React 的调度系统 Scheduler

> React 版本为 18.2.0

React 使用了全新的 Fiber 架构，将原本需要一次性递归找出所有的改变，并一次性更新真实 DOM 的流程，改成通过时间分片，先分成一个个小的异步任务在空闲时间找出改变，最后一次性更新 DOM。

这里需要使用调度器，在浏览器空闲的时候去做这些异步小任务。

Scheduler

做这个调度工作的在 React 中叫做 Scheduler（调度器）模块。

其实浏览器是提供一个 **requestIdleCallback** 的方法，让我们可以在浏览器空闲的时去调用传入去的回调函数。但因为兼容性不好，给的优先级可能太低，执行是在渲染帧执行等缺点。

所以 React 实现了 requestIdleCallback 的替代方案，也就是这个 Scheduler。它的底层是 基于 MessageChannel 的。

为什么是 MessageChannel?

选择 MessageChannel 的原因，是首先异步得是个宏任务，因为宏任务中会在下次事件循环中执行，不会阻塞当前页面的更新。MessageChannel 是一个宏任务。

没选常见的 setTimeout，是因为MessageChannel 能较快执行，在 0～1ms 内触发，像 setTimeout 即便设置 timeout 为 0 还是需要 4～5ms。相同时间下，MessageChannel 能够完成更多的任务。

若浏览器不支持 MessageChannel，还是得降级为 setTimeout。

其实如果 setImmediate 存在的话，会优先使用 setImmediate，但它只在少量环境（比如 IE 的低版本、Node.js）中存在。

逻辑是在 packages/scheduler/src/forks/Scheduler.js 中实现的：

```js 复制代码
// Capture local references to native APIs, in case a polyfill overrides them.
const localSetTimeout = typeof setTimeout === 'function' ? setTimeout : null;
const localClearTimeout =
  typeof clearTimeout === 'function' ? clearTimeout : null;
const localSetImmediate =
  typeof setImmediate !== 'undefined' ? setImmediate : null; // IE and Node.js + jsdom

/***** 异步选择策略 *****/
// 【1】 优先使用 setImmediate
if (typeof localSetImmediate === 'function') {
  // Node.js and old IE.
  schedulePerformWorkUntilDeadline = () => {
    localSetImmediate(performWorkUntilDeadline);
  };
}
// 【2】 然后是 MessageChannel
else if (typeof MessageChannel !== 'undefined') {
  // DOM and Worker environments.
  // We prefer MessageChannel because of the 4ms setTimeout clamping.
  const channel = new MessageChannel();
  const port = channel.port2;
  channel.port1.onmessage = performWorkUntilDeadline;
  schedulePerformWorkUntilDeadline = () => {
    port.postMessage(null);
  };
}
// 【3】 最后是 setTimeout（兜底）
else {
  // We should only fallback here in non-browser environments.
  schedulePerformWorkUntilDeadline = () => {
    localSetTimeout(performWorkUntilDeadline, 0);
  };
}
```

另外，也没有选择使用 requestAnimationFrame，是因为它的机制比较特别，是在更新页面前执行，但更新页面的时机并没有规定，执行时机并不稳定。

底层的异步循环

requestHostCallback 方法，用于请求宿主（指浏览器）去执行函数。该方法会将传入的函数保存起来到 scheduledHostCallback 上，

然后调用 schedulePerformWorkUntilDeadline 方法。

schedulePerformWorkUntilDeadline 方法一调用，就停不下来了。

它会异步调用 performWorkUntilDeadline，后者又调用回 schedulePerformWorkUntilDeadline，最终实现 **不断地异步循环执行 performWorkUntilDeadline。**

```js
// 请求宿主（指浏览器）执行函数
function requestHostCallback(callback) {
  scheduledHostCallback = callback;
  if (!isMessageLoopRunning) {
    isMessageLoopRunning = true;
    schedulePerformWorkUntilDeadline();
  }
}
```

isMessageLoopRunning 是一个 flag，表示是否正在走循环。防止同一时间调用多次 schedulePerformWorkUntilDeadline。

React 会调度 workLoopSync / workLoopConcurrent

我们在 React 项目启动后，执行一个更新操作，会调用 ensureRootIsScheduled 方法。

```js
function ensureRootIsScheduled(root, currentTime) {
  // 最高优先级
  if (newCallbackPriority === SyncLane) {
    // Special case: Sync React callbacks are scheduled on a special
    // internal queue
    if (root.tag === LegacyRoot) {
      // Legacy Mode，即 ReactDOM.render() 启用的同步模式
      scheduleLegacySyncCallback(performSyncWorkOnRoot.bind(null, root));
    } else {
      scheduleSyncCallback(performSyncWorkOnRoot.bind(null, root));
    }
    // 立即执行优先级，去清空需要同步执行的任务
    scheduleCallback(ImmediateSchedulerPriority, flushSyncCallbacks);
  } else {
    // 初始化 schedulerPriorityLevel 并计算出 Scheduler 支持的优先级值
    let schedulerPriorityLevel;
    // ...
```

```
    scheduleCallback(
      schedulerPriorityLevel,
      performConcurrentWorkOnRoot.bind(null, root), // 并发模式
    );
  }
}
```

该方法有很多分支，最终会根据条件调用：

1. performSyncWorkOnRoot（立即执行）
2. performConcurrentWorkOnRoot（并发执行，且会用 scheduler 的 scheduleCallback 进行异步调用）

performSyncWorkOnRoot 最终会执行重要的 workLoopSync 方法：

```
// 调用链路：
// performSyncWorkOnRoot -> renderRootSync -> workLoopSync
function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
```

workInProgress 表示一个需要进行处理的 FiberNode。

performUnitOfWork 方法用于处理一个 workInProgress，进行调和操作，计算出新的 fiberNode。

同样，performConcurrentWorkOnRoot 最终会执行重要的 workLoopConcurrent 方法。

```
// 调用链路：
// performConcurrentWorkOnRoot -> performConcurrentWorkOnRoot -> renderRootConcurrent
function workLoopConcurrent() {
  while (workInProgress !== null && !shouldYield()) {
    performUnitOfWork(workInProgress);
  }
}
```

和 workLoopSync 很相似，但循环条件里多了一个来自 Scheduler 的 `shouldYield()` 决定是否将进程让出给浏览器，这样就能做到中断 Fiber 的调和阶段，做到时间分片。

scheduleCallback

上面的 workLoopSync 和 workLoopConcurrent 都是通过 scheduleCallback 去调度的。

scheduleCallback 方法传入优先级 priorityLevel、需要指定的回调函数 callback， 以及一个可选项 options。

scheduleCallback 的实现如下（做了简化）：

```
function unstable_scheduleCallback(priorityLevel, callback, options) {
  var currentTime = getCurrentTime();

  var startTime;
  if (options?.delay) {
    startTime = currentTime + options.delay;
  }
  // 有效期时长，根据优先级设置。
  var timeout;
  // ...
  // 计算出 过期时间点
  var expirationTime = startTime + timeout;

  // 创建一个任务
  var newTask = {
    id: taskIdCounter++,
    callback, // 这个就是任务本身
    priorityLevel,
    startTime,
    expirationTime,
    sortIndex: -1,
  };

  // 说明新任务是加了 option.delay 的任务，需要延迟执行
  // 我们会放到未逾期队列（timerQueue）中
  if (startTime > currentTime) {
    newTask.sortIndex = startTime;
    push(timerQueue, newTask);
    // 没有需要逾期的任务，且优先级最高的未逾期任务就是这个新任务
    if (peek(taskQueue) === null && newTask === peek(timerQueue)) {
      // 那，用 setTimeout 延迟 options.delay 执行 handleTimeout
        requestHostTimeout(handleTimeout, startTime - currentTime);
    }
  }
  // 立即执行的任务，加入到逾期队列（taskQueue）
  else {
    newTask.sortIndex = expirationTime;
```

```
    push(taskQueue, newTask);

    // Schedule a host callback, if needed. If we're already performing work,
    // wait until the next time we yield.
    if (!isHostCallbackScheduled && !isPerformingWork) {
      isHostCallbackScheduled = true;
      requestHostCallback(flushWork);
    }
  }
}
```

`push / peek / pop` 这些是 scheduler 提供的操作 **优先级队列** 的操作方法。

优先级队列的底层实现是小顶堆，实现原理不展开讲。我们只需要记住优先级队列的特性：**就是出队的时候，会取优先级最高的任务**。在 scheduler 中，**sortIndex 最小的任务的优先级最高**。

`push(queue, task)` 表示入队，加一个新任务；`peek(queue)` 表示得到最高优先级（不出队）；`pop(queue)` 表示将最高优先级任务出队。

taskQueue 为逾期的任务队列，需要赶紧执行。新生成的任务（没有设置 options.delay）会放到 taskQueue，并以 expirationTime 作为优先级（sortIndex）来比较。

timerQueue 是还没逾期的任务队列，以 startTime 作为优先级来比较。如果逾期了，就会 取出放到 taskQueue 里。

handleTimeout

```
// 如果没有逾期的任务，且优先级最高的未逾期任务就是这个新任务
// 延迟执行 handleTimeout
if (peek(taskQueue) === null && newTask === peek(timerQueue)) {
  requestHostTimeout(handleTimeout, startTime - currentTime);
}
```

requestHostTimeout 其实就是 setTimeout 定时器的简单封装，在 newTask 过期的时间点（`startTime - currentTime` 后）执行 handleTimeout。

```
function handleTimeout(currentTime) {
  isHostTimeoutScheduled = false;
  advanceTimers(currentTime); // 更新 timerQueue 和 taskQueue
```

```
  if (!isHostCallbackScheduled) {
    if (peek(taskQueue) !== null) { // 有要执行的逾期任务
      isHostCallbackScheduled = true;
      requestHostCallback(flushWork); // 清空 taskQueue 任务
    } else { // 没有逾期任务
      const firstTimer = peek(timerQueue);
      if (firstTimer !== null) { // 但有未逾期任务，用 setTimeout 晚点再调用自己
        requestHostTimeout(handleTimeout, firstTimer.startTime - currentTime);
      }
    }
  }
}
```

handleTimeout 下会调用 advanceTimers 方法，根据当前时间要将 timerTask 中逾期的任务搬到 taskQueue 下。

（advanceTimers 这个方法会在多个位置被调用。搬一搬，更健康）

搬完后，看看 taskQueue 有没有任务要做，有的话就调用 flushWork 清空 taskQueue 任务。没有的话看看有没有未逾期任务，用定时器在它过期的时间点再递归执行 handleTimeout。

workLoop

flushWork 会 调用 workLoop。flushWork 还需要做一些额外的修改模块文件变量的操作。

```
function flushWork(hasTimeRemaining, initialTime) {
  // ...
        return workLoop(hasTimeRemaining, initialTime);
}
```

workLoop 会不停地从 taskQueue 取出任务来执行。其核心逻辑为：

```
function workLoop(hasTimeRemaining, initialTime) {
  // 更新 taskQueue，并取出一个任务
  let currentTime = initialTime;
  advanceTimers(currentTime);
  currentTask = peek(taskQueue);

  while (currentTask !== null) {
    if (
```

```
      currentTask.expirationTime > currentTime &&
      (!hasTimeRemaining || shouldYieldToHost())
    ) {
      // This currentTask hasn't expired, and we've reached the deadline.
      break;
    }
    // 执行任务
    const callback = currentTask.callback;
    callback();

    // 更新 taskQueue，并取出一个任务
    currentTime = getCurrentTime();
    advanceTimers(currentTime);
    currentTask = peek(taskQueue);
  }
  return currentTask !== null;
}
```

shouldYieldToHost

上面的循环并不是一直会执行到 currentTask 为 null 为止，在必要的时候还是会跳出的。我们是通过 shouldYieldToHost 方法判断是否要跳出。

此外，Fiber 异步更新的 workLoopConcurrent 方法用到的 shouldYield，其实就是这个 shouldYieldToHost。

shouldYieldToHost 核心实现：

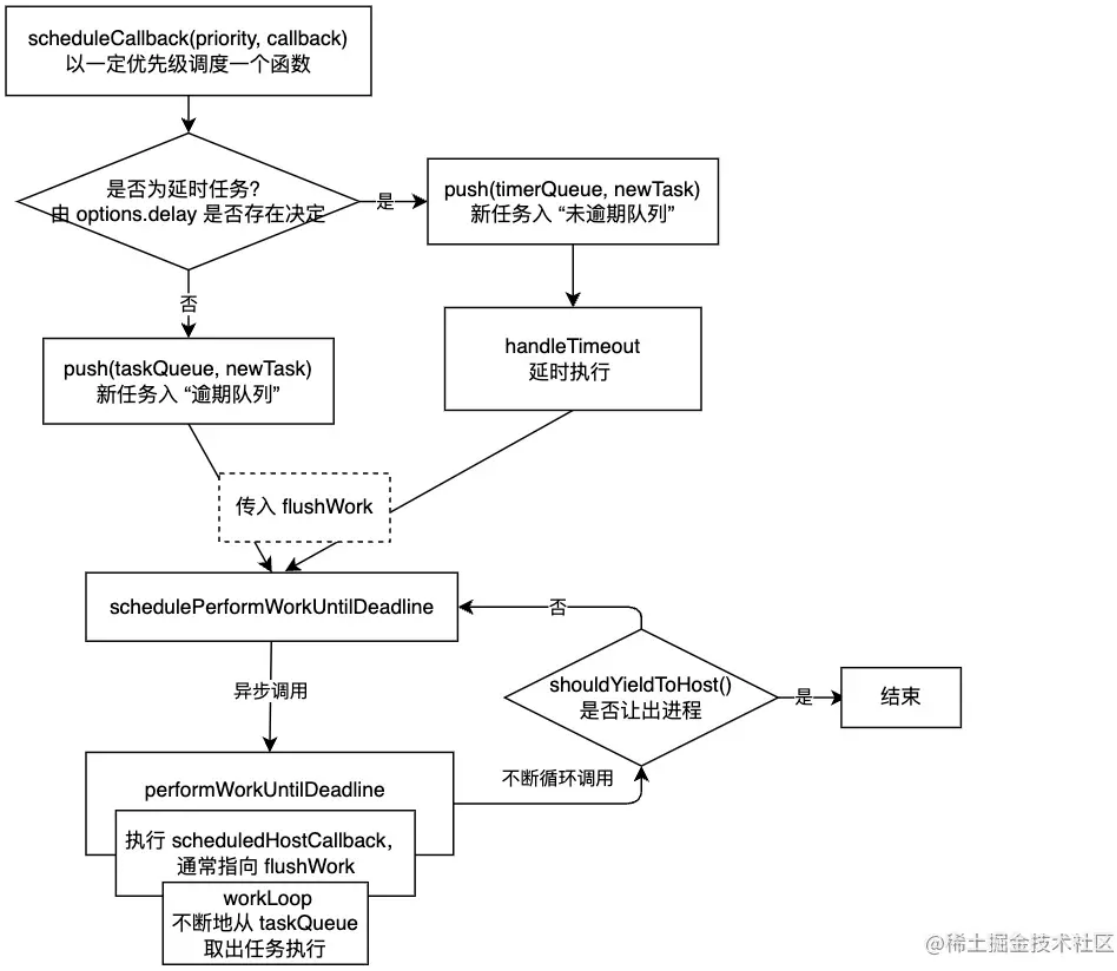
```
const frameYieldMs = 5;
var frameInterval = frameYieldMs;

function shouldYieldToHost() {
  var timeElapsed = getCurrentTime() - startTime;
  // 经过的时间小于 5 ms，不需要让出进程
  if (timeElapsed < frameInterval) {
    return false;
  }
  return true;
}

export {
  // 会重命名为 unstable_shouldYield 导出
  shouldYieldToHost as unstable_shouldYield,
}
```

计算经过的时间，如果小于帧间隔时间（frameInterval，通常为 5ms），不需要让出进程，否则让出。

startTime 是模块文件的最外层变量，会在 performWorkUntilDeadline 方法中赋值，也就是任务开始调度的时候。

流程图

试着画一下 Scheduler 的调度流程图。



结尾

Scheduler 一套下来还是挺复杂的。

首先是 Scheduler 底层大多数情况下会使用 MessageChannel，作为循环执行异步任务的能力。通过它来不断地执行任务队列中的任务。

任务队列是特殊的优先级队列，特性是出队时，拿到优先级最高的任务（在 Scheduler 中对比的是 sortIndex，值是一个时间戳）。

任务队列在 Scheduler 中有两种。一种是逾期任务 taskQueue，需要赶紧执行，另一种是延期任务 timerQueue，还不到时间执行。Scheduler 会根据当前时间，将逾期的 timerQueue 任务放到 taskQueue 中，然后从 taskQueue 取出优先级最高的任务去执行。

Scheduler 向外暴露 scheduleCallback 方法，该方法接受一个优先级和一个函数（就是任务），对于 React 来说，它通常是 workLoopSync 或 workLoopConcurrent。

scheduleCallback 会设置新任务的过期时间（根据优先级），并判断是否为延时任务（根据 options.delay）决定放入哪个任务队列中。然后启用循环执行异步任务，不断地清空执行 taskQueue。

Scheduler 也向外暴露了 shouldYield，通过它可以知道是否执行时间过长，应该让出进程给浏览器。该方法同时也在 Scheduler 内部的循环执行异步任务中作为一种打断循环的判断条件。

React 的并发模式下，可以用它作为暂停调和阶段的依据。