## ◆ 并发数据结构设计思路

购票系统建立在一个三维数组上，其中数组的维度是routenum * stationnum*coachnum,数组的每个元素都是一个车厢内seatnum长度的bit串。那么每个车次其实对应一个[stationnum][coachnum]的二维数组。另外系统维护一个ConcurrentHashMap纪录已经卖出去的票。

例有5个车次，每个车次有3站，每个车次有4个车厢，每个车厢内8个座位，那么一个车次对应的数组初始化如下，1代表该座位有票，0表示没票：

| | 车厢1 | 车厢2 | 车厢3 | 车厢4 |
|---|---|---|---|---|
| 北京 | 11111111 | 11111111 | 11111111 | 11111111 |
| 上海 | 11111111 | 11111111 | 11111111 | 11111111 |
| 广州 | 11111111 | 11111111 | 11111111 | 11111111 |

### ➢ 查询行为实现思路

针对北京到广州的查询购买和退票行为，都不需要检查和修改广州对应的bit串，因为最后一站是下车站，不需要座位。

因为系统指明查询和购买可以同时进行，并且查询的行为频度最高，因此系统设计时查询是不加锁的。例如来了一个查询从北京到广州的票，则对每个车厢遍历，把每个车厢内北京和上海对应的bit串与操作，操作结果中1的个数即为该车厢北京到广州的票数，然后把每个车厢的票数相加，即为最终结果，然后结果返回。

### ➢ 购买行为实现思路

购买涉及到对数组的修改，并且同一张票不能卖给两个人，系统在设计时，例如来了一个购买从北京到广州的票，则从第一车厢开始，对北京和上海对应的bit串加锁并对bit串与操作，与操作结果中1的个数如果大于0，则证明有票，修改bit串，释放锁，在hashmap中纪录票，返回票。否则如果与操作结果中1的个数等于0，则证明该车厢没有票，释放锁，检查下一个车厢。

### ➢ 退票行为实现思路

退票受限验证退票的票id是否在hashmap中，如果不在，则直接返回false，否则给这张票的车厢和站对应的bit串加锁，修改，释放锁，并更新hashmap。

## ◆ 多线程测试程序设计思路

在测试程序中，生成thread_num个thread，并复写run方法，在方法里面，生成一个10以内的随机数，如果随机数小于3则买票，等于3则退票，否则查询，用来控制购买查询和退票3:6:1的行为比例。用一个linklist纪录已经买的票，退票从linklist里面随机选一张退，并根据返回结果判断购买的票是否成功退。同时随机生成一张票，然后退票，检查退票结果，对无效票返false。

对查询和购买行为测试时，具体任务是以随机数生成的，首先生成一个1到routenum的随机数表示车次，然后生成1到stationnum的随机数a表示起始站，然后生成a到stationnum的随机数b表示终点站，b比a大保证生成的任务都是有效的，使得测试的时间都是在有锁争用的任务上计算。系统纪录开始和结束的时间，计算在给定参数下跑完所有任务需要的时间。

## ◆ 方法是否可线性化

方法都是可线性化的。考虑系统内已经没有北京到上海的票了，这时候同时来了一个购买北京到上海的票的行为和一个退北京到上海的票的行为，退票行为可以直接请求要退票的座位的锁，买票行为需要从第一个车厢开始遍历，如果买到票，那么证明买票行为遍历到退票座位的时候退票行为完成，即退票的可线性化点在买票之前，如果没有买到票，则买票的可线性化点在退票之前。

### ➢ 查询可线性化点

```
//只需要验证从[start:end) 全是1的便是有座
BitSet res = new BitSet(this.seatNum);
int ticket_sum = 0;
for(int i = 0; i<this.coachNum;i++){
    res.set(0,this.seatNum);
    for(int j=start;j<end;j++){
        res.and(bs[route-1][j][i]);
    }
    ticket_sum += res.cardinality();
}
return ticket_sum;
```

其中蓝色阴影为查询的可线性化点，即计算座位个数的时候。

➢ **购买可线性化点**

```
try{
    for(int j = start; j < end; j++){
        res.and(bs[route-1][j][i]);
    }
    pos = res.nextSetBit(0);
    if(pos!=-1){
        //找到票,车厢是coachnum，座位号是pos
        for(int k = start;k<end;k++){
            bs[route-1][k][i].set(pos,false);//将指定的座位置为0
        }
        isSuccess = true;
        break;
    }
}finally{
    for(int p=start;p<end;p++){
        bs_lock[route-1][p][i].unlock();
    }
}
```

其中蓝色阴影为购买的可线性化点，即找到座位并把这个座位置为被购买状态时。

➢ **退票可线性化点**

```
for(int i =start;i<end;i++){//对这个车次这个区间的这个车厢的所有位置加锁
    bs_lock[route_refound][i][coach_refound].lock();
}
try{
    for(int i = start;i<end;i++){
        bs[route_refound][i][coach_refound].set(seat_refound);//把这个位置重新设置为1
    }
}finally{
    for(int i =start;i<end;i++){
        bs_lock[route_refound][i][coach_refound].unlock();;
    }
}
```

其中蓝色阴影为退票的可线性化点，即退票将原来数组的位置重新置为1。

◆ **分析每个方法的特性**

查询是没有用锁的的，所以是wait-free的，也是starvation-free的。购买和退票都加了锁，是starvation-free的，但是在锁争用时可能会block，因此不是lock-free的。

◆ **系统正确性和性能分析**

➢ **正确性分析**

系统对数组的修改仅发生在买票和退票行为上，查询是无锁的，如果一个bit串中有一位为0，说明对应车厢对应站已经卖出，则对查询区间的车厢内的bit串与操作，结果为1证明这个位置每一站都没有卖出，那么说明有一个座，1的个数即为这个车厢这个区间空座的个数。

买票首先在指定的站区间上加锁，如果有票则修改，没有票释放锁，退票也是先加锁再修改然后释放锁，即对数组的修改都是加锁的，不存在两个函数调用同时修改同一个bit串。

- **性能分析**

下表为给定默认参数下本地的执行时间，但是每次运行所需要的时间有区别，这个可能与任务都是以随机数生成有关，当随机数不够随机时，争用会比较多，另外在测试程序说明中已经指出保证了生成的购买和查询任务都是有效的，即终点站大于起点站，使得购买和查询都是存在锁争用，使时间的计算更合理。

| 线程数 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| 执行时间 | 25ms | 21ms | 22ms | 24ms | 27ms |
| 平均每个任务时间 | 0.0025 | 0.0021 | 0.0022 | 0.0024 | 0.0027 |
| 吞吐率 | 400/ms | 476/ms | 455/ms | 417/ms | 370/ms |