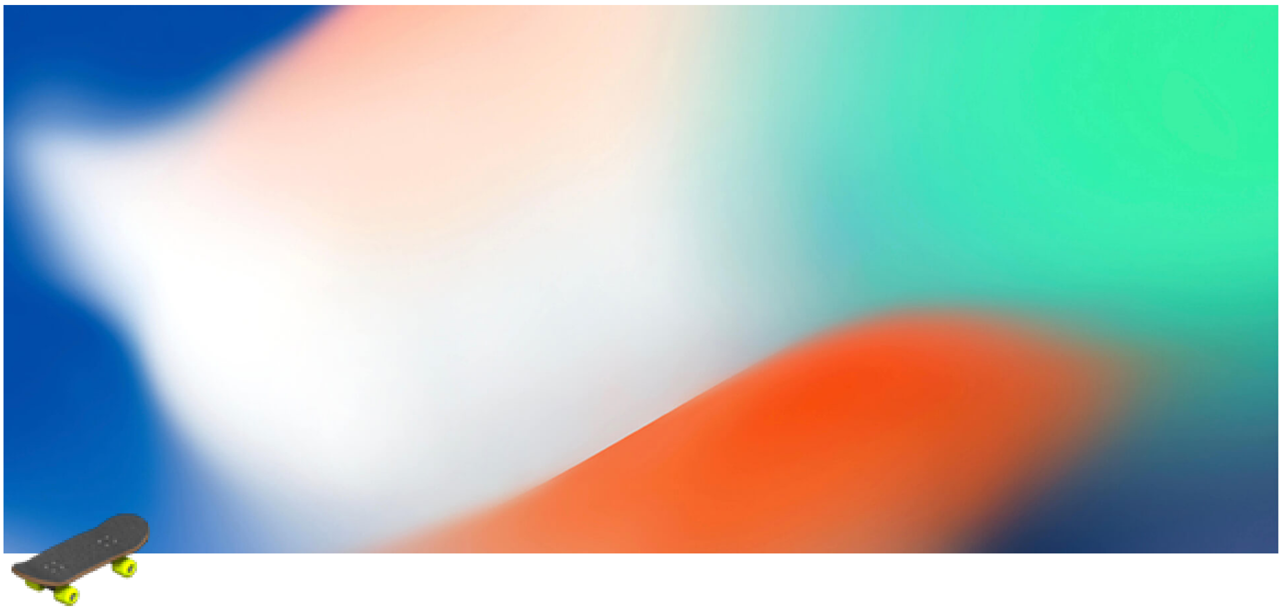

5-7 总结：源码实践 React 底层任务调度机制

最小堆、调度策略(任务优先级、任务到达时间、延迟时间)、具体处理策略。

```TypeScript
// ! 实现一个单线程任务调度器
import { getCurrentTime, isFn } from "shared/utils";
import { peek, pop, push } from "./SchedulerMinHeap";
import {
  PriorityLevel,
  NormalPriority,
  IdlePriority,
  ImmediatePriority,
  LowPriority,
  UserBlockingPriority,
  NoPriority,
} from "./SchedulerPriorities";
import {
  lowPriorityTimeout,
  maxSigned31BitInt,
  normalPriorityTimeout,
  userBlockingPriorityTimeout,
} from "./SchedulerFeatureFlags";
```

```
type Callback = (arg: boolean) => Callback | null | undefined;

export type Task = {
  id: number;
  callback: Callback | null;
  priorityLevel: PriorityLevel;
  startTime: number;
  expirationTime: number;
  sortIndex: number;
};

// 任务池，最小堆
const taskQueue: Array<Task> = []; // 没有延迟的任务
const timerQueue: Array<Task> = []; // 有延迟的任务

//标记task的唯一性
let taskIdCounter = 1;

let currentTask: Task | null = null;
let currentPriorityLevel: PriorityLevel = NoPriority;

// 记录时间切片的起始值，时间戳
let startTime = -1;

// 时间切片，这是个时间段
let frameInterval = 5;

// 锁
// 是否有 work 在执行
let isPerformingWork = false;

// 主线程是否在调度
let isHostCallbackScheduled = false;

let isMessageLoopRunning = false;

// 是否有任务在倒计时
var isHostTimeoutScheduled = false;

let taskTimeoutID = -1;
```

```
function shouldYieldToHost() {
  const timeElapsed = getCurrentTime() - startTime;

  if (timeElapsed < frameInterval) {
    return false;
  }

  return true;
}

// 任务调度器的入口函数
function scheduleCallback(
  priorityLevel: PriorityLevel,
  callback: Callback,
  options?: { delay: number }
) {
  const currentTime = getCurrentTime();
  let startTime;

  if (typeof options === "object" && options !== null) {
    let delay = options.delay;
    if (typeof delay === "number" && delay > 0) {
      // 有效的延迟时间
      startTime = currentTime + delay;
    } else {
      // 无效的延迟时间
      startTime = currentTime;
    }
  } else {
    // 无延迟
    startTime = currentTime;
  }

  // expirationTime 是过期时间，理论上的任务执行时间

  let timeout: number;
  switch (priorityLevel) {
    case ImmediatePriority:
      // 立即超时，SVVVVIP
      timeout = -1;
      break;
    case UserBlockingPriority:
      // 最终超时，VIP
      timeout = userBlockingPriorityTimeout;
      break;
    case IdlePriority:
      // 永不超时
      timeout = maxSigned31BitInt;
      break;
    case LowPriority:
      // 最终超时
      timeout = lowPriorityTimeout;
      break;
    case NormalPriority:
    default:
      timeout = normalPriorityTimeout;
      break;
  }

  const expirationTime = startTime + timeout;
  const newTask: Task = {
    id: taskIdCounter++,
    callback,
    priorityLevel,
    startTime,
    expirationTime,
    sortIndex: -1,
  };

  if (startTime > currentTime) {
    // newTask任务有延迟
    newTask.sortIndex = startTime;
    // 任务在timerQueue到达开始时间之后，就会被推入 taskQueue
    push(timerQueue, newTask);
    // 每次只倒计时一个任务
    if (peek(taskQueue) === null && newTask === peek(timerQueue)) {
      if (isHostTimeoutScheduled) {
        // newTask 才是堆顶任务，才应该最先到达执行时间，newTask应该被倒计时，
        cancelHostTimeout();
      } else {
        isHostTimeoutScheduled = true;
      }

      requestHostTimeout(handleTimeout, startTime - currentTime);
    }
  } else {
    newTask.sortIndex = expirationTime;
    push(taskQueue, newTask);

    if (!isHostCallbackScheduled && !isPerformingWork) {
      isHostCallbackScheduled = true;
      requestHostCallback();
    }
  }
}

function requestHostCallback() {
  if (!isMessageLoopRunning) {
    isMessageLoopRunning = true;
    schedulePerformWorkUntilDeadline();
  }
}

function performWorkUntilDeadline() {
  if (isMessageLoopRunning) {
    const currentTime = getCurrentTime();
    // 记录了一个work的起始时间，其实就是一个时间切片的起始时间，这是个时间戳
    startTime = currentTime;
    let hasMoreWork = true;
    try {
      hasMoreWork = flushWork(currentTime);
    } finally {
      if (hasMoreWork) {
        schedulePerformWorkUntilDeadline();
      } else {
        isMessageLoopRunning = false;
      }
    }
  }
}

const channel = new MessageChannel();
const port = channel.port2;
channel.port1.onmessage = performWorkUntilDeadline;
function schedulePerformWorkUntilDeadline() {
  port.postMessage(null);
}

function flushWork(initialTime: number) {
  isHostCallbackScheduled = false;
  isPerformingWork = true;

  let previousPriorityLevel = currentPriorityLevel;
  try {
    return workLoop(initialTime);
  } finally {
    currentTask = null;
    currentPriorityLevel = previousPriorityLevel;
    isPerformingWork = false;
  }
}
// 取消某个任务，由于最小堆没法直接删除，因此只能初步把 task.callback 设置为null
// 调度过程中，当这个任务位于堆顶时，删掉
function cancelCallback() {
  currentTask!.callback = null;
}

function getCurrentPriorityLevel(): PriorityLevel {
  return currentPriorityLevel;
}

// 有很多task，每个task都有一个callback，callback执行完了，就执行下一个task
// 一个work就是一个时间切片内执行的一些task
// 时间切片要循环，就是work要循环(loop)
// 返回为true，表示还有任务没有执行完，需要继续执行
function workLoop(initialTime: number): boolean {
  let currentTime = initialTime;
  advanceTimers(currentTime);
  currentTask = peek(taskQueue);
  while (currentTask !== null) {
    if (currentTask.expirationTime > currentTime && shouldYieldToHost(
      break;
    }
    // 执行任务
    const callback = currentTask.callback;
    if (typeof callback === "function") {
      // 有效的任务
      currentTask.callback = null;
      currentPriorityLevel = currentTask.priorityLevel;
      const didUserCallbackTimeout = currentTask.expirationTime <= cur
      const continuationCallback = callback(didUserCallbackTimeout);
      currentTime = getCurrentTime();
      if (typeof continuationCallback === "function") {
        currentTask.callback = continuationCallback;
        advanceTimers(currentTime);
        return true;
      } else {
        if (currentTask === peek(taskQueue)) {
          pop(taskQueue);
        }
        advanceTimers(currentTime);
      }
    } else {
      // 无效的任务
      pop(taskQueue);
    }

    currentTask = peek(taskQueue);
  }

  if (currentTask !== null) {
    return true;
  } else {
    const firstTimer = peek(timerQueue);
    if (firstTimer !== null) {
      requestHostTimeout(handleTimeout, firstTimer.startTime - current
    }

    return false;
  }
}

function requestHostTimeout(
  callback: (currentTime: number) => void,
  ms: number
) {
  taskTimeoutID = setTimeout(() => {
    callback(getCurrentTime());
  }, ms);
}

// delay任务处理逻辑
function cancelHostTimeout() {
  clearTimeout(taskTimeoutID);
  taskTimeoutID = -1;
}

function advanceTimers(currentTime: number) {
  let timer = peek(timerQueue);
  while (timer !== null) {
    if (timer.callback === null) {
      // 无效的任务
      pop(timerQueue);
    } else if (timer.startTime <= currentTime) {
      // 有效的任务
      // 任务已经到达开始时间，可以推入taskQueue
      pop(timerQueue);
      timer.sortIndex = timer.expirationTime;
      push(taskQueue, timer);
    } else {
      return;
    }
    timer = peek(timerQueue);
  }
}

function handleTimeout(currentTime: number) {
  isHostTimeoutScheduled = false;
  //  把延迟任务从timerQueue中推入taskQueue
  advanceTimers(currentTime);

  if (!isHostCallbackScheduled) {
    if (peek(taskQueue) !== null) {
      isHostCallbackScheduled = true;
      requestHostCallback();
    } else {
      const firstTimer = peek(timerQueue);
      if (firstTimer !== null) {
        requestHostTimeout(handleTimeout, firstTimer.startTime - curre
      }
    }
  }
}
// todo 实现一个单线程任务调度器
export {
  ImmediatePriority,
  UserBlockingPriority,
  NormalPriority,
  IdlePriority,
  LowPriority,
  scheduleCallback, // 某个任务进入调度器，等待调度
  cancelCallback, // 取消某个任务，由于最小堆没法直接删除，因此只能初步把 task.
  getCurrentPriorityLevel, // 获取当前正在执行任务的优先级
  shouldYieldToHost as shouldYield, // 把控制权交换给主线程
};
```