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=166 sentences=753, words=7660

1. React面试题

- 为什么不能在条件和循环里使用Hooks?
- 为什么不能在函数组件外部使用Hooks?
- React Hooks的状态保存在了哪里?
- 为什么传入二次相同的状态，函数组件不更新?
- 函数组件的useState和类组件的setState有什么区别?

2.前置知识

2.1 位操作

2.1.1 按位与(&)

- 两个输入数的同一位都为1才为1

2.1.2 按位或(|)

- 两个输入数的同一位只要有一个为1就是1

2.1.3 位操作

```
const NoFlags =    0b000;
const HasEffect =  0b001;
const Layout =     0b010;
const Passive =    0b100;

let layoutTag = HasEffect|Layout;
if((layoutTag & Layout) !== NoFlags){
    console.log('useLayoutEffect');
}
let tag = HasEffect|Passive;
if((tag & Passive) !== NoFlags){
    console.log('useEffect');
}
```

2.2 Fiber

2.2.1 Fiber是一种数据结构

- React目前的做法是使用链表, 每个 VirtualDOM节点内部表示为一个 Fiber

```
let virtualDOM = (
    <div key="A">
        <div key="B1">B1div>
        <div key="B2">B2div>
    div>
)
```

2.2.2 Fiber树

- current fiber树 当渲染完成后会产生一个current Fiber树
- workInProgress fiber树 在render阶段，会基于current树创建新的workInProgress fiber树,更新完成后会把workInProgress fiber树赋给current fiber树
- workInProgress fiber树的每个节点会有一个alternate指针指向current树对应的fiber节点

2.2.3 Fiber是一个执行单元

- Fiber是一个执行单元,每次执行完一个执行单元, React 就会检查现在还剩多少时间，如果没有时间就将控制权让出去

2.3 循环链表

- 循环链表是另一种形式的链式存储结构
- 它的特点是表中最后一个结点的指针域指向头结点，整个链表形成一个环

```
function dispatchAction(queue, action) {
    const update = { action, next: null };
    const pending = queue.pending;
    if (pending === null) {
        update.next = update;
    } else {
        update.next = pending.next;
        pending.next = update;
    }
    queue.pending = update;
}
let queue = { pending: null };
dispatchAction(queue, 'action1');
dispatchAction(queue, 'action2');
dispatchAction(queue, 'action3');
const pendingQueue = queue.pending;
if (pendingQueue !== null) {
    const first = pendingQueue.next;
    let update = first;
    do {
        const action = update.action;
        console.log(action);
        update = update.next;
    } while (update !== null && update !== first);
}
```

3.使用useReducer

3.1 renderWithHooks

3.2 hooks更新

3.3 src\index.js

src\index.js

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';
const reducer = (state, action) => {
  if (action.type === 'add')
    return state + 1;
  else
    return state;
}
function Counter() {
  const [number, setNumber] = useReducer(reducer, 0);
  return (
    <div onClick={() => {setNumber({ type: 'add' })}}>{number}div>
  )
}
ReactDOM.render(<Counter/>,document.getElementById('root'));
```

4.实现useReducer

4.1 src\index.js

src\index.js

```
import * as React from 'react';
import { IndeterminateComponent } from './ReactWorkTags';
import {render} from './ReactFiberWorkLoop';
import {useReducer} from './ReactFiberHooks'
const reducer = (state, action) => {
  if (action.type === 'add')
    return state + 1;
  else
    return state;
}
function Counter() {
  const [number, setNumber] = useReducer(reducer, 0);
  return (
    <div onClick={() => {setNumber({ type: 'add' })}}>{number}div>
  )
}
let workInProgress = {
  tag:IndeterminateComponent,
  type: Counter,
  alternate:null
}
render(workInProgress);
```

src\ReactWorkTags.js

```
export const FunctionComponent = 0;
export const ClassComponent = 1;
export const IndeterminateComponent = 2;
export const HostRoot = 3;
export const HostComponent = 5;
```

4.3 ReactFiberWorkLoop.js

src\ReactFiberWorkLoop.js

```
import {beginWork} from './ReactFiberBeginWork';
let workInProgress;

export function workLoop(){
    while(workInProgress){
        workInProgress = performUnitOfWork(workInProgress);
    }
}
export function performUnitOfWork(unitOfWork){
    let current = unitOfWork.alternate;
    return beginWork(current,unitOfWork);
}

export function render(fiber){
    workInProgress=fiber;
    workLoop();
}
```

4.4 ReactFiberBeginWork.js

ReactFiberBeginWork.js

```
import { IndeterminateComponent,FunctionComponent,HostComponent} from './ReactWorkTags';
import { renderWithHooks } from './ReactFiberHooks';
export function beginWork(current, workInProgress) {
    switch (workInProgress.tag) {
        case IndeterminateComponent: {
            return mountIndeterminateComponent(
                current,
                workInProgress,
                workInProgress.type,
            );
        }
        default:
            break;
    }
}
export function mountIndeterminateComponent(_current, workInProgress, Component) {
    let value = renderWithHooks(_current,workInProgress,Component);
    window.counter = value;
    console.log('Counter的render结果 ', value.props.children);
    workInProgress.tag = FunctionComponent;
    reconcileChildren(null, workInProgress, value);
    return workInProgress.child;
}
function reconcileChildren(current, workInProgress, nextChildren) {
    let childFiber = {
        tag: HostComponent,
        type: nextChildren.type
    };
    workInProgress.child = childFiber;
}
```

4.5 ReactFiberHooks.js

src\ReactFiberHooks.js

```
let ReactCurrentDispatcher = {
    current: null
}
let currentlyRenderingFiber = null;
let workInProgressHook = null;

const HooksDispatcherOnMount = {
    useReducer: mountReducer
}

export function useReducer(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useReducer(reducer, initialArg);
}

export function renderWithHooks(_current, workInProgress, Component) {
    currentlyRenderingFiber = workInProgress;
    ReactCurrentDispatcher.current = HooksDispatcherOnMount;
    let children = Component();
    window.counter = children;
    currentlyRenderingFiber = null;
    return children;
}
export function mountReducer(reducer, initialArg) {
    const hook = mountWorkInProgressHook();
    let initialState = initialArg;
    hook.memoizedState = initialState;
    const queue = (hook.queue = {pending: null, lastRenderedReducer: reducer, lastRenderedState: initialState});
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}

export function mountWorkInProgressHook() {
    const hook = {
        memoizedState: null,
        queue: null,
        next: null,
    };
    if (workInProgressHook === null) {
        currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
    } else {
        workInProgressHook = workInProgressHook.next = hook;
    }
    return workInProgressHook;
}

export function dispatchAction(fiber, queue, action) {
   console.log('dispatchAction');
}
```

5.useReducer更新

5.1 ReactFiberHooks.js

src\ReactFiberHooks.js

```
+import { scheduleUpdateOnFiber } from './ReactFiberWorkLoop';
let ReactCurrentDispatcher = {
    current: null
}
let currentlyRenderingFiber = null;
let workInProgressHook = null;
+let currentHook = null;
const HooksDispatcherOnMount = {
    useReducer: mountReducer
}
+const HooksDispatcherOnUpdate = {
+    useReducer: updateReducer
+}
export function useReducer(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useReducer(reducer, initialArg);
}

export function renderWithHooks(_current, workInProgress, Component) {
    currentlyRenderingFiber = workInProgress;
+   workInProgress.memoizedState = null;
-   ReactCurrentDispatcher.current = HooksDispatcherOnMount;
+   if (_current !== null) {
+       ReactCurrentDispatcher.current = HooksDispatcherOnUpdate;
+   } else {
+       ReactCurrentDispatcher.current = HooksDispatcherOnMount;
+   }
    let children = Component();
    window.counter = children;
    currentlyRenderingFiber = null;
+   currentHook = null;
+   workInProgressHook = null;
    return children;
}
+function updateReducer(reducer) {
+    const hook = updateWorkInProgressHook();
+    const queue = hook.queue;
+    queue.lastRenderedReducer = reducer;
+    const current = currentHook;
+    const pendingQueue = queue.pending;
+    if (pendingQueue !== null) {
+        const first = pendingQueue.next;
+        let newState = current.memoizedState;
+        let update = first;
+        do {
+            const action = update.action;
+            newState = reducer(newState, action);
+            update = update.next;
+        } while (update !== null && update !== first);
```

```
+        queue.pending = null;
+        hook.memoizedState = newState;
+        queue.lastRenderedState = newState;
+    }
+    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
+    return [hook.memoizedState, dispatch];
+}
+function updateWorkInProgressHook() {
+    let nextCurrentHook;
+    if (currentHook === null) {
+        const current = currentlyRenderingFiber.alternate;
+        nextCurrentHook = current.memoizedState;
+    } else {
+        nextCurrentHook = currentHook.next;
+    }
+    currentHook = nextCurrentHook;
+    const newHook = {
+        memoizedState: currentHook.memoizedState,
+        queue: currentHook.queue,
+        next: null,
+    };
+    if (workInProgressHook === null) {
+        currentlyRenderingFiber.memoizedState = workInProgressHook = newHook;
+    } else {
+        workInProgressHook = workInProgressHook.next = newHook;
+    }
+    return workInProgressHook;
+}
export function mountReducer(reducer, initialArg) {
    const hook = mountWorkInProgressHook();
    let initialState = initialArg;
    hook.memoizedState = initialState;
    const queue = (hook.queue = {pending: null, lastRenderedReducer: reducer, lastRenderedState: initialState});
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}

export function mountWorkInProgressHook() {
    const hook = {
        memoizedState: null,
        queue: null,
        next: null,
    };
    if (workInProgressHook
        currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
    } else {
        workInProgressHook = workInProgressHook.next = hook;
    }
    return workInProgressHook;
}

export function dispatchAction(fiber, queue, action) {
+  const update = { action, next: null };
+  const pending = queue.pending;
+  if (pending === null) {
+      update.next = update;
+  } else {
+     update.next = pending.next;
+     pending.next = update;
+  }
+  queue.pending = update;
+  const lastRenderedReducer = queue.lastRenderedReducer;
+  const currentState = queue.lastRenderedState;
+  const eagerState = lastRenderedReducer(currentState, action);
+  if (Object.is(eagerState, currentState)) {
+     return
+  }
+  scheduleUpdateOnFiber(fiber);
}
```

5.2 ReactFiberWorkLoop.js

src\ReactFiberWorkLoop.js

```
import { beginWork } from './ReactFiberBeginWork';
let workInProgress;

export function workLoop() {
    while (workInProgress) {
        workInProgress = performUnitOfWork(workInProgress);
    }
}
export function performUnitOfWork(unitOfWork) {
    let current = unitOfWork.alternate;
    return beginWork(current, unitOfWork);
}
+export function scheduleUpdateOnFiber(fiber) {
+    let newFiber = {
+        ...fiber,
+        alternate: fiber
+    }
+    workInProgress = newFiber;
+    workLoop();
}
export function render(fiber) {
    workInProgress = fiber;
    workLoop();
}
```

5.3 ReactFiberBeginWork.js

src\ReactFiberBeginWork.js

```
import { IndeterminateComponent, FunctionComponent, HostComponent } from './ReactWorkTags';
import { renderWithHooks } from './ReactFiberHooks';
export function beginWork(current, workInProgress) {
+    if (current !== null) {
+        switch (workInProgress.tag) {
+            case FunctionComponent: {
+                const Component = workInProgress.type;
+                return updateFunctionComponent(
+                    current,
+                    workInProgress,
+                    Component,
+                );
+            }
+            default:
+                break;
+        }
+    } else {
        switch (workInProgress.tag) {
            case IndeterminateComponent: {
                return mountIndeterminateComponent(
                    current,
                    workInProgress,
                    workInProgress.type,
                );
            }
            default:
                break;
        }
    }
}
+function updateFunctionComponent(current,workInProgress,Component){
+    let nextChildren = renderWithHooks(current,workInProgress,Component);
+    window.counter = nextChildren;
+    console.log('Counter的render结果 ', nextChildren.props.children);
+    reconcileChildren(current, workInProgress, nextChildren);
+    return workInProgress.child;
+}
export function mountIndeterminateComponent(_current, workInProgress, Component) {
    let value = renderWithHooks(_current, workInProgress, Component);
    window.counter = value;
    console.log('Counter的render结果 ', value.props.children);
    workInProgress.tag = FunctionComponent;
    reconcileChildren(null, workInProgress, value);
    return workInProgress.child;
}
function reconcileChildren(current, workInProgress, nextChildren) {
    let childFiber = {
        tag: HostComponent,
        type: nextChildren.type
    };
    workInProgress.child = childFiber;
}
```

6.useState

6.1 src\index.js

src\index.js

```
import * as React from 'react';
import { IndeterminateComponent } from './ReactWorkTags';
import {render} from './ReactFiberWorkLoop';
+import {useReducer,useState} from './ReactFiberHooks'

function Counter() {
+  const [number, setNumber] = useState(0);
  return (
+    {setNumber(number+1)}}>{number}
  )
}
let workInProgress = {
  tag:IndeterminateComponent,
  type: Counter,
  alternate:null
}
render(workInProgress);
```

6.2 src\ReactFiberHooks.js

src\ReactFiberHooks.js

```
import { scheduleUpdateOnFiber } from './ReactFiberWorkLoop';
let ReactCurrentDispatcher = {
    current: null
}
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;
const HooksDispatcherOnMount = {
    useReducer: mountReducer,
+   useState: mountState
}
const HooksDispatcherOnUpdate = {
    useReducer: updateReducer,
+   useState: updateState
}
+function mountState(initialState) {
+    const hook = mountWorkInProgressHook();
+    hook.memoizedState = initialState;
+    const queue = (hook.queue = { pending: null,lastRenderedReducer: basicStateReducer, lastRenderedState: initialState });
+    const dispatch = dispatchAction.bind(null, currentlyRenderingFiber, queue)
+    return [hook.memoizedState, dispatch];
+}
+
```

```
+function basicStateReducer(state, action) {
+    return typeof action === 'function' ? action(state) : action;
+}
+
+function updateState(initialState) {
+    return updateReducer(basicStateReducer, initialState);
+}
+
+export function useState(initialState) {
+    return ReactCurrentDispatcher.current.useState(initialState);
+}
export function useReducer(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useReducer(reducer, initialArg);
}

export function renderWithHooks(_current, workInProgress, Component) {
    currentlyRenderingFiber = workInProgress;
    workInProgress.memoizedState = null;
    if (_current !== null) {
        ReactCurrentDispatcher.current = HooksDispatcherOnUpdate;
    } else {
        ReactCurrentDispatcher.current = HooksDispatcherOnMount;
    }
    let children = Component();
    window.counter = children;
    currentlyRenderingFiber = null;
    currentHook = null;
    workInProgressHook = null;
    return children;
}
function updateReducer(reducer) {
    const hook = updateWorkInProgressHook();
    const queue = hook.queue;
    queue.lastRenderedReducer = reducer;
    const current = currentHook;
    const pendingQueue = queue.pending;
    if (pendingQueue !== null) {
        const first = pendingQueue.next;
        let newState = current.memoizedState;
        let update = first;
        do {
            const action = update.action;
            newState = reducer(newState, action);
            update = update.next;
        } while (update !== null && update !== first);
        queue.pending = null;
        hook.memoizedState = newState;
        queue.lastRenderedState = newState;
    }
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}
function updateWorkInProgressHook() {
    let nextCurrentHook;
    if (currentHook
        const current = currentlyRenderingFiber.alternate;
        nextCurrentHook = current.memoizedState;
    } else {
        nextCurrentHook = currentHook.next;
    }
    currentHook = nextCurrentHook;

    const newHook = {
        memoizedState: currentHook.memoizedState,
        queue: currentHook.queue,
        next: null,
    };
    if (workInProgressHook
        currentlyRenderingFiber.memoizedState = workInProgressHook = newHook;
    } else {
        workInProgressHook = workInProgressHook.next = newHook;
    }
    return workInProgressHook;
}
export function mountReducer(reducer, initialArg) {
    const hook = mountWorkInProgressHook();
    let initialState = initialArg;
    hook.memoizedState = initialState;
    const queue = (hook.queue = {pending: null, lastRenderedReducer: reducer, lastRenderedState: initialState});
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}

export function mountWorkInProgressHook() {
    const hook = {
        memoizedState: null,
        queue: null,
        next: null,
    };
    if (workInProgressHook
        currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
    } else {
        workInProgressHook = workInProgressHook.next = hook;
    }
    return workInProgressHook;
}

export function dispatchAction(fiber, queue, action) {
   const update = { action, next: null };
   const pending = queue.pending;
   if (pending
       update.next = update;
   } else {
       update.next = pending.next;
```

```
        pending.next = update;
    }
    queue.pending = update;
const lastRenderedReducer = queue.lastRenderedReducer;
    const currentState = queue.lastRenderedState;
    const eagerState = lastRenderedReducer(currentState, action);
    if (Object.is(eagerState, currentState)) {
        return
    }
    scheduleUpdateOnFiber(fiber);
}
```

7.useEffect

- React工作的三个阶段
    - scheduler(调度) 确定最高优的任务并进入 reconciler
    - reconciler(协调) 找出变化的内容
    - renderer(渲染) 把变化的内容更新到DOM上
        - beforeMutation 更新DOM前
        - mutation 更新DOM
        - layout 更新DOM后

类型 fiberFlags hookFlags useEffect UpdateEffect或PassiveEffect HookHasEffect或HookPassive useLayoutEffect UpdateEffect HookHasEffect或HookLayout 阶段 useEffect useLayoutEffect commitBeforeMutationEffects 调度flushPassiveEffects 无 commitMutationEffects 无 执行destroy commitLayoutEffects 注册destroy、create 执行create commit完成后 执行flushPassiveEffects 无

7.1 src\index.js

src\index.js

```
import * as React from 'react';
import { IndeterminateComponent } from './ReactWorkTags';
import {render} from './ReactFiberWorkLoop';
import {useReducer,useState,useEffect} from './ReactFiberHooks'

function Counter() {
  const [number, setNumber] = useState(0);
+ useEffect(()=>{
+   console.log('useEffect1');
+   return ()=>{
+     console.log('destroy useEffect1');
+   }
+ });
+ useEffect(()=>{
+   console.log('useEffect2');
+   return ()=>{
+     console.log('destroy useEffect2');
+   }
+ });
+ useEffect(()=>{
+   console.log('useEffect3');
+   return ()=>{
+     console.log('destroy useEffect3');
+   }
  });
  return (
     {setNumber(number+1)}}>{number}
  )
}
let workInProgress = {
  tag:IndeterminateComponent,
  type: Counter,
  alternate:null,
  updateQueue:null
}
render(workInProgress);
```

7.2 ReactFiberFlags.js

- NoFlags 没有任何副作用
- PerformedWork 有工作要做
- Update 有 useLayoutEffect对应副作用
- Passive 有 useEffect对应的副作用

src\ReactFiberFlags.js

```
export const NoFlags =  0b00000000000000000;
export const PerformedWork =  0b00000000000000001;
export const Update =  0b00000000000000100;
export const Passive =  0b00000000100000000;
```

- NoFlags
- HasEffect 有effect
- Layout useLayoutEffect创建的effect
- Passive useEffect创建的effect

src\ReactHookEffectTags.js

```
export const NoFlags =  0b000;
export const HasEffect =  0b001;
export const Layout =  0b010;
export const Passive =  0b100;
```

7.4 ReactFiberHooks.js

src\ReactFiberHooks.js

```
import { scheduleUpdateOnFiber } from './ReactFiberWorkLoop';
+import { Update as UpdateEffect, Passive as PassiveEffect } from './ReactFiberFlags';
+import { HasEffect as HookHasEffect, Passive as HookPassive} from './ReactHookEffectTags';
let ReactCurrentDispatcher = {
    current: null
```

```
}
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;
const HooksDispatcherOnMount = {
    useReducer: mountReducer,
    useState: mountState,
+   useEffect: mountEffect
}
const HooksDispatcherOnUpdate = {
    useReducer: updateReducer,
    useState: updateState,
+   useEffect: updateEffect
}
+export function mountEffect(create, deps) {
+    return mountEffectImpl(
+        UpdateEffect | PassiveEffect,
+        HookPassive,
+        create,
+        deps
+    );
+}
+function mountEffectImpl(fiberFlags, hookFlags, create, deps) {
+    const hook = mountWorkInProgressHook();
+    const nextDeps = deps === undefined ? null : deps;
+    currentlyRenderingFiber.flags |= fiberFlags;
+    hook.memoizedState = pushEffect(
+        HookHasEffect | hookFlags,
+        create,
+        undefined,
+        nextDeps,
+    );
+}
+function pushEffect(tag, create, destroy, deps) {
+    const effect = { tag, create, destroy, deps, next: null};
+    let componentUpdateQueue = (currentlyRenderingFiber.updateQueue);
+    if (componentUpdateQueue === null) {
+        componentUpdateQueue = createFunctionComponentUpdateQueue();
+        currentlyRenderingFiber.updateQueue = componentUpdateQueue;
+        componentUpdateQueue.lastEffect = effect.next = effect;
+    } else {
+        const lastEffect = componentUpdateQueue.lastEffect;
+        if (lastEffect === null) {
+            componentUpdateQueue.lastEffect = effect.next = effect;
+        } else {
+            const firstEffect = lastEffect.next;
+            lastEffect.next = effect;
+            effect.next = firstEffect;
+            componentUpdateQueue.lastEffect = effect;
+        }
+    }
+    return effect;
+}
+function createFunctionComponentUpdateQueue() {
+    return {
+        lastEffect: null,
+    };
+}
+export function updateEffect(create, deps,) {
+    return updateEffectImpl(
+        PassiveEffect,
+        HookPassive,
+        create,
+        deps,
+    );
+}
+function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
+    const hook = updateWorkInProgressHook();
+    const nextDeps = deps === undefined ? null : deps;
+    let destroy = undefined;
+    if (currentHook !== null) {
+        const prevEffect = currentHook.memoizedState;
+        destroy = prevEffect.destroy;
+        if (nextDeps !== null) {
+            const prevDeps = prevEffect.deps;
+            if (areHookInputsEqual(nextDeps, prevDeps)) {
+                pushEffect(hookFlags, create, destroy, nextDeps);
+                return;
+            }
+        }
+    }
+    currentlyRenderingFiber.flags |= fiberFlags;
+    hook.memoizedState = pushEffect(HookHasEffect | hookFlags, create, destroy, nextDeps)
+}
+function areHookInputsEqual(nextDeps, prevDeps) {
+    if (prevDeps === null) {
+        return false;
+    }
+    for (let i = 0; i < prevDeps.length && i < nextDeps.length; i++) {
+        if (Object.is(nextDeps[i], prevDeps[i])) {
+            continue;
+        }
+        return false;
+    }
+    return true;
+}
+function mountState(initialState) {
+    const hook = mountWorkInProgressHook();
+    hook.memoizedState = initialState;
+    const queue = (hook.queue = { pending: null,lastRenderedReducer: basicStateReducer, lastRenderedState: initialState });
+    const dispatch = dispatchAction.bind(null, currentlyRenderingFiber, queue)
+    return [hook.memoizedState, dispatch];
+}
+export function useEffect(reducer, initialArg) {
```

```
+    return ReactCurrentDispatcher.current.useEffect(reducer, initialArg);
+}
function basicStateReducer(state, action) {
    return typeof action
}

function updateState(initialState) {
    return updateReducer(basicStateReducer, initialState);
}

export function useState(initialState) {
    return ReactCurrentDispatcher.current.useState(initialState);
}
export function useReducer(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useReducer(reducer, initialArg);
}

export function renderWithHooks(_current, workInProgress, Component) {
    currentlyRenderingFiber = workInProgress;
    workInProgress.memoizedState = null;
+   workInProgress.updateQueue = null;
    if (_current !== null) {
        ReactCurrentDispatcher.current = HooksDispatcherOnUpdate;
    } else {
        ReactCurrentDispatcher.current = HooksDispatcherOnMount;
    }
    let children = Component();
    window.counter = children;
    currentlyRenderingFiber = null;
    currentHook = null;
    workInProgressHook = null;
    return children;
}
function updateReducer(reducer) {
    const hook = updateWorkInProgressHook();
    const queue = hook.queue;
    queue.lastRenderedReducer = reducer;
    const current = currentHook;;
    const pendingQueue = queue.pending;
    if (pendingQueue !== null) {
        const first = pendingQueue.next;
        let newState = current.memoizedState;
        let update = first;
        do {
            const action = update.action;
            newState = reducer(newState, action);
            update = update.next;
        } while (update !== null && update !== first);
        queue.pending = null;
        hook.memoizedState = newState;
        queue.lastRenderedState = newState;
    }
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}
function updateWorkInProgressHook() {
    let nextCurrentHook;
    if (currentHook
        const current = currentlyRenderingFiber.alternate;
        nextCurrentHook = current.memoizedState;
    } else {
        nextCurrentHook = currentHook.next;
    }
    currentHook = nextCurrentHook;

    const newHook = {
        memoizedState: currentHook.memoizedState,
        queue: currentHook.queue,
        next: null,
    };
    if (workInProgressHook
        currentlyRenderingFiber.memoizedState = workInProgressHook = newHook;
    } else {
        workInProgressHook = workInProgressHook.next = newHook;
    }
    return workInProgressHook;
}
export function mountReducer(reducer, initialArg) {
    const hook = mountWorkInProgressHook();
    let initialState = initialArg;
    hook.memoizedState = initialState;
    const queue = (hook.queue = {pending: null, lastRenderedReducer: reducer, lastRenderedState: initialState});
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}

export function mountWorkInProgressHook() {
    const hook = {
        memoizedState: null,
        queue: null,
        next: null,
    };
    if (workInProgressHook
        currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
    } else {
        workInProgressHook = workInProgressHook.next = hook;
    }
    return workInProgressHook;
}

export function dispatchAction(fiber, queue, action) {
   const update = { action, next: null };
   const pending = queue.pending;
   if (pending
```

```
        update.next = update;
    } else {
        update.next = pending.next;
        pending.next = update;
    }
    queue.pending = update;
       const lastRenderedReducer = queue.lastRenderedReducer;
    const currentState = queue.lastRenderedState;
    const eagerState = lastRenderedReducer(currentState, action);
    if (Object.is(eagerState, currentState)) {
        return
    }
    scheduleUpdateOnFiber(fiber);
}
```

7.5 ReactFiberWorkLoop.js

src\ReactFiberWorkLoop.js

```
import { beginWork } from './ReactFiberBeginWork';
+import {Update, Passive, NoFlags } from './ReactFiberFlags';
+import { commitLifeCycles as commitLayoutEffectOnFiber } from './ReactFiberCommitWork';
let workInProgress;
+let finishedWork = null
+let pendingPassiveHookEffectsMount = [];
+let pendingPassiveHookEffectsUnmount = [];
export function workLoop() {
    while (workInProgress) {
        workInProgress = performUnitOfWork(workInProgress);
    }
+   commitRoot();
}
+function commitRoot() {
+    if (!finishedWork) return;
+    commitBeforeMutationEffects();
+    commitMutationEffects();
+    commitLayoutEffects();
+}
+export function enqueuePendingPassiveHookEffectMount(fiber, effect,) {
+    pendingPassiveHookEffectsMount.push(effect, fiber);
+}
+
+export function enqueuePendingPassiveHookEffectUnmount(fiber, effect) {
+    pendingPassiveHookEffectsUnmount.push(effect, fiber);
+}
+function commitLayoutEffects() {
+    const flags = finishedWork.flags;
+    if (flags & Update) {
+        const current = finishedWork.alternate;
+        commitLayoutEffectOnFiber(finishedWork, current, finishedWork);
+    }
+}
+function commitMutationEffects() {
+
+}
+
+function commitBeforeMutationEffects() {
+    const flags = finishedWork.flags;
+    if ((flags & Passive) !== NoFlags) {
+        setTimeout(flushPassiveEffects);
+    }
+}
+function flushPassiveEffects() {
+    const unmountEffects = pendingPassiveHookEffectsUnmount;
+    pendingPassiveHookEffectsUnmount = [];
+    for (let i = 0; i < unmountEffects.length; i += 2) {
+        const effect = unmountEffects[i];
+        const destroy = effect.destroy;
+        effect.destroy = undefined;
+        if (typeof destroy === 'function') {
+            destroy();
+        }
+    }
+
+    const mountEffects = pendingPassiveHookEffectsMount;
+    pendingPassiveHookEffectsMount = [];
+    for (let i = 0; i < mountEffects.length; i += 2) {
+        const effect = mountEffects[i]
+        const create = effect.create;
+        effect.destroy = create();
+    }
+}
export function performUnitOfWork(unitOfWork) {
    let current = unitOfWork.alternate;
    return beginWork(current, unitOfWork);
}
export function scheduleUpdateOnFiber(fiber) {
    let newFiber = {
        ...fiber,
        alternate: fiber
    }
+   finishedWork = workInProgress = newFiber;
    workLoop();
}
export function render(fiber) {
+   finishedWork = workInProgress = fiber;
    workLoop();
}
```

7.6 ReactFiberCommitWork.js

src\ReactFiberCommitWork.js

```
import { FunctionComponent } from './ReactWorkTags';
import {
    Layout as HookLayout, HasEffect as HookHasEffect,
    Passive as HookPassive, NoFlags as NoHookEffect
} from './ReactHookEffectTags';
import {
    enqueuePendingPassiveHookEffectMount,
    enqueuePendingPassiveHookEffectUnmount,
} from './ReactFiberWorkLoop';
export function commitLifeCycles(finishedRoot, current, finishedWork) {
    switch (finishedWork.tag) {
        case FunctionComponent:
            schedulePassiveEffects(finishedWork);
            break;
        default:
            break;
    }
}
function schedulePassiveEffects(finishedWork) {
    const updateQueue = finishedWork.updateQueue;
    const lastEffect = updateQueue !== null ? updateQueue.lastEffect : null;
    if (lastEffect !== null) {
        const firstEffect = lastEffect.next;
        let effect = firstEffect;
        do {
            const { next, tag } = effect;
            if ((tag & HookPassive) !== NoHookEffect && (tag & HookHasEffect) !== NoHookEffect) {
                enqueuePendingPassiveHookEffectUnmount(finishedWork, effect);
                enqueuePendingPassiveHookEffectMount(finishedWork, effect);
            }
            effect = next;
        } while (effect !== firstEffect);
    }
}
```

8.useLayoutEffect

8.1.src\index.js

```
import * as React from 'react';
import { IndeterminateComponent } from './ReactWorkTags';
import {render} from './ReactFiberWorkLoop';
+import {useReducer,useState,useEffect,useLayoutEffect} from './ReactFiberHooks'

function Counter() {
  const [number, setNumber] = useState(0);
  useEffect(()=>{
    console.log('useEffect1');
    return ()=>{
      console.log('destroy useEffect1');
    }
  });
+ useLayoutEffect(()=>{
+   console.log('LayoutEffect2');
+   return ()=>{
+     console.log('destroy LayoutEffect2');
+   }
+ });
  useEffect(()=>{
    console.log('useEffect3');
    return ()=>{
      console.log('destroy useEffect3');
    }
  });
  return (
     {setNumber(number+1)}}>{number}
  )
}
let workInProgress = {
  tag:IndeterminateComponent,
  type: Counter,
  alternate:null,
  updateQueue:null
}
render(workInProgress);
```

8.2 src\ReactFiberHooks.js

src\ReactFiberHooks.js

```
import { scheduleUpdateOnFiber } from './ReactFiberWorkLoop';
import { Update as UpdateEffect, Passive as PassiveEffect } from './ReactFiberFlags';
+import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout} from './ReactHookEffectTags';
let ReactCurrentDispatcher = {
    current: null
}
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;
const HooksDispatcherOnMount = {
    useReducer: mountReducer,
    useState: mountState,
    useEffect: mountEffect,
+   useLayoutEffect: mountLayoutEffect,
}
const HooksDispatcherOnUpdate = {
    useReducer: updateReducer,
    useState: updateState,
    useEffect: updateEffect,
+   useLayoutEffect: updateLayoutEffect,
}
+export function useLayoutEffect(reducer, initialArg) {
```

```
+    return ReactCurrentDispatcher.current.useLayoutEffect(reducer, initialArg);
+}
+function updateLayoutEffect(create, deps,) {
+    return updateEffectImpl(UpdateEffect, HookLayout, create, deps);
+}
+function mountLayoutEffect(create, deps,) {
+    return mountEffectImpl(UpdateEffect, HookLayout, create, deps);
+}
export function mountEffect(create, deps) {
    return mountEffectImpl(
        UpdateEffect | PassiveEffect,
        HookPassive,
        create,
        deps
    );
}
function mountEffectImpl(fiberFlags, hookFlags, create, deps) {
    const hook = mountWorkInProgressHook();
    const nextDeps = deps
    currentlyRenderingFiber.flags |= fiberFlags;
    hook.memoizedState = pushEffect(
        HookHasEffect | hookFlags,
        create,
        undefined,
        nextDeps,
    );
}
function pushEffect(tag, create, destroy, deps) {
    const effect = { tag, create, destroy, deps, next: null};
    let componentUpdateQueue = (currentlyRenderingFiber.updateQueue);
    if (componentUpdateQueue
        componentUpdateQueue = createFunctionComponentUpdateQueue();
        currentlyRenderingFiber.updateQueue = componentUpdateQueue;
        componentUpdateQueue.lastEffect = effect.next = effect;
    } else {
        const lastEffect = componentUpdateQueue.lastEffect;
        if (lastEffect
            componentUpdateQueue.lastEffect = effect.next = effect;
        } else {
            const firstEffect = lastEffect.next;
            lastEffect.next = effect;
            effect.next = firstEffect;
            componentUpdateQueue.lastEffect = effect;
        }
    }
    return effect;
}
function createFunctionComponentUpdateQueue() {
    return {
        lastEffect: null,
    };
}
export function updateEffect(create, deps,) {
    return updateEffectImpl(
        PassiveEffect,
        HookPassive,
        create,
        deps,
    );
}
function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
    const hook = updateWorkInProgressHook();
    const nextDeps = deps
    let destroy = undefined;
    if (currentHook !== null) {
        const prevEffect = currentHook.memoizedState;
        destroy = prevEffect.destroy;
        if (nextDeps !== null) {
            const prevDeps = prevEffect.deps;
            if (areHookInputsEqual(nextDeps, prevDeps)) {
                pushEffect(hookFlags, create, destroy, nextDeps);
                return;
            }
        }
    }
    currentlyRenderingFiber.flags |= fiberFlags;
    hook.memoizedState = pushEffect(HookHasEffect | hookFlags, create, destroy, nextDeps)
}
function areHookInputsEqual(nextDeps, prevDeps) {
    if (prevDeps
        return false;
    }
    for (let i = 0; i < prevDeps.length && i < nextDeps.length; i++) {
        if (Object.is(nextDeps[i], prevDeps[i])) {
            continue;
        }
        return false;
    }
    return true;
}
function mountState(initialState) {
    const hook = mountWorkInProgressHook();
    hook.memoizedState = initialState;
    const queue = (hook.queue = { pending: null,lastRenderedReducer: basicStateReducer, lastRenderedState: initialState });
    const dispatch = dispatchAction.bind(null, currentlyRenderingFiber, queue)
    return [hook.memoizedState, dispatch];
}

function basicStateReducer(state, action) {
    return typeof action
}

function updateState(initialState) {
    return updateReducer(basicStateReducer, initialState);
```

```
}
export function useEffect(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useEffect(reducer, initialArg);
}
export function useState(initialState) {
    return ReactCurrentDispatcher.current.useState(initialState);
}
export function useReducer(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useReducer(reducer, initialArg);
}

export function renderWithHooks(_current, workInProgress, Component) {
    currentlyRenderingFiber = workInProgress;
    workInProgress.memoizedState = null;
    workInProgress.updateQueue = null;
    if (_current !== null) {
        ReactCurrentDispatcher.current = HooksDispatcherOnUpdate;
    } else {
        ReactCurrentDispatcher.current = HooksDispatcherOnMount;
    }
    let children = Component();
    window.counter = children;
    currentlyRenderingFiber = null;
    currentHook = null;
    workInProgressHook = null;
    return children;
}
function updateReducer(reducer) {
    const hook = updateWorkInProgressHook();
    const queue = hook.queue;
    queue.lastRenderedReducer = reducer;
    const current = currentHook;
    const pendingQueue = queue.pending;
    if (pendingQueue !== null) {
        const first = pendingQueue.next;
        let newState = current.memoizedState;
        let update = first;
        do {
            const action = update.action;
            newState = reducer(newState, action);
            update = update.next;
        } while (update !== null && update !== first);
        queue.pending = null;
        hook.memoizedState = newState;
        queue.lastRenderedState = newState;
    }
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}
function updateWorkInProgressHook() {
    let nextCurrentHook;
    if (currentHook
        const current = currentlyRenderingFiber.alternate;
        nextCurrentHook = current.memoizedState;
    } else {
        nextCurrentHook = currentHook.next;
    }
    currentHook = nextCurrentHook;

    const newHook = {
        memoizedState: currentHook.memoizedState,
        queue: currentHook.queue,
        next: null,
    };
    if (workInProgressHook
        currentlyRenderingFiber.memoizedState = workInProgressHook = newHook;
    } else {
        workInProgressHook = workInProgressHook.next = newHook;
    }
    return workInProgressHook;
}
export function mountReducer(reducer, initialArg) {
    const hook = mountWorkInProgressHook();
    let initialState = initialArg;
    hook.memoizedState = initialState;
    const queue = (hook.queue = {pending: null, lastRenderedReducer: reducer, lastRenderedState: initialState});
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}

export function mountWorkInProgressHook() {
    const hook = {
        memoizedState: null,
        queue: null,
        next: null,
    };
    if (workInProgressHook
        currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
    } else {
        workInProgressHook = workInProgressHook.next = hook;
    }
    return workInProgressHook;
}

export function dispatchAction(fiber, queue, action) {
   const update = { action, next: null };
   const pending = queue.pending;
   if (pending
       update.next = update;
   } else {
       update.next = pending.next;
       pending.next = update;
   }
   queue.pending = update;
```

```
      const lastRenderedReducer = queue.lastRenderedReducer;
   const currentState = queue.lastRenderedState;
   const eagerState = lastRenderedReducer(currentState, action);
   if (Object.is(eagerState, currentState)) {
       return
   }
   scheduleUpdateOnFiber(fiber);
}
```

8.3 ReactFiberWorkLoop.js

src\ReactFiberWorkLoop.js

```
import { beginWork } from './ReactFiberBeginWork';
import {Update, Passive, NoFlags } from './ReactFiberFlags';
import { commitLifeCycles as commitLayoutEffectOnFiber } from './ReactFiberCommitWork';
+import { FunctionComponent } from './ReactWorkTags';
+import { HasEffect as HookHasEffect, Layout as HookLayout } from './ReactHookEffectTags';
let workInProgress;
let finishedWork = null
let pendingPassiveHookEffectsMount = [];
let pendingPassiveHookEffectsUnmount = [];
export function workLoop() {
    while (workInProgress) {
        workInProgress = performUnitOfWork(workInProgress);
    }
    commitRoot();
}
function commitRoot() {
    if (!finishedWork) return;
    commitBeforeMutationEffects();
+   commitMutationEffects();
    commitLayoutEffects();
}
export function enqueuePendingPassiveHookEffectMount(fiber, effect,) {
    pendingPassiveHookEffectsMount.push(effect, fiber);
}

export function enqueuePendingPassiveHookEffectUnmount(fiber, effect) {
    pendingPassiveHookEffectsUnmount.push(effect, fiber);
}
function commitLayoutEffects() {
    const flags = finishedWork.flags;
    if (flags & Update) {
        const current = finishedWork.alternate;
        commitLayoutEffectOnFiber(finishedWork, current, finishedWork);
    }
}
+function commitMutationEffects() {
+    const flags = finishedWork.flags;
+    const primaryFlags = flags & (Update);
+    switch (primaryFlags) {
+        case Update: {
+            const current = finishedWork.alternate;
+            commitWork(current, finishedWork);
+            break;
+        }
+        default:
+            break;
+    }
+}
+function commitWork(current, finishedWork) {
+    switch (finishedWork.tag) {
+        case FunctionComponent:
+            commitHookEffectListUnmount(
+                HookLayout | HookHasEffect,
+                finishedWork
+            );
+            break;
+        default:
+            break;
+    }
+}
+function commitHookEffectListUnmount(tag, finishedWork) {
+    const updateQueue = finishedWork.updateQueue;
+    const lastEffect = updateQueue !== null ? updateQueue.lastEffect : null;
+    if (lastEffect !== null) {
+        const firstEffect = lastEffect.next;
+        let effect = firstEffect;
+        do {
+            if ((effect.tag & tag) === tag) {
+                const destroy = effect.destroy;
+                effect.destroy = undefined;
+                if (destroy !== undefined) {
+                    destroy();
+                }
+            }
+            effect = effect.next;
+        } while (effect !== firstEffect);
+    }
+}
function commitBeforeMutationEffects() {
    const flags = finishedWork.flags;
    if ((flags & Passive) !== NoFlags) {
        setTimeout(flushPassiveEffects);
    }
}
function flushPassiveEffects() {
    const unmountEffects = pendingPassiveHookEffectsUnmount;
    pendingPassiveHookEffectsUnmount = [];
    for (let i = 0; i < unmountEffects.length; i += 2) {
        const effect = unmountEffects[i];
        const destroy = effect.destroy;
        effect.destroy = undefined;
```

```
        if (typeof destroy
            destroy();
        }
    }

    const mountEffects = pendingPassiveHookEffectsMount;
    pendingPassiveHookEffectsMount = [];
    for (let i = 0; i < mountEffects.length; i += 2) {
        const effect = mountEffects[i]
        const create = effect.create;
        effect.destroy = create();
    }
}
export function performUnitOfWork(unitOfWork) {
    let current = unitOfWork.alternate;
    return beginWork(current, unitOfWork);
}
export function scheduleUpdateOnFiber(fiber) {
    let newFiber = {
        ...fiber,
        alternate: fiber
    }
    finishedWork = workInProgress = newFiber;
    workLoop();
}
export function render(fiber) {
    finishedWork = workInProgress = fiber;
    workLoop();
}
```

8.4 ReactFiberCommitWork.js

src\ReactFiberCommitWork.js

```
import { FunctionComponent } from './ReactWorkTags';
import {
    Layout as HookLayout, HasEffect as HookHasEffect,
    Passive as HookPassive, NoFlags as NoHookEffect
} from './ReactHookEffectTags';
import {
    enqueuePendingPassiveHookEffectMount,
    enqueuePendingPassiveHookEffectUnmount,
} from './ReactFiberWorkLoop';
export function commitLifeCycles(finishedRoot, current, finishedWork) {
    switch (finishedWork.tag) {
        case FunctionComponent:
+           commitHookEffectListMount(HookLayout | HookHasEffect, finishedWork);
            schedulePassiveEffects(finishedWork);
            break;
        default:
            break;
    }
}
+function commitHookEffectListMount(tag, finishedWork) {
+    const updateQueue = finishedWork.updateQueue;
+    const lastEffect = updateQueue !== null ? updateQueue.lastEffect : null;
+    if (lastEffect !== null) {
+        const firstEffect = lastEffect.next;
+        let effect = firstEffect;
+        do {
+            if ((effect.tag & tag) === tag) {
+                const create = effect.create;
+                effect.destroy = create();
+            }
+            effect = effect.next;
+        } while (effect !== firstEffect);
+    }
+}
function schedulePassiveEffects(finishedWork) {
    const updateQueue = finishedWork.updateQueue;
    const lastEffect = updateQueue !== null ? updateQueue.lastEffect : null;
    if (lastEffect !== null) {
        const firstEffect = lastEffect.next;
        let effect = firstEffect;
        do {
            const { next, tag } = effect;
            if ((tag & HookPassive) !== NoHookEffect && (tag & HookHasEffect) !== NoHookEffect) {
                enqueuePendingPassiveHookEffectUnmount(finishedWork, effect);
                enqueuePendingPassiveHookEffectMount(finishedWork, effect);
            }
            effect = next;
        } while (effect !== firstEffect);
    }
}
```

9.useRef

9.1.src\index.js

```
import * as React from 'react';
import { IndeterminateComponent } from './ReactWorkTags';
import {render} from './ReactFiberWorkLoop';
+import {useReducer,useState,useEffect,useLayoutEffect,useRef} from './ReactFiberHooks'

function Counter() {
  const [number, setNumber] = useState(0);
+ const numberRef = useRef();
  useEffect(()=>{
    console.log('useEffect1');
+   numberRef.current = number;
    setTimeout(()=>{
+     console.log(numberRef.current);
    },3000)
  });
  return (
     {setNumber(number+1)}}>{number}
  )
}
let workInProgress = {
  tag:IndeterminateComponent,
  type: Counter,
  alternate:null,
  updateQueue:null
}
render(workInProgress);
```

9.2 ReactFiberHooks.js

src\ReactFiberHooks.js

```
import { scheduleUpdateOnFiber } from './ReactFiberWorkLoop';
import { Update as UpdateEffect, Passive as PassiveEffect } from './ReactFiberFlags';
import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout} from './ReactHookEffectTags';
let ReactCurrentDispatcher = {
    current: null
}
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;
const HooksDispatcherOnMount = {
    useReducer: mountReducer,
    useState: mountState,
    useEffect: mountEffect,
    useLayoutEffect: mountLayoutEffect,
+   useRef: mountRef
}
const HooksDispatcherOnUpdate = {
    useReducer: updateReducer,
    useState: updateState,
    useEffect: updateEffect,
    useLayoutEffect: updateLayoutEffect,
+   useRef: updateRef
}
+function updateRef(initialValue) {
+    const hook = updateWorkInProgressHook()
+    return hook.memoizedState
+}
+function mountRef(initialValue) {
+    const hook = mountWorkInProgressHook();
+    const ref = { current: initialValue };
+    hook.memoizedState = ref;
+    return ref;
+}
+export function useRef(initialValue) {
+    return ReactCurrentDispatcher.current.useRef(initialValue);
+}
export function useLayoutEffect(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useLayoutEffect(reducer, initialArg);
}
function updateLayoutEffect(create, deps,) {
    return updateEffectImpl(UpdateEffect, HookLayout, create, deps);
}
function mountLayoutEffect(create, deps,) {
    return mountEffectImpl(UpdateEffect, HookLayout, create, deps);
}
export function mountEffect(create, deps) {
    return mountEffectImpl(
        UpdateEffect | PassiveEffect,
        HookPassive,
        create,
        deps
    );
}
function mountEffectImpl(fiberFlags, hookFlags, create, deps) {
    const hook = mountWorkInProgressHook();
    const nextDeps = deps
    currentlyRenderingFiber.flags |= fiberFlags;
    hook.memoizedState = pushEffect(
        HookHasEffect | hookFlags,
        create,
        undefined,
        nextDeps,
    );
}
function pushEffect(tag, create, destroy, deps) {
    const effect = { tag, create, destroy, deps, next: null};
    let componentUpdateQueue = (currentlyRenderingFiber.updateQueue);
    if (componentUpdateQueue
        componentUpdateQueue = createFunctionComponentUpdateQueue();
        currentlyRenderingFiber.updateQueue = componentUpdateQueue;
        componentUpdateQueue.lastEffect = effect.next = effect;
    } else {
        const lastEffect = componentUpdateQueue.lastEffect;
```

```
        if (lastEffect
            componentUpdateQueue.lastEffect = effect.next = effect;
        } else {
            const firstEffect = lastEffect.next;
            lastEffect.next = effect;
            effect.next = firstEffect;
            componentUpdateQueue.lastEffect = effect;
        }
    }
    return effect;
}
function createFunctionComponentUpdateQueue() {
    return {
        lastEffect: null,
    };
}
export function updateEffect(create, deps,) {
    return updateEffectImpl(
        PassiveEffect,
        HookPassive,
        create,
        deps,
    );
}
function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
    const hook = updateWorkInProgressHook();
    const nextDeps = deps
    let destroy = undefined;
    if (currentHook !== null) {
        const prevEffect = currentHook.memoizedState;
        destroy = prevEffect.destroy;
        if (nextDeps !== null) {
            const prevDeps = prevEffect.deps;
            if (areHookInputsEqual(nextDeps, prevDeps)) {
                pushEffect(hookFlags, create, destroy, nextDeps);
                return;
            }
        }
    }
    currentlyRenderingFiber.flags |= fiberFlags;
    hook.memoizedState = pushEffect(HookHasEffect | hookFlags, create, destroy, nextDeps)
}
function areHookInputsEqual(nextDeps, prevDeps) {
    if (prevDeps
        return false;
    }
    for (let i = 0; i < prevDeps.length && i < nextDeps.length; i++) {
        if (Object.is(nextDeps[i], prevDeps[i])) {
            continue;
        }
        return false;
    }
    return true;
}
function mountState(initialState) {
    const hook = mountWorkInProgressHook();
    hook.memoizedState = initialState;
    const queue = (hook.queue = { pending: null ,lastRenderedReducer: basicStateReducer, lastRenderedState: initialState});
    const dispatch = dispatchAction.bind(null, currentlyRenderingFiber, queue)
    return [hook.memoizedState, dispatch];
}

function basicStateReducer(state, action) {
    return typeof action
}

function updateState(initialState) {
    return updateReducer(basicStateReducer, initialState);
}
export function useEffect(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useEffect(reducer, initialArg);
}
export function useState(initialState) {
    return ReactCurrentDispatcher.current.useState(initialState);
}
export function useReducer(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useReducer(reducer, initialArg);
}

export function renderWithHooks(_current, workInProgress, Component) {
    currentlyRenderingFiber = workInProgress;
    workInProgress.memoizedState = null;
    workInProgress.updateQueue = null;
    if (_current !== null) {
        ReactCurrentDispatcher.current = HooksDispatcherOnUpdate;
    } else {
        ReactCurrentDispatcher.current = HooksDispatcherOnMount;
    }
    let children = Component();
    window.counter = children;
    currentlyRenderingFiber = null;
    currentHook = null;
    workInProgressHook = null;
    return children;
}
function updateReducer(reducer) {
    const hook = updateWorkInProgressHook();
    const queue = hook.queue;
    queue.lastRenderedReducer = reducer;
    const current = currentHook;
    const pendingQueue = queue.pending;
    if (pendingQueue !== null) {
        const first = pendingQueue.next;
        let newState = current.memoizedState;
```

```
        let update = first;
        do {
            const action = update.action;
            newState = reducer(newState, action);
            update = update.next;
        } while (update !== null && update !== first);
        queue.pending = null;
        hook.memoizedState = newState;
        queue.lastRenderedState = newState;
    }
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}
function updateWorkInProgressHook() {
    let nextCurrentHook;
    if (currentHook
        const current = currentlyRenderingFiber.alternate;
        nextCurrentHook = current.memoizedState;
    } else {
        nextCurrentHook = currentHook.next;
    }
    currentHook = nextCurrentHook;

    const newHook = {
        memoizedState: currentHook.memoizedState,
        queue: currentHook.queue,
        next: null,
    };
    if (workInProgressHook
        currentlyRenderingFiber.memoizedState = workInProgressHook = newHook;
    } else {
        workInProgressHook = workInProgressHook.next = newHook;
    }
    return workInProgressHook;
}
export function mountReducer(reducer, initialArg) {
    const hook = mountWorkInProgressHook();
    let initialState = initialArg;
    hook.memoizedState = initialState;
    const queue = (hook.queue = {pending: null, lastRenderedReducer: reducer, lastRenderedState: initialState});
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}

export function mountWorkInProgressHook() {
    const hook = {
        memoizedState: null,
        queue: null,
        next: null,
    };
    if (workInProgressHook
        currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
    } else {
        workInProgressHook = workInProgressHook.next = hook;
    }
    return workInProgressHook;
}

export function dispatchAction(fiber, queue, action) {
   const update = { action, next: null };
   const pending = queue.pending;
   if (pending
       update.next = update;
   } else {
       update.next = pending.next;
       pending.next = update;
   }
   queue.pending = update;
      const lastRenderedReducer = queue.lastRenderedReducer;
   const currentState = queue.lastRenderedState;
   const eagerState = lastRenderedReducer(currentState, action);
   if (Object.is(eagerState, currentState)) {
       return
   }
   scheduleUpdateOnFiber(fiber);
}
```

10.useMemo

10.1.src\index.js

```
import * as React from 'react';
import { IndeterminateComponent } from './ReactWorkTags';
import {render} from './ReactFiberWorkLoop';
+import {useReducer,useState,useEffect,useLayoutEffect,useRef,useMemo} from './ReactFiberHooks'
let lastData;
+function Counter() {
+  const [number, setNumber] = useState(0);
+  const [age, setAge] = useState(0);
+  const data = useMemo(()=>({age}),[age]);
+  console.log("lastData === data",lastData === data);
+  lastData = data;
+  return (
+     {setNumber(number+1)}}>{number}
+  )
+}
let workInProgress = {
  tag:IndeterminateComponent,
  type: Counter,
  alternate:null,
  updateQueue:null
}
render(workInProgress);
```

10.2 ReactFiberHooks.js

src\ReactFiberHooks.js

```
import { scheduleUpdateOnFiber } from './ReactFiberWorkLoop';
import { Update as UpdateEffect, Passive as PassiveEffect } from './ReactFiberFlags';
import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout} from './ReactHookEffectTags';
let ReactCurrentDispatcher = {
    current: null
}
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;
const HooksDispatcherOnMount = {
    useReducer: mountReducer,
    useState: mountState,
    useEffect: mountEffect,
    useLayoutEffect: mountLayoutEffect,
    useRef: mountRef,
+   useMemo: mountMemo
}
const HooksDispatcherOnUpdate = {
    useReducer: updateReducer,
    useState: updateState,
    useEffect: updateEffect,
    useLayoutEffect: updateLayoutEffect,
    useRef: updateRef,
+   useMemo: updateMemo
}
+function mountMemo(nextCreate, deps) {
+    const hook = mountWorkInProgressHook();
+    const nextDeps = deps === undefined ? null : deps;
+    const nextValue = nextCreate();
+    hook.memoizedState = [nextValue, nextDeps];
+    return nextValue;
+}
+function updateMemo(nextCreate,deps) {
+    const hook = updateWorkInProgressHook();
+    const nextDeps = deps === undefined ? null : deps;
+    const prevState = hook.memoizedState;
+    if (prevState !== null) {
+        if (nextDeps !== null) {
+            const prevDeps = prevState[1];
+            if (areHookInputsEqual(nextDeps, prevDeps)) {
+                return prevState[0];
+            }
+        }
+    }
+    const nextValue = nextCreate();
+    hook.memoizedState = [nextValue, nextDeps];
+    return nextValue;
+}
function updateRef(initialValue) {
    const hook = updateWorkInProgressHook()
    return hook.memoizedState
}
function mountRef(initialValue) {
    const hook = mountWorkInProgressHook();
    const ref = { current: initialValue };
    hook.memoizedState = ref;
    return ref;
}
+export function useMemo(nextCreate, deps) {
+    return ReactCurrentDispatcher.current.useMemo(nextCreate, deps);
+}
export function useRef(initialValue) {
    return ReactCurrentDispatcher.current.useRef(initialValue);
}
export function useLayoutEffect(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useLayoutEffect(reducer, initialArg);
}
function updateLayoutEffect(create, deps,) {
    return updateEffectImpl(UpdateEffect, HookLayout, create, deps);
}
function mountLayoutEffect(create, deps,) {
    return mountEffectImpl(UpdateEffect, HookLayout, create, deps);
}
export function mountEffect(create, deps) {
    return mountEffectImpl(
        UpdateEffect | PassiveEffect,
        HookPassive,
        create,
        deps
    );
}
function mountEffectImpl(fiberFlags, hookFlags, create, deps) {
    const hook = mountWorkInProgressHook();
    const nextDeps = deps
    currentlyRenderingFiber.flags |= fiberFlags;
    hook.memoizedState = pushEffect(
        HookHasEffect | hookFlags,
        create,
        undefined,
        nextDeps,
    );
}
function pushEffect(tag, create, destroy, deps) {
    const effect = { tag, create, destroy, deps, next: null};
    let componentUpdateQueue = (currentlyRenderingFiber.updateQueue);
    if (componentUpdateQueue
        componentUpdateQueue = createFunctionComponentUpdateQueue();
        currentlyRenderingFiber.updateQueue = componentUpdateQueue;
        componentUpdateQueue.lastEffect = effect.next = effect;
    } else {
```

```
        const lastEffect = componentUpdateQueue.lastEffect;
        if (lastEffect
            componentUpdateQueue.lastEffect = effect.next = effect;
        } else {
            const firstEffect = lastEffect.next;
            lastEffect.next = effect;
            effect.next = firstEffect;
            componentUpdateQueue.lastEffect = effect;
        }
    }
    return effect;
}
function createFunctionComponentUpdateQueue() {
    return {
        lastEffect: null,
    };
}
export function updateEffect(create, deps,) {
    return updateEffectImpl(
        PassiveEffect,
        HookPassive,
        create,
        deps,
    );
}
function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
    const hook = updateWorkInProgressHook();
    const nextDeps = deps
    let destroy = undefined;
    if (currentHook !== null) {
        const prevEffect = currentHook.memoizedState;
        destroy = prevEffect.destroy;
        if (nextDeps !== null) {
            const prevDeps = prevEffect.deps;
            if (areHookInputsEqual(nextDeps, prevDeps)) {
                pushEffect(hookFlags, create, destroy, nextDeps);
                return;
            }
        }
    }
    currentlyRenderingFiber.flags |= fiberFlags;
    hook.memoizedState = pushEffect(HookHasEffect | hookFlags, create, destroy, nextDeps)
}
function areHookInputsEqual(nextDeps, prevDeps) {
    if (prevDeps
        return false;
    }
    for (let i = 0; i < prevDeps.length && i < nextDeps.length; i++) {
        if (Object.is(nextDeps[i], prevDeps[i])) {
            continue;
        }
        return false;
    }
    return true;
}
function mountState(initialState) {
    const hook = mountWorkInProgressHook();
    hook.memoizedState = initialState;
    const queue = (hook.queue = { pending: null,lastRenderedReducer: basicStateReducer, lastRenderedState: initialState });
    const dispatch = dispatchAction.bind(null, currentlyRenderingFiber, queue)
    return [hook.memoizedState, dispatch];
}

function basicStateReducer(state, action) {
    return typeof action
}

function updateState(initialState) {
    return updateReducer(basicStateReducer, initialState);
}
export function useEffect(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useEffect(reducer, initialArg);
}
export function useState(initialState) {
    return ReactCurrentDispatcher.current.useState(initialState);
}
export function useReducer(reducer, initialArg) {
    return ReactCurrentDispatcher.current.useReducer(reducer, initialArg);
}

export function renderWithHooks(_current, workInProgress, Component) {
    currentlyRenderingFiber = workInProgress;
    workInProgress.memoizedState = null;
    workInProgress.updateQueue = null;
    if (_current !== null) {
        ReactCurrentDispatcher.current = HooksDispatcherOnUpdate;
    } else {
        ReactCurrentDispatcher.current = HooksDispatcherOnMount;
    }
    let children = Component();
    window.counter = children;
    currentlyRenderingFiber = null;
    currentHook = null;
    workInProgressHook = null;
    return children;
}
function updateReducer(reducer) {
    const hook = updateWorkInProgressHook();
    const queue = hook.queue;
    queue.lastRenderedReducer = reducer;
    const current = currentHook;
    const pendingQueue = queue.pending;
    if (pendingQueue !== null) {
        const first = pendingQueue.next;
```

```
        let newState = current.memoizedState;
        let update = first;
        do {
            const action = update.action;
            newState = reducer(newState, action);
            update = update.next;
        } while (update !== null && update !== first);
        queue.pending = null;
        hook.memoizedState = newState;
        queue.lastRenderedState = newState;
    }
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}
function updateWorkInProgressHook() {
    let nextCurrentHook;
    if (currentHook
        const current = currentlyRenderingFiber.alternate;
        nextCurrentHook = current.memoizedState;
    } else {
        nextCurrentHook = currentHook.next;
    }
    currentHook = nextCurrentHook;

    const newHook = {
        memoizedState: currentHook.memoizedState,
        queue: currentHook.queue,
        next: null,
    };
    if (workInProgressHook
        currentlyRenderingFiber.memoizedState = workInProgressHook = newHook;
    } else {
        workInProgressHook = workInProgressHook.next = newHook;
    }
    return workInProgressHook;
}
export function mountReducer(reducer, initialArg) {
    const hook = mountWorkInProgressHook();
    let initialState = initialArg;
    hook.memoizedState = initialState;
    const queue = (hook.queue = {pending: null, lastRenderedReducer: reducer, lastRenderedState: initialState});
    const dispatch = dispatchAction.bind(null,currentlyRenderingFiber,queue);
    return [hook.memoizedState, dispatch];
}

export function mountWorkInProgressHook() {
    const hook = {
        memoizedState: null,
        queue: null,
        next: null,
    };
    if (workInProgressHook
        currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
    } else {
        workInProgressHook = workInProgressHook.next = hook;
    }
    return workInProgressHook;
}

export function dispatchAction(fiber, queue, action) {
   const update = { action, next: null };
   const pending = queue.pending;
   if (pending
       update.next = update;
   } else {
       update.next = pending.next;
       pending.next = update;
   }
   queue.pending = update;
   const lastRenderedReducer = queue.lastRenderedReducer;
   const currentState = queue.lastRenderedState;
   const eagerState = lastRenderedReducer(currentState, action);
   if (Object.is(eagerState, currentState)) {
       return
   }
   scheduleUpdateOnFiber(fiber);
}
```