第二周 - 线程池和任务队列

第二周 - 线程池和任务队列

队列的基础知识 队列的典型应用场景 经典面试题

LeetCode #86 分隔链表

LeetCode #138 复制带随机指针的链表

LeetCode #622 设计循环队列

LeetCode #641 设计双端循环队列

LeetCode #1670 设计前中后队列

LeetCode #933 最近请求次数

LeetCode #面试题 17.09 第K个数

LeetCode #859 亲密字符串

LeetCode #860 柠檬水找零

LeetCode #969 煎饼排序

LeetCode #621 任务调度

队列的基础知识

队列是连续的存储区,可以存储一系列的元素。是FIFO(先入先出,First-In-First-Out)结构。

队列通常具有头尾指针(左闭右开区间),头指针指向第一个元素,尾指针指向最后一个元素的下一位。

队列支持(从队尾)入队 (enqueue)、(从队首)出队 (dequeue)操作。

循环队列可以通过取模操作更充分地利用空间。

队列的典型应用场景

- CPU的超线程技术
- 线程池的任务队列

经典面试题

LeetCode #86 分隔链表

使用两个链表,一个用于插入小于x的元素,一个用于插入大于等于x的元素,最后合并两个链表即可。

LeetCode #138 复制带随机指针的链表

难点在于复制随机指针。

这里可以使用一个小技巧对节点进行复制:

将原本的 $A \to B \to C$ 复制成 $A \to A' \to B \to B' \to C \to C'$ 。

然后将复制节点中的随机指针域向后推进一格,这样复制节点的随机指针域,就指向了随机指针的复制节点。

最后将复制的节点拆下来即可。

LeetCode #622 设计循环队列 LeetCode #641 设计双端循环队列

按照逻辑实现即可。

LeetCode #1670 设计前中后队列

为便于后续操作, 首先实现一个基于双向链表的双端队列。

使用两个双端队列,一个存<mark>放前半</mark>部分元素,一个存放后半部分元素。通过维护两个队列首尾节点的方式,平衡两个队列的元素数量,使得中间的元素始终处于固定的位置(如维持在第二个链表头部、或第一个链表尾部)。

LeetCode #933 最近请求次数

使用队列对过程进行模拟。不断弹出队首的过期元素,然后返回队列大小即可。

LeetCode #面试题 17.09 第K个数

先考虑一个问题。对于初始状态{1},我们应该怎么获得后续元素呢? 我们可以将{1}直接从数组中弹出,然后将他的3倍、5倍、7倍分别加入数组。

然后我们选择{3,5,7}中的最小值3,将3从数组中弹出,将3的3倍、5倍、7倍分别加入数组。重复上述过程,直到我们得到第K个数。

由于我们每次都需要获得数组的最小值,因此能动态维护当前最值的优先队列(即最大/小堆)结构可以对此过程进行优化,此时的时间复杂度是 $O(n \lg n)$ 。

这样结束了吗? 我们不妨继续模拟一下这个过程。

注意带下划线的元素,这些是当前轮次中被新添加的元素。

 $\{1\} \rightarrow \{3, 5, 7\} \rightarrow \{5, 7, 9, 15, 21\} \rightarrow \{7, 9, 15, 15, 21, 25, 35\}$

在第3轮处理的时候,出现了重复元素15 (5×3) ,而在之前也得到过这个元素 (3×5) 。

解决重复元素的方案也很简单。

如果当前正在处理的元素不包含因子3,那么它的3倍就无需被加入队列。 如果当前正在处理的元素不包含因子3和5,那么它的3倍和5倍就无需被加入队列。

因为这些元素的3倍/5倍一定在之前被其他元素的倍率得到过,可以通过 质因数分解的表示形式对此进行证明。

由于在过程中遍历了所有数组中元素的3、5、7倍并选择了当前的最小值,因此这个做法是不重复、不遗漏的。

分析到这里其实已经可以AC这道题了。但我们还可以想一下是否可以继续 优化。

能否优化掉这个lg n呢?

这个lg n来自于优先队列的维护过程。实际上我们的枚举过程本身是有序的,而被弹出的元素的3倍、5倍、7倍中,我们并不一定需要把5倍和7倍也提前加入队列参与这个排序过程。因为5倍值和7倍值之间,大概率还存在着其他元素的3倍值和5倍值。

我们可以使用3个指针p3, p5, p7来指向我们结果数组中的元素。指针p3指向的值, 其3倍在当前循环结束后加入结果数组; p5, p7同理。

循环的每一轮, $3 \times ans[p3]$, $5 \times ans[p5]$, $7 \times ans[p7]$ 中的最小值将被加入结果数组,然后对应的指针向前推进一格,直到结果数组中有K个数。

对于不重复的证明是显然的,在推进过程中,每次循环结束后,加入数组中的值一定严格小于指针正在指向的值的倍率,因此数组是严格单调递增的——3个if中,可能有两个是同时成立的,这导致了p3, p5, p7可能同时被推进一格。

对于不遗漏的严格证明,细节较为繁多,在此不做赘述。由于每个元素都是由数组内的元素的倍率生成的,我们可以通过考虑指针推进的过程,来 直观得到这一结论。

LeetCode #859 亲密字符串

LeetCode #860 柠檬水找零

分情况讨论即可,注意不要漏掉样例中给出的边缘情况。

LeetCode #969 煎饼排序

每次将第N大的元素先翻转到第1位,再翻转到第N位,这样第N位就无需在后续进程中再进行处理,只需要考虑前N-1位即可。

由于每个元素只需要2次翻转即可归位,因此所需的次数最多只需2N次,符合题目需求。

对于这种做法,可行的优化主要有两个。

一是可以去除值为"1"的翻转(值为"1"的翻转相当于未操作);二是可以跳过已经在正确位置上的元素。

对于煎饼排序的最优解,在《编程之美》中有更加详细的讨论,感兴趣的同学可以自行阅读。

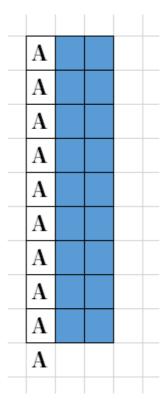
LeetCode #621 任务调度

首先考虑这样一个问题,对于存在冷却时间的情况,任务调度最少要花费 多少的空间?

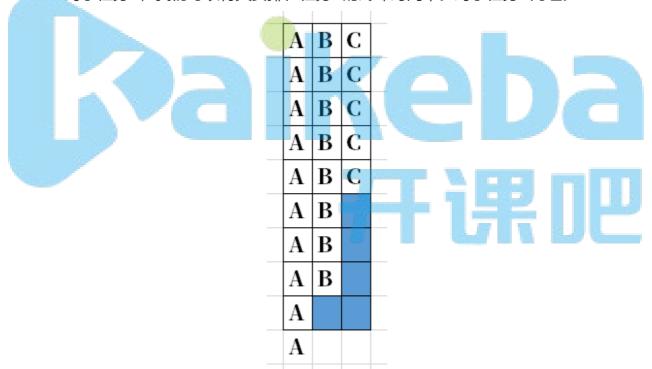
我们首先对任务进行分类,根据任务出现的次数对任务进行排序。

以{"A": 10, "B": 8, "C": 5, "D": 3, "E": 1}, 且冷却时间为2为例。

无论如何安排,任务A都至少要花费9 * (1 + n) + 1的时间,原因如图所示(带有黑色边框的部分是无法节省的时间部分,蓝色部分为冷却时间)。



对于任务B, 我们可以将其安排在任务A的冷却时间中。对于任务C同理。



由于冷却时间n=2,因此接在C后面的元素,在右边竖着接下去的时候,由于同名任务相隔距离大于2,就不再受冷却的限制,可以理解为可以任意排布。

A	В	C	D	E	
A	В	C	D		
A	В	C	D		
A	В	C			
A	В	C			
A	В				
A	В				
A	В				
A					
A					

但是我们注意到,任务的安排还可以再优化。黄色部分的内容可以按顺序 地接在蓝色部分,让冷却时间得到利用。

因此,可以得到如下计算式:

最短调度时间 = 白色部分 + 红色部分 + max(0, (黄色部分 - 蓝色部分))

开课吧