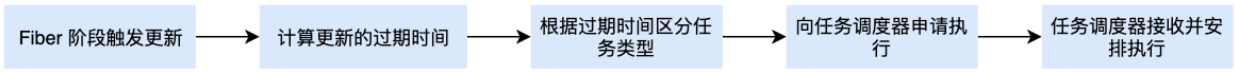
19 React Fiber 结点的更新任务如何被调度器执行

更新时间：2020-09-09 18:08:59

> 没有智慧的头脑，就象没有腊烛的灯笼。——列夫·托尔斯泰

前言

还记得 React 对「更新」如何定义的吗？Fiber 结点的更新对象中一个重要的属性是过期时间（expirationTime），该属性的值根据根据任务调度器返回的当前任务优先级来确定。确定好了 Fiber 结点的更新对象的过期时间后就对对该 Fiber 结点的更新任务进行调度。本节将会介绍 React Fiber 结点的更新任务如何被调度器执行。



在本章第一节中有介绍，在调用 computeExpirationForFiber 函数计算 Fiber 结点更新对象的过期时间时会使用任务调度器暴露出的 getCurrentPriorityLevel 函数，那么该函数是如何返回优先级的呢？

任务调度器如何返回 Fiber 结点更新任务的优先级

```
function unstable_getCurrentPriorityLevel() {
  return currentPriorityLevel;
}
```

代码示例 4.4.1 getCurrentPriorityLevel 函数返回当前任务优先级

额，unstable_getCurrentPriorityLevel 函数这么简单呀！问题是变量 currentPriorityLevel 的值由谁来确定呢？在上一节中有提到变量 currentPriorityLevel 有个初始值 NormalPriority（也就是数值 3），事实上是该变量的值是任务调度器在调度任务的过程中动态修改的，那么什么时候才会修改 currentPriorityLevel 的值呢？

1. ImmediatePriority 修改时机 — 执行同步任务

```
function scheduleCallbackForRoot(root, priorityLevel, expirationTime) {
  // 应用程序首次渲染时
  if (expirationTime === Sync) {
    // 调用scheduleSyncCallback立即执行该更新任务
    root.callbackNode = scheduleSyncCallback(runRootCallback.bind(null, root, renderRoot.bind(null, root, expirationTime)));
  }
}
function scheduleSyncCallback(callback) {
  // 修改调度器当前任务的优先级为ImmediatePriority
  immediateQueueCallbackNode = Scheduler_scheduleCallback(Scheduler_ImmediatePriority, flushSyncCallbackQueueImpl);
  // ...
}
```

代码示例 4.4.2 应用程序首次渲染时修改当前任务优先级

在本章第一节代码示例 4.1.5 中提到，如果检测到当前的 mode 为非批量更新（这里的批量更新指的是一个 Fiber 结点有多个更新要处理）返回的过期时间值为 Sync，也就是按照同步任务执行，因此会向任务调度器发送立即执行的请求。同时，任务调度器将当前的任务优先级设置为 ImmediatePriority。

2. UserBlockingPriority 修改时机 — 处理交互事件队列时

```
// 源码位置：packages/react-dom/src/events/DOMEventResponderSystem.js
function processEventQueue() {
  // ...
  switch (currentEventQueuePriority) {
    // 如果是用户交互阻塞事件
    if (enableUserBlockingEvents) {
      // 按照UserBlockingPriority级别执行任务
      runWithPriority(
        UserBlockingPriority,
        batchedEventUpdates.bind(null, processEvents, eventQueue),
      );
    } else {
      batchedEventUpdates(processEvents, eventQueue);
    }
  }
}
```

代码示例 4.4.3 处理事件队列时修改当前任务优先级

在处理交互事件队列时 React 会通过 runWithPriority 函数将 currentPriorityLevel 的值修改为 UserBlockingPriority。

更新任务以 callback 的形式向任务调度器申请执行

前面提到过一个问题，React 是如何定义「执行一个任务」呢？事实上，**执行一个带有更新处理逻辑的 callback 就是执行更新任务**。

scheduleCallbackForRoot 函数是 React 更新任务调度的入口函数，在该函数内部会检查当前的更新任务属于 **同步任务**（需要立即执行）还是 **非同步任务**，根据任务类型执行不同的调度逻辑，见代码示例 4.4.4。

```
// 源码位置：packages/react-reconciler/src/ReactFiberWorkLoop.js
function scheduleCallbackForRoot(root, priorityLevel, expirationTime) {
  // ...
  if (expirationTime === Sync) {
    // 执行同步任务，会告诉调度器该任务需要立即执行
    root.callbackNode = scheduleSyncCallback(runRootCallback.bind(null, root, renderRoot.bind(null, root, expirationTime)));
  } else {
    // 执行非同步任务
    root.callbackNode = scheduleCallback(priorityLevel, runRootCallback.bind(null, root, renderRoot.bind(null, root, expirationTime)), options);
  }
}
```

代码示例 4.4.4 更新任务处理入口

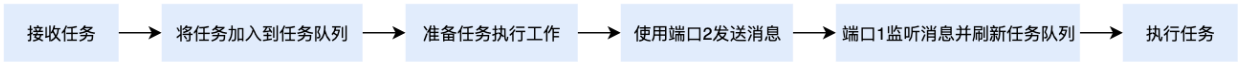
scheduleSyncCallback 函数用于执行同步任务，它会告诉任务调度器当前任务需要立即执行。scheduleCallback 函数用于执行非同步任务，同时将当前任务的优先级传给任务调度器（感觉这里只是为了同步 Fiber 结点里面的优先级和任务队列中的优先级）。两个函数有个共同的参数就是

```
runRootCallback.bind(null, root, renderRoot.bind(null, root, expirationTime))
```

这个表达式中用了两个 bind，但最终要执行的是 renderRoot 函数，该函数使得更新任务进入渲染阶段。

> renderRoot 函数在下一章 — React 应用程序首次渲染流程中将会详细介绍。

任务调度器接收任务并安排执行



这一部分在上一节中已经有过介绍，下面主要介绍任务调度器是如何准备任务执行工作以及使用端口发送消息的。

任务执行准备工作

这个过程的工作主要包括准备更新任务的 callback，检查待执行任务是否已经就绪等，见代码示例 4.4.5。

```
const channel = new MessageChannel();
const port = channel.port2;

requestHostCallback = function(callback) {
  // scheduledHostCallback为任务调度器维护的任务变量，存放即将要执行的任务
  scheduledHostCallback = callback;
  // ...
}

const onAnimationFrame = rAFTime => {
  if (scheduledHostCallback === null) {
    // 如果没有待执行的任务则退出
    prevRAFTime = -1;
    prevRAFInterval = -1;
    return;
  }

  // ...
  // 发送通道消息
  port.postMessage(null);
  // ...
}

// 监听到通道消息后立即执行任务
channel.port1.onmessage = performWorkUntilDeadline;
```

代码示例 4.4.5

在 performWorkUntilDeadline 函数中执行任务，见代码示例 4.4.6。

```
const performWorkUntilDeadline = () => {
  if (scheduledHostCallback !== null) {
    const currentTime = getCurrentTime();
    const hasTimeRemaining = frameDeadline - currentTime > 0;
    try {
      // 调用更新任务的callback，执行该任务
      const hasMoreWork = scheduledHostCallback(
        hasTimeRemaining,
        currentTime,
      );
      // 检查是否还有其他任务
      if (!hasMoreWork) {
        scheduledHostCallback = null;
      }
    } catch (error) {
      port.postMessage(null);
      throw error;
    }
  }
  needsPaint = false;
};
```

监听动画帧执行完成，然后发送消息通知

动画帧执行完成后，浏览器会有短暂的时间空隙，这时会执行回调函数 onAnimationFrame，在该回调函数内部通过 MessageChannel 的一个接口发送通道消息，在另一个接口处可通过 onmessage 事件监听到消息，然后执行下一个任务。

```
// 源码位置：packages/scheduler/src/forks/SchedulerHostConfig.default.js
// 重新完成绘制之后使用postMessage通知其他任务可以开始执行
const channel = new MessageChannel();
const port = channel.port2;
// 监听通道消息
channel.port1.onmessage = performWorkUntilDeadline;
// 动画帧执行结束的回调函数
const onAnimationFrame = rAFTime => {
  ...
  port.postMessage(null);
}
// 任务正式执行函数
const performWorkUntilDeadline = () => {
  if (scheduledHostCallback !== null) {
    const currentTime = getCurrentTime();
    const hasTimeRemaining = frameDeadline - currentTime > 0;
    try {
      // 正式执行任务，scheduledHostCallback为任务的回调函数
      const hasMoreWork = scheduledHostCallback(
        hasTimeRemaining,
        currentTime,
      );
      if (!hasMoreWork) {
        scheduledHostCallback = null;
      }
    } catch (error) {
      port.postMessage(null);
      throw error;
    }
  }
  needsPaint = false;
};
```

代码示例 4.4.2

> 注：这部分还有更多的内容可以去探索，有兴趣的话可以自行去了解。

小结

本文介绍了从 React Fiber 结点创建更新时，过期时间的计算到更新任务被执行的整体过程。在这个过程中 React 会经常和任务调度器进行对话，从任务调度器哪里获取最新的执行权。当 Fiber 结点获得到任务执行权后就开始了更新渲染阶段，这个过程主要分为 render 阶段和 commit 阶段，在下一章将会详细介绍 React 在这两个阶段具体做了哪些事情。

}

18 React 中的任务调度器（scheduler）

20 React 应用程序首次渲染时内部运行流程概述