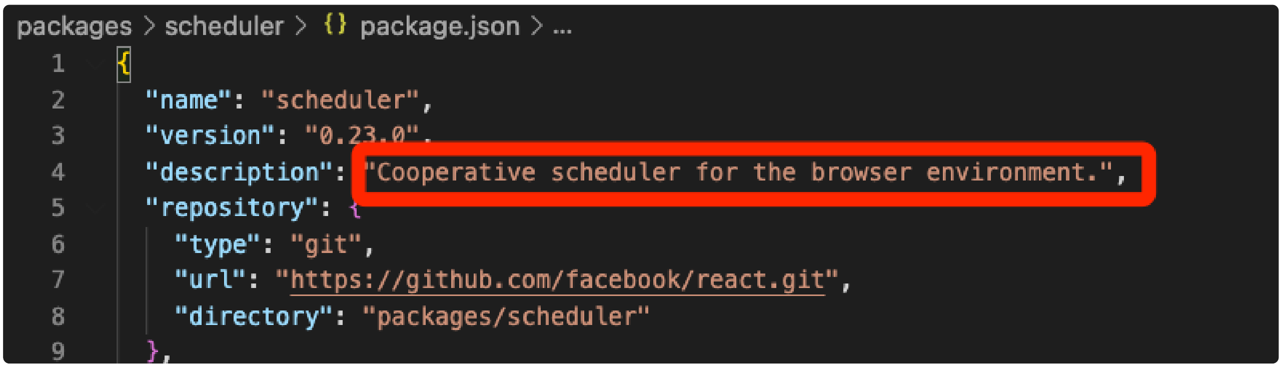
6-1 剖析 React 中的任务调度器场景：合作式调度器 & 抢占式调度器

合作式调度器 & 抢占式调度器

Cooperative Scheduler & Preemptive Scheduler

React 中用到了两种任务调度方案：合作式调度和抢占式调度。



packages > scheduler > {} package.json > ...

```
{
  "name": "scheduler",
  "version": "0.23.0",
  "description": "Cooperative scheduler for the browser environment.",
  "repository": {
    "type": "git",
    "url": "https://github.com/facebook/react.git",
    "directory": "packages/scheduler"
  },
```

在浏览器环境中，合作式调度器(Cooperative Scheduler)，是一种调度机制，用于管理和分配任务的执行。

传统的抢占式调度器(Preemptive Scheduler)虽然 CPU 利用率比较高，但是容易出现“饿死现象”。(对于饿死现象，常见解决办法是定期检查，对于长时间没法得到处理的低优先级任务，即快要饿死的任务，提高它们的优先级，以此避免任务饿死。React 也是采用的这种方案。)

与传统的抢占式调度器不同，Cooperative Scheduler 依赖于任务主动释放执行权，而不是由 scheduler 强制中断任务。

在 Cooperative Scheduler 中，每个任务负责自己的执行，并在适当的时机将执行权交还给 scheduler。这种方式可以避免长时间运行的任务阻塞其它任务的执行，提高整体的响应性和性能。

Cooperative Scheduler 在处理 IO 操作、事件处理等场景下非常有用，可以避免阻塞浏览器的主线程，提升用户体验。

如何避免饿死

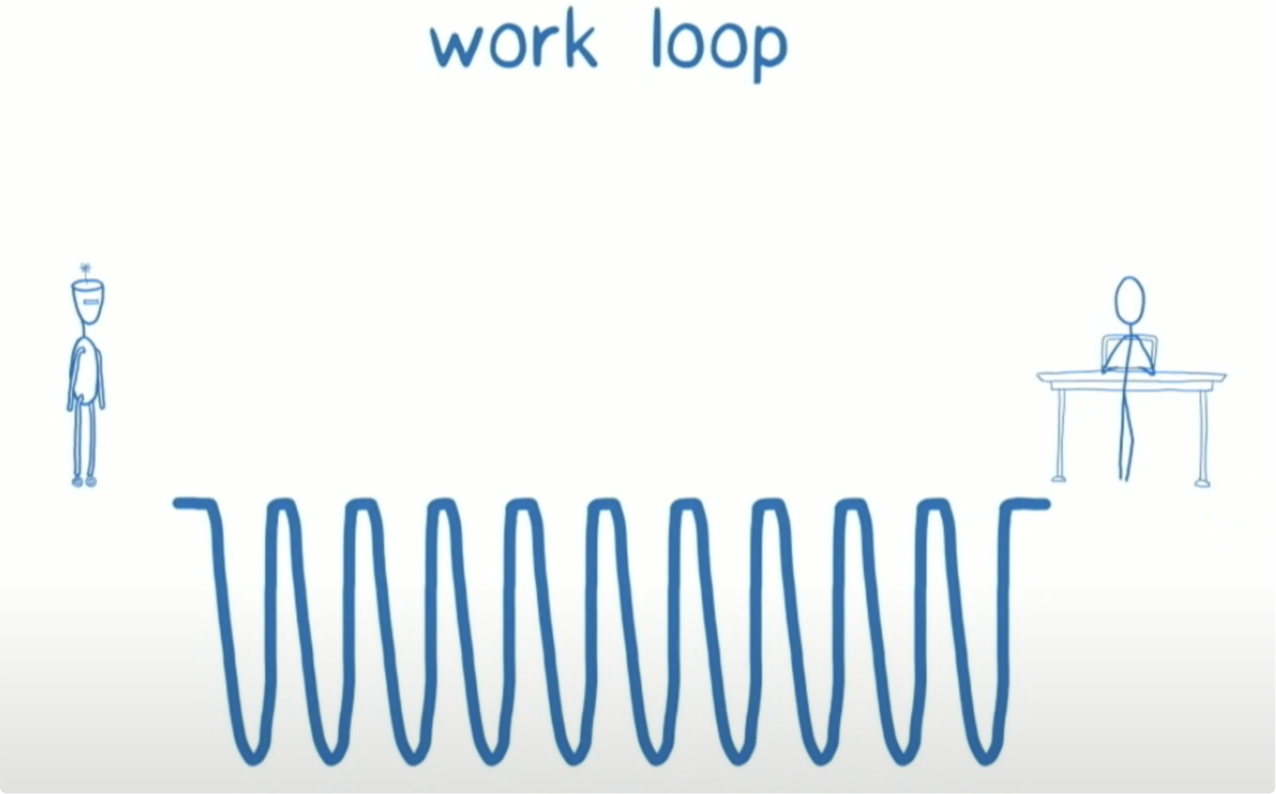
React 中用到了两种避免任务“`饿死`”的方案：

时间切片

react/schduler

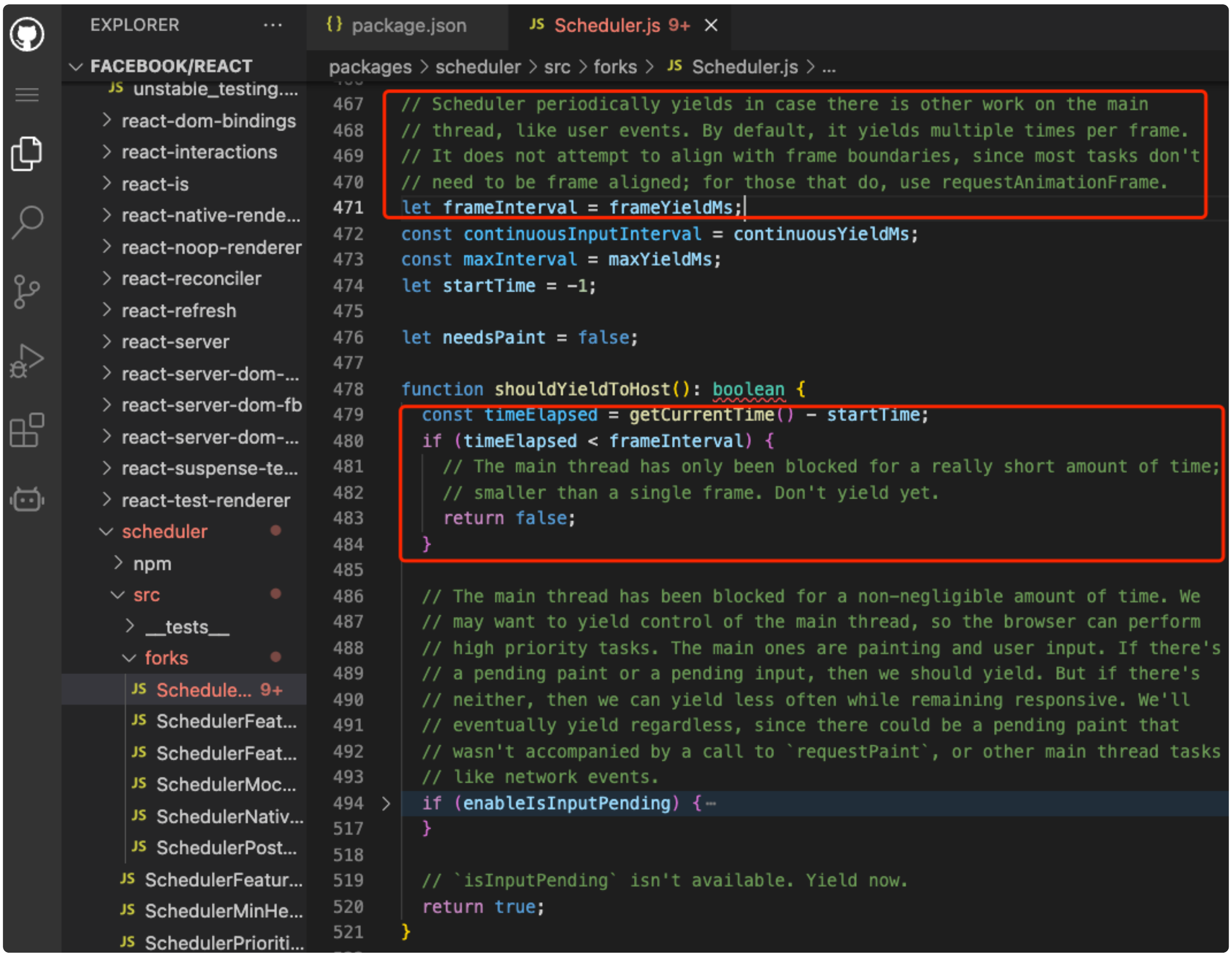
时间切片，time slices，即划分时间段，避免某些任务长期占着主线程导致其它高优先级任务无法得到立即处理。

这种方式可以避免长时间运行的任务阻塞其它任务的执行，提高整体的响应性和性能。



代码实现

https://github1s.com/facebook/react/blob/HEAD/packages/scheduler/src/forks/Scheduler.js



```
// Scheduler periodically yields in case there is other work on the main
// thread, like user events. By default, it yields multiple times per frame.
// It does not attempt to align with frame boundaries, since most tasks don't
// need to be frame aligned; for those that do, use requestAnimationFrame.
let frameInterval = frameYieldMs;
const continuousInputInterval = continuousYieldMs;
const maxInterval = maxYieldMs;
let startTime = -1;

let needsPaint = false;

function shouldYieldToHost(): boolean {
  const timeElapsed = getCurrentTime() - startTime;
  if (timeElapsed < frameInterval) {
    // The main thread has only been blocked for a really short amount of time;
    // smaller than a single frame. Don't yield yet.
    return false;
  }

  // The main thread has been blocked for a non-negligible amount of time. We
  // may want to yield control of the main thread, so the browser can perform
  // high priority tasks. The main ones are painting and user input. If there's
  // a pending paint or a pending input, then we should yield. But if there's
  // neither, then we can yield less often while remaining responsive. We'll
  // eventually yield regardless, since there could be a pending paint that
  // wasn't accompanied by a call to `requestPaint`, or other main thread tasks
  // like network events.
  if (enableIsInputPending) {⋯
  }

  // `isInputPending` isn't available. Yield now.
  return true;
}
```

aging

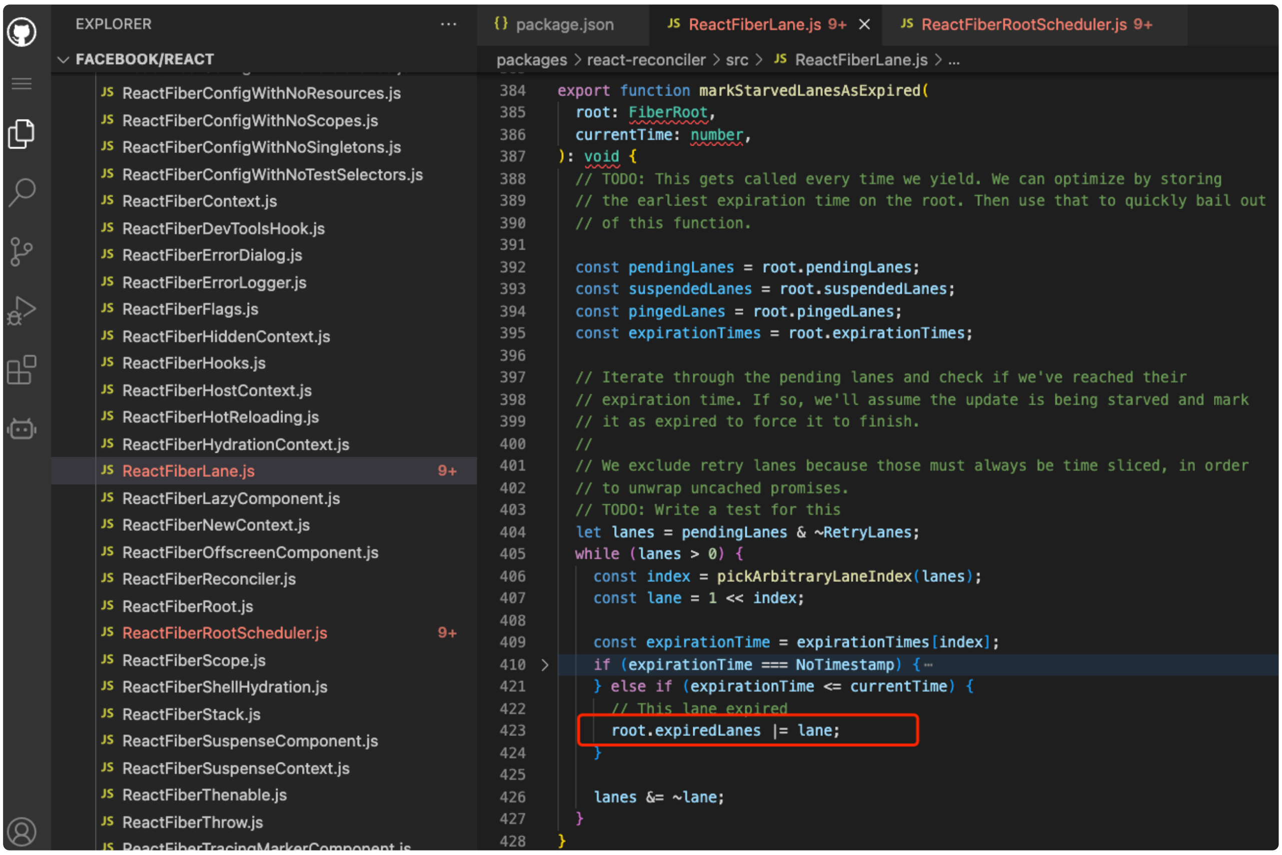
react/react-scheduler

对于长时间没法得到处理的低优先级任务，即快要饿死的任务，提高它们的优先级，以此避免任务饿死。

代码实现

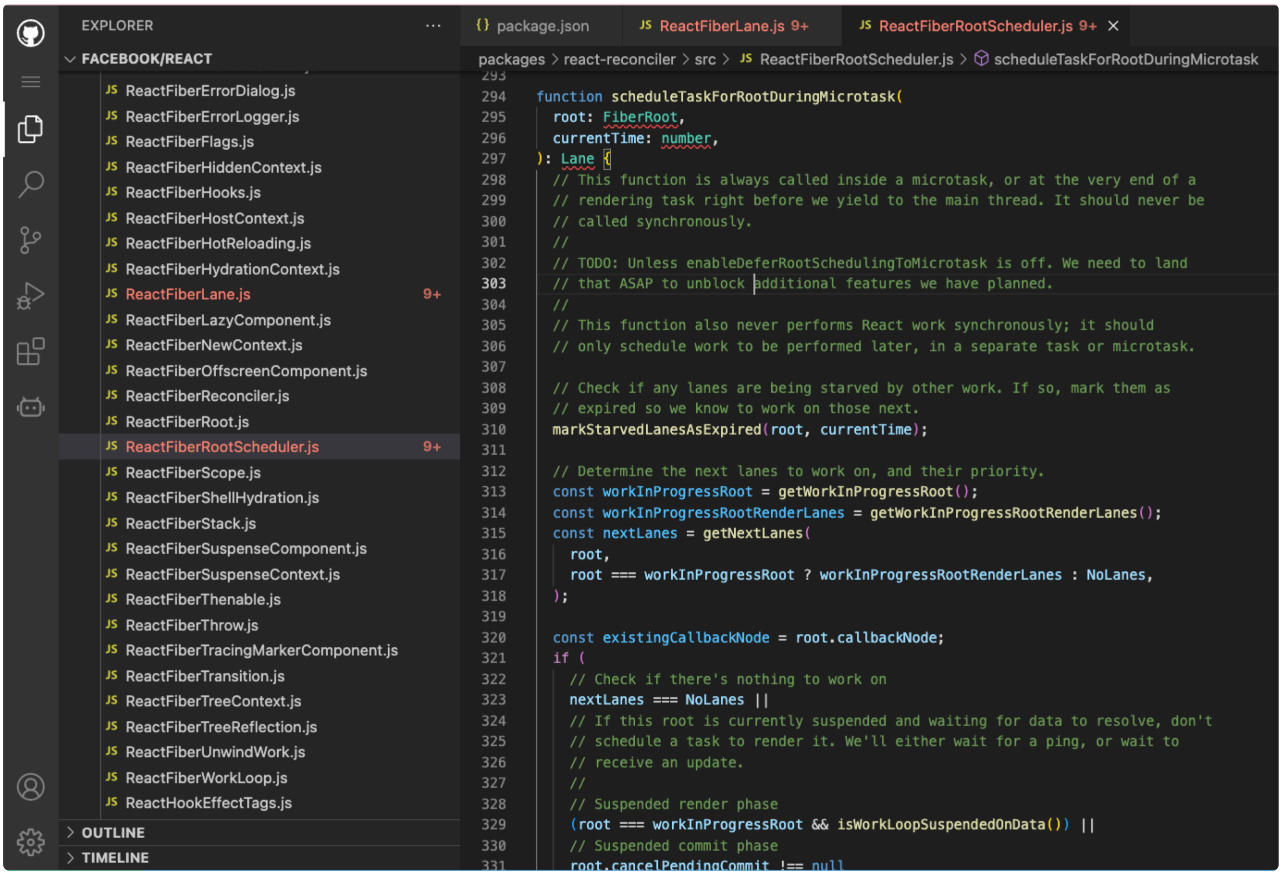
`markStarvedLanesAsExpired` 函数用于把饿死任务标记为过期，相当于上文提到的提高优先级，以使之得到尽快的完成。

这里具体的做法是，遍历待处理的任务，即遍历 lanes，检查它们是否过期。如果已经过期，就认为该任务处于即将饿死的状态，然后标记为已经过期，以强制它的完成。


EXPLORER
FACEBOOK/REACT
ReactFiberConfigWithNoResources.js
ReactFiberConfigWithNoScopes.js
ReactFiberConfigWithNoSingletons.js
ReactFiberConfigWithNoTestSelectors.js
ReactFiberContext.js
ReactFiberDevToolsHook.js
ReactFiberErrorDialog.js
ReactFiberErrorLogger.js
ReactFiberFlags.js
ReactFiberHiddenContext.js
ReactFiberHooks.js
ReactFiberHostContext.js
ReactFiberHotReloading.js
ReactFiberHydrationContext.js
ReactFiberLane.js 9+
ReactFiberLazyComponent.js
ReactFiberNewContext.js
ReactFiberOffscreenComponent.js
ReactFiberReconciler.js
ReactFiberRoot.js
ReactFiberRootScheduler.js 9+
ReactFiberScope.js
ReactFiberShellHydration.js
ReactFiberStack.js
ReactFiberSuspenseComponent.js
ReactFiberSuspenseContext.js
ReactFiberThenable.js
ReactFiberThrow.js
ReactFiberTracingMarkerComponent.js
package.json
ReactFiberLane.js 9+
ReactFiberRootScheduler.js 9+
packages > react-reconciler > src > ReactFiberLane.js > ...

```
export function markStarvedLanesAsExpired(
  root: FiberRoot,
  currentTime: number,
): void {
  // TODO: This gets called every time we yield. We can optimize by storing
  // the earliest expiration time on the root. Then use that to quickly bail out
  // of this function.

  const pendingLanes = root.pendingLanes;
  const suspendedLanes = root.suspendedLanes;
  const pingedLanes = root.pingedLanes;
  const expirationTimes = root.expirationTimes;

  // Iterate through the pending lanes and check if we've reached their
  // expiration time. If so, we'll assume the update is being starved and mark
  // it as expired to force it to finish.
  //
  // We exclude retry lanes because those must always be time sliced, in order
  // to unwrap uncached promises.
  // TODO: Write a test for this
  let lanes = pendingLanes & ~RetryLanes;
  while (lanes > 0) {
    const index = pickArbitraryLaneIndex(lanes);
    const lane = 1 << index;

    const expirationTime = expirationTimes[index];
    if (expirationTime === NoTimestamp) {⋯
    } else if (expirationTime <= currentTime) {
      // This lane expired
      root.expiredLanes |= lane;
    }

    lanes &= ~lane;
  }
}
```

`scheduleTaskForRootDuringMicrotask` 函数会在执行微任务的函数内部调用，或者是在交换控制权给主线程之前在渲染任务的结尾调用。它不能被同步调用。



```
function scheduleTaskForRootDuringMicrotask(
  root: FiberRoot,
  currentTime: number,
): Lane {
  // This function is always called inside a microtask, or at the very end of a
  // rendering task right before we yield to the main thread. It should never be
  // called synchronously.
  //
  // TODO: Unless enableDeferRootSchedulingToMicrotask is off. We need to land
  // that ASAP to unblock additional features we have planned.
  //
  // This function also never performs React work synchronously; it should
  // only schedule work to be performed later, in a separate task or microtask.

  // Check if any lanes are being starved by other work. If so, mark them as
  // expired so we know to work on those next.
  markStarvedLanesAsExpired(root, currentTime);

  // Determine the next lanes to work on, and their priority.
  const workInProgressRoot = getWorkInProgressRoot();
  const workInProgressRootRenderLanes = getWorkInProgressRootRenderLanes();
  const nextLanes = getNextLanes(
    root,
    root === workInProgressRoot ? workInProgressRootRenderLanes : NoLanes,
  );

  const existingCallbackNode = root.callbackNode;
  if (
    // Check if there's nothing to work on
    nextLanes === NoLanes ||
    // If this root is currently suspended and waiting for data to resolve, don't
    // schedule a task to render it. We'll either wait for a ping, or wait to
    // receive an update.
    //
    // Suspended render phase
    (root === workInProgressRoot && isWorkLoopSuspendedOnData()) ||
    // Suspended commit phase
    root.cancelPendingCommit !== null
```