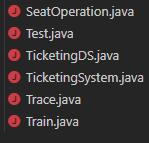
用于列车售票的可线性化并发数据结构

软709 冯欣宇 202028015059010

1. 系统设计与实现

项目整体目录为：



其中，TicketingSystem.java是规范文件，Trace.java是trace生成程序，用于正确性验证，TicketingDS.java是并发数据结构的主要实现，Train.java为火车类及相关方法，SeatOpreation用于维护座位余座数量的相关方法，Test.java为测试类。

1. 首先是TicketingDS.java的相关设计，维护了车次数目辆火车（Train.java类），

并且采用了ConcurrentHashMap类来保存和维护已售出车票，用于退票的合法性验证，能够提高效率。类中的主要方法有buyTicket，inquiry，和refundTicket。

1. 调用buyTicket方法时，首先采用hash的方法随机生成一个车厢号和座位号，用于尝试购票，然后调用Train类的trainBuyTicket，若成功，则将票的tid原子性的加1，返回票的对象，并存入soldtickets中，否则返回空。
2. 调用inquiry方法时，调用Train类的ticketinquiry方法，返回正确的票余数。
3. 调用refundTicket方法时，首先利用票的tid在soldtickets中查找相同的票对象，若存在且相同，则调用Train类的refundTicket方法并返回true，否则返回false。
4. 然后是Train类的相关方法实现

每一个火车对象需要维护车厢数，座位数，车站数，总座位数，以及本项目最核心的两个属性，AtomicLong[] seats以及SeatOperation seatops。

首先对AtomicLong[] seats进行解释，对于火车上的一个座位我们用一个原子类型的long型变量来存储其在所有车站之间是否被占用了，由于long型的长度是64bit，因此可以存储64个boolean类型的变量，比如，如果车站数目是4，则我们需要存储1→2，2→3，3→4，共三个boolean类型的变量，我们可以用一个long型变量来代替，若seats = 0 = 0x000，则均未被占用，若seats = 1 = 0x001，则代表3→4被占用了，其余没有被占用，并且由于seats是原子的，能够保证线程安全。采用long类型来存储的主要好处是我们可以利用位运算来大大加快效率。

然后对SeatOperation seatops进行解释。

对于一个查询，购票，买票都需要可线性化的系统，并且查询的操作是最多的，自然的想法就是需要把查询操作做的足够快，因此，我们采用一个余票表的结构seatops来维护这辆火车中，任意两站之间的余票数，这样，就可以做到查询只需要调用数组中对应的值即可，效率大大提高了，而如果查询，购票，退票，对余票表的操作都是原子化的，自然也就做到了可线性化，具体后面再阐述。

介绍完AtomicLong[] seats以及SeatOperation seatops后，Train类中的方法为：

1. trainBuyTicket方法：

首先调用seatops中的hasRemainSeat来查询是否还有余票，若有，则从拿到的随机车厢，随机座位开始遍历，查询每一个座位上的seats是否被占用了，若成功找到了未被占用的，则原子性的将其设为占用的，再调用seatops的refreshSeatNum函数，来刷新对应区间的余票数。

1. trainRefundTicket方法：

利用拿到了票的信息，得到需要操作的座位的seats对象，原子性的将其设为未被占用的，再调用seatops的refreshSeatNum函数，来刷新对应区间的余票数。

1. 然后是本项目的核心并发数据结构，SeatOperation类的实现

利用SeatOperation类，我们可以做到整个系统是无锁的，并且可线性化。

SeatOperation类中主要维护的变量有：

AtomicStampedReference<int[][]> ramainseatsptr；

ThreadLocal<int[][]> LocalRamain；

ramainseatsptr是一个原子指针，指向了当前正确的余票表。LocalRamain是ThreadLocal类型的指针，指向了每个线程所自己持有的一个余票表。初始化时，首先将ramainseatsptr指向一个int[][]数组，其中存储着车站之间的余票信息，自然的，初始值为allseatnum = coachnum \* seatnum。

类中的主要方法有：

1. getCurrentSeatsNum方法，查询当前余票。

采用while循环+CAS原语的方法，首先得到当前的ramainseatsptr指针，并获取其中的余票信息，然后CAS操作判断当前指针和时间戳是否发生改变了，若改变了，则重新读取，若没有，则返回。

1. hasRemainSeat方法，直接调用getCurrentSeatsNum即可
2. refreshSeatNum，刷新余票信息，需要的参数有long oldseat, long newseat, boolean ops。其中，oldseat为刷新前对应座位的seats变量（AtomicLong类型），newseat为刷新后对应座位的seats变量（AtomicLong类型）,ops为操作类型，true时为买票，false时为退票。

当需要刷新余票时，采用的也是while循环+CAS原语的方法，首先得到当前的ramainseatsptr指针，并获取其中的余票信息，然后根据其余票信息，在线程本地的LocalRamain中，利用位运算判断哪些区间发生了改变，生成新的余票信息，然后用CAS操作原子性的将ramainseatsptr设为线程自己的LocalRamain，若失败，说明已经发生了改变，需要重新获取ramainseatsptr并计算。若成功，则退出。这里需要注意一个问题是，当ramainseatsptr已经指向线程自己的LocalRamain时，不能直接在自己的LocalRamain上进行余票信息的改变，否则其他线程读取的时候可能会引发线程不安全，需要重新新建一个数组进行改变，设置成功后再将其设为LocalRamain。

1. 本项目是lock-free且可线性化的

查询，买票，退票的操作都是采用CAS原语对ramainseatsptr原子性的改变，自然是lock-free的，但不是wait-free的，因为在while循环+CAS原语的过程中，可能会有线程一直CAS失败，但这种可能性极低。

可线性化的，由于三种方法都是对同一变量的CAS操作，自然是可线性化的，可线性化点为各自的CAS操作。

1. 性能测试

测试环境：

课程所提供的128核服务器

java8

routenum = 20;

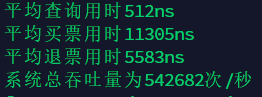
coachnum = 10;

seatnum = 100;

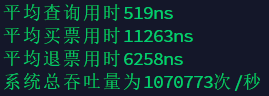
stationnum = 16;

testnum = 100000;

4线程时：



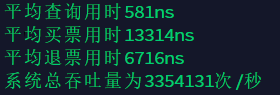
8线程时：



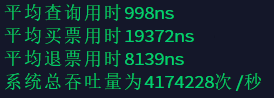
16线程时：



32线程时：



64线程时：



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 线程数 | 查询(ns) | 买票(ns) | 退票(ns) | 吞吐量(次/秒) | 吞吐量/线程数 |
| 4 | 512 | 11305 | 5583 | 542682 | 135671 |
| 8 | 519 | 11263 | 6258 | 1070773 | 133847 |
| 16 | 502 | 10326 | 5096 | 2025636 | 126602 |
| 32 | 581 | 13314 | 6716 | 3354131 | 104817 |
| 64 | 998 | 19372 | 8139 | 4174228 | 65222 |

可以看到，实际测试的性能还是比较理想的，单线程吞吐量直到64线程时由于争用提升明显，CAS失败率变高，才有明显下降。总体比较符合预期。