此次作业有两个不同版本，最终按ticketingsystem文件夹中为准，ticketingsystem1中为版本二（对照版本）。

数据结构设计：

private static boolean[][][][] *routes*;  
private static AtomicInteger[][][] *ticketNum*;  
private static AtomicLong *tid* = new AtomicLong();

private ConcurrentHashMap sold = new ConcurrentHashMap<Long,Ticket>();

路程段：

路程段的含义就是每两个车站之间就是一个路程段，比如总共有n个station则有n-1个路程段。当一个乘客从站1到站3，则他会占用路程段1和路程段2，也就是说同一个座位可以支持同时卖出1-2,2-3这样形式的车票，这个比较符合认知。

routes:

此为最基本的存储结构，一个boolean的4维数组。之所以用数组是因为数组访问比较高效。其中routes[i][j][k][l]对应了route为i的列车的第j个车厢的第k个座位的第l段路程是否被占用。

ticketNum:

这个数据结构不影响整体，是为了加速查询余票构造的，其中存的都是AtomicInteger。因为查询操作在这样一个系统中会经常被使用，所以另外维护一个数据结构来加速查询非常必要。ticketNum[i][j][k]代表route为i的列车的从站j到站k还有多少张空余座位。具体查询方法会在下面函数介绍中讲解。

tid:

很简单的原子数据，保证不重复即可。

sold:

为了保证退票的正确性引入的线程安全且高效的数据结构ConcurrentHashMap，其在每次买票成功后会新增一个元素，在退票时会检查退的票和对应tid的票是否一致，一致才退票成功，并且从sold中删除对应票。

函数实现思路：

buyTicket():

函数一开始会判断购票需求的合法性，比如出发站要小于终点站等等约束条件，如果合法则会从该列车的某个座位开始循环遍历座位，每次进入新的车厢都会调用查询余票函数，当发现余票为0则购票失败。对遍历过程中的一个座位，该座位数组中有stationnum-1个路程段，此时假设买n-(n+m)站的票，那么则需要将数组中第n-1到第n+m-2的路程段占用。此处使用了java中的sychronized关键字来对该座位进行加锁保证此时没有其他进程干扰，之后判断路程段如果是false(该位置该路程段空闲)则设置为true(占用该位置的该路程段)，依次占用该票所需的所有路程段，如果某次占用尝试失败则意味着该座位不能满足买票需求，则逆序把之前占用的重新设置为空闲，函数开始尝试下一个座位；如果成功则表示占用成功，然后会按规则更新ticketNum（具体方法见下文），最后设置买到票的座位车厢等信息并返回。

inquiry():

此处介绍一下ticketNum数据结构，它是一张3维的表，记录了某列车每两站之间的余票数。为了更清楚讲解，现假设某座位有以下这种一般情况：

下表表示的不是座位而是路程段，如图所示的10个路程段则有11个站。打\*表示该路程段被占用，如下图则第1站到第3站和第9站到第10站已经没票了。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1\* | 2\* | 3 | 4 | 5 | 6 | 7 | 8 | 9\* | 10 | ... |

现在假设新的购票需求为第6站到第8站，那么表就会变为：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1\* | 2\* | 3 | 4 | 5 | 6\* | 7\* | 8 | 9\* | 10 | ... |

此时第6、7路程段被占用，确认购票成功后，就会去更新ticketNum结构，此时我们可以看到买了6到8站，不光是6到8站的余票数量会有影响，3到7,、6到9等等也会受到影响。所以算法会从成功买票的起始站往前找直到找到一个已经被占用的路程段，比如这个例子中会找到3，对应第三站往后还是空闲的。同样的也会从第7路程段往后找直到找到被占用的9路程段。此时我们得到了向前第一个空闲的3路程段和向后最后一个空闲的8路程段，它们对应着从第3站开始到第9站结束的这些站的余票都会受影响。所以我们从3-7开始，对3-7、3-8、3-9、4-7、4-8、4-9、5-7、5-8、5-9、6-7、6-8、6-9、7-8、7-9对应ticketNum的余票数减1(注意此处需要从路程段转化到车站对应)。此处减1其实代表了买了这张票以后造成的哪些路段的影响，需要注意的是比如3-4、4-6、8-9这些路段，虽然也在3-9区间内，但其实买了这张票对这些票并没有影响，所以不需要减。同样的对于1-7、6-10这类的票，虽然买的票中包含了这些车站，但因为它们本身已经被占用了，也就是说它们本身就没有余票，当前票的成交不会对这个座位的这些余票产生影响。最终我们就能完成对ticketNum的更新，针对这个座位来对相应的数组中的值作减1，可以看到这是能保证正确性的。

有了这个数据结构，就能大大加快查询速度，查询函数简化为查表只有O(1)复杂度，当然查询操作对表中数值来说是原子的，使用了AtomicInteger的get()操作。虽然它需要对座位加锁，增加了锁的粒度，但因为查询操作正常情况下占比很大，所以这个优化是非常值得的，具体优化数据见后文分析。

refundTicket():

退票函数是买票函数的一个简化版，首先检查要退的票的正确性，然后查找到该座位，再获取该座位的锁，之后将routes数组里相应路程段的状态改为空闲，最后再更新ticketNum表并从sold中移除该票，更新的过程其实就是buyTicket里更新过程的逆操作，即将此时能释放出的所有可能余票都加1，当然头尾两个站还是得重新找，因为此时这个座位的其它路程段可能和购票时相比发生了变动。

版本二的实现思路：

此次作业有第二个版本，该版本的查询余票不是像之前那样构造表来查询，而是遍历所有座位来累加取得余票数量。但不同的地方不止这个，在优化查询版本中每次购票退票都需要对整个座位加锁，因为在更新ticketNum的时候还需要检查座位对应路程段的状态信息。而如果没有查询优化，则我们不需要对座位加锁，我们只要把routes数据结构换成AtomicBoolean的数组，这样在遍历查找的时候调用compareAndSet，如果所有需要占用的路程段都设置占用状态成功则表示购票成功，否则回退这些设置，退票同理，这样能避免锁的争用。

测试程序思路：

测试程序参考Trace中的测试样例，启动多个进程，然后在每次相关函数调用的前后取得系统时间，并计数调用该函数的总次数，最后对所有运行时间取平均得到平均运行时间，即得到不同函数调用的延迟，对所有操作次数累加除以总运行时间则得到吞吐量。重复以上过程多次取平均减小误差。

正确性及同步分析：

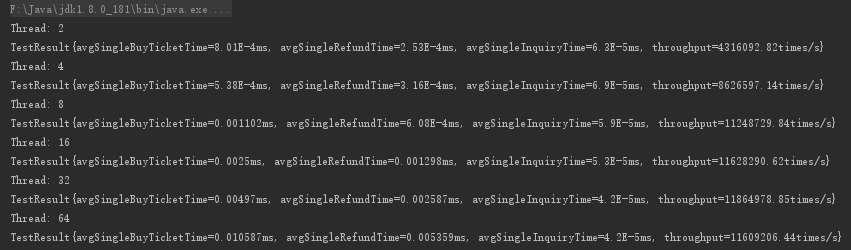
对版本一（即对座位加锁）:

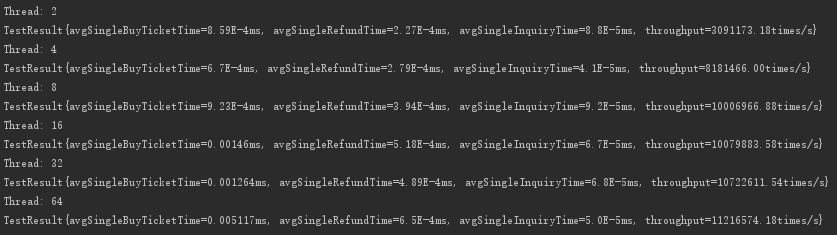
访问座位状态信息的就buyTicket和refundTicket，通过对当前访问的座位加锁，使得访问某一个座位的仅可能是一个进程。所以座位占用状态的检查和修改都能保证不受其他进程的干扰，所以是正确的，同时这两个方法显然也是可线性化的，线性化点可以认为是在成功修改完ticketNum的时刻。而ticketNum表里的元素都是AtomicInteger，所以在对相应余票作增减的时候调用的都是CAS操作，也能保证正确性，其中查询操作保证的是静态一致性。

由上文叙述可知，buyTicket和refundTicket都要争用座位锁，所以它们不是lock-free也不是starvation-free，另外因为只有一个锁的争用，故不存在死锁，是deadlock-free的。此实现通过随机定位起始遍历点来避免有进程饥饿，实际效果较好。对于inquiry函数，它只是返回数组元素，用到了AtomicInteger的get操作，所以是wait-free的。

性能分析：

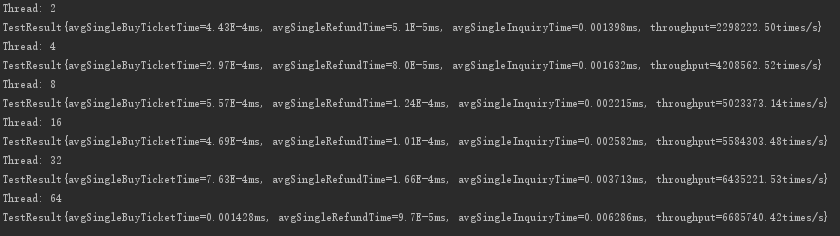
1. 分析整个系统，当多个进程同时检查车厢时，如果遍历比较集中，那很容易出现竞争情况。所以此次系统实现在buyTicket时的检查顺序采用随机的起始位置，这样不同进程买票过程尽量分散，则能减少竞争。下图1展示了不采用随机起始位置的性能，下图2则为采用随机起始位置的性能。可以看到在线程数比较多的时候，buyTicket的延迟在采用随机起始的时候能下降50%以上，这个优化是有很大提升的。





1. 查询不优化实现：

此次作业有第二个版本，该版本的查询余票不是像之前那样构造表来查询，而是遍历所有座位来累加取得余票数量。但不同的地方不止这个，在优化查询版本中每次购票退票都需要对整个座位加锁，因为在更新ticketNum的时候还需要检查座位对应路程段的状态信息。而如果没有查询优化，则我们不需要对座位加锁，我们只要把routes数据结构换成AtomicBoolean的数组，这样在遍历查找的时候调用compareAndSet，如果所有需要的路程段都设置占用状态成功则表示购票成功，否则回退这些设置，退票同理，这样能避免锁的争用，同时在正确性保证下它们能实现wait-free，因为当CAS失败时也就是表示这个座位该区间不空闲，那么就立即找下一个座位，不会自旋等待，所以能实现wait-free。这两个函数也是可线性化的，线性化点在设置完需要设置的AtomicBoolean时。看下图为查询不优化的测试结果，可以看到购票和退票操作的延迟都更低，性能提高了一个数量级，但是查询的性能则相比查询优化差了两个数量级。在实际环境中，查询操作应该是远远多于购票和退票操作的，所以查询优化能显著提升系统性能，从吞吐量可见，查询优化的吞吐量提高了一个数量级。



1. 从前面的图片可以看到，随着并发程度的提高，buyTicket和refundTicket在优化查询和不优化的两个版本的实现中平均耗费时间都有所上升，因为越多线程的并发导致了更多的争用所以是合理的。另外可以看到在查询优化版本中查询的时间几乎不会因为并发数变多而变长，查询不优化版本的查询则会随着并发变大而耗时更长，可见优化查询能在牺牲部分买退票的性能下大大提升查询的性能，是非常有必要的。
2. 可以看到吞吐量在并发程度较低时会随着并发的线程数增多而变大，那是因为系统还没到承载的极限，而一般在大于等于32个线程并发后，吞吐量不会再有太大的变化，这可以看作是整个系统能承载的最大的吞吐量（与机器也有关）。