性能评测报告

阐述并发数据结构和多线程测试程序的设计思路，分析系统的正确性和性能，解释所实现的每个方法如何满足可线性化、是否deadlock-free、starvation-free、lock-free 或wait-free

并发数据结构实现

本程序中一共实现了四种不同的并发数据结构。其中实现一与实现二的存储结构相同，而数据操作方法不同，实现三与实现四同上。

TicketDS内部结构

在TicketDS内部分别定义多个内部类真正实现业务逻辑，不指定数据结构等。

```
private TicketingSystem actualImpl;
```

使用actualImpl存储真正调用的类。

在实现三个接口的方法时，调用actualImpl对应的方法传递下去。

但在此处进行一些所有内部子类都含有的逻辑，即票的有效无效检测。将购买票的id存入线程安全的跳表集合，当退票的参数id不在已售出的票id内时，判定该票为无效票。

```
@Override
public Ticket buyTicket(String passenger, int route, int departure, int arrival) {
    Ticket boughtTicket = actualImpl.buyTicket(passenger, route, departure, arrival);
    if(boughtTicket != null) {
        soldTickIds.add(boughtTicket.tid);
    }
    return boughtTicket;
}

@Override
public int inquiry(int route, int departure, int arrival) {
    return actualImpl.inquiry(route, departure, arrival);
}

@Override
public boolean refundTicket(Ticket ticket) {
    if (!soldTickIds.contains(ticket.tid)) {
        // 无效票
        return false;
    }
    boolean refundSuccess = actualImpl.refundTicket(ticket);
```

```
    if (!refundSuccess) {
        soldTickIds.remove(ticket.tid);
    }
    return refundSuccess;
}
```

下面介绍内部的四个真正实现类。

实现一

数据结构

数据结构如下所述：

> 按照车次建立数组，如Route[5]:
> 每个车次下的每个站点建立数组，如station[10]:
> 每个站点包含一个位向量，涵盖所有车厢的所有座位，1表示座位被所占用，如[1, 1, 1, 0, 0, 0, 1]表示有7个座位，其中前三个与最后一个座位已经被占用。

通过以上描述，全部数据使用一个二维数组保存。其中一维下标为车次，二维下标为站点。

车厢概念被消除，仅保留座位概念。通过取模方式计算车厢号码。

```
protected final BitSet[][] data;
```

Inquery操作实现:

```
public int inquiry(int route, int departure, int arrival) {
  if (isParamsInvalid(route, departure, arrival)) {
    return -1;
  }
  BitSet[] station2seats = data[route - 1];
  BitSet seatsMerged = (BitSet) station2seats[departure - 1].clone();
  for (int i = departure; i <= arrival - 1; i++) {
    try {
      seatsMerged.or(station2seats[i]);
      if (seatsMerged.size() != station2seats[i].size()) {
        throw new Exception();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  return SEAT_NUM * COACH_NUM - seatsMerged.cardinality();
}
```

正确性和性能

首先进行参数检查。随后找到对应车次的每个站点的位图数组。每次从内存读取该车次的每个站点的位图，从departure到arrival的每个站点的位向量进行或，获取所有进行或操作后仍为0的位置，`seatsMerged.cardinality()`将会返回位图中设置为1的数量，用每个车次的座位总数减去即得到了空位的数量。

由于使用的是位图，以上操作速度会非常快，性能应该会是最好的。

满足要求

由于查询只要求静态一致性，所以我们不需要进行加锁。BitSet对象中的数据将从内存读取，线程间可见性能够保障。

由于没有进行加互斥锁，所以deadlock-free，starvation-free、lock-free或wait-free都是能够得到实现的。

BuyTicket操作实现：

```
1  public Ticket buyTicket(String passenger, int route, int departure, int
   arrival) {
2    if (isParamsInvalid(route, departure, arrival)) {
3      return null;
4    }
5    BitSet[] station2seats;
6    int seatIdx;
7    synchronized (station2seats = data[route - 1]) {
8      seatIdx = doBuyTicket(station2seats, departure, arrival);
9    }
10   if (seatIdx == -1) {
11     return null;
12   }
13   CoachSeatPair p = new CoachSeatPair(seatIdx);
14   return buildTicket(currTid.getAndIncrement(), passenger, route, p.coach,
   p.seat, departure, arrival);
15 }
```

正确性和性能

购票前首先也需要检查参数。购票前，我们使用synchronized关键字对对应的车次进行加锁。

同时由于jdk目前对于synchronized关键字的锁性能优化十分良好，其中对于竞争不激烈到激烈的过程中，它的锁分为：无锁，轻量级锁，偏向锁，重量级锁。jvm内部在竞争激烈时会逐步地升级锁。在竞争不激烈时锁的竞争很少，速度快。

```
1  protected int doBuyTicket(BitSet[] station2seats, int departure, int
   arrival) {
2    BitSet seatsMerged = (BitSet) station2seats[departure - 1].clone();
3    for (int i = departure; i <= arrival - 1; i++) {
4      seatsMerged.or(station2seats[i]);
5    }
6    int seatIdx = seatsMerged.previousClearBit(SEAT_NUM * COACH_NUM - 1);
```

```
    if (seatIdx == -1) {
      // 无票
      return seatIdx;
    }
    for (int i = departure - 1; i <= arrival - 1; i++) {
      station2seats[i].set(seatIdx);
    }
    return seatIdx;
  }
```

内部处理使用位图，速度同样是非常快的。

满足要求

使用了synchronized关键字，能够保障多线程对于同一个车次的数据访问是可线性化的。

由于使用synchronized同步块，其中的锁的申请和释放由jvm执行的，同时在代码块内部没有更改该锁对象的锁状态关键字等数据，所以不可能出现死锁与饥饿的情况，所以deadlock-free，starvation-free是能够得到实现的。

而lock-free和wait-free，由于加了互斥锁，所以无法实现。（若持有锁的线程被永久挂起，系统将无法进一步取得进展）

RefundTicket操作实现：

```
public boolean refundTicket(Ticket ticket) {
  if (isParamsInvalid(ticket.route, ticket.departure, ticket.arrival)) {
    return false;
  }
  BitSet[] station2seats;
  synchronized (station2seats = data[ticket.route - 1]) {
    return doRefundTicket(station2seats, ticket);
  }
}
```

正确性和性能

退票前首先也需要检查参数。购票前，我们使用synchronized关键字对对应的车次进行加锁。

性能分析同BuyTicket中一致。

```
protected boolean doRefundTicket(BitSet[] station2seats, Ticket ticket) {
  // 检查座位所有departure到arrival的站是否全部被占用
  CoachSeatPair p = new CoachSeatPair(ticket.coach, ticket.seat);
  for (int i = ticket.departure - 1; i <= ticket.arrival - 1; i++) {
    if (!station2seats[i].get(p.seatIdx)) {
      return false;
    }
  }
  for (int i = ticket.departure - 1; i <= ticket.arrival - 1; i++) {
    station2seats[i].set(p.seatIdx, false);
  }
  return true;
}
```

具体的doRefundTicket由于使用位图，对于位图的操作速度都是很快的。

满足要求

与BuyTicket基本一致。

使用了synchronized关键字，能够保障多线程对于同一个车次的数据访问是可线性化的。

而lock-free和wait-free，由于加了互斥锁，所以无法实现。

实现二

数据结构

实现二继承了实现一的数据结构，与实现一保持一致。

Inquery操作实现：

与实现一完全一致，不再赘述。

BuyTicket操作实现：

```
public Ticket buyTicket(String passenger, int route, int departure, int
arrival) {
  if (isParamsInvalid(route, departure, arrival)) {
    return null;
  }
  int seatIdx;
  ReentrantLock lock = locks[route - 1];
  try {
    lock.lock();
    BitSet[] station2seats = data[route - 1];
    seatIdx = doBuyTicket(station2seats, departure, arrival);
  } finally {
    lock.unlock();
```

```
  }
  if (seatIdx == -1) {
    return null;
  }
  CoachSeatPair p = new CoachSeatPair(seatIdx);
  return buildTicket(currTid.getAndIncrement(), passenger, route, p.coach,
p.seat, departure, arrival);
}
```

正确性和性能

与实现一中的唯一区别即为，将synchronized关键字替换为了ReentrantLock进行加锁的操作。

性能相比synchronized可能较差。但内部由于使用位图操作，同时由于车次能够造成分流，速度仍会较快。

满足要求

使用了ReentrantLock进行加锁，能够保障多线程对于同一个车次的数据访问是可线性化的。

使用了try-finally结构进行上锁-释放锁，同时由于代码块内部没有对其他资源的需求，也没有对lock对象本身做出改变，所以能够保障deadlock-free，starvation-free的实现。

而lock-free和wait-free，由于加了互斥锁，所以无法实现。

RefundTicket操作实现：

```
public boolean refundTicket(Ticket ticket) {
    if (isParamsInvalid(ticket.route, ticket.departure, ticket.arrival)) {
        return false;
    }
    BitSet[] station2seats = data[ticket.route - 1];
    ReentrantLock lock = locks[ticket.route - 1];
    try {
        lock.lock();
        return doRefundTicket(station2seats, ticket);
    } finally {
        lock.unlock();
    }
}
```

正确性和性能

与上面BuyTicket中的分析基本一致。

满足要求

与上面BuyTicket中的分析基本一致。能够保障deadlock-free，starvation-free的实现。

实现三

数据结构

> 数据结构：只建立一个超长数组。数组长度为 ROUTE_NUM * COACH_NUM * SEAT_NUM
> 将全部车次，全部车厢，全部座位放置到一个长数组中，避免嵌套数组的多次访存开销。
> 每个元素为int或long类型，该数字的每个比特表示在某个车站是否有人已经占座。
> 当车站数量<32时，使用int数组，当车站数量>32且<=64时，使用long数组。

数据使用两种数组存储，分别是站点数较少时的intArray（只有32个比特，所以只支持32个及以内的站点数），以及longArray（支持64个站点）。

使用VarHandle提供的变量访问类，用来针对数组内的每个元素进行不同类型的访存操作。

其中包括Plain，Opaque，Acquire，Release以及Volatile五类访问变量的类型。

其中主要使用getOpaque保障变量在多线程间的线程可见性。

以及使用VarHandle支持的CompareAndSet原语，可以对数组内的任意元素进行CAS修改，避免加锁。

```
protected VarHandle vh;
protected long[] longArray;
protected int[] intArray;
protected boolean useLong;
```

Inquery操作实现:

```
public int inquiry(int route, int departure, int arrival) {
    if (isParamsInvalid(route, departure, arrival)) {
        return -1;
    }
    int res = 0;
    for (int i = (route - 1) * COACH_NUM * SEAT_NUM; i < route * COACH_NUM * SEAT_NUM; i++) {
        if (checkSeatInfo(readSeatInfoOpaque(i), departure, arrival, false)) {
            // 当前座位idx为i，如果检查从departure 到 arrival之间的位数都为false，即可证明该座位的该区间没有被占用，可以增加余票数量。
            res++;
        }
    }
    return res;
}
```

正确性和性能

首先进行参数检查。随后通过遍历整个车次对应的数组范围，检查每个座位信息，查看其是否满足从departure到arrival的位数都为false，如果都为false即可证明该座位在该区间内是可以乘坐的，即可将余票增加1。

在读取座位信息的过程中使用了readSeatInfoOpaque函数，如下所示：

```
protected long readSeatInfoOpaque(int seatIdx) {
    if (useLong) {
        return (long) vh.getOpaque(longArray, seatIdx);
    } else {
        return (int) vh.getOpaque(intArray, seatIdx);
    }
}
```

该函数中使用了VarHandle类的getOpaque方法，该方法能够确保读取内存中的数据，避免了读取缓存，从而保证了静态一致性的实现。

由于没有进行加锁等操作，性能还算可以，但由于需要遍历每个座位的信息，循环次数较大，会比较耗时。

满足要求

查询只要求静态一致性，由于使用VarHandle类的getOpaque方法，保证查询操作在没有其他buy和refund操作的情况下，能够查询到内存中的精确数据。所以能够保证静态一致性的实现。

由于没有进行加互斥锁，所以deadlock-free，starvation-free、lock-free或wait-free都是能够得到实现的。

BuyTicket操作实现：

```
public Ticket buyTicket(String passenger, int route, int departure, int arrival) {
    if (isParamsInvalid(route, departure, arrival)) {
        return null;
    }
    int left = (route - 1) * COACH_NUM * SEAT_NUM;
    int right = route * COACH_NUM * SEAT_NUM - 1;
    int startIdx = super.random.nextInt(left, right + 1);
    int currIdx = startIdx;
    boolean buyResult = false;
    while (currIdx >= left) {
        buyResult = setOccupiedInverted(route, currIdx, departure, arrival, true);
        if (buyResult) {
            break;
        }
        currIdx--;
    }
    if (!buyResult) {
        currIdx = startIdx + 1;
        while (currIdx <= right) {
            if (checkSeatInfo(readSeatInfoPlain(currIdx), departure, arrival, true)) {
                currIdx++;
                continue;
            }
```

```
            buyResult = setOccupiedInverted(route, currIdx, departure, arrival, true);
            if (buyResult) {
                break;
            }
            currIdx++;
        }
    }
    if (!buyResult) {
        // 无余票
        return null;
    }
    Seat s = new Seat(currIdx, 0);
    return buildTicket(currTid.getAndIncrement(), passenger, route, s.getCoach(), s.getSeat(), departure, arrival);
}
```

正确性和性能

首先进行参数检查。随后在整条车次对应的数组范围内，生成一个随机起始搜索点，该随机数生成使用了ThreadLocalRandom类，能够提升生成随机数的性能。

从该随机起始搜索点先向左侧搜索，每遇到一个新的点位，便尝试调用setOccupiedInverted进行购买。若向左侧一直没有搜索到可以购买的座位，即再从一开始生成的位置向右侧开始搜索。

搜索结束后若仍没有成功购买，则认为无余票。

在尝试购买时使用了setOccupiedInverted函数，如下所示：

```
protected boolean setOccupiedInverted(int route, int seatIdx, int departure, int arrival, boolean occupied) {
    long expectedSeatInfo, newSeatInfo;
    do {
        expectedSeatInfo = readSeatInfoOpaque(seatIdx);
        if (!checkSeatInfo(expectedSeatInfo, departure, arrival, !occupied)) {
            // 从departure到arrival站区间内，有与occupied相反的位数，即
            // 若occupied为true，即我们想将departure到arrival的站点全部设置为true，首先需要确保这些站点当前为false。
            // 若有invert，则认为当前座位已被占，无法购买。
            return false;
        }
        newSeatInfo = expectedSeatInfo;
        for (int i = departure - 1; i <= arrival - 1; i++) {
            if (occupied) {
                // 置1
                newSeatInfo = newSeatInfo | (0x1L << i);
            } else {
                // 置0
                newSeatInfo = newSeatInfo & ~(0x1L << i);
```

```
            }
        }
    } while (useLong ?
            !vh.compareAndSet(longArray, seatIdx, expectedSeatInfo,
newSeatInfo) :
            !vh.compareAndSet(intArray, seatIdx, (int) expectedSeatInfo,
(int) newSeatInfo));
    return true;
}
```

该函数的含义是：将某个座位对应的站点信息，从departure到arrival的站点进行翻转。举例说明：

该座位的站点信息为整数i，该整数i的前10位表示在该站点是否被占用，如：[0,0,0,0,1,1,1,0,0,0]表示该座位从第四站到第六站被占用了。

那么此时想要购买该座位的第一到第二站，则需要调用该函数将该数字的前两位进行翻转。该翻转过程使用VarHandle提供的CompareAndSet原语。当修改前的读取的内容与写入时的数据不一致时，将不会进行修改。这样可以保证多线程的安全性。

使用该方法的查找座位的速度比较慢，需要多次搜索。而CompareAndSet使用了原子原语保证正确性，没有加锁，性能较好。

满足要求

通过使用CAS操作进行数据写入，使得多个线程若同时进入循环内部，调用该段代码，只有一个能够调用成功，其余线程即会重新循环进入下一次调用，保证了该操作的可线性化性。

而同时CAS操作避免了加锁，不可能出现死锁以及饥饿的问题。同时多个线程同时执行CAS时，在一定时间内，至少会有一个线程执行成功，系统将会取得进展，能够保障lock-free与wait-free的实现。

RefundTicket操作实现：

```
@Override
public boolean refundTicket(Ticket ticket) {
    if (isParamsInvalid(ticket.route, ticket.departure, ticket.arrival)) {
        return false;
    }
    Seat s = new Seat(ticket.route, ticket.coach, ticket.seat);
    return setOccupiedInverted(ticket.route, s.getSeatIdx(),
ticket.departure, ticket.arrival, false);
}
```

不需要进行随机搜索，直接调用setOccupiedInverted进行退票即可。

正确性和性能

与BuyTicket中实现相比，避免了随机搜索查找的开销，其余一致，不再赘述。

满足要求

与BuyTicket中实现一致，不再赘述。

实现四

实现四继承了实现三，仅仅修改了setOccupiedInverted的实现。

```
protected boolean setOccupiedInverted(int route, int seatIdx, int
departure, int arrival, boolean occupied) {
    long expectedSeatInfo, newSeatInfo;
    synchronized (seatIndices[seatIdx]) {
        expectedSeatInfo = readSeatInfoOpaque(seatIdx);
        if (!checkSeatInfo(expectedSeatInfo, departure, arrival,
!occupied)) {
            return false;
        }
        newSeatInfo = expectedSeatInfo;
        for (int i = departure - 1; i <= arrival - 1; i++) {
            if (occupied) {
                // 置1
                newSeatInfo = newSeatInfo | (0x1L << i);
            } else {
                // 置0
                newSeatInfo = newSeatInfo & ~(0x1L << i);
            }
        }

        if (useLong) {
            vh.setOpaque(longArray, seatIdx, newSeatInfo);
        } else {
            vh.setOpaque(intArray, seatIdx, (int) newSeatInfo);
        }
    }
    return true;
}
```

该实现没有采用CAS操作，它采用了对于每个座位的信息修改过程中，对于每个座位生成一个锁，使用synchronized保证对于该座位的信息修改的原子性。

BuyTicket与RefundTicket操作实现：

只修改了setOccupiedInverted的内部，基本与实现三一致。

正确性和性能

对于每个座位进行了加锁，多个线程同时对同一个座位进行写入时，只有一个线程能够进入同步块，其余线程阻塞等待。使用synchronized关键字使得若同一个座位的内容被访问到时，才会产生较大的性能损失。

满足要求

使用了synchronized关键字，能够保障多线程对于同一个座位的数据访问是可线性化的。

而lock-free和wait-free，由于加了互斥锁，所以无法实现。

多线程测试程序的设计思路

TicketingDS内部对于测试的支持

TicketingDS内部有定义一个enum，如下所示：

```
public enum ImplType {
    One(ImplOne.class),
    Two(ImplTwo.class),
    Three(ImplThree.class),
    Four(ImplFour.class);
    private final Class<? extends TicketingSystem> implClass;

    ImplType(Class<? extends TicketingSystem> implClass) {
        this.implClass = implClass;
    }
}
```

该枚举内部包含了四个内部的实现TicketingSystem接口的类。即上面所说的实现一到四。

通过使用如下所示的switchImplType方法，使用java的反射，动态更改TicketingDS的实现类，提供方便地动态测试能力。

```
public void switchImplType(ImplType type) throws NoSuchMethodException,
IllegalAccessException, InvocationTargetException, InstantiationException {
    this.actualImpl =
type.implClass.getConstructor(TicketingDS.class).newInstance(this);
    this.currTid = new AtomicLong(1);
}
```

Test类实现

首先定义常量以及部分类变量

```
private static final int INQUIRY_RATIO = 60;
private static final int PURCHASE_RATIO = 30;
private static final int REFUND_RATIO = 10;
private static final int ROUTE_NUM = 5;
private static final int COACH_NUM = 8;
private static final int SEAT_NUM = 100;
private static final int STATION_NUM = 10;
private static final int FUNC_CALL_COUNT = 10000;
private static int THREAD_NUM = 16;
private static final int REPEAT_MULTI_THREAD_TEST_COUNT = 50;
private static final boolean PRINT_BUY_INFO = false;
private static TicketingDS tds;
private static Map<String, TicketConsumerStatistics> thread2Statistics;
```

其中thread2Statistics存储当每个线程运行结束后，存储的统计数据。

其代表统计数据的类为TicketConsumerStatistics，如下所示：

```
private static class TicketConsumerStatistics {
    private int tryBuyCount = 0;
    private int tryInqCount = 0;
    private int tryRefCount = 0;
    private long buyExecTime = 0;
    private long inqExecTime = 0;
    private long refExecTime = 0;
    private int funcCallCount = 0;
    private long fullExecTime = 0;
    ...
}
```

分别对应存储了每个操作的执行次数，以及执行时间。

开始测试的入口

```
private static void doVariousThreadNumMultiThreadTest() throws ... {
    int[] THREAD_NUMS = new int[]{4, 8, 16, 32, 64};
    for (int thread_num : THREAD_NUMS) {
        System.out.println("-------------- START TEST THREAD NUM = " +
thread_num + " --------------");
        THREAD_NUM = thread_num;
        tds = new TicketingDS(ROUTE_NUM, COACH_NUM, SEAT_NUM, STATION_NUM,
THREAD_NUM);
        for (TicketingDS.ImplType value : TicketingDS.ImplType.values()) {
            repeatDoMultiThreadTest(value,
REPEAT_MULTI_THREAD_TEST_COUNT);
        }
        System.out.println("--------------- END TEST THREAD NUM = " +
thread_num + " ---------------\n\n");
    }
}
```

对于4，8，16，32，64个线程，分别repeatDoMultiThreadTest进行测试，该函数接受一个repeatCount参数，表示一共重复执行repeatCount次测试后，取其平均执行的测试结果。

以下定义了repeatDoMultiThreadTest：

```
private static void repeatDoMultiThreadTest(TicketingDS.ImplType implType,
int repeatCount) throws InterruptedException, InvocationTargetException,
NoSuchMethodException, InstantiationException, IllegalAccessException {
    thread2Statistics = new ConcurrentHashMap<>();
    for (int i = 0; i < THREAD_NUM; i++) {
        thread2Statistics.put(String.valueOf(i + 1), new
TicketConsumerStatistics());
    }
    for (int rc = 0; rc < repeatCount; rc++) {
        System.gc();
        Thread.sleep(100);
        tds.switchImplType(implType);

        Thread[] threads = new Thread[THREAD_NUM];
        for (int i = 0; i < THREAD_NUM; i++) {
            Thread t = new Thread(new TicketConsumerRunner(),
String.valueOf(i + 1));
            threads[i] = t;
            t.start();
        }
        for (Thread thread : threads) {
            thread.join();
        }
    }
    long fullExecNanoTime =
thread2Statistics.values().stream().mapToLong(TicketConsumerStatistics::ge
tFullExecTime).sum();
```

```
22      // ... do calculate
23      int fullFuncCallCount =
thread2Statistics.values().stream().mapToInt(TicketConsumerStatistics::get
FuncCallCount).sum();
24      // ... do calculate
25      double throughPut_nano = (1. * fullFuncCallCount) / fullExecNanoTime;
26      // ... do output
27  }
```

首先初始化统计数据，在每次repeat起始时，调用System.gc()，建议jvm进行gc保证内存的可用空间。

启动THREAD_NUM个线程，其执行Runner为TicketConsumerRunner，在其内部再进行具体的tds操作调用。

在最后，将统计数据内部的数据进行stream处理，统计方法执行总数，执行时间等等数据，最后再进行打印屏幕的操作。

TicketConsumerRunner内部较为简单，逻辑即为按照ThreadLocalRandom提供的随机数，选择执行inquiry，buyTicket以及refundTicket操作，在执行次数结束后，将执行过程中统计的执行次数与执行时间添加到线程安全的thread2Statistics中即可。

执行效率测试结果

执行环境

jdk版本：15

macOS Mojave (13-inch, 2018)

处理器：2.3 GHz Intel Core i5

内存：8 GB 2133MHz LPDDR3

执行配置

```java
private static final int INQUIRY_RATIO = 60;                    // Inquiry比例
private static final int PURCHASE_RATIO = 30;                   // BuyTicket比例
private static final int REFUND_RATIO = 10;                     // RefundTicket比例
private static final int ROUTE_NUM = 5;                         // 车次
private static final int COACH_NUM = 8;                         // 车厢
private static final int SEAT_NUM = 100;                        // 每个车厢的座位
private static final int STATION_NUM = 10;                      // 车站数量
private static final int FUNC_CALL_COUNT = 10000;               // 函数调用次数
private static final int REPEAT_MULTI_THREAD_TEST_COUNT = 100;  // 重复测试次数
```

线程数为4时

```
实现一:
=================================================================
ImplOne class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 3999800, fullExecNanoTime = 3884778981
throughPut(func/nano) = 0.0010296081243135206
inquiry average exec time = 433.81664893488505
buyTicket average exec time = 1739.3105879938842
refundTicket average exec time = 1893.7532088666874
=================================================================
实现二:
=================================================================
ImplTwo class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 3999800, fullExecNanoTime = 4219040635
throughPut(func/nano) = 9.480354293861879E-4
inquiry average exec time = 401.38024907206324
buyTicket average exec time = 1951.540142212024
refundTicket average exec time = 2285.0436961319897
=================================================================
实现三:
=================================================================
ImplThree class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 3999820, fullExecNanoTime = 30637425821
throughPut(func/nano) = 1.3055339646904598E-4
inquiry average exec time = 11720.85150478742
buyTicket average exec time = 1814.057772844913
refundTicket average exec time = 805.317415117751
=================================================================
```

```
实现四:
================================================================
ImplFour class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 3999818, fullExecNanoTime = 31076042834
throughPut(func/nano) = 1.2871066053570494E-4
inquiry average exec time = 11649.687005480131
buyTicket average exec time = 2328.090833817173
refundTicket average exec time = 842.4305665783182
================================================================
```

可以看出在线程数较少时，实现一和实现二基本类似，其中查询由于使用了位图且没有加锁，速度非常快。

而在实现三和四中，查询和买票的速度由于遍历次数太多，速度很慢。而由于退票时O(1)操作，速度较快。

线程数为8时

```
实现一:
================================================================
ImplOne class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 7999575, fullExecNanoTime = 9820759372
throughPut(func/nano) = 8.145576830654883E-4
inquiry average exec time = 623.388149636633
buyTicket average exec time = 2011.0215481618384
refundTicket average exec time = 2501.054555188766
================================================================
实现二:
================================================================
ImplTwo class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 7999615, fullExecNanoTime = 13503673989
throughPut(func/nano) = 5.924028532173119E-4
inquiry average exec time = 532.641400252485
buyTicket average exec time = 3279.026995234713
refundTicket average exec time = 3852.280971630549
================================================================
实现三:
================================================================
ImplThree class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 7999582, fullExecNanoTime = 102429645145
throughPut(func/nano) = 7.809830824538878E-5
inquiry average exec time = 17839.18831150443
buyTicket average exec time = 6646.10917340269
refundTicket average exec time = 1019.423725587191
================================================================
```

```
实现四:
================================================================
ImplFour class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 7999600, fullExecNanoTime = 107281032118
throughPut(func/nano) = 7.456676955905048E-5
inquiry average exec time = 17769.988552876268
buyTicket average exec time = 8804.491509036307
refundTicket average exec time = 1063.3038823242707
================================================================
```

与线程数为4的情况基本一致。

线程数为16时

```
实现一:
================================================================
ImplOne class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 15995344, fullExecNanoTime = 23266901690
throughPut(func/nano) = 6.874720241275065E-4
inquiry average exec time = 544.9923443566737
buyTicket average exec time = 2677.4759257249384
refundTicket average exec time = 3250.7533553457483
================================================================
实现二:
================================================================
ImplTwo class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 15999138, fullExecNanoTime = 44586895152
throughPut(func/nano) = 3.5883050267254007E-4
inquiry average exec time = 508.4379137496409
buyTicket average exec time = 5977.371707116696
refundTicket average exec time = 6902.917358281934
================================================================
实现三:
================================================================
ImplThree class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 15998875, fullExecNanoTime = 347008702980
throughPut(func/nano) = 4.6105111666095885E-5
inquiry average exec time = 27622.852706428646
buyTicket average exec time = 16432.061597342337
refundTicket average exec time = 1858.4909222980189
================================================================
实现四:
================================================================
ImplFour class repeat multi thread test finished.
repeat test count: 100
```

```
fullFuncCallCount = 15999001, fullExecNanoTime = 426699894021
throughPut(func/nano) = 3.7494738630548174E-5
inquiry average exec time = 27711.380776626298
buyTicket average exec time = 32869.56391313051
refundTicket average exec time = 1826.1805284481966
======================================================================
```

线程数上升后，可以看出实现一和二的查询由于没有加锁，速度还是非常快。但是可以初步看出buy和refund操作的速度差距逐渐变大，能够看出synchronized关键字的性能要比使用reentrantLock强。

实现三与四的吞吐量差距不大，但是实现四由于在buyTicket当中使用了synchronized关键字，可以看出速度相较实现三变慢了很多。

线程数为32时

```
实现一:
======================================================================
ImplOne class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 31764569, fullExecNanoTime = 27796375572
throughPut(func/nano) = 0.0011427593830613393
inquiry average exec time = 428.8016216661912
buyTicket average exec time = 1444.0819557603795
refundTicket average exec time = 1921.4668905413669
======================================================================
实现二:
======================================================================
ImplTwo class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 31998402, fullExecNanoTime = 161439671417
throughPut(func/nano) = 1.982065605011538E-4
inquiry average exec time = 502.659941603634
buyTicket average exec time = 11557.817529917895
refundTicket average exec time = 12754.70503400484
======================================================================
实现三:
======================================================================
ImplThree class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 31977892, fullExecNanoTime = 1468735406494
throughPut(func/nano) = 2.1772398117870685E-5
inquiry average exec time = 55439.37159505918
buyTicket average exec time = 40530.63646105173
refundTicket average exec time = 4774.249218990254
======================================================================
实现四:
======================================================================
ImplFour class repeat multi thread test finished.
repeat test count: 100
```

```
fullFuncCallCount = 31992407, fullExecNanoTime = 1677412285797
throughPut(func/nano) = 1.907247685669551E-5
inquiry average exec time = 36442.80297253684
buyTicket average exec time = 100517.34567188489
refundTicket average exec time = 3870.2879462739143
======================================================================
```

随着线程数变多，可以看出执行速度变得越来越慢，由于本机的CPU核心数不足以覆盖全部的线程数，而该执行的任务又是计算密集的程序，所以当线程数变多时会造成线程切换的成本非常高。

其中实现三和四的buyTicket速度明显变慢。

线程数为64时

```
实现一:
======================================================================
ImplOne class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 62747311, fullExecNanoTime = 155811395270
throughPut(func/nano) = 4.02713234749406E-4
inquiry average exec time = 452.69949784385705
buyTicket average exec time = 5820.715558113958
refundTicket average exec time = 5187.217593759577
======================================================================
实现二:
======================================================================
ImplTwo class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 63995235, fullExecNanoTime = 617751986800
throughPut(func/nano) = 1.0359373400237845E-4
inquiry average exec time = 473.48292587131607
buyTicket average exec time = 22970.10727816432
refundTicket average exec time = 24808.23110215771
======================================================================
实现三:
======================================================================
ImplThree class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 63798659, fullExecNanoTime = 6333133600467
throughPut(func/nano) = 1.0073790168471346E-5
inquiry average exec time = 109142.76832977039
buyTicket average exec time = 102129.92184692953
refundTicket average exec time = 29211.161166907637
======================================================================
实现四:
======================================================================
ImplFour class repeat multi thread test finished.
repeat test count: 100
fullFuncCallCount = 63928033, fullExecNanoTime = 6888455809275
```

```
throughPut(func/nano) = 9.280459187082791E-6
inquiry average exec time = 41530.65531394724
buyTicket average exec time = 272429.03126243706
refundTicket average exec time = 9882.533491990436
================================================================================
```

在线程为64个时，可以看出依然是实现1的性能最优，其通过使用位图的实现方式，以及使用synchronized优化的锁，保证大量线程同时执行时的速度。而和实现二的对比看出其使用的ReentrantLock性能会较差。

实现三和四的时间复杂度较高，速度较差。实现三和四能够对比出不加锁的CAS操作确实与synchronized关键字加锁后更快。但是由于在本场景中，能够使用CAS操作的数据结构设计被局限在实现三和四的结构中，不能使用实现一和二的位图实现。