

性能评测报告

2021E8013282114 吴钰轩

Saturday 11th December, 2021

1 并发数据结构设计

1.1 TrainTicket 类

TrainTicket 类封装的对象是某一个车次上所有座位的售票情况。

在该类的构造函数中，定义该车次的总座位数，车厢数和经过站台数，并且初始化一个 AtomicLong 类型的数组，用来记录每个座位在各个站台的售票情况，代码如下：

```
public TrainTicket(int coachnum, int seatnum, int stationnum){
    seatNum = coachnum * seatnum;
    coachNum = coachnum;
    stationNum = stationnum;
    seatState = new AtomicLong[seatNum];
    for(int i = 0; i < seatNum; i++){
        seatState[i] = new AtomicLong(0);
    }
}
```

对买票操作，在购票时对座位进行上锁并分配座位，有空余座位时返回座位数，没有空余座位时返回-1。对每个座位的售出情况，采用了二进制的方式进行表示，对从 departure 到 arrival 的购票请求，将 1 左移 (departure - arrival) 位并减 1，之后再左移 departure 位，这样得到的二进制表示中，数字 0 的位表示此次购票不经过该站，数字 1 的位表示此次购票需要经过该站。同样的，在 AtomicLong 的数组中也用这样的二进制数记录售票情况。在判断是否有空余座位时，遍历搜索 seatState 数组，将座位状态和购票状

态进行按位与操作，若两者没有交集则得到结果 0，此时可以在该座位上购票，并用 compareAndSet 更新该座位的状态。代码如下：

```
public int lockForSeat(final int departure, final int arrival){
    // use binary to record which stations are used
    long passStations = (1 << (arrival - departure)) - 1;
    passStations = passStations << departure;

    // search for empty seat
    for(int i = 0; i < seatNum; i++){
        long temp = seatState[i].get();
        // add stations to empty seat
        // spin lock
        while((temp & passStations) == 0){
            if(seatState[i].compareAndSet(temp, (temp |
                passStations))){
                return i + 1;
            }
            temp = seatState[i].get();
        }
    }

    // no more seat
    return -1;
}
```

对于退票操作，同样定义方法 unlockForSeat，计算得到要退票的座位 seatnum 上的退票站台的二进制表示 passStations，用 compareAndSet 更新 seatState 的新值，代码如下：

```
public boolean unlockForSeat(final int seatnum, final int departure,
    final int arrival){

    long passStations = (1 << (arrival - departure)) - 1;
    passStations = passStations << departure;

    // spin lock
    while(true){
        long temp = seatState[seatnum - 1].get();
```

```

        if(seatState[seatnum - 1].compareAndSet(temp, (temp &
            ~passStations))){
            return true;
        }
    }
}

```

最后定义查询余票的方法 `searchForSeat`，采用按位与的方式遍历 `seatState` 数组，在查询过程中不对操作上锁，所以不能保证并发的购票和退票操作不会对查询造成影响。代码如下：

```

public AtomicInteger searchForSeat(final int departure, final int
    arrival){
    AtomicInteger ticketNum = new AtomicInteger(0);
    long passStations = (1 << (arrival - departure)) - 1;
    passStations = passStations << departure;

    // search for tickets
    for(int i = 0; i < seatNum; i++){
        long temp = seatState[i].get();
        if((passStations & temp) == 0){
            ticketNum.getAndIncrement();
        }
    }

    return ticketNum;
}

```

1.2 TicketingDS 类

定义 `ticketId` 表示 `ticket` 的 id 标号，每个 `ticket` 有唯一不同的 id。定义 `trains` 数组，数组元素为 `TrainTicket` 类型，表示每一个车次的票务信息。定义 `ConcurrentHashMap` 类型的字典 `soldTicket` 记录已售出的票信息。

在构造函数中，根据传入参数初始化以上变量，代码如下：

```

public TicketingDS(int routenum, int coachnum, int seatnum, int
    stationnum, int threadnum){
    trains = new TrainTicket[routenum];
}

```

```

        ticketId = new AtomicInteger(1);
        for(int i = 0; i < routenum; i++){
            trains[i] = new TrainTicket(coachnum, seatnum, stationnum);
        }
        // ConcurrentHashMap for multiple situation
        soldTicket = new ConcurrentHashMap<Long, Ticket>();
    }

```

在购票操作中, 根据车次调用对应 trains[route-1] 的 lockForSeat 方法, 并记录 ticket 的信息, 计算购得车票的车厢号和座位号, 并添加到 soldTicket 中, 代码如下:

```

public Ticket buyTicket(String passenger, int route, int departure,
    int arrival){
    // buy ticket
    int seat = trains[route - 1].lockForSeat(departure - 1, arrival
        - 1);
    if(seat < 0){
        return null;
    }
    // create ticket info
    Ticket ticket = new Ticket();
    ticket.tid = ticketId.getAndIncrement();
    ticket.passenger = passenger;
    ticket.route = route;
    ticket.departure = departure;
    ticket.arrival = arrival;
    // calculate coach and seat
    ticket.coach = ((seat - 1) / (trains[route - 1].seatNum /
        trains[route - 1].coachNum)) + 1;
    ticket.seat = ((seat - 1) % (trains[route - 1].seatNum /
        trains[route - 1].coachNum)) + 1;
    soldTicket.put(ticket.tid, ticket);
    return ticket;
}

```

在退票操作中, 根据票务信息重新计算出座位号, 调用 trains[route-1] 的 unlockForSeat 方法删除已购得的车票, 代码如下:

```

public boolean refundTicket(Ticket ticket){
    // info error
    if(!soldTicket.containsKey(ticket.tid) || !(ticket ==
        soldTicket.get(ticket.tid))){
        return false;
    }

    // calculate seat
    int seat = (ticket.coach - 1) * (trains[ticket.route -
        1].seatNum / trains[ticket.route - 1].coachNum) +
        ticket.seat;

    // refund ticket
    if(trains[ticket.route - 1].unlockForSeat(seat, ticket.departure
        - 1, ticket.arrival - 1)){
        return soldTicket.remove(ticket.tid, ticket);
    }

    return false;
}

```

查询操作调用 trains[route-1] 中的 searchForSeat 方法，代码如下：

```

public int inquiry(int route, int departure, int arrival){
    int restSeat = trains[route - 1].searchForSeat(departure - 1,
        arrival - 1).get();
    return restSeat;
}

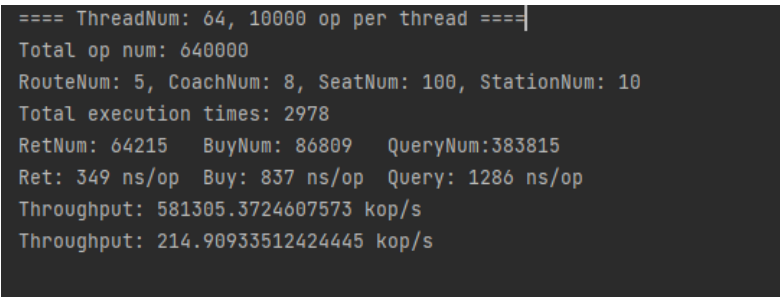
```

2 测试程序设计

Test.java 程序基本复制于 Trace.java 中的 Trace 类，执行过程中对各种操作总时间和总次数都做了统计，并记录了所有操作的执行时间，用于计算程序的吞吐量。下面给出在如下参数情况下的运行结果：

```
final static int[] threadnums = {1, 2, 4, 8, 16, 32, 64};
final static int routenum = 5;
final static int coachnum = 8;
final static int seatnum = 100;
final static int stationnum = 10;

final static int testnum = 10000;
final static int retpc = 10; // return ticket operation is 10% percent
final static int buypc = 40; // buy ticket operation is 30% percent
final static int inqpc = 100; //inquiry ticket operation is 60%
    percent
```



```
==== ThreadNum: 64, 10000 op per thread ====
Total op num: 640000
RouteNum: 5, CoachNum: 8, SeatNum: 100, StationNum: 10
Total execution times: 2978
RetNum: 64215   BuyNum: 86809   QueryNum:383815
Ret: 349 ns/op  Buy: 837 ns/op  Query: 1286 ns/op
Throughput: 581305.3724607573 kop/s
Throughput: 214.90933512424445 kop/s
```

图 1: result

3 系统正确性

3.1 可线性化

在 TrainTicket 类中，购票方法和退票方法均对所要操作的数据结构进行上锁，并采用原子操作 `compareAndSet()` 方法修改 `seatState` 数组元素的状态。退票和购票操作均可以看作是在

$$seatState[i].compareAndSet(temp, (temp | passStations))$$

上述操作完成的时间点上退票和购票在并发数据结构中完成。`lockForSeat()` 方法和 `unlockForSeat()` 方法中 `compareAndSet()` 所在的行就是两者的可线性化点。

```
while((temp & passStations) == 0){
    if(seatState[i].compareAndSet(temp, (temp | passStations))){
        return i + 1;
    }
    temp = seatState[i].get();
}
```

图 2: lockForSeat

如果 `compareAndSet()` 操作没有成功，线程会不断自旋并探测，重复 `compareAndSet()` 操作。对退票方法同理。

```
while(true){
    long temp = seatState[seatnum - 1].get();
    if(seatState[seatnum - 1].compareAndSet(temp, (temp & ~passStations))){
        return true;
    }
}
```

图 3: lockForSeat

3.2 deadlock-free

该程序是 `deadlock-free` 的，在死锁的过程中，需要两个进程相互等待，进程对请求的资源进行保持。而在本程序中，在循环等待的过程中，涉及到的操作只有：

seatState[i].compareAndSet(temp, (temp|passStations)

这样一个原子操作，并且在条件不满足时不会对 `temp` 的值进行修改，即没有对资源进行保持操作。要使得程序死锁，即说明两个进程在相互等待时，需要其中一个进程不断修改 `temp` 的值，这显然不可能出现，因为原子操作 `compareAndSet()` 一旦完成即退出循环。

3.3 starvation-free

由于使用了原子操作 `compareAndSet()`，在买票和退票的进程中，进程不会在等待时不断被其他进程所打断，一旦 `temp` 修改完成即退出等待，不会出现 starvation-free 的现象。

3.4 lock-free

购票和退票操作的可线性化点均在 `compareAndSet()` 操作上，那么当 `temp` 的值没有改变时，`compareAndSet()` 操作就可以完成，即完成购票和退票操作。能够对 `temp` 值进行修改的只有 `compareAndSet()` 操作，若所有进程都在等待状态，那么 `temp` 的值必然不变，此时任意进程都能完成 `compareAndSet()` 操作。所以，不存在所有进程均等待且 `temp` 值不断变化的情况，即至少有一个进程在同一时间能够执行。

3.5 wait-free

wait-free 要保证每个线程在有限步骤下完成，由于进程时 starvation-free，保证了每个进程都能得到执行。购票和退票操作在执行时只需要修改 `temp` 的值即可，即完成 `compareAndSet()` 操作，所以所有的进程均能在有限步内完成。

4 系统性能

在 route=5, coach=8, seat=100, station=10 的情况下, 分别对 thread 在各种情况下的吞吐量和执行时间进行测量。

4.1 程序总执行时间

在 Test.java 中采用 System.nanoTime() 测量程序执行时间, 结果如下图所示:

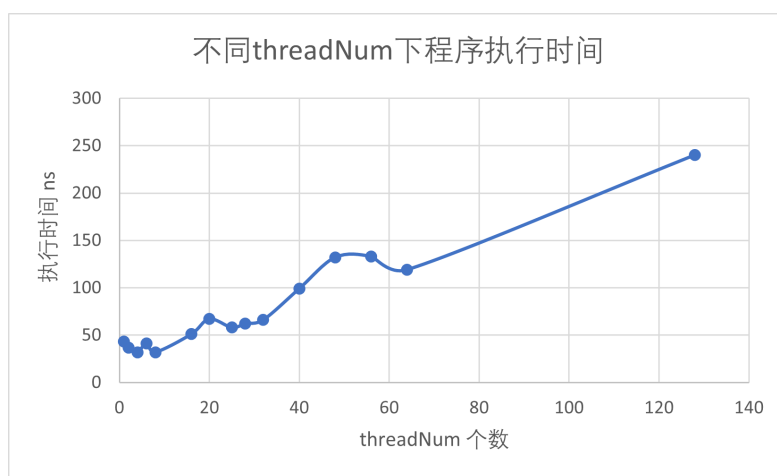


图 4: 程序执行时间和线程数关系

4.2 程序总吞吐量

每个线程执行操作 10000 次, 其中退票操作、购票操作和查询操作分别占 10%、30% 和 60%, 用总执行操作数除以执行总时间得到程序总吞吐量, 如下图:

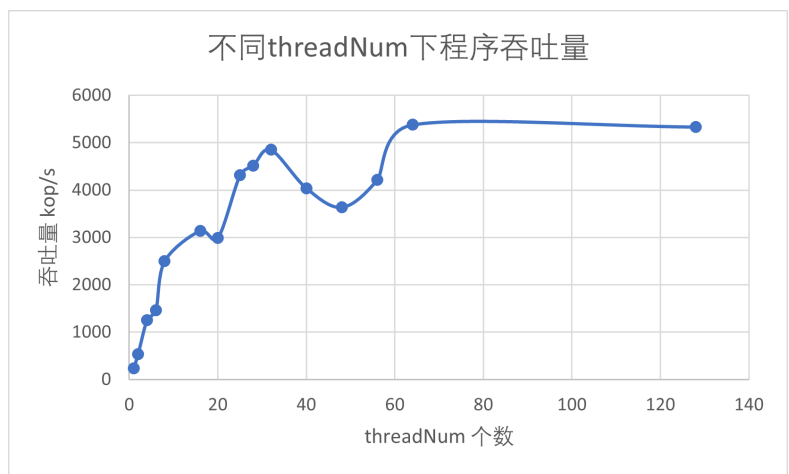


图 5: 程序总吞吐量和线程数关系