用于列车售票的可线性化并发数据结构

梁梓睿 201928013229078

方案设计

1. 主要的思路：
    - 考虑到在性能测试中，查票操作的比例要远远高于买票和退票，故优先考虑对查票友好的设计方案。给定出发站和目的地，能够想到最快的查票方式就是：*为每个出发地和目的地维护一个原子变量，代表该区间内剩余的座位数，查询的时候就直接返回该变量即可。*接下来的买票和退票都要围绕这一思路而服务。
2. 数据结构解释：
    - `State`类，代表某次班车某个区间的状态；含有两个成员变量`ConcurrentHashMap<Integer, Boolean>[] coach`和`AtomicInteger left`。
        - 首先第一个`coach`变量使用了并发哈希表，`key`代表座位号，`value`规定为`true`，`coach`是一个数组，下标即为该列车的车厢号-1。整个数组变量含义为：某节车厢下，有哪些空闲的座位。例如：第8节车厢的27号座位在当前区间是空闲的，就有`coach[7].get(27) == true`；49号座位在该区间不是空闲的，就有`coach[7].get(49) == null`。
        - 第二个变量`left`即为上面谈到的原子变量，代表该列车在该区间剩余的座位数。
        - 总体代码如下：

```
class State {
    ConcurrentHashMap<Integer, Boolean>[] coach;
    AtomicInteger left;

    State(int coachNum, int seatNum) {
        //coach在代码中从0开始计数
        coach = (ConcurrentHashMap<Integer, Boolean>
[]) new ConcurrentHashMap[coachNum];
        for (int i = 0; i < coachNum; i++) {
            coach[i] = new ConcurrentHashMap<>
(seatNum);
            for (int j = 1; j <= seatNum; j++) {
                coach[i].put(j, true);
            }
        }
        left = new AtomicInteger(coachNum *
seatNum);
    }
}
```

- 另外，我创建了Route类，代表了每次班车；含有成员变量State[][] states，代表该列车在每个区间的状态。如果想要获取该列车departure到arrival的状态，只需读取states[departure][arrival]即可。代码如下：

```
class Route {
    ...
    State[][] states; //states[1][2]: 1->2, states[2][5]: 2->5, 以此类推

    Route(int departure, int stationNum, int routID, int coachNum, int seatNum) {
        ...
        states = new State[stationNum+1][stationNum+1];
        for (int i = 1; i < arrival; i++) {
            for (int j = i + 1; j <= arrival; j++) {
                states[i][j] = new State(coachNum, seatNum);
            }
        }
    }
}
```

3. 查票具体实现：
   - 因为有了上面谈到的State类，查询的操作非常简单，代码如下：

```
    @Override
    public int inquiry(int routeID, int departure, int arrival) {
        //直接读取剩余空闲座位数
        Route route = routes[routeID];
        State s = route.states[departure][arrival];
        return s.left.get();
    }
```

4. 买票的具体实现：
   - 买票的操作稍微比较复杂，假设我们购买某个区间的车票，首先随便挑选该区间中的某个空闲座位，*那么我们要从所有跟该区间有交集的区间中购买该座位*；如果在某个区间购买该座位失败，则要换一个座位重新尝试，如果所有座位都尝试失败，则代表没有空闲的座位，购票失败。
   - 例如，8号列车，一共有8个站，我们要买从第6站到第8站的车票，那么我们要首先从该列车的中的空闲座位随便挑选一个，例如为第6号车厢的27号座位。那么我们要从下列所有区间中都尝试"占领"该座位：(1,7)、(1,8)；(2,7)，(2,8)；(3,7)、(3,8)；(4,7)、(4,8)；(5,7)、(5,8)；(6,7)、(6,8)；(7,8)

     具体如下图所示：
     - 值得注意的是，假设在执行过程中，在占领(5,8)区间的27号座位时，发现在该区间下27号不是空闲的，代表购买27号座位失败。*但是因为我们在之前已经把(1,7)、(1,8)等等区间的27号座位"占领"了，在重新开始换座位尝试之前，我们要把这些区间的27号座位的状态"复原"*，以供其他人购买。
   - 具体代码如下：

```
    @Override
    public Ticket buyTicket(String passenger, int routeID, int departure, int arrival) {
        Route route = routes[routeID];
        State[][] states = route.states;
        //错开买的车厢, 提高吞吐量
        int hash = Thread.currentThread().hashCode();
        int begin = hash % route.coachNum, cur = begin;
        //购票的目标区间, 在循环占座的过程中不会变
        State targetInterval = states[departure][arrival];
        do {
            var map = targetInterval.coach[cur];
            //从当前车厢开始尝试所有车厢内的座位
            for (var entry : map.entrySet()) {
                int seat = entry.getKey();
                if (map.get(seat) == null)
                    //在foreach循环中, entrySet是一份拷贝
                    //也就是说当前循环的键值对可能已经不在map真正的entrySet里
                    //所以要先判断一下是否在当前map里,如果不在就不要浪费时间去循环了
                    continue;
                //从各个区间跟目标区间有交集的区间的map中占领该座位
                boolean flag = true;
                Try: for (int i = 1; i < arrival; i++) {
                    for (int j = Math.max(departure, i) + 1;
                         j <= route.stationNum; j++) {
                        State s = states[i][j];
                        var m = s.coach[cur];
                        if (m.remove(seat) == null) {
                            //目标座位已经被其他线程从这个map中删除了, 占座失败
                            flag = false;
                            //复原之前区间的状态
                            recovery(routeID, departure, arrival,
                                     cur , seat, i, j);
                            break Try;  //换一个座位重新尝试
                        } else {
                            //占座成功,空闲座位减一
                            int prev = s.left.getAndDecrement();
                            ...
                        }
                    }
                }
                if (flag) {
                    //购票成功, 生成车票
                    return ticketGenerator(passenger, routeID, departure,
```

```
                                                arrival, cur,
seat);
                }
            }
            //这节车厢的所有座位都尝试过了，换下一节车厢试试
            cur = (cur + 1) % route.coachNum;
        } while (cur != begin);
        //所有车厢遍历完毕，买不到座位
        return null;
    }
```

5. 退票的具体实现：
    - 整体思路为：从车票中获取班车号，出发站，到达站，车厢号，座位号信息。*然后在所有跟出发站和到达站有交集的区间中，复原该座位的状态*。
    - 对于重复退票，退不存在的票等业务问题的考虑：
        - 我最初的想法是创建一个并发哈希表，存储所有的有效车票，然后退票的时候要先查这个表，但是这样会增加线程之间的竞争程度，并不算非常好的一个方案。
        - 后来我想到，由于每个线程只可能会退自己买到的票，不可能存在某个线程退别人买到的车票，因此我创建一个`ThreadLocal<Set<Long>> tickets`的变量，用于存该线程所有的有效车票的`tid`。在退票的时候，首先查询在`tickets`里是否存在目标车票，如果有，则删除该车票，然后做座位复原工作；如果不存在，则返回`false`。
    - 具体代码如下：

```
public class TicketingDS implements TicketingSystem {
    //用于记录某个线程购买的车票
    public static final ThreadLocal<Set<Long>> tickets =
ThreadLocal.withInitial(HashSet::new);
    ...
    @Override
    public boolean refundTicket(Ticket ticket) {
        if (ticket == null) {
            return false;
        } else {
            //判断是否为有效票
            Set<Long> set = tickets.get();
            boolean isSuccessful =
set.remove(ticket.tid);
            if (isSuccessful) {
                //恢复相应区间的座位空闲情况，剩余座位数
                //票的coach从1开始计数，代码中是从0开始计数，
所以要-1
                recovery(ticket.route, ticket.departure,
ticket.arrival,
                        ticket.coach - 1, ticket.seat,
-1, -1);
            }
            return isSuccessful;
        }
    }
}
```

6. 复原座位状态函数recovery的具体实现：

   - Talk is cheap，here is the code：

```
    /**
     * 复原从departure到arrival之间所有区间目标座位的状态
     * @param routeID 列车ID
     * @param departure 始发站
     * @param arrival 终点站
     * @param targetSeat 目标座位
     * @param failedStart :
     * 如果是占座调用的，则表示占座失败区间的始发站；如果是退票调用的则为-1
     * @param failedEnd :
     * 如果是占座调用的，则表示占座失败区间的终点站；如果是退票调用的则为-1
     */
    private void recovery(int routeID, int departure, int arrival, int targetCoach, int targetSeat, int failedStart, int failedEnd) {
        Route route = routes[routeID];  //目标车次
        State[][] states = route.states;
        for (int i = 1; i < arrival; i++) {
            for (int j = Math.max(departure, i) + 1;
                 j <= route.stationNum; j++) {
                State s = states[i][j];
                var map = s.coach[targetCoach];
                if (i == failedStart && j == failedEnd)
                    //占座调用的recovery，失败区间之前的区间全部复原完毕，直接退出
                    return;
                Boolean flag = map.put(targetSeat, true);
                ...
                int prev = s.left.getAndIncrement();
                ...
            }
        }
    }
```

7. 车票生成函数ticketGenerator的具体实现：

   - 该函数唯一值得说明的一点是如何确定tid，使得其全局唯一并且不重复。起初我的构想是利用原子变量设置一个计数器，通过getAndIncrement()直接赋值给tid。但考虑到tid并不需要严格单调递增，并且使用原子变量会增加线程之间的竞争，最后的想法如下：

     - 利用一个ThreadLocal<Integer> cnt变量记录每个线程的购票次数
     - 将线程ID和cnt以类似字符串拼接的方式拼接起来，即为tid。

     因为线程ID全局唯一，并且每个线程的cnt都是严格单调递增的，因此tid是全局唯一并且不重复，同时有效地减少了竞争。

   - 具体代码如下：

```
public class TicketingDS implements TicketingSystem {
    //用于记录某个线程购买的车票
```

```
    public static final ThreadLocal<Set<Long>> tickets =
        ThreadLocal.withInitial(HashSet::new);
    //每个线程的计数器
    private static final ThreadLocal<Integer> cnt =
        ThreadLocal.withInitial(() -> 1);
    ...
    //生成车票
    private Ticket ticketGenerator(String passenger, int
routeID, int departure, int arrival, int coach, int seat)
{
        //购票次数加1
        int prev = cnt.get();
        cnt.set(prev + 1);
        //将线程ID和当前线程的购票次数拼接在一起当tid
        long tid = getTid(prev);
        //设置车票的其他信息
        ...
        //将票放入有效票集合中
        Set<Long> set = tickets.get();
        set.add(tid);
        return ticket;
    }

    private long getTid(int n) {
        //例如: threadID = 14, n = 27, 那么返回结果就位1427
        long tid = Thread.currentThread().getId();
        int tmp = n;
        while (tmp >= 1) {
            tmp /= 10;
            tid *= 10;
        }
        return tid + n;
    }
}
```

正确性分析

1. 是否会出现以下问题：（1）某一班车的某个区间有两张不同的车票对应到同一个座位上？（2）某个区间卖出的票数大于该列车的总座位数?
    - 不会，因为在买票的时候，每个线程都需要从对应的ConcurrentHashMap中把座位删除掉，如果删除失败，就代表这次尝试是不成功，会换一个座位尝试。出现上述问题可能会出现的唯一情况就是，某个座位被成功占领了，但是没有从ConcurrentHashMap中删除掉。但是在上述中提到，在删除ConcurrentHashMap中某个键值对时，是有锁保护的，因此不会出现这种问题。
    - 为了更进一步验证问题(1)不会出现，测试的时候，我加入了以下代码：

```
public class TicketingDS implements TicketingSystem {
    private static final List<Ticket> validator = new
LinkedList<>();
    private static final Object lock = new Object();
    ...
```

```
    @Override
    public boolean refundTicket(Ticket ticket) {
        if (ticket == null) {
            return false;
        } else {
            //判断是否为有效票
            Set<Long> set = tickets.get();
            boolean isSuccessful =
set.remove(ticket.tid);
            synchronized (lock) {
                for (Ticket t : validator) {
                    if (t.tid == ticket.tid) {
                        validator.remove(t);
                        break;
                    }
            }
            ...
        }
    }
    ...
    //生成车票
    private Ticket ticketGenerator(String passenger, int
routeID, int departure, int arrival, int coach, int seat)
{
        Ticket ticket = new Ticket();
        //车票信息设置
        ...
        synchronized (lock) {
            boolean flag = true;
            for (Ticket t : validator)
                if (t.route == ticket.route &&
                    t.coach == ticket.coach && t.seat ==
ticket.seat) {
                    if (!(t.departure >= ticket.arrival
|| t.arrival <= ticket.departure)) {
                        flag = false;
                        break;
                    }
                }
            if (!flag)
                Singleton.getInstance().errorMsg("...");
        }
        return ticket;
    }
}
```

validator存储所有的有效票，在生成车票的时候，用一个粗粒度的锁锁住validator，然后遍历validator看是否有区间冲突的票出现。经过多次测试，皆没有问题。

2. 是否会出现当没有线程进行买票或者退票的时候，查询的票数不准确?
    - 不会，因为无论是买票还是退票，在操作执行完之前，都会对每个区间的剩余票数进行相应的设置才会返回。而且剩余票数是原子变量，因此保证了并发时候的正确性。

3. 其他的正确性验证：

   - 为了防止不明原因出错，测试的时候，我在buyTicket和recovery函数中加入了以下代码：

     buyTicket：

```
                //占座成功
                int prev = s.left.getAndDecrement();
                if (prev < 0) {
                //座位计数器的值低于0
                Singleton.getInstance().errorMsg(

                    "\n*******************************" +
                    "\nseat less than 0: " +
                    "\nrouteID: " + routeID +
                    "\ncount: " + prev +
                    "\nfrom: " + i + " to: " + j);
            }
```

     revocery：

```
                //将目标座位放回到对应的哈希表中
                Boolean flag = map.put(targetSeat, true);
                 if (flag != null) {
                     //不知道为啥被删除的座位会出现这个map里
                     Singleton.getInstance().errorMsg(

 "\n*******************************" +
                             "\nseat should not appear: "
+
                             "\nrouteID: " + routeID +
                             "\ncoach: "+ (targetCoach+1)
+
                             "\nseat: " + targetSeat +
                             "\nfrom: " + i + " to: " + j
+
                             "\nstate: " + flag);
                 }
                 int prev = s.left.getAndIncrement();
                 if (prev > route.total_seat) {
                     //座位计数器的值超过了总的座位数
                     Singleton.getInstance().errorMsg(

 "\n*******************************" +
                             "\nseat greater than seatNum:
" +
                             "\nrouteID: " + routeID +
                             "\ncount: " + prev + ",
limit: " + route.total_seat +
                             "\nfrom: " + i + " to: " +
j);
             }
```

在多次的测试中，都没有错误信息的输出，因此在一定程度上验证了程序的正确性。（虽然座位计数器的值确实可能会有溢出的问题，但这是因为有买票或者退票的操作在进行，不会造成静态不一致性）

- 如果在多线程执行完毕后，其中没有票卖光或者退票的情况出现，那么我们可以根据其输出的trace文件，单线程重新执行一遍相同的操作，最后两个得出来的`TicketingDS`所有列车的所有区间的剩余票数都应该相同。因此我采用以下方法验证：
  - 调整参数，使得不会出现票卖光和退票的情况出现（因为trace中卖光票和退票操作的输出信息跟其对应的买票操作的输出信息的前后关系不一定跟实际执行中一样）
  - 将多线程执行的结果输出到`log.txt`中，并且将`TicketingDS`序列化成`tds.bin`，
  - 验证程序中，创建一个新的`TicketingDS single`，将`log.txt`的结果读出，单线程执行对应的操作
  - 将`tds.bin`反序列化成`multiple`，与`single`每个区间的剩余票数做对比
  - 代码如下：

    `Trace.java`:

```
public static void main(String[] args) throws
InterruptedException {
    ...
    serialize(tds);
}
private static void serialize(Object obj) {
    File bin = new File("tds.bin");
    try (FileOutputStream fos = new
FileOutputStream(bin);
        ObjectOutputStream oos = new
ObjectOutputStream(fos)) {
        oos.writeObject(obj);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

    `Test.java`:

```
package ticketingsystem;

import java.io.*;

public class Test {
    private final static int threadNum = 96;
    private final static int routeNum = 20;
    private final static int coachNum = 15;
    private final static int seatNum = 100;
    private final static int stationNum = 10;

```

```
12      public static void main(String[] args) throws
    Exception {
13          final String file = "log.txt";
14          try (FileReader fr = new FileReader(new
    File(file));
15               BufferedReader br = new
    BufferedReader(fr)) {
16              String line;
17              final TicketingDS single = new
    TicketingDS(routeNum, coachNum, seatNum, stationNum,
    threadNum);
18              while ((line = br.readLine()) != null) {
19                  String[] info = line.split(" ");
20                  if (info[0].equals("TicketBought"))
    {
21                      int routeID =
    Integer.parseInt(info[3]);
22                      int departure =
    Integer.parseInt(info[5]);
23                      int arrival =
    Integer.parseInt(info[6]);
24                      //单线程执行买票操作
25                      single.buyTicket("", routeID,
    departure, arrival);
26                  }
27              }
28          }
29          //反序列化
30          final TicketingDS multiple =
    (TicketingDS)deSerialize();
31          //逐一比较两者的剩余票数
32          for (int k = 1; k <= routeNum; k++) {
33              for (int i = 1; i < stationNum; i++) {
34                  for (int j = i + 1; j <= stationNum;
    j++) {
35                      int left1 = single.inquiry(k, i,
    j);
36                      int left2 = multiple.inquiry(k,
    i, j);
37                      if (left1 != left2) {
38                          System.out.println("error");
39                      }
40                  }
41              }
42          }
43      }
44      private static Object deSerialize() {
45          Object obj = null;
46          File bin = new File("tds.bin");
47          try (FileInputStream fis = new
    FileInputStream(bin);
48          ObjectInputStream ois = new
    ObjectInputStream(fis)) {
49              obj = ois.readObject();
50          } catch (Exception e) {
```

```
51                 e.printStackTrace();
52             }
53             return obj;
54         }
55     }
```

多次试验证明，两者的剩余票数都一样，一定程度上证明了程序的正确性。

- 以上辅助性的测试都在一定程度上证明了各个方法的可线性化和正确性。
- (测试代码在最终提交版本中都被删除了，如果老师需要看这些代码，请通知我，谢谢！)

性能分析

1. `buyTicket`方法：
    - 该算法的时间复杂度波动比较大，取决于座位的状态。假设一共有S个车站，每节车厢的座位数为N，在最好的情况下，时间复杂度为O(S)；在最坏情况下的时间复杂度为$O(S^2N)$。
    - 在`ConcurrentHashMap`的源码中，删除哈希表中的键值对需要获取table中对应"bucket"的锁；因此在占座的过程中，可能会发生某个线程总是获取不了锁，但是总会有一个线程能够获取锁，因此该方法是lock-free的。
2. `refundTicket`方法：
    - 主要取决于`recovery`方法的复杂度，对于`recovery`方法来说：
        - 该函数的时间复杂度取决于要恢复的区间，假设一共有S个车站，在最好的情况下，时间复杂度为O(S)；在最坏情况下的时间复杂度为$O(S^2)$。
        - 在`ConcurrentHashMap`的源码中，调用`put(key, value)`时，如果key对应的"bucket"为空，那么不需要获取锁。又因为我在初始化时，为每一个座位都分配了一个"bucket"，并且在`recovery`阶段，该座位对应的"bucket"必定为空，所以所有调用该方法的线程都会返回，是wait-free的。
3. `inquiry`方法：
    - 因为直接读取剩余票数即可，故时间复杂度为O(1)。其中并没有涉及到锁的操作，任何一个线程执行该函数最终都肯定可以获得返回值，所以是wait-free的。