# 用于列车售票的可线性化并发数据结构

潘浩彬
2016281013229093

2016 年 12 月 20 日

## 数据结构与测试程序说明

### 1. 并发数据结构与算法设计

这部分的代码在 performance_test_src 文件夹中。

首先是数据结构的设计，根据题目我们可以获取到几个关键的概念，如票、车次、出发站、终止站、车厢、座位。由于票是需要返回的，因此需要为票设计一个专门的类 Ticket（如下）。

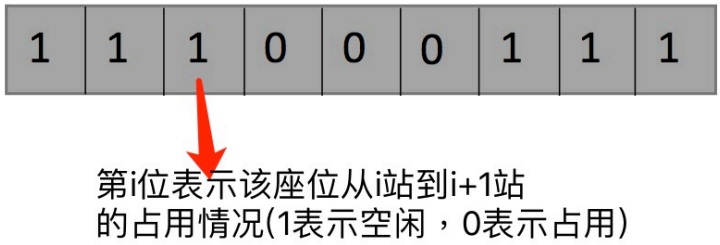
```
class Ticket {
    //原子计数器 用于生成tid
    static AtomicInteger counter = new AtomicInteger(0);

    long tid;           // 票的tid
    String passenger; // 乘客名字
    int route;          // 车次
    int coach;          // 车厢号
    int seat;           // 座位号
    int departure;      // 起始站
    int arrival;        // 终止站

    // 0 表示退票后的废票  1表示成功的购票  2表示失败的购票
    int type;           // 主要用于正确性的检验
    int timeStamp;      // 线性化点的时间戳
}
```

为了表示方便，我们将车次也设计成一个 Route 类，这个类里面需要保存的信息有该车次的车站数、车厢数、每个车厢的座位数等信息。这个数据结构如下设计:

```
class Route {
```

```
    SeatPair[] seats;           // 该车次的座位列表
    int station;                // 该车次的车站数目
    int coach_per_route;        // 每个车次的车厢数
    int seat_per_coach;         // 每个车厢的座位数
}
```

除此之外，由于题目要求我们需要完成购票、退票和查询操作，而每张票对应到每个车次从 A 到 B 站的 C 车厢 D 座位信息，也就是说调用方法的时候，我们需要保存好座位的占用情况，以便下次调用时能记住那些座位在哪个区间是否已经售出。而这里为了提高并发性，将每个座位的区间占用编码成为一个整型的数字, 具体从高位往低位的顺序，第 i 位表示某个座位从第 i 站到第 i+1 站的占用情况, 如下图所示:



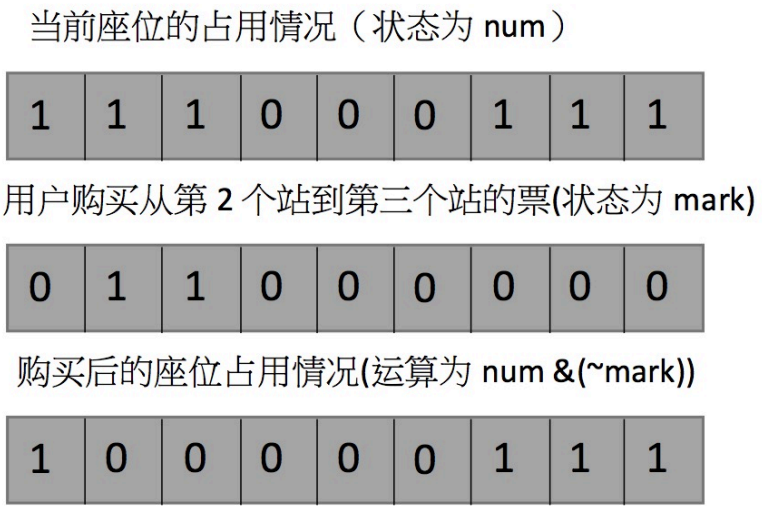

因此可以给座位设计一个数据结构:

```
class SeatPair {
    int coachId;            //座位的车厢id
    int seatId;             //座位的座位id
    // 座位的占用情况(使用int的前k位来编码)
    AtomicInteger info;
    // 对于验证程序，因为要保证线性化点的前后顺序
    // 采用了reference+timestamp的方式
    // AtomicStampedReference<Integer> info;
}
```

最后在原有的 TicketingDS 的接口中加入成员变量，表示有多趟车次，到此数据结构设计完毕。数据结构设计的重点在于考虑并发性的要求，希望在购票和退票的时候能够避免粗粒度所来实现互斥，因此更多考虑在调用方法时，特定方法只修正对应的座位，而不会影响其他座位。因此才有座位的编码方法，这样能保证每次退购票只要修改找到的座位的 info 域。

说完了数据结构的设计，接下来就主要分析 Inquiry 方法、buyTicket 方法以及 refundTicket 方法的设计思路，当然单线程的逻辑跟多线程的逻辑很相近，唯一不同的是如何保证原子的

修改，并且保证正确性。在解释对应代码之前，我们首先来看看，如果通过修改座位的位编码实现购票或者退票后的状态迁移。如下图所示，原先座位编码为 info[111000111], 其中中间三个区间被占用了，这时候如果用户希望能够订下这个位置的前三个区间，于是产生一个 num 为 [111000000], 表示希望购买该位置的前三个区间，于是只需要简单的做一个位操作 [info&(~num)], 购买成功后，即可以将该座位的状态更新为 [000000111], 这样就是一个正确的购买行为。




同理对于退票操作来说，假设原先座位编码为 info[111000111], 其中中间三个区间被 userA 购买了，这时候如果 userA 希望能够退掉这个座位的中间那三个区间的票，于是产生一个 num 为 [000111000], 希望退掉该票，这时只需要简单做一个位操作 [info|num], 即可以将原来的 0 变为 1，退票成功后，即可以将该座位的状态更新为 [111111111], 这样就是一个正确的退票行为。

最后是查询剩余票的操作，查询操作的话，只需要遍历所有的座位，查看查询区间跟该座位是否有区间冲突就可以了，具体方法是，假设座位状态为 info[111000111], 用户要查询的是区间 4 到 8，于是产生了 num[000111100], 如果条件 [info&num == num] 成立，即说明区间没重复，满足需求。

首先是 Inquiry 方法，这里的 Inquiry 方法并没有采用粗略式的估计，其时间复杂度跟座位总数相关。主要是思路是遍历查询车次的所有座位，同时设计一个计数器。如果发现座位满足要求，那么计数器加一，最后返回计数器的值。

```
1    @Override
2    public int inquiry(int route, int departure, int arrival) {
3        // 获取车次的站数
4        int sta = routes[route].station;
5        // 计算出用户区间的mark, 即num
6        int num = calIntervalNum(departure, arrival, sta);
```

```
7
8           SeatPair[] sp = routes[route].seats;
9           int cnt = 0;
10          // 遍历该车次的所有座位
11          for (int i = 0; i < sp.length; i++) {
12              // 检查该区间是否可用
13              if ((sp[i].info.get() & num) == num) {
14                  cnt++;
15              }
16          }
17          return cnt;
18      }
```

接下来是 buyTicket 方法, 购票方法接受乘客名，车次，起始站以及终止站作为参数，同样采用遍历该车次的所有座位来寻找满足要求的座位，然后通过 CAS 操作对座位对应的 info 信息进行修改。这里可能有种情况，就是 CAS 失败，一般是因为有别的线程抢先修改了 info，那么我采用的策略是重新获得该 info 值，并判断是否仍然符合要求。这里每次都一定能保证一个线程成功订到票，因此不会出现问题。

```
1       @Override
2       public Ticket buyTicket(String passenger,
3               int route, int departure, int arrival) {
4           SeatPair[] sp = routes[route].seats;
5           int sta = routes[route].station;
6           // 计算用户购买区间的mark, 即num
7           int num = calIntervalNum(departure, arrival, sta);
8
9           // true表示购票成功, false为失败
10          boolean flag = false;
11
12          // 遍历该车次的所有座位
13          for (int i = 0; i < sp.length; i++) {
14              // 获取当前座位的状态
15              int val = sp[i].info.get();
16              // 如果flag为false, 并且该座位满足区间要求
17              while(!flag && (val&num) == num) {
18                  // 如果CAS成功, 则此处为线性化点。
19                  flag = sp[i].info.compareAndSet(val, val&(~num));
```

```
20                  // 如果CAS失败，重新获取座位状态，然后重购
21                  val = sp[i].info.get();
22              }
23              // 购票成功，返回已购买的票
24              if (flag) {
25                  Ticket res = new Ticket(passenger,
26                          route, departure, arrival, sp[i]);
27                  return res;
28              }
29          }
30          // 没可用的票，返回空
31          return null;
32      }
```

最后是退票方法，购票方法接受 Ticket 作为参数，通过 Ticket 信息很容易就可以定位到对应车次上的特定座位，然后通过 CAS 操作对座位对应的 info 信息进行修改。这里同样可能有种情况，就是 CAS 失败，一般是因为有别的线程抢先修改了 info，那么我采用的策略是重新获得该 info 值，并判断是否仍然符合要求。这里每次都一定能保证一个线程成功退到票，因此不会出现问题。

```
1       @Override
2       public boolean refundTicket(Ticket ticket) {
3           int route = ticket.route;
4           int coach = ticket.coach;
5           int seat = ticket.seat;
6           int dep = ticket.departure;
7           int arr = ticket.arrival;
8
9           Route rou = routes[route];
10
11          int stn = rou.station;
12          // 找到票信息对应的位置
13          int sid = (coach-1) * rou.seat_per_coach + seat-1;
14          SeatPair[] sp = rou.seats;
15          boolean flag = false;
16          int val = sp[sid].info.get();
17          // 计算用户退票区间的mark，即num
18          int num = calIntervalNum(dep, arr, stn);
```

```
19          // 满足区间要求，则执行CAS，修改座位info
20          while (!flag && (val & num) == 0) {
21              flag = sp[sid].info.compareAndSet(val, val|num);
22              // 重新获取座位状态
23              val = sp[sid].info.get();
24          }
25          return flag;
26      }
```

## 2. 多线程测试程序设计

测试的代码在 validation_src 文件夹中。测试程序主要作用是验证多线程的正确性，因此在保证了单线程的正确性下，考虑如下问题：

1. 每张车票都有一个唯一的编号 tid, 且不能重复。
2. 每一个 tid 的车票只能出售一次，退票后，原车票的 tid 作废。
3. 每个区段有余票时，系统必须满足该区段的购票请求。
4. 车票不能超卖, 系统不能卖无座车票。

首先应该明确，通过对 log 进行分析来验证正确性存在一定难度，因为打 log 时，不同线程的 log 会覆盖，因此很难进行分析。因此我们希望能在主程序中对多线程的结果进行验证，于是我采用的思路是将多线程过程中的所有购票、退票记录全部用一个 ArrayList 保存起来, 可以看到在最开始的 Ticket 数据结构中，我预留了一个 type 来表示这三类票，以便验证。记录下来之后，等多线程的程序跑完后，通过调用 Analyse 方法对这个 Ticket 的 list 进行分析，来判断是否存在上述的几个问题。而这里我们采用另外的算法来验证，通过维护一个 test 五维数组来记录每个区间每个位置的售出情况，然后一次判断每次购买和退票是否有效。ticket id 的话是最好验证的，只要通过 HashSet 来判重就可以了。具体 Analyse 逻辑分三块：

1. tid 判重 (解决了第一个问题和第二个问题的验证，由于 tid 是递增的，因此不会用回退票的 tid)

```
1   // HashSet存储tid
2   HashSet<Long> idset = new HashSet<Long>();
3   for (int i = 0; i < history.size(); i++) {
4       if (history.get(i).type == 1) {
5           Long tid = history.get(i).tid;
6           // 如果tid已存在则报错
7           if (idset.contains(tid)) {
```

```
                System.out.println("出现重复的tid!");
            } else {
                idset.add(tid);
            }
        }
    }
```

2. 每个区段有余票时，系统必须满足该区段的购票请求。这一块不好验证，因为把操作串行化之后，会使得原本有交叠的方法调用变成串行执行，因而如果在购票过程中，原本无余票，这时有个退票方法进来，退了一张票，但是购票线程并没有看到，返回了 null，这个过程其实也是合理的，因为购票的线性化点可以在退票之前。因此这一块的验证可能存在一些问题, 所以代码中这一块注释掉了。

```
if (k.type == 2){       // 购买失败的检验
    // 看看route车次dep到arr是否有座位
    for (int j = 1; j <= coachnum; j++) {
        for (int l = 1; l <= seatnum; l++) {
            if (test[route][dep][arr][j][l] == false) {
                System.out.println("存在座位，但是购票失败！");
            }
        }
    }
}
```

3. 车票不能超卖, 系统不能卖无座车票。这一块的验证，只要判断购买该票时，该座位是否是可用状态。

```
if (k.type == 1) {  // 购票检验
    // 判断购买的位置是否有人
    if (test[route][dep][arr][coach][seat]) {
        System.out.println("购买的座位被重复购买！");
    } else {
        // 如果该票能成功购买，
        // 需要更新与该票座位一样并且区间有交叠的座位信息。
        for (int j = 1; j < stationnum; j++) {
            for (int l = j+1; l <= stationnum; l++) {
                // 如果存在交叠，座位设置为不可用
                if (l > dep && j < arr) {
```

```
                    test[route][j][l][coach][seat] = true;
                }
            }
        }
    }
}
```

4. 除了上面的问题，我还额外验证了退票的合理性程序

```
if (k.type == 0) { // 退票检验
    if (!test[route][dep][arr][coach][seat]) {
        System.out.println("退了未购买的座位");
    }
    // 成功退票  区间座位更新
    else {
        test[route][dep][arr][coach][seat] = false;
        for (int j = 1; j < stationnum; j++) {
            for (int l = j+1; l <= stationnum; l++) {
                // 判断区间是否有交叠
                if (l > dep && j < arr) {
                    test[route][j][l][coach][seat] = false;
                }
            }
        }
    }
}
```

在讲完上述程序验证逻辑之后，为了能够进行正确性验证，也要求我们对原程序的一些接口进行改动。由于购票和退票的记录要保持跟实际情况相符，因此同一个座位必须严格按照 CAS 成功的先后顺序来进行保存，而原程序仅仅使用了 AtomicInteger 并不能保证前后。因此修改后的程序使用了 AtomicStampedReference<Integer> info 来保存座位的修改信息和时间戳，同时在 Ticket 中引入一个时间戳 ts，每次购票退票之后都会把 CAS 成功的时间戳打到 Ticket 上，这样就能保证记录的有序性，使得我们能够正确验证。而验证前，需要对操作的记录进行排序，根据每个座位时间戳的顺序排序，代码如下:

```
// history是购票记录
Collections.sort(history, new Comparator<Ticket>() {
    @Override
```

```
4       public int compare(Ticket o1, Ticket o2) {
5           //System.out.println("hah");
6           int route_a = o1.route;
7           int route_b = o2.route;
8           int coach_a = o1.coach;
9           int coach_b = o2.coach;
10          int seat_a = o1.seat;
11          int seat_b = o2.seat;
12          int ts_a = o1.timeStamp;
13          int ts_b = o2.timeStamp;
14          // 排列的key先后为:车次, 车厢, 座位, 时间戳。
15          if (route_a > route_b) return 1;
16          else if (route_a < route_b) return -1;
17          else {
18              if (coach_a > coach_b) return 1;
19              else if (coach_a < coach_b) return -1;
20              else {
21                  if (seat_a > seat_b) return 1;
22                  else if (seat_a < seat_b) return -1;
23                  else {
24                      if (ts_a > ts_b) return 1;
25                      else if (ts_a < ts_b) return -1;
26                      else return 0;
27                  }
28              }
29          }
30      }
31  });
```

# 系统正确性验证以及性能测试

## 3. 系统正确性

根据前面的验证程序，我们分别在远程主机上进行了测试，测试结果如下：

```
[student078@panda4 test]$ java Main
4个线程，每个线程执行100000次操作！
执行时间：0.662s
Tid重复检测通过！
退票检测通过！
购票检测通过！
8个线程，每个线程执行100000次操作！
执行时间：0.717s
Tid重复检测通过！
退票检测通过！
购票检测通过！
16个线程，每个线程执行100000次操作！
执行时间：4.466s
Tid重复检测通过！
退票检测通过！
购票检测通过！
32个线程，每个线程执行100000次操作！
执行时间：4.616s
Tid重复检测通过！
退票检测通过！
购票检测通过！
64个线程，每个线程执行100000次操作！
执行时间：26.06s
Tid重复检测通过！
退票检测通过！
购票检测通过！
```

### 4. 性能测试

本程序并没有采用锁来互斥，因此在性能上还是不错的，具体测试结果如下：

```
[student078@panda4 caltime]$ java Main
4个线程，每个线程执行100000次操作！
线程数为：4
有效执行操作数：399999
执行时间：0.925s
吞吐率为：432431 ops

8个线程，每个线程执行100000次操作！
线程数为：8
有效执行操作数：789256
执行时间：1.126s
吞吐率为：700937 ops

16个线程，每个线程执行100000次操作！
线程数为：16
有效执行操作数：1540283
执行时间：0.74s
吞吐率为：2081463 ops

32个线程，每个线程执行100000次操作！
线程数为：32
有效执行操作数：3058700
执行时间：1.748s
吞吐率为：1749828 ops

64个线程，每个线程执行100000次操作！
线程数为：64
有效执行操作数：6088878
执行时间：1.338s
吞吐率为：4550730 ops
```

## 程序理论分析

本程序的正确性已经在前面得到了验证,而现在从可线性化、是否 deadloock-free、starvation-free、lock-free 或 wait-free 这几个角度来分析下程序中的三个方法。

首先对于 Inquiry 方法，对于 Inquiry 方法，其是可线性化的，如果 Inquiry 方法跟其他调用没有过 overlap 的话，那么其返回的结果是可以保证正确的，如果跟 refundTicket 和 buyTicket 方法会产生 overlap，那么 Inquiry 方法的线性化点可以在 refundTicket 和 buyTicket 方法的前后，可以根据具体的结果来找到可线性化点。而且由于 Inquiry 方法采用无锁的方法进行遍历，一定能保持所有调用在有穷步内完成，因此是 deadlock-free、starvation-free 以及 wait-free 的。

对于购票 buyTicket 方法，也是一个可线性化的方法，既能保证无 overlap 的情况下正确运行，又能保证有 overlap 时能找到一个线性化点，即 CAS 操作成功那行，同时方法本身无锁，采用 CAS 操作，能保证每次多个线程竞争一个座位时，总有一个能成功，而其他的会接着竞争下一个位置，因此购票方法是 deadlock-free、starvation-free 以及 lock-free 的。

最后是 refundTicket 方法，可线性化点依然是在 CAS 操作成功的时候，同时，如果有多个线程想退同一个位置，那么 CAS 操作能保证每次有一个线程能成功完成，因此退票方法是 deadlock-free、starvation-free 以及 lock-free 的。