8-4 scheduleUpdateOnFiber 调度更新

页面初次渲染、类组件 setState/forceUpdate、函数组件 setState 都会走到更新，都会调用 `scheduleUpdateOnFiber` 函数。

DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberWorkLoop.js

```
744  export function scheduleUpdateOnFiber(
745    root: FiberRoot,
746    fiber: Fiber,
747    lane: Lane,
748  ) {
```

从根节点开始更新。

如果以下源码没有特殊标记路径，那么路径都和 `scheduleUpdateOnFiber` 一样，即 src/react/packages/react-reconciler/src/ReactFiberWorkLoop.js。

1. `markRootUpdated`

标记根节点有一个 pending update，即待处理的更新。

Flow

```
markRootUpdated(root, lane);
```

源码：

```Flow
function markRootUpdated(root: FiberRoot, updatedLanes: Lanes) {
  _markRootUpdated(root, updatedLanes);

  if (enableInfiniteRenderLoopDetection) {
    // ? sy
    // Check for recursive updates
    if (executionContext & RenderContext) {
      workInProgressRootDidIncludeRecursiveRenderUpdate = true;
    } else if (executionContext & CommitContext) {
      didIncludeCommitPhaseUpdate = true;
    }
    // getRootForUpdatedFiber中也有这个检测
    // 如果循环超过限制次数(50次)，抛出错误。比如在类组件的render函数里执行setSt
    throwIfInfiniteUpdateLoopDetected();
  }
}
```

`_markRootUpdated` 的实现来自 react/packages/react-reconciler/src/ReactFiberLazyComponent.js 的 `markRootUpdated`

```Flow
export function markRootUpdated(root: FiberRoot, updateLane: Lane) {
  root.pendingLanes |= updateLane;

  // 如果update是idle的，将不会处理它，因为我们直到所有常规update完成后才会处理它
  if (updateLane !== IdleLane) {
    root.suspendedLanes = NoLanes;
    root.pingedLanes = NoLanes;
  }
}
```

2. `ensureRootIsScheduled`(root)

每次 root: FiberRoot 接收 update 的时候，这个函数都会被调用。

1. 确保 root 在 root 调度中

2. 确保有一个待处理的微任务来处理根调度。

react/packages/react-reconciler/src/ReactFiberRootScheduler.js

Flow

```
// 单向链表
// 一般应用中，只有一个根节点，但是React支持多根节点
let firstScheduledRoot: FiberRoot | null = null;
let lastScheduledRoot: FiberRoot | null = null;
// 用于没有同步work的时候，快速退出flushSync
let mightHavePendingSyncWork: boolean = false;

// 用于防止重复的微任务被调度。
let didScheduleMicrotask: boolean = false;

export function ensureRootIsScheduled(root: FiberRoot): void {
  // 把root添加到调度中
  if (root === lastScheduledRoot || root.next !== null) {
    // ? 后两次
    // 已经添加到了调度中
  } else {
    // ? sy 前一次
    if (lastScheduledRoot === null) {
      // ? sy
      firstScheduledRoot = lastScheduledRoot = root;
    } else {
      // 多个根节点
      lastScheduledRoot.next = root;
      lastScheduledRoot = root;
    }
  }

  // 每当root接收到update，我们将其设置为true，直到下次处理调度为止。如果为fals
  mightHavePendingSyncWork = true;

  // 在当前事件结束时，逐个检查每个root，并确保为每个root安排了正确优先级的任务。
  if (__DEV__ && ReactCurrentActQueue.current !== null) {
    // 略
  } else {
    //
    if (!didScheduleMicrotask) {
      // ? sy
```

```
      didScheduleMicrotask = true;
      scheduleImmediateTask(processRootScheduleInMicrotask);
    }
  }
}
```

scheduleImmediateTask

react/packages/react-reconciler/src/ReactFiberRootScheduler.js

DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberRootScheduler.js > ...

```
465  function scheduleImmediateTask(cb: () => mixed) {
466  >  if (__DEV__ && ReactCurrentActQueue.current !== null) {…
476     }
477
478     // TODO: Can we land supportsMicrotasks? Which environments don't support it?
479     // Alternatively, can we move this check to the host config?
480     if (supportsMicrotasks) {
481       // ? sy
482       scheduleMicrotask(() => {
483         // In Safari, appending an iframe forces microtasks to run.
484         // https://github.com/facebook/react/issues/22459
485         // We don't support running callbacks in the middle of render
486         // or commit so we need to check against that.
487         const executionContext = getExecutionContext();
488  >      if ((executionContext & (RenderContext | CommitContext)) !== NoContext) {…
499         }
500         cb();
501       });
502     } else {
503       // If microtasks are not supported, use Scheduler.
504       Scheduler_scheduleCallback(ImmediateSchedulerPriority, cb);
505     }
506  }
```

这里的 scheduleMicrotask 来源于 react/packages/react-dom-bindings/src/client/ReactFiberConfigDOM.js

DebugReact > src > react > packages > react-dom-bindings > src > client > JS ReactFiberConfigDOM.js

```
652  // -------------------
653  //     Microtasks
654  // -------------------
655  export const supportsMicrotasks = true;
656  export const scheduleMicrotask: any =
657    typeof queueMicrotask === 'function'
658      ? queueMicrotask
659      : typeof localPromise !== 'undefined'
660      ? callback =>
661          localPromise.resolve(null).then(callback).catch(handleErrorInNextTick)
662      : scheduleTimeout; // TODO: Determine the best fallback here.
```

关于 queueMicrotask

`Window` 或 `Worker` 接口的 **`queueMicrotask()`** 方法，**将微任务加入队列**以在控制返回浏览器的事件循环之前的安全时间执行。

更多参考 MDN

`processRootScheduleInMicrotask`

这个函数总是在 microtask 中被调用，它绝对不应该被同步调用。

react/packages/react-reconciler/src/ReactFiberRootScheduler.js

Flow

```
function processRootScheduleInMicrotask() {
  didScheduleMicrotask = false;

  // 我们将在遍历所有roots并调度它们时重新计算这个
  mightHavePendingSyncWork = false;

  const currentTime = now();

  let prev = null;
  let root = firstScheduledRoot;
  while (root !== null) {
    const next = root.next;

    if (
      currentEventTransitionLane !== NoLane &&
      shouldAttemptEagerTransition()
    ) {
      // ? sy-no
      upgradePendingLaneToSync(root, currentEventTransitionLane);
    }

    const nextLanes = scheduleTaskForRootDuringMicrotask(root, current
    if (nextLanes === NoLane) {
      // 页面初次渲染，再执行这里。nextLanes=0
      // root 没有更多pending work。从调度中移除它。为了防止微妙的重入bug，这个
      // 将其置null，以便我们知道它已从调度中移除
      root.next = null;
```

```
      if (prev === null) {
        // ? sy
        // This is the new head of the list
        firstScheduledRoot = next;
      } else {
        prev.next = next;
      }
      if (next === null) {
        // ? sy
        // This is the new tail of the list
        lastScheduledRoot = prev;
      }
    } else {
      // ? sy 1
      // 页面初次渲染，先执行这里。nextLanes=32
      // This root still has work. Keep it in the list.
      prev = root;
      if (includesSyncLane(nextLanes)) {
        mightHavePendingSyncWork = true;
      }
    }
    root = next;
  }

  currentEventTransitionLane = NoLane;

  // 在microtask结束时，flush任何pending的同步work。这必须放在最后，因为它执行
  flushSyncWorkOnAllRoots();
}
```

scheduleTaskForRootDuringMicrotask

返回 nextLanes

react/packages/react-reconciler/src/ReactFiberRootScheduler.js

DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberRootScheduler.js > scheduleTaskForRootDuringMicrotask

```
377     const newCallbackPriority = getHighestPriorityLane(nextLanes);
378
379     if (
380       newCallbackPriority === existingCallbackPriority &&
381       // Special case related to `act`. If the currently scheduled task is a
382       // Scheduler task, rather than an `act` task, cancel it and re-schedule
383       // on the `act` queue.
384     >   !(…
388       )
389     > ) {…
392     > } else {…
395     }
396
397     let schedulerPriorityLevel;
398     switch (lanesToEventPriority(nextLanes)) {
399       case DiscreteEventPriority:
400         schedulerPriorityLevel = ImmediateSchedulerPriority;
401         break;
402       case ContinuousEventPriority:
403         schedulerPriorityLevel = UserBlockingSchedulerPriority;
404         break;
405       case DefaultEventPriority:
406         // ? sy 页面初次渲染
407         schedulerPriorityLevel = NormalSchedulerPriority;
408         break;
409       case IdleEventPriority:
410         schedulerPriorityLevel = IdleSchedulerPriority;
411         break;
412       default:
413         schedulerPriorityLevel = NormalSchedulerPriority;
414         break;
415     }
416
417
418     // ? sy
419     const newCallbackNode = scheduleCallback(
420       schedulerPriorityLevel,
421       performConcurrentWorkOnRoot.bind(null, root),
422     );
423
424     root.callbackPriority = newCallbackPriority;
425     root.callbackNode = newCallbackNode;
426     return newCallbackPriority;
427   }      Andrew Clark, 11个月前 • Move update scheduling to microtask (#26512) …
428 }
```

markStarvedLanesAsExpired

检查是否有 lanes 挨饿，如果有，则标记他们过期，以便下次执行。详情参考 lanes 章节。

getWorkInProgressRoot

返回 ReactFiberWorkLoop.js 的全局变量：

JavaScript

```
// The root we're working on
let workInProgressRoot: FiberRoot | null = null;

export function getWorkInProgressRoot(): FiberRoot | null {
```

```
  return workInProgressRoot;
}
```

getWorkInProgressRootRenderLanes

返回 ReactFiberWorkLoop.js 的全局变量：

JavaScript

```
// The lanes we're rendering
export function getWorkInProgressRootRenderLanes(): Lanes {
  return workInProgressRootRenderLanes;
}
```

getNextLanes

获取下一个 lanes。详情参考 lanes 章节。

lanesToEventPriority

JavaScript

```
export function lanesToEventPriority(lanes: Lanes): EventPriority {
  // 根据优先级最高的lane，返回对应的 EventPriority。这里对应Scheduler包中的优
  const lane = getHighestPriorityLane(lanes);
  if (!isHigherEventPriority(DiscreteEventPriority, lane)) {
    return DiscreteEventPriority;
  }
  if (!isHigherEventPriority(ContinuousEventPriority, lane)) {
    return ContinuousEventPriority;
  }
  if (includesNonIdleWork(lane)) {
    return DefaultEventPriority;
  }
  return IdleEventPriority;
}
```

scheduleCallback

进入 scheduler 调度器

DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberRootScheduler.js

```
454
455   function scheduleCallback(
456     priorityLevel: PriorityLevel,
457     callback: RenderTaskFn,
458   ) {
459 >   if (__DEV__ && ReactCurrentActQueue.current !== null) {…
465     } else {
466       return Scheduler_scheduleCallback(priorityLevel, callback);
467     }
468   }
```

`performConcurrentWorkOnRoot`

详情参考下节 render 阶段。