link: null
title: 珠峰架构师成长计划
description: src\react-dom\index.js
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=69 sentences=104, words=1133

## 7.初次渲染

```
import React from 'react';
import ReactDOM from './react-dom';
ReactDOM.render(<h1>helloh1>, document.getElementById('root'));
```

src\react-dom\index.js

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

src\react-reconciler\ReactFiberRoot.js

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

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

src\react-reconciler\ReactFiber.js

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

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

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

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

src\react-reconciler\ReactUpdateQueue.js

```

export function initializeUpdateQueue(fiber) {
    const queue = {

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

export function createUpdate() {
    return {};
}

export function enqueueUpdate(fiber, update) {

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

    const pending = sharedQueue.pending;

    if (!pending) {

        update.next = update;
    } else {

        update.next = pending.next;

        pending.next = update;
    }

    sharedQueue.pending = update;
}
```

src\react-reconciler\ReactFiberReconciler.js

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

export function updateContainer(element, container) {

    const current = container.current;

    const update = createUpdate();

    update.payload = { element };

    enqueueUpdate(current, update);

    scheduleUpdateOnFiber(current);
}
```

src\react-reconciler\ReactFiberWorkLoop.js

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

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

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

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

export function scheduleUpdateOnFiber(fiber) {

    const root = markUpdateLaneFromFiberToRoot(fiber);

    performSyncWorkOnRoot(root);
}

function performSyncWorkOnRoot(root) {
    console.log(root);
}
```

src\react-reconciler\ReactWorkTags.js

```

export const HostRoot = 3;
```

## 8.同步渲染级

src\react-reconciler\ReactFiberReconciler.js

```
import { createUpdate, enqueueUpdate } from './ReactUpdateQueue';
+import { scheduleUpdateOnFiber, requestUpdateLane, requestEventTime } from './ReactFiberWorkLoop';
/**
 * 把element元素渲染到容器中
 * @param {*} element 要渲染的虚拟DOM
 * @param {*} container 容器
 */
export function updateContainer(element, container) {
    //获取HostRootFiber
    const current = container.current;
+   //获取事件开始时间，一般是performance.now()
+   const eventTime = requestEventTime();
+   //获取更新优先级
+   const lane = requestUpdateLane(current);
    //创建一个更新对象
+   const update = createUpdate(eventTime, lane);
    //更新对象的payload为{ element }
    update.payload = { element };
    //把更新对象添加到更新队列中
    enqueueUpdate(current, update);
    //开始从HostRootFiber调度更新
+   scheduleUpdateOnFiber(current, lane, eventTime);
}
```

src\react-reconciler\ReactFiber.js

```
import { HostRoot } from './ReactWorkTags';
/**
 * 创建根fiber
 * @returns 根fiber
 */
export function createHostRootFiber() {
    return createFiber(HostRoot);
}
/**
 * 创建fiber
 * @param {*} tag fiber类型
 * @param {*} pendingProps 新属性对象
 * @param {*} key 唯一标识
 * @returns 创建的fiber
 */
const createFiber = function (tag, pendingProps, key) {
    return new FiberNode(tag, pendingProps, key);
};
/**
 * fiber构建函数
 * @param {*} tag fiber类型
 * @param {*} pendingProps  新属性对象
 * @param {*} key 唯一标识
 */
function FiberNode(tag, pendingProps, key) {
    this.tag = tag;
    this.pendingProps = pendingProps;
    this.key = key;
}
+/**
+ * 基于老的current创建新的workInProgress
+ * @param {*} current 老的fiber
+ * @returns
+*/
+export function createWorkInProgress(current, pendingProps) {
+    let workInProgress = createFiber(current.tag, pendingProps, current.key);
+    workInProgress.childLanes = current.childLanes;
+    workInProgress.lanes = current.lanes;
+    current.alternate = workInProgress;
+    return workInProgress;
+}
```

src\react-reconciler\ReactUpdateQueue.js

```
/**
 * 初始化fiber节点上的更新队列
 * @param {*} fiber
 */
export function initializeUpdateQueue(fiber) {
    const queue = {
        //这是一个环状链表，存放着等待生效的更新
        shared: {
            pending: null
        }
    };
    fiber.updateQueue = queue;
}
/**
 * 创建更新对象
 * @returns 更新对象
 */
export function createUpdate(eventTime, lane) {
    return {
+        // 任务时间，通过performance.now()获取的毫秒数
+        eventTime,
+        // 更新优先级
+        lane,
+        //更新所携带的状态
+        //根组件中为React.element，即ReactDOM.render的第一个参数
+        payload: null,
+        // 指向下一个update
+        next: null
    };
}
/**
 * 把更新对象添加到fiber的更新队列中
 * @param {*} fiber fiber节点
 * @param {*} update 新的更新对象
 */
export function enqueueUpdate(fiber, update) {
    //取出fiber上的更新队列
    const updateQueue = fiber.updateQueue;
    const sharedQueue = updateQueue.shared;
    //取出等待生效的更新环状链表
    const pending = sharedQueue.pending;
    //如果环状链表为空
    if (!pending) {
        //构建环状链表
        update.next = update;
    } else {
        //让新的更新的next指向第一个更新
        update.next = pending.next;
        //让原来的pending指向新的更新
        pending.next = update;
    }
    //sharedQueue的pending指向新的更新
    sharedQueue.pending = update;
}
```

src\react-reconciler\ReactFiberWorkLoop.js

```
import { HostRoot } from './ReactWorkTags';
+import { NoTimestamp, SyncLane, mergeLanes, markRootUpdated, NoLanes, getNextLanes, +includesSomeLane } from './ReactFiberLane';
+import { now } from '../scheduler';
+import { createWorkInProgress } from './ReactFiber';
+let currentEventTime = NoTimestamp;
+let workInProgress = null;
+export let subtreeRenderLanes = NoLanes;
/**
 * 从触发状态更新的fiber通过一直往上找return得到rootFiber
 * 找的过程都会将lane收集到每个parent.childLanes上
 * @param {*} sourceFiber 更新来源fiber
 * @returns
 */
+function markUpdateLaneFromFiberToRoot(sourceFiber, lane) {
+   //更新现有fiber上的lanes
+   sourceFiber.lanes = mergeLanes(sourceFiber.lanes, lane);
    let node = sourceFiber;
    // 从产生更新的fiber节点开始，向上收集childLanes
    let parent = node.return;
    //到rootFiber，其parent为null，则会跳出while
    while (parent) {
+       parent.childLanes = mergeLanes(parent.childLanes, lane);
        node = parent;
        parent = parent.return;
    }
    //如果找到的是HostRoot就返回FiberRootNode,其实就是容器div#root
    if (node.tag
        return node.stateNode;
    }
}

+export function scheduleUpdateOnFiber(fiber, lane, eventTime) {
+    //向上获取HostRoot节点并向上收集fiber.childLanes
+    const root = markUpdateLaneFromFiberToRoot(fiber, lane);
+    //在root上标记更新，将update的lane放到root.pendingLane
+    markRootUpdated(root, lane, eventTime);
+    if (lane === SyncLane) {
+        //执行HostRoot上的更新
+        performSyncWorkOnRoot(root);
+    }
+}

/**
 * 开始执行FiberRootNode上的工作
 * @param {*} root  FiberRootNode
 */
function performSyncWorkOnRoot(root) {
+   let lanes = getNextLanes(root, NoLanes);
+   renderRootSync(root, lanes);
}
+/**
+ * 刷新栈帧: 重置FiberRoot上的全局属性和fiber树构造循环过程中的全局变量
+ * @param {*} root
+ * @param {*} lanes
+ */
+function prepareFreshStack(root, lanes) {
+    root.finishedWork = null;
+    root.finishedLanes = NoLanes;
+    workInProgress = createWorkInProgress(root.current, null);
+    subtreeRenderLanes = lanes;
+}
+
+function renderRootSync(root, lanes) {
+    prepareFreshStack(root, lanes);
+    workLoopSync();
+}
+function workLoopSync() {
+    while (workInProgress) {
+        performUnitOfWork(workInProgress);
+    }
+}
+function performUnitOfWork(unitOfWork) {
+    if (includesSomeLane(subtreeRenderLanes, unitOfWork.lanes)) {
+        console.log('处理', unitOfWork);
+        workInProgress = null;
+    } else {
+        workInProgress = null;
+    }
+}
+export function requestEventTime() {
+    currentEventTime = now();
+    return currentEventTime;
+}
+
+export function requestUpdateLane(fiber) {
+    return SyncLane;
+}
```

src\react-reconciler\ReactFiberLane.js

```
export const NoLanePriority = 0;
export const DefaultLanePriority = 8;

export const NoLanes = 0b0000000000000000000000000000000;

export const SyncLane = 0b0000000000000000000000000000001;

export const NoTimestamp = -1;

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;
    let nextLanes = NoLanes;
    nextLanes = getHighestPriorityLanes(pendingLanes);
    nextLanes = pendingLanes & getEqualOrHigherPriorityLanes(nextLanes);
    return nextLanes;
}

function getHighestPriorityLanes(lanes) {
    if ((SyncLane & lanes) !== NoLanes) {
        return SyncLane;
    }
}

function getEqualOrHigherPriorityLanes(lanes) {
    return (getLowestPriorityLane(lanes) << 1) - 1;
}

function getLowestPriorityLane(lanes) {
    const index = 31 - Math.clz32(lanes);
    return index < 0 ? NoLanes : 1 << index;
}

export function includesSomeLane(a, b) {
    return (a & b) !== NoLanes;
}
```

## 9.异步渲染

react-reconciler\ReactFiberWorkLoop.js

```
import { HostRoot } from './ReactWorkTags';
+import {
+   NoTimestamp, SyncLane, mergeLanes, markRootUpdated, NoLanes, getNextLanes,
+   includesSomeLane, schedulerPriorityToLanePriority, findUpdateLane, returnNextLanesPriority,
+   lanePriorityToSchedulerPriority, markStarvedLanesAsExpired, markRootFinished
+} from './ReactFiberLane';
import { now } from '../scheduler';
import { createWorkInProgress } from './ReactFiber';
+import {
+    scheduleCallback, getCurrentPriorityLevel, shouldYield,
+    ImmediatePriority as ImmediateSchedulerPriority,
+    runWithPriority
+} from './SchedulerWithReactIntegration';
let currentEventTime = NoTimestamp;
let workInProgress = null;
//表示需要更新的fiber节点的lane的集合，在后面更新fiber节点的时候会根据这个值判断是否需要更新
export let subtreeRenderLanes = NoLanes;
+let currentEventWipLanes = NoLanes;
+let workInProgressRootIncludedLanes = NoLanes;
+//是在任务执行阶段赋予的需要更新的fiber节点上的lane的值
+//当新的更新任务产生时，workInProgressRootRenderLanes不为空，则表示有任务正在执行
+//那么则直接返回这个正在执行的任务的lane，那么当前新的任务则会和现有的任务进行一次批量更新
+//表示当前是否有任务正在执行，有值则表示有任务正在执行，反之则没有任务在执行
+let workInProgressRootRenderLanes = NoLanes;

/**
 * 从触发状态更新的fiber通过一直往上找return得到rootFiber
 * 找的过程都会将lane收集到每个parent.childLanes上
 * @param {*} sourceFiber 更新来源fiber
 * @returns
 */
function markUpdateLaneFromFiberToRoot(sourceFiber, lane) {
    //更新现有fiber上的lanes
    sourceFiber.lanes = mergeLanes(sourceFiber.lanes, lane);
    let node = sourceFiber;
    // 从产生更新的fiber节点开始，向上收集childLanes
    let parent = node.return;
    //到rootFiber，其parent为null，则会跳出while
    while (parent) {
        parent.childLanes = mergeLanes(parent.childLanes, lane);
        node = parent;
        parent = parent.return;
    }
    //如果找到的是HostRoot就返回FiberRootNode,其实就是容器div#root
    if (node.tag
        return node.stateNode;
    }
}

export function scheduleUpdateOnFiber(fiber, lane, eventTime) {
    //向上获取HostRoot节点并向上收集fiber.childLanes
    const root = markUpdateLaneFromFiberToRoot(fiber, lane);
    //在root上标记更新，将update的lane放到root.pendingLane
    markRootUpdated(root, lane, eventTime);
    if (lane
        //执行HostRoot上的更新
        performSyncWorkOnRoot(root);
+   } else {
+       ensureRootIsScheduled(root, eventTime);
+   }
}

+function ensureRootIsScheduled(root, currentTime) {
+    //为当前任务根据优先级添加过期时间
+    //并检查未执行的任务中是否有任务过期，有任务过期则expiredLanes中添加该任务的lane
+    //在后续任务执行中以同步模式执行，避免饥饿问题
+    markStarvedLanesAsExpired(root, currentTime);
+    //获取优先级最高的任务的优先级
+    const nextLanes = getNextLanes(root, workInProgressRootRenderLanes);
+    //如果nextLanes为空则表示没有任务需要执行，则直接中断更新
+    if (nextLanes === NoLanes) {
+        return;
+    }
+    const newCallbackPriority = returnNextLanesPriority();
+    const schedulerPriorityLevel = lanePriorityToSchedulerPriority(newCallbackPriority);
+    let newCallbackNode = scheduleCallback(schedulerPriorityLevel, performConcurrentWorkOnRoot.bind+(null, root));
+    root.callbackPriority = newCallbackPriority;
+    root.callbackNode = newCallbackNode;
+}
+function performConcurrentWorkOnRoot(root) {
+    currentEventTime = NoTimestamp;
+    currentEventWipLanes = NoLanes;
+    const originalCallbackNode = root.callbackNode;
+    //获取本次渲染的优先级
+    let lanes = getNextLanes(root, workInProgressRootRenderLanes);
+    //构造fiber树
+    let exitStatus = renderRootConcurrent(root, lanes);
+    const finishedWork = root.current.alternate;
+    root.finishedWork = finishedWork;
+    root.finishedLanes = lanes;
+    //渲染fiber树
+    finishConcurrentRender(root, exitStatus, lanes);
+    //退出前再次检测, 是否还有其他更新, 是否需要发起新调度
+    if (root.callbackNode === originalCallbackNode) {
+        //渲染被阻断, 返回一个新的performConcurrentWorkOnRoot函数, 等待下一次调用
+        return performConcurrentWorkOnRoot.bind(null, root);
+    }
+    return null;
+}
+function finishConcurrentRender(root, exitStatus, lanes) {
+    commitRoot(root);
+}
+function commitRoot(root) {
+    const renderPriorityLevel = getCurrentPriorityLevel();
+    runWithPriority(ImmediateSchedulerPriority, commitRootImpl.bind(null, root, renderPriorityLevel));
+    return null;
+}
+function commitRootImpl(root, renderPriorityLevel) {
+    //设置局部变量
+    const finishedWork = root.finishedWork;
+    const lanes = root.finishedLanes;
+    //清空FiberRoot对象上的属性
+    root.finishedWork = null;
+    root.finishedLanes = NoLanes;
+    root.callbackNode = null;
+    //将finishedWork.lanes和finishedWork.childLanes进行合并操作,获取到剩下还需要做更新的lanes
+    let remainingLanes = mergeLanes(finishedWork.lanes, finishedWork.childLanes);
+    //然后调用markRootFinished清空掉已经执行完成的lanes的数据，将剩下的lanes重新挂载到pendingLanes上，准备下一+次的执行
+    markRootFinished(root, remainingLanes);
+}
+function renderRootConcurrent(root, lanes) {
+    prepareFreshStack(root, lanes);
+    workLoopConcurrent();
+}
+function workLoopConcurrent() {
+    while (workInProgress !== null && !shouldYield()) {
+        performUnitOfWork(workInProgress);
+    }
+}
+
+
+/**
+ * 开始执行FiberRootNode上的工作
+ * @param {*} root  FiberRootNode
+ */
+function performSyncWorkOnRoot(root) {
+    let lanes = getNextLanes(root, NoLanes);
+    let exitStatus = renderRootSync(root, lanes);
+    const finishedWork = root.current.alternate;
+    root.finishedWork = finishedWork;
+    root.finishedLanes = lanes;
+    commitRoot(root);
+}
+/**
+ * 刷新栈帧: 重置FiberRoot上的全局属性和fiber树构造循环过程中的全局变量
+ * @param {*} root
+ * @param {*} lanes
+ */
+function prepareFreshStack(root, lanes) {
+    root.finishedWork = null;
+    root.finishedLanes = NoLanes;
+    workInProgress = createWorkInProgress(root.current, null);
+    subtreeRenderLanes = workInProgressRootIncludedLanes = lanes;
+}
+
+function renderRootSync(root, lanes) {
```

```
+    prepareFreshStack(root, lanes);
+    workLoopSync();
+}
+function workLoopSync() {
+    while (workInProgress) {
+        performUnitOfWork(workInProgress);
+    }
+}
+function performUnitOfWork(unitOfWork) {
+    if (includesSomeLane(subtreeRenderLanes, unitOfWork.lanes)) {
+        console.log('处理', unitOfWork);
+        workInProgress = null;
+    } else {
+        workInProgress = null;
+    }
+}
+export function requestEventTime() {
+    currentEventTime = now();
+    return currentEventTime;
+}
+
+export function requestUpdateLane(fiber) {
+    if (currentEventWipLanes === NoLanes) {
+        currentEventWipLanes = workInProgressRootIncludedLanes;
+    }
+    const schedulerPriority = getCurrentPriorityLevel();//97
+    let lane;
+    const schedulerLanePriority = schedulerPriorityToLanePriority(schedulerPriority);//8
+    lane = findUpdateLane(schedulerLanePriority, currentEventWipLanes);
+    return lane;
+}
```

src\scheduler\src\Scheduler.js

```
import { requestHostCallback, shouldYieldToHost, getCurrentTime, requestHostTimeout } from './SchedulerHostConfig';
import { push, pop, peek } from './SchedulerMinHeap';
import { ImmediatePriority, UserBlockingPriority, NormalPriority, LowPriority, IdlePriority } from './SchedulerPriorities';
// 不同优先级对应的不同的任务过期时间间隔
let maxSigned31BitInt = 1073741823;
let IMMEDIATE_PRIORITY_TIMEOUT = -1;//立即执行的优先级，级别最高
let USER_BLOCKING_PRIORITY_TIMEOUT = 250;//用户阻塞级别的优先级
let NORMAL_PRIORITY_TIMEOUT = 5000;//正常的优先级
let LOW_PRIORITY_TIMEOUT = 10000;//较低的优先级
let IDLE_PRIORITY_TIMEOUT = maxSigned31BitInt;//优先级最低，表示任务可以闲置
//下一个任务ID编号
let taskIdCounter = 1;
//任务队列
let taskQueue = [];
//延迟队列
let timerQueue = [];
let currentTask;
let currentPriorityLevel = NormalPriority;

/**
 * 调度一个任务
 * @param {*} callback 要执行的任务
 */
export function scheduleCallback(priorityLevel, callback, options) {
    // 获取当前时间，它是计算任务开始时间、过期时间和判断任务是否过期的依据
    let currentTime = getCurrentTime();
    // 确定任务开始时间
    let startTime;
    if (typeof options
        var delay = options.delay;
        if (typeof delay
            startTime = currentTime + delay;
        } else {
            startTime = currentTime;
        }
    } else {
        startTime = currentTime;
    }
    // 计算过期时间
    let timeout;
    switch (priorityLevel) {
        case ImmediatePriority://1
            timeout = IMMEDIATE_PRIORITY_TIMEOUT;//-1
            break;
        case UserBlockingPriority://2
            timeout = USER_BLOCKING_PRIORITY_TIMEOUT;//250
            break;
        case IdlePriority://5
            timeout = IDLE_PRIORITY_TIMEOUT;//1073741823
            break;
        case LowPriority://4
            timeout = LOW_PRIORITY_TIMEOUT;//10000
            break;
        case NormalPriority://3
        default:
            timeout = NORMAL_PRIORITY_TIMEOUT;//5000
            break;
    }
    //计算超时时间
    let expirationTime = startTime + timeout;
    //创建新任务
    let newTask = {
        id: taskIdCounter++,//任务ID
        callback,//真正的任务函数
        priorityLevel,//任务优先级，参与计算任务过期时间
        startTime,
        expirationTime,//表示任务何时过期，影响它在taskQueue中的排序
        //为小顶堆的队列提供排序依据
        sort
    };
    if (startTime > currentTime) {
        newTask.sortIndex = startTime;
        push(timerQueue, newTask);
        if (peek(taskQueue)
            requestHostTimeout(handleTimeout, startTime - currentTime);
        }
    } else {
        newTask.sortIndex = expirationTime;
        //把此工作添加到任务队列中
        push(taskQueue, newTask);
        //taskQueue.push(callback);
        //开始调度flushWork
        requestHostCallback(flushWork);
    }
    return newTask;
}
/**
 * 处理超时任务
 * @param {*} currentTime
 */
function handleTimeout(currentTime) {
    advanceTimers(currentTime);
    if (peek(taskQueue) !== null) {
        requestHostCallback(flushWork);
    } else {
        const firstTimer = peek(timerQueue);
        if (firstTimer !== null) {
            requestHostTimeout(handleTimeout, firstTimer.startTime - currentTime);
        }
    }
}
function advanceTimers(currentTime) {
    let timer = peek(timerQueue);
    while (timer !== null) {
        if (timer.callback
            pop(timerQueue);
        } else if (timer.startTime  currentTime && shouldYieldToHost()) {
            break;
        }
        //执行当前的工作
        const callback = currentTask.callback;
        if (typeof callback
            currentTask.callback = null;
            const didUserCallbackTimeout = currentTask.expirationTime +/**
+ *
+ * @param {*} priorityLevel
+ * @param {*} eventHandler
+ * @returns
+ */
+function runWithPriority(priorityLevel, eventHandler) {
+    switch (priorityLevel) {
+        case ImmediatePriority:
+        case UserBlockingPriority:
+        case NormalPriority:
+        case LowPriority:
+        case IdlePriority:
+            break;
+        default:
+            priorityLevel = NormalPriority;
+    }
+    var previousPriorityLevel = currentPriorityLevel;
+    currentPriorityLevel = priorityLevel;
+    try {
+        return eventHandler();
+    } finally {
+        currentPriorityLevel = previousPriorityLevel;
+    }
+}
export function getCurrentPriorityLevel() {
    return currentPriorityLevel;
}
export {
    shouldYieldToHost as shouldYield,
    ImmediatePriority,
    UserBlockingPriority,
    NormalPriority,
    IdlePriority,
    LowPriority,
    getCurrentTime as now,
+   runWithPriority
}
```

src\react-reconciler\ReactFiberLane.js

```
import { ImmediatePriority as ImmediateSchedulerPriority, UserBlockingPriority as UserBlockingSchedulerPriority, NormalPriority as NormalSchedulerPriority,
LowPriority as LowSchedulerPriority, IdlePriority as IdleSchedulerPriority, NoPriority as NoSchedulerPriority } from './SchedulerWithReactIntegration';
+export const SyncLanePriority = 15;
+export const InputDiscreteLanePriority = 12;
+export const InputContinuousLanePriority = 10;
export const DefaultLanePriority = 8;
+export const TransitionPriority = 6;
+export const IdleLanePriority = 2;
+export const NoLanePriority = 0;

+//lane使用31位二进制来表示优先级车道共31条, 位数越小(1的位置越靠右)表示优先级越高
+const TotalLanes = 31;
+//没有优先级
export const NoLanes = 0b0000000000000000000000000000000;
+//同步优先级，表示同步的任务一次只能执行一个，例如：用户的交互事件产生的更新任务
export const SyncLane = 0b0000000000000000000000000000001;
+// 连续触发优先级，例如：滚动事件，拖动事件等
+export const InputContinuousHydrationLane = 0b0000000000000000000000000000010;
+export const InputContinuousLane = 0b0000000000000000000000000000100;
+// 默认优先级，例如使用setTimeout，请求数据返回等造成的更新
```

```
+export const DefaultLanes = 0b0000000000000000000111000000000;
+const IdleLanes = 0b0110000000000000000000000000000;
+export const NoTimestamp = -1;
+let return_highestLanePriority = DefaultLanePriority;

/**
 * 把 a 和 b合并成一个Lanes(优先级分组)
 * 生成一个新的优先级范围
 */
export function mergeLanes(a, b) {
    return a | b;
}

export function markRootUpdated(root, updateLane) {
    //将本次更新的lane放入root的pendingLanes
    root.pendingLanes |= updateLane;
}
+/**
+ * 获取优先级最高的任务的优先级
+ * @param {*} root
+ * @param {*} wipLanes
+ * @returns
+ */
+export function getNextLanes(root, wipLanes) {
+    // 该函数从root.pendingLanes中找出优先级最高的lane
+    const pendingLanes = root.pendingLanes;
+    let nextLanes = NoLanes;
+    let nextLanePriority = NoLanePriority;
+    const expiredLanes = root.expiredLanes;
+    if (expiredLanes !== NoLanes) {
+        nextLanes = expiredLanes;
+        nextLanePriority = return_highestLanePriority = SyncLanePriority;
+    } else {
+        nextLanes = getHighestPriorityLanes(pendingLanes);
+        nextLanePriority = return_highestLanePriority;
+    }
+    if (nextLanes === NoLanes) {
+        return NoLanes;
+    }
+    nextLanes = pendingLanes & getEqualOrHigherPriorityLanes(nextLanes);
+    return nextLanes;
+}
+/**
+ * 找到对应优先级范围内优先级最高的那一批lanes
+ * @param {*} lanes
+ */
+function getHighestPriorityLanes(lanes) {
+    if ((SyncLane & lanes) !== NoLanes) {
+        return SyncLane;
+    }
+    const defaultLanes = DefaultLanes & lanes;
+    if (defaultLanes !== NoLanes) {
+        return_highestLanePriority = DefaultLanePriority;
+        return defaultLanes;
+    }
+
+    const idleLanes = IdleLanes & lanes;
+
+    if (idleLanes !== NoLanes) {
+        return_highestLanePriority = IdleLanePriority;
+        return idleLanes;
+    }
+    return_highestLanePriority = DefaultLanePriority;
+    return lanes;
+}
+function getHighestPriorityLane(lanes) {
+    return lanes & -lanes;
+}
+
+function getEqualOrHigherPriorityLanes(lanes) {
+    return (getLowestPriorityLane(lanes) << 1) - 1;
+}
+/**
+ * 找到lanes中优先级最低的那一个lane
+ * @param {*} lanes
+ * @returns
+ */
+function getLowestPriorityLane(lanes) {
+    const index = 31 - Math.clz32(lanes);
+    return index < 0 ? NoLanes : 1 << index;
+}
+
+export function includesSomeLane(a, b) {
+    return (a & b) !== NoLanes;
+}
+
+export function schedulerPriorityToLanePriority(schedulerPriorityLevel) {
+    switch (schedulerPriorityLevel) {
+        case ImmediateSchedulerPriority:
+            return SyncLanePriority;
+        case UserBlockingSchedulerPriority:
+            return InputContinuousLanePriority;
+        case NormalSchedulerPriority:
+        case LowSchedulerPriority:
+            return DefaultLanePriority;
+        case IdleSchedulerPriority:
+            return IdleLanePriority;
+        default:
+            return NoLanePriority;
+    }
+}
+
+export function findUpdateLane(lanePriority, wipLanes) {
+    switch (lanePriority) {
```

```
+        case DefaultLanePriority:
+            {
+                let lane = pickArbitraryLane(DefaultLanes & ~wipLanes);//512
+                return lane;
+            }
+        default:
+            break;
+    }
+}
+export function pickArbitraryLane(lanes) {
+    return getHighestPriorityLane(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 pickArbitraryLaneIndex(lanes) {
+    return 31 - Math.clz32(lanes);
+}
+
+function computeExpirationTime(lane, currentTime) {
+    getHighestPriorityLanes(lane);
+    const priority = return_highestLanePriority;
+    if (priority >= InputContinuousLanePriority) {
+        return currentTime + 250;
+    } else if (priority >= TransitionPriority) {
+        return currentTime + 5000;
+    } else {
+        return NoTimestamp;
+    }
+}
+export function lanePriorityToSchedulerPriority(lanePriority) {
+    switch (lanePriority) {
+        case SyncLanePriority:
+            return ImmediateSchedulerPriority;
+        case InputDiscreteLanePriority:
+        case InputContinuousLanePriority:
+            return UserBlockingSchedulerPriority;
+        case DefaultLanePriority:
+            return NormalSchedulerPriority;
+        case IdleLanePriority:
+            return IdleSchedulerPriority;
+        case NoLanePriority:
+            return NoSchedulerPriority;
+        default:
+            break;
+    }
+}
+
+export function returnNextLanesPriority() {
+    return return_highestLanePriority;
+}
+
+export function createLaneMap(initial) {
+    const laneMap = [];
+    for (let i = 0; i < TotalLanes; i++) {
+        laneMap.push(initial);
+    }
+    return laneMap;
+}
+export function markRootFinished(root, remainingLanes) {
+    //从pendingLanes中删除还未执行的lanes，那么就找到了已经执行过的lanes
+    const noLongerPendingLanes = root.pendingLanes & ~remainingLanes;
+    // 将剩下的lanes重新挂载到pendingLanes上，准备下一次的执行
+    root.pendingLanes = remainingLanes;
+    // 从expiredLanes中删除掉已经执行的lanes
+    root.expiredLanes &= remainingLanes;
+    const expirationTimes = root.expirationTimes;
+    const eventTimes = root.eventTimes;
+    let lanes = noLongerPendingLanes;
+    //取出已经执行的lane，清空它们所有的数据
+    //eventTimes中的事件触发时间，expirationTimes中的任务过期时间等
+    while (lanes > 0) {
+        const index = pickArbitraryLaneIndex(lanes);
+        const lane = 1 << index;
+        eventTimes[index] = NoTimestamp;
+        expirationTimes[index] = NoTimestamp;
+        lanes &= ~lane;
+    }
+}
```

src\react-reconciler\ReactFiberRoot.js

```
import { createHostRootFiber } from './ReactFiber';
import { initializeUpdateQueue } from './ReactUpdateQueue';
+import { NoTimestamp, createLaneMap, NoLanes } from './ReactFiberLane';
export function createFiberRoot(containerInfo) {
    const root = new FiberRootNode(containerInfo);
    const hostRootFiber = createHostRootFiber();
    root.current = hostRootFiber;
    hostRootFiber.stateNode = root;
    initializeUpdateQueue(hostRootFiber);
    return root;
}
function FiberRootNode(containerInfo) {
    this.containerInfo = containerInfo;
+   this.eventTimes = createLaneMap(NoLanes);
+   this.expirationTimes = createLaneMap(NoTimestamp);
}
```

src\react\shared\ReactTypes.js

```
export const DiscreteEvent = 0;
export const UserBlockingEvent = 1;
export const ContinuousEvent = 2;
```

src\react-dom\ReactDOMEventListener.js

```
import * as Scheduler from 'scheduler';
const { UserBlockingPriority, runWithPriority } = Scheduler;
export function createEventListenerWrapperWithPriority(targetContainer, domEventName, eventSystemFlags) {
    const eventPriority = getEventPriorityForPluginSystem(domEventName);
    let listenerWrapper;
    switch (eventPriority) {
        case DiscreteEvent:
            listenerWrapper = dispatchDiscreteEvent;
            break;
        case UserBlockingEvent:
            listenerWrapper = dispatchUserBlockingUpdate;
            break;
        case ContinuousEvent:
        default:
            listenerWrapper = dispatchEvent;
            break;
    }
    return listenerWrapper.bind(null, domEventName, eventSystemFlags, targetContainer);
}
function dispatchUserBlockingUpdate(domEventName, eventSystemFlags, container, nativeEvent) {
    runWithPriority(UserBlockingPriority, dispatchEvent.bind(null, domEventName, eventSystemFlags, container, nativeEvent));
}
```

src\react-dom\ReactDOMEventListener.js

```
import * as Scheduler from 'scheduler';
const { UserBlockingPriority, runWithPriority } = Scheduler;
export function createEventListenerWrapperWithPriority(targetContainer, domEventName, eventSystemFlags) {
    const eventPriority = getEventPriorityForPluginSystem(domEventName);
    let listenerWrapper;
    switch (eventPriority) {
        case DiscreteEvent:
            listenerWrapper = dispatchDiscreteEvent;
            break;
        case UserBlockingEvent:
            listenerWrapper = dispatchUserBlockingUpdate;
            break;
        case ContinuousEvent:
        default:
            listenerWrapper = dispatchEvent;
            break;
    }
    return listenerWrapper.bind(null, domEventName, eventSystemFlags, targetContainer);
}
function dispatchUserBlockingUpdate(domEventName, eventSystemFlags, container, nativeEvent) {
    runWithPriority(UserBlockingPriority, dispatchEvent.bind(null, domEventName, eventSystemFlags, container, nativeEvent));
}
```

src\react-reconciler\SchedulerWithReactIntegration.js

```
import * as Scheduler from '../scheduler';
const {
    getCurrentPriorityLevel: Scheduler_getCurrentPriorityLevel,
    ImmediatePriority: Scheduler_ImmediatePriority,
    UserBlockingPriority: Scheduler_UserBlockingPriority,
    NormalPriority: Scheduler_NormalPriority,
    LowPriority: Scheduler_LowPriority,
    IdlePriority: Scheduler_IdlePriority,
    scheduleCallback: Scheduler_scheduleCallback,
    shouldYield: Scheduler_shouldYield,
    runWithPriority: Scheduler_runWithPriority
} = Scheduler;

export const ImmediatePriority = 99;
export const UserBlockingPriority = 98;
export const NormalPriority = 97;
export const LowPriority = 96;
export const IdlePriority = 95;
export const NoPriority = 90;

export function getCurrentPriorityLevel() {
    switch (Scheduler_getCurrentPriorityLevel()) {
        case Scheduler_ImmediatePriority:
            return ImmediatePriority;
        case Scheduler_UserBlockingPriority:
            return UserBlockingPriority;
        case Scheduler_NormalPriority:
            return NormalPriority;
        case Scheduler_LowPriority:
            return LowPriority;
        case Scheduler_IdlePriority:
            return IdlePriority;
        default:
            break;
    }
}

export function scheduleCallback(reactPriorityLevel, callback, options) {
    const priorityLevel = reactPriorityToSchedulerPriority(reactPriorityLevel);
    return Scheduler_scheduleCallback(priorityLevel, callback, options);
}

function reactPriorityToSchedulerPriority(reactPriorityLevel) {
    switch (reactPriorityLevel) {
        case ImmediatePriority:
            return Scheduler_ImmediatePriority;
        case UserBlockingPriority:
            return Scheduler_UserBlockingPriority;
        case NormalPriority:
            return Scheduler_NormalPriority;
        case LowPriority:
            return Scheduler_LowPriority;
        case IdlePriority:
            return Scheduler_IdlePriority;
        default:
            break;
    }
}
export function runWithPriority(reactPriorityLevel, fn) {
    const priorityLevel = reactPriorityToSchedulerPriority(reactPriorityLevel);
    return Scheduler_runWithPriority(priorityLevel, fn);
}
export const shouldYield = Scheduler_shouldYield;
```