5-6 如何调度延迟任务

React `scheduler` 中有延迟任务的代码逻辑，但是这块代码在 React 目前的版本中并没有用到 ~

接下来这块逻辑的讲解，大家可以大作学习**任务调度器**的拓展内容来学习。

任务池

延迟任务在到执行时间之前有自己的任务池：

TypeScript

```
const timerQueue: Array<Task> = [];
```

把有延迟的任务放入任务池

TypeScript

```
// 任务调度器的入口函数
function scheduleCallback(
```

```
  priorityLevel: PriorityLevel,
  callback: Callback,
  options?: { delay: number }
) {
  var currentTime = getCurrentTime();

  var startTime;
  if (typeof options === "object" && options !== null) {
    var delay = options.delay;
    if (typeof delay === "number" && delay > 0) {
      startTime = currentTime + delay;
    } else {
      startTime = currentTime;
    }
  } else {
    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) {
      // This is a delayed task.
      newTask.sortIndex = startTime;
      push(timerQueue, newTask);
      if (peek(taskQueue) === null && newTask === peek(timerQueue)) {
        // All tasks are delayed, and this is the task with the earliest
        if (isHostTimeoutScheduled) {
          // Cancel an existing timeout.
          cancelHostTimeout();
        } else {
          isHostTimeoutScheduled = true;
        }
        requestHostTimeout(handleTimeout, startTime - currentTime);
      }
    } else {
      newTask.sortIndex = expirationTime;
      push(taskQueue, newTask);

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

倒计时延迟的任务

TypeScript

```
let taskTimeoutID = -1;

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;
}
```

倒计时结束

TypeScript

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

function handleTimeout(currentTime: number) {
  isHostTimeoutScheduled = false;
  advanceTimers(currentTime);

  if (!isHostCallbackScheduled) {
    if (peek(taskQueue) !== null) {
      isHostCallbackScheduled = true;
      requestHostCallback();
    } else {
      const firstTimer = peek(timerQueue);
      if (firstTimer !== null) {
```

```
        requestHostTimeout(handleTimeout, firstTimer.startTime - curre
      }
    }
  }
}

function advanceTimers(currentTime: number) {
  // Check for tasks that are no longer delayed and add them to the qu
  let timer = peek(timerQueue);
  while (timer !== null) {
    if (timer.callback === null) {
      // Timer was cancelled.
      pop(timerQueue);
    } else if (timer.startTime <= currentTime) {
      // Timer fired. Transfer to the task queue.
      pop(timerQueue);
      timer.sortIndex = timer.expirationTime;
      push(taskQueue, timer);
    } else {
      // Remaining timers are pending.
      return;
    }
    timer = peek(timerQueue);
  }
}
```