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=1310 sentences=4737, words=48662

1.React 前置知识

1.1 React 是什么？#

- React (https://zh-hans.reactjs.org/)是一个用于构建用户界面的 JavaScript 库
- 可以通过组件化的方式构建 构建快速响应的大型 Web应用程序

1.2 JSX 是什么

- jsx (https://zh-hans.reactjs.org/docs/introducing-jsx.html)
- JSX 是一个 JavaScript的语法扩展,JSX 可以很好地描述 UI 应该呈现出它应有交互的本质形式
- repl (https://babeljs.io/repl)可以在线转换代码
- astexplorer (https://astexplorer.net/)可以把代码转换成 AST 树
- react/jsx-runtime 和 react/jsx-dev-runtime 中的函数只能由编译器转换使用。如果你需要在代码中手动创建元素，你可以继续使用 React.createElement

1.2.1 旧转换

1.2.1.1 jsx.js#

```
const babel = require("@babel/core");
const sourceCode = `

   helloworld

`;
const result = babel.transform(sourceCode, {
  plugins: [["@babel/plugin-transform-react-jsx", { runtime: "classic" }]],
});
console.log(result.code);
```

1.2.1.2 转译结果

```
React.createElement(
  "h1",
  null,
  "hello",
  React.createElement(
    "span",
    {
      style: {
        color: "red",
      },
    },
    "world"
  )
);
```

1.2.2 新转换

1.2.2.1 jsx.js#

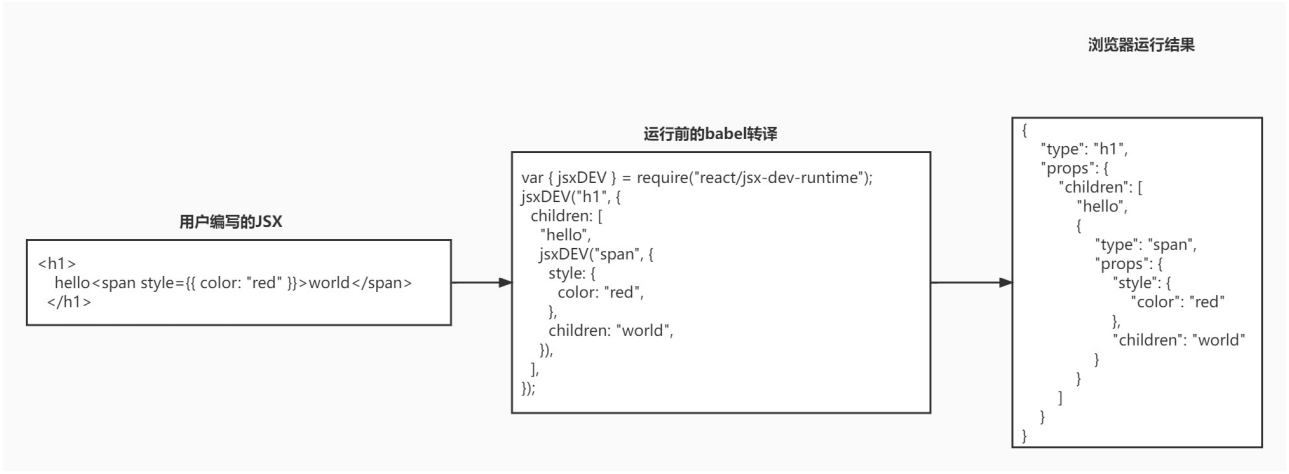
```
const babel = require("@babel/core");
const sourceCode = `

   helloworld

`;
const result = babel.transform(sourceCode, {
+ plugins: [["@babel/plugin-transform-react-jsx", { runtime: "automatic" }]],
});
console.log(result.code);
```

1.2.2.2 转译结果

```
var { jsxDEV } = require("react/jsx-dev-runtime");
jsxDEV("h1", {
  children: [
    "hello",
    jsxDEV("span", {
      style: {
        color: "red",
      },
      children: "world",
    }),
  ],
});
```

1.3 Virtual DOM

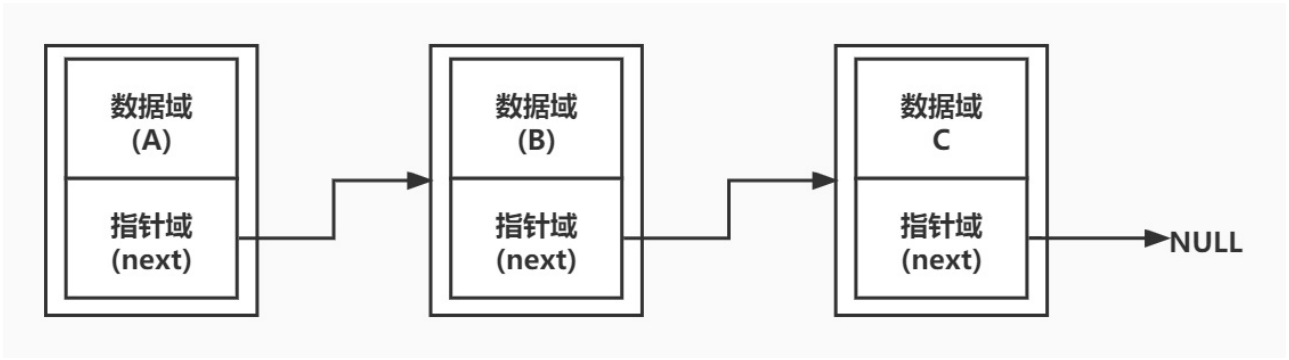
- React.createElement 函数所返回的就是一个虚拟 DOM
- 虚拟 DOM 就是一个描述真实 DOM 的纯 JS 对象



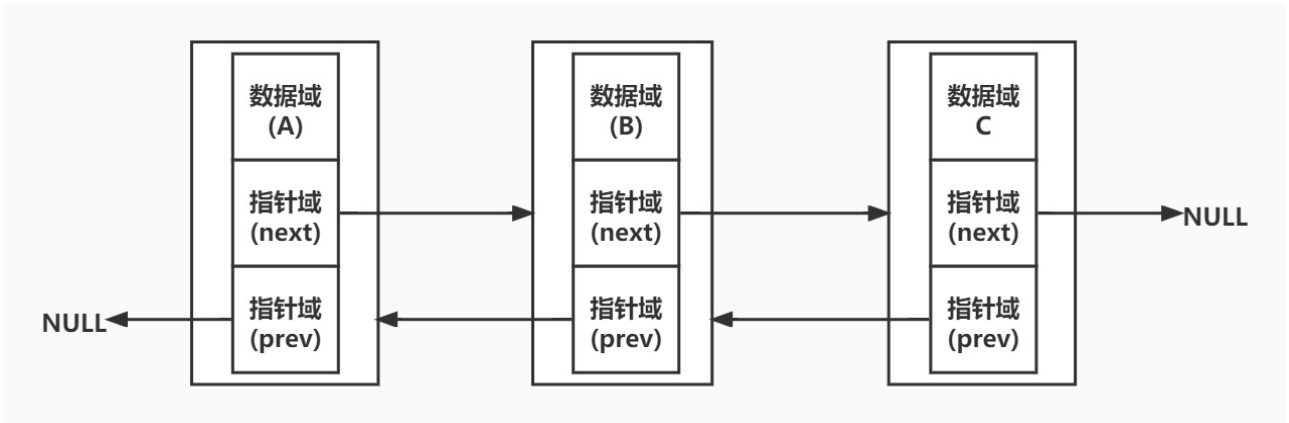
1.4 链表

1.4.1 链表分类

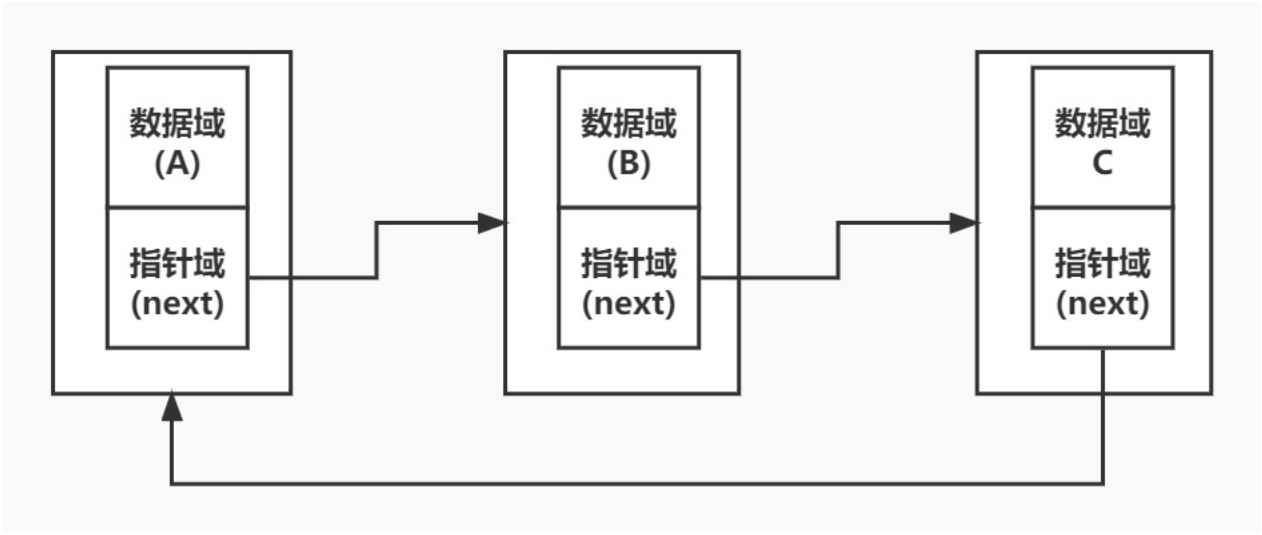
1.4.1.1 单向链表

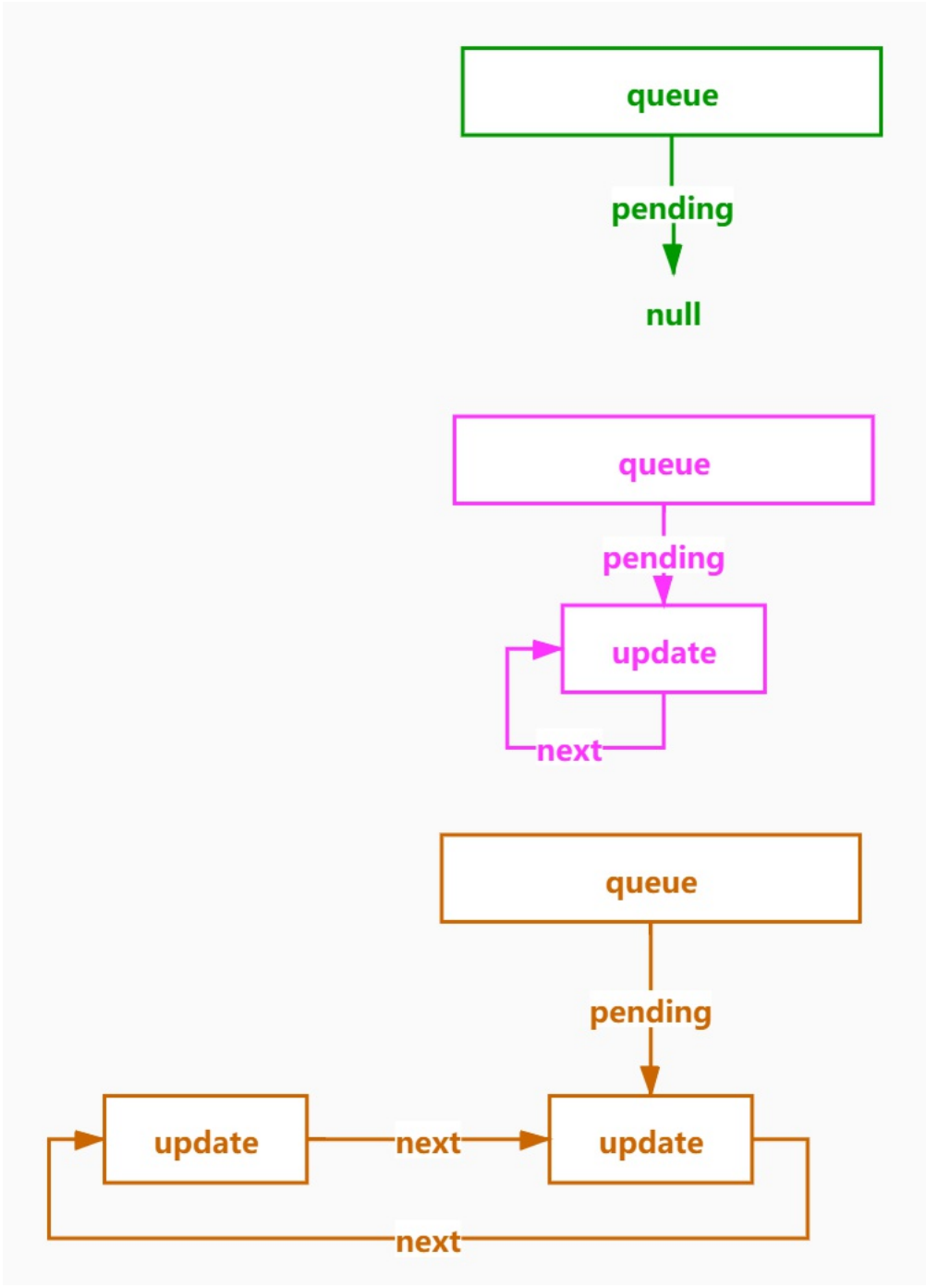


1.4.1.2 双向链表



1.4.1.3 循环链表


数据域
(A)
指针域
(next)
数据域
(B)
指针域
(next)
数据域
C
指针域
(next)


queue
pending
null
queue
pending
update
next
queue
pending
update
next
update
next

1.4.1.4 示例

```
const UpdateState = 0;
function initializeUpdateQueue(fiber) {
  const queue = {
    shared: {
      pending: null,
    },
  };
  fiber.updateQueue = queue;
}
function createUpdate() {
  const update = { tag: UpdateState };
  return update;
}
function enqueueUpdate(fiber, update) {
  const updateQueue = fiber.updateQueue;
  const sharedQueue = updateQueue.shared;
  const pending = sharedQueue.pending;
  if (pending === null) {
    update.next = update;
  } else {
    update.next = pending.next;
    pending.next = update;
  }
  updateQueue.shared.pending = update;
}
function getStateFromUpdate(update, prevState) {
  switch (update.tag) {
    case UpdateState: {
      const { payload } = update;
      const partialState = payload;
      return Object.assign({}, prevState, partialState);
    }
    default:
      return prevState;
  }
}
function processUpdateQueue(workInProgress) {
  const queue = workInProgress.updateQueue;
  const pendingQueue = queue.shared.pending;
  if (pendingQueue !== null) {
    queue.shared.pending = null;
    const lastPendingUpdate = pendingQueue;
    const firstPendingUpdate = lastPendingUpdate.next;
    lastPendingUpdate.next = null;
    let newState = workInProgress.memoizedState;
    let update = firstPendingUpdate;
    while (update) {
      newState = getStateFromUpdate(update, newState);
      update = update.next;
    }
    workInProgress.memoizedState = newState;
  }
}
let fiber = { memoizedState: { id: 1 } };
initializeUpdateQueue(fiber);
let update1 = createUpdate();
update1.payload = { name: "zhufeng" };
enqueueUpdate(fiber, update1);
let update2 = createUpdate();
update2.payload = { age: 14 };
enqueueUpdate(fiber, update2);
processUpdateQueue(fiber);
console.log(fiber);
```

1.5 fiber

1.5.1 性能瓶颈

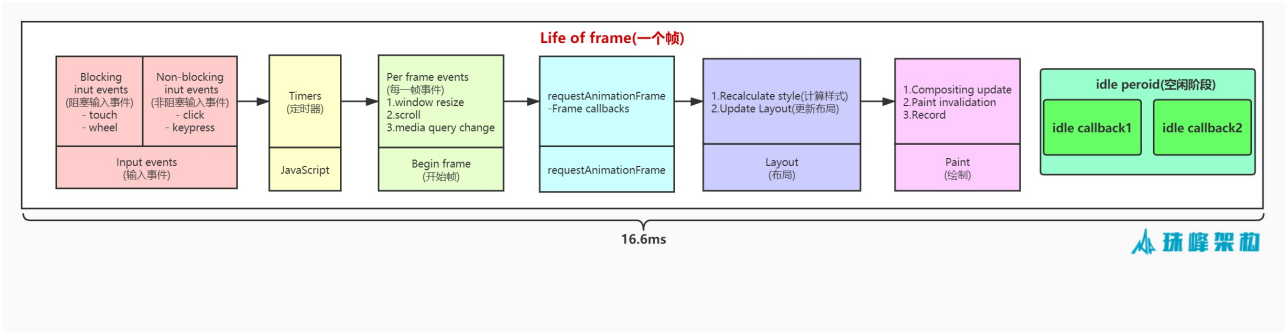
- JS 任务执行时间过长
    - 浏览器刷新频率为 60Hz,大概 16.6 毫秒渲染一次，而 JS 线程和渲染线程是互斥的，所以如果 JS 线程执行任务时间超过 16.6ms 的话，就会导致掉帧，导致卡顿，解决方案就是 React 利用空闲的时间进行更新，不影响渲染进行的渲染
    - 把一个耗时任务切分成一个个小任务，分布在每一帧里的方式就叫时间切片

1.5.2 屏幕刷新率

- 目前大多数设备的屏幕刷新率为 60 次/秒
- 浏览器渲染动画或页面的每一帧的速率也需要跟设备屏幕的刷新率保持一致
- 页面是一帧一帧绘制出来的，当每秒绘制的帧数（FPS）达到 60 时，页面是流畅的,小于这个值时，用户会感觉到卡顿
- 每个帧的预算时间是 16.66 毫秒 (1 秒/60)
- 1s 60 帧，所以每一帧分到的时间是 1000/60 ≈ 16 ms,所以我们书写代码时力求不让一帧的工作量超过 16ms

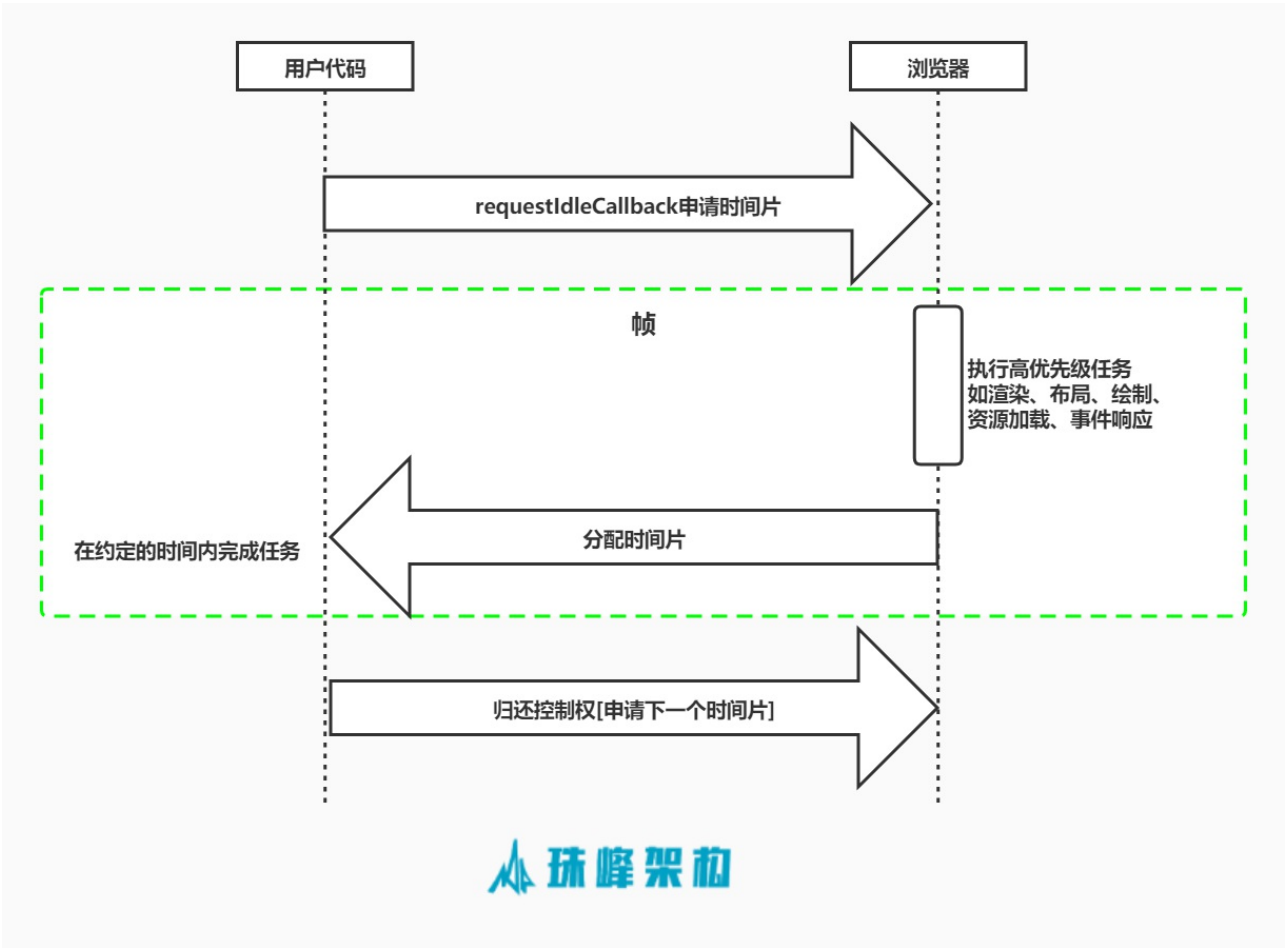
1.5.3 帧

- 每个帧的开头包括样式计算、布局和绘制
- JavaScript 执行 Javascript 引擎和页面渲染引擎在同一个渲染线程,GUI 渲染和 Javascript 执行两者是互斥的
- 如果某个任务执行时间过长，浏览器会推迟渲染



1.5.4 requestIdleCallback

- 我们希望快速响应用户，让用户觉得够快，不能阻塞用户的交互
- `requestIdleCallback` 使开发者能够在主事件循环上执行后台和低优先级工作，而不会影响延迟关键事件，如动画和输入响应
- 正常帧任务完成后没超过 16 ms,说明时间有富余，此时就会执行 `requestIdleCallback` 里注册的任务



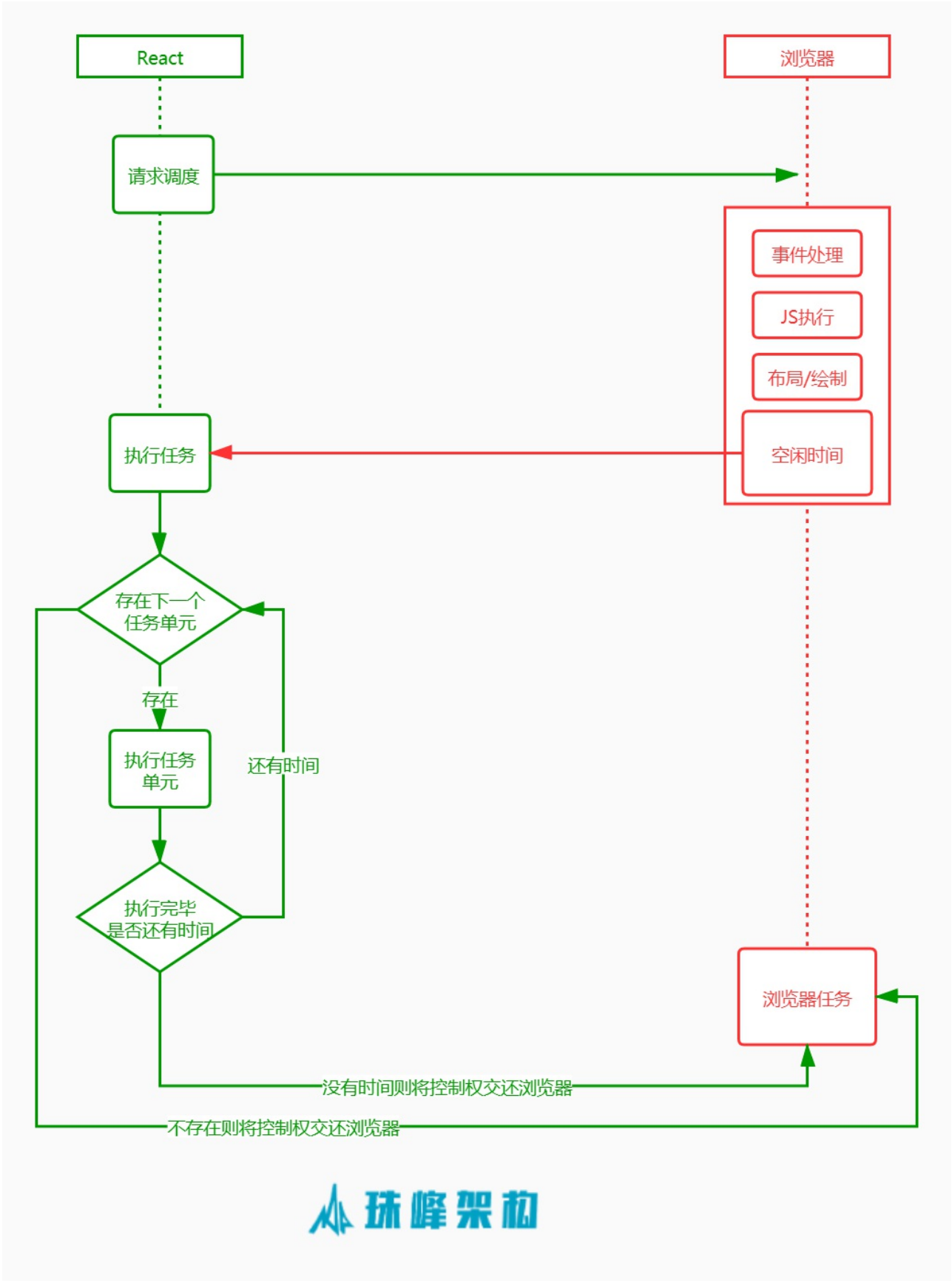
```
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  head>
  <body>
    <script>
      function sleep(d) {
        for (var t = Date.now(); Date.now() - t const works = [
        () => {
          console.log("第1个任务开始");
          sleep(20);
          console.log("第1个任务结束");
        },
        () => {
          console.log("第2个任务开始");
          sleep(20);
          console.log("第2个任务结束");
        },
        () => {
          console.log("第3个任务开始");
          sleep(20);
          console.log("第3个任务结束");
        },
      ];

      requestIdleCallback(workLoop);
      function workLoop(deadline) {
        console.log("本帧剩余时间", parseInt(deadline.timeRemaining()));
        while (deadline.timeRemaining() > 1 && works.length > 0) {
          performUnitOfWork();
        }
        if (works.length > 0) {
          console.log(`只剩下${parseInt(deadline.timeRemaining())}ms,时间片到了等待下次空闲时间的调度`);
          requestIdleCallback(workLoop);
        }
      }
      function performUnitOfWork() {
        works.shift()();
      }
    script>
  body>
html>
```

1.5.5 fiber

- 我们可以通过某些调度策略合理分配 CPU 资源，从而提高用户的响应速度
- 通过 Fiber 架构，让自己的调和过程变成可被中断。 适时地让出 CPU 执行权，除了可以让浏览器及时地响应用户的交互

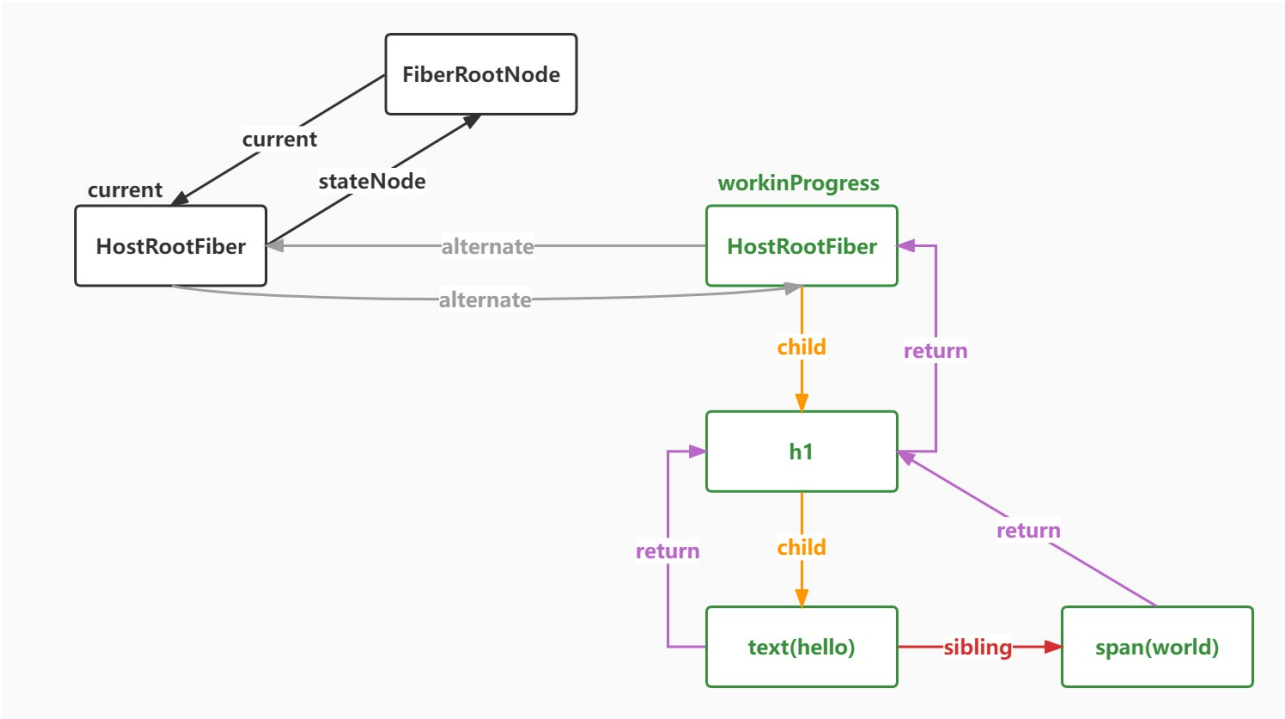
1.5.5.1 Fiber 是一个执行单元

- 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

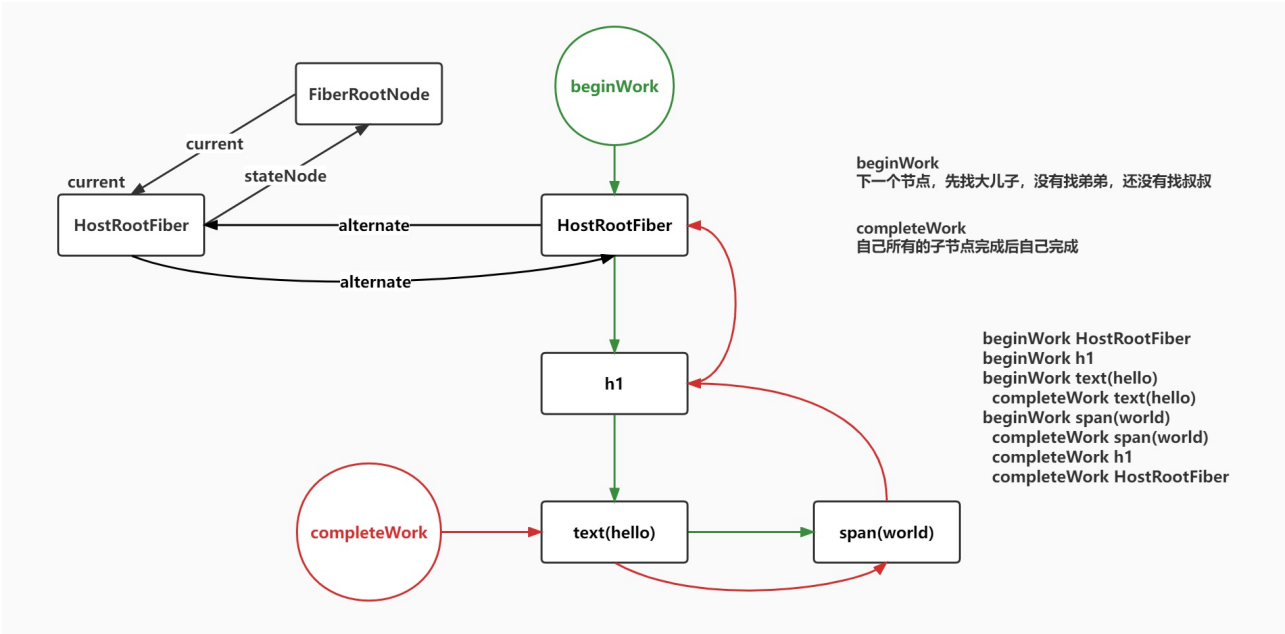


1.5.5.2 Fiber是一种数据结构

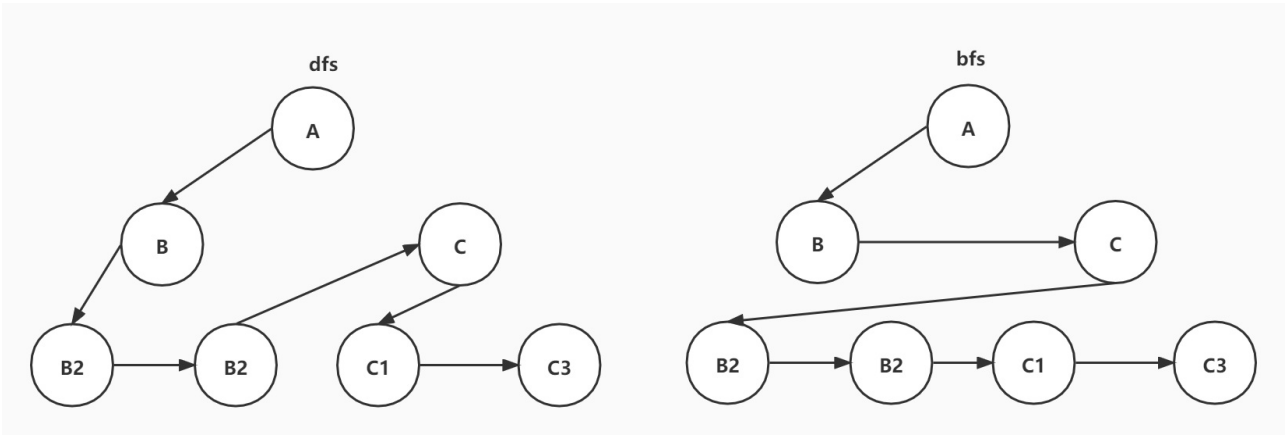
- React 目前的做法是使用链表, 每个虚拟节点内部表示为一个 `Fiber`
- 从顶点开始遍历
- 如果有第一个儿子，先遍历第一个儿子
- 如果没有第一个儿子，标志着此节点遍历完成
- 如果有弟弟遍历弟弟
- 如果有没有下一个弟弟，返回父节点标识完成父节点遍历，如果有叔叔遍历叔叔
- 没有父节点遍历结束



1.5.5.3 递归构建 fiber 树 #



1.6 树的遍历 #



1.6.1 深度优先(DFS) #

- 深度优先搜索英文缩写为 DFS 即 Depth First Search
- 其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个节点只能访问一次
- 应用场景
    - React 虚拟 DOM 的构建
    - React 的 fiber 树构建

```
function dfs(node) {
  console.log(node.name);
  node.children &&
    node.children.forEach((child) => {
      dfs(child);
    });
}
let root = {
  name: "A",
  children: [
    {
      name: "B",
      children: [{ name: "B1" }, { name: "B2" }],
    },
    {
      name: "C",
      children: [{ name: "C1" }, { name: "C2" }],
    },
  ],
};
dfs(root);
```

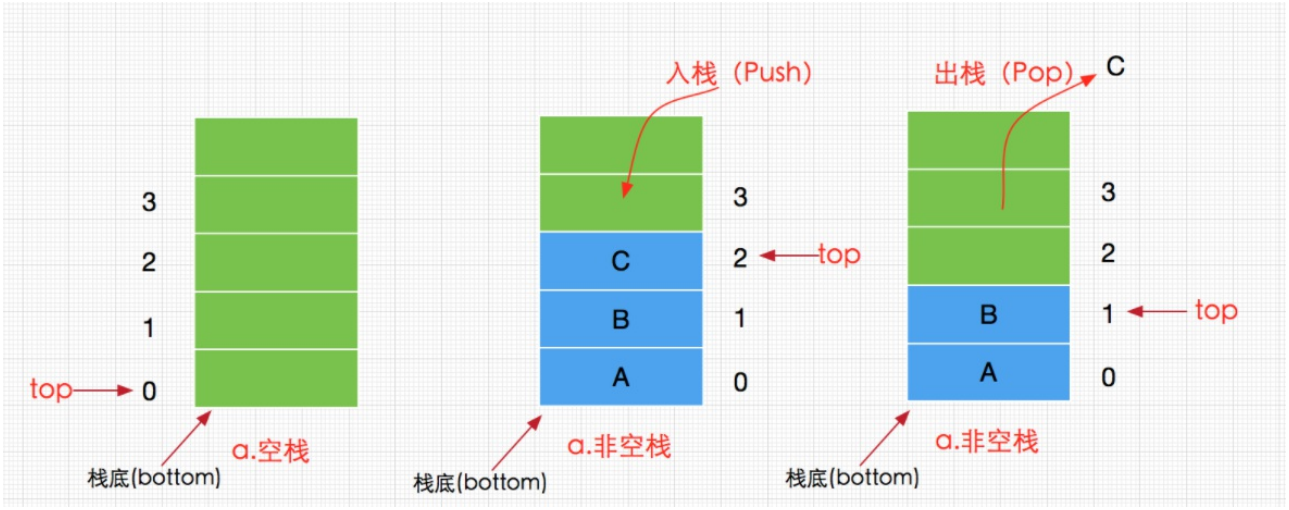
1.6.2 广度优先(BFS)

- 宽度优先搜索算法（又称广度优先搜索），其英文全称是 Breadth First Search
- 算法首先搜索距离为 k的所有顶点，然后再去搜索距离为 k+1的其他顶点

```
function bfs(node) {
  const stack = [];
  stack.push(node);
  let current;
  while ((current = stack.shift())) {
    console.log(current.name);
    current.children &&
      current.children.forEach((child) => {
        stack.push(child);
      });
  }
}
let root = {
  name: "A",
  children: [
    {
      name: "B",
      children: [{ name: "B1" }, { name: "B2" }],
    },
    {
      name: "C",
      children: [{ name: "C1" }, { name: "C2" }],
    },
  ],
};
bfs(root);
```

1.6.3 栈

- 栈（stack）又名堆栈，它是一种运算受限的线性表
- 限定仅在表尾进行插入和删除操作的线性表，这一端被称为栈顶，相对地，把另一端称为栈底
- 向一个栈插入新元素又称作进栈、入栈或压栈，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素
- 从一个栈删除元素又称作出栈或退栈，它是把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素



```
class Stack {
  constructor() {
    this.data = [];
    this.top = 0;
  }
  push(node) {
    this.data[this.top++] = node;
  }
  pop() {
    return this.data[--this.top];
  }
  peek() {
    return this.data[this.top - 1];
  }
  size() {
    return this.top;
  }
  clear() {
    this.top = 0;
  }
}

const stack = new Stack();
stack.push("1");
stack.push("2");
stack.push("3");
console.log("stack.size()", stack.size());
console.log("stack.peek", stack.peek());
console.log("stack.pop()", stack.pop());
console.log("stack.peek()", stack.peek());
stack.push("4");
console.log("stack.peek", stack.peek());
stack.clear();
console.log("stack.size", stack.size());
stack.push("5");
console.log("stack.peek", stack.peek());
```

1.7 位运算

1.7.1 比特

- 比特(bit)是表示信息的最小单位
- 比特(bit)是二进制单位(binary unit)的缩写
- 比特(bit)只有两种状态：0 和 1
- 一般来说 n 比特的信息量可以表示出 2 的 n 次方种选择


```
0b1000=2*2*2=Math.pow(2,3)=8
```

1.7.2 位运算

- Binary Bitwise Operators (https://262.ecma-international.org/5.1/#sec-11.10)
- 按位与 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Bitwise_AND)
- 按位或 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Bitwise_OR)

运算 使用 说明 按位与(&) x & y 每一个比特位都为 1 时，结果为 1，否则为 0 按位或(

) x

y 每一个比特位都为 0 时，结果为 0，否则为 1

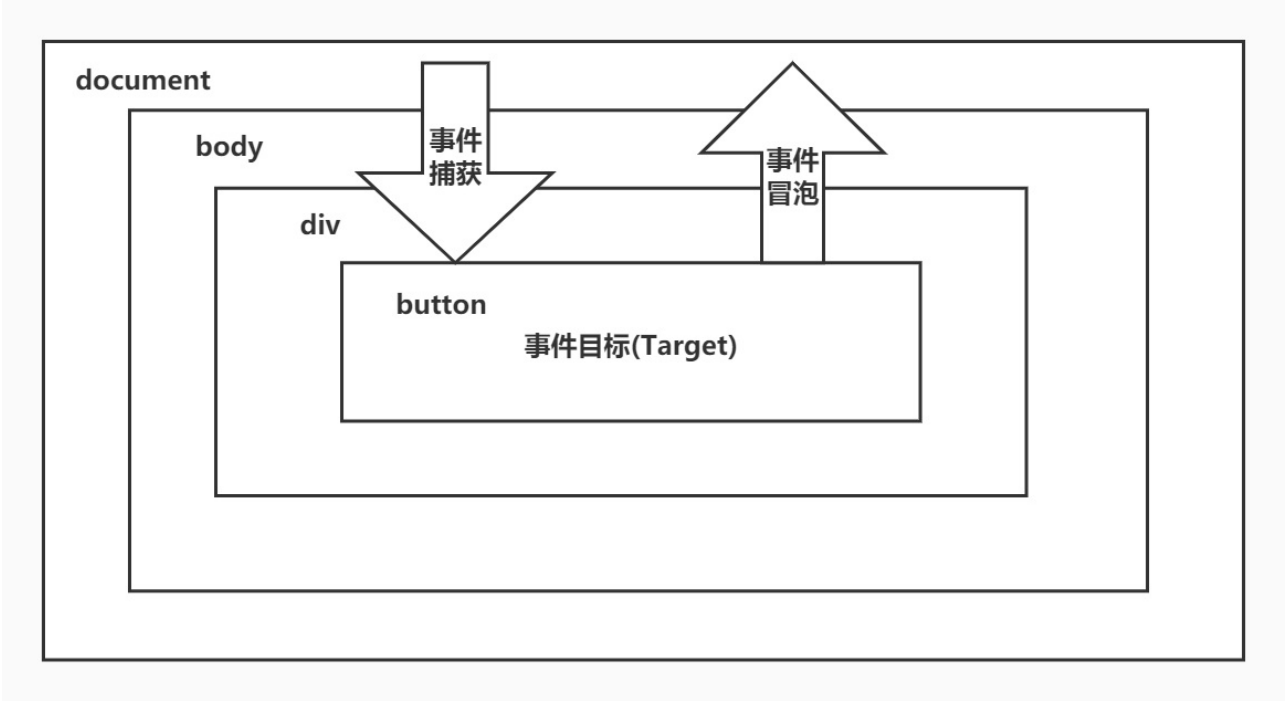
1.7.3 使用

```
const Placement = 0b001;
const Update = 0b010;

let flags = 0b000;

flags |= Placement;
flags |= Update;
console.log(flags.toString(2));

flags = flags & ~Placement;
console.log(flags.toString(2));

console.log((flags & Placement) === Placement);
console.log((flags & Update) === Update);

console.log((flags & Placement) === 0);
console.log((flags & Update) === 0);
```

1.8 事件

- 事件是用户或浏览器自身执行的某种动作，而响应某个事件的函数叫做事件处理程序

1.8.1 DOM 事件流



- 事件流包含三个阶段
    - 事件捕获阶段
    - 处于目标阶段
    - 事件冒泡阶段
- 首先发生的是事件捕获，然后是实际的目标接收到事件，最后阶段是冒泡阶段

1.8.2 事件捕获

- 是先由最上一级的节点先接收事件,然后向下传播到具体的节点 `document->body->div->button`

1.8.3 目标阶段

- 在目标节点上触发,称为目标阶段
- 事件目标是真正触发事件的对象

```
let target = event.target || event.srcElement;
```

1.8.4 事件冒泡

- 事件开始时由最具体的元素(文档中嵌套层次最深的那个节点)接收,然后逐级向上传播 `button->div->body->document`

1.8.5 addEventListener

- 任何发生在 W3C 事件模型中的事件，首是进入捕获阶段，直到达到目标元素，再进入冒泡阶段
- 可以选择是在捕获阶段还是冒泡阶段绑定事件处理函数
- `useCapture`参数是 `true`，则在捕获阶段绑定函数，反之 `false`，在冒泡阶段绑定函数

```
element.addEventListener(event, function, useCapture)
```

1.8.6 阻止冒泡

- 如果想要阻止事件的传播
    - 在微软的模型中你必须设置事件的 `cancelBubble`的属性为 true
    - 在 W3C 模型中你必须调用事件的 `stopPropagation()`方法

```
function stopPropagation(event) {
  if (!event) {
    window.event.cancelBubble = true;
  }
  if (event.stopPropagation) {
    event.stopPropagation();
  }
}
```

1.8.7 阻止默认行为

- 取消默认事件

```
function preventDefault(event) {
  if (!event) {
    window.event.returnValue = false;
  }
  if (event.preventDefault) {
    event.preventDefault();
  }
}
```

1.8.8 事件代理

- 事件代理又称之为事件委托
- 事件代理是把原本需要绑定在 `&#x5B50;&#x5143;&#x7D20;`的事件委托给 `&#x7236;&#x5143;&#x7D20;`，让父元素负责事件监听

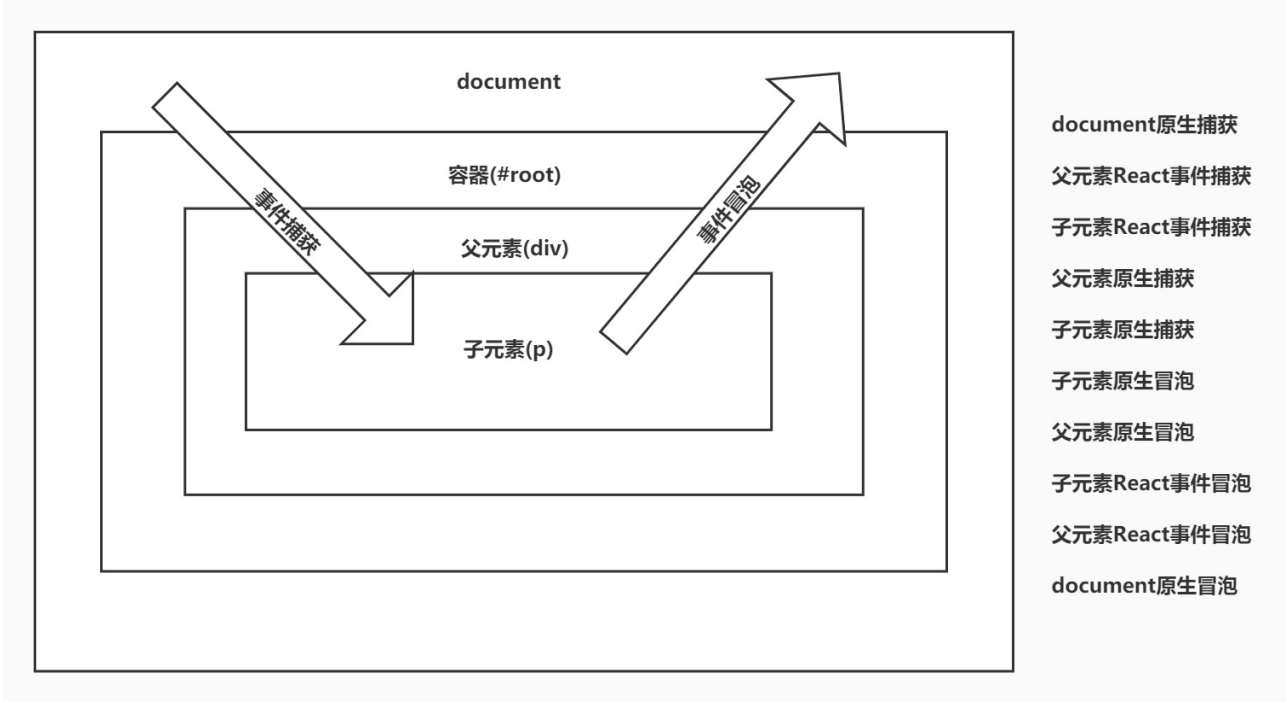
- 事件代理是利用 事件冒泡来实现的
- 优点
    - 可以大量节省内存占用，减少事件注册
    - 当新增子对象时无需再次对其绑定

```
<body>
  <ul id="list" onclick="show(event)">
    <li>item 1li>
    <li>item 2li>
    <li>item 3li>
    <li>item nli>
  ul>
  <script>
    function show(event) {
      alert(event.target.innerHTML);
    }
  script>
body>
```

1.8.9 事件系统

- 合成事件是围绕浏览器原生事件充当跨浏览器包装器的对象,它们将不同浏览器的行为合并为一个 API,这样做是为了确保事件在不同浏览器中显示一致的属性

1.8.10.1 使用



```
import * as React from "react";
import * as ReactDOM from "react-dom";
class App extends React.Component {
  parentRef = React.createRef();
  childRef = React.createRef();
  componentDidMount() {
    this.parentRef.current.addEventListener(
      "click",
      () => {
        console.log("父元素原生捕获");
      },
      true
    );
    this.parentRef.current.addEventListener("click", () => {
      console.log("父元素原生冒泡");
    });
    this.childRef.current.addEventListener(
      "click",
      () => {
        console.log("子元素原生捕获");
      },
      true
    );
    this.childRef.current.addEventListener("click", () => {
      console.log("子元素原生冒泡");
    });
    document.addEventListener(
      "click",
      () => {
        console.log("document原生捕获");
      },
      true
    );
    document.addEventListener("click", () => {
      console.log("document原生冒泡");
    });
  }
  parentBubble = () => {
    console.log("父元素React事件冒泡");
  };
  childBubble = () => {
    console.log("子元素React事件冒泡");
  };
  parentCapture = () => {
    console.log("父元素React事件捕获");
  };
  childCapture = () => {
    console.log("子元素React事件捕获");
  };
  render() {
    return (
      <div ref={this.parentRef} onClick={this.parentBubble} onClickCapture={this.parentCapture}>
        <p ref={this.childRef} onClick={this.childBubble} onClickCapture={this.childCapture}>
          事件执行顺序
        p>
      div>
    );
  }
}
ReactDOM.render(<App />, document.getElementById("root"));
```

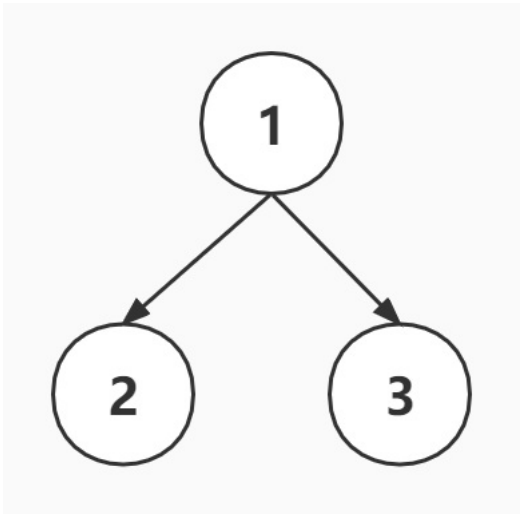
1.8.10.2 简易实现

```
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>eventtitle>
  head>
  <body>
    <div id="root">
      <div id="parent">
        <div id="child">点击div>
      div>
    div>
    <script>
      let root = document.getElementById("root");
      let parent = document.getElementById("parent");
      let child = document.getElementById("child");

      root.addEventListener("click", (event) => dispatchEvent(event, true), true);

      root.addEventListener("click", (event) => dispatchEvent(event, false), false);
      function dispatchEvent(event, isCapture) {

        let paths = [];
        let currentTarget = event.target;
        while (currentTarget) {
          paths.push(currentTarget);
          currentTarget = currentTarget.parentNode;
        }
        if (isCapture) {

          for (let i = paths.length - 1; i >= 0; i--) {

            let handler = paths[i].onClickCapture;
            handler && handler();
          }
        } else {
          for (let i = 0; i < paths.length; i++) {

            let handler = paths[i].onClick;
            handler && handler();
          }
        }
      }
      root.addEventListener("click", (event) => console.log("根元素原生事件捕获"), true);
      root.addEventListener("click", (event) => console.log("根元素原生事件冒泡"), false);
      parent.addEventListener(
        "click",
        () => {
          console.log("父元素原生事件捕获");
        },
        true
      );
      parent.addEventListener(
        "click",
        () => {
          console.log("父元素原生事件冒泡");
        },
        false
      );
      child.addEventListener(
        "click",
        () => {
          console.log("子元素原生事件捕获");
        },
        true
      );
      child.addEventListener(
        "click",
        () => {
          console.log("子元素原生事件冒泡");
        },
        false
      );
      parent.onClick = () => {
        console.log("React:父元素React事件冒泡");
      };
      parent.onClickCapture = () => {
        console.log("React:父元素React事件捕获");
      };
      child.onClick = () => {
        console.log("React:子元素React事件冒泡");
      };
      child.onClickCapture = () => {
        console.log("React:子元素React事件捕获");
      };
    script>
  body>
html>
```

1.9 最小堆

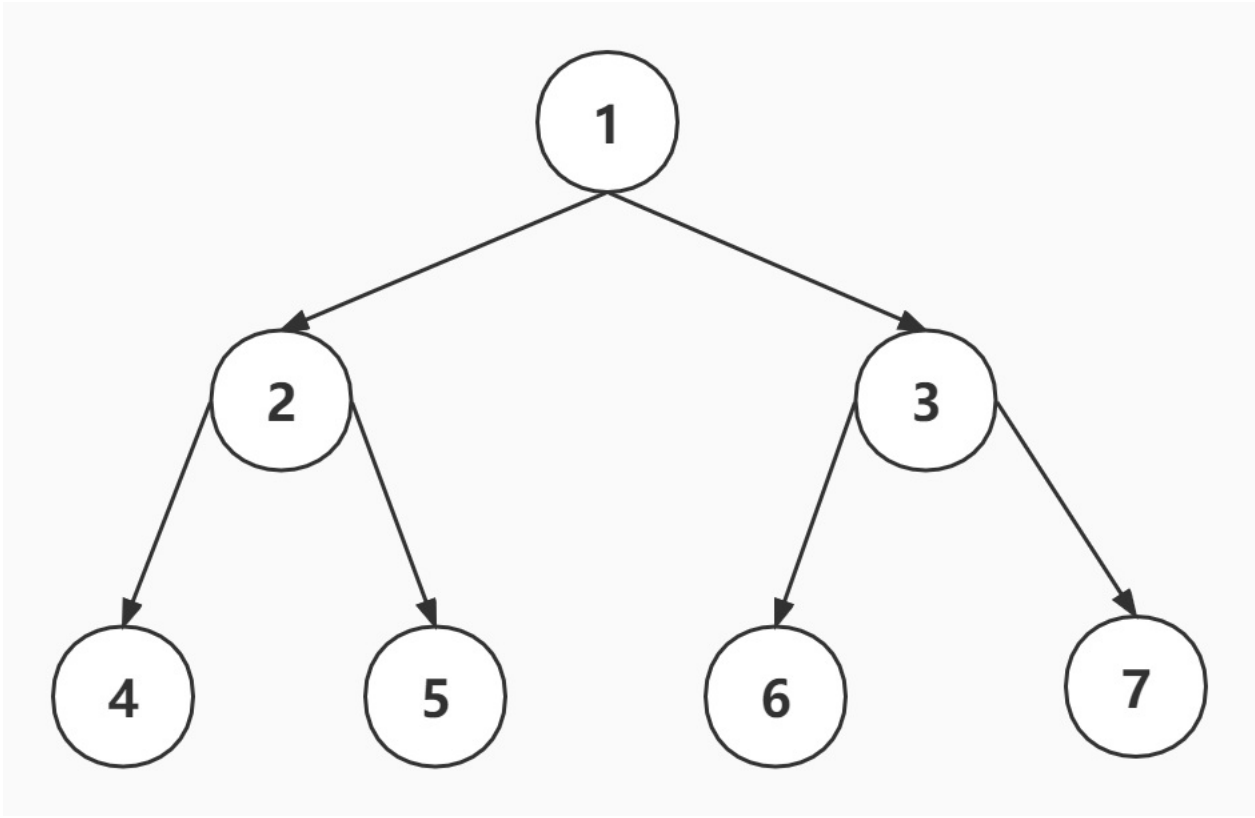
1.9.1 二叉树

- 每个节点最多有两个子节点



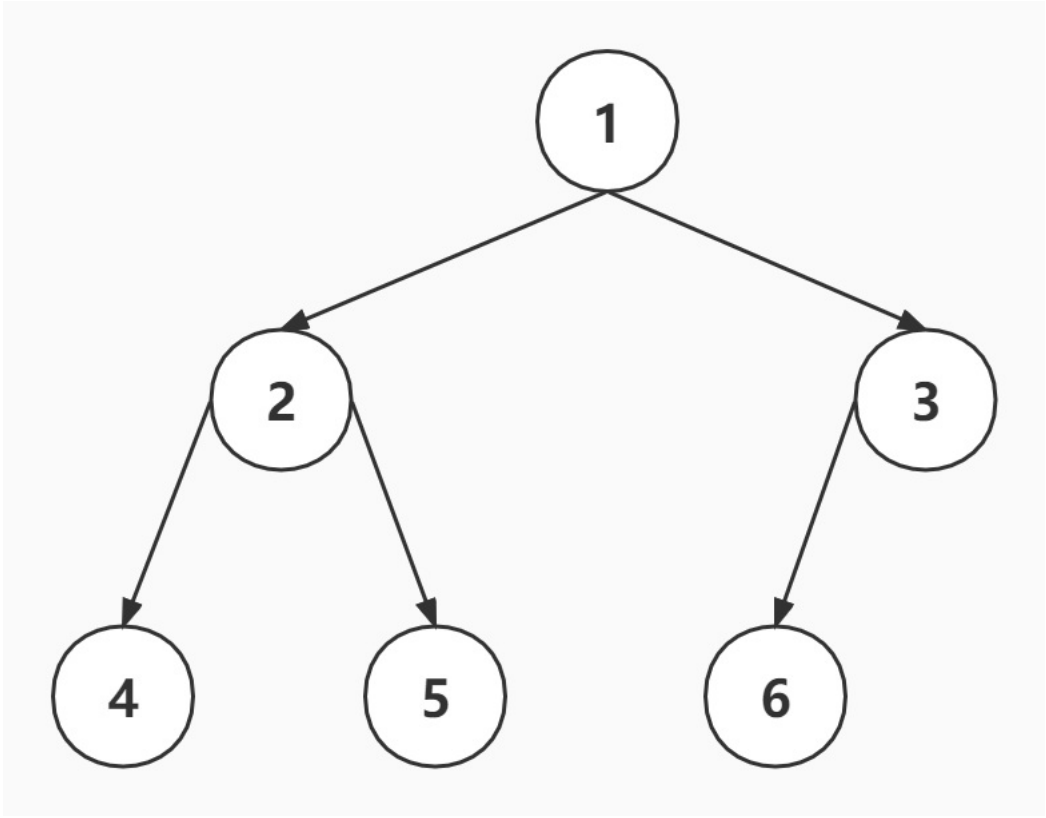
1.9.2 满二叉树

- 除最后一层无任何子节点外，每一层上的所有结点都有两个子结点的二叉树



1.9.3 完全二叉树

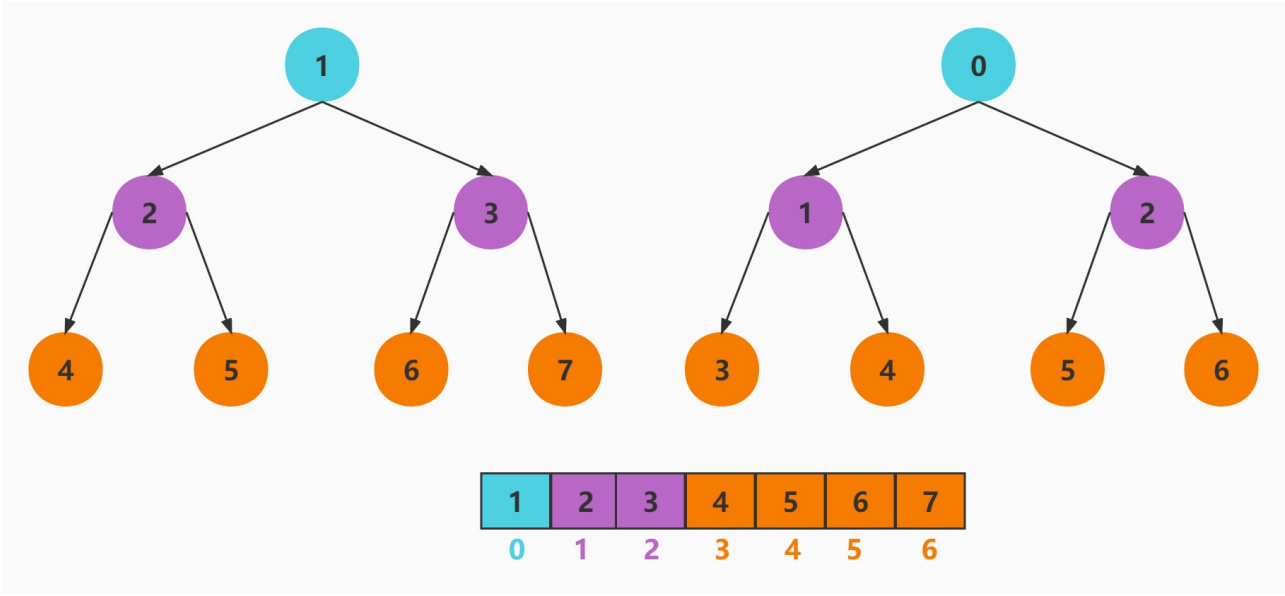
- 叶子结点只能出现在最下层和次下层
- 且最下层的叶子结点集中在树的左部



1.9.4 最小堆

- processon (https://www.processon.com/diagraming/61f26156e0b34d06c3b5bf48)
- 最小堆是一种经过排序的完全二叉树
- 其中任一非终端节点的数据值均不大于其左子节点和右子节点的值
- 根结点值是所有堆结点值中最小者
- 编号关系
    - 左子节点编号=父节点编号 _2 1_2=2
    - 右子节点编号=左子节点编号+1

    - 父节点编号=子节点编号/2 2/2=1
- 索引关系
    - 左子节点索引=(父节点索引+1) _2-1 (0+1)_2-1=1
    - 右子节点索引=左子节点索引+1
    - 父节点索引=(子节点索引-1)/2 (1-1)/2=0
- Unsigned_right_shift (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Unsigned_right_shift)



1.9.5 SchedulerMinHeap.js

- peek() 查看堆的顶点
- pop() 弹出堆的定点后需要调用 `siftDown`函数向下调整堆
- push() 添加新节点后需要调用 `siftUp`函数向上调整堆
- siftDown() 向下调整堆结构, 保证最小堆
- siftUp() 需要向上调整堆结构, 保证最小堆

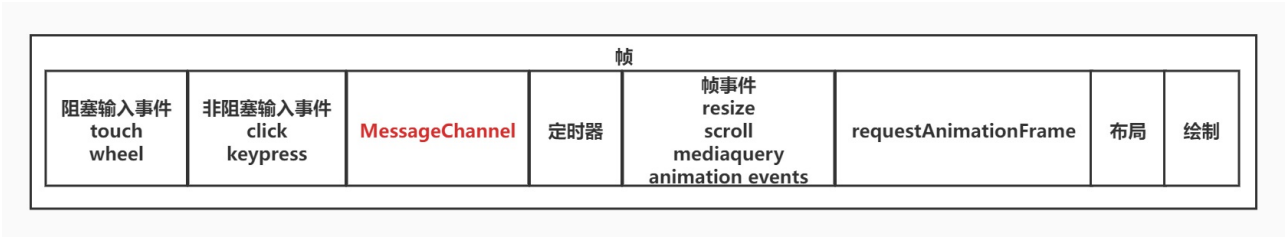
react\packages\scheduler\src\SchedulerMinHeap.js

```
export function push(heap, node) {
  const index = heap.length;
  heap.push(node);
  siftUp(heap, node, index);
}
export function peek(heap) {
  const first = heap[0];
  return first === undefined ? null : first;
}
export function pop(heap) {
  const first = heap[0];
  if (first !== undefined) {
    const last = heap.pop();
    if (last !== first) {
      heap[0] = last;
      siftDown(heap, last, 0);
    }
    return first;
  } else {
    return null;
  }
}

function siftUp(heap, node, i) {
  let index = i;
  while (true) {
    const parentIndex = index - 1 >>> 1;
    const parent = heap[parentIndex];
    if (parent !== undefined && compare(parent, node) > 0) {
      heap[parentIndex] = node;
      heap[index] = parent;
      index = parentIndex;
    } else {

      return;
    }
  }
}

function siftDown(heap, node, i) {
  let index = i;
  const length = heap.length;
  while (index < length) {
    const leftIndex = (index + 1) * 2 - 1;
    const left = heap[leftIndex];
    const rightIndex = leftIndex + 1;
    const right = heap[rightIndex];
    if (left !== undefined && compare(left, node) < 0) {
      if (right !== undefined && compare(right, left) < 0) {
        heap[index] = right;
        heap[rightIndex] = node;
        index = rightIndex;
      } else {
        heap[index] = left;
        heap[leftIndex] = node;
        index = leftIndex;
      }
    } else if (right !== undefined && compare(right, node) < 0) {
      heap[index] = right;
      heap[rightIndex] = node;
      index = rightIndex;
    } else {
      return;
    }
  }
}

function compare(a, b) {
  const diff = a.sortIndex - b.sortIndex;
  return diff !== 0 ? diff : a.id - b.id;
}
```

```
const { push, pop, peek } = require('./SchedulerMinHeap');
let heap = [];
push(heap, { sortIndex: 1 });
push(heap, { sortIndex: 2 });
push(heap, { sortIndex: 3 });
console.log(peek(heap));
push(heap, { sortIndex: 4 });
push(heap, { sortIndex: 5 });
push(heap, { sortIndex: 6 });
push(heap, { sortIndex: 7 });
console.log(peek(heap));
pop(heap);
console.log(peek(heap));
```

1.10 MessageChannel

- 目前 `requestIdleCallback` 目前只有Chrome支持
- 所以目前 React利用MessageChannel (https://developer.mozilla.org/zh-CN/docs/Web/API/MessageChannel)模拟了requestIdleCallback，将回调延迟到绘制操作之后执行
- MessageChannel API允许我们创建一个新的消息通道，并通过它的两个MessagePort属性发送数据
- MessageChannel创建了一个通信的管道，这个管道有两个端口，每个端口都可以通过postMessage发送数据，而一个端口只要绑定了onmessage回调方法，就可以接收从另一个端口传过来的数据
- MessageChannel是一个宏任务




```
var channel = new MessageChannel();
```

```
var channel = new MessageChannel();
var port1 = channel.port1;
var port2 = channel.port2
port1.onmessage = function(event) {
    console.log("port1收到来自port2的数据: " + event.data);
}
port2.onmessage = function(event) {
    console.log("port2收到来自port1的数据: " + event.data);
}
port1.postMessage("发送给port2");
port2.postMessage("发送给port1");
```

1.11 二进制

- 计算机用二进制来存储数字
- 为了简化运算，二进制数都是用一个字节(8个二进制位)来简化说明
- 在线工具 (unit.html)

1.11.1 ES5规范

- Binary Bitwise Operators (https://262.ecma-international.org/5.1/#sec-11.10)
- ToInt32: (Signed 32 Bit Integer) (https://262.ecma-international.org/5.1/#sec-9.5)
- 位运算只支持整数运算
- 位运算中的左右操作数都会转换为有符号32位整型, 且返回结果也是有符号32位整型
- 操作数的大小超过Int32范围(-2^31 ~ 2^31-1). 超过范围的二进制位会被截断, 取低位32bit

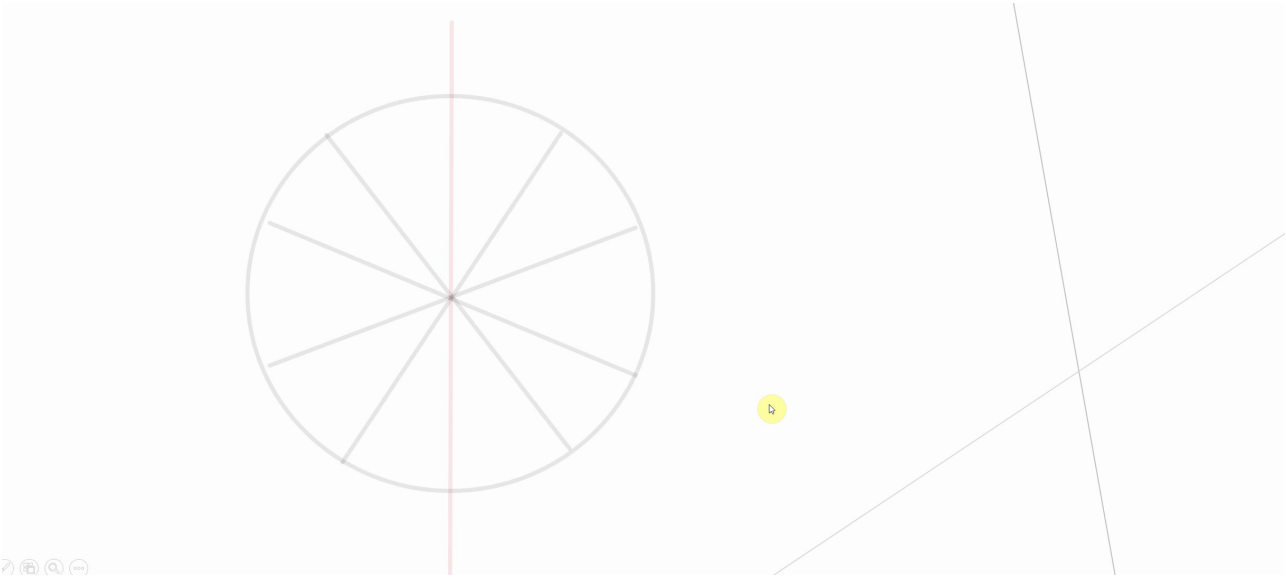
1.11.2 真值

- 8位二进制数能表示的真值范围是[-2^8, +2^8]

```
+ 00000001 # +1
- 00000001 # -1
```

1.11.3 原码

- 由于计算机只能存储0和1，不能存储正负
- 所以用8个二进制位的最高位来表示符号，0表示正，1表示负，用后七位来表示真值的绝对值
- 这种表示方法称为原码表示法，简称原码
- 由于 10000000的意思是-0，这个没有意义，所有这个数字被用来表示-128
- 由于最高位被用来表示符号了，现在能表示的范围是[-2^7, +2^7-1]，即[-128, +127]

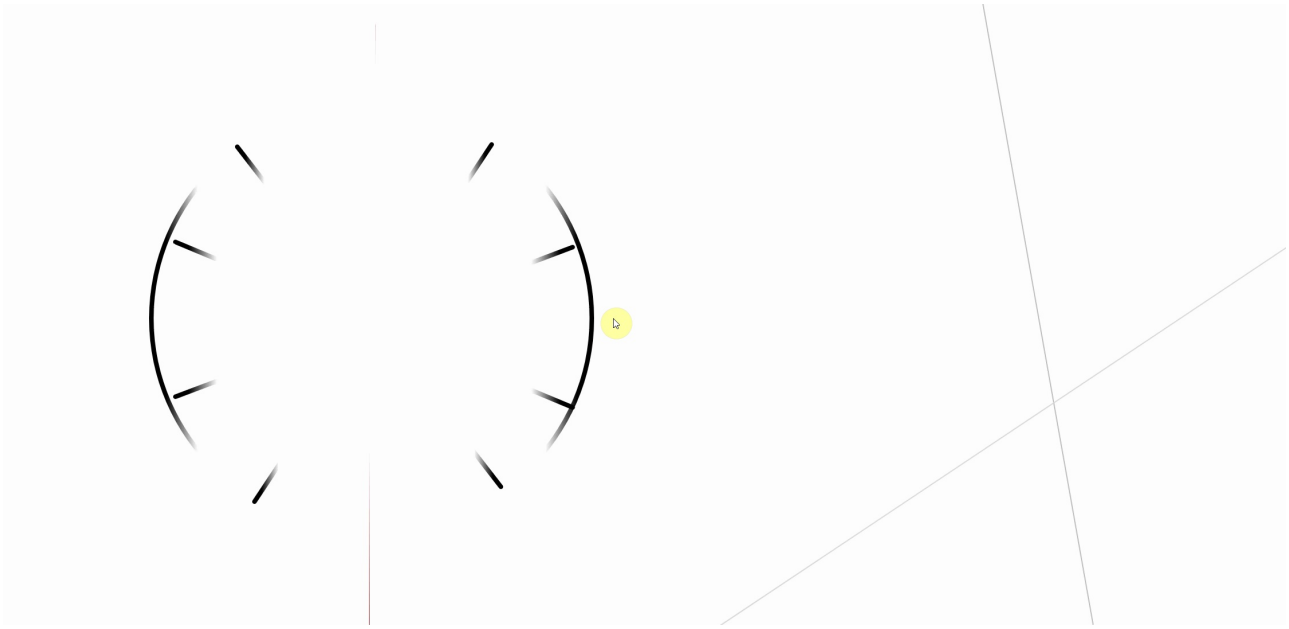


```
0 0000001 # +1
1 0000001 # -1
```

1.11.4 反码

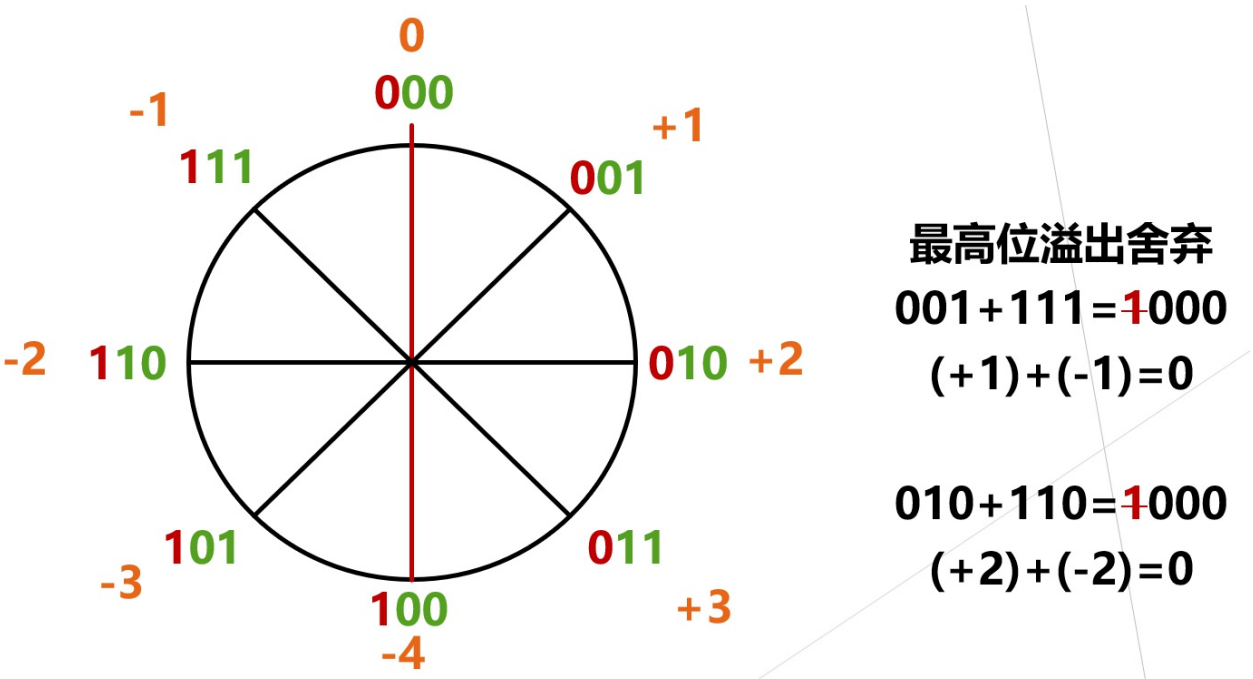
- 反码是另一种表示数字的方法
- 其规则是整数的反码何其原码一样
- 负数的反码将其原码的符号位不变，其余各位按位取反
- 反码的表示范围是[-2^7, +2^7-1]，即[-128, +127]

```
0 0000001 # +1
1 1111110 # -1
```



1.11.5 补码

- 补码是为了简化运算，将减法变为加法而发明的数字表示法
- 其规则是整数的补码和原码一样，负数的补码是其反码末尾加1
- 8位补码表示的范围是[-2^7, +2^7-1]，即[-128, +127]
- 快速计算负数补码的规则就是，由其原码低位向高位找到第一个1，1和其低位不变，1前面的高位按位取反即可




```
0 0000001 # +1
1 1111111 # -1
```

1.11.6 二进制数整数

- 只要对js中的任何数字做位运算操作系统内部都会将其转换成整形
- js中的这种整形是区分正负数的
- js中的整数的表示范围是[-2^31, +2^31-1]，即[-2147483648, +2147483647]

1.11.7 ~非

- ~操作符会将操作数的每一位取反，如果是1则变为0，如果是0则边为1

```
0b00000011
3
~0b00000011 =>  0b11111100
-4
(~0b00000011).toString();
'-4'
(~0b00000011).toString(2);
'-100'

求补码的真值
1 表示负号
剩下的 1111100 开始转换
1111100 减1
1111011 取反
0000100 4
```

1.11.8 getHighestPriorityLane

- 可以找到最右边的1
- 最右边的1右边的全是0，全是0取反就全是1，再加上就会全部进位到1取反的位置
- 最右边的1和右边的数跟原来的值是完全一样的，左边的全是反的

```
function getHighestPriorityLane(lanes) {
    return lanes & -lanes;
}

lanes=0b00001100=12
-lanes=-12
1
0001100
1110011
1110100
11110100
00001100
```

1.11.9 左移

- 左移的规则就是每一位都向左移动一位，末尾补0，其效果相当于×2
- 计算机就是用移位操作来计算乘法的

```
(0b00000010<<1).toString(2)
```

1.11.10 >> 有符号右移

- 有符号右移也就是移位的时候高位补的是其符号位，整数则补0，负数则补1

```
(-0b111>>1).toString(2) "-100"
-0b111 -7
100000111 原码
111111000 反码
111111001 补码
111111100

1
111111100
111111011
000000100
1000000100
-100
-4
```

1.11.11 >>>无符号右移

- 右移的时候高位始终补0
- 正数和有符号右移没有区别
- 负数右移后会变为正数

```
(0b111>>>1).toString(2)
>>> "11"
```

```
(-0b111>>>1).toString(2)
>>> "11111111111111111111111111111100"

00000000000000000000000000000111
11111111111111111111111111111000
11111111111111111111111111111001
01111111111111111111111111111100
2147483644
```

1.12 更新优先级

1.12.1 lane

- React中用lane(车道)模型来表示任务优先级
- 一共有31条优先级，数字越小优先级越高，某些车道的优先级相同
- clz32 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Math/clz32)函数返回开头的 0 的个数

Offscreen	Idle		IdleHydration	SelectiveHydration	Retry				Transition								TransitionHydration	Default			DefaultHydration	InputContinuous		InputContinuousHydration	InputDiscrete		InputDiscreteHydration	SyncBatched	Sync
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

```
const SyncLanePriority = 15;
const SyncBatchedLanePriority = 14;

const InputDiscreteHydrationLanePriority = 13;
const InputDiscreteLanePriority = 12;

const InputContinuousHydrationLanePriority = 11;
const InputContinuousLanePriority = 10;

const DefaultHydrationLanePriority = 9;
const DefaultLanePriority = 8;

const TransitionHydrationPriority = 7;
const TransitionPriority = 6;
const RetryLanePriority = 5;
const SelectiveHydrationLanePriority = 4;

const IdleHydrationLanePriority = 3;
const IdleLanePriority = 2;

const OffscreenLanePriority = 1;

const NoLanePriority = 0;

const TotalLanes = 31;

const NoLanes = 0b0000000000000000000000000000000;
const NoLane = 0b0000000000000000000000000000000;

const SyncLane = 0b0000000000000000000000000000001;
const SyncBatchedLane = 0b0000000000000000000000000000010;

const InputDiscreteHydrationLane = 0b0000000000000000000000000000100;
const InputDiscreteLanes = 0b0000000000000000000000000011000;

const InputContinuousHydrationLane = 0b0000000000000000000000000100000;
const InputContinuousLanes = 0b0000000000000000000000011000000;

const DefaultHydrationLane = 0b0000000000000000000000100000000;
const DefaultLanes = 0b0000000000000000000111000000000;

const TransitionHydrationLane = 0b0000000000000000001000000000000;
const TransitionLanes = 0b0000000001111111110000000000000;

const RetryLanes = 0b0000011110000000000000000000000;
const SomeRetryLane = 0b0000010000000000000000000000000;

const SelectiveHydrationLane = 0b0000100000000000000000000000000;

const NonIdleLanes = 0b0000111111111111111111111111111;
const IdleHydrationLane = 0b0001000000000000000000000000000;

const IdleLanes = 0b0110000000000000000000000000000;

const OffscreenLane = 0b1000000000000000000000000000000;

function getHighestPriorityLane(lanes) {
    return lanes & -lanes;
}

function getLowestPriorityLane(lanes) {
    const index = 31 - Math.clz32(lanes);
    return index < 0 ? NoLanes : 1 << index;
}
console.log(getLowestPriorityLane(InputDiscreteLanes));
console.log('00000000000000000000000011000');
console.log(Math.clz32(InputDiscreteLanes));
console.log(31 - Math.clz32(InputDiscreteLanes));
```

1.12.2 Hydration

- 水合反应(hydrated reaction)，也叫作水化
- 是指物质溶解在水里时，与水发生的化学作用,水合分子的过程
- 组件在服务器端拉取数据(水)，并在服务器端首次渲染
- 脱水: 对组件进行脱水，变成HTML字符串，脱去动态数据，成为风干标本快照
- 注水：发送到客户端后，重新注入数据(水)，重新变成可交互组件


这个是晒干之后的银耳
呈自然的淡黄色

1.12.3 ReactFiberLane.js

ReactFiberLane.js

```
const NoLanes = 0b00;
const NoLane = 0b00;
const SyncLane = 0b01;
const SyncBatchedLane = 0b10;
/**
 * 判断subset是不是set的子集
 * @param {*} set
 * @param {*} subset
 * @returns
 */
function isSubsetOfLanes(set, subset) {
    return (set & subset) === subset;
}
/**
 * 合并两个车道
 * @param {*} a
 * @param {*} b
 * @returns
 */
function mergeLanes(a, b) {
    return a | b;
}
module.exports = {
    NoLane,
    NoLanes,
    SyncLane,
    SyncBatchedLane,
    isSubsetOfLanes,
    mergeLanes
}
```

1.12.4 ReactUpdateQueue.js

ReactUpdateQueue.js

```
const { NoLane, NoLanes, isSubsetOfLanes, mergeLanes } = require('./ReactFiberLane');
function initializeUpdateQueue(fiber) {
  const queue = {
    baseState: fiber.memoizedState,
    firstBaseUpdate: null,
    lastBaseUpdate: null,
    shared: {

      pending: null
    }
  }
  fiber.updateQueue = queue;
}
function enqueueUpdate(fiber, update) {
  const updateQueue = fiber.updateQueue;
  if (updateQueue === null) {
    return;
  }
  const sharedQueue = updateQueue.shared;
  const pending = sharedQueue.pending;
  if (pending === null) {
    update.next = update;
  } else {
    update.next = pending.next;
    pending.next = update;
  }
  sharedQueue.pending = update;
}

function processUpdateQueue(fiber, renderLanes) {

  const queue = fiber.updateQueue;

  let firstBaseUpdate = queue.firstBaseUpdate;
  let lastBaseUpdate = queue.lastBaseUpdate;

  let pendingQueue = queue.shared.pending;
  if (pendingQueue !== null) {

    queue.shared.pending = null;
```

```
    const lastPendingUpdate = pendingQueue;

    const firstPendingUpdate = lastPendingUpdate.next;

    lastPendingUpdate.next = null;

    if (lastBaseUpdate === null) {
      firstBaseUpdate = firstPendingUpdate;
    } else {
      lastBaseUpdate.next = firstPendingUpdate;
    }

    lastBaseUpdate = lastPendingUpdate;
  }

  if (firstBaseUpdate !== null) {

    let newState = queue.baseState;
    let newLanes = NoLanes;
    let newBaseState = null;
    let newFirstBaseUpdate = null;
    let newLastBaseUpdate = null;
    let update = firstBaseUpdate;
    do {

      const updateLane = update.lane;

      if (!isSubsetOfLanes(renderLanes, updateLane)) {
        const clone = {
          id: update.id,
          lane: updateLane,
          payload: update.payload
        };
        if (newLastBaseUpdate === null) {
          newFirstBaseUpdate = newLastBaseUpdate = clone;
          newBaseState = newState;
        } else {
          newLastBaseUpdate = newLastBaseUpdate.next = clone;
        }

        newLanes = mergeLanes(newLanes, updateLane);
      } else {

        if (newLastBaseUpdate !== null) {
          const clone = {
            id: update.id,

            lane: NoLane,
            payload: update.payload
          };
          newLastBaseUpdate = newLastBaseUpdate.next = clone;
        }
        newState = getStateFromUpdate(update, newState);
      }
      update = update.next;
      if (!update) {
        break;
      }
    } while (true);

    if (!newLastBaseUpdate) {
      newBaseState = newState;
    }
    queue.baseState = newBaseState;
    queue.firstBaseUpdate = newFirstBaseUpdate;
    queue.lastBaseUpdate = newLastBaseUpdate;
    fiber.lanes = newLanes;
    fiber.memoizedState = newState;
  }
}

function getStateFromUpdate(update, prevState) {
  const payload = update.payload;
  let partialState = payload(prevState);
  return Object.assign({}, prevState, partialState);
}
module.exports = {
  initializeUpdateQueue,
  enqueueUpdate,
  processUpdateQueue
}
```

1.12.5 processUpdateQueue.js

use.js

```
const { initializeUpdateQueue, enqueueUpdate, processUpdateQueue } = require('./ReactUpdateQueue');
const { SyncBatchedLane, SyncLane } = require('./ReactFiberLane');

let fiber = { memoizedState: { msg: '' } };
initializeUpdateQueue(fiber);
let update1 = { id: 'A', payload: (state) => ({ msg: state.msg + 'A' }), lane: SyncBatchedLane };
enqueueUpdate(fiber, update1);
let update2 = { id: 'B', payload: (state) => ({ msg: state.msg + 'B' }), lane: SyncLane };
enqueueUpdate(fiber, update2);
let update3 = { id: 'C', payload: (state) => ({ msg: state.msg + 'C' }), lane: SyncBatchedLane };
enqueueUpdate(fiber, update3);
let update4 = { id: 'D', payload: (state) => ({ msg: state.msg + 'D' }), lane: SyncLane };
enqueueUpdate(fiber, update4);

processUpdateQueue(fiber, SyncLane);
console.log('memoizedState', fiber.memoizedState);
console.log('updateQueue', printQueue(fiber.updateQueue));

let update5 = { id: 'E', payload: (state) => ({ msg: state.msg + 'E' }), lane: SyncLane };
enqueueUpdate(fiber, update5);
processUpdateQueue(fiber, SyncLane);
console.log('memoizedState', fiber.memoizedState);
console.log('updateQueue', printQueue(fiber.updateQueue));

processUpdateQueue(fiber, SyncBatchedLane);
console.log('memoizedState', fiber.memoizedState);
console.log('updateQueue', printQueue(fiber.updateQueue));

let update6 = { id: 'F', payload: (state) => ({ msg: state.msg + 'F' }), lane: SyncLane };
enqueueUpdate(fiber, update6);
processUpdateQueue(fiber, SyncLane);
console.log('memoizedState', fiber.memoizedState);
console.log('updateQueue', printQueue(fiber.updateQueue));

function printQueue(queue) {
  const { baseState, firstBaseUpdate } = queue
  let state = baseState.msg + '|';
  let update = firstBaseUpdate;
  while (update) {
    state += ((update.id) + "=>");
    update = update.next;
  }
  state += "null";
  console.log(state);
}
```

2.创建项目

- vitejs (https://cn.vitejs.dev/)

2.1 创建目录

```
mkdir react182
cd react182
npm init -y
```

2.2 安装

```
npm install vite @vitejs/plugin-react --save
```

2.3 vite.config.js

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";
import path from "path";

export default defineConfig({
  define: {
    __DEV__: true,
    __PROFILE__: true,
    __UMD__: true,
    __EXPERIMENTAL__: true,
  },
  resolve: {
    alias: {
      react: path.posix.resolve("src/react"),
      "react-dom": path.posix.resolve("src/react-dom"),
      "react-dom-bindings": path.posix.resolve("src/react-dom-bindings"),
      "react-reconciler": path.posix.resolve("src/react-reconciler"),
      scheduler: path.posix.resolve("src/scheduler"),
      shared: path.posix.resolve("src/shared"),
    },
  },
  plugins: [react()],
  optimizeDeps: {
    force: true,
  },
});
```

2.4 jsconfig.json

jsconfig.json

```
{
  "compilerOptions": {
    "baseUrl": "./",
    "paths": {
      "react/*": ["src/react/*"],
      "react-dom/*": ["src/react-dom/*"],
      "react-dom-bindings/*": ["react-dom-bindings/*"],
      "react-reconciler/*": ["src/react-reconciler/*"],
      "scheduler/*": ["scheduler/*"],
      "shared/*": ["src/shared/*"]
    }
  },
  "exclude": ["node_modules", "dist"]
}
```

2.5 main.jsx

src\main.jsx

```
console.log("main");
```

2.6 index.html

index.html

```
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>React18title>
  head>

  <body>
    <div id="root">div>
    <script type="module" src="/src/main.jsx">script>
  body>
html>
```

2.7 package.json

package.json

```
{
  "scripts": {
    "dev": "vite"
  }
}
```

3.实现虚拟 DOM

3.1 main.jsx

main.jsx

```
let element = (
  <h1>
    hello<span style={{ color: "red" }}>worldspan>
  h1>
);
console.log(element);
```

3.2 jsx-dev-runtime.js

src\react\jsx-dev-runtime.js

```
export { jsxDEV } from "./src/jsx/ReactJSXElement";
```

3.3 ReactJSXElement.js

src\react\src\jsx\ReactJSXElement.js

```
import hasOwnProperty from "shared/hasOwnProperty";
import { REACT_ELEMENT_TYPE } from "shared/ReactSymbols";

const RESERVED_PROPS = {
  key: true,
  ref: true,
  __self: true,
  __source: true,
};
function hasValidRef(config) {
  return config.ref !== undefined;
}

const ReactElement = (type, key, ref, props) => {
  const element = {
    $typeof: REACT_ELEMENT_TYPE,
    type,
    key,
    ref,
    props,
  };
  return element;
};

export function jsxDEV(type, config, maybeKey) {
  let propName;
  const props = {};
  let key = null;
  let ref = null;

  if (maybeKey !== undefined) {
    key = "" + maybeKey;
  }

  if (hasValidRef(config)) {
    ref = config.ref;
  }

  for (propName in config) {
    if (hasOwnProperty.call(config, propName) && !RESERVED_PROPS.hasOwnProperty(propName)) {
      props[propName] = config[propName];
    }
  }
  return ReactElement(type, key, ref, props);
}
```

3.4 ReactSymbols.js

src\shared\ReactSymbols.js

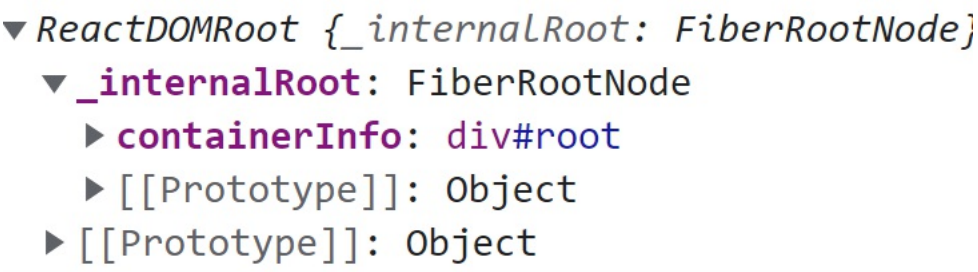
```
export const REACT_ELEMENT_TYPE = Symbol.for("react.element");
```

3.5 hasOwnProperty.js

src\shared\hasOwnProperty.js

```
const { hasOwnProperty } = Object.prototype;
export default hasOwnProperty;
```

4.创建 ReactDOMRoot



4.1 main.jsx

src\main.jsx

```
+import { createRoot } from "react-dom/client";
let element = (

    helloworld

);
+const root = createRoot(document.getElementById("root"));
+console.log(root);
```

4.2 client.js

src\react-dom\client.js

```
export { createRoot } from "./src/client/ReactDOMRoot";
```

4.3 ReactDOMRoot.js

src\react-dom\src\client\ReactDOMRoot.js

```
import { createContainer } from "react-reconciler/src/ReactFiberReconciler";

function ReactDOMRoot(internalRoot) {
  this._internalRoot = internalRoot;
}

export function createRoot(container) {
  const root = createContainer(container);
  return new ReactDOMRoot(root);
}
```

4.4 ReactFiberReconciler.js

src\react-reconciler\src\ReactFiberReconciler.js

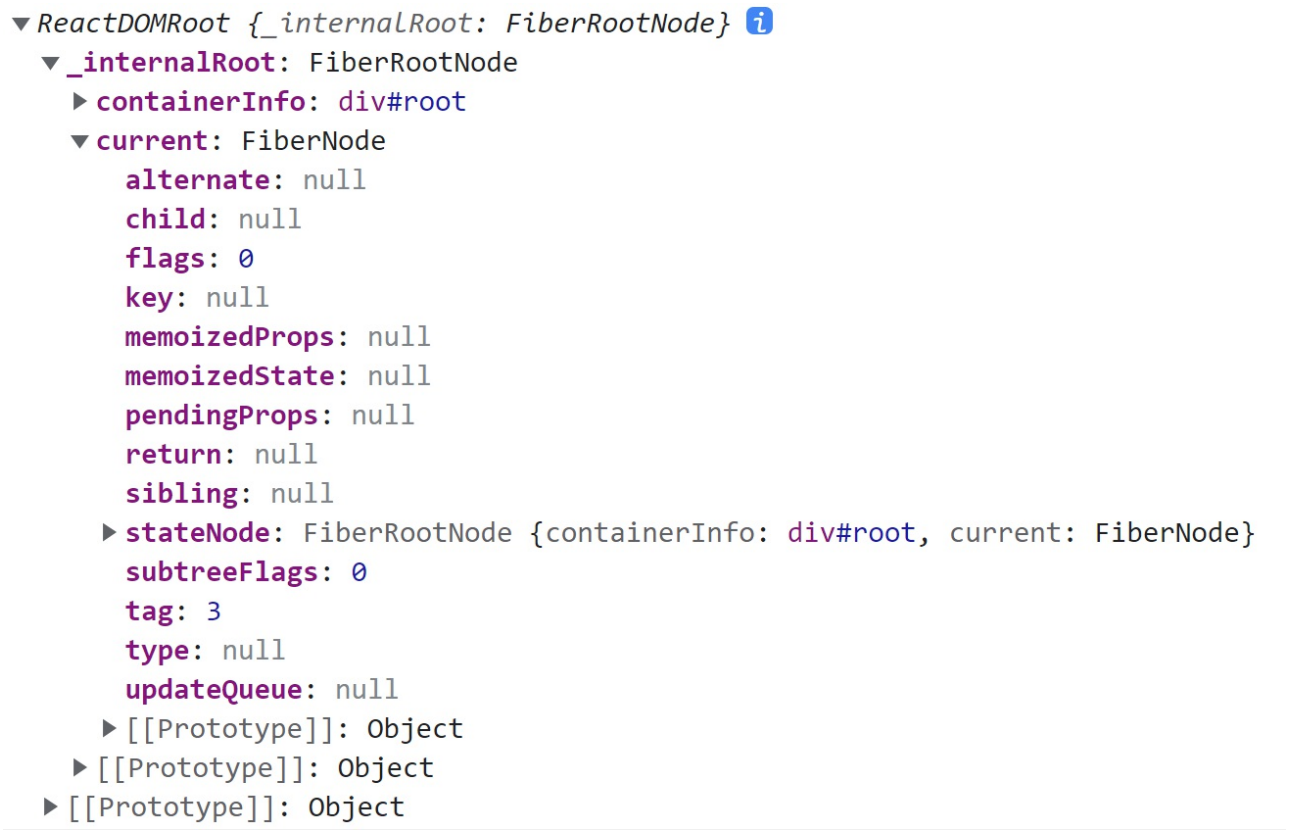
```
import { createFiberRoot } from "./ReactFiberRoot";
export function createContainer(containerInfo) {
  return createFiberRoot(containerInfo);
}
```

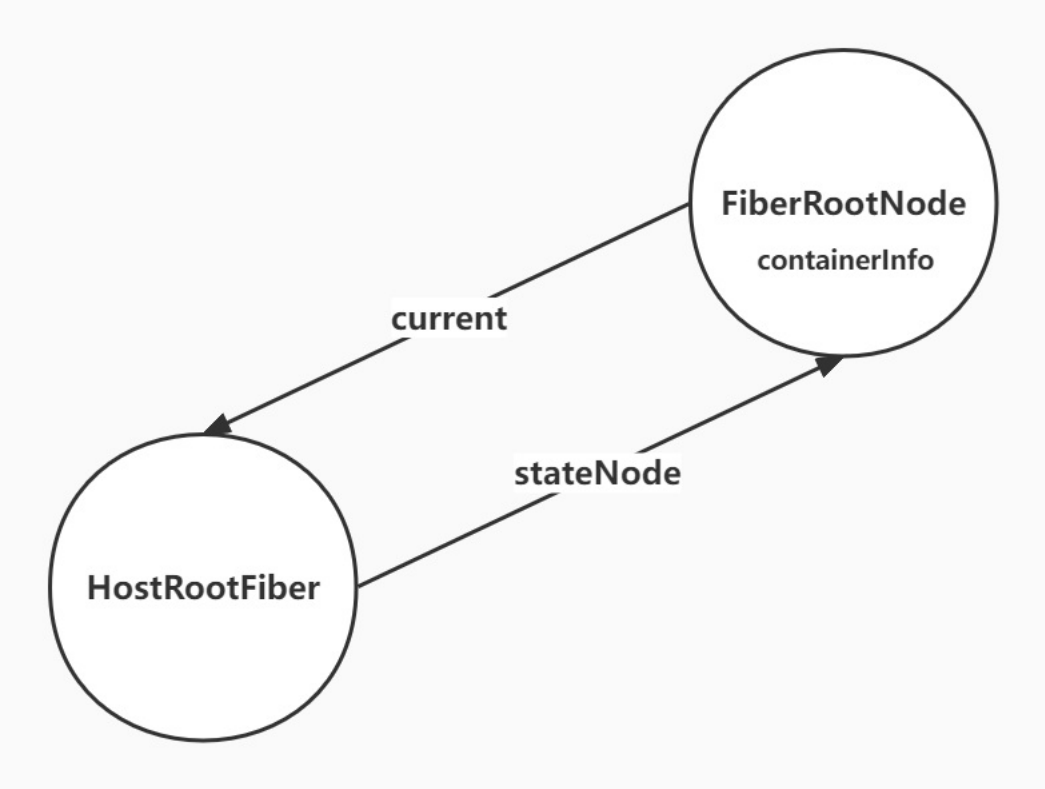
4.5 ReactFiberRoot.js

src\react-reconciler\src\ReactFiberRoot.js

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

export function createFiberRoot(containerInfo) {
  const root = new FiberRootNode(containerInfo);
  return root;
}
```

5.创建 RootFiber





5.1 ReactFiberRoot.js

src\react-reconciler\src\ReactFiberRoot.js

```
+import { createHostRootFiber } from "./ReactFiber";
function FiberRootNode(containerInfo) {
  this.containerInfo = containerInfo;
}

export function createFiberRoot(containerInfo) {
  const root = new FiberRootNode(containerInfo);
+ const uninitializedFiber = createHostRootFiber();
+ root.current = uninitializedFiber;
+ uninitializedFiber.stateNode = root;
  return root;
}
```

5.2 ReactFiber.js

src\react-reconciler\src\ReactFiber.js

```
import { HostRoot } from "./ReactWorkTags";
import { NoFlags } from "./ReactFiberFlags";
export function FiberNode(tag, pendingProps, key) {
  this.tag = tag;
  this.key = key;
  this.type = null;
  this.stateNode = null;

  this.return = null;
  this.child = null;
  this.sibling = null;

  this.pendingProps = pendingProps;
  this.memoizedProps = null;
  this.updateQueue = null;
  this.memoizedState = null;

  this.flags = NoFlags;
  this.subtreeFlags = NoFlags;
  this.alternate = null;
}
function createFiber(tag, pendingProps, key) {
  return new FiberNode(tag, pendingProps, key);
}
export function createHostRootFiber() {
  return createFiber(HostRoot, null, null);
}
```

src\react-reconciler\src\ReactWorkTags.js

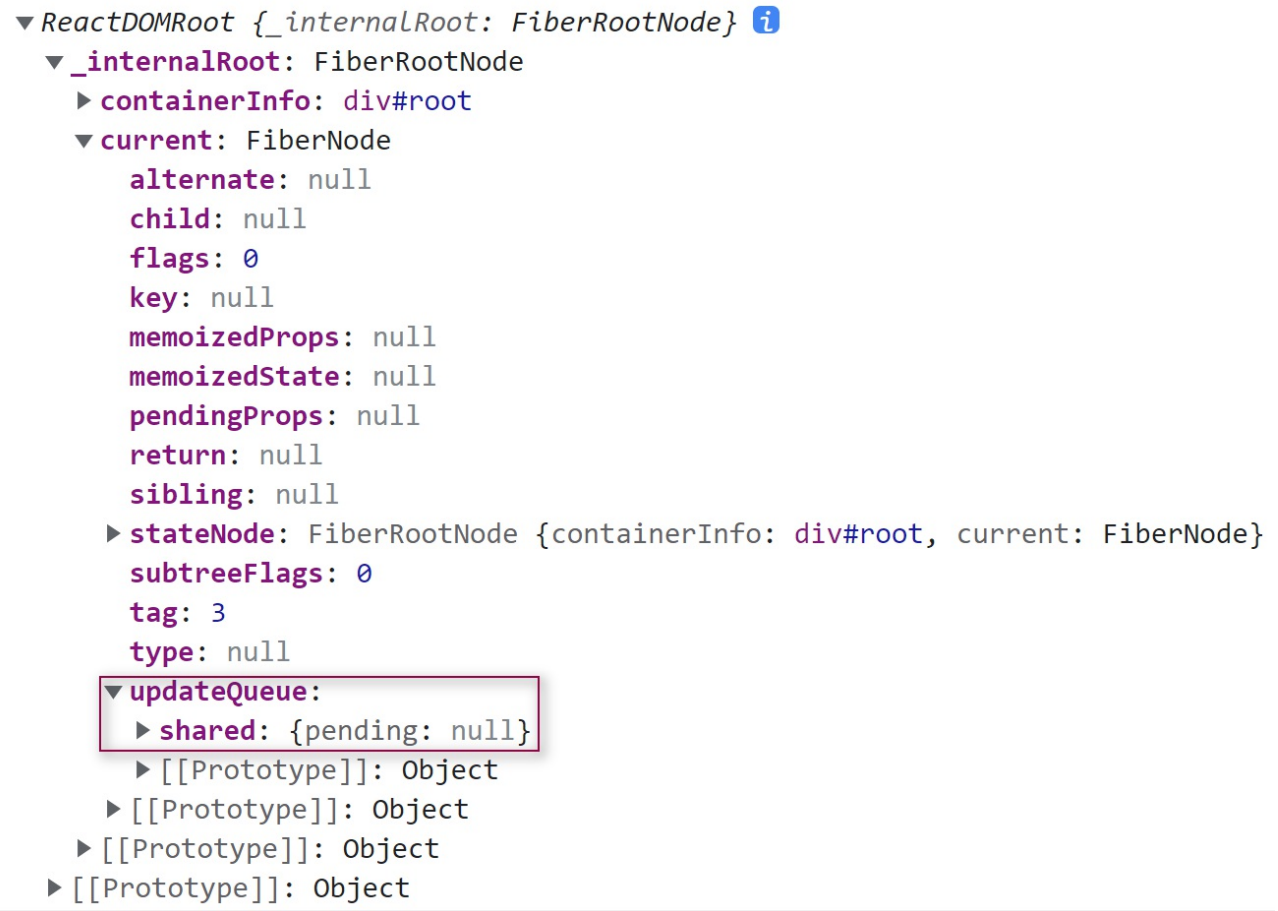
```
export const HostRoot = 3;
```

5.4 ReactFiberFlags.js

src\react-reconciler\src\ReactFiberFlags.js

```
export const NoFlags = 0b00000000000000000000000000;
```

6.初始化 UpdateQueue



6.1 ReactFiberRoot.js

src\react-reconciler\src\ReactFiberRoot.js

```
import { createHostRootFiber } from "./ReactFiber";
+import { initializeUpdateQueue } from "./ReactFiberClassUpdateQueue";
function FiberRootNode(containerInfo) {
  this.containerInfo = containerInfo;
}

export function createFiberRoot(containerInfo) {
  const root = new FiberRootNode(containerInfo);
  const uninitializedFiber = createHostRootFiber();
  root.current = uninitializedFiber;
  uninitializedFiber.stateNode = root;
+ initializeUpdateQueue(uninitializedFiber);
  return root;
}
```

6.2 ReactFiberClassUpdateQueue.js

src\react-reconciler\src\ReactFiberClassUpdateQueue.js

```
export function initializeUpdateQueue(fiber) {
  const queue = {
    shared: {
      pending: null,
    },
  };
  fiber.updateQueue = queue;
}
```

7. enqueueUpdate

7.1 main.jsx

src\main.jsx

```
import { createRoot } from "react-dom/client";
let element = (

    helloworld

);
const root = createRoot(document.getElementById("root"));
+root.render(element);
```

7.2 ReactDOMRoot.js

src\react-dom\src\client\ReactDOMRoot.js

```
import {
  createContainer,
+ updateContainer,
} from "react-reconciler/src/ReactFiberReconciler";

function ReactDOMRoot(internalRoot) {
  this._internalRoot = internalRoot;
}
+ReactDOMRoot.prototype.render = function render(children) {
+  const root = this._internalRoot;
+  root.containerInfo.innerHTML = "";
+  updateContainer(children, root);
+};
export function createRoot(container) {
  const root = createContainer(container);
  return new ReactDOMRoot(root);
}
```

7.3 ReactFiberReconciler.js

src\react-reconciler\src\ReactFiberReconciler.js

```
import { createFiberRoot } from "./ReactFiberRoot";
+import { createUpdate, enqueueUpdate } from "./ReactFiberClassUpdateQueue";
export function createContainer(containerInfo) {
  return createFiberRoot(containerInfo);
}
+export function updateContainer(element, container) {
+  const current = container.current;
+  const update = createUpdate();
+  update.payload = { element };
+  const root = enqueueUpdate(current, update);
+  console.log(root);
+}
```

7.4 ReactFiberClassUpdateQueue.js

src\react-reconciler\src\ReactFiberClassUpdateQueue.js

```
+import { markUpdateLaneFromFiberToRoot } from "./ReactFiberConcurrentUpdates";
+export const UpdateState = 0;
export function initializeUpdateQueue(fiber) {
  const queue = {
    shared: {
      pending: null,
    },
  };
  fiber.updateQueue = queue;
}
+export function createUpdate() {
+  const update = { tag: UpdateState };
+  return update;
+}
+export function enqueueUpdate(fiber, update) {
+  const updateQueue = fiber.updateQueue;
+  const sharedQueue = updateQueue.shared;
+  const pending = sharedQueue.pending;
+  if (pending === null) {
+    update.next = update;
+  } else {
+    update.next = pending.next;
+    pending.next = update;
+  }
+  updateQueue.shared.pending = update;
+  return markUpdateLaneFromFiberToRoot(fiber);
+}
```

7.5 ReactFiberConcurrentUpdates.js

src\react-reconciler\src\ReactFiberConcurrentUpdates.js

```
import { HostRoot } from "./ReactWorkTags";
export function markUpdateLaneFromFiberToRoot(sourceFiber) {
  let node = sourceFiber;
  let parent = sourceFiber.return;
  while (parent !== null) {
    node = parent;
    parent = parent.return;
  }
  if (node.tag === HostRoot) {
    const root = node.stateNode;
    return root;
  }
  return null;
}
```

8.performConcurrentWorkOnRoot

8.1 ReactFiberReconciler.js

src\react-reconciler\src\ReactFiberReconciler.js

```
import { createFiberRoot } from "./ReactFiberRoot";
import { createUpdate, enqueueUpdate } from "./ReactFiberClassUpdateQueue";
+import { scheduleUpdateOnFiber } from "./ReactFiberWorkLoop";
export function createContainer(containerInfo) {
  return createFiberRoot(containerInfo);
}
export function updateContainer(element, container) {
  const current = container.current;
  const update = createUpdate();
  update.payload = { element };
  const root = enqueueUpdate(current, update);
+ scheduleUpdateOnFiber(root);
}
```

8.2 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import { scheduleCallback } from "scheduler";
export function scheduleUpdateOnFiber(root) {
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  scheduleCallback(performConcurrentWorkOnRoot.bind(null, root));
}
function performConcurrentWorkOnRoot(root) {
  console.log("performConcurrentWorkOnRoot");
}
```

8.3 scheduler\index.js

src\scheduler\index.js

```
export * from "./src/forks/Scheduler";
```

8.4 Scheduler.js

src\scheduler\src\forks\Scheduler.js

```
export function scheduleCallback(callback) {
  requestIdleCallback(callback);
}
```

9.prepareFreshStack

9.1 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import { scheduleCallback } from "scheduler";
+import { createWorkInProgress } from "./ReactFiber";
+let workInProgress = null;
export function scheduleUpdateOnFiber(root) {
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  scheduleCallback(performConcurrentWorkOnRoot.bind(null, root));
}
function performConcurrentWorkOnRoot(root) {
+ renderRootSync(root);
}
+function prepareFreshStack(root) {
+  workInProgress = createWorkInProgress(root.current, null);
+  console.log(workInProgress);
+}
+function renderRootSync(root) {
+  prepareFreshStack(root);
+}
```

9.2 ReactFiber.js

src\react-reconciler\src\ReactFiber.js

```
import { HostRoot } from "./ReactWorkTags";
import { NoFlags } from "./ReactFiberFlags";
export function FiberNode(tag, pendingProps, key) {
  this.tag = tag;
  this.key = key;
  this.type = null;
  this.stateNode = null;

  this.return = null;
  this.child = null;
  this.sibling = null;

  this.pendingProps = pendingProps;
  this.memoizedProps = null;
  this.updateQueue = null;
  this.memoizedState = null;

  this.flags = NoFlags;
  this.subtreeFlags = NoFlags;
  this.alternate = null;
}
function createFiber(tag, pendingProps, key) {
  return new FiberNode(tag, pendingProps, key);
}
export function createHostRootFiber() {
  return createFiber(HostRoot, null, null);
}
+// We use a double buffering pooling technique because we know that we'll
+// only ever need at most two versions of a tree. We pool the "other" unused
+// node that we're free to reuse. This is lazily created to avoid allocating
+// extra objects for things that are never updated. It also allow us to
+// reclaim the extra memory if needed.

+//我们使用双缓冲池技术，因为我们知道一棵树最多只需要两个版本
+//我们将"其他"未使用的我们可以自由重用的节点
+//这是延迟创建的，以避免分配从未更新的内容的额外对象。它还允许我们如果需要，回收额外的内+存
+export function createWorkInProgress(current, pendingProps) {
+  let workInProgress = current.alternate;
+  if (workInProgress === null) {
+    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.type = current.type;
+    workInProgress.flags = NoFlags;
+    workInProgress.subtreeFlags = NoFlags;
+  }
+  workInProgress.child = current.child;
+  workInProgress.memoizedProps = current.memoizedProps;
+  workInProgress.memoizedState = current.memoizedState;
+  workInProgress.updateQueue = current.updateQueue;
+  workInProgress.sibling = current.sibling;
+  workInProgress.index = current.index;
+  return workInProgress;
+}
```

10.beginWork

```
beginWork ▶ FiberNode {tag: 3, key: null, type: null, stateNode: FiberRootNode, return: null, …}
beginWork ▶ FiberNode {tag: 5, key: null, type: 'h1', stateNode: null, return: FiberNode, …}
beginWork ▶ FiberNode {tag: 6, key: null, type: null, stateNode: null, return: FiberNode, …}
```

10.1 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import { scheduleCallback } from "scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
let workInProgress = null;
export function scheduleUpdateOnFiber(root) {
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  scheduleCallback(performConcurrentWorkOnRoot.bind(null, root));
}
function performConcurrentWorkOnRoot(root) {
  renderRootSync(root);
}
function prepareFreshStack(root) {
  workInProgress = createWorkInProgress(root.current, null);

}
function renderRootSync(root) {
  prepareFreshStack(root);
+ workLoopSync();
}

+function workLoopSync() {
+  while (workInProgress !== null) {
+    performUnitOfWork(workInProgress);
+  }
+}
+function performUnitOfWork(unitOfWork) {
+  const current = unitOfWork.alternate;
+  const next = beginWork(current, unitOfWork);
+  unitOfWork.memoizedProps = unitOfWork.pendingProps;
+  if (next === null) {
+    //completeUnitOfWork(unitOfWork);
+    workInProgress = null;
+  } else {
+    workInProgress = next;
+  }
+}
```

10.2 ReactFiberBeginWork.js

src\react-reconciler\src\ReactFiberBeginWork.js

```
import { HostRoot, HostComponent, HostText } from "./ReactWorkTags";
import { processUpdateQueue } from "./ReactFiberClassUpdateQueue";
import { mountChildFibers, reconcileChildFibers } from "./ReactChildFiber";
import { shouldSetTextContent } from "react-dom-bindings/src/client/ReactDOMHostConfig";
import logger, { indent } from "shared/logger";
function reconcileChildren(current, workInProgress, nextChildren) {
  if (current === null) {
    workInProgress.child = mountChildFibers(workInProgress, null, nextChildren);
  } else {
    workInProgress.child = reconcileChildFibers(workInProgress, current.child, nextChildren);
  }
}
function updateHostRoot(current, workInProgress) {
  processUpdateQueue(workInProgress);
  const nextState = workInProgress.memoizedState;
  const nextChildren = nextState.element;
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
function updateHostComponent(current, workInProgress) {
  const { type } = workInProgress;
  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 beginWork(current, workInProgress) {
  logger(" ".repeat(indent.number) + "beginWork", workInProgress);
  switch (workInProgress.tag) {
    case HostRoot:
      return updateHostRoot(current, workInProgress);
    case HostComponent:
      return updateHostComponent(current, workInProgress);
    case HostText:
    default:
      return null;
  }
}
```

src\react-reconciler\src\ReactWorkTags.js

```
export const HostRoot = 3;
+export const IndeterminateComponent = 2;
+export const HostComponent = 5;
+export const HostText = 6;
```

10.4 ReactFiberClassUpdateQueue.js

src\react-reconciler\src\ReactFiberClassUpdateQueue.js

```
import { markUpdateLaneFromFiberToRoot } from "./ReactFiberConcurrentUpdates";
+import assign from "shared/assign";
export const UpdateState = 0;
export function initializeUpdateQueue(fiber) {
  const queue = {
    shared: {
      pending: null,
    },
  };
  fiber.updateQueue = queue;
}
export function createUpdate() {
  const update = { tag: UpdateState };
  return update;
}
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;
  }
  updateQueue.shared.pending = update;
  return markUpdateLaneFromFiberToRoot(fiber);
}

+function getStateFromUpdate(update, prevState) {
+  switch (update.tag) {
+    case UpdateState: {
+      const { payload } = update;
+      const partialState = payload;
+      return assign({}, prevState, partialState);
+    }
+    default:
+      return prevState;
+  }
+}
+export function processUpdateQueue(workInProgress) {
+  const queue = workInProgress.updateQueue;
+  const pendingQueue = queue.shared.pending;
+  if (pendingQueue !== null) {
+    queue.shared.pending = null;
+    const lastPendingUpdate = pendingQueue;
+    const firstPendingUpdate = lastPendingUpdate.next;
+    lastPendingUpdate.next = null;
+    let newState = workInProgress.memoizedState;
+    let update = firstPendingUpdate;
+    while (update) {
+      newState = getStateFromUpdate(update, newState);
+      update = update.next;
+    }
+    workInProgress.memoizedState = newState;
+  }
+}
```

10.5 ReactChildFiber.js

src\react-reconciler\src\ReactChildFiber.js

```
import { REACT_ELEMENT_TYPE } from "shared/ReactSymbols";
import isArray from "shared/isArray";
import { createFiberFromElement, FiberNode, createFiberFromText } from "./ReactFiber";
import { Placement } from "./ReactFiberFlags";
import { HostText } from "./ReactWorkTags";
function createChildReconciler(shouldTrackSideEffects) {
  function reconcileSingleElement(returnFiber, currentFirstChild, element) {
    const created = createFiberFromElement(element);
    created.return = returnFiber;
    return created;
  }
  function placeSingleChild(newFiber) {
    if (shouldTrackSideEffects) newFiber.flags |= Placement;
    return newFiber;
  }
  function reconcileSingleTextNode(returnFiber, currentFirstChild, content) {
    const created = new FiberNode(HostText, { content }, null);
    created.return = returnFiber;
    return created;
  }
  function createChild(returnFiber, newChild) {
    if ((typeof newChild === "string" && newChild !== "") || typeof newChild === "number") {
      const created = createFiberFromText(`${newChild}`);
      created.return = returnFiber;
      return created;
    }

    if (typeof newChild === "object" && newChild !== null) {
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          const created = createFiberFromElement(newChild);
          created.return = returnFiber;
          return created;
        }
        default:
          break;
      }
    }
    return null;
  }
  function placeChild(newFiber, newIndex) {
    newFiber.index = newIndex;
    if (shouldTrackSideEffects) newFiber.flags |= Placement;
  }
  function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
    let resultingFirstChild = null;
    let previousNewFiber = null;
    let newIdx = 0;
    for (; newIdx < newChildren.length; newIdx++) {
      const newFiber = createChild(returnFiber, newChildren[newIdx]);
      if (newFiber === null) {
        continue;
      }
      placeChild(newFiber, newIdx);
      if (previousNewFiber === null) {
        resultingFirstChild = newFiber;
      } else {
        previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
    }
    return resultingFirstChild;
  }
  function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
    if (typeof newChild === "object" && newChild !== null) {
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          return placeSingleChild(reconcileSingleElement(returnFiber, currentFirstChild, newChild));
        }
        default:
          break;
      }
      if (isArray(newChild)) {
        return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
      }
    }
    if (typeof newChild === "string") {
      return placeSingleChild(reconcileSingleTextNode(returnFiber, currentFirstChild, newChild));
    }
    return null;
  }
  return reconcileChildFibers;
}
export const reconcileChildFibers = createChildReconciler(true);
export const mountChildFibers = createChildReconciler(false);
```

10.6 ReactDOMHostConfig.js

src\react-dom-bindings\src\client\ReactDOMHostConfig.js

```
export function shouldSetTextContent(type, props) {
  return typeof props.children === "string" || typeof props.children === "number";
}
```

10.7 logger.js

src\shared\logger.js

```
import * as ReactWorkTags from "react-reconciler/src/ReactWorkTags";
const ReactWorkTagsMap = new Map();
for (let tag in ReactWorkTags) {
  ReactWorkTagsMap.set(ReactWorkTags[tag], tag);
}

export default function logger(prefix, workInProgress) {
  let tagValue = workInProgress.tag;
  let tagName = ReactWorkTagsMap.get(tagValue);
  let str = ` ${tagName} `;
  if (tagName === "HostComponent") {
    str += ` ${workInProgress.type} `;
  } else if (tagName === "HostText") {
    str += ` ${workInProgress.pendingProps} `;
  }
  console.log(`${prefix} ${str}`);
}

let indent = { number: 0 };
export { indent };
```

10.8 assign.js

src\shared\assign.js

```
const { assign } = Object;
export default assign;
```

10.9 isArray.js

src\shared\isArray.js

```
const { isArray } = Array;
export default isArray;
```

10.10 ReactFiber.js

src\react-reconciler\src\ReactFiber.js

```
import {
  HostRoot,
+ IndeterminateComponent,
+ HostComponent,
+ HostText,
} from "./ReactWorkTags";
import { NoFlags } from "./ReactFiberFlags";
export function FiberNode(tag, pendingProps, key) {
  this.tag = tag;
  this.key = key;
  this.type = null;
  this.stateNode = null;

  this.return = null;
  this.child = null;
  this.sibling = null;

  this.pendingProps = pendingProps;
  this.memoizedProps = null;
  this.updateQueue = null;
  this.memoizedState = null;

  this.flags = NoFlags;
  this.subtreeFlags = NoFlags;
  this.alternate = null;
}
function createFiber(tag, pendingProps, key) {
  return new FiberNode(tag, pendingProps, key);
}
export function createHostRootFiber() {
  return createFiber(HostRoot, null, null);
}
// We use a double buffering pooling technique because we know that we'll
// only ever need at most two versions of a tree. We pool the "other" unused
// node that we're free to reuse. This is lazily created to avoid allocating
// extra objects for things that are never updated. It also allow us to
// reclaim the extra memory if needed.

//我们使用双缓冲池技术，因为我们知道一棵树最多只需要两个版本
//我们将"其他"未使用的我们可以自由重用的节点
//这是延迟创建的，以避免分配从未更新的内容的额外对象。它还允许我们如果需要，回收额外的内存
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.type = current.type;
    workInProgress.flags = NoFlags;
    workInProgress.subtreeFlags = NoFlags;
  }
  workInProgress.child = current.child;
  workInProgress.memoizedProps = current.memoizedProps;
  workInProgress.memoizedState = current.memoizedState;
  workInProgress.updateQueue = current.updateQueue;
  workInProgress.sibling = current.sibling;
  workInProgress.index = current.index;
  return workInProgress;
}
+export function createFiberFromTypeAndProps(type, key, pendingProps) {
+  let fiberTag = IndeterminateComponent;
+  if (typeof type === "string") {
+    fiberTag = HostComponent;
+  }
+  const fiber = createFiber(fiberTag, pendingProps, key);
+  fiber.type = type;
+  return fiber;
+}
+export function createFiberFromElement(element) {
+  const { type } = element;
+  const { key } = element;
+  const pendingProps = element.props;
+  const fiber = createFiberFromTypeAndProps(type, key, pendingProps);
+  return fiber;
+}
+
+export function createFiberFromText(content) {
+  const fiber = createFiber(HostText, content, null);
+  return fiber;
+}
```

10.11 ReactFiberFlags.js

src\react-reconciler\src\ReactFiberFlags.js

```
export const NoFlags = 0b00000000000000000000000000;
+export const Placement = 0b00000000000000000000000010;
+export const MutationMask = Placement;
```

11.completeUnitOfWork

```
beginWork ▶ FiberNode {tag: 3, key: null, type: null, stateNode: FiberRootNode, return: null, …}
beginWork ▶ FiberNode {tag: 5, key: null, type: 'h1', stateNode: null, return: FiberNode, …}
beginWork ▶ FiberNode {tag: 6, key: null, type: null, stateNode: null, return: FiberNode, …}
completeWork ▶ FiberNode {tag: 6, key: null, type: null, stateNode: null, return: FiberNode, …}
beginWork ▶ FiberNode {tag: 5, key: null, type: 'span', stateNode: null, return: FiberNode, …}
completeWork ▶ FiberNode {tag: 5, key: null, type: 'span', stateNode: null, return: FiberNode, …}
completeWork ▶ FiberNode {tag: 5, key: null, type: 'h1', stateNode: null, return: FiberNode, …}
completeWork ▶ FiberNode {tag: 3, key: null, type: null, stateNode: FiberRootNode, return: null, …}
```

11.1 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import { scheduleCallback } from "scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
+import { completeWork } from "./ReactFiberCompleteWork";
let workInProgress = null;
export function scheduleUpdateOnFiber(root) {
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  scheduleCallback(performConcurrentWorkOnRoot.bind(null, root));
}
function performConcurrentWorkOnRoot(root) {
  renderRootSync(root);
}
function prepareFreshStack(root) {
  workInProgress = createWorkInProgress(root.current, null);
}
function renderRootSync(root) {
  prepareFreshStack(root);
  workLoopSync();
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
+   completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

+function completeUnitOfWork(unitOfWork) {
+  let completedWork = unitOfWork;
+  do {
+    const current = completedWork.alternate;
+    const returnFiber = completedWork.return;
+    completeWork(current, completedWork);
+    const siblingFiber = completedWork.sibling;
+    if (siblingFiber !== null) {
+      workInProgress = siblingFiber;
+      return;
+    }
+    completedWork = returnFiber;
+    workInProgress = completedWork;
+  } while (completedWork !== null);
+}
```

11.2 ReactFiberCompleteWork.js

src\react-reconciler\src\ReactFiberCompleteWork.js

```
import {
  appendInitialChild,
  createInstance,
  createTextInstance,
  finalizeInitialChildren,
} from "react-dom-bindings/src/client/ReactDOMHostConfig";
import { HostComponent, HostRoot, HostText } from "./ReactWorkTags";
import { NoFlags } from "./ReactFiberFlags";
import logger, { indent } from "shared/logger";
function bubbleProperties(completedWork) {
  let subtreeFlags = NoFlags;
  let child = completedWork.child;
  while (child !== null) {
    subtreeFlags |= child.subtreeFlags;
    subtreeFlags |= child.flags;
    child = child.sibling;
  }
  completedWork.subtreeFlags |= subtreeFlags;
}

function appendAllChildren(parent, workInProgress) {

  let node = workInProgress.child;
  while (node !== null) {

    if (node.tag === HostComponent || node.tag === HostText) {
      appendInitialChild(parent, node.stateNode);

    } else if (node.child !== null) {

      node = node.child;
      continue;
    }
    if (node === workInProgress) {
      return;
    }

    while (node.sibling === null) {

      if (node.return === null || node.return === workInProgress) {
        return;
      }
      node = node.return;
    }

    node = node.sibling;
  }
}

export function completeWork(current, workInProgress) {
  indent.number -= 2;
  logger(" ".repeat(indent.number) + "completeWork", workInProgress);
  const newProps = workInProgress.pendingProps;
  switch (workInProgress.tag) {
    case HostComponent: {
      const { type } = workInProgress;
      const instance = createInstance(type, newProps, workInProgress);
      appendAllChildren(instance, workInProgress);
      workInProgress.stateNode = instance;
      finalizeInitialChildren(instance, type, newProps);
      bubbleProperties(workInProgress);
      break;
    }
    case HostRoot:
      bubbleProperties(workInProgress);
      break;
    case HostText: {
      const newText = newProps;
      workInProgress.stateNode = createTextInstance(newText);
      bubbleProperties(workInProgress);
      break;
    }
    default:
      break;
  }
}
```

11.3 ReactDOMHostConfig.js

src\react-dom-bindings\src\client\ReactDOMHostConfig.js

```
+import { setInitialProperties } from "./ReactDOMComponent";
export function shouldSetTextContent(type, props) {
  return (
    typeof props.children
  );
}
+export const appendInitialChild = (parent, child) => {
+  parent.appendChild(child);
+};
+export const createInstance = (type, props, internalInstanceHandle) => {
+  const domElement = document.createElement(type);
+  return domElement;
+};
+export const createTextInstance = (content) => document.createTextNode+(content);
+export function finalizeInitialChildren(domElement, type, props) {
+  setInitialProperties(domElement, type, props);
+}
```

11.4 ReactDOMComponent.js

src\react-dom-bindings\src\client\ReactDOMComponent.js

```
import { setValueForStyles } from "./CSSPropertyOperations";
import setTextContent from "./setTextContent";
import { setValueForProperty } from "./DOMPropertyOperations";
const CHILDREN = "children";
const STYLE = "style";
function setInitialDOMProperties(tag, domElement, nextProps) {
  for (const propKey in nextProps) {
    if (nextProps.hasOwnProperty(propKey)) {
      const nextProp = nextProps[propKey];
      if (propKey === STYLE) {
        setValueForStyles(domElement, nextProp);
      } else if (propKey === CHILDREN) {
        if (typeof nextProp === "string") {
          setTextContent(domElement, nextProp);
        } else if (typeof nextProp === "number") {
          setTextContent(domElement, `${nextProp}`);
        }
      } else if (nextProp != null) {
        setValueForProperty(domElement, propKey, nextProp);
      }
    }
  }
}
export function setInitialProperties(domElement, tag, props) {
  setInitialDOMProperties(tag, domElement, props);
}
```

11.5 CSSPropertyOperations.js

src\react-dom-bindings\src\client\CSSPropertyOperations.js

```
export function setValueForStyles(node, styles) {
  const { style } = node;
  for (const styleName in styles) {
    if (styles.hasOwnProperty(styleName)) {
      const styleValue = styles[styleName];
      style[styleName] = styleValue;
    }
  }
}
```

11.6 setTextContent.js

src\react-dom-bindings\src\client\setTextContent.js

```
function setTextContent(node, text) {
  node.textContent = text;
}

export default setTextContent;
```

11.7 DOMPropertyOperations.js

src\react-dom-bindings\src\client\DOMPropertyOperations.js

```
export function setValueForProperty(node, name, value) {
  if (value === null) {
    node.removeAttribute(name);
  } else {
    node.setAttribute(name, value);
  }
}
```

12.commitRoot

12.1 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import { scheduleCallback } from "scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
+import { MutationMask, NoFlags } from "./ReactFiberFlags";
let workInProgress = null;
export function scheduleUpdateOnFiber(root) {
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  scheduleCallback(performConcurrentWorkOnRoot.bind(null, root));
}
function performConcurrentWorkOnRoot(root) {
  renderRootSync(root);
+ const finishedWork = root.current.alternate;
+ printFiber(finishedWork);
+ console.log(`~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`);
+ root.finishedWork = finishedWork;
+ commitRoot(root);
}
+function commitRoot(root) {
+  const { finishedWork } = root;
+  const subtreeHasEffects =
+    (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
+  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
+  if (subtreeHasEffects || rootHasEffect) {
+    console.log("commitRoot");
+  }
+  root.current = finishedWork;
+}
+function printFiber(fiber) {
+  /*
+  fiber.flags &= ~Forked;
+  fiber.flags &= ~PlacementDEV;
+  fiber.flags &= ~Snapshot;
```

```
+  fiber.flags &= ~PerformedWork;
+  */
+  if (fiber.flags !== 0) {
+    console.log(
+      getFlags(fiber.flags),
+      getTag(fiber.tag),
+      typeof fiber.type === "function" ? fiber.type.name : fiber.type,
+      fiber.memoizedProps
+    );
+    if (fiber.deletions) {
+      for (let i = 0; i < fiber.deletions.length; i++) {
+        const childToDelete = fiber.deletions[i];
+        console.log(getTag(childToDelete.tag), childToDelete.type, childToDelete.+memoizedProps);
+      }
+    }
+  }
+  let child = fiber.child;
+  while (child) {
+    printFiber(child);
+    child = child.sibling;
+  }
+}
+function getTag(tag) {
+  switch (tag) {
+    case FunctionComponent:
+      return `FunctionComponent`;
+    case HostRoot:
+      return `HostRoot`;
+    case HostComponent:
+      return `HostComponent`;
+    case HostText:
+      return HostText;
+    default:
+      return tag;
+  }
+}
+function getFlags(flags) {
+  if (flags === (Update | Placement | ChildDeletion)) {
+    return `自己移动和子元素有删除`;
+  }
+  if (flags === (ChildDeletion | Update)) {
+    return `自己有更新和子元素有删除`;
+  }
+  if (flags === ChildDeletion) {
+    return `子元素有删除`;
+  }
+  if (flags === (Placement | Update)) {
+    return `移动并更新`;
+  }
+  if (flags === Placement) {
+    return `插入`;
+  }
+  if (flags === Update) {
+    return `更新`;
+  }
+  return flags;
+}

function prepareFreshStack(root) {
  workInProgress = createWorkInProgress(root.current, null);
}
function renderRootSync(root) {
  prepareFreshStack(root);
  workLoopSync();
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
}
```

13.commitMutationEffectsOnFiber

13.1 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import { scheduleCallback } from "scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
import { MutationMask, NoFlags } from "./ReactFiberFlags";
+import { commitMutationEffectsOnFiber } from "./ReactFiberCommitWork";
let workInProgress = null;
export function scheduleUpdateOnFiber(root) {
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  scheduleCallback(performConcurrentWorkOnRoot.bind(null, root));
}
function performConcurrentWorkOnRoot(root) {
  renderRootSync(root);
  const finishedWork = root.current.alternate;
  root.finishedWork = finishedWork;
  commitRoot(root);
}
function commitRoot(root) {
  const { finishedWork } = root;
  const subtreeHasEffects =
    (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
+   commitMutationEffectsOnFiber(finishedWork, root);
  }
  root.current = finishedWork;
}
function prepareFreshStack(root) {
  workInProgress = createWorkInProgress(root.current, null);
}
function renderRootSync(root) {
  prepareFreshStack(root);
  workLoopSync();
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
}
```

13.2 ReactFiberCommitWork.js

src\react-reconciler\src\ReactFiberCommitWork.js

```
import { HostRoot, HostComponent, HostText } from "./ReactWorkTags";
import { MutationMask, Placement } from "./ReactFiberFlags";
function recursivelyTraverseMutationEffects(root, parentFiber) {
  if (parentFiber.subtreeFlags & MutationMask) {
    let { child } = parentFiber;
    while (child !== null) {
      commitMutationEffectsOnFiber(child, root);
      child = child.sibling;
    }
  }
}
function commitPlacement(finishedWork) {
  console.log("commitPlacement", finishedWork);
}
function commitReconciliationEffects(finishedWork) {
  const { flags } = finishedWork;
  if (flags & Placement) {
    commitPlacement(finishedWork);
    finishedWork.flags &= ~Placement;
  }
}
export function commitMutationEffectsOnFiber(finishedWork, root) {
  switch (finishedWork.tag) {
    case HostRoot:
    case HostComponent:
    case HostText: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    default: {
      break;
    }
  }
}
```

14.commitPlacement

14.1 ReactFiberCommitWork.js

src\react-reconciler\src\ReactFiberCommitWork.js

```
import { HostRoot, HostComponent, HostText } from "./ReactWorkTags";
import { MutationMask, Placement } from "./ReactFiberFlags";
+import {
+  insertBefore,
+  appendChild,
+} from "react-dom-bindings/src/client/ReactDOMHostConfig";
function recursivelyTraverseMutationEffects(root, parentFiber) {
  if (parentFiber.subtreeFlags & MutationMask) {
    let { child } = parentFiber;
    while (child !== null) {
      commitMutationEffectsOnFiber(child, root);
      child = child.sibling;
    }
  }
}
+function isHostParent(fiber) {
+  return fiber.tag === HostComponent || fiber.tag === HostRoot;
+}
+function getHostParentFiber(fiber) {
+  let parent = fiber.return;
+  while (parent !== null) {
+    if (isHostParent(parent)) {
+      return parent;
+    }
+    parent = parent.return;
+  }
+  return parent;
+}
+function insertOrAppendPlacementNode(node, before, parent) {
+  const { tag } = node;
+  const isHost = tag === HostComponent || tag === HostText;
+  if (isHost) {
+    const { stateNode } = node;
+    if (before) {
+      insertBefore(parent, stateNode, before);
+    } else {
+      appendChild(parent, stateNode);
+    }
+  } else {
+    const { child } = node;
+    if (child !== null) {
+      insertOrAppendPlacementNode(child, before, parent);
+      let { sibling } = child;
+      while (sibling !== null) {
+        insertOrAppendPlacementNode(sibling, before, parent);
+        sibling = sibling.sibling;
+      }
+    }
+  }
+}
+function getHostSibling(fiber) {
+  let node = fiber;
+  siblings: while (true) {
+    // 如果我们没有找到任何东西，让我们试试下一个弟弟
+    while (node.sibling === null) {
+      if (node.return === null || isHostParent(node.return)) {
+        // 如果我们是根Fiber或者父亲是原生节点，我们就是最后的弟弟
+        return null;
+      }
+      node = node.return;
+    }
```

```
+    // node.sibling.return = node.return
+    node = node.sibling;
+    while (node.tag !== HostComponent && node.tag !== HostText) {
+      // 如果它不是原生节点，并且，我们可能在其中有一个原生节点
+      // 试着向下搜索，直到找到为止
+      if (node.flags & Placement) {
+        // 如果我们没有孩子，可以试试弟弟
+        continue siblings;
+      } else {
+        // node.child.return = node
+        node = node.child;
+      }
+    } // Check if this host node is stable or about to be placed.

+    // 检查此原生节点是否稳定可以放置
+    if (!(node.flags & Placement)) {
+      // 找到它了!

+      return node.stateNode;
+    }
+  }
+}
function commitPlacement(finishedWork) {
+  const parentFiber = getHostParentFiber(finishedWork);
+  switch (parentFiber.tag) {
+    case HostComponent: {
+      const parent = parentFiber.stateNode;
+      const before = getHostSibling(finishedWork);
+      insertOrAppendPlacementNode(finishedWork, before, parent);
+      break;
+    }
+    case HostRoot: {
+      const parent = parentFiber.stateNode.containerInfo;
+      const before = getHostSibling(finishedWork);
+      insertOrAppendPlacementNode(finishedWork, before, parent);
+      break;
+    }
+    default:
+      break;
+  }
}
function commitReconciliationEffects(finishedWork) {
  const { flags } = finishedWork;
  if (flags & Placement) {
    commitPlacement(finishedWork);
    finishedWork.flags &= ~Placement;
  }
}
export function commitMutationEffectsOnFiber(finishedWork, root) {
  switch (finishedWork.tag) {
    case HostRoot:
    case HostComponent:
    case HostText: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    default: {
      break;
    }
  }
}
```

14.2 ReactDOMHostConfig.js

src\react-dom-bindings\src\client\ReactDOMHostConfig.js

```
import { setInitialProperties } from "./ReactDOMComponent";
export function shouldSetTextContent(type, props) {
  return (
    typeof props.children
  );
}
export const appendInitialChild = (parent, child) => {
  parent.appendChild(child);
};
export const createInstance = (type, props, internalInstanceHandle) => {
  const domElement = document.createElement(type);
  return domElement;
};
export const createTextInstance = (content) => document.createTextNode(content);
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);
+}
```

15.函数组件

15.1 src\main.jsx

src\main.jsx

```
import { createRoot } from "react-dom/client";
+function FunctionComponent() {
+  return (
+
+      helloworld
+
+  );
+}
+let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

src\react-reconciler\src\ReactWorkTags.js

```
+export const FunctionComponent = 0;
+export const IndeterminateComponent = 2;
export const HostRoot = 3;
export const HostComponent = 5;
export const HostText = 6;
```

15.3 ReactFiberBeginWork.js

src\react-reconciler\src\ReactFiberBeginWork.js

```
import {
  HostRoot,
  HostComponent,
  HostText,
+ IndeterminateComponent,
+ FunctionComponent,
} from "./ReactWorkTags";
import { processUpdateQueue } from "./ReactFiberClassUpdateQueue";
import { mountChildFibers, reconcileChildFibers } from "./ReactChildFiber";
import { shouldSetTextContent } from "react-dom-bindings/src/client/ReactDOMHostConfig";
import logger, { indent } from "shared/logger";
+import { renderWithHooks } from "react-reconciler/src/ReactFiberHooks";
function reconcileChildren(current, workInProgress, nextChildren) {
  if (current
    workInProgress.child = mountChildFibers(workInProgress, null, nextChildren);
  } else {
    workInProgress.child = reconcileChildFibers(
      workInProgress,
      current.child,
      nextChildren
    );
  }
}
function updateHostRoot(current, workInProgress) {
  processUpdateQueue(workInProgress);
  const nextState = workInProgress.memoizedState;
  const nextChildren = nextState.element;
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
function updateHostComponent(current, workInProgress) {
  const { type } = workInProgress;
  const nextProps = workInProgress.pendingProps;
  let nextChildren = nextProps.children;
  const isDirectTextChild = shouldSetTextContent(type, nextProps);
  if (isDirectTextChild) {
    nextChildren = null;
  }
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
+function mountIndeterminateComponent(_current, workInProgress, Component) {
+  const props = workInProgress.pendingProps;
+  const value = renderWithHooks(null, workInProgress, Component, props);
+  workInProgress.tag = FunctionComponent;
+  reconcileChildren(null, workInProgress, value);
+  return workInProgress.child;
+}
export function beginWork(current, workInProgress) {
  logger(" ".repeat(indent.number) + "beginWork", workInProgress);
  indent.number += 2;
  switch (workInProgress.tag) {
+   case IndeterminateComponent: {
+     return mountIndeterminateComponent(
+       current,
+       workInProgress,
+       workInProgress.type
+     );
+   }
    case HostRoot:
      return updateHostRoot(current, workInProgress);
    case HostComponent:
      return updateHostComponent(current, workInProgress);
    case HostText:
    default:
      return null;
  }
}
```

15.4 ReactFiberHooks.js

src\react-reconciler\src\ReactFiberHooks.js

```
export function renderWithHooks(current, workInProgress, Component, props) {
  const children = Component(props);
  return children;
}
```

15.5 ReactFiberCommitWork.js

src\react-reconciler\src\ReactFiberCommitWork.js

```
import {
  HostRoot,
  HostComponent,
  HostText,
+ FunctionComponent,
} from "./ReactWorkTags";
import { MutationMask, Placement } from "./ReactFiberFlags";
import {
  insertBefore,
  appendChild,
} from "react-dom-bindings/src/client/ReactDOMHostConfig";
function recursivelyTraverseMutationEffects(root, parentFiber) {
  if (parentFiber.subtreeFlags & MutationMask) {
    let { child } = parentFiber;
    while (child !== null) {
      commitMutationEffectsOnFiber(child, root);
      child = child.sibling;
    }
  }
}
function isHostParent(fiber) {
  return fiber.tag
}
function getHostParentFiber(fiber) {
  let parent = fiber.return;
  while (parent !== null) {
    if (isHostParent(parent)) {
      return parent;
    }
    parent = parent.return;
  }
  return parent;
}
function insertOrAppendPlacementNode(node, before, parent) {
  const { tag } = node;
  const isHost = tag
  if (isHost) {
    const { stateNode } = node;
    if (before) {
      insertBefore(parent, stateNode, before);
    } else {
      appendChild(parent, stateNode);
    }
  } else {
    const { child } = node;
    if (child !== null) {
      insertOrAppendPlacementNode(child, before, parent);
      let { sibling } = child;
      while (sibling !== null) {
        insertOrAppendPlacementNode(sibling, before, parent);
        sibling = sibling.sibling;
      }
    }
  }
}
function getHostSibling(fiber) {
  let node = fiber;
  siblings: while (true) {
    // 如果我们没有找到任何东西，让我们试试下一个弟弟
    while (node.sibling
      if (node.return
        // 如果我们是根Fiber或者父亲是原生节点，我们就是最后的弟弟
        return null;
      }
      node = node.return;
    }
    // node.sibling.return = node.return
    node = node.sibling;
    while (node.tag !== HostComponent && node.tag !== HostText) {
      // 如果它不是原生节点，并且，我们可能在其中有一个原生节点
      // 试着向下搜索，直到找到为止
      if (node.flags & Placement) {
        // 如果我们没有孩子，可以试试弟弟
        continue siblings;
      } else {
        // node.child.return = node
        node = node.child;
      }
    } // Check if this host node is stable or about to be placed.

    // 检查此原生节点是否稳定可以放置
    if (!(node.flags & Placement)) {
      // 找到它了!

      return node.stateNode;
    }
  }
}
function commitPlacement(finishedWork) {
  const parentFiber = getHostParentFiber(finishedWork);
  switch (parentFiber.tag) {
    case HostComponent: {
      const parent = parentFiber.stateNode;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    case HostRoot: {
      const parent = parentFiber.stateNode.containerInfo;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
```

```
    default:
      break;
  }
}
function commitReconciliationEffects(finishedWork) {
  const { flags } = finishedWork;
  if (flags & Placement) {
    commitPlacement(finishedWork);
    finishedWork.flags &= ~Placement;
  }
}
export function commitMutationEffectsOnFiber(finishedWork, root) {
  switch (finishedWork.tag) {
    case HostRoot:
+   case FunctionComponent:
    case HostComponent:
    case HostText: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    default: {
      break;
    }
  }
}
```

16.注册事件名

16.1 src\main.jsx

src\main.jsx

```
import { createRoot } from "react-dom/client";
function FunctionComponent() {
  return (
+
+     onClick={() => console.log("onClick FunctionComponent")}
+     onClickCapture={() => console.log("onClickCapture FunctionComponent")}
+   >
+     hello
+
+       style={{ color: "red" }}
+       onClick={() => console.log("onClick span")}
+       onClickCapture={() => console.log("onClickCapture span")}
+     >
+       world
+
+
  );
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

16.2 ReactDOMRoot.js

src\react-dom\src\client\ReactDOMRoot.js

```
import {
  createContainer,
  updateContainer,
} from "react-reconciler/src/ReactFiberReconciler";
+import { listenToAllSupportedEvents } from "react-dom-bindings/src/events/DOMPluginEventSystem";

function ReactDOMRoot(internalRoot) {
  this._internalRoot = internalRoot;
}
ReactDOMRoot.prototype.render = function render(children) {
  const root = this._internalRoot;
  root.containerInfo.innerHTML = "";
  updateContainer(children, root);
};
export function createRoot(container) {
  const root = createContainer(container);
+ listenToAllSupportedEvents(container);
  return new ReactDOMRoot(root);
}
```

16.3 DOMPluginEventSystem.js

src\react-dom-bindings\src\events\DOMPluginEventSystem.js

```
import { allNativeEvents } from "./EventRegistry";
import * as SimpleEventPlugin from "./plugins/SimpleEventPlugin";
SimpleEventPlugin.registerEvents();
export function listenToAllSupportedEvents(rootContainerElement) {
  allNativeEvents.forEach((domEventName) => {
    console.log(domEventName);
  });
}
```

16.4 EventRegistry.js

src\react-dom-bindings\src\events\EventRegistry.js

```
export const allNativeEvents = new Set();
export function registerTwoPhaseEvent(registrationName, dependencies) {
  registerDirectEvent(registrationName, dependencies);
  registerDirectEvent(registrationName + "Capture", dependencies);
}
export function registerDirectEvent(registrationName, dependencies) {
  for (let i = 0; i < dependencies.length; i++) {
    allNativeEvents.add(dependencies[i]);
  }
}
```

16.5 SimpleEventPlugin.js

src\react-dom-bindings\src\events\plugins\SimpleEventPlugin.js

```
import { registerSimpleEvents } from "../DOMEventProperties";
export { registerSimpleEvents as registerEvents };
```

16.6 DOMEventProperties.js

src\react-dom-bindings\src\events\DOMEventProperties.js

```
import { registerTwoPhaseEvent } from "./EventRegistry";

const simpleEventPluginEvents = ["click"];
function registerSimpleEvent(domEventName, reactName) {
  registerTwoPhaseEvent(reactName, [domEventName]);
}

export function registerSimpleEvents() {
  for (let i = 0; i < simpleEventPluginEvents.length; i++) {
    const eventName = simpleEventPluginEvents[i];
    const domEventName = eventName.toLowerCase();
    const capitalizedEvent = eventName[0].toUpperCase() + eventName.slice(1);
    registerSimpleEvent(domEventName, `on${capitalizedEvent}`);
  }
}
```

17.listenToNativeEvent

17.1 DOMPluginEventSystem.js

src\react-dom-bindings\src\events\DOMPluginEventSystem.js

```
import { allNativeEvents } from "./EventRegistry";
import * as SimpleEventPlugin from "./plugins/SimpleEventPlugin";
+import { createEventListenerWrapperWithPriority } from "./ReactDOMEventListener";
+import { IS_CAPTURE_PHASE } from "./EventSystemFlags";
+import { addEventCaptureListener, addEventBubbleListener } from "./EventListener";

SimpleEventPlugin.registerEvents();

export function listenToAllSupportedEvents(rootContainerElement) {
  allNativeEvents.forEach((domEventName) => {
+   listenToNativeEvent(domEventName, true, rootContainerElement);
+   listenToNativeEvent(domEventName, false, rootContainerElement);
  });
}

+export function listenToNativeEvent(domEventName, isCapturePhaseListener, +target) {
+  let eventSystemFlags = 0; // 冒泡 = 0 捕获 = 4
+  if (isCapturePhaseListener) {
+    eventSystemFlags |= IS_CAPTURE_PHASE;
+  }
+  addTrappedEventListener(target, domEventName, eventSystemFlags, isCapturePhaseListener);
+}
+function addTrappedEventListener(targetContainer, domEventName, +eventSystemFlags, isCapturePhaseListener) {
+  const listener = createEventListenerWrapperWithPriority(targetContainer, domEventName, eventSystemFlags);
+  if (isCapturePhaseListener) {
+    addEventCaptureListener(targetContainer, domEventName, listener);
+  } else {
+    addEventBubbleListener(targetContainer, domEventName, listener);
+  }
+}
```

17.2 EventSystemFlags.js

src\react-dom-bindings\src\events\EventSystemFlags.js

```
export const IS_CAPTURE_PHASE = 1 << 2;
```

17.3 ReactDOMEventListener.js

src\react-dom-bindings\src\events\ReactDOMEventListener.js

```
export function createEventListenerWrapperWithPriority(targetContainer, domEventName, eventSystemFlags) {
  const listenerWrapper = dispatchDiscreteEvent;
  return listenerWrapper.bind(null, domEventName, eventSystemFlags, targetContainer);
}
function dispatchDiscreteEvent(domEventName, eventSystemFlags, container, nativeEvent) {
  dispatchEvent(domEventName, eventSystemFlags, container, nativeEvent);
}
export function dispatchEvent(domEventName, eventSystemFlags, targetContainer, nativeEvent) {
  console.log("dispatchEvent", domEventName, eventSystemFlags, targetContainer, nativeEvent);
}
```

17.4 EventListener.js

src\react-dom-bindings\src\events\EventListener.js

```
export function addEventCaptureListener(target, eventType, listener) {
  target.addEventListener(eventType, listener, true);
  return listener;
}

export function addEventBubbleListener(target, eventType, listener) {
  target.addEventListener(eventType, listener, false);
  return listener;
}
```

18.extractEvents

18.1 ReactDOMEventListener.js

src\react-dom-bindings\src\events\ReactDOMEventListener.js

```
+import getEventTarget from "./getEventTarget";
+import { getClosestInstanceFromNode } from "../client/ReactDOMComponentTree";
+import { dispatchEventForPluginEventSystem } from "./DOMPluginEventSystem";

+export function createEventListenerWrapperWithPriority(
+  targetContainer,
+  domEventName,
+  eventSystemFlags
+) {
+  const listenerWrapper = dispatchDiscreteEvent;
+  return listenerWrapper.bind(null, domEventName, eventSystemFlags, targetContainer);
+}

+function dispatchDiscreteEvent(domEventName, eventSystemFlags, container, nativeEvent) {
+  dispatchEvent(domEventName, eventSystemFlags, container, nativeEvent);
+}
export function dispatchEvent(domEventName, eventSystemFlags, targetContainer, nativeEvent) {
+ const nativeEventTarget = getEventTarget(nativeEvent);
+ const targetInst = getClosestInstanceFromNode(nativeEventTarget);
+ dispatchEventForPluginEventSystem(
+   domEventName,
+   eventSystemFlags,
+   nativeEvent,
+   targetInst,
+   targetContainer
+ );
}
```

18.2 getEventTarget.js

src\react-dom-bindings\src\events\getEventTarget.js

```
function getEventTarget(nativeEvent) {
  const target = nativeEvent.target || nativeEvent.srcElement || window;
  return target;
}
export default getEventTarget;
```

18.3 ReactDOMComponentTree.js

src\react-dom-bindings\src\client\ReactDOMComponentTree.js

```
const randomKey = Math.random().toString(36).slice(2);
const internalInstanceKey = "__reactFiber{{content}}quot; + randomKey;
const internalPropsKey = "__reactProps{{content}}quot; + randomKey;

export function getClosestInstanceFromNode(targetNode) {
  const targetInst = targetNode[internalInstanceKey];
  if (targetInst) {
    return targetInst;
  }
  return null;
}

export function getFiberCurrentPropsFromNode(node) {
  return node[internalPropsKey] || null;
}

export function precacheFiberNode(hostInst, node) {
  node[internalInstanceKey] = hostInst;
}

export function updateFiberProps(node, props) {
  node[internalPropsKey] = props;
}
```

18.4 DOMPluginEventSystem.js

src\react-dom-bindings\src\events\DOMPluginEventSystem.js

```
import { allNativeEvents } from "./EventRegistry";
import * as SimpleEventPlugin from "./plugins/SimpleEventPlugin";
import { createEventListenerWrapperWithPriority } from "./ReactDOMEventListener";
import { IS_CAPTURE_PHASE } from "./EventSystemFlags";
import { addEventCaptureListener, addEventBubbleListener } from "./EventListener";
+import getEventTarget from "./getEventTarget";
+import getListener from "./getListener";
+import { HostComponent } from "react-reconciler/src/ReactWorkTags";

SimpleEventPlugin.registerEvents();

export function listenToAllSupportedEvents(rootContainerElement) {
  allNativeEvents.forEach((domEventName) => {
    listenToNativeEvent(domEventName, true, rootContainerElement);
    listenToNativeEvent(domEventName, false, rootContainerElement);
  });
}
```

```
export function listenToNativeEvent(domEventName, isCapturePhaseListener, target) {
  let eventSystemFlags = 0; // 冒泡 = 0 捕获 = 4
  if (isCapturePhaseListener) {
    eventSystemFlags |= IS_CAPTURE_PHASE;
  }
  addTrappedEventListener(target, domEventName, eventSystemFlags, isCapturePhaseListener);
}
function addTrappedEventListener(targetContainer, domEventName, eventSystemFlags, isCapturePhaseListener) {
  const listener = createEventListenerWrapperWithPriority(targetContainer, domEventName, eventSystemFlags);
  if (isCapturePhaseListener) {
    addEventCaptureListener(targetContainer, domEventName, listener);
  } else {
    addEventBubbleListener(targetContainer, domEventName, listener);
  }
}
+export function dispatchEventForPluginEventSystem(
+  domEventName,
+  eventSystemFlags,
+  nativeEvent,
+  targetInst,
+  targetContainer
+) {
+  dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent, targetInst, targetContainer);
+}

+function dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent, targetInst, targetContainer) {
+  const nativeEventTarget = getEventTarget(nativeEvent);
+  const dispatchQueue = [];
+  extractEvents(
+    dispatchQueue,
+    domEventName,
+    targetInst,
+    nativeEvent,
+    nativeEventTarget,
+    eventSystemFlags,
+    targetContainer
+  );
+  console.log("dispatchQueue", dispatchQueue);
+}

+function extractEvents(
+  dispatchQueue,
+  domEventName,
+  targetInst,
+  nativeEvent,
+  nativeEventTarget,
+  eventSystemFlags,
+  targetContainer
+) {
+  SimpleEventPlugin.extractEvents(
+    dispatchQueue,
+    domEventName,
+    targetInst,
+    nativeEvent,
+    nativeEventTarget,
+    eventSystemFlags,
+    targetContainer
+  );
+}

+export function accumulateSinglePhaseListeners(targetFiber, reactName, nativeEventType, inCapturePhase) {
+  const captureName = reactName + "Capture";
+  const reactEventName = inCapturePhase ? captureName : reactName;
+  const listeners = [];
+  let instance = targetFiber;
+  while (instance !== null) {
+    const { stateNode, tag } = instance;
+    if (tag === HostComponent && stateNode !== null) {
+      if (reactEventName !== null) {
+        const listener = getListener(instance, reactEventName);
+        if (listener !== null && listener !== undefined) {
+          listeners.push(createDispatchListener(instance, listener, stateNode));
+        }
+      }
+    }
+    instance = instance.return;
+  }
+  return listeners;
+}
+function createDispatchListener(instance, listener, currentTarget) {
+  return {
+    instance,
+    listener,
+    currentTarget,
+  };
+}
```

18.5 getListener.js

src\react-dom-bindings\src\events\getListener.js

```
import { getFiberCurrentPropsFromNode } from "../client/ReactDOMComponentTree";

export default function getListener(inst, registrationName) {
  const stateNode = inst.stateNode;
  if (stateNode === null) {
    return null;
  }
  const props = getFiberCurrentPropsFromNode(stateNode);
  if (props === null) {
    return null;
  }
  const listener = props[registrationName];
  return listener;
}
```

18.6 SimpleEventPlugin.js

src\react-dom-bindings\src\events\plugins\SimpleEventPlugin.js

```
import { registerSimpleEvents, topLevelEventsToReactNames } from "../DOMEventProperties";
import { SyntheticMouseEvent } from "../SyntheticEvent";
import { IS_CAPTURE_PHASE } from "../EventSystemFlags";
import { accumulateSinglePhaseListeners } from "../DOMPluginEventSystem";

+function extractEvents(dispatchQueue, domEventName, targetInst, nativeEvent, nativeEventTarget, eventSystemFlags) {
+  const reactName = topLevelEventsToReactNames.get(domEventName);
+  let SyntheticEventCtor;
+  switch (domEventName) {
+    case "click":
+      SyntheticEventCtor = SyntheticMouseEvent;
+      break;
+    default:
+      break;
+  }
+  const inCapturePhase = (eventSystemFlags & IS_CAPTURE_PHASE) !== 0;
+  const listeners = accumulateSinglePhaseListeners(targetInst, reactName, nativeEvent.type, inCapturePhase);
+  if (listeners.length > 0) {
+    const event = new SyntheticEventCtor(reactName, domEventName, targetInst, nativeEvent, nativeEventTarget);
+    dispatchQueue.push({
+      event,
+      listeners,
+    });
+  }
+}
+export { registerSimpleEvents as registerEvents, extractEvents };
```

18.7 SyntheticEvent.js

src\react-dom-bindings\src\events\SyntheticEvent.js

```
import assign from "shared/assign";

function functionThatReturnsTrue() {
  return true;
}
function functionThatReturnsFalse() {
  return false;
}
const MouseEventInterface = {
  clientX: 0,
  clientY: 0,
};
function createSyntheticEvent(Interface) {
  function SyntheticBaseEvent(reactName, reactEventType, targetInst, nativeEvent, nativeEventTarget) {
    this._reactName = reactName;
    this.type = reactEventType;
    this._targetInst = targetInst;
    this.nativeEvent = nativeEvent;
    this.target = nativeEventTarget;
    for (const propName in Interface) {
      if (!Interface.hasOwnProperty(propName)) {
        continue;
      }
      this[propName] = nativeEvent[propName];
    }
    this.isDefaultPrevented = functionThatReturnsFalse;
    this.isPropagationStopped = functionThatReturnsFalse;
    return this;
  }

  assign(SyntheticBaseEvent.prototype, {
    preventDefault() {
      const event = this.nativeEvent;
      if (event.preventDefault) {
        event.preventDefault();
      } else {
        event.returnValue = false;
      }
      this.isDefaultPrevented = functionThatReturnsTrue;
    },
    stopPropagation() {
      const event = this.nativeEvent;
      if (event.stopPropagation) {
        event.stopPropagation();
      } else {
        event.cancelBubble = true;
      }
      this.isPropagationStopped = functionThatReturnsTrue;
    },
  });
  return SyntheticBaseEvent;
}
export const SyntheticMouseEvent = createSyntheticEvent(MouseEventInterface);
```

18.8 ReactDOMHostConfig.js

src\react-dom-bindings\src\client\ReactDOMHostConfig.js

```
import { setInitialProperties } from "./ReactDOMComponent";
+import { precacheFiberNode, updateFiberProps } from "./ReactDOMComponentTree";

export function shouldSetTextContent(type, props) {
  return typeof props.children
}
export const appendInitialChild = (parent, child) => {
  parent.appendChild(child);
};
export const createInstance = (type, props, internalInstanceHandle) => {
  const domElement = document.createElement(type);
+ precacheFiberNode(internalInstanceHandle, domElement);
+ updateFiberProps(domElement, props);
  return domElement;
};
export const createTextInstance = (content) => document.createTextNode(content);
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);
}
```

18.9 DOMEventProperties.js

src\react-dom-bindings\src\events\DOMEventProperties.js

```
import { registerTwoPhaseEvent } from "./EventRegistry";
+export const topLevelEventsToReactNames = new Map();
const simpleEventPluginEvents = ["click"];
function registerSimpleEvent(domEventName, reactName) {
+ topLevelEventsToReactNames.set(domEventName, reactName);
  registerTwoPhaseEvent(reactName, [domEventName]);
}

export function registerSimpleEvents() {
  for (let i = 0; i < simpleEventPluginEvents.length; i++) {
    const eventName = simpleEventPluginEvents[i]; // click
    const domEventName = eventName.toLowerCase(); // click
    const capitalizedEvent = eventName[0].toUpperCase() + eventName.slice(1); // Click
    registerSimpleEvent(domEventName, `on${capitalizedEvent}`); // click=>onClick
  }
}
```

19.processDispatchQueue

19.1 DOMPluginEventSystem.js

src\react-dom-bindings\src\events\DOMPluginEventSystem.js

```
import { allNativeEvents } from "./EventRegistry";
import * as SimpleEventPlugin from "./plugins/SimpleEventPlugin";
import { createEventListenerWrapperWithPriority } from "./ReactDOMEventListener";
import { IS_CAPTURE_PHASE } from "./EventSystemFlags";
import { addEventCaptureListener, addEventBubbleListener } from "./EventListener";
import getEventTarget from "./getEventTarget";
import getListener from "./getListener";
import { HostComponent } from "react-reconciler/src/ReactWorkTags";

SimpleEventPlugin.registerEvents();

export function listenToAllSupportedEvents(rootContainerElement) {
  allNativeEvents.forEach((domEventName) => {
    listenToNativeEvent(domEventName, true, rootContainerElement);
    listenToNativeEvent(domEventName, false, rootContainerElement);
  });
}

export function listenToNativeEvent(domEventName, isCapturePhaseListener, target) {
  let eventSystemFlags = 0; // 冒泡 = 0 捕获 = 4
  if (isCapturePhaseListener) {
    eventSystemFlags |= IS_CAPTURE_PHASE;
  }
  addTrappedEventListener(target, domEventName, eventSystemFlags, isCapturePhaseListener);
}
function addTrappedEventListener(targetContainer, domEventName, eventSystemFlags, isCapturePhaseListener) {
  const listener = createEventListenerWrapperWithPriority(targetContainer, domEventName, eventSystemFlags);
  if (isCapturePhaseListener) {
    addEventCaptureListener(targetContainer, domEventName, listener);
  } else {
    addEventBubbleListener(targetContainer, domEventName, listener);
  }
}
export function dispatchEventForPluginEventSystem(
  domEventName,
  eventSystemFlags,
  nativeEvent,
  targetInst,
  targetContainer
) {
  dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent, targetInst, targetContainer);
}

function dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent, targetInst, targetContainer) {
  const nativeEventTarget = getEventTarget(nativeEvent);
  const dispatchQueue = [];
```

```
  extractEvents(
    dispatchQueue,
    domEventName,
    targetInst,
    nativeEvent,
    nativeEventTarget,
    eventSystemFlags,
    targetContainer
  );
+ processDispatchQueue(dispatchQueue, eventSystemFlags);
}

+export function processDispatchQueue(dispatchQueue, eventSystemFlags) {
+  const inCapturePhase = (eventSystemFlags & IS_CAPTURE_PHASE) !== 0;
+  for (let i = 0; i < dispatchQueue.length; i++) {
+    const { event, listeners } = dispatchQueue[i];
+    processDispatchQueueItemsInOrder(event, listeners, inCapturePhase); //  event system doesn't use pooling.

+  }
+}
+function processDispatchQueueItemsInOrder(event, dispatchListeners, inCapturePhase) {
+  if (inCapturePhase) {
+    for (let i = dispatchListeners.length - 1; i >= 0; i--) {
+      const { currentTarget, listener } = dispatchListeners[i];
+      if (event.isPropagationStopped()) {
+        return;
+      }
+      executeDispatch(event, listener, currentTarget);
+    }
+  } else {
+    for (let i = 0; i < dispatchListeners.length; i++) {
+      const { currentTarget, listener } = dispatchListeners[i];
+      if (event.isPropagationStopped()) {
+        return;
+      }
+      executeDispatch(event, listener, currentTarget);
+    }
+  }
+}
+function executeDispatch(event, listener, currentTarget) {
+  event.currentTarget = currentTarget;
+  listener(event);
+  event.currentTarget = null;
+}
function extractEvents(
  dispatchQueue,
  domEventName,
  targetInst,
  nativeEvent,
  nativeEventTarget,
  eventSystemFlags,
  targetContainer
) {
  SimpleEventPlugin.extractEvents(
    dispatchQueue,
    domEventName,
    targetInst,
    nativeEvent,
    nativeEventTarget,
    eventSystemFlags,
    targetContainer
  );
}

export function accumulateSinglePhaseListeners(targetFiber, reactName, nativeEventType, inCapturePhase) {
  const captureName = reactName + "Capture";
  const reactEventName = inCapturePhase ? captureName : reactName;
  const listeners = [];
  let instance = targetFiber;
  while (instance !== null) {
    const { stateNode, tag } = instance;
    if (tag
      if (reactEventName !== null) {
        const listener = getListener(instance, reactEventName);
        if (listener !== null && listener !== undefined) {
          listeners.push(createDispatchListener(instance, listener, stateNode));
        }
      }
    }
    instance = instance.return;
  }

  return listeners;
}
function createDispatchListener(instance, listener, currentTarget) {
  return {
    instance,
    listener,
    currentTarget,
  };
}
```

20.dispatchReducerAction

20.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";
+const reducer = (state, action) => {
+  if (action.type === "add") return state + 1;
+  return state;
+};
function FunctionComponent() {
+  const [number, setNumber] = React.useReducer(reducer, 0);
+  return  setNumber({ type: "add" })}>{number};
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

20.2 ReactFiberHooks.js

src\react-reconciler\src\ReactFiberHooks.js

```
+import ReactSharedInternals from "shared/ReactSharedInternals";
+
+const { ReactCurrentDispatcher } = ReactSharedInternals;
+let currentlyRenderingFiber = null;
+let workInProgressHook = null;
+
+function mountWorkInProgressHook() {
+  const hook = {
+    memoizedState: null,
+    queue: null,
+    next: null,
+  };
+  if (workInProgressHook === null) {
+    currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
+  } else {
+    workInProgressHook = workInProgressHook.next = hook;
+  }
+  return workInProgressHook;
+}
+function dispatchReducerAction(fiber, queue, action) {
+  console.log("dispatchReducerAction", action);
+}
+const HooksDispatcherOnMountInDEV = {
+  useReducer:mountReducer
+};
+function useReducer(reducer, initialArg){
+    const hook = mountWorkInProgressHook();
+    hook.memoizedState = initialArg;
+    const queue = {
+      pending: null,
+      dispatch: null,
+    };
+    hook.queue = queue;
+    const dispatch = (queue.dispatch = dispatchReducerAction.bind(null, currentlyRenderingFiber, queue));
+    return [hook.memoizedState, dispatch];
}
export function renderWithHooks(current, workInProgress, Component, props) {
+ currentlyRenderingFiber = workInProgress;
+ if (current !== null && current.memoizedState !== null) {
+ } else {
+   ReactCurrentDispatcher.current = HooksDispatcherOnMountInDEV;
+ }
  const children = Component(props);
+ currentlyRenderingFiber = null;
  return children;
}
```

20.3 react\index.js

src\react\index.js

```
export { __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED, useReducer } from "./src/React";
```

20.4 React.js

src\react\src\React.js

```
import { useReducer } from "./ReactHooks";
import ReactSharedInternals from "./ReactSharedInternals";

export { useReducer, ReactSharedInternals as __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED };
```

20.5 ReactHooks.js

src\react\src\ReactHooks.js

```
import ReactCurrentDispatcher from "./ReactCurrentDispatcher";

function resolveDispatcher() {
  const dispatcher = ReactCurrentDispatcher.current;
  return dispatcher;
}

export function useReducer(reducer, initialArg, init) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useReducer(reducer, initialArg, init);
}
```

20.6 ReactCurrentDispatcher.js

src\react\src\ReactCurrentDispatcher.js

```
const ReactCurrentDispatcher = {
  current: null,
};
export default ReactCurrentDispatcher;
```

src\react\src\ReactSharedInternals.js

```
import ReactCurrentDispatcher from "./ReactCurrentDispatcher";

const ReactSharedInternals = {
  ReactCurrentDispatcher,
};
export default ReactSharedInternals;
```

src\shared\ReactSharedInternals.js

```
import * as React from "react";

const ReactSharedInternals = React.__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED;
export default ReactSharedInternals;
```

21.HooksDispatcherOnUpdateInDEV

21.1 ReactFiberHooks.js

src\react-reconciler\src\ReactFiberHooks.js

```
import ReactSharedInternals from "shared/ReactSharedInternals";
+import { enqueueConcurrentHookUpdate } from "./ReactFiberConcurrentUpdates";
+import { scheduleUpdateOnFiber } from "./ReactFiberWorkLoop";

const { ReactCurrentDispatcher } = ReactSharedInternals;
let currentlyRenderingFiber = null;
let workInProgressHook = null;
+let currentHook = null;

function mountWorkInProgressHook() {
  const hook = {
    memoizedState: null,
    queue: null,
    next: null,
  };
  if (workInProgressHook
    currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
  } else {
    workInProgressHook = workInProgressHook.next = hook;
  }
  return workInProgressHook;
}
function dispatchReducerAction(fiber, queue, action) {
+ const update = {
+   action,
+   next: null,
+ };
+ const root = enqueueConcurrentHookUpdate(fiber, queue, update);
+ scheduleUpdateOnFiber(root, fiber);
}
const HooksDispatcherOnMountInDEV = {
  useReducer:mountReducer
};
function mountReducer(reducer, initialArg) {
    const hook = mountWorkInProgressHook();
    hook.memoizedState = initialArg;
    const queue = {
      pending: null,
      dispatch: null,
    };
    hook.queue = queue;
    const dispatch = (queue.dispatch = dispatchReducerAction.bind(null, currentlyRenderingFiber, queue));
    return [hook.memoizedState, dispatch];
}
+function updateWorkInProgressHook() {
+  if (currentHook === null) {
+    const current = currentlyRenderingFiber.alternate
+    currentHook = current.memoizedState
+  } else {
+    currentHook = currentHook.next
+  }
+  const newHook = {
+    memoizedState: currentHook.memoizedState,
+    queue: currentHook.queue,
+    next: null
+  }
+  if (workInProgressHook === null) {
+    currentlyRenderingFiber.memoizedState = workInProgressHook = newHook
+  } else {
+    workInProgressHook = workInProgressHook.next = newHook
+  }
+  return workInProgressHook
+}
+const HooksDispatcherOnUpdateInDEV = {
+  useReducer: updateReducer
+};
+function updateReducer(reducer) {
+  const hook = updateWorkInProgressHook()
+  const queue = hook.queue
+  queue.lastRenderedReducer = reducer
+  const current = currentHook
+  const pendingQueue = queue.pending
+  let newState = current.memoizedState
+  if (pendingQueue !== null) {
+    queue.pending = null
+    const first = pendingQueue.next
+    let update = first
+    do {
+      if (update.hasEagerState) {
+        newState = update.eagerState
+      } else {
+        const action = update.action
+        newState = reducer(newState, action)
+      }
+      update = update.next
+    } while (update !== null && update !== first)
+  }
+  hook.memoizedState = queue.lastRenderedState = newState
+  return [hook.memoizedState, queue.dispatch]
+},
export function renderWithHooks(current, workInProgress, Component, props) {
  currentlyRenderingFiber = workInProgress;
  if (current !== null && current.memoizedState !== null) {
+   ReactCurrentDispatcher.current = HooksDispatcherOnUpdateInDEV;
  } else {
    ReactCurrentDispatcher.current = HooksDispatcherOnMountInDEV;
  }
  const children = Component(props);
  currentlyRenderingFiber = null;
+ workInProgressHook = null;
+ currentHook = null;
  return children;
}
```

21.2 ReactFiberConcurrentUpdates.js

src\react-reconciler\src\ReactFiberConcurrentUpdates.js

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

+const concurrentQueues = [];
+let concurrentQueuesIndex = 0;

export function markUpdateLaneFromFiberToRoot(sourceFiber) {
  let node = sourceFiber;
  let parent = sourceFiber.return;
  while (parent !== null) {
    node = parent;
    parent = parent.return;
  }
  if (node.tag
    const root = node.stateNode;
    return root;
  }
  return null;
}
+export function enqueueConcurrentHookUpdate(fiber, queue, update) {
+  enqueueUpdate(fiber, queue, update);
+  return getRootForUpdatedFiber(fiber);
+}
+function enqueueUpdate(fiber, queue, update) {
+  concurrentQueues[concurrentQueuesIndex++] = fiber;
+  concurrentQueues[concurrentQueuesIndex++] = queue;
+  concurrentQueues[concurrentQueuesIndex++] = update;
+}
+function getRootForUpdatedFiber(sourceFiber) {
+  let node = sourceFiber;
+  let parent = node.return;
+  while (parent !== null) {
+    node = parent;
+    parent = node.return;
+  }
+  return node.tag === HostRoot ? node.stateNode : null;
+}

+export function finishQueueingConcurrentUpdates() {
+  const endIndex = concurrentQueuesIndex;
+  concurrentQueuesIndex = 0;
+  let i = 0;
+  while (i < endIndex) {
+    const fiber = concurrentQueues[i++];
+    const queue = concurrentQueues[i++];
+    const update = concurrentQueues[i++];
+    if (queue !== null && update !== null) {
+      const pending = queue.pending;
+      if (pending === null) {
+        update.next = update;
+      } else {
+        update.next = pending.next;
+        pending.next = update;
+      }
+      queue.pending = update;
+    }
+  }
+}
```

21.3 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import { scheduleCallback } from "scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
import { MutationMask, NoFlags } from "./ReactFiberFlags";
import { commitMutationEffectsOnFiber } from "./ReactFiberCommitWork";
+import { finishQueueingConcurrentUpdates } from "./ReactFiberConcurrentUpdates";

let workInProgress = null;
export function scheduleUpdateOnFiber(root) {
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  scheduleCallback(performConcurrentWorkOnRoot.bind(null, root));
}
function performConcurrentWorkOnRoot(root) {
  renderRootSync(root);
  const finishedWork = root.current.alternate;
  root.finishedWork = finishedWork;
  commitRoot(root);
}
function commitRoot(root) {
  const { finishedWork } = root;
  const subtreeHasEffects = (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
    commitMutationEffectsOnFiber(finishedWork, root);
  }
  root.current = finishedWork;
}
function prepareFreshStack(root) {
  workInProgress = createWorkInProgress(root.current, null);
+ finishQueueingConcurrentUpdates();
}
function renderRootSync(root) {
  prepareFreshStack(root);
  workLoopSync();
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
}
```

21.4 ReactFiberBeginWork.js

src\react-reconciler\src\ReactFiberBeginWork.js

```
import { HostRoot, HostComponent, HostText, IndeterminateComponent, FunctionComponent } from "./ReactWorkTags";
import { processUpdateQueue } from "./ReactFiberClassUpdateQueue";
import { mountChildFibers, reconcileChildFibers } from "./ReactChildFiber";
import { shouldSetTextContent } from "react-dom-bindings/src/client/ReactDOMHostConfig";
import logger, { indent } from "shared/logger";
import { renderWithHooks } from "react-reconciler/src/ReactFiberHooks";
function reconcileChildren(current, workInProgress, nextChildren) {
  if (current
    workInProgress.child = mountChildFibers(workInProgress, null, nextChildren);
  } else {
    workInProgress.child = reconcileChildFibers(workInProgress, current.child, nextChildren);
  }
}
function updateHostRoot(current, workInProgress) {
  processUpdateQueue(workInProgress);
  const nextState = workInProgress.memoizedState;
  const nextChildren = nextState.element;
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
function updateHostComponent(current, workInProgress) {
  const { type } = workInProgress;
  const nextProps = workInProgress.pendingProps;
  let nextChildren = nextProps.children;
  const isDirectTextChild = shouldSetTextContent(type, nextProps);
  if (isDirectTextChild) {
    nextChildren = null;
  }
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
function mountIndeterminateComponent(_current, workInProgress, Component) {
  const props = workInProgress.pendingProps;
  const value = renderWithHooks(null, workInProgress, Component, props);
  workInProgress.tag = FunctionComponent;
  reconcileChildren(null, workInProgress, value);
  return workInProgress.child;
}
+function updateFunctionComponent(current, workInProgress, Component, nextProps) {
+  const nextChildren = renderWithHooks(current, workInProgress, Component, nextProps);
+  reconcileChildren(current, workInProgress, nextChildren);
+  return workInProgress.child;
+}
export function beginWork(current, workInProgress) {
  //logger(" ".repeat(indent.number) + "beginWork", workInProgress);
  indent.number += 2;
  switch (workInProgress.tag) {
    case IndeterminateComponent: {
      return mountIndeterminateComponent(current, workInProgress, workInProgress.type);
    }
+   case FunctionComponent: {
+     const Component = workInProgress.type;
+     const resolvedProps = workInProgress.pendingProps;
+     return updateFunctionComponent(current, workInProgress, Component, resolvedProps);
+   }
    case HostRoot:
      return updateHostRoot(current, workInProgress);
    case HostComponent:
      return updateHostComponent(current, workInProgress);
    case HostText:
    default:
      return null;
  }
}
```

21.5 ReactChildFiber.js

src\react-reconciler\src\ReactChildFiber.js

```
import { REACT_ELEMENT_TYPE } from "shared/ReactSymbols";
import isArray from "shared/isArray";
import { createFiberFromElement, FiberNode, createFiberFromText, createWorkInProgress } from "./ReactFiber";
import { Placement } from "./ReactFiberFlags";
import { HostText } from "./ReactWorkTags";
function createChildReconciler(shouldTrackSideEffects) {
+ function useFiber(fiber, pendingProps) {
+   const clone = createWorkInProgress(fiber, pendingProps);
+   clone.index = 0;
+   clone.sibling = null;
+   return clone;
+ }
  function reconcileSingleElement(returnFiber, currentFirstChild, element) {
+   const key = element.key;
+   let child = currentFirstChild;
+   while (child !== null) {
+     if (child.key === key) {
+       const elementType = element.type;
+       if (child.type === elementType) {
+         const existing = useFiber(child, element.props);
+         existing.return = returnFiber;
+         return existing;
+       }
+     }
+     child = child.sibling;
+   }
    const created = createFiberFromElement(element);
    created.return = returnFiber;
    return created;
  }
  function placeSingleChild(newFiber) {
+   if (shouldTrackSideEffects && newFiber.alternate === null) {
      newFiber.flags |= Placement;
    }
    return newFiber;
```

```
  }
  function reconcileSingleTextNode(returnFiber, currentFirstChild, content) {
    const created = new FiberNode(HostText, { content }, null);
    created.return = returnFiber;
    return created;
  }
  function createChild(returnFiber, newChild) {
    if ((typeof newChild
      const created = createFiberFromText(`${newChild}`);
      created.return = returnFiber;
      return created;
    }

    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          const created = createFiberFromElement(newChild);
          created.return = returnFiber;
          return created;
        }
        default:
          break;
      }
    }
    return null;
  }
  function placeChild(newFiber, newIndex) {
    newFiber.index = newIndex;
    if (shouldTrackSideEffects) newFiber.flags |= Placement;
  }
  function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
    let resultingFirstChild = null;
    let previousNewFiber = null;
    let newIdx = 0;
    for (; newIdx < newChildren.length; newIdx++) {
      const newFiber = createChild(returnFiber, newChildren[newIdx]);
      if (newFiber
        continue;
      }
      placeChild(newFiber, newIdx);
      if (previousNewFiber
        resultingFirstChild = newFiber;
      } else {
        previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
    }
    return resultingFirstChild;
  }
  function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          return placeSingleChild(reconcileSingleElement(returnFiber, currentFirstChild, newChild));
        }
        default:
          break;
      }
      if (isArray(newChild)) {
        return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
      }
    }
    if (typeof newChild
      return placeSingleChild(reconcileSingleTextNode(returnFiber, currentFirstChild, newChild));
    }
    return null;
  }
  return reconcileChildFibers;
}
export const reconcileChildFibers = createChildReconciler(true);
export const mountChildFibers = createChildReconciler(false);
```

21.6 ReactFiberCompleteWork.js

src\react-reconciler\src\ReactFiberCompleteWork.js

```
import {
  appendInitialChild,
  createInstance,
  createTextInstance,
  finalizeInitialChildren,
+ prepareUpdate,
} from "react-dom-bindings/src/client/ReactDOMHostConfig";
import { HostComponent, HostRoot, HostText } from "./ReactWorkTags";
+import { NoFlags, Update } from "./ReactFiberFlags";
import logger, { indent } from "shared/logger";
function bubbleProperties(completedWork) {
  let subtreeFlags = NoFlags;
  let child = completedWork.child;
  while (child !== null) {
    subtreeFlags |= child.subtreeFlags;
    subtreeFlags |= child.flags;
    child = child.sibling;
  }
  completedWork.subtreeFlags |= subtreeFlags;
}

function appendAllChildren(parent, workInProgress) {
  // 我们只有创建的顶级fiber，但需要递归其子节点来查找所有终端节点
  let node = workInProgress.child;
  while (node !== null) {
    // 如果是原生节点，直接添加到父节点上
    if (node.tag
      appendInitialChild(parent, node.stateNode);
      // 再看看第一个节节点是不是原生节点
    } else if (node.child !== null) {
      // node.child.return = node
      node = node.child;
      continue;
    }
    if (node
      return;
    }
    // 如果没有弟弟就找父亲的弟弟
    while (node.sibling
      // 如果找到了根节点或者回到了原节点结束
      if (node.return
        return;
      }
      node = node.return;
    }
    // node.sibling.return = node.return
    // 下一个弟弟节点
    node = node.sibling;
  }
}
+function markUpdate(workInProgress) {
+  workInProgress.flags |= Update;
+}
+function updateHostComponent(current, workInProgress, type, newProps) {
+  const oldProps = current.memoizedProps;
+  const instance = workInProgress.stateNode;
+  const updatePayload = prepareUpdate(instance, type, oldProps, newProps);
+  workInProgress.updateQueue = updatePayload;
+  if (updatePayload) {
+    markUpdate(workInProgress);
+  }
+}
export function completeWork(current, workInProgress) {
  indent.number -= 2;
  //logger(" ".repeat(indent.number) + "completeWork", workInProgress);
  const newProps = workInProgress.pendingProps;
  switch (workInProgress.tag) {
    case HostComponent: {
      const { type } = workInProgress;
+     if (current !== null && workInProgress.stateNode != null) {
+       updateHostComponent(current, workInProgress, type, newProps);
+       console.log("updatePayload", workInProgress.updateQueue);
+     } else {
        const instance = createInstance(type, newProps, workInProgress);
        appendAllChildren(instance, workInProgress);
        workInProgress.stateNode = instance;
        finalizeInitialChildren(instance, type, newProps);
+     }
+     bubbleProperties(workInProgress);
      break;
    }
    case HostRoot:
      bubbleProperties(workInProgress);
      break;
    case HostText: {
      const newText = newProps;
      workInProgress.stateNode = createTextInstance(newText);
      bubbleProperties(workInProgress);
      break;
    }
    default:
      break;
  }
}
```

21.7 ReactFiberFlags.js

src\react-reconciler\src\ReactFiberFlags.js

```
export const NoFlags = 0b00000000000000000000000000;
export const Placement = 0b00000000000000000000000010;
+export const Update = 0b00000000000000000000000100;
+export const MutationMask = Placement | Update;
```

21.8 ReactDOMHostConfig.js

src\react-dom-bindings\src\client\ReactDOMHostConfig.js

```
+import { setInitialProperties, diffProperties } from "./ReactDOMComponent";
import { precacheFiberNode, updateFiberProps } from "./ReactDOMComponentTree";

export function shouldSetTextContent(type, props) {
  return typeof props.children
}
export const appendInitialChild = (parent, child) => {
  parent.appendChild(child);
};
export const createInstance = (type, props, internalInstanceHandle) => {
  const domElement = document.createElement(type);
  precacheFiberNode(internalInstanceHandle, domElement);
  updateFiberProps(domElement, props);
  return domElement;
};
export const createTextInstance = (content) => document.createTextNode(content);
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);
}

+export function prepareUpdate(domElement, type, oldProps, newProps) {
+  return diffProperties(domElement, type, oldProps, newProps);
+}
```

21.9 ReactDOMComponent.js

src\react-dom-bindings\src\client\ReactDOMComponent.js

```
import { setValueForStyles } from "./CSSPropertyOperations";
import setTextContent from "./setTextContent";
import { setValueForProperty } from "./DOMPropertyOperations";
const CHILDREN = "children";
const STYLE = "style";
function setInitialDOMProperties(tag, domElement, nextProps) {
  for (const propKey in nextProps) {
    if (nextProps.hasOwnProperty(propKey)) {
      const nextProp = nextProps[propKey];
      if (propKey
        setValueForStyles(domElement, nextProp);
      } else if (propKey
        if (typeof nextProp
          setTextContent(domElement, nextProp);
        } else if (typeof nextProp
          setTextContent(domElement, `${nextProp}`);
        }
      } else if (nextProp != null) {
        setValueForProperty(domElement, propKey, nextProp);
      }
    }
  }
}
export function setInitialProperties(domElement, tag, props) {
  setInitialDOMProperties(tag, domElement, props);
}

+export function diffProperties(domElement, tag, lastProps, nextProps) {
+  let updatePayload = null;
+  let propKey;
+  let styleName;
+  let styleUpdates = null;
+  for (propKey in lastProps) {
+    if (nextProps.hasOwnProperty(propKey) || !lastProps.hasOwnProperty(propKey) || lastProps[propKey] == null) {
+      continue;
+    }
+    if (propKey === STYLE) {
+      const lastStyle = lastProps[propKey];
+      for (styleName in lastStyle) {
+        if (lastStyle.hasOwnProperty(styleName)) {
+          if (!styleUpdates) {
+            styleUpdates = {};
+          }
+          styleUpdates[styleName] = "";
+        }
+      }
+    } else {
+      (updatePayload = updatePayload || []).push(propKey, null);
+    }
+  }
+  for (propKey in nextProps) {
+    const nextProp = nextProps[propKey];
+    const lastProp = lastProps != null ? lastProps[propKey] : undefined;
+    if (!nextProps.hasOwnProperty(propKey) || nextProp === lastProp || (nextProp == null && lastProp == null)) {
+      continue;
+    }
+    if (propKey === STYLE) {
+      if (lastProp) {
+        for (styleName in lastProp) {
+          if (lastProp.hasOwnProperty(styleName) && (!nextProp || !nextProp.hasOwnProperty(styleName))) {
+            if (!styleUpdates) {
+              styleUpdates = {};
+            }
+            styleUpdates[styleName] = "";
+          }
+        }
+        for (styleName in nextProp) {
+          if (nextProp.hasOwnProperty(styleName) && lastProp[styleName] !== nextProp[styleName]) {
+            if (!styleUpdates) {
+              styleUpdates = {};
+            }
+            styleUpdates[styleName] = nextProp[styleName];
+          }
+        }
+      } else {
+        if (!styleUpdates) {
+          if (!updatePayload) {
+            updatePayload = [];
+          }
+          updatePayload.push(propKey, styleUpdates);
+        }
+        styleUpdates = nextProp;
+      }
+    } else if (propKey === CHILDREN) {
+      if (typeof nextProp === "string" || typeof nextProp === "number") {
+        (updatePayload = updatePayload || []).push(propKey, "" + nextProp);
+      }
+    } else {
+      (updatePayload = updatePayload || []).push(propKey, nextProp);
+    }
+  }
+  if (styleUpdates) {
+    (updatePayload = updatePayload || []).push(STYLE, styleUpdates);
+  }
+  return updatePayload;
+}
```

22.commitUpdate

22.1 DOMPluginEventSystem.js

src\react-dom-bindings\src\events\DOMPluginEventSystem.js

```
import { allNativeEvents } from "./EventRegistry";
import * as SimpleEventPlugin from "./plugins/SimpleEventPlugin";
import { createEventListenerWrapperWithPriority } from "./ReactDOMEventListener";
import { IS_CAPTURE_PHASE } from "./EventSystemFlags";
import { addEventCaptureListener, addEventBubbleListener } from "./EventListener";
import getEventTarget from "./getEventTarget";
import getListener from "./getListener";
import { HostComponent } from "react-reconciler/src/ReactWorkTags";

SimpleEventPlugin.registerEvents();
+const listeningMarker = "_reactListening" + Math.random().toString(36).slice(2);
export function listenToAllSupportedEvents(rootContainerElement) {
+ if (!rootContainerElement[listeningMarker]) {
+   rootContainerElement[listeningMarker] = true;
    allNativeEvents.forEach((domEventName) => {
      listenToNativeEvent(domEventName, true, rootContainerElement);
      listenToNativeEvent(domEventName, false, rootContainerElement);
    });
+ }
}

export function listenToNativeEvent(domEventName, isCapturePhaseListener, target) {
  let eventSystemFlags = 0; // 冒泡 = 0 捕获 = 4
  if (isCapturePhaseListener) {
    eventSystemFlags |= IS_CAPTURE_PHASE;
  }
  addTrappedEventListener(target, domEventName, eventSystemFlags, isCapturePhaseListener);
}
function addTrappedEventListener(targetContainer, domEventName, eventSystemFlags, isCapturePhaseListener) {
  const listener = createEventListenerWrapperWithPriority(targetContainer, domEventName, eventSystemFlags);
  if (isCapturePhaseListener) {
    addEventCaptureListener(targetContainer, domEventName, listener);
  } else {
    addEventBubbleListener(targetContainer, domEventName, listener);
  }
}
export function dispatchEventForPluginEventSystem(
  domEventName,
  eventSystemFlags,
  nativeEvent,
  targetInst,
  targetContainer
) {
  dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent, targetInst, targetContainer);
}

function dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent, targetInst, targetContainer) {
  const nativeEventTarget = getEventTarget(nativeEvent);
  const dispatchQueue = [];
  extractEvents(
    dispatchQueue,
    domEventName,
    targetInst,
    nativeEvent,
    nativeEventTarget,
    eventSystemFlags,
    targetContainer
  );
  processDispatchQueue(dispatchQueue, eventSystemFlags);
}

export function processDispatchQueue(dispatchQueue, eventSystemFlags) {
  const inCapturePhase = (eventSystemFlags & IS_CAPTURE_PHASE) !== 0;
  for (let i = 0; i < dispatchQueue.length; i++) {
    const { event, listeners } = dispatchQueue[i];
    processDispatchQueueItemsInOrder(event, listeners, inCapturePhase); //  event system doesn't use pooling.

  }
}
function processDispatchQueueItemsInOrder(event, dispatchListeners, inCapturePhase) {
  if (inCapturePhase) {
    for (let i = dispatchListeners.length - 1; i >= 0; i--) {
      const { currentTarget, listener } = dispatchListeners[i];
      if (event.isPropagationStopped()) {
        return;
      }
      executeDispatch(event, listener, currentTarget);
    }
  } else {
    for (let i = 0; i < dispatchListeners.length; i++) {
      const { currentTarget, listener } = dispatchListeners[i];
      if (event.isPropagationStopped()) {
        return;
      }
      executeDispatch(event, listener, currentTarget);
    }
  }
}
function executeDispatch(event, listener, currentTarget) {
  event.currentTarget = currentTarget;
  listener(event);
  event.currentTarget = null;
}
function extractEvents(
  dispatchQueue,
  domEventName,
  targetInst,
  nativeEvent,
  nativeEventTarget,
  eventSystemFlags,
  targetContainer
) {
  SimpleEventPlugin.extractEvents(
```

```
    dispatchQueue,
    domEventName,
    targetInst,
    nativeEvent,
    nativeEventTarget,
    eventSystemFlags,
    targetContainer
  );
}

export function accumulateSinglePhaseListeners(targetFiber, reactName, nativeEventType, inCapturePhase) {
  const captureName = reactName + "Capture";
  const reactEventName = inCapturePhase ? captureName : reactName;
  const listeners = [];
  let instance = targetFiber;
  while (instance !== null) {
    const { stateNode, tag } = instance;
    if (tag
      if (reactEventName !== null) {
        const listener = getListener(instance, reactEventName);
        if (listener !== null && listener !== undefined) {
          listeners.push(createDispatchListener(instance, listener, stateNode));
        }
      }
    }
    instance = instance.return;
  }

  return listeners;
}
function createDispatchListener(instance, listener, currentTarget) {
  return {
    instance,
    listener,
    currentTarget,
  };
}
```

22.2 ReactFiberCompleteWork.js

src\react-reconciler\src\ReactFiberCompleteWork.js

```
import {
  appendInitialChild,
  createInstance,
  createTextInstance,
  finalizeInitialChildren,
  prepareUpdate,
} from "react-dom-bindings/src/client/ReactDOMHostConfig";
+import { HostComponent, HostRoot, HostText, FunctionComponent } from "./ReactWorkTags";
import { NoFlags, Update } from "./ReactFiberFlags";
import logger, { indent } from "shared/logger";
function bubbleProperties(completedWork) {
  let subtreeFlags = NoFlags;
  let child = completedWork.child;
  while (child !== null) {
    subtreeFlags |= child.subtreeFlags;
    subtreeFlags |= child.flags;
    child = child.sibling;
  }
  completedWork.subtreeFlags |= subtreeFlags;
}

function appendAllChildren(parent, workInProgress) {
  // 我们只有创建的顶级fiber，但需要递归其子节点来查找所有终端节点
  let node = workInProgress.child;
  while (node !== null) {
    // 如果是原生节点，直接添加到父节点上
    if (node.tag
      appendInitialChild(parent, node.stateNode);
      // 再看看第一个节节点是不是原生节点
    } else if (node.child !== null) {
      // node.child.return = node
      node = node.child;
      continue;
    }
    if (node
      return;
    }
    // 如果没有弟弟就找父亲的弟弟
    while (node.sibling
      // 如果找到了根节点或者回到了原节点结束
      if (node.return
        return;
      }
      node = node.return;
    }
    // node.sibling.return = node.return
    // 下一个弟弟节点
    node = node.sibling;
  }
}
function markUpdate(workInProgress) {
  workInProgress.flags |= Update;
}
function updateHostComponent(current, workInProgress, type, newProps) {
  const oldProps = current.memoizedProps;
  const instance = workInProgress.stateNode;
  const updatePayload = prepareUpdate(instance, type, oldProps, newProps);
  workInProgress.updateQueue = updatePayload;
  if (updatePayload) {
    markUpdate(workInProgress);
  }
}
export function completeWork(current, workInProgress) {
  indent.number -= 2;
  //logger(" ".repeat(indent.number) + "completeWork", workInProgress);
  const newProps = workInProgress.pendingProps;
  switch (workInProgress.tag) {
    case HostComponent: {
      const { type } = workInProgress;
      if (current !== null && workInProgress.stateNode != null) {
        updateHostComponent(current, workInProgress, type, newProps);
      } else {
        const instance = createInstance(type, newProps, workInProgress);
        appendAllChildren(instance, workInProgress);
        workInProgress.stateNode = instance;
        finalizeInitialChildren(instance, type, newProps);
      }
      bubbleProperties(workInProgress);
      break;
    }
+   case FunctionComponent:
+     bubbleProperties(workInProgress);
+     break;
    case HostRoot:
      bubbleProperties(workInProgress);
      break;
    case HostText: {
      const newText = newProps;
      workInProgress.stateNode = createTextInstance(newText);
      bubbleProperties(workInProgress);
      break;
    }
    default:
      break;
  }
}
```

22.3 ReactFiberCommitWork.js

src\react-reconciler\src\ReactFiberCommitWork.js

```
+import { HostRoot, HostComponent, HostText, FunctionComponent } from "./ReactWorkTags";
+import { MutationMask, Placement, Update } from "./ReactFiberFlags";
+import { insertBefore, appendChild, commitUpdate } from "react-dom-bindings/src/client/ReactDOMHostConfig";
```

```
function recursivelyTraverseMutationEffects(root, parentFiber) {
  if (parentFiber.subtreeFlags & MutationMask) {
    let { child } = parentFiber;
    while (child !== null) {
      commitMutationEffectsOnFiber(child, root);
      child = child.sibling;
    }
  }
}
function isHostParent(fiber) {
  return fiber.tag
}
function getHostParentFiber(fiber) {
  let parent = fiber.return;
  while (parent !== null) {
    if (isHostParent(parent)) {
      return parent;
    }
    parent = parent.return;
  }
  return parent;
}
function insertOrAppendPlacementNode(node, before, parent) {
  const { tag } = node;
  const isHost = tag
  if (isHost) {
    const { stateNode } = node;
    if (before) {
      insertBefore(parent, stateNode, before);
    } else {
      appendChild(parent, stateNode);
    }
  } else {
    const { child } = node;
    if (child !== null) {
      insertOrAppendPlacementNode(child, before, parent);
      let { sibling } = child;
      while (sibling !== null) {
        insertOrAppendPlacementNode(sibling, before, parent);
        sibling = sibling.sibling;
      }
    }
  }
}
function getHostSibling(fiber) {
  let node = fiber;
  siblings: while (true) {
    // 如果我们没有找到任何东西，让我们试试下一个弟弟
    while (node.sibling
      if (node.return
        // 如果我们是根Fiber或者父亲是原生节点，我们就是最后的弟弟
        return null;
      }
      node = node.return;
    }
    // node.sibling.return = node.return
    node = node.sibling;
    while (node.tag !== HostComponent && node.tag !== HostText) {
      // 如果它不是原生节点，并且，我们可能在其中有一个原生节点
      // 试着向下搜索，直到找到为止
      if (node.flags & Placement) {
        // 如果我们没有孩子，可以试试弟弟
        continue siblings;
      } else {
        // node.child.return = node
        node = node.child;
      }
    } // Check if this host node is stable or about to be placed.

    // 检查此原生节点是否稳定可以放置
    if (!(node.flags & Placement)) {
      // 找到它了!

      return node.stateNode;
    }
  }
}
function commitPlacement(finishedWork) {
  const parentFiber = getHostParentFiber(finishedWork);
  switch (parentFiber.tag) {
    case HostComponent: {
      const parent = parentFiber.stateNode;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    case HostRoot: {
      const parent = parentFiber.stateNode.containerInfo;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    default:
      break;
  }
}
function commitReconciliationEffects(finishedWork) {
  const { flags } = finishedWork;
  if (flags & Placement) {
    commitPlacement(finishedWork);
    finishedWork.flags &= ~Placement;
  }
}
export function commitMutationEffectsOnFiber(finishedWork, root) {
```

```
+ const current = finishedWork.alternate;
+ const flags = finishedWork.flags;
  switch (finishedWork.tag) {
+    case HostRoot: {
+      recursivelyTraverseMutationEffects(root, finishedWork);
+      commitReconciliationEffects(finishedWork);
+      break;
+    }
+    case FunctionComponent: {
+      recursivelyTraverseMutationEffects(root, finishedWork);
+      commitReconciliationEffects(finishedWork);
+      break;
+    }
+    case HostComponent: {
+      recursivelyTraverseMutationEffects(root, finishedWork);
+      commitReconciliationEffects(finishedWork);
+      if (flags & Update) {
+        const instance = finishedWork.stateNode;
+        if (instance != null) {
+          const newProps = finishedWork.memoizedProps;
+          const oldProps = current !== null ? current.memoizedProps : newProps;
+          const type = finishedWork.type;
+          const updatePayload = finishedWork.updateQueue;
+          finishedWork.updateQueue = null;
+          if (updatePayload !== null) {
+            commitUpdate(instance, updatePayload, type, oldProps, newProps, finishedWork);
+          }
+        }
+      }
+      break;
+    }
+    case HostText: {
+      recursivelyTraverseMutationEffects(root, finishedWork);
+      commitReconciliationEffects(finishedWork);
+      break;
+    }
    default: {
      break;
    }
  }
}
```

22.4 ReactDOMHostConfig.js

src\react-dom-bindings\src\client\ReactDOMHostConfig.js

```
+import { setInitialProperties, diffProperties, updateProperties } from "./ReactDOMComponent";
import { precacheFiberNode, updateFiberProps } from "./ReactDOMComponentTree";

export function shouldSetTextContent(type, props) {
  return typeof props.children
}
export const appendInitialChild = (parent, child) => {
  parent.appendChild(child);
};
export const createInstance = (type, props, internalInstanceHandle) => {
  const domElement = document.createElement(type);
  precacheFiberNode(internalInstanceHandle, domElement);
  updateFiberProps(domElement, props);
  return domElement;
};
export const createTextInstance = (content) => document.createTextNode(content);
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);
}

export function prepareUpdate(domElement, type, oldProps, newProps) {
  return diffProperties(domElement, type, oldProps, newProps);
}

+export function commitUpdate(domElement, updatePayload, type, oldProps, newProps) {
+  updateProperties(domElement, updatePayload, type, oldProps, newProps);
+  updateFiberProps(domElement, newProps);
+}
```

22.5 ReactDOMComponent.js

src\react-dom-bindings\src\client\ReactDOMComponent.js

```
import { setValueForStyles } from "./CSSPropertyOperations";
import setTextContent from "./setTextContent";
import { setValueForProperty } from "./DOMPropertyOperations";
const CHILDREN = "children";
const STYLE = "style";
function setInitialDOMProperties(tag, domElement, nextProps) {
  for (const propKey in nextProps) {
    if (nextProps.hasOwnProperty(propKey)) {
      const nextProp = nextProps[propKey];
      if (propKey
        setValueForStyles(domElement, nextProp);
      } else if (propKey
        if (typeof nextProp
          setTextContent(domElement, nextProp);
        } else if (typeof nextProp
          setTextContent(domElement, `${nextProp}`);
        }
      } else if (nextProp != null) {
```

```
        setValueForProperty(domElement, propKey, nextProp);
      }
    }
  }
}
export function setInitialProperties(domElement, tag, props) {
  setInitialDOMProperties(tag, domElement, props);
}

export function diffProperties(domElement, tag, lastProps, nextProps) {
  let updatePayload = null;
  let propKey;
  let styleName;
  let styleUpdates = null;
  for (propKey in lastProps) {
    if (nextProps.hasOwnProperty(propKey) || !lastProps.hasOwnProperty(propKey) || lastProps[propKey] == null) {
      continue;
    }
    if (propKey
      const lastStyle = lastProps[propKey];
      for (styleName in lastStyle) {
        if (lastStyle.hasOwnProperty(styleName)) {
          if (!styleUpdates) {
            styleUpdates = {};
          }
          styleUpdates[styleName] = "";
        }
      }
    } else {
      (updatePayload = updatePayload || []).push(propKey, null);
    }
  }
  for (propKey in nextProps) {
    const nextProp = nextProps[propKey];
    const lastProp = lastProps != null ? lastProps[propKey] : undefined;
    if (!nextProps.hasOwnProperty(propKey) || nextProp
      continue;
    }
    if (propKey
      if (lastProp) {
        for (styleName in lastProp) {
          if (lastProp.hasOwnProperty(styleName) && (!nextProp || !nextProp.hasOwnProperty(styleName))) {
            if (!styleUpdates) {
              styleUpdates = {};
            }
            styleUpdates[styleName] = "";
          }
        }
        for (styleName in nextProp) {
          if (nextProp.hasOwnProperty(styleName) && lastProp[styleName] !== nextProp[styleName]) {
            if (!styleUpdates) {
              styleUpdates = {};
            }
            styleUpdates[styleName] = nextProp[styleName];
          }
        }
      } else {
        if (!styleUpdates) {
          if (!updatePayload) {
            updatePayload = [];
          }
          updatePayload.push(propKey, styleUpdates);
        }
        styleUpdates = nextProp;
      }
    } else if (propKey
      if (typeof nextProp
        (updatePayload = updatePayload || []).push(propKey, "" + nextProp);
      }
    } else {
      (updatePayload = updatePayload || []).push(propKey, nextProp);
    }
  }
  if (styleUpdates) {
    (updatePayload = updatePayload || []).push(STYLE, styleUpdates);
  }
  return updatePayload;
}

+export function updateProperties(domElement, updatePayload) {
+  updateDOMProperties(domElement, updatePayload);
+}
+
+function updateDOMProperties(domElement, updatePayload) {
+  for (let i = 0; i < updatePayload.length; i += 2) {
+    const propKey = updatePayload[i];
+    const propValue = updatePayload[i + 1];
+    if (propKey === STYLE) {
+      setValueForStyles(domElement, propValue);
+    } else if (propKey === CHILDREN) {
+      setTextContent(domElement, propValue);
+    } else {
+      setValueForProperty(domElement, propKey, propValue);
+    }
+  }
+}
```

23.useState

23.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

function FunctionComponent() {
  console.log("FunctionComponent render");
+ const [number, setNumber] = React.useState(0);
+ return  setNumber(number + 1)}>{number};
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

23.2 react\index.js

src\react\index.js

```
+export { __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED, useReducer, useState } from "./src/React";
```

23.3 React.js

src\react\src\React.js

```
+import { useReducer, useState } from "./ReactHooks";
import ReactSharedInternals from "./ReactSharedInternals";

+export { useReducer, useState, ReactSharedInternals as __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED };
```

23.4 ReactHooks.js

src\react\src\ReactHooks.js

```
import ReactCurrentDispatcher from "./ReactCurrentDispatcher";

function resolveDispatcher() {
  const dispatcher = ReactCurrentDispatcher.current;
  return dispatcher;
}

export function useReducer(reducer, initialArg, init) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useReducer(reducer, initialArg, init);
}

+export function useState(initialState) {
+  const dispatcher = resolveDispatcher();
+  return dispatcher.useState(initialState);
+}
```

23.5 objectIs.js

src\shared\objectIs.js

```
const objectIs = Object.is;
export default objectIs;
```

23.6 ReactFiberHooks.js

src\react-reconciler\src\ReactFiberHooks.js

```
import ReactSharedInternals from "shared/ReactSharedInternals";
import { enqueueConcurrentHookUpdate } from "./ReactFiberConcurrentUpdates";
import { scheduleUpdateOnFiber } from "./ReactFiberWorkLoop";
+import is from "shared/objectIs";

const { ReactCurrentDispatcher } = ReactSharedInternals;
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;

const HooksDispatcherOnMountInDEV = {
  useReducer: mountReducer,
+ useState: mountState,
};
const HooksDispatcherOnUpdateInDEV = {
  useReducer: updateReducer,
+ useState: updateState,
};
+function basicStateReducer(state, action) {
+  return typeof action === "function" ? action(state) : action;
+}
function mountReducer(reducer, initialArg) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialArg;
  const queue = {
    pending: null,
    dispatch: null,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchReducerAction.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function updateReducer(reducer) {
  const hook = updateWorkInProgressHook()
  const queue = hook.queue
  queue.lastRenderedReducer = reducer
  const current = currentHook
  const pendingQueue = queue.pending
  let newState = current.memoizedState
  if (pendingQueue !== null) {
    queue.pending = null
    const first = pendingQueue.next
    let update = first
    do {
```

```
      if (update.hasEagerState) {
        newState = update.eagerState
      } else {
        const action = update.action
        newState = reducer(newState, action)
      }
      update = update.next
    } while (update !== null && update !== first)
  }
  hook.memoizedState = queue.lastRenderedState = newState
  return [hook.memoizedState, queue.dispatch]
}
+function mountState(initialState) {
+  const hook = mountWorkInProgressHook();
+  hook.memoizedState =  hook.baseState = initialState;
+  const queue = {
+    pending: null,
+    dispatch: null,
+    lastRenderedReducer: basicStateReducer,
+    lastRenderedState: initialState,
+  };
+  hook.queue = queue;
+  const dispatch = (queue.dispatch = dispatchSetState.bind(null, currentlyRenderingFiber, queue));
+  return [hook.memoizedState, dispatch];
+}
+function dispatchSetState(fiber, queue, action) {
+  const update = {
+    action,
+    hasEagerState: false,
+    eagerState: null,
+    next: null,
+  };
+  const lastRenderedReducer = queue.lastRenderedReducer;
+  const currentState = queue.lastRenderedState;
+  const eagerState = lastRenderedReducer(currentState, action);
+  update.hasEagerState = true;
+  update.eagerState = eagerState;
+  if (is(eagerState, currentState)) {
+    return;
+  }
+  const root = enqueueConcurrentHookUpdate(fiber, queue, update);
+  scheduleUpdateOnFiber(root, fiber);
+}
+function updateState(initialState) {
+  return updateReducer(basicStateReducer, initialState);
+}
function mountWorkInProgressHook() {
  const hook = {
    memoizedState: null,
    queue: null,
    next: null,
  };
  if (workInProgressHook
    currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
  } else {
    workInProgressHook = workInProgressHook.next = hook;
  }
  return workInProgressHook;
}
function dispatchReducerAction(fiber, queue, action) {
  const update = {
    action,
    next: null,
  };
  const root = enqueueConcurrentHookUpdate(fiber, queue, update);
  scheduleUpdateOnFiber(root, fiber);
}

function updateWorkInProgressHook() {
  if (currentHook
    const current = currentlyRenderingFiber.alternate
    currentHook = current.memoizedState
  } else {
    currentHook = currentHook.next
  }
  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 renderWithHooks(current, workInProgress, Component, props) {
  currentlyRenderingFiber = workInProgress;
  if (current !== null && current.memoizedState !== null) {
    ReactCurrentDispatcher.current = HooksDispatcherOnUpdateInDEV;
  } else {
    ReactCurrentDispatcher.current = HooksDispatcherOnMountInDEV;
  }
  const children = Component(props);
  currentlyRenderingFiber = null;
  workInProgressHook = null;
  currentHook = null;
  return children;
}
```

24.单节点(key 相同,类型相同)

24.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

+function FunctionComponent() {
+  const [number, setNumber] = React.useState(0);
+  return number === 0 ? (
+    setNumber(number + 1)} key="title" id="title">
+     title
+
+  ) : (
+    setNumber(number + 1)} key="title" id="title2">
+     title2
+
+  );
+}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

25.单节点 key 不同,类型相同

- 单节点 key 不同,类型相同，删除老节点，添加新节点

25.1 main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

function FunctionComponent() {
  const [number, setNumber] = React.useState(0);
+  return number === 0 ? (
+    setNumber(number + 1)} key="title1" id="title">
+     title
+
+  ) : (
+    setNumber(number + 1)} key="title2" id="title2">
+     title2
+
  );
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

25.2 ReactFiberFlags.js

src\react-reconciler\src\ReactFiberFlags.js

```
export const NoFlags = 0b00000000000000000000000000;
export const Placement = 0b00000000000000000000000010;
export const Update = 0b00000000000000000000000100;
+export const ChildDeletion = 0b00000000000000000000001000;
export const MutationMask = Placement | Update;
```

25.3 ReactFiber.js

src\react-reconciler\src\ReactFiber.js

```
export function FiberNode(tag, pendingProps, key) {
  this.tag = tag;
  this.key = key;
  this.type = null;
  this.stateNode = null;

  this.return = null;
  this.child = null;
  this.sibling = null;

  this.pendingProps = pendingProps;
  this.memoizedProps = null;
  this.updateQueue = null;
  this.memoizedState = null;

  this.flags = NoFlags;
  this.subtreeFlags = NoFlags;
+ this.deletions = null;
  this.alternate = null;
}
```

25.4 ReactDOMHostConfig.js

src\react-dom-bindings\src\client\ReactDOMHostConfig.js

```
+export function removeChild(parentInstance, child) {
+  parentInstance.removeChild(child);
+}
```

25.5 ReactChildFiber.js

src\react-reconciler\src\ReactChildFiber.js

```
import { REACT_ELEMENT_TYPE } from "shared/ReactSymbols";
import isArray from "shared/isArray";
import { createFiberFromElement, FiberNode, createFiberFromText, createWorkInProgress } from "./ReactFiber";
+import { Placement, ChildDeletion } from "./ReactFiberFlags";
import { HostText } from "./ReactWorkTags";
function createChildReconciler(shouldTrackSideEffects) {
  function useFiber(fiber, pendingProps) {
    const clone = createWorkInProgress(fiber, pendingProps);
```

```
    clone.index = 0;
    clone.sibling = null;
    return clone;
  }
+ function deleteChild(returnFiber, childToDelete) {
+   if (!shouldTrackSideEffects) {
+     return;
+   }
+   const deletions = returnFiber.deletions;
+   if (deletions === null) {
+     returnFiber.deletions = [childToDelete];
+     returnFiber.flags |= ChildDeletion;
+   } else {
+     deletions.push(childToDelete);
+   }
+ }
  function reconcileSingleElement(returnFiber, currentFirstChild, element) {
    const key = element.key;
    let child = currentFirstChild;
    while (child !== null) {
      if (child.key
        const elementType = element.type;
        if (child.type
          const existing = useFiber(child, element.props);
          existing.return = returnFiber;
          return existing;
        }
+     } else {
+       deleteChild(returnFiber, child);
+     }
      child = child.sibling;
    }
    const created = createFiberFromElement(element);
    created.return = returnFiber;
    return created;
  }
  function placeSingleChild(newFiber) {
    if (shouldTrackSideEffects && newFiber.alternate
      newFiber.flags |= Placement;
    }
    return newFiber;
  }
  function reconcileSingleTextNode(returnFiber, currentFirstChild, content) {
    const created = new FiberNode(HostText, { content }, null);
    created.return = returnFiber;
    return created;
  }
  function createChild(returnFiber, newChild) {
    if ((typeof newChild
      const created = createFiberFromText(`${newChild}`);
      created.return = returnFiber;
      return created;
    }

    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          const created = createFiberFromElement(newChild);
          created.return = returnFiber;
          return created;
        }
        default:
          break;
      }
    }
    return null;
  }
  function placeChild(newFiber, newIndex) {
    newFiber.index = newIndex;
    if (shouldTrackSideEffects) newFiber.flags |= Placement;
  }
  function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
    let resultingFirstChild = null;
    let previousNewFiber = null;
    let newIdx = 0;
    for (; newIdx < newChildren.length; newIdx++) {
      const newFiber = createChild(returnFiber, newChildren[newIdx]);
      if (newFiber
        continue;
      }
      placeChild(newFiber, newIdx);
      if (previousNewFiber
        resultingFirstChild = newFiber;
      } else {
        previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
    }
    return resultingFirstChild;
  }
  function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          return placeSingleChild(reconcileSingleElement(returnFiber, currentFirstChild, newChild));
        }
        default:
          break;
      }
      if (isArray(newChild)) {
        return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
      }
    }
    if (typeof newChild
```

```
      return placeSingleChild(reconcileSingleTextNode(returnFiber, currentFirstChild, newChild));
    }
    return null;
  }
  return reconcileChildFibers;
}
export const reconcileChildFibers = createChildReconciler(true);
export const mountChildFibers = createChildReconciler(false);
```

25.6 ReactFiberCommitWork.js

src\react-reconciler\src\ReactFiberCommitWork.js

```
import { HostRoot, HostComponent, HostText, FunctionComponent } from "./ReactWorkTags";
import { MutationMask, Placement, Update } from "./ReactFiberFlags";
import {
  insertBefore,
  appendChild,
  commitUpdate,
+ removeChild,
} from "react-dom-bindings/src/client/ReactDOMHostConfig";
+let hostParent = null;
+function commitDeletionEffects(root, returnFiber, deletedFiber) {
+  let parent = returnFiber;
+  findParent: while (parent !== null) {
+    switch (parent.tag) {
+      case HostComponent: {
+        hostParent = parent.stateNode;
+        break findParent;
+      }
+      case HostRoot: {
+        hostParent = parent.stateNode.containerInfo;
+        break findParent;
+      }
+      default:
+        break;
+    }
+    parent = parent.return;
+  }
+  commitDeletionEffectsOnFiber(root, returnFiber, deletedFiber);
+  hostParent = null;
+}
+function commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor, deletedFiber) {
+  switch (deletedFiber.tag) {
+    case HostComponent:
+    case HostText: {
+      recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor, deletedFiber);
+      if (hostParent !== null) {
+          removeChild(hostParent, deletedFiber.stateNode);
+      }
+      break;
+    }
+    default:
+      break;
+  }
+}
+function recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor, parent) {
+  let child = parent.child;
+  while (child !== null) {
+    commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor, child);
+    child = child.sibling;
+  }
+}
function recursivelyTraverseMutationEffects(root, parentFiber) {
+ const deletions = parentFiber.deletions;
+ if (deletions !== null) {
+   for (let i = 0; i < deletions.length; i++) {
+     const childToDelete = deletions[i];
+     commitDeletionEffects(root, parentFiber, childToDelete);
+   }
+ }
  if (parentFiber.subtreeFlags & MutationMask) {
    let { child } = parentFiber;
    while (child !== null) {
      commitMutationEffectsOnFiber(child, root);
      child = child.sibling;
    }
  }
}
function isHostParent(fiber) {
  return fiber.tag
}
function getHostParentFiber(fiber) {
  let parent = fiber.return;
  while (parent !== null) {
    if (isHostParent(parent)) {
      return parent;
    }
    parent = parent.return;
  }
  return parent;
}
function insertOrAppendPlacementNode(node, before, parent) {
  const { tag } = node;
  const isHost = tag
  if (isHost) {
    const { stateNode } = node;
    if (before) {
      insertBefore(parent, stateNode, before);
    } else {
      appendChild(parent, stateNode);
    }
  } else {
```

```
    const { child } = node;
    if (child !== null) {
      insertOrAppendPlacementNode(child, before, parent);
      let { sibling } = child;
      while (sibling !== null) {
        insertOrAppendPlacementNode(sibling, before, parent);
        sibling = sibling.sibling;
      }
    }
  }
}
function getHostSibling(fiber) {
  let node = fiber;
  siblings: while (true) {
    // 如果我们没有找到任何东西，让我们试试下一个弟弟
    while (node.sibling
      if (node.return
        // 如果我们是根Fiber或者父亲是原生节点，我们就是最后的弟弟
        return null;
      }
      node = node.return;
    }
    // node.sibling.return = node.return
    node = node.sibling;
    while (node.tag !== HostComponent && node.tag !== HostText) {
      // 如果它不是原生节点，并且，我们可能在其中有一个原生节点
      // 试着向下搜索，直到找到为止
      if (node.flags & Placement) {
        // 如果我们没有孩子，可以试试弟弟
        continue siblings;
      } else {
        // node.child.return = node
        node = node.child;
      }
    } // Check if this host node is stable or about to be placed.

    // 检查此原生节点是否稳定可以放置
    if (!(node.flags & Placement)) {
      // 找到它了!

      return node.stateNode;
    }
  }
}
function commitPlacement(finishedWork) {
  const parentFiber = getHostParentFiber(finishedWork);
  switch (parentFiber.tag) {
    case HostComponent: {
      const parent = parentFiber.stateNode;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    case HostRoot: {
      const parent = parentFiber.stateNode.containerInfo;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    default:
      break;
  }
}
function commitReconciliationEffects(finishedWork) {
  const { flags } = finishedWork;
  if (flags & Placement) {
    commitPlacement(finishedWork);
    finishedWork.flags &= ~Placement;
  }
}
export function commitMutationEffectsOnFiber(finishedWork, root) {
  const current = finishedWork.alternate;
  const flags = finishedWork.flags;
  switch (finishedWork.tag) {
    case HostRoot: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    case FunctionComponent: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    case HostComponent: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      if (flags & Update) {
        const instance = finishedWork.stateNode;
        if (instance != null) {
          const newProps = finishedWork.memoizedProps;
          const oldProps = current !== null ? current.memoizedProps : newProps;
          const type = finishedWork.type;
          const updatePayload = finishedWork.updateQueue;
          finishedWork.updateQueue = null;
          if (updatePayload !== null) {
            commitUpdate(instance, updatePayload, type, oldProps, newProps, finishedWork);
          }
        }
      }
      break;
    }
    case HostText: {
      recursivelyTraverseMutationEffects(root, finishedWork);
```

```
      commitReconciliationEffects(finishedWork);
      break;
    }
    default: {
      break;
    }
  }
}
```

26.单节点 key 相同,类型不同

26.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

function FunctionComponent() {
  const [number, setNumber] = React.useState(0);
+  return number === 0 ? (
+     setNumber(number + 1)} key="title1" id="title1">
+      title1
+
+  ) : (
+     setNumber(number + 1)} key="title1" id="title1">
+      title1
+
+  );
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

26.2 ReactChildFiber.js

src\react-reconciler\src\ReactChildFiber.js

```
import { REACT_ELEMENT_TYPE } from "shared/ReactSymbols";
import isArray from "shared/isArray";
import { createFiberFromElement, FiberNode, createFiberFromText, createWorkInProgress } from "./ReactFiber";
import { Placement, ChildDeletion } from "./ReactFiberFlags";
import { HostText } from "./ReactWorkTags";
function createChildReconciler(shouldTrackSideEffects) {
  function useFiber(fiber, pendingProps) {
    const clone = createWorkInProgress(fiber, pendingProps);
    clone.index = 0;
    clone.sibling = null;
    return clone;
  }
  function deleteChild(returnFiber, childToDelete) {
    if (!shouldTrackSideEffects) {
      return;
    }
    const deletions = returnFiber.deletions;
    if (deletions
      returnFiber.deletions = [childToDelete];
      returnFiber.flags |= ChildDeletion;
    } else {
      deletions.push(childToDelete);
    }
  }
+ function deleteRemainingChildren(returnFiber, currentFirstChild) {
+   if (!shouldTrackSideEffects) {
+     return null;
+   }
+   let childToDelete = currentFirstChild;
+   while (childToDelete !== null) {
+     deleteChild(returnFiber, childToDelete);
+     childToDelete = childToDelete.sibling;
+   }
+   return null;
+ }
  function reconcileSingleElement(returnFiber, currentFirstChild, element) {
    const key = element.key;
    let child = currentFirstChild;
    while (child !== null) {
      if (child.key
        const elementType = element.type;
        if (child.type
+         deleteRemainingChildren(returnFiber, child.sibling);
          const existing = useFiber(child, element.props);
          existing.return = returnFiber;
          return existing;
        }
+       deleteRemainingChildren(returnFiber, child);
+       break;
      } else {
        deleteChild(returnFiber, child);
      }
      child = child.sibling;
    }
    const created = createFiberFromElement(element);
    created.return = returnFiber;
    return created;
  }
  function placeSingleChild(newFiber) {
    if (shouldTrackSideEffects && newFiber.alternate
      newFiber.flags |= Placement;
    }
    return newFiber;
  }
```

```
  function reconcileSingleTextNode(returnFiber, currentFirstChild, content) {
    const created = new FiberNode(HostText, { content }, null);
    created.return = returnFiber;
    return created;
  }
  function createChild(returnFiber, newChild) {
    if ((typeof newChild
      const created = createFiberFromText(`${newChild}`);
      created.return = returnFiber;
      return created;
    }

    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          const created = createFiberFromElement(newChild);
          created.return = returnFiber;
          return created;
        }
        default:
          break;
      }
    }
    return null;
  }
  function placeChild(newFiber, newIndex) {
    newFiber.index = newIndex;
    if (shouldTrackSideEffects) newFiber.flags |= Placement;
  }
  function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
    let resultingFirstChild = null;
    let previousNewFiber = null;
    let newIdx = 0;
    for (; newIdx < newChildren.length; newIdx++) {
      const newFiber = createChild(returnFiber, newChildren[newIdx]);
      if (newFiber
        continue;
      }
      placeChild(newFiber, newIdx);
      if (previousNewFiber
        resultingFirstChild = newFiber;
      } else {
        previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
    }
    return resultingFirstChild;
  }
  function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          return placeSingleChild(reconcileSingleElement(returnFiber, currentFirstChild, newChild));
        }
        default:
          break;
      }
      if (isArray(newChild)) {
        return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
      }
    }
    if (typeof newChild
      return placeSingleChild(reconcileSingleTextNode(returnFiber, currentFirstChild, newChild));
    }
    return null;
  }
  return reconcileChildFibers;
}
export const reconcileChildFibers = createChildReconciler(true);
export const mountChildFibers = createChildReconciler(false);
```

27.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

27.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

function FunctionComponent() {
  const [number, setNumber] = React.useState(0);
+ return number === 0 ? (
+    setNumber(number + 1)}>
+     A
+
+       B
+
+     C
+
+ ) : (
+    setNumber(number + 1)}>
+
+       B2
+
+
+ );
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

28.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

29.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

29.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

function FunctionComponent() {
  console.log("FunctionComponent");
  const [number, setNumber] = React.useState(0);
+ return number === 0 ? (
+    setNumber(number + 1)}>
+     A
+
+       B
+
+
+       C
+
+
+ ) : (
+    setNumber(number + 1)}>
+     A2
+
+       B2
+
+
+       C2
+
+
+ );
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

29.2 ReactChildFiber.js

src\react-reconciler\src\ReactChildFiber.js

```
import { REACT_ELEMENT_TYPE } from "shared/ReactSymbols";
import isArray from "shared/isArray";
import { createFiberFromElement, FiberNode, createFiberFromText, createWorkInProgress } from "./ReactFiber";
import { Placement, ChildDeletion } from "./ReactFiberFlags";
import { HostText } from "./ReactWorkTags";
function createChildReconciler(shouldTrackSideEffects) {
  function useFiber(fiber, pendingProps) {
    const clone = createWorkInProgress(fiber, pendingProps);
    clone.index = 0;
```

```
    clone.sibling = null;
    return clone;
  }
  function deleteChild(returnFiber, childToDelete) {
    if (!shouldTrackSideEffects) {
      return;
    }
    const deletions = returnFiber.deletions;
    if (deletions
      returnFiber.deletions = [childToDelete];
      returnFiber.flags |= ChildDeletion;
    } else {
      deletions.push(childToDelete);
    }
  }
  function deleteRemainingChildren(returnFiber, currentFirstChild) {
    if (!shouldTrackSideEffects) {
      return null;
    }
    let childToDelete = currentFirstChild;
    while (childToDelete !== null) {
      deleteChild(returnFiber, childToDelete);
      childToDelete = childToDelete.sibling;
    }
    return null;
  }
  function reconcileSingleElement(returnFiber, currentFirstChild, element) {
    const key = element.key;
    let child = currentFirstChild;
    while (child !== null) {
      if (child.key
        const elementType = element.type;
        if (child.type
          deleteRemainingChildren(returnFiber, child.sibling);
          const existing = useFiber(child, element.props);
          existing.return = returnFiber;
          return existing;
        }
        deleteRemainingChildren(returnFiber, child);
        break;
      } else {
        deleteChild(returnFiber, child);
      }
      child = child.sibling;
    }
    const created = createFiberFromElement(element);
    created.return = returnFiber;
    return created;
  }
  function placeSingleChild(newFiber) {
    if (shouldTrackSideEffects && newFiber.alternate
      newFiber.flags |= Placement;
    }
    return newFiber;
  }
  function reconcileSingleTextNode(returnFiber, currentFirstChild, content) {
    const created = new FiberNode(HostText, { content }, null);
    created.return = returnFiber;
    return created;
  }
  function createChild(returnFiber, newChild) {
    if ((typeof newChild
      const created = createFiberFromText(`${newChild}`);
      created.return = returnFiber;
      return created;
    }

    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          const created = createFiberFromElement(newChild);
          created.return = returnFiber;
          return created;
        }
        default:
          break;
      }
    }
    return null;
  }
  function placeChild(newFiber, newIndex) {
+   newFiber.index = newIndex;
+   if (!shouldTrackSideEffects) {
+     return;
+   }
+   const current = newFiber.alternate;
+   if (current !== null) {
+     return;
+   } else {
+     newFiber.flags |= Placement;
+   }
  }
+ function updateElement(returnFiber, current, element) {
+   const elementType = element.type;
+   if (current !== null) {
+     if (current.type === elementType) {
+       const existing = useFiber(current, element.props);
+       existing.return = returnFiber;
+       return existing;
+     }
+   }
+   const created = createFiberFromElement(element);
+   created.return = returnFiber;
+   return created;
```

```
+ }
+ function updateSlot(returnFiber, oldFiber, newChild) {
+   const key = oldFiber !== null ? oldFiber.key : null;
+   if (typeof newChild === "object" && newChild !== null) {
+     switch (newChild.$typeof) {
+       case REACT_ELEMENT_TYPE: {
+         if (newChild.key === key) {
+           return updateElement(returnFiber, oldFiber, newChild);
+         }
+       }
+       default:
+         return null;
+     }
+     return null
+   }
+ }

  function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
    let resultingFirstChild = null;
    let previousNewFiber = null;
    let newIdx = 0;
+   let oldFiber = currentFirstChild;
+   let nextOldFiber = null;
+   for (; oldFiber !== null && newIdx < newChildren.length; newIdx++) {
+     nextOldFiber = oldFiber.sibling;
+     const newFiber = updateSlot(returnFiber, oldFiber, newChildren[newIdx]);
+     if (newFiber === null) {
+       break;
+     }
+     if (shouldTrackSideEffects) {
+       if (oldFiber && newFiber.alternate === null) {
+         deleteChild(returnFiber, oldFiber);
+       }
+     }
+     placeChild(newFiber, newIdx);
+     if (previousNewFiber === null) {
+       resultingFirstChild = newFiber;
+     } else {
+       previousNewFiber.sibling = newFiber;
+     }
+     previousNewFiber = newFiber;
+     oldFiber = nextOldFiber;
+   }
+   if (newIdx === newChildren.length) {
+     deleteRemainingChildren(returnFiber, oldFiber);
+     return resultingFirstChild;
+   }
+   if (oldFiber === null) {
+     for (; newIdx < newChildren.length; newIdx++) {
+       const newFiber = createChild(returnFiber, newChildren[newIdx]);
+       if (newFiber === null) {
+         continue;
+       }
+       placeChild(newFiber, newIdx);
+       if (previousNewFiber === null) {
+         resultingFirstChild = newFiber;
+       } else {
+         previousNewFiber.sibling = newFiber;
+       }
+       previousNewFiber = newFiber;
+     }
+   }
    return resultingFirstChild;
  }
  function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          return placeSingleChild(reconcileSingleElement(returnFiber, currentFirstChild, newChild));
        }
        default:
          break;
      }
      if (isArray(newChild)) {
        return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
      }
    }
    if (typeof newChild
      return placeSingleChild(reconcileSingleTextNode(returnFiber, currentFirstChild, newChild));
    }
    return null;
  }
  return reconcileChildFibers;
}
export const reconcileChildFibers = createChildReconciler(true);
export const mountChildFibers = createChildReconciler(false);
```

30.多个节点的类型和 key 全部相同，有新增元素

30.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

function FunctionComponent() {
  console.log("FunctionComponent");
  const [number, setNumber] = React.useState(0);
+ return number === 0 ? (
+    setNumber(number + 1)}>
+     A
+
+       B
+
+     C
+
+ ) : (
+    setNumber(number + 1)}>
+     A
+
+       B2
+
+     C2
+     D
+
+ );
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

31.多个节点的类型和 key 全部相同，有删除老元素

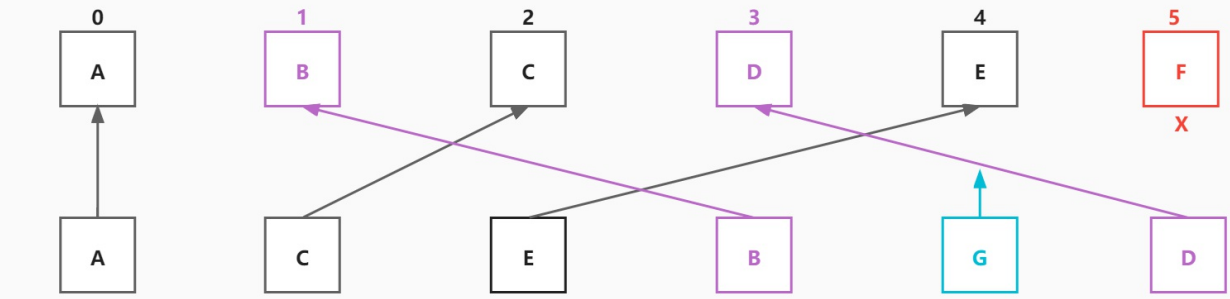
31.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

function FunctionComponent() {
  console.log("FunctionComponent");
  const [number, setNumber] = React.useState(0);
+ return number === 0 ? (
+    setNumber(number + 1)}>
+     A
+
+       B
+
+     C
+
+ ) : (
+    setNumber(number + 1)}>
+     A
+
+       B2
+
+
+ );
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

32.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



lastPlacedIndex=0 lastPlacedIndex=2 lastPlacedIndex=4

- 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,默认为0
- 继续循环剩下的虚拟DOM节点，从C开始
- 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节点
- 如果是复用老的fiber,则判断老fiber的索引是否小于lastPlacedIndex
- 如果小于lastPlacedIndex则需要移动老fiber，lastPlacedIndex不变
- 如果大于lastPlacedIndex则不需要移动老fiber,更新lastPlacedIndex为老fiber的index
- 虚拟DOM循环结束后把map中所有的剩下的fiber全部标记为删除
- (删除#li#F)=>(移动#li#B)=>(添加#li#G)=>(移动#li#D)=>null

```
FunctionComponent
插入 FunctionComponent FunctionComponent ▶ {}
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
FunctionComponent
自己有更新和子元素有删除 HostComponent ul ▶ {children: Array(6), onClick: ƒ}
HostComponent li ▶ {children: 'F'}
更新 HostComponent li ▶ {children: 'A2'}
更新 HostComponent li ▶ {children: 'C2'}
更新 HostComponent li ▶ {children: 'E2'}
移动并更新 HostComponent li ▶ {id: 'b2', children: 'B2'}
插入 HostComponent li ▶ {children: 'G'}
移动并更新 HostComponent li ▶ {children: 'D2'}
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

32.1 src\main.jsx #

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

function FunctionComponent() {
  console.log("FunctionComponent");
  const [number, setNumber] = React.useState(0);
  return number
     setNumber(number + 1)}>
+     A
+
+       B
+
+     C
+     D
+     E
+     F

  ) : (
     setNumber(number + 1)}>
+     A2
+     C2
+     E2
+
+       B2
+
+     G
+     D2

  );
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

32.2 ReactFiber.js

src\react-reconciler\src\ReactFiber.js

```
export function FiberNode(tag, pendingProps, key) {
  this.tag = tag;
  this.key = key;
  this.type = null;
  this.stateNode = null;

  this.return = null;
  this.child = null;
  this.sibling = null;

  this.pendingProps = pendingProps;
  this.memoizedProps = null;
  this.updateQueue = null;
  this.memoizedState = null;

  this.flags = NoFlags;
  this.subtreeFlags = NoFlags;
  this.deletions = null;
  this.alternate = null;

+ this.index = 0;
}
```

32.3 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import { scheduleCallback } from "scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
import { MutationMask, NoFlags, Placement, Update,ChildDeletion } from "./ReactFiberFlags";
import { commitMutationEffectsOnFiber } from "./ReactFiberCommitWork";
import { finishQueueingConcurrentUpdates } from "./ReactFiberConcurrentUpdates";
import { FunctionComponent, IndeterminateComponent, HostRoot, HostComponent, HostText } from "./ReactWorkTags";

let workInProgress = null;
export function scheduleUpdateOnFiber(root) {
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  scheduleCallback(performConcurrentWorkOnRoot.bind(null, root));
}
function performConcurrentWorkOnRoot(root) {
  renderRootSync(root);
  const finishedWork = root.current.alternate;
+ printFiber(finishedWork);
+ console.log(`~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`);
  root.finishedWork = finishedWork;
  commitRoot(root);
}
function commitRoot(root) {
  const { finishedWork } = root;
  const subtreeHasEffects = (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
    commitMutationEffectsOnFiber(finishedWork, root);
  }
  root.current = finishedWork;
}
function prepareFreshStack(root) {
  workInProgress = createWorkInProgress(root.current, null);
  finishQueueingConcurrentUpdates();
}
function renderRootSync(root) {
```

```
  prepareFreshStack(root);
  workLoopSync();
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
}

+function printFiber(fiber) {
+  /*
+  fiber.flags &= ~Forked;
+  fiber.flags &= ~PlacementDEV;
+  fiber.flags &= ~Snapshot;
+  fiber.flags &= ~PerformedWork;
+  */
+  if (fiber.flags !== 0) {
+    console.log(
+      getFlags(fiber.flags),
+      getTag(fiber.tag),
+      typeof fiber.type === "function" ? fiber.type.name : fiber.type,
+      fiber.memoizedProps
+    );
+    if (fiber.deletions) {
+      for (let i = 0; i < fiber.deletions.length; i++) {
+        const childToDelete = fiber.deletions[i];
+        console.log(getTag(childToDelete.tag), childToDelete.type, childToDelete.+memoizedProps);
+      }
+    }
+  }
+  let child = fiber.child;
+  while (child) {
+    printFiber(child);
+    child = child.sibling;
+  }
+}
+function getTag(tag) {
+  switch (tag) {
+    case FunctionComponent:
+      return `FunctionComponent`;
+    case HostRoot:
+      return `HostRoot`;
+    case HostComponent:
+      return `HostComponent`;
+    case HostText:
+      return HostText;
+    default:
+      return tag;
+  }
+}
+function getFlags(flags) {
+  if (flags === (Update | Placement | ChildDeletion)) {
+    return `自己移动和子元素有删除`;
+  }
+  if (flags === (ChildDeletion | Update)) {
+    return `自己有更新和子元素有删除`;
+  }
+  if (flags === ChildDeletion) {
+    return `子元素有删除`;
+  }
+  if (flags === (Placement | Update)) {
+    return `移动并更新`;
+  }
+  if (flags === Placement) {
+    return `插入`;
+  }
+  if (flags === Update) {
+    return `更新`;
+  }
+  return flags;
+}
```

32.4 ReactChildFiber.js

src\react-reconciler\src\ReactChildFiber.js

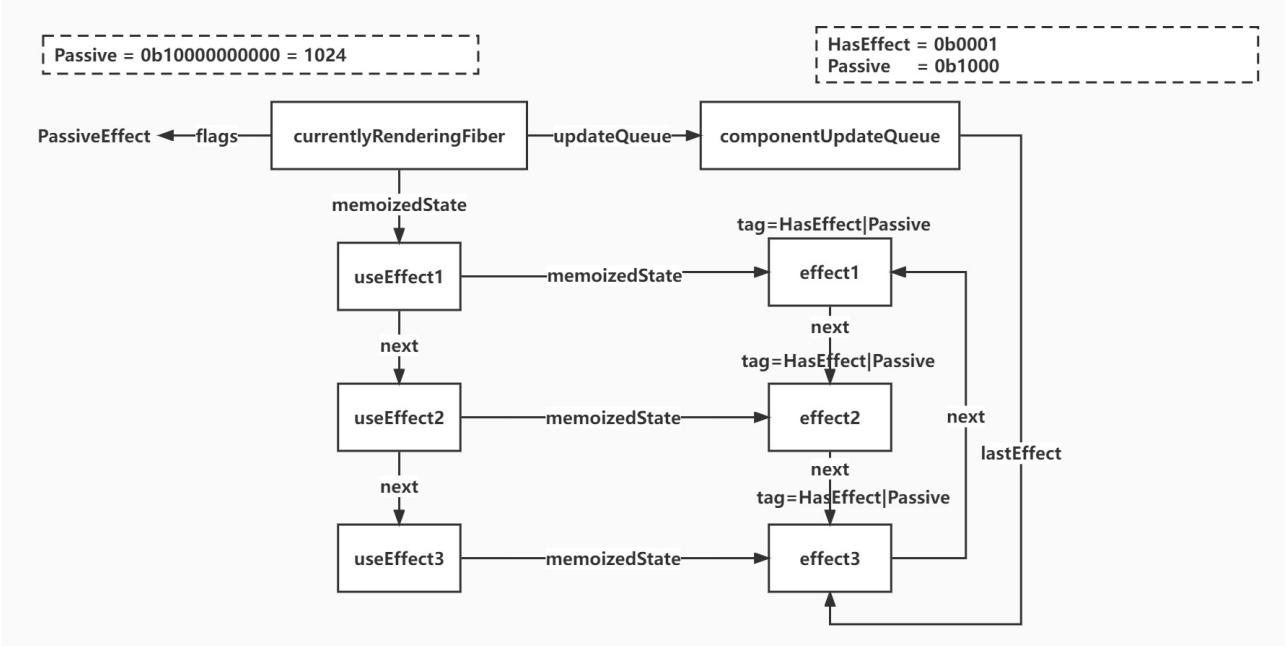
```
import { REACT_ELEMENT_TYPE } from "shared/ReactSymbols";
import isArray from "shared/isArray";
import { createFiberFromElement, FiberNode, createFiberFromText, createWorkInProgress } from "./ReactFiber";
import { Placement, ChildDeletion } from "./ReactFiberFlags";
import { HostText } from "./ReactWorkTags";
function createChildReconciler(shouldTrackSideEffects) {
  function useFiber(fiber, pendingProps) {
    const clone = createWorkInProgress(fiber, pendingProps);
    clone.index = 0;
    clone.sibling = null;
    return clone;
  }
  function deleteChild(returnFiber, childToDelete) {
    if (!shouldTrackSideEffects) {
      return;
    }
    const deletions = returnFiber.deletions;
    if (deletions
      returnFiber.deletions = [childToDelete];
      returnFiber.flags |= ChildDeletion;
    } else {
      deletions.push(childToDelete);
    }
  }
  function deleteRemainingChildren(returnFiber, currentFirstChild) {
    if (!shouldTrackSideEffects) {
      return null;
    }
    let childToDelete = currentFirstChild;
    while (childToDelete !== null) {
      deleteChild(returnFiber, childToDelete);
      childToDelete = childToDelete.sibling;
    }
    return null;
  }
  function reconcileSingleElement(returnFiber, currentFirstChild, element) {
    const key = element.key;
    let child = currentFirstChild;
    while (child !== null) {
      if (child.key
        const elementType = element.type;
        if (child.type
          deleteRemainingChildren(returnFiber, child.sibling);
          const existing = useFiber(child, element.props);
          existing.return = returnFiber;
          return existing;
        }
        deleteRemainingChildren(returnFiber, child);
        break;
      } else {
        deleteChild(returnFiber, child);
      }
      child = child.sibling;
    }
    const created = createFiberFromElement(element);
    created.return = returnFiber;
    return created;
  }
  function placeSingleChild(newFiber) {
    if (shouldTrackSideEffects && newFiber.alternate
      newFiber.flags |= Placement;
    }
    return newFiber;
  }
  function reconcileSingleTextNode(returnFiber, currentFirstChild, content) {
    const created = new FiberNode(HostText, { content }, null);
    created.return = returnFiber;
    return created;
  }
  function createChild(returnFiber, newChild) {
    if ((typeof newChild
      const created = createFiberFromText(`${newChild}`);
      created.return = returnFiber;
      return created;
    }

    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          const created = createFiberFromElement(newChild);
          created.return = returnFiber;
          return created;
        }
        default:
          break;
      }
    }
    return null;
  }
+ function placeChild(newFiber, lastPlacedIndex, newIndex) {
+   newFiber.index = newIndex;
+   if (!shouldTrackSideEffects) {
+     return lastPlacedIndex;
+   }
+   const current = newFiber.alternate;
+   if (current !== null) {
+     const oldIndex = current.index;
+     if (oldIndex < lastPlacedIndex) {
+       newFiber.flags |= Placement;
+       return lastPlacedIndex;
+     } else {
+       return oldIndex;
+     }
```

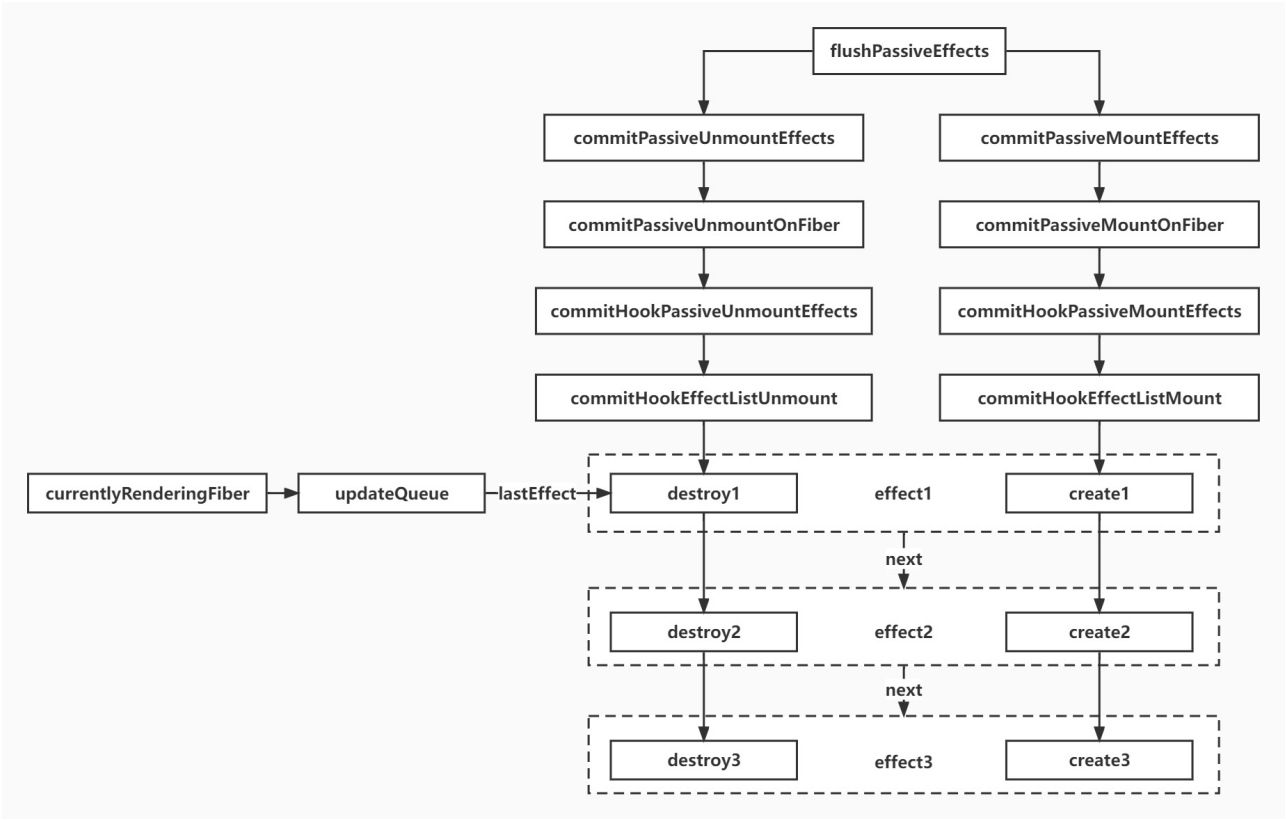
```
+   } else {
+     newFiber.flags |= Placement;
+     return lastPlacedIndex;
+   }
+ }
  function updateElement(returnFiber, current, element) {
    const elementType = element.type;
    if (current !== null) {
      if (current.type
        const existing = useFiber(current, element.props);
        existing.return = returnFiber;
        return existing;
      }
    }
    const created = createFiberFromElement(element);
    created.return = returnFiber;
    return created;
  }
  function updateSlot(returnFiber, oldFiber, newChild) {
    const key = oldFiber !== null ? oldFiber.key : null;
    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          if (newChild.key
            return updateElement(returnFiber, oldFiber, newChild);
          } else {
            return null;
          }
        }
        default:
          return null;
      }
    }
  }
+ function mapRemainingChildren(returnFiber, currentFirstChild) {
+   const existingChildren = new Map();
+   let existingChild = currentFirstChild;
+   while (existingChild !== null) {
+     if (existingChild.key !== null) {
+       existingChildren.set(existingChild.key, existingChild);
+     } else {
+       existingChildren.set(existingChild.index, existingChild);
+     }
+     existingChild = existingChild.sibling;
+   }
+   return existingChildren;
+ }
+ function updateTextNode(returnFiber, current, textContent) {
+   if (current === null || current.tag !== HostText) {
+     const created = createFiberFromText(textContent);
+     created.return = returnFiber;
+     return created;
+   } else {
+     const existing = useFiber(current, textContent);
+     existing.return = returnFiber;
+     return existing;
+   }
+ }
+ function updateFromMap(existingChildren, returnFiber, newIdx, newChild) {
+   if ((typeof newChild === "string" && newChild !== "") || typeof newChild === "number") {
+     const matchedFiber = existingChildren.get(newIdx) || null;
+     return updateTextNode(returnFiber, matchedFiber, "" + newChild);
+   }
+   if (typeof newChild === "object" && newChild !== null) {
+     switch (newChild.$typeof) {
+       case REACT_ELEMENT_TYPE: {
+         const matchedFiber = existingChildren.get(newChild.key === null ? newIdx : newChild.key) || null;
+         return updateElement(returnFiber, matchedFiber, newChild);
+       }
+     }
+   }
+   return null;
+ }
  function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
    let resultingFirstChild = null;
    let previousNewFiber = null;
    let newIdx = 0;
    let oldFiber = currentFirstChild;
    let nextOldFiber = null;
+   let lastPlacedIndex = 0;
    for (; oldFiber !== null && newIdx < newChildren.length; newIdx++) {
      nextOldFiber = oldFiber.sibling;
      const newFiber = updateSlot(returnFiber, oldFiber, newChildren[newIdx]);
      if (newFiber
        break;
      }
      if (shouldTrackSideEffects) {
        if (oldFiber && newFiber.alternate
          deleteChild(returnFiber, oldFiber);
        }
      }
+     lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
      if (previousNewFiber
        resultingFirstChild = newFiber;
      } else {
        previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
      oldFiber = nextOldFiber;
    }
    if (newIdx
      deleteRemainingChildren(returnFiber, oldFiber);
      return resultingFirstChild;
    }
```

```
    if (oldFiber
      for (; newIdx < newChildren.length; newIdx++) {
        const newFiber = createChild(returnFiber, newChildren[newIdx]);
        if (newFiber
          continue;
        }
+       lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
        if (previousNewFiber
          resultingFirstChild = newFiber;
        } else {
          previousNewFiber.sibling = newFiber;
        }
        previousNewFiber = newFiber;
      }
    }
+   const existingChildren = mapRemainingChildren(returnFiber, oldFiber);
+   for (; newIdx < newChildren.length; newIdx++) {
+     const newFiber = updateFromMap(existingChildren, returnFiber, newIdx, newChildren[newIdx]);
+     if (newFiber !== null) {
+       if (shouldTrackSideEffects) {
+         if (newFiber.alternate !== null) {
+           existingChildren.delete(newFiber.key === null ? newIdx : newFiber.key);
+         }
+       }
+       lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
+       if (previousNewFiber === null) {
+         resultingFirstChild = newFiber;
+       } else {
+         previousNewFiber.sibling = newFiber;
+       }
+       previousNewFiber = newFiber;
+     }
+   }
+   if (shouldTrackSideEffects) {
+     existingChildren.forEach((child) => deleteChild(returnFiber, child));
+   }
    return resultingFirstChild;
  }
  function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          return placeSingleChild(reconcileSingleElement(returnFiber, currentFirstChild, newChild));
        }
        default:
          break;
      }
      if (isArray(newChild)) {
        return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
      }
    }
    if (typeof newChild
      return placeSingleChild(reconcileSingleTextNode(returnFiber, currentFirstChild, newChild));
    }
    return null;
  }
  return reconcileChildFibers;
}
export const reconcileChildFibers = createChildReconciler(true);
export const mountChildFibers = createChildReconciler(false);
```

33.useEffect

- 在函数组件主体内（这里指在 React 渲染阶段）改变 DOM、添加订阅、设置定时器、记录日志以及执行其他包含副作用的操作都是不被允许的，因为这可能会产生莫名其妙的 bug 并破坏 UI 的一致性
- 使用 useEffect 完成副作用操作。赋值给 useEffect 的函数会在组件渲染到屏幕之后执行。你可以把 effect 看作从 React 的纯函数式世界通往命令式世界的逃生通道
- useEffect 就是一个 Effect Hook，给函数组件增加了操作副作用的能力。它跟 class 组件中的 componentDidMount、componentDidUpdate 和 componentWillUnmount 具有相同的用途，只不过被合并成了一个 API
- 该 Hook 接收一个包含命令式、且可能有副作用代码的函数





33.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

function Counter() {
+ const [number, setNumber] = React.useState(0);
+ React.useEffect(() => {
+   console.log("useEffect1");
+   return () => {
+     console.log("destroy useEffect1");
+   };
+ });
+ React.useEffect(() => {
+   console.log("useEffect2");
+   return () => {
+     console.log("destroy useEffect2");
+   };
+ });
+ React.useEffect(() => {
+   console.log("useEffect3");
+   return () => {
+     console.log("destroy useEffect3");
+   };
+ });
+ return (
+
+     onClick={() => {
+       setNumber(number + 1);
+     }}
+   >
+     {number}
+
+ );
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

33.2 react\index.js

src\react\index.js

```
+export { __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED, useReducer, useState, useEffect } from "./src/React";
```

33.3 React.js

src\react\src\React.js

```
+import { useReducer, useState, useEffect } from "./ReactHooks";
import ReactSharedInternals from "./ReactSharedInternals";

+export { useReducer, useState, useEffect, ReactSharedInternals as __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED };
```

33.4 ReactHooks.js

src\react\src\ReactHooks.js

```
import ReactCurrentDispatcher from "./ReactCurrentDispatcher";

function resolveDispatcher() {
  const dispatcher = ReactCurrentDispatcher.current;
  return dispatcher;
}

export function useReducer(reducer, initialArg, init) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useReducer(reducer, initialArg, init);
}

export function useState(initialState) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useState(initialState);
}

+export function useEffect(create,deps) {
+  const dispatcher = resolveDispatcher();
+  return dispatcher.useEffect(create,deps);
+}
```

src\react-reconciler\src\ReactHookEffectTags.js

```
export const HasEffect = 0b0001;
export const Passive = 0b1000;
```

33.6 ReactFiberFlags.js

src\react-reconciler\src\ReactFiberFlags.js

```
export const NoFlags = 0b00000000000000000000000000;
export const Placement = 0b00000000000000000000000010;
export const Update = 0b00000000000000000000000100;
export const ChildDeletion = 0b00000000000000000000001000;
export const MutationMask = Placement | Update;
+export const Passive = 0b00000000000000010000000000;
```

33.7 ReactFiberHooks.js

src\react-reconciler\src\ReactFiberHooks.js

```
import ReactSharedInternals from "shared/ReactSharedInternals";
import { enqueueConcurrentHookUpdate } from "./ReactFiberConcurrentUpdates";
import { scheduleUpdateOnFiber } from "./ReactFiberWorkLoop";
import is from "shared/objectIs";
+import { Passive as PassiveEffect } from "./ReactFiberFlags";
+import { HasEffect as HookHasEffect, Passive as HookPassive } from "./ReactHookEffectTags";

const { ReactCurrentDispatcher } = ReactSharedInternals;
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;

const HooksDispatcherOnMountInDEV = {
  useReducer: mountReducer,
  useState: mountState,
+ useEffect: mountEffect,
};
const HooksDispatcherOnUpdateInDEV = {
  useReducer: updateReducer,
  useState: updateState,
+ useEffect: updateEffect,
};
+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;
+  if (currentHook !== null) {
+    const prevEffect = currentHook.memoizedState;
+    destroy = prevEffect.destroy;
+    if (nextDeps !== null) {
+      const prevDeps = prevEffect.deps;
+      if (areHookInputsEqual(nextDeps, prevDeps)) {
+        hook.memoizedState = 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 (is(nextDeps[i], prevDeps[i])) {
+      continue;
+    }
+    return false;
+  }
+
+  return true;
+}
+function mountEffect(create, deps) {
+  return mountEffectImpl(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,
+  };
+}
function basicStateReducer(state, action) {
  return typeof action
}
function mountReducer(reducer, initialArg) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialArg;
  const queue = {
    pending: null,
    dispatch: null,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchReducerAction.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function updateReducer(reducer) {
  const hook = updateWorkInProgressHook();
  const queue = hook.queue;
  queue.lastRenderedReducer = reducer;
  const current = currentHook;
  const pendingQueue = queue.pending;
  let baseQueue = null;
  let newState = current.memoizedState;
  if (pendingQueue !== null) {
    baseQueue = pendingQueue;
    queue.pending = null;
  }
  if (baseQueue !== null) {
    const first = baseQueue.next;
    let update = first;
    do {
      if (update.hasEagerState) {
        newState = update.eagerState;
      } else {
        const action = update.action;
        newState = reducer(newState, action);
      }
      update = update.next;
    } while (update !== null && update !== first);
  }
  hook.memoizedState = newState;
  queue.lastRenderedState = newState;
  const dispatch = queue.dispatch;
  return [hook.memoizedState, dispatch];
}
function mountState(initialState) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialState;
  const queue = {
    pending: null,
    dispatch: null,
    lastRenderedReducer: basicStateReducer,
    lastRenderedState: initialState,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchSetState.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function dispatchSetState(fiber, queue, action) {
  const update = {
    action,
    hasEagerState: false,
    eagerState: null,
    next: null,
  };
  const lastRenderedReducer = queue.lastRenderedReducer;
  const currentState = queue.lastRenderedState;
  const eagerState = lastRenderedReducer(currentState, action);
  update.hasEagerState = true;
  update.eagerState = eagerState;
  if (is(eagerState, currentState)) {
    return;
```

```
  }
  const root = enqueueConcurrentHookUpdate(fiber, queue, update);
  scheduleUpdateOnFiber(root, fiber);
}
function updateState(initialState) {
  return updateReducer(basicStateReducer, initialState);
}
function mountWorkInProgressHook() {
  const hook = {
    memoizedState: null,
    queue: null,
    next: null,
  };
  if (workInProgressHook
    currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
  } else {
    workInProgressHook = workInProgressHook.next = hook;
  }
  return workInProgressHook;
}
function dispatchReducerAction(fiber, queue, action) {
  const update = {
    action,
    next: null,
  };
  const root = enqueueConcurrentHookUpdate(fiber, queue, update);
  scheduleUpdateOnFiber(root, fiber);
}

function updateWorkInProgressHook() {
  let nextCurrentHook;
  if (currentHook
    const current = currentlyRenderingFiber.alternate;
    if (current !== null) {
      nextCurrentHook = current.memoizedState;
    } else {
      nextCurrentHook = null;
    }
  } else {
    nextCurrentHook = currentHook.next;
  }

  let nextWorkInProgressHook;
  if (workInProgressHook
    nextWorkInProgressHook = currentlyRenderingFiber.memoizedState;
  } else {
    nextWorkInProgressHook = workInProgressHook.next;
  }

  if (nextWorkInProgressHook !== null) {
    workInProgressHook = nextWorkInProgressHook;
    nextWorkInProgressHook = workInProgressHook.next;
    currentHook = nextCurrentHook;
  } else {
    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 renderWithHooks(current, workInProgress, Component, props) {
  currentlyRenderingFiber = workInProgress;
+ workInProgress.updateQueue = null;
+ workInProgress.memoizedState = null;
  if (current !== null && current.memoizedState !== null) {
    ReactCurrentDispatcher.current = HooksDispatcherOnUpdateInDEV;
  } else {
    ReactCurrentDispatcher.current = HooksDispatcherOnMountInDEV;
  }
  const children = Component(props);
  currentlyRenderingFiber = null;
  workInProgressHook = null;
  currentHook = null;
  return children;
}
```

33.8 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import { scheduleCallback } from "scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
+import { MutationMask, NoFlags, Placement, Update, ChildDeletion, Passive } from "./ReactFiberFlags";
+import { commitMutationEffects, commitPassiveUnmountEffects, commitPassiveMountEffects } from "./ReactFiberCommitWork";
import { finishQueueingConcurrentUpdates } from "./ReactFiberConcurrentUpdates";
import { FunctionComponent, IndeterminateComponent, HostRoot, HostComponent, HostText } from "./ReactWorkTags";

let workInProgress = null;
+let rootDoesHavePassiveEffects = false;
+let rootWithPendingPassiveEffects = null;
export function scheduleUpdateOnFiber(root) {
  ensureRootIsScheduled(root);
```

```
}
function ensureRootIsScheduled(root) {
  scheduleCallback(performConcurrentWorkOnRoot.bind(null, root));
}
function performConcurrentWorkOnRoot(root) {
  renderRootSync(root);
  const finishedWork = root.current.alternate;
  printFiber(finishedWork);
  console.log(`~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`);
  root.finishedWork = finishedWork;
  commitRoot(root);
}
+export function flushPassiveEffects() {
+  if (rootWithPendingPassiveEffects !== null) {
+    const root = rootWithPendingPassiveEffects;
+    commitPassiveUnmountEffects(root.current);
+    commitPassiveMountEffects(root, root.current);
+  }
+}
function commitRoot(root) {
  const { finishedWork } = root;
+  if ((finishedWork.subtreeFlags & Passive) !== NoFlags || (finishedWork.flags & Passive) !== NoFlags) {
+   if (!rootDoesHavePassiveEffects) {
+     rootDoesHavePassiveEffects = true;
+     scheduleCallback(flushPassiveEffects);
+   }
+ }
  const subtreeHasEffects = (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
+   commitMutationEffects(finishedWork, root);
+   root.current = finishedWork;
+   if (rootDoesHavePassiveEffects) {
+     rootDoesHavePassiveEffects = false;
+     rootWithPendingPassiveEffects = root;
+   }
  }
  root.current = finishedWork;
}
function prepareFreshStack(root) {
  workInProgress = createWorkInProgress(root.current, null);
  finishQueueingConcurrentUpdates();
}
function renderRootSync(root) {
  prepareFreshStack(root);
  workLoopSync();
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
}

function printFiber(fiber) {
  /*
  fiber.flags &= ~Forked;
  fiber.flags &= ~PlacementDEV;
  fiber.flags &= ~Snapshot;
  fiber.flags &= ~PerformedWork;
  */
  if (fiber.flags !== 0) {
    console.log(
      getFlags(fiber.flags),
      getTag(fiber.tag),
      typeof fiber.type
      fiber.memoizedProps
    );
    if (fiber.deletions) {
      for (let i = 0; i < fiber.deletions.length; i++) {
        const childToDelete = fiber.deletions[i];
        console.log(getTag(childToDelete.tag), childToDelete.type, childToDelete.memoizedProps);
      }
    }
  }
  let child = fiber.child;
  while (child) {
    printFiber(child);
```

```
    child = child.sibling;
  }
}
function getTag(tag) {
  switch (tag) {
    case FunctionComponent:
      return `FunctionComponent`;
    case HostRoot:
      return `HostRoot`;
    case HostComponent:
      return `HostComponent`;
    case HostText:
      return HostText;
    default:
      return tag;
  }
}
function getFlags(flags) {
  if (flags
    return `自己移动和子元素有删除`;
  }
  if (flags
    return `自己有更新和子元素有删除`;
  }
  if (flags
    return `子元素有删除`;
  }
  if (flags
    return `移动并更新`;
  }
  if (flags
    return `插入`;
  }
  if (flags
    return `更新`;
  }
  return flags;
}
```

33.9 ReactFiberCommitWork.js

src\react-reconciler\src\ReactFiberCommitWork.js

```
import { HostRoot, HostComponent, HostText, FunctionComponent } from "./ReactWorkTags";
+import { MutationMask, Placement, Update, Passive } from "./ReactFiberFlags";
import { insertBefore, appendChild, commitUpdate, removeChild } from "react-dom-bindings/src/client/ReactDOMHostConfig";
+import { HasEffect as HookHasEffect, Passive as HookPassive } from "./ReactHookEffectTags";

+export function commitMutationEffects(finishedWork, root) {
+  commitMutationEffectsOnFiber(finishedWork, root);
+}
+export function commitPassiveUnmountEffects(finishedWork) {
+  commitPassiveUnmountOnFiber(finishedWork);
+}
+function commitPassiveUnmountOnFiber(finishedWork) {
+  switch (finishedWork.tag) {
+    case FunctionComponent: {
+      recursivelyTraversePassiveUnmountEffects(finishedWork);
+      if (finishedWork.flags & Passive) {
+        commitHookPassiveUnmountEffects(finishedWork, finishedWork.return, HookPassive | HookHasEffect);
+      }
+      break;
+    }
+    default: {
+      recursivelyTraversePassiveUnmountEffects(finishedWork);
+      break;
+    }
+  }
+}
+function recursivelyTraversePassiveUnmountEffects(parentFiber) {
+  if (parentFiber.subtreeFlags & Passive) {
+    let child = parentFiber.child;
+    while (child !== null) {
+      commitPassiveUnmountOnFiber(child);
+      child = child.sibling;
+    }
+  }
+}
+function commitHookPassiveUnmountEffects(finishedWork, nearestMountedAncestor, +hookFlags) {
+  commitHookEffectListUnmount(hookFlags, finishedWork, nearestMountedAncestor);
+}
+
+function commitHookEffectListUnmount(flags, 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 & flags) === flags) {
+        const destroy = effect.destroy;
+        effect.destroy = undefined;
+        if (destroy !== undefined) {
+          destroy();
+        }
+      }
+      effect = effect.next;
+    } while (effect !== firstEffect);
+  }
+}

+export function commitPassiveMountEffects(root, finishedWork) {
+  commitPassiveMountOnFiber(root, finishedWork);
+}
```

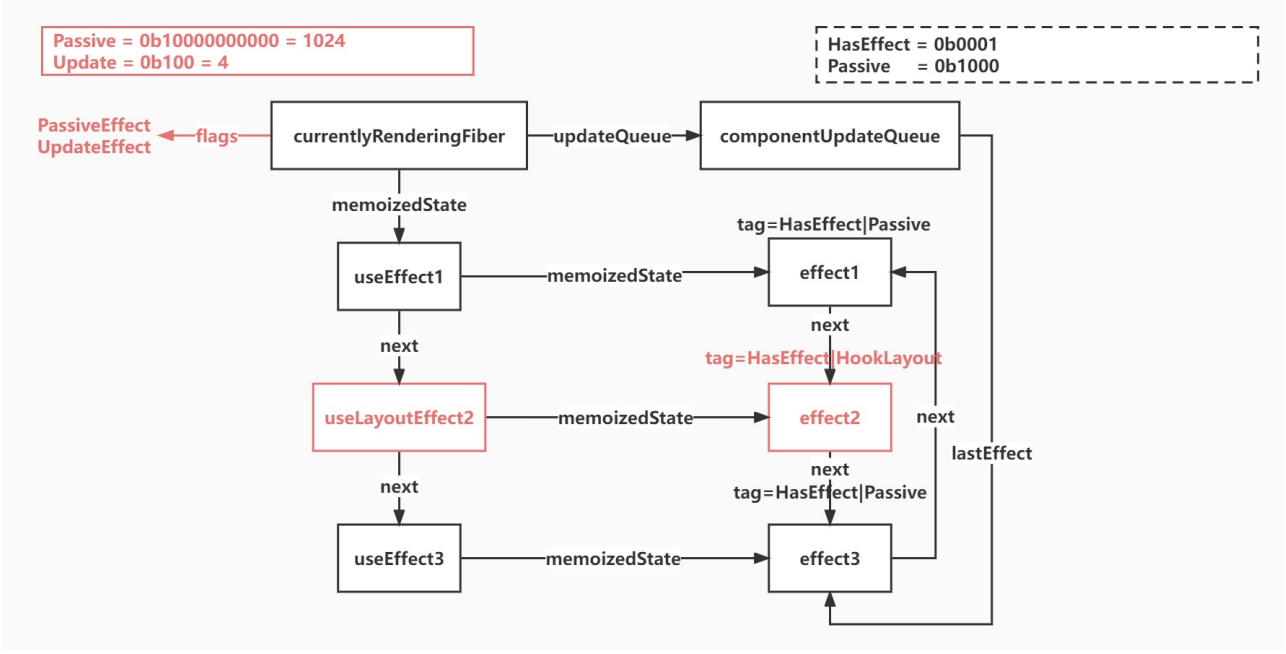
```
+function commitPassiveMountOnFiber(finishedRoot, finishedWork) {
+  const flags = finishedWork.flags;
+  switch (finishedWork.tag) {
+    case FunctionComponent: {
+      recursivelyTraversePassiveMountEffects(finishedRoot, finishedWork);
+      if (flags & Passive) {
+        commitHookPassiveMountEffects(finishedWork, HookPassive | HookHasEffect);
+      }
+      break;
+    }
+    case HostRoot: {
+      recursivelyTraversePassiveMountEffects(finishedRoot, finishedWork);
+      break;
+    }
+    default:
+      break;
+  }
+}
+function commitHookPassiveMountEffects(finishedWork, hookFlags) {
+  commitHookEffectListMount(hookFlags, finishedWork);
+}
+function commitHookEffectListMount(flags, 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 & flags) === flags) {
+        const create = effect.create;
+        effect.destroy = create();
+      }
+      effect = effect.next;
+    } while (effect !== firstEffect);
+  }
+}
+function recursivelyTraversePassiveMountEffects(root, parentFiber) {
+  if (parentFiber.subtreeFlags & Passive) {
+    let child = parentFiber.child;
+    while (child !== null) {
+      commitPassiveMountOnFiber(root, child);
+      child = child.sibling;
+    }
+  }
+}
let hostParent = null;
function commitDeletionEffects(root, returnFiber, deletedFiber) {
  let parent = returnFiber;
  findParent: while (parent !== null) {
    switch (parent.tag) {
      case HostComponent: {
        hostParent = parent.stateNode;
        break findParent;
      }
      case HostRoot: {
        hostParent = parent.stateNode.containerInfo;
        break findParent;
      }
      default:
        break;
    }
    parent = parent.return;
  }
  commitDeletionEffectsOnFiber(root, returnFiber, deletedFiber);
  hostParent = null;
}
function commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor, deletedFiber) {
  switch (deletedFiber.tag) {
    case HostComponent:
    case HostText: {
      recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor, deletedFiber);
      if (hostParent !== null) {
        removeChild(hostParent, deletedFiber.stateNode);
      }
      break;
    }
    default:
      break;
  }
}
function recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor, parent) {
  let child = parent.child;
  while (child !== null) {
    commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor, child);
    child = child.sibling;
  }
}
function recursivelyTraverseMutationEffects(root, parentFiber) {
  const deletions = parentFiber.deletions;
  if (deletions !== null) {
    for (let i = 0; i < deletions.length; i++) {
      const childToDelete = deletions[i];
      commitDeletionEffects(root, parentFiber, childToDelete);
    }
  }
  if (parentFiber.subtreeFlags & MutationMask) {
    let { child } = parentFiber;
    while (child !== null) {
      commitMutationEffectsOnFiber(child, root);
      child = child.sibling;
    }
  }
}
function isHostParent(fiber) {
```

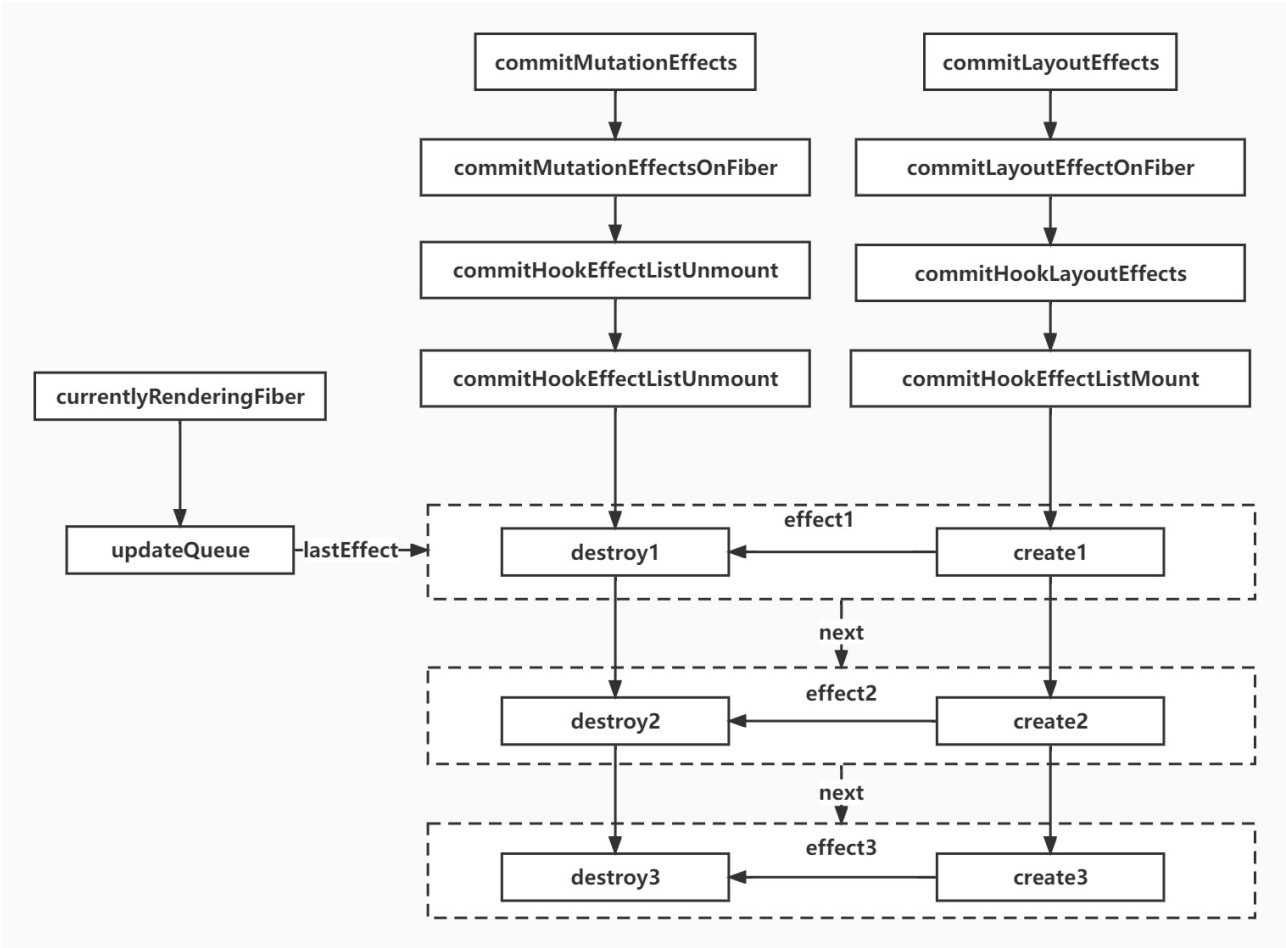
```
  return fiber.tag
}
function getHostParentFiber(fiber) {
  let parent = fiber.return;
  while (parent !== null) {
    if (isHostParent(parent)) {
      return parent;
    }
    parent = parent.return;
  }
  return parent;
}
function insertOrAppendPlacementNode(node, before, parent) {
  const { tag } = node;
  const isHost = tag
  if (isHost) {
    const { stateNode } = node;
    if (before) {
      insertBefore(parent, stateNode, before);
    } else {
      appendChild(parent, stateNode);
    }
  } else {
    const { child } = node;
    if (child !== null) {
      insertOrAppendPlacementNode(child, before, parent);
      let { sibling } = child;
      while (sibling !== null) {
        insertOrAppendPlacementNode(sibling, before, parent);
        sibling = sibling.sibling;
      }
    }
  }
}
function getHostSibling(fiber) {
  let node = fiber;
  siblings: while (true) {
    // 如果我们没有找到任何东西，让我们试试下一个弟弟
    while (node.sibling
      if (node.return
        // 如果我们是根Fiber或者父亲是原生节点，我们就是最后的弟弟
        return null;
      }
      node = node.return;
    }
    // node.sibling.return = node.return
    node = node.sibling;
    while (node.tag !== HostComponent && node.tag !== HostText) {
      // 如果它不是原生节点，并且，我们可能在其中有一个原生节点
      // 试着向下搜索，直到找到为止
      if (node.flags & Placement) {
        // 如果我们没有孩子，可以试试弟弟
        continue siblings;
      } else {
        // node.child.return = node
        node = node.child;
      }
    } // Check if this host node is stable or about to be placed.

    // 检查此原生节点是否稳定可以放置
    if (!(node.flags & Placement)) {
      // 找到它了!

      return node.stateNode;
    }
  }
}
function commitPlacement(finishedWork) {
  const parentFiber = getHostParentFiber(finishedWork);
  switch (parentFiber.tag) {
    case HostComponent: {
      const parent = parentFiber.stateNode;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    case HostRoot: {
      const parent = parentFiber.stateNode.containerInfo;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    default:
      break;
  }
}
function commitReconciliationEffects(finishedWork) {
  const { flags } = finishedWork;
  if (flags & Placement) {
    commitPlacement(finishedWork);
    finishedWork.flags &= ~Placement;
  }
}
export function commitMutationEffectsOnFiber(finishedWork, root) {
  const current = finishedWork.alternate;
  const flags = finishedWork.flags;
  switch (finishedWork.tag) {
    case HostRoot: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    case FunctionComponent: {
      recursivelyTraverseMutationEffects(root, finishedWork);
```

```
      commitReconciliationEffects(finishedWork);
      break;
    }
    case HostComponent: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      if (flags & Update) {
        const instance = finishedWork.stateNode;
        if (instance != null) {
          const newProps = finishedWork.memoizedProps;
          const oldProps = current !== null ? current.memoizedProps : newProps;
          const type = finishedWork.type;
          const updatePayload = finishedWork.updateQueue;
          finishedWork.updateQueue = null;
          if (updatePayload !== null) {
            commitUpdate(instance, updatePayload, type, oldProps, newProps, finishedWork);
          }
        }
      }
      break;
    }
    case HostText: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    default: {
      break;
    }
  }
}
```

34.useLayoutEffect

- 其函数签名与 useEffect 相同，但它会在所有的 DOM 变更之后同步调用 effect
- useEffect不会阻塞浏览器渲染，而 useLayoutEffect 会浏览器渲染
- useEffect会在浏览器渲染结束后执行,useLayoutEffect 则是在 DOM 更新完成后,浏览器绘制之前执行






```
6292486 'FunctionComponent' 'Counter' ▶ {}
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
useLayoutEffect2
useEffect1
useEffect3
6292484 'FunctionComponent' 'Counter' ▶ {}
更新 HostComponent div ▶ {children: 1, onClick: ƒ}
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
destroy useLayoutEffect2
useLayoutEffect2
destroy useEffect1
destroy useEffect3
useEffect1
useEffect3
```

34.1 src\main.jsx #

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

function Counter() {
  const [number, setNumber] = React.useState(0);
  React.useEffect(() => {
    console.log("useEffect1");
    return () => {
      console.log("destroy useEffect1");
    };
  });
+ React.useLayoutEffect(() => {
+   console.log("useLayoutEffect2");
+   return () => {
+     console.log("destroy useLayoutEffect2");
+   };
+ });
  React.useEffect(() => {
    console.log("useEffect3");
    return () => {
      console.log("destroy useEffect3");
    };
  });
  return (
     {
       setNumber(number + 1);
     }}
   >
     {number}

  );
}
let element = ;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

33.2 react\index.js

src\react\index.js

```
export {
  __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED,
  useReducer,
  useState,
  useEffect,
+ useLayoutEffect,
} from "./src/React";
```

34.3 React.js

src\react\src\React.js

```
import { useReducer, useState, useEffect, useLayoutEffect } from "./ReactHooks";
import ReactSharedInternals from "./ReactSharedInternals";

export {
  useReducer,
  useState,
  useEffect,
+ useLayoutEffect,
  ReactSharedInternals as __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED,
};
```

34.4 ReactHooks.js

src\react\src\ReactHooks.js

```
import ReactCurrentDispatcher from "./ReactCurrentDispatcher";

function resolveDispatcher() {
  const dispatcher = ReactCurrentDispatcher.current;
  return dispatcher;
}

export function useReducer(reducer, initialArg, init) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useReducer(reducer, initialArg, init);
}

export function useState(initialState) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useState(initialState);
}

export function useEffect(create,deps) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useEffect(create,deps);
}
+export function useLayoutEffect(create,deps) {
+  const dispatcher = resolveDispatcher();
+  return dispatcher.useLayoutEffect(create,deps);
+}
```

src\react-reconciler\src\ReactHookEffectTags.js

```
export const HasEffect = 0b0001;
export const Passive = 0b1000;
+export const Layout = 0b0100;
```

34.6 ReactFiberFlags.js

src\react-reconciler\src\ReactFiberFlags.js

```
export const NoFlags = 0b0000000000000000000000000;
export const Placement = 0b0000000000000000000000010;
export const Update = 0b0000000000000000000000100;
export const ChildDeletion = 0b0000000000000000000001000;
export const MutationMask = Placement | Update;
export const Passive = 0b0000000000000000010000000000;
+export const LayoutMask = Update;
```

34.7 ReactFiberHooks.js

src\react-reconciler\src\ReactFiberHooks.js

```
import ReactSharedInternals from "shared/ReactSharedInternals";
import { enqueueConcurrentHookUpdate } from "./ReactFiberConcurrentUpdates";
import { scheduleUpdateOnFiber } from "./ReactFiberWorkLoop";
import is from "shared/objectIs";
+import { Passive as PassiveEffect, Update as UpdateEffect } from "./ReactFiberFlags";
+import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout } from "./ReactHookEffectTags";

const { ReactCurrentDispatcher } = ReactSharedInternals;
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;

const HooksDispatcherOnMountInDEV = {
  useReducer: mountReducer,
  useState: mountState,
  useEffect: mountEffect,
+ useLayoutEffect: mountLayoutEffect,
};
const HooksDispatcherOnUpdateInDEV = {
  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) {
+  const fiberFlags = UpdateEffect;
+  return mountEffectImpl(fiberFlags, HookLayout, create, deps);
+}
+function updateEffect(create, deps) {
+  return updateEffectImpl(PassiveEffect, HookPassive, create, deps);
+}
function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
  const hook = updateWorkInProgressHook();
  const nextDeps = deps
  let destroy;
  if (currentHook !== null) {
    const prevEffect = currentHook.memoizedState;
    destroy = prevEffect.destroy;
    if (nextDeps !== null) {
      const prevDeps = prevEffect.deps;
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        hook.memoizedState = 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 (is(nextDeps[i], prevDeps[i])) {
      continue;
    }
    return false;
  }

  return true;
}
function mountEffect(create, deps) {
  return mountEffectImpl(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,
  };
}
function basicStateReducer(state, action) {
  return typeof action
}
function mountReducer(reducer, initialArg) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialArg;
  const queue = {
    pending: null,
    dispatch: null,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchReducerAction.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function updateReducer(reducer) {
  const hook = updateWorkInProgressHook();
  const queue = hook.queue;
  queue.lastRenderedReducer = reducer;
  const current = currentHook;
  const pendingQueue = queue.pending;
  let baseQueue = null;
  let newState = current.memoizedState;
  if (pendingQueue !== null) {
    baseQueue = pendingQueue;
    queue.pending = null;
  }
  if (baseQueue !== null) {
    const first = baseQueue.next;
    let update = first;
    do {
      if (update.hasEagerState) {
        newState = update.eagerState;
      } else {
        const action = update.action;
        newState = reducer(newState, action);
      }
      update = update.next;
    } while (update !== null && update !== first);
  }
  hook.memoizedState = newState;
  queue.lastRenderedState = newState;
  const dispatch = queue.dispatch;
  return [hook.memoizedState, dispatch];
}
function mountState(initialState) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialState;
  const queue = {
    pending: null,
    dispatch: null,
    lastRenderedReducer: basicStateReducer,
    lastRenderedState: initialState,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchSetState.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function dispatchSetState(fiber, queue, action) {
  const update = {
    action,
    hasEagerState: false,
    eagerState: null,
    next: null,
  };
  const lastRenderedReducer = queue.lastRenderedReducer;
  const currentState = queue.lastRenderedState;
  const eagerState = lastRenderedReducer(currentState, action);
  update.hasEagerState = true;
  update.eagerState = eagerState;
  if (is(eagerState, currentState)) {
    return;
  }
  const root = enqueueConcurrentHookUpdate(fiber, queue, update);
  scheduleUpdateOnFiber(root, fiber);
}
function updateState(initialState) {
  return updateReducer(basicStateReducer, initialState);
}
function mountWorkInProgressHook() {
  const hook = {
    memoizedState: null,
    queue: null,
    next: null,
  };
  if (workInProgressHook
    currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
  } else {
    workInProgressHook = workInProgressHook.next = hook;
```

```
  }
  return workInProgressHook;
}
function dispatchReducerAction(fiber, queue, action) {
  const update = {
    action,
    next: null,
  };
  const root = enqueueConcurrentHookUpdate(fiber, queue, update);
  scheduleUpdateOnFiber(root, fiber);
}

function updateWorkInProgressHook() {
  let nextCurrentHook;
  if (currentHook
    const current = currentlyRenderingFiber.alternate;
    if (current !== null) {
      nextCurrentHook = current.memoizedState;
    } else {
      nextCurrentHook = null;
    }
  } else {
    nextCurrentHook = currentHook.next;
  }

  let nextWorkInProgressHook;
  if (workInProgressHook
    nextWorkInProgressHook = currentlyRenderingFiber.memoizedState;
  } else {
    nextWorkInProgressHook = workInProgressHook.next;
  }

  if (nextWorkInProgressHook !== null) {
    workInProgressHook = nextWorkInProgressHook;
    nextWorkInProgressHook = workInProgressHook.next;
    currentHook = nextCurrentHook;
  } else {
    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 renderWithHooks(current, workInProgress, Component, props) {
  currentlyRenderingFiber = workInProgress;
  workInProgress.updateQueue = null;
  workInProgress.memoizedState = null;
  if (current !== null && current.memoizedState !== null) {
    ReactCurrentDispatcher.current = HooksDispatcherOnUpdateInDEV;
  } else {
    ReactCurrentDispatcher.current = HooksDispatcherOnMountInDEV;
  }
  const children = Component(props);
  currentlyRenderingFiber = null;
  workInProgressHook = null;
  currentHook = null;
  return children;
}
```

34.8 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import { scheduleCallback } from "scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
import { MutationMask, NoFlags, Placement, Update, ChildDeletion, Passive } from "./ReactFiberFlags";
import {
  commitMutationEffects,
  commitPassiveUnmountEffects,
  commitPassiveMountEffects,
+  commitLayoutEffects,
} from "./ReactFiberCommitWork";
import { finishQueueingConcurrentUpdates } from "./ReactFiberConcurrentUpdates";
import { FunctionComponent, IndeterminateComponent, HostRoot, HostComponent, HostText } from "./ReactWorkTags";

let workInProgress = null;
let rootDoesHavePassiveEffects = false;
let rootWithPendingPassiveEffects = null;
export function scheduleUpdateOnFiber(root) {
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  scheduleCallback(performConcurrentWorkOnRoot.bind(null, root));
}
function performConcurrentWorkOnRoot(root) {
  renderRootSync(root);
  const finishedWork = root.current.alternate;
  printFiber(finishedWork);
  console.log(`~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`);
  root.finishedWork = finishedWork;
  commitRoot(root);
}
```

```
export function flushPassiveEffects() {
  if (rootWithPendingPassiveEffects !== null) {
    const root = rootWithPendingPassiveEffects;
    commitPassiveUnmountEffects(root.current);
    commitPassiveMountEffects(root, root.current);
  }
}
function commitRoot(root) {
  const { finishedWork } = root;
  if ((finishedWork.subtreeFlags & Passive) !== NoFlags || (finishedWork.flags & Passive) !== NoFlags) {
    if (!rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = true;
      scheduleCallback(flushPassiveEffects);
    }
  }
  const subtreeHasEffects = (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
    commitMutationEffects(finishedWork, root);
+   commitLayoutEffects(finishedWork, root);
    root.current = finishedWork;
    if (rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = false;
      rootWithPendingPassiveEffects = root;
    }
  }
  root.current = finishedWork;
}
function prepareFreshStack(root) {
  workInProgress = createWorkInProgress(root.current, null);
  finishQueueingConcurrentUpdates();
}
function renderRootSync(root) {
  prepareFreshStack(root);
  workLoopSync();
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
}

function printFiber(fiber) {
  /*
  fiber.flags &= ~Forked;
  fiber.flags &= ~PlacementDEV;
  fiber.flags &= ~Snapshot;
  fiber.flags &= ~PerformedWork;
  */
  if (fiber.flags !== 0) {
    console.log(
      getFlags(fiber.flags),
      getTag(fiber.tag),
      typeof fiber.type
      fiber.memoizedProps
    );
    if (fiber.deletions) {
      for (let i = 0; i < fiber.deletions.length; i++) {
        const childToDelete = fiber.deletions[i];
        console.log(getTag(childToDelete.tag), childToDelete.type, childToDelete.memoizedProps);
      }
    }
  }
  let child = fiber.child;
  while (child) {
    printFiber(child);
    child = child.sibling;
  }
}
function getTag(tag) {
  switch (tag) {
    case FunctionComponent:
      return `FunctionComponent`;
    case HostRoot:
      return `HostRoot`;
    case HostComponent:
      return `HostComponent`;
```

```
    case HostText:
      return HostText;
    default:
      return tag;
  }
}
function getFlags(flags) {
  if (flags
    return `自己移动和子元素有删除`;
  }
  if (flags
    return `自己有更新和子元素有删除`;
  }
  if (flags
    return `子元素有删除`;
  }
  if (flags
    return `移动并更新`;
  }
  if (flags
    return `插入`;
  }
  if (flags
    return `更新`;
  }
  return flags;
}
```

34.9 ReactFiberCommitWork.js

src\react-reconciler\src\ReactFiberCommitWork.js

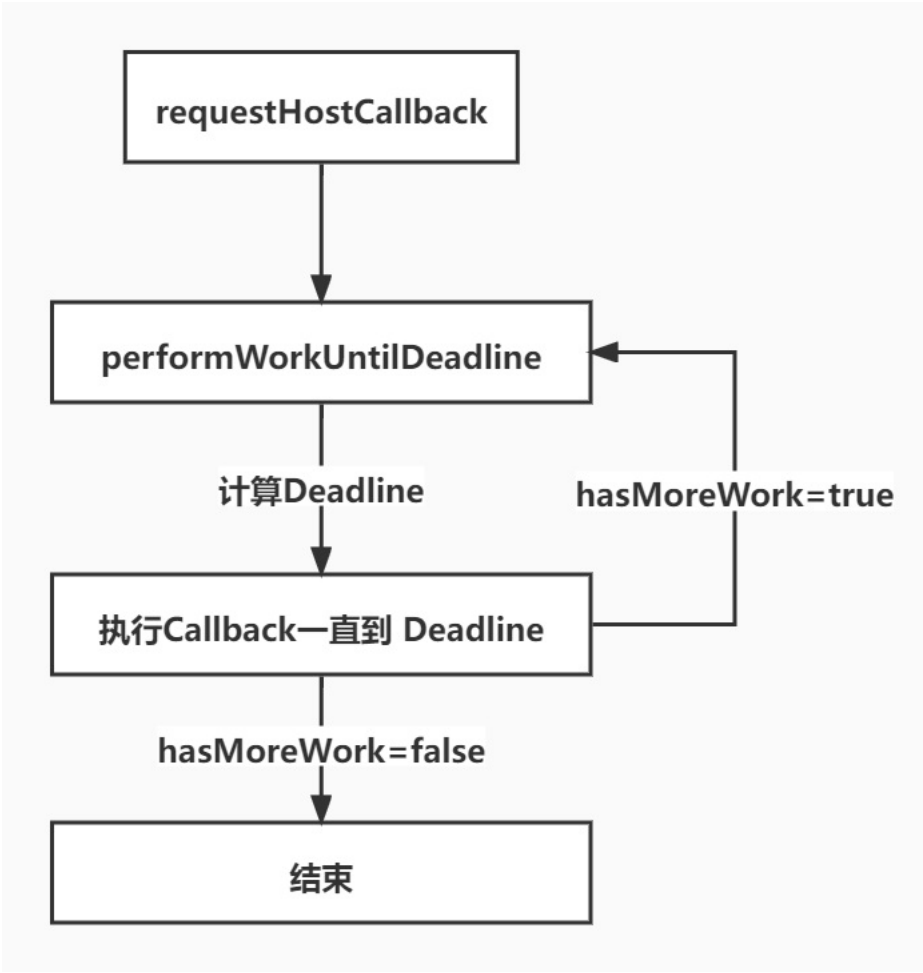
```
import { HostRoot, HostComponent, HostText, FunctionComponent } from "./ReactWorkTags";
+import { Passive, MutationMask, Placement, Update, LayoutMask } from "./ReactFiberFlags";
import { insertBefore, appendChild, commitUpdate, removeChild } from "react-dom-bindings/src/client/ReactDOMHostConfig";
+import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout } from "./ReactHookEffectTags";

export function commitMutationEffects(finishedWork, root) {
  commitMutationEffectsOnFiber(finishedWork, root);
}
export function commitPassiveUnmountEffects(finishedWork) {
  commitPassiveUnmountOnFiber(finishedWork);
}
function commitPassiveUnmountOnFiber(finishedWork) {
  switch (finishedWork.tag) {
    case FunctionComponent: {
      recursivelyTraversePassiveUnmountEffects(finishedWork);
      if (finishedWork.flags & Passive) {
        commitHookPassiveUnmountEffects(finishedWork, finishedWork.return, HookPassive | HookHasEffect);
      }
      break;
    }
    default: {
      recursivelyTraversePassiveUnmountEffects(finishedWork);
      break;
    }
  }
}
function recursivelyTraversePassiveUnmountEffects(parentFiber) {
  if (parentFiber.subtreeFlags & Passive) {
    let child = parentFiber.child;
    while (child !== null) {
      commitPassiveUnmountOnFiber(child);
      child = child.sibling;
    }
  }
}
function commitHookPassiveUnmountEffects(finishedWork, nearestMountedAncestor, hookFlags) {
  commitHookEffectListUnmount(hookFlags, finishedWork, nearestMountedAncestor);
}

function commitHookEffectListUnmount(flags, 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 & flags)
        const destroy = effect.destroy;
        effect.destroy = undefined;
        if (destroy !== undefined) {
          destroy();
        }
      }
      effect = effect.next;
    } while (effect !== firstEffect);
  }
}

export function commitPassiveMountEffects(root, finishedWork) {
  commitPassiveMountOnFiber(root, finishedWork);
}
function commitPassiveMountOnFiber(finishedRoot, finishedWork) {
  const flags = finishedWork.flags;
  switch (finishedWork.tag) {
    case FunctionComponent: {
      recursivelyTraversePassiveMountEffects(finishedRoot, finishedWork);
      if (flags & Passive) {
        commitHookPassiveMountEffects(finishedWork, HookPassive | HookHasEffect);
      }
      break;
    }
```

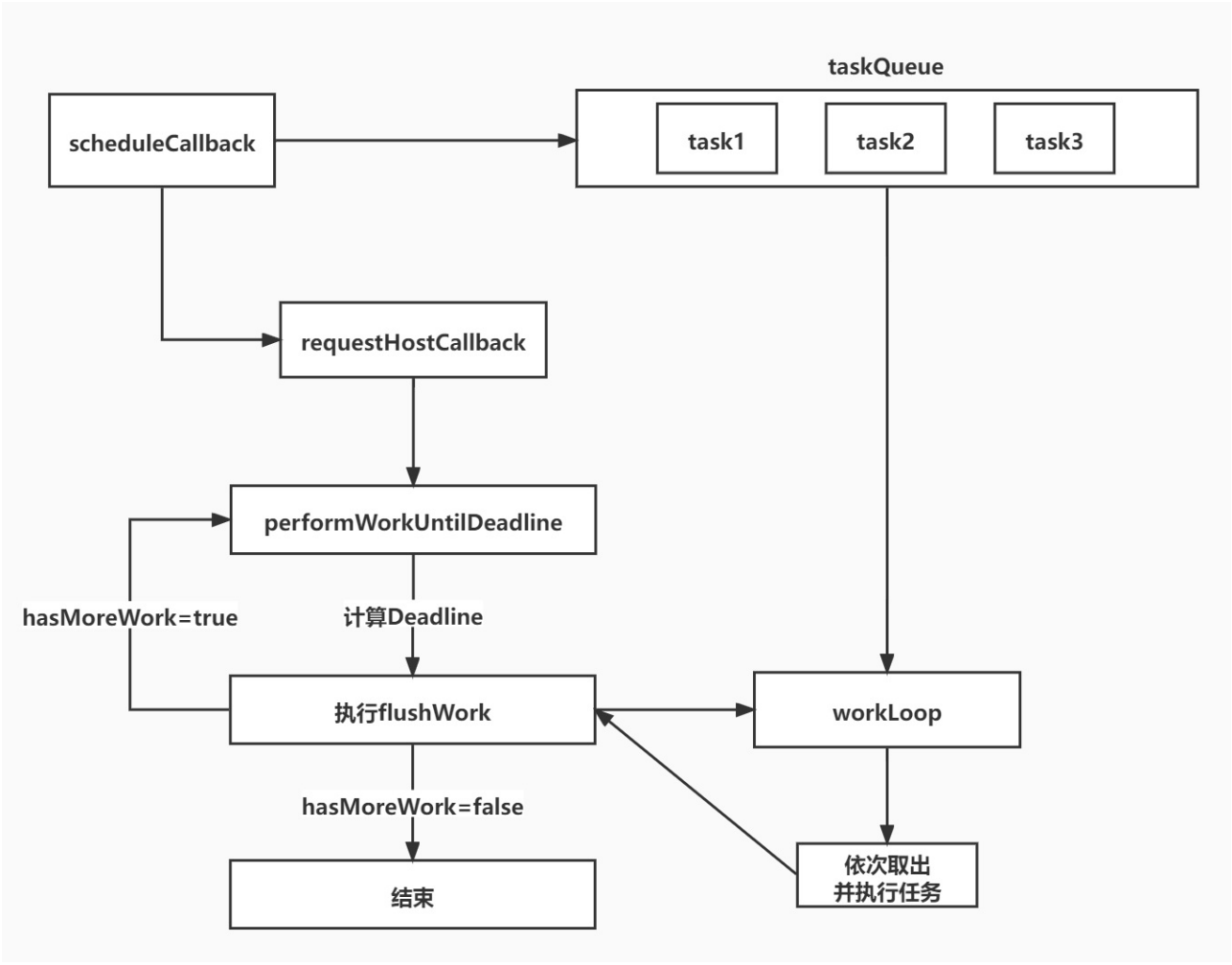
```
    case HostRoot: {
      recursivelyTraversePassiveMountEffects(finishedRoot, finishedWork);
      break;
    }
    default:
      break;
  }
}
function commitHookPassiveMountEffects(finishedWork, hookFlags) {
  commitHookEffectListMount(hookFlags, finishedWork);
}
function commitHookEffectListMount(flags, 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 & flags)
        const create = effect.create;
        effect.destroy = create();
      }
      effect = effect.next;
    } while (effect !== firstEffect);
  }
}
function recursivelyTraversePassiveMountEffects(root, parentFiber) {
  if (parentFiber.subtreeFlags & Passive) {
    let child = parentFiber.child;
    while (child !== null) {
      commitPassiveMountOnFiber(root, child);
      child = child.sibling;
    }
  }
}
let hostParent = null;
function commitDeletionEffects(root, returnFiber, deletedFiber) {
  let parent = returnFiber;
  findParent: while (parent !== null) {
    switch (parent.tag) {
      case HostComponent: {
        hostParent = parent.stateNode;
        break findParent;
      }
      case HostRoot: {
        hostParent = parent.stateNode.containerInfo;
        break findParent;
      }
      default:
        break;
    }
    parent = parent.return;
  }
  commitDeletionEffectsOnFiber(root, returnFiber, deletedFiber);
  hostParent = null;
}
function commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor, deletedFiber) {
  switch (deletedFiber.tag) {
    case HostComponent:
    case HostText: {
      recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor, deletedFiber);
      if (hostParent !== null) {
        removeChild(hostParent, deletedFiber.stateNode);
      }
      break;
    }
    default:
      break;
  }
}
function recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor, parent) {
  let child = parent.child;
  while (child !== null) {
    commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor, child);
    child = child.sibling;
  }
}
function recursivelyTraverseMutationEffects(root, parentFiber) {
  const deletions = parentFiber.deletions;
  if (deletions !== null) {
    for (let i = 0; i < deletions.length; i++) {
      const childToDelete = deletions[i];
      commitDeletionEffects(root, parentFiber, childToDelete);
    }
  }
  if (parentFiber.subtreeFlags & MutationMask) {
    let { child } = parentFiber;
    while (child !== null) {
      commitMutationEffectsOnFiber(child, root);
      child = child.sibling;
    }
  }
}
function isHostParent(fiber) {
  return fiber.tag
}
function getHostParentFiber(fiber) {
  let parent = fiber.return;
  while (parent !== null) {
    if (isHostParent(parent)) {
      return parent;
    }
    parent = parent.return;
  }
```

```
  return parent;
}
function insertOrAppendPlacementNode(node, before, parent) {
  const { tag } = node;
  const isHost = tag
  if (isHost) {
    const { stateNode } = node;
    if (before) {
      insertBefore(parent, stateNode, before);
    } else {
      appendChild(parent, stateNode);
    }
  } else {
    const { child } = node;
    if (child !== null) {
      insertOrAppendPlacementNode(child, before, parent);
      let { sibling } = child;
      while (sibling !== null) {
        insertOrAppendPlacementNode(sibling, before, parent);
        sibling = sibling.sibling;
      }
    }
  }
}
function getHostSibling(fiber) {
  let node = fiber;
  siblings: while (true) {
    // 如果我们没有找到任何东西，让我们试试下一个弟弟
    while (node.sibling
      if (node.return
        // 如果我们是根Fiber或者父亲是原生节点，我们就是最后的弟弟
        return null;
      }
      node = node.return;
    }
    // node.sibling.return = node.return
    node = node.sibling;
    while (node.tag !== HostComponent && node.tag !== HostText) {
      // 如果它不是原生节点，并且，我们可能在其中有一个原生节点
      // 试着向下搜索，直到找到为止
      if (node.flags & Placement) {
        // 如果我们没有孩子，可以试试弟弟
        continue siblings;
      } else {
        // node.child.return = node
        node = node.child;
      }
    } // Check if this host node is stable or about to be placed.

    // 检查此原生节点是否稳定可以放置
    if (!(node.flags & Placement)) {
      // 找到它了！

      return node.stateNode;
    }
  }
}
function commitPlacement(finishedWork) {
  const parentFiber = getHostParentFiber(finishedWork);
  switch (parentFiber.tag) {
    case HostComponent: {
      const parent = parentFiber.stateNode;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    case HostRoot: {
      const parent = parentFiber.stateNode.containerInfo;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    default:
      break;
  }
}
function commitReconciliationEffects(finishedWork) {
  const { flags } = finishedWork;
  if (flags & Placement) {
    commitPlacement(finishedWork);
    finishedWork.flags &= ~Placement;
  }
}
export function commitMutationEffectsOnFiber(finishedWork, root) {
  const current = finishedWork.alternate;
  const flags = finishedWork.flags;
  switch (finishedWork.tag) {
    case HostRoot: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    case FunctionComponent: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
+     if (flags & Update) {
+       commitHookEffectListUnmount(HookLayout | HookHasEffect, finishedWork, finishedWork.return);
+     }
      break;
    }
    case HostComponent: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      if (flags & Update) {
```

```
        const instance = finishedWork.stateNode;
        if (instance != null) {
          const newProps = finishedWork.memoizedProps;
          const oldProps = current !== null ? current.memoizedProps : newProps;
          const type = finishedWork.type;
          const updatePayload = finishedWork.updateQueue;
          finishedWork.updateQueue = null;
          if (updatePayload !== null) {
            commitUpdate(instance, updatePayload, type, oldProps, newProps, finishedWork);
          }
        }
      }
      break;
    }
    case HostText: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    default: {
      break;
    }
  }
}

+export function commitLayoutEffects(finishedWork, root) {
+  const current = finishedWork.alternate;
+  commitLayoutEffectOnFiber(root, current, finishedWork);
+}
+function commitLayoutEffectOnFiber(finishedRoot, current, finishedWork) {
+  const flags = finishedWork.flags;
+  switch (finishedWork.tag) {
+    case FunctionComponent: {
+      recursivelyTraverseLayoutEffects(finishedRoot, finishedWork);
+      if (flags & Update) {
+        commitHookLayoutEffects(finishedWork, HookLayout | HookHasEffect);
+      }
+      break;
+    }
+    case HostRoot: {
+      recursivelyTraverseLayoutEffects(finishedRoot, finishedWork);
+      break;
+    }
+    default:
+      break;
+  }
+}
+function recursivelyTraverseLayoutEffects(root, parentFiber) {
+  if (parentFiber.subtreeFlags & LayoutMask) {
+    let child = parentFiber.child;
+    while (child !== null) {
+      const current = child.alternate;
+      commitLayoutEffectOnFiber(root, current, child);
+      child = child.sibling;
+    }
+  }
+}
+function commitHookLayoutEffects(finishedWork, hookFlags) {
+  commitHookEffectListMount(hookFlags, finishedWork);
+}
```

35.Scheduler_scheduleCallback


requestHostCallback
performWorkUntilDeadline
计算Deadline
hasMoreWork=true
执行Callback一直到 Deadline
hasMoreWork=false
结束



35.1 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
+import { NormalPriority as NormalSchedulerPriority, scheduleCallback as Scheduler_scheduleCallback } from "./Scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
import { MutationMask, NoFlags, Placement, Update, ChildDeletion, Passive } from "./ReactFiberFlags";
import {
  commitMutationEffects,
  commitPassiveUnmountEffects,
  commitPassiveMountEffects,
  commitLayoutEffects,
} from "./ReactFiberCommitWork";
import { finishQueueingConcurrentUpdates } from "./ReactFiberConcurrentUpdates";
import { FunctionComponent, IndeterminateComponent, HostRoot, HostComponent, HostText } from "./ReactWorkTags";

let workInProgress = null;
let rootDoesHavePassiveEffects = false;
let rootWithPendingPassiveEffects = null;
export function scheduleUpdateOnFiber(root) {
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
+ Scheduler_scheduleCallback(NormalSchedulerPriority, performConcurrentWorkOnRoot.bind(null, root));
}
function performConcurrentWorkOnRoot(root) {
  renderRootSync(root);
  const finishedWork = root.current.alternate;
  printFiber(finishedWork);
  console.log(`~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`);
  root.finishedWork = finishedWork;
  commitRoot(root);
}
export function flushPassiveEffects() {
  if (rootWithPendingPassiveEffects !== null) {
    const root = rootWithPendingPassiveEffects;
    commitPassiveUnmountEffects(root.current);
    commitPassiveMountEffects(root, root.current);
  }
}
function commitRoot(root) {
  const { finishedWork } = root;
  if ((finishedWork.subtreeFlags & Passive) !== NoFlags || (finishedWork.flags & Passive) !== NoFlags) {
    if (!rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = true;
+     Scheduler_scheduleCallback(NormalSchedulerPriority, flushPassiveEffects);
    }
```

```
  }
  const subtreeHasEffects = (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
    commitMutationEffects(finishedWork, root);
    commitLayoutEffects(finishedWork, root);
    root.current = finishedWork;
    if (rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = false;
      rootWithPendingPassiveEffects = root;
    }
  }
  root.current = finishedWork;
}
function prepareFreshStack(root) {
  workInProgress = createWorkInProgress(root.current, null);
  finishQueueingConcurrentUpdates();
}
function renderRootSync(root) {
  prepareFreshStack(root);
  workLoopSync();
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
}

function printFiber(fiber) {
  /*
  fiber.flags &= ~Forked;
  fiber.flags &= ~PlacementDEV;
  fiber.flags &= ~Snapshot;
  fiber.flags &= ~PerformedWork;
  */
  if (fiber.flags !== 0) {
    console.log(
      getFlags(fiber.flags),
      getTag(fiber.tag),
      typeof fiber.type
      fiber.memoizedProps
    );
    if (fiber.deletions) {
      for (let i = 0; i < fiber.deletions.length; i++) {
        const childToDelete = fiber.deletions[i];
        console.log(getTag(childToDelete.tag), childToDelete.type, childToDelete.memoizedProps);
      }
    }
  }
  let child = fiber.child;
  while (child) {
    printFiber(child);
    child = child.sibling;
  }
}
function getTag(tag) {
  switch (tag) {
    case FunctionComponent:
      return `FunctionComponent`;
    case HostRoot:
      return `HostRoot`;
    case HostComponent:
      return `HostComponent`;
    case HostText:
      return HostText;
    default:
      return tag;
  }
}
function getFlags(flags) {
  if (flags
    return `自己移动和子元素有删除`;
  }
  if (flags
    return `自己有更新和子元素有删除`;
  }
  if (flags
```

```
    return `子元素有删除`;
  }
  if (flags
    return `移动并更新`;
  }
  if (flags
    return `插入`;
  }
  if (flags
    return `更新`;
  }
  return flags;
}
```

35.2 Scheduler.js

src\react-reconciler\src\Scheduler.js

```
import * as Scheduler from "scheduler";

export const scheduleCallback = Scheduler.unstable_scheduleCallback;
export const NormalPriority = Scheduler.unstable_NormalPriority;
```

35.3 Scheduler.js

src\scheduler\src\forks\Scheduler.js

```
import {
  ImmediatePriority,
  UserBlockingPriority,
  NormalPriority,
  LowPriority,
  IdlePriority,
} from "../SchedulerPriorities";
import { push, pop, peek } from "../SchedulerMinHeap";
import { frameYieldMs } from "../SchedulerFeatureFlags";

const maxSigned31BitInt = 1073741823;
const IMMEDIATE_PRIORITY_TIMEOUT = -1;
const USER_BLOCKING_PRIORITY_TIMEOUT = 250;
const NORMAL_PRIORITY_TIMEOUT = 5000;
const LOW_PRIORITY_TIMEOUT = 10000;
const IDLE_PRIORITY_TIMEOUT = maxSigned31BitInt;

const taskQueue = [];
let taskIdCounter = 1;
let scheduledHostCallback = null;
let startTime = -1;
let currentTask = null;
const frameInterval = frameYieldMs;
const channel = new MessageChannel();
const port = channel.port2;

const getCurrentTime = () => performance.now();
channel.port1.onmessage = performWorkUntilDeadline;

function schedulePerformWorkUntilDeadline() {
  port.postMessage(null);
}
function performWorkUntilDeadline() {
  if (scheduledHostCallback !== null) {
    startTime = getCurrentTime();
    let hasMoreWork = true;
    try {
      hasMoreWork = scheduledHostCallback(startTime);
    } finally {
      if (hasMoreWork) {
        schedulePerformWorkUntilDeadline();
      } else {
        scheduledHostCallback = null;
      }
    }
  }
}
function requestHostCallback(callback) {
  scheduledHostCallback = callback;
  schedulePerformWorkUntilDeadline();
}

function unstable_scheduleCallback(priorityLevel, callback) {
  const currentTime = getCurrentTime();
  const startTime = currentTime;
  let timeout;
  switch (priorityLevel) {
    case ImmediatePriority:
      timeout = IMMEDIATE_PRIORITY_TIMEOUT;
      break;
    case UserBlockingPriority:
      timeout = USER_BLOCKING_PRIORITY_TIMEOUT;
      break;
    case IdlePriority:
      timeout = IDLE_PRIORITY_TIMEOUT;
      break;
    case LowPriority:
      timeout = LOW_PRIORITY_TIMEOUT;
      break;
    case NormalPriority:
    default:
      timeout = NORMAL_PRIORITY_TIMEOUT;
      break;
  }
  const expirationTime = startTime + timeout;
  const newTask = {
```

```
    id: taskIdCounter++,
    callback,
    priorityLevel,
    startTime,
    expirationTime,
    sortIndex: -1,
  };
  newTask.sortIndex = expirationTime;
  push(taskQueue, newTask);
  requestHostCallback(flushWork);
  return newTask;
}

function flushWork(initialTime) {
  return workLoop(initialTime);
}

function shouldYieldToHost() {
  const timeElapsed = getCurrentTime() - startTime;
  if (timeElapsed < frameInterval) {
    return false;
  }
  return true;
}
function workLoop(initialTime) {
  let currentTime = initialTime;
  currentTask = peek(taskQueue);
  while (currentTask !== null) {
    if (currentTask.expirationTime > currentTime && shouldYieldToHost()) {
      break;
    }
    const callback = currentTask.callback;
    if (typeof callback === "function") {
      currentTask.callback = null;
      const didUserCallbackTimeout = currentTask.expirationTime const continuationCallback = callback(didUserCallbackTimeout);
      currentTime = getCurrentTime();
      if (typeof continuationCallback === "function") {
        currentTask.callback = continuationCallback;
        return true;
      }
      if (currentTask === peek(taskQueue)) {
        pop(taskQueue);
      }
    } else {
      pop(taskQueue);
    }
    currentTask = peek(taskQueue);
  }
  if (currentTask !== null) {
    return true;
  }
  return false;
}

export { NormalPriority as unstable_NormalPriority, unstable_scheduleCallback };
```

35.4 SchedulerFeatureFlags.js

src\scheduler\src\SchedulerFeatureFlags.js

```
export const frameYieldMs = 5;
```

35.5 SchedulerMinHeap.js

src\scheduler\src\SchedulerMinHeap.js

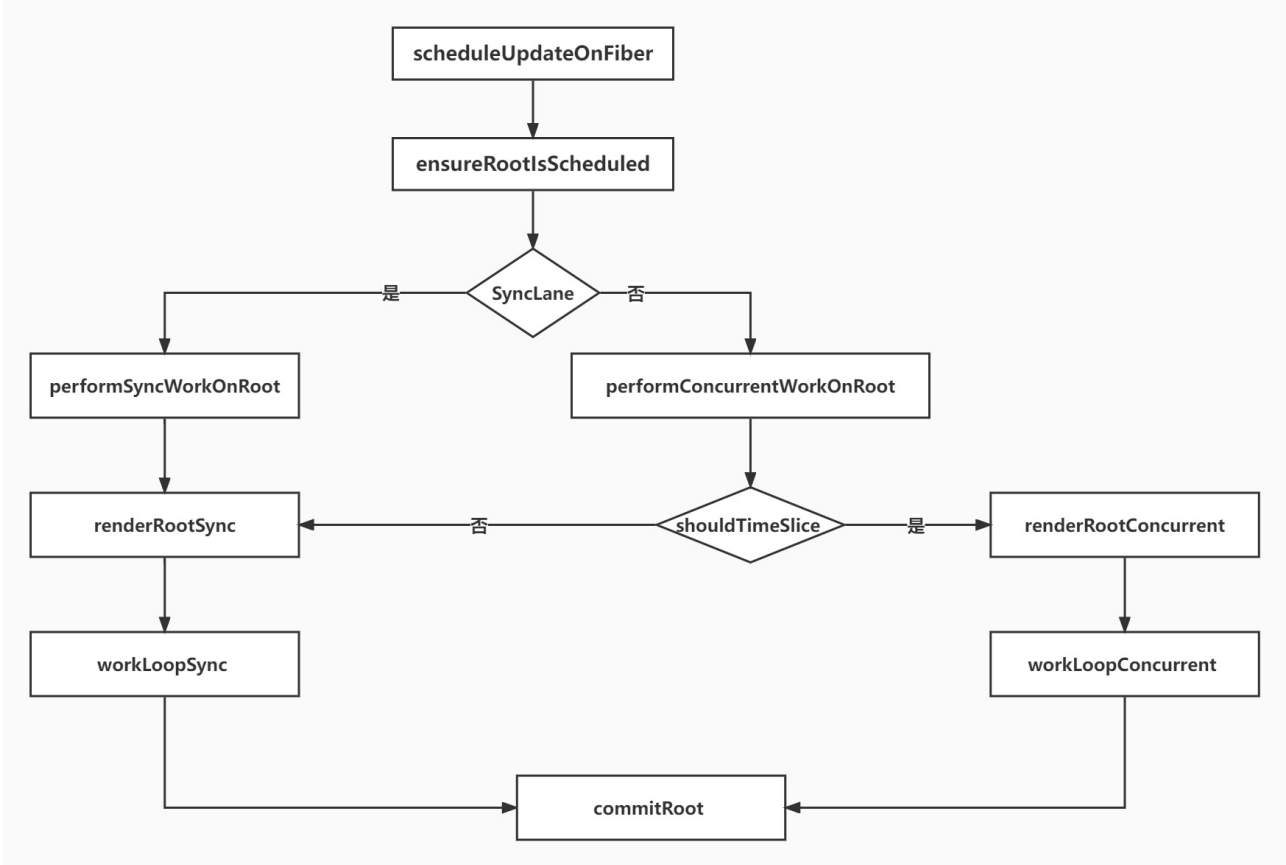
```
export function push(heap, node) {
  const index = heap.length;
  heap.push(node);
  siftUp(heap, node, index);
}
export function peek(heap) {
  return heap.length === 0 ? null : heap[0];
}
export function pop(heap) {
  if (heap.length === 0) {
    return null;
  }
  const first = heap[0];
  const last = heap.pop();
  if (last !== first) {
    heap[0] = last;
    siftDown(heap, last, 0);
  }
  return first;
}
function siftUp(heap, node, i) {
  let index = i;
  while (index > 0) {
    const parentIndex = (index - 1) >>> 1;
    const parent = heap[parentIndex];
    if (compare(parent, node) > 0) {
      heap[parentIndex] = node;
      heap[index] = parent;
      index = parentIndex;
    } else {
      return;
    }
  }
}
function siftDown(heap, node, i) {
  let index = i;
  const length = heap.length;
  const halfLength = length >>> 1;
  while (index < halfLength) {
    const leftIndex = (index + 1) * 2 - 1;
    const left = heap[leftIndex];
    const rightIndex = leftIndex + 1;
    const right = heap[rightIndex];
    if (compare(left, node) < 0) {
      if (rightIndex < length && compare(right, left) < 0) {
        heap[index] = right;
        heap[rightIndex] = node;
        index = rightIndex;
      } else {
        heap[index] = left;
        heap[leftIndex] = node;
        index = leftIndex;
      }
    } else if (rightIndex < length && compare(right, node) < 0) {
      heap[index] = right;
      heap[rightIndex] = node;
      index = rightIndex;
    } else {
      return;
    }
  }
}
function compare(a, b) {
  const diff = a.sortIndex - b.sortIndex;
  return diff !== 0 ? diff : a.id - b.id;
}
```

35.6 SchedulerPriorities.js

src\scheduler\src\SchedulerPriorities.js

```
export const NoPriority = 0;
export const ImmediatePriority = 1;
export const UserBlockingPriority = 2;
export const NormalPriority = 3;
export const LowPriority = 4;
export const IdlePriority = 5;
```

36.初次渲染



36.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";
+let element = hello;
const root = createRoot(document.getElementById("root"));
root.render(element);
```

36.2 ReactFiberReconciler.js

src\react-reconciler\src\ReactFiberReconciler.js

```
import { createFiberRoot } from "./ReactFiberRoot";
import { createUpdate, enqueueUpdate } from "./ReactFiberClassUpdateQueue";
+import { scheduleUpdateOnFiber, requestUpdateLane } from "./ReactFiberWorkLoop";
export function createContainer(containerInfo) {
  return createFiberRoot(containerInfo);
}
export function updateContainer(element, container) {
  const current = container.current;
+ const lane = requestUpdateLane(current);
+ const update = createUpdate(lane);
  update.payload = { element };
+ const root = enqueueUpdate(current, update, lane);
+ scheduleUpdateOnFiber(root, current, lane);
}
```

36.3 ReactFiberClassUpdateQueue.js

src\react-reconciler\src\ReactFiberClassUpdateQueue.js

```
import assign from "shared/assign";
+import { enqueueConcurrentClassUpdate } from './ReactFiberConcurrentUpdates'
+import { mergeLanes, NoLanes, NoLane, isSubsetOfLanes } from './ReactFiberLane';

export const UpdateState = 0;
export function initializeUpdateQueue(fiber) {
  const queue = {
+   baseState: fiber.memoizedState,
+   firstBaseUpdate: null,
+   lastBaseUpdate: null,
    shared: {
      pending: null,
    },
  };
  fiber.updateQueue = queue;
}
+export function createUpdate(lane) {
+ const update = { tag: UpdateState, lane, next: null };
  return update;
}
+export function enqueueUpdate(fiber, update, lane) {
+  const updateQueue = fiber.updateQueue
+  const sharedQueue = updateQueue.shared
+  return enqueueConcurrentClassUpdate(fiber, sharedQueue, update, lane);
```

```
+}

function getStateFromUpdate(update, prevState, nextProps) {
  switch (update.tag) {
    case UpdateState: {
      const { payload } = update;
+     let partialState;
+     if (typeof payload === 'function') {
+       partialState = payload.call(null, prevState, nextProps);
+     } else {
+       partialState = payload;
+     }
+     return assign({}, prevState, partialState);
    }
    default:
      return prevState;
  }
}
+export const processUpdateQueue = (workInProgress, props, workInProgressRootRenderLanes) => {
+  // 获取新的更新队列
+  const queue = workInProgress.updateQueue
+  // 第一个跳过的更新
+  let firstBaseUpdate = queue.firstBaseUpdate;
+  // 最后一个跳过的更新
+  let lastBaseUpdate = queue.lastBaseUpdate;
+  // 获取待生效的队列
+  const pendingQueue = queue.shared.pending
+  /**   如果有新链表合并新旧链表开始  */
+  // 如果有新的待生效的队列
+  if (pendingQueue !== null) {
+    // 先清空待生效的队列
+    queue.shared.pending = null
+    // 最后一个待生效的更新
+    const lastPendingUpdate = pendingQueue
+    // 第一个待生效的更新
+    const firstPendingUpdate = lastPendingUpdate.next
+    // 把环状链表剪开
+    lastPendingUpdate.next = null
+    // 如果没有老的更新队列
+    if (lastBaseUpdate === null) {
+      // 第一个基本更新就是待生效队列的第一个更新
+      firstBaseUpdate = firstPendingUpdate;
+    } else {
+      // 否则把待生效更新队列添加到基本更新的尾部
+      lastBaseUpdate.next = firstPendingUpdate;
+    }
+    // 最后一个基本更新肯定就是最后一个待生效的更新
+    lastBaseUpdate = lastPendingUpdate;
+    /**  合并新旧链表结束  */
+  }
+
+  // 如果有更新
+  if (firstBaseUpdate !== null) {
+    // 基本状态
+    let newState = queue.baseState;
+    // 新的车道
+    let newLanes = NoLanes;
+    // 新的基本状态
+    let newBaseState = null;
+    // 新的第一个基本更新
+    let newFirstBaseUpdate = null;
+    // 新的最后一个基本更新
+    let newLastBaseUpdate = null;
+    // 第一个更新
+    let update = firstBaseUpdate;
+    do {
+      const updateLane = update.lane;
+      const shouldSkipUpdate = !isSubsetOfLanes(workInProgressRootRenderLanes, updateLane);
+      // 判断优先级是否足够,如果不够就跳过此更新
+      if (shouldSkipUpdate) {
+        // 复制一个新的更新并添加新的基本链表中
+        const clone = {
+          lane: updateLane,
+          tag: update.tag,
+          payload: update.payload,
+          next: null
+        };
+        if (newLastBaseUpdate === null) {
+          newFirstBaseUpdate = newLastBaseUpdate = clone;
+          newBaseState = newState;
+        } else {
+          newLastBaseUpdate = newLastBaseUpdate.next = clone;
+        }
+        // 保存此fiber上还剩下的更新车道
+        newLanes = mergeLanes(newLanes, updateLane);
+      } else {
+        // 如果已经有跳过的更新了，即使优先级足够也需要添到新的基本链表中
+        if (newLastBaseUpdate !== null) {
+          const clone = {
+            lane: NoLane,
+            tag: update.tag,
+            payload: update.payload,
+            next: null
+          };
+          newLastBaseUpdate = newLastBaseUpdate.next = clone;
+        }
+        // 根据更新计算新状态
+        newState = getStateFromUpdate(update, newState, props);
+        update = update.next;
+      }
+    } while (update);
+    // 如果没有跳过的更新
+    if (newLastBaseUpdate === null) {
+      newBaseState = newState;
```

```
+    }
+    queue.baseState = newBaseState;
+    queue.firstBaseUpdate = newFirstBaseUpdate;
+    queue.lastBaseUpdate = newLastBaseUpdate;
+    workInProgress.lanes = newLanes;
+    workInProgress.memoizedState = newState;
+  }
+}

+export function cloneUpdateQueue(current, workInProgress) {
+  const queue = workInProgress.updateQueue;
+  const currentQueue = current.updateQueue;
+  if (queue === currentQueue) {
+    const clone = {
+      baseState: currentQueue.baseState,
+      firstBaseUpdate: currentQueue.firstBaseUpdate,
+      lastBaseUpdate: currentQueue.lastBaseUpdate,
+      shared: currentQueue.shared,
+    };
+    workInProgress.updateQueue = clone;
+  }
+}
```

36.4 ReactFiberConcurrentUpdates.js

src\react-reconciler\src\ReactFiberConcurrentUpdates.js

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

const concurrentQueues = [];
let concurrentQueuesIndex = 0;

export function markUpdateLaneFromFiberToRoot(sourceFiber) {
  let node = sourceFiber;
  let parent = sourceFiber.return;
  while (parent !== null) {
    node = parent;
    parent = parent.return;
  }
  if (node.tag
    const root = node.stateNode;
    return root;
  }
  return null;
}
export function enqueueConcurrentHookUpdate(fiber, queue, update) {
  enqueueUpdate(fiber, queue, update);
  return getRootForUpdatedFiber(fiber);
}
+export function enqueueConcurrentClassUpdate(fiber, queue, update, lane) {
+  enqueueUpdate(fiber, queue, update, lane);
+  return getRootForUpdatedFiber(fiber);
+}

+function enqueueUpdate(fiber, queue, update, lane) {
  concurrentQueues[concurrentQueuesIndex++] = fiber;
  concurrentQueues[concurrentQueuesIndex++] = queue;
  concurrentQueues[concurrentQueuesIndex++] = update;
+ concurrentQueues[concurrentQueuesIndex++] = lane;
}
function getRootForUpdatedFiber(sourceFiber) {
  let node = sourceFiber;
  let parent = node.return;
  while (parent !== null) {
    node = parent;
    parent = node.return;
  }
  return node.tag
}

export function finishQueueingConcurrentUpdates() {
  const endIndex = concurrentQueuesIndex;
  concurrentQueuesIndex = 0;
  let i = 0;
  while (i < endIndex) {
    const fiber = concurrentQueues[i++];
    const queue = concurrentQueues[i++];
    const update = concurrentQueues[i++];
+   const lane = concurrentQueues[i++]
    if (queue !== null && update !== null) {
      const pending = queue.pending;
      if (pending
        update.next = update;
      } else {
        update.next = pending.next;
        pending.next = update;
      }
      queue.pending = update;
    }
  }
}
```

36.5 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
+import {
+  scheduleCallback as Scheduler_scheduleCallback,
+  ImmediatePriority as ImmediateSchedulerPriority,
+  UserBlockingPriority as UserBlockingSchedulerPriority,
+  NormalPriority as NormalSchedulerPriority,
+  IdlePriority as IdleSchedulerPriority,
+} from "./Scheduler";
import { createWorkInProgress } from "./ReactFiber";
```

```
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
+import { MutationMask, NoFlags, Passive } from "./ReactFiberFlags";
import {
  commitMutationEffects,
  commitPassiveUnmountEffects,
  commitPassiveMountEffects,
  commitLayoutEffects,
} from "./ReactFiberCommitWork";
import { finishQueueingConcurrentUpdates } from "./ReactFiberConcurrentUpdates";
+import {
+  NoLane, markRootUpdated, NoLanes,
+  getNextLanes, getHighestPriorityLane, SyncLane,
+  includesBlockingLane
+} from './ReactFiberLane';
+import {
+  getCurrentUpdatePriority, lanesToEventPriority, DiscreteEventPriority, ContinuousEventPriority,
+  DefaultEventPriority, IdleEventPriority,
+} from './ReactEventPriorities';
+import { getCurrentEventPriority } from 'react-dom-bindings/src/client/ReactDOMHostConfig';

let workInProgress = null;
let rootDoesHavePassiveEffects = false;
let rootWithPendingPassiveEffects = null;
+let workInProgressRootRenderLanes = NoLanes;

+export function scheduleUpdateOnFiber(root, fiber, lane) {
+ markRootUpdated(root, lane);
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
+ const nextLanes = getNextLanes(root, NoLanes);
+ const newCallbackPriority = getHighestPriorityLane(nextLanes);
+ if (newCallbackPriority === SyncLane) {
+   // TODO
+ } else {
+   let schedulerPriorityLevel;
+   switch (lanesToEventPriority(nextLanes)) {
+     case DiscreteEventPriority:
+       schedulerPriorityLevel = ImmediateSchedulerPriority;
+       break;
+     case ContinuousEventPriority:
+       schedulerPriorityLevel = UserBlockingSchedulerPriority;
+       break;
+     case DefaultEventPriority:
+       schedulerPriorityLevel = NormalSchedulerPriority;
+       break;
+     case IdleEventPriority:
+       schedulerPriorityLevel = IdleSchedulerPriority;
+       break;
+     default:
+       schedulerPriorityLevel = NormalSchedulerPriority;
+       break;
+   }
+   Scheduler_scheduleCallback(schedulerPriorityLevel, performConcurrentWorkOnRoot.bind(null, root))
  }
}
+function performConcurrentWorkOnRoot(root,didTimeout) {
+ const lanes = getNextLanes(root, NoLanes);
+ if (lanes === NoLanes) {
+   return null;
+ }
+ const shouldTimeSlice = !includesBlockingLane(root, lanes) && (!didTimeout);
+ if (shouldTimeSlice) {
+   renderRootConcurrent(root, lanes)
+ } else {
+   renderRootSync(root, lanes);
+ }
  const finishedWork = root.current.alternate;
  root.finishedWork = finishedWork;
  commitRoot(root);
}
+function renderRootConcurrent(root, lanes) {
+  console.log(root, lanes);
+}
export function flushPassiveEffects() {
  if (rootWithPendingPassiveEffects !== null) {
    const root = rootWithPendingPassiveEffects;
    commitPassiveUnmountEffects(root.current);
    commitPassiveMountEffects(root, root.current);
  }
}
function commitRoot(root) {
  const { finishedWork } = root;
  if ((finishedWork.subtreeFlags & Passive) !== NoFlags || (finishedWork.flags & Passive) !== NoFlags) {
    if (!rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = true;
      Scheduler_scheduleCallback(NormalSchedulerPriority, flushPassiveEffects);
    }
  }
  const subtreeHasEffects = (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
    commitMutationEffects(finishedWork, root);
    commitLayoutEffects(finishedWork, root);
    root.current = finishedWork;
    if (rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = false;
      rootWithPendingPassiveEffects = root;
    }
  }
  root.current = finishedWork;
}
+function prepareFreshStack(root, lanes) {
```

```
  workInProgress = createWorkInProgress(root.current, null);
+ workInProgressRootRenderLanes = lanes;
  finishQueueingConcurrentUpdates();
}
+function renderRootSync(root, lanes) {
+ prepareFreshStack(root, lanes);
  workLoopSync();
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
+ const next = beginWork(current, unitOfWork, workInProgressRootRenderLanes);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
}

+export function requestUpdateLane() {
+  const updateLane = getCurrentUpdatePriority();
+  if (updateLane !== NoLane) {
+    return updateLane;
+  }
+  const eventLane = getCurrentEventPriority();
+  return eventLane;
+}
```

36.6 ReactFiberBeginWork.js

src\react-reconciler\src\ReactFiberBeginWork.js

```
import { HostRoot, HostComponent, HostText, IndeterminateComponent, FunctionComponent } from "./ReactWorkTags";
+import { processUpdateQueue, cloneUpdateQueue } from "./ReactFiberClassUpdateQueue";
import { mountChildFibers, reconcileChildFibers } from "./ReactChildFiber";
import { shouldSetTextContent } from "react-dom-bindings/src/client/ReactDOMHostConfig";
import { renderWithHooks } from "react-reconciler/src/ReactFiberHooks";

function reconcileChildren(current, workInProgress, nextChildren) {
  if (current
    workInProgress.child = mountChildFibers(workInProgress, null, nextChildren);
  } else {
    workInProgress.child = reconcileChildFibers(workInProgress, current.child, nextChildren);
  }
}
+function updateHostRoot(current, workInProgress, renderLanes) {
+ const nextProps = workInProgress.pendingProps;
+ cloneUpdateQueue(current, workInProgress);
+ processUpdateQueue(workInProgress, nextProps, renderLanes)
  const nextState = workInProgress.memoizedState;
  const nextChildren = nextState.element;
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
function updateHostComponent(current, workInProgress) {
  const { type } = workInProgress;
  const nextProps = workInProgress.pendingProps;
  let nextChildren = nextProps.children;
  const isDirectTextChild = shouldSetTextContent(type, nextProps);
  if (isDirectTextChild) {
    nextChildren = null;
  }
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
function mountIndeterminateComponent(_current, workInProgress, Component) {
  const props = workInProgress.pendingProps;
  const value = renderWithHooks(null, workInProgress, Component, props);
  workInProgress.tag = FunctionComponent;
  reconcileChildren(null, workInProgress, value);
  return workInProgress.child;
}
function updateFunctionComponent(current, workInProgress, Component, nextProps,workInProgressRootRenderLanes) {
  const nextChildren = renderWithHooks(current, workInProgress, Component, nextProps,workInProgressRootRenderLanes);
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
+export function beginWork(current, workInProgress, renderLanes) {
  switch (workInProgress.tag) {
    case IndeterminateComponent: {
+     return mountIndeterminateComponent(current, workInProgress, workInProgress.type, renderLanes);
    }
    case FunctionComponent: {
      const Component = workInProgress.type;
      const resolvedProps = workInProgress.pendingProps;
+     return updateFunctionComponent(current, workInProgress, Component, resolvedProps, renderLanes);
    }
    case HostRoot:
+     return updateHostRoot(current, workInProgress, renderLanes);
    case HostComponent:
+     return updateHostComponent(current, workInProgress, renderLanes);
    case HostText:
    default:
      return null;
  }
}
```

36.7 ReactFiberLane.js

src\react-reconciler\src\ReactFiberLane.js

```
export const TotalLanes = 31;
export const NoLanes = 0b0000000000000000000000000000000;
export const NoLane = 0b0000000000000000000000000000000;
export const SyncLane = 0b0000000000000000000000000000001;
export const InputContinuousLane = 0b0000000000000000000000000000100;
export const DefaultLane = 0b0000000000000000000000000010000;
export const NonIdleLanes = 0b0001111111111111111111111111111;
export const IdleLane = 0b0100000000000000000000000000000;

export function mergeLanes(a, b) {
  return a | b;
}
export function markRootUpdated(root, updateLane) {
  root.pendingLanes |= updateLane;
}

export function getNextLanes(root) {
  const pendingLanes = root.pendingLanes;
  if (pendingLanes === NoLanes) {
    return NoLanes;
  }
  const nextLanes = getHighestPriorityLanes(pendingLanes);
  return nextLanes;
}

function getHighestPriorityLanes(lanes) {
  return getHighestPriorityLane(lanes);
}

export function getHighestPriorityLane(lanes) {
  return lanes & -lanes;
}

export function includesNonIdleWork(lanes) {
  return (lanes & NonIdleLanes) !== NoLanes;
}
export function includesBlockingLane(root, lanes) {
  const SyncDefaultLanes = InputContinuousLane | DefaultLane;
  return (lanes & SyncDefaultLanes) !== NoLanes;
}
export function isSubsetOfLanes(set, subset) {
  return (set & subset) === subset;
}
```

36.8 ReactEventPriorities.js

src\react-reconciler\src\ReactEventPriorities.js

```
import {
  NoLane, DefaultLane, getHighestPriorityLane,
  includesNonIdleWork, SyncLane, InputContinuousLane, IdleLane
} from './ReactFiberLane';

export const DefaultEventPriority = DefaultLane;
export const DiscreteEventPriority = SyncLane;
export const ContinuousEventPriority = InputContinuousLane;
export const IdleEventPriority = IdleLane;

let currentUpdatePriority = NoLane;

export function getCurrentUpdatePriority() {
  return currentUpdatePriority;
}
export function setCurrentUpdatePriority(newPriority) {
  currentUpdatePriority = newPriority;
}
export function isHigherEventPriority(a, b) {
  return a !== 0 && a < b;
}
export function lanesToEventPriority(lanes) {
  const lane = getHighestPriorityLane(lanes);
  if (!isHigherEventPriority(DiscreteEventPriority, lane)) {
    return DiscreteEventPriority;
  }
  if (!isHigherEventPriority(ContinuousEventPriority, lane)) {
    return ContinuousEventPriority;
  }
  if (includesNonIdleWork(lane)) {
    return DefaultEventPriority;
  }
  return IdleEventPriority;
}
```

36.9 ReactDOMHostConfig.js

src\react-dom-bindings\src\client\ReactDOMHostConfig.js

```
import { setInitialProperties, diffProperties, updateProperties } from "./ReactDOMComponent";
import { precacheFiberNode, updateFiberProps } from "./ReactDOMComponentTree";
+import { getEventPriority } from '../events/ReactDOMEventListener';
+import { DefaultEventPriority } from 'react-reconciler/src/ReactEventPriorities';

export function shouldSetTextContent(type, props) {
  return typeof props.children
}
export const appendInitialChild = (parent, child) => {
  parent.appendChild(child);
};
export const createInstance = (type, props, internalInstanceHandle) => {
  const domElement = document.createElement(type);
  precacheFiberNode(internalInstanceHandle, domElement);
  updateFiberProps(domElement, props);
  return domElement;
};
export const createTextInstance = (content) => document.createTextNode(content);
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);
}

export function prepareUpdate(domElement, type, oldProps, newProps) {
  return diffProperties(domElement, type, oldProps, newProps);
}

export function commitUpdate(domElement, updatePayload, type, oldProps, newProps) {
  updateProperties(domElement, updatePayload, type, oldProps, newProps);
  updateFiberProps(domElement, newProps);
}
export function removeChild(parentInstance, child) {
  parentInstance.removeChild(child);
}
+export function getCurrentEventPriority() {
+  const currentEvent = window.event;
+  if (currentEvent === undefined) {
+    return DefaultEventPriority;
+  }
+  return getEventPriority(currentEvent.type);
+}
```

36.10 ReactDOMEventListener.js

src\react-dom-bindings\src\events\ReactDOMEventListener.js

```
import getEventTarget from "./getEventTarget";
import { getClosestInstanceFromNode } from "../client/ReactDOMComponentTree";
import { dispatchEventForPluginEventSystem } from "./DOMPluginEventSystem";
+import { DiscreteEventPriority, ContinuousEventPriority, DefaultEventPriority } from 'react-reconciler/src/ReactEventPriorities';

export function createEventListenerWrapperWithPriority(
  targetContainer,
  domEventName,
  eventSystemFlags
) {
  const listenerWrapper = dispatchDiscreteEvent;
  return listenerWrapper.bind(null, domEventName, eventSystemFlags, targetContainer);
}
function dispatchDiscreteEvent(domEventName, eventSystemFlags, container, nativeEvent) {
  dispatchEvent(domEventName, eventSystemFlags, container, nativeEvent);
}
export function dispatchEvent(domEventName, eventSystemFlags, targetContainer, nativeEvent) {
  const nativeEventTarget = getEventTarget(nativeEvent);
  const targetInst = getClosestInstanceFromNode(nativeEventTarget);
  dispatchEventForPluginEventSystem(
    domEventName,
    eventSystemFlags,
    nativeEvent,
    targetInst,
    targetContainer
  );
}
+export function getEventPriority(domEventName) {
+  switch (domEventName) {
+    case 'click':
+      return DiscreteEventPriority;
+    case 'drag':
+      return ContinuousEventPriority;
+    default:
+      return DefaultEventPriority;
+  }
+}
```

36.11 Scheduler.js

src\react-reconciler\src\Scheduler.js

```
import * as Scheduler from 'scheduler'

export const scheduleCallback = Scheduler.unstable_scheduleCallback
export const NormalPriority = Scheduler.unstable_NormalPriority
+export const ImmediatePriority = Scheduler.unstable_ImmediatePriority;
+export const UserBlockingPriority = Scheduler.unstable_UserBlockingPriority;
+export const LowPriority = Scheduler.unstable_LowPriority;
+export const IdlePriority = Scheduler.unstable_IdlePriority;
```

37.更新渲染

37.1 main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";
+function FunctionComponent() {
+  const [number, setNumber] = React.useState(0);
+  return  {
+    setNumber(number + 1)
+  }}>{number}
+}
const element = ;
const container = document.getElementById("root");
const root = createRoot(container);
root.render(element);
```

37.2 ReactFiberHooks.js

src\react-reconciler\src\ReactFiberHooks.js

```
import ReactSharedInternals from "shared/ReactSharedInternals";
import { enqueueConcurrentHookUpdate } from "./ReactFiberConcurrentUpdates";
+import { scheduleUpdateOnFiber, requestUpdateLane } from "./ReactFiberWorkLoop";
import is from "shared/objectIs";
import { Passive as PassiveEffect, Update as UpdateEffect } from "./ReactFiberFlags";
import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout } from "./ReactHookEffectTags";

const { ReactCurrentDispatcher } = ReactSharedInternals;
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;

const HooksDispatcherOnMountInDEV = {
  useReducer: mountReducer,
  useState: mountState,
  useEffect: mountEffect,
  useLayoutEffect: mountLayoutEffect,
};
const HooksDispatcherOnUpdateInDEV = {
  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) {
  const fiberFlags = UpdateEffect;
  return mountEffectImpl(fiberFlags, HookLayout, create, deps);
}
function updateEffect(create, deps) {
  return updateEffectImpl(PassiveEffect, HookPassive, create, deps);
}
function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
  const hook = updateWorkInProgressHook();
  const nextDeps = deps
  let destroy;
  if (currentHook !== null) {
    const prevEffect = currentHook.memoizedState;
    destroy = prevEffect.destroy;
    if (nextDeps !== null) {
      const prevDeps = prevEffect.deps;
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        hook.memoizedState = 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 (is(nextDeps[i], prevDeps[i])) {
      continue;
    }
    return false;
  }

  return true;
}
function mountEffect(create, deps) {
  return mountEffectImpl(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,
  };
}
function basicStateReducer(state, action) {
  return typeof action
}
function mountReducer(reducer, initialArg) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialArg;
  const queue = {
    pending: null,
    dispatch: null,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchReducerAction.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function updateReducer(reducer) {
  const hook = updateWorkInProgressHook();
  const queue = hook.queue;
  queue.lastRenderedReducer = reducer;
  const current = currentHook;
  const pendingQueue = queue.pending;
  let baseQueue = null;
  let newState = current.memoizedState;
  if (pendingQueue !== null) {
    baseQueue = pendingQueue;
    queue.pending = null;
  }
  if (baseQueue !== null) {
    const first = baseQueue.next;
    let update = first;
    do {
      if (update.hasEagerState) {
        newState = update.eagerState;
      } else {
        const action = update.action;
        newState = reducer(newState, action);
      }
      update = update.next;
    } while (update !== null && update !== first);
  }
  hook.memoizedState = newState;
  queue.lastRenderedState = newState;
  const dispatch = queue.dispatch;
  return [hook.memoizedState, dispatch];
}
function mountState(initialState) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialState;
  const queue = {
    pending: null,
    dispatch: null,
    lastRenderedReducer: basicStateReducer,
    lastRenderedState: initialState,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchSetState.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function dispatchSetState(fiber, queue, action) {
+ const lane = requestUpdateLane(fiber);
  const update = {
+   lane,
    action,
    hasEagerState: false,
    eagerState: null,
    next: null,
  };
  const lastRenderedReducer = queue.lastRenderedReducer;
  const currentState = queue.lastRenderedState;
  const eagerState = lastRenderedReducer(currentState, action);
  update.hasEagerState = true;
  update.eagerState = eagerState;
  if (is(eagerState, currentState)) {
    return;
  }
+ const root = enqueueConcurrentHookUpdate(fiber, queue, update, lane);
+ scheduleUpdateOnFiber(root, fiber, lane);
}
function updateState(initialState) {
  return updateReducer(basicStateReducer, initialState);
```

```
}
function mountWorkInProgressHook() {
  const hook = {
    memoizedState: null,
    queue: null,
    next: null,
  };
  if (workInProgressHook
    currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
  } else {
    workInProgressHook = workInProgressHook.next = hook;
  }
  return workInProgressHook;
}
function dispatchReducerAction(fiber, queue, action) {
  const update = {
    action,
    next: null,
  };
  const root = enqueueConcurrentHookUpdate(fiber, queue, update);
  scheduleUpdateOnFiber(root, fiber);
}

function updateWorkInProgressHook() {
  let nextCurrentHook;
  if (currentHook
    const current = currentlyRenderingFiber.alternate;
    if (current !== null) {
      nextCurrentHook = current.memoizedState;
    } else {
      nextCurrentHook = null;
    }
  } else {
    nextCurrentHook = currentHook.next;
  }

  let nextWorkInProgressHook;
  if (workInProgressHook
    nextWorkInProgressHook = currentlyRenderingFiber.memoizedState;
  } else {
    nextWorkInProgressHook = workInProgressHook.next;
  }

  if (nextWorkInProgressHook !== null) {
    workInProgressHook = nextWorkInProgressHook;
    nextWorkInProgressHook = workInProgressHook.next;
    currentHook = nextCurrentHook;
  } else {
    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 renderWithHooks(current, workInProgress, Component, props) {
  currentlyRenderingFiber = workInProgress;
  workInProgress.updateQueue = null;
  workInProgress.memoizedState = null;
  if (current !== null && current.memoizedState !== null) {
    ReactCurrentDispatcher.current = HooksDispatcherOnUpdateInDEV;
  } else {
    ReactCurrentDispatcher.current = HooksDispatcherOnMountInDEV;
  }
  const children = Component(props);
  currentlyRenderingFiber = null;
  workInProgressHook = null;
  currentHook = null;
  return children;
}
```

37.3 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import {
  scheduleCallback as Scheduler_scheduleCallback,
  ImmediatePriority as ImmediateSchedulerPriority,
  UserBlockingPriority as UserBlockingSchedulerPriority,
  NormalPriority as NormalSchedulerPriority,
  IdlePriority as IdleSchedulerPriority,
} from "./Scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
import { MutationMask, NoFlags, Passive } from "./ReactFiberFlags";
import {
  commitMutationEffects,
  commitPassiveUnmountEffects,
  commitPassiveMountEffects,
  commitLayoutEffects,
} from "./ReactFiberCommitWork";
import { finishQueueingConcurrentUpdates } from "./ReactFiberConcurrentUpdates";
import {
  NoLane, markRootUpdated, NoLanes,
  getNextLanes, getHighestPriorityLane, SyncLane,
  includesBlockingLane
```

```
} from './ReactFiberLane';
import {
  getCurrentUpdatePriority, lanesToEventPriority, DiscreteEventPriority, ContinuousEventPriority,
+ DefaultEventPriority, IdleEventPriority, setCurrentUpdatePriority
} from './ReactEventPriorities';
import { getCurrentEventPriority } from 'react-dom-bindings/src/client/ReactDOMHostConfig';
+import { scheduleSyncCallback, flushSyncCallbacks } from './ReactFiberSyncTaskQueue';

let workInProgress = null;
let rootDoesHavePassiveEffects = false;
let rootWithPendingPassiveEffects = null;
let workInProgressRootRenderLanes = NoLanes;

export function scheduleUpdateOnFiber(root, fiber, lane) {
  markRootUpdated(root, lane);
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  const nextLanes = getNextLanes(root, NoLanes);
  const newCallbackPriority = getHighestPriorityLane(nextLanes);
  if (newCallbackPriority
+   scheduleSyncCallback(performSyncWorkOnRoot.bind(null, root));
+   queueMicrotask(flushSyncCallbacks);
  } else {
    let schedulerPriorityLevel;
    switch (lanesToEventPriority(nextLanes)) {
      case DiscreteEventPriority:
        schedulerPriorityLevel = ImmediateSchedulerPriority;
        break;
      case ContinuousEventPriority:
        schedulerPriorityLevel = UserBlockingSchedulerPriority;
        break;
      case DefaultEventPriority:
        schedulerPriorityLevel = NormalSchedulerPriority;
        break;
      case IdleEventPriority:
        schedulerPriorityLevel = IdleSchedulerPriority;
        break;
      default:
        schedulerPriorityLevel = NormalSchedulerPriority;
        break;
    }
    Scheduler_scheduleCallback(schedulerPriorityLevel, performConcurrentWorkOnRoot.bind(null, root))
  }
}
+function performSyncWorkOnRoot(root) {
+  const lanes = getNextLanes(root, NoLanes);
+  renderRootSync(root, lanes);
+  const finishedWork = root.current.alternate
+  root.finishedWork = finishedWork
+  commitRoot(root)
+  return null;//如果没有任务了一定要返回null
+}
function performConcurrentWorkOnRoot(root) {
  const lanes = getNextLanes(root, NoLanes);
  if (lanes
    return null;
  }
  const shouldTimeSlice = !includesBlockingLane(root, lanes) && (!didTimeout);
  if (shouldTimeSlice) {
    renderRootConcurrent(root, lanes)
  } else {
    renderRootSync(root, lanes);
  }
  const finishedWork = root.current.alternate;
  root.finishedWork = finishedWork;
  commitRoot(root);
}
function renderRootConcurrent(root, lanes) {
  console.log(root, lanes);
}
export function flushPassiveEffects() {
  if (rootWithPendingPassiveEffects !== null) {
    const root = rootWithPendingPassiveEffects;
    commitPassiveUnmountEffects(root.current);
    commitPassiveMountEffects(root, root.current);
  }
}
+function commitRoot(root) {
+  const previousPriority = getCurrentUpdatePriority();
+  try {
+    setCurrentUpdatePriority(DiscreteEventPriority);
+    commitRootImpl(root);
+  } finally {
+    setCurrentUpdatePriority(previousPriority);
+  }
+}
+function commitRootImpl(root) {
  const { finishedWork } = root;
  if ((finishedWork.subtreeFlags & Passive) !== NoFlags || (finishedWork.flags & Passive) !== NoFlags) {
    if (!rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = true;
      Scheduler_scheduleCallback(NormalSchedulerPriority, flushPassiveEffects);
    }
  }
  const subtreeHasEffects = (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
    commitMutationEffects(finishedWork, root);
    commitLayoutEffects(finishedWork, root);
    root.current = finishedWork;
    if (rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = false;
      rootWithPendingPassiveEffects = root;
```

```
    }
  }
  root.current = finishedWork;
+}
function prepareFreshStack(root, lanes) {
  workInProgress = createWorkInProgress(root.current, null);
  workInProgressRootRenderLanes = lanes;
  finishQueueingConcurrentUpdates();
}
function renderRootSync(root, lanes) {
  prepareFreshStack(root, lanes);
  workLoopSync();
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork, workInProgressRootRenderLanes);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
}

export function requestUpdateLane() {
  const updateLane = getCurrentUpdatePriority();
  if (updateLane !== NoLane) {
    return updateLane;
  }
  const eventLane = getCurrentEventPriority();
  return eventLane;
}
```

37.4 ReactFiberSyncTaskQueue.js

src\react-reconciler\src\ReactFiberSyncTaskQueue.js

```
import { DiscreteEventPriority, getCurrentUpdatePriority, setCurrentUpdatePriority } from './ReactEventPriorities';

let syncQueue = null;
let isFlushingSyncQueue = false;

export function scheduleSyncCallback(callback) {
  if (syncQueue === null) {
    syncQueue = [callback];
  } else {
    syncQueue.push(callback);
  }
}

export function flushSyncCallbacks() {
  if (!isFlushingSyncQueue && syncQueue !== null) {
    isFlushingSyncQueue = true;
    let i = 0;
    const previousUpdatePriority = getCurrentUpdatePriority();
    try {
      const isSync = true;
      const queue = syncQueue;
      setCurrentUpdatePriority(DiscreteEventPriority);
      for (; i < queue.length; i++) {
        let callback = queue[i];
        do {
          callback = callback(isSync);
        } while (callback !== null);
      }
      syncQueue = null;
    } finally {
      setCurrentUpdatePriority(previousUpdatePriority);
      isFlushingSyncQueue = false;
    }
  }
  return null;
}
```

37.5 ReactFiberConcurrentUpdates.js

src\react-reconciler\src\ReactFiberConcurrentUpdates.js

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

const concurrentQueues = [];
let concurrentQueuesIndex = 0;

export function markUpdateLaneFromFiberToRoot(sourceFiber) {
  let node = sourceFiber;
  let parent = sourceFiber.return;
  while (parent !== null) {
    node = parent;
    parent = parent.return;
  }
  if (node.tag
    const root = node.stateNode;
    return root;
  }
  return null;
}
+export function enqueueConcurrentHookUpdate(fiber, queue, update, lane) {
+ enqueueUpdate(fiber, queue, update, lane);
  return getRootForUpdatedFiber(fiber);
}
export function enqueueConcurrentClassUpdate(fiber, queue, update, lane) {
  enqueueUpdate(fiber, queue, update, lane);
  return getRootForUpdatedFiber(fiber);
}

function enqueueUpdate(fiber, queue, update, lane) {
  concurrentQueues[concurrentQueuesIndex++] = fiber;
  concurrentQueues[concurrentQueuesIndex++] = queue;
  concurrentQueues[concurrentQueuesIndex++] = update;
  concurrentQueues[concurrentQueuesIndex++] = lane;
}
function getRootForUpdatedFiber(sourceFiber) {
  let node = sourceFiber;
  let parent = node.return;
  while (parent !== null) {
    node = parent;
    parent = node.return;
  }
  return node.tag
}

export function finishQueueingConcurrentUpdates() {
  const endIndex = concurrentQueuesIndex;
  concurrentQueuesIndex = 0;
  let i = 0;
  while (i < endIndex) {
    const fiber = concurrentQueues[i++];
    const queue = concurrentQueues[i++];
    const update = concurrentQueues[i++];
    const lane = concurrentQueues[i++]
    if (queue !== null && update !== null) {
      const pending = queue.pending;
      if (pending
        update.next = update;
      } else {
        update.next = pending.next;
        pending.next = update;
      }
      queue.pending = update;
    }
  }
}
```

37.6 ReactDOMEventListener.js

src\react-dom-bindings\src\events\ReactDOMEventListener.js

```
import getEventTarget from "./getEventTarget";
import { getClosestInstanceFromNode } from "../client/ReactDOMComponentTree";
import { dispatchEventForPluginEventSystem } from "./DOMPluginEventSystem";
import {
  DiscreteEventPriority, ContinuousEventPriority, DefaultEventPriority,
+ getCurrentUpdatePriority, setCurrentUpdatePriority
} from 'react-reconciler/src/ReactEventPriorities';

export function createEventListenerWrapperWithPriority(
  targetContainer,
  domEventName,
  eventSystemFlags
) {
  const listenerWrapper = dispatchDiscreteEvent;
  return listenerWrapper.bind(null, domEventName, eventSystemFlags, targetContainer);
}
+function dispatchDiscreteEvent(domEventName, eventSystemFlags, container, nativeEvent) {
+  const previousPriority = getCurrentUpdatePriority();
+  try {
+    setCurrentUpdatePriority(DiscreteEventPriority);
     dispatchEvent(domEventName, eventSystemFlags, container, nativeEvent)
+  } finally {
+    setCurrentUpdatePriority(previousPriority);
+  }
+}
export function dispatchEvent(domEventName, eventSystemFlags, targetContainer, nativeEvent) {
  const nativeEventTarget = getEventTarget(nativeEvent);
  const targetInst = getClosestInstanceFromNode(nativeEventTarget);
  dispatchEventForPluginEventSystem(
    domEventName,
    eventSystemFlags,
    nativeEvent,
    targetInst,
    targetContainer
  );
}

export function getEventPriority(domEventName) {
  switch (domEventName) {
    case 'click':
      return DiscreteEventPriority;
    case 'drag':
      return ContinuousEventPriority;
    default:
      return DefaultEventPriority;
  }
}
```

38.并发渲染

38.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

+function FunctionComponent() {
+  console.log('FunctionComponent');
+  const [number, setNumber] = React.useState(0);
+  React.useEffect(() => {
+    setNumber(number => number + 1)
+  }, []);
+  return ( setNumber(number + 1)}>{number})
+}
const element = ;
const container = document.getElementById("root");
const root = createRoot(container, { unstable_concurrentUpdatesByDefault: true });
root.render(element);
```

38.2 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import {
  scheduleCallback as Scheduler_scheduleCallback,
  ImmediatePriority as ImmediateSchedulerPriority,
  UserBlockingPriority as UserBlockingSchedulerPriority,
  NormalPriority as NormalSchedulerPriority,
  IdlePriority as IdleSchedulerPriority,
+ shouldYield
} from "./Scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
import { MutationMask, NoFlags, Passive } from "./ReactFiberFlags";
import {
  commitMutationEffects,
  commitPassiveUnmountEffects,
  commitPassiveMountEffects,
  commitLayoutEffects,
} from "./ReactFiberCommitWork";
import { finishQueueingConcurrentUpdates } from "./ReactFiberConcurrentUpdates";
import {
  NoLane, markRootUpdated, NoLanes,
  getNextLanes, getHighestPriorityLane, SyncLane,
  includesBlockingLane
} from './ReactFiberLane';
import {
  getCurrentUpdatePriority, lanesToEventPriority, DiscreteEventPriority, ContinuousEventPriority,
  DefaultEventPriority, IdleEventPriority, setCurrentUpdatePriority
} from './ReactEventPriorities';
import { getCurrentEventPriority } from 'react-dom-bindings/src/client/ReactDOMHostConfig';
import { scheduleSyncCallback, flushSyncCallbacks } from './ReactFiberSyncTaskQueue';
```

```
let workInProgress = null;
let rootDoesHavePassiveEffects = false;
let rootWithPendingPassiveEffects = null;
let workInProgressRootRenderLanes = NoLanes;

+const RootInProgress = 0;
+const RootCompleted = 5;
+let workInProgressRoot = null;
+let workInProgressRootExitStatus = RootInProgress;

export function scheduleUpdateOnFiber(root, fiber, lane) {
  markRootUpdated(root, lane);
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  const nextLanes = getNextLanes(root, NoLanes);
+ if (nextLanes === NoLanes) {
+   root.callbackNode = null;
+   root.callbackPriority = NoLane;
+   return;
+ }
  const newCallbackPriority = getHighestPriorityLane(nextLanes);
+ let newCallbackNode;
  if (newCallbackPriority
    scheduleSyncCallback(performSyncWorkOnRoot.bind(null, root));
    queueMicrotask(flushSyncCallbacks);
+   newCallbackNode = null;
  } else {
    let schedulerPriorityLevel;
    switch (lanesToEventPriority(nextLanes)) {
      case DiscreteEventPriority:
        schedulerPriorityLevel = ImmediateSchedulerPriority;
        break;
      case ContinuousEventPriority:
        schedulerPriorityLevel = UserBlockingSchedulerPriority;
        break;
      case DefaultEventPriority:
        schedulerPriorityLevel = NormalSchedulerPriority;
        break;
      case IdleEventPriority:
        schedulerPriorityLevel = IdleSchedulerPriority;
        break;
      default:
        schedulerPriorityLevel = NormalSchedulerPriority;
        break;
    }
+   newCallbackNode = Scheduler_scheduleCallback(schedulerPriorityLevel, performConcurrentWorkOnRoot.bind(null, root))
  }
+ root.callbackNode = newCallbackNode;
}
function performSyncWorkOnRoot(root) {
  const lanes = getNextLanes(root, NoLanes);
  renderRootSync(root, lanes);
  const finishedWork = root.current.alternate
  root.finishedWork = finishedWork
  commitRoot(root)
  return null;
}
function performConcurrentWorkOnRoot(root, didTimeout) {
+ const originalCallbackNode = root.callbackNode;
  const lanes = getNextLanes(root, NoLanes);
  if (lanes
    return null;
  }
  const shouldTimeSlice = !includesBlockingLane(root, lanes) && (!didTimeout);
+ const exitStatus = shouldTimeSlice ? renderRootConcurrent(root, lanes) : renderRootSync(root, lanes);
+ if (exitStatus !== RootInProgress) {
+   const finishedWork = root.current.alternate
+   root.finishedWork = finishedWork
+   commitRoot(root)
+ }
+ if (root.callbackNode === originalCallbackNode) {
+   return performConcurrentWorkOnRoot.bind(null, root);
+ }
+ return null;
}
function renderRootConcurrent(root, lanes) {
+ if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
+   prepareFreshStack(root, lanes);
+ }
+ workLoopConcurrent();
+ if (workInProgress !== null) {
+   return RootInProgress;
+ }
+ workInProgressRoot = null;
+ workInProgressRootRenderLanes = NoLanes;
+ return workInProgressRootExitStatus;
}
+function workLoopConcurrent() {
+  sleep(6);
+  performUnitOfWork(workInProgress);
+  console.log('shouldYield()', shouldYield(), workInProgress?.type);
+}
export function flushPassiveEffects() {
  if (rootWithPendingPassiveEffects !== null) {
    const root = rootWithPendingPassiveEffects;
    commitPassiveUnmountEffects(root.current);
    commitPassiveMountEffects(root, root.current);
  }
}
function commitRoot(root) {
  const previousPriority = getCurrentUpdatePriority();
  try {
```

```
    setCurrentUpdatePriority(DiscreteEventPriority);
    commitRootImpl(root);
  } finally {
    setCurrentUpdatePriority(previousPriority);
  }
}
function commitRootImpl(root) {
  const { finishedWork } = root;
+ root.callbackNode = null;
+ root.callbackPriority = NoLane;
+ workInProgressRoot = null;
+ workInProgressRootRenderLanes = NoLanes;
  if ((finishedWork.subtreeFlags & Passive) !== NoFlags || (finishedWork.flags & Passive) !== NoFlags) {
    if (!rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = true;
      Scheduler_scheduleCallback(NormalSchedulerPriority, flushPassiveEffects);
    }
  }
  const subtreeHasEffects = (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
    commitMutationEffects(finishedWork, root);
    commitLayoutEffects(finishedWork, root);
    root.current = finishedWork;
    if (rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = false;
      rootWithPendingPassiveEffects = root;
    }
  }
  root.current = finishedWork;
}
function prepareFreshStack(root, lanes) {
+ workInProgressRoot = root;
  workInProgress = createWorkInProgress(root.current, null);
  workInProgressRootRenderLanes = lanes;
  finishQueueingConcurrentUpdates();
}
function renderRootSync(root, lanes) {
+ //不是一个根，或者是更高优先级的更新
+ if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
    prepareFreshStack(root, lanes)
+ }
  workLoopSync();
+ return workInProgressRootExitStatus;
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork, workInProgressRootRenderLanes);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
+ if (workInProgressRootExitStatus === RootInProgress) {
+   workInProgressRootExitStatus = RootCompleted;
+ }
}

export function requestUpdateLane() {
  const updateLane = getCurrentUpdatePriority();
  if (updateLane !== NoLane) {
    return updateLane;
  }
  const eventLane = getCurrentEventPriority();
  return eventLane;
}

+function sleep(time) {
+  const timeStamp = new Date().getTime();
+  const endTime = timeStamp + time;
+  while (true) {
+    if (new Date().getTime() > endTime) {
+      return;
+    }
+  }
+}
```

38.3 Scheduler.js

src\react-reconciler\src\Scheduler.js

```
import * as Scheduler from 'scheduler'
export const scheduleCallback = Scheduler.unstable_scheduleCallback
export const NormalPriority = Scheduler.unstable_NormalPriority
export const ImmediatePriority = Scheduler.unstable_ImmediatePriority;
export const UserBlockingPriority = Scheduler.unstable_UserBlockingPriority;
export const LowPriority = Scheduler.unstable_LowPriority;
export const IdlePriority = Scheduler.unstable_IdlePriority;
+export const shouldYield = Scheduler.unstable_shouldYield
```

38.4 Scheduler.js

src\scheduler\src\forks\Scheduler.js

```
import {
  ImmediatePriority,
  UserBlockingPriority,
  NormalPriority,
  LowPriority,
  IdlePriority,
} from "../SchedulerPriorities";
import { push, pop, peek } from "../SchedulerMinHeap";
import { frameYieldMs } from "../SchedulerFeatureFlags";

const maxSigned31BitInt = 1073741823;
const IMMEDIATE_PRIORITY_TIMEOUT = -1;
const USER_BLOCKING_PRIORITY_TIMEOUT = 250;
const NORMAL_PRIORITY_TIMEOUT = 5000;
const LOW_PRIORITY_TIMEOUT = 10000;
const IDLE_PRIORITY_TIMEOUT = maxSigned31BitInt;

const taskQueue = [];
let taskIdCounter = 1;
let scheduledHostCallback = null;
let startTime = -1;
let currentTask = null;
const frameInterval = frameYieldMs;
const channel = new MessageChannel();
const port = channel.port2;

const getCurrentTime = () => performance.now();
channel.port1.onmessage = performWorkUntilDeadline;

function schedulePerformWorkUntilDeadline() {
  port.postMessage(null);
}
function performWorkUntilDeadline() {
  if (scheduledHostCallback !== null) {
    startTime = getCurrentTime();
    let hasMoreWork = true;
    try {
      hasMoreWork = scheduledHostCallback(startTime);
    } finally {
      if (hasMoreWork) {
        schedulePerformWorkUntilDeadline();
      } else {
        scheduledHostCallback = null;
      }
    }
  }
}
function requestHostCallback(callback) {
  scheduledHostCallback = callback;
  schedulePerformWorkUntilDeadline();
}

function unstable_scheduleCallback(priorityLevel, callback) {
  const currentTime = getCurrentTime();
  const startTime = currentTime;
  let timeout;
  switch (priorityLevel) {
    case ImmediatePriority:
      timeout = IMMEDIATE_PRIORITY_TIMEOUT;
      break;
    case UserBlockingPriority:
      timeout = USER_BLOCKING_PRIORITY_TIMEOUT;
      break;
    case IdlePriority:
      timeout = IDLE_PRIORITY_TIMEOUT;
      break;
    case LowPriority:
      timeout = LOW_PRIORITY_TIMEOUT;
      break;
    case NormalPriority:
    default:
      timeout = NORMAL_PRIORITY_TIMEOUT;
      break;
  }
  const expirationTime = startTime + timeout;
  const newTask = {
    id: taskIdCounter++,
    callback,
    priorityLevel,
    startTime,
    expirationTime,
    sort
  };
  newTask.sortIndex = expirationTime;
  push(taskQueue, newTask);
  requestHostCallback(flushWork);
  return newTask;
}

function flushWork(initialTime) {
  return workLoop(initialTime);
```

```
}

function shouldYieldToHost() {
  const timeElapsed = getCurrentTime() - startTime;
  if (timeElapsed < frameInterval) {
    return false;
  }
  return true;
}
function workLoop(initialTime) {
  let currentTime = initialTime;
  currentTask = peek(taskQueue);
  while (currentTask !== null) {
    if (currentTask.expirationTime > currentTime && shouldYieldToHost()) {
      break;
    }
    const callback = currentTask.callback;
    if (typeof callback
      currentTask.callback = null;
      const didUserCallbackTimeout = currentTask.expirationTime + shouldYieldToHost as unstable_shouldYield
};
```

38.5 ReactFiberLane.js

src\react-reconciler\src\ReactFiberLane.js

```
+import { allowConcurrentByDefault } from 'shared/ReactFeatureFlags';

export const TotalLanes = 31;
export const NoLanes = 0b0000000000000000000000000000000;
export const NoLane = 0b0000000000000000000000000000000;
export const SyncLane = 0b0000000000000000000000000000001;// 1
export const InputContinuousLane = 0b0000000000000000000000000000100;// 4
export const DefaultLane = 0b0000000000000000000000000010000;// 16
export const NonIdleLanes = 0b0001111111111111111111111111111;
export const IdleLane = 0b0100000000000000000000000000000;

export function mergeLanes(a, b) {
  return a | b;
}
export function markRootUpdated(root, updateLane) {
  root.pendingLanes |= updateLane;
}

export function getNextLanes(root) {
  const pendingLanes = root.pendingLanes;
  if (pendingLanes
    return NoLanes;
  }
  const nextLanes = getHighestPriorityLanes(pendingLanes);
  return nextLanes;
}

function getHighestPriorityLanes(lanes) {
  return getHighestPriorityLane(lanes);
}

export function getHighestPriorityLane(lanes) {
  return lanes & -lanes;
}

export function includesNonIdleWork(lanes) {
  return (lanes & NonIdleLanes) !== NoLanes;
}
export function includesBlockingLane(root, lanes) {
+ if (allowConcurrentByDefault) {
+   return false;
+ }
  const SyncDefaultLanes = InputContinuousLane | DefaultLane;
  return (lanes & SyncDefaultLanes) !== NoLanes;
}
export function isSubsetOfLanes(set, subset) {
  return (set & subset)
}
```

38.6 ReactFeatureFlags.js

src\shared\ReactFeatureFlags.js

```
export const allowConcurrentByDefault = true;
```

39.批量更新

39.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

+function FunctionComponent() {
+  console.log('FunctionComponent');
+  const [number, setNumber] = React.useState(0);
+  React.useEffect(() => {
+    setNumber(number => number + 1)
+    setNumber(number => number + 1)
+  }, []);
+  return ( {
+    setNumber(number => number + 1)
+    setNumber(number => number + 1)
+  }}>{number})
+}
const element = ;
const container = document.getElementById("root");
const root = createRoot(container, { unstable_concurrentUpdatesByDefault: true });
root.render(element);
```

39.2 ReactFiberHooks.js

src\react-reconciler\src\ReactFiberHooks.js

```
import ReactSharedInternals from "shared/ReactSharedInternals";
import { enqueueConcurrentHookUpdate } from "./ReactFiberConcurrentUpdates";
import { scheduleUpdateOnFiber, requestUpdateLane } from "./ReactFiberWorkLoop";
import is from "shared/objectIs";
import { Passive as PassiveEffect, Update as UpdateEffect } from "./ReactFiberFlags";
import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout } from "./ReactHookEffectTags";
+import { NoLanes } from './ReactFiberLane';

const { ReactCurrentDispatcher } = ReactSharedInternals;
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;

const HooksDispatcherOnMountInDEV = {
  useReducer: mountReducer,
  useState: mountState,
  useEffect: mountEffect,
  useLayoutEffect: mountLayoutEffect,
};
const HooksDispatcherOnUpdateInDEV = {
  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) {
  const fiberFlags = UpdateEffect;
  return mountEffectImpl(fiberFlags, HookLayout, create, deps);
}
function updateEffect(create, deps) {
  return updateEffectImpl(PassiveEffect, HookPassive, create, deps);
}
function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
  const hook = updateWorkInProgressHook();
  const nextDeps = deps
  let destroy;
  if (currentHook !== null) {
    const prevEffect = currentHook.memoizedState;
    destroy = prevEffect.destroy;
    if (nextDeps !== null) {
      const prevDeps = prevEffect.deps;
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        hook.memoizedState = 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 (is(nextDeps[i], prevDeps[i])) {
      continue;
    }
    return false;
  }

  return true;
}
function mountEffect(create, deps) {
  return mountEffectImpl(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,
  };
}
function basicStateReducer(state, action) {
  return typeof action
}
function mountReducer(reducer, initialArg) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialArg;
  const queue = {
    pending: null,
    dispatch: null,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchReducerAction.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function updateReducer(reducer) {
  const hook = updateWorkInProgressHook();
  const queue = hook.queue;
  queue.lastRenderedReducer = reducer;
  const current = currentHook;
  const pendingQueue = queue.pending;
  let baseQueue = null;
  let newState = current.memoizedState;
  if (pendingQueue !== null) {
    baseQueue = pendingQueue;
    queue.pending = null;
  }
  if (baseQueue !== null) {
    const first = baseQueue.next;
    let update = first;
    do {
      if (update.hasEagerState) {
        newState = update.eagerState;
      } else {
        const action = update.action;
        newState = reducer(newState, action);
      }
      update = update.next;
    } while (update !== null && update !== first);
  }
  hook.memoizedState = newState;
  queue.lastRenderedState = newState;
  const dispatch = queue.dispatch;
  return [hook.memoizedState, dispatch];
}
function mountState(initialState) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialState;
  const queue = {
    pending: null,
    dispatch: null,
    lastRenderedReducer: basicStateReducer,
    lastRenderedState: initialState,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchSetState.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function dispatchSetState(fiber, queue, action) {
  const lane = requestUpdateLane(fiber);
  const update = {
    lane,
    action,
    hasEagerState: false,
    eagerState: null,
    next: null,
  };
+ const alternate = fiber.alternate;
+ if (fiber.lanes === NoLanes && (alternate === null || alternate.lanes === NoLanes)) {
    const lastRenderedReducer = queue.lastRenderedReducer;
    const currentState = queue.lastRenderedState;
    const eagerState = lastRenderedReducer(currentState, action);
    update.hasEagerState = true;
    update.eagerState = eagerState;
    if (is(eagerState, currentState)) {
      return;
```

```
    }
+ }
  const root = enqueueConcurrentHookUpdate(fiber, queue, update, lane);
  scheduleUpdateOnFiber(root, fiber, lane);
}
function updateState(initialState) {
  return updateReducer(basicStateReducer, initialState);
}
function mountWorkInProgressHook() {
  const hook = {
    memoizedState: null,
    queue: null,
    next: null,
  };
  if (workInProgressHook
    currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
  } else {
    workInProgressHook = workInProgressHook.next = hook;
  }
  return workInProgressHook;
}
function dispatchReducerAction(fiber, queue, action) {
  const update = {
    action,
    next: null,
  };
  const root = enqueueConcurrentHookUpdate(fiber, queue, update);
  scheduleUpdateOnFiber(root, fiber);
}

function updateWorkInProgressHook() {
  let nextCurrentHook;
  if (currentHook
    const current = currentlyRenderingFiber.alternate;
    if (current !== null) {
      nextCurrentHook = current.memoizedState;
    } else {
      nextCurrentHook = null;
    }
  } else {
    nextCurrentHook = currentHook.next;
  }

  let nextWorkInProgressHook;
  if (workInProgressHook
    nextWorkInProgressHook = currentlyRenderingFiber.memoizedState;
  } else {
    nextWorkInProgressHook = workInProgressHook.next;
  }

  if (nextWorkInProgressHook !== null) {
    workInProgressHook = nextWorkInProgressHook;
    nextWorkInProgressHook = workInProgressHook.next;
    currentHook = nextCurrentHook;
  } else {
    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 renderWithHooks(current, workInProgress, Component, props) {
  currentlyRenderingFiber = workInProgress;
  workInProgress.updateQueue = null;
  workInProgress.memoizedState = null;
  if (current !== null && current.memoizedState !== null) {
    ReactCurrentDispatcher.current = HooksDispatcherOnUpdateInDEV;
  } else {
    ReactCurrentDispatcher.current = HooksDispatcherOnMountInDEV;
  }
  const children = Component(props);
  currentlyRenderingFiber = null;
  workInProgressHook = null;
  currentHook = null;
  return children;
}
```

39.3 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import {
  scheduleCallback as Scheduler_scheduleCallback,
  ImmediatePriority as ImmediateSchedulerPriority,
  UserBlockingPriority as UserBlockingSchedulerPriority,
  NormalPriority as NormalSchedulerPriority,
  IdlePriority as IdleSchedulerPriority,
  shouldYield
} from "./Scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
import { MutationMask, NoFlags, Passive } from "./ReactFiberFlags";
import {
  commitMutationEffects,
```

```
  commitPassiveUnmountEffects,
  commitPassiveMountEffects,
  commitLayoutEffects,
} from "./ReactFiberCommitWork";
import { finishQueueingConcurrentUpdates } from "./ReactFiberConcurrentUpdates";
import {
  NoLane, markRootUpdated, NoLanes,
  getNextLanes, getHighestPriorityLane, SyncLane,
  includesBlockingLane
} from './ReactFiberLane';
import {
  getCurrentUpdatePriority, lanesToEventPriority, DiscreteEventPriority, ContinuousEventPriority,
  DefaultEventPriority, IdleEventPriority, setCurrentUpdatePriority
} from './ReactEventPriorities';
import { getCurrentEventPriority } from 'react-dom-bindings/src/client/ReactDOMHostConfig';
import { scheduleSyncCallback, flushSyncCallbacks } from './ReactFiberSyncTaskQueue';

let workInProgress = null;
let rootDoesHavePassiveEffects = false;
let rootWithPendingPassiveEffects = null;
let workInProgressRootRenderLanes = NoLanes;

const RootInProgress = 0;
const RootCompleted = 5;
let workInProgressRoot = null;
let workInProgressRootExitStatus = RootInProgress;

export function scheduleUpdateOnFiber(root, fiber, lane) {
  markRootUpdated(root, lane);
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
  const nextLanes = getNextLanes(root, NoLanes);
  if (nextLanes
    root.callbackNode = null;
    root.callbackPriority = NoLane;
    return;
  }
  const newCallbackPriority = getHighestPriorityLane(nextLanes);
+ const existingCallbackPriority = root.callbackPriority;
+ if (existingCallbackPriority === newCallbackPriority) {
+   return;
+ }
+  let newCallbackNode;
  if (newCallbackPriority
    scheduleSyncCallback(performSyncWorkOnRoot.bind(null, root));
    queueMicrotask(flushSyncCallbacks);
    newCallbackNode = null;
  } else {
    let schedulerPriorityLevel;
    switch (lanesToEventPriority(nextLanes)) {
      case DiscreteEventPriority:
        schedulerPriorityLevel = ImmediateSchedulerPriority;
        break;
      case ContinuousEventPriority:
        schedulerPriorityLevel = UserBlockingSchedulerPriority;
        break;
      case DefaultEventPriority:
        schedulerPriorityLevel = NormalSchedulerPriority;
        break;
      case IdleEventPriority:
        schedulerPriorityLevel = IdleSchedulerPriority;
        break;
      default:
        schedulerPriorityLevel = NormalSchedulerPriority;
        break;
    }
+   newCallbackNode = Scheduler_scheduleCallback(schedulerPriorityLevel, performConcurrentWorkOnRoot.bind(null, root))
  }
+ root.callbackPriority = newCallbackPriority;
+ root.callbackNode = newCallbackNode;
}
function performSyncWorkOnRoot(root) {
  const lanes = getNextLanes(root, NoLanes);
  renderRootSync(root, lanes);
  const finishedWork = root.current.alternate
  root.finishedWork = finishedWork
  commitRoot(root)
  return null;
}
function performConcurrentWorkOnRoot(root, didTimeout) {
  const originalCallbackNode = root.callbackNode;
  const lanes = getNextLanes(root, NoLanes);
  if (lanes
    return null;
  }
  const shouldTimeSlice = !includesBlockingLane(root, lanes) && (!didTimeout);
  const exitStatus = shouldTimeSlice ? renderRootConcurrent(root, lanes) : renderRootSync(root, lanes);
  if (exitStatus !== RootInProgress) {
    const finishedWork = root.current.alternate
    root.finishedWork = finishedWork
    commitRoot(root)
  }
  if (root.callbackNode
    return performConcurrentWorkOnRoot.bind(null, root);
  }
  return null;
}
function renderRootConcurrent(root, lanes) {
  if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
    prepareFreshStack(root, lanes);
  }
  workLoopConcurrent();
  if (workInProgress !== null) {
```

```
    return RootInProgress;
  }
  workInProgressRoot = null;
  workInProgressRootRenderLanes = NoLanes;
  return workInProgressRootExitStatus;
}
function workLoopConcurrent() {
  while (workInProgress !== null && !shouldYield()) {
    performUnitOfWork(workInProgress);
  }
}
export function flushPassiveEffects() {
  if (rootWithPendingPassiveEffects !== null) {
    const root = rootWithPendingPassiveEffects;
    commitPassiveUnmountEffects(root.current);
    commitPassiveMountEffects(root, root.current);
  }
}
function commitRoot(root) {
  const previousPriority = getCurrentUpdatePriority();
  try {
    setCurrentUpdatePriority(DiscreteEventPriority);
    commitRootImpl(root);
  } finally {
    setCurrentUpdatePriority(previousPriority);
  }
}
function commitRootImpl(root) {
  const { finishedWork } = root;
  root.callbackNode = null;
+ root.callbackPriority = NoLane;
  if ((finishedWork.subtreeFlags & Passive) !== NoFlags || (finishedWork.flags & Passive) !== NoFlags) {
    if (!rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = true;
      Scheduler_scheduleCallback(NormalSchedulerPriority, flushPassiveEffects);
    }
  }
  const subtreeHasEffects = (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
    commitMutationEffects(finishedWork, root);
    commitLayoutEffects(finishedWork, root);
    root.current = finishedWork;
    if (rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = false;
      rootWithPendingPassiveEffects = root;
    }
  }
  root.current = finishedWork;
}
function prepareFreshStack(root, lanes) {
  workInProgress = createWorkInProgress(root.current, null);
  workInProgressRootRenderLanes = lanes;
  finishQueueingConcurrentUpdates();
}
function renderRootSync(root, lanes) {
  if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
    prepareFreshStack(root, lanes)
  }
  workLoopSync();
  workInProgressRoot = null;
  workInProgressRootRenderLanes = NoLanes;
  return workInProgressRootExitStatus;
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork, workInProgressRootRenderLanes);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
  if (workInProgressRootExitStatus
    workInProgressRootExitStatus = RootCompleted;
  }
}

export function requestUpdateLane() {
  const updateLane = getCurrentUpdatePriority();
  if (updateLane !== NoLane) {
    return updateLane;
```

```
  }
  const eventLane = getCurrentEventPriority();
  return eventLane;
}
```

40.高优更新打断低优更新(useRef)

40.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

function FunctionComponent() {
+function FunctionComponent() {
+  const [numbers, setNumbers] = React.useState(new Array(10).fill('A'));
+  const divRef = React.useRef();
+  React.useEffect(() => {
+    setTimeout(() => {
+      divRef.current.click();
+    }, 10);
+    setNumbers(numbers => numbers.map(item => item + 'B'))
+  }, []);
+  return ( {
+    setNumbers(numbers => numbers.map(item => item + 'C'))
+  }}>{numbers.map((number, index) => {number})})
+}
const element = ;
const container = document.getElementById("root");
const root = createRoot(container, { unstable_concurrentUpdatesByDefault: true });
root.render(element);
```

40.2 react\index.js

src\react\index.js

```
export {
  __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED,
  useReducer,
  useState,
  useEffect,
  useLayoutEffect,
+ useRef
} from "./src/React";
```

40.3 React.js

src\react\src\React.js

```
+import { useReducer, useState, useEffect, useLayoutEffect, useRef } from "./ReactHooks";
import ReactSharedInternals from "./ReactSharedInternals";

export {
  useReducer,
  useState,
  useEffect,
  useLayoutEffect,
+ useRef,
  ReactSharedInternals as __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED,
};
```

40.4 ReactHooks.js

src\react\src\ReactHooks.js

```
import ReactCurrentDispatcher from "./ReactCurrentDispatcher";

function resolveDispatcher() {
  const dispatcher = ReactCurrentDispatcher.current;
  return dispatcher;
}

export function useReducer(reducer, initialArg, init) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useReducer(reducer, initialArg, init);
}

export function useState(initialState) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useState(initialState);
}

export function useEffect(create, deps) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useEffect(create, deps);
}
export function useLayoutEffect(create, deps) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useLayoutEffect(create, deps);
}
+export function useRef(initialValue) {
+  const dispatcher = resolveDispatcher();
+  return dispatcher.useRef(initialValue);
+}
```

40.5 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import {
  scheduleCallback as Scheduler_scheduleCallback,
  ImmediatePriority as ImmediateSchedulerPriority,
  UserBlockingPriority as UserBlockingSchedulerPriority,
  NormalPriority as NormalSchedulerPriority,
```

```
  IdlePriority as IdleSchedulerPriority,
  shouldYield,
+ cancelCallback as Scheduler_cancelCallback
} from "./Scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
import { MutationMask, NoFlags, Passive } from "./ReactFiberFlags";
import {
  commitMutationEffects,
  commitPassiveUnmountEffects,
  commitPassiveMountEffects,
  commitLayoutEffects,
} from "./ReactFiberCommitWork";
import { finishQueueingConcurrentUpdates } from "./ReactFiberConcurrentUpdates";
import {
  NoLane, markRootUpdated, NoLanes,
  getNextLanes, getHighestPriorityLane, SyncLane,
  includesBlockingLane
} from './ReactFiberLane';
import {
  getCurrentUpdatePriority, lanesToEventPriority, DiscreteEventPriority, ContinuousEventPriority,
  DefaultEventPriority, IdleEventPriority, setCurrentUpdatePriority
} from './ReactEventPriorities';
import { getCurrentEventPriority } from 'react-dom-bindings/src/client/ReactDOMHostConfig';
import { scheduleSyncCallback, flushSyncCallbacks } from './ReactFiberSyncTaskQueue';

let workInProgress = null;
let rootDoesHavePassiveEffects = false;
let rootWithPendingPassiveEffects = null;
let workInProgressRootRenderLanes = NoLanes;

const RootInProgress = 0;
const RootCompleted = 5;
let workInProgressRoot = null;
let workInProgressRootExitStatus = RootInProgress;

export function scheduleUpdateOnFiber(root, fiber, lane) {
  markRootUpdated(root, lane);
  ensureRootIsScheduled(root);
}
function ensureRootIsScheduled(root) {
+ const existingCallbackNode = root.callbackNode;
+ const nextLanes = getNextLanes(root, root === workInProgressRoot ? workInProgressRootRenderLanes : NoLanes);
  if (nextLanes
    root.callbackNode = null;
    root.callbackPriority = NoLane;
    return;
  }
  const newCallbackPriority = getHighestPriorityLane(nextLanes);
  const existingCallbackPriority = root.callbackPriority;
  if (existingCallbackPriority
    return;
  }
+ if (existingCallbackNode != null) {
+   Scheduler_cancelCallback(existingCallbackNode);
+ }
  let newCallbackNode;
  if (newCallbackPriority
    scheduleSyncCallback(performSyncWorkOnRoot.bind(null, root));
    queueMicrotask(flushSyncCallbacks);
    newCallbackNode = null;
  } else {
    let schedulerPriorityLevel;
    switch (lanesToEventPriority(nextLanes)) {
      case DiscreteEventPriority:
        schedulerPriorityLevel = ImmediateSchedulerPriority;
        break;
      case ContinuousEventPriority:
        schedulerPriorityLevel = UserBlockingSchedulerPriority;
        break;
      case DefaultEventPriority:
        schedulerPriorityLevel = NormalSchedulerPriority;
        break;
      case IdleEventPriority:
        schedulerPriorityLevel = IdleSchedulerPriority;
        break;
      default:
        schedulerPriorityLevel = NormalSchedulerPriority;
        break;
    }
    newCallbackNode = Scheduler_scheduleCallback(schedulerPriorityLevel, performConcurrentWorkOnRoot.bind(null, root))
  }
  root.callbackPriority = newCallbackPriority;
  root.callbackNode = newCallbackNode;
}
function performSyncWorkOnRoot(root) {
  const lanes = getNextLanes(root, NoLanes);
  renderRootSync(root, lanes);
  const finishedWork = root.current.alternate
  root.finishedWork = finishedWork
  commitRoot(root)
  return null;
}
function performConcurrentWorkOnRoot(root, didTimeout) {
  const originalCallbackNode = root.callbackNode;
  const lanes = getNextLanes(root, NoLanes);
  if (lanes
    return null;
  }
  const shouldTimeSlice = !includesBlockingLane(root, lanes) && (!didTimeout);
  const exitStatus = shouldTimeSlice ? renderRootConcurrent(root, lanes) : renderRootSync(root, lanes);
  if (exitStatus !== RootInProgress) {
    const finishedWork = root.current.alternate
```

```
    root.finishedWork = finishedWork
    commitRoot(root)
  }
  if (root.callbackNode
    return performConcurrentWorkOnRoot.bind(null, root);
  }
  return null;
}
function renderRootConcurrent(root, lanes) {
  if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
    prepareFreshStack(root, lanes);
  }
  workLoopConcurrent();
  if (workInProgress !== null) {
    return RootInProgress;
  }
  workInProgressRoot = null;
  workInProgressRootRenderLanes = NoLanes;
  return workInProgressRootExitStatus;
}
function workLoopConcurrent() {
  while (workInProgress !== null && !shouldYield()) {
    performUnitOfWork(workInProgress);
  }
}
export function flushPassiveEffects() {
  if (rootWithPendingPassiveEffects !== null) {
    const root = rootWithPendingPassiveEffects;
    commitPassiveUnmountEffects(root.current);
    commitPassiveMountEffects(root, root.current);
  }
}
function commitRoot(root) {
  const previousPriority = getCurrentUpdatePriority();
  try {
    setCurrentUpdatePriority(DiscreteEventPriority);
    commitRootImpl(root);
  } finally {
    setCurrentUpdatePriority(previousPriority);
  }
}
function commitRootImpl(root) {
  const { finishedWork } = root;
  console.log('commit', finishedWork.child.memoizedState.memoizedState);
  root.callbackNode = null;
  root.callbackPriority = NoLane;
  if ((finishedWork.subtreeFlags & Passive) !== NoFlags || (finishedWork.flags & Passive) !== NoFlags) {
    if (!rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = true;
      Scheduler_scheduleCallback(NormalSchedulerPriority, flushPassiveEffects);
    }
  }
  const subtreeHasEffects = (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
    commitMutationEffects(finishedWork, root);
    commitLayoutEffects(finishedWork, root);
    root.current = finishedWork;
    if (rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = false;
      rootWithPendingPassiveEffects = root;
    }
  }
  root.current = finishedWork;
}
function prepareFreshStack(root, lanes) {
  workInProgress = createWorkInProgress(root.current, null);
  workInProgressRootRenderLanes = lanes;
  finishQueueingConcurrentUpdates();
}
function renderRootSync(root, lanes) {
  if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
    prepareFreshStack(root, lanes)
  }
  workLoopSync();
  workInProgressRoot = null;
  workInProgressRootRenderLanes = NoLanes;
  return workInProgressRootExitStatus;
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork, workInProgressRootRenderLanes);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
```

```
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
  if (workInProgressRootExitStatus
    workInProgressRootExitStatus = RootCompleted;
  }
}

export function requestUpdateLane() {
  const updateLane = getCurrentUpdatePriority();
  if (updateLane !== NoLane) {
    return updateLane;
  }
  const eventLane = getCurrentEventPriority();
  return eventLane;
}
```

40.6 ReactFiberLane.js

src\react-reconciler\src\ReactFiberLane.js

```
import { allowConcurrentByDefault } from 'shared/ReactFeatureFlags';

export const TotalLanes = 31;
export const NoLanes = 0b0000000000000000000000000000000;
export const NoLane = 0b0000000000000000000000000000000;
export const SyncLane = 0b0000000000000000000000000000001;// 1
export const InputContinuousLane = 0b0000000000000000000000000000100;// 4
export const DefaultLane = 0b0000000000000000000000000010000;// 16
export const NonIdleLanes = 0b0001111111111111111111111111111;
export const IdleLane = 0b0100000000000000000000000000000;

export function mergeLanes(a, b) {
  return a | b;
}
export function markRootUpdated(root, updateLane) {
  root.pendingLanes |= updateLane;
}

+export function getNextLanes(root, wipLanes) {
  const pendingLanes = root.pendingLanes;
  if (pendingLanes
    return NoLanes;
  }
  const nextLanes = getHighestPriorityLanes(pendingLanes);
+ if (wipLanes !== NoLanes && wipLanes !== nextLanes) {
+   if (nextLanes >= wipLanes) {
+     return wipLanes;
+   }
+ }
  return nextLanes;
}

function getHighestPriorityLanes(lanes) {
  return getHighestPriorityLane(lanes);
}

export function getHighestPriorityLane(lanes) {
  return lanes & -lanes;
}

export function includesNonIdleWork(lanes) {
  return (lanes & NonIdleLanes) !== NoLanes;
}
export function includesBlockingLane(root, lanes) {
  if (allowConcurrentByDefault) {
    return false;
  }
  const SyncDefaultLanes = InputContinuousLane | DefaultLane;
  return (lanes & SyncDefaultLanes) !== NoLanes;
}
export function isSubsetOfLanes(set, subset) {
  return (set & subset)
}
```

40.7 ReactFiberHooks.js

src\react-reconciler\src\ReactFiberHooks.js

```
import ReactSharedInternals from "shared/ReactSharedInternals";
import { enqueueConcurrentHookUpdate } from "./ReactFiberConcurrentUpdates";
import { scheduleUpdateOnFiber, requestUpdateLane } from "./ReactFiberWorkLoop";
import is from "shared/objectIs";
import { Passive as PassiveEffect, Update as UpdateEffect } from "./ReactFiberFlags";
import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout } from "./ReactHookEffectTags";
import { NoLanes } from './ReactFiberLane';

const { ReactCurrentDispatcher } = ReactSharedInternals;
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;

const HooksDispatcherOnMountInDEV = {
  useReducer: mountReducer,
  useState: mountState,
  useEffect: mountEffect,
  useLayoutEffect: mountLayoutEffect,
+ useRef: mountRef,
};
const HooksDispatcherOnUpdateInDEV = {
  useReducer: updateReducer,
```

```
  useState: updateState,
  useEffect: updateEffect,
  useLayoutEffect: updateLayoutEffect,
+ useRef: updateRef
};
+function mountRef(initialValue) {
+  const hook = mountWorkInProgressHook();
+  const ref = {
+    current: initialValue,
+  };
+  hook.memoizedState = ref;
+  return ref;
+}
+function updateRef() {
+  const hook = updateWorkInProgressHook();
+  return hook.memoizedState;
+}
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) {
  const fiberFlags = UpdateEffect;
  return mountEffectImpl(fiberFlags, HookLayout, create, deps);
}
function updateEffect(create, deps) {
  return updateEffectImpl(PassiveEffect, HookPassive, create, deps);
}
function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
  const hook = updateWorkInProgressHook();
  const nextDeps = deps
  let destroy;
  if (currentHook !== null) {
    const prevEffect = currentHook.memoizedState;
    destroy = prevEffect.destroy;
    if (nextDeps !== null) {
      const prevDeps = prevEffect.deps;
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        hook.memoizedState = 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 (is(nextDeps[i], prevDeps[i])) {
      continue;
    }
    return false;
  }

  return true;
}
function mountEffect(create, deps) {
  return mountEffectImpl(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,
  };
}
function basicStateReducer(state, action) {
  return typeof action
}
```

```
function mountReducer(reducer, initialArg) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialArg;
  const queue = {
    pending: null,
    dispatch: null,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchReducerAction.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function updateReducer(reducer) {
  const hook = updateWorkInProgressHook();
  const queue = hook.queue;
  queue.lastRenderedReducer = reducer;
  const current = currentHook;
  const pendingQueue = queue.pending;
  let baseQueue = null;
  let newState = current.memoizedState;
  if (pendingQueue !== null) {
    baseQueue = pendingQueue;
    queue.pending = null;
  }
  if (baseQueue !== null) {
    const first = baseQueue.next;
    let update = first;
    do {
      if (update.hasEagerState) {
        newState = update.eagerState;
      } else {
        const action = update.action;
        newState = reducer(newState, action);
      }
      update = update.next;
    } while (update !== null && update !== first);
  }
  hook.memoizedState = newState;
  queue.lastRenderedState = newState;
  const dispatch = queue.dispatch;
  return [hook.memoizedState, dispatch];
}
function mountState(initialState) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialState;
  const queue = {
    pending: null,
    dispatch: null,
    lastRenderedReducer: basicStateReducer,
    lastRenderedState: initialState,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchSetState.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function dispatchSetState(fiber, queue, action) {
  const lane = requestUpdateLane(fiber);
  const update = {
    lane,
    action,
    hasEagerState: false,
    eagerState: null,
    next: null,
  };
  const alternate = fiber.alternate;
  if (fiber.lanes
    const lastRenderedReducer = queue.lastRenderedReducer;
    const currentState = queue.lastRenderedState;
    const eagerState = lastRenderedReducer(currentState, action);
    update.hasEagerState = true;
    update.eagerState = eagerState;
    if (is(eagerState, currentState)) {
      return;
    }
  }
  const root = enqueueConcurrentHookUpdate(fiber, queue, update, lane);
  scheduleUpdateOnFiber(root, fiber, lane);
}
function updateState(initialState) {
  return updateReducer(basicStateReducer, initialState);
}
function mountWorkInProgressHook() {
  const hook = {
    memoizedState: null,
    queue: null,
    next: null,
  };
  if (workInProgressHook
    currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
  } else {
    workInProgressHook = workInProgressHook.next = hook;
  }
  return workInProgressHook;
}
function dispatchReducerAction(fiber, queue, action) {
  const update = {
    action,
    next: null,
  };
  const root = enqueueConcurrentHookUpdate(fiber, queue, update);
  scheduleUpdateOnFiber(root, fiber);
}

function updateWorkInProgressHook() {
  let nextCurrentHook;
```

```
  if (currentHook
    const current = currentlyRenderingFiber.alternate;
    if (current !== null) {
      nextCurrentHook = current.memoizedState;
    } else {
      nextCurrentHook = null;
    }
  } else {
    nextCurrentHook = currentHook.next;
  }

  let nextWorkInProgressHook;
  if (workInProgressHook
    nextWorkInProgressHook = currentlyRenderingFiber.memoizedState;
  } else {
    nextWorkInProgressHook = workInProgressHook.next;
  }

  if (nextWorkInProgressHook !== null) {
    workInProgressHook = nextWorkInProgressHook;
    nextWorkInProgressHook = workInProgressHook.next;
    currentHook = nextCurrentHook;
  } else {
    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 renderWithHooks(current, workInProgress, Component, props) {
  currentlyRenderingFiber = workInProgress;
  workInProgress.updateQueue = null;
  workInProgress.memoizedState = null;
  if (current !== null && current.memoizedState !== null) {
    ReactCurrentDispatcher.current = HooksDispatcherOnUpdateInDEV;
  } else {
    ReactCurrentDispatcher.current = HooksDispatcherOnMountInDEV;
  }
  const children = Component(props);
  currentlyRenderingFiber = null;
  workInProgressHook = null;
  currentHook = null;
  return children;
}
```

40.8 ReactFiberFlags.js

src\react-reconciler\src\ReactFiberFlags.js

```
export const NoFlags = 0b00000000000000000000000000;
export const Placement = 0b00000000000000000000000010;
export const Update = 0b00000000000000000000000100;
export const ChildDeletion = 0b00000000000000000000001000;
export const Passive = 0b00000000000000010000000000;
export const LayoutMask = Update;
+export const Ref = 0b00000000000000000100000000;
+export const MutationMask = Placement | Update | Ref;
```

40.9 ReactFiberCompleteWork.js

src\react-reconciler\src\ReactFiberCompleteWork.js

```
import {
  appendInitialChild,
  createInstance,
  createTextInstance,
  finalizeInitialChildren,
  prepareUpdate,
} from "react-dom-bindings/src/client/ReactDOMHostConfig";
import { HostComponent, HostRoot, HostText, FunctionComponent } from "./ReactWorkTags";
+import { Ref, NoFlags, Update } from "./ReactFiberFlags";
+
+function markRef(workInProgress) {
+  workInProgress.flags |= Ref;
+}

function bubbleProperties(completedWork) {
  let subtreeFlags = NoFlags;
  let child = completedWork.child;
  while (child !== null) {
    subtreeFlags |= child.subtreeFlags;
    subtreeFlags |= child.flags;
    child = child.sibling;
  }
  completedWork.subtreeFlags |= subtreeFlags;
}

function appendAllChildren(parent, workInProgress) {
  // 我们只有创建的顶级fiber，但需要递归其子节点来查找所有终端节点
  let node = workInProgress.child;
  while (node !== null) {
    // 如果是原生节点，直接添加到父节点上
    if (node.tag
      appendInitialChild(parent, node.stateNode);
      // 再看看第一个节节点是不是原生节点
```

```
    } else if (node.child !== null) {
      // node.child.return = node
      node = node.child;
      continue;
    }
    if (node
      return;
    }
    // 如果没有弟弟就找父亲的弟弟
    while (node.sibling
      // 如果找到了根节点或者回到了原节点结束
      if (node.return
        return;
      }
      node = node.return;
    }
    // node.sibling.return = node.return
    // 下一个弟弟节点
    node = node.sibling;
  }
}
function markUpdate(workInProgress) {
  workInProgress.flags |= Update;
}
function updateHostComponent(current, workInProgress, type, newProps) {
  const oldProps = current.memoizedProps;
  const instance = workInProgress.stateNode;
  const updatePayload = prepareUpdate(instance, type, oldProps, newProps);
  workInProgress.updateQueue = updatePayload;
  if (updatePayload) {
    markUpdate(workInProgress);
  }
}
export function completeWork(current, workInProgress) {
  const newProps = workInProgress.pendingProps;
  switch (workInProgress.tag) {
    case HostComponent: {
      const { type } = workInProgress;
      if (current !== null && workInProgress.stateNode != null) {
        updateHostComponent(current, workInProgress, type, newProps);
+       if (current.ref !== workInProgress.ref) {
+         markRef(workInProgress);
+       }
      } else {
        const instance = createInstance(type, newProps, workInProgress);
        appendAllChildren(instance, workInProgress);
        workInProgress.stateNode = instance;
        finalizeInitialChildren(instance, type, newProps);
+       if (workInProgress.ref !== null) {
+         markRef(workInProgress);
+       }
      }
      bubbleProperties(workInProgress);
      return null;
      break;
    }
    case FunctionComponent:
      bubbleProperties(workInProgress);
      break;
    case HostRoot:
      bubbleProperties(workInProgress);
      break;
    case HostText: {
      const newText = newProps;
      workInProgress.stateNode = createTextInstance(newText);
      bubbleProperties(workInProgress);
      break;
    }
    default:
      break;
  }
}
```

40.10 ReactFiberCommitWork.js

src\react-reconciler\src\ReactFiberCommitWork.js

```
import { HostRoot, HostComponent, HostText, FunctionComponent } from "./ReactWorkTags";
+import { Passive, MutationMask, Placement, Update, LayoutMask, Ref } from "./ReactFiberFlags";
import { insertBefore, appendChild, commitUpdate, removeChild } from "react-dom-bindings/src/client/ReactDOMHostConfig";
import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout } from "./ReactHookEffectTags";

export function commitMutationEffects(finishedWork, root) {
  commitMutationEffectsOnFiber(finishedWork, root);
}
export function commitPassiveUnmountEffects(finishedWork) {
  commitPassiveUnmountOnFiber(finishedWork);
}
function commitPassiveUnmountOnFiber(finishedWork) {
  switch (finishedWork.tag) {
    case FunctionComponent: {
      recursivelyTraversePassiveUnmountEffects(finishedWork);
      if (finishedWork.flags & Passive) {
        commitHookPassiveUnmountEffects(finishedWork, finishedWork.return, HookPassive | HookHasEffect);
      }
      break;
    }
    default: {
      recursivelyTraversePassiveUnmountEffects(finishedWork);
      break;
    }
  }
}
function recursivelyTraversePassiveUnmountEffects(parentFiber) {
```

```
  if (parentFiber.subtreeFlags & Passive) {
    let child = parentFiber.child;
    while (child !== null) {
      commitPassiveUnmountOnFiber(child);
      child = child.sibling;
    }
  }
}
function commitHookPassiveUnmountEffects(finishedWork, nearestMountedAncestor, hookFlags) {
  commitHookEffectListUnmount(hookFlags, finishedWork, nearestMountedAncestor);
}

function commitHookEffectListUnmount(flags, 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 & flags)
        const destroy = effect.destroy;
        effect.destroy = undefined;
        if (destroy !== undefined) {
          destroy();
        }
      }
      effect = effect.next;
    } while (effect !== firstEffect);
  }
}

export function commitPassiveMountEffects(root, finishedWork) {
  commitPassiveMountOnFiber(root, finishedWork);
}
function commitPassiveMountOnFiber(finishedRoot, finishedWork) {
  const flags = finishedWork.flags;
  switch (finishedWork.tag) {
    case FunctionComponent: {
      recursivelyTraversePassiveMountEffects(finishedRoot, finishedWork);
      if (flags & Passive) {
        commitHookPassiveMountEffects(finishedWork, HookPassive | HookHasEffect);
      }
      break;
    }
    case HostRoot: {
      recursivelyTraversePassiveMountEffects(finishedRoot, finishedWork);
      break;
    }
    default:
      break;
  }
}
function commitHookPassiveMountEffects(finishedWork, hookFlags) {
  commitHookEffectListMount(hookFlags, finishedWork);
}
function commitHookEffectListMount(flags, 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 & flags)
        const create = effect.create;
        effect.destroy = create();
      }
      effect = effect.next;
    } while (effect !== firstEffect);
  }
}
function recursivelyTraversePassiveMountEffects(root, parentFiber) {
  if (parentFiber.subtreeFlags & Passive) {
    let child = parentFiber.child;
    while (child !== null) {
      commitPassiveMountOnFiber(root, child);
      child = child.sibling;
    }
  }
}
let hostParent = null;
function commitDeletionEffects(root, returnFiber, deletedFiber) {
  let parent = returnFiber;
  findParent: while (parent !== null) {
    switch (parent.tag) {
      case HostComponent: {
        hostParent = parent.stateNode;
        break findParent;
      }
      case HostRoot: {
        hostParent = parent.stateNode.containerInfo;
        break findParent;
      }
      default:
        break;
    }
    parent = parent.return;
  }
  commitDeletionEffectsOnFiber(root, returnFiber, deletedFiber);
  hostParent = null;
}
function commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor, deletedFiber) {
  switch (deletedFiber.tag) {
    case HostComponent:
    case HostText: {
```

```
      recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor, deletedFiber);
      if (hostParent !== null) {
        removeChild(hostParent, deletedFiber.stateNode);
      }
      break;
    }
    default:
      break;
  }
}
function recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor, parent) {
  let child = parent.child;
  while (child !== null) {
    commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor, child);
    child = child.sibling;
  }
}
function recursivelyTraverseMutationEffects(root, parentFiber) {
  const deletions = parentFiber.deletions;
  if (deletions !== null) {
    for (let i = 0; i < deletions.length; i++) {
      const childToDelete = deletions[i];
      commitDeletionEffects(root, parentFiber, childToDelete);
    }
  }
  if (parentFiber.subtreeFlags & MutationMask) {
    let { child } = parentFiber;
    while (child !== null) {
      commitMutationEffectsOnFiber(child, root);
      child = child.sibling;
    }
  }
}
function isHostParent(fiber) {
  return fiber.tag
}
function getHostParentFiber(fiber) {
  let parent = fiber.return;
  while (parent !== null) {
    if (isHostParent(parent)) {
      return parent;
    }
    parent = parent.return;
  }
  return parent;
}
function insertOrAppendPlacementNode(node, before, parent) {
  const { tag } = node;
  const isHost = tag
  if (isHost) {
    const { stateNode } = node;
    if (before) {
      insertBefore(parent, stateNode, before);
    } else {
      appendChild(parent, stateNode);
    }
  } else {
    const { child } = node;
    if (child !== null) {
      insertOrAppendPlacementNode(child, before, parent);
      let { sibling } = child;
      while (sibling !== null) {
        insertOrAppendPlacementNode(sibling, before, parent);
        sibling = sibling.sibling;
      }
    }
  }
}
function getHostSibling(fiber) {
  let node = fiber;
  siblings: while (true) {
    // 如果我们没有找到任何东西，让我们试试下一个弟弟
    while (node.sibling
      if (node.return
        // 如果我们是根Fiber或者父亲是原生节点，我们就是最后的弟弟
        return null;
      }
      node = node.return;
    }
    // node.sibling.return = node.return
    node = node.sibling;
    while (node.tag !== HostComponent && node.tag !== HostText) {
      // 如果它不是原生节点，并且，我们可能在其中有一个原生节点
      // 试着向下搜索，直到找到为止
      if (node.flags & Placement) {
        // 如果我们没有孩子，可以试试弟弟
        continue siblings;
      } else {
        // node.child.return = node
        node = node.child;
      }
    } // Check if this host node is stable or about to be placed.

    // 检查此原生节点是否稳定可以放置
    if (!(node.flags & Placement)) {
      // 找到它了!

      return node.stateNode;
    }
  }
}
function commitPlacement(finishedWork) {
  const parentFiber = getHostParentFiber(finishedWork);
  switch (parentFiber.tag) {
```

```
    case HostComponent: {
      const parent = parentFiber.stateNode;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    case HostRoot: {
      const parent = parentFiber.stateNode.containerInfo;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    default:
      break;
  }
}
function commitReconciliationEffects(finishedWork) {
  const { flags } = finishedWork;
  if (flags & Placement) {
    commitPlacement(finishedWork);
    finishedWork.flags &= ~Placement;
  }
}
export function commitMutationEffectsOnFiber(finishedWork, root) {
  const current = finishedWork.alternate;
  const flags = finishedWork.flags;
  switch (finishedWork.tag) {
    case HostRoot: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    case FunctionComponent: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      if (flags & Update) {
        commitHookEffectListUnmount(HookLayout | HookHasEffect, finishedWork, finishedWork.return);
      }
      break;
    }
    case HostComponent: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
+     if (flags & Ref) {
+       commitAttachRef(finishedWork);
+     }
      if (flags & Update) {
        const instance = finishedWork.stateNode;
        if (instance != null) {
          const newProps = finishedWork.memoizedProps;
          const oldProps = current !== null ? current.memoizedProps : newProps;
          const type = finishedWork.type;
          const updatePayload = finishedWork.updateQueue;
          finishedWork.updateQueue = null;
          if (updatePayload !== null) {
            commitUpdate(instance, updatePayload, type, oldProps, newProps, finishedWork);
          }
        }
      }

      break;
    }
    case HostText: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    default: {
      break;
    }
  }
}
+function commitAttachRef(finishedWork) {
+  const ref = finishedWork.ref;
+  if (ref !== null) {
+    const instance = finishedWork.stateNode;
+    if (typeof ref === "function") {
+      ref(instance)
+    } else {
+      ref.current = instance;
+    }
+  }
+}
export function commitLayoutEffects(finishedWork, root) {
  const current = finishedWork.alternate;
  commitLayoutEffectOnFiber(root, current, finishedWork);
}
function commitLayoutEffectOnFiber(finishedRoot, current, finishedWork) {
  const flags = finishedWork.flags;
  switch (finishedWork.tag) {
    case FunctionComponent: {
      recursivelyTraverseLayoutEffects(finishedRoot, finishedWork);
      if (flags & Update) {
        commitHookLayoutEffects(finishedWork, HookLayout | HookHasEffect);
      }
      break;
    }
    case HostRoot: {
      recursivelyTraverseLayoutEffects(finishedRoot, finishedWork);
      break;
    }
    default:
      break;
  }
```

```
}
function recursivelyTraverseLayoutEffects(root, parentFiber) {
  if (parentFiber.subtreeFlags & LayoutMask) {
    let child = parentFiber.child;
    while (child !== null) {
      const current = child.alternate;
      commitLayoutEffectOnFiber(root, current, child);
      child = child.sibling;
    }
  }
}
function commitHookLayoutEffects(finishedWork, hookFlags) {
  commitHookEffectListMount(hookFlags, finishedWork);
}
```

40.11 ReactFiber.js

src\react-reconciler\src\ReactFiber.js

```
import { HostRoot, IndeterminateComponent, HostComponent, HostText } from "./ReactWorkTags";
import { NoFlags } from "./ReactFiberFlags";
export function FiberNode(tag, pendingProps, key) {
  this.tag = tag;
  this.key = key;
  this.type = null;
  this.stateNode = null;

  this.return = null;
  this.child = null;
  this.sibling = null;

  this.pendingProps = pendingProps;
  this.memoizedProps = null;
  this.updateQueue = null;
  this.memoizedState = null;

  this.flags = NoFlags;
  this.subtreeFlags = NoFlags;
  this.deletions = null;
  this.alternate = null;

  this.index = 0;
+ this.ref = null;
}
function createFiber(tag, pendingProps, key) {
  return new FiberNode(tag, pendingProps, key);
}
export function createHostRootFiber() {
  return createFiber(HostRoot, null, null);
}
// We use a double buffering pooling technique because we know that we'll
// only ever need at most two versions of a tree. We pool the "other" unused
// node that we're free to reuse. This is lazily created to avoid allocating
// extra objects for things that are never updated. It also allow us to
// reclaim the extra memory if needed.

//我们使用双缓冲池技术，因为我们知道一棵树最多只需要两个版本
//我们将"其他"未使用的我们可以自由重用的节点
//这是延迟创建的，以避免分配从未更新的内容的额外对象。它还允许我们如果需要，回收额外的内存
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.type = current.type;
    workInProgress.flags = NoFlags;
    workInProgress.subtreeFlags = NoFlags;
  }
  workInProgress.child = current.child;
  workInProgress.memoizedProps = current.memoizedProps;
  workInProgress.memoizedState = current.memoizedState;
  workInProgress.updateQueue = current.updateQueue;
  workInProgress.sibling = current.sibling;
  workInProgress.index = current.index;
+ workInProgress.ref = current.ref;
  return workInProgress;
}
export function createFiberFromTypeAndProps(type, key, pendingProps) {
  let fiberTag = IndeterminateComponent;
  if (typeof type
    fiberTag = HostComponent;
  }
  const fiber = createFiber(fiberTag, pendingProps, key);
  fiber.type = type;
  return fiber;
}
export function createFiberFromElement(element) {
  const { type } = element;
  const { key } = element;
  const pendingProps = element.props;
  const fiber = createFiberFromTypeAndProps(type, key, pendingProps);
  return fiber;
}

export function createFiberFromText(content) {
  const fiber = createFiber(HostText, content, null);
  return fiber;
}
```

40.12 ReactChildFiber.js

src\react-reconciler\src\ReactChildFiber.js

```
import { REACT_ELEMENT_TYPE } from "shared/ReactSymbols";
import isArray from "shared/isArray";
import { createFiberFromElement, FiberNode, createFiberFromText, createWorkInProgress } from "./ReactFiber";
import { Placement, ChildDeletion } from "./ReactFiberFlags";
import { HostText } from "./ReactWorkTags";
function createChildReconciler(shouldTrackSideEffects) {
  function useFiber(fiber, pendingProps) {
    const clone = createWorkInProgress(fiber, pendingProps);
    clone.index = 0;
    clone.sibling = null;
    return clone;
  }
  function deleteChild(returnFiber, childToDelete) {
    if (!shouldTrackSideEffects) {
      return;
    }
    const deletions = returnFiber.deletions;
    if (deletions
      returnFiber.deletions = [childToDelete];
      returnFiber.flags |= ChildDeletion;
    } else {
      deletions.push(childToDelete);
    }
  }
  function deleteRemainingChildren(returnFiber, currentFirstChild) {
    if (!shouldTrackSideEffects) {
      return null;
    }
    let childToDelete = currentFirstChild;
    while (childToDelete !== null) {
      deleteChild(returnFiber, childToDelete);
      childToDelete = childToDelete.sibling;
    }
    return null;
  }
  function reconcileSingleElement(returnFiber, currentFirstChild, element) {
    const key = element.key;
    let child = currentFirstChild;
    while (child !== null) {
      if (child.key
        const elementType = element.type;
        if (child.type
          deleteRemainingChildren(returnFiber, child.sibling);
          const existing = useFiber(child, element.props);
+         existing.ref = element.ref;
          existing.return = returnFiber;
          return existing;
        }
        deleteRemainingChildren(returnFiber, child);
        break;
      } else {
        deleteChild(returnFiber, child);
      }
      child = child.sibling;
    }
    const created = createFiberFromElement(element);
+   created.ref = element.ref;
    created.return = returnFiber;
    return created;
  }
  function placeSingleChild(newFiber) {
    if (shouldTrackSideEffects && newFiber.alternate
      newFiber.flags |= Placement;
    }
    return newFiber;
  }
  function reconcileSingleTextNode(returnFiber, currentFirstChild, content) {
    const created = new FiberNode(HostText, { content }, null);
    created.return = returnFiber;
    return created;
  }
  function createChild(returnFiber, newChild) {
    if ((typeof newChild
      const created = createFiberFromText(`${newChild}`);
      created.return = returnFiber;
      return created;
    }

    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          const created = createFiberFromElement(newChild);
+         created.ref = newChild.ref;
          created.return = returnFiber;
          return created;
        }
        default:
          break;
      }
    }
    return null;
  }
  function placeChild(newFiber, lastPlacedIndex, newIndex) {
    newFiber.index = newIndex;
    if (!shouldTrackSideEffects) {
      return lastPlacedIndex;
    }
    const current = newFiber.alternate;
    if (current !== null) {
      const oldIndex = current.index;
```

```
      if (oldIndex < lastPlacedIndex) {
        newFiber.flags |= Placement;
        return lastPlacedIndex;
      } else {
        return oldIndex;
      }
    } else {
      newFiber.flags |= Placement;
      return lastPlacedIndex;
    }
  }
  function updateElement(returnFiber, current, element) {
    const elementType = element.type;
    if (current !== null) {
      if (current.type
        const existing = useFiber(current, element.props);
+       existing.ref = element.ref;
        existing.return = returnFiber;
        return existing;
      }
    }
    const created = createFiberFromElement(element);
+   created.ref = element.ref;
    created.return = returnFiber;
    return created;
  }
  function updateSlot(returnFiber, oldFiber, newChild) {
    const key = oldFiber !== null ? oldFiber.key : null;
    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          if (newChild.key
            return updateElement(returnFiber, oldFiber, newChild);
          } else {
            return null;
          }
        }
        default:
          return null;
      }
    }
  }
  function mapRemainingChildren(returnFiber, currentFirstChild) {
    const existingChildren = new Map();
    let existingChild = currentFirstChild;
    while (existingChild !== null) {
      if (existingChild.key !== null) {
        existingChildren.set(existingChild.key, existingChild);
      } else {
        existingChildren.set(existingChild.index, existingChild);
      }
      existingChild = existingChild.sibling;
    }
    return existingChildren;
  }
  function updateTextNode(returnFiber, current, textContent) {
    if (current
      const created = createFiberFromText(textContent, returnFiber.mode);
      created.return = returnFiber;
      return created;
    } else {
      const existing = useFiber(current, textContent);
      existing.return = returnFiber;
      return existing;
    }
  }
  function updateFromMap(existingChildren, returnFiber, newIdx, newChild) {
    if ((typeof newChild
      const matchedFiber = existingChildren.get(newIdx) || null;
      return updateTextNode(returnFiber, matchedFiber, "" + newChild);
    }
    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          const matchedFiber = existingChildren.get(newChild.key
          return updateElement(returnFiber, matchedFiber, newChild);
        }
      }
    }
    return null;
  }
  function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren) {
    let resultingFirstChild = null;
    let previousNewFiber = null;
    let newIdx = 0;
    let oldFiber = currentFirstChild;
    let nextOldFiber = null;
    let lastPlacedIndex = 0;
    for (; oldFiber !== null && newIdx < newChildren.length; newIdx++) {
      nextOldFiber = oldFiber.sibling;
      const newFiber = updateSlot(returnFiber, oldFiber, newChildren[newIdx]);
      if (newFiber
        break;
      }
      if (shouldTrackSideEffects) {
        if (oldFiber && newFiber.alternate
          deleteChild(returnFiber, oldFiber);
        }
      }
      lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
      if (previousNewFiber
        resultingFirstChild = newFiber;
      } else {
        previousNewFiber.sibling = newFiber;
```

```
      }
      previousNewFiber = newFiber;
      oldFiber = nextOldFiber;
    }
    if (newIdx
      deleteRemainingChildren(returnFiber, oldFiber);
      return resultingFirstChild;
    }
    if (oldFiber
      for (; newIdx < newChildren.length; newIdx++) {
        const newFiber = createChild(returnFiber, newChildren[newIdx]);
        if (newFiber
          continue;
        }
        lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
        if (previousNewFiber
          resultingFirstChild = newFiber;
        } else {
          previousNewFiber.sibling = newFiber;
        }
        previousNewFiber = newFiber;
      }
    }
    const existingChildren = mapRemainingChildren(returnFiber, oldFiber);
    for (; newIdx < newChildren.length; newIdx++) {
      const newFiber = updateFromMap(existingChildren, returnFiber, newIdx, newChildren[newIdx]);
      if (newFiber !== null) {
        if (shouldTrackSideEffects) {
          if (newFiber.alternate !== null) {
            existingChildren.delete(newFiber.key
          }
        }
        lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
        if (previousNewFiber
          resultingFirstChild = newFiber;
        } else {
          previousNewFiber.sibling = newFiber;
        }
        previousNewFiber = newFiber;
      }
    }
    if (shouldTrackSideEffects) {
      existingChildren.forEach((child) => deleteChild(returnFiber, child));
    }
    return resultingFirstChild;
  }
  function reconcileChildFibers(returnFiber, currentFirstChild, newChild) {
    if (typeof newChild
      switch (newChild.$typeof) {
        case REACT_ELEMENT_TYPE: {
          return placeSingleChild(reconcileSingleElement(returnFiber, currentFirstChild, newChild));
        }
        default:
          break;
      }
      if (isArray(newChild)) {
        return reconcileChildrenArray(returnFiber, currentFirstChild, newChild);
      }
    }
    if (typeof newChild
      return placeSingleChild(reconcileSingleTextNode(returnFiber, currentFirstChild, newChild));
    }
    return null;
  }
  return reconcileChildFibers;
}
export const reconcileChildFibers = createChildReconciler(true);
export const mountChildFibers = createChildReconciler(false);
```

41.饥饿问题

41.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

+let counter = 0;
+let timer;
+let bCounter = 0;
+let cCounter = 0;
+function FunctionComponent() {
+  const [numbers, setNumbers] = React.useState(new Array(100).fill('A'));
+  const divRef = React.useRef();
+  const updateB = (numbers) => new Array(100).fill(numbers[0] + 'B')
+  updateB.id = 'updateB' + (bCounter++);
+  const updateC = (numbers) => new Array(100).fill(numbers[0] + 'C')
+  updateC.id = 'updateC' + (cCounter++);
+  React.useEffect(() => {
+    timer = setInterval(() => {
+      console.log(divRef);
+      divRef.current.click();
+      if (counter++ === 0) {
+        setNumbers(updateB)
+      }
+      divRef.current.click();
+      if (counter++ > 10) {
+        clearInterval(timer);
+      }
+    });
+  }, []);
+  return ( setNumbers(updateC)}>
+    {numbers.map((number, index) => {number})})
+}

const element = ;
const container = document.getElementById("root");
const root = createRoot(container, { unstable_concurrentUpdatesByDefault: true });
root.render(element);
```

41.2 Scheduler.js

src\scheduler\src\forks\Scheduler.js

```
import {
  ImmediatePriority,
  UserBlockingPriority,
  NormalPriority,
  LowPriority,
  IdlePriority,
} from "../SchedulerPriorities";
import { push, pop, peek } from "../SchedulerMinHeap";
import { frameYieldMs } from "../SchedulerFeatureFlags";

const maxSigned31BitInt = 1073741823;
const IMMEDIATE_PRIORITY_TIMEOUT = -1;
const USER_BLOCKING_PRIORITY_TIMEOUT = 250;
const NORMAL_PRIORITY_TIMEOUT = 5000;
const LOW_PRIORITY_TIMEOUT = 10000;
const IDLE_PRIORITY_TIMEOUT = maxSigned31BitInt;

const taskQueue = [];
let taskIdCounter = 1;
let scheduledHostCallback = null;
let startTime = -1;
let currentTask = null;
const frameInterval = frameYieldMs;
const channel = new MessageChannel();
const port = channel.port2;

const getCurrentTime = () => performance.now();
channel.port1.onmessage = performWorkUntilDeadline;

function schedulePerformWorkUntilDeadline() {
  port.postMessage(null);
}
function performWorkUntilDeadline() {
  if (scheduledHostCallback !== null) {
    startTime = getCurrentTime();
    let hasMoreWork = true;
    try {
      hasMoreWork = scheduledHostCallback(startTime);
    } finally {
      if (hasMoreWork) {
        schedulePerformWorkUntilDeadline();
      } else {
        scheduledHostCallback = null;
      }
    }
  }
}
function requestHostCallback(callback) {
  scheduledHostCallback = callback;
  schedulePerformWorkUntilDeadline();
}

function unstable_scheduleCallback(priorityLevel, callback) {
  const currentTime = getCurrentTime();
  const startTime = currentTime;
  let timeout;
  switch (priorityLevel) {
    case ImmediatePriority:
      timeout = IMMEDIATE_PRIORITY_TIMEOUT;
      break;
    case UserBlockingPriority:
      timeout = USER_BLOCKING_PRIORITY_TIMEOUT;
      break;
    case IdlePriority:
```

```
      timeout = IDLE_PRIORITY_TIMEOUT;
      break;
    case LowPriority:
      timeout = LOW_PRIORITY_TIMEOUT;
      break;
    case NormalPriority:
    default:
      timeout = NORMAL_PRIORITY_TIMEOUT;
      break;
  }
  const expirationTime = startTime + timeout;
  const newTask = {
    id: taskIdCounter++,
    callback,
    priorityLevel,
    startTime,
    expirationTime,
    sort
  };
  newTask.sortIndex = expirationTime;
  push(taskQueue, newTask);
  requestHostCallback(flushWork);
  return newTask;
}

function flushWork(initialTime) {
  return workLoop(initialTime);
}

function shouldYieldToHost() {
  const timeElapsed = getCurrentTime() - startTime;
  if (timeElapsed < frameInterval) {
    return false;
  }
  return true;
}
function workLoop(initialTime) {
  let currentTime = initialTime;
  currentTask = peek(taskQueue);
  while (currentTask !== null) {
    if (currentTask.expirationTime > currentTime && shouldYieldToHost()) {
      break;
    }
    const callback = currentTask.callback;
    if (typeof callback
      currentTask.callback = null;
      const didUserCallbackTimeout = currentTask.expirationTime +function unstable_cancelCallback(task) {
+  task.callback = null;
+}

export {
  NormalPriority as unstable_NormalPriority,
  unstable_scheduleCallback,
  shouldYieldToHost as unstable_shouldYield,
+ unstable_cancelCallback,
+ getCurrentTime as unstable_now
};
```

41.3 Scheduler.js

src\react-reconciler\src\Scheduler.js

```
import * as Scheduler from 'scheduler'
export const scheduleCallback = Scheduler.unstable_scheduleCallback
export const NormalPriority = Scheduler.unstable_NormalPriority
export const ImmediatePriority = Scheduler.unstable_ImmediatePriority;
export const UserBlockingPriority = Scheduler.unstable_UserBlockingPriority;
export const LowPriority = Scheduler.unstable_LowPriority;
export const IdlePriority = Scheduler.unstable_IdlePriority;
export const shouldYield = Scheduler.unstable_shouldYield
+export const cancelCallback = Scheduler.unstable_cancelCallback
+export const now = Scheduler.unstable_now;
```

41.4 ReactFiberWorkLoop.js

src\react-reconciler\src\ReactFiberWorkLoop.js

```
import {
  scheduleCallback as Scheduler_scheduleCallback,
  ImmediatePriority as ImmediateSchedulerPriority,
  UserBlockingPriority as UserBlockingSchedulerPriority,
  NormalPriority as NormalSchedulerPriority,
  IdlePriority as IdleSchedulerPriority,
  shouldYield,
+ cancelCallback as Scheduler_cancelCallback,
+ now
} from "./Scheduler";
import { createWorkInProgress } from "./ReactFiber";
import { beginWork } from "./ReactFiberBeginWork";
import { completeWork } from "./ReactFiberCompleteWork";
import { MutationMask, NoFlags, Passive } from "./ReactFiberFlags";
import {
  commitMutationEffects,
  commitPassiveUnmountEffects,
  commitPassiveMountEffects,
  commitLayoutEffects,
} from "./ReactFiberCommitWork";
import { finishQueueingConcurrentUpdates } from "./ReactFiberConcurrentUpdates";
import {
  NoLane, markRootUpdated, NoLanes,
  getNextLanes, getHighestPriorityLane, SyncLane,
+ includesBlockingLane, markStarvedLanesAsExpired, includesExpiredLane,
+ mergeLanes, markRootFinished, NoTimestamp
```

```
} from './ReactFiberLane';
import {
  getCurrentUpdatePriority, lanesToEventPriority, DiscreteEventPriority, ContinuousEventPriority,
  DefaultEventPriority, IdleEventPriority, setCurrentUpdatePriority
} from './ReactEventPriorities';
import { getCurrentEventPriority } from 'react-dom-bindings/src/client/ReactDOMHostConfig';
import { scheduleSyncCallback, flushSyncCallbacks } from './ReactFiberSyncTaskQueue';

let workInProgress = null;
let rootDoesHavePassiveEffects = false;
let rootWithPendingPassiveEffects = null;
let workInProgressRootRenderLanes = NoLanes;

const RootInProgress = 0;
const RootCompleted = 5;
let workInProgressRoot = null;
let workInProgressRootExitStatus = RootInProgress;
+let currentEventTime = NoTimestamp;

+function cancelCallback(callbackNode) {
+  console.log('cancelCallback');
+  return Scheduler_cancelCallback(callbackNode);
+}

+export function scheduleUpdateOnFiber(root, fiber, lane, eventTime) {
  markRootUpdated(root, lane);
+ ensureRootIsScheduled(root, eventTime);
}
+function ensureRootIsScheduled(root, currentTime) {
  const existingCallbackNode = root.callbackNode;
+ markStarvedLanesAsExpired(root, currentTime);
  const nextLanes = getNextLanes(root, root
  if (nextLanes
    root.callbackNode = null;
    root.callbackPriority = NoLane;
    return;
  }
  const newCallbackPriority = getHighestPriorityLane(nextLanes);
  const existingCallbackPriority = root.callbackPriority;
  if (existingCallbackPriority
    return;
  }
  if (existingCallbackNode != null) {
    cancelCallback(existingCallbackNode);
  }
  let newCallbackNode;
  if (newCallbackPriority
    scheduleSyncCallback(performSyncWorkOnRoot.bind(null, root));
    queueMicrotask(flushSyncCallbacks);
    newCallbackNode = null;
  } else {
    let schedulerPriorityLevel;
    switch (lanesToEventPriority(nextLanes)) {
      case DiscreteEventPriority:
        schedulerPriorityLevel = ImmediateSchedulerPriority;
        break;
      case ContinuousEventPriority:
        schedulerPriorityLevel = UserBlockingSchedulerPriority;
        break;
      case DefaultEventPriority:
        schedulerPriorityLevel = NormalSchedulerPriority;
        break;
      case IdleEventPriority:
        schedulerPriorityLevel = IdleSchedulerPriority;
        break;
      default:
        schedulerPriorityLevel = NormalSchedulerPriority;
        break;
    }
    newCallbackNode = Scheduler_scheduleCallback(schedulerPriorityLevel, performConcurrentWorkOnRoot.bind(null, root))
  }
  root.callbackPriority = newCallbackPriority;
  root.callbackNode = newCallbackNode;
}
function performSyncWorkOnRoot(root) {
  const lanes = getNextLanes(root, NoLanes);
  renderRootSync(root, lanes);
  const finishedWork = root.current.alternate
  root.finishedWork = finishedWork
  commitRoot(root)
  return null;
}
function performConcurrentWorkOnRoot(root, didTimeout) {
  const originalCallbackNode = root.callbackNode;
  const lanes = getNextLanes(root, NoLanes);
  if (lanes
    return null;
  }
+ const nonIncludesBlockingLane = !includesBlockingLane(root, lanes);
+ const nonIncludesExpiredLane = !includesExpiredLane(root, lanes);
+ const nonTimeout = !didTimeout;
+ const shouldTimeSlice = nonIncludesBlockingLane && nonIncludesExpiredLane && nonTimeout;
  const exitStatus = shouldTimeSlice ? renderRootConcurrent(root, lanes) : renderRootSync(root, lanes);
  if (exitStatus !== RootInProgress) {
    const finishedWork = root.current.alternate
    root.finishedWork = finishedWork
    commitRoot(root)
  }
  if (root.callbackNode
    return performConcurrentWorkOnRoot.bind(null, root);
  }
  return null;
}
function renderRootConcurrent(root, lanes) {
```

```
 if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
    prepareFreshStack(root, lanes);
  }
  workLoopConcurrent();
  if (workInProgress !== null) {
    return RootInProgress;
  }
  workInProgressRoot = null;
  workInProgressRootRenderLanes = NoLanes;
  return workInProgressRootExitStatus;
}
function workLoopConcurrent() {
  while (workInProgress !== null && !shouldYield()) {
+   sleep(5)
    performUnitOfWork(workInProgress);
  }
}
export function flushPassiveEffects() {
  if (rootWithPendingPassiveEffects !== null) {
    const root = rootWithPendingPassiveEffects;
    commitPassiveUnmountEffects(root.current);
    commitPassiveMountEffects(root, root.current);
  }
}
function commitRoot(root) {
  const previousPriority = getCurrentUpdatePriority();
  try {
    setCurrentUpdatePriority(DiscreteEventPriority);
    commitRootImpl(root);
  } finally {
    setCurrentUpdatePriority(previousPriority);
  }
}
function commitRootImpl(root) {
  const { finishedWork } = root;
+ console.log('commit', finishedWork.child.memoizedState.memoizedState[0]);
  root.callbackNode = null;
  root.callbackPriority = NoLane;
+ const remainingLanes = mergeLanes(finishedWork.lanes, finishedWork.childLanes);
+ markRootFinished(root, remainingLanes);
  if ((finishedWork.subtreeFlags & Passive) !== NoFlags || (finishedWork.flags & Passive) !== NoFlags) {
    if (!rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = true;
      Scheduler_scheduleCallback(NormalSchedulerPriority, flushPassiveEffects);
    }
  }
  const subtreeHasEffects = (finishedWork.subtreeFlags & MutationMask) !== NoFlags;
  const rootHasEffect = (finishedWork.flags & MutationMask) !== NoFlags;
  if (subtreeHasEffects || rootHasEffect) {
    commitMutationEffects(finishedWork, root);
    commitLayoutEffects(finishedWork, root);
    root.current = finishedWork;
    if (rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = false;
      rootWithPendingPassiveEffects = root;
    }
  }
  root.current = finishedWork;
+ ensureRootIsScheduled(root, now());
}
function prepareFreshStack(root, lanes) {
  workInProgressRoot = root;
  workInProgress = createWorkInProgress(root.current, null);
  workInProgressRootRenderLanes = lanes;
  finishQueueingConcurrentUpdates();
}
function renderRootSync(root, lanes) {
  if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
    prepareFreshStack(root, lanes)
  }
  workLoopSync();
  workInProgressRoot = null;
  workInProgressRootRenderLanes = NoLanes;
  return workInProgressRootExitStatus;
}

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
function performUnitOfWork(unitOfWork) {
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork, workInProgressRootRenderLanes);
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}

function completeUnitOfWork(unitOfWork) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    completeWork(current, completedWork);
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    completedWork = returnFiber;
```

```
    workInProgress = completedWork;
  } while (completedWork !== null);
  if (workInProgressRootExitStatus
    workInProgressRootExitStatus = RootCompleted;
  }
}

export function requestUpdateLane() {
  const updateLane = getCurrentUpdatePriority();
  if (updateLane !== NoLane) {
    return updateLane;
  }
  const eventLane = getCurrentEventPriority();
  return eventLane;
}

+export function requestEventTime() {
+  currentEventTime = now();
+  return currentEventTime;
+}

+function sleep(time) {
+  const timeStamp = new Date().getTime();
+  const endTime = timeStamp + time;
+  while (true) {
+    if (new Date().getTime() > endTime) {
+      return;
+    }
+  }
+}
```

41.5 ReactFiberLane.js

src\react-reconciler\src\ReactFiberLane.js

```
import { allowConcurrentByDefault } from 'shared/ReactFeatureFlags';

+export const NoTimestamp = -1;

export const TotalLanes = 31;
export const NoLanes = 0b0000000000000000000000000000000;
export const NoLane = 0b0000000000000000000000000000000;
export const SyncLane = 0b0000000000000000000000000000001;// 1
export const InputContinuousLane = 0b0000000000000000000000000000100;// 4
export const DefaultLane = 0b0000000000000000000000000010000;// 16
export const NonIdleLanes = 0b0001111111111111111111111111111;
export const IdleLane = 0b0100000000000000000000000000000;

export function mergeLanes(a, b) {
  return a | b;
}
export function markRootUpdated(root, updateLane) {
  root.pendingLanes |= updateLane;
}

export function getNextLanes(root, wipLanes) {
  const pendingLanes = root.pendingLanes;
  if (pendingLanes
    return NoLanes;
  }
  const nextLanes = getHighestPriorityLanes(pendingLanes);
  if (wipLanes !== NoLanes && wipLanes !== nextLanes) {
    if (nextLanes >= wipLanes) {
      return wipLanes;
    }
  }
  return nextLanes;
}

function getHighestPriorityLanes(lanes) {
  return getHighestPriorityLane(lanes);
}

export function getHighestPriorityLane(lanes) {
  return lanes & -lanes;
}

export function includesNonIdleWork(lanes) {
  return (lanes & NonIdleLanes) !== NoLanes;
}
export function includesBlockingLane(root, lanes) {
  if (allowConcurrentByDefault) {
    return false;
  }
  const SyncDefaultLanes = InputContinuousLane | DefaultLane;
  return (lanes & SyncDefaultLanes) !== NoLanes;
}
export function isSubsetOfLanes(set, subset) {
  return (set & subset)
}

+function pickArbitraryLaneIndex(lanes) {
+  return 31 - Math.clz32(lanes);
+}
+
+export function markStarvedLanesAsExpired(root, currentTime) {
+  const pendingLanes = root.pendingLanes;
+  const expirationTimes = root.expirationTimes;
+  let lanes = pendingLanes
+  while (lanes > 0) {
+    const index = pickArbitraryLaneIndex(lanes);
+    const lane = 1 << index;
+    const expirationTime = expirationTimes[index];
+    if (expirationTime === NoTimestamp) {
```

```
+      expirationTimes[index] = computeExpirationTime(lane, currentTime);
+    } else if (expirationTime
+      root.expiredLanes |= lane;
+    }
+    lanes &= ~lane;
+  }
+}
+
+function computeExpirationTime(lane, currentTime) {
+  switch (lane) {
+    case SyncLane:
+    case InputContinuousLane:
+      return currentTime + 250;
+    case DefaultLane:
+      return currentTime + 5000;
+    case IdleLane:
+      return NoTimestamp;
+    default:
+      return NoTimestamp;
+  }
+}
+export function createLaneMap(initial) {
+  const laneMap = [];
+  for (let i = 0; i < TotalLanes; i++) {
+    laneMap.push(initial);
+  }
+  return laneMap;
+}
+export function includesExpiredLane(root, lanes) {
+  return (lanes & root.expiredLanes) !== NoLanes;
+}
+export function markRootFinished(root, remainingLanes) {
+  const noLongerPendingLanes = root.pendingLanes & ~remainingLanes;
+  root.pendingLanes = remainingLanes;
+  let lanes = noLongerPendingLanes;
+  const expirationTimes = root.expirationTimes;
+  while (lanes > 0) {
+    const index = pickArbitraryLaneIndex(lanes);
+    const lane = 1 << index;
+    expirationTimes[index] = NoTimestamp;
+    lanes &= ~lane;
+  }
+}
```

41.6 ReactFiber.js

src\react-reconciler\src\ReactFiber.js

```
import { HostRoot, IndeterminateComponent, HostComponent, HostText } from "./ReactWorkTags";
import { NoFlags } from "./ReactFiberFlags";
+import { NoLanes } from './ReactFiberLane';

export function FiberNode(tag, pendingProps, key) {
  this.tag = tag;
  this.key = key;
  this.type = null;
  this.stateNode = null;

  this.return = null;
  this.child = null;
  this.sibling = null;

  this.pendingProps = pendingProps;
  this.memoizedProps = null;
  this.updateQueue = null;
  this.memoizedState = null;

  this.flags = NoFlags;
  this.subtreeFlags = NoFlags;
  this.deletions = null;
  this.alternate = null;

  this.index = 0;
  this.ref = null;
+ this.lanes = NoLanes;
+ this.childLanes = NoLanes;
}
function createFiber(tag, pendingProps, key) {
  return new FiberNode(tag, pendingProps, key);
}
export function createHostRootFiber() {
  return createFiber(HostRoot, null, null);
}
// We use a double buffering pooling technique because we know that we'll
// only ever need at most two versions of a tree. We pool the "other" unused
// node that we're free to reuse. This is lazily created to avoid allocating
// extra objects for things that are never updated. It also allow us to
// reclaim the extra memory if needed.

//我们使用双缓冲池技术，因为我们知道一棵树最多只需要两个版本
//我们将"其他"未使用的我们可以自由重用的节点
//这是延迟创建的，以避免分配从未更新的内容的额外对象。它还允许我们如果需要，回收额外的内存
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.type = current.type;
    workInProgress.flags = NoFlags;
    workInProgress.subtreeFlags = NoFlags;
+   workInProgress.deletions = null;
  }
  workInProgress.child = current.child;
  workInProgress.memoizedProps = current.memoizedProps;
  workInProgress.memoizedState = current.memoizedState;
  workInProgress.updateQueue = current.updateQueue;
  workInProgress.sibling = current.sibling;
  workInProgress.index = current.index;
  workInProgress.ref = current.ref;
+ workInProgress.flags = current.flags;
+ workInProgress.childLanes = current.childLanes;
+ workInProgress.lanes = current.lanes;
  return workInProgress;
}
export function createFiberFromTypeAndProps(type, key, pendingProps) {
  let fiberTag = IndeterminateComponent;
  if (typeof type
    fiberTag = HostComponent;
  }
  const fiber = createFiber(fiberTag, pendingProps, key);
  fiber.type = type;
  return fiber;
}
export function createFiberFromElement(element) {
  const { type } = element;
  const { key } = element;
  const pendingProps = element.props;
  const fiber = createFiberFromTypeAndProps(type, key, pendingProps);
  return fiber;
}

export function createFiberFromText(content) {
  const fiber = createFiber(HostText, content, null);
  return fiber;
}
```

41.7 ReactFiberBeginWork.js

src\react-reconciler\src\ReactFiberBeginWork.js

```
import { HostRoot, HostComponent, HostText, IndeterminateComponent, FunctionComponent } from "./ReactWorkTags";
import { processUpdateQueue, cloneUpdateQueue } from "./ReactFiberClassUpdateQueue";
import { mountChildFibers, reconcileChildFibers } from "./ReactChildFiber";
import { shouldSetTextContent } from "react-dom-bindings/src/client/ReactDOMHostConfig";
import { renderWithHooks } from "react-reconciler/src/ReactFiberHooks";
+import { NoLanes } from './ReactFiberLane';

function reconcileChildren(current, workInProgress, nextChildren) {
  if (current
    workInProgress.child = mountChildFibers(workInProgress, null, nextChildren);
  } else {
    workInProgress.child = reconcileChildFibers(workInProgress, current.child, nextChildren);
  }
}
function updateHostRoot(current, workInProgress, renderLanes) {
  const nextProps = workInProgress.pendingProps;
  cloneUpdateQueue(current, workInProgress);
  processUpdateQueue(workInProgress, nextProps, renderLanes)
  const nextState = workInProgress.memoizedState;
  const nextChildren = nextState.element;
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
function updateHostComponent(current, workInProgress) {
  const { type } = workInProgress;
  const nextProps = workInProgress.pendingProps;
  let nextChildren = nextProps.children;
  const isDirectTextChild = shouldSetTextContent(type, nextProps);
  if (isDirectTextChild) {
    nextChildren = null;
  }
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
function mountIndeterminateComponent(_current, workInProgress, Component) {
  const props = workInProgress.pendingProps;
  const value = renderWithHooks(null, workInProgress, Component, props);
  workInProgress.tag = FunctionComponent;
  reconcileChildren(null, workInProgress, value);
  return workInProgress.child;
}
+function updateFunctionComponent(current, workInProgress, Component, nextProps, renderLanes) {
+ const nextChildren = renderWithHooks(current, workInProgress, Component, nextProps, renderLanes);
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
export function beginWork(current, workInProgress, renderLanes) {
+ workInProgress.lanes = NoLanes;
  switch (workInProgress.tag) {
    case IndeterminateComponent: {
      return mountIndeterminateComponent(current, workInProgress, workInProgress.type, renderLanes);
    }
    case FunctionComponent: {
      const Component = workInProgress.type;
      const resolvedProps = workInProgress.pendingProps;
+     return updateFunctionComponent(current, workInProgress, Component, resolvedProps, renderLanes);
    }
    case HostRoot:
      return updateHostRoot(current, workInProgress, renderLanes);
    case HostComponent:
      return updateHostComponent(current, workInProgress, renderLanes);
    case HostText:
    default:
      return null;
  }
}
```

41.8 ReactFiberCompleteWork.js

src\react-reconciler\src\ReactFiberCompleteWork.js

```
import {
  appendInitialChild,
  createInstance,
  createTextInstance,
  finalizeInitialChildren,
  prepareUpdate,
} from "react-dom-bindings/src/client/ReactDOMHostConfig";
import { HostComponent, HostRoot, HostText, FunctionComponent } from "./ReactWorkTags";
import { Ref, NoFlags, Update } from "./ReactFiberFlags";
+import { NoLanes, mergeLanes } from './ReactFiberLane';

function markRef(workInProgress) {
  workInProgress.flags |= Ref;
}

function bubbleProperties(completedWork) {
+ let newChildLanes = NoLanes;
  let subtreeFlags = NoFlags;
  let child = completedWork.child;
  while (child !== null) {
+   newChildLanes = mergeLanes(newChildLanes, mergeLanes(child.lanes, child.childLanes));
    subtreeFlags |= child.subtreeFlags;
    subtreeFlags |= child.flags;
    child = child.sibling;
  }
+ completedWork.childLanes = newChildLanes;
  completedWork.subtreeFlags |= subtreeFlags;
}

function appendAllChildren(parent, workInProgress) {
  // 我们只有创建的顶级fiber，但需要递归其子节点来查找所有终端节点
  let node = workInProgress.child;
  while (node !== null) {
```

```
    // 如果是原生节点，直接添加到父节点上
    if (node.tag
      appendInitialChild(parent, node.stateNode);
      // 再看看第一个节节点是不是原生节点
    } else if (node.child !== null) {
      // node.child.return = node
      node = node.child;
      continue;
    }
    if (node
      return;
    }
    // 如果没有弟弟就找父亲的弟弟
    while (node.sibling
      // 如果找到了根节点或者回到了原节点结束
      if (node.return
        return;
      }
      node = node.return;
    }
    // node.sibling.return = node.return
    // 下一个弟弟节点
    node = node.sibling;
  }
}
function markUpdate(workInProgress) {
  workInProgress.flags |= Update;
}
function updateHostComponent(current, workInProgress, type, newProps) {
  const oldProps = current.memoizedProps;
  const instance = workInProgress.stateNode;
  const updatePayload = prepareUpdate(instance, type, oldProps, newProps);
  workInProgress.updateQueue = updatePayload;
  if (updatePayload) {
    markUpdate(workInProgress);
  }
}
export function completeWork(current, workInProgress) {
  const newProps = workInProgress.pendingProps;
  switch (workInProgress.tag) {
    case HostComponent: {
      const { type } = workInProgress;
      if (current !== null && workInProgress.stateNode != null) {
        updateHostComponent(current, workInProgress, type, newProps);
        if (current.ref !== workInProgress.ref) {
          markRef(workInProgress);
        }
      } else {
        const instance = createInstance(type, newProps, workInProgress);
        appendAllChildren(instance, workInProgress);
        workInProgress.stateNode = instance;
        finalizeInitialChildren(instance, type, newProps);
        if (workInProgress.ref !== null) {
          markRef(workInProgress);
        }
      }
      bubbleProperties(workInProgress);
      return null;
    }
    case FunctionComponent:
      bubbleProperties(workInProgress);
      break;
    case HostRoot:
      bubbleProperties(workInProgress);
      break;
    case HostText: {
      const newText = newProps;
      workInProgress.stateNode = createTextInstance(newText);
      bubbleProperties(workInProgress);
      break;
    }
    default:
      break;
  }
}
```

41.9 ReactFiberConcurrentUpdates.js

src\react-reconciler\src\ReactFiberConcurrentUpdates.js

```
import { HostRoot } from "./ReactWorkTags";
+import { mergeLanes, NoLanes } from './ReactFiberLane';

const concurrentQueues = [];
let concurrentQueuesIndex = 0;

export function markUpdateLaneFromFiberToRoot(sourceFiber) {
  let node = sourceFiber;
  let parent = sourceFiber.return;
  while (parent !== null) {
    node = parent;
    parent = parent.return;
  }
  if (node.tag
    const root = node.stateNode;
    return root;
  }
  return null;
}
export function enqueueConcurrentHookUpdate(fiber, queue, update, lane) {
  enqueueUpdate(fiber, queue, update, lane);
  return getRootForUpdatedFiber(fiber);
}
export function enqueueConcurrentClassUpdate(fiber, queue, update, lane) {
  enqueueUpdate(fiber, queue, update, lane);
  return getRootForUpdatedFiber(fiber);
}

function enqueueUpdate(fiber, queue, update, lane) {
  concurrentQueues[concurrentQueuesIndex++] = fiber;
  concurrentQueues[concurrentQueuesIndex++] = queue;
  concurrentQueues[concurrentQueuesIndex++] = update;
  concurrentQueues[concurrentQueuesIndex++] = lane;
+ fiber.lanes = mergeLanes(fiber.lanes, lane);
}
function getRootForUpdatedFiber(sourceFiber) {
  let node = sourceFiber;
  let parent = node.return;
  while (parent !== null) {
    node = parent;
    parent = node.return;
  }
  return node.tag
}

export function finishQueueingConcurrentUpdates() {
  const endIndex = concurrentQueuesIndex;
  concurrentQueuesIndex = 0;
  let i = 0;
  while (i < endIndex) {
    const fiber = concurrentQueues[i++];
    const queue = concurrentQueues[i++];
    const update = concurrentQueues[i++];
    const lane = concurrentQueues[i++]
    if (queue !== null && update !== null) {
      const pending = queue.pending;
      if (pending
        update.next = update;
      } else {
        update.next = pending.next;
        pending.next = update;
      }
      queue.pending = update;
    }
  }
}
```

41.10 ReactFiberHooks.js

src\react-reconciler\src\ReactFiberHooks.js

```
import ReactSharedInternals from "shared/ReactSharedInternals";
import { enqueueConcurrentHookUpdate } from "./ReactFiberConcurrentUpdates";
+import { scheduleUpdateOnFiber, requestUpdateLane, requestEventTime } from "./ReactFiberWorkLoop";
import is from "shared/objectIs";
import { Passive as PassiveEffect, Update as UpdateEffect } from "./ReactFiberFlags";
import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout } from "./ReactHookEffectTags";
+import { NoLanes, NoLane, mergeLanes, isSubsetOfLanes } from './ReactFiberLane';

const { ReactCurrentDispatcher } = ReactSharedInternals;
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;
+let renderLanes = NoLanes;

const HooksDispatcherOnMountInDEV = {
  useReducer: mountReducer,
  useState: mountState,
  useEffect: mountEffect,
  useLayoutEffect: mountLayoutEffect,
  useRef: mountRef,
};
const HooksDispatcherOnUpdateInDEV = {
  useReducer: updateReducer,
  useState: updateState,
  useEffect: updateEffect,
  useLayoutEffect: updateLayoutEffect,
  useRef: updateRef
};
function mountRef(initialValue) {
  const hook = mountWorkInProgressHook();
  const ref = {
    current: initialValue,
  };
```

```
  hook.memoizedState = ref;
  return ref;
}
function updateRef() {
  const hook = updateWorkInProgressHook();
  return hook.memoizedState;
}
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) {
  const fiberFlags = UpdateEffect;
  return mountEffectImpl(fiberFlags, HookLayout, create, deps);
}
function updateEffect(create, deps) {
  return updateEffectImpl(PassiveEffect, HookPassive, create, deps);
}
function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
  const hook = updateWorkInProgressHook();
  const nextDeps = deps
  let destroy;
  if (currentHook !== null) {
    const prevEffect = currentHook.memoizedState;
    destroy = prevEffect.destroy;
    if (nextDeps !== null) {
      const prevDeps = prevEffect.deps;
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        hook.memoizedState = 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 (is(nextDeps[i], prevDeps[i])) {
      continue;
    }
    return false;
  }

  return true;
}
function mountEffect(create, deps) {
  return mountEffectImpl(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,
  };
}
function basicStateReducer(state, action) {
  return typeof action
}
function mountReducer(reducer, initialArg) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialArg;
  const queue = {
    pending: null,
    dispatch: null,
+   lastRenderedReducer: reducer,
+   lastRenderedState: initialArg
  };
  hook.queue = queue;
```

```
  const dispatch = (queue.dispatch = dispatchReducerAction.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function updateReducer(reducer) {
+  const hook = updateWorkInProgressHook();
+  const queue = hook.queue;
+  queue.lastRenderedReducer = reducer;
+  const current = currentHook;
+  let baseQueue = current.baseQueue;
+  const pendingQueue = queue.pending;
+  if (pendingQueue !== null) {
+    if (baseQueue !== null) {
+      const baseFirst = baseQueue.next;
+      const pendingFirst = pendingQueue.next;
+      baseQueue.next = pendingFirst;
+      pendingQueue.next = baseFirst;
+    }
+    current.baseQueue = baseQueue = pendingQueue;
+    queue.pending = null;
+  }
+  if (baseQueue !== null) {
+    printQueue(baseQueue)
+    const first = baseQueue.next;
+    let newState = current.baseState;
+    let newBaseState = null;
+    let newBaseQueueFirst = null;
+    let newBaseQueueLast = null;
+    let update = first;
+    do {
+      const updateLane = update.lane;
+      const shouldSkipUpdate = !isSubsetOfLanes(renderLanes, updateLane);
+      if (shouldSkipUpdate) {
+        const clone = {
+          lane: updateLane,
+          action: update.action,
+          hasEagerState: update.hasEagerState,
+          eagerState: update.eagerState,
+          next: null,
+        };
+        if (newBaseQueueLast === null) {
+          newBaseQueueFirst = newBaseQueueLast = clone;
+          newBaseState = newState;
+        } else {
+          newBaseQueueLast = newBaseQueueLast.next = clone;
+        }
+        currentlyRenderingFiber.lanes = mergeLanes(currentlyRenderingFiber.lanes, updateLane);
+      } else {
+        if (newBaseQueueLast !== null) {
+          const clone = {
+            lane: NoLane,
+            action: update.action,
+            hasEagerState: update.hasEagerState,
+            eagerState: update.eagerState,
+            next: null,
+          };
+          newBaseQueueLast = newBaseQueueLast.next = clone;
+        }
+        if (update.hasEagerState) {
+          newState = update.eagerState;
+        } else {
+          const action = update.action;
+          newState = reducer(newState, action);
+        }
+      }
+      update = update.next;
+    } while (update !== null && update !== first);
+    if (newBaseQueueLast === null) {
+      newBaseState = newState;
+    } else {
+      newBaseQueueLast.next = newBaseQueueFirst;
+    }
+    hook.memoizedState = newState;
+    hook.baseState = newBaseState;
+    hook.baseQueue = newBaseQueueLast;
+    queue.lastRenderedState = newState;
+  }
+  if (baseQueue === null) {
+    queue.lanes = NoLanes;
+  }
+  const dispatch = queue.dispatch;
+  return [hook.memoizedState, dispatch];
}
function mountState(initialState) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialState;
  const queue = {
    pending: null,
    dispatch: null,
    lastRenderedReducer: basicStateReducer,
    lastRenderedState: initialState,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchSetState.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function dispatchSetState(fiber, queue, action) {
  const lane = requestUpdateLane(fiber);
  const update = {
    lane,
    action,
    hasEagerState: false,
    eagerState: null,
    next: null,
  };
```

```
  const alternate = fiber.alternate;
  if (fiber.lanes
    const lastRenderedReducer = queue.lastRenderedReducer;
    const currentState = queue.lastRenderedState;
    const eagerState = lastRenderedReducer(currentState, action);
    update.hasEagerState = true;
    update.eagerState = eagerState;
    if (is(eagerState, currentState)) {
      return;
    }
  }
  const root = enqueueConcurrentHookUpdate(fiber, queue, update, lane);
+ const eventTime = requestEventTime();
+ scheduleUpdateOnFiber(root, fiber, lane, eventTime);
}
function updateState(initialState) {
  return updateReducer(basicStateReducer, initialState);
}
function mountWorkInProgressHook() {
  const hook = {
    memoizedState: null,
    queue: null,
    next: null,
+   baseState: null,
+   baseQueue: null,
  };
  if (workInProgressHook
    currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
  } else {
    workInProgressHook = workInProgressHook.next = hook;
  }
  return workInProgressHook;
}
function dispatchReducerAction(fiber, queue, action) {
  const update = {
    action,
    next: null,
  };
  const root = enqueueConcurrentHookUpdate(fiber, queue, update);
  scheduleUpdateOnFiber(root, fiber);
}

function updateWorkInProgressHook() {
  let nextCurrentHook;
  if (currentHook
    const current = currentlyRenderingFiber.alternate;
    if (current !== null) {
      nextCurrentHook = current.memoizedState;
    } else {
      nextCurrentHook = null;
    }
  } else {
    nextCurrentHook = currentHook.next;
  }

  let nextWorkInProgressHook;
  if (workInProgressHook
    nextWorkInProgressHook = currentlyRenderingFiber.memoizedState;
  } else {
    nextWorkInProgressHook = workInProgressHook.next;
  }

  if (nextWorkInProgressHook !== null) {
    workInProgressHook = nextWorkInProgressHook;
    nextWorkInProgressHook = workInProgressHook.next;
    currentHook = nextCurrentHook;
  } else {
    currentHook = nextCurrentHook;
    const newHook = {
      memoizedState: currentHook.memoizedState,
      queue: currentHook.queue,
      next: null,
+     baseState: currentHook.baseState,
+     baseQueue: currentHook.baseQueue,
    };
    if (workInProgressHook
      currentlyRenderingFiber.memoizedState = workInProgressHook = newHook;
    } else {
      workInProgressHook = workInProgressHook.next = newHook;
    }
  }
  return workInProgressHook;
}

export function renderWithHooks(current, workInProgress, Component, props, nextRenderLanes) {
+ renderLanes = nextRenderLanes;
  currentlyRenderingFiber = workInProgress;
  workInProgress.updateQueue = null;
+ workInProgress.memoizedState = null;
  if (current !== null && current.memoizedState !== null) {
    ReactCurrentDispatcher.current = HooksDispatcherOnUpdateInDEV;
  } else {
    ReactCurrentDispatcher.current = HooksDispatcherOnMountInDEV;
  }
  const children = Component(props);
  currentlyRenderingFiber = null;
  workInProgressHook = null;
  currentHook = null;
+ renderLanes = NoLanes;
  return children;
}

+function printQueue(queue) {
+  const first = queue.next;
```

```
+  let desc = '';
+  let update = first;
+  do {
+    desc += ("=>" + (update.action.id));
+    update = update.next;
+  } while (update !== null && update !== first);
+  desc += "=>null";
+  console.log(desc);
+}
```

41.11 ReactFiberReconciler.js

src\react-reconciler\src\ReactFiberReconciler.js

```
import { createFiberRoot } from "./ReactFiberRoot";
import { createUpdate, enqueueUpdate } from "./ReactFiberClassUpdateQueue";
+import { scheduleUpdateOnFiber, requestUpdateLane, requestEventTime } from "./ReactFiberWorkLoop";
export function createContainer(containerInfo) {
  return createFiberRoot(containerInfo);
}
export function updateContainer(element, container) {
  const current = container.current;
+ const eventTime = requestEventTime();
  const lane = requestUpdateLane(current);
  const update = createUpdate(lane);
  update.payload = { element };
  const root = enqueueUpdate(current, update, lane);
+ scheduleUpdateOnFiber(root, current, lane, eventTime);
}
```

41.12 ReactFiberRoot.js

src\react-reconciler\src\ReactFiberRoot.js

```
import { createHostRootFiber } from "./ReactFiber";
import { initializeUpdateQueue } from "./ReactFiberClassUpdateQueue";
+import { NoTimestamp, createLaneMap, NoLanes } from 'react-reconciler/src/ReactFiberLane';

function FiberRootNode(containerInfo) {
  this.containerInfo = containerInfo;
+ this.expirationTimes = createLaneMap(NoTimestamp);
+ this.expiredLanes = NoLanes;
}

export function createFiberRoot(containerInfo) {
  const root = new FiberRootNode(containerInfo);
  const uninitializedFiber = createHostRootFiber();
  root.current = uninitializedFiber;
  uninitializedFiber.stateNode = root;
  initializeUpdateQueue(uninitializedFiber);
  return root;
}
```

42.context

42.1 src\main.jsx

src\main.jsx

```
import * as React from "react";
import { createRoot } from "react-dom/client";

+const NameContext = React.createContext('');
+const AgeContext = React.createContext('');
+
+function Child() {
+  const name = React.useContext(NameContext);
+  const age = React.useContext(AgeContext);
+  return {name + age}
+}
+function App() {
+  const [name, setName] = React.useState('a');
+  const [age, setAge] = React.useState('1');
+  return (
+
+       {
+        setName(name + 'a')
+      }}>setName
+       {
+        setAge(age + '1')
+      }}>setAge
+
+
+
+
+
+
+  )
+}

const element = ;
const container = document.getElementById("root");
const root = createRoot(container, { unstable_concurrentUpdatesByDefault: true });
root.render(element);
```

42.2 react\index.js

src\react\index.js

```
export {
  __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED,
  useReducer,
  useState,
  useEffect,
  useLayoutEffect,
  useRef,
+ createContext,
+ useContext,
} from "./src/React";
```

42.3 React.js

src\react\src\React.js

```
+import { useReducer, useState, useEffect, useLayoutEffect, useRef, useContext } from "./ReactHooks";
import ReactSharedInternals from "./ReactSharedInternals";
+import { createContext } from './ReactContext';

export {
  useReducer,
  useState,
  useEffect,
  useLayoutEffect,
  useRef,
+ createContext,
+ useContext,
  ReactSharedInternals as __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED,
};
```

42.4 ReactContext.js

src\react\src\ReactContext.js

```
import { REACT_PROVIDER_TYPE, REACT_CONTEXT_TYPE } from 'shared/ReactSymbols';

export function createContext(defaultValue) {
  const context = {
    $typeof: REACT_CONTEXT_TYPE,
    _currentValue: defaultValue,
    Provider: null
  };
  context.Provider = {
    $typeof: REACT_PROVIDER_TYPE,
    _context: context
  };
  return context;
}
```

42.5 ReactHooks.js

src\react\src\ReactHooks.js

```
import ReactCurrentDispatcher from "./ReactCurrentDispatcher";

function resolveDispatcher() {
  const dispatcher = ReactCurrentDispatcher.current;
  return dispatcher;
}

export function useReducer(reducer, initialArg, init) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useReducer(reducer, initialArg, init);
}

export function useState(initialState) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useState(initialState);
}

export function useEffect(create, deps) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useEffect(create, deps);
}
export function useLayoutEffect(create, deps) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useLayoutEffect(create, deps);
}
export function useRef(initialValue) {
  const dispatcher = resolveDispatcher();
  return dispatcher.useRef(initialValue);
}
+export function useContext(Context) {
+  const dispatcher = resolveDispatcher();
+  return dispatcher.useContext(Context);
+}
```

42.6 ReactFiber.js

src\react-reconciler\src\ReactFiber.js

```
+import { HostRoot, IndeterminateComponent, HostComponent, HostText, ContextProvider } from "./ReactWorkTags";
import { NoFlags } from "./ReactFiberFlags";
import { NoLanes } from './ReactFiberLane';
+import { REACT_PROVIDER_TYPE } from 'shared/ReactSymbols';

export function FiberNode(tag, pendingProps, key) {
  this.tag = tag;
  this.key = key;
  this.type = null;
  this.stateNode = null;

  this.return = null;
  this.child = null;
  this.sibling = null;
```

```
  this.pendingProps = pendingProps;
  this.memoizedProps = null;
  this.updateQueue = null;
  this.memoizedState = null;

  this.flags = NoFlags;
  this.subtreeFlags = NoFlags;
  this.deletions = null;
  this.alternate = null;

  this.index = 0;
  this.ref = null;
  this.lanes = NoLanes;
  this.childLanes = NoLanes;
}
function createFiber(tag, pendingProps, key) {
  return new FiberNode(tag, pendingProps, key);
}
export function createHostRootFiber() {
  return createFiber(HostRoot, null, null);
}
// We use a double buffering pooling technique because we know that we'll
// only ever need at most two versions of a tree. We pool the "other" unused
// node that we're free to reuse. This is lazily created to avoid allocating
// extra objects for things that are never updated. It also allow us to
// reclaim the extra memory if needed.

//我们使用双缓冲池技术，因为我们知道一棵树最多只需要两个版本
//我们将"其他"未使用的我们可以自由重用的节点
//这是延迟创建的，以避免分配从未更新的内容的额外对象。它还允许我们如果需要，回收额外的内存
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.type = current.type;
    workInProgress.flags = NoFlags;
    workInProgress.subtreeFlags = NoFlags;
    workInProgress.deletions = null;
  }
  workInProgress.child = current.child;
  workInProgress.memoizedProps = current.memoizedProps;
  workInProgress.memoizedState = current.memoizedState;
  workInProgress.updateQueue = current.updateQueue;
  workInProgress.sibling = current.sibling;
  workInProgress.index = current.index;
  workInProgress.ref = current.ref;
  workInProgress.flags = current.flags;
  workInProgress.childLanes = current.childLanes;
  workInProgress.lanes = current.lanes;
  return workInProgress;
}
export function createFiberFromTypeAndProps(type, key, pendingProps) {
  let fiberTag = IndeterminateComponent;
  if (typeof type
    fiberTag = HostComponent;
  } else {
+   getTag: switch (type) {
+     default:
+       {
+         if (typeof type === 'object' && type !== null) {
+           switch (type.$typeof) {
+             case REACT_PROVIDER_TYPE:
+               fiberTag = ContextProvider;
+               break getTag;
+             default:
+               break;
+           }
+         }
+       }
+   }
  }
  const fiber = createFiber(fiberTag, pendingProps, key);
  fiber.type = type;
  return fiber;
}
export function createFiberFromElement(element) {
  const { type } = element;
  const { key } = element;
  const pendingProps = element.props;
  const fiber = createFiberFromTypeAndProps(type, key, pendingProps);
  return fiber;
}

export function createFiberFromText(content) {
  const fiber = createFiber(HostText, content, null);
  return fiber;
}
```

42.7 ReactFiberBeginWork.js

src\react-reconciler\src\ReactFiberBeginWork.js

```
import {
  HostRoot, HostComponent, HostText, IndeterminateComponent,
+ FunctionComponent, ContextProvider
} from "./ReactWorkTags";
import { processUpdateQueue, cloneUpdateQueue } from "./ReactFiberClassUpdateQueue";
import { mountChildFibers, reconcileChildFibers } from "./ReactChildFiber";
import { shouldSetTextContent } from "react-dom-bindings/src/client/ReactDOMHostConfig";
import { renderWithHooks } from "react-reconciler/src/ReactFiberHooks";
import { NoLanes } from './ReactFiberLane';
+import { pushProvider } from './ReactFiberNewContext';

function reconcileChildren(current, workInProgress, nextChildren) {
  if (current
    workInProgress.child = mountChildFibers(workInProgress, null, nextChildren);
  } else {
    workInProgress.child = reconcileChildFibers(workInProgress, current.child, nextChildren);
  }
}
function updateHostRoot(current, workInProgress, renderLanes) {
  const nextProps = workInProgress.pendingProps;
  cloneUpdateQueue(current, workInProgress);
  processUpdateQueue(workInProgress, nextProps, renderLanes)
  const nextState = workInProgress.memoizedState;
  const nextChildren = nextState.element;
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
function updateHostComponent(current, workInProgress) {
  const { type } = workInProgress;
  const nextProps = workInProgress.pendingProps;
  let nextChildren = nextProps.children;
  const isDirectTextChild = shouldSetTextContent(type, nextProps);
  if (isDirectTextChild) {
    nextChildren = null;
  }
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
function mountIndeterminateComponent(_current, workInProgress, Component) {
  const props = workInProgress.pendingProps;
  const value = renderWithHooks(null, workInProgress, Component, props);
  workInProgress.tag = FunctionComponent;
  reconcileChildren(null, workInProgress, value);
  return workInProgress.child;
}
function updateFunctionComponent(current, workInProgress, Component, nextProps, renderLanes) {
  const nextChildren = renderWithHooks(current, workInProgress, Component, nextProps, renderLanes);
  reconcileChildren(current, workInProgress, nextChildren);
  return workInProgress.child;
}
export function beginWork(current, workInProgress, renderLanes) {
  workInProgress.lanes = NoLanes;
  switch (workInProgress.tag) {
    case IndeterminateComponent: {
      return mountIndeterminateComponent(current, workInProgress, workInProgress.type, renderLanes);
    }
    case FunctionComponent: {
      const Component = workInProgress.type;
      const resolvedProps = workInProgress.pendingProps;
      return updateFunctionComponent(current, workInProgress, Component, resolvedProps, renderLanes);
    }
    case HostRoot:
      return updateHostRoot(current, workInProgress, renderLanes);
    case HostComponent:
      return updateHostComponent(current, workInProgress, renderLanes);
    case HostText:
      return null;
+   case ContextProvider:
+     return updateContextProvider(current, workInProgress, renderLanes);
    default:
      return null;
  }
}

+function updateContextProvider(current, workInProgress, renderLanes) {
+  const providerType = workInProgress.type;
+  const context = providerType._context;
+  const newProps = workInProgress.pendingProps;
+  const newValue = newProps.value;
+  pushProvider(context, newValue);
+  const newChildren = newProps.children;
+  reconcileChildren(current, workInProgress, newChildren, renderLanes);
+  return workInProgress.child;
+}
```

42.8 ReactFiberCompleteWork.js

src\react-reconciler\src\ReactFiberCompleteWork.js

```
import {
  appendInitialChild,
  createInstance,
  createTextInstance,
  finalizeInitialChildren,
  prepareUpdate,
} from "react-dom-bindings/src/client/ReactDOMHostConfig";
+import { HostComponent, HostRoot, HostText, FunctionComponent, ContextProvider } from "./ReactWorkTags";
import { Ref, NoFlags, Update } from "./ReactFiberFlags";
import { NoLanes, mergeLanes } from './ReactFiberLane';

function markRef(workInProgress) {
  workInProgress.flags |= Ref;
}
```

```
function bubbleProperties(completedWork) {
  let newChildLanes = NoLanes;
  let subtreeFlags = NoFlags;
  let child = completedWork.child;
  while (child !== null) {
    newChildLanes = mergeLanes(newChildLanes, mergeLanes(child.lanes, child.childLanes));
    subtreeFlags |= child.subtreeFlags;
    subtreeFlags |= child.flags;
    child = child.sibling;
  }
  completedWork.childLanes = newChildLanes;
  completedWork.subtreeFlags |= subtreeFlags;
}

function appendAllChildren(parent, workInProgress) {
  // 我们只有创建的顶级fiber，但需要递归其子节点来查找所有终端节点
  let node = workInProgress.child;
  while (node !== null) {
    // 如果是原生节点，直接添加到父节点上
    if (node.tag
      appendInitialChild(parent, node.stateNode);
      // 再看看第一个节节点是不是原生节点
    } else if (node.child !== null) {
      // node.child.return = node
      node = node.child;
      continue;
    }
    if (node
      return;
    }
    // 如果没有弟弟就找父亲的弟弟
    while (node.sibling
      // 如果找到了根节点或者回到了原节点结束
      if (node.return
        return;
      }
      node = node.return;
    }
    // node.sibling.return = node.return
    // 下一个弟弟节点
    node = node.sibling;
  }
}
function markUpdate(workInProgress) {
  workInProgress.flags |= Update;
}
function updateHostComponent(current, workInProgress, type, newProps) {
  const oldProps = current.memoizedProps;
  const instance = workInProgress.stateNode;
  const updatePayload = prepareUpdate(instance, type, oldProps, newProps);
  workInProgress.updateQueue = updatePayload;
  if (updatePayload) {
    markUpdate(workInProgress);
  }
}
export function completeWork(current, workInProgress) {
  const newProps = workInProgress.pendingProps;
  switch (workInProgress.tag) {
    case HostComponent: {
      const { type } = workInProgress;
      if (current !== null && workInProgress.stateNode != null) {
        updateHostComponent(current, workInProgress, type, newProps);
        if (current.ref !== workInProgress.ref) {
          markRef(workInProgress);
        }
      } else {
        const instance = createInstance(type, newProps, workInProgress);
        appendAllChildren(instance, workInProgress);
        workInProgress.stateNode = instance;
        finalizeInitialChildren(instance, type, newProps);
        if (workInProgress.ref !== null) {
          markRef(workInProgress);
        }
      }
      bubbleProperties(workInProgress);
      return null;
    }
    case FunctionComponent:
      bubbleProperties(workInProgress);
      break;
    case HostRoot:
      bubbleProperties(workInProgress);
      break;
    case HostText: {
      const newText = newProps;
      workInProgress.stateNode = createTextInstance(newText);
      bubbleProperties(workInProgress);
      break;
    }
+   case ContextProvider: {
+     bubbleProperties(workInProgress);
+     break;
+   }
    default:
      break;
  }
}
```

42.9 ReactFiberCommitWork.js

src\react-reconciler\src\ReactFiberCommitWork.js

```
+import { HostRoot, HostComponent, HostText, FunctionComponent, ContextProvider } from "./ReactWorkTags";
```

```
import { Passive, MutationMask, Placement, Update, LayoutMask, Ref } from "./ReactFiberFlags";
import { insertBefore, appendChild, commitUpdate, removeChild } from "react-dom-bindings/src/client/ReactDOMHostConfig";
import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout } from "./ReactHookEffectTags";

export function commitMutationEffects(finishedWork, root) {
  commitMutationEffectsOnFiber(finishedWork, root);
}
export function commitPassiveUnmountEffects(finishedWork) {
  commitPassiveUnmountOnFiber(finishedWork);
}
function commitPassiveUnmountOnFiber(finishedWork) {
  switch (finishedWork.tag) {
    case FunctionComponent: {
      recursivelyTraversePassiveUnmountEffects(finishedWork);
      if (finishedWork.flags & Passive) {
        commitHookPassiveUnmountEffects(finishedWork, finishedWork.return, HookPassive | HookHasEffect);
      }
      break;
    }
    default: {
      recursivelyTraversePassiveUnmountEffects(finishedWork);
      break;
    }
  }
}
function recursivelyTraversePassiveUnmountEffects(parentFiber) {
  if (parentFiber.subtreeFlags & Passive) {
    let child = parentFiber.child;
    while (child !== null) {
      commitPassiveUnmountOnFiber(child);
      child = child.sibling;
    }
  }
}
function commitHookPassiveUnmountEffects(finishedWork, nearestMountedAncestor, hookFlags) {
  commitHookEffectListUnmount(hookFlags, finishedWork, nearestMountedAncestor);
}

function commitHookEffectListUnmount(flags, 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 & flags)
        const destroy = effect.destroy;
        effect.destroy = undefined;
        if (destroy !== undefined) {
          destroy();
        }
      }
      effect = effect.next;
    } while (effect !== firstEffect);
  }
}

export function commitPassiveMountEffects(root, finishedWork) {
  commitPassiveMountOnFiber(root, finishedWork);
}
function commitPassiveMountOnFiber(finishedRoot, finishedWork) {
  const flags = finishedWork.flags;
  switch (finishedWork.tag) {
    case FunctionComponent: {
      recursivelyTraversePassiveMountEffects(finishedRoot, finishedWork);
      if (flags & Passive) {
        commitHookPassiveMountEffects(finishedWork, HookPassive | HookHasEffect);
      }
      break;
    }
    case HostRoot: {
      recursivelyTraversePassiveMountEffects(finishedRoot, finishedWork);
      break;
    }
    default:
      break;
  }
}
function commitHookPassiveMountEffects(finishedWork, hookFlags) {
  commitHookEffectListMount(hookFlags, finishedWork);
}
function commitHookEffectListMount(flags, 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 & flags)
        const create = effect.create;
        effect.destroy = create();
      }
      effect = effect.next;
    } while (effect !== firstEffect);
  }
}
function recursivelyTraversePassiveMountEffects(root, parentFiber) {
  if (parentFiber.subtreeFlags & Passive) {
    let child = parentFiber.child;
    while (child !== null) {
      commitPassiveMountOnFiber(root, child);
      child = child.sibling;
    }
  }
```

```
}
let hostParent = null;
function commitDeletionEffects(root, returnFiber, deletedFiber) {
  let parent = returnFiber;
  findParent: while (parent !== null) {
    switch (parent.tag) {
      case HostComponent: {
        hostParent = parent.stateNode;
        break findParent;
      }
      case HostRoot: {
        hostParent = parent.stateNode.containerInfo;
        break findParent;
      }
      default:
        break;
    }
    parent = parent.return;
  }
  commitDeletionEffectsOnFiber(root, returnFiber, deletedFiber);
  hostParent = null;
}
function commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor, deletedFiber) {
  switch (deletedFiber.tag) {
    case HostComponent:
    case HostText: {
      recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor, deletedFiber);
      if (hostParent !== null) {
        removeChild(hostParent, deletedFiber.stateNode);
      }
      break;
    }
    default:
      break;
  }
}
function recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor, parent) {
  let child = parent.child;
  while (child !== null) {
    commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor, child);
    child = child.sibling;
  }
}
function recursivelyTraverseMutationEffects(root, parentFiber) {
  const deletions = parentFiber.deletions;
  if (deletions !== null) {
    for (let i = 0; i < deletions.length; i++) {
      const childToDelete = deletions[i];
      commitDeletionEffects(root, parentFiber, childToDelete);
    }
  }
  if (parentFiber.subtreeFlags & MutationMask) {
    let { child } = parentFiber;
    while (child !== null) {
      commitMutationEffectsOnFiber(child, root);
      child = child.sibling;
    }
  }
}
function isHostParent(fiber) {
  return fiber.tag
}
function getHostParentFiber(fiber) {
  let parent = fiber.return;
  while (parent !== null) {
    if (isHostParent(parent)) {
      return parent;
    }
    parent = parent.return;
  }
  return parent;
}
function insertOrAppendPlacementNode(node, before, parent) {
  const { tag } = node;
  const isHost = tag
  if (isHost) {
    const { stateNode } = node;
    if (before) {
      insertBefore(parent, stateNode, before);
    } else {
      appendChild(parent, stateNode);
    }
  } else {
    const { child } = node;
    if (child !== null) {
      insertOrAppendPlacementNode(child, before, parent);
      let { sibling } = child;
      while (sibling !== null) {
        insertOrAppendPlacementNode(sibling, before, parent);
        sibling = sibling.sibling;
      }
    }
  }
}
function getHostSibling(fiber) {
  let node = fiber;
  siblings: while (true) {
    // 如果我们没有找到任何东西，让我们试试下一个弟弟
    while (node.sibling
      if (node.return
        // 如果我们是根Fiber或者父亲是原生节点，我们就是最后的弟弟
        return null;
      }
      node = node.return;
```

```
    }
    // node.sibling.return = node.return
    node = node.sibling;
    while (node.tag !== HostComponent && node.tag !== HostText) {
      // 如果它不是原生节点，并且，我们可能在其中有一个原生节点
      // 试着向下搜索，直到找到为止
      if (node.flags & Placement) {
        // 如果我们没有孩子，可以试试弟弟
        continue siblings;
      } else {
        // node.child.return = node
        node = node.child;
      }
    } // Check if this host node is stable or about to be placed.

    // 检查此原生节点是否稳定可以放置
    if (!(node.flags & Placement)) {
      // 找到它了!

      return node.stateNode;
    }
  }
}
function commitPlacement(finishedWork) {
  const parentFiber = getHostParentFiber(finishedWork);
  switch (parentFiber.tag) {
    case HostComponent: {
      const parent = parentFiber.stateNode;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    case HostRoot: {
      const parent = parentFiber.stateNode.containerInfo;
      const before = getHostSibling(finishedWork);
      insertOrAppendPlacementNode(finishedWork, before, parent);
      break;
    }
    default:
      break;
  }
}
function commitReconciliationEffects(finishedWork) {
  const { flags } = finishedWork;
  if (flags & Placement) {
    commitPlacement(finishedWork);
    finishedWork.flags &= ~Placement;
  }
}
export function commitMutationEffectsOnFiber(finishedWork, root) {
  const current = finishedWork.alternate;
  const flags = finishedWork.flags;
  switch (finishedWork.tag) {
    case HostRoot: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
    case FunctionComponent: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      if (flags & Update) {
        commitHookEffectListUnmount(HookLayout | HookHasEffect, finishedWork, finishedWork.return);
      }
      break;
    }
    case HostComponent: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      if (flags & Ref) {
        commitAttachRef(finishedWork);
      }
      if (flags & Update) {
        const instance = finishedWork.stateNode;
        if (instance != null) {
          const newProps = finishedWork.memoizedProps;
          const oldProps = current !== null ? current.memoizedProps : newProps;
          const type = finishedWork.type;
          const updatePayload = finishedWork.updateQueue;
          finishedWork.updateQueue = null;
          if (updatePayload !== null) {
            commitUpdate(instance, updatePayload, type, oldProps, newProps, finishedWork);
          }
        }
      }

      break;
    }
    case HostText: {
      recursivelyTraverseMutationEffects(root, finishedWork);
      commitReconciliationEffects(finishedWork);
      break;
    }
+   case ContextProvider: {
+     recursivelyTraverseMutationEffects(root, finishedWork);
+     commitReconciliationEffects(finishedWork);
+     break;
    }
    default: {
      break;
    }
  }
}
function commitAttachRef(finishedWork) {
```

```
  const ref = finishedWork.ref;
  if (ref !== null) {
    const instance = finishedWork.stateNode;
    if (typeof ref
      ref(instance)
    } else {
      ref.current = instance;
    }
  }
}
export function commitLayoutEffects(finishedWork, root) {
  const current = finishedWork.alternate;
  commitLayoutEffectOnFiber(root, current, finishedWork);
}
function commitLayoutEffectOnFiber(finishedRoot, current, finishedWork) {
  const flags = finishedWork.flags;
  switch (finishedWork.tag) {
    case FunctionComponent: {
      recursivelyTraverseLayoutEffects(finishedRoot, finishedWork);
      if (flags & Update) {
        commitHookLayoutEffects(finishedWork, HookLayout | HookHasEffect);
      }
      break;
    }
    case HostRoot: {
      recursivelyTraverseLayoutEffects(finishedRoot, finishedWork);
      break;
    }
    default:
      break;
  }
}
function recursivelyTraverseLayoutEffects(root, parentFiber) {
  if (parentFiber.subtreeFlags & LayoutMask) {
    let child = parentFiber.child;
    while (child !== null) {
      const current = child.alternate;
      commitLayoutEffectOnFiber(root, current, child);
      child = child.sibling;
    }
  }
}
function commitHookLayoutEffects(finishedWork, hookFlags) {
  commitHookEffectListMount(hookFlags, finishedWork);
}
```

42.10 ReactFiberHooks.js

src\react-reconciler\src\ReactFiberHooks.js

```
import ReactSharedInternals from "shared/ReactSharedInternals";
import { enqueueConcurrentHookUpdate } from "./ReactFiberConcurrentUpdates";
import { scheduleUpdateOnFiber, requestUpdateLane, requestEventTime } from "./ReactFiberWorkLoop";
import is from "shared/objectIs";
import { Passive as PassiveEffect, Update as UpdateEffect } from "./ReactFiberFlags";
import { HasEffect as HookHasEffect, Passive as HookPassive, Layout as HookLayout } from "./ReactHookEffectTags";
import { NoLanes, NoLane, mergeLanes, isSubsetOfLanes } from './ReactFiberLane';
+import { readContext } from './ReactFiberNewContext';

const { ReactCurrentDispatcher } = ReactSharedInternals;
let currentlyRenderingFiber = null;
let workInProgressHook = null;
let currentHook = null;
let renderLanes = NoLanes;

const HooksDispatcherOnMountInDEV = {
  useReducer: mountReducer,
  useState: mountState,
  useEffect: mountEffect,
  useLayoutEffect: mountLayoutEffect,
  useRef: mountRef,
+ useContext: readContext
};
const HooksDispatcherOnUpdateInDEV = {
  useReducer: updateReducer,
  useState: updateState,
  useEffect: updateEffect,
  useLayoutEffect: updateLayoutEffect,
  useRef: updateRef,
+ useContext: readContext
};
function mountRef(initialValue) {
  const hook = mountWorkInProgressHook();
  const ref = {
    current: initialValue,
  };
  hook.memoizedState = ref;
  return ref;
}
function updateRef() {
  const hook = updateWorkInProgressHook();
  return hook.memoizedState;
}
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) {
  const fiberFlags = UpdateEffect;
  return mountEffectImpl(fiberFlags, HookLayout, create, deps);
}
```

```
function updateEffect(create, deps) {
  return updateEffectImpl(PassiveEffect, HookPassive, create, deps);
}
function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
  const hook = updateWorkInProgressHook();
  const nextDeps = deps
  let destroy;
  if (currentHook !== null) {
    const prevEffect = currentHook.memoizedState;
    destroy = prevEffect.destroy;
    if (nextDeps !== null) {
      const prevDeps = prevEffect.deps;
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        hook.memoizedState = 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 (is(nextDeps[i], prevDeps[i])) {
      continue;
    }
    return false;
  }

  return true;
}
function mountEffect(create, deps) {
  return mountEffectImpl(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,
  };
}
function basicStateReducer(state, action) {
  return typeof action
}
function mountReducer(reducer, initialArg) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialArg;
  const queue = {
    pending: null,
    dispatch: null,
    lastRenderedReducer: reducer,
    lastRenderedState: initialArg
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchReducerAction.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function updateReducer(reducer) {
  const hook = updateWorkInProgressHook();
  const queue = hook.queue;
  queue.lastRenderedReducer = reducer;
  const current = currentHook;
  let baseQueue = current.baseQueue;
  const pendingQueue = queue.pending;
  if (pendingQueue !== null) {
    if (baseQueue !== null) {
      const baseFirst = baseQueue.next;
      const pendingFirst = pendingQueue.next;
      baseQueue.next = pendingFirst;
      pendingQueue.next = baseFirst;
    }
```

```
    current.baseQueue = baseQueue = pendingQueue;
    queue.pending = null;
  }
  if (baseQueue !== null) {
    printQueue(baseQueue)
    const first = baseQueue.next;
    let newState = current.baseState;
    let newBaseState = null;
    let newBaseQueueFirst = null;
    let newBaseQueueLast = null;
    let update = first;
    do {
      const updateLane = update.lane;
      const shouldSkipUpdate = !isSubsetOfLanes(renderLanes, updateLane);
      if (shouldSkipUpdate) {
        const clone = {
          lane: updateLane,
          action: update.action,
          hasEagerState: update.hasEagerState,
          eagerState: update.eagerState,
          next: null,
        };
        if (newBaseQueueLast
          newBaseQueueFirst = newBaseQueueLast = clone;
          newBaseState = newState;
        } else {
          newBaseQueueLast = newBaseQueueLast.next = clone;
        }
        currentlyRenderingFiber.lanes = mergeLanes(currentlyRenderingFiber.lanes, updateLane);
      } else {
        if (newBaseQueueLast !== null) {
          const clone = {
            lane: NoLane,
            action: update.action,
            hasEagerState: update.hasEagerState,
            eagerState: update.eagerState,
            next: null,
          };
          newBaseQueueLast = newBaseQueueLast.next = clone;
        }
        if (update.hasEagerState) {
          newState = update.eagerState;
        } else {
          const action = update.action;
          newState = reducer(newState, action);
        }
      }
      update = update.next;
    } while (update !== null && update !== first);
    if (newBaseQueueLast
      newBaseState = newState;
    } else {
      newBaseQueueLast.next = newBaseQueueFirst;
    }
    hook.memoizedState = newState;
    hook.baseState = newBaseState;
    hook.baseQueue = newBaseQueueLast;
    queue.lastRenderedState = newState;
  }
  if (baseQueue
    queue.lanes = NoLanes;
  }
  const dispatch = queue.dispatch;
  return [hook.memoizedState, dispatch];
}
function mountState(initialState) {
  const hook = mountWorkInProgressHook();
  hook.memoizedState = initialState;
  const queue = {
    pending: null,
    dispatch: null,
    lastRenderedReducer: basicStateReducer,
    lastRenderedState: initialState,
  };
  hook.queue = queue;
  const dispatch = (queue.dispatch = dispatchSetState.bind(null, currentlyRenderingFiber, queue));
  return [hook.memoizedState, dispatch];
}
function dispatchSetState(fiber, queue, action) {
  const lane = requestUpdateLane(fiber);
  const update = {
    lane,
    action,
    hasEagerState: false,
    eagerState: null,
    next: null,
  };
  const alternate = fiber.alternate;
  if (fiber.lanes
    const lastRenderedReducer = queue.lastRenderedReducer;
    const currentState = queue.lastRenderedState;
    const eagerState = lastRenderedReducer(currentState, action);
    update.hasEagerState = true;
    update.eagerState = eagerState;
    if (is(eagerState, currentState)) {
      return;
    }
  }
  const root = enqueueConcurrentHookUpdate(fiber, queue, update, lane);
  const eventTime = requestEventTime();
  scheduleUpdateOnFiber(root, fiber, lane, eventTime);
}
function updateState(initialState) {
  return updateReducer(basicStateReducer, initialState);
```

```
}
function mountWorkInProgressHook() {
  const hook = {
    memoizedState: null,
    queue: null,
    next: null,
    baseState: null,
    baseQueue: null,
  };
  if (workInProgressHook
    currentlyRenderingFiber.memoizedState = workInProgressHook = hook;
  } else {
    workInProgressHook = workInProgressHook.next = hook;
  }
  return workInProgressHook;
}
function dispatchReducerAction(fiber, queue, action) {
  const update = {
    action,
    next: null,
  };
  const root = enqueueConcurrentHookUpdate(fiber, queue, update);
  scheduleUpdateOnFiber(root, fiber);
}

function updateWorkInProgressHook() {
  let nextCurrentHook;
  if (currentHook
    const current = currentlyRenderingFiber.alternate;
    if (current !== null) {
      nextCurrentHook = current.memoizedState;
    } else {
      nextCurrentHook = null;
    }
  } else {
    nextCurrentHook = currentHook.next;
  }

  let nextWorkInProgressHook;
  if (workInProgressHook
    nextWorkInProgressHook = currentlyRenderingFiber.memoizedState;
  } else {
    nextWorkInProgressHook = workInProgressHook.next;
  }

  if (nextWorkInProgressHook !== null) {
    workInProgressHook = nextWorkInProgressHook;
    nextWorkInProgressHook = workInProgressHook.next;
    currentHook = nextCurrentHook;
  } else {
    currentHook = nextCurrentHook;
    const newHook = {
      memoizedState: currentHook.memoizedState,
      queue: currentHook.queue,
      next: null,
      baseState: currentHook.baseState,
      baseQueue: currentHook.baseQueue,
    };
    if (workInProgressHook
      currentlyRenderingFiber.memoizedState = workInProgressHook = newHook;
    } else {
      workInProgressHook = workInProgressHook.next = newHook;
    }
  }
  return workInProgressHook;
}

export function renderWithHooks(current, workInProgress, Component, props, nextRenderLanes) {
  renderLanes = nextRenderLanes;
  currentlyRenderingFiber = workInProgress;
  workInProgress.updateQueue = null;
  workInProgress.memoizedState = null;
  if (current !== null && current.memoizedState !== null) {
    ReactCurrentDispatcher.current = HooksDispatcherOnUpdateInDEV;
  } else {
    ReactCurrentDispatcher.current = HooksDispatcherOnMountInDEV;
  }
  const children = Component(props);
  currentlyRenderingFiber = null;
  workInProgressHook = null;
  currentHook = null;
  renderLanes = NoLanes;
  return children;
}

function printQueue(queue) {
  const first = queue.next;
  let desc = '';
  let update = first;
  do {
    desc += ("=>" + (update.action.id));
    update = update.next;
  } while (update !== null && update !== first);
  desc += "=>null";
  console.log(desc);
}
```

42.11 ReactFiberNewContext.js

src\react-reconciler\src\ReactFiberNewContext.js

```
export function pushProvider(context, nextValue) {
  context._currentValue = nextValue;
}

export function readContext(context) {
  return context._currentValue;
}
```

src\react-reconciler\src\ReactWorkTags.js

```
export const FunctionComponent = 0;
export const IndeterminateComponent = 2;
export const HostRoot = 3;
export const HostComponent = 5;
export const HostText = 6;
+export const ContextProvider = 10;
```

42.13 ReactSymbols.js

src\shared\ReactSymbols.js

```
export const REACT_ELEMENT_TYPE = Symbol.for("react.element");
+export const REACT_PROVIDER_TYPE = Symbol.for('react.provider');
+export const REACT_CONTEXT_TYPE = Symbol.for('react.context');
```