给定 Ticket 类：

```
class Ticket{
    long tid;
    String passenger;
    int route;
    int coach;
    int seat;
    int departure;
    int arrival;
}
```

其中，tid 是车票编号，passenger 是乘客名字，route 是列车车次，coach 是车厢号，seat 是座位号，departure 是出发站编号，arrival 是到达站编号。
给定 TicketingSystem 接口：

```
public interface TicketingSystem {
    Ticket buyTicket(String passenger, int route, int departure, int arrival);
    int inquiry(int route, int departure, int arrival);
    boolean refundTicket(Ticket ticket);
    boolean buyTicketReplay(Ticket ticket);
    boolean refundTicketReplay(Ticket ticket);
}
```

其中,

- buyTicket 是购票方法，即乘客 passenger 购买 route 车次从 departure站到 arrival 站的车票 1 张。若购票成功，返回有效的 Ticket 对象；若失败（即无余票），返回空对象（即 return null）。
- refundTicket 是退票方法，对有效的 Ticket 对象返回 true，对错误或无效的 Ticket 对象返回 false。
- inquriy 是查询余票方法，即查询 route 车次从 departure 站到 arrival站的余票数。

正确性要求

- 每张车票都有一个唯一的编号 tid，不能重复。
- 每一个 tid 的车票只能出售一次。退票后，原车票的 tid 作废。（TODO： tofix）
- 每个区段有余票时，系统必须满足该区段的购票请求。
- 车票不能超卖，系统不能卖无座车票。
- 买票、退票和查询余票方法都需满足可线性化要求。

设计思路

方案一， 首先，通过对全局的大锁的方案，会直接将线程排队编程单一线程操作，而对于本问题中，数据都存储在内存中，不存在大量的网络或磁盘io，多线程并没有因为dma等操作让出cpu使用时间，反而导致了竞态下的不断争用，使得线程越大时，吞吐量越小。

对于全局的粗粒度锁，我们可以将其更换为细粒度锁，对于每一个座位都设置一把细粒度锁，需要读写时，先获取这把粗粒度锁，再进行读写。使得系统不会阻塞在一个一个地方。

但是细粒度锁带来的一个问题。在查询操作中，因为查询操作需要遍历整个车车次的所有位置来判断是每个位置。对当前的沉着区间是否存在余票？在这个过程中便可能出现， 在查询的整个毒过程中，可能会出现某些买票或者退票操作已经提交，因为读取开始的时间和最后一次读取的时间不一样，导致这两个时间所看到的整体的列车状况是不一样的。这也对应了数据库中不可重复读的问题。

为了解决这一问题，本方案实现了一个简易的多版本机制。

数据库中mvcc的实现方式中，在读取和写操作过程中，都会获取一个单调递增的事务id。然后会先生成一个Read View，Read View 有四个重要的字段：

- m_ids ： 指的是在创建 Read View 时，当前中「活跃事务」的id 列表，注意是一个列表，“活跃事务”指的就是，启动了但还没提交的事务。
- min_trx_id ： 指的是在创建 Read View 时，当前系统中「活跃事务」中事务 id 最小的事务，也就是 m_ids 的最小值。
- max_trx_id ： 这个并不是 m_ids 的最大值，而是创建 Read View 时当前系统中应该给下一个事务的 id 值，也就是全局最大的事务 id + 1。
- creator_trx_id ： 指的是创建该 Read View 的事务的事务 id。

线程更新系统中的值时，多版本系统会维护一个版本链用于查询数据的旧值。版本链维护了事务id递增的历史版本，每次更新操作会将更新值和版本信息存储起来，并且可以回溯历史版本。

系统读取时，首先获取一个Read View。

- 如果记录的 trx_id 值小于 Read View 中的 `min_trx_id` 值，表示这个版本的记录是在创建 Read View 前已经提交的事务生成的，所以该版本的记录对当前事务可见。
- 如果记录的 trx_id 值大于等于 Read View 中的 `max_trx_id` 值，表示这个版本的记录是在创建 Read View 后才启动的事务生成的，所以该版本的记录对当前事务不可见。
- 如果记录的 trx_id 值在 Read View 的 min_trx_id 和 max_trx_id

  之间，需要判断 trx_id 是否在 m_ids 列表中：
  - 如果记录的 trx_id **在** `m_ids` 列表中，表示生成该版本记录的活跃事务依然活跃着（还没提交事务），所以该版本的记录对当前事务**不可见。**
  - 如果记录的 trx_id **不在** `m_ids`列表中，表示生成该版本记录的活跃事务已经被提交，所以该版本的记录对当前事务**可见。**

但是在本方案中，为了简化实验，可以将读的可线性化点设置在获取到Read View的时刻，而将写操作的可线性化设置在成功修改seat并且交还活跃事务的时刻。减少了实现的工作量。

具体实现

基础数据类型

因为并发操作时对Route隔离的，不同Route之间的并发操作互相不影响，所以我们可以针对每个route建立一个并发系统，

```
public class Route {
    int routeId;
    int coachnum;
    int seatnum;
    int stationnum;
    int totalseats;
    int threadnum;
    Seat[] seats;
    AtomicLong gobalID;
    View preView;
```

```
    volatile boolean isCacheViewUpdate;
    HashSet<Long> theadViewId;
    long gobalID;
    ReentrantReadWriteLock activeLock;
}
```

使用gobalID来座位全局事务ID分配器， seats中维护每个seat的位置。

```
public class VersionState {
    long version;
    long state;

    public Boolean isVisable(View view) {
        if (version > view.viewId) {
            return false;
        } else if (version < view.minId) {
            return true;
        } else {
            if (view.activeIds.contains(Long.valueOf(version))) {
                return false;
            } else {
                return true;
            }
        }
    }
}

public class Seat {
    ReentrantLock seatLock;
    HashMap<Long,Ticket>soldTickets;
    ArrayList<VersionState> versionStates;
    volatile int tail;
}
```

每个seat中维护了细粒度锁和版本信息，为了简化操作，对于每个seat直接使用ArrayList维护了多版本，通过tail获取到版本链的末端。

View生成

```
public class View {
    long viewId;
    HashSet<Long>activeIds;
    long minId;
}

public View createView() {
    activeLock.readLock().lock();
    try {
        long viewId = gobalID;
        HashSet<Long> actives = new HashSet<Long>();
        actives.addAll(activeIds);
        preView=new View(viewId, actives);
        return preView;
    } finally {
        activeLock.readLock().unlock();
    }
}
```

因为activeIds是创建和提交时的临界区，我们这里通过读写锁获取资源，将route中正在活跃的Id加入view中，并且设置自身Id。

查询操作

```
public long readView(View view) {
    int localTail = tail;
    while (localTail >= 0) {
        if (versionStates.get(localTail).isVisable(view)) {
            break;
        }
        localTail -= 1;
    }
    return versionStates.get(localTail).state;
}

public class VersionState {

    public Boolean isVisable(View view) {
        if (version > view.viewId) {
            return false;
        } else if (version < view.minId) {
            return true;
        } else {
            if (view.activeIds.contains(Long.valueOf(version))) {
                return false;
            } else {
                return true;
            }
        }
    }
}
```

通过view读取多版本的过程中，首先获取到最大的版本，从版本号大到版本号小的记录，从后往前检查该条记录是否对view可见，返回第一个可见的记录。当记录版本大于读操作的viewId时，是不可见的。当记录版本小于读操作的minId时，是可见的。当记录版本处在前两者之间是， 如果活跃事务包括记录版本是对该view不可见的，否则对该view可见。

逐步检查过程，可以使用二分查找进一步优化。

写操作

写操作设计对版本数组的修改，写操作设计到读版本数组的争用，所以需要对该seat进行进行加锁。

写操作前先通过golabalId自增获取到一个版本号，然后通过将当前Id添加进activeIds，在关闭事务时，会执行对应的逆操作。

```
public long beginTrx() {
    activeLock.writeLock().lock();
    try {
        isCacheViewUpdate = false;
        gobalID += 1l;
        long viewId = gobalID;
        activeIds.add(viewId);
        return viewId;
    } finally {
        activeLock.writeLock().unlock();
    }
}
```

```
public boolean closeTrx(long versionId) {
    activeLock.writeLock().lock();
    try {
        isCacheViewUpdate = false;
        activeIds.remove(versionId);
        return true;
    } finally {
        activeLock.writeLock().unlock();
    }
}
```

seat在实际执行写操作时，通过修改版本数组和tail尾指针实现，为了保证版本链是递增的，在提交前需要确认当前写版本号是否大于版本数组末尾的版本号，如果小于版本数组末尾，需要通过回到route中重新进行获取新版本再重新进行。因为写操作不仅涉及到对seat本身的修改，也涉及到系统版本计数器的修改。对此，本方案中采用两阶段提交的方式，当不满足条件要求时返回失败或者要求重试，成功时则进行commit。 退票操作同理。

```
public Result tryBuyTicket(long version, String passenger, int departure, int arrival) {
    seatLock.lock();
    lastState = versionStates.get(tail);
    if (lastState.version > version) {
        seatLock.unlock();
        return Result.SMALLVERSION;
    }
    if (checkState(lastState.state, departure, arrival)) {
        versionStates.add(new VersionState(version, stateBuy(lastState.state, departure,
arrival)));
        tail += 1;
        soldTickets.put(version, issueTicket(passenger, version, routeId, coachId, seatId,
departure, arrival));
        return Result.SUCCESSED;
    } else {
        seatLock.unlock();
        return Result.NOTAVAILABLE;
    }
}

public Ticket commitBuyTicket() {
    VersionState lastState = versionStates.get(tail);
    Ticket ticket = soldTickets.get(lastState.version);
    seatLock.unlock();
    return ticket;
}
```

无等待快照优化view生成

从上述代码来看， 进行read_view的创建过程中，因为需要获取m_ids（当前活跃事务列表），会对activeIds设置读写锁， commit过程中也会去获取这把mutex。 这个地方还是一个critical section。本质上该机制使用使用多版本机制将存储的数据上的blocking转为在activeIds上事务状态的blocking。

因为数据都是在内存中， 不存在网络IO等，实际上用处时极为有限的。

5 routes 8 coachs 100 seats 30 stations 10000 operator

ThreadNum: 4 BuyAvgTime(ms): 0.02193 RefundAvgTime(ms): 0.01938 InquiryAvgTime(ms): 0.04036 ThroughOut(op/ms): 103

ThreadNum: 8 BuyAvgTime(ms): 0.04368 RefundAvgTime(ms): 0.02943 InquiryAvgTime(ms): 0.03770 ThroughOut(op/ms): 189

ThreadNum: 16 BuyAvgTime(ms): 0.08887 RefundAvgTime(ms): 0.05051 InquiryAvgTime(ms): 0.06689 ThroughOut(op/ms): 213

ThreadNum: 32 BuyAvgTime(ms): 0.32423 RefundAvgTime(ms): 0.19161 InquiryAvgTime(ms): 0.18576 ThroughOut(op/ms): 142

ThreadNum: 64 BuyAvgTime(ms): 1.15540 RefundAvgTime(ms): 0.76702 InquiryAvgTime(ms): 0.50090 ThroughOut(op/ms): 95

我们可以看到随着线程数据增加，吞吐量反而减小。

对此，我们可以通过无等待快照来优化view生成过程，

```
public View createView() {
    Long viewId = gobalID.get();
    HashSet<Long> actives = new HashSet<Long>();
    ArrayList<Long> activeIds = theadViewId.scan();
    for (Long id : activeIds) {
        if (id != 0) {
            actives.add(id);
        }
    }
    return new View(viewId, actives);
}

public long beginTrx() {
    Long viewId = gobalID.incrementAndGet();
    theadViewId.update(viewId);
    return viewId;
}

public boolean closeTrx() {
    theadViewId.update(0L);
    return true;
}
```

将对activeId的操作转为无等待快照的scan和update操作。

50 routes 20 coachs 100 seats 30 stations 100000 operator

ThreadNum: 4 BuyAvgTime(ms): 0.00395 RefundAvgTime(ms): 0.00399 InquiryAvgTime(ms): 0.06909 ThroughOut(op/ms): 73

ThreadNum: 8 BuyAvgTime(ms): 0.00408 RefundAvgTime(ms): 0.00334 InquiryAvgTime(ms): 0.10029 ThroughOut(op/ms): 104

ThreadNum: 16 BuyAvgTime(ms): 0.00513 RefundAvgTime(ms): 0.00521 InquiryAvgTime(ms): 0.14610 ThroughOut(op/ms): 140

ThreadNum: 32 BuyAvgTime(ms): 0.03360 RefundAvgTime(ms): 0.00798 InquiryAvgTime(ms): 0.20398 ThroughOut(op/ms): 188

ThreadNum: 64 BuyAvgTime(ms): 0.10345 RefundAvgTime(ms): 0.01508 InquiryAvgTime(ms): 0.29483 ThroughOut(op/ms): 263

可以看到随着线程数量的增加，吞吐量在不断变大，但是并为近线性增长。

不过当route比较小时，吞吐量的增长曲线比较快

5 routes 20 coachs 100 seats 30 stations 100000 operator

ThreadNum: 4 BuyAvgTime(ms): 0.01221 RefundAvgTime(ms): 0.00490 InquiryAvgTime(ms): 0.04306 ThroughOut(op/ms): 111

ThreadNum: 8 BuyAvgTime(ms): 0.01799 RefundAvgTime(ms): 0.00336 InquiryAvgTime(ms): 0.04176 ThroughOut(op/ms): 223

ThreadNum: 16 BuyAvgTime(ms): 0.02822 RefundAvgTime(ms): 0.00355 InquiryAvgTime(ms): 0.04927 ThroughOut(op/ms): 366

ThreadNum: 32 BuyAvgTime(ms): 0.04341 RefundAvgTime(ms): 0.00547 InquiryAvgTime(ms): 0.06175 ThroughOut(op/ms): 577

ThreadNum: 64 BuyAvgTime(ms): 0.07242 RefundAvgTime(ms): 0.00863 InquiryAvgTime(ms): 0.08093 ThroughOut(op/ms): 817

但是实际上route之间是相互隔离的，不应该对多线程操作有影响。 猜测可能是因为线程切换route过程中cache开销较大，但是因为实验设备限制，没有办法在较多线程情况下运行profiler，无法验证。

正确性和进展性

读操作使用了多版本机制来保证读正确性，可线性化点为view生成的位置。因为使用无等待快照生成view，遍历seat时使用版本进行比较，所以整个过程时无等待的。

购票操作过程中，每轮循环中，首先使用inquiry进行查询，查询到余票为0则退出，否则遍历seat进行操作，购票成功进行提交，否则开始下一轮循环。 该方法有两个可线性化点，购票成功时为购票操作提交时，购票失败为inquiry为0时的可线性化点。 过程中因为设计到细粒度锁，所以是无死锁的。

退票操作分析同上，但是因为inquiry，所以线性化点为退票提交时刻。 同理该操作也是无死锁的。 如果进一步，多版本使用无锁双向链表的化， 购退票可以做到无锁操作。

程序实现在src/main/java下，也可在https://github.com/my-vegetable-has-exploded/TrainTicketingSystem中ViewSnapshot分支中找到代码。