用于列车销售的可线性化并发数据结构

性能评测报告

黄成 201828013229149

一、设计思路

1.并发数据结构设计思路

该并发数据结构用于模拟简化的并发列车车票销售系统，涉及的基本信息包括列车数(route)、每趟列车车厢数(coach)、每个车厢座位数(seat)，车站数(station)，并发线程数(thread)。每张车票(ticket)包含的信息有车票编号(tid)、乘客姓名(passenger)、车次、车厢、座位、出发站、到达站。该系统具有以下特点：

(1) 所有列车的车厢、座位相同；

(2) 所有列车均单向发车，从第一个车站单程发往最后一个车站；

(3) 每张车票的 tid 全局唯一；

(4) 支持退票。

基于以上规则与特点，设计如下 TicketingDS 类：

```
public class TicketingDS implements TicketingSystem {
    private int routeNum;
    private int coachNum;
    private int seatNum;
    private int stationNum;
    private int threadNum;

    private AtomicBoolean[][][] isSeatLocked; // a lock per seat for buy and refund
    private BitSet[][][] isSeatSold; // whether basic sections of a seat is sold or not
    private AtomicLong[] ticketIds; // ticket id in a route, to calculate global id
    private ArrayList<ArrayList<ArrayList<Ticket>>> soldTickets; // a sold ticket list for each coach

    public TicketingDS(int routenum, int coachnum, int seatnum, int stationnum, int threadnum) { ... }
    @Override
    public Ticket buyTicket(String passenger, int route, int departure, int arrival) { ... }
    @Override
    public int inquiry(int route, int departure, int arrival) { ... }
    @Override
    public boolean refundTicket(Ticket ticket) { ... }
}
```

其中，

(1) int routeNum, coachNum, seatNum, stationNum, threadNum 用来存储基本信息；

(2) AtomicBoolean isSeatLocked 标记每个座位是否正在被购票/退票，相当于锁；

(3) BitSet[][][] isSeatSold 记录每个座位在每个区间是否被售出，每个座位一个 bitmap，售出时对应的位为 1，否则为 0；

(4) AtomicLong[] ticketIds 记录每趟列车内部唯一的车票编号，可根据它得到一个全局唯一的编号；

(5) ArrayList<ArrayList<ArrayList<Ticket>>> soldTickets 记录每趟列车的每个车厢中已售出车票的列表，售出时添加，退票时删除；

(6) 方法包括一个构造方法和 TicketingSystem 接口的 3 个方法。

几个方法的具体实现思路如下：

(1) buyTicket 方法：

Step 1: 对于指定车次，无锁遍历每个车厢的每个座位，测试是否可用（即指定的出发

站到到达站之间是否都未售出）；

Step 2: 如果可用，尝试使用 CAS 对指定座位加锁（乐观锁）。加锁成功则继续，否则进入 step 6；

Step 3: 加锁成功后再次检查该座位，仍然可用则继续，否则放弃座位，继续 step 1；

Step 4: 将该座位从指定出发站到到达站之间标记为已售出，释放锁；

Step 5: 生成车票，加入对应车次的对应车厢的已售出列表，并返回车票；

对于 step 2 中加锁失败的座位：

Step 6: 把这个座位记录在一个表中，继续 Step1；如果所有座位遍历后仍然不能出票，则尝试从这个表中出票；

Step 7: 如果最终无法出票，返回 null。

(2) inquiry 方法：无锁遍历每个车厢的每个座位，测试是否可用，可用时计数器+1；返回最终得到的数值。

(3) refundTicket 方法：

Step 1: 检查对应车次的对应车厢的已售出列表是否包含这张票，如果包含则将它删除并继续，否则返回 false；

Step 2: 对对应座位加锁，将出发站到到达站之间改为未售出，释放锁，并返回 true。

2.程序测试方法

(1) 单线程简单测试：购买 1000 张票，然后全部退票。

(2) 单独的多线程购票测试、查询测试、购票和退票(1:1)测试、购票和查询(1:1)测试。

(3) 不同线程数、不同操作数量下查询、购票和退票(6:3:1)混合测试。

二、系统分析

1.正确性分析

没有想到比较严格地测试系统正确性的方法。

从逻辑上暂时没有发现漏洞：

(1) 不上溢的情况下，每趟列车每次出票 tid 应该是单调递增的等差数列，公差是每趟列车的座位数（同时保证了唯一性和一次性）；

(2) 区段有余票时一定能被遍历检查到并出票（即使加锁失败也会一直重复尝试，除非检查到区段已被售出）；

(3) 区段无余票时，遍历不会返回任何座位；

(4) 系统静态时，查询计数不会遗漏或重复；

(5) 保证对已售出列表的读写操作互斥的条件下，所有的无效票（包括重复退同一张票）都不会出现在列表中。

从已有的测试程序输出看，没有出现退错票的信息，购票、查询和退票结果看起来也没有异常。

方法分析：

(1) buyTicket 方法中虽然没有使用显式的锁，而使用了原子类型的变量和 CAS 原语，但实际产生了使用锁的效果。该方法是 deadlock-free 的，一个线程有可能发现了某个座位可用，但是 CAS 操作一直失败导致一直等待；不过整个系统是有进展的，至少有一个线程 CAS 成功。

该方法是否可线性化仍然存疑：如果对于出票的座位，该方法必然是可线性化的，线性化点在 CAS 操作成功之后；但是从整个方法来看，假如在遍历过程中，i)已经遍历过的座位

s1 有退票，ii)s1 退票完成后，尚未遍历的座位 s2 有退票，iii)最终买到了 s2 的票，这样 refund(s1)和 refund(s2)有了偏序关系，无法插入购票的线性化点。

(2) inquiry 方法是 wait-free 的，无锁遍历所有座位后完成。

该方法不可线性化，只满足静态一致性。在购票/退票的同时查询，无法保证结果的正确性。

(3) refundTicket 方法也使用原子变量和 CAS 原语实现了锁的功能，是 deadlock-free 的。

该方法可线性化，可线性化点在 CAS 成功之后。

2.性能分析

(1) 单独测试

测试参数：routeNum = 10, coachNum = 24, seatNum = 200, stationNum = 60，每个线程操作数：10k，线程数：16。

测试结果：

i) inquiry: 0.4~0.6 Mops/sec

ii) buy: 0.2~0.3 Mops/sec

iii) buy and refund (1:1): 0.6~0.9 Mops/sec

iv) buy and inquiry (1:1): 0.55~0.65 Mops/sec

结果分析：

可以发现，三种操作中速度最快的是退票操作，因为每个车厢维护了一个列表，车厢的座位数有限，因此在这个列表上查询和删除都很快，同时也减少的冲突；其次是查询操作，需要遍历所有座位；最慢的是购票操作，可能是因为线程间的冲突较为频繁。

退票操作本身速度较快，在混合测试的场景下发生的比例又最低，因此不再对它进行优化。可以尝试使用更细粒度的已购车票列表，或者使用 CLH/MCS 锁代替 TTAS 锁。

查询操作简单，基本没有可以优化的空间。尝试使用多线程查询，但结果导致了性能降低。猜想这是由于查询本身开销不大，多线程反而增加了创建线程和线程同步的开销。

对于购票操作，首先尝试使用更细粒度的 tid 生成机制，但测试后发现没有明显的效果。使用多线程可能是有效的，但是涉及比较复杂的同步机制，鉴于时间和精力有限，没有继续。

(2) 混合测试

测试参数：routeNum = 10, coachNum = 24, seatNum = 200, stationNum = 60，动态改变每个线程的操作数和线程数。

测试结果见下页图，不同的线表示不同的每个线程的操作数。

图 1 为平均延迟与线程数的关系，随着线程数增加，平均延迟呈上升趋势；每个线程的操作数越多，上升约缓慢。图 2 位吞吐量与线程数的关系，随着线程数增加，吞吐量呈上升趋势；每个线程的操作数越多，上升越明显。

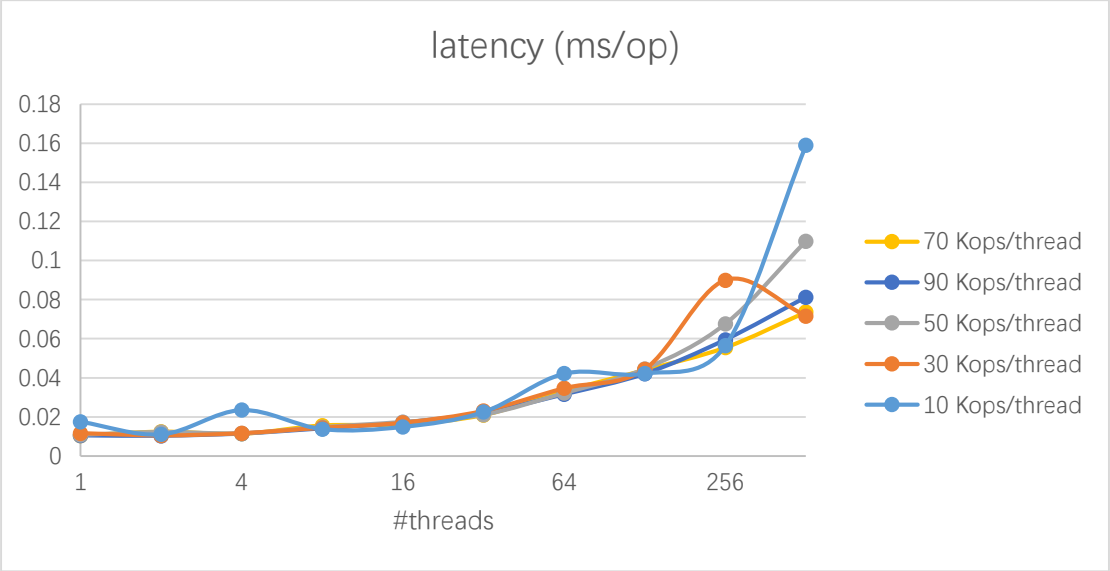

图 1 平均延迟-线程数

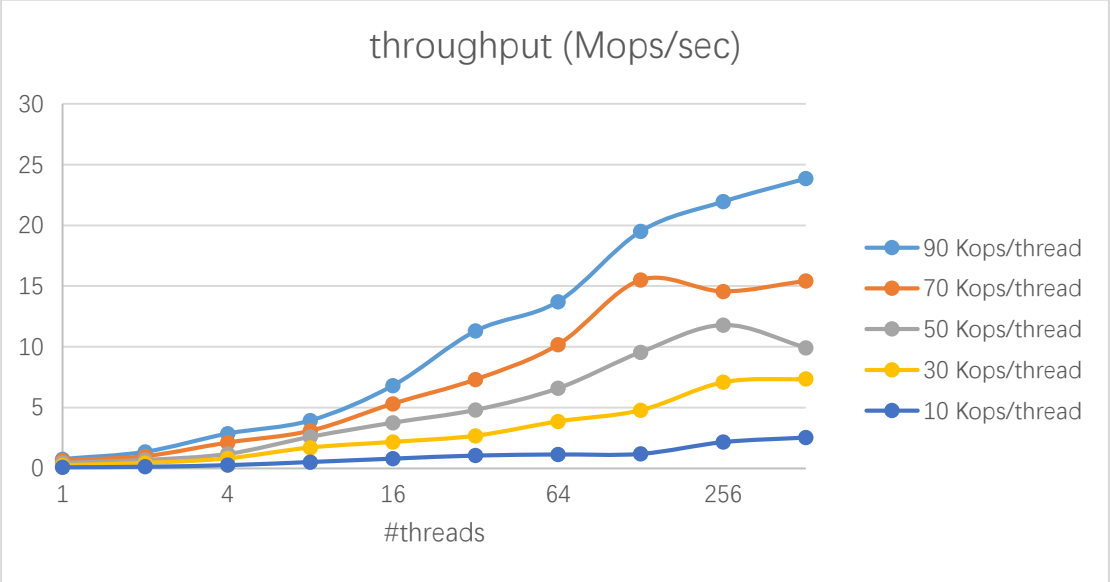

图 2 吞吐量-线程数