# 实验报告

202118013229064 於修远

## 一、实验概述

### 1、实验背景

本实验设计并实现了一个用于列车售票场景的并发系统。通过编写 `buyTicket` 和 `refundTicket` 方法，实现了可线性化的并发售票、退票过程；通过编写 `inquiry` 方法，实现了并发无锁的余票查询功能。并且，在现有功能的基础上，还增加了一些优化设计的尝试，并在理论上分析了各种优化思想的适用场景。最后，性能评估显示，基础的实验设计在要求场景下已有较好的性能，而加入各种优化模块后的性能表现也基本论证了对各自最佳适用场景的分析。

### 2、应用场景

基于实验要求和评分环境的设置，本项目适用场景的限制如下所示。相关参数理论上都可以直接扩展，在后续的“数据结构设计”部分也会介绍目前的设计考虑和扩展方式。

- 出票次数至多为线程数的 12 万倍
- 列车车次数的上限为 64
- 每辆列车的总座位数（车厢数乘每车厢座位数）的上限为 16384
- 车站总数的上限为 32
- 测试场景中，查询、购票、退票操作的比例为 7 : 2 : 1

## 二、数据结构设计

### 1、座位为单位的余票结构

为了兼顾存储和操作效率，基于位图（bitmap）的余票表示方式是一种比较自然的想法。具体而言，假设每车次的总座位数为 N ，则可以为每一车次申请元素个数为 N 的 `AtomicIntegerArray` 数组。对于其中的每个元素，若由低到高第 k 位为 0 ，则表示该座位存在一张乘坐区间包含站点 k 到站点 k+1 的已售票。

显然，位图表示的位数决定了系统所支持的车站数的上限，当前为 33 。如果采用长的位图，如元素位宽为 64 的 `AtomicLongArray` ，则可以支持至多 65 个车站。但如果需要提供更多车站情况下的支持，将不得不引入锁或其他机制，或多或少会影响系统性能。此外，目前的整型位宽是在项目要求之下的一种折中方案。考虑到减少内存开销和减少 Cache 行替换、行对齐的需要，位图的位宽必然是在为 2 的幂次的前提下，越接近车站总数越好。

之所以要采用原子类型，是因为在这一版本的实现中，对座位位图的修改恰为购票和退票操作的可线性化点，必须保证更新的原子性。

具体的修改操作采用自旋的 CAS 来实现。不论是购票还是退票，开始访问位图数组时一定携带了一个区间位图，表示希望购票或退票的区间。对于退票，可以直接索引到要清空记录的位置，自旋地尝试将被退票的区间对应的比特位清 0 直至成功。对于购票，首先遍历位图数组，查找在所需购票的区间有完整空闲比特位的座位，随后自旋地尝试将购票区间的对应比特位置 1 。值得一提的是，由于可能存在并发的购票请求，所以购票操作的 CAS 失败后必须重新检查该座位的目标区间是否依然空闲，若否，则放弃这一座位，继续向后查找。

最后，对于查询操作，基于位图的实现可以保证查询的无锁化，只需原子地获取位图的当前值并与查询区间进行比对即可。当然，查询操作本身不是可线性化的，因为已查询部分可能又被退票和购票操作修改，但在串行场景下的正确性可以保证，并行场景下也一定能得到一个事务级的合理结果（至多忽略或额外统计若干个并发的购票、退票操作带来的影响）。

### 2、全局出票记录

座位级的位图虽然可以便捷地实现购票和退票的记录，但不足以保存已售票的完整信息。例如，对于连续的两位置 1 的比特位，仅靠位图无法判断这是一张跨越两站的车票还是两张间隔一站的车票；此外，还有一些必要信息（如购票人姓名等）也未被记录在内，导致无法应对“退假票”的场景需求。基于这些考虑，在原有系统上必须额外加入一层全局记录，作为购票后与退票前的过渡操作。

理论上说，全局的出票记录可以用任何一种数据结构实现，如链表或哈希，但为了更好的查询性能，本项目中选择了直接申请足够长度的数组。出票记录包括两个全局数组：记录哈希与乘客姓名，数组长度均为预设的最大操作数，这也等价于出票的最大数目，即 TID 的实际上限。

在购票操作中，如果查询到了可购票的座位，并成功执行了 CAS 操作，系统需要额外地把购票信息的哈希值原子地写入记录哈希数组，然后将姓名字符串替换进入乘客姓名数组，两个数组被写入位置的下标均为 TID 的值。至此，一张票才被真正地“售出”。

在这里，哈希值实际上是车票中乘客姓名和 TID 以外所有信息拼接得到的 32 位整型数据，其中起始站、终点站、车次号均为 6 位，剩余 14 位为座位号。事实上，这一规划直接决定了大多数系统可接受参数的范围，除了前文已经提到的车站数为位图宽度所限制。如果采用更长的哈希记录，如 64 位的长整型，则可以允许系统管理更多的车次、列车有更多的总座位数。值得一提的是，由于该数据的初始值为 -1 ，即各比特位全 1 ，所以系统不应允许一张合法票的哈希值与默认值相等的情况；而由于终点站和起点站一定不相同，所以这一点总能被满足。

而在退票时，则首先需要比对乘客姓名，确认一致后再比对姓名以外的信息，也即根据 TID 索引哈希值进行比对。这里采用了一个取巧的方式：在比对哈希记录时使用 CAS 方法，若哈希记录与所退票的哈希信息一致，则将哈希记录重置为缺省值 -1 。

根据要求，每张票的 TID 是唯一的，这也就意味着出票记录数组的每个元素最多只会被某个购票操作写一次；同时，更新乘客姓名是在哈希记录之后，而退票时则颠倒顺序先检查姓名再检查哈希记录，所以如果一个退票操作能对哈希记录成功执行 CAS ，则意味着其退票信息与记录信息完全一致，是合法的退票；另一方面，如果一个退票操作不成功，则意味着或是退票信息有误，或是 CAS 操作失败。而 CAS 一定严格在购票完成后（否则无法通过姓名检查），所以 CAS 失败等价于哈希值有误或哈希记录已被其他退票操作改回缺省值。此时，退票操作直接返回失败，就能保证至多只有一个操作能成功进入后续修改位图的环节，避免一张票被重复退票的情况。

综上所述，全局出票记录的设计可以保证非法的退票在常数步操作内被拒绝，合法的退票操作则在常数步操作后进入实际的修改工作中。也即是说，这一结构不仅能对退票的正确性提供保障，还不会对退票操作的性能产生明显影响。

## 三、理论性质分析

### 1、可线性化分析

如上所述，以购票操作的 CAS 操作和退票操作的 CAS 操作为可线性化点，本系统整体是可线性化的。值得注意的是，全局的出票记录为本系统的可线性化点提供了两个额外保障：

- 对于一张已售票，有且仅有一个合法的退票请求能运行到可线性化点，其它都会在常数步内被“驳回”。被“驳回”的退票请求或是非法的，或是对同一张票的并发退票请求中失败的。这些操作因为在常数步内被处理完成，也不会对系统的数据结构作出任何改变，所以不纳入“可线性化”的考量中。
- 对于同一张票，退票请求的可线性化点一定严格位于购票请求的可线性化点之后；更具体而言，一个能被成功执行的退票请求，其对出票记录的访问一定在对应的购票请求更新出票记录

之后。

因此，可以将每一个成功的 CAS 操作视为一个可线性化点，因为这意味着一个操作对系统产生了实质性的影响。如果存在并发的退票和购票请求，且串行场景下退票后的购票可以成功，那么在并行情况下如果购票失败，则意味着退票的 CAS 成功后于购票的查询步骤，这依然不改变可线性化点的正确性；而在其他情况下，线性化序列的建立都较为直观。

### 2、无锁性分析

首先，由于本实现中未使用任何的锁机制，在可线性化点采用的也是 CAS 原子操作，所以本实现是无锁的；由于 CAS 操作本身的性质，所以在一段时间内针对同一位图的 CAS 操作总有一个可以成功，所以本实现同时也是无饥饿的。

但是，本实现并不是无等待的。以购票操作为例，理论上有可能出现为购买某个座位某区间的票而进入 CAS 循环，但该座的其他区间却被反复购票、退票，导致一个购票请求永远无法成功的情况；同理，退票过程的 CAS 也有可能持续失败而不得不反复尝试。这些情况都意味着，一个操作可能需要无限步来完成，所以不是无等待的。

不过，根据后续引入购票的随机查询起点的优化，对同一座位的竞争事实上会得到缓解；此外，由于全局出票记录的存在，实际情况中退票的 CAS 尝试一定会比对应的购票时的 CAS 延迟相当一段时间，所以某一线程的 CAS 操作持续失败的情况极其罕有。

## 四、优化设计

### 1、随机化购票查询起点

考虑系统运行的起始阶段，如果简单地从位图数组的起点开始查找，那么由于大量购票和退票都会集中在数组的前半部分，对同一 Cache 行的写入可能会十分频繁，进而影响并发的效率。同时，即使一个购票线程成功进行了写入，其他线程相当于会同时浪费时间在重新尝试检查、再写入或向后查找新座位上。因此，为了避免读写过于集中的情况，本项目尝试在购票操作中，随机确定余票位图的查询起点，查询到数组末尾后返回数组开头继续，直至回到最初确定的随机起点。

需要明确的是，这一设计的出发点在于避免对同一 Cache 行的并发写，所以查询操作中并不需要引入类似的随机性；相反，查询操作使用相同的起点更有利于 Cache 的共享，因此也不该引入随机的查询起点。通过引入购票时的随机查询起点，可以令系统提高约 7% 的吞吐率。

### 2、余票位图的缓存行排布

正如前面所提到的，如果原子修改的对象落在同一缓存行上，可能会影响写入的效率，增加额外的同步开销。因此，引入 padding 方法是一种常用的思路，本质上是通过额外的内存开销，来减少原子对象共享 Cache 行的概率。在目前的实现中，由于位图的位宽为 32 比特，实验机器的 L1 Cache 行宽度为 64 字节，所以每个 Cache 行实际包含的位图个数为 16 。为了减少额外的处理开销，使用 padding 方法后每个 Cache 行中的位图个数减少为 `16 >> k` ，其中 k 为不超过 4 的自然数。

然而，经过实际测试，这一优化策略的实际效果并不理想，反而会降低系统的吞吐率。主要原因在于：

- 加入 padding 后，访问位图数组的每个元素前都不得不进行一次位移操作，积少成多后会产生不小的开销。
- 在随机确定查询起点的优化作用下，原子写的位置恰好位于同一 Cache 行的概率已经大大下降。并且即使不加入随机化查询起点的设计，由于测试负载中的购票操作占比较低，对同一 Cache 行进行写入的概率本来就比较低，所以不易得到明显的优化效果。
- 最重要的一点在于，在 padding 方法的实现中，虽然减少每一条 Cache 行中的位图数量可以减少写同一行的概率，但同时也降低了读同一行的概率；在查询或购票失败导致的遍历时，更是不得不访问成倍的内存，执行成倍的 Cache 行替换。在查询负载占比高、购票失败负载高的测试条件下，选择 padding 方法对系统性能的影响弊大于利。

## 3、TID 的线程级分配

在目前的实现中，购票时 TID 的生成逻辑是简单地维护一个全局 TID 号，出票时调用 `getAndIncrease` 方法获取绝对唯一的 TID 号（除非溢出）。但是，这一设计中可能存在的问题是，如果有多个线程同时试图获取 TID 号，则会产生竞争，全局 TID 可能成为性能瓶颈。并且，考虑退票记录的结构，其中的哈希记录数组也可能会因为连续的 TID 号（即数组下标）发生不同线程对同一 Cache 行并发写入的情况。因此，可以考虑采用线程级的 TID 分配方法来进行优化。

具体来说，在系统初始化时直接申请一个元素个数不少于最大接收线程数的数组，用于存储每个线程被分配到的局部 TID 。假设局部 TID 的分配粒度为 N ，当一个购票操作申请 TID 时，首先检查当前保存的局部 TID 是否能被 N 整除。若是，则原子地为全局 TID 增加 N ，并获取到相应的 N 个 TID 的使用权。随后，获得局部 TID 的值作为本次出票的 TID ，并为局部 TID 增加 1 ，以供后续购票操作使用。

审视整个过程，首先可以将对全局 TID 的操作次数减少到原本的 1 / N ，减少竞争的场景；其次由于局部的 TID 不涉及并发访问，所以能够使大部分分配操作、局部 TID 的修改更加轻量级。需要注意的是，记录局部 TID 的数组必须进行 padding 操作，否则会因为共享 Cache 行，反而拖慢性能。

## 4、余票售空时的缓存记录

测试结果显示，购票和查询操作的平均延迟十分接近，由此推测相当一部分的购票实际上是以失败告终（从而和查询一样访问了整个位图数组）。进一步进行测试，发现在实验给定的负载比例和参数配置条件下，有约 38% 的查询或购票操作，面对的是无票的场景；而如果去除退票的情况，以 7 : 3 的比例发起查询和购票请求，无票操作的概率更是会达到 58% 左右。考虑到“有票时必须出票”的需求，有票场景下的优化较难实现；但无票场景下的快速返回则很有希望为提高吞吐率做出贡献。

基于这些考量，本项目中尝试实现了“空票缓存”结构，即为每一车次维护一个最小的无票区间。其基本思想是，如果区间 A 不存在余票，且区间 A 为区间 B 的子集，则区间 B 也一定不存在余票。而这一实现的主要难点在于，如何保证空票缓存的正确性和高效性。

本项目通过“带时间戳的原子操作”来实现这一结构。与空票缓存相关的操作有 3 类。首先是查询操作 `readCache` ，适用于购票或查询操作，若缓存中记录的最小空票区间为输入的乘坐区间的子集，则购票操作直接被判定为失败，查询操作直接返回 0 值。其次是更新操作 `updateCache` ，在购票操作的查询开始前，会首先获取一个时间戳，如果遍历位图后发现没有余票，则将先前获取的时间戳和当前的购票区间更新到空票缓存中。最后是重置操作 `flushCache` ，在退票操作的最后，如果发现退票区间与最小空票区间有所重合，则需要清空缓存内容，因为这意味着原先保存的区间可能不再无票。

空票缓存正确性的关键在于更新和重置的逻辑。在此，我们采用相对保守的维护策略：重置操作无视时间戳，更新操作则只有时间戳较新时才能成功。这是因为，理论上说存在“在获取到时间戳和重置缓存的两个时间点之间，一个购票操作完整进行了从获取时间戳到查询失败的过程并更新缓存”的情况，此时出于谨慎的考量，更新的缓存也应被重置刷新。

此外，为了尽可能避免空票缓存的维护成为新的瓶颈，维护的逻辑应该被尽可能简化，维护频率也应适当下降。测试发现，如果在查询中也加入维护，会严重影响系统的吞吐量，可以认为是过于频繁的空票缓存访问导致了性能的下降；但即使只在购票失败时更新缓存，测试的效果也并不理想。分析发现，如果测试过程中不包含退票，即以 7 : 3 的比例发起查询和购票请求，则缓存的命中率可以达到 39% 左右，也就意味着有约 22% 的查询和购票操作能被高效地直接返回；然而，由于退票的刷新具有很严厉的强制性，在项目要求的测试负载下，空票缓存仅能达到 7% 左右的命中率，也就是仅仅增益约 2% 的查询和购票操作，收益极低的同时又不得不付出高代价来维护，系统的整体吞吐也就因此下降了。

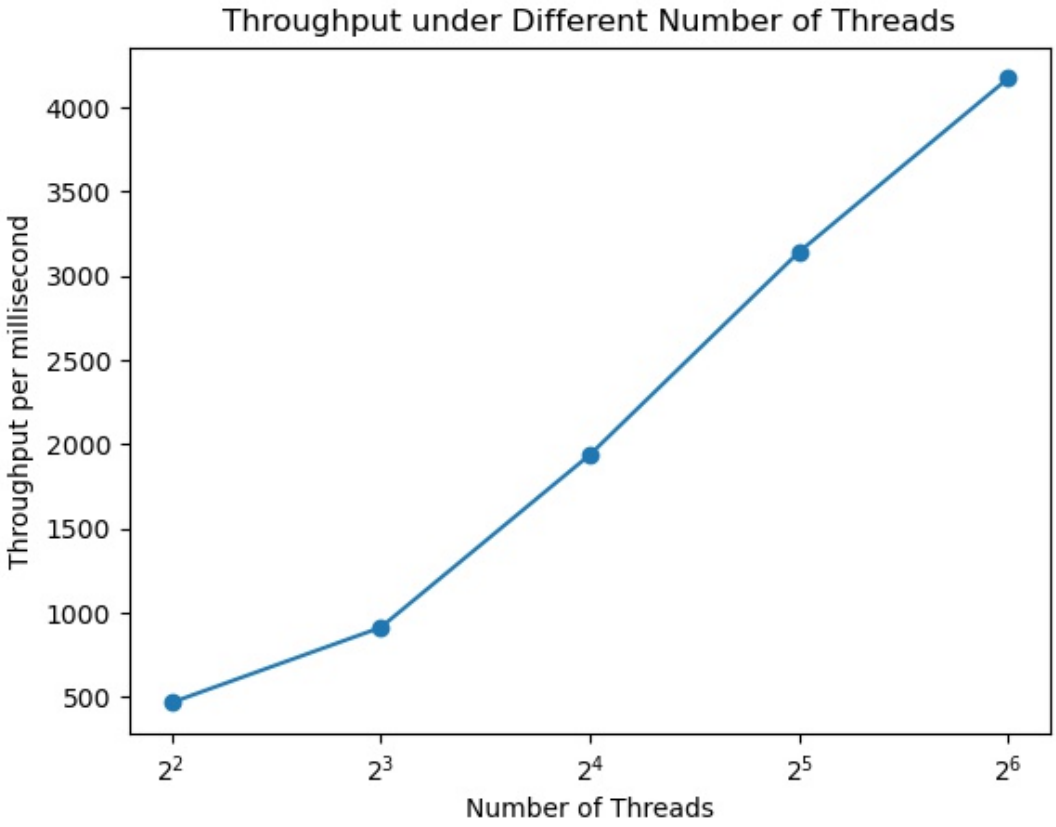
综上所述，空票缓存的设计在负载的退票比例极小，甚至几乎没有退票负载时，可以得到一定的优化效果，但在本项目中难以发挥其效用。另一方面，由于维护开销较大，应注意尽可能减少缓存的更新频率。

# 五、性能评测

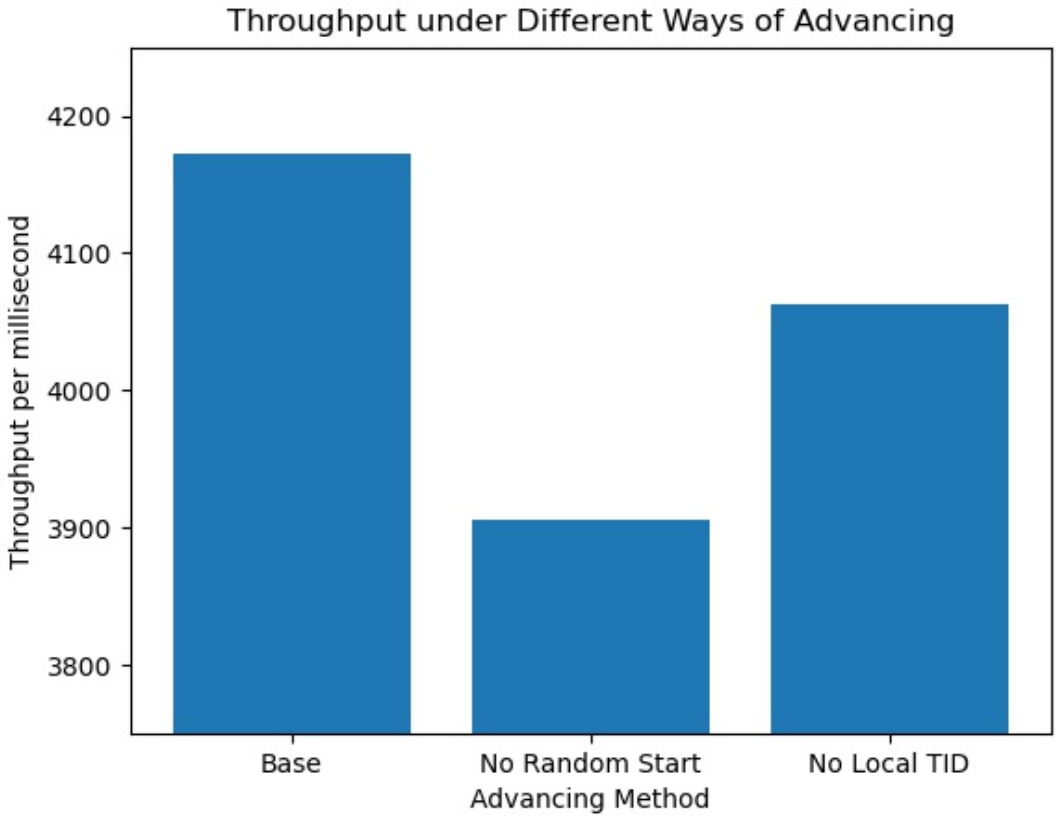
本报告的评测部分使用的实验机器为课程提供的远程服务器，配置参数不赘述。各版本的系统均能够通过串行测试，并行吞吐量测试程序借用了课程群中陈宇哲同学编写的脚本，并在必要部分增加了需要的统计信息。测试结果均为运行 10 次测试的平均值。对比测试使用的负载为：

- 4 或 8 或 16 或 32 或 64 线程
- 每线程执行 10 万次操作，其中查询、购票、退票的比例为 7 : 2 : 1

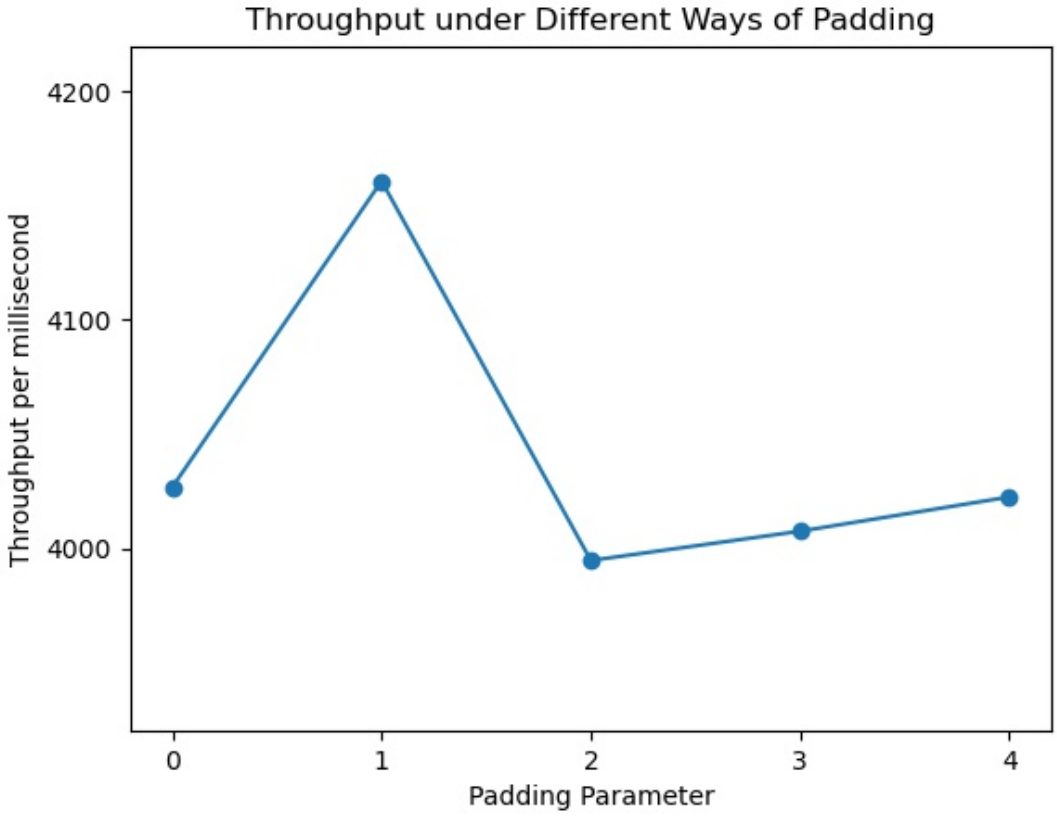
首先，对不同线程数量的负载下本系统的吞吐量进行了测试。可以发现，在这一范围内，随着负载成倍增加，系统的吞吐量基本保持线性上升，且线程数越多，吞吐量越高。




为了验证在系统的最终版本中所引入的“购票时随机化查询起点”和“线程级局部分配 TID ”这两大优化的作用，以最终系统为标准，分别测试了 64 线程负载下取消一种优化时的系统性能，结果如下图所示。显然，这两项优化都能在一定程度上提高系统的吞吐量。




最后，假设 Padding 方法有效，为了获取最优的 Padding 参数，还对各种不同程度的 Padding 后的系统进行了性能测试，结果如下图所示。可以看到，当 Padding 参数为 1，即令每个位图额外占据一倍的空间时，可以得到最好的性能。但值得一提的是，即使是这一峰值点，其吞吐量也略小于不加入 Padding 逻辑时的吞吐量，且运行结果的方差会有所增大，推测是 Padding 逻辑导致的频繁位运算或多或少影响了系统效率，因此在最终版本内没有加入这一优化。



# 六、实验总结

本项目从基于位图的数据结构出发，设计了并发的列车售票系统，并通过了正确性测试。在基础的功能上，提出了四种可能的优化，并根据实际测试结果，将其中的两种优化最终加入了系统中，使系统的性能有了进一步的提升。

限于时间和其他安排，本系统未能进一步完善，例如查询过程可以通过进一步分配多线程任务来提速，对于查询占比极高的本项目负载来说使非常有吸引力的想法。此前也思考过直接在后台检索余票并提供余票队列，来直接响应请求的思路，但基于这一思路的系统的实现和维护难度过高，故没有能实现。