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：直接举例子说明。假设一共有20次班车，每班车1500个座位，每班车一共10个站。那么每次班车在初始化阶段，都只有1->10的1500张票。
    - 如果购买了班车1的4-6区间，那么我们就将班车1的1->10的1500张票拆成1->10的1499张票，　1->4的1张票和6->10的1张票。
    - 接着如果想买3->5的一张票，那么我们继续将班车1的1->10的1499张票拆成1->10的1498张票，　1->3的1张票和5->10的1张票。
    - 然后如果我们相查7->10区间的票，那么我们就将出发站小于等于7和终点站大于等于10的票全部加起来，有：1->10的1498张票，5->10的1张票和6->10的1张票，一共1500张票。
    - 然后如果我们想退票4-6区间的那张票，那么我们怎么拆分就怎么合并，将1->4的1张票和6->10的1张票合并成一张“大票”，有1->4的0张票，6->10的0张票和1->10的1499票。
2. 数据结构解释：
    - `Seat`类，代表某次班车某节车厢的某个座位。
        - 关键成员变量：
            - `int bitmap`： 位图，初始化为`0x7fffffff`，表示当前座位在各个区间段的空闲状态。从低位往高位走，如果当前的bit=0表示该区间这个座位被占领，否则是空闲的。例如`bitmap = 0111...10001`，低位的第1位"1"表示1->2是空闲的，第2、3、4位的"0"表示2->3、3->4、4->5的区间是被占领的，以此类推。
            - `AtomicBoolean occupying`：表示当前是否有线程在操作这个座位，包括：查询其状态，占座中，释放座位中，相当于座位的锁。
        - 类方法：
            - `isOccupied(int depature)`，利用位操作查询该座位从departure到departure+1区间是否空闲
            - `take(int departure, int arrival)`，利用位操作在从departure到arrival区间占领该座位
            - `realease(int departure, int arrival)`，利用位操作在从departure到arrival区间释放该座位
        - 总体代码如下：

```
class Seat {
    private int bitmap;
    AtomicBoolean occupying;
    Seat() {
        bitmap = Integer.MAX_VALUE;
        occupying = new AtomicBoolean(false);
    }
    boolean isOccupied(int departure) {
        int res = bitmap & (1 << (departure - 1));
```

```
        return res == 0;
    }
    void take(int departure, int arrival) {
        for (int i = departure; i < arrival; i++)
            bitmap &= ~(1 << (i - 1));
    }
    void release(int departure, int arrival) {
        for (int i = departure; i < arrival; i++)
            bitmap |= (1 << (i - 1));
    }
}
```

- Coach类，代表了某次班车的某节车厢（主要用于加速买票操作）
  - 关键成员变量：
    - `Seat[] seats`：该节车厢下的所有座位对象
    - `AtomicInteger[][] left`：该车厢在某个区间*最少*有多少个空闲座位。例如：`left[2][5].get() == 10`，表示该车厢在2->5区间最少有10个空闲座位。
- Route类，代表了每次班车。
  - 关键成员变量：
    - `Coach[] coaches`：该班车下的所有车厢对象
    - `AtomicInteger[][] left`：同Coach类的成员变量`left`
    - 总体代码如下：

```
    Route (int stationNum, int coachNum, int seatNum) {
        this.stationNum = stationNum;
        this.coachNum = coachNum;
        //初始化
        left = new AtomicInteger[stationNum + 1][stationNum + 1];
        for (int i = 1; i < stationNum; i++) {
            left[i] = new AtomicInteger[stationNum + 1];
            for (int j = i + 1; j <= stationNum; j++)
                left[i][j] = new AtomicInteger(0);
        }
        left[1][10].set(coachNum * seatNum);
        //初始化车厢
        coaches = new Coach[coachNum];
        for (int i = 0; i < coachNum; i++) {
            coaches[i] = new Coach(stationNum, seatNum);
        }
    }
```

3. 查票的具体实现：

- 需要将该班车所有i小于等于departure且j大于等于arrival的`left[i][j]`全部加起来，即为查询结果。因为`left[departure][arrival]`可能只是从某个更大的区间的票中分裂得到了，所以要把包含departure到arrival的所有区间的`left[i][j]`加起来。
- 代码：

```
    @Override
    public int inquiry(int routeID, int departure, int arrival) {
        return inquiry(routes[routeID-1].left, departure, arrival);
    }
    //给定一班列车或一个车厢的剩余座位数组，查询目标区间还剩多少个座位
    private int inquiry(AtomicInteger[][] left,int departure, int arrival)
    {
        int stationNum = left.length-1, res = 0;
        for (int i = 1; i <= departure; i++) {
            for (int j = arrival; j <= stationNum; j++)
                res += left[i][j].get();
        }
        return res;
    }
```

4. 买票的具体实现：总体思路遍历该列车下的所有车厢，尝试占领该车厢下的空闲座位。
   - 第一步：查询当前车厢是否有空闲座位，如果没有则换一节车厢
   - 第二步：通过CAS操作获取某个座位的锁，查询这个座位在目标区间段是否全程空闲，*如果是，则占领该座位，并且进行拆票操作*；如果不是，则换一个座位重新尝试。
   - 总体代码：

```
    @Override
    public Ticket buyTicket(String passenger, int routeID, int departure, int arrival) {
        Route route = routes[routeID-1];
        for (int cur = 0; cur < route.coachNum; cur++) {
            Coach coach = route.coaches[cur];
            //先查一下这个车厢还有没有座位，没了就略过
            if (inquiry(coach.left, departure, arrival) > 0) {
                for (int now = 0; now < coach.seatNum; now++) {
                    Seat seat = coach.seats[now];
                    //获取座位锁
                    if (seat.occupying.compareAndSet(false, true)) {
                        boolean hasFound = true;
                        for (int i = departure; i < arrival; i++) {
                            if (seat.isOccupied(i)) {
```

```
                                //当前座位在目标区间内不全空闲
                                hasFound = false;
                                break;
                            }
                        }
                        if (hasFound) {
                            //OK, 座位可用, 先占领座位
                            seat.take(departure, arrival);
                            //拆票操作
                            spilt(route, coach, seat, departure, arrival);
                            //释放座位锁
                            seat.occupying.set(false);
                            return ticketGenerator(passenger, routeID, departure, arrival, cur, now);
                        } else {
                            //释放座位锁
                            seat.occupying.set(false);
                        }
                    }
                }
            }
        }
        return null;
    }
```

5. 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：

```
    private Ticket ticketGenerator(String passenger, int routeID,
                            int departure,int arrival, int coach, int seat) {
        Ticket ticket = new Ticket();
        int prev = cnt.get();
        cnt.set(prev + 1);
        long tid = getTid(prev);
        ticket.tid = tid;
        ticket.passenger = passenger;
        ticket.route = routeID;
        ticket.coach = coach + 1;
```

```
        ticket.seat = seat + 1;
        ticket.departure = departure;
        ticket.arrival = arrival;
        //加入到有效票集合中
        validTickets.put(tid, ticket);
        return ticket;
    }
```

6. 退票的具体实现：总体思路是从车票中获取班车号，出发站，到达站，车厢号，座位号信息。*复原该座位的状态，并进行合并票操作*。
    - 首先根据车票信息查询有效票集合中是否存在该票，如果不存在，则是无效票，返回退票失败
    - 如果是有效票，则从有效票集合中删除该票，并复原座位状态，然后执行合并票操作
    - 具体代码：

```
    @Override
    public boolean refundTicket(Ticket ticket) {
        if (ticket == null)
            //空票，无效票
            return false;
        Ticket t = validTickets.get(ticket.tid);
        if (t == null) {
            //不存在该tid对应的KV对，无效票
            return false;
        } else {
            //查询两张票的每个域是否相同
            if (isSame(t, ticket)) {
                if (validTickets.remove(ticket.tid) !=
null) {
                    Route r = routes[ticket.route-1];
                    Coach c = r.coaches[ticket.coach-1];
                    Seat s = c.seats[ticket.seat-1];
                    //TTAS 抢座位锁
                    while (true) {
                        while (s.occupying.get()); //wait
                        if
(s.occupying.compareAndSet(false, true))
                            //success
                            break;
                    }
                    s.release(ticket.departure,
ticket.arrival); //释放座位
                    merge(r, c, s, ticket.departure,
ticket.arrival);
                    s.occupying.set(false); //释放锁
                    return true;
                } else  {
                    //有人抢先把票给退了
                    return false;
                }
            } else {
                //同一个tid，但有的域不同，是无效票
```

```
                return false;
            }
        }
    }
```

7. 拆票操作：同时将目标列车和车厢的票进行拆分。
    - 第一步：先找到目标座位的某个区间i到j，需要满足以下条件：
        - 条件一：i ≤ departure且j ≥ arrival
        - 条件二：该座位在区间i到j下全程空闲
        - 条件三：j - i 的差值最大
    - 第二步：将区间i到j的票拆成一张始发站到i的票和一张j到终点站的票
    - 具体代码如下：

```
    //将票拆开
    private void spilt(Route r, Coach c, Seat s, int departure, int arrival) {
        //寻找该座位的最大空闲区间
        int res = getMaxIdleInterval(s, departure, arrival, r.stationNum);
        int from = res / 100, to = res % 100;
        //拆票
        r.left[from][to].getAndDecrement();
        c.left[from][to].getAndDecrement();
        if (from < departure) {
            r.left[from][departure].getAndIncrement();
            c.left[from][departure].getAndIncrement();
        }
        if (arrival < to) {
            r.left[arrival][to].getAndIncrement();
            c.left[arrival][to].getAndIncrement();
        }
    }
```

8. 合并票操作：同时将目标列车和车厢的票进行合并。
    - 第一步：先找到目标座位的某个区间i到j，需要满足以下条件：
        - 条件一：i ≤ departure且j ≥ arrival
        - 条件二：该座位在区间i到departure和arrival到j下全程空闲
        - 条件三：j - i 的差值最大
    - 第二步：将始发站到i的票的一张票和j到终点站的一张票合并成一张新的从i到j的票
    - 具体代码如下：

```
    //将票合并
    private void merge(Route r, Coach c, Seat s, int departure, int arrival) {
        //寻找该座位的空闲区间
        int res = getMaxIdleInterval(s, departure, arrival, r.stationNum);
        int from = res / 100, to = res % 100;
        //合并票
        r.left[from][to].getAndIncrement();
```

```
        c.left[from][to].getAndIncrement();
        if (from < departure) {
            r.left[from][departure].getAndDecrement();
            c.left[from][departure].getAndDecrement();
        }
        if (arrival < to) {
            r.left[arrival][to].getAndDecrement();
            c.left[arrival][to].getAndDecrement();
        }
    }
```

9. 寻找空闲区间操作：
    - Talk is cheap, here is the code

```
    //从[departure, arrival]左右扩散，看该座位空闲的最大区间段
    private int getMaxIdleInterval(Seat s, int departure,
                                   int arrival, int stationNum) {
        int from = departure - 1, to = arrival;
        while (from >= 1) {
            if (s.isOccupied(from)) {
                break;
            }
            from--;
        }
        //比如是从from = 2退出来，那么代表2-3不是空的，所以from = 2 + 1 = 3
        //正常循环结束退出来的，那么此时from = 0，所以from = 0 + 1 = 1
        //故都需要from++
        from++;
        while (to < stationNum) {
            if (s.isOccupied(to)) {
                break;
            }
            to++;
        }
        //比如是to = 7退出来的，那么代表7-8不是空的，此时to = 7没错
        //正常循环结束退出来的，那么此时to = stationNum，也同样正确
        return from * 100 + to;
    }
```

正确性分析

1. 总体思路是：单线程重复执行多线程并发执行的操作，最后比较两者的状态是否相同
    - 第一步：多线程并发执行查票和买票操作，把操作记录在log日志中。
    - 第二步：多线程全部执行完后，单线程读取日志内容，根据信息购买对应的座位，并在买的过程中检查座位是否空闲，如果不是，则表明并发有错误。
    - 第四步：比较两者最后各列车各区间剩下的票数是否相等，如果有不相等的，表明并发有错误。

    - 第五步：多线程并发地将第一步买到的票全部回，在退票的时候检查对应座位在目标区间是否空闲，如果是空闲的，则表明有错误；最后再检查各列车是否回复到原始状态，如果不是，则表明有错误。
2. 解释：这样能够检测并发买票或并发退票的正确性，如果最后多线程某次班车的某个区间的剩余票数跟单线程的不一样，则并发执行过程中必定出了问题。
3. 缺陷：(1)无法检测并发买票和退票的正确性，因为在日志中，信息的记录不一定对应每个方法的线性化点，所以日志中就可能会出现对某张票的退回记录输出在它的购买之前。如果我们单线程严格根据日志内容执行，那么肯定会出错

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(1)，即第一个座位尝试就成功了，然后发现该座位只有在目标区间是空着的；最坏情况下，时间复杂度为O(NS)，即虽然查过每节车厢都有余票剩下，但很不巧所有的余票都在查完后被买了，然后检查座位状态的时候又恰好在最后一站是不空闲的(O(S)复杂度)，因此遍历完所有座位(O(N)复杂度)最后还是发现没票。
    - 采用了无锁结构，每个执行该方法的线程最终都会返回，所以是wait-free的
2. `refundTicket`方法：
    - 耗时操作主要是：检查该座位时，查询其空闲空间的操作。最好情况下，该座位全程不空，是O(1)的；最坏情况下，除了目标区间外的所有区间都是空闲的，是O(S)
    - 采用了TTAS，所以是Deadlock-free的。
3. `inquiry`方法：
    - 假设一共有S个车站，最好情况下，查询的目标区间恰好就是始发站和终点站，是O(1)的；最坏情况下，查询的目标区间恰好位于整个区间的中间，是O(S)的。
    - 采用了无锁结构，每个执行该方法的线程最终都会返回，所以是wait-free的
4. 实际测试数据：默认testnum = 100000

线程数(个)	总耗时(ms)	吞吐率(次/s)	查询耗时(ns)	买票耗时(ns)	退票耗时(ns)
4	422.10	947640	236.51	14133.78	2985.55
8	606.46	1319123	346.64	21678.45	2692.48
16	801.73	1995690	462.91	20789.11	2109.22
32	1089.20	2937949	407.36	31465.56	1540.21
64	1740.15	3677839	419.48	57144.56	2100.04