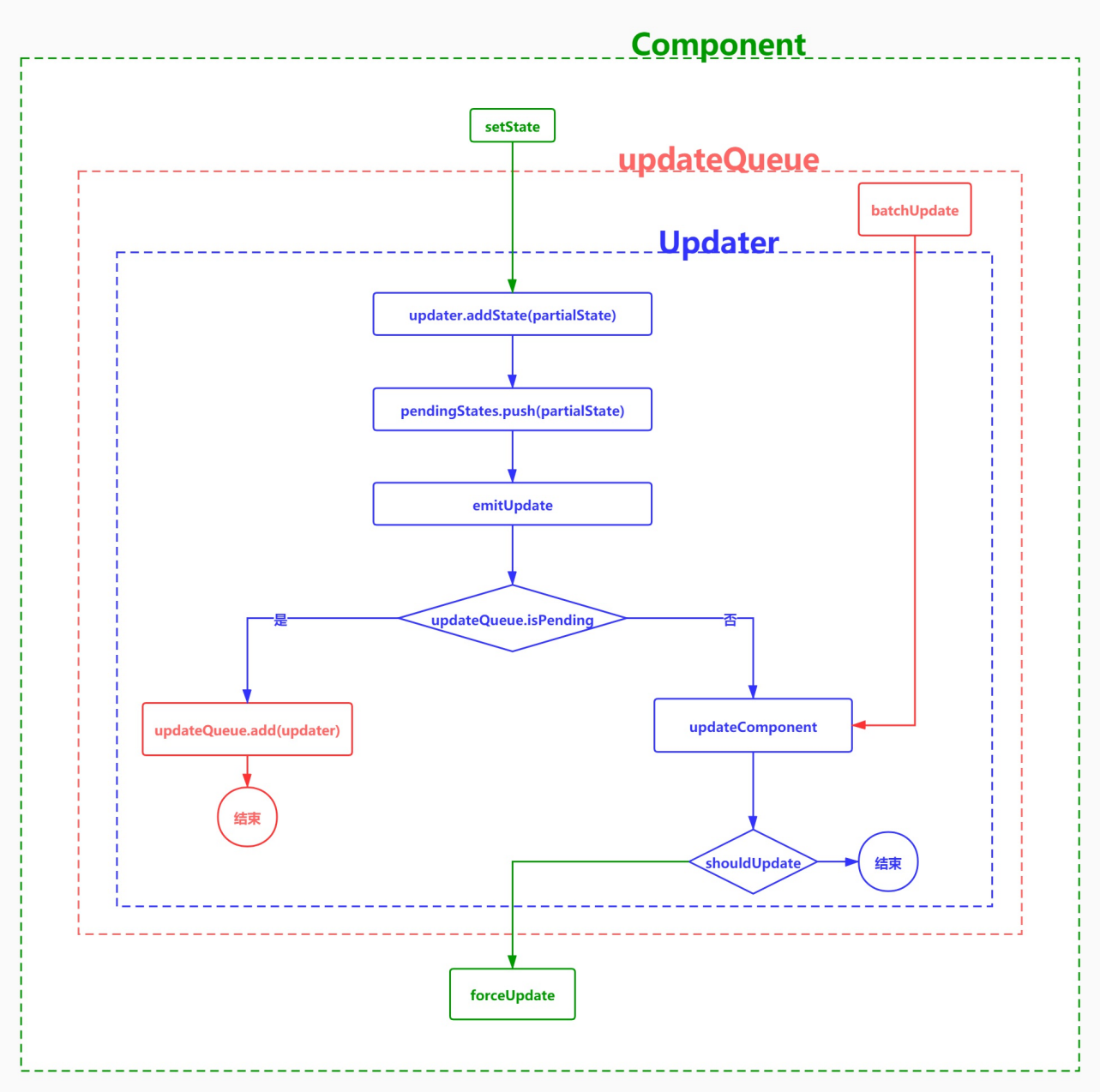
```
+import { TEXT, ELEMENT, FUNCTION_COMPONENT, CLASS_COMPONENT } from './constants';
import { setProps,onlyOne,flatten } from './utils';
export function createDOM(element) {
    element = onlyOne(element);
    let { $typeof } = element;
    let dom = null;
    if (!$typeof) {
        dom = document.createTextNode(element);
    } else if ($typeof
        dom = document.createTextNode(element.content);
    } else if ($typeof
        dom = createNativeDOM(element);
+    } else if ($typeof === FUNCTION_COMPONENT) {
+        dom = createFunctionComponentDOM(element);
+    } else if ($typeof === CLASS_COMPONENT) {
+        dom = createClassComponentDOM(element);
    }
    element.dom = dom;
    return dom;
}

function createNativeDOM(element) {
    let { type, props } = element;
    let dom = document.createElement(type);
    createChildren(element, dom);
    setProps(dom, props);
    return dom;
}

function createChildren(element, parentNode) {
    element.props.children && flatten(element.props.children).forEach((childElement, index) => {
        childElement._mountIndex = index;
        let childDOM = createDOM(childElement);
        childElement.dom = childDOM;
        parentNode.appendChild(childDOM);
    });
}
+function createFunctionComponentDOM(element) {
+    let { type, props } = element;
+    let renderElement = type(props);
+    element.renderElement = renderElement;
+    let newDOM = createDOM(renderElement);
+    renderElement.dom = newDOM;
+    return newDOM;
+}
+function createClassComponentDOM(element) {
+    let { type, props } = element;
+    let componentInstance = new type(props);
+    element.componentInstance = componentInstance;
+    let renderElement = componentInstance.render();
+    componentInstance.renderElement = renderElement;
+    let newDOM = createDOM(renderElement);
+    return newDOM;
+}
export function ReactElement($typeof, type, key, ref, props) {
    let element = {
        $typeof,
        type,
        props,
        key,
        ref
    };
    return element;
}
```

2.5 react\component.js

src\react\component.js

```
class Component {
    constructor(props) {
        this.props = props;
    }
}
Component.prototype.isReactComponent = {};
export {
    Component
}
```

3. 组件更新



3.1 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
class Counter extends React.Component {
    constructor(props) {
        super(props);
        this.state = { number: 0 };
+        setInterval(() => {
+            this.setState({ number: this.state.number + 1 });
+        }, 2000);
    }
    render() {
+        //return
+        //return
+        return
    }
}
+class ClassCounter extends React.Component {
+    render() {
+        return (
+
+        )
+    }
+}
+function FunctionCounter(props) {
+    return (
+
+    )
+}
+let element = React.createElement(Counter, {});
ReactDOM.render(
    element,
    document.getElementById('root')
);
```

3.2 react\utils.js

src\react\utils.js

```
import { addEvent } from './event';
export function setProps(elem, props) {
    for (let key in props) {
        if (key !== 'children') {
            let value = props[key];
            setProp(elem, key, value);
        }
    }
}
function setProp(elem, key, value) {
    if (/^on/.test(key)) {
        addEvent(elem, key, value);
    } else if (key
        if (value) {
            for (let styleName in value) {
                if (value.hasOwnProperty(styleName)) {
                    elem.style[styleName] = value[styleName];
                }
            }
        }
    } else {
        elem.setAttribute(key, value);
    }
    return elem;
}

export function patchProps(elem, oldProps, newProps) {
    for (let key in oldProps) {
        if (key !== 'children') {
            if (newProps.hasOwnProperty(key)) {
                setProp(elem, key, newProps[key]);
            } else {
                elem.removeAttribute(key);
            }
        }
    }
    for (let key in newProps) {
        if (key !== 'children' && !newProps.hasOwnProperty(key)) {
            setProp(elem, key, newProps[key])
        }
    }
}
export function onlyOne(obj) {
    return Array.isArray(obj) ? obj[0] : obj;
}
+export function isFunction(obj) {
+    return typeof obj === 'function';
+}
```

3.3 react\vdom.js

src\react\vdom.js

```
import { TEXT, ELEMENT, FUNCTION_COMPONENT, CLASS_COMPONENT } from './constants';
+import { setProps, onlyOne, flatten,patchProps } from './utils';
export function createDOM(element) {
+    element = onlyOne(element);
    let { $typeof } = element;
    let dom = null;
    if (!$typeof) {
        dom = document.createTextNode(element);
    } else if ($typeof
        dom = document.createTextNode(element.content);
    } else if ($typeof
        dom = createNativeDOM(element);
    } else if ($typeof
```

```
        dom = createFunctionComponentDOM(element);
    } else if ($typeof
        dom = createClassComponentDOM(element);
    }
    element.dom = dom;
    return dom;
}

function createNativeDOM(element) {
    let { type, props } = element;
    let dom = document.createElement(type);
    createChildren(element, dom);
    setProps(dom, props);
    return dom;
}

function createChildren(element, parentNode) {
    element.props.children && flatten(element.props.children).forEach((childElement, index) => {
        childElement._mountIndex = index;
        let childDOM = createDOM(childElement);
        childElement.dom = childDOM;
        parentNode.appendChild(childDOM);
    });
}
function createFunctionComponentDOM(element) {
    let { type, props } = element;
    let renderElement = type(props);
    element.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    renderElement.dom = newDOM;
    return newDOM;
}
function createClassComponentDOM(element) {
    let { type, props } = element;
    let componentInstance = new type(props);
    element.componentInstance = componentInstance;
    let renderElement = componentInstance.render();
    componentInstance.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    return newDOM;
}
+export function compareTwoElements(oldElement, newElement) {
+    oldElement = onlyOne(oldElement);
+    newElement = onlyOne(newElement);
+    let currentDOM = oldElement.dom;
+    let currentElement = oldElement;
+    if (newElement == null) {//如果新节点没有了，直接删除拉倒
+        currentDOM.parentNode.removeChild(currentDOM);
+        currentElement = null;
+    } else if (oldElement.type !== newElement.type) {//如果类型不同
+        let newDOM = createDOM(newElement);
+        currentDOM.parentNode.replaceChild(newDOM, currentDOM);
+        currentElement = newElement;
+    } else {
+        updateElement(oldElement, newElement);
+    }
+    return currentElement;
+}

+function updateElement(oldElement, newElement) {
+    let currentDOM = newElement.dom = oldElement.dom;
+    if (oldElement.$typeof === TEXT && newElement.$typeof === TEXT) {//如果都是文本类型，则直接改文本
+        if (oldElement.content !== newElement.content) {
+            currentDOM.textContent = newElement.content;
+        }
+    } else if (oldElement.$typeof === ELEMENT) {
+        updateDOMProps(currentDOM, oldElement.props, newElement.props);
+        oldElement.props = newElement.props;
+    } else if (oldElement.$typeof === CLASS_COMPONENT) {
+        updateClassComponent(oldElement, newElement);
+        newElement.componentInstance = oldElement.componentInstance;
+    } else if (oldElement.$typeof === FUNCTION_COMPONENT) {
+        updateFunctionComponent(oldElement, newElement);
+    }
+}
+function updateDOMProps(dom, oldProps, newProps) {
+    return patchProps(dom, oldProps, newProps);
+}

+function updateClassComponent(oldElement, newElement) {
+    let componentInstance = oldElement.componentInstance;
+    let updater = componentInstance.$updater;
+    let nextProps = newElement.props;
+    updater.emitUpdate(nextProps);
+}

+function updateFunctionComponent(oldElement, newElement) {
+    let newRenderElement = newElement.type(newElement.props);
+    var oldRenderElement = oldElement.renderElement;
+    var currentElement = compareTwoElements(oldRenderElement, newRenderElement);
+    newElement.renderElement = currentElement;
+}

export function ReactElement($typeof, type, key, ref, props) {
    let element = {
        $typeof,
        type,
        props,
        key,
        ref
    };
    return element;
}
```

3.4 src\component.js

src\component.js

```
import { isFunction } from './utils';
import { compareTwoElements } from './vdom';
+export let updateQueue = {
+    updaters: [],
+    isPending: false,
+    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;
+    },
+};
+class Updater {
+    constructor(instance) {
+        this.instance = instance;
+        this.pendingStates = [];
+        this.nextProps = null;
+    }
+    addState(partialState) {
+        this.pendingStates.push(partialState);
+        this.emitUpdate();
+    }
+    emitUpdate(nextProps) {
+        this.nextProps = nextProps;
+        nextProps || !updateQueue.isPending
+            ? this.updateComponent()
+            : updateQueue.add(this);
+    }
+    updateComponent() {
+        let { instance, pendingStates, nextProps } = this;
+        if (nextProps || pendingStates.length > 0) {
+            shouldUpdate(
+                instance,
+                nextProps,
+                this.getState()
+            );
+        }
+    }

+    getState() {
+        let { instance, pendingStates } = this;
+        let { state } = instance;
+        if (pendingStates.length) {
+            pendingStates.forEach(nextState => {
+                if (isFunction(nextState)) {
+                    nextState = nextState.call(instance, state);
+                }
+                state = { ...state, ...nextState };
+            });
+            pendingStates.length = 0;
+        }
+        return state;
+    }
+}
+function shouldUpdate(component, nextProps, nextState) {
+    component.props = nextProps;
+    component.state = nextState;
+    if (component.shouldComponentUpdate && !component.shouldComponentUpdate(nextProps, nextState)) {
+        return;
+    }
+    component.forceUpdate();
+}

class Component {
    constructor(props) {
        this.props = props;
+        this.$updater = new Updater(this);
+        this.state = {};
+        this.nextProps = null;
    }
+    setState(partialState) {
+        this.$updater.addState(partialState);
+    }
+    forceUpdate() {
+        let { props, state, renderElement: oldRenderElement } = this;
+        if (this.componentWillUpdate) {
+            this.componentWillUpdate(props, state);
+        }
+        let newRenderElement = this.render();
+        let currentElement = compareTwoElements(oldRenderElement, newRenderElement);
+        this.renderElement = currentElement;
+        if (this.componentDidUpdate) {
+            this.componentDidUpdate(props, state);
+        }
+    }
+}
Component.prototype.isReactComponent = {};
export {
    Component
}
```

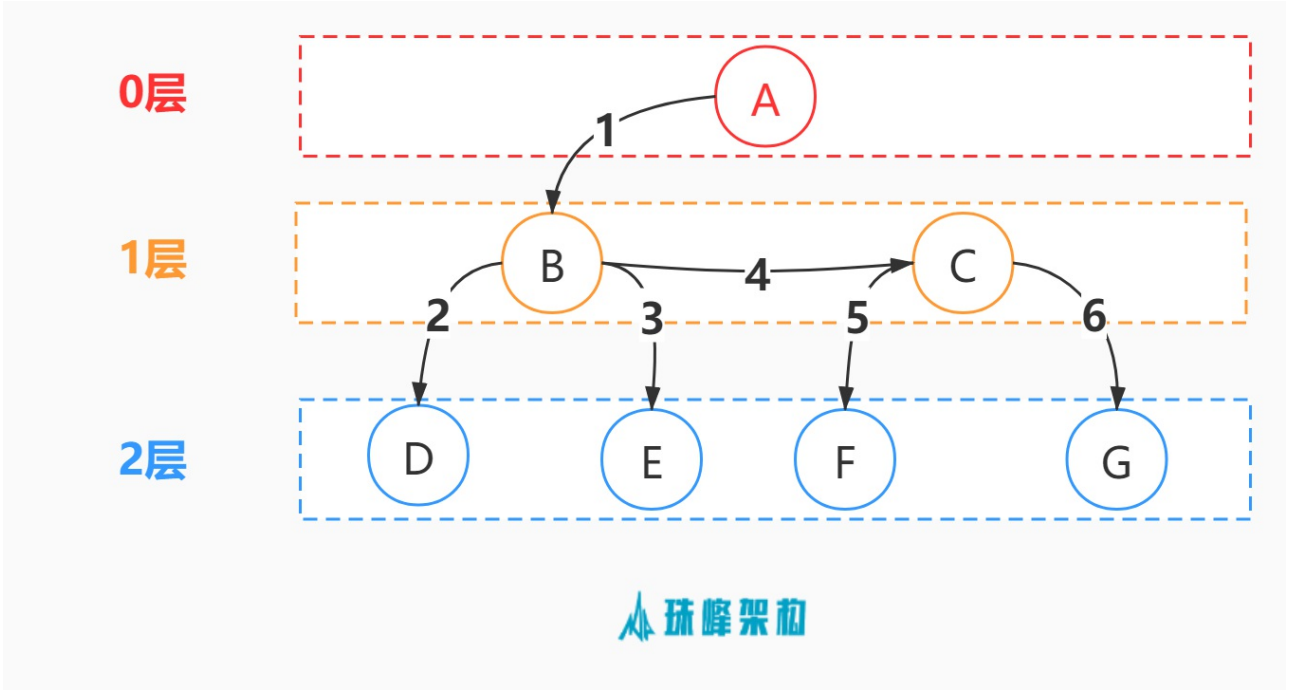
3.5 react\event.js

src\react\event.js

```
+import { updateQueue } from './component';
export function addEvent(elem, eventType, listener) {
    eventType = eventType.toLowerCase();
    let eventStore = elem.eventStore || (elem.eventStore = {});
    eventStore[eventType] = listener;
    document.addEventListener(eventType.substr(2), dispatchEvent, false)
}
function dispatchEvent(event) {
    let { target, type } = event
    let eventType = 'on' + type;
    let syntheticEvent;
+   updateQueue.isPending = true;
    while (target) {
        let { eventStore } = target
        let listener = eventStore && eventStore[eventType];
        if (listener) {
            if (!syntheticEvent) {
                syntheticEvent = createSyntheticEvent(event);
            }
            syntheticEvent.currentTarget = target;
            listener.call(target, syntheticEvent);
        }
        target = target.parentNode;
    }
+    updateQueue.isPending = false;
+    updateQueue.batchUpdate();
}
function createSyntheticEvent(nativeEvent) {
    let syntheticEvent = {}
    syntheticEvent.nativeEvent = nativeEvent
    for (let key in nativeEvent) {
        if (typeof nativeEvent[key] == 'function') {
            syntheticEvent[key] = nativeEvent[key].bind(nativeEvent)
        } else {
            syntheticEvent[key] = nativeEvent[key]
        }
    }
    return syntheticEvent
}
```

4. 比较更新子节点

4.1 深度优先遍历



```
let tree = {
    value: 'A',
    left: {
        value: 'B',
        left: {
            value: 'D',

        },
        right: {
            value: 'E'
        }
    },
    right: {
        value: 'C',
        left: {
            value: 'F',

        },
        right: {
            value: 'G'
        }
    }
}
let depth = 0;
function visit(tree) {
    depth++;
    if (tree) {
        console.log(depth, tree.value);
        visit(tree.left);
        visit(tree.right);
    }
    depth--;
}
visit(tree);
console.log(depth);
```

4.2 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';

class Counter extends React.Component {
    constructor(props) {
        super(props);
        this.state = { number: 0 };
    }
    handleClick = () => {
        this.setState({ number: this.state.number + 1 });
        console.log(this.state.number);
        this.setState({ number: this.state.number + 1 });
        console.log(this.state.number);
        setTimeout(() => {
            this.setState({ number: this.state.number + 1 });
            console.log(this.state.number);
        }, 0);
    }
    render() {
        let p = React.createElement('p', { style: { color: 'red' } }, this.state.number);
        let button = React.createElement('button', { onClick: this.handleClick }, '+');
        return React.createElement('div', { id: 'counter' }, p, button);
    }
}
let element = React.createElement(Counter, {});
ReactDOM.render(
    element,
    document.getElementById('root')
);
```

4.3 react\utils.js

src\react\utils.js

```
import { addEvent } from './event';
export function setProps(elem, props) {
    for (let key in props) {
        if (key !== 'children') {
            let value = props[key];
            setProp(elem, key, value);
        }
    }
}
function setProp(elem, key, value) {
    if (/^on/.test(key)) {
        addEvent(elem, key, value);
    } else if (key
        if (value) {
            for (let styleName in value) {
                if (value.hasOwnProperty(styleName)) {
                    elem.style[styleName] = value[styleName];
                }
            }
        }
    } else {
        elem.setAttribute(key, value);
    }
    return elem;
}

export function patchProps(elem, oldProps, newProps) {
    for (let key in oldProps) {
        if (key !== 'children') {
            if (newProps.hasOwnProperty(key)) {
                setProp(elem, key, newProps[key]);
            } else {
                elem.removeAttribute(key);
            }
        }
    }
    for (let key in newProps) {
        if (key !== 'children' && !newProps.hasOwnProperty(key)) {
            setProp(elem, key, newProps[key])
        }
    }
}
export function onlyOne(obj) {
    return Array.isArray(obj) ? obj[0] : obj;
}

export function isFunction(obj) {
    return typeof obj
}
```

4.4 react\vdom.js

src\react\vdom.js

```
import { TEXT, ELEMENT, FUNCTION_COMPONENT, CLASS_COMPONENT } from './constants';
+import { setProps, onlyOne, patchProps, deepEqual } from './utils';
export function createDOM(element) {
    element = onlyOne(element);
    let { $typeof } = element;
    let dom = null;
    if (!$typeof) {
        dom = document.createTextNode(element);
    } else if ($typeof
        dom = document.createTextNode(element.content);
    } else if ($typeof
        dom = createNativeDOM(element);
    } else if ($typeof
        dom = createFunctionComponentDOM(element);
    } else if ($typeof
        dom = createClassComponentDOM(element);
    }
    element.dom = dom;
    return dom;
}

function createNativeDOM(element) {
    let { type, props } = element;
    let dom = document.createElement(type);
    createChildren(element, dom);
    setProps(dom, props);
    return dom;
}

function createChildren(element, parentNode) {
    element.props.children && flatten(element.props.children).forEach((childElement, index) => {
        childElement._mountIndex = index;
        let childDOM = createDOM(childElement);
        childElement.dom = childDOM;
        parentNode.appendChild(childDOM);
    });
}
function createFunctionComponentDOM(element) {
    let { type, props } = element;
    let renderElement = type(props);
    element.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    renderElement.dom = newDOM;
    return newDOM;
}
function createClassComponentDOM(element) {
    let { type, props } = element;
    let componentInstance = new type(props);
    element.componentInstance = componentInstance;
    let renderElement = componentInstance.render();
```

```
    componentInstance.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    return newDOM;
}
export function compareTwoElements(oldElement, newElement) {
    oldElement = onlyOne(oldElement);
    newElement = onlyOne(newElement);
    let currentDOM = oldElement.dom;
    let currentElement = oldElement;
    if (newElement == null) {//如果新节点没有了，直接删除拉倒
        currentDOM.parentNode.removeChild(currentDOM);
        currentElement = null;
    } else if (oldElement.type !== newElement.type) {//如果类型不同
        let newDOM = createDOM(newElement);
        currentDOM.parentNode.replaceChild(newDOM, currentDOM);
        currentElement = newElement;
    } else {
        updateElement(oldElement, newElement);
    }
    return currentElement;
}

function updateElement(oldElement, newElement) {
    let currentDOM = newElement.dom = oldElement.dom;
    if (oldElement.$typeof
        if (oldElement.content !== newElement.content) {
            currentDOM.textContent = newElement.content;
        }
    } else if (oldElement.$typeof
+        updateChildrenElements(currentDOM, oldElement.props.children, newElement.props.children);
        updateDOMProps(currentDOM, oldElement.props, newElement.props);
        oldElement.props = newElement.props;
    } else if (oldElement.$typeof
        updateClassComponent(oldElement, newElement);
        newElement.componentInstance = oldElement.componentInstance;
    } else if (oldElement.$typeof
        updateFunctionComponent(oldElement, newElement);
    }
}

+function updateChildrenElements(dom, oldChildrenElements, newChildrenElement) {
+    diff(dom, oldChildrenElements, newChildrenElement);
+}

+function diff(parentNode, oldChildrenElements, newChildrenElements) {
+    let oldChildrenElementMap = getChildrenElementMap(oldChildrenElements);
+    let newChildrenElementMap = getNewChildren(oldChildrenElementMap, newChildrenElements);
+}
+function getChildrenElementMap(childrenElements) {
+    let childrenElementMap = {};
+    for (let i = 0; i < childrenElements.length; i++) {
+        let key = childrenElements[i].key || i.toString();
+        childrenElementMap[key] = childrenElements[i];
+    }
+    return childrenElementMap;
+}

+function getNewChildren(oldChildrenElementMap, newChildrenElements) {
+    let newChildrenElementMap = {};
+    newChildrenElements.forEach((newChildElement, index) => {
+        if (newChildElement) {
+            let newKey = newChildElement.key || index.toString();
+            let oldChildElement = oldChildrenElementMap[newKey];
+            if (canDeepCompare(oldChildElement, newChildElement)) {
+                updateElement(oldChildElement, newChildElement);
+                newChildrenElements[index] = oldChildElement;
+            }
+            newChildrenElementMap[newKey] = newChildrenElements[index];
+        }
+    });
+    return newChildrenElementMap;
+}

+function canDeepCompare(oldChildElement, newChildElement) {
+    if (!!oldChildElement && !!newChildElement) {
+        return oldChildElement.type === newChildElement.type;
+    }
+    return false;
+}

+function updateDOMProps(dom, oldProps, newProps) {
+    patchProps(dom, oldProps, newProps);
+}

function updateClassComponent(oldElement, newElement) {
    let componentInstance = oldElement.componentInstance;
    let updater = componentInstance.$updater;
    let nextProps = newElement.props;
    updater.emitUpdate(nextProps);
}
function updateFunctionComponent(oldElement, newElement) {
    let newRenderElement = newElement.type(newElement.props);
    var oldRenderElement = oldElement.renderElement;
    var currentElement = compareTwoElements(oldRenderElement, newRenderElement);
    newElement.renderElement = currentElement;
}

export function ReactElement($typeof, type, key, ref, props) {
    let element = {
        $typeof,
        type,
        props,
        key,
        ref
```

```
    };
    return element;
}
```

5. key的处理

5.1 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';

+class App extends React.Component {
+    constructor(props) {
+        super(props);
+        this.state = { odd: true };
+        setTimeout(() => {
+            this.setState({ odd: !this.state.odd });
+        }, 1000);
+    }
+    handleClick = () => {
+        this.setState({ number: this.state.number + 1 });
+    }
+    render() {
+        if (this.state.odd) {
+            return React.createElement('ul', { key: 'wrapper' },
+                React.createElement('li', { key: 'A' }, 'A'),
+                React.createElement('li', { key: 'B' }, 'B'),
+                React.createElement('li', { key: 'C' }, 'C'),
+                React.createElement('li', { key: 'D' }, 'D'),
+            );
+        } else {
+            return React.createElement('ul', { key: 'wrapper' },
+                React.createElement('li', { key: 'A' }, 'A'),
+                React.createElement('li', { key: 'C' }, 'C1'),
+                React.createElement('li', { key: 'B' }, 'B1'),
+                React.createElement('li', { key: 'E' }, 'E1'),
+                React.createElement('li', { key: 'F' }, 'F1')
+            );
+        }
+    }
+}

+class Todos extends React.Component {
+    constructor(props) {
+        super(props);
+        this.state = { list: [], text: '' };
+    }
+    add = () => {
+        if (this.state.text && this.state.text.length > 0) {
+            this.setState({ list: [...this.state.list, this.state.text] });
+        }
+    }
+    onChange = (event) => {
+        this.setState({ text: event.target.value });
+    }
+    onDel = (index) => {
+        this.state.list.splice(index, 1);
+        this.setState({ list: this.state.list });
+    }
+    render() {
+        var createItem = (itemText, index) => {
+            return React.createElement("li", {}, itemText, React.createElement('button',
+                { onClick: () => this.onDel(index) }, 'X'));
+        };
+        var lists = this.state.list.map(createItem);
+        let ul = React.createElement("ul", {}, ...lists);
+        var input = React.createElement("input", { onKeyup: this.onChange, value: this.state.text });
+        var button = React.createElement("button", { onClick: this.add }, 'Add')
+        return React.createElement('div', {}, input, button, ul);
+    }
+}
let element = React.createElement(Todos, {});
ReactDOM.render(
    element,
    document.getElementById('root')
);
```

5.2 react\constants.js

src\react\constants.js

```
export const TEXT = Symbol.for('TEXT');
export const ELEMENT = Symbol.for('ELEMENT');
export const FUNCTION_COMPONENT = Symbol.for('FUNCTION_COMPONENT');
export const CLASS_COMPONENT = Symbol.for('CLASS_COMPONENT');

+export const MOVE = 'MOVE';
+export const INSERT = 'INSERT';
+export const REMOVE = 'REMOVE';
```

5.3 react\vdom.js

src\react\vdom.js

```
+import { TEXT, ELEMENT, FUNCTION_COMPONENT, CLASS_COMPONENT, MOVE, INSERT, REMOVE } from './constants';
+import { setProps, onlyOne,flatten, patchProps} from './utils';
+const diffQueue = [];
+let updateDepth = 0;

export function createDOM(element) {
    element = onlyOne(element);
    let { $typeof } = element;
```

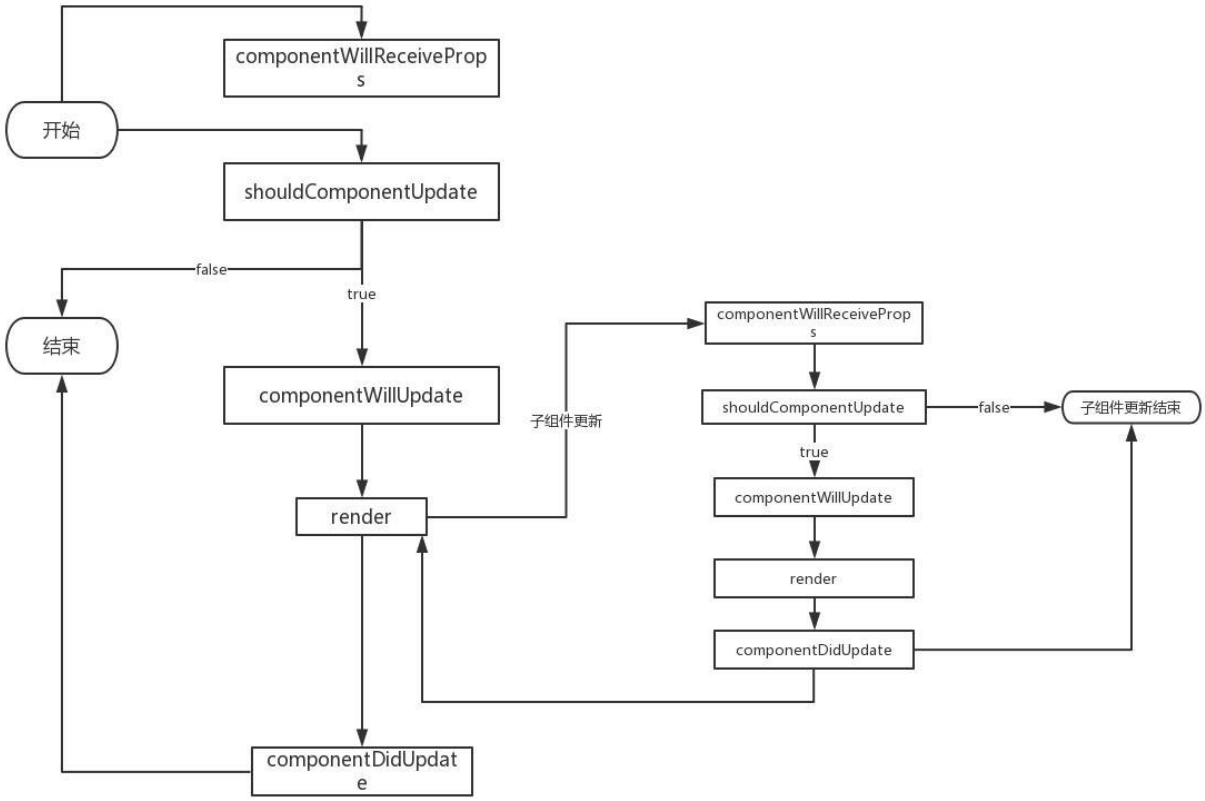
```
    let dom = null;
    if (!$typeof) {
        dom = document.createTextNode(element);
    } else if ($typeof
        dom = document.createTextNode(element.content);
    } else if ($typeof
        dom = createNativeDOM(element);
    } else if ($typeof
        dom = createFunctionComponentDOM(element);
    } else if ($typeof
        dom = createClassComponentDOM(element);
    }
    element.dom = dom;
    return dom;
}

function createNativeDOM(element) {
    let { type, props } = element;
    let dom = document.createElement(type);
    createChildren(element, dom);
    setProps(dom, props);
    return dom;
}

function createChildren(element, parentNode) {
    element.props.children && flatten(element.props.children).forEach((childElement, index) => {
        childElement._mountIndex = index;
        let childDOM = createDOM(childElement);
        parentNode.appendChild(childDOM);
    });
}
function createFunctionComponentDOM(element) {
    let { type, props } = element;
    let renderElement = type(props);
    element.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    renderElement.dom = newDOM;
    return newDOM;
}
function createClassComponentDOM(element) {
    let { type, props } = element;
    let componentInstance = new type(props);
    element.componentInstance = componentInstance;
    let renderElement = componentInstance.render();
    componentInstance.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    return newDOM;
}
export function compareTwoElements(oldElement, newElement) {
    oldElement = onlyOne(oldElement);
    newElement = onlyOne(newElement);
    let currentDOM = oldElement.dom;
    let currentElement = oldElement;
    if (newElement == null) {//如果新节点没有了，直接删除拉倒
        currentDOM.parentNode.removeChild(currentDOM);
        currentElement = null;
    } else if (oldElement.type !== newElement.type) {//如果类型不同
        let newDOM = createDOM(newElement);
        currentDOM.parentNode.replaceChild(newDOM, currentDOM);
        currentElement = newElement;
    } else {
        updateElement(oldElement, newElement);
    }
    return currentElement;
}

function updateElement(oldElement, newElement) {
    let currentDOM = newElement.dom = oldElement.dom;
    if (oldElement.$typeof
        if (oldElement.content !== newElement.content) {
            currentDOM.textContent = newElement.content;
        }
    } else if (oldElement.$typeof
+        updateChildrenElements(currentDOM, oldElement.props.children, newElement.props.children);
        updateDOMProps(currentDOM, oldElement.props, newElement.props);
        oldElement.props = newElement.props;
    } else if (oldElement.$typeof
        updateClassComponent(oldElement, newElement);
        newElement.componentInstance = oldElement.componentInstance;
    } else if (oldElement.$typeof
        updateFunctionComponent(oldElement, newElement);
    }
}
function updateChildrenElements(dom, oldChildrenElements, newChildrenElement) {
+    updateDepth++;
    diff(dom, flatten(oldChildrenElements), (newChildrenElement), diffQueue);
+    updateDepth--;
+    if (updateDepth === 0) {
+        patch(diffQueue);
+        diffQueue.length = 0;
+    }
+}

function diff(parentNode, oldChildrenElements, newChildrenElements, diffQueue) {
    let oldChildrenElementMap = getChildrenElementMap(oldChildrenElements);
    let newChildrenElementMap = getNewChildren(oldChildrenElementMap, newChildrenElements);
+    let lastIndex = 0;
+    for (let i = 0; i < newChildrenElements.length; i++) {
+        let newElement = newChildrenElements[i];//取得新元素
+        if (newElement) {
+            let newKey = (newElement.key) || i.toString();//取得新key
+            let oldElement = oldChildrenElementMap[newKey];
+            if (oldElement === newElement) {
+                if (oldElement._mountIndex < lastIndex) {
```

```
+                    diffQueue.push({
+                        parentNode,
+                        type: MOVE,
+                        fromIndex: oldElement._mountIndex,
+                        toIndex: i
+                    });
+                }
+                lastIndex = Math.max(oldElement._mountIndex, lastIndex);
+            } else {
+                diffQueue.push({
+                    parentNode,
+                    type: INSERT,
+                    toIndex: i,
+                    dom: createDOM(newElement)
+                });
+            }
+            newElement._mountIndex = i;
+        }
+
+    }
+    for (let oldKey in oldChildrenElementMap) {
+        if (!newChildrenElementMap.hasOwnProperty(oldKey)) {
+            let oldElement = oldChildrenElementMap[oldKey];
+            diffQueue.push({
+                parentNode,
+                type: REMOVE,
+                fromIndex: oldElement._mountIndex
+            });
+        }
+    }
+}
+function patch(diffQueue) {
+    let deleteChildren = [];
+    let deleteMap = {};
+    for (let i = 0; i < diffQueue.length; i++) {
+        let difference = diffQueue[i];
+        if (difference.type === MOVE || difference.type === REMOVE) {
+            let fromIndex = difference.fromIndex;
+            let oldChild = difference.parentNode.children[fromIndex];
+            deleteMap[fromIndex] = oldChild;
+            deleteChildren.push(oldChild);
+        }
+    }
+    deleteChildren.forEach(child => {
+        child.parentNode.removeChild(child);
+    });
+
+    for (let k = 0; k < diffQueue.length; k++) {
+        let difference = diffQueue[k];
+        switch (difference.type) {
+            case INSERT:
+                insertChildAt(difference.parentNode, difference.dom, difference.toIndex);
+                break;
+            case MOVE:
+                insertChildAt(difference.parentNode, deleteMap[difference.fromIndex], difference.toIndex);
+                break;
+            default:
+                break;
+        }
+    }
+}
+
+function insertChildAt(parentNode, childNode, index) {
+    let oldChild = parentNode.children[index]
+    oldChild ? parentNode.insertBefore(childNode, oldChild) : parentNode.appendChild(childNode);
+}

function getChildrenElementMap(childrenElements) {
    let childrenElementMap = {};
    for (let i = 0; i < childrenElements.length; i++) {
        let key = childrenElements[i].key || i.toString();
        childrenElementMap[key] = childrenElements[i];
    }
    return childrenElementMap;
}

function getNewChildren(oldChildrenElementMap, newChildrenElements) {
    let newChildrenElementMap = {};
    newChildrenElements.forEach((newChildElement, index) => {
        if (newChildElement) {
            let newKey = newChildElement.key || index.toString();
            let oldChildElement = oldChildrenElementMap[newKey];
            if (canDeepCompare(oldChildElement, newChildElement)) {
                updateElement(oldChildElement, newChildElement);
                newChildrenElements[index] = oldChildElement;
            }
            newChildrenElementMap[newKey] = newChildrenElements[index];
        }
    });
    return newChildrenElementMap;
}

function canDeepCompare(oldChildElement, newChildElement) {
    if (!!oldChildElement && !!newChildElement) {
        return oldChildElement.type
    }
    return false;
}

function updateDOMProps(dom, oldProps, newProps) {
    return patchProps(dom, oldProps, newProps);
}

function updateClassComponent(oldElement, newElement) {
```

```
    let componentInstance = oldElement.componentInstance;
    let updater = componentInstance.$updater;
    let nextProps = newElement.props;
    updater.emitUpdate(nextProps);
}

function updateFunctionComponent(oldElement, newElement) {
    let newRenderElement = newElement.type(newElement.props);
    var oldRenderElement = oldElement.renderElement;
    var currentElement = compareTwoElements(oldRenderElement, newRenderElement);
    newElement.renderElement = currentElement;
}
export function ReactElement($typeof, type, key, ref, props) {
    let element = {
        $typeof,
        type,
        props,
        key,
        ref
    };
    return element;
}
```

6. 支持旧版生命周期 **#**

```
Counter constructor
Counter  componentWillMount
Counter render
ChildCounter componentWillMount
ChildCounter render
ChildCounter componentDidMount
Counter  componentDidMount
Counter  shouldComponentUpdate
Counter  componentWillUpdate
Counter render
ChildCounter componentWillReceiveProps
ChildCounter shouldComponentUpdate
Counter  componentDidUpdate
Counter  shouldComponentUpdate
Counter  componentWillUpdate
Counter render
ChildCounter componentWillReceiveProps
ChildCounter shouldComponentUpdate
ChildCounter  componentWillUpdate
ChildCounter render
ChildCounter  componentDidUpdate
Counter  componentDidUpdate
Counter  shouldComponentUpdate
Counter  componentWillUpdate
Counter render
ChildCounter  componentWillUnmount
Counter  componentDidUpdate
```



6.1 src\index.js

src\index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
+class Counter extends React.Component { // 他会比较两个状态相等就不会刷新视图 PureComponent是浅比较
+    static defaultProps = {
+        name: '珠峰架构'
+    };
+    constructor(props) {
+        super(props);
+        this.state = { number: 0 }
+        console.log('Counter constructor')
+    }
+    componentWillMount() { // 取本地的数据 同步的方式：采用渲染之前获取数据，只渲染一次
+        console.log('Counter  componentWillMount');
+    }
+    componentDidMount() {
+        console.log('Counter  componentDidMount');
+    }
+    handleClick = () => {
+        this.setState({ number: this.state.number + 1 });
+    };
+    // react可以shouldComponentUpdate方法中优化 PureComponent 可以帮我们做这件事
+    shouldComponentUpdate(nextProps, nextState) { // 代表的是下一次的属性 和 下一次的状态
+        console.log('Counter  shouldComponentUpdate');
+        return nextState.number > 1;
+        // return nextState.number!==this.state.number; //如果此函数种返回了false 就不会调用render方法了
+    } //不要随便用setState 可能会死循环
+    componentWillUpdate() {
+        console.log('Counter  componentWillUpdate');
+    }
+    componentDidUpdate() {
+        console.log('Counter  componentDidUpdate');
+    }
+    render() {
+        console.log('Counter render');
+        return (
+
+                {this.state.number}
+                {this.state.number > 3 ? null : }
+                +
+
+        )
+    }
+}
+class ChildCounter extends React.Component {
+    componentWillUnmount() {
+        console.log('ChildCounter  componentWillUnmount')
+    }
+    componentWillMount() {
+        console.log('ChildCounter componentWillMount')
+    }
+    render() {
+        console.log('ChildCounter render')
+        return (
+            {this.props.n}
+        )
+    }
+    componentDidMount() {
+        console.log('ChildCounter componentDidMount')
+    }
+    componentWillReceiveProps(newProps) { // 第一次不会执行，之后属性更新时才会执行
+        console.log('ChildCounter componentWillReceiveProps')
+    }
+    shouldComponentUpdate(nextProps, nextState) {
+        console.log('ChildCounter shouldComponentUpdate')
+        return nextProps.n > 2; //子组件判断接收的属性 是否满足更新条件 为true则更新
+    }
+    componentWillUpdate() {
+        console.log('ChildCounter  componentWillUpdate');
+    }
+    componentDidUpdate() {
+        console.log('ChildCounter  componentDidUpdate');
+    }
+}
let element = React.createElement(Counter, {});
ReactDOM.render(
    element,
    document.getElementById('root')
);
```

6.2 react\vdom.js

src\react\vdom.js

```
import { TEXT, ELEMENT, FUNCTION_COMPONENT, CLASS_COMPONENT, MOVE, INSERT, REMOVE } from './constants';
import { setProps, onlyOne, patchProps, flatten } from './utils';
const diffQueue = [];
let updateDepth = 0;

export function createDOM(element) {
    element = onlyOne(element);
    let { $typeof } = element;
    let dom = null;
    if (!$typeof) {
        dom = document.createTextNode(element);
    } else if ($typeof
        dom = document.createTextNode(element.content);
    } else if ($typeof
        dom = createNativeDOM(element);
    } else if ($typeof
        dom = createFunctionComponentDOM(element);
    } else if ($typeof
        dom = createClassComponentDOM(element);
    }
    element.dom = dom;
```

```
    return dom;
}

function createNativeDOM(element) {
    let { type, props } = element;
    let dom = document.createElement(type);
    createChildren(element, dom);
    setProps(dom, props);
    return dom;
}

function createChildren(element, parentNode) {
    element.props.children && flatten(element.props.children).forEach((childElement, index) => {
        childElement._mountIndex = index;
        let childDOM = createDOM(childElement);
        parentNode.appendChild(childDOM);
    });
}
function createFunctionComponentDOM(element) {
    let { type, props } = element;
    let renderElement = type(props);
    element.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    renderElement.dom = newDOM;
    return newDOM;
}
function createClassComponentDOM(element) {
    let { type, props } = element;
    let componentInstance = new type(props);
+    if (componentInstance.componentWillMount)
+        componentInstance.componentWillMount();
    element.componentInstance = componentInstance;
    let renderElement = componentInstance.render();
    componentInstance.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
+    if (componentInstance.componentDidMount)
+        componentInstance.componentDidMount();
    return newDOM;
}
export function compareTwoElements(oldElement, newElement) {
    oldElement = onlyOne(oldElement);
    newElement = onlyOne(newElement);
    let currentDOM = oldElement.dom;
    let currentElement = oldElement;
    if (newElement == null) {//如果新节点没有了，直接删除拉倒
        currentDOM.parentNode.removeChild(currentDOM);
        currentElement = null;
    } else if (oldElement.type !== newElement.type) {//如果类型不同
        let newDOM = createDOM(newElement);
        currentDOM.parentNode.replaceChild(newDOM, currentDOM);
        currentElement = newElement;
    } else {
        updateElement(oldElement, newElement);
    }
    return currentElement;
}

function updateElement(oldElement, newElement) {
    let currentDOM = newElement.dom = oldElement.dom;
    if (oldElement.$typeof
        if (oldElement.content !== newElement.content) {
            currentDOM.textContent = newElement.content;
        }
    } else if (oldElement.$typeof
        updateChildrenElements(currentDOM, oldElement.props.children, newElement.props.children);
        updateDOMProps(currentDOM, oldElement.props, newElement.props);
        oldElement.props = newElement.props;
    } else if (oldElement.$typeof
        updateClassComponent(oldElement, newElement);
        newElement.componentInstance = oldElement.componentInstance;
    } else if (oldElement.$typeof
        updateFunctionComponent(oldElement, newElement);
    }
}
function updateChildrenElements(dom, oldChildrenElements, newChildrenElement) {
    updateDepth++;
    diff(dom, flatten(oldChildrenElements), flatten(newChildrenElement), diffQueue);
    updateDepth--;
    if (updateDepth
        patch(diffQueue);
        diffQueue.length = 0;
    }
}

function diff(parentNode, oldChildrenElements, newChildrenElements, diffQueue) {
    let oldChildrenElementMap = getChildrenElementMap(oldChildrenElements);
    let newChildrenElementMap = getNewChildren(oldChildrenElementMap, newChildrenElements);
    let lastIndex = 0;

    for (let i = 0; i < newChildrenElements.length; i++) {
        let newElement = newChildrenElements[i];//取得新元素
        if (newElement) {
            let newKey = (newElement.key) || i.toString();//取得新key
            let oldElement = oldChildrenElementMap[newKey];
            if (oldElement
                if (oldElement._mountIndex < lastIndex) {
                    diffQueue.push({
                        parentNode,
                        type: MOVE,
                        from
                        to
                    });
                }
                lastIndex = Math.max(oldElement._mountIndex, lastIndex);
```

```
            } else {
                diffQueue.push({
                    parentNode,
                    type: INSERT,
                    to
                    dom: createDOM(newElement)
                });
            }
            newElement._mountIndex = i;
+        } else {
+            if (oldChildrenElements[i].componentInstance && oldChildrenElements[i].componentInstance.componentWillUnmount) {
+                oldChildrenElements[i].componentInstance.componentWillUnmount();
+            }
+        }
    }
    for (let oldKey in oldChildrenElementMap) {
        if (!newChildrenElementMap.hasOwnProperty(oldKey)) {
            let oldElement = oldChildrenElementMap[oldKey];
            diffQueue.push({
                parentNode,
                type: REMOVE,
                from
            });
        }
    }
}
function patch(diffQueue) {
    let deleteChildren = [];
    let deleteMap = {};
    for (let i = 0; i < diffQueue.length; i++) {
        let difference = diffQueue[i];
        if (difference.type
            let fromIndex = difference.fromIndex;
            let oldChild = difference.parentNode.children[fromIndex];
            deleteMap[fromIndex] = oldChild;
            deleteChildren.push(oldChild);
        }
    }
    deleteChildren.forEach(child => {
        child.parentNode.removeChild(child);
    });

    for (let k = 0; k < diffQueue.length; k++) {
        let difference = diffQueue[k];
        switch (difference.type) {
            case INSERT:
                insertChildAt(difference.parentNode, difference.dom, difference.toIndex);
                break;
            case MOVE:
                insertChildAt(difference.parentNode, deleteMap[difference.fromIndex], difference.toIndex);
                break;
            default:
                break;
        }
    }
}

function insertChildAt(parentNode, childNode, index) {
    let oldChild = parentNode.children[index]
    oldChild ? parentNode.insertBefore(childNode, oldChild) : parentNode.appendChild(childNode);
}

function getChildrenElementMap(childrenElements) {
    let childrenElementMap = {};
    for (let i = 0; i < childrenElements.length; i++) {
        let key = childrenElements[i].key || i.toString();
        childrenElementMap[key] = childrenElements[i];
    }
    return childrenElementMap;
}

function getNewChildren(oldChildrenElementMap, newChildrenElements) {
    let newChildrenElementMap = {};
    newChildrenElements.forEach((newChildElement, index) => {
        if (newChildElement) {
            let newKey = newChildElement.key || index.toString();
            let oldChildElement = oldChildrenElementMap[newKey];
            if (canDeepCompare(oldChildElement, newChildElement)) {
                updateElement(oldChildElement, newChildElement);
                newChildrenElements[index] = oldChildElement;
            }
            newChildrenElementMap[newKey] = newChildrenElements[index];
        }
    });
    return newChildrenElementMap;
}

function canDeepCompare(oldChildElement, newChildElement) {
    if (!!oldChildElement && !!newChildElement) {
        return oldChildElement.type
    }
    return false;
}

function updateDOMProps(dom, oldProps, newProps) {
    return patchProps(dom, oldProps, newProps);
}

function updateClassComponent(oldElement, newElement) {
    let componentInstance = oldElement.componentInstance;
    let updater = componentInstance.$updater;
    let nextProps = newElement.props;
+    if (componentInstance.componentWillReceiveProps) {
+        componentInstance.componentWillReceiveProps(nextProps);
```

```
+    }
    updater.emitUpdate(nextProps);
}

function updateFunctionComponent(oldElement, newElement) {
    let newRenderElement = newElement.type(newElement.props);
    var oldRenderElement = oldElement.renderElement;
    var currentElement = compareTwoElements(oldRenderElement, newRenderElement);
    newElement.renderElement = currentElement;
}
export function ReactElement($typeof, type, key, ref, props) {
    let element = {
        $typeof,
        type,
        props,
        key,
        ref
    };
    return element;
}
```

7. 新版生命周期和ref

7.1 getDerivedStateFromProps

7.1.1 index.js

src/index.js

```
import React from './react';
import ReactDOM from './react-dom';
class Counter extends React.Component {
    static defaultProps = {
        name: '珠峰架构'
    };
    constructor(props) {
        super(props);
        this.state = { number: 0 }
    }

    handleClick = () => {
        this.setState({ number: this.state.number + 1 });
    };

    render() {
        return (
            <div>
                <p>{this.state.number}p>
                <ChildCounter number={this.state.number} />
                <button onClick={this.handleClick}>+button>
            div>
        )
    }
}
class ChildCounter extends React.Component {
    constructor(props) {
        super(props);
        this.state = { number: 0 };
    }
    static getDerivedStateFromProps(nextProps, prevState) {
        const { number } = nextProps;

        if (number % 2 == 0) {
            return { number: number * 2 };
        } else {
            return { number: number * 3 };
        }

        return null;
    }
    render() {
        return (<div>
            {this.state.number}
        div>)
    }

}

ReactDOM.render(
    <Counter />,
    document.getElementById('root')
);
```

7.1.2 react\vdom.js

src\react\vdom.js

```
import { TEXT, ELEMENT, FUNCTION_COMPONENT, CLASS_COMPONENT, MOVE, INSERT, REMOVE } from './constants';
import { setProps, onlyOne, patchProps, flatten } from './utils';
const diffQueue = [];
let updateDepth = 0;

export function createDOM(element) {
    element = onlyOne(element);
    let { $typeof } = element;
    let dom = null;
    if (!$typeof) {
        dom = document.createTextNode(element);
    } else if ($typeof
        dom = document.createTextNode(element.content);
    } else if ($typeof
        dom = createNativeDOM(element);
    } else if ($typeof
        dom = createFunctionComponentDOM(element);
```

```
    } else if ($typeof
        dom = createClassComponentDOM(element);
    }
    element.dom = dom;
    return dom;
}

function createNativeDOM(element) {
    let { type, props } = element;
    let dom = document.createElement(type);
    createChildren(element, dom);
    setProps(dom, props);
    return dom;
}

function createChildren(element, parentNode) {
    element.props.children && flatten(element.props.children).forEach((childElement, index) => {
        childElement._mountIndex = index;
        let childDOM = createDOM(childElement);
        parentNode.appendChild(childDOM);
    });
}
function createFunctionComponentDOM(element) {
    let { type, props } = element;
    let renderElement = type(props);
    element.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    renderElement.dom = newDOM;
    return newDOM;
}
function createClassComponentDOM(element) {
    let { type, props } = element;
    let componentInstance = new type(props);
    if (componentInstance.componentWillMount)
        componentInstance.componentWillMount();
    element.componentInstance = componentInstance;
    let renderElement = componentInstance.render();
    componentInstance.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    if (componentInstance.componentDidMount)
        componentInstance.componentDidMount();
    return newDOM;
}
export function compareTwoElements(oldElement, newElement) {
    oldElement = onlyOne(oldElement);
    newElement = onlyOne(newElement);
    let currentDOM = oldElement.dom;
    let currentElement = oldElement;
    if (newElement == null) {//如果新节点没有了，直接删除拉倒
        currentDOM.parentNode.removeChild(currentDOM);
        currentElement = null;
    } else if (oldElement.type !== newElement.type) {//如果类型不同
        let newDOM = createDOM(newElement);
        currentDOM.parentNode.replaceChild(newDOM, currentDOM);
        currentElement = newElement;
    } else {
        updateElement(oldElement, newElement);
    }
    return currentElement;
}

function updateElement(oldElement, newElement) {
    let currentDOM = newElement.dom = oldElement.dom;
    if (oldElement.$typeof
        if (oldElement.content !== newElement.content) {
            currentDOM.textContent = newElement.content;
        }
    } else if (oldElement.$typeof
        updateChildrenElements(currentDOM, oldElement.props.children, newElement.props.children);
        updateDOMProps(currentDOM, oldElement.props, newElement.props);
        oldElement.props = newElement.props;
    } else if (oldElement.$typeof
        updateClassComponent(oldElement, newElement);
        newElement.componentInstance = oldElement.componentInstance;
    } else if (oldElement.$typeof
        updateFunctionComponent(oldElement, newElement);
    }
}
function updateChildrenElements(dom, oldChildrenElements, newChildrenElement) {
    updateDepth++;
    diff(dom, flatten(oldChildrenElements), flatten(newChildrenElement), diffQueue);
    updateDepth--;
    if (updateDepth
        patch(diffQueue);
        diffQueue.length = 0;
    }
}

function diff(parentNode, oldChildrenElements, newChildrenElements, diffQueue) {
    let oldChildrenElementMap = getChildrenElementMap(oldChildrenElements);
    let newChildrenElementMap = getNewChildren(oldChildrenElementMap, newChildrenElements);
    let lastIndex = 0;

    for (let i = 0; i < newChildrenElements.length; i++) {
        let newElement = newChildrenElements[i];//取得新元素
        if (newElement) {
            let newKey = (newElement.key) || i.toString();//取得新key
            let oldElement = oldChildrenElementMap[newKey];
            if (oldElement
                if (oldElement._mountIndex < lastIndex) {
                    diffQueue.push({
                        parentNode,
                        type: MOVE,
                        from
```

```
                        to
                    });
                }
                lastIndex = Math.max(oldElement._mountIndex, lastIndex);
            } else {
                diffQueue.push({
                    parentNode,
                    type: INSERT,
                    to
                    dom: createDOM(newElement)
                });
            }
            newElement._mountIndex = i;
        } else {
            if (oldChildrenElements[i].componentInstance && oldChildrenElements[i].componentInstance.componentWillUnmount) {
                oldChildrenElements[i].componentInstance.componentWillUnmount();
            }
        }

    }
    for (let oldKey in oldChildrenElementMap) {
        if (!newChildrenElementMap.hasOwnProperty(oldKey)) {
            let oldElement = oldChildrenElementMap[oldKey];
            diffQueue.push({
                parentNode,
                type: REMOVE,
                from
            });
        }
    }
}
function patch(diffQueue) {
    let deleteChildren = [];
    let deleteMap = {};
    for (let i = 0; i < diffQueue.length; i++) {
        let difference = diffQueue[i];
        if (difference.type
            let fromIndex = difference.fromIndex;
            let oldChild = difference.parentNode.children[fromIndex];
            deleteMap[fromIndex] = oldChild;
            deleteChildren.push(oldChild);
        }
    }
    deleteChildren.forEach(child => {
        child.parentNode.removeChild(child);
    });

    for (let k = 0; k < diffQueue.length; k++) {
        let difference = diffQueue[k];
        switch (difference.type) {
            case INSERT:
                insertChildAt(difference.parentNode, difference.dom, difference.toIndex);
                break;
            case MOVE:
                insertChildAt(difference.parentNode, deleteMap[difference.fromIndex], difference.toIndex);
                break;
            default:
                break;
        }
    }
}

function insertChildAt(parentNode, childNode, index) {
    let oldChild = parentNode.children[index]
    oldChild ? parentNode.insertBefore(childNode, oldChild) : parentNode.appendChild(childNode);
}

function getChildrenElementMap(childrenElements) {
    let childrenElementMap = {};
    for (let i = 0; i < childrenElements.length; i++) {
        let key = childrenElements[i].key || i.toString();
        childrenElementMap[key] = childrenElements[i];
    }
    return childrenElementMap;
}

function getNewChildren(oldChildrenElementMap, newChildrenElements) {
    let newChildrenElementMap = {};
    newChildrenElements.forEach((newChildElement, index) => {
        if (newChildElement) {
            let newKey = newChildElement.key || index.toString();
            let oldChildElement = oldChildrenElementMap[newKey];
            if (canDeepCompare(oldChildElement, newChildElement)) {
                updateElement(oldChildElement, newChildElement);
                newChildrenElements[index] = oldChildElement;
            }
            newChildrenElementMap[newKey] = newChildrenElements[index];
        }
    });
    return newChildrenElementMap;
}

function canDeepCompare(oldChildElement, newChildElement) {
    if (!!oldChildElement && !!newChildElement) {
        return oldChildElement.type
    }
    return false;
}

function updateDOMProps(dom, oldProps, newProps) {
    return patchProps(dom, oldProps, newProps);
}

function updateClassComponent(oldElement, newElement) {
```

```
    let componentInstance = oldElement.componentInstance;
    let updater = componentInstance.$updater;
    let nextProps = newElement.props;
    if (componentInstance.componentWillReceiveProps) {
        componentInstance.componentWillReceiveProps(nextProps);
    }
+   if (newElement.type.getDerivedStateFromProps) {
+       let newState = newElement.type.getDerivedStateFromProps(nextProps, componentInstance.state);
+       if (newState)
+           componentInstance.state = newState;
+   }
    updater.emitUpdate(nextProps);
}

function updateFunctionComponent(oldElement, newElement) {
    let newRenderElement = newElement.type(newElement.props);
    var oldRenderElement = oldElement.renderElement;
    var currentElement = compareTwoElements(oldRenderElement, newRenderElement);
    newElement.renderElement = currentElement;
}
export function ReactElement($typeof, type, key, ref, props) {
    let element = {
        $typeof,
        type,
        props,
        key,
        ref
    };
    return element;
}
```

7.2.1 index.js

src/index.js

```
import React from './react';
import ReactDOM from './react-dom';
+class ScrollingList extends React.Component {
+    wrapper
+    timeID
+    constructor(props) {
+        super(props);
+        this.state = { messages: [] }
+        this.wrapper = React.createRef();
+    }

+    addMessage = () => {
+        this.setState(state => ({
+            messages: [`${state.messages.length}`, ...state.messages],
+        }))
+    }
+    componentDidMount() {
+        this.timeID = window.setInterval(() => {//设置定时器
+            this.addMessage();
+        }, 3000)
+    }
+    componentWillUnmount() {//清除定时器
+        window.clearInterval(this.timeID);
+    }
+    getSnapshotBeforeUpdate = () => {//很关键的，我们获取当前rootNode的scrollHeight，传到componentDidUpdate 的参数perScrollHeight
+        return this.wrapper.current.scrollHeight;
+    }
+    componentDidUpdate(pervProps, pervState, prevScrollHeight) {
+        const curScrollTop = this.wrapper.current.scrollTop;//当前向上卷去的高度
+        //当前向上卷去的高度加上增加的内容高度
+        this.wrapper.current.scrollTop = curScrollTop + (this.wrapper.current.scrollHeight - prevScrollHeight);
+    }
+    render() {
+        let style = {
+            height: '100px',
+            width: '200px',
+            border: '1px solid red',
+            overflow: 'auto'
+        }
+        return (
+
+                {this.state.messages.map((message, index) => (
+                    {message}
+                ))}
+
+        );
+    }
+}

ReactDOM.render(
    ,
    document.getElementById('root')
);
```

7.2.2 react\component.js

src\react\component.js

```
import { isFunction } from './utils';
import { compareTwoElements } from './vdom';
export let updateQueue = {
    updaters: [],
    isPending: false,
    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;
    },
};
class Updater {
    constructor(instance) {
        this.instance = instance;
        this.pendingStates = [];
        this.nextProps = null;
    }
    addState(partialState) {
        this.pendingStates.push(partialState);
        this.emitUpdate();
    }
    emitUpdate(nextProps) {
        this.nextProps = nextProps;
        nextProps || !updateQueue.isPending
            ? this.updateComponent()
            : updateQueue.add(this);
    }
    updateComponent() {
        let { instance, pendingStates, nextProps } = this;
        if (nextProps || pendingStates.length > 0) {
            shouldUpdate(
                instance,
                nextProps,
                this.getState()
            );
        }
    }

    getState() {
        let { instance, pendingStates } = this;
        let { state } = instance;
        if (pendingStates.length) {
            pendingStates.forEach(nextState => {
                if (isFunction(nextState)) {
                    nextState = nextState.call(instance, state);
                }
                state = { ...state, ...nextState };
            });
            pendingStates.length = 0;
        }
        return state;
    }
}
function shouldUpdate(component, nextProps, nextState) {
    component.props = nextProps;
    component.state = nextState;
    if (component.shouldComponentUpdate && !component.shouldComponentUpdate(nextProps, nextState)) {
        return;
    }
    component.forceUpdate();
}

class Component {
    constructor(props) {
        this.props = props;
        this.$updater = new Updater(this);
        this.state = {};
        this.nextProps = null;
    }
    setState(partialState) {
        this.$updater.addState(partialState);
    }
    forceUpdate() {
        let { props, state, renderElement: oldRenderElement } = this;
        if (this.componentWillUpdate) {
            this.componentWillUpdate(props, state);
        }
+        let { getSnapshotBeforeUpdate } = this;
+        let extraArgs = getSnapshotBeforeUpdate && getSnapshotBeforeUpdate();
        let newRenderElement = this.render();
        let currentElement = compareTwoElements(oldRenderElement, newRenderElement);
        this.renderElement = currentElement;
        if (this.componentDidUpdate) {
+            this.componentDidUpdate(props, state, extraArgs);
        }
    }
}
Component.prototype.isReactComponent = {};
export {
    Component
}
```

7.2.3 react\v dom.js

src\react\vdom.js

```
import { TEXT, ELEMENT, FUNCTION_COMPONENT, CLASS_COMPONENT, MOVE, INSERT, REMOVE } from './constants';
import { setProps, onlyOne, patchProps, flatten } from './utils';
const diffQueue = [];
let updateDepth = 0;

export function createDOM(element) {
    element = onlyOne(element);
    let { $typeof } = element;
    let dom = null;
    if (!$typeof) {
        dom = document.createTextNode(element);
    } else if ($typeof
        dom = document.createTextNode(element.content);
    } else if ($typeof
        dom = createNativeDOM(element);
    } else if ($typeof
        dom = createFunctionComponentDOM(element);
    } else if ($typeof
        dom = createClassComponentDOM(element);
    }
    element.dom = dom;
    return dom;
}

function createNativeDOM(element) {
    let { type, props, ref } = element;
    let dom = document.createElement(type);
    createChildren(element, dom);
    setProps(dom, props);
+    if (ref)
+        ref.current = dom;
    return dom;
}

function createChildren(element, parentNode) {
    element.props.children && flatten(element.props.children).forEach((childElement, index) => {
        childElement._mountIndex = index;
        let childDOM = createDOM(childElement);
        parentNode.appendChild(childDOM);
    });
}
function createFunctionComponentDOM(element) {
    let { type, props } = element;
    let renderElement = type(props);
    element.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    renderElement.dom = newDOM;
    return newDOM;
}
function createClassComponentDOM(element) {
    let { type, props, ref } = element;
    let componentInstance = new type(props);
+    if (ref)
+        ref.current = componentInstance;
    if (componentInstance.componentWillMount)
        componentInstance.componentWillMount();
    element.componentInstance = componentInstance;
    let renderElement = componentInstance.render();
    componentInstance.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    if (componentInstance.componentDidMount)
        componentInstance.componentDidMount();
    return newDOM;
}
export function compareTwoElements(oldElement, newElement) {
    oldElement = onlyOne(oldElement);
    newElement = onlyOne(newElement);
    let currentDOM = oldElement.dom;
    let currentElement = oldElement;
    if (newElement == null) {//如果新节点没有了，直接删除拉倒
        currentDOM.parentNode.removeChild(currentDOM);
        currentElement = null;
    } else if (oldElement.type !== newElement.type) {//如果类型不同
        let newDOM = createDOM(newElement);
        currentDOM.parentNode.replaceChild(newDOM, currentDOM);
        currentElement = newElement;
    } else {
        updateElement(oldElement, newElement);
    }
    return currentElement;
}

function updateElement(oldElement, newElement) {
    let currentDOM = newElement.dom = oldElement.dom;
    if (oldElement.$typeof
        if (oldElement.content !== newElement.content) {
            currentDOM.textContent = newElement.content;
        }
    } else if (oldElement.$typeof
        updateChildrenElements(currentDOM, oldElement.props.children, newElement.props.children);
        updateDOMProps(currentDOM, oldElement.props, newElement.props);
        oldElement.props = newElement.props;
    } else if (oldElement.$typeof
        updateClassComponent(oldElement, newElement);
        newElement.componentInstance = oldElement.componentInstance;
    } else if (oldElement.$typeof
        updateFunctionComponent(oldElement, newElement);
    }
}
function updateChildrenElements(dom, oldChildrenElements, newChildrenElement) {
    updateDepth++;
    diff(dom, flatten(oldChildrenElements), flatten(newChildrenElement), diffQueue);
```

```
    updateDepth--;
    if (updateDepth
        patch(diffQueue);
        diffQueue.length = 0;
    }
}

function diff(parentNode, oldChildrenElements, newChildrenElements, diffQueue) {
    let oldChildrenElementMap = getChildrenElementMap(oldChildrenElements);
    let newChildrenElementMap = getNewChildren(oldChildrenElementMap, newChildrenElements);
    let lastIndex = 0;

    for (let i = 0; i < newChildrenElements.length; i++) {
        let newElement = newChildrenElements[i];//取得新元素
        if (newElement) {
            let newKey = (newElement.key) || i.toString();//取得新key
            let oldElement = oldChildrenElementMap[newKey];
            if (oldElement
                if (oldElement._mountIndex < lastIndex) {
                    diffQueue.push({
                        parentNode,
                        type: MOVE,
                        from
                        to
                    });
                }
                lastIndex = Math.max(oldElement._mountIndex, lastIndex);
            } else {
                diffQueue.push({
                    parentNode,
                    type: INSERT,
                    to
                    dom: createDOM(newElement)
                });
            }
            newElement._mountIndex = i;
        } else {
            if (oldChildrenElements[i].componentInstance && oldChildrenElements[i].componentInstance.componentWillUnmount) {
                oldChildrenElements[i].componentInstance.componentWillUnmount();
            }
        }

    }
    for (let oldKey in oldChildrenElementMap) {
        if (!newChildrenElementMap.hasOwnProperty(oldKey)) {
            let oldElement = oldChildrenElementMap[oldKey];
            diffQueue.push({
                parentNode,
                type: REMOVE,
                from
            });
        }
    }
}
function patch(diffQueue) {
    let deleteChildren = [];
    let deleteMap = {};
    for (let i = 0; i < diffQueue.length; i++) {
        let difference = diffQueue[i];
        if (difference.type
            let fromIndex = difference.fromIndex;
            let oldChild = difference.parentNode.children[fromIndex];
            deleteMap[fromIndex] = oldChild;
            deleteChildren.push(oldChild);
        }
    }
    deleteChildren.forEach(child => {
        child.parentNode.removeChild(child);
    });

    for (let k = 0; k < diffQueue.length; k++) {
        let difference = diffQueue[k];
        switch (difference.type) {
            case INSERT:
                insertChildAt(difference.parentNode, difference.dom, difference.toIndex);
                break;
            case MOVE:
                insertChildAt(difference.parentNode, deleteMap[difference.fromIndex], difference.toIndex);
                break;
            default:
                break;
        }
    }
}

function insertChildAt(parentNode, childNode, index) {
    let oldChild = parentNode.children[index]
    oldChild ? parentNode.insertBefore(childNode, oldChild) : parentNode.appendChild(childNode);
}

function getChildrenElementMap(childrenElements) {
    let childrenElementMap = {};
    for (let i = 0; i < childrenElements.length; i++) {
        let key = childrenElements[i].key || i.toString();
        childrenElementMap[key] = childrenElements[i];
    }
    return childrenElementMap;
}

function getNewChildren(oldChildrenElementMap, newChildrenElements) {
    let newChildrenElementMap = {};
    newChildrenElements.forEach((newChildElement, index) => {
        if (newChildElement) {
            let newKey = newChildElement.key || index.toString();
```

```
            let oldChildElement = oldChildrenElementMap[newKey];
            if (canDeepCompare(oldChildElement, newChildElement)) {
                updateElement(oldChildElement, newChildElement);
                newChildrenElements[index] = oldChildElement;
            }
            newChildrenElementMap[newKey] = newChildrenElements[index];
        }
    });
    return newChildrenElementMap;
}

function canDeepCompare(oldChildElement, newChildElement) {
    if (!!oldChildElement && !!newChildElement) {
        return oldChildElement.type
    }
    return false;
}

function updateDOMProps(dom, oldProps, newProps) {
    return patchProps(dom, oldProps, newProps);
}

function updateClassComponent(oldElement, newElement) {
    let componentInstance = oldElement.componentInstance;
    let updater = componentInstance.$updater;
    let nextProps = newElement.props;
    if (componentInstance.componentWillReceiveProps) {
        componentInstance.componentWillReceiveProps(nextProps);
    }
    if (newElement.type.getDerivedStateFromProps) {
        let newState = newElement.type.getDerivedStateFromProps(nextProps, componentInstance.state);
        if (newState)
            componentInstance.state = newState;
    }
    updater.emitUpdate(nextProps);
}

function updateFunctionComponent(oldElement, newElement) {
    let newRenderElement = newElement.type(newElement.props);
    var oldRenderElement = oldElement.renderElement;
    var currentElement = compareTwoElements(oldRenderElement, newRenderElement);
    newElement.renderElement = currentElement;
}
export function ReactElement($typeof, type, key, ref, props) {
    let element = {
        $typeof,
        type,
        props,
        key,
        ref
    };
    return element;
}
```

8. context

8.1 src\index.js

src\index.js

```
import React, { Component } from './react';
import ReactDOM from './react-dom';
let ThemeContext = React.createContext(null);
let root = document.querySelector('#root');
class Header extends Component {
    static contextType = ThemeContext;
    render() {
        return (
            <div style={{ border: `5px solid ${this.context.color}`, padding: '5px' }}>
                header
                <Title />
            div>
        )
    }
}
class Title extends Component {
    static contextType = ThemeContext;
    render() {
        return (
            <div style={{ border: `5px solid ${this.context.color}` }}>
                title
            div>
        )
    }
}
class Main extends Component {
    static contextType = ThemeContext;
    render() {
        return (
            <div style={{ border: `5px solid ${this.context.color}`, margin: '5px', padding: '5px' }}>
                main
                <Content />
            div>
        )
    }
}
class Content extends Component {
    static contextType = ThemeContext;
    render() {
        return (
            <div style={{ border: `5px solid ${this.context.color}`, padding: '5px' }}>
                Content
                <button onClick={() => this.context.changeColor('red')} style={{ color: 'red' }}>红色button>
```

```
                <button onClick={() => this.context.changeColor('green')} style={{ color: 'green' }}>绿色button>
            div>
        )
    }
}

class ClassPage extends Component {
    constructor(props) {
        super(props);
        this.state = { color: 'red' };
    }
    changeColor = (color) => {
        this.setState({ color });
    }
    render() {
        let contextVal = { changeColor: this.changeColor, color: this.state.color };
        return (
            <ThemeContext.Provider value={contextVal}>
                <div style={{ margin: '10px', border: `5px solid ${this.state.color}`, padding: '5px', width: '200px' }}>
                    page
                    <Header />
                    <Main />
                div>
            ThemeContext.Provider>

        )
    }
}
class FunctionHeader extends Component {
    render() {
        return (
            <ThemeContext.Consumer>
                {
                    (value) => (
                        <div style={{ border: `5px solid ${value.color}`, padding: '5px' }}>
                            header
                            <FunctionTitle />
                        div>
                    )
                }
            ThemeContext.Consumer>
        )
    }
}
class FunctionTitle extends Component {
    render() {
        return (
            <ThemeContext.Consumer>
                {
                    (value) => (
                        <div style={{ border: `5px solid ${value.color}` }}>
                            title
                        div>
                    )
                }
            ThemeContext.Consumer>
        )
    }
}
class FunctionMain extends Component {
    render() {
        return (
            <ThemeContext.Consumer>
                {
                    (value) => (
                        <div style={{ border: `5px solid ${value.color}`, margin: '5px', padding: '5px' }}>
                            main
                            <FunctionContent />
                        div>
                    )
                }
            ThemeContext.Consumer>
        )
    }
}
class FunctionContent extends Component {
    render() {
        return (
            <ThemeContext.Consumer>
                {
                    (value) => (
                        <div style={{ border: `5px solid ${value.color}`, padding: '5px' }}>
                            Content
                            <button onClick={() => value.changeColor('red')} style={{ color: 'red' }}>红色button>
                            <button onClick={() => value.changeColor('green')} style={{ color: 'green' }}>绿色button>
                        div>
                    )
                }
            ThemeContext.Consumer>
        )
    }
}
class FunctionPage extends Component {
    constructor(props) {
        super(props);
        this.state = { color: 'red' };
    }
    changeColor = (color) => {
        this.setState({ color });
    }
    render() {
        let contextVal = { changeColor: this.changeColor, color: this.state.color };
        return (
            <ThemeContext.Provider value={contextVal}>
```

```
                <div style={{ margin: '10px', border: `5px solid ${this.state.color}`, padding: '5px', width: '200px' }}>
                    page
                    <FunctionHeader />
                    <FunctionMain />
                div>
            ThemeContext.Provider>

        )
    }
}
ReactDOM.render(<FunctionPage />, root);
```

8.2 react\index.js

src\react\index.js

```
import { TEXT, ELEMENT, FUNCTION_COMPONENT, CLASS_COMPONENT } from './constants';
import { ReactElement } from './vdom';
import { Component } from './component';
import { onlyOne } from './utils';
function createElement(type, config = {}, ...children) {
    delete config.__source;
    delete config.__self;
    let { key, ref, ...props } = config;
    let $typeof = null;
    if (typeof type
        $typeof = ELEMENT;
    } else if (typeof type
        $typeof = CLASS_COMPONENT;
    } else if (typeof type
        $typeof = FUNCTION_COMPONENT;
    }
    props.children = children.map(item => typeof item
        : { $typeof: TEXT, type: TEXT, content: item });
    return ReactElement($typeof, type, key, ref, props);
}
+function createContext(defaultValue) {
+    Provider.value = defaultValue;
+    function Provider(props) {
+        Provider.value = props.value;
+        return props.children;
+    }
+    function Consumer(props) {
+        return onlyOne(props.children)(Provider.value);
+    }
+    return {
+        Provider,
+        Consumer
+    }
+}
export {
    Component
}
const React = {
    createElement,
    Component,
+    createContext
}
export default React;
```

8.3 react\vdom.js

src\react\vdom.js

```
import { TEXT, ELEMENT, FUNCTION_COMPONENT, CLASS_COMPONENT, MOVE, INSERT, REMOVE } from './constants';
import { setProps, onlyOne, patchProps, flatten } from './utils';
const diffQueue = [];
let updateDepth = 0;

export function createDOM(element) {
    element = onlyOne(element);
    let { $typeof } = element;
    let dom = null;
    if (!$typeof) {
        dom = document.createTextNode(element);
    } else if ($typeof
        dom = document.createTextNode(element.content);
    } else if ($typeof
        dom = createNativeDOM(element);
    } else if ($typeof
        dom = createFunctionComponentDOM(element);
    } else if ($typeof
        dom = createClassComponentDOM(element);
    }
    element.dom = dom;
    return dom;
}

function createNativeDOM(element) {
    let { type, props, ref } = element;
    let dom = document.createElement(type);
    createChildren(element, dom);
    setProps(dom, props);
    if (ref)
        ref.current = dom;
    return dom;
}

function createChildren(element, parentNode) {
    element.props.children && flatten(element.props.children).forEach((childElement, index) => {
        childElement._mountIndex = index;
        let childDOM = createDOM(childElement);
        parentNode.appendChild(childDOM);
    });
```

```
}
function createFunctionComponentDOM(element) {
    let { type, props } = element;
    let renderElement = type(props);
    element.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    renderElement.dom = newDOM;
    return newDOM;
}
function createClassComponentDOM(element) {
    let { type, props, ref } = element;
    let componentInstance = new type(props);
+    if (element.type.contextType) {
+        componentInstance.context = element.type.contextType.Provider.value;
+    }
    if (ref)
        ref.current = componentInstance;
    if (componentInstance.componentWillMount)
        componentInstance.componentWillMount();
    element.componentInstance = componentInstance;
    let renderElement = componentInstance.render();
    componentInstance.renderElement = renderElement;
    let newDOM = createDOM(renderElement);
    if (componentInstance.componentDidMount)
        componentInstance.componentDidMount();
    return newDOM;
}
export function compareTwoElements(oldElement, newElement) {
    oldElement = onlyOne(oldElement);
    newElement = onlyOne(newElement);
    let currentDOM = oldElement.dom;
    let currentElement = oldElement;
    if (newElement == null) {//如果新节点没有了，直接删除拉倒
        currentDOM.parentNode.removeChild(currentDOM);
        currentElement = null;
    } else if (oldElement.type !== newElement.type) {//如果类型不同
        let newDOM = createDOM(newElement);
        currentDOM.parentNode.replaceChild(newDOM, currentDOM);
        currentElement = newElement;
    } else {
        updateElement(oldElement, newElement);
    }
    return currentElement;
}

function updateElement(oldElement, newElement) {
    let currentDOM = newElement.dom = oldElement.dom;
    if (oldElement.$typeof
        if (oldElement.content !== newElement.content) {
            currentDOM.textContent = newElement.content;
        }
    } else if (oldElement.$typeof
        updateChildrenElements(currentDOM, oldElement.props.children, newElement.props.children);
        updateDOMProps(currentDOM, oldElement.props, newElement.props);
        oldElement.props = newElement.props;
    } else if (oldElement.$typeof
        updateClassComponent(oldElement, newElement);
        newElement.componentInstance = oldElement.componentInstance;
    } else if (oldElement.$typeof
        updateFunctionComponent(oldElement, newElement);
    }
}
function updateChildrenElements(dom, oldChildrenElements, newChildrenElement) {
    updateDepth++;
    diff(dom, flatten(oldChildrenElements), flatten(newChildrenElement), diffQueue);
    updateDepth--;
    if (updateDepth
        patch(diffQueue);
        diffQueue.length = 0;
    }
}

function diff(parentNode, oldChildrenElements, newChildrenElements, diffQueue) {
    let oldChildrenElementMap = getChildrenElementMap(oldChildrenElements);
    let newChildrenElementMap = getNewChildren(oldChildrenElementMap, newChildrenElements);
    let lastIndex = 0;

    for (let i = 0; i < newChildrenElements.length; i++) {
        let newElement = newChildrenElements[i];//取得新元素
        if (newElement) {
            let newKey = (newElement.key) || i.toString();//取得新key
            let oldElement = oldChildrenElementMap[newKey];
            if (oldElement
                if (oldElement._mountIndex < lastIndex) {
                    diffQueue.push({
                        parentNode,
                        type: MOVE,
                        from
                        to
                    });
                }
                lastIndex = Math.max(oldElement._mountIndex, lastIndex);
            } else {
                diffQueue.push({
                    parentNode,
                    type: INSERT,
                    to
                    dom: createDOM(newElement)
                });
            }
            newElement._mountIndex = i;
        } else {
            if (oldChildrenElements[i].componentInstance && oldChildrenElements[i].componentInstance.componentWillUnmount) {
                oldChildrenElements[i].componentInstance.componentWillUnmount();
```

```
            }
        }

    }
    for (let oldKey in oldChildrenElementMap) {
        if (!newChildrenElementMap.hasOwnProperty(oldKey)) {
            let oldElement = oldChildrenElementMap[oldKey];
            diffQueue.push({
                parentNode,
                type: REMOVE,
                from
            });
        }
    }
}
function patch(diffQueue) {
    let deleteChildren = [];
    let deleteMap = {};
    for (let i = 0; i < diffQueue.length; i++) {
        let difference = diffQueue[i];
        if (difference.type
            let fromIndex = difference.fromIndex;
            let oldChild = difference.parentNode.children[fromIndex];
            deleteMap[fromIndex] = oldChild;
            deleteChildren.push(oldChild);
        }
    }
    deleteChildren.forEach(child => {
        child.parentNode.removeChild(child);
    });

    for (let k = 0; k < diffQueue.length; k++) {
        let difference = diffQueue[k];
        switch (difference.type) {
            case INSERT:
                insertChildAt(difference.parentNode, difference.dom, difference.toIndex);
                break;
            case MOVE:
                insertChildAt(difference.parentNode, deleteMap[difference.fromIndex], difference.toIndex);
                break;
            default:
                break;
        }
    }
}

function insertChildAt(parentNode, childNode, index) {
    let oldChild = parentNode.children[index]
    oldChild ? parentNode.insertBefore(childNode, oldChild) : parentNode.appendChild(childNode);
}

function getChildrenElementMap(childrenElements) {
    let childrenElementMap = {};
    for (let i = 0; i < childrenElements.length; i++) {
        let key = childrenElements[i].key || i.toString();
        childrenElementMap[key] = childrenElements[i];
    }
    return childrenElementMap;
}

function getNewChildren(oldChildrenElementMap, newChildrenElements) {
    let newChildrenElementMap = {};
    newChildrenElements.forEach((newChildElement, index) => {
        if (newChildElement) {
            let newKey = newChildElement.key || index.toString();
            let oldChildElement = oldChildrenElementMap[newKey];
            if (canDeepCompare(oldChildElement, newChildElement)) {
                updateElement(oldChildElement, newChildElement);
                newChildrenElements[index] = oldChildElement;
            }
            newChildrenElementMap[newKey] = newChildrenElements[index];
        }
    });
    return newChildrenElementMap;
}

function canDeepCompare(oldChildElement, newChildElement) {
    if (!!oldChildElement && !!newChildElement) {
        return oldChildElement.type
    }
    return false;
}

function updateDOMProps(dom, oldProps, newProps) {
    return patchProps(dom, oldProps, newProps);
}

function updateClassComponent(oldElement, newElement) {
    let componentInstance = oldElement.componentInstance;
    let updater = componentInstance.$updater;
    let nextProps = newElement.props;
+    if (oldElement.type.contextType) {
+        componentInstance.context = oldElement.type.contextType.Provider.value;
+    }
    if (componentInstance.componentWillReceiveProps) {
        componentInstance.componentWillReceiveProps(nextProps);
    }
    if (newElement.type.getDerivedStateFromProps) {
        let newState = newElement.type.getDerivedStateFromProps(nextProps, componentInstance.state);
        if (newState)
            componentInstance.state = newState;
    }
    updater.emitUpdate(nextProps);
}
```

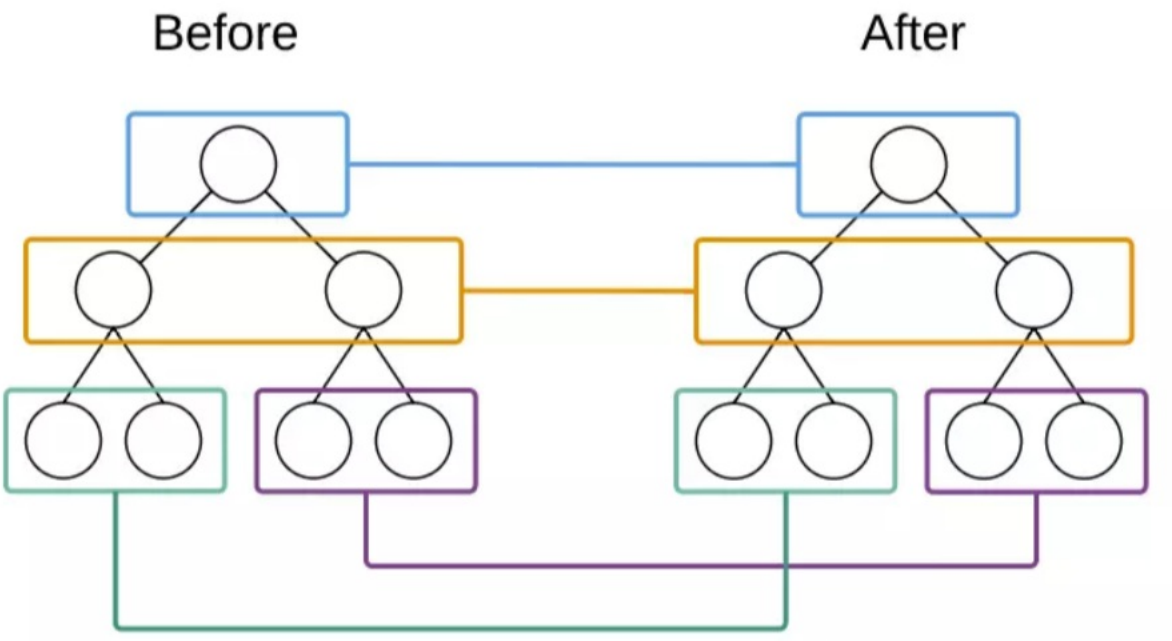
```
function updateFunctionComponent(oldElement, newElement) {
    let newRenderElement = newElement.type(newElement.props);
    var oldRenderElement = oldElement.renderElement;
    var currentElement = compareTwoElements(oldRenderElement, newRenderElement);
    newElement.renderElement = currentElement;
}
export function ReactElement($typeof, type, key, ref, props) {
    let element = {
        $typeof,
        type,
        props,
        key,
        ref
    };
    return element;
}
```

9. DOM-DIFF

- DOM 节点跨层级的移动操作特别少，可以忽略不计
- 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
- 对于同一层级的一组子节点，它们可以通过唯一 key进行区分
- DIFF算法在执行时有三个维度，分别是Tree DIFF、Component DIFF和Element DIFF，执行时按顺序依次执行，它们的差异仅仅因为DIFF粒度不同、执行先后顺序不同

9.1 Tree DIFF

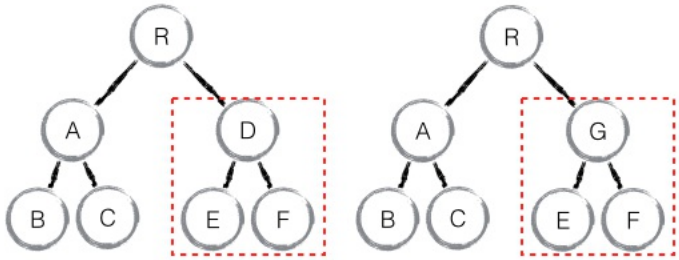
- Tree DIFF是对树的每一层进行遍历，如果某组件不存在了，则会直接销毁



- 当出现节点跨层级移动时，并不会出现想象中的移动操作，而是以 A 为根节点的树被整个重新创建

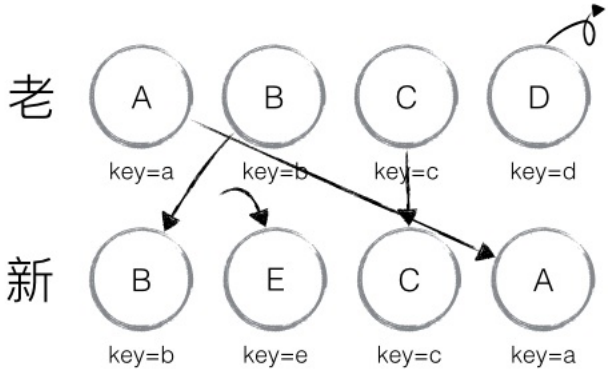
9.2 Component DIFF

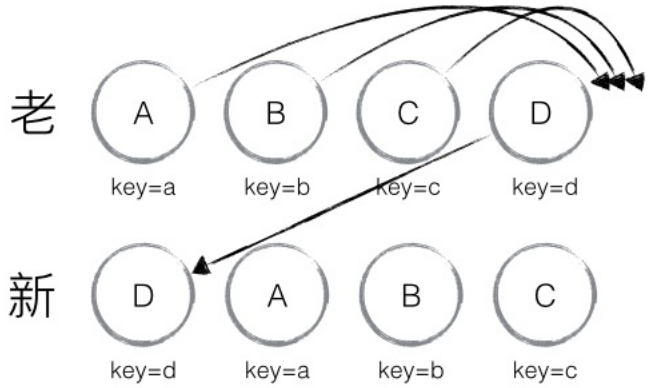
- 如果是同一类型的组件，按照原策略继续比较
- 类型不同则直接替换



9.3 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(插入)、MOVE(移动)和 REMOVE(删除)
    - INSERT: 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: 在老集合有新 component 类型，就需要做更新和移动操作，可以复用以前的 DOM 节点
    - REMOVE: 老 component 不在新集合里的，也需要执行删除操作


老
A
B
C
D
key=a
key=b
key=c
key=d
新
B
E
C
A
key=b
key=e
key=c
key=a


老
A
B
C
D
key=a
key=b
key=c
key=d
新
D
A
B
C
key=d
key=a
key=b
key=c