性能测评报告

中文题目 用于列车售票的可线性化并发数据结构

学生姓名 郭智方

摘 要

本文设计了一个用于列车售票的可线性化并发数据结构，该结构可以用于并发地购票、售票和查询余票。其中购票和售票满足线性一致性，查询余票满足静态一致性。

本文使用列车当前承载量分两种搜索方式：当列车当前承载量小于最大座位数目时，可以直接购买当前承载量之后的空座位；当列车当前承载量大于最大座位数目时，则需要搜索上一次的退票位置或购票的位置。

考虑到线程多少对于数据处理的快慢问题，本文针对不同的线程采用不同的处理方式，当线程较多时，最大承载量会较快地增加到最大值，此时使用 LongAdder 并发计数，虽然增加时结果会出现问题，但是可以采用遍历的方法避免其错误，同时采用分布式的方式形成车票编号，并使用额外线程维护一个全局缓存，在高并发条件下快速更新时间戳，这样做虽然会失去一定精度但是能够有效提升并发性；当线程较少时，最大承载量会不会较快达到最大值，采用 getAndIncrement()获取准确的最大承载量反而会提升性能，对于车票编号可以在退票时使用 getAndIncrement()递增更新对应标签戳。在多线程和少线程获得的时间戳和标签戳都是在退票时更新，且无论是否完成退票，之后与列次、起止站和座位一同生成编号，这样不仅能防止再次购买相同车票时获得相同的编号，还能保持系统的可线性化。

本文使用一维数组存储每个座位被占用的情况，在购票、退票和查询余票过程中遍历对应的座位，并使用 compareAndSet()实现了无锁的性质，为了提升查询速度，本文使用并发式哈希表的结构记录上次退票位置或上次购票的位置。

本文通过了性能测试、串行正确性测试和并发正确性测试。对于正确性测试，本文采用老师提供的串行正确性测和他人已实现的并发正确性测试，并通过了 50 轮测试，满足可线性化要求。对于性能测试，本文在测试阶段对系统首先进行预热，之后再计算性能平均值，并将性能测试结果与初步实现的单列车加锁的性能进行对比，平均吞吐量提升 201.35%。

关键词 列车售票 并发数据结构 静态一致性 可线性化 无锁

目录

第 1 章 设计思路

1.1 题目要求

用于列车售票的可线性化并发数据结构的原题目可以参见 project.pdf，题目要求实现可线性化的并发数据结构用于列车售票、退票和查询余票。其中要求：

（1） 每张车票都有一个唯一的编号 tid，不能重复；

（2） 每一个 tid 的车票只能出售一次。退票后，原车票的 tid 作废。

（3） 每个区段有余票时，系统必须满足该区段的购票请求。

（4） 车票不能超卖，系统不能卖无座车票。

（5） 买票、退票方法需满足可线性化要求。查询余票方法的约束放松到静态一致性。查询结果允许不精确，但是在某个车次查票过程中没有其他购票和退票的“静止”状态下，该车次查询余票的结果必须准确。

1.2 题目分析

综合分析题目要求和题目本身，我有以下几个观点：

（1） 各个车次之间的行为不会产生竞争，因此各个车次之间的行为可以并发处理。

（2） 可以将整个车厢看作一个整体，即使用 coachnum*seatnum 来作为车次的总座位数目 seatsum。

（3） 当车次总的购票数目 capicity 小于 seatsum 时，说明在[capacity, seatsum]这个区间中的票都还没有被购买，因此直接分配给购买车票的人，同样查询余票时，在[capacity, seatsum]区间的余票可以不用遍历，直接加入余票总数中。经过实验可知，这种方法在线程较少，seatsum 较大时对性能提升有一定的帮助。

（4） 我主要想到四种结构以供参考：使用数组存储每个区间的余票数目，每次购票或退票对多个区间的余票进行加减，但是这样无法确定购票时获得的空座位置；使用数组存储任意站点间的余票，每次购票或退票对涉及到的区间进行加减，但是这样由于并发竞争较为复杂，会严重影响性能；使用数组存储每个座位在每个站点间的使用情况，但是这样在查询空座位时需要对每个座位的每个区间进行遍历；最后我使用数组 seatmap 存储每个座位的使用情况，该数组中存储每个数组在每个站的使用情况。

（5） 车票编号要求唯一且不能重复，我考虑使用分布式的方式根据每张车票的购买时间、车次号、出发站编号、到达站编号和座位号生成一个唯一的车票编号，

或使用 getAndIncrement()获得一个递增的车次编号。由于高并发下获取时间戳非常占用系统时间，因此我使用额外线程维护时间戳。

（6） 目前还存在的问题是当火车购票后，如果出现退票，购票时如何快速知道退票的位子，对此我使用 ConcurrentHashMap()构造 priorefund，这是一种类似于哈希表的结构，它采用数组加分段锁的实现，我用它来存储上次退票的位置，或者是购票的下一个位置，这样在查询余票或购票时可以不同每一个座位都遍历一遍。当然这样做也有一定的问题，即某张车票退票后会使得除该起止点以外的车票同样也能购买，如果同时修改其他起止点的 priorefund 中的数值会引发更多的竞争，具体解决办法见后文。

1.3 竞争分析

由题目要求可知，每个线程是一个票务代理，按照 60%查询余票，30%购票和 10%退票的比率，由于查询余票满足静态一致性，因此查询余票不会引发竞争。但是购票与购票之间，如果区间相交会引发竞争；退票与退票之间，如果退票区间相交，对于 seatmap 的修改和对于 priorefund 的修改会引发竞争；对于购票与退票之间，如果区间相交，二者的先后顺序会对最后的结果产生影响。

综上所述，最小的竞争为区间和区间之间，对此我使用 compareAndSet()实现了无锁操作，有效地提升了并发性能。

第 2 章 数据设计

2.1 数据类设计

（1） TicketingDS 类：系统的接口程序，使用 Seat 接口来调用 Seat_lessthread 类或 Seat_morethread 类的方法。

（2） Seat 接口：用于调用 Seat_lessthread 类或 Seat_morethread 类的 buyTicket、refundTicket 和 inquery 方法。

（3） Seat_lessthread 类：描述了线程较少时，线程不会在很快的情况下占用所有的座位，此时对每个座位的购票、查询和退票操作。

（4） Seat_morethread 类：描述了线程较多时，线程会在很快的情况下占用所有的座位，此时对每个座位的购票、查询和退票操作。

（5） SystemClock 类：描述了线程较少时，采用额外线程高并发获取时间戳从而形成车票编号。

2.2 数据类实现

2.2.1 TicketingDS 类

TicketingDS 类主要用于实现如图 2-1 所示的参数

```
1. public class TicketingDS implements TicketingSystem {
2.     private static AtomicLong[][][][] tid1; //车票编号
3.     private int routenum; //列车车次数目
4.     private int coachnum; //车厢数目
5.     private int seatnum; //座位数目
6.     private int stationnum; //站数目
7.     private int threadnum; //线程数目
8.     private int threadrange1=256;//线程阈值
9.     private Seat[] routemap;//车次图，用于存储每个车次的座位占用情况
10.    private int seatsum; //单趟列车总座位数目，因为每节车厢的座位数目没有意义
11.    private ConcurrentHashMap<Long, Ticket> soldTicket;//存储已购买车票
12.    private long timetag=0L;//时间戳
13. …
14. }
```

图 2-1 TicketingDS 类参数

在这里主要值得注意的是对于 tid 参数的维护，当线程数目较少时，每次购票的时间间隔较长，一般不会发生冲突，所以我在退票时，无论是否退票成功都使用 getAndIncrement()更新对应票的编号；当线程数目较多时，我在退票时，无论是否退票成

功都使用额外的线程更新时间戳。这样做能够防止退票后再次获取票编号重复的问题，且能保持可线性化。

```
1.  public long tidcoder(int route, int departure, int arrival, int i) {
2.      long coder;
3.      if(threadnum>threadrange1){//获取系统时间较长
4.
    coder=(timetag<<25)+((long)i<<14)+((long)route<<10)+((long)departure<<5)+((long)arrival);
5.      }
6.      else{
7.          coder=(tid1[route-1][departure-1][arrival-
    1][i].get()<<25)+((long)i<<14)+((long)route<<10)+((long)departure<<5)+((long)arrival);
8.      }
9.      return coder;
10.  }
11.  @Override
12.  public Ticket buyTicket(String passenger, int route, int departure, int arrival) {
13.      if
    (route<=0||route>this.routenum||arrival>this.stationnum||arrival<=0||departure>=arrival||departure<=
    0||departure>=this.stationnum){
14.          return null;
15.      }
16.      int i = routemap[route - 1].buyTicket(departure, arrival);
17.       if (i >= 0){
18.          Ticket ticket = new Ticket();
19.          ticket.tid=tidcoder(route, departure, arrival, i);
20.          ticket.passenger=passenger;
21.          ticket.route=route;
22.          ticket.coach=i/seatnum+1;
23.          ticket.seat=i%seatnum+1;
24.          ticket.departure=departure;
25.          ticket.arrival=arrival;
26.          soldTicket.put(ticket.tid,ticket);
27.         return ticket;
28.          }
29.      else{
30.          return null;
31.      }
32.  …
33.  }
```

图 2-2 tid 参数在不同线程下的获取方法

2.2.2 Seat 接口

Seat 接口中主要包含 Seat_lessthread 类或 Seat_morethread 类的 buyTicket、refundTicket 和 inquery 方法，具体实现如图 2-3 所示。该接口主要用于在不同线程下对 TicketingDS 类进行不同的初始化。

```
1.  public interface Seat{
2.      public int buyTicket(int departure, int arrival);
3.      public boolean refundTicket(Ticket ticket, int i);
4.      public int inquery(int departure, int arrival);
5.  }
```

图 2-3 Seat 接口的实现

2.2.3 Seat_lessthread 类

Seat_lessthread 类实现方法主要包括 seatsum 记录单趟列车总座位数目，seatmap 记录每个座位在每一站被占用的情况，priorefund 记录上一次退票后购票的票数，capicity 记录当前整个车次卖出去了多少张票，具体参数可见图 2-4。

```
1.  public class Seat_lessthread implements Seat{
2.      private int seatsum; //单趟列车总座位数目，因为每节车厢的座位数目没有意义
3.      private AtomicLong[] seatmap;//座位图，用于存储座位在每一站被占用情况
4.      private ConcurrentHashMap<Long, AtomicInteger> priorefund;//退票购票记录
5.      private AtomicInteger capicity; //当前整个车次承载量
6.
7.      public Seat_lessthread(int seatsum){
8.       this.seatsum=seatsum;
9.          seatmap=new AtomicLong[seatsum];
10.         for (int i=0; i<seatsum; i++) {
11.             seatmap[i] = new AtomicLong();
12.         }
13.         priorefund = new ConcurrentHashMap<Long, AtomicInteger>();
14.         capicity = new AtomicInteger();
15.     }
16. …
17. }
```

图 2-4 Seat_lessthread 类参数

（1） encoder(int departure, int arrival)

为了加快运算速度，我采用编码的方式对每个座位在每个站点的占用情况进行编码，如果该座位被占用，则为 1，否则为 0 或不存在，具体代码如图 2-5 所示，示意图如图 2-5 所示。

```
1.  private long encoder(int departure, int arrival){//输出当前出发到截至的位置编码
2.      long coder = 0;
3.      for (int j = departure - 1; j < arrival - 1; j++){
4.          coder += 1 << j;
5.      }
6.      return coder;
7.  }
```

图 2-5 encoder()代码

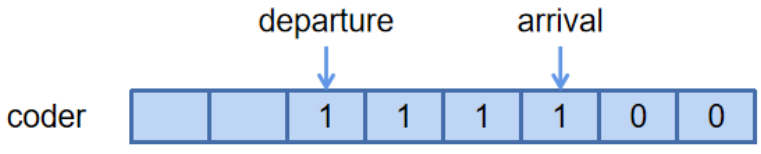

图 2-6 encoder()示意图

（2） buyTicket(int departure, int arrival)

在 Seat_lessthread 类中 buyTicket()主要分为两种模式：第一种模式当整个车次承载量 capicity 小于整个列车总座位数目 seatsum 时，可以认为 capicity 之后的票都没有人占用可以直接购买，此时 priorefund 只记录成功退票的位置。当

然可能存在这样一种情况，即搜索到 seatsum 时仍然没有找到空座位，这是因为其他购票线程占用了，对此就需要从上次退票的位置 priorefund.get(coder)开始购买直到 capicity。当然可能存在这样一种情况，即搜索到 capicity 时仍然没有找到空座位，对此就需要考虑其他线程在 priorefund.get(coder)之前退票的情况了，这就需要从第一个座位开始搜索直到 priorefund.get(coder)。如果仍然没有则认为票已经售空，具体示意图如图 2-7 所示。

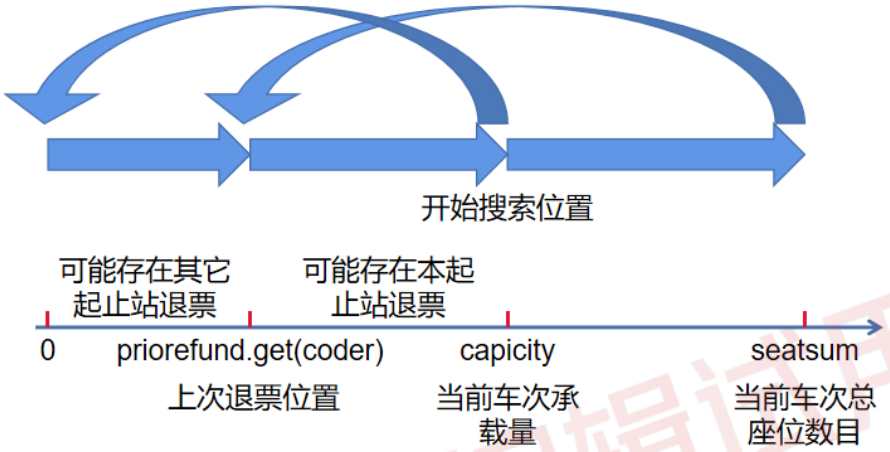

图 2-7 buyTicket()模式一搜索示意图

第二种模式当整个车次承载量 capicity 大于整个列车总座位数目 seatsum 时，可以认为整个车次的的位置都已经被占用，只是占用的站点不相同，此时 priorefund 不仅要录成功退票的位置，还需要记录购票后的下一个位置。这种模式下需要从 priorefund.get(coder)开始搜索，直到 seatsum。当然可能存在这样一种情况，即搜索到 seatsum 时仍然没有找到空座位，对此就需要考虑其他线程在 priorefund.get(coder)之前退票的情况了，这就需要从第一个座位开始搜索直到 priorefund.get(coder)。如果仍然没有则认为票已经售空，具体示意图如图 2-8 所示。

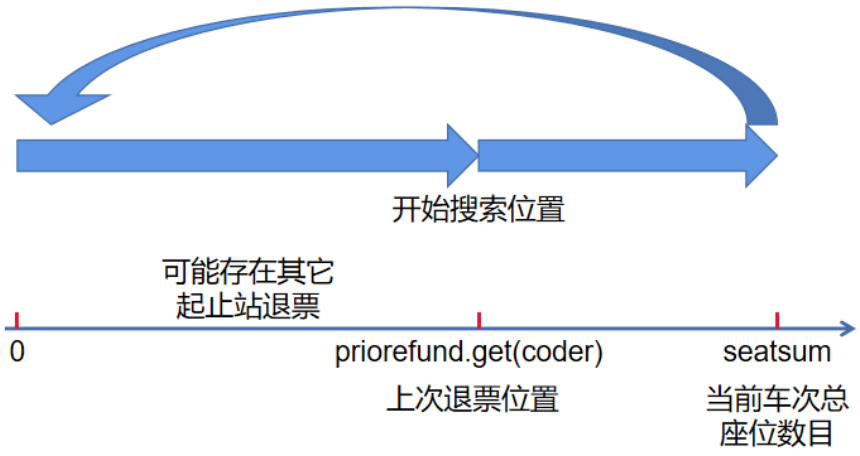

图 2-8 buyTicket()模式二搜索示意图

对于如何购票，我采用了 compareAndSet()，通过原子操作实现了无锁购票，具体程序如图 2-9 所示。具体做法是从 seatmap 搜索到一个座位将其编码与当前购买车票的编码进行&操作，如果等于 0 则表示该座位可以被占用，之后如果能通过 compareAndSet()修改 seatmap 则表明购票成功，否则再次循环或再次搜索新的票。

```
1.  public int buyTicket(int departure, int arrival){//购买车票
2.      long coder = encoder(departure,arrival);
3.      if (!priorefund.containsKey(coder)) { //如果没有买过该票，创建该票
4.          priorefund.put(coder, new AtomicInteger());
5.      }
6.      long newseat, oldseat;
7.      int end = seatsum;
8.      int currnum = capicity.get();//获取当前座位数目
9.      if(currnum<seatsum){
10.         end =currnum;
11.         for (int i = currnum; i < seatsum; i++){
12.             AtomicLong currmap=seatmap[i];
13.             oldseat=currmap.get();//座位占用情况
14.             while ((oldseat & coder) == 0){//座位未被占用
15.                 newseat = oldseat | coder;
16.                 if (currmap.compareAndSet(oldseat, newseat)){//执行成功说明购票成功
17.                     capicity.getAndIncrement();
18.                     return i;
19.                 }
20.                 oldseat=currmap.get();
21.             }
22.         }
23.     }
24.     AtomicInteger currpriorefund = priorefund.get(coder);//退回车票
25.     currnum = currpriorefund.get();
26.     for (int i = currnum; i <end; i++){
27.         AtomicLong currmap=seatmap[i];
28.         oldseat=currmap.get();//座位占用情况
29.         while ((oldseat & coder) == 0){//座位未被占用
30.             newseat = oldseat | coder;
31.             if (currmap.compareAndSet(oldseat, newseat)){//执行成功说明购票成功
32.                 currpriorefund.compareAndSet(currnum,i+1);
33.                 capicity.getAndIncrement();
34.                 return i;
35.             }
36.             oldseat=currmap.get();
37.         }
38.     }
39.     for (int i = 0; i <currnum; i++){
40.         AtomicLong currmap=seatmap[i];
41.         oldseat=currmap.get();//座位占用情况
42.         while ((oldseat & coder) == 0){//座位未被占用
43.             newseat = oldseat | coder;
44.             if (currmap.compareAndSet(oldseat, newseat)){//执行成功说明购票成功
```

```
45.                     currpriorefund.compareAndSet(currnum,i+1);
46.                     capicity.getAndIncrement();
47.                     return i;
48.                 }
49.                     oldseat=currmap.get();
50.                 }
51.             }
52.             return -1;
53.     }
```

图 2-9 buyTicket()程序

（3） refundTicket(Ticket ticket, int i)

refundTicket()代码我采用类似于 buyTicket()的方式，但是因为退票只需要考虑是否买过这张票，那么我只需要检查所退的票有没有被其他线程退掉或是否存在即可，具体程序如图 2-10 所示。如果能通过 compareAndSet()修改 seatmap 则表明退票成功，否则再次循环继续退票。

```
1.  public boolean refundTicket(Ticket ticket, int i){//查询车票
2.      AtomicLong currmap = seatmap[i];
3.      long oldseat = currmap.get();//旧座位是否被占
4.      long newseat;
5.      long coder = encoder(ticket.departure, ticket.arrival);
6.      AtomicInteger currpriorefund = priorefund.get(coder);
7.      while ((oldseat & coder) == coder){//旧座位没有被退才会执行
8.          newseat = (~coder) &oldseat;
9.          if (currmap.compareAndSet(oldseat,newseat)) {//执行成功说明退票成功
10.             int currnum = currpriorefund.get();
11.             while (i<currnum) {//执行循环说明该票没有从中剔除
12.                 currpriorefund.compareAndSet(currnum, i);//i 已经隐含减 1
13.                 currnum = currpriorefund.get();
14.             }
15.         capicity.getAndDecrement();
16.         return true;
17.     }
18. oldseat = currmap.get();
19. }
20. return false;
21. }
```

图 2-10 refundTicket()程序

（4） inquery(int departure, int arrival)

由题意可知，查询票只需要满足静态一致性即可，inquery()主要分为两种模式：第一种模式当整个车次承载量 capicity 小于整个列车总座位数目 seatsum 时，可以认为 capicity 之后的票都没有人占用，因此可以搜索到 capicity 之后直接加上 capicity，具体示意图如图 2-11 所示。

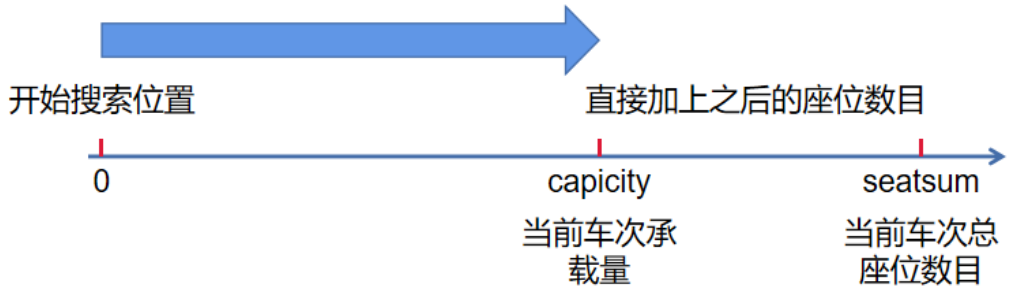

图 2-11 inquery()模式一搜索示意图

第二种模式当整个车次承载量 capicity 大于整个列车总座位数目 seatsum 时，可以认为整个车次的的位置都已经被占用，只是占用的站点不相同，此时需要对所有座位进行查询，具体示意图如图 2-12 所示。

图 2-12 inquery()模式二搜索示意图

inquery()的具体程序如图 2-13 所示。

```
1.  public int inquery(int departure, int arrival) {//查询车票
2.      int num = 0;//余票数
3.      long coder = encoder(departure, arrival);
4.      for (int i = 0; i < seatsum; i++) {
5.          AtomicLong currcoder = seatmap[i];
6.          if ((currcoder.get() & coder) == 0)//座位未被占用
7.          num++;
8.      }
9.      return num;
10. }
```

图 2-13 inquery() 程序

2.2.4 Seat_morehread 类

在 Seat_lessthread 中采用的 capicity 主要针对线程较少，总座位数目较多的情况，当线程数目较多时，capicity 会较快到达 seatnum，此时由于 capicity 采用的是 getAndIncrement() 反而会影响性能，因此在线程较多的时候使用 LongAdder，在低竞争的并发环境下 AtomicInteger 的性能是要比 LongAdder 的性能好，而高竞争环境下 LongAdder 的性能比 AtomicInteger 好，当有 1000 个线程运行时，LongAdder 的性能比 AtomicInteger 快了约 1.53 倍。

2.2.5 SystemClock 类

Java 获取当前时间戳一般是通过 System.currentTimeMillis()来获取。这是一个 native 方法，用于获取当前时间与 1970 年 1 月 1 日 0 点之间的差，虽然返回值的时间单位是毫秒，但该值的粒度取决于基础操作系统。为了能够解决高并发条件下的性能较慢的问题，最常见的办法是用单个调度线程（守护线程）来按毫秒更新时间戳，相当于维护一个全局内存缓存。其他线程取时间戳时相当于从内存取，不会再造成时钟资源的争用，代价就是牺牲了一些精确度。具体代码如图 2-15 所示。

```
1.  public class SystemClock {
2.      private static final String THREAD_NAME = "system.clock";
3.      private static final SystemClock MILLIS_CLOCK = new SystemClock(1);
4.      private final long precision;
5.      private final AtomicLong now;
6.
7.      private SystemClock(long precision) {
8.          this.precision = precision;
9.          now = new AtomicLong(System.currentTimeMillis());
10.         scheduleClockUpdating();
11.     }
12.     public static SystemClock millisClock() {
13.         return MILLIS_CLOCK;
14.     }
15.     private void scheduleClockUpdating() {
16.         ScheduledExecutorService scheduler =
    Executors.newSingleThreadScheduledExecutor(runnable -> {
17.             Thread thread = new Thread(runnable, THREAD_NAME);
18.             thread.setDaemon(true);
19.             return thread;
20.         });
21.         scheduler.scheduleAtFixedRate(() ->
22.         now.set(System.currentTimeMillis()), precision, precision,
    TimeUnit.MILLISECONDS);
23.     }
24.     public long now() {
25.         return now.get();
26.     }
27. }
```

图 2-15 SystemClock 程序

2.3 数据结构说明

本文中使用了哈希表的结构，但是没有使用传统的 HashMap，因为这种结构可能会形成环状链导致空转，无法在并发条件下使用。而 HashTable 虽然在并发条件下是安全的，但是它相当于在整个哈希表的结构上加了一把大锁，所有相关操作都是 synchronized，多线程访问时只允许一个线程访问或操作该对象，其他线程只能阻塞，效率较低。

因此本文使用 ConcurrentHashMap，这种容器中有多把锁，每一把锁锁住一段数据，这样在多线程访问数据时就不会存在较多的竞争了。ConcurrentHashMap 是由 Segment 数

组结构和 HashEntry 数组结构组成。Segment 是一种可重入锁 ReentrantLock，在 ConcurrentHashMap 里扮演锁的角色，HashEntry 则用于存储键值对数据。一个 ConcurrentHashMap 里包含一个 Segment 数组，Segment 的结构和 HashMap 类似，是一种数组和链表结构， 一个 Segment 里包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素， 每个 Segment 守护者一个 HashEntry 数组里的元素,当对 HashEntry 数组的数据进行修改时，必须首先获得它对应的 Segment 锁，具体结构如图 2-15 所示。

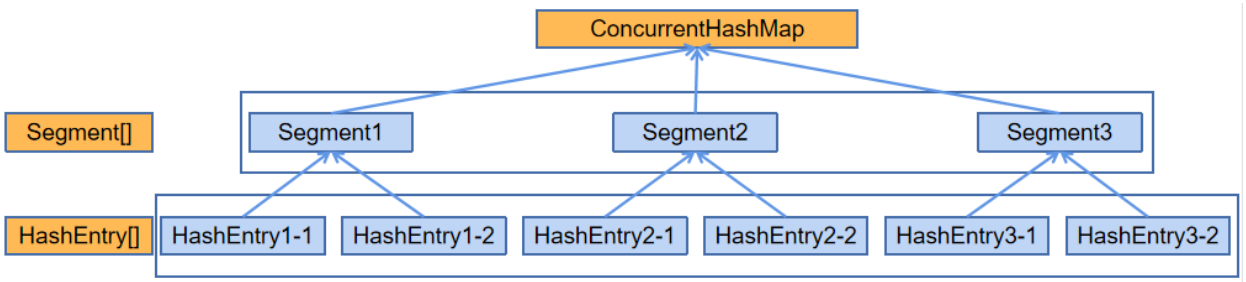

图 2-15 ConcurrentHashMap 结构

第 3 章 正确性分析与测试

3.1 正确性说明

用于列车售票的可线性化并发数据结构的要求可以参见 1.1 题目要求，本文将针对每一条要求提出解决方案：

（1） 每张车票都有一个唯一的编号 tid，不能重复：

在线程较少时，每次退票本文都会通过原语 getAndIncrement()生成一个递增的标签戳；在线程较多时，每次退票本文会通过额外线程获取并发时间戳。购票本文会根据时间戳或者标签戳、列次、出发站、到达站和座位编号生成一个唯一的编号。因此满足每张车票都有一个唯一的编号 tid，不能重复。

（2） 每一个 tid 的车票只能出售一次。退票后，原车票的 tid 作废。

在线程较少时，getAndIncrement()只会生成新值，因此车票不会二次售出；在线程较多时，时间戳会不断更新，再加对应的列次、出发站、到达站和座位编号，满足每个 tid 只出售一次。

（3） 每个区段有余票时，系统必须满足该区段的购票请求。

如果区段有票，buyTicket()的多段搜索机制会保证每一个可能卖出票的座位都被搜索到，因此必定满足有余票时能够售出车票。

（4） 车票不能超卖，系统不能卖无座车票。

如果一个区段的车票被卖光了，buyTicket()无法搜索到合适的车票会返回-1 代表没有车票可以被卖出。

（5） 买票、退票方法需满足可线性化要求。查询余票方法的约束放松到静态一致性。查询结果允许不精确，但是在某个车次查票过程中没有其 他购票和退票的“ 静止”状态下，该车次查询余票的结果必须准确。

买票和退票方法都采用 compareAndSet()进行操作，满足可线性化要求，具体验证可以查看第 2.2 正确性测试。而查询票方法针对每一个座位进行查询，满足静态一致性。

3.2 串行正确性测试

本文使用老师提供的 verify.sh 进行串行正确性测试。循环处理 100 轮，每次测试 1000 个数据，其参数如图 3-1 所示。

```
1.  final static int routenum = 3; // route is designed from 1 to 3
```

```
2.  final static int coachnum = 3; // coach is arranged from 1 to 5
3.  final static int seatnum = 3; // seat is allocated from 1 to 20
4.  final static int stationnum = 3; // station is designed from 1 to 5
```

图 3-1 正确性测试参数

最终结果如图 3-2 所示。

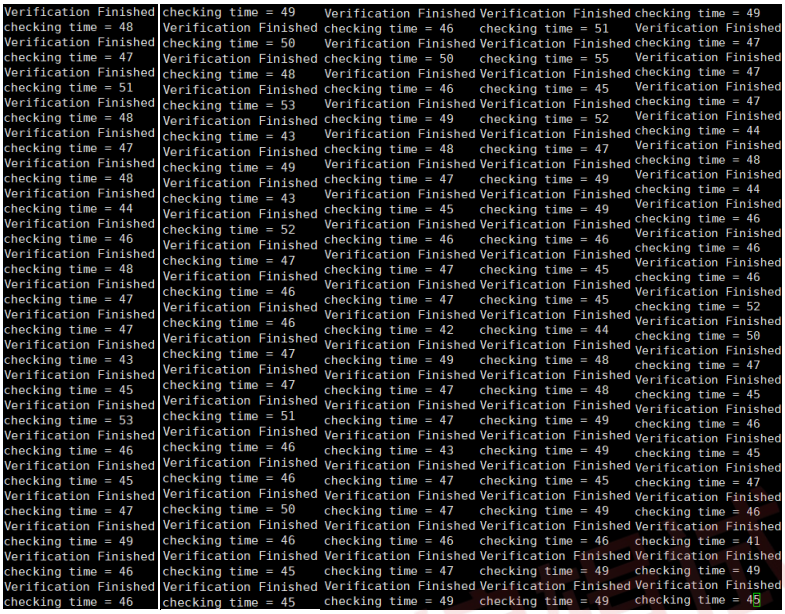

图 3-2 正确性测试结果

3.3 并发正确性测试

在这里为了验证模型在并发条件下是否可以线性化，本文采用已实现的验证工具对多线程验证工具进行验证。判断一段历史能否线性化就是观察该历史能否找到一段全序列，因此可以根据历史的偏序关系建立图。首先根据事件是否重叠建立单向或双向边，之后根据深度优先搜索查找合法的路径。如果存在合法的路径则认为可线性化，否则认为不能线性化，最终结果图如图 3-3 所示。

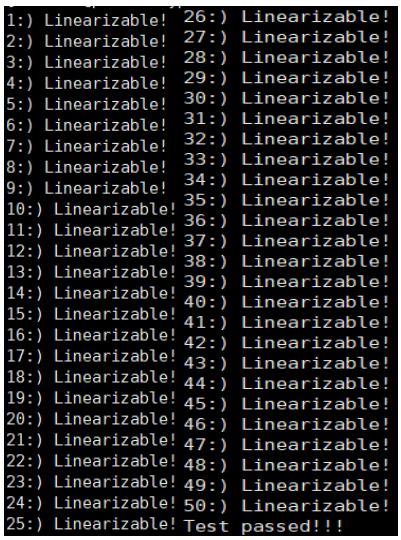

图 3-3 多线程可线性化测试结果

第 4 章 性能分析与测试

4.1 性能分析

4.1.1 购票

在每一时刻首先检查座位能否被占用，如果座位能够被占用，那么会构造被占用后的新座位。只有当新座位通过 compareAndSet()能够替换掉旧座位时，我认为此时才算购票成功。如果座位已经被其他购票者占用，座位会根据被占用后的座位还能否购票，进行更新 oldseat=currmap.get()或选择 i++来购买下一个座位，主要程序如图 4-1 所示，因此认为可线性化点为图 4-1 的第 6 行。

```
1.   for (int i = currnum; i <seatsum; i++){
2.         AtomicLong currmap=seatmap[i];
3.         oldseat=currmap.get();//座位占用情况
4.         while ((oldseat & coder) == 0){//座位未被占用
5.               newseat = oldseat | coder;//如果座位被占用
6.               if (currmap.compareAndSet(oldseat, newseat)){//执行成功说明购票成功
7.                     currpriorefund.compareAndSet(currnum,i+1);
8.                     return i;
9.               }
10.              oldseat=currmap.get();
11.        }
12. }
```

图 4-1 buyTicket()程序

根据上述描述可知，购票满足可线性化、无锁、无死锁，但是可能会存在饥饿，因为如果存在其他的购票或退票行为使得第 4 行的座位始终满足条件，那么这些行为会打断第 6 行的购票，使得购票不成功但是始终在循环。

4.1.2 退票

退票首先查询该座位是否被占用，即座位是否被购买，如果被购买则生成新座位 newseat，如果过能够通过 compareAndSet()更新 seatmap 则说明退票成功。当然如果座位被其他线程的相同乘客退票，则跳出循环，反之继续退票，具体程序可见图 4-2 所示。

```
1.         while ((oldseat & coder) == coder){//座位被占用
2.               newseat = (~coder) &oldseat;//如果座位未被占用
3.               if (currmap.compareAndSet(oldseat,newseat)) {//执行成功说明退票成功
4.                     int currnum = currpriorefund.get();
5.                     while (i<currnum) {//循环执行直到更小的座位被退
6.                           currpriorefund.compareAndSet(currnum, i);
7.                           currnum = currpriorefund.get();
8.                     }
9.                     return true;
```

```
10.             }
11.         oldseat = currmap.get();
12.         }
13.         return false;
```

图 4-2 refundTicket ()程序

当前最坏的情况是该座位始终再进行购票或买票，因此会不断打断退票的程序，所以退票满足可线性化、无锁、无死锁，但是可能会存在饥饿。

3.3.3 查询余票

查询余票需要对每个座位进行检测，在检测过程中之前检测过的座位，之后将检测的座位都可能会被购买或退票，因此查询余票满足静态一致性，其代码如图 4-3 所示。

```
1.  for (int i = 0; i < end; i++) {
2.          AtomicLong currcoder = seatmap[i];
3.          if ((currcoder.get() & coder) == 0)
4.              num++;
5.          }
6.  return num;
```

图 4-3 inquery ()程序

4.2 性能测试

test.sh 对多线程进行测试，valid.sh 包含预热对单线程进行测试，并且票务存在错误。

为了对性能结果形成对比，本文首先实现了对每个列次进行加锁，并与本设计进行对比，其中性能测试参数如图 4-4 所示，对每个列次加锁的性能结果图如图 4-5 所示，本设计性能结果图如图 4-6 所示，两者性能结果对比图如图 4-7 所示，平均性能提升 201.35%。

```
1.          final static int routenum = 5; // route is designed from 1 to 3
2.          final static int coachnum = 8; // coach is arranged from 1 to 5
3.          final static int seatnum = 8; // seat is allocated from 1 to 20
4.          final static int stationnum = 10; // station is designed from 1 to 5
5.          final static int testnum = 100000;
6.          final static int retpc = 10; // return ticket operation is 10% percent
7.          final static int buypc = 30; // buy ticket operation is 30% percent
8.          final static int inqpc = 100; //inquiry ticket operation is 60% percent
```

图 4-4 性能测试参数

```
[user035@panda7 myproject_1]$ ./test.sh
ThreadNum: 4 BuyAvgTime(纳秒): 15930 RefundAvgTime(纳秒): 11464 InquiryAvgTime(纳秒): 14127 ThroughOut(操作数/毫秒): 233
ThreadNum: 8 BuyAvgTime(纳秒): 19023 RefundAvgTime(纳秒): 17565 InquiryAvgTime(纳秒): 18323 ThroughOut(操作数/毫秒): 398
ThreadNum: 16 BuyAvgTime(纳秒): 47245 RefundAvgTime(纳秒): 47280 InquiryAvgTime(纳秒): 47419 ThroughOut(操作数/毫秒): 320
ThreadNum: 32 BuyAvgTime(纳秒): 95635 RefundAvgTime(纳秒): 90567 InquiryAvgTime(纳秒): 97743 ThroughOut(操作数/毫秒): 313
ThreadNum: 64 BuyAvgTime(纳秒): 716991 RefundAvgTime(纳秒): 708928 InquiryAvgTime(纳秒): 724674 ThroughOut(操作数/毫秒): 88
ThreadNum: 128 BuyAvgTime(纳秒): 574931 RefundAvgTime(纳秒): 571344 InquiryAvgTime(纳秒): 575821 ThroughOut(操作数/毫秒): 221
```

图 4-4 对每个列次进行加锁的性能

```
[user035@panda7 myproject_ok]$ ./test.sh
ThreadNum: 4 BuyAvgTime(纳秒): 2570 RefundAvgTime(纳秒): 3540 InquiryAvgTime(纳秒): 971 ThroughOut(操作数/毫秒): 1194
ThreadNum: 8 BuyAvgTime(纳秒): 2144 RefundAvgTime(纳秒): 3329 InquiryAvgTime(纳秒): 415 ThroughOut(操作数/毫秒): 2666
ThreadNum: 16 BuyAvgTime(纳秒): 1861 RefundAvgTime(纳秒): 993 InquiryAvgTime(纳秒): 2306 ThroughOut(操作数/毫秒): 2725
ThreadNum: 32 BuyAvgTime(纳秒): 386 RefundAvgTime(纳秒): 1293 InquiryAvgTime(纳秒): 839 ThroughOut(操作数/毫秒): 5047
ThreadNum: 64 BuyAvgTime(纳秒): 609 RefundAvgTime(纳秒): 1141 InquiryAvgTime(纳秒): 2467 ThroughOut(操作数/毫秒): 5643
ThreadNum: 128 BuyAvgTime(纳秒): 660 RefundAvgTime(纳秒): 8478 InquiryAvgTime(纳秒): 2369 ThroughOut(操作数/毫秒): 8398
```

图 4-4 本设计性能性能

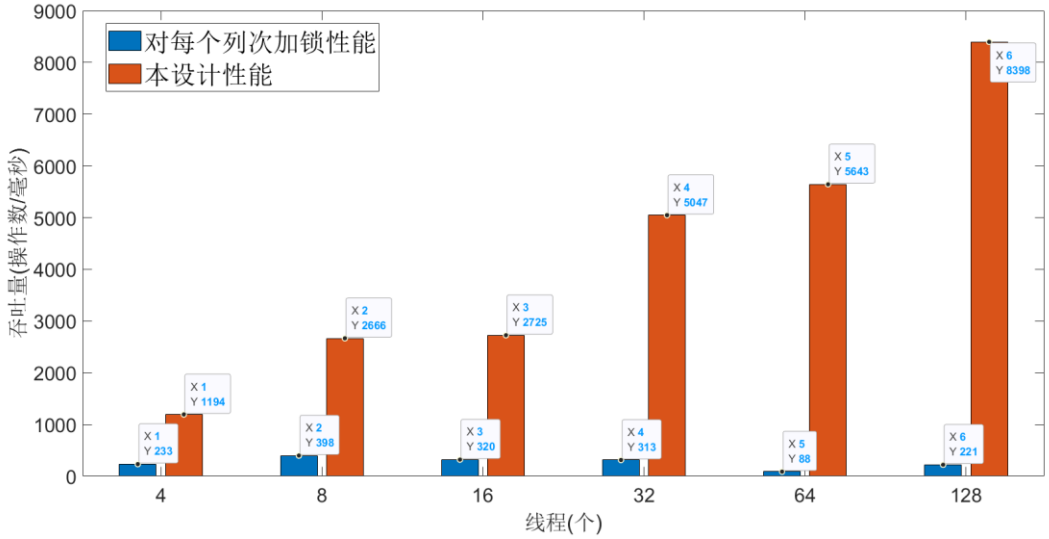

图 4-6 性能对比

4.3 测试说明

如表 4-1 为测试说明，其中 ticketingsystem1 为读写锁实现的基线设计，用于与本设计进行对比，本设计文件位于 ticketingsystem。

脚本文件	说明
trace.sh	验证模型正确性
valid.sh/test2.sh	测试模型在单一线程下的性能（包含预热）
test1.sh	测试模型在不同线程下的性能（不包含预热）
replay.sh	测试模型串行正确性
verify.sh	验证模型串行正确性
clean.sh	清除生成的类文件
check.sh	验证模型并发正确性

表 4-1 测试说明

第 5 章 总结与展望

5.1 总结分析

本文设计了一个用于列车售票的可线性化并发数据结构，该结构可以用于并发地购票、售票和查询余票。其中购票和售票满足线性一致性，查询余票满足静态一致性。本文通过了性能测试和正确性测试，并将性能测试结果与初步实现的单列车加锁的性能进行对比，平均吞吐量提升 201.35%。

5.1 展望

对于本模型，我认为还有可以进行改进的地方，具体如下：

（1） 本文的 tid 编码采用 getAndIncrement()，虽然满足无重复编码但是会极大降低性能，应该采用更好的办法解决。

（2） 对于票务的调度可以进一步优化，对于购票和退票的请求可以进一步优化。

（3） 可以使购票类似于入队、退票类似于出队，采用不同的标识符确定是否能够购票和退票。