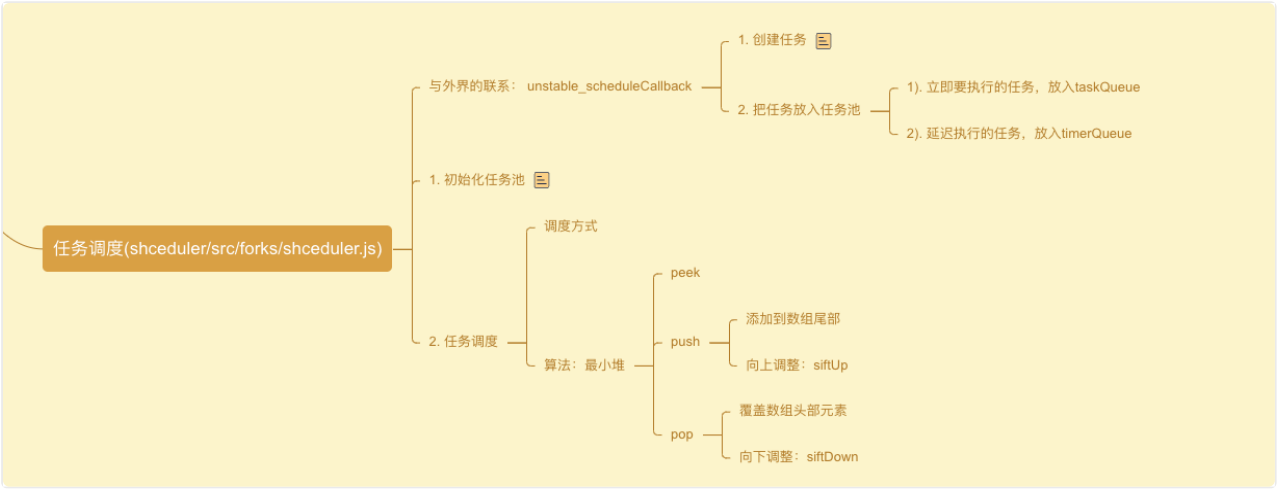
6-2 剖析 React 任务调度源码



在 React 应用运行过程当中，有一些任务要执行，这些任务分别有不同的**优先级**，比如从高优先级到低优先级分别为：

react/packages/scheduler/src/SchedulerPriorities.js

TypeScript

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

export const NoPriority = 0;
export const ImmediatePriority = 1;
export const UserBlockingPriority = 2;
export const NormalPriority = 3;
```

```
export const LowPriority = 4;
export const IdlePriority = 5;
```

那么，在每次只能执行一个任务的前提下，React 如何依次执行这些任务呢?

这个时候就需要一种调度策略了。

单线程任务调度器实现

任务池

在 React `scheduler` 中，调度任务包括了两种：分别是可以立即执行的任务与需要延迟执行的任务。

因此任务池也定义了两个：

react/packages/scheduler/src/forks/Scheduler.js

TypeScript

```
export type Callback = boolean => ?Callback;
export opaque type Task = {
  id: number,
  callback: Callback | null,
  priorityLevel: PriorityLevel,
  startTime: number,
  expirationTime: number,
  sortIndex: number,
  isQueued?: boolean,
};

// 任务存储，最小堆
const taskQueue: Array<Task> = []; // 最小堆
const timerQueue: Array<Task> = []; // 最小堆

var taskIdCounter = 1;
```

但是实际上，React 中实际需要调度的任务目前只有前者。

为了源码保持一致，在 Mini React 中，这两种我们都实现了 ~

一些常量

react/packages/scheduler/src/forks/Scheduler.js

```JavaScript
// 任务池，最小堆
const taskQueue: Array<Task> = []; // 没有延迟的任务
const timerQueue: Array<Task> = []; // 有延迟的任务

//标记task的唯一性
let taskIdCounter = 1;

let currentTask: Task | null = null;
let currentPriorityLevel: PriorityLevel = NormalPriority;

// 记录时间切片的起始值，时间戳
let startTime = -1;

// 时间切片，这是个时间段
let frameInterval = 5;

// 锁
// 是否有 work 在执行
let isPerformingWork = false;

// 主线程是否在调度
let isHostCallbackScheduled = false;

let isMessageLoopRunning = false;

// 是否有任务在倒计时
var isHostTimeoutScheduled = false;

let taskTimeoutID = -1;
```

调度任务入口-scheduleCallback

react/packages/scheduler/src/forks/Scheduler.js

JavaScript

```
function unstable_scheduleCallback(
  priorityLevel: PriorityLevel,
  callback: Callback,
  options?: {delay: number},
): Task {


  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:
      // Times out immediately
      timeout = -1;
      break;
    case UserBlockingPriority:
      // Eventually times out
      timeout = userBlockingPriorityTimeout; // 250
      break;
    case IdlePriority:
      // Never times out
      timeout = maxSigned31BitInt; // Math.pow(2, 30) - 1， 最大的31位整
      break;
    case LowPriority:
```

```
      // Eventually times out
      timeout = lowPriorityTimeout; // 10000
      break;
    case NormalPriority:
    default:
      // Eventually times out
      timeout = normalPriorityTimeout; // 5000
      break;
  }

  var expirationTime = startTime + timeout;

  var newTask: Task = {
    id: taskIdCounter++,
    callback,
    priorityLevel,
    startTime,
    expirationTime,
    sortIndex: -1,
  };


  if (startTime > currentTime) {
    // 有delay的任务
    newTask.sortIndex = startTime;
    push(timerQueue, newTask);
    if (peek(taskQueue) === null && newTask === peek(timerQueue)) {
      // 所有任务都延迟了，而这是延迟时间最短的任务。
      if (isHostTimeoutScheduled) {
        // 取消现有的setTimeout
        cancelHostTimeout();
      } else {
        isHostTimeoutScheduled = true;
      }
      // setTimeout
      requestHostTimeout(handleTimeout, startTime - currentTime);
    }
  } else {
    // 没有delay的任务
    newTask.sortIndex = expirationTime;
    push(taskQueue, newTask);
```

```
    // 如果需要的话，调度一个HostCallback。如果我们已经在执行work，就等到下次我
    if (!isHostCallbackScheduled && !isPerformingWork) {
      isHostCallbackScheduled = true;
      requestHostCallback();
    }
  }

  return newTask;
}
```