并发大作业性能评测报告

一、设计思路

(a) 数据结构的设计

TicketingDS 数据结构整体设计是将买票、查询、退票的操作按照车次号分配给下层对应的 mRoute 数据结构进行处理，并根据 mRoute 相应方法的返回情况来决定向上层的返回。

对于基准测试程序给定的车次数量 routeNum，TicketingDS 在初始化时将建立 routeNum 个 mRoute 对象：

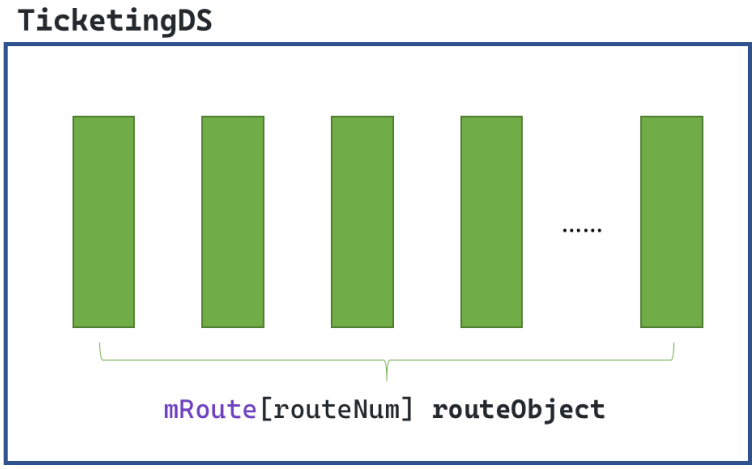

图 1　ticketingDS 数据结构示意图

为记录售出的票并在退票时进行快速检查，创建了两个 Array 分别作为“tid-ticket 信息”和“tid-passenger”的 hashmap，在初始化时根据测试的输入指定一个足够大的 tid 上界 MAX_TID，并为这两个 Array 都分配 MAX_TID * sizeof(type)的空间。

这种设计较之于 concurrentHashmap，优势在于初始化时就分配好该功能需要的全部内存空间，从而减少了系统运行过程中的内存申请、内存释放，劣势在于可能会较多冗余空间。鉴于测评机的内存资源充足，此处认为这种“空间换时间”是可以接受的。

mRoute 数据结构记录了线路编号、线路数量、车厢数量、车厢容量、车次容量、车站数量等基本信息，同时还用一个 AtomicLong 记录了本车次当前分配的 tid.

为了尽可能减少 tid 分配带来的线程竞争，同时又保证 tid 的唯一性，对于车次 k，它的任意 tid（不失一般性记为 n）必然满足 $n \bmod routeSize = k$.

mRoute 数据结构的核心成员是 routeRecord. 该数据结构整体设计如图：

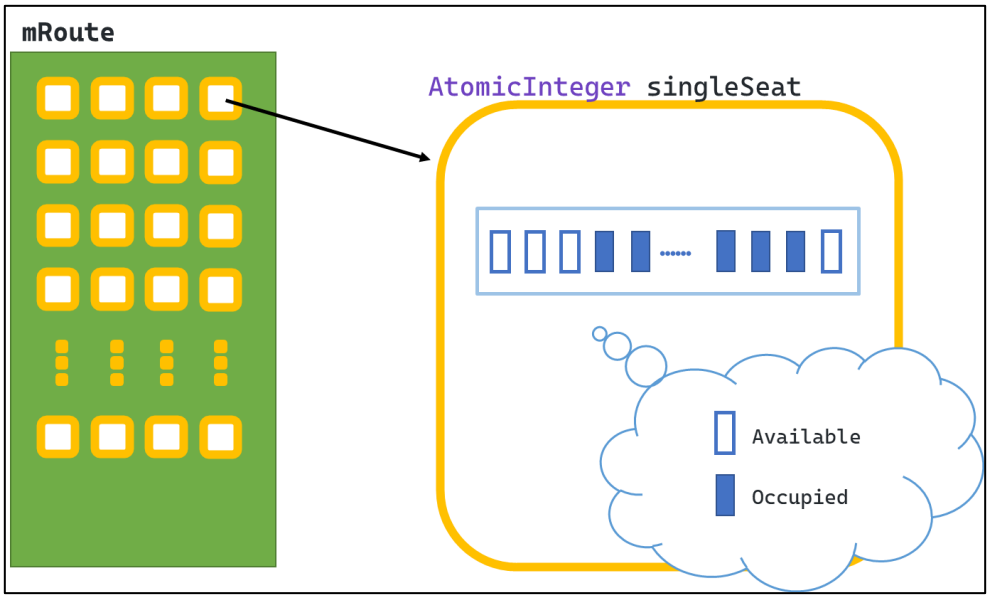

图 2　mRoute 数据结构示意图

上图中具体展示了routeRecord数组的结构，该数组的长度是一趟车次的座位总数，数组的元素是 AtomicInteger. 对该 AtomicInteger 元素中的某一 bit，其值为 0 代表了某一座位在对应区间的车票未被售出，其值为 1 代表了某一座位在对应区间的车票已被售出或该区间对于当前情形不可达。由于不可达和已售出实际上的效果是相同的，故可以归一化设计，从而使标识空闲区间的 bitmap 的设计趋于精简。

鉴于 routeRecord 已经采用 atomic 类型，如果只考虑买票和退票的过程，则并不需要显式加锁即可保证可线性化。但是，此时不加锁只能实现静态一致的查询。要想实现查询的可线性化，则必须限制“读类操作”和“写类操作”不同时出现。这个要求与读写锁很相似但不完全相同，因为本实现中对原子类型的使用保证了“多写”的合法性。

出于以上考虑，设计中单独实现了一个简单的、允许多读多写的读写锁 mReadWriteLock. 在这个锁中，读和写的地位是平等的，只是两者不能同时出现而已。该锁内部采用 TTAS 的思路来管理锁的获取和释放。以 mReadWriteLock 内部的读锁类为例：

```
class ReadLock implements Lock {
    public void lock() {
        while (true) {
            int res = counter.get();
            if (res >= 0) {
                if (counter.compareAndSet(res, res + 1)) {
                    break;
                }
            }
        }
    }

    public void unlock() {
        counter.getAndDecrement();
    }
```

图 3　mReadWriteLock 的内部类 ReadLock

成员 counter 记录当前持有读锁/写锁的线程数，记 counter 的绝对值为 c，则正值代表有 c 个线程持有读锁，负值代表有 c 个线程持有写锁。

(b) 测试程序的设计

测试程序提供了模拟作业要求的基础测试步骤，按照给定的车次车辆信息、各操作占比、测试操作数等信息进行测试，使用 System.currentTimeMillis() 方法计时并在最后输出各操作平均用时以及吞吐量数据。

二、系统正确性分析

以下将对任务提出的每个正确性要求做出分析。

(a) 每张车票都有一个唯一的编号 tid，不能重复

首先，根据前面已经提到的设计细节，不同车次之间的 tid 在模 k 的情形下不可能相等（k是车次数量）。

其次，同一车次的 tid 采用原子操作单调自增，也就不可能有重复的情况出现。

(b) 每一个 tid 的车票只能出售一次；退票后，原车票的 tid 作废

在 (a) 中提到过 tid 原子操作单调递增，不存在复用的情况，故该要求亦满足。

(c) 每个区段有余票时，系统必须满足该区段的购票请求

购票时，经过下图中红框部分的检查确认为某座有票，此时 CAS 返回 true，这意味着新的区间座位信息已经更新，并且程序根据该座号成功出票。

```
for (int fakeSeatTag = rand.nextInt(routeSize),
        end = fakeSeatTag + routeSize; fakeSeatTag != end; fakeSeatTag++) {
    while (true) {
        int seatTag = fakeSeatTag % routeSize;
        int seatRecord = routeRecord.get(seatTag);
        if ((seatRecord & tourCover) != 0) {
            // seat is already occupied
            break;
        }

        if (routeRecord.compareAndSet(seatTag, seatRecord, seatRecord | tourCover)) {
            // successfully add a ticket
            Ticket t = getTicket(passenger, seatTag, departure, arrival);
            writeLock.unlock();
            return t;
        }
    }
}
```

图 4 mRoute 中的 buyTicket 方法片段

(d) 车票不能超卖，系统不能卖无座车票

同上，由于 routeRecord 的原子性，车票售卖遵循可线性化要求，每一次出售都不会售出非法票。

(e) 买票、退票和查询余票方法都需满足可线性化要求

由于买票和退票的CAS操作均在持有写锁后才有可能进行，因此买票和退票方法的可线性化点均在 CAS 操作完成对 routeRecord 修改的时刻；查询余票方法的可线性化点在获取到读锁的时刻。

综上，3 个方法均满足可线性化要求。

三、系统性能分析

这是一个 deadlock-free 的系统。首先在实现 3 个方法均可线性化时使用读写锁，因此这个系统是阻塞式的。其次，这个系统中总是存在能够获得锁的线程，这说明每一时刻总有线程可以获得进展。

尽管该系统在理论上不能保证无饥饿，但由于读操作和写操作的比例较为均衡(6:4)，实际上很难发生饥饿的情形。可以想象这样的情境：一共有 64 个线程正在工作，读写锁的 counter 值为 0，其中 35 个线程申请了读锁，另外的 29 个线程申请了写锁，并且最先申请到的锁是读锁，则这 35 个线程逐个完成了读操作后又被分配了新的操作，……直至最初的这 35 个线程中所有线程的读操作结束，则所有线程都在申请写锁并且都能够申请到写锁。最开始申请到读锁的线程一直都被分配读操作的可能性是极小的，并且实地测试为读写锁添加防饥饿设置并不会对性能造成显著影响。

四、测试展示

测试配置:

- 列车数= 50，每趟车的车站数=30，每趟车的车厢数=20，每个车厢的座位数=100;
- 每个线程 100 万条操作，比例分别为买票 30%，退票 10%，查票 60%

分别测试线程数为 4、8、16、32、 64 的情况，以下为命令行原始输出:

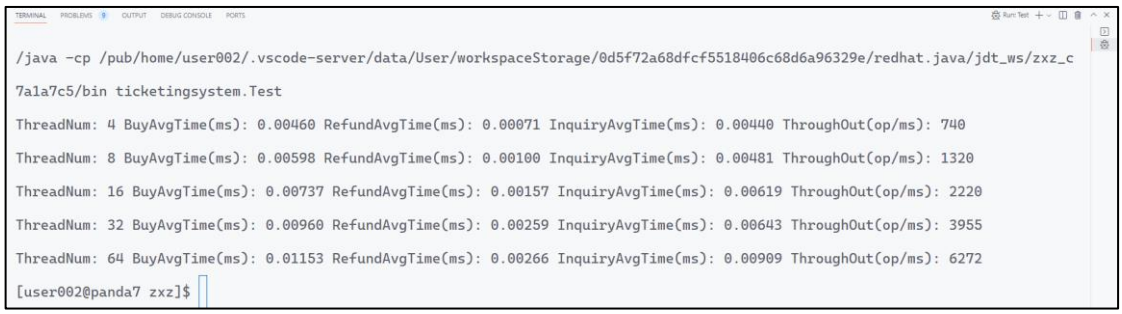

图5 测试命令行原始输出

以下是测试结果的具体数值：

测试线程数	buyTicket操作平均执行时间(ms)	refundTicket操作平均执行时间(ms)	inquiry操作平均执行时间(ms)	总吞吐量(ops/ms)
4	0.0046	0.00071	0.0044	740
8	0.00598	0.001	0.00481	1320
16	0.00737	0.00157	0.00619	2220
32	0.0096	0.00259	0.00643	3955
64	0.01153	0.00266	0.00909	6272

表 1　测试结果

作出下图，分析吞吐量结果与并发线程数之间的关系，可知在测试使用的线程数范围内，线程数的增加能够提升吞吐量，且整体呈线性关系（拟合得到的线性方程为 $y = 91.244x + 638.54$，判定系数 $R^2 = 0.9858$），这符合该设计的预期和相关理论知识。

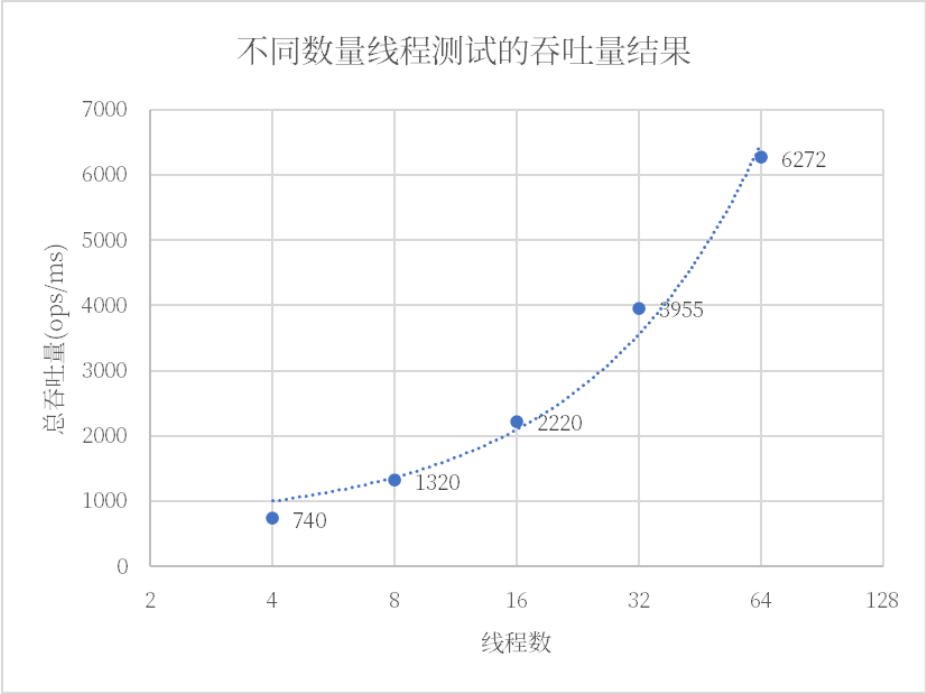

图 6 不同数量线程测试的吞吐量结果

五、总结与讨论

Q：为什么没有考虑将读写锁的粒度缩小?

补充：目前读写锁会锁住整个记录，将读写锁粒度缩小即为每两个相邻车站之间的路段设一单独的读写锁，这样如果同一车次有两个不相交的退票/买票操作，则两者可以不冲突地并发完成，提升并发度。

A：尝试过这一方案，发现反而导致了性能的下降，分析原因是修改记录采用 CAS 操作，在本设计中，无论锁粒度几何，最终座位的购票记录都要由 CAS 操作进行原子地修改，因此对性能提升不多；此外，线程越多，能够享受到这一改进优势的情形就越少，而由于锁数量的增加，锁的开销反而会成为性能负担。

Q：为什么对 ticketsPassengers 数据结构使用 volatile 限定符?

A：该数据结构的更新不依赖于旧值，故不需要 atomic 特性；但是由于退假票的可能性，该数据结构仍然需要满足某种形式的互斥，“允许多个线程读、一个线程写”的 volatile 特性恰好能较为圆满地解决这个问题。

Q：为什么将 MAX_TID 设置为 threadNum * 102000 + routeNum * 30300?

A: 根据测试配置和估算得到，退票操作数暗示了购票成功的可能次数，买票相当于组合数学中的插板模型，可以根据这两方面的信息估算出一个有效并且实用的购票上界。不开固定长度的数组是因为，如果申请的内存空间有较多冗余，也会增加不必要的访存开销。

Q：为什么 mReadWriteLock 采用类似于 TTAS 的设计(自旋+CAS)，而非传统读写锁中的管程锁?

A：实现过程中两种方案都尝试过，采用类 TTAS 设计的 mReadWriteLock 性能比采用管程锁时提升了约 50%-100%，分析原因是管程锁涉及较多系统中断操作，在系统调用方面的开销大于类 TTAS 的设计。