---

link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=371 sentences=1093, words=10763

1.核心知识

1.1 fiber

- 为了让渲染的过程可以不断，我们可以把整个渲染任务分成若干个task(工作单元),每个工作单元就是一个fiber
- 每个虚拟DOM节点内部表示为一个fiber对象
- render阶段会根据虚拟DOM以深度优先的方式构建Fiber树

1.1.1 fiber数据结构

```
let element = {
    type: 'div',
    key: 'A',
    props: {
        style,
        children: [
            'A文本',
            { type: 'div', key: 'B1', props: { style, children: 'B1文本' } },
            { type: 'div', key: 'B2', props: { style, children: 'B2文本' } }
        ]
    }
}
```

1.1.2 构建和完成

1.1.3 案例

1.2 updateQueue

- updateQueue (https://www.processon.com/diagraming/618e3c0af346fb6e389c44ad)

```
function initializeUpdateQueue(fiber) {
    const queue = {

        shared: {
            pending: null
        }
    };
    fiber.updateQueue = queue;
}

function createUpdate() {
    return {};
}

function enqueueUpdate(fiber, update) {

    const updateQueue = fiber.updateQueue;
    const sharedQueue = updateQueue.shared;

    const pending = sharedQueue.pending;

    if (!pending) {

        update.next = update;
    } else {

        update.next = pending.next;

        pending.next = update;
    }

    sharedQueue.pending = update;
}
let fiber = { baseState: { number: 0 } };
initializeUpdateQueue(fiber);
const update1 = createUpdate();
update1.payload = { number: 1 };
enqueueUpdate(fiber, update1);
const update2 = createUpdate();
update2.payload = { number: 2 };
enqueueUpdate(fiber, update2);
const update3 = createUpdate();
update3.payload = { number: 3 };
enqueueUpdate(fiber, update3);
console.log(fiber.updateQueue.shared.pending);
```

1.3 位运算

1.3.1 按位与(&)

- 两个输入数的同一位都为1才为1

1.3.2 按位或(|)

- 两个输入数的同一位只要有一个为1就是1

```
const NoFlags = 0b00000000000000000000;
const Placement = 0b00000000000000000010;
const Update = 0b00000000000000000100;
const Deletion = 0b00000000000000001000;
const PlacementAndUpdate = 0b00000000000000000110;
010
100
110
console.log((Placement | Update) === PlacementAndUpdate);
110
010
100
console.log((PlacementAndUpdate & Placement) === Update);
```

1.4 collectEffectList

- 为了避免遍历fiber树寻找有副作用的fiber节点，所以有了effectList
- 在 fiber树构建过程中，每当一个 fiber节点的 flags字段不为 NoFlags时(代表需要执行副作用),就把该 fiber节点添加到 effectList中
- effectList是一个单向链表， firstEffect代表链表中的第一个fiber节点， lastEffect代表链表中的最后一个fiber节点
- fiber树的构建是 深度优先的，也就是先向下构建子级Fiber节点，子级节点构建完成后，再向上构建父级Fiber节点，所以EffectList中总是子级Fiber节点在前面
- fiber节点构建完成的操作执行在 completeUnitOfWork方法，在这个方法里，不仅会对节点完成构建，也会将有flags的Fiber节点添加到fffectList

```
function collectEffectList(returnFiber, completedWork) {
    if (returnFiber) {
        if (!returnFiber.firstEffect) {
            returnFiber.firstEffect = completedWork.firstEffect;
        }
        if (completedWork.lastEffect) {
            if (returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = completedWork.firstEffect;
            }
            returnFiber.lastEffect = completedWork.lastEffect;
        }

        const flags = completedWork.flags;
        if (flags) {
            if (returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = completedWork;
            } else {
                returnFiber.firstEffect = completedWork;
            }
            returnFiber.lastEffect = completedWork;
        }
    }
}

let rootFiber = { key: 'root' };
let fiberA = { key: 'A', flags: 'Placement' };
let fiberB = { key: 'B', flags: 'Placement' };
let fiberC = { key: 'C', flags: 'Placement' };
collectEffectList(fiberA, fiberB);
collectEffectList(fiberA, fiberC);
collectEffectList(rootFiber, fiberA);
let effectLists = '';
let nextEffect = rootFiber.firstEffect;
while (nextEffect) {
    effectLists += `${nextEffect.key}=>`;
    nextEffect = nextEffect.nextEffect;
}
effectLists += 'null';
console.log(effectLists);
```

2.实现虚拟DOM

- 虚拟DOM就是一个描述真实DOM的纯JS对象
- babeljs (https://www.babeljs.cn/repl)

2.1 package.json

package.json

```
{
  "scripts": {
+   "start": "set DISABLE_NEW_JSX_TRANSFORM=true&&react-scripts start",
+   "build": "set DISABLE_NEW_JSX_TRANSFORM=true&&react-scripts build",
+   "test": "set DISABLE_NEW_JSX_TRANSFORM=true&&react-scripts test",
+   "eject": "set DISABLE_NEW_JSX_TRANSFORM=true&&react-scripts eject"
  }
}
```

2.2 src\index.js

src\index.js

```
import React from './react';

let element = <div key="title" id="title">divdiv>;
console.log(element);
```

2.3 ReactSymbols.js

src\ReactSymbols.js

```
export let REACT_ELEMENT_TYPE = Symbol.for('react.element');
```

2.4 react.js

src\react.js

```
import { REACT_ELEMENT_TYPE } from './ReactSymbols';
const RESERVED_PROPS = {
    key: true,
    ref: true,
    __self: true,
    __source: true

};

function createElement(type, config, children) {
    let propName;

    const props = {};
    let key = null;
    let ref = null;
    if (config) {
        if (config.ref) {
            ref = config.ref;
        }
        if (config.key) {
            key = '' + config.key;
        }

        for (propName in config) {
            if (!RESERVED_PROPS.hasOwnProperty(propName)) {
                props[propName] = config[propName];
            }
        }
    }
    const childrenLength = arguments.length - 2;
    if (childrenLength === 1) {
        props.children = children;
    } else if (childrenLength > 1) {
        const childArray = Array(childrenLength);
        for (let i = 0; i < childrenLength; i++) {
            childArray[i] = arguments[i + 2];
        }
        props.children = childArray;
    }
    return {

        $typeof: REACT_ELEMENT_TYPE,

        type,
        ref,
        key,
        props
    }
}
const React = {
    createElement
}
export default React;
```

3.开始WorkOnRoot

- 开始WorkOnRoot (https://www.processon.com/diagraming/618dec810e3e744ad43d37a9)

3.1 从render到执行工作循环

3.2 fiber结构

- fiber结构 (https://www.processon.com/diagraming/618d10607d9c08562ae923b9)

3.3 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
let element = <div key="title" id="title">divdiv>;
console.log(element);
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

3.4 react-dom.js

src\react-dom.js

```
import { createFiberRoot } from './ReactFiberRoot';
import { updateContainer } from './ReactFiberReconciler';
function render(element, container) {
    let fiberRoot = createFiberRoot(container);
    updateContainer(element, fiberRoot);
}
const ReactDOM = {
    render
}
export default ReactDOM;
```

3.5 ReactFiberRoot.js

src\ReactFiberRoot.js

```
import { createHostRootFiber } from './ReactFiber';
import { initializeUpdateQueue } from './ReactUpdateQueue';
export function createFiberRoot(containerInfo) {
    const root = new FiberRootNode(containerInfo);
    const hostRootFiber = createHostRootFiber();
    root.current = hostRootFiber;
    hostRootFiber.stateNode = root;
    initializeUpdateQueue(hostRootFiber);
    return root;
}

function FiberRootNode(containerInfo) {
    this.containerInfo = containerInfo;
}
```

3.6 ReactFiber.js

src\ReactFiber.js

```
import { HostRoot } from './ReactWorkTags';

export function createHostRootFiber() {
    return createFiber(HostRoot);
}

const createFiber = function (tag, pendingProps, key) {
    return new FiberNode(tag, pendingProps, key);
};

function FiberNode(tag, pendingProps, key) {
    this.tag = tag;
    this.pendingProps = pendingProps;
    this.key = key;
}
```

src\ReactWorkTags.js

```

export const HostRoot = 3;
```

3.8 ReactUpdateQueue.js

src\ReactUpdateQueue.js

```

export function initializeUpdateQueue(fiber) {
    const queue = {

        shared: {
            pending: null
        }
    };
    fiber.updateQueue = queue;
}

export function createUpdate() {
    return {};
}

export function enqueueUpdate(fiber, update) {

    const updateQueue = fiber.updateQueue;
    const sharedQueue = updateQueue.shared;

    const pending = sharedQueue.pending;

    if (!pending) {

        update.next = update;
    } else {

        update.next = pending.next;

        pending.next = update;
    }

    sharedQueue.pending = update;
}
```

3.9 ReactFiberReconciler.js

src\ReactFiberReconciler.js

```

import { createUpdate, enqueueUpdate } from './ReactUpdateQueue';
import { scheduleUpdateOnFiber } from './ReactFiberWorkLoop';

export function updateContainer(element, container) {

    const current = container.current;

    const update = createUpdate();

    update.payload = { element };

    enqueueUpdate(current, update);

    scheduleUpdateOnFiber(current);
}
```

3.10 ReactFiberWorkLoop.js

src\ReactFiberWorkLoop.js

```
import { HostRoot } from './ReactWorkTags';

function markUpdateLaneFromFiberToRoot(sourceFiber) {
    let node = sourceFiber;
    let parent = node.return;

    while (parent) {
        node = parent;
        parent = parent.return;
    }

    if (node.tag === HostRoot) {
        return node.stateNode;
    }
}

export function scheduleUpdateOnFiber(fiber) {

    const root = markUpdateLaneFromFiberToRoot(fiber);

    performSyncWorkOnRoot(root);
}

function performSyncWorkOnRoot(root) {

    console.log(root);
}
```

4.创建createWorkInProgress

- 创建createWorkInProgress (https://www.processon.com/diagraming/618e41131efad41bf2c534f6)

4.1 src\ReactFiberWorkLoop.js

src\ReactFiberWorkLoop.js

```
import { HostRoot } from './ReactWorkTags';
+import { createWorkInProgress } from './ReactFiber';
+//正在调度的fiberRoot根节点
+let workInProgressRoot = null;
+//正在处理的fiber节点
+let workInProgress = null;
/**
 * 向上获取HostRoot节点
 * @param {*} sourceFiber 更新来源fiber
 * @returns
 */
function markUpdateLaneFromFiberToRoot(sourceFiber) {
    let node = sourceFiber;
    let parent = node.return;
    //一直向上找父亲，找不到为止
    while (parent) {
        node = parent;
        parent = parent.return;
    }
    //如果找到的是HostRoot就返回FiberRootNode,其实就是容器div#root
    if (node.tag
        return node.stateNode;
    }
}
/**
 * 向上查找到根节点开始调度更新
 * @param {*} fiber
 */
export function scheduleUpdateOnFiber(fiber) {
    //向上获取HostRoot节点
    const root = markUpdateLaneFromFiberToRoot(fiber);
    //执行HostRoot上的更新
    performSyncWorkOnRoot(root);
}
/**
 * 开始执行FiberRootNode上的工作
 * @param {*} root  FiberRootNode
 */
function performSyncWorkOnRoot(root) {
+   //先赋值给当前正在执行工作的FiberRootNode根节点
+   workInProgressRoot = root;
+   //创建一个新的处理中的fiber节点
+   workInProgress = createWorkInProgress(workInProgressRoot.current);
+   console.log(workInProgress);
}
```

4.2 src\ReactFiber.js

src\ReactFiber.js

```
import { HostRoot } from './ReactWorkTags';
+import { NoFlags } from './ReactFiberFlags';
/**
 * 创建根fiber
 * @returns 根fiber
 */
export function createHostRootFiber() {
    return createFiber(HostRoot);
}
/**
 * 创建fiber
 * @param {*} tag fiber类型
 * @param {*} pendingProps 新属性对象
 * @param {*} key 唯一标识
 * @returns 创建的fiber
 */
const createFiber = function (tag, pendingProps, key) {
    return new FiberNode(tag, pendingProps, key);
};
/**
 * fiber构建函数
 * @param {*} tag fiber类型
 * @param {*} pendingProps  新属性对象
 * @param {*} key 唯一标识
 */
function FiberNode(tag, pendingProps, key) {
    this.tag = tag;
    this.pendingProps = pendingProps;
    this.key = key;
}
+/**
+ * 基于老的current创建新的workInProgress
+ * @param {*} current 老的fiber
+ * @returns
+ */
+export function createWorkInProgress(current, pendingProps) {
+    let workInProgress = current.alternate;
+    if (!workInProgress) {
+        workInProgress = createFiber(current.tag, pendingProps, current.key);
+        workInProgress.type = current.type;
+        workInProgress.stateNode = current.stateNode;
+        workInProgress.alternate = current;
+        current.alternate = workInProgress;
+    } else {
+        workInProgress.pendingProps = pendingProps;
+        workInProgress.flags = NoFlags;
+    }
+    //清空原来的child
+    workInProgress.child = null;
+    //清空原来的sibling
+    workInProgress.sibling = null;
+    //清空原来的副作用链
+    workInProgress.firstEffect = workInProgress.nextEffect = workInProgress.lastEffect = null;
+    workInProgress.updateQueue = current.updateQueue;
+    return workInProgress;
+}
```

4.3 src\ReactFiberFlags.js

src\ReactFiberFlags.js

```
export const NoFlags = 0b00000000000000000000;
```

5.初次渲染

- 5.初次渲染 (https://www.processon.com/diagraming/618fbad7e0b34d73f7f763d4)

5.1 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
let element = <div key="title" id="title" style={{ border: '1px solid red' }}>divdiv>;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

5.2 ReactFiberWorkLoop.js

src\ReactFiberWorkLoop.js

```
+import { HostRoot, HostComponent } from './ReactWorkTags';
import { createWorkInProgress } from './ReactFiber';
+import { beginWork } from './ReactFiberBeginWork';
+import { completeWork } from './ReactFiberCompleteWork';
+import { Placement } from './ReactFiberFlags';
+import { commitPlacement } from './ReactFiberCommitWork';
//正在调度的fiberRoot根节点
let workInProgressRoot = null;
//正在处理的fiber节点
let workInProgress = null;
/**
 * 向上获取HostRoot节点
 * @param {*} sourceFiber 更新来源fiber
 * @returns
 */
function markUpdateLaneFromFiberToRoot(sourceFiber) {
    let node = sourceFiber;
    let parent = node.return;
    //一直向上找父亲，找不到为止
```

```
    while (parent) {
        node = parent;
        parent = parent.return;
    }
    //如果找到的是HostRoot就返回FiberRootNode,其实就是容器div#root
    if (node.tag
        return node.stateNode;
    }
}
/**
 * 向上查找到根节点开始调度更新
 * @param {*} fiber
 */
export function scheduleUpdateOnFiber(fiber) {
    //向上获取HostRoot节点
    const root = markUpdateLaneFromFiberToRoot(fiber);
    //执行HostRoot上的更新
    performSyncWorkOnRoot(root);
}
/**
 * 开始执行FiberRootNode上的工作
 * @param {*} root  FiberRootNode
 */
function performSyncWorkOnRoot(root) {
    //先赋值给当前正在执行工作的FiberRootNode根节点
    workInProgressRoot = root;
    //创建一个新的处理中的fiber节点
    workInProgress = createWorkInProgress(workInProgressRoot.current);
+   workLoopSync();
+   commitRoot();
}
+function commitRoot(root) {
+    //构建成功的新的fiber树
+    const finishedWork = workInProgressRoot.current.alternate;
+    //当前完成的构建工作等于finishedWork
+    workInProgressRoot.finishedWork = finishedWork;
+    commitMutationEffects(workInProgressRoot);
+}
+function getFlag(flags) {
+    switch (flags) {
+        case Placement:
+            return '添加';
+        default:
+            break;
+    }
+}
+function commitMutationEffects(root) {
+    const finishedWork = root.finishedWork;
+    let nextEffect = finishedWork.firstEffect;
+    let effectList = '';
+    while (nextEffect) {
+        effectList += `(${getFlag(nextEffect.flags)}#${nextEffect.type}#${nextEffect.+key})=>`;
+        const flags = nextEffect.flags;
+        if (flags === Placement) {
+            commitPlacement(nextEffect);
+        }
+        nextEffect = nextEffect.nextEffect;
+    }
+    effectList += 'null';
+    console.log(effectList);
+    root.current = finishedWork;
+}
+function workLoopSync() {
+    while (workInProgress) {
+        performUnitOfWork(workInProgress);
+    }
+}
+/**
+ * 执行单个工作单元
+ * @param {*} unitOfWork 单个fiber
+ */
+function performUnitOfWork(unitOfWork) {
+    //获取当前fiber的alternate
+    const current = unitOfWork.alternate;
+    //开始构建此fiber的子fiter链表
+    let next = beginWork(current, unitOfWork);
+    //更新属性
+    unitOfWork.memoizedProps = unitOfWork.pendingProps;
+    //如果有子fiber，就继续执行
+    if (next) {
+        workInProgress = next;
+    } else {
+        //如果没有子fiber，就完成当前的fiber
+        completeUnitOfWork(unitOfWork);
+    }
+}
+
+function completeUnitOfWork(unitOfWork) {
+    //尝试完成当前的工作单元，然后移动到下一个弟弟
+    //如果没有下一个弟弟，返回到父Fiber
+    let completedWork = unitOfWork;
+    do {
+        const current = completedWork.alternate;
+        const returnFiber = completedWork.return;
+        completeWork(current, completedWork);
+        collectEffectList(returnFiber, completedWork)
+        const siblingFiber = completedWork.sibling;
+        if (siblingFiber) {
+            //如果此 returnFiber 中还有更多工作要做，请执行下一步
+            workInProgress = siblingFiber;
+            return;
+        }
+        //否则，返回父级
+        completedWork = returnFiber;
```

```
+        //更新我们正在处理的下一件事
+        workInProgress = completedWork;
+    } while (completedWork);
+}
+function collectEffectList(returnFiber, completedWork) {
+    if (returnFiber) {
+        if (!returnFiber.firstEffect) {
+            returnFiber.firstEffect = completedWork.firstEffect;
+        }
+        if (completedWork.lastEffect) {
+            if (returnFiber.lastEffect) {
+                returnFiber.lastEffect.nextEffect = completedWork.firstEffect;
+            }
+            returnFiber.lastEffect = completedWork.lastEffect;
+        }
+        //如果这个fiber有副作用，我们会在孩子们的副使用后面添加它自己的副作用
+        const flags = completedWork.flags;
+        if (flags) {
+            if (returnFiber.lastEffect) {
+                returnFiber.lastEffect.nextEffect = completedWork;
+            } else {
+                returnFiber.firstEffect = completedWork;
+            }
+            returnFiber.lastEffect = completedWork;
+        }
+    }
+}
```

src\ReactWorkTags.js

```
//根fiber,对应的其实是容器containerInfo
export const HostRoot = 3;
+export const HostComponent = 5;
```

5.4 ReactFiberBeginWork.js

src\ReactFiberBeginWork.js

```
import { shouldSetTextContent } from './ReactDOMHostConfig';
import { reconcileChildFibers, mountChildFibers } from './ReactChildFiber';
import { HostRoot, HostComponent } from './ReactWorkTags';

export function beginWork(current, workInProgress) {
    switch (workInProgress.tag) {
        case HostRoot:
            return updateHostRoot(current, workInProgress);
        case HostComponent:
            return updateHostComponent(current, workInProgress);
        default:
            break;
    }
}

function updateHostRoot(current, workInProgress) {
    const updateQueue = workInProgress.updateQueue;
    const nextChildren = updateQueue.shared.pending.payload.element;
    reconcileChildren(current, workInProgress, nextChildren);
    updateQueue.shared.pending = null;
    return workInProgress.child;
}

function updateHostComponent(current, workInProgress) {

    const type = workInProgress.type;

    const nextProps = workInProgress.pendingProps;

    let nextChildren = nextProps.children;

    const isDirectTextChild = shouldSetTextContent(type, nextProps);
    if (isDirectTextChild) {

        nextChildren = null;
    }
    reconcileChildren(current, workInProgress, nextChildren);
    return workInProgress.child;
}
export function reconcileChildren(current, workInProgress, nextChildren) {

    if (current) {
        workInProgress.child = reconcileChildFibers(
            workInProgress,
            current && current.child,
            nextChildren
        );

    } else {
        workInProgress.child = mountChildFibers(
            workInProgress,
            current && current.child,
            nextChildren
        );
    }

}
```

5.5 ReactFiberCompleteWork.js

src\ReactFiberCompleteWork.js

```
import { createInstance, finalizeInitialChildren } from './ReactDOMHostConfig';
import { HostComponent } from './ReactWorkTags';
export function completeWork(current, workInProgress) {
    const newProps = workInProgress.pendingProps;
    switch (workInProgress.tag) {
        case HostComponent: {
            if (current && workInProgress.stateNode) {

            } else {
                const type = workInProgress.type;
                const instance = createInstance(type, newProps);
                workInProgress.stateNode = instance;
                finalizeInitialChildren(instance, type, newProps);
            }
            break;
        }
        default:
            break;
    }
}
```

5.6 ReactFiberFlags.js

src\ReactFiberFlags.js

```
export const NoFlags = 0b00000000000000000000;
+export const Placement = 0b00000000000000000010;
```

5.7 ReactDOMHostConfig.js

src\ReactDOMHostConfig.js

```
import { createElement, setInitialProperties } from './ReactDOMComponent'
export function shouldSetTextContent(type, props) {
    return (
        typeof props.children === 'string' ||
        typeof props.children === 'number'
    );
}
export function createInstance(type) {
    return createElement(type);
}
export function finalizeInitialChildren(domElement, type, props) {
    setInitialProperties(domElement, type, props);
}
export function appendChild(parentInstance, child) {
    parentInstance.appendChild(child);
}
export function insertBefore(parentInstance, child, beforeChild) {
    parentInstance.insertBefore(child, beforeChild);
}
```

5.8 src\ReactDOMComponent.js

src\ReactDOMComponent.js

```
export function createElement(type) {
    return document.createElement(type);
}
export function setInitialProperties(domElement, tag, rawProps) {
    for (const propKey in rawProps) {
        const nextProp = rawProps[propKey];
        if (propKey === children) {
            if (typeof nextProp === 'string' || typeof nextProp === 'number') {
                domElement.textContent = nextProp;
            }
        } else if (propKey === 'style') {
            let styleObj = rawProps[propKey];
            for (let styleProp in styleObj) {
                domElement.style[styleProp] = styleObj[styleProp];
            }
        } else {
            domElement[propKey] = nextProp;
        }
    }
}
```

5.9 src\ReactChildFiber.js

src\ReactChildFiber.js

```
import { Placement } from './ReactFiberFlags';
import { createFiberFromElement } from './ReactFiber';
import { REACT_ELEMENT_TYPE } from './ReactSymbols';
function ChildReconciler(shouldTrackSideEffects) {
    function placeSingleChild(newFiber) {

        if (shouldTrackSideEffects && !newFiber.alternate) {
            newFiber.flags = Placement;
        }
        return newFiber;
    }
    function reconcileSingleElement(returnFiber, currentFirstChild, element) {

        const created = createFiberFromElement(element);

        created.return = returnFiber;
        return created;
    }
    function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
        const isObject = typeof newChild === 'object' && (newChild);
        if (isObject) {
            switch (newChild.$typeof) {
                case REACT_ELEMENT_TYPE:
                    return placeSingleChild(
                        reconcileSingleElement(returnFiber, currentFirstChild, newChild)
                    );
                default:
                    break;
            }
        }
    }
    return reconcileChildFibers;
}

export const reconcileChildFibers = ChildReconciler(true);
export const mountChildFibers = ChildReconciler(false);
```

5.10 ReactFiber.js

src\ReactFiber.js

```
+import { HostRoot, HostComponent } from './ReactWorkTags';
import { NoFlags } from './ReactFiberFlags';

/**
 * 创建根fiber
 * @returns 根fiber
 */
export function createHostRootFiber() {
    return createFiber(HostRoot);
}
/**
 * 创建fiber
 * @param {*} tag fiber类型
 * @param {*} pendingProps 新属性对象
 * @param {*} key 唯一标识
 * @returns 创建的fiber
 */
const createFiber = function (tag, pendingProps, key) {
    return new FiberNode(tag, pendingProps, key);
};
/**
 * fiber构建函数
 * @param {*} tag fiber类型
 * @param {*} pendingProps  新属性对象
 * @param {*} key 唯一标识
 */
function FiberNode(tag, pendingProps, key) {
    this.tag = tag;
    this.pendingProps = pendingProps;
    this.key = key;
}
/**
 * 基于老的current创建新的workInProgress
 * @param {*} current 老的fiber
 * @returns
 */
export function createWorkInProgress(current, pendingProps) {
    let workInProgress = current.alternate;
    if (!workInProgress) {
        workInProgress = createFiber(current.tag, pendingProps, current.key);
        workInProgress.type = current.type;
        workInProgress.stateNode = current.stateNode;
        workInProgress.alternate = current;
        current.alternate = workInProgress;
    } else {
        workInProgress.pendingProps = pendingProps;
        workInProgress.flags = NoFlags;
    }
    //清空原来的child
    workInProgress.child = null;
    //清空原来的sibling
    workInProgress.sibling = null;
    //清空原来的副作用链
    workInProgress.firstEffect = workInProgress.nextEffect = workInProgress.lastEffect = null;
    workInProgress.updateQueue = current.updateQueue;
    return workInProgress;
}
+export function createFiberFromElement(element) {
+    const { key, type, props } = element;
+    let fiberTag;
+    if (typeof type === 'string') {
+        fiberTag = HostComponent;
+    }
+    const fiber = createFiber(fiberTag, props, key);
+    fiber.type = type;
+    return fiber;
+}
```

5.11 ReactFiberCommitWork.js

src\ReactFiberCommitWork.js

```
import { HostComponent, HostRoot } from './ReactWorkTags';
import { appendChild, insertBefore } from './ReactDOMHostConfig';
import { Placement } from './ReactFiberFlags';
function getParentStateNode(fiber) {
    const parent = fiber.return;
    do {
        if (parent.tag === HostComponent) {
            return parent.stateNode;
        } else if (parent.tag === HostRoot) {
            return parent.stateNode.containerInfo
        }
        parent = parent.return;
    } while (parent);
}
export function commitPlacement(finishedWork) {
    let stateNode = finishedWork.stateNode;
    let parentStateNode = getParentStateNode(finishedWork);
     appendChild(parentStateNode, stateNode);
}
```

6.单节点key相同,类型相同

- 单节点key相同,类型相同的时候,复用老节点，只更新属性
- fiber结构 (https://www.processon.com/diagraming/618fbda40e3e744ad4402b1c)
- 单节点DIFF流程 (https://www.processon.com/diagraming/6185f0781efad40ab186a60a)

6.1 .eslintrc

.eslintrc

```
{
  "rules": {
    "react-hooks/rules-of-hooks": "off"
  }
}
```

6.2 public\index.html

public\index.html

```
   React App

     1.key相同,类型相同
     <div key="title" id="title">
         div
     </div>
     复用老节点，只更新属性
     <div key="title" id="title2">
         div2
     </div>
```

6.3 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
//1. key相同,类型相同,复用老节点，只更新属性
single1.addEventListener('click', () => {
  let element = (
    title
  );
  ReactDOM.render(element, root);
});
single1Update.addEventListener('click', () => {
  let element = (
    title2
  );
  ReactDOM.render(element, root);
});
```

6.4 src\react-dom.js

src\react-dom.js

```
import { createFiberRoot } from './ReactFiberRoot';
import { updateContainer } from './ReactFiberReconciler';
function render(element, container) {
+   let fiberRoot = container._reactRootContainer;
+   if (!fiberRoot) {
+       fiberRoot = container._reactRootContainer = createFiberRoot(container);
+   }
    updateContainer(element, fiberRoot);
}
const ReactDOM = {
    render
}
export default ReactDOM;
```

6.5 ReactFiberWorkLoop.js

src\ReactFiberWorkLoop.js

```
import { HostRoot, HostComponent } from './ReactWorkTags';
import { createWorkInProgress } from './ReactFiber';
import { beginWork } from './ReactFiberBeginWork';
import { completeWork } from './ReactFiberCompleteWork';
+import { Placement, Update, Deletion } from './ReactFiberFlags';
+import { commitPlacement,commitWork,commitDeletion } from './ReactFiberCommitWork';
//正在调度的fiberRoot根节点
let workInProgressRoot = null;
//正在处理的fiber节点
let workInProgress = null;
+//最大更新深度
+const NESTED_UPDATE_LIMIT = 50;
+let nestedUpdateCount = 0;
/**
 * 向上获取HostRoot节点
 * @param {*} sourceFiber 更新来源fiber
 * @returns
 */
function markUpdateLaneFromFiberToRoot(sourceFiber) {
    let node = sourceFiber;
    let parent = node.return;
    //一直向上找父亲，找不到为止
    while (parent) {
        node = parent;
        parent = parent.return;
    }
    //如果找到的是HostRoot就返回FiberRootNode,其实就是容器div#root
    if (node.tag
        return node.stateNode;
    }
}
/**
 * 向上查找到根节点开始调度更新
 * @param {*} fiber
 */
export function scheduleUpdateOnFiber(fiber) {
    checkForNestedUpdates();
    //向上获取HostRoot节点
```

```
    const root = markUpdateLaneFromFiberToRoot(fiber);
    //执行HostRoot上的更新
    performSyncWorkOnRoot(root);
}
+function checkForNestedUpdates() {
+    if (++nestedUpdateCount > NESTED_UPDATE_LIMIT) {
+        throw new Error('Maximum update depth exceeded');
+    }
+}
/**
 * 开始执行FiberRootNode上的工作
 * @param {*} root  FiberRootNode
 */
function performSyncWorkOnRoot(root) {
    //先赋值给当前正在执行工作的FiberRootNode根节点
    workInProgressRoot = root;
    //创建一个新的处理中的fiber节点
    workInProgress = createWorkInProgress(workInProgressRoot.current);
    workLoopSync();
    commitRoot();
}
function commitRoot(root) {
    //构建成功的新的fiber树
    const finishedWork = workInProgressRoot.current.alternate;
    //当前完成的构建工作等于finishedWork
    workInProgressRoot.finishedWork = finishedWork;
    commitMutationEffects(workInProgressRoot);
+   nestedUpdateCount--;
}
function getFlag(flags) {
    switch (flags) {
        case Placement:
            return '添加';
+       case Update:
+           return '更新';
+       case Deletion:
+           return '删除';
    }
}
function commitMutationEffects(root) {
    const finishedWork = root.finishedWork;
    let nextEffect = finishedWork.firstEffect;
    let effectList = '';
    while (nextEffect) {
        effectList += `(${getFlag(nextEffect.flags)}#${nextEffect.type}#${nextEffect.key})=>`;
        const flags = nextEffect.flags;
+       const current = nextEffect.alternate;
        if (flags
            commitPlacement(nextEffect);
+       } else if (flags === Update) {
+           commitWork(current, nextEffect);
+       } else if (flags === Deletion) {
+           commitDeletion(nextEffect);
+       }
        nextEffect = nextEffect.nextEffect;
    }
    effectList += 'null';
    console.log(effectList);
    root.current = finishedWork;
}

function commitPlacement(nextEffect) {
    let stateNode = fiber.stateNode;
    let parentStateNode = getParentStateNode(nextEffect);
    parentStateNode.appendChild(stateNode);
}
function getParentStateNode(fiber) {
    const parent = fiber.return;
    do {
        if (parent.tag
            return parent.stateNode;
        } else if (parent.tag
            return parent.stateNode.containerInfo
        }
        parent = parent.return;
    } while (parent);
}
function workLoopSync() {
    while (workInProgress) {
        performUnitOfWork(workInProgress);
    }
}
/**
 * 执行单个工作单元
 * @param {*} unitOfWork 单个fiber
 */
function performUnitOfWork(unitOfWork) {
    //获取当前fiber的alternate
    const current = unitOfWork.alternate;
    //开始构建此fiber的子fiter链表
    let next = beginWork(current, unitOfWork);
    //更新属性
    unitOfWork.memoizedProps = unitOfWork.pendingProps;
    //如果有子fiber，就继续执行
    if (next) {
        workInProgress = next;
    } else {
        //如果没有子fiber，就完成当前的fiber
        completeUnitOfWork(unitOfWork);
    }
}

function completeUnitOfWork(unitOfWork) {
    //尝试完成当前的工作单元，然后移动到下一个弟弟
```

```
    //如果没有下一个弟弟，返回到父Fiber
    let completedWork = unitOfWork;
    do {
        const current = completedWork.alternate;
        const returnFiber = completedWork.return;
        completeWork(current, completedWork);
        collectEffectList(returnFiber, completedWork)
        const siblingFiber = completedWork.sibling;
        if (siblingFiber) {
            //如果此 returnFiber 中还有更多工作要做，请执行下一步
            workInProgress = siblingFiber;
            return;
        }
        //否则，返回父级
        completedWork = returnFiber;
        //更新我们正在处理的下一件事
        workInProgress = completedWork;
    } while (completedWork);
}
function collectEffectList(returnFiber, completedWork) {
    if (returnFiber) {
        if (!returnFiber.firstEffect) {
            returnFiber.firstEffect = completedWork.firstEffect;
        }
        if (completedWork.lastEffect) {
            if (returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = completedWork.firstEffect;
            }
            returnFiber.lastEffect = completedWork.lastEffect;
        }
        //如果这个fiber有副作用，我们会在孩子们的副使用后面添加它自己的副作用
        const flags = completedWork.flags;
        if (flags) {
            if (returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = completedWork;
            } else {
                returnFiber.firstEffect = completedWork;
            }
            returnFiber.lastEffect = completedWork;
        }
    }
}
```

6.6 ReactFiberFlags.js

src\ReactFiberFlags.js

```
export const NoFlags = 0b00000000000000000000;//0
export const Placement = 0b00000000000000000010;//2
+export const Update = 0b00000000000000000100;//4
+export const Deletion = 0b00000000000000001000;//8
```

6.7 src\ReactFiberCommitWork.js

src\ReactFiberCommitWork.js

```
import { updateProperties } from './ReactDOMComponent';
import { HostComponent } from './ReactWorkTags';
import {appendChild,removeChild} from './ReactFiberHostConfig';
+/**
+ * 更新DOM节点的属性
+ * @param {*} current
+ * @param {*} finishedWork
+ */
+export function commitWork(current, finishedWork) {
+    switch (finishedWork.tag) {
+        case HostComponent: {
+            const updatePayload = finishedWork.updateQueue;
+            finishedWork.updateQueue = null;
+            if (updatePayload) {
+                updateProperties(finishedWork.stateNode, updatePayload);
+            }
+        }
+        default:
+            break;
+    }
+}
+function commitDeletion(fiber) {
+    if (!fiber) {
+        return;
+    }
+    let parentStateNode = getParentStateNode(fiber);
+    removeChild(parentStateNode, fiber.stateNode);
+}
function getParentStateNode(fiber) {
    const parent = fiber.return;
    do {
        if (parent.tag
            return parent.stateNode;
        } else if (parent.tag
            return parent.stateNode.containerInfo
        }
        parent = parent.return;
    } while (parent);
}
export function commitPlacement(finishedWork) {
    let stateNode = finishedWork.stateNode;
    let parentStateNode = getParentStateNode(finishedWork);
     appendChild(parentStateNode, stateNode);
}
```

6.8 src\ReactDOMComponent.js

src\ReactDOMComponent.js

```
export function createElement(type) {
    return document.createElement(type);
}
/**
 * 设置初始化属性
 * @param {*} domElement DOM节点
 * @param {*} tag 标签
 * @param {*} rawProps 属性对象
 */
export function setInitialProperties(domElement, tag, rawProps) {
    for (const propKey in rawProps) {
        const nextProp = rawProps[propKey];
        if (propKey
            if (typeof nextProp
                domElement.textContent = nextProp;
            }
        } else if (propKey
            let styleObj = rawProps[propKey];
            for (let styleProp in styleObj) {
                domElement.style[styleProp] = styleObj[styleProp];
            }
        } else {
            domElement[propKey] = nextProp;
        }
    }
}
+/**
+ * 比较属性的差异
+ * @param {*} domElement DOM节点
+ * @param {*} tag 标签
+ * @param {*} lastProps 老属性
+ * @param {*} nextProps 新属性
+ * @returns
+ */
+export function diffProperties(domElement, tag, lastProps = {}, nextProps = {}) {
+    let updatePayload = null;
+    let propKey;
+    for (propKey in lastProps) {
+        //老有新没有
+        if (lastProps.hasOwnProperty(propKey) && (!nextProps.hasOwnProperty(propKey))) {
+            // 对于所有其他已删除的属性，我们将其添加到队列中。
+            // 我们用改为提交阶段中允许的属性列表
+            (updatePayload = updatePayload || []).push(propKey, null);
+        }
+    }
+    for (propKey in nextProps) {
+        const nextProp = nextProps[propKey];
+        if (propKey === "children") {
+            if (typeof nextProp === 'string' || typeof nextProp === 'number') {
+                if (nextProp !== lastProps[propKey]) {
+                    (updatePayload = updatePayload || []).push(propKey, '' + nextProp);
+                }
+            }
+        } else {
+            let oldProp = lastProps[propKey];
+            if (oldProp !== nextProp) {
+                (updatePayload = updatePayload || []).push(propKey, nextProp);
+            }
+        }
+    }
+    return updatePayload;
+}
+/**
+ * 更新DOM节点的属性
+ * @param {*} domElement DOM节点
+ * @param {*} updatePayload 要更新的属性
+ */
+export function updateProperties(domElement, updatePayload) {
+    for (let i = 0; i < updatePayload.length; i += 2) {
+        const propKey = updatePayload[i];
+        const propValue = updatePayload[i + 1];
+        if (propKey === "children") {
+            domElement.textContent = propValue;
+        } else {
+            domElement.setAttribute(propKey, propValue);
+        }
+    }
+
+}
```

6.9 src\ReactChildFiber.js

src\ReactChildFiber.js

```
+import { Placement, Deletion } from './ReactFiberFlags';
+import { createFiberFromElement, createWorkInProgress } from './ReactFiber';
import { REACT_ELEMENT_TYPE } from './ReactSymbols';
function ChildReconciler(shouldTrackSideEffects) {
    function placeSingleChild(newFiber) {
        //如果要跟踪副作用并且没有老的fiber的话，就把此fiber标记为新建
        if (shouldTrackSideEffects && !newFiber.alternate) {
            newFiber.flags = Placement;
        }
        return newFiber;
    }
+   function deleteChild(returnFiber, childToDelete) {
+       if (!shouldTrackSideEffects) {
+           return;
+       }
+       //把此fiber添加到父亲待删除的副作用链表中
+       const last = returnFiber.lastEffect;
+       if (last) {//如果父亲有尾部，就把此fiber添加到尾部的nextEffect上
+           last.nextEffect = childToDelete;
+           //然后让父亲的尾部等于自己
+           returnFiber.lastEffect = childToDelete;
+       } else {
+           //如果父亲的副作用链表为空，头和尾向childToDelete
+           returnFiber.firstEffect = returnFiber.lastEffect = childToDelete;
+       }
+       //清空它的nextEffect
+       childToDelete.nextEffect = null;
+       //把此fiber标记为删除
+       childToDelete.flags = Deletion;
+   }
+   function deleteRemainingChildren(returnFiber, currentFirstChild) {
+       let childToDelete = currentFirstChild;
+       while (childToDelete) {
+           deleteChild(returnFiber, childToDelete);
+           childToDelete = childToDelete.sibling;
+       }
+       return null;
+   }
+   /**
+ * 复用老的fiber
+ * @param {*} fiber 老fiber
+ * @param {*} pendingProps 新属性
+ * @returns
+ */
+   function useFiber(fiber, pendingProps) {
+       //根据老的fiber创建新的fiber
+       return createWorkInProgress(fiber, pendingProps);
+   }
+   function reconcileSingleElement(returnFiber, currentFirstChild, element) {
+       //新jsx元素的key
+       const key = element.key;
+       //第一个老的fiber节点
+       let child = currentFirstChild;
+       //判断是否有老的fiber节点
+       while (child) {
+           //先判断两者的key是相同
+           if (child.key === key) {
+               //再判断类型是否相同,如果相同
+               if (child.type === element.type) {
+                   //删除剩下的所有的老fiber节点
+                   deleteRemainingChildren(returnFiber, child.sibling);
+                   //复用这个老的fiber节点,并传递新的属性
+                   const existing = useFiber(child, element.props);
+                   //让新的fiber的return指向当前的父fiber
+                   existing.return = returnFiber;
+                   //返回复用的fiber
+                   return existing;
+               }else{
+                   //如果key相同但类型不同，不再进行后续匹配，老fiber全部全部删除
+                   deleteRemainingChildren(returnFiber, child);
+                   break;
+               }
+           } else {
+               //如果key不一样，则把当前的老fiber标记为删除，继续匹配弟弟
+               deleteChild(returnFiber, child);
+           }
+           //找弟弟继续匹配
+           child = child.sibling;
+       }
        //根据虚拟DOM创建fiber节点
        const created = createFiberFromElement(element);
        //让新的fiber的return指向当前的父fiber
        created.return = returnFiber;
        return created;
    }
    function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
        const isObject = typeof newChild
        if (isObject) {
            switch (newChild.$typeof) {
                case REACT_ELEMENT_TYPE:
                    return placeSingleChild(
                        reconcileSingleElement(returnFiber, currentFirstChild, newChild)
                    );
                default:
                    break;
            }
        }
    }
    return reconcileChildFibers;
}

export const reconcileChildFibers = ChildReconciler(true);
export const mountChildFibers = ChildReconciler(false);
```

6.10 ReactDOMHostConfig.js

src\ReactDOMHostConfig.js

```
+import { createElement, setInitialProperties, diffProperties } from './ReactDOMComponent'
export function shouldSetTextContent(type, props) {
    return (
        typeof props.children
        typeof props.children
    );
}
export function createInstance(type) {
    return createElement(type);
}
export function finalizeInitialChildren(domElement, type, props) {
    setInitialProperties(domElement, type, props);
}
+export function prepareUpdate(domElement, type, oldProps, newProps) {
+    return diffProperties(
+        domElement,
+        type,
+        oldProps,
+        newProps
+    );
+}
+export function removeChild(parentInstance, child) {
+  parentInstance.removeChild(child);
+}
```

6.11 ReactFiberCompleteWork.js

src\ReactFiberCompleteWork.js

```
+import { createInstance, finalizeInitialChildren, prepareUpdate } from './ReactDOMHostConfig';
import { HostComponent } from './ReactWorkTags';
+import { Update } from './ReactFiberFlags';
export function completeWork(current, workInProgress) {
    const newProps = workInProgress.pendingProps;
    switch (workInProgress.tag) {
        case HostComponent: {
            if (current && workInProgress.stateNode) {
+               updateHostComponent(current, workInProgress, workInProgress.tag, newProps);
            } else {
                const type = workInProgress.type;
                const instance = createInstance(type, newProps);
                workInProgress.stateNode = instance;
                finalizeInitialChildren(instance, type, newProps);
            }
            break;
        }
        default:
            break;
    }
}
+/**
+ * 更新原生DOM组件
+ * @param {*} current 老fiber
+ * @param {*} workInProgress 新fiber
+ * @param {*} type 类型
+ * @param {*} newProps  新属性
+ */
+function updateHostComponent(current, workInProgress, type, newProps) {
+    //如果我们有一个替代方案，这意味着这是一个更新，我们需要安排一个副作用来进行更新
+    const oldProps = current.memoizedProps;
+    const instance = workInProgress.stateNode;
+    //如果我们因为我们的一个孩子更新而得到更新，我们不会有 newProps 所以我们必须重用它们。
+    const updatePayload = prepareUpdate(instance, type, oldProps, newProps);
+    workInProgress.updateQueue = updatePayload;
+    if (updatePayload) {
+        markUpdate(workInProgress);
+    }
+}
+function markUpdate(workInProgress) {
+    //用更新效果标记fiber。 这会将 Placement 转换为 PlacementAndUpdate。
+    workInProgress.flags |= Update;
+}
```

7.单节点key相同,类型不同

- 单节点key相同,类型不同，删除老节点，添加新节点

7.1 public\index.html

public\index.html

```
  React App

   2.key相同,类型不同
   <div key="title" id="title">
        div
   </div>
   删除老节点，添加新节点
   <p key="title" id="title">
       p
   </p>
```

7.2 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
//2.key相同,类型不同，删除老节点，添加新节点
single2.addEventListener('click', () => {
  let element = (
    title
  );
  ReactDOM.render(element, root);
});
single2Update.addEventListener('click', () => {
  let element = (
    title
  );
  ReactDOM.render(element, root);
});
```

8.单节点类型相同,key不同

- 3.类型相同,key不同,删除老节点，添加新节点

8.1 public\index.html

```
  React App

   3.类型相同,key不同
   <div key="title1" id="title">
      title
   </div>
   删除老节点，添加新节点
   <div key="title2" id="title">
      title
   </div>
```

8.2 src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
//3.类型相同,key不同,删除老节点，添加新节点
single3.addEventListener('click', () => {
  let element = (
    title
  );
  ReactDOM.render(element, root);
});
single3Update.addEventListener('click', () => {
  let element = (
    title
  );
  ReactDOM.render(element, root);
});
```

9.原来多个节点，现在只有一个节点

- 原来多个节点，现在只有一个节点,删除多余节点
- fiber结构 (https://www.processon.com/diagraming/618fe7621e0853689b0d6159)

9.1 public\index.html

```
  React App

   4.原来多个节点，现在只有一个节点
   <ul key="ul">
       <li key="A">A</li>
       <li key="B" id="B">B</li>
       <li key="C">C</li>
    </ul>
   保留并更新这一个节点，删除其它节点
   <ul key="ul">
       <li key="B" id="B2">B2</li>
   </ul>
```

9.2 src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
//4.原来多个节点，现在只有一个节点,删除多余节点
single4.addEventListener('click', () => {
  let element = (

     A
     B
     C

  );
  ReactDOM.render(element, root);
});
single4Update.addEventListener('click', () => {
  let element = (

     B2

  );
  ReactDOM.render(element, root);
});
```

9.3 ReactChildFiber.js

src\ReactChildFiber.js

```
import { Placement, Deletion } from './ReactFiberFlags';
import { createFiberFromElement, createWorkInProgress } from './ReactFiber';
import { REACT_ELEMENT_TYPE } from './ReactSymbols';
function ChildReconciler(shouldTrackSideEffects) {
    function placeSingleChild(newFiber) {
        //如果要跟踪副作用并且没有老的fiber的话，就把此fiber标记为新建
        if (shouldTrackSideEffects && !newFiber.alternate) {
            newFiber.flags = Placement;
        }
        return newFiber;
    }
    function deleteChild(returnFiber, childToDelete) {
        if (!shouldTrackSideEffects) {
            return;
        }
        //把此fiber添加到父亲待删除的副作用链表中
        const last = returnFiber.lastEffect;
        if (last) {//如果父亲有尾部，就把此fiber添加到尾部的nextEffect上
            last.nextEffect = childToDelete;
            //然后让父亲的尾部等于自己
            returnFiber.lastEffect = childToDelete;
        } else {
            //如果父亲的副作用链表为空，头和尾向childToDelete
            returnFiber.firstEffect = returnFiber.lastEffect = childToDelete;
        }
        //清空它的nextEffect
        childToDelete.nextEffect = null;
        //把此fiber标记为删除
        childToDelete.flags = Deletion;
    }
    function deleteRemainingChildren(returnFiber, currentFirstChild) {
        let childToDelete = currentFirstChild;
        while (childToDelete) {
            deleteChild(returnFiber, childToDelete);
            childToDelete = childToDelete.sibling;
        }
        return null;
    }
    /**
 * 复用老的fiber
 * @param {*} fiber 老fiber
 * @param {*} pendingProps 新属性
 * @returns
 */
    function useFiber(fiber, pendingProps) {
        //根据老的fiber创建新的fiber
        return createWorkInProgress(fiber, pendingProps);
    }
    /**
     * 单节点DIFF
     * @param {*} returnFiber 当前父亲fiber
     * @param {*} currentFirstChild 当前第一个子节fiber
     * @param {*} element JSX虚拟DOM
     * @returns
     */
    function reconcileSingleElement(returnFiber, currentFirstChild, element) {
        //新jsx元素的key
        const key = element.key;
        //第一个老的fiber节点
        let child = currentFirstChild;
        //判断是否有老的fiber节点
        while (child) {
            //先判断两者的key是相同
            if (child.key
                //再判断类型是否相同,如果相同
                if (child.type
                    //删除剩下的所有的老fiber节点
                    deleteRemainingChildren(returnFiber, child.sibling);
                    //复用这个老的fiber节点,并传递新的属性
                    const existing = useFiber(child, element.props);
                    //让新的fiber的return指向当前的父fiber
                    existing.return = returnFiber;
                    //返回复用的fiber
                    return existing;
                } else {
                    //如果key相同但类型不同，不再进行后续匹配，后面的全部删除
                    deleteRemainingChildren(returnFiber, child);
                    break;
                }
            } else {
                //如果key不一样，则把当前的老fiber标记为删除，继续匹配弟弟
                deleteChild(returnFiber, child);
            }
            //找弟弟继续匹配
            child = child.sibling;
        }
        //根据虚拟DOM创建fiber节点
        const created = createFiberFromElement(element);
        //让新的fiber的return指向当前的父fiber
        created.return = returnFiber;
        return created;
    }
+   function createChild(returnFiber, newChild) {
+       if (typeof newChild === 'object' && newChild) {
+           switch (newChild.$typeof) {
+               case REACT_ELEMENT_TYPE: {
+                   const created = createFiberFromElement(newChild);
+                   created.return = returnFiber;
+                   return created;
+               }
+               default:
+                   break;
```

```
+           }
+       }
+   }
+   /**
+    * 协调子节点
+    * @param {*} returnFiber 父fiber
+    * @param {*} currentFirstChild 当前的第一个子fiber
+    * @param {*} newChildren JSX对象
+    * @returns
+    */
+   function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
+       //该算法无法通过两端搜索进行优化，因为fiber上没有反向指针
+       //我在试着看看我们能用这个模型走多远。如果它最终不值得，我们可以稍后添加它
+       //返回的第一个fiber子节点
+       let resultingFirstChild = null;
+       //上一个新的fiber节点
+       let previousNewFiber = null;
+       //比较中的旧的fiber节点
+       let oldFiber = currentFirstChild;
+       //新的索引
+       let newIdx = 0;
+       //如果老fiber完成，新的JSX没有完成
+       if (!oldFiber) {
+           for (; newIdx < newChildren.length; newIdx++) {
+               const newFiber = createChild(returnFiber, newChildren[newIdx]);
+               if (!previousNewFiber) {
+                   resultingFirstChild = newFiber;
+               } else {
+                   previousNewFiber.sibling = newFiber;
+               }
+               previousNewFiber = newFiber;
+           }
+           return resultingFirstChild;
+       }
+       return resultingFirstChild;
+   }
    function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
        const isObject = typeof newChild
        if (isObject) {
            switch (newChild.$typeof) {
                case REACT_ELEMENT_TYPE:
                    return placeSingleChild(
                        reconcileSingleElement(returnFiber, currentFirstChild, newChild)
                    );
                default:
                    break;
            }
        }
+       if (Array.isArray(newChild)) {
+           return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
+       }
    }
    return reconcileChildFibers;
}

export const reconcileChildFibers = ChildReconciler(true);
export const mountChildFibers = ChildReconciler(false);
```

9.4 src\ReactFiberCompleteWork.js

src\ReactFiberCompleteWork.js

```
+import { createInstance, finalizeInitialChildren, prepareUpdate, appendInitialChild } from './ReactDOMHostConfig';
import { HostComponent } from './ReactWorkTags';
import { Update } from './ReactFiberFlags';
export function completeWork(current, workInProgress) {
    const newProps = workInProgress.pendingProps;
    switch (workInProgress.tag) {
        case HostComponent: {
            if (current && workInProgress.stateNode) {
                updateHostComponent(current, workInProgress, workInProgress.tag, newProps);
            } else {
                const type = workInProgress.type;
                const instance = createInstance(type, newProps);
+               appendAllChildren(instance, workInProgress);
                workInProgress.stateNode = instance;
                finalizeInitialChildren(instance, type, newProps);
            }
            break;
        }
        default:
            break;
    }
}
/**
 * 更新原生DOM组件
 * @param {*} current 老fiber
 * @param {*} workInProgress 新fiber
 * @param {*} type 类型
 * @param {*} newProps  新属性
 */
function updateHostComponent(current, workInProgress, type, newProps) {
    //如果我们有一个替代方案，这意味着这是一个更新，我们需要安排一个副作用来进行更新
    const oldProps = current.memoizedProps;
    const instance = workInProgress.stateNode;
    //如果我们因为我们的一个孩子更新而得到更新，我们不会有 newProps 所以我们必须重用它们。
    const updatePayload = prepareUpdate(instance, type, oldProps, newProps);
    workInProgress.updateQueue = updatePayload;
    if (updatePayload) {
        markUpdate(workInProgress);
    }
}
function markUpdate(workInProgress) {
    //用更新效果标记fiber。 这会将 Placement 转换为 PlacementAndUpdate。
    workInProgress.flags |= Update;
}
+function appendAllChildren(parent, workInProgress) {
+    let node = workInProgress.child;
+    //我们只有创建的顶层 Fiber，但我们需要向下递归它的子节点以找到所有终端节点。
+    while (node) {
+        if (node.tag === HostComponent) {
+            let instance = node.stateNode;
+            appendInitialChild(parent, instance);
+        }
+        if (node === workInProgress) {
+            return;
+        }
+        while (!(node.sibling)) {
+            if (!(node.return) || node.return === workInProgress) {
+                return;
+            }
+            node = node.return;
+        }
+        node = node.sibling;
+    }
+}
```

9.5 ReactDOMHostConfig.js

src\ReactDOMHostConfig.js

```
import { createElement, setInitialProperties, diffProperties } from './ReactDOMComponent'
export function shouldSetTextContent(type, props) {
    return (
        typeof props.children
        typeof props.children
    );
}
export function createInstance(type) {
    return createElement(type);
}
export function finalizeInitialChildren(domElement, type, props) {
    setInitialProperties(domElement, type, props);
}
export function prepareUpdate(domElement, type, oldProps, newProps) {
    return diffProperties(
        domElement,
        type,
        oldProps,
        newProps
    );
}
export function appendChild(parentInstance, child) {
    parentInstance.appendChild(child);
}
export function insertBefore(parentInstance, child, beforeChild) {
    parentInstance.insertBefore(child, beforeChild);
}
+export function appendInitialChild(parentInstance, child) {
+    parentInstance.appendChild(child);
+}
```

10.多节点DIFF

- DOM DIFF的三个规则

    - 只对同级元素进行比较，不同层级不对比
    - 不同的类型对应不同的元素
    - 可以通过key来标识同一个节点
- 第1轮遍历
    - 如果key不同则直接结束本轮循环
    - newChildren或oldFiber遍历完，结束本轮循环
    - key相同而type不同，标记老的oldFiber为删除，继续循环
    - key相同而type也相同，则可以复用老节oldFiber节点，继续循环
- 第2轮遍历
    - newChildren遍历完而oldFiber还有，遍历剩下所有的oldFiber标记为删除，DIFF结束
    - oldFiber遍历完了，而newChildren还有，将剩下的newChildren标记为插入，DIFF结束
    - newChildren和oldFiber都同时遍历完成，diff结束
    - newChildren和oldFiber都没有完成，则进行 节点移动的逻辑
- 第3轮遍历
    - 处理节点移动的情况

11.多个节点的数量和key相同，有的type不同

- 多个节点的数量和key相同，有的type不同，则更新属性，type不同的删除老节点，删除新节点
- fiber图 (https://www.processon.com/diagraming/6190e63d5653bb36b39d05bf)
- 流程图 (https://www.processon.com/diagraming/619612845653bb30803edb4f)

11.1 public\index.html

public\index.html

```
  React App

    5.多个节点的数量和key相同，有的type不同
     <ul key="ul">
         <li key="A">A</li>
         <li key="B" id="B">B</li>
         <li key="C"id="C">C</li>
      </ul>
    更新属性，type不同的删除老节点，删除新节点
     <ul key="ul">
         <li key="A">A</li>
         <p key="B" id="B2">B2</p>
         <li key="C" id="C2" >C2</li>
      </ul>
```

11.2 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
//多节点diff
//5.多个节点的数量、类型和key全部相同，只更新属性
multil.addEventListener('click', () => {
  let element = (

      A
      B
      C

  );
  ReactDOM.render(element, root);
});
multilUpdate.addEventListener('click', () => {
  let element = (

      A
      B2
      C2

  );
  ReactDOM.render(element, root);
});
```

11.3 src\ReactChildFiber.js

src\ReactChildFiber.js

```
import { Placement, Deletion } from './ReactFiberFlags';
import { createFiberFromElement, createWorkInProgress } from './ReactFiber';
import { REACT_ELEMENT_TYPE } from './ReactSymbols';
function ChildReconciler(shouldTrackSideEffects) {
    function placeSingleChild(newFiber) {
        //如果要跟踪副作用并且没有老的fiber的话，就把此fiber标记为新建
        if (shouldTrackSideEffects && !newFiber.alternate) {
            newFiber.flags = Placement;
        }
        return newFiber;
    }
    function deleteChild(returnFiber, childToDelete) {
        if (!shouldTrackSideEffects) {
            return;
        }
        //把此fiber添加到父亲待删除的副作用链表中
        const last = returnFiber.lastEffect;
        if (last) {//如果父亲有尾部，就把此fiber添加到尾部的nextEffect上
            last.nextEffect = childToDelete;
            //然后让父亲的尾部等于自己
            returnFiber.lastEffect = childToDelete;
```

```
        } else {
            //如果父亲的副作用链表为空，头和尾向childToDelete
            returnFiber.firstEffect = returnFiber.lastEffect = childToDelete;
        }
        //清空它的nextEffect
        childToDelete.nextEffect = null;
        //把此fiber标记为删除
        childToDelete.flags = Deletion;
    }
    function deleteRemainingChildren(returnFiber, currentFirstChild) {
        let childToDelete = currentFirstChild;
        while (childToDelete) {
            deleteChild(returnFiber, childToDelete);
            childToDelete = childToDelete.sibling;
        }
        return null;
    }
    /**
 * 复用老的fiber
 * @param {*} fiber 老fiber
 * @param {*} pendingProps 新属性
 * @returns
 */
    function useFiber(fiber, pendingProps) {
        //根据老的fiber创建新的fiber
        return createWorkInProgress(fiber, pendingProps);
    }
    /**
     * 单节点DIFF
     * @param {*} returnFiber 当前父亲fiber
     * @param {*} currentFirstChild 当前第一个子节fiber
     * @param {*} element JSX虚拟DOM
     * @returns
     */
    function reconcileSingleElement(returnFiber, currentFirstChild, element) {
        //新jsx元素的key
        const key = element.key;
        //第一个老的fiber节点
        let child = currentFirstChild;
        //判断是否有老的fiber节点
        while (child) {
            //先判断两者的key是相同
            if (child.key
                //再判断类型是否相同,如果相同
                if (child.type
                    //删除剩下的所有的老fiber节点
                    deleteRemainingChildren(returnFiber, child.sibling);
                    //复用这个老的fiber节点,并传递新的属性
                    const existing = useFiber(child, element.props);
                    //让新的fiber的return指向当前的父fiber
                    existing.return = returnFiber;
                    //返回复用的fiber
                    return existing;
                } else {
                    //如果key相同但类型不同，不再进行后续匹配，后面的全部删除
                    deleteRemainingChildren(returnFiber, child);
                    break;
                }
            } else {
                //如果key不一样，则把当前的老fiber标记为删除，继续匹配弟弟
                deleteChild(returnFiber, child);
            }
            //找弟弟继续匹配
            child = child.sibling;
        }
        //根据虚拟DOM创建fiber节点
        const created = createFiberFromElement(element);
        //让新的fiber的return指向当前的父fiber
        created.return = returnFiber;
        return created;
    }
    function createChild(returnFiber, newChild) {
        if (typeof newChild
            switch (newChild.$typeof) {
                case REACT_ELEMENT_TYPE: {
                    const created = createFiberFromElement(newChild);
                    created.return = returnFiber;
                    return created;
                }
                default:
                    break;
            }
        }
    }
+   /**
+    * 更新元素
+    * @param {*} returnFiber 父fiber
+    * @param {*} current 老的的子fiber节点
+    * @param {*} element 新的JSX节点
+    * @returns
+    */
+   function updateElement(returnFiber, current, element) {
+       //如果老fiber存在
+       if (current) {
+           //而且新老type类型也一样
+           if (current.type === element.type) {
+               //复用老的fiber
+               const existing = useFiber(current, element.props);
+               existing.return = returnFiber;
+               return existing;
+           }
+       }
+       //如果老fiber不存在，则创建新的fiber
+       const created = createFiberFromElement(element);
```

```
+       created.return = returnFiber;
+       return created;
+   }
+   function updateSlot(returnFiber, oldFiber, newChild) {
+       //获取老fiber节点的key
+       const key = oldFiber ? oldFiber.key : null;
+       //如果新的JSX子节点是一个对象
+       if (typeof newChild === 'object' && newChild) {
+           //如果新老key是一样的，则表示找到了可复用的节点
+           if (newChild.key === key) {
+               return updateElement(returnFiber, oldFiber, newChild);
+           } else {
+               //key不存在，不能复用
+               return null;
+           }
+       }
+   }
+    function placeChild(newFiber, newIndex) {
+        newFiber.index = newIndex;
+        if (!shouldTrackSideEffects) {
+            return;
+        }
+        const current = newFiber.alternate;
+        if (current) {
+
+        } else {
+            newFiber.flags = Placement;
+            return;
+        }
+    }
    /**
     * 协调子节点
     * @param {*} returnFiber 父fiber
     * @param {*} currentFirstChild 当前的第一个子fiber
     * @param {*} newChildren JSX对象
     * @returns
     */
    function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
        //该算法无法通过两端搜索进行优化，因为fiber上没有反向指针
        //我在试着看看我们能用这个模型走多远。如果它最终不值得，我们可以稍后添加它
        //返回的第一个fiber子节点
        let resultingFirstChild = null;
        //上一个新的fiber节点
        let previousNewFiber = null;
        //比较中的旧的fiber节点
        let oldFiber = currentFirstChild;
+       //下一个老的fiber
+       let nextOldFiber = null;
        //新的索引
        let newIdx = 0;
+       //先处理节点更新的情况
+       for (; oldFiber && newIdx < newChildren.length; newIdx++) {
+           //先缓存下下一个老的fiber节点
+           nextOldFiber = oldFiber.sibling;
+           //更新复用的新fiber
+           //判断key和type是否相同，如果相同表示可以复用，newIdx++后和下一个oldFiber比较
+           const newFiber = updateSlot(returnFiber, oldFiber, newChildren[newIdx]);
+           if (!newFiber) {
+               break;
+           }
+           //如果类型相同，但类型不同，但没有重用现有fiber，因此需要删除现有子级fiber
+           if (oldFiber && !(newFiber.alternate)) {
+               deleteChild(returnFiber, oldFiber);
+           }
+           //放置此newFiber到newIdx，就是设置newFiber的index索引
+           placeChild(newFiber, newIdx);
+           //如果previousNewFiber不存在表示这是第一个fiber
+           if (!previousNewFiber) {
+               resultingFirstChild = newFiber;
+           } else {
+               //否则上一个新fiber的sibling等于这个newFiber
+               previousNewFiber.sibling = newFiber;
+           }
+           //让当前的newFiber等于previousNewFiber
+           previousNewFiber = newFiber;
+           //下一个老fiber
+           oldFiber = nextOldFiber;
+       }
        //如果老fiber完成，新的JSX没有完成
        if (!oldFiber) {
            for (; newIdx < newChildren.length; newIdx++) {
                const newFiber = createChild(returnFiber, newChildren[newIdx]);
                if (!previousNewFiber) {
                    resultingFirstChild = newFiber;
                } else {
                    previousNewFiber.sibling = newFiber;
                }
                previousNewFiber = newFiber;
            }
            return resultingFirstChild;
        }
        return resultingFirstChild;
    }
    function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
        const isObject = typeof newChild
        if (isObject) {
            switch (newChild.$typeof) {
                case REACT_ELEMENT_TYPE:
                    return placeSingleChild(
                        reconcileSingleElement(returnFiber, currentFirstChild, newChild)
                    );
                default:
                    break;
            }
```

```
        }
        if (Array.isArray(newChild)) {
            return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
        }
    }
    return reconcileChildFibers;
}

export const reconcileChildFibers = ChildReconciler(true);
export const mountChildFibers = ChildReconciler(false);
```

11.4 src\ReactFiberWorkLoop.js

src\ReactFiberWorkLoop.js

```
import { HostRoot, HostComponent } from './ReactWorkTags';
import { createWorkInProgress } from './ReactFiber';
import { beginWork } from './ReactFiberBeginWork';
import { completeWork } from './ReactFiberCompleteWork';
import { Placement, Update, Deletion } from './ReactFiberFlags';
import { commitWork } from './ReactFiberCommitWork';
//正在调度的fiberRoot根节点
let workInProgressRoot = null;
//正在处理的fiber节点
let workInProgress = null;
//最大更新深度
const NESTED_UPDATE_LIMIT = 50;
let nestedUpdateCount = 0;
/**
 * 向上获取HostRoot节点
 * @param {*} sourceFiber 更新来源fiber
 * @returns
 */
function markUpdateLaneFromFiberToRoot(sourceFiber) {
    let node = sourceFiber;
    let parent = node.return;
    //一直向上找父亲，找不到为止
    while (parent) {
        node = parent;
        parent = parent.return;
    }
    //如果找到的是HostRoot就返回FiberRootNode,其实就是容器div#root
    if (node.tag
        return node.stateNode;
    }
}
/**
 * 向上查找到根节点开始调度更新
 * @param {*} fiber
 */
export function scheduleUpdateOnFiber(fiber) {
    checkForNestedUpdates();
    //向上获取HostRoot节点
    const root = markUpdateLaneFromFiberToRoot(fiber);
    //执行HostRoot上的更新
    performSyncWorkOnRoot(root);
}
function checkForNestedUpdates() {
    if (++nestedUpdateCount > NESTED_UPDATE_LIMIT) {
        throw new Error('Maximum update depth exceeded');
    }
}
/**
 * 开始执行FiberRootNode上的工作
 * @param {*} root  FiberRootNode
 */
function performSyncWorkOnRoot(root) {
    //先赋值给当前正在执行工作的FiberRootNode根节点
    workInProgressRoot = root;
    //创建一个新的处理中的fiber节点
    workInProgress = createWorkInProgress(workInProgressRoot.current);
    workLoopSync();
    commitRoot();
}
function commitRoot(root) {
    //构建成功的新的fiber树
    const finishedWork = workInProgressRoot.current.alternate;
    //当前完成的构建工作等于finishedWork
    workInProgressRoot.finishedWork = finishedWork;
    commitMutationEffects(workInProgressRoot);
    nestedUpdateCount--;
}
function getFlag(flags) {
    switch (flags) {
        case Placement:
            return '添加';
        case Update:
            return '更新';
        case Deletion:
            return '删除';
    }
}
function commitMutationEffects(root) {
    const finishedWork = root.finishedWork;
    let nextEffect = finishedWork.firstEffect;
    let effectList = '';
    while (nextEffect) {
        effectList += `(${getFlag(nextEffect.flags)}#${nextEffect.type}#${nextEffect.key})=>`;
        const flags = nextEffect.flags;
        const current = nextEffect.alternate;
        if (flags
            commitPlacement(nextEffect);
        } else if (flags
            commitWork(current, nextEffect);
```

```
        } else if (flags
            commitDeletion(nextEffect);
        }
        nextEffect = nextEffect.nextEffect;
    }
    effectList += 'null';
    console.log(effectList);
    root.current = finishedWork;
}
function commitDeletion(fiber) {
    if (!fiber) {
        return;
    }
    let parentStateNode = getParentStateNode(fiber);
    parentStateNode.removeChild(fiber.stateNode);
}
function workLoopSync() {
    while (workInProgress) {
        performUnitOfWork(workInProgress);
    }
}
/**
 * 执行单个工作单元
 * @param {*} unitOfWork 单个fiber
 */
function performUnitOfWork(unitOfWork) {
    //获取当前fiber的alternate
    const current = unitOfWork.alternate;
    //开始构建此fiber的子fiber链表
    let next = beginWork(current, unitOfWork);
    //更新属性
    unitOfWork.memoizedProps = unitOfWork.pendingProps;
    //如果有子fiber，就继续执行
    if (next) {
        workInProgress = next;
    } else {
        //如果没有子fiber，就完成当前的fiber
        completeUnitOfWork(unitOfWork);
    }
}

function completeUnitOfWork(unitOfWork) {
    //尝试完成当前的工作单元，然后移动到下一个弟弟
    //如果没有下一个弟弟，返回到父Fiber
    let completedWork = unitOfWork;
    do {
        const current = completedWork.alternate;
        const returnFiber = completedWork.return;
        completeWork(current, completedWork);
        collectEffectList(returnFiber, completedWork)
        const siblingFiber = completedWork.sibling;
        if (siblingFiber) {
            //如果此 returnFiber 中还有更多工作要做，请执行下一步
            workInProgress = siblingFiber;
            return;
        }
        //否则，返回父级
        completedWork = returnFiber;
        //更新我们正在处理的下一件事
        workInProgress = completedWork;
    } while (completedWork);
}
function collectEffectList(returnFiber, completedWork) {
    if (returnFiber) {
        if (!returnFiber.firstEffect) {
            returnFiber.firstEffect = completedWork.firstEffect;
        }
        if (completedWork.lastEffect) {
            if (returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = completedWork.firstEffect;
            }
            returnFiber.lastEffect = completedWork.lastEffect;
        }
        //如果这个fiber有副作用，我们会在孩子们的副使用后面添加它自己的副作用
        const flags = completedWork.flags;
        if (flags) {
            if (returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = completedWork;
            } else {
                returnFiber.firstEffect = completedWork;
            }
            returnFiber.lastEffect = completedWork;
        }
    }
}
```

11.5 ReactFiberCommitWork.js

src\ReactFiberCommitWork.js

```
import { updateProperties } from './ReactDOMComponent';
import { HostComponent, HostRoot } from './ReactWorkTags';
import { appendChild, insertBefore } from './ReactDOMHostConfig';
import { Placement } from './ReactFiberFlags';
/**
 * 更新DOM节点的属性
 * @param {*} current
 * @param {*} finishedWork
 */
export function commitWork(current, finishedWork) {
    switch (finishedWork.tag) {
        case HostComponent: {
            const updatePayload = finishedWork.updateQueue;
            finishedWork.updateQueue = null;
            if (updatePayload) {
                updateProperties(finishedWork.stateNode, updatePayload);
            }
        }
        default:
            break;
    }
}
function getParentStateNode(fiber) {
    const parent = fiber.return;
    do {
        if (parent.tag
            return parent.stateNode;
        } else if (parent.tag
            return parent.stateNode.containerInfo
        }
        parent = parent.return;
    } while (parent);
}
+function getHostSibling(fiber) {
+    let node = fiber.sibling;
+    while (node) {
+        if (!(node.flags & Placement)) {
+            return node.stateNode;
+        }
+        node = node.sibling;
+    }
+    return null;
+}
export function commitPlacement(finishedWork) {
    let stateNode = finishedWork.stateNode;
    let parentStateNode = getParentStateNode(finishedWork);
+   let before = getHostSibling(finishedWork);
+   if (before) {
+       insertBefore(parentStateNode, stateNode, before);
+   } else {
+       appendChild(parentStateNode, stateNode);
+   }
}
```

12. 多个节点的类型和key全部相同，有新增元素

12.1 public\index.html

```
   React App

    6.多个节点的类型和key全部相同，有新增元素
      <ul key="ul">
          <li key="A">A</li>
          <li key="B" id="B">B</li>
          <li key="C">C</li>
      </ul>
    增加新元素并更新老元素
    <ul key="ul">
        <li key="A">A</li>
        <li key="B" id="B2">B2</li>
        <li key="C">C</li>
        <li key="D">D</li>
    </ul>
```

12.2 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
//6.多个节点的类型和key全部相同，有新增元素
multi2.addEventListener('click', () => {
  let element = (

      A
      B
      C

  );
  ReactDOM.render(element, root);
});
//增加新元素并更新老元素
multi2Update.addEventListener('click', () => {
  let element = (

      A
      B2
      C
      D

  );
  ReactDOM.render(element, root);
});
```

12.3 src\ReactChildFiber.js

src\ReactChildFiber.js

```
import { Placement, Deletion } from './ReactFiberFlags';
import { createFiberFromElement, createWorkInProgress } from './ReactFiber';
import { REACT_ELEMENT_TYPE } from './ReactSymbols';
function ChildReconciler(shouldTrackSideEffects) {
    function placeSingleChild(newFiber) {
        //如果要跟踪副作用并且没有老的fiber的话，就把此fiber标记为新建
        if (shouldTrackSideEffects && !newFiber.alternate) {
            newFiber.flags = Placement;
        }
        return newFiber;
    }
    function deleteChild(returnFiber, childToDelete) {
        if (!shouldTrackSideEffects) {
            return;
        }
        //把此fiber添加到父亲待删除的副作用链表中
        const last = returnFiber.lastEffect;
        if (last) {//如果父亲有尾部，就把此fiber添加到尾部的nextEffect上
            last.nextEffect = childToDelete;
            //然后让父亲的尾部等于自己
            returnFiber.lastEffect = childToDelete;
        } else {
            //如果父亲的副作用链表为空，头和尾向childToDelete
            returnFiber.firstEffect = returnFiber.lastEffect = childToDelete;
        }
        //清空它的nextEffect
        childToDelete.nextEffect = null;
        //把此fiber标记为删除
        childToDelete.flags = Deletion;
    }
    function deleteRemainingChildren(returnFiber, currentFirstChild) {
        let childToDelete = currentFirstChild;
        while (childToDelete) {
            deleteChild(returnFiber, childToDelete);
            childToDelete = childToDelete.sibling;
        }
        return null;
    }
    /**
  * 复用老的fiber
  * @param {*} fiber 老fiber
  * @param {*} pendingProps 新属性
  * @returns
  */
    function useFiber(fiber, pendingProps) {
        //根据老的fiber创建新的fiber
        return createWorkInProgress(fiber, pendingProps);
    }
    /**
     * 单节点DIFF
     * @param {*} returnFiber 当前父亲fiber
     * @param {*} currentFirstChild 当前第一个子节fiber
     * @param {*} element JSX虚拟DOM
     * @returns
     */
    function reconcileSingleElement(returnFiber, currentFirstChild, element) {
        //新jsx元素的key
        const key = element.key;
        //第一个老的fiber节点
        let child = currentFirstChild;
        //判断是否有老的fiber节点
        while (child) {
            //先判断两者的key是相同
            if (child.key
                //再判断类型是否相同,如果相同
                if (child.type
                    //删除剩下的所有的老fiber节点
                    deleteRemainingChildren(returnFiber, child.sibling);
                    //复用这个老的fiber节点,并传递新的属性
                    const existing = useFiber(child, element.props);
                    //让新的fiber的return指向当前的父fiber
                    existing.return = returnFiber;
                    //返回复用的fiber
                    return existing;
```

```
            } else {
                //如果key相同但类型不同，不再进行后续匹配，后面的全部删除
                deleteRemainingChildren(returnFiber, child);
                break;
            }
        } else {
            //如果key不一样，则把当前的老fiber标记为删除，继续匹配弟弟
            deleteChild(returnFiber, child);
        }
        //找弟弟继续匹配
        child = child.sibling;
    }
    //根据虚拟DOM创建fiber节点
    const created = createFiberFromElement(element);
    //让新的fiber的return指向当前的父fiber
    created.return = returnFiber;
    return created;
}
function createChild(returnFiber, newChild) {
    if (typeof newChild
        switch (newChild.$typeof) {
            case REACT_ELEMENT_TYPE: {
                const created = createFiberFromElement(newChild);
                created.return = returnFiber;
                return created;
            }
            default:
                break;
        }
    }
}
/**
 * 更新元素
 * @param {*} returnFiber 父fiber
 * @param {*} current 老的的子fiber节点
 * @param {*} element 新的JSX节点
 * @returns
 */
function updateElement(returnFiber, current, element) {
    //如果老fiber存在
    if (current) {
        //而且新老type类型也一样
        if (current.type
            //复用老的fiber
            const existing = useFiber(current, element.props);
            existing.return = returnFiber;
            return existing;
        }
    }
    //如果老fiber不存在，则创建新的fiber
    const created = createFiberFromElement(element);
    created.return = returnFiber;
    return created;
}
function updateSlot(returnFiber, oldFiber, newChild) {
    //获取老fiber节点的key
    const key = oldFiber ? oldFiber.key : null;
    //如果新的JSX子节点是一个对象
    if (typeof newChild
        //如果新老key是一样的，则表示找到了可复用的节点
        if (newChild.key
            return updateElement(returnFiber, oldFiber, newChild);
        } else {
            //key不存在，不能复用
            return null;
        }
    }
}
function placeChild(newFiber, newIndex) {
    newFiber.index = newIndex;
    if (!shouldTrackSideEffects) {
        return;
    }
    const current = newFiber.alternate;
    if (current) {

    } else {
        newFiber.flags = Placement;
        return;
    }
}
/**
 * 协调子节点
 * @param {*} returnFiber 父fiber
 * @param {*} currentFirstChild 当前的第一个子fiber
 * @param {*} newChildren JSX对象
 * @returns
 */
function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
    //该算法无法通过两端搜索进行优化，因为fiber上没有反向指针
    //我在试着看看我们能用这个模型走多远。如果它最终不值得，我们可以稍后添加它
    //返回的第一个fiber子节点
    let resultingFirstChild = null;
    //上一个新的fiber节点
    let previousNewFiber = null;
    //比较中的旧的fiber节点
    let oldFiber = currentFirstChild;
    //下一个老的fiber
    let nextOldFiber = null;
    //新的索引
    let newIdx = 0;
    //先处理节点更新的情况
    for (; oldFiber && newIdx < newChildren.length; newIdx++) {
        //先缓存下下一个老的fiber节点
```

```
            nextOldFiber = oldFiber.sibling;
            //更新复用的新fiber
            //判断key和type是否相同，如果相同表示可以复用，newIdx++后和下一个oldFiber比较
            const newFiber = updateSlot(returnFiber, oldFiber, newChildren[newIdx]);
            if (!newFiber) {
                break;
            }
            //如果类型相同，但类型不同，但没有重用现有fiber，因此需要删除现有子级fiber
            if (oldFiber && !(newFiber.alternate)) {
                deleteChild(returnFiber, oldFiber);
            }
            //放置此newFiber到newIdx，就是设置newFiber的index索引
            placeChild(newFiber, newIdx);
            //如果previousNewFiber不存在表示这是第一个fiber
            if (!previousNewFiber) {
                resultingFirstChild = newFiber;
            } else {
                //否则上一个新fiber的sibling等于这个newFiber
                previousNewFiber.sibling = newFiber;
            }
            //让当前的newFiber等于previousNewFiber
            previousNewFiber = newFiber;
            //下一个老fiber
            oldFiber = nextOldFiber;
        }
        //如果老fiber完成，新的JSX没有完成
        if (!oldFiber) {
            for (; newIdx < newChildren.length; newIdx++) {
                const newFiber = createChild(returnFiber, newChildren[newIdx]);
+               placeChild(newFiber, newIdx);
                if (!previousNewFiber) {
                    resultingFirstChild = newFiber;
                } else {
                    previousNewFiber.sibling = newFiber;
                }
                previousNewFiber = newFiber;
            }
            return resultingFirstChild;
        }
        return resultingFirstChild;
    }
    function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
        const isObject = typeof newChild
        if (isObject) {
            switch (newChild.$typeof) {
                case REACT_ELEMENT_TYPE:
                    return placeSingleChild(
                        reconcileSingleElement(returnFiber, currentFirstChild, newChild)
                    );
                default:
                    break;
            }
        }
        if (Array.isArray(newChild)) {
            return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
        }
    }
    return reconcileChildFibers;
}

export const reconcileChildFibers = ChildReconciler(true);
export const mountChildFibers = ChildReconciler(false);
```

13. 7.多个节点的类型和key全部相同，有删除老元素

- 多个节点的类型和key全部相同，有删除老元素，删除老元素并更新老元素

13.1 public\index.html

public\index.html

```
  React App

    7.多个节点的类型和key全部相同，有删除老元素
    <ul key="ul">
        <li key="A">A</li>
        <li key="B" id="B">B</li>
        <li key="C">C</li>
    </ul>
    删除老元素并更新老元素
    <ul key="ul">
        <li key="A">A</li>
        <li key="B" id="B2">B2</li>
    </ul>
```

13.2 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
//7.多个节点的类型和key全部相同，有删除老元素
multi3.addEventListener('click', () => {
  let element = (

      A
      B
      C

  );
  ReactDOM.render(element, root);
});
multi3Update.addEventListener('click', () => {
  let element = (

      A
      B2

  );
  ReactDOM.render(element, root);
});
```

13.3 ReactChildFiber.js

src\ReactChildFiber.js

```
import { Placement, Deletion } from './ReactFiberFlags';
import { createFiberFromElement, createWorkInProgress } from './ReactFiber';
import { REACT_ELEMENT_TYPE } from './ReactSymbols';
function ChildReconciler(shouldTrackSideEffects) {
    function placeSingleChild(newFiber) {
        //如果要跟踪副作用并且没有老的fiber的话，就把此fiber标记为新建
        if (shouldTrackSideEffects && !newFiber.alternate) {
            newFiber.flags = Placement;
        }
        return newFiber;
    }
    function deleteChild(returnFiber, childToDelete) {
        if (!shouldTrackSideEffects) {
            return;
        }
        //把此fiber添加到父亲待删除的副作用链表中
        const last = returnFiber.lastEffect;
        if (last) {//如果父亲有尾部，就把此fiber添加到尾部的nextEffect上
            last.nextEffect = childToDelete;
            //然后让父亲的尾部等于自己
            returnFiber.lastEffect = childToDelete;
        } else {
            //如果父亲的副作用链表为空，头和尾向childToDelete
            returnFiber.firstEffect = returnFiber.lastEffect = childToDelete;
        }
        //清空它的nextEffect
        childToDelete.nextEffect = null;
        //把此fiber标记为删除
        childToDelete.flags = Deletion;
    }
    function deleteRemainingChildren(returnFiber, currentFirstChild) {
        let childToDelete = currentFirstChild;
        while (childToDelete) {
            deleteChild(returnFiber, childToDelete);
            childToDelete = childToDelete.sibling;
        }
        return null;
    }
    /**
  * 复用老的fiber
  * @param {*} fiber 老fiber
  * @param {*} pendingProps 新属性
  * @returns
  */
    function useFiber(fiber, pendingProps) {
        //根据老的fiber创建新的fiber
        return createWorkInProgress(fiber, pendingProps);
    }
    /**
     * 单节点DIFF
     * @param {*} returnFiber 当前父亲fiber
     * @param {*} currentFirstChild 当前第一个子节fiber
     * @param {*} element JSX虚拟DOM
     * @returns
     */
    function reconcileSingleElement(returnFiber, currentFirstChild, element) {
        //新jsx元素的key
        const key = element.key;
        //第一个老的fiber节点
        let child = currentFirstChild;
        //判断是否有老的fiber节点
        while (child) {
            //先判断两者的key是相同
            if (child.key
                //再判断类型是否相同,如果相同
                if (child.type
                    //删除剩下的所有的老fiber节点
                    deleteRemainingChildren(returnFiber, child.sibling);
                    //复用这个老的fiber节点,并传递新的属性
                    const existing = useFiber(child, element.props);
                    //让新的fiber的return指向当前的父fiber
                    existing.return = returnFiber;
                    //返回复用的fiber
                    return existing;
                } else {
                    //如果key相同但类型不同，不再进行后续匹配，后面的全部删除
                    deleteRemainingChildren(returnFiber, child);
```

```
                break;
            }
        } else {
            //如果key不一样，则把当前的老fiber标记为删除，继续匹配弟弟
            deleteChild(returnFiber, child);
        }
        //找弟弟继续匹配
        child = child.sibling;
    }
    //根据虚拟DOM创建fiber节点
    const created = createFiberFromElement(element);
    //让新的fiber的return指向当前的父fiber
    created.return = returnFiber;
    return created;
}
function createChild(returnFiber, newChild) {
    if (typeof newChild
        switch (newChild.$typeof) {
            case REACT_ELEMENT_TYPE: {
                const created = createFiberFromElement(newChild);
                created.return = returnFiber;
                return created;
            }
            default:
                break;
        }
    }
}
/**
 * 更新元素
 * @param {*} returnFiber 父fiber
 * @param {*} current 老的的子fiber节点
 * @param {*} element 新的JSX节点
 * @returns
 */
function updateElement(returnFiber, current, element) {
    //如果老fiber存在
    if (current) {
        //而且新老type类型也一样
        if (current.type
            //复用老的fiber
            const existing = useFiber(current, element.props);
            existing.return = returnFiber;
            return existing;
        }
    }
    //如果老fiber不存在，则创建新的fiber
    const created = createFiberFromElement(element);
    created.return = returnFiber;
    return created;
}
function updateSlot(returnFiber, oldFiber, newChild) {
    //获取老fiber节点的key
    const key = oldFiber ? oldFiber.key : null;
    //如果新的JSX子节点是一个对象
    if (typeof newChild
        //如果新老key是一样的，则表示找到了可复用的节点
        if (newChild.key
            return updateElement(returnFiber, oldFiber, newChild);
        } else {
            //key不存在，不能复用
            return null;
        }
    }
}
function placeChild(newFiber, newIndex) {
    newFiber.index = newIndex;
    if (!shouldTrackSideEffects) {
        return;
    }
    const current = newFiber.alternate;
    if (current) {

    } else {
        newFiber.flags = Placement;
        return;
    }
}
/**
 * 协调子节点
 * @param {*} returnFiber 父fiber
 * @param {*} currentFirstChild 当前的第一个子fiber
 * @param {*} newChildren JSX对象
 * @returns
 */
function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
    //该算法无法通过两端搜索进行优化，因为fiber上没有反向指针
    //我在试着看看我们能用这个模型走多远。如果它最终不值得，我们可以稍后添加它
    //返回的第一个fiber子节点
    let resultingFirstChild = null;
    //上一个新的fiber节点
    let previousNewFiber = null;
    //比较中的旧的fiber节点
    let oldFiber = currentFirstChild;
    //下一个老的fiber
    let nextOldFiber = null;
    //新的索引
    let newIdx = 0;
    //先处理节点更新的情况
    for (; oldFiber && newIdx < newChildren.length; newIdx++) {
        //先缓存下下一个老的fiber节点
        nextOldFiber = oldFiber.sibling;
        //更新复用的新fiber
        //判断key和type是否相同，如果相同表示可以复用，newIdx++后和下一个oldFiber比较
```

```
            const newFiber = updateSlot(returnFiber, oldFiber, newChildren[newIdx]);
            if (!newFiber) {
                break;
            }
            //如果类型相同，但类型不同，但没有重用现有fiber，因此需要删除现有子级fiber
            if (oldFiber && !(newFiber.alternate)) {
                deleteChild(returnFiber, oldFiber);
            }
            //放置此newFiber到newIdx，就是设置newFiber的index索引
            placeChild(newFiber, newIdx);
            //如果previousNewFiber不存在表示这是第一个fiber
            if (!previousNewFiber) {
                resultingFirstChild = newFiber;
            } else {
                //否则上一个新fiber的sibling等于这个newFiber
                previousNewFiber.sibling = newFiber;
            }
            //让当前的newFiber等于previousNewFiber
            previousNewFiber = newFiber;
            //下一个老fiber
            oldFiber = nextOldFiber;
        }
+       //如果新的遍历完了，删除掉所有老的剩下的fiber，直接返回resultingFirstChild
+       if (newIdx === newChildren.length) {
+           deleteRemainingChildren(returnFiber, oldFiber);
+           return resultingFirstChild;
+       }
        //如果老fiber完成，新的JSX没有完成
        if (!oldFiber) {
            for (; newIdx < newChildren.length; newIdx++) {
                const newFiber = createChild(returnFiber, newChildren[newIdx]);
                placeChild(newFiber, newIdx);
                if (!previousNewFiber) {
                    resultingFirstChild = newFiber;
                } else {
                    previousNewFiber.sibling = newFiber;
                }
                previousNewFiber = newFiber;
            }
            return resultingFirstChild;
        }
        return resultingFirstChild;
    }
    function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
        const isObject = typeof newChild
        if (isObject) {
            switch (newChild.$typeof) {
                case REACT_ELEMENT_TYPE:
                    return placeSingleChild(
                        reconcileSingleElement(returnFiber, currentFirstChild, newChild)
                    );
                default:
                    break;
            }
        }
        if (Array.isArray(newChild)) {
            return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
        }
    }
    return reconcileChildFibers;
}

export const reconcileChildFibers = ChildReconciler(true);
export const mountChildFibers = ChildReconciler(false);
```

14. 多个节点数量不同、key不同

- 多个节点数量不同、key不同 (https://www.processon.com/diagraming/6193773ef346fb6e38a56734)
- 第一轮比较A和A，相同可以复用，更新，然后比较B和C，key不同直接跳出第一个循环
- 把剩下oldFiber的放入existingChildren这个map中
- 然后声明一个 lastPlacedIndex变量，表示不需要移动的老节点的索引
- 继续循环剩下的虚拟DOM节点
- 如果能在map中找到相同key相同type的节点则可以复用老fiber,并把此老fiber从map中删除
- 如果能在map中找不到相同key相同type的节点则创建新的fiber
- 如果是复用老的fiber,则判断老fiber的索引是否小于lastPlacedIndex，如果是要移动老fiber，不变
- 如果是复用老的fiber,则判断老fiber的索引是否小于lastPlacedIndex，如果否则更新lastPlacedIndex为老fiber的index
- 把所有的map中剩下的fiber全部标记为删除
- (删除#li#F)=>(添加#li#B)=>(添加#li#G)=>(添加#li#D)=>null

14.1 public\index.html

public\index.html

```
  React App

  9.多个节点数量不同、key不同
    <ul key="ul">
       <li key="A">A</li>
       <li key="B" id="b">B</li>
       <li key="C">C</li>
       <li key="D">D</li>
       <li key="E">E</li>
       <li key="F">F</li>
    </ul>
   处理节点移动的情况
    <ul key="ul">
       <li key="A">A</li>
       <li key="C">C</li>
       <li key="E">E</li>
       <li key="B" id="b2">B2</li>
       <li key="G">G</li>
       <li key="D">D</li>
    </ul>
```

14.2 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
//9.多个节点数量不同、key不同
multi5.addEventListener('click', () => {
  let element = (

      A
      B
      C
      D
      E
      F

  );
  ReactDOM.render(element, root);
});
multi5Update.addEventListener('click', () => {
  let element = (

      A
      C
      E
      B2
      G
      D

  );
  ReactDOM.render(element, root);
});
```

14.3 src\ReactChildFiber.js

src\ReactChildFiber.js

```
import { Placement, Deletion } from './ReactFiberFlags';
import { createFiberFromElement, createWorkInProgress } from './ReactFiber';
import { REACT_ELEMENT_TYPE } from './ReactSymbols';
function ChildReconciler(shouldTrackSideEffects) {
    function placeSingleChild(newFiber) {
        //如果要跟踪副作用并且没有老的fiber的话，就把此fiber标记为新建
        if (shouldTrackSideEffects && !newFiber.alternate) {
            newFiber.flags = Placement;
        }
        return newFiber;
    }
    function deleteChild(returnFiber, childToDelete) {
        if (!shouldTrackSideEffects) {
            return;
        }
        //把此fiber添加到父亲待删除的副作用链表中
        const last = returnFiber.lastEffect;
        if (last) {//如果父亲有尾部，就把此fiber添加到尾部的nextEffect上
            last.nextEffect = childToDelete;
            //然后让父亲的尾部等于自己
            returnFiber.lastEffect = childToDelete;
        } else {
            //如果父亲的副作用链表为空，头和尾向childToDelete
            returnFiber.firstEffect = returnFiber.lastEffect = childToDelete;
        }
        //清空它的nextEffect
        childToDelete.nextEffect = null;
        //把此fiber标记为删除
        childToDelete.flags = Deletion;
    }
    function deleteRemainingChildren(returnFiber, currentFirstChild) {
        let childToDelete = currentFirstChild;
        while (childToDelete) {
            deleteChild(returnFiber, childToDelete);
            childToDelete = childToDelete.sibling;
        }
        return null;
    }
    /**
  * 复用老的fiber
  * @param {*} fiber 老fiber
  * @param {*} pendingProps 新属性
  * @returns
```

```
 */
  function useFiber(fiber, pendingProps) {
      //根据老的fiber创建新的fiber
      return createWorkInProgress(fiber, pendingProps);
  }
  /**
   * 单节点DIFF
   * @param {*} returnFiber 当前父亲fiber
   * @param {*} currentFirstChild 当前第一个子节fiber
   * @param {*} element JSX虚拟DOM
   * @returns
   */
  function reconcileSingleElement(returnFiber, currentFirstChild, element) {
      //新jsx元素的key
      const key = element.key;
      //第一个老的fiber节点
      let child = currentFirstChild;
      //判断是否有老的fiber节点
      while (child) {
          //先判断两者的key是相同
          if (child.key
              //再判断类型是否相同,如果相同
              if (child.type
                  //删除剩下的所有的老fiber节点
                  deleteRemainingChildren(returnFiber, child.sibling);
                  //复用这个老的fiber节点,并传递新的属性
                  const existing = useFiber(child, element.props);
                  //让新的fiber的return指向当前的父fiber
                  existing.return = returnFiber;
                  //返回复用的fiber
                  return existing;
              } else {
                  //如果key相同但类型不同，不再进行后续匹配，后面的全部删除
                  deleteRemainingChildren(returnFiber, child);
                  break;
              }
          } else {
              //如果key不一样，则把当前的老fiber标记为删除，继续匹配弟弟
              deleteChild(returnFiber, child);
          }
          //找弟弟继续匹配
          child = child.sibling;
      }
      //根据虚拟DOM创建fiber节点
      const created = createFiberFromElement(element);
      //让新的fiber的return指向当前的父fiber
      created.return = returnFiber;
      return created;
  }
  function createChild(returnFiber, newChild) {
      if (typeof newChild
          switch (newChild.$typeof) {
              case REACT_ELEMENT_TYPE: {
                  const created = createFiberFromElement(newChild);
                  created.return = returnFiber;
                  return created;
              }
              default:
                  break;
          }
      }
  }
  /**
   * 更新元素
   * @param {*} returnFiber 父fiber
   * @param {*} current 老的的子fiber节点
   * @param {*} element 新的JSX节点
   * @returns
   */
  function updateElement(returnFiber, current, element) {
      //如果老fiber存在
      if (current) {
          //而且新老type类型也一样
          if (current.type
              //复用老的fiber
              const existing = useFiber(current, element.props);
              existing.return = returnFiber;
              return existing;
          }
      }
      //如果老fiber不存在，则创建新的fiber
      const created = createFiberFromElement(element);
      created.return = returnFiber;
      return created;
  }
  function updateSlot(returnFiber, oldFiber, newChild) {
      //获取老fiber节点的key
      const key = oldFiber ? oldFiber.key : null;
      //如果新的JSX子节点是一个对象
      if (typeof newChild
          //如果新老key是一样的，则表示找到了可复用的节点
          if (newChild.key
              return updateElement(returnFiber, oldFiber, newChild);
          } else {
              //key不存在，不能复用
              return null;
          }
      }
  }
+ function placeChild(newFiber, lastPlacedIndex, newIndex) {
      newFiber.index = newIndex;
      if (!shouldTrackSideEffects) {
+         return lastPlacedIndex;
      }
```

```
        const current = newFiber.alternate;
        if (current) {
+           const oldIndex = current.index;
+           if (oldIndex < lastPlacedIndex) {
+               //这是一个移动操作
+               newFiber.flags = Placement;
+               return lastPlacedIndex;
+           } else {
+               //这个项目可以保留在原地
+               return oldIndex;
+           }
        } else {
            newFiber.flags = Placement;
+           return lastPlacedIndex;
        }
    }
+   function updateFromMap(existingChildren, returnFiber, newIdx, newChild) {
+       if (typeof newChild === 'object' && newChild) {
+           switch (newChild.$typeof) {
+               case REACT_ELEMENT_TYPE: {
+                   const matchedFiber =
+                       existingChildren.get(newChild.key || newIdx) || null;
+                   return updateElement(returnFiber, matchedFiber, newChild);
+               }
+               default:
+                   break;
+           }
+       }
+       return null;
+   }
+   function mapRemainingChildren(returnFiber, currentFirstChild,) {
+       //将剩余的子对象添加到临时map中以便我们可以通过key快速找到它们
+       //隐式的key值为null的话会使用索引作
+       const existingChildren = new Map();
+       let existingChild = currentFirstChild;
+       while (existingChild) {
+           if (existingChild.key) {
+               existingChildren.set(existingChild.key, existingChild);
+           } else {
+               existingChildren.set(existingChild.index, existingChild);
+           }
+           existingChild = existingChild.sibling;
+       }
+       return existingChildren;
+   }
    /**
     * 协调子节点
     * @param {*} returnFiber 父fiber
     * @param {*} currentFirstChild 当前的第一个子fiber
     * @param {*} newChildren JSX对象
     * @returns
     */
    function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
        //该算法无法通过两端搜索进行优化，因为fiber上没有反向指针
        //我在试着看看我们能用这个模型走多远。如果它最终不值得，我们可以稍后添加它
        //返回的第一个fiber子节点
        let resultingFirstChild = null;
        //上一个新的fiber节点
        let previousNewFiber = null;
        //比较中的旧的fiber节点
        let oldFiber = currentFirstChild;
+       //上次不需要移动的放置索引
+       let lastPlacedIndex = 0;
        //下一个老的fiber
        let nextOldFiber = null;
        //新的索引
        let newIdx = 0;
        //先处理节点更新的情况
        for (; oldFiber && newIdx < newChildren.length; newIdx++) {
            //先缓存下下一个老的fiber节点
            nextOldFiber = oldFiber.sibling;
            //更新复用的新fiber
            //判断key和type是否相同，如果相同表示可以复用，newIdx++后和下一个oldFiber比较
            const newFiber = updateSlot(returnFiber, oldFiber, newChildren[newIdx]);
            if (!newFiber) {
                break;
            }
            //如果类型相同，但类型不同，但没有重用现有fiber，因此需要删除现有子级fiber
            if (oldFiber && !(newFiber.alternate)) {
                deleteChild(returnFiber, oldFiber);
            }
            //放置此newFiber到newIdx，就是设置newFiber的index索引
+           lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
            //如果previousNewFiber不存在表示这是第一个fiber
            if (!previousNewFiber) {
                resultingFirstChild = newFiber;
            } else {
                //否则上一个新fiber的sibling等于这个newFiber
                previousNewFiber.sibling = newFiber;
            }
            //让当前的newFiber等于previousNewFiber
            previousNewFiber = newFiber;
            //下一个老fiber
            oldFiber = nextOldFiber;
        }
        //如果新的遍历完了，删除掉所有老的剩下的fiber，直接返回resultingFirstChild
        if (newIdx
            deleteRemainingChildren(returnFiber, oldFiber);
            return resultingFirstChild;
        }
        //如果老fiber完成，新的JSX没有完成
        if (!oldFiber) {
            for (; newIdx < newChildren.length; newIdx++) {
                debugger
```

```
                const newFiber = createChild(returnFiber, newChildren[newIdx]);
+               lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
                if (!previousNewFiber) {
                    resultingFirstChild = newFiber;
                } else {
                    previousNewFiber.sibling = newFiber;
                }
                previousNewFiber = newFiber;
            }
            return resultingFirstChild;
        }
+       //将所有儿子添加到key map以进行快速查找。
+       const existingChildren = mapRemainingChildren(returnFiber, oldFiber);
+       // Keep scanning and use the map to restore deleted items as moves.

+       //继续扫描并使用map在移动时还原已删除的项目。
+       for (; newIdx < newChildren.length; newIdx++) {
+           const newFiber = updateFromMap(existingChildren, returnFiber, newIdx, newChildren[newIdx]);
+           if (newFiber) {
+               if (newFiber.alternate) {
+                   //新fiber正在处理中，但如果存在current
+                   //这意味着我们重用了fiber。我们需要删除它从子列表中删除
+                   //这样我们就不会将其添加到删除列表中
+                   existingChildren.delete(newFiber.key || newIdx);
+               }
+               lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
+               if (!previousNewFiber) {
+                   resultingFirstChild = newFiber;
+               } else {
+                   previousNewFiber.sibling = newFiber;
+               }
+               previousNewFiber = newFiber;
+           }
+       }
+       //已删除上面未使用的所有现有子项
+       //我们需要将它们添加到删除列表中
+       existingChildren.forEach(child => deleteChild(returnFiber, child));
        return resultingFirstChild;
    }
    function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
        const isObject = typeof newChild
        if (isObject) {
            switch (newChild.$typeof) {
                case REACT_ELEMENT_TYPE:
                    return placeSingleChild(
                        reconcileSingleElement(returnFiber, currentFirstChild, newChild)
                    );
                default:
                    break;
            }
        }
        if (Array.isArray(newChild)) {
            return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
        }
    }
    return reconcileChildFibers;
}

export const reconcileChildFibers = ChildReconciler(true);
export const mountChildFibers = ChildReconciler(false);
```

15.4 src\ReactFiberFlags.js

src\ReactFiberFlags.js

```
export const NoFlags = 0b00000000000000000000;//0
export const Placement = 0b00000000000000000010;//2
export const Update = 0b00000000000000000100;//4
+export const PlacementAndUpdate = 0b00000000000000000110;
export const Deletion = 0b00000000000000001000;//8
```

15.5 src\ReactFiberWorkLoop.js

src\ReactFiberWorkLoop.js

```
import { HostRoot, HostComponent } from './ReactWorkTags';
import { createWorkInProgress } from './ReactFiber';
import { beginWork } from './ReactFiberBeginWork';
import { completeWork } from './ReactFiberCompleteWork';
+import { Placement, Update, Deletion, PlacementAndUpdate } from './ReactFiberFlags';
import { commitWork } from './ReactFiberCommitWork';
//正在调度的fiberRoot根节点
let workInProgressRoot = null;
//正在处理的fiber节点
let workInProgress = null;
//最大更新深度
const NESTED_UPDATE_LIMIT = 50;
let nestedUpdateCount = 0;
/**
 * 向上获取HostRoot节点
 * @param {*} sourceFiber 更新来源fiber
 * @returns
 */
function markUpdateLaneFromFiberToRoot(sourceFiber) {
    let node = sourceFiber;
    let parent = node.return;
    //一直向上找父亲，找不到为止
    while (parent) {
        node = parent;
        parent = parent.return;
    }
    //如果找到的是HostRoot就返回FiberRootNode,其实就是容器div#root
    if (node.tag
        return node.stateNode;
```

```
    }
}
/**
 * 向上查找到根节点开始调度更新
 * @param {*} fiber
 */
export function scheduleUpdateOnFiber(fiber) {
    checkForNestedUpdates();
    //向上获取HostRoot节点
    const root = markUpdateLaneFromFiberToRoot(fiber);
    //执行HostRoot上的更新
    performSyncWorkOnRoot(root);
}
function checkForNestedUpdates() {
    if (++nestedUpdateCount > NESTED_UPDATE_LIMIT) {
        throw new Error('Maximum update depth exceeded');
    }
}
/**
 * 开始执行FiberRootNode上的工作
 * @param {*} root  FiberRootNode
 */
function performSyncWorkOnRoot(root) {
    //先赋值给当前正在执行工作的FiberRootNode根节点
    workInProgressRoot = root;
    //创建一个新的处理中的fiber节点
    workInProgress = createWorkInProgress(workInProgressRoot.current);
    workLoopSync();
    commitRoot();
}
function commitRoot(root) {
    //构建成功的新的fiber树
    const finishedWork = workInProgressRoot.current.alternate;
    //当前完成的构建工作等于finishedWork
    workInProgressRoot.finishedWork = finishedWork;
    commitMutationEffects(workInProgressRoot);
    nestedUpdateCount--;
}
function getFlag(flags) {
    switch (flags) {
        case Placement:
            return '添加';
        case Update:
            return '更新';
+       case PlacementAndUpdate:
+           return '移动更新';
        case Deletion:
            return '删除';
    }
}
function commitMutationEffects(root) {
    const finishedWork = root.finishedWork;
    let nextEffect = finishedWork.firstEffect;
    let effectList = '';
    while (nextEffect) {
        effectList += `(${getFlag(nextEffect.flags)}#${nextEffect.type}#${nextEffect.key})=>`;
        const flags = nextEffect.flags;
        const current = nextEffect.alternate;
        if (flags
            commitPlacement(nextEffect);
+       } else if (flags === PlacementAndUpdate) {
+           commitPlacement(nextEffect);
+           nextEffect.flags &= ~Placement;
+           commitWork(current, nextEffect);
+       } else if (flags === Update) {
            commitWork(current, nextEffect);
        } else if (flags
            commitDeletion(nextEffect);
        }
        nextEffect = nextEffect.nextEffect;
    }
    effectList += 'null';
    console.log(effectList);
    root.current = finishedWork;
}
function commitDeletion(fiber) {
    if (!fiber) {
        return;
    }
    let parentStateNode = getParentStateNode(fiber);
    parentStateNode.removeChild(fiber.stateNode);
}
function commitPlacement(nextEffect) {
    let stateNode = nextEffect.stateNode;
    let parentStateNode = getParentStateNode(nextEffect);
    let before = getHostSibling(nextEffect);
    if (before) {
        parentStateNode.insertBefore(stateNode, before);
    } else {
        parentStateNode.appendChild(stateNode);
    }
}
function getHostSibling(fiber) {
    let node = fiber.sibling;
    while (node) {
        if (!(node.flags & Placement)) {
            return node.stateNode;
        }
        node = node.sibling;
    }
    return null;
}
function getParentStateNode(fiber) {
    const parent = fiber.return;
```

```
    do {
        if (parent.tag
            return parent.stateNode;
        } else if (parent.tag
            return parent.stateNode.containerInfo
        }
        parent = parent.return;
    } while (parent);
}
function workLoopSync() {
    while (workInProgress) {
        performUnitOfWork(workInProgress);
    }
}
/**
 * 执行单个工作单元
 * @param {*} unitOfWork 单个fiber
 */
function performUnitOfWork(unitOfWork) {
    //获取当前fiber的alternate
    const current = unitOfWork.alternate;
    //开始构建此fiber的子fiter链表
    let next = beginWork(current, unitOfWork);
    //更新属性
    unitOfWork.memoizedProps = unitOfWork.pendingProps;
    //如果有子fiber，就继续执行
    if (next) {
        workInProgress = next;
    } else {
        //如果没有子fiber，就完成当前的fiber
        completeUnitOfWork(unitOfWork);
    }
}

function completeUnitOfWork(unitOfWork) {
    //尝试完成当前的工作单元，然后移动到下一个弟弟
    //如果没有下一个弟弟，返回到父Fiber
    let completedWork = unitOfWork;
    do {
        const current = completedWork.alternate;
        const returnFiber = completedWork.return;
        completeWork(current, completedWork);
        collectEffectList(returnFiber, completedWork)
        const siblingFiber = completedWork.sibling;
        if (siblingFiber) {
            //如果此 returnFiber 中还有更多工作要做，请执行下一步
            workInProgress = siblingFiber;
            return;
        }
        //否则，返回父级
        completedWork = returnFiber;
        //更新我们正在处理的下一件事
        workInProgress = completedWork;
    } while (completedWork);
}
function collectEffectList(returnFiber, completedWork) {
    if (returnFiber) {
        if (!returnFiber.firstEffect) {
            returnFiber.firstEffect = completedWork.firstEffect;
        }
        if (completedWork.lastEffect) {
            if (returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = completedWork.firstEffect;
            }
            returnFiber.lastEffect = completedWork.lastEffect;
        }
        //如果这个fiber有副作用，我们会在孩子们的副使用后面添加它自己的副作用
        const flags = completedWork.flags;
        if (flags) {
            if (returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = completedWork;
            } else {
                returnFiber.firstEffect = completedWork;
            }
            returnFiber.lastEffect = completedWork;
        }
    }
}
```