用于列车售票的可线性化并发数据结构

实验报告

姓名：申庆瑜

学号：202128015121006

Email：shen99855@outlook.com

实现语言：C++

时间：2021年11月

1. 数据结构设计

1.区间售票策略

对于火车售票系统，首先应考虑区间售票的问题：即，同一车次同一座位从起点（假设是北京）到终点（假设是上海）之间，可以按区段划分成不同的车票出售，即，从北京到郑州一张票，从郑州到上海还可以再卖一张。

注意：乘车区间数刚好为车站数-1.

所以，对于给定的车次，车厢，座位号。

乘车区间的表示应当是一个布尔型数组。假设车站数为6（编号0-5）

乘车区间的表示为

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0-1区间 | 1-2区间 | 2-3区间 | 3-4区间 | 4-5区间 |
| 1 | 1 | 1 | 1 | 1 |

如果购买0-3站点的一张票，

将前3个区间置零，票务中心剩余的票为

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0-1区间 | 1-2区间 | 2-3区间 | 3-4区间 | 4-5区间 |
| 0 | 0 | 0 | 1 | 1 |

该方法在DataStaion类中Rtickets数组实现。

2.动态座位策略

如果按照每一个座位进行分票，Ticket将是一个四维数组。

bool Tickets [Station\_num-1][Route\_num][Coach\_num][Seat\_num]

注：C++不支持布尔数组，故程序中采用char类型数组

考虑查询和购买车票时，只关心车次和起点终点，车厢号和座位号不发挥作用，可以将其合并，将每个车次整车上的余票数设成一个int型变量，

int Tickets[Station\_num-1][Route\_num];

在查询时，无需进行求和，可以极大提高查询效率。

在出售车票时，通过对除以座位总数和对座位总数取模可以快速分配车厢和座位号，不会损失售票效率。

查询余票检索的是对应区间所有子区间的余票的最小值。（参见DataStaion类中Inquire函数）

出售车票为区间剩余子区间余票最小值，如此保证了出售的同一车次，同一座位号车票的区间不交叠特性（参见DataStaion类中BuyTicket函数）

注意Tickets初始值是Coach\_num\*Seat\_num，访问过程中只减不增，这样不会发生座位分配重复的问题。

3.数据域拆分策略

被退还的车票不能重新加到Ticket上，原因在于会影响车厢和座位号的分配，所以，我将其额外记录在一个新的数组Rtickets：

bool Rtickets [Station\_num-1][Route\_num][Coach\_num][Seat\_num]

Rtickets是精确到每一个座位的票。

所以Tickets中存了没被卖过的票。

Rtickets中存了被卖过又被退了的票。

查询余票时，除了在Tickets中查，还应该在Rtickets中在查，返回的是二者的总和。购票也是先买没买过的票，再试图买二手票。

最不理想的情况是在Rtickets中查票，在Tickets中查票要比Rtickets中快得多。

4.细粒度锁

对于

int Tickets[Stationplusr\_num][Route\_num];

bool Rtickets[Stationplusr\_num][Route\_num][Coach\_num][Seat\_num] ;

分别划分R锁和RS锁

mutex Rlock[Route\_num];

mutex RSRlock[Route\_num][Seat\_num];

Rlock[i]用于锁定Tickets[Stationplusr\_num][i];

RSRlock[i][j] 用于锁定 Rtickets[Stationplusr\_num][i][Coach\_num][j]

这样，不同车次的初次购票请求可以天然地并发执行。

不同车次不同座位的二手票购票请求可以天然地并发执行。

5.买票时先查后锁策略

从TTAS锁的原理中获得启发，

由于查票是只读操作，在Rtickets当中买票时，不用着急加锁，可以在无锁情况下，先查找要买的票，找到待购车票后再加锁进行购票。

策略1使得本系统特性能与日常生活习惯保持一致，大大提高了售票效率

策略2-5都是为了加速，加速，加速！重要的事情说三遍。所以虽然我这里只用了简单的固定数组，没用什么队列啊，栈啊，也没用无锁数据结构。但是从我数据存储的设计和查票，买票，退票方法执行策略的设计上，天然地有高效率的优势。因为时间来不及，也因为我用C++来写，有太多底层的架构需要自己搭建，所以在并发加速方面，只有4和5两条策略，进一步加速呢，考虑到RTickets它是一个稀疏矩阵，可以做成哈希查找表，但是确实没有时间了。

1. 程序简单说明

1.开发平台：

采用Visual Studio 2017，Windows平台进行开发，除了标准库函数以外，没有使用第三方库。项目工程文件为[releaseticket\periallelThread.sln](releaseticket/periallelThread.sln)

注意必须定义#define Record才会生成trace.txt

生成的trace文件保存在：[releaseticket\periallelThread\data](releaseticket/periallelThread/data)目录下

所有线程并发的trace为[releaseticket\periallelThread\data\trace.txt](releaseticket/periallelThread/data/trace.txt)，其余的trace是不同线程各自的执行历史，trace0对应线程0，trace1对应线程1·，以此类推。

2.实验参数的设定

请打开base.h

//#define CULock 1//粗粒度锁

//IOLock 和Culock不能同时定义

#define Record // 生成并发trace

#define LocalTrace // 生成每个线程的trace

#ifdef Record

#define IOLock 1

//IOLock 和Culock不能同时定义

#define GUItrace 1

#endif

#define Thread\_Num 1

//线程数目

#define Route\_num 10//车次数目

#define Station\_num 8//站点数目

#define Coach\_num 10//车厢数目

#define Seat\_num 15//座位数目

#define Stationplusr\_num (Station\_num-1)

#define Full\_train\_num (Coach\_num\*Seat\_num)

#define Buypc 30//买票百分比

#define Retpc 10 //退票百分比

#define Inqpc 60//查票百分比

#define Tast\_num 100000//测试数目

1、注意性能测试与正确性检验应该分开进行。因为在生成trace时，需要不同线程同时访问一个文件输出流，所以需要一把额外的IO锁，才能保证生成正确的trace，这把额外的IO锁会带来系统额外开销。由于C++的随机数是伪随机数，所以我这里生成的trace的运行和不生成trace的运行它的执行历史其实是一样的。

重要：生成trece时，定义Record。不生成trcae时，不定义Record。

如果想要改变程序随机过程，请在main函数开始时重置随机数种子： srand(115200);

程序运行结束后，会打印runtime的值，runtime为系统运行总时间，单位是毫秒，可以作为评价性能的指标。

3.main函数介绍

int main()

{

//reset random seed here

srand(115200);

chrono::time\_point<chrono::steady\_clock> start = chrono::steady\_clock::now();

chrono::time\_point<chrono::steady\_clock> c = start;

thread\* Threads= new thread[Thread\_Num];

ThreadBase TB[Thread\_Num];

TicketServiceStation TS[Thread\_Num];

char fname[] = "data/trace0.txt";

//for queuelock

TB[0].lock.init\_space();

clock\_t StartTime = clock();//timer initer

cout << "start!\n";

char\*fn;

chronostart = chrono::steady\_clock::now();//refreshclock

for (int i = 0; i < Thread\_Num; i++)

{

fname[10] = char(i+48);

fn = new char[16];

for (int j = 0; j < 16; j++)

{

fn[j] = fname[j];

}

TB[i].ThreadBase\_init(i,fn);

TB[i].clock\_0 = StartTime;

TB[i].TS = TS + i;

//\*(Threads+i) = thread(&ThreadBase::adder, TB + i, 1000);

\*(Threads+i) = thread(&ThreadBase::trace, TB+i);

}

for (int i = 0; i < Thread\_Num; i++)

{

Threads[i].join();

}

long runtime = clock() - StartTime;

cout << "end processing!\n";

cout << "runtime=" << runtime;

CombineTrace.close();

return 0;

}

关键在于\*(Threads+i) = thread(&ThreadBase::trace, TB+i);这句，创建了Thread\_Num个并发的ThreadBase对象，并运行trace函数。

trace是仿照trace.java写的，trace函数的定义如下

void ThreadBase::trace()

{

srand(tid);//tid = id;

int rep = Tast\_num / Thread\_Num;

int i;

recorder.open(filename);

long long preTime;

long long postTime;

while (rep--)

{

int route = randeng(Route\_num); //随机数

int departure = randeng(Stationplusr\_num);

int arrival = randeng(Stationplusr\_num);

if (departure > arrival)//swap

{

int rem = departure;departure = arrival;arrival = rem;

}

arrival++;

{

i = randeng(Inqpc+Buypc+Retpc);

if (i < Inqpc)//查

{

preTime = chrononanoclock();

int leftTicket=(\*TS).inquiry(route,departure,arrival);

postTime = chrononanoclock();

recorder << preTime << " " << postTime << " " << tid << " " << "RemainTicket" << " "

<< leftTicket << " " << route+1 << " " << departure+1 << " " << arrival+1 << endl;

}

else if (i < Inqpc + Buypc)//买

{

preTime = chrononanoclock();

Ticket ticket = (\*TS).buyTicket("aperson",route, departure, arrival);

postTime = chrononanoclock();

if (ticket.tid > -1) //success

{

TS->sold\_tickets.push\_back(ticket);

recorder << preTime << " " << postTime << " " << tid << " " <<

"TicketBought" << " " << ticket.tid << " " << "aperson" << " " << ticket.route + 1 <<

" " << ticket.coach + 1 << " " << ticket.departure + 1 << " " << ticket.arrival + 1 << " "

<< ticket.seat + 1 << endl;

}

else//fail to buy

{

recorder << preTime << " " << postTime << " " << tid << " " << "TicketSoldOut"

<< " " << route + 1 << " " << departure + 1 << " " << arrival + 1 << endl;

}

}

//下续

//接上

else if (i < Inqpc + Buypc + Retpc)//退票

{

int r = (\*TS).sold\_tickets.size();

if (r)//有票可以退

{

r = randeng(r);

Ticket ticket = TS->sold\_tickets[r];//get it

vector<Ticket>::iterator iter = TS->sold\_tickets.begin();

TS->sold\_tickets.erase(iter+r);//pop

preTime = chrononanoclock();

bool succret = (\*TS).refundTicket(ticket);

postTime = chrononanoclock();

if (succret) //成功退票

{

recorder << preTime << " " << postTime << " " << tid << " "

<< "TicketRefund" << " " << ticket.tid << " " << "aperson" << " "

<< ticket.route + 1 << " " << ticket.coach + 1 << " " << ticket.departure + 1 << " "

<< ticket.arrival + 1 << " " << ticket.seat + 1 <<endl ;

}

else//失败退票

{

recorder << preTime << " " << postTime << " " << tid << " " << "ErrOfRefund" << endl;

}

}

}

else

{

cout << "err from rep(i)=" << i <<"from thread"<<tid<< endl;

break;

}

}

}

recorder.close();

cout << "end trace";

return;

}

TS是TicketServiceStaion类，就是要求实现的并发数据结构,但其实在我的程序实现当中这里他只是一个接口。

还有一个DataStation，才是真正的票务中心。这里就不展开讲了。

根据我这个系统设计的结构，以下情况下此系统性能将大幅提升：

1. 车次数较多（最好大于线程数）
2. 车厢座位数较多，至少是车箱数的十倍以上吧。
3. 查询百分比>购票百分比>退票百分比。
4. 程序运行
5. 正确性验证：

一定要定义#define Record

//#define CULock 1//粗粒度锁

//IOLock 和Culock不能同时定义

#define Record // 生成并发trace

#define LocalTrace // 生成每个线程的trace

#ifdef Record

#define IOLock 1

//IOLock 和Culock不能同时定义

#define GUItrace 1

#endif

#define Thread\_Num 1

//线程数目

#define Route\_num 3//车次数目

#define Station\_num 3//站点数目

#define Coach\_num 3//车厢数目

#define Seat\_num 3//座位数目

#define Stationplusr\_num (Station\_num-1)

#define Full\_train\_num (Coach\_num\*Seat\_num)

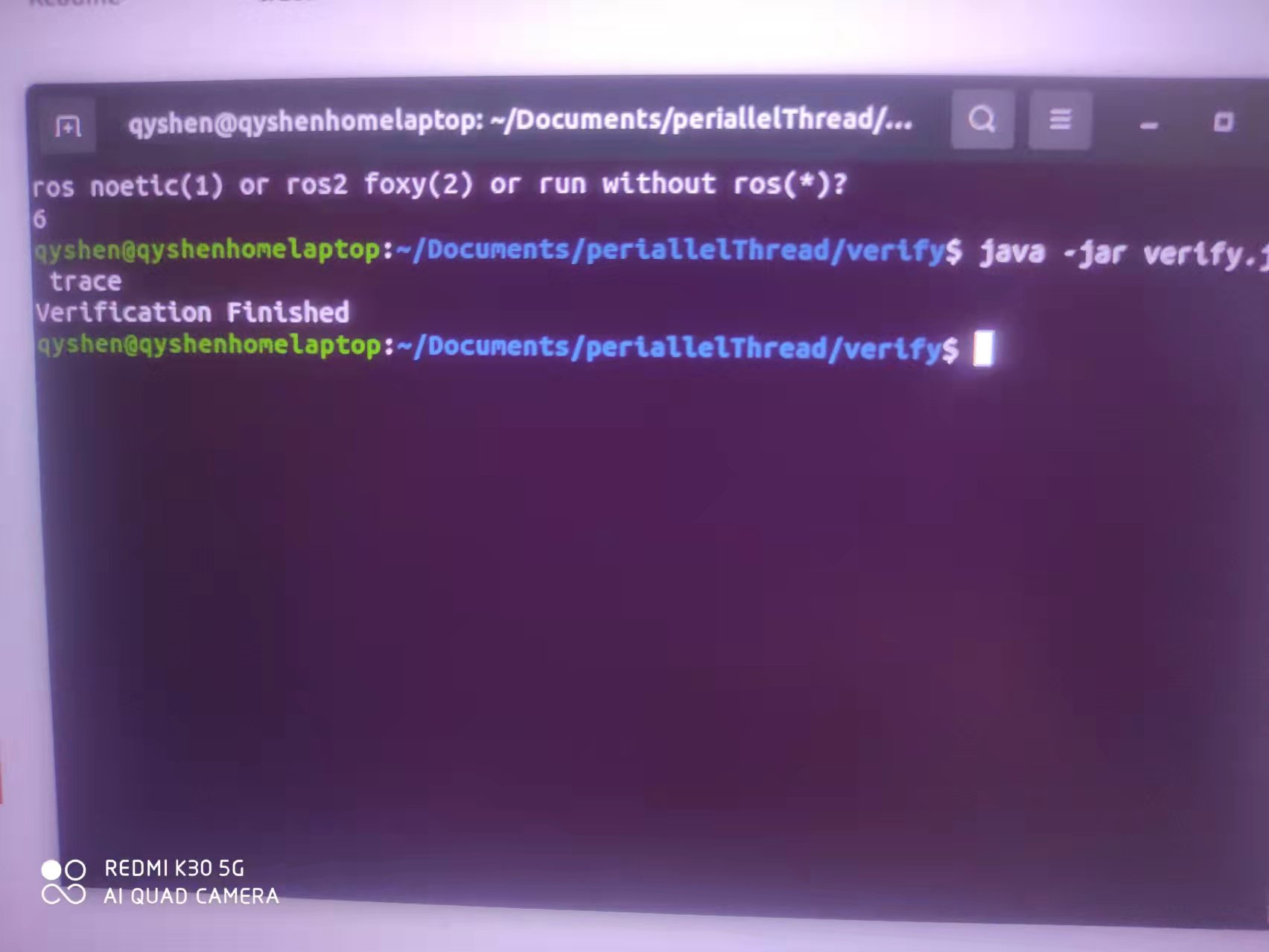
#define Buypc 30//买票百分比

#define Retpc 10 //退票百分比

#define Inqpc 0//查票百分比

#define Tast\_num 100000//测试数目

在只有买票和卖票的情况下验证正确性：



增加查询操作再次进行验证：（#define Inqpc 60）

1. 性能测试
2. 结果展示
3. 展望