

性能评测报告

并发数据结构设计思路

代码的设计由以下两部分组成:

1. 采用BitMap 位图索引，压缩数据存储和查找的时间 利用 Java的 Long.BitCount完成对于比特串中 1 出现次数的计数。提高查找效率
2. 采用CLH锁，对于每趟列车上一把锁，锁的个数增多，可减少争用。

位图索引的设计

由于需要实现的三个方法中，均传递了参数 `route`, 所以可以将每趟列车分别对应一个位图，所有的列车合起来，构成一个位图的列表。

这样实现的好处是，在执行查询、买票、退票的操作时，都可以直接索引对应列车的位图，而需要对全部的列车信息进行操作，减少了需要操作的范围，也便于锁的分配。

```
public class TicketingDS implements TicketingSystem{
    public static ArrayList<long[][]> BitMapTotal;
    public static long[][] BitMap;

    // 在构造函数中，进行初始化
    public TicketingDS(int routenum, int coachnum, int seatnum, int stationnum,
int threadnum){
        BitMapTotal = new ArrayList<long[][]>();

        for(int i = 0; i <= routenum; i++){
            long[][] BitMapToadd = new long[(coachnum * seatnum)/64 + 1]
[stationnum + 1];
            BitMapTotal.add(BitMapToadd);
        }
    }
}
```

在上述构造中，涉及到下列的细节。

细节一: BitMap 采用 Long[][] 类型

这个二维数组，其中每一列代表某一列车的 第 `x` 个车厢的第 `y` 个座位。以缺省值为例子，假设取

```
BitMapTotal.get(1)[0][0];
```

前一个 `BitMapTotal` 的下标从 0 开始，所以默认下标为 0 的第 0 趟列车无用，这里仅进行初始化和添加，在后续索引中都不会用到。

二维数组的行表示某一站的情况，例如出发站从 1 - 10,则除了第 0 站不需要使用后，其余每行均表示站点的对应值。对于每一列来说，直接表示列车的情况。假设我们有 9 个车厢，100 个座位，则每一行均需要至少 $9 * 100 = 900$ 个 bit 位来进行存储。

- 选用 Long 而不是 Byte 或 int 等其他类型: 位图运算的好处在于，把对于单个元素的赋值转换为 位运算。现在 1 bit 表示列车上某 x 车厢 y 座位的情况，则long类型可以表示 64 个座位。在进行修改时，只需要确定待修改参数的位置，进行适当的 & | ~ (与或非)等操作。

如果选用 Long, 每一行都需要记录 900 个座位的情况。经过计算得到 $900/64 = 14.0625$ 所以至少需要 15 个 Long 类型的元素，一共存储了 $15 * 64 = 960$ 个 bit位。

但如果采用 Byte,每次只能存储 8 个座位的信息，每一行需要 $900 / 8 = 112.5$, 至少遍历 113 个 Byte 类型的元素。在位运算中，我们希望分隔的数据块约少，一次遍历的速度也越快。

```
// 表示1号列车初始化时，具有 9 个车厢，每个车厢 100 个座位
// 一共有 10 个站点，一共 15 个 Long 类型数据，每块初始化为 64bit 的0， 显示为 15 个 0
0 0000000000000000
1 0000000000000000
2 0000000000000000
3 0000000000000000
4 0000000000000000
5 0000000000000000
6 0000000000000000
7 0000000000000000
8 0000000000000000
9 0000000000000000
10 0000000000000000
```

细节二: x 车厢的 y 座位在每一行中 第 m 个 (从0开始)long块的 第 n 个偏移 (从 0 开始)

实际上，需要存储 9 个车厢，每车厢100个座位的情况。由上述求解过程得知，需要有 15 个 Long 块。实际上，每一行的 bit 从 0 - 959, 我们假设第 [0] 个 Long 块，存储 第 1 个车厢的 1-64 号元素，形成如下的对应关系:

```
/*
 * 9 coach 100 seat
 * [0] 1: 1 - 64 [1] 1: 65 - 100(36) 2: 1 - 28
 * [2] 2: 29 - 92 [3] 2: 93 - 100(8) 3: 1 - 56
 * [4] 3: 57 - 100(44) 4: 1 - 20 [5] 4: 21 - 84
 * [6] 4: 85 - 100(16) 5: 1 - 48 [7] 5: 49 - 100(52) 6: 1 - 12
 * [8] 6: 13 - 76 [9] 6: 77 - 100(24) 7: 1 - 40
 * [10] 7: 41- 100(60) 8: 1 - 4 [11] 8: 5 - 68
 * [12] 8: 69 - 100(32) 9: 1 - 32 [13] 9: 33 - 96
 * [14] 9: 97- 100(4) add 60' 1
 * **/
```

由上，我们知道每一行的第 0 个 Long块，表示 1 号车厢的 1-64 在这一站的(空/非空)情况，第 1 个 Long 块，表示了 1 号车厢的 65-100 座位在这一站的 (空/非空)情况，2 号车厢的 1-28 座位在这一站的 (空/非空)情况

等。每一行都对应应有 15 个 Long 块。现在需要寻找这样的对应关系,有如下两个需求:

1. 知道 x 车厢的 y 座位, 希望知道 这个座位在 每一行的 第 几个 Long 块, 偏移为几?
2. 知道第 m 个Long块(从 0 开始) 的 第 n 个偏移(从 0 开始), 需要确定是 第 x 个车厢的 第 y 个座位?

这两个关系实际上并不复杂, 因为 Long 是以 64 存储数据, 而 座位是以 seatnum 存储数据, 只需要确定该元素是第几个 bit 位置, 之后除以 或 模 64/seatnum 就可以得到结果。可以看成是两种模数运算, 一种是 64 模数运算, 一种是 seatnum(100)模数运算。区别在于, 64模数运算从下标 0 开始, 座位从下标 1 开始, 所以在确定数据是 第几个 bit 位时, 需要进行 - 1操作。

```
int x = 9, y = 97;
int longNumber = (x - 1) * seatnum + (y - 1);
int longBegin = (longNumber) >>> 6; // >>> 6 == / 64
int longBias = (longNumber & 0x3F); // & 0x3F == % 64
System.out.println(x + " th coach " + y + " th seat lie at " + longBegin + " byte " + longBias + " position(start all at 0)");
// 执行结果:
// 9 th coach 97 th seat lie at 14 byte 0 position(start all at 0)
```

上述计算回答了问题 1: 也举出一个例子: 9 号车厢的 97 座位在 下标为 14 的Long块的下标 0 处。

```
int m = 7, n = 0;
int nNumber = m * 64 + n; // m = 3, n = 0, represent the 4 [long] position[1]
int coachBegin = nNumber / seatnum + 1;
int seatBegin = nNumber % seatnum + 1;
System.out.println(m + " [long] and the " + n + " bias lie at coach: " + coachBegin + " seat: " + seatBegin);
// 执行结果:
// 7 [long] and the 0 bias lie at coach: 5 seat: 49
```

上述计算回答了问题 2: 也举出一个例子: 下标为 7 的 Long块偏移 0 处是 第 5 号车厢的 第 49 个座位。

初始化过程

初始化需要完成两步:

1. 第一步, 把第0 行的所有数据赋值1 默认 0 表示座位为空, 1表示座位满。实际上没有 0 号站点, 但由于下标的设置, 所以在初始化过程中, 需要把第 0 行的 15 个Long块全部赋值为 0xFFFFFFFFFFFFFFFFL。实际在数据转换中, 64bit全1 表示 - 1, 会观察到, 完成第一步初始化后, 第 0 行的 15 个Long 块全部变为 -1。

```
for(int i = 0; i <= routenum; i++){ // 一共 5 趟列车
    for(int k = 0; k < (coachnum * seatnum)/64 + 1; k++){
        BitMapTotal.get(i)[k][0] = 0xFFFFFFFFFFFFFFFFL;
    }
}
```

2. 第二步，把每一行最后一个Long块补全的部分添 1。实际只有900个座位，但15个Long块有960个座位，所以需要把除了第0行外的每行最后60个bit赋值为 1。这个赋值操作需要先构造出，前 4 bit为0，后60 bit为1的数据。这样的数据可以由 0xFFFFFFFFFFFFFFFF，经过无符号数右移得到。

```
int last = (coachnum * seatnum) / 64;
int start = coachnum * seatnum; // 需要补 1 的bit位开始
int end = (last + 1) * 64; // 需要补 1 的 bit 位结束
long originalValue = 0xFFFFFFFFFFFFFFFFL;
long secondValue = originalValue >>> (64 - (end - start));
// 由于需要补60位，实际是全1右移 4 位得到

for(int i = 0; i <= routenum; i++){
    for(int l = 1; l <= stationnum; l++){
        BitMapTotal.get(i)[last][l] = secondValue;
    }
}
```

完成这两步初始化之后，打印结果如下：

```
0 -1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
1 0000000000000001152921504606846975
2 0000000000000001152921504606846975
3 0000000000000001152921504606846975
4 0000000000000001152921504606846975
5 0000000000000001152921504606846975
6 0000000000000001152921504606846975
7 0000000000000001152921504606846975
8 0000000000000001152921504606846975
9 0000000000000001152921504606846975
10 0000000000000001152921504606846975
```

查询过程 Inquiry

查询过程相对简单，由于已经知道列车的起点站和终点站，只需要对列车起点站到终点站中，每一列的数据进行操作，希望求得全0的满足条件的bit，再取反，利用Long.bitCount进行判断，求出1的数目。位运算可以大大缩短操作的时间。

```
@Override
public int inquiry(int route, int departure, int arrival){
    lock[route].lock();
    int totalLeft = 0;
    long compare = 0L;
    for(int k = 0; k < (coachnum * seatnum) / 64 + 1; k++){
        compare = 0L;
        for(int l = departure; l < arrival; l++){
            compare = compare | BitMapTotal.get(route)[k][l];
        }
    }
}
```

```

        totalLeft += Long.BitCount(~compare);
    }
    lock[route].unlock();
    return totalLeft;
}

```

买票过程 Buy

首先需要有位运算的一个前置知识，想要知道Long块的最后一个 1 出现的位置。

```

假设 n          = 11001011100110100
那么 n - 1      = 11001011100110011
所以 n ^ (n - 1) = 0000000000000111

```

将原来的数据和 - 1 后的数据进行异或，得到的1的个数，也就是原来 n 中最后的1 出现的位置。这里我们实际希望找到在出发站到到达站(包含前不包含后)的所有站点中，该位置上的 bit 都是 0，这样就说明这个位置上为可卖的票。现在的 Long.BitCount方法和 (n ^ (n - 1)) 可以得到每一个 Long 块上，最后一个 1 出现的位置。所以不妨取反，得到最后一个 0 出现的位置。例如初始化时，每一个Long块都为 64位的0，所以返回值为 1,表示最低位即满足条件。在这里的设计中，最先卖出的是 第1个车厢第64号座位上的票。

```

@Override
public Ticket buyTicket(String passenger, int route, int departure, int arrival){
    lock[route].lock();
    long compare = 0L;
    long compareMinus = 0L;
    long compareNOR = 0L;

    for(int k = 0; k < (coachnum * seatnum)/64 + 1; k++){
        compare = 0L;    // 每一轮需要清零，因为只相对于特定的 bit 块来计算，彼此是独立
        的
        for(int l = departure; l < arrival; l++){
            compare = compare | BitMapTotal.get(route)[k][l];
        }
        if(compare == ~0L && k == coachnum * seatnum / 64){
            lock[route].unlock();
            return null;
        }
        if(compare == ~0L && k < coachnum * seatnum / 64){
            continue;
        }
        compareMinus = (~compare) - 1;
        compareNOR = (~compare) ^ compareMinus;
        int Left = long.BitCount(compareNOR);
        Ticket buyItem = new Ticket();
        buyItem.tid = Counter.getAndIncrement();
        buyItem.passenger = passenger;
        buyItem.route = route;
    }
}

```

```

        buyItem.coach = (k * 64 + (64 - Left))/ seatnum + 1;
        buyItem.seat = (k * 64 + (64 - Left)) % seatnum + 1;
        buyItem.departure = departure;
        buyItem.arrival = arrival;

        long originalValue = 1L;
        long shiftValue = originalValue << (Left - 1);
        for(int ll = departure; ll < arrival; ll++){
            BitMapTotal.get(route)[k][ll] = BitMapTotal.get(route)[k][ll] |
shiftValue;
        }
        lock[route].unlock();
        return buyItem;
    }
    lock[route].unlock();
    return null;
}

```

这里需要特殊考虑 `Long = 0xFFFFFFFFFFFFFFFFL` 的情况，此时的 `compare` 和 `compare - 1` 由于在溢出情况下，反而判断失败。所以需要特别考虑 `compare = 0xFFFFFFFFFFFFFFFFL` 的情况。最后在卖出票之后，通过之前的转换关系，确定是哪一趟列车的哪一个座位。之后需要对特定的座位进行赋值 1 操作，只需要将 1L 左移到对应的 bit 位，进行或操作即可。

退票过程

退票过程和买票类似，这里需要进行一个正确性检查。在这里的检查是，如果对应的座位(需要退的票的起点站和终点站之间)，出现空位。说明要退的票不合法，返回 `false`。退票需要将原先的座位转换为对应的 Long 块的位置和偏移，并完成赋 0 操作。赋 0 需要进行与操作。将原来的 1 左移到需要的位置后，取反，和对应的 Long 块相与。

```

@Override
public boolean refundTicket(Ticket ticket){
    int Refroute = ticket.route;
    int Refcoach = ticket.coach;
    int Refdeparture = ticket.departure;
    int Refarrival = ticket.arrival;
    int RefSeat = ticket.seat;
    int longNumber = (Refcoach - 1) * seatnum + RefSeat - 1;
    int longBegin = (longNumber) >>> 6;
    int longBias = (longNumber & 0x3F);

    long originalValue = 1L;
    long secondValue = originalValue << (64 - longBias - 1);
    long startCompare = 0L;
    lock[ticket.route].lock();

    for(int l = Refdeparture; l < Refarrival; l++){
        startCompare = startCompare | BitMapTotal.get(Refroute)[longBegin][l];
    }
}

```

```

        if((startCompare & secondValue) != secondValue){
            lock[ticket.route].unlock();
            return false;
        }

        for(int l = Refdeparture; l < Refarrival; l++){
            BitMapTotal.get(Refroute)[longBegin][l] = BitMapTotal.get(Refroute)
[longBegin][l] & (~secondValue);
        }
        lock[ticket.route].unlock();
        return true;
    }
}

```

测试程序设计思路

测试程序，将总的测试次数平均分配到每个线程。设计一个随机变量，如果随机变量的值为 0,1,2 表示执行买操作。如果随机变量的值为 3，执行退票操作，随机变量的值为 4-9,执行查询操作。这样保证了查询次数的分配比例。设置全局的计数数组(记录三种操作的时间和次数)，并设置每个线程局部的计数(三种操作的时间和计数)，在每个线程结束时，写入数组中。

由于需要设置每个线程唯一的变量作为区分，则可以设计：

```

class ThreadToAllocate{
    private AtomicInteger newId = new AtomicInteger(0);

    private ThreadLocal<Integer> threadGetId =
        new ThreadLocal<Integer>(){
            @Override protected Integer initialValue(){
                return newId.getAndIncrement();
            }
        };
    public int get(){
        return threadGetId.get();
    }
}

```

从而确定本地变量写入数组中的哪一位。设计进行 turnNum次运算，设置全局的记录，进行总和的运算。

```

static int[] CountBuy;
static int[] CountInq;
static int[] CountRef;
static long[] BuyTime;
static long[] RefTime;
static long[] InqTime;

```

为记录每一轮，每个线程运行的次数和时间。

```
static long GlobalBuy;
static long GlobalInq;
static long GlobalRef;
static long GlobalThroughput;
// 上述为 turnNum 轮次平均
int totalBuy = 0;
int totalInq = 0;
int totalRef = 0;
long totalBuyTime = 0;
long totalInqTime = 0;
long totalRefTime = 0;
// 上述为每一轮的测试结果。
```

为记录20轮之后的总数和总吞吐量，计算平均值。采用 `ArrayList<List>` 的列表，为每个线程记录售出票的列表，并把所有线程串成一个列表。一共 3 个 for 循环，最外层希望进行 `turnNum` 次测试，求出总的平均值。中间的 for 循环对线程进行遍历，里层的 for 循环表示需要每个线程进行 `testnum / threadnum` 次测试。

```
static ArrayList<List<Ticket>> hasBeensold;
```

```
public class Test {
    static int refRatio = 10;
    static int buyRatio = 20;
    static int inqRatio = 30;
    static int testnum = 10000;
    static int[] CountBuy;
    static int[] CountInq;
    static int[] CountRef;
    static long[] BuyTime;
    static long[] RefTime;
    static long[] InqTime;
    static ThreadToAllocate threadId;
    static ArrayList<List<Ticket>> hasBeensold;
    static long GlobalBuy;
    static long GlobalInq;
    static long GlobalRef;
    static long GlobalThroughput;

    public static void main(String[] args) throws InterruptedException {
        // *****
        // routenum, coachnum, seatnum, stationnum, threadnum

        int turnNum = 20;
        for(int turn = 0; turn < turnNum; turn++){
            CountBuy = new int[threadnum];
            CountInq = new int[threadnum];
            CountRef = new int[threadnum];
            BuyTime = new long[threadnum];
            RefTime = new long[threadnum];
```



```

        InqTime = new long[threadnum];
        final TicketingDS tds = new
TicketingDS(routenum,coachnum,seatnum,stationnum,threadnum);
        //
        *****
        // initialization
        hasBeensold = new ArrayList<List<Ticket>>();
        threadId = new ThreadToAllocate();
        // 4, 8 , 16, 32, 64
        Thread[] threadsToCom = new Thread[threadnum];
        // 30 : 10 : 60 method Call
        // verage conduct
        int Averagecount = testnum / threadnum;

        // Initializting ThreadLocal Variables
        for(int i = 0; i < threadnum; i++){
            List<Ticket> threadBuyingTicket = new ArrayList<Ticket>();
            hasBeensold.add(threadBuyingTicket);
        }

        for(int i = 0; i < threadnum; i++){
            threadsToCom[i] = new Thread(()-> {
                Random rand = new Random();
                long threadBuyTime = 0;
                long threadRefTime = 0;
                long threadInqTime = 0;
                int countBuy = 0;
                int countRef = 0;
                int countInq = 0;
                // methodList = [buyTicket | buyTicket | buyTicket | refund |
inquiry | inquiry | inquiry | inquiry | inquiry | inquiry]
                for(int testRound = 0; testRound < Averagecount; testRound++){
                    int methodCount = rand.nextInt(10);
                    if(methodCount < 3){ // buy
                        countBuy += 1;
                        Ticket ticketBuy = new Ticket();
                        String passenger = "passenger" +
rand.nextInt(testnum);

                        int route = rand.nextInt(routenum) + 1;
                        int departure = rand.nextInt(stationnum - 1) + 1;
                        int arrival = departure + rand.nextInt(stationnum -
departure) + 1;

                        // preparing for buy method

                        long buyTimeStart = System.nanoTime();
                        ticketBuy = tds.buyTicket(passenger, route, departure,
arrival);

                        long buyTimeEnd = System.nanoTime();
                        threadBuyTime += buyTimeEnd - buyTimeStart;

                        hasBeensold.get(threadId.get()).add(ticketBuy);
                    }
                }
            });
        }
    }

```

```

        else if(methodCount == 3){ // refund
            if(hasBeensold.get(threadId.get()).size() == 0){
                continue;
            }
            // ThreadToAllocate --> thread
            int n =
rand.nextInt(hasBeensold.get(threadId.get()).size());
            Ticket ticketRefund =
hasBeensold.get(threadId.get()).remove(n);
            // prepare
            if(ticketRefund == null){
                continue;
            }
            countRef += 1;
            long refundTimeStart = System.nanoTime();
            boolean flag = tds.refundTicket(ticketRefund);
            long refundTimeEnd = System.nanoTime();
            threadRefTime += refundTimeEnd - refundTimeStart;
        }

        else{ // inquiry
            countInq += 1;
            Ticket inqTicket = new Ticket();
            inqTicket.passenger = "Passenger" +
rand.nextInt(testnum);

            inqTicket.route = rand.nextInt(routenum) + 1;
            inqTicket.departure = rand.nextInt(stationnum - 1) +
1;

            inqTicket.arrival = inqTicket.departure +
rand.nextInt(stationnum - inqTicket.departure) + 1;
            long inqTimeStart = System.nanoTime();
            inqTicket.seat = tds.inquiry(inqTicket.route,
inqTicket.departure, inqTicket.arrival);
            long inqTimeEnd = System.nanoTime();
            threadInqTime += inqTimeEnd - inqTimeStart;
        }
    }
    BuyTime[threadId.get()] = threadBuyTime;
    RefTime[threadId.get()] = threadRefTime;
    InqTime[threadId.get()] = threadInqTime;
    CountRef[threadId.get()] = countRef;
    CountBuy[threadId.get()] = countBuy;
    CountInq[threadId.get()] = countInq;
});
}

long finalstart = System.nanoTime();
for(int i = 0; i < threadnum; i++){
    threadsToCom[i].start();
}

for(int i = 0; i < threadnum; i++){
    threadsToCom[i].join();
}

```

```

        long finalend = System.nanoTime();

        int totalBuy = 0;
        int totalInq = 0;
        int totalRef = 0;
        long totalBuyTime = 0;
        long totalInqTime = 0;
        long totalRefTime = 0;
        for(int i = 0; i < threadnum; i++){
            totalBuy += CountBuy[i];
            totalInq += CountInq[i];
            totalRef += CountRef[i];
            totalBuyTime += BuyTime[i];
            totalInqTime += InqTime[i];
            totalRefTime += RefTime[i];
        }

        GlobalBuy += totalBuyTime / totalBuy;
        GlobalInq += totalInqTime / totalInq;
        GlobalRef += totalRefTime / totalRef;
        GlobalThroughput += 1000000L *(totalBuy + totalInq +
totalRef)/(finalend - finalstart);

    }
    System.out.println("Average BuyTime = " + GlobalBuy / turnNum);
    System.out.println("Average RefTime = " + GlobalInq/ turnNum);
    System.out.println("Average InqTime = " + GlobalRef / turnNum);
    System.out.println("Throughput = " + GlobalThroughput/turnNum);
}
}

```

CLH实现

实现了课本的 CLHLock，其具体实现如下: 设置了可以重复利用的队列锁。

```

class Qnode{
    public AtomicBoolean locked = new AtomicBoolean(true);
}

class CLHLock{
    AtomicReference<Qnode> tail = new AtomicReference<Qnode>();
    ThreadLocal<Qnode> myNode = new ThreadLocal<Qnode>();
    ThreadLocal<Qnode> myPred = new ThreadLocal<Qnode>();

    public CLHLock(){
        tail.set(new Qnode());
        tail.get().locked.set(false);
    }

    public void lock(){
        if(myNode.get() == null)    myNode.set(new Qnode());
    }
}

```

```
        myNode.get().locked.set(true);
        Qnode pred = tail.getAndSet(myNode.get());
        myPred.set(pred);
        while(pred.locked.get()){};
    }

    public void unlock(){
        myNode.get().locked.set(false);
        myNode.set(myPred.get());
    }
}
```

对于每一个车厢，设置锁的数组，并对于请求的每个车厢，分配其中的一把锁。在车厢数增多的情况下，并行性良好。

```
public static CLHLock[] lock;

lock = new CLHLock[routenum + 1];
for(int i = 0; i <= routenum; i++){
    lock[i] = new CLHLock();
}
```

正确性要求

程序实现的锁，满足可线性化要求，无死锁、无饥饿、无锁、无等待。证明如下: 由于每个列车分配一把锁，如果对于不同的列车进行请求，需要抢夺不同的锁。列车的数量决定了争用的情况。死锁、饥饿等情况均出现在对同一把锁，也就是对同一个列车的锁的抢夺上。由于三个操作，均在一开始就上锁，程序返回之前解锁，相当于一把大锁的情况。抢到锁的线程执行操作，其余线程此时只能阻塞等待，当线程执行完之后，才释放锁，交给其他希望得到锁的线程。由于程序三个操作的执行是完整的，临界区保护了所有全局共享资源的全部操作，所以对于单个线程每个操作都完整执行完，满足顺序一致性、静态一致性、可线性化。对于同一把锁，使用完之后线程释放，程序执行有限次操作，不存在死锁、饥饿等问题。

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  PORTS

● [user014@panda7 ~]$ cd myproject/
● [user014@panda7 myproject]$ ./verilin.sh
route: 3, coach: 3, seatnum: 5, station: 5, refundRatio: 10, buyRatio: 30, inquiryRatio: 60
history size = 4000, region size = 4000, max_region_size = 1
Verification Finished.
37ms
VeriLin

kettingDS.java  J Replay.java  J GenerateHistory.java  $ mysh.sh  J Test.java  $ replay

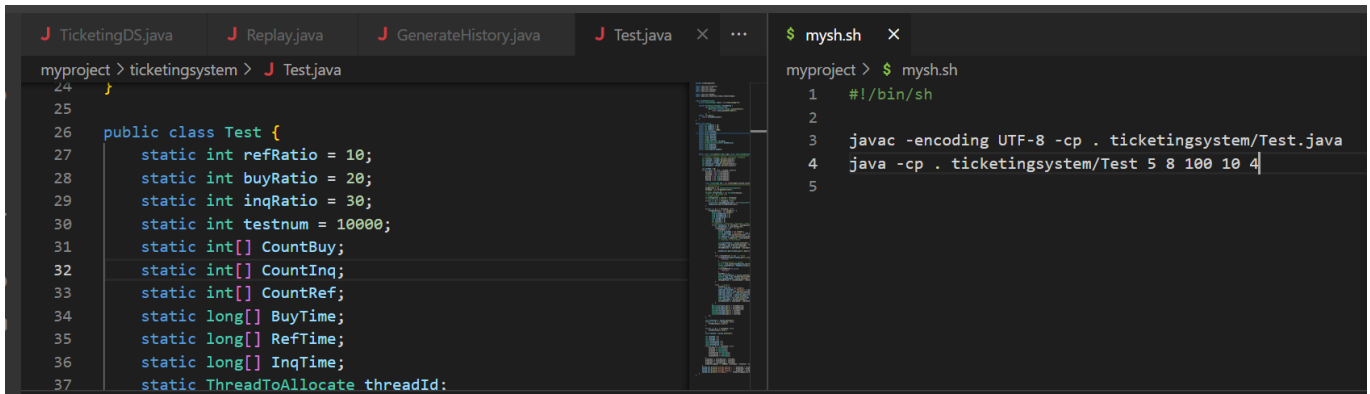
myproject > $ verilin.sh
1  #!/bin/sh
2
3  javac -encoding UTF-8 -cp . ticketingsystem/GenerateHistory.java
4  java -cp . ticketingsystem/GenerateHistory 4 1000 0 0 > history
5  java -Xss1024m -Xmx400g -jar VeriLin.jar 4 history 1 failedHistory
6

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  PORTS

● [user014@panda7 myproject]$ ./clean.sh
⊗ [user014@panda7 myproject]$ ./clean.sh
rm: cannot remove 'ticketingsystem/*.class': No such file or directory
rm: cannot remove 'failedHistory': No such file or directory
rm: cannot remove 'history': No such file or directory
● [user014@panda7 myproject]$ ./verilin.sh
route: 3, coach: 3, seatnum: 5, station: 5, refundRatio: 10, buyRatio: 30, inquiryRatio: 60
history size = 3998, region size = 3977, max_region_size = 4
Verification Finished.
41ms
VeriLin
● [user014@panda7 myproject]$ ./verilin.sh
route: 3, coach: 3, seatnum: 5, station: 5, refundRatio: 10, buyRatio: 30, inquiryRatio: 60
history size = 4000, region size = 3910, max_region_size = 64
Verification Finished.
56ms
VeriLin
```

性能分析:

5 个车次 · 8 个 车厢, 100 个座位 · 10 个车站 · 查询 10000 次 (线程数 4, 8, 16, 32, 64)



The screenshot shows an IDE with several tabs: TicketingDS.java, Replay.java, GenerateHistory.java, and Test.java. The Test.java file is open, showing a public class Test with static variables and arrays. To the right, a terminal window titled 'mysh.sh' shows the execution of a shell script. The script contains the following commands:

```
1 #!/bin/sh
2
3 javac -encoding UTF-8 -cp . ticketingsystem/Test.java
4 java -cp . ticketingsystem/Test 5 8 100 10 4
5
```

- threadNum = 4

```
● [user014@panda7 myproject]$ ./mysh.sh
Average BuyTime = 1186
Average RefTime = 1648
Average InqTime = 1091
Throughput = 1197
```

- threadNum = 8

```
● [user014@panda7 myproject]$ ./mysh.sh
Average BuyTime = 2701
Average RefTime = 3499
Average InqTime = 2844
Throughput = 1132
```

- threadNum = 16

```
● [user014@panda7 myproject]$ ./mysh.sh
Average BuyTime = 4129
Average RefTime = 4340
Average InqTime = 4017
Throughput = 1251
```

- threadNum = 32

```
● [user014@panda7 myproject]$ ./mysh.sh
Average BuyTime = 7947
Average RefTime = 7838
Average InqTime = 8038
Throughput = 1134
```

- threadNum = 64

```
● [user014@panda7 myproject]$ ./mysh.sh
Average BuyTime = 22784
Average RefTime = 22872
Average InqTime = 23688
Throughput = 995
```

50 个车次 · 20 个车厢 · 100 个座位 · 30 个车站 · 每个线程 100 万条操作

- threadNum = 4

```
● [user014@panda7 myproject]$ ./mysh.sh
Average BuyTime = 1368
Average RefTime = 1205
Average InqTime = 1219
Throughput = 1723
```

- threadNum = 8

```
● [user014@panda7 myproject]$ ./mysh.sh
Average BuyTime = 1436
Average RefTime = 1296
Average InqTime = 1315
Throughput = 3215
```

- threadNum = 16

```
● [user014@panda7 myproject]$ ./mysh.sh
Average BuyTime = 2065
Average RefTime = 1878
Average InqTime = 1953
Throughput = 5077
```

- threadNum = 32

```
● [user014@panda7 myproject]$ ./mysh.sh
Average BuyTime = 3460
Average RefTime = 3218
Average InqTime = 3477
Throughput = 6751
```

- threadNum = 64

```
● [user014@panda7 myproject]$ ./mysh.sh
Average BuyTime = 5997
Average RefTime = 5762
Average InqTime = 6315
Throughput = 8650
```