课 程：并发数据结构与多核编程

任课教师：林惠民、吕毅、吴鹏

用于列车售票的可线性化并发数据结构

—— 评测报告

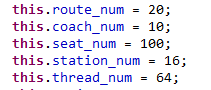
姓 名：韩路通

学 号：2020E8018682108

培养单位：中国科学院信息工程研究所

**一、并发数据结构的设计思路**

1、座位的表示

****

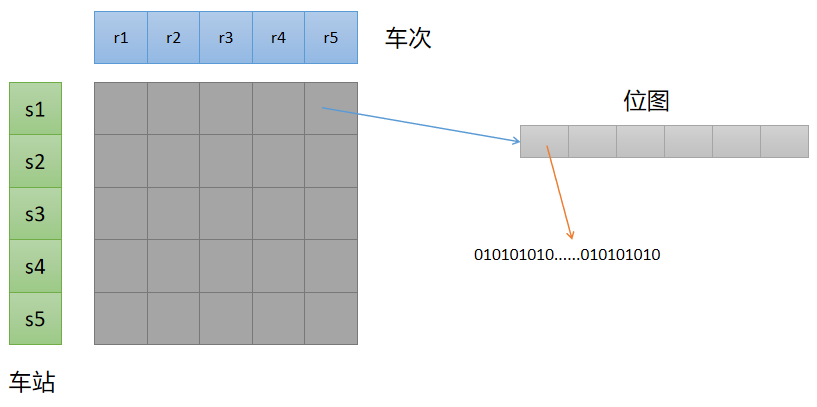
在进行列车座位表示的设计时，我使用了数组加位图的数据结构，使用位图的话可以加速查询以及定位空座位。

首先介绍整体结构的实现，座位表示的整体结构是一个二维数组，二维数组的大小是（station\_num -1）\*（route\_num），因为到达站的车票不需要占用，如下图所示，每次购票、退票、查询时，都需要首先确定车辆的车次，然后从起始站遍历到到达站。



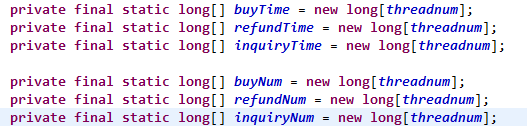
对于具体到确定车站确定车次的座位表示情况，也就是上述二维数组的基础元素的数据结构实现，我使用了位图的方式：

位图的长度是（coach\_num \* seat\_num），每个座位占用一位，位为1的时候表示票已经这个座位的票已经售出，0的时候表示这个座位的票还未售出，由于默认情况下（coach\_num \* seat\_num）是1000，但是java是没有相应的二进制位数为1000的基本类型变量的，所以在实现位图的时候我同样借助了数组，位图使用了一个一维数组，数组的元素是一个Long类型的变量，这样的话一个Long类型具有64位，就可以根据每辆火车的总的座位数来确定位图的数组长度。这样的话整体的数据结构就变成如下图所示的三维数组结构：

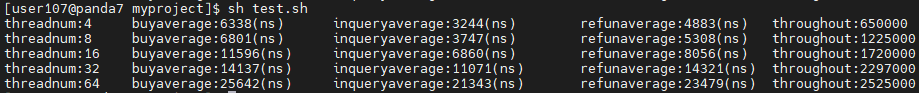


**二、多线程测试程序的设计思路**

根据项目设计书的要求，我基于项目提供的Trace.java代码实现了一个简单的测试每个方法执行的平均时间以及每个线程下系统的总体吞吐量的测试代码Test.java。



相对于Trace.java中的代码，我通过添加了相应的数组，来分别记录每个方法调用所使用的时间以及每个方法调用的次数（如上图），这样可以获得每次方法调用的平均时间。此外，通过计算运行开始和结束之间所花费的时间，可以计算系统总体的吞吐量。下图是在服务器上测试性能的得到的结果：



**三、方法的实现**

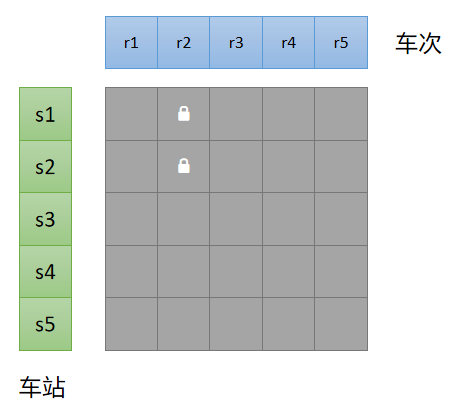
我的方法实现比较粗糙，方法实现的整体思路是，在买票和退票时需要单独对相应的车票进行加锁，所以借助于读写锁的读可以并发、写需要互斥的特点，系统在购票和退票时需要获得写锁，而在查询时只需要获得读锁就可以了。锁的声明如下图：



具体每个方法的实现过程如下：

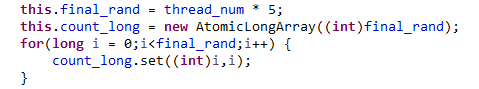
**购票：**

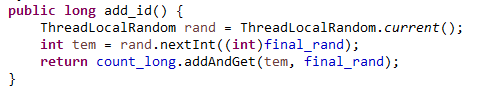
购票时需要申请写锁，写锁并不是锁定所有车站所有车次的座位，而是只根据购票的请求，锁定相应车次的从出发站至到达站（不包括到达站）的所有座位，这样的话锁定的范围就相应的变得很小。如下图，购买r2车次1到3站的车票只需要加上如下的锁：



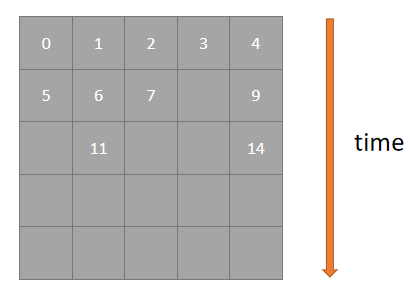
请求到锁之后，会对相应的座位进行查询，而由于座位的基础实现是位图，利用位图可以快速的定位到空位的位置，接下来便可以进行相应的操作：更改位图、获得id、将id和相应的车票信息放入一个hash表中，购票成功，返回车票。

为了使每一个车票的id都不会重复，并且使得相应的瓶颈不会出现在id生成上，我使用了如下的id生成代码：





通过一个长度为5倍线程数量的原子类型的Long数组，初始化时每个位置初始为元素的索引值，然后每次调用这个方法生成id就会随机的获取一个位置，由于数组长度是线程的5倍，所以会尽量小的减少冲突，获得相应的位置后，会利用原子类型的addAndGet来加上数组的长度值，这样就能确保每个id不会重复。即id会出现如下的更新情况（横坐标是数组长度）：



除此之外，为了提高买票的速度，在购票线程获得锁时，我加入了随机购票的操作，即首先随机得到某个票，然后在位图上验证这个票座位是否可以购买，如果可以的话就直接进行购买，由于此时是有锁的，所以其它线程并不能对座位进行操作，这样的话就能减少查询空位置的时间，若随机几次（我默认为了15次）之后仍未购到票，接下来在进行查位置、购票的流程。

加写锁->随机购票->(随机购票失败)查询座位->再次购票->售票记录放进hash表->购票成功->释放写锁

**退票：**

由于退票和购票时均是遍历相应的车站对座位进行加锁，为了防止票退到一半即被购买(也就是在更改位图的时候还未更改完，前面的座位就被购买或查询到)的情况发生，在进行退票时我同样使用了写锁，这样退票时候的流程就变成了：

加写锁->验证票的正确性->更改位图->从hash表中删除->释放写锁

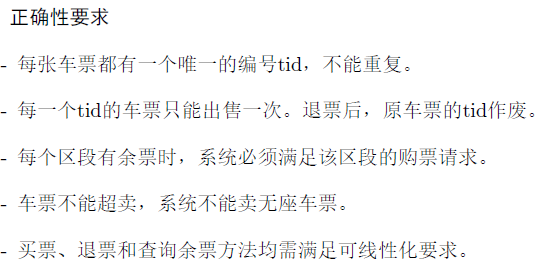
**查询：**

查询的时候需要请求读锁，这样的话就可以进行查询的并发，查询票的数量时只需要遍历位图即可。

加读锁->查询位图->返回结果->释放读锁

**四、系统的正确性和性能**

对于系统的正确性要求如下图：



对于车票的id,由于车票的id是通过Long类型原子的累加实现的，所以一旦车票被卖出，就不会再产生同一个id，这样就保证tid唯一且只会售卖一次。

由于通过读写锁实现了购票、查询、退票的互斥，所以系统能够保证相应区间有票时年就能够买到票。

同样的，根据坐位的数据结构实现，以及锁的互斥，保证了不会出现车票超卖以及售卖无座车票的情况发生。

可线性化分析，由于对同一车次同一区间的购买、查询、退票均需要取得锁之后才能操作，所以这样保证了三个方法均具满足可线性化，可线性化点在相应的锁释放的时候，也就是离开临界区的时候，而不同车次的方法之间不会互相影响。

满足deadlock-free：若很多线程尝试获取一个锁，必定会有某个线程最终得到锁，由于采用读写锁，且每个方法都需要单独申请锁，且最后都会强制释放锁，不会出现死锁的情况。

不满足starvation-free：由于在使用读写锁时使用的是非公平模式，所以会出现饥饿的情况。

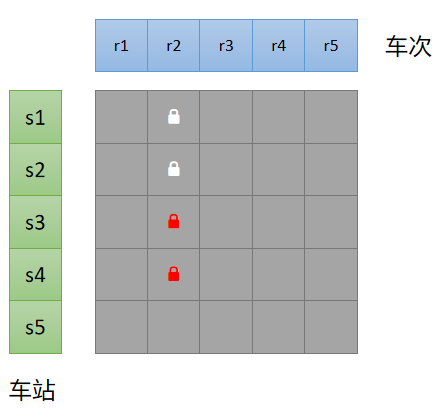
满足lock-free ：某个调用购票或者查询或者退票操作的线程最终会返回。

不满足wait-free：若有的线程进入饥饿状态，则这个方法调用就不能在有限步内返回，即不满足wait-free。

**性能分析：**

本数据结构的时候本质上来讲是一个基于位图数组的细粒度锁，一方面，通过将锁具体到相应车次的具体区间，另一方面使用了读写锁来尽可能降低互斥带来的性能损耗，因为方法调用70%是查询操作，而通过读锁可以实现查询操作之间的并发。

系统的瓶颈：由于在加锁时是对相应车次的对应区间进行加锁，所以系统的性能会受到影响；另一方面，再加锁的时候会出现同一车次的请求区间出现重叠，进而导致申请锁时只申请到了一部分，从而导致加锁失败，如下图，当白色线程申请s1到s4的锁时，由于是遍历申请，当其申请到s3时s3已经被红色线程加锁，这时白色线程就只能进行等待红色线程释放锁，这也很大的影响了系统的性能。



注：test\_result.txt是利用胡起同学提供的测试包进行测试的结果。