课程大作业:用于列车售票的可线性化并发数据结构

并发数据结构与多核编程21-22秋季

主讲教师：林惠民 吴鹏 吕毅

朱奎

202128013329011

设计要求

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），返回无效的Ticket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。

每位学生使用Java语言设计并完成一个用于列车售票的可线性化并发数据结构：TicketingDS类，该类实现TicketingSystem接口，同时提供构造函数

```
TicketingDS ( routenum , coachnum , seatnum , stationnum ,
threadnum ) ;
```

其中，routenum是车次总数（缺省为5个），coachnum是列车的车厢数目（缺省为8个），seatnum是每节车厢的座位数（缺省为100个），stationnum是每个车次经停站的数量（缺省为10个，含始发站和终点站），threadnum是并发购票的线程数（缺省为16个）。为简单起见，假设每个车次的coachnum、seatnum和stationnum都相同。车票涉及的各项参数均从1开始计数，例如车厢从1到8编号，车站从1到10编号等。

为简单起见，假设每个车次的coachnum、seatnum和stationnum都相同。车票涉及的各项参数均从1开始计数，例如车厢从1到8编号，车站从1到10编号等。

每位学生需编写多线程测试程序，在main方法中用下述语句创建TicketingDS类的一个实例。

```
final TicketingDS tds = new TicketingDS ( routenum , coachnum ,
seatnum , stationnum , threadnum ) ;
```

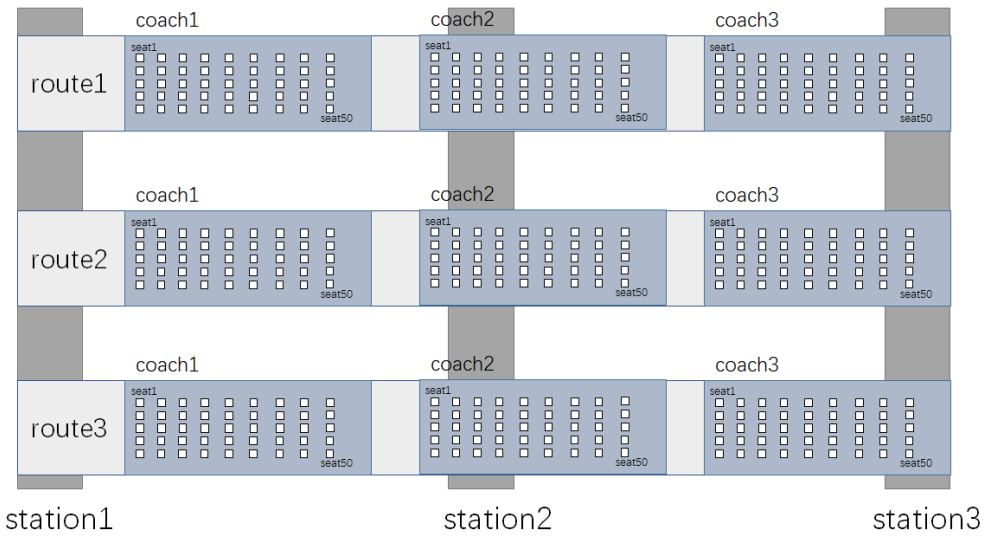
系统中同时存在threadnum个线程（缺省为16个），每个线程是一个票务代理，按照60%查询余票，30%购票和10% 退票的比率反复调用TicketingDS类的三种方法若干次（缺省为总共10000次）。按照线程数为4，8，16，32，64个的情况分别给出每种方法调用的平均执行时间，同时计算系统的总吞吐率（单位时间内完成的方法调用总数）。

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。
- 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

思路

结构设计



此示意图为routenum为3，coachnum为3，stationnum为3，seatnum为50的情况。

根据此关系，我们设计对应的类。

- TicketingDS

  实现TicketingSystem的**查票、买票、退票**方法。

```
//产生自增ticketID的原子变量
AtomicLong ticketID;
//记录已售出票的哈希表，用来检测退票的真实性
ConcurrentHashMap<Long, Ticket> mapTicket;
//记录某类余票剩余数量的哈希表，每次买票退票进行维护，用来快速查询
ConcurrentHashMap<Integer, LongAdder> remainTicketMap;
```

- Route

  车次类，包含车次对应的车厢数据结构。

```
Coach[] coachs;
```

- Coach

  车厢类，包含此车厢中的座位数据结构。

```
Seat[] seats;
```

- Seat

  座位类，提供**售出此座位的某两站区间、退还此座位的某两站区间、判断此座位在某两站间是否可用**的方法

```
//用来表示某站点到某站点的二进制数
AtomicInteger section;
```

区间二进制表示

格式

- 每个座位对应一个二进制串，用来表示一个座位在哪些车站区间是被占用的。通过二进制编码，可以通过位运算实现快速的区间检测，这样单次判断座位操作的时间复杂度可以从$O(stationnum)$降低到$O(1)$。
- 单个二进制位含义
  - 从**低到高第i位为1表示此座位从第i个站台到第i+1个站台的区间被占用**
  - 从**低到高第i位为0表示此座位从第i个站台到第i+1个站台的区间空闲**
- 二进制串存储方式

- 对于一个座位的二进制串举例，某个座位2--5被占用（2站台上车，5站台下车） 0...01110表示 1-2站空闲 2-3被占用 3-4空闲 4-5空闲...

位运算操作

- **判断某车站区间是否空闲**

```
(section.get() & (((1<<(arrival-1)) ) - (1<<(departure-1)))) == 0;
```

某个座位：

  - 0....011000：1-2站空闲 2-3站空闲 3-4站空闲 4-5被占用 5-6站被占用 7-8站空闲
  - 某用户2站上车，4站下车，想判断买票是否区间冲突：arrival = 2，departure = 4
  - (((1<<(arrival-1))) - (1<<(departure-1)))) 得到0....000110
  - 0....011000 & 0....000110 == 0 代表此座位2站上车，4站下车是可用的
- **占用某车站区间**

```
newSec = oldSec | (((1 << (arrival - 1))) - (1 << (departure - 1)))
```

某个座位：

  - 0....011000：1-2站空闲 2-3站空闲 3-4站空闲 4-5被占用 5-6站被占用 7-8站空闲
  - 某用户2站上车，4站下车，想要买票：arrival = 2，departure = 4
  - (((1<<(arrival-1))) - (1<<(departure-1)))) 得到0....000110
  - 0....011000 | 0....000110 == 0....011110 在之前的基础上，标记了此座位2站上车，4站下车的区间被占用
- **标记某车站区间空闲**

```
newSec = oldSec & (~(((1<<(arrival-1)) ) - (1<<(departure-1))))
```

某个座位：

  - 0....011110：1-2站空闲 2-3站被占用 3-4站被占用 4-5被占用 5-6站被占用 7-8站空闲

- 某用户2站上车，4站下车，退票，解除对此车票的2-4车站区间的占用：arrival = 2，departure = 4
- (((1<<(arrival-1))) - (1<<(departure-1)))) 得到0....000110
- 0....011110 | ~0....000110 == 0....011000 标记了此座位解除了2站上车，4站下车的区间占用

买票

步骤

- 在buyTicket方法中，首先**调用inquiry查询是否存在此路线的余票**。若余票已经不存在，则不需要去查找合适的座位，直接返回null。
- 为了减轻多个线程都按照顺序访问座位所产生的带来的争用问题，设计让每一个线程**随机去访问车厢中的座位**，即每个线程随机产生要查找的车厢号和座位号，然后以此为起点，直到遍历所有的车厢和座位，找到区间空闲的就对此座位执行买票操作。
- 买票操作采用位运算的方法，实现快速的区间操作。同时用CAS操作，实现**lockfree**的买票操作。
- 买票成功则产生Ticket对象返回给用户，并且将此对象存入记录车票的哈希表（用来判断退票操作是否是假票）。
- 买票操作后，记录变化，维护一个记录余票的哈希表（为了实现快速查询余票）。

TicketingDS类

```
public Ticket buyTicket(String passenger, int route, int
departure, int arrival) {
    if(inquiry(route,departure,arrival) == 0){
        return null;
    }
    int randCoachNo =
ThreadLocalRandom.current().nextInt(1,coachnum);//随机产生coach
查找起始点
    int randSeatNo = ThreadLocalRandom.current().nextInt(1,
seatnum);//随机产生seat查找起始点
    int i = randCoachNo;
    do{
        i = i % coachnum + 1;
        int j = randSeatNo;
        do {
```

```
            j = j % seatnum + 1;
            if
(routes[route].coachs[i].seats[j].isFree(departure, arrival)) {
                BinaryChange outcome =
routes[route].coachs[i].seats[j].sell(departure, arrival);
                if (outcome != null) {
                    decreaseRemainTicket(route, outcome);//更新
余票哈希表
                    return issueTicket(passenger, route, i, j,
departure, arrival);
                }
            }
        }while (j != randSeatNo);
    }while (i != randCoachNo);
    return null;
}
```

Seat类

```
public BinaryChange sell(int departure, int arrival) {
        while (true){
            int oldSec = section.get();
            //判断如果被占用了，则退出循环
            if((oldSec & (((1<<(arrival-1)) ) - (1<<(departure-
1)))) != 0){
                return null;
            }
            //cas操作成功则返回成功
            int update = oldSec | (((1 << (arrival - 1))) - (1
<< (departure - 1)));
            if(section.compareAndSet(oldSec, update)) {
                return new BinaryChange(oldSec, update);
            }
        }
    }
```

判断section是否发生变化，若未被其他线程提前占用，且CAS操作断定Section未发生，则可以修改section标记departure到arrival的区间占用，买票操作执行成功

退票

步骤

- 首先使用checkTicket(ticket)方法，通过ConcurrentHashMap判断是否售出过此票。

```
public boolean checkTicket(Ticket ticket){
    Ticket ticket1 = mapTicket.get(ticket.tid);
    if(ticket1 == null)
        return false;
    return ticket.departure == ticket1.departure &&
            ticket.arrival == ticket1.arrival &&
            ticket.seat == ticket1.seat &&
            ticket.route == ticket1.route;

}
```

- 若此票不是假票
    - 从ConcurrentHashMap中删除此票的对象（并发安全）。
    - 直接用CAS操作，标记车票对应区间变为空闲（不冲突）。
- increaseRemainTicket刷新余票表

TicketingDS类

```
public boolean refundTicket(Ticket ticket) {
    if(!checkTicket(ticket))//找不到票
        return false;
    mapTicket.remove(ticket.tid);
    BinaryChange outcome =
routes[ticket.route].coachs[ticket.coach].seats[ticket.seat].re
fund(ticket.departure,ticket.arrival);
    increaseRemainTicket(ticket.route, outcome);//刷新余票
    return true;
}
```

Seat类

```
public BinaryChange refund(int departure, int arrival) {
        while (true){
            int oldSec = section.get();
            int newSec = oldSec & (~(((1<<(arrival-1)) ) -
(1<<(departure-1))));
            //cas操作成功则返回成功
            if(section.compareAndSet(oldSec, newSec)) {
                return new BinaryChange(oldSec, newSec);
            }
        }
}
```

查票

思路

- 为了优化查询速度，建立一张记录余票变化的表remainTicketMap，初始时任意车次任意座位任区间的**余票变化量为0，初始余票数为**$coachnum \times seatnum$
- remainTicketMap中的Key为二进制串，将之前的二进制串左移三位，留出前3位存储车厢号，用来表示某车次某车厢某座位的某上下车区间，Value为对应的余票变化量。
- 对于记录余票变化量的数据结构，本可以采用AtomicLong当作并发计数器，但是考虑到高并发情况下，理想的方式应争用较低，解决访问共享对象时的顺序瓶颈问题。在jdk8中，引入了**LongAdder**，非常适合多线程原子计数器。对于此系统，LongAdder在多线程环境中，原子自增长性能要好很多。

  ```
  ConcurrentHashMap<Integer, LongAdder> remainTicketMap;
  ```

- 查票时，直接从remainTicketMap查出余票变化量，用初始余票数为$coachnum \times seatnum$减去变化量，返回当前余票。

更新余票的方法

```
//计算某座位购买后， 会引起哪些类型余票数量的减少
    public void decreaseRemainTicket(int route, BinaryChange
binaryChange){
        //算法：找出哪些类型的票在此票购买前是可买的，购买后导致此票
不能买，对相应类型车票余票数-1
        for(int departure = 1; departure <= stationnum-1;
departure++){
            for(int arrival = departure + 1; arrival <=
stationnum; arrival++) {
                //若退票前是不可买的，退票后导致此类票可买
                int thisKind =
TicketingDS.stationToBinary(departure, arrival);
                if (((thisKind & binaryChange.before) == 0) &&
((thisKind & binaryChange.after) != 0)){

 remainTicketMap.get(routeAndBinaryToKey(route,
thisKind)).decrement();//减少余票
                }
            }
        }
    }
//计算某座位退退票后，引起哪些类型余票数量的增加
    public void increaseRemainTicket(int route, BinaryChange
binaryChange){
        //算法：找出哪些类型的票在此票退票前是不可买的，退票后导致此
类票可买，对相应类型车票余票数+1
        for(int i = 1; i <= stationnum-1; i++){
            for(int j = i + 1; j <= stationnum; j++) {
                int thisKind = TicketingDS.stationToBinary(i,
j);//某一种类型的票
                //若退票前是不可买的(二进制串冲突)，退票后导致此类
票可买
                if (((thisKind & binaryChange.before) != 0) &&
((thisKind & binaryChange.after) == 0)){

 remainTicketMap.get(routeAndBinaryToKey(route,
thisKind)).increment();//增加余票
                }
```

```
            }
        }
    }
```

查询余票方法

```
public int inquiry(int route, int departure, int arrival) {
    //seatnum*coachnum是初始座位数量 + 表中记录的变化量 即为余票
    return seatnum*coachnum +
remainTicketMap.get(routeAndStationToKey(route, departure,
arrival)).intValue();
}
```

正确性

题目要求的正确性

- 每张车票都有一个唯一的编号tid,不能重复

  AtomicLong方法产生余票，其getAndIncrement原理是，不断的loop循环判断，当前值有没有在累加操作前被其他线程修改了，如果没改就赋值，改了就重新累加，再判断赋值。从而形成了类似同步的机制。保证变量的原子性。

- 每一个tid的车票只能出售一次。退票后,原车票的tid作废。

  采用哈希表存储产生过的车票，退票时进行判断，是否存在，且车票信息是否一致。判断成功后从表中移除此票。

- 每个区段有余票时，系统必须满足该区段的购票请求。

  由于区间的表示方法可以精确地确定某车次某车厢某座位的某些区间是否可售，且买票的时候会遍历到每一个座位，判断是否可以购买，所以会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。

  余票查询和车票空闲检测防止了超卖和无座车票现象。

- 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。

见后文**查票**正确性。

买票

使用CAS操作实现的并发购票操作

```
while (true){
            int oldSec = section.get();
            //判断如果被占用了，则退出循环
            if((oldSec & (((1<<(arrival-1)) ) - (1<<(departure-
1)))) != 0){
                return null;
            }
            //cas操作成功则返回成功
            int update = oldSec | (((1 << (arrival - 1))) - (1
<< (departure - 1)));
            if(section.compareAndSet(oldSec, update)) {
                return new BinaryChange(oldSec, update);
            }
}
```

- 先获取目前的区间状态，判断此区间是否被其他购票者提前占用，若已占用则退出循环，继续访问其他的座位。
- CAS操作，section未被改变，则可以修改section，标记上departure到arrival的区间占用，买票操作执行成功。此方法不会产生死锁，调用购票方法的线程中至少有一个最终会返回，购票成功，满足Lock-free。

退票

```
while (true){
            int oldSec = section.get();
            int newSec = oldSec & (~(((1<<(arrival-1)) ) -
(1<<(departure-1))));
            //cas操作成功则返回成功
            if(section.compareAndSet(oldSec, newSec)) {
                return new BinaryChange(oldSec, newSec);
            }
}
```

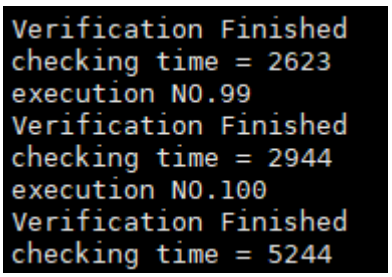
- 并发退票操作比买票操作要更容易，因为修改区间取消占用并不会导致冲突。
- 所以不需要进行占用判断，直接进行CAS修改区间即可，不会产生死锁。调用退票方法的线程中至少有一个最终会返回，退票成功，满足Lock-free。

查票

- ConcurrentHashMap是线程安全的，多个线程在获取同一个LongAdder对象，改变余票数量的方法也是可并发的。
- 对于并发计数器LongAdder：在最初无竞争时，只更新base的值，当有多线程竞争时通过分段的思想，让不同的线程更新不同的段，最后把这些段相加就得到了完整的LongAdder存储的值。
- 其满足静态一致性。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，可以得到准确的余票结果。
- 无等待（Wait-free）：调用查询方法的所有线程最终都会返回

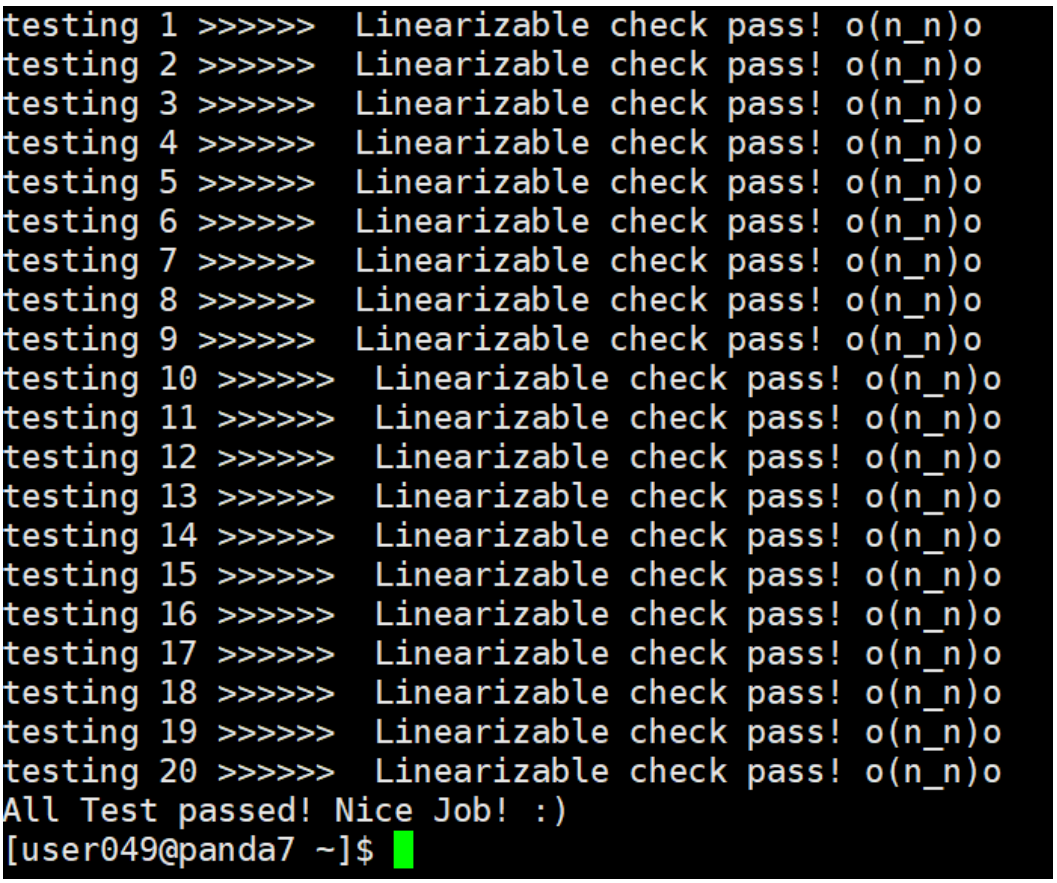
串行测试

运行Verify 100次，每轮调用10000个方法，通过串行正确性检验



可线性化测试

借用张逸舟同学实现的可线性化测试工具checker.jar运行20次，通过可线性化检验(正确性不确定)



> *使用DFS找一条事件发生的合法路径，如果事件A的结束时间先于B的开始时间，建立A到B的单向边；如果事件A和B时间重叠，则建立一条双向边，如果能从没有父节点的“孤儿”节点开始，能找到一条包含所有事件的合法路径，则是可线性化。根据题目要求，在检测的时候只考虑了trace记录里的买票、退票方法*
>
> *——张逸舟*

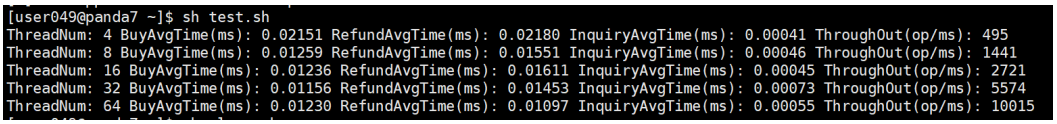
性能

对售票系统进行性能评价，按照60%查询余票，30%购票和10% 退票的比率反复调用TicketingDS类的三种方法若干次（缺省为总共10000次）。按照线程数为4，8，16，32，64个的情况分别给出每种方法调用的平均执行时间，同时计算系统的总吞吐率（单位时间内完成的方法调用总数）。

测试结果

256线程 512GB　服务器

```
[user049@panda7 ~]$ sh test.sh
ThreadNum: 4 BuyAvgTime(ms): 0.02151 RefundAvgTime(ms): 0.02180
InquiryAvgTime(ms): 0.00041 ThroughOut(op/ms): 495
ThreadNum: 8 BuyAvgTime(ms): 0.01259 RefundAvgTime(ms): 0.01551
InquiryAvgTime(ms): 0.00046 ThroughOut(op/ms): 1441
ThreadNum: 16 BuyAvgTime(ms): 0.01236 RefundAvgTime(ms):
0.01611 InquiryAvgTime(ms): 0.00045 ThroughOut(op/ms): 2721
ThreadNum: 32 BuyAvgTime(ms): 0.01156 RefundAvgTime(ms):
0.01453 InquiryAvgTime(ms): 0.00073 ThroughOut(op/ms): 5574
ThreadNum: 64 BuyAvgTime(ms): 0.01230 RefundAvgTime(ms):
0.01097 InquiryAvgTime(ms): 0.00055 ThroughOut(op/ms): 10015
```



线程数	购买平均时间(ms)	退票平均时间(ms)	查询平均时间(ms)	吞吐量(op/ms)
4	0.02151	0.02180	0.00041	495
8	0.01259	0.01551	0.00046	1441
16	0.01236	0.01611	0.00045	2721
32	0.01156	0.01453	0.00073	5574
64	0.01230	0.01097	0.00055	10015