link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=124 sentences=227, words=3289

1.基础知识

1.1 最小堆

1.1.1 二叉树

- 每个节点最多有两个子节点

1.1.2 满二叉树

- 除最后一层无任何子节点外，每一层上的所有结点都有两个子结点的二叉树

1.1.3 完全二叉树

- 叶子结点只能出现在最下层和次下层
- 且最下层的叶子结点集中在树的左部

1.1.4 最小堆

- processon (https://www.processon.com/diagraming/61f26156e0b34d06c3b5bf48)
- 最小堆是一种经过排序的完全二叉树
- 其中任一非终端节点的数据值均不大于其左子节点和右子节点的值
- 根结点值是所有堆结点值中最小者
- 编号关系
    - 左子节点编号=父节点编号 _2 1_2=2
    - 右子节点编号=左子节点编号+1
    - 父节点编号=子节点编号/2 2/2=1
- 索引关系
    - 左子节点索引=(父节点索引+1) _2-1 (0+1)_2-1=1
    - 右子节点索引=左子节点索引+1
    - 父节点索引=(子节点索引-1)/2 (1-1)/2=0
- Unsigned_right_shift (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Unsigned_right_shift)

react\packages\scheduler\src\SchedulerMinHeap.js

```
export function push(heap, node) {
  const index = heap.length;
  heap.push(node);
  siftUp(heap, node, index);
}
export function peek(heap) {
  const first = heap[0];
  return first === undefined ? null : first;
}
export function pop(heap) {
  const first = heap[0];
  if (first !== undefined) {
    const last = heap.pop();
    if (last !== first) {
      heap[0] = last;
      siftDown(heap, last, 0);
    }
    return first;
  } else {
    return null;
  }
}

function siftUp(heap, node, i) {
  let index = i;
  while (true) {
    const parentIndex = index - 1 >>> 1;
    const parent = heap[parentIndex];
    if (parent !== undefined && compare(parent, node) > 0) {
      heap[parentIndex] = node;
      heap[index] = parent;
      index = parentIndex;
    } else {

      return;
    }
  }
}

function siftDown(heap, node, i) {
  let index = i;
  const length = heap.length;
  while (index < length) {
    const leftIndex = (index + 1) * 2 - 1;
    const left = heap[leftIndex];
    const rightIndex = leftIndex + 1;
    const right = heap[rightIndex];
    if (left !== undefined && compare(left, node) < 0) {
      if (right !== undefined && compare(right, left) < 0) {
        heap[index] = right;
        heap[rightIndex] = node;
        index = rightIndex;
      } else {
        heap[index] = left;
        heap[leftIndex] = node;
        index = leftIndex;
      }
    } else if (right !== undefined && compare(right, node) < 0) {
      heap[index] = right;
      heap[rightIndex] = node;
      index = rightIndex;
    } else {
      return;
    }
  }
}

function compare(a, b) {
  const diff = a.sortIndex - b.sortIndex;
  return diff !== 0 ? diff : a.id - b.id;
}
```

1.2 MessageChannel

- 目前 `requestIdleCallback` 目前只有Chrome支持
- 所以目前 React利用MessageChannel (https://developer.mozilla.org/zh-CN/docs/Web/API/MessageChannel)模拟了requestIdleCallback，将回调延迟到绘制操作之后执行
- MessageChannel API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
- MessageChannel创建了一个通信的管道，这个管道有两个端口，每个端口都可以通过postMessage发送数据，而一个端口只要绑定了onmessage回调方法，就可以接收从另一个端口传过来的数据
- MessageChannel是一个宏任务

```
var channel = new MessageChannel();
```

```
var channel = new MessageChannel();
var port1 = channel.port1;
var port2 = channel.port2;
port1.onmessage = function(event) {
    console.log("port1收到来自port2的数据：" + event.data);
}
port2.onmessage = function(event) {
    console.log("port2收到来自port1的数据：" + event.data);
}
port1.postMessage("发送给port2");
port2.postMessage("发送给port1");
```

2.实现基本任务调度

2.1 src\index.js

src\index.js

```
import { scheduleCallback } from "./scheduler";
function calculate() {
  let result = 0;
  for (let i = 0; i < 100000000; i++) {

    result += 1;
  }
  console.log(result);
}
scheduleCallback(calculate);
```

2.2 scheduler\index.js

src\scheduler\index.js

```
export * from './src/Scheduler';
```

2.3 Scheduler.js

src\scheduler\src\Scheduler.js

```

function scheduleCallback(callback) {

    callback();
}

export {
    scheduleCallback
}
```

3.实现时间切片

- 任务执行和用户交互是互斥的，如果任务执行时间过长会引起页面卡顿
- 可以把任务的执行时间切成多个时间片，每个帧最大执行时间为5ms
- 如果任务执行超过时间分片，则会任务会暂停执行，等下一帧的时间再执行

3.1 src\index.js

```
+import { scheduleCallback, shouldYield } from "./scheduler";
let result = 0;
let i = 0;
/**
 * 总任务
 * @returns
 */
function calculate() {
+  for (; i < 10000000 && (!shouldYield()); i++) {//7个0
+    result += 1;
+  }
+  if (result < 10000000) {
+    return calculate;
+  } else {
+    console.log('result', result);
+    return null;
+  }
}
scheduleCallback(calculate);
```

3.2 Scheduler.js

src\scheduler\src\Scheduler.js

```
+import { requestHostCallback, shouldYieldToHost } from './SchedulerHostConfig';
/**
 * 调度一个工作
 * @param {*} callback 要执行的工作
 */
function scheduleCallback(callback) {
    //执行工作
+   requestHostCallback(callback);
}

export {
    scheduleCallback,
+   shouldYieldToHost as shouldYield
}
```

3.3 SchedulerHostConfig.js

src\scheduler\src\SchedulerHostConfig.js

```
let deadline = 0;

let scheduledHostCallback = null;

let yieldInterval = 5;

const channel = new MessageChannel();
channel.port1.onmessage = performWorkUntilDeadline;

export function getCurrentTime() {
    return performance.now();
}

export function shouldYieldToHost() {
    const currentTime = getCurrentTime();
    return currentTime >= deadline;
};

export function performWorkUntilDeadline() {
    const currentTime = getCurrentTime();

    deadline = currentTime + yieldInterval;

    const hasMoreWork = scheduledHostCallback();

    if (hasMoreWork) {
        channel.port2.postMessage(null);
    }
}

export function requestHostCallback(callback) {
    scheduledHostCallback = callback;
    channel.port2.postMessage(null);
};
```

4.调度多个任务

- 如果同时调度多个任务需要保证任务有序执行

4.1 src\index.js

src\index.js

```
import { scheduleCallback, shouldYield } from "./scheduler";
let result = 0;
let i = 0;
/**
 * 总任务
 * @returns
 */
function calculate() {
  for (; i < 10000000 && (!shouldYield()); i++) {//7个0
    result += 1;
  }
  if (result < 10000000) {
    return calculate;
  } else {
    console.log('result', result);
    return null;
  }
}
+let result2 = 0;
+let i2 = 0;
+/**
+ * 总任务
+ * @returns
+ */
+function calculate2() {
+  for (; i2 < 10000000 && (!shouldYield()); i2++) {
+    result2 += 1;
+  }
+  if (result2 < 10000000) {
+    return calculate;
+  } else {
+    console.log('result2', result2);
+    return null;
+  }
+}
scheduleCallback(calculate);
+scheduleCallback(calculate2);
```

4.2 Scheduler.js

src\scheduler\src\Scheduler.js

```
+import { requestHostCallback, shouldYieldToHost as shouldYield } from './SchedulerHostConfig';
+//任务队列
+let taskQueue = [];
+let currentTask;
/**
 * 调度一个工作
 * @param {*} callback 要执行的工作
 */
function scheduleCallback(callback) {
+   //把此工作添加到任务队列中
+   taskQueue.push(callback);
+   //开始调度flushWork
+   requestHostCallback(flushWork);
}
+/**
+ * 清空任务队列
+ * @returns 队列中是否还有任务
+ */
+function flushWork() {
+    return workLoop();
+}
+/**
+ * 清空任务队列
+ * @returns 队列中是否还有任务
+ */
+function workLoop() {
+    //取出第一个任务
+    currentTask = taskQueue[0];
+    //如果任务存在
+    while (currentTask) {
+        //如果当前的时间片到期了,退出工作循环
+        if (shouldYield()) {
+            break;
+        }
+        //执行当前的工作
+        const continuationCallback = currentTask();
+        //如果返回函数说明任务尚未结束,下次还执行它
+        if (typeof continuationCallback === 'function') {
+            currentTask = continuationCallback;
+        } else {
+            //否则表示此任务执行结束，可以把此任务移除队列
+            taskQueue.shift();
+        }
+        //还取第一个任务
+        currentTask = taskQueue[0];
+    }
+    return currentTask;
+}
+
+export {
+    scheduleCallback,
+    shouldYield
+}
```

5.任务优先级

- 如果想后开始的任务先执行就需要增加任务优先级

5.1 src\index.js

```
import {
  scheduleCallback,
  shouldYield,
+ ImmediatePriority,//-1
+ UserBlockingPriority,//250
+ NormalPriority,//5000
+ LowPriority,//10000
+ IdlePriority,//1073741823
} from "./scheduler";
let result = 0;
let i = 0;
/**
 * 总任务
 * @returns
 */
+function calculate(didTimeout) {
+ for (; i < 10000000 && (!shouldYield() || didTimeout); i++) {//7个0
    result += 1;
  }
  if (result < 10000000) {
    return calculate;
  } else {
    console.log('result', result);
    return null;
  }
}

let result2 = 0;
let i2 = 0;
+function calculate2(didTimeout) {
+  for (; i2 < 10000000 && (!shouldYield() || didTimeout); i2++) {
    result2 += 1;
  }
  if (result2 < 10000000) {
    return calculate2;
  } else {
    console.log('result2', result2);
    return null;
  }
}
+let result3 = 0;
+let i3 = 0;
+function calculate3(didTimeout) {
+  for (; i3 < 10000000 && (!shouldYield() || didTimeout); i3++) {
+    result3 += 1;
+  }
+  if (result3 < 10000000) {
+    return calculate3;
+  } else {
+    console.log('result3', result3);
+    return null;
+  }
+}
+scheduleCallback(ImmediatePriority, calculate);//-1
+scheduleCallback(LowPriority, calculate2);//10000
+scheduleCallback(UserBlockingPriority, calculate3);//250
```

5.2 SchedulerHostConfig.js

src\scheduler\src\SchedulerHostConfig.js

```
//截止时间
let deadline = 0;
//当前正在调度执行的工作
let scheduledHostCallback = null;
//每帧的时间片
let yieldInterval = 5;

const channel = new MessageChannel();
channel.port1.onmessage = performWorkUntilDeadline;

/**
 * 获取当前的时间戳
 * @returns 当前的时间戳
 */
export function getCurrentTime() {
    return performance.now();
}
/**
 * 判断是否到达了本帧的截止时间
 * @returns 是否需要暂停执行
 */
export function shouldYieldToHost() {
    const currentTime = getCurrentTime();
    return currentTime >= deadline;
};

/**
 * 执行工作直到截止时间
 */
export function performWorkUntilDeadline() {
    const currentTime = getCurrentTime();
    //计算截止时间
    deadline = currentTime + yieldInterval;
    //执行工作
+   const hasMoreWork = scheduledHostCallback(currentTime);
    //如果此工作还没有执行完，则再次调度
    if (hasMoreWork) {
        channel.port2.postMessage(null);
    }
}

/**
 * 请求宿主的回调函数执行
 * @param {*} callback
 */
export function requestHostCallback(callback) {
    scheduledHostCallback = callback;
    channel.port2.postMessage(null);
};
```

5.3 Scheduler.js

src\scheduler\src\Scheduler.js

```
+import { requestHostCallback, shouldYieldToHost as shouldYield, getCurrentTime } from './+SchedulerHostConfig';
+import { push, pop, peek } from './SchedulerMinHeap';
+import { ImmediatePriority, UserBlockingPriority, NormalPriority, LowPriority, IdlePriority } from './SchedulerPriorities';
+// 不同优先级对应的不同的任务过期时间间隔
+let maxSigned31BitInt = 1073741823;
+let IMMEDIATE_PRIORITY_TIMEOUT = -1;//立即执行的优先级，级别最高
+let USER_BLOCKING_PRIORITY_TIMEOUT = 250;//用户阻塞级别的优先级
+let NORMAL_PRIORITY_TIMEOUT = 5000;//正常的优先级
+let LOW_PRIORITY_TIMEOUT = 10000;//较低的优先级
+let IDLE_PRIORITY_TIMEOUT = maxSigned31BitInt;//优先级最低，表示任务可以闲置
//任务队列
let taskQueue = [];
let currentTask;
+//下一个任务ID编号
+let taskIdCounter = 1;
/**
 * 调度一个工作
 * @param {*} callback 要执行的工作
 */
+function scheduleCallback(priorityLevel, callback) {
+    // 获取当前时间，它是计算任务开始时间、过期时间和判断任务是否过期的依据
+    let currentTime = getCurrentTime();
+    // 确定任务开始时间
+    let startTime = currentTime;
+    // 计算过期时间
+    let timeout;
+    switch (priorityLevel) {
+        case ImmediatePriority://1
+            timeout = IMMEDIATE_PRIORITY_TIMEOUT;//-1
+            break;
+        case UserBlockingPriority://2
+            timeout = USER_BLOCKING_PRIORITY_TIMEOUT;//250
+            break;
+        case IdlePriority://5
+            timeout = IDLE_PRIORITY_TIMEOUT;//1073741823
+            break;
+        case LowPriority://4
+            timeout = LOW_PRIORITY_TIMEOUT;//10000
+            break;
+        case NormalPriority://3
+        default:
+            timeout = NORMAL_PRIORITY_TIMEOUT;//5000
+            break;
+    }
+    //计算超时时间
+    let expirationTime = startTime + timeout;
+    //创建新任务
+    let newTask = {
+        id: taskIdCounter++,//任务ID
+        callback,//真正的任务函数
+        priorityLevel,//任务优先级，参与计算任务过期时间
+        expirationTime,//表示任务何时过期，影响它在taskQueue中的排序
+        //为小顶堆的队列提供排序依据
+        sortIndex: -1
+    };
+    newTask.sortIndex = expirationTime;
+    //把此工作添加到任务队列中
+    push(taskQueue, newTask);
+    //开始调度flushWork
+    requestHostCallback(flushWork);
+}
/**
 * 清空任务队列
 * @returns 队列中是否还有任务
 */
+function flushWork(initialTime) {
+    return workLoop(initialTime);
+}
/**
 * 清空任务队列
 * @returns 队列中是否还有任务
 */
+function workLoop(initialTime) {
+    //当前时间
+    let currentTime = initialTime;
+    //取出第一个任务
+    currentTask = peek(taskQueue);
+    //如果任务存在
+    while (currentTask) {
+        //如果当前任务的过期时间大于当前时间,并且当前的时间片到期了,退出工作循环
+        if (currentTask.expirationTime > currentTime && shouldYield()) {
+            break;
+        }
+        //执行当前的工作
+        //const continuationCallback = currentTask();
+        const callback = currentTask.callback;
+        if (typeof callback === 'function') {
+            currentTask.callback = null;
+            const didUserCallbackTimeout = currentTask.expirationTime
+            const continuationCallback = callback(didUserCallbackTimeout);
+            //如果返回函数说明任务尚未结束,下次还执行它
+            if (typeof continuationCallback === 'function') {
+                //currentTask = continuationCallback;
+                currentTask.callback = continuationCallback;
+            } else {
+                //否则表示此任务执行结束，可以把此任务移除队列
+                //taskQueue.shift();
+                pop(taskQueue);
+            }
+        } else {
+            pop(taskQueue);
+        }
+        //还取第一个任务
+        currentTask = peek(taskQueue);
+    }
+    return currentTask;
+}

export {
    scheduleCallback,
    shouldYield,
+   ImmediatePriority,
+   UserBlockingPriority,
+   NormalPriority,
+   IdlePriority,
+   LowPriority
}
```

5.4 SchedulerPriorities.js

src\scheduler\src\SchedulerPriorities.js

```
export const NoPriority = 0;
export const ImmediatePriority = 1;
export const UserBlockingPriority = 2;
export const NormalPriority = 3;
export const LowPriority = 4;
export const IdlePriority = 5;
```

5.5 SchedulerMinHeap.js

src\scheduler\src\SchedulerMinHeap.js

```
export function push(heap, node) {
    const index = heap.length;
    heap.push(node);
    siftUp(heap, node, index);
}
export function peek(heap) {
    const first = heap[0];
    return first === undefined ? null : first;
}
export function pop(heap) {
    const first = heap[0];
    if (first !== undefined) {
        const last = heap.pop();
        if (last !== first) {
            heap[0] = last;
            siftDown(heap, last, 0);
        }
        return first;
    } else {
        return null;
    }
}

function siftUp(heap, node, i) {
    let index = i;
    while (true) {
        const parentIndex = index - 1 >>> 1;
        const parent = heap[parentIndex];
        if (parent !== undefined && compare(parent, node) > 0) {
            heap[parentIndex] = node;
            heap[index] = parent;
            index = parentIndex;
        } else {

            return;
        }
    }
}

function siftDown(heap, node, i) {
    let index = i;
    const length = heap.length;
    while (index < length) {
        const leftIndex = (index + 1) * 2 - 1;
        const left = heap[leftIndex];
        const rightIndex = leftIndex + 1;
        const right = heap[rightIndex];
        if (left !== undefined && compare(left, node) < 0) {
            if (right !== undefined && compare(right, left) < 0) {
                heap[index] = right;
                heap[rightIndex] = node;
                index = rightIndex;
            } else {
                heap[index] = left;
                heap[leftIndex] = node;
                index = leftIndex;
            }
        } else if (right !== undefined && compare(right, node) < 0) {
            heap[index] = right;
            heap[rightIndex] = node;
            index = rightIndex;
        } else {
            return;
        }
    }
}

function compare(a, b) {
    const diff = a.sortIndex - b.sortIndex;
    return diff !== 0 ? diff : a.id - b.id;
}
```

6.延迟任务

6.1 src\index.js

```
import {
  scheduleCallback,
  shouldYield,
  ImmediatePriority,//-1
  UserBlockingPriority,//250
  NormalPriority,//5000
  LowPriority,//10000
  IdlePriority,//1073741823
} from "./scheduler";
let result = 0;
let i = 0;
/**
 * 总任务
 * @returns
 */
function calculate(didTimeout) {
  for (; i < 10000000 && (!shouldYield() || didTimeout); i++) {//7个0
    result += 1;
  }
  if (result < 10000000) {
    return calculate;
  } else {
    console.log('result', result);
    return null;
  }
}

let result2 = 0;
let i2 = 0;
+console.time('cost');
function calculate2(didTimeout) {
+ if (i2 === 0)
+   console.timeEnd('cost');
  for (; i2 < 10000000 && (!shouldYield() || didTimeout); i2++) {
    result2 += 1;
  }
  if (result2 < 10000000) {
    return calculate2;
  } else {
    console.log('result2', result2);
    return null;
  }
}

scheduleCallback(ImmediatePriority, calculate);//-1
+scheduleCallback(LowPriority, calculate2, { delay: 10000 });//10000
```

6.2 SchedulerHostConfig.js

src\scheduler\src\SchedulerHostConfig.js

```
//截止时间
let deadline = 0;
//当前正在调度执行的工作
let scheduledHostCallback = null;
//每帧的时间片
let yieldInterval = 5;
+let taskTimeoutID = -1;
const channel = new MessageChannel();
channel.port1.onmessage = performWorkUntilDeadline;

/**
 * 获取当前的时间戳
 * @returns 当前的时间戳
 */
export function getCurrentTime() {
    return performance.now();
}
/**
 * 判断是否到达了本帧的截止时间
 * @returns 是否需要暂停执行
 */
export function shouldYieldToHost() {
    const currentTime = getCurrentTime();
    return currentTime >= deadline;
};

/**
 * 执行工作直到截止时间
 */
export function performWorkUntilDeadline() {
    const currentTime = getCurrentTime();
    //计算截止时间
    deadline = currentTime + yieldInterval;
    //执行工作
    const hasMoreWork = scheduledHostCallback(currentTime);
    //如果此工作还没有执行完，则再次调度
    if (hasMoreWork) {
        channel.port2.postMessage(null);
    }
}

/**
 * 请求宿主的回调函数执行
 * @param {*} callback
 */
export function requestHostCallback(callback) {
    scheduledHostCallback = callback;
    channel.port2.postMessage(null);
};

+export function requestHostTimeout(callback, ms) {
+    taskTimeoutID = setTimeout(() => {
+        callback(getCurrentTime());
+    }, ms);
+};
```

6.3 Scheduler.js

src\scheduler\src\Scheduler.js

```
+import { requestHostCallback, shouldYieldToHost as shouldYield, getCurrentTime, requestHostTimeout } from './SchedulerHostConfig';
import { push, pop, peek } from './SchedulerMinHeap';
import { ImmediatePriority, UserBlockingPriority, NormalPriority, LowPriority, IdlePriority } from './SchedulerPriorities';
// 不同优先级对应的不同的任务过期时间间隔
let maxSigned31BitInt = 1073741823;
let IMMEDIATE_PRIORITY_TIMEOUT = -1;//立即执行的优先级，级别最高
let USER_BLOCKING_PRIORITY_TIMEOUT = 250;//用户阻塞级别的优先级
let NORMAL_PRIORITY_TIMEOUT = 5000;//正常的优先级
let LOW_PRIORITY_TIMEOUT = 10000;//较低的优先级
let IDLE_PRIORITY_TIMEOUT = maxSigned31BitInt;//优先级最低，表示任务可以闲置
//已经可以开始执行队列
let taskQueue = [];
+//尚未可以开始执行队列
+let timerQueue = [];
let currentTask;
//下一个任务ID编号
let taskIdCounter = 1;
/**
 * 调度一个工作
 * @param {*} callback 要执行的工作
 */
+function scheduleCallback(priorityLevel, callback, options) {
    // 获取当前时间，它是计算任务开始时间、过期时间和判断任务是否过期的依据
    let currentTime = getCurrentTime();
    // 确定任务开始时间
+    let 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;
+    }
    // 计算过期时间
    let timeout;
    switch (priorityLevel) {
        case ImmediatePriority://1
            timeout = IMMEDIATE_PRIORITY_TIMEOUT;//-1
            break;
        case UserBlockingPriority://2
            timeout = USER_BLOCKING_PRIORITY_TIMEOUT;//250
            break;
        case IdlePriority://5
            timeout = IDLE_PRIORITY_TIMEOUT;//1073741823
            break;
        case LowPriority://4
            timeout = LOW_PRIORITY_TIMEOUT;//10000
            break;
        case NormalPriority://3
        default:
            timeout = NORMAL_PRIORITY_TIMEOUT;//5000
            break;
    }
    //计算超时时间
    let expirationTime = startTime + timeout;
    //创建新任务
    let newTask = {
        id: taskIdCounter++,//任务ID
        callback,//真正的任务函数
        priorityLevel,//任务优先级，参与计算任务过期时间
+       startTime,
        expirationTime,//表示任务何时过期，影响它在taskQueue中的排序
        //为小顶堆的队列提供排序依据
        sort
    };
+   if (startTime > currentTime) {
+       newTask.sortIndex = startTime;
+       push(timerQueue, newTask);
+       if (peek(taskQueue) === null && newTask === peek(timerQueue)) {
+           requestHostTimeout(handleTimeout, startTime - currentTime);
+       }
+   } else {
        newTask.sortIndex = expirationTime;
        //把此工作添加到任务队列中
        push(taskQueue, newTask);
        //开始调度flushWork
        requestHostCallback(flushWork);
+   }
}
+/**
+ * 处理超时任务
+ * @param {*} currentTime
+ */
+function handleTimeout(currentTime) {
+    advanceTimers(currentTime);
+    if (peek(taskQueue) !== null) {
+        requestHostCallback(flushWork);
+    } else {
+        const firstTimer = peek(timerQueue);
+        if (firstTimer !== null) {
+            requestHostTimeout(handleTimeout, firstTimer.startTime - currentTime);
+        }
+    }
+}
+function advanceTimers(currentTime) {
+    let timer = peek(timerQueue);
+    while (timer !== null) {
+        if (timer.callback === null) {
+            pop(timerQueue);
+        } else if (timer.startTime
+            pop(timerQueue);
+            timer.sortIndex = timer.expirationTime;
+            push(taskQueue, timer);
+        } else {
+            return;
+        }
+        timer = peek(timerQueue);
+    }
+}
/**
 * 清空任务队列
 * @returns 队列中是否还有任务
 */
function flushWork(initialTime) {
    return workLoop(initialTime);
}
/**
 * 清空任务队列
 * @returns 队列中是否还有任务
 */
function workLoop(initialTime) {
    //当前时间
    let currentTime = initialTime;
    //取出第一个任务
    currentTask = peek(taskQueue);
    //如果任务存在
    while (currentTask) {
        //如果当前任务的过期时间大于当前时间,并且当前的时间片到期了,退出工作循环
        if (currentTask.expirationTime > currentTime && shouldYield()) {
            break;
        }
        //执行当前的工作
        //const continuationCallback = currentTask();
        const callback = currentTask.callback;
        if (typeof callback
            currentTask.callback = null;
            const didUserCallbackTimeout = currentTask.expirationTime +    if (currentTask !== null) {
+       return true;
+   } else {
+       const firstTimer = peek(timerQueue);
+       if (firstTimer !== null) {
+           requestHostTimeout(handleTimeout, firstTimer.startTime - currentTime);
+       }
+       return false;
+   }
}

export {
    scheduleCallback,
    shouldYield,
    ImmediatePriority,
    UserBlockingPriority,
    NormalPriority,
    IdlePriority,
    LowPriority
}
```

7.取消任务

7.1 src\index.js

src\index.js

```
import {
  scheduleCallback,
  shouldYield,
  ImmediatePriority,//-1
  UserBlockingPriority,//250
  NormalPriority,//5000
  LowPriority,//10000
  IdlePriority,//1073741823
+ cancelCallback
} from "./scheduler";
let result = 0;
let i = 0;
/**
 * 总任务
 * @returns
 */
function calculate(didTimeout) {
  for (; i < 10000000 && (!shouldYield() || didTimeout); i++) {//7个0
    result += 1;
  }
  if (result < 10000000) {
    return calculate;
  } else {
    console.log('result', result);
    return null;
  }
}

let result2 = 0;
let i2 = 0;
console.time('cost');
function calculate2(didTimeout) {
  if (i2
    console.timeEnd('cost');
  for (; i2 < 10000000 && (!shouldYield() || didTimeout); i2++) {
    result2 += 1;
  }
  if (result2 < 10000000) {
    return calculate2;
  } else {
    console.log('result2', result2);
    return null;
  }
}

scheduleCallback(ImmediatePriority, calculate);//-1
+const task = scheduleCallback(LowPriority, calculate2, { delay: 10000 });//10000
+cancelCallback(task);
```

7.2 Scheduler.js

src\scheduler\src\Scheduler.js

```
import { requestHostCallback, shouldYieldToHost as shouldYield, getCurrentTime, requestHostTimeout } from './SchedulerHostConfig';
import { push, pop, peek } from './SchedulerMinHeap';
import { ImmediatePriority, UserBlockingPriority, NormalPriority, LowPriority, IdlePriority } from './SchedulerPriorities';
// 不同优先级对应的不同的任务过期时间间隔
let maxSigned31BitInt = 1073741823;
let IMMEDIATE_PRIORITY_TIMEOUT = -1;//立即执行的优先级，级别最高
let USER_BLOCKING_PRIORITY_TIMEOUT = 250;//用户阻塞级别的优先级
let NORMAL_PRIORITY_TIMEOUT = 5000;//正常的优先级
let LOW_PRIORITY_TIMEOUT = 10000;//较低的优先级
let IDLE_PRIORITY_TIMEOUT = maxSigned31BitInt;//优先级最低，表示任务可以闲置
//已经可以开始执行队列
let taskQueue = [];
//尚未可以开始执行队列
let timerQueue = [];
let currentTask;
//下一个任务ID编号
let taskIdCounter = 1;
/**
 * 调度一个工作
 * @param {*} callback 要执行的工作
 */
function scheduleCallback(priorityLevel, callback, options) {
    // 获取当前时间，它是计算任务开始时间、过期时间和判断任务是否过期的依据
    let currentTime = getCurrentTime();
    // 确定任务开始时间
    let startTime;
    if (typeof options
        var delay = options.delay;
        if (typeof delay
            startTime = currentTime + delay;
        } else {
            startTime = currentTime;
        }
    } else {
        startTime = currentTime;
    }
    // 计算过期时间
    let timeout;
    switch (priorityLevel) {
        case ImmediatePriority://1
            timeout = IMMEDIATE_PRIORITY_TIMEOUT;//-1
            break;
        case UserBlockingPriority://2
            timeout = USER_BLOCKING_PRIORITY_TIMEOUT;//250
            break;
        case IdlePriority://5
            timeout = IDLE_PRIORITY_TIMEOUT;//1073741823
            break;
        case LowPriority://4
            timeout = LOW_PRIORITY_TIMEOUT;//10000
            break;
        case NormalPriority://3
        default:
            timeout = NORMAL_PRIORITY_TIMEOUT;//5000
            break;
    }
    //计算超时时间
    let expirationTime = startTime + timeout;
    //创建新任务
    let newTask = {
        id: taskIdCounter++,//任务ID
        callback,//真正的任务函数
        priorityLevel,//任务优先级，参与计算任务过期时间
        startTime,
        expirationTime,//表示任务何时过期，影响它在taskQueue中的排序
        //为小顶堆的队列提供排序依据
        sort
    };
    if (startTime > currentTime) {
        newTask.sortIndex = startTime;
        push(timerQueue, newTask);
        if (peek(taskQueue)
            requestHostTimeout(handleTimeout, startTime - currentTime);
        }
    } else {
        newTask.sortIndex = expirationTime;
        //把此工作添加到任务队列中
        push(taskQueue, newTask);
        //开始调度flushWork
        requestHostCallback(flushWork);
    }
+   return newTask;
}
/**
 * 处理超时任务
 * @param {*} currentTime
 */
function handleTimeout(currentTime) {
    advanceTimers(currentTime);
    if (peek(taskQueue) !== null) {
        requestHostCallback(flushWork);
    } else {
        const firstTimer = peek(timerQueue);
        if (firstTimer !== null) {
            requestHostTimeout(handleTimeout, firstTimer.startTime - currentTime);
        }
    }
}
function advanceTimers(currentTime) {
    let timer = peek(timerQueue);
    while (timer !== null) {
        if (timer.callback
            pop(timerQueue);
        } else if (timer.startTime  currentTime && shouldYield()) {
            break;
        }
        //执行当前的工作
        //const continuationCallback = currentTask();
        const callback = currentTask.callback;
        if (typeof callback
            currentTask.callback = null;
            const didUserCallbackTimeout = currentTask.expirationTime +export function cancelCallback(task) {
+    //清空回调以表示任务已取消
+    //无法从队列中删除，因为无法从基于数组的堆中删除任意节点，只能删除第一个节点
+    task.callback = null;
+}
export {
    scheduleCallback,
    shouldYield,
    ImmediatePriority,
    UserBlockingPriority,
    NormalPriority,
    IdlePriority,
    LowPriority,
+   cancelCallback
}
```