React 中是如何实现状态更新调度的?

前面我们讲了 `render` 阶段和 `commit` 阶段的工作，讲了 `render` 阶段中 Diff 算法的实现，现在我们已经对整体的流程有了一定的认识

- 在 `render` 阶段的 `beginWork` 阶段会创建子 `Fiber` 节点，通过 Diff 算法，打上相应的 `effectTag`
- 在 `render` 阶段的 `completeWork` 阶段会根据 `Fiber` 节点生成对应的 `DOM` 节点，并连接子节点
- 在 `commit` 阶段会对所有的 `effectTag` 执行相应的 DOM 操作，更新视图

在这些流程之前还有着，从**触发状态更新到 `render` 阶段**的过程，也就是我们本章需要学习的内容：**状态更新**

首先我们先了解一下什么是**最小更新单元**

最小更新单元

我们从实例来看，对于一个组件来说，如果想要触发更新，那么可以有以下这些情况

- 组件本身的 `state` 发生改变
- 组件 `props` 的改变，也就是父组件状态导致子组件更新
- `context` 改变，该组件消费的 `context` 发生了更新

但是归根结底，无论是哪种场景下触发的更新，最终的本质都是 `state` 的变化导致的
对于 React 来说，能够触发 state 更新的基本都是**在组件层面**上，毕竟 Fiber 没有办法实现自我更新，只能依赖组件进行 `state` 更想你
因此，我们可以认为 **最小的更新单元是组件，更新源自于 state 的变化**

那么如何触发更新呢?

触发更新

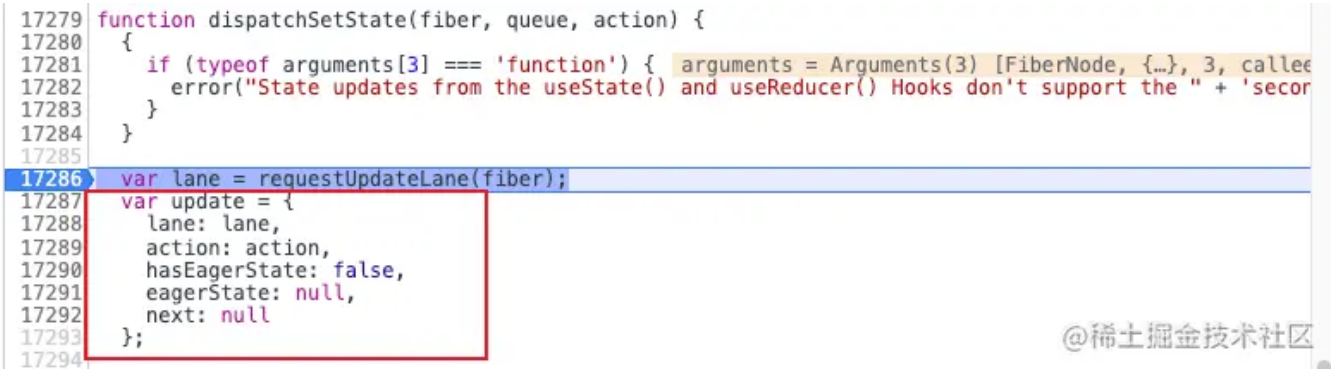
竟然组件是最小的更新单元，那么我们可以知道，触发更新可以分为

- 类组件的 `setState` 状态更新
- 函数组件的 `useState` 状态更新

那么，状态更新都会触发哪些流程呢?

创建 Update 对象

首先，每次状态更新都会创建一个保存更新状态内容的对象，也就是 Update 对象，保存在 updateQueue 链表中，记录当前 Fiber 节点收集到的更新。在 `render` 阶段的 `beginWork` 中会根据 `Update` 对象来计算新的 `state`



这个 Update 会被保存在一个环状链表中，接下来会调用 `scheduleUpdateOnFiber` 方法，来调度这个 Update

标记 RootFiber

我们知道 `render` 阶段是从 `rootFiber` 开始向下遍历，因此我们需要在 `render` 阶段开始之前，让 `rootFiber` 知道本次调度的相关信息。
因此我们需要从当前触发更新的节点对应的 Fiber **一直遍历到应用的根节点 rootFiber**，并通知沿途的 Fiber，你有子孙节点被更新了，打上**本次触发更新的优先级标记**，并返回 rootFiber 这一步叫做 `markUpdateLaneFromFiberToRoot`

调度更新

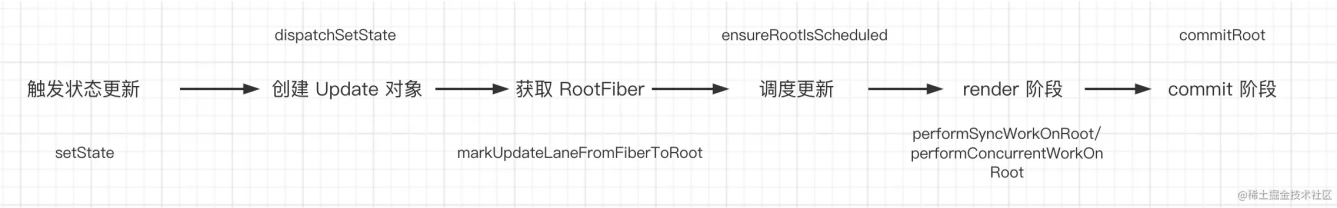
经过了上面的处理，我们已经获取到了 `RootFiber`，并且在这个 Fiber 上同时标记了子树中的 Update
接下来，需要通知 Scheduler 调度器来根据更新的优先级，以及任务的类型，来发起异步或是同步的更新调度，这一步调用的是 `ensureRootIsScheduled` 方法

- 对于 legacy 模式下，最后执行的是 `performSyncWorkOnRoot` 方法
- 对于 concurrent 模式，最后执行的是 `performConcurrentWorkOnRoot` 方法

总结

当调度的回调函数被执行，会进入组件的 `render` 阶段

- 在 `render` 阶段的 `reconcile` 也就是 diff 算法中，会根据 `Update` 对象，返回对应的 `state`，根据 `state` 判断本次是否需要更新视图，如果需要更新视图就会被标记为 `effectTag`
- 在 `commit` 阶段，标记了 `effectTag` 的 `Fiber`，就会执行对应的视图更新



优先级更新

在 React 18 的更新中，全面启用了 `concurrent` 模式，使用 `legacy` 模式将会报 **warning 警告**，可以看出 `concurrent` 模式会是 React 的未来。

`legacy` 模式是我们之前常用的，它**构建 dom 的过程是同步的**，所以在 `reconcile` 阶段的 Diff 中，如果特别耗时，那么导致的结果就是 js 会一直**阻塞高优先级的任务**，表现为页面的卡顿和无法响应。

`concurrent` 模式是 react 18 中全面开启的模式，它用**时间片调度实现了异步可中断**的任务，根据设备性能的不同，时间片的长度也不一样，在每个时间片中，如果任务到了过期时间，就会主动让出线程给高优先级的任务。

采用 `ReactDOM.render` 来渲染的应用，就是 `legacy` 模式，都是同步的，在状态更新时没有优先级的概念，任务之间需要依次执行。

对于 `concurrent` 模式和 `blocking` 模式来说，也就是采用 `ReactDOM.createRoot` 或 `ReactDOM.createBlockingRoot` 创建的应用，会采用并发的方式来更新状态。**当有高优先级任务存在时，会中断当前正在执行的低优先级任务，先完成高优先级更新后，再基于更新结果重新进行低优先级的更新。**

> blocking mode 是实验中的模式，为了 legacy 迁移至 concurrent 而存在

那么什么是优先级呢?

优先级与Update

> 优先级的概念只存在与 concurrent 模式中

React 源码中优先级的分类，一共有以下 6 种优先级

```javascript
export type PriorityLevel = 0 | 1 | 2 | 3 | 4 | 5;

// TODO: Use symbols?
export const NoPriority = 0;
export const ImmediatePriority = 1; // 最高
export const UserBlockingPriority = 2; // 用户触发的更新，onClick 等
export const NormalPriority = 3; // 一般的优先级，请求数据更新状态
export const LowPriority = 4; //
export const IdlePriority = 5; // 空闲优先级
```

优先级的计算公式，只执行**高于本次更新**的优先级的 Update
baseState + Update1（NormalPriority） + Update2（UserBlockingPriority） = newState
对于上面的公式来说，`Update2` 的优先级高于 `Update1`，那么会先执行 `Update2` 的更新，再基于 `Update2` 更新的结果进行 `Update1` 的更新，也就是下面两步

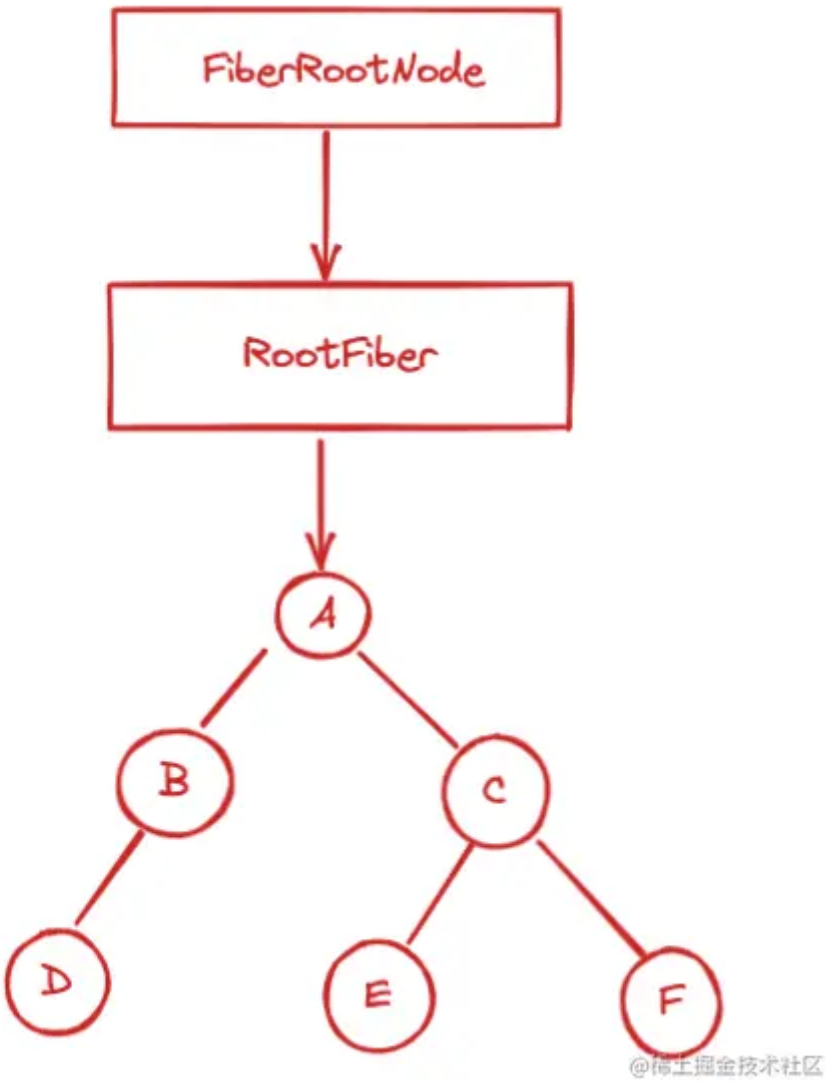
1. `baseState` + `Update2` = `newState1`
2. `newState1` + `Update1` = `newState`

在知道了不同事件触发更新的优先级之后，我们再来看看它的更新流程

更新流程

在上面我们知道了不同优先级的更新会有中断的可能，那么具体流程是怎么样的呢，我们通过下面这个例子来了解一下
下图是一个组件树的结构，F 组件触发了一次更新，它的优先级是 **NormalPriority**。

可以假设是在 `componentDidMount` 中去请求了一次数据，在请求成功后调用了 `setState` 去更新状态，这里调用 `setState` 就会创建一次更新，因此这个更新的优先级是 **NormalPriority**



- 首先它会从当前的 Fiber 节点，也就是 F 节点，开始向上遍历，并通知沿途的 Fiber 节点有更新，一直到 `FiberRootNode`，在 `FiberRootNode` 上保存当前更新的优先级，在这里就是 **NormalPriority**
- 接下来，就会以 **NormalPriority** 优先级，来调度整个应用的根节点 `FiberRootNode`，整个应用中只有一个被调度的任务，它的优先级是 **NormalPriority**，于是就调用 **NormalPriority** 的回调函数，这个回调函数就是 `render` 阶段的入口，由于优先级是作用在**整个组件树**的，我们会从 `FiberRootNode` 开始向下，采用深度优先遍历的方式，依次以 **NormalPriority** 来执行每一个组件的 Diff
- 在这个组件树中，只有 F 组件存在 **NormalPriority** 优先级对应的 Update，因此只有 F 组件会在 Diff 中得到一个 state
- 如果在这个计算过程中，F 组件**又触发了一次更新**，这个更新的优先级是 `UserBlockingPriority`，那么又会从 F 组件向上遍历，直到 `FiberRootNode`，mark 一下

- 注册一个 `UserBlockingPriority` 的调度，接下来 scheduler 就会调度，**NormalPriority** 和 `UserBlockingPriority`，`UserBlockingPriority` 的优先级高于 **NormalPriority**
- 因此之前正在执行的 **NormalPriority** 的 `render` 阶段就会被中断，重新从根节点向下深度优先遍历，执行 `UserBlockingPriority` 优先级的更新
- 在执行完 `render --> commit` 阶段后，基于当前的计算结果，再去执行刚刚被中断的低先级的更新

例子

> 例子学习自：React 技术揭秘

在下面的例子中，通过 `useEffect` 触发了两个更新，一个是 `useState` 的回调更新，一个是**事件触发的更新**，`onClick` 触发的更新优先级比 `normal` 要高，又因为我们两次更新时间间隔很短，并且操作很多，第一次更新还没有在 20ms 内完成就触发了优先级更高的更新，因此会调度 `normal` 和 `userBlocking` 两个优先级的事件，`userBlocking` 优先级的更新，会中断正在执行的 `normal` 的 render 阶段
因此页面会从 0 变为 2 再变为 3

```javascript
const [count, updateCount] = useState(0)
const buttonRef = useRef(null)

const onClick = () => {
  updateCount(count => count + 2)
}

useEffect(() => {
  const button = buttonRef.current
  setTimeout(() => updateCount(1), 1000)
  setTimeout(() => button.click(), 1020)
},[])

return (
  <div className="App">
    <button ref={buttonRef} onClick={onClick}>
      增加 2
    </button>
    <div>
      {
        Array.from(new Array(4000)).map((v, index) => (
          <span key={index}>{count}</span>
```

```
        ))
      }
    </div>
  </div>
);
```

以上的讨论都是基于 `concurrent mode` 下进行的，如果是在同步模式下，也就是 ReactDOM.render ，页面的结果展示将会是从 0 变为 1 变为 3，这是因为同步模式下没有优先级的概念，不会中断第一次的 `updateCount`

状态更新调度源码解析

在前面几节，我们介绍了 React 中，状态更新的主要流程，以及 `concurrent` 模式下，优先级的概念以及优先级更新的流程，本节我们将**从源码的角度来解析 React 是如何实现状态更新**的。
无论是 `setState` 还是 `useState` 进行的更新，**都会**创建更新任务，也就是创建 Update 对象，并添加到 Fiber 的 `UpdateQueue` 中，如果是 Function Component 会添加到 `baseQueue` 中。
接下来就会进入核心的 `reconciler` 阶段，主要分为 4 个子阶段

1. 任务输入：触发的更新都会 dispatch 到 `scheduleUpdateOnFiber` 这个函数中，来处理更新任务
2. 调度任务：通过 `Scheduler` 来调度任务，等待空闲时间回调
3. 执行任务会调：构造 Fiber 树， `render` 阶段的 `completeWork` 阶段会创建 Fiber 对应的 DOM 节点
4. 输出 DOM 节点： `commit` 阶段会与渲染器交互，渲染出 DOM 节点

对于**不同形式触发**的状态更新来说，它们都会进入**一套相同的 render 到 commit 的流程**，这是因为在每次更新时都会创建一个保存**更新状态相关内容**的对象 **Update**。在 render 阶段的 **beginWork** 中会根据 Update 来计算 `newState`
在初始化阶段完成之后，如果触发了 state 的更新，那么会发生什么呢?

触发更新

以 `useState` 和 `setState` 来分别看函数组件和类组件的更新流程

类组件

触发 `setState` 本质上是调用了 `enqueueSetState`

```javascript 复制代码
enqueueSetState(inst,payload,callback){
    const update = createUpdate(eventTime, lane);
    ...
    enqueueUpdate(fiber, update, lane);
    ...
    const root = scheduleUpdateOnFiber(fiber, lane, eventTime);
}
```

函数组件

对于函数组件而言，会调用 `dispatchSetState`

```javascript 复制代码
function dispatchSetState(fiber, queue, action) {
    const lane = requestUpdateLane(fiber);
    const update: Update<S, A> = {
      lane,
      action,
      hasEagerState: false,
      eagerState: null,
      next: (null: any),
    };
    ...
    scheduleUpdateOnFiber(fiber, lane, eventTime);
}
```

无论是通过什么方式来触发，都会**创建一个 Update 对象**，这样验证了我们之前一直所说的，然后它会被**保存到环状链表 pending 中**，最后都是会**调用** `scheduleUpdateOnFiber` 方法，这个也就是整个更新的入口，接下来我们看看它都做了些什么

更新入口 scheduleUpdateOnFiber

从前面我们知道了，无论是什么方式触发的更新，最后都会调用 `scheduleUpdateOnFiber` 方法，这也是任务调度的入口，它的核心流程如下

1. 首先会检查当前的更新是否存在嵌套更新
2. 递归向上通知沿途的父节点，子节点存在某种优先级的更新

3. 标记 RootFiber 有待处理的更新，为 render 阶段做准备
4. 开始可中断更新

```javascript
// react-reconciler/src/ReactFiberWorkLoop.new.js
export function scheduleUpdateOnFiber(
  fiber: Fiber,
  lane: Lane,
  eventTime: number,
): FiberRoot | null {
  // 检查是否有死循环
  checkForNestedUpdates();
  ...
  // 自底向上标记更新优先级
  const root = markUpdateLaneFromFiberToRoot(fiber, lane);
  ...
  // 标记 root 有更新，lane 插入到 root.pendingLanes 中
  markRootUpdated(root, lane, eventTime);
  ...
  if (root === workInProgressRoot) {
    // 在渲染过程中接收到一个更新，在根节点上标记一个交错更新，
    if (
      deferRenderPhaseUpdateToNextBatch ||
      (executionContext & RenderContext) === NoContext
    ) {
      workInProgressRootUpdatedLanes = mergeLanes(
        workInProgressRootUpdatedLanes,
        lane,
      );
    }
    if (workInProgressRootExitStatus === RootSuspendedWithDelay) {
      // 执行高优先级更新
      markRootSuspended(root, workInProgressRootRenderLanes);
    }
  }

  // 执行可中断更新
  ensureRootIsScheduled(root, eventTime);
  ...
  return root;
}
```

1.检查是否死循环 checkForNestedUpdates

在 `scheduleUpdateOnFiber` 函数中，首先会调用 `checkForNestedUpdates` 方法，检查是否有嵌套更新，也可以说是循环更新，无限调用，这种情况会抛出异常

这里的 `NESTED_UPDATE_LIMIT` 的值是 50，也就是说当循环次数超过 50 次时，会认为是死循环，会抛出错误

```javascript 复制代码
function checkForNestedUpdates() {

  if (nestedUpdateCount > NESTED_UPDATE_LIMIT) {
    nestedUpdateCount = 0;
    rootWithNestedUpdates = null;
    throw new Error(
      'Maximum update depth exceeded. This can happen when a component ' +
        'repeatedly calls setState inside componentWillUpdate or ' +
        'componentDidUpdate. React limits the number of nested updates to ' +
        'prevent infinite loops.',
    );
  }
}
```

2.递归向上通知 parent 有更新 markUpdateLaneFromFiberToRoot

接下来，会调用 `markUpdateLaneFromFiberToRoot` 方法，更新当前 Fiber 节点的 `lanes` 字段，并向上归并在父节点的 `childLanes` 字段中添加为本次更新的优先级 `lanes`。最后返回当前的 rootFiber 节点

注意这里会对 Fiber 节点的 alternate Fiber 的 lane 进行更新，这个非常重要，下一节会讲到

```javascript 复制代码
function markUpdateLaneFromFiberToRoot(
  sourceFiber: Fiber,
  lane: Lane,
): FiberRoot | null {
  // 更新当前 Fiber 的优先级
  sourceFiber.lanes = mergeLanes(sourceFiber.lanes, lane);
  let alternate = sourceFiber.alternate;
  if (alternate !== null) {
    alternate.lanes = mergeLanes(alternate.lanes, lane);
  }

  let node = sourceFiber;
  let parent = sourceFiber.return;
  // 归并更新 父节点 的优先级
  while (parent !== null) {
    parent.childLanes = mergeLanes(parent.childLanes, lane);
    alternate = parent.alternate;
    if (alternate !== null) {
      alternate.childLanes = mergeLanes(alternate.childLanes, lane);
    }
```

```
    node = parent;
    parent = parent.return;
  }
  if (node.tag === HostRoot) {
    const root: FiberRoot = node.stateNode;
    return root;
  } else {
    return null;
  }
}
```

这里采用 `mergeLanes` 来合并优先级，因为可能会有多个更新存在，都需要在后续进行调度，lanes 更新的操作很简单，只需要将当前的优先级 `lane` 与之前的 `lane` 进行**二进制或运算**即可

```javascript 复制代码
export function mergeLanes(a: Lanes | Lane, b: Lanes | Lane): Lanes {
  return a | b;
}
```

3. 标记 RootFiber 有待处理更新

接下来会将更新的 lane 通过二进制运算添加到 root Fiber 的 `pendingLanes` 中，在 root Fiber 标记一个更新

```javascript 复制代码
export function markRootUpdated(
  root: FiberRoot,
  updateLane: Lane,
  eventTime: number,
) {
  // 设置更新的优先级
  root.pendingLanes |= updateLane;

  if (updateLane !== IdleLane) {
    root.suspendedLanes = NoLanes;
    root.pingedLanes = NoLanes;
  }
  const eventTimes = root.eventTimes;
  const index = laneToIndex(updateLane);
  eventTimes[index] = eventTime;
}
```

4. 开始可中断更新 ensureRootIsScheduled

在函数的最后，会调用 `ensureRootIsScheduled` 方法，传入已经被标记过的 root Fiber 节点以及创建 update 的时间，开始可中断更新

```javascript 复制代码
ensureRootIsScheduled(root, eventTime);
```

接下来详细看看 `ensureRootIsScheduled` 方法做了什么

注册调度任务

`ensureRootIsScheduled` 函数的作用是为 `root` 安排调度任务，每个更新任务的 `update` 都会经过 `ensureRootIsScheduled` 的处理，它主要会做以下几件事：

- 首先会计算**最新的调度更新优先级** `newCallbackPriority` ，判断是否和 `rootFiber` 上的 `calllbackPriority` 优先级是否相等，如果相等，则会因为**优先级没有改变，重用当前任务**，直接退出
- 如果不相等，会进入真正的调度任务函数 `sheduleSyncCallback` 函数中
- 最后会将 `newCallbackPriority` 赋值给 `callbackPriority`

核心代码如下

```javascript 复制代码
// 注册任务
function ensureRootIsScheduled(root: FiberRoot, currentTime: number) {
  ...
  // 注册的新任务
  let newCallbackNode;
  // 如果新渲染任务的优先级是同步优先级
  // if 逻辑处理的是同步任务，同步任务不需经过 Scheduler
  if (newCallbackPriority === SyncLane) {
    // 同步任务不经过 Scheduler
    if (root.tag === LegacyRoot) {
      // Legacy 模式
      scheduleLegacySyncCallback(performSyncWorkOnRoot.bind(null, root));
    } else {
      // 非 legacy模式
      scheduleSyncCallback(performSyncWorkOnRoot.bind(null, root));
    }
    // React18 新增加的
    if (supportsMicrotasks) {
        scheduleMicrotask(() => {
```

```
          if (executionContext === NoContext) {
            flushSyncCallbacks();
          }
        });
    } else {
      scheduleCallback(ImmediateSchedulerPriority, flushSyncCallbacks);
    }
    ...
  } else {
    ...
    // 调度优先级任务
    newCallbackNode = scheduleCallback(
      schedulerPriorityLevel,
      performConcurrentWorkOnRoot.bind(null, root),
    );
  }
  ...
  root.callbackPriority = newCallbackPriority;
  root.callbackNode = newCallbackNode;
}
```

如果任务是**同步任务**，就不需要 Scheduler 调度，直接通过 `scheduleSyncCallback` 和 `scheduleLegacySyncCallback` 处理，当 JS 主线程空闲的时候，则执行 `performSyncWorkOnRoot` 函数来执行同步任务
如果任务是**并发任务**，则需要经过 Scheduler 调度，会通过 `scheduleCallback` 回调函数注册调度任务。

开始调度任务

分为同步和并发两种情况来讨论

同步情况

当同步状态下触发多次 `useState` 的时候，会执行以下步骤

- 首先第一次进入到 `ensureRootIsScheduled` ，计算出新的更新任务的优先级 `newCallbackPriority`，和之前的 `callbackPriority` 进行对比，如果相等那就直接退出
- 同步状态下更新的优先级 `newCallbackPriority` 是等于 `SyncLane` 的，那么会执行两个函数， `scheduleSyncCallback` 和 `scheduleMicrotask` 。

最终都会进入 `scheduleSyncCallback` 的逻辑，这个方法非常简单，就是将任务放入 `syncQueue` 队列当中

```javascript 复制代码
export function scheduleSyncCallback(callback: SchedulerCallback) {
  // 将任务放入队列中
  if (syncQueue === null) {
    syncQueue = [callback];
  } else {
    syncQueue.push(callback);
  }
}
```

接着会在下面的流程中通过 `scheduleMicrotask` 来执行 `flushSyncCallbacks` 方法，这个方法会立即执行更新队列，发起更新任务，**目的就是让任务不延时到下一帧**。同步情况也需要调度是为了保证更新的连续性，一个一个任务依次执行。

```javascript 复制代码
 scheduleMicrotask(() => {
   if (executionContext === NoContext) {
     flushSyncCallbacks();
   }
 });
```

`scheduleMicrotask` 是一个 `ployfill` 实现，本质上就是一个 `Promise.resolve` 以及**不兼容**情况下使用的 `setTimeout`

```javascript 复制代码
export const scheduleMicrotask: any =
  typeof queueMicrotask === 'function'
    ? queueMicrotask
    : typeof localPromise !== 'undefined'
    ? callback =>
        localPromise
          .resolve(null)
          .then(callback)
          .catch(handleErrorInNextTick)
    : scheduleTimeout; // TODO: Determine the best fallback here.
```

异步情况

上面是在同步情况下的更新逻辑，有时候更新是在 `setTimeout` 等方法中触发的，那么他们会进入下面这些逻辑

- 首先会判断 `existingCallbackPriority === newCallbackPriority` 是否相等，**来尝试复用它**
- 接下来会执行 `scheduleCallback` ，得到最新的 `newCallbackNode` ，赋值给 `root`

`scheduleCallback` 会调用 `Scheduler_scheduleCallback` 方法，具体来看看这个方法的实现

```javascript 复制代码
function scheduleCallback(priorityLevel, callback) {
    return Scheduler_scheduleCallback(priorityLevel, callback);
}
```

`Scheduler_scheduleCallback` 方法最终是由 `unstable_scheduleCallback` 方法导入的，这个方法在 `scheduler/src/forks/scheduler.js` 目录下，比较难找
通过调用 `unstable_scheduleCallback` 方法创建调度任务，然后**根据任务是否超时**，将任务插入到超时队列 `timerQueue` 和调度任务队列 `taskQueue`
将任务插入调度任务队列 `taskQueue` 之后，会通过 `requestHostCallback` 函数去调度任务。

核心流程如下

1. 通过 `startTime` 和 `currentTime` 比较，来**判断任务是否过期**，过期存入 `taskQueue` ，未过期存入 `timerQueue`
2. 如果**有过期任务存在，并且没有正在调度的任务**，那么通过 requestHostCallback 来调度
3. 如果**没有过期任务**，通过 `requestHostTimeout` 来延时执行

```javascript 复制代码
function unstable_scheduleCallback(priorityLevel, callback, options) {
  // 获取当前时间戳
  var currentTime = getCurrentTime();
  var startTime;
   ...
// 根据调度优先级设置相应的超时时间
  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;
```

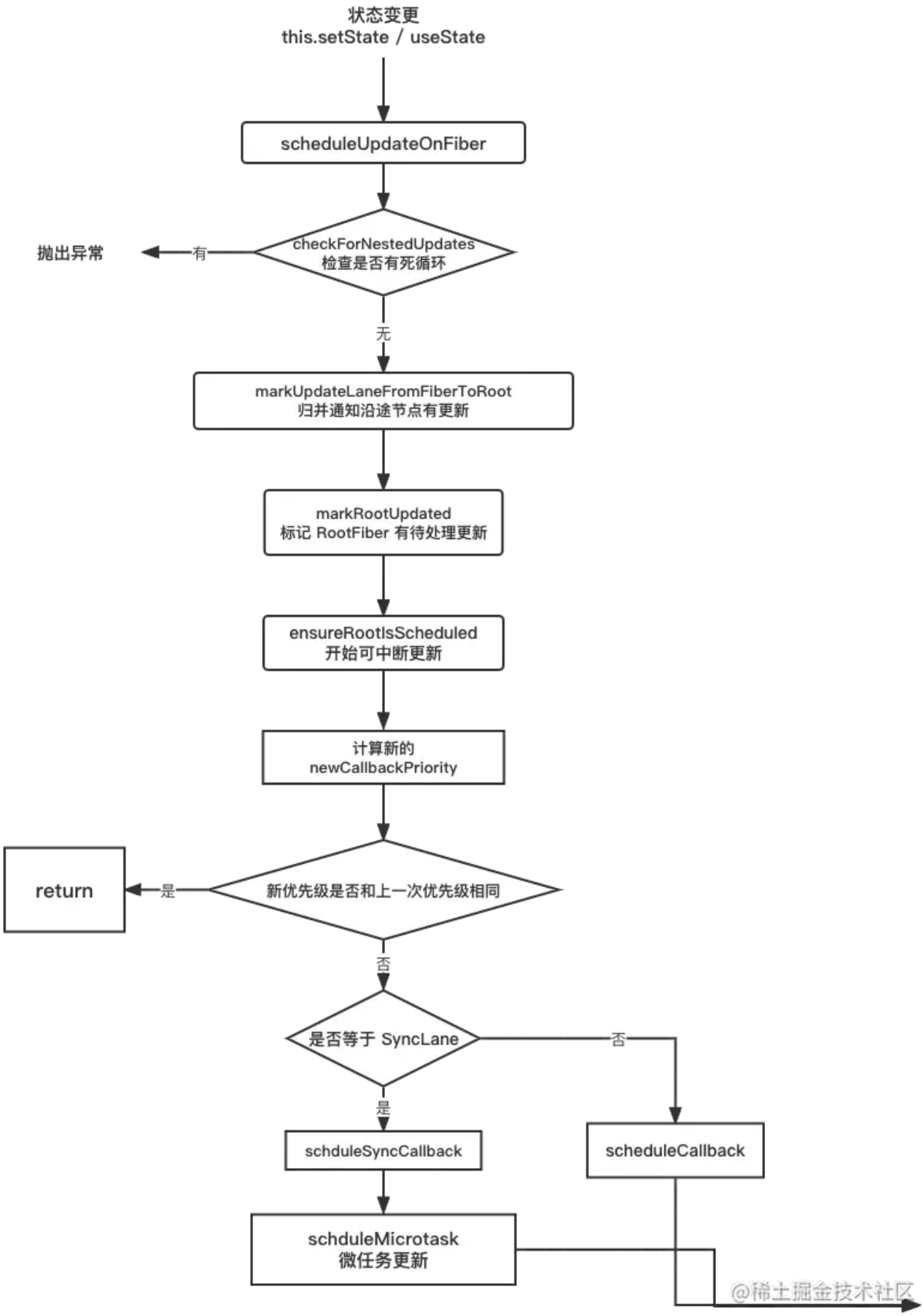
```
    case NormalPriority:
    default:
      timeout = NORMAL_PRIORITY_TIMEOUT;
      break;
  }
  // 过期时间
  var expirationTime = startTime + timeout;
  ...
  // 表示这个任务将会延迟执行
  if (startTime > currentTime) {
    // 当前任务已超时，插入超时队列
    newTask.sortIndex = startTime;
    push(timerQueue, newTask);
    if (peek(taskQueue) === null && newTask === peek(timerQueue)) {
      // 这个任务是最早延迟执行的
      if (isHostTimeoutScheduled) {
        // 取消现有的定时器
        cancelHostTimeout();
      } else {
        isHostTimeoutScheduled = true;
      }
      requestHostTimeout(handleTimeout, startTime - currentTime);
    }
  } else {
    // 任务未超时，插入调度任务队列
    newTask.sortIndex = expirationTime;
    // taskQueue 是一个二叉堆结构，以最小堆的形式存储 task
    push(taskQueue, newTask);
    if (enableProfiling) {
      markTaskStart(newTask, currentTime);
      newTask.isQueued = true;
    }
    // 符合更新调度执行的标志
    if (!isHostCallbackScheduled && !isPerformingWork) {
      isHostCallbackScheduled = true;
      requestHostCallback(flushWork);
    }
  }

  return newTask;
}
```

- `taskQueue` 里存放的是过期的任务，根据过期时间来排序，需要在调度的 `workLoop` 中循环执行完这些任务
- `timerQueue` 里存的都是没有过期的任务，依据任务的开始时间(`startTime`)排序，在调度 `workLoop` 中 会用 `advanceTimers` 检查任务是否过期，如果过期了，放入 `taskQueue` 队列。

总结

至此状态更新的大致流程我们已经讲解完毕，后面省略了一部分关于 Scheduler 部分的内容，会在后面 Schduler 部分单独讲解
以下就是完整流程图


状态变更
this.setState / useState
scheduleUpdateOnFiber
checkForNestedUpdates
检查是否有死循环
有
抛出异常
无
markUpdateLaneFromFiberToRoot
归并通知沿途节点有更新
markRootUpdated
标记 RootFiber 有待处理更新
ensureRootIsScheduled
开始可中断更新
计算新的
newCallbackPriority
新优先级是否和上一次优先级相同
是
return
否
是否等于 SyncLane
否
是
schduleSyncCallback
scheduleCallback
schduleMicrotask
微任务更新
@稀土掘金技术社区