18 React 中的任务调度器（scheduler）

更新时间：2020-09-04 09:39:15

> 耐心和恒心总会得到报酬的。——爱因斯坦

前言

任务调度器（`scheduler`）是 React 源码的一部分，但是它是一个独立于`react`和`react-dom`的模块。调度器在 React 应用程序运行中的主要工作就是负责更新任务的管理，如任务的时间分片，高优先级任务要先于低优先级任务执行等。本节主要介绍任务调度器的结构、调度器（与浏览器）的通信原理以及简单说明调度一个任务的流程。

任务调度器的基本结构

调度器中比较重要的几个属性就是`firstTask`，`firstDelayedTask`，`isSchedulerPaused`，`currentPriorityLevel`，见代码示例 4.3.1。

```
// 源码位置：packages/scheduler/src/Scheduler.js
// firstTask是任务队列（taskQueue）的入口
var firstTask = null;
// 第一个被延期执行的任务
var firstDelayedTask = null;
// debug时可以暂停调度器的工作
var isSchedulerPaused = false;
// 当前任务
var currentTask = null;
// 当前任务的优先级
var currentPriorityLevel = NormalPriority;
// 将任务加入到任务队列
function insertScheduledTask(newTask, expirationTime) { ... }
// 从任务队列中取出任务
function flushTask(task, currentTime) { ... }
```

代码示例 4.3.1 任务调度器的基本结构

任务调度器暴露出去的一些属性

调度器会将一些属性（方法/函数）暴露出去，这些属性可以直接在 react-dom 模块使用，见代码示例 4.3.2。

```
// 源码位置：packages/scheduler/src/Scheduler.js
export {
  // 任务优先级标识
  ImmediatePriority as unstable_ImmediatePriority,
  UserBlockingPriority as unstable_UserBlockingPriority,
  NormalPriority as unstable_NormalPriority,
  IdlePriority as unstable_IdlePriority,
  LowPriority as unstable_LowPriority,
  // 常用的任务处理方法
  unstable_runWithPriority
  unstable_next,
  unstable_scheduleCallback,
  unstable_cancelCallback,
  unstable_wrapCallback,
  unstable_getCurrentPriorityLevel,
  unstable_shouldYield,
  unstable_requestPaint,
  unstable_continueExecution,
  unstable_pauseExecution,
  unstable_getFirstCallbackNode,
  getCurrentTime as unstable_now,
  forceFrameRate as unstable_forceFrameRate,
}
```

代码示例 4.3.2 任务调度器暴露出的属性

调度器（ scheduler ）中很多方法名中带有 unstable ，表示该方法的逻辑「不稳定」，后续可能会继续修改。**此外，方法名里面的 callback 参数表示的是任务内容（对于程序来说，执行任务就是执行回调函数）**。下面对任务调度器（ scheduler ）中的部分调度方法做出简要介绍。

unstable_runWithPriority 函数

```
// 源码位置：packages/scheduler/src/Scheduler.js
function unstable_runWithPriority(priorityLevel, eventHandler) {
  switch (priorityLevel) {
    case ImmediatePriority:
    case UserBlockingPriority:
    case NormalPriority:
    case LowPriority:
    case IdlePriority:
      break;
    default:
      priorityLevel = NormalPriority;
  }
  var previousPriorityLevel = currentPriorityLevel;
  // 修改任务调度器中当前任务的优先级
  currentPriorityLevel = priorityLevel;

  try {
    // 执行回调（任务）
    return eventHandler();
  } finally {
    // 还原任务队列中当前任务的优先级
    currentPriorityLevel = previousPriorityLevel;
  }
}
```

代码示例 4.3.3 unstable_runWithPriority 函数

`unstable_runWithPriority` 函数会立即执行传入的回调函数（任务）。

unstable_scheduleCallback 函数

```
// 源码位置：packages/scheduler/src/Scheduler.js
function unstable_scheduleCallback(priorityLevel, callback, options) {
  var currentTime = getCurrentTime();
  var startTime;
  var timeout;
  ...
  // 根据options计算超时时间
  ...
  // 创新新的任务对象
  var expirationTime = startTime + timeout;
  var newTask = {
    callback,
    priorityLevel,
    startTime,
    expirationTime,
    next: null,
    previous: null,
  };
  if (startTime > currentTime) {
    // 如果是超时/延期任务，则先将该任务加入到超时任务队列
    insertDelayedTask(newTask, startTime);
    ...
    // 执行超时/延期任务
    requestHostTimeout(handleTimeout, startTime - currentTime);
  } else {
    // 不是超时任务，则将该任务加入到任务队列
    insertScheduledTask(newTask, expirationTime);
    ...
    // 从任务队列中取出任务执行
    requestHostCallback(flushWork);
  }
}
```

代码示例 4.3.4 unstable_scheduleCallback 函数

unstable_scheduleCallback 函数是分配任务执行权的入口。在该函数中会先根据 options 里面的信息计算出超时时间，判断是否是超时/延期任务，如果是则调度执行延期任务队列（优先级比较高），否则调度执行（正常）任务队列。

任务调度器内部的通信原理

window.requestAnimationFrame(callback) — 与浏览器通信

window.requestAnimationFrame(callback) 告诉浏览器——我们希望执行一个动画（帧），并且要求浏览器在下次重绘之前调用指定的回调函数（也是我们这里的任务）更新动画。该方法需要传入一个回调函数作为参数，该回调函数会在浏览器下一次重绘之前执行。

当我们调用此方法真棒更新动画（帧）时，将使浏览器在下一次重绘之前调用我们传入给该方法的动画函数（即回调函数）。回调函数执行次数通常是每秒 60 次，但在大多数遵循 W3C 建议的浏览器中，回调函数执行次数通常与浏览器屏幕刷新次数相匹配。为了提高性能和电池寿命，因此在大多数浏览器里，当 requestAnimationFrame() 运行在后台标签页或者隐藏的 iframe 里时，requestAnimationFrame() 会被暂停调用以提升性能和电池寿命。点击了解更多

window.MessageChannel — 调度器内部通信

Channel Messaging API 的 MessageChannel 接口允许我们创建一个新的消息通道，并通过它的两个 MessagePort 属性发送数据。点击了解更多

```
var channel = new MessageChannel();
var port1 = channel.port1;
var port2 = channel.port2;
port1.onmessage = function(event){
  console.log(event.data) // 动画帧执行完成，浏览器有空隙了
}
port2.postMessage('动画帧执行完成，浏览器有空隙了')
```

代码示例 4.3.5 使用 window.MessageChannel 传递消息

一般情况下，onmessage 的回调函数的调用时机是在浏览器的一帧绘制完成之后。看到这里，我们可以想一想，任务调度器是如何调度一个任务的呢？

任务调度器执行一个任务的流程

整体流程

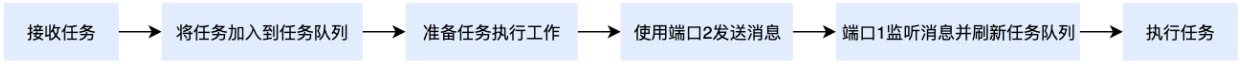

图 4.3.1 任务调度器执行一个任务的流程

流程说明

1. 任务调度器接收任务，并根据任务的特性分别将其加入到延期队列或者正常任务队列。如果当前没有正在执行的任务，则调用 requestHostCallback 函数准备执行任务。
2. 在准备执行任务时，会检测浏览器动画帧是否正在执行。如果正处于动画帧直接的时间空隙中，则使用 window.requestAnimationFrame 请求执行 callback。如果浏览器时机成熟，则通过 MessageChannel 发生消息，消息发送方式为 port.postMessage(null) 。

3. MessageChannel 的另一个端口以 `channel.port1.onmessage = performWorkUntilDeadline` 的方式监听消息，当收到消息后正式执行 callback（执行任务）。
4. 执行 callback，实际上是调用 `flushTask` 函数从任务队列中取出优先级最高的任务执行。

小结

本节主要介绍了 React 中的任务调度器，首先是其核心结构以及暴露出去的一些属性和方法，然后是任务调度器的通信原理包括与浏览器的通信和其内部通信。在文章最后简单说明了一个任务被执行的整体流程。**下一节，将会结合 Fiber 结点的更新介绍一个更新任务如何才能被执行。**

}

17 从设计者的角度理解 React 中的任务与任务队列

19 React Fiber 结点的更新任务如何被调度器执行