react源码中的协调与调度

requestEventTime

其实在 `React` 执行过程中，会有数不清的任务要去执行，但是他们会有一个 `优先级的判定` ，假如两个事件的 `优先级一样` ，那么 `React` 是怎么去判定他们两谁先执行呢？

javascript 复制代码

```javascript
// packages/react-reconciler/src/ReactFiberWorkLoop.old.js
export function requestEventTime() {
  if ((executionContext & (RenderContext | CommitContext)) !== NoContext) {
    // We're inside React, so it's fine to read the actual time.
    // react事件正在执行
    // executionContext
    // RenderContext 正在计算
    // CommitContext 正在提交
    // export const NoContext = /*             */ 0b0000000;
    // const BatchedContext = /*               */ 0b0000001;
    // const EventContext = /*                 */ 0b0000010;
    // const DiscreteEventContext = /*         */ 0b0000100;
    // const LegacyUnbatchedContext = /*       */ 0b0001000;
    // const RenderContext = /*                */ 0b0010000;
    // const CommitContext = /*                */ 0b0100000;
    // export const RetryAfterError = /*       */ 0b1000000;
    return now();
  }
  // 没有在react事件执行 NoTimestamp === -1
  if (currentEventTime !== NoTimestamp) {
    // 浏览器事件正在执行，返回上次的 currentEventTime
    return currentEventTime;
  }
  // 重新计算currentEventTime，当执行被中断后
  currentEventTime = now();
  return currentEventTime;
}
```

- `RenderContext` 与 `CommitContext` 表示正在计算更新和正在提交更新，返回 `now()` 。
- 如果是浏览器事件正在执行中，返回上一次的 `currentEventTime` 。
- 如果终止或者 `中断react` 任务执行的时候，则重新获取执行时间 `now(`)。
- 获取的时间 `越小` ，则执行的优先级 `越高` 。

`now()` 并不是单纯的 `new Date()` ，而是判定两次更新任务的时间是否 `小于10ms` ，来决定是否 `复用` 上一次的更新时间 `Scheduler_now` 的。

javascript 复制代码
```javascript
export const now = initialTimeMs < 10000 ? Scheduler_now : () => Scheduler_now() - initialTimeMs;
```

其实各位猜想一下，对于 `10ms` 级别的任务间隙时间，几乎是可以忽略不计的，那么这里就可以视为同样的任务，不需要有很大的性能开销，有利于 `批量更新` 。

requestUpdateLane

requestEventTime位每一个需要执行的任务打上了触发更新时间标签，那么任务的优先级还需要进一步的确立，requestUpdateLane就是用来获取每一个任务执行的优先级的。

javascript 复制代码
```javascript
// packages/react-reconciler/src/ReactFiberWorkLoop.old.js
export function requestUpdateLane(fiber: Fiber): Lane {
  // Special cases
  const mode = fiber.mode;
  if ((mode & BlockingMode) === NoMode) {
    return (SyncLane: Lane);
  } else if ((mode & ConcurrentMode) === NoMode) {
    return getCurrentPriorityLevel() === ImmediateSchedulerPriority
      ? (SyncLane: Lane)
      : (SyncBatchedLane: Lane);
  } else if (
    !deferRenderPhaseUpdateToNextBatch &&
    (executionContext & RenderContext) !== NoContext &&
    workInProgressRootRenderLanes !== NoLanes
  ) {
    // This is a render phase update. These are not officially supported. The
    // old behavior is to give this the same "thread" (expiration time) as
    // whatever is currently rendering. So if you call `setState` on a component
    // that happens later in the same render, it will flush. Ideally, we want to
    // remove the special case and treat them as if they came from an
    // interleaved event. Regardless, this pattern is not officially supported.
    // This behavior is only a fallback. The flag only exists until we can roll
    // out the setState warning, since existing code might accidentally rely on
    // the current behavior.
    return pickArbitraryLane(workInProgressRootRenderLanes);
  }

  // The algorithm for assigning an update to a lane should be stable for all
  // updates at the same priority within the same event. To do this, the inputs
  // to the algorithm must be the same. For example, we use the `renderLanes`
  // to avoid choosing a lane that is already in the middle of rendering.
```

```
//
// However, the "included" lanes could be mutated in between updates in the
// same event, like if you perform an update inside `flushSync`. Or any other
// code path that might call `prepareFreshStack`.
//
// The trick we use is to cache the first of each of these inputs within an
// event. Then reset the cached values once we can be sure the event is over.
// Our heuristic for that is whenever we enter a concurrent work loop.
//
// We'll do the same for `currentEventPendingLanes` below.
if (currentEventWipLanes === NoLanes) {
  currentEventWipLanes = workInProgressRootIncludedLanes;
}

const isTransition = requestCurrentTransition() !== NoTransition;
if (isTransition) {
  if (currentEventPendingLanes !== NoLanes) {
    currentEventPendingLanes =
      mostRecentlyUpdatedRoot !== null
        ? mostRecentlyUpdatedRoot.pendingLanes
        : NoLanes;
  }
  return findTransitionLane(currentEventWipLanes, currentEventPendingLanes);
}

// TODO: Remove this dependency on the Scheduler priority.
// To do that, we're replacing it with an update lane priority.

// 获取执行任务的优先级，便于调度
const schedulerPriority = getCurrentPriorityLevel();

// The old behavior was using the priority level of the Scheduler.
// This couples React to the Scheduler internals, so we're replacing it
// with the currentUpdateLanePriority above. As an example of how this
// could be problematic, if we're not inside `Scheduler.runWithPriority`,
// then we'll get the priority of the current running Scheduler task,
// which is probably not what we want.
let lane;
if (
  // TODO: Temporary. We're removing the concept of discrete updates.
  (executionContext & DiscreteEventContext) !== NoContext &&

  // 用户block的类型事件
  schedulerPriority === UserBlockingSchedulerPriority
) {
  // 通过findUpdateLane函数重新计算lane
  lane = findUpdateLane(InputDiscreteLanePriority, currentEventWipLanes);
} else {
  // 根据优先级计算法则计算lane
```

```javascript
    const schedulerLanePriority = schedulerPriorityToLanePriority(
      schedulerPriority,
    );

    if (decoupleUpdatePriorityFromScheduler) {
      // In the new strategy, we will track the current update lane priority
      // inside React and use that priority to select a lane for this update.
      // For now, we're just logging when they're different so we can assess.
      const currentUpdateLanePriority = getCurrentUpdateLanePriority();

      if (
        schedulerLanePriority !== currentUpdateLanePriority &&
        currentUpdateLanePriority !== NoLanePriority
      ) {
        if (__DEV__) {
          console.error(
            'Expected current scheduler lane priority %s to match current update lane priority %s',
            schedulerLanePriority,
            currentUpdateLanePriority,
          );
        }
      }
    }
    // 根据计算得到的 schedulerLanePriority，计算更新的优先级 lane
    lane = findUpdateLane(schedulerLanePriority, currentEventWipLanes);
  }

  return lane;
}
```

- 通过 `getCurrentPriorityLevel` 获得所有执行任务的调度优先级 `schedulerPriority` 。
- 通过 `findUpdateLane` 计算 `lane` ，作为更新中的优先级。

findUpdateLane

javascript 复制代码

```javascript
export function findUpdateLane(
  lanePriority: LanePriority,  wipLanes: Lanes,
): Lane {
  switch (lanePriority) {
    case NoLanePriority:
      break;
    case SyncLanePriority:
      return SyncLane;
    case SyncBatchedLanePriority:
      return SyncBatchedLane;
    case InputDiscreteLanePriority: {
      const lane = pickArbitraryLane(InputDiscreteLanes & ~wipLanes);
      if (lane === NoLane) {
        // Shift to the next priority level
        return findUpdateLane(InputContinuousLanePriority, wipLanes);
      }
      return lane;
    }
    case InputContinuousLanePriority: {
      const lane = pickArbitraryLane(InputContinuousLanes & ~wipLanes);
      if (lane === NoLane) {
        // Shift to the next priority level
        return findUpdateLane(DefaultLanePriority, wipLanes);
      }
      return lane;
    }
    case DefaultLanePriority: {
      let lane = pickArbitraryLane(DefaultLanes & ~wipLanes);
      if (lane === NoLane) {
        // If all the default lanes are already being worked on, look for a
        // lane in the transition range.
        lane = pickArbitraryLane(TransitionLanes & ~wipLanes);
        if (lane === NoLane) {
          // All the transition lanes are taken, too. This should be very
          // rare, but as a last resort, pick a default lane. This will have
          // the effect of interrupting the current work-in-progress render.
          lane = pickArbitraryLane(DefaultLanes);
        }
      }
      return lane;
    }
    case TransitionPriority: // Should be handled by findTransitionLane instead
    case RetryLanePriority: // Should be handled by findRetryLane instead
      break;
    case IdleLanePriority:
      let lane = pickArbitraryLane(IdleLanes & ~wipLanes);
      if (lane === NoLane) {
        lane = pickArbitraryLane(IdleLanes);
      }
      return lane;
    default:
      // The remaining priorities are not valid for updates
      break;
  }
  invariant(
    false,
    'Invalid update priority: %s. This is a bug in React.',
    lanePriority,
  );
}
```

相关参考视频讲解：进入学习

lanePriority: LanePriority

javascript 复制代码

```javascript
export opaque type LanePriority =
  | 0
  | 1
  | 2
  | 3
  | 4
  | 5
  | 6
  | 7
  | 8
  | 9
  | 10
  | 11
  | 12
  | 13
  | 14
  | 15
  | 16
  | 17;
export opaque type Lanes = number;
export opaque type Lane = number;
export opaque type LaneMap<T> = Array<T>;

import {
  ImmediatePriority as ImmediateSchedulerPriority,
  UserBlockingPriority as UserBlockingSchedulerPriority,
  NormalPriority as NormalSchedulerPriority,
  LowPriority as LowSchedulerPriority,
  IdlePriority as IdleSchedulerPriority,
  NoPriority as NoSchedulerPriority,
} from './SchedulerWithReactIntegration.new';

// 同步任务
export const SyncLanePriority: LanePriority = 15;
export const SyncBatchedLanePriority: LanePriority = 14;

// 用户事件
const InputDiscreteHydrationLanePriority: LanePriority = 13;
export const InputDiscreteLanePriority: LanePriority = 12;

const InputContinuousHydrationLanePriority: LanePriority = 11;
export const InputContinuousLanePriority: LanePriority = 10;
```

```
const DefaultHydrationLanePriority: LanePriority = 9;
export const DefaultLanePriority: LanePriority = 8;

const TransitionHydrationPriority: LanePriority = 7;
export const TransitionPriority: LanePriority = 6;

const RetryLanePriority: LanePriority = 5;

const SelectiveHydrationLanePriority: LanePriority = 4;

const IdleHydrationLanePriority: LanePriority = 3;
const IdleLanePriority: LanePriority = 2;

const OffscreenLanePriority: LanePriority = 1;

export const NoLanePriority: LanePriority = 0;
```

createUpdate

javascript 复制代码

```javascript
export function createUpdate(eventTime: number, lane: Lane): Update<*> {
  const update: Update<*> = {
    eventTime, // 更新时间
    lane, // 优先级

    tag: UpdateState,//更新
    payload: null,// 需要更新的内容
    callback: null, // 更新完后的回调

    next: null, // 指向下一个更新
  };
  return update;
}
```

`createUpdate` 函数入参为 `eventTime` 和 `lane` ，输出一个 `update` 对象，而对象中的 `tag` 表示此对象要进行什么样的操作。

javascript 复制代码

```javascript
export const UpdateState = 0;// 更新
export const ReplaceState = 1;//替换
export const ForceUpdate = 2;//强制更新
export const CaptureUpdate = 3;//xx更新
```

- `createUpdate` 就是单纯的给每一个任务进行包装，作为一个个体推入到更新队列中。

enqueueUpdate

javascript 复制代码

```javascript
export function enqueueUpdate<State>(fiber: Fiber, update: Update<State>) {
  // 获取当前更新队列？为啥呢？因为无法保证react是不是还有正在更新或者没有更新完毕的任务
  const updateQueue = fiber.updateQueue;
  //  如果更新队列为空，则表示fiber还未渲染，直接退出
  if (updateQueue === null) {
    // Only occurs if the fiber has been unmounted.
    return;
  }

  const sharedQueue: SharedQueue<State> = (updateQueue: any).shared;
  const pending = sharedQueue.pending;
  if (pending === null) {
    // This is the first update. Create a circular list.
     // 还记得那个更新对象吗？update.next =>
     // 如果pedding位null，表示第一次渲染，那么他的指针为update本身
    update.next = update;
  } else {
    // 将update插入到更新队列循环当中
    update.next = pending.next;
    pending.next = update;
  }
  sharedQueue.pending = update;

  if (__DEV__) {
    if (
      currentlyProcessingQueue === sharedQueue &&
      !didWarnUpdateInsideUpdate
    ) {
      console.error(
        'An update (setState, replaceState, or forceUpdate) was scheduled ' +
          'from inside an update function. Update functions should be pure, ' +
          'with zero side-effects. Consider using componentDidUpdate or a ' +
          'callback.',
      );
      didWarnUpdateInsideUpdate = true;
    }
  }
}
```

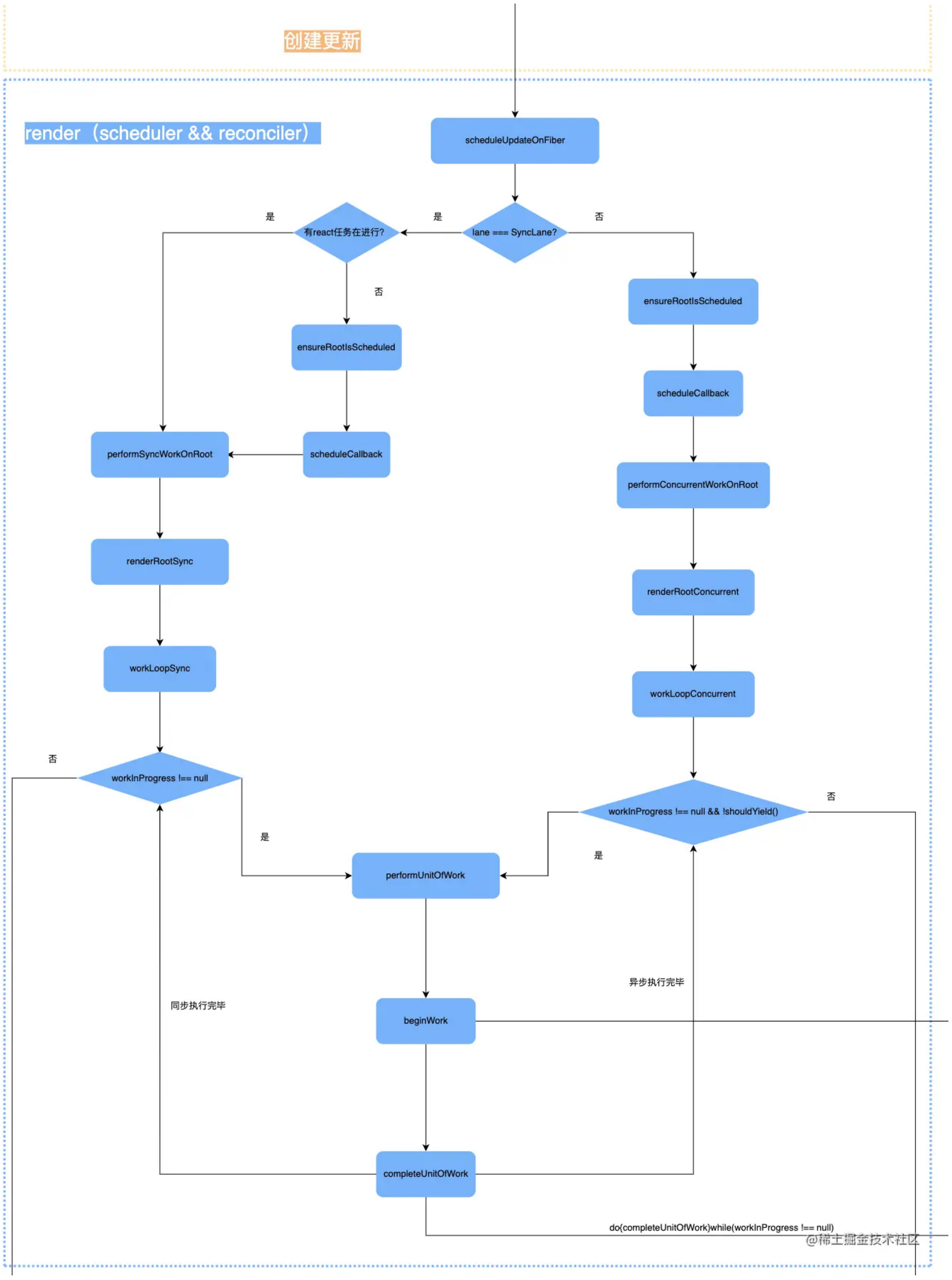
- 这一步就是把需要更新的对象，与 `fiber` 更新队列关联起来。

总结

`React` 通过获取事件的优先级，处理具有同样优先级的事件，创建更新对象并与 `fiber` 的更新队列关联起来。到这一步 `updateContainer` 这个流程就走完了，也下面就是开始他的 `协调阶段` 了。

协调与调度

`协调` 与 `调度` 的流程大致如图所示：



reconciler流程

`React` 的 `reconciler` 流程以 `scheduleUpdateOnFiber` 为入口，并在 `checkForNestedUpdates` 里面处理任务更新的嵌套层数，如果嵌套层数过大（ `>50` ），就会认为是无效更新，则会抛出异

常。之后便根据 `markUpdateLaneFromFiberToRoot` 对当前的 `fiber` 树，自底向上的递归 `fiber` 的 `lane` ，根据 `lane` 做二进制比较或者位运算处理。详情如下：

- 如果当前执行任务的优先级为同步，则去判断有无正在执行的 `React` 任务。如果没有则执行 `ensureRootIsScheduled` ，进行调度处理。
- 如果当前任务优先级是异步执行，则执行 `ensureRootIsScheduled` 进行调度处理。

javascript 复制代码

```javascript
export function scheduleUpdateOnFiber(
  fiber: Fiber,  lane: Lane,  eventTime: number,
) {
  // 检查嵌套层数，避免是循环做无效操作
  checkForNestedUpdates();
  warnAboutRenderPhaseUpdatesInDEV(fiber);

  // 更新当前更新队列里面的任务优先级,自底而上更新child.fiberLanes
  const root = markUpdateLaneFromFiberToRoot(fiber, lane);
  if (root === null) {
    warnAboutUpdateOnUnmountedFiberInDEV(fiber);
    return null;
  }

  // Mark that the root has a pending update.
  // 标记root有更新的，执行它
  markRootUpdated(root, lane, eventTime);

  if (root === workInProgressRoot) {
    // Received an update to a tree that's in the middle of rendering. Mark
    // that there was an interleaved update work on this root. Unless the
    // `deferRenderPhaseUpdateToNextBatch` flag is off and this is a render
    // phase update. In that case, we don't treat render phase updates as if
    // they were interleaved, for backwards compat reasons.
    if (
      deferRenderPhaseUpdateToNextBatch ||
      (executionContext & RenderContext) === NoContext
    ) {
      workInProgressRootUpdatedLanes = mergeLanes(
        workInProgressRootUpdatedLanes,
        lane,
      );
    }
    if (workInProgressRootExitStatus === RootSuspendedWithDelay) {
      // The root already suspended with a delay, which means this render
      // definitely won't finish. Since we have a new update, let's mark it as
      // suspended now, right before marking the incoming update. This has the
      // effect of interrupting the current render and switching to the update.
      // TODO: Make sure this doesn't override pings that happen while we've
      // already started rendering.
      markRootSuspended(root, workInProgressRootRenderLanes);
    }
  }

  // TODO: requestUpdateLanePriority also reads the priority. Pass the
  // priority as an argument to that function and this one.
  // 获取当前优先级层次
  const priorityLevel = getCurrentPriorityLevel();

  // 同步任务，采用同步更新的方式
  if (lane === SyncLane) {
    if (
      // Check if we're inside unbatchedUpdates
      (executionContext & LegacyUnbatchedContext) !== NoContext &&
      // Check if we're not already rendering
      (executionContext & (RenderContext | CommitContext)) === NoContext
    ) {
      // Register pending interactions on the root to avoid losing traced interaction data.
      // 同步而且没有react任务在执行，调用performSyncWorkOnRoot
      schedulePendingInteractions(root, lane);

      // This is a legacy edge case. The initial mount of a ReactDOM.render-ed
      // root inside of batchedUpdates should be synchronous, but layout updates
      // should be deferred until the end of the batch.




      performSyncWorkOnRoot(root);




    } else {
      // 如果有正在执行的react任务，那么执行它ensureRootIsScheduled去复用当前正在执行的任务
      // 跟本次更新一起进行
      ensureRootIsScheduled(root, eventTime);






      schedulePendingInteractions(root, lane);
      if (executionContext === NoContext) {
        // Flush the synchronous work now, unless we're already working or inside
        // a batch. This is intentionally inside scheduleUpdateOnFiber instead of
        // scheduleCallbackForFiber to preserve the ability to schedule a callback
        // without immediately flushing it. We only do this for user-initiated
        // updates, to preserve historical behavior of legacy mode.
        resetRenderTimer();
        flushSyncCallbackQueue();
      }
    }

  } else {
    // Schedule a discrete update but only if it's not Sync.
    // 如果此次是异步任务
    if (
      (executionContext & DiscreteEventContext) !== NoContext &&
      // Only updates at user-blocking priority or greater are considered
      // discrete, even inside a discrete event.
      (priorityLevel === UserBlockingSchedulerPriority ||
        priorityLevel === ImmediateSchedulerPriority)
    ) {
      // This is the result of a discrete event. Track the lowest priority
      // discrete update per root so we can flush them early, if needed.
      if (rootsWithPendingDiscreteUpdates === null) {
        rootsWithPendingDiscreteUpdates = new Set([root]);
      } else {
        rootsWithPendingDiscreteUpdates.add(root);
      }
    }

    // Schedule other updates after in case the callback is sync.
    // 可以中断更新，只要调用ensureRootIsScheduled => performConcurrentWorkOnRoot
    ensureRootIsScheduled(root, eventTime);





    schedulePendingInteractions(root, lane);
  }

  // We use this when assigning a lane for a transition inside
  // `requestUpdateLane`. We assume it's the same as the root being updated,
  // since in the common case of a single root app it probably is. If it's not
  // the same root, then it's not a huge deal, we just might batch more stuff
  // together more than necessary.
  mostRecentlyUpdatedRoot = root;
}
```

同步任务类型执行机制

当任务的类型为同步任务，并且当前的 `js` 主线程空闲，会通过 `performSyncWorkOnRoot(root)` 方法开始执行同步任务。

`performSyncWorkOnRoot` 里面主要做了两件事：

- `renderRootSync` 从根节点开始进行同步渲染任务
- `commitRoot` 执行 `commit` 流程

当前 `js` 线程中有正在执行的任务时候，就会触发 `ensureRootIsScheduled` 函数。 `ensureRootIsScheduled` 里面主要是处理当前加入的更新任务的 `lane` 是否有变化：

- 如果没有变化则表示跟当前的 `schedule` 一起执行。
- 如果有则创建新的 `schedule` 。
- 调用 `performSyncWorkOnRoot` 执行同步任务。

javascript 复制代码

```javascript
function ensureRootIsScheduled(root: FiberRoot, currentTime: number) {
  const existingCallbackNode = root.callbackNode;

  // 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 nextLanes = getNextLanes(
    root,
    root === workInProgressRoot ? workInProgressRootRenderLanes : NoLanes,
  );
  // This returns the priority level computed during the `getNextLanes` call.
  const newCallbackPriority = returnNextLanesPriority();

  if (nextLanes === NoLanes) {
    // Special case: There's nothing to work on.
    if (existingCallbackNode !== null) {
      cancelCallback(existingCallbackNode);
      root.callbackNode = null;
      root.callbackPriority = NoLanePriority;
    }
    return;
  }

  // Check if there's an existing task. We may be able to reuse it.
  if (existingCallbackNode !== null) {
    const existingCallbackPriority = root.callbackPriority;
    if (existingCallbackPriority === newCallbackPriority) {
      // The priority hasn't changed. We can reuse the existing task. Exit.
      return;
    }
    // The priority changed. Cancel the existing callback. We'll schedule a new
    // one below.
```

```
      cancelCallback(existingCallbackNode);
    }

    // Schedule a new callback.
    let newCallbackNode;
    if (newCallbackPriority === SyncLanePriority) {
      // Special case: Sync React callbacks are scheduled on a special
      // internal queue
      // 同步任务调用performSyncWorkOnRoot
      newCallbackNode = scheduleSyncCallback(
        performSyncWorkOnRoot.bind(null, root),
      );
    } else if (newCallbackPriority === SyncBatchedLanePriority) {
      newCallbackNode = scheduleCallback(
        ImmediateSchedulerPriority,
        performSyncWorkOnRoot.bind(null, root),
      );
    } else {
      // 异步任务调用 performConcurrentWorkOnRoot
      const schedulerPriorityLevel = lanePriorityToSchedulerPriority(
        newCallbackPriority,
      );
      newCallbackNode = scheduleCallback(
        schedulerPriorityLevel,
        performConcurrentWorkOnRoot.bind(null, root),
      );
    }

    root.callbackPriority = newCallbackPriority;
    root.callbackNode = newCallbackNode;
  }
```

所以任务类型为同步的时候，不管 `js` 线程空闲与否，都会走到 `performSyncWorkOnRoot` ，进而走 `renderRootSync` 、 `workLoopSync` 流程，而在 `workLoopSync` 中，只要 `workInProgress fiber` 不为 `null` ，则会一直循环执行 `performUnitOfWork` ，而 `performUnitOfWork` 中会去执行 `beginWork` 和 `completeWork` ，也就是上一章里面说的 `beginWork` 流程去创建每一个 `fiber` 节点

javascript 复制代码

```javascript
// packages/react-reconciler/src/ReactFiberWorkLoop.old.js

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
```

异步任务类型执行机制

异步任务则会去执行 `performConcurrentWorkOnRoot` ，进而去执行 `renderRootConcurrent` 、 `workLoopConcurrent` ，但是与同步任务不同的是异步任务是可以中断的，这个可中断的关键字就在于 `shouldYield` ，它本身返回值是一个 `false` ，为 `true` 则可以中断。

javascript 复制代码

```javascript
// packages/react-reconciler/src/ReactFiberWorkLoop.old.js

function workLoopConcurrent() {
  while (workInProgress !== null && !shouldYield()) {
    performUnitOfWork(workInProgress);
  }
}
```

每一次在执行 `performUnitOfWork` 之前都会关注一下 `shouldYield()` 返回值，也就是说的 `reconciler` 过程可中断的意思。

shouldYield

javascript 复制代码

```javascript
// packages\scheduler\src\SchedulerPostTask.js
export function unstable_shouldYield() {
  return getCurrentTime() >= deadline;
}
```

`getCurrentTime` 为 `new Date()` ， `deadline` 为浏览器处理每一帧结束 `时间戳` ，所以这里表示的是，在浏览器每一帧空闲的时候，才会去处理此任务，如果当前任务在浏览器执行的 `某一帧` 里面，则会 `中断当前任务` ，等待浏览器当前帧执行完毕，等到 `下一帧空闲` 的时候，才会去执行当前任务。

所以不管在 `workLoopConcurrent` 还是 `workLoopSync` 中，都会根据当前的 `workInProgress fiber` 是否为 `null` 来进行循环调用 `performUnitOfWork` 。根据流程图以及上面说的这一些，可以看得出来从 `beginWork` 到 `completeUnitOfWork` 这个过程究竟干了什么。

这三章将会讲解 `fiber` 树的 `reconcileChildren` 过程、 `completeWork` 过程、 `commitMutationEffects` .. `insertOrAppendPlacementNodeIntoContainer(DOM)` 过程。这里将详细解读 `v17` 版本的 `React` 的 `diff算法` 、 `虚拟dom到真实dom的创建` ， `函数生命钩子` 的执行流程等。

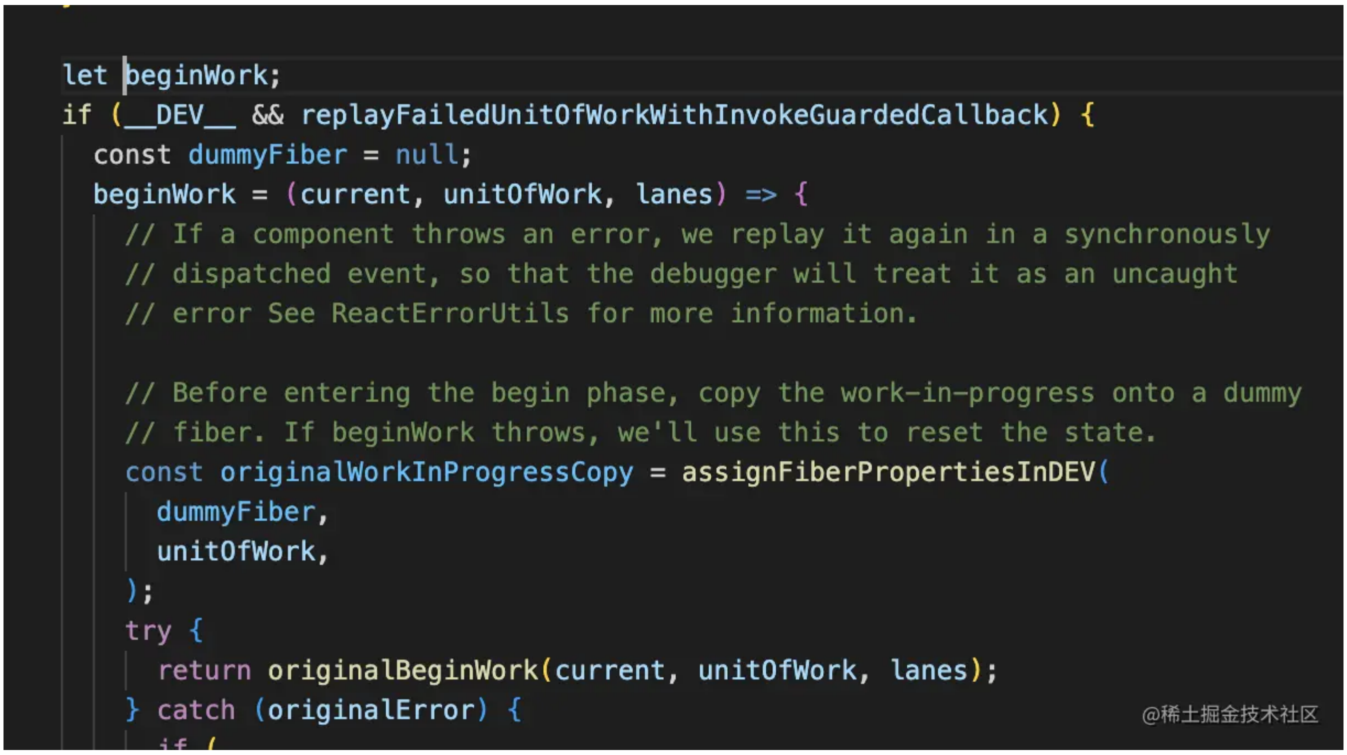
performUnitOfWork

javascript 复制代码

```javascript
function performUnitOfWork(unitOfWork: Fiber): void {
  // The current, flushed, state of this fiber is the alternate. Ideally
  // nothing should rely on this, but relying on it here means that we don't
  // need an additional field on the work in progress.
  const current = unitOfWork.alternate;
  setCurrentDebugFiberInDEV(unitOfWork);

  let next;
  if (enableProfilerTimer && (unitOfWork.mode & ProfileMode) !== NoMode) {
    startProfilerTimer(unitOfWork);
    next = beginWork(current, unitOfWork, subtreeRenderLanes);
    stopProfilerTimerIfRunningAndRecordDelta(unitOfWork, true);
  } else {
    // beginWork
    next = beginWork(current, unitOfWork, subtreeRenderLanes);
  }

  resetCurrentDebugFiberInDEV();
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next === null) {
    // If this doesn't spawn new work, complete the current work.
    // completeUnitOfWork
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }

  ReactCurrentOwner.current = null;
}
```

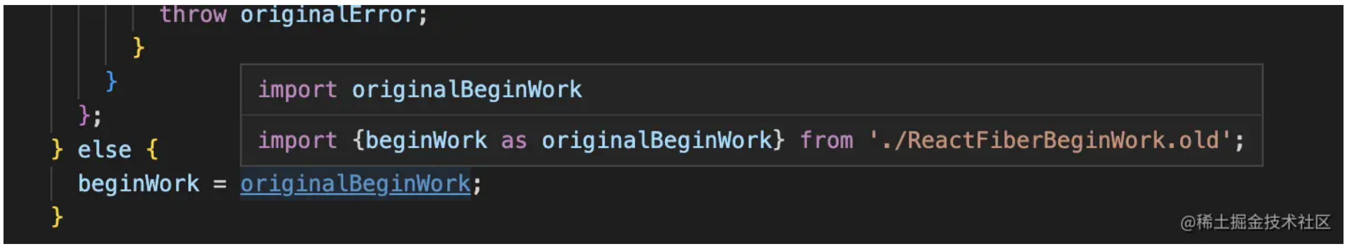
所以在 `performUnitOfWork` 里面，每一次执行 `beginWork` ，进行workIngProgress更新，当遍历完毕整棵fiber树之后便会执行 `completeUnitOfWork` 。

beginWork



```
let beginWork;
if (__DEV__ && replayFailedUnitOfWorkWithInvokeGuardedCallback) {
  const dummyFiber = null;
  beginWork = (current, unitOfWork, lanes) => {
    // If a component throws an error, we replay it again in a synchronously
    // dispatched event, so that the debugger will treat it as an uncaught
    // error See ReactErrorUtils for more information.

    // Before entering the begin phase, copy the work-in-progress onto a dummy
    // fiber. If beginWork throws, we'll use this to reset the state.
    const originalWorkInProgressCopy = assignFiberPropertiesInDEV(
      dummyFiber,
      unitOfWork,
    );
    try {
      return originalBeginWork(current, unitOfWork, lanes);
    } catch (originalError) {
      if (
```

@稀土掘金技术社区



```
        throw originalError;
      }
    }
  };
} else {
  beginWork = originalBeginWork;
}
```

```
import originalBeginWork
import {beginWork as originalBeginWork} from './ReactFiberBeginWork.old';
```

@稀土掘金技术社区

我们可以看到 `beginWork` 就是 `originBeginWork` 得实际执行。我们翻开 `beginWork` 的源码可以看到，它便是根据不同的 `workInProgress.tag` 执行不同组件类型的处理函数，这里就不去拆分的太细，只有有想法便会单独出一篇文章讲述这个的细节，但是最后都会去调用 `reconcileChildren` 。

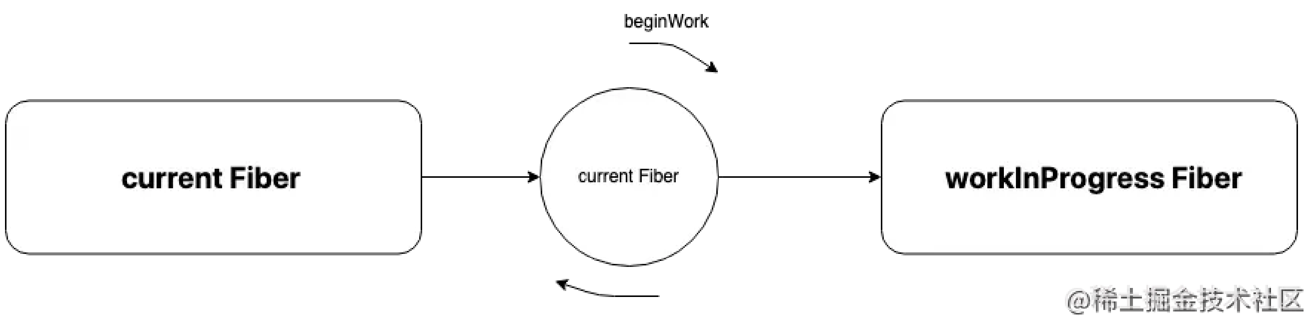
completeUnitOfWork

当遍历完毕执行 `beginWork` ，遍历完毕之后就会走 `completeUnitOfWork` 。

javascript 复制代码

```javascript
function completeUnitOfWork(unitOfWork: Fiber): void {
  // Attempt to complete the current unit of work, then move to the next
  // sibling. If there are no more siblings, return to the parent fiber.
  let completedWork = unitOfWork;
  do {
    // The current, flushed, state of this fiber is the alternate. Ideally
    // nothing should rely on this, but relying on it here means that we don't
    // need an additional field on the work in progress.
    const current = completedWork.alternate;
    const returnFiber = completedWork.return;
    // Check if the work completed or if something threw.
    if ((completedWork.flags & Incomplete) === NoFlags) {
      setCurrentDebugFiberInDEV(completedWork);
      let next;
      if (
        !enableProfilerTimer ||
        (completedWork.mode & ProfileMode) === NoMode
      ) {
        // 绑定事件，更新props，更新dom
        next = completeWork(current, completedWork, subtreeRenderLanes);
      } else {
        startProfilerTimer(completedWork);
        next = completeWork(current, completedWork, subtreeRenderLanes);
        // Update render duration assuming we didn't error.
        stopProfilerTimerIfRunningAndRecordDelta(completedWork, false);
      }
      resetCurrentDebugFiberInDEV();

      if (next !== null) {
        // Completing this fiber spawned new work. Work on that next.
        workInProgress = next;
        return;
      }

      resetChildLanes(completedWork);

      if (
        returnFiber !== null &&
        // Do not append effects to parents if a sibling failed to complete
        (returnFiber.flags & Incomplete) === NoFlags
      ) {
        // Append all the effects of the subtree and this fiber onto the effect
        // list of the parent. The completion order of the children affects the
        // side-effect order.

        // 把已收集到的副作用，合并到父级effect lists中
        if (returnFiber.firstEffect === null) {
          returnFiber.firstEffect = completedWork.firstEffect;
        }
        if (completedWork.lastEffect !== null) {
          if (returnFiber.lastEffect !== null) {
            returnFiber.lastEffect.nextEffect = completedWork.firstEffect;
          }
          returnFiber.lastEffect = completedWork.lastEffect;
        }

        // If this fiber had side-effects, we append it AFTER the children's
        // side-effects. We can perform certain side-effects earlier if needed,
        // by doing multiple passes over the effect list. We don't want to
        // schedule our own side-effect on our own list because if end up
        // reusing children we'll schedule this effect onto itself since we're
        // at the end.
        const flags = completedWork.flags;

        // Skip both NoWork and PerformedWork tags when creating the effect
        // list. PerformedWork effect is read by React DevTools but shouldn't be
        // committed.
        // 跳过NoWork，PerformedWork在commit阶段用不到

        if (flags > PerformedWork) {
          if (returnFiber.lastEffect !== null) {
            returnFiber.lastEffect.nextEffect = completedWork;
          } else {
            returnFiber.firstEffect = completedWork;
          }
          returnFiber.lastEffect = completedWork;
        }
      }
    } else {
      // This fiber did not complete because something threw. Pop values off
      // the stack without entering the complete phase. If this is a boundary,
      // capture values if possible.
      const next = unwindWork(completedWork, subtreeRenderLanes);

      // Because this fiber did not complete, don't reset its expiration time.

      if (next !== null) {
        // If completing this work spawned new work, do that next. We'll come
        // back here again.
        // Since we're restarting, remove anything that is not a host effect
        // from the effect tag.
        next.flags &= HostEffectMask;
        workInProgress = next;
        return;
      }

      if (
        enableProfilerTimer &&
        (completedWork.mode & ProfileMode) !== NoMode
      ) {
        // Record the render duration for the fiber that errored.
        stopProfilerTimerIfRunningAndRecordDelta(completedWork, false);

        // Include the time spent working on failed children before continuing.
        let actualDuration = completedWork.actualDuration;
        let child = completedWork.child;
        while (child !== null) {
          actualDuration += child.actualDuration;
          child = child.sibling;
        }
        completedWork.actualDuration = actualDuration;
      }

      if (returnFiber !== null) {
        // Mark the parent fiber as incomplete and clear its effect list.
        returnFiber.firstEffect = returnFiber.lastEffect = null;
        returnFiber.flags |= Incomplete;
      }
    }

    // 兄弟层指针
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      // If there is more work to do in this returnFiber, do that next.
      workInProgress = siblingFiber;
      return;
    }
    // Otherwise, return to the parent
    completedWork = returnFiber;
    // Update the next thing we're working on in case something throws.
    workInProgress = completedWork;
  } while (completedWork !== null);

  // We've reached the root.
  if (workInProgressRootExitStatus === RootIncomplete) {
    workInProgressRootExitStatus = RootCompleted;
  }
}
```

他的作用便是逐层收集 `fiber` 树上已经被打上的副作用标签 `flags` ，一直收集到 `root` 上面以便于在 `commit` 阶段进行 `dom` 的 `增删改` 。



scheduler流程

在这里应该有很多人不明白， `协调` 和 `调度` 是什么意思，通俗来讲：

- 协调就是协同合作
- 调度就是执行命令

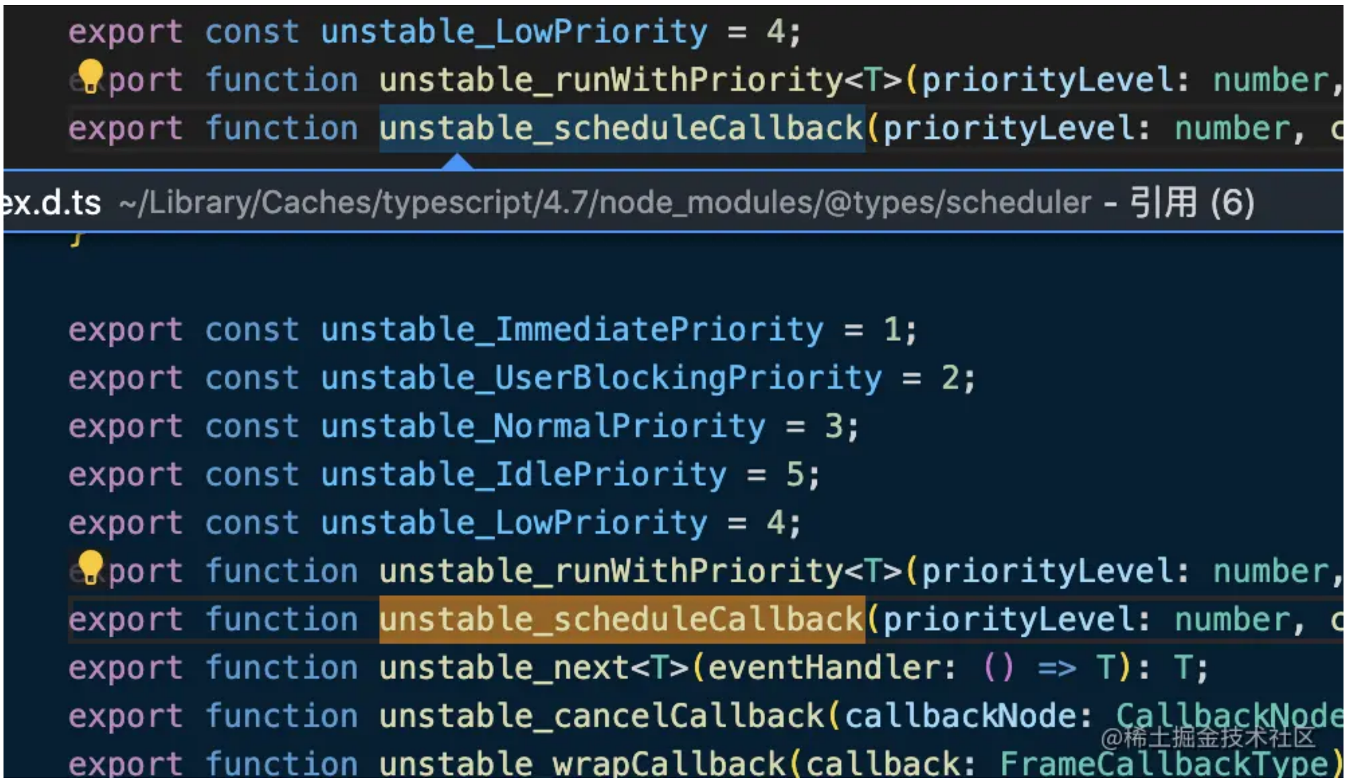
所以在 `React` 中协调就是一个 `js` 线程中，需要安排很多模块去完成整个流程，例如：同步异步 `lane` 的处理， `reconcileChildren` 处理 `fiber` 节点等，保证整个流程有条不紊的执行。调度表现为让 `空闲的js线程` （帧层面）去执行其他任务，这个过程称之为调度，那么它到底是怎么去做的呢？ 我们回到处理异步任务那里，我们会发现 `performConcurrentWorkOnRoot` 这个函数外面包裹了一层 `scheduleCallback` ：

javascript 复制代码

```javascript
newCallbackNode = scheduleCallback(
   schedulerPriorityLevel,
   performConcurrentWorkOnRoot.bind(null, root),
)
```

javascript 复制代码

```javascript
export function scheduleCallback(
  reactPriorityLevel: ReactPriorityLevel,  callback: SchedulerCallback,  options: SchedulerCallbackO
) {
  const priorityLevel = reactPriorityToSchedulerPriority(reactPriorityLevel);
  return Scheduler_scheduleCallback(priorityLevel, callback, options);
}
```



```
export const unstable_LowPriority = 4;
export function unstable_runWithPriority<T>(priorityLevel: number,
export function unstable_scheduleCallback(priorityLevel: number, c
```

ex.d.ts ~/Library/Caches/typescript/4.7/node_modules/@types/scheduler - 引用 (6)

```
export const unstable_ImmediatePriority = 1;
export const unstable_UserBlockingPriority = 2;
export const unstable_NormalPriority = 3;
export const unstable_IdlePriority = 5;
export const unstable_LowPriority = 4;
export function unstable_runWithPriority<T>(priorityLevel: number,
export function unstable_scheduleCallback(priorityLevel: number, c
export function unstable_next<T>(eventHandler: () => T): T;
export function unstable_cancelCallback(callbackNode: CallbackNode
export function unstable_wrapCallback(callback: FrameCallbackType)
```

我们几经周折找到了声明函数的地方

javascript 复制代码

```javascript
// packages/scheduler/src/Scheduler.js
function unstable_scheduleCallback(priorityLevel, callback, options) {
  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;
  }

  var timeout;
  switch (priorityLevel) {
    case ImmediatePriority:
      timeout = IMMEDIATE_PRIORITY_TIMEOUT;
      break;
    case UserBlockingPriority:
      timeout = USER_BLOCKING_PRIORITY_TIMEOUT;
      break;
    case IdlePriority:
      timeout = IDLE_PRIORITY_TIMEOUT;
      break;
    case LowPriority:
      timeout = LOW_PRIORITY_TIMEOUT;
      break;
    case NormalPriority:
    default:
      timeout = NORMAL_PRIORITY_TIMEOUT;
      break;
  }

  var expirationTime = startTime + timeout;

  var newTask = {
    id: taskIdCounter++,
    callback,
    priorityLevel,
    startTime,
    expirationTime,
    sortIndex: -1,
  };
  if (enableProfiling) {
    newTask.isQueued = false;
  }
  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 delay.
      if (isHostTimeoutScheduled) {
        // Cancel an existing timeout.
        cancelHostTimeout();
      } else {
        isHostTimeoutScheduled = true;
      }
      // Schedule a timeout.
      requestHostTimeout(handleTimeout, startTime - currentTime);
    }
  } else {
    newTask.sortIndex = expirationTime;
    push(taskQueue, newTask);
    if (enableProfiling) {
      markTaskStart(newTask, currentTime);
      newTask.isQueued = true;
    }
    // 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);
    }
  }

  return newTask;
}
```

- 当 `starttime > currentTime` 的时候，表示任务超时，插入超时队列。
- 任务没有超时，插入调度队列
- 执行 `requestHostCallback` 调度任务。

javascript 复制代码

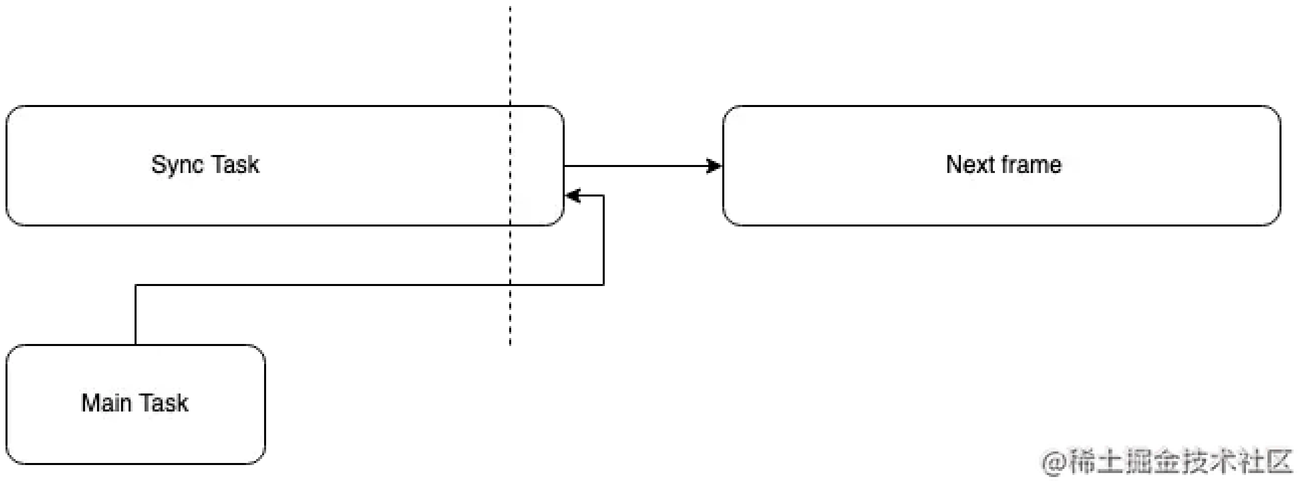
```javascript
// 创建消息通道
const channel = new MessageChannel();
const port = channel.port2;
channel.port1.onmessage = performWorkUntilDeadline;

// 告知scheduler开始调度
requestHostCallback = function(callback) {
  scheduledHostCallback = callback;
  if (!isMessageLoopRunning) {
    isMessageLoopRunning = true;
    port.postMessage(null);
  }
};
```

`react` 通过 `new MessageChannel()` 创建了消息通道，当发现 `js` 线程空闲时，通过 `postMessage` 通知 `scheduler` 开始调度。 `performWorkUntilDeadline` 函数功能为处理 `react` 调度开始时间更新到结束时间。 这里我们要关注一下设备帧速率。

```javascript
forceFrameRate = function(fps) {
  if (fps < 0 || fps > 125) {
    // Using console['error'] to evade Babel and ESLint
    console['error'](
      'forceFrameRate takes a positive int between 0 and 125, ' +
        'forcing frame rates higher than 125 fps is not supported',
    );
    return;
  }
  if (fps > 0) {
    yieldInterval = Math.floor(1000 / fps);
  } else {
    // reset the framerate
    yieldInterval = 5;
  }
};
```

performWorkUntilDeadline

```javascript
const performWorkUntilDeadline = () => {
  if (scheduledHostCallback !== null) {
    const currentTime = getCurrentTime();
    // Yield after `yieldInterval` ms, regardless of where we are in the vsync
    // cycle. This means there's always time remaining at the beginning of
    // the message event.
    // 更新当前帧结束时间
    deadline = currentTime + yieldInterval;
    const hasTimeRemaining = true;
    try {
      const hasMoreWork = scheduledHostCallback(
        hasTimeRemaining,
        currentTime,
      );
      // 还有任务就继续执行
      if (!hasMoreWork) {
        isMessageLoopRunning = false;
        scheduledHostCallback = null;
      } else {
        // If there's more work, schedule the next message event at the end
        // of the preceding one.
        // 没有就postMessage
        port.postMessage(null);
      }
    } catch (error) {
      // If a scheduler task throws, exit the current browser task so the
      // error can be observed.
      port.postMessage(null);
      throw error;
    }
  } else {
    isMessageLoopRunning = false;
  }
  // Yielding to the browser will give it a chance to paint, so we can
  // reset this.
  needsPaint = false;
};
```



总结

本文讲了 React 在状态改变的时候，会根据当前任务优先级，等一些列操作去创建 workInProgress fiber 链表树，在协调阶段，会根据浏览器每一帧去做比较，假如浏览器 每一帧 执行时间戳高于当前时间，则表示当前帧没有空闲时间，当前任务则必须要等到 下一个空闲帧 才能去执行的 可中断 的策略。还有关于 beginWork 的遍历执行更新 fiber 的节点。那么到这里这一章就讲述完毕了，下一章讲一讲React的diff算法