基于内存池的分配器优化算法

摘要：。堆内存随机分配容易形成内存碎片化，而碎片化对内存分配器性能有重要的影响。但是对于已知内存大小类型的小内存分配应用场景，可设计内存池化的分配器降低碎片化影响并提升分配性能。该分配器采用固定池和动态池相结合的方案,实现基础场景无锁设计,而极端分配场景使用读写锁动态拓展。针对内存分配块的查找性能优化，对已知类型采用哈希表快速查找，而随机类型通过优化折半和插值查找算法，使用混合区间查找固定内存块。最后，性能测试表明，对于已知固定类型分配请求，其性能整体上优于未池化的直接随机分配方案。

关键词：内存池； 内存分配器；哈希表查找；区间查找

English

Abstract：.

Key words：;;

1. 引言

堆内存分配器是应用程序内存管理重要基础组件，降低了应用编程对内存管理的难度。堆内存分配器主要关注点是：

- 额外的空间损耗尽量少
- 分配性能
- 内存碎片
- 通用性，兼容性，可移植性，易调试

目前主流的内存分配器有 ptmalloc（glibc 默认分配算法），Google 的 tcmalloc，facebook 的 jemalloc 等。ptmalloc 是由最早被收纳到 C 标准库里，现在是 ptmalloc2，其性能低于后来者 tcmalloc，jemalloc 分配器，好在由于是 C 标准库支持，其通用性和可移植性比较有优势。tcmalloc，jemalloc 性能虽然比较好，但其代价是相对组件臃肿，额外开销（如碎片化整理合并，缓存管理等）较多，对于小型轻量化应用不够友好。

显然，内存分配器主要聚焦点是性能和利用率，一个是运行速度，一个是内存使用效率。

当前由于内存随机分配，大小无法完全确定，就容易把大块的内存分割成碎片化，从而影响继续分配更大的内存，造成空间利用率下降，因此分配器往往需要自动整合碎片内存空间，自动整合碎片需要操作锁，并且需要额外的内存管理线程负责自动整合，对分配性能有些许影响。大部分算法实现上基本采用独立的线程级缓存来分配小内存，独立的服务线程独立整合碎片化内存。优化基本上是尽可能减小操作锁和减小锁粒度来实现。

内存分配器，设计主要点是小内存碎片化管理和尽可能无锁设计。而基于内存池的内存分配器有两个优势：不会碎片化和支持无锁模式。

第一，不会产生碎片化。基于内存池的分配器，其设计预先申请一系列固定长度的内存块，，不需要进行碎片合并等整理操作，申请的内存都是按照固定长度分配，其性能相对比较高。

第二，无锁设计。内存池一般实现采用无锁队列设计（其实有锁，如 spdk 采用 CAS 指令，锁粒度非常小，性能消耗非常小）内存池分配单元支持多线程分配，且性能高。

但是，基于内存池的分配器的缺点也是明显的，如下几个点:

- 适用于若干固定长度的内存场无法满足随机分配的要求；
- 内存池扩容性能；
- 内存池查找性能；
- 内存独占，无法被系统其它组件共享；

下面针对优缺点研究其设计要点。

2. 算法和设计

针对以上优缺点，充分利用其优势的同时，需要解决其不足点，因此基于内存池的分配器设计要点可以归纳为：内存块管理；内存池动态拓展；支持任意长度分配；查找性能优化；

2.1 内存池动态拓展

由于一般无锁内存池的内存容量需要一次申请，一次释放，无法直接动态扩容。只能通过增加内存池链表来实现扩容。

维护内存池链表需要额外增加锁保护，否则无法支持多线程。

由于堆内存使用存在波峰和波谷效应，波峰和波谷运行时间占比都比较低，为提高分配性能，这里采用划分为固定组和动态组的设计方案，动态组用于满足波峰内存使用的场景，固定组用于满足基础平均值。正常申请内存，优先固定组，若固定组申请失败，则加锁访问动态组。

固定组内存池，利用已知业务内存容量需求，提前规划一定容量。采用全程无锁设计，其生命周期跟随进程初始化和消亡，需要提前初始化后使用，禁止动态修改固定组的内存池。由于固定组的生命周期与进程一致，期间多线程对固定组的都

是读访问，不存在竞争修改问题，所以可以全程无锁，性能较高。

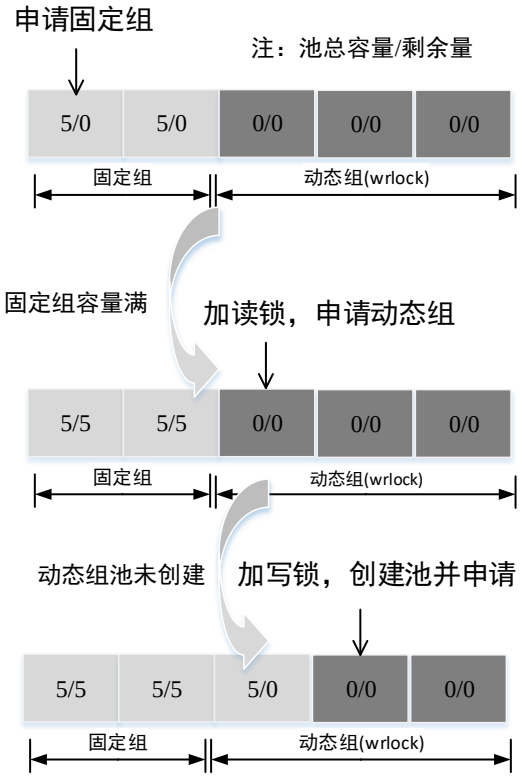
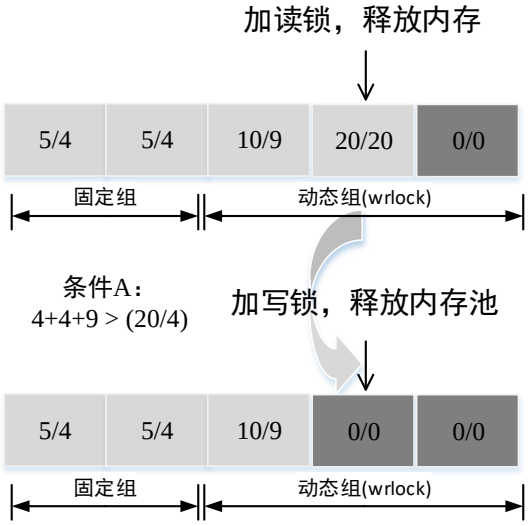


图 2-1 内存块申请示意图

内存池动态组，采用读写锁设计方案。因为申请内存池扩容操作频率远小于访问扩容内存池的频率（申请新内存池容量增长），一次申请，N 次使用。优先加读锁，申请分配内存，若分配失败，则加写锁，执行扩容，如图 2-1。



2-2 内存块释放示意图

动态组内存池释放，遵从本次释放时若该内存池已经没有人使用，且固定组和其它动态组池空闲容量大于该内存池的四分之一容量（该点表示为**条件 A**，其说明内存块余量足够，不需要继续保留当前内存池），则可以释放当前内存池，以实现缩小内存，其流程如图 2-2 所示。

这里可能存在的问题是，如果短时间内存申请量突增，导致申请的动态池满足不了**条件 A**，则该内存池将一直保留无法释放，直到再次申请量上来时才能触发释放动作，因此可能造成内存池浪费（更优的方案是起一个自动定时回收的线程，但分配器变复杂），但至多只浪费一个内存池空间，可根据业务场景评估影响。

2.2 支持任意长度分配

小内存分配，因为内存池的内存块是定长，无法支持任意长度的小内存分配。这里解决方案是：对于任意长度的小内存申请，考虑其碎片化对系统性能的影响远大于其分配锁消耗的性能影响，因此采用区间定长分配的方案。即任意长度若在固定长度区间内，则选择区间上限对应的内存池予以分配，如图 2-3。优点是避免碎片化内存申请，缺点是造成一定量空间浪费和额外的查找性能消耗。针对空间浪费的缺点，考虑小内存实际内存占用不大，可以忽略。对于查找性能消耗则参考 2.3 查找性能优化。

针对大内存（比如大于 1k）的任意长度，则直接使用系统内存分配，不从内存池里获取，从而满足大内存的任意长度分配。

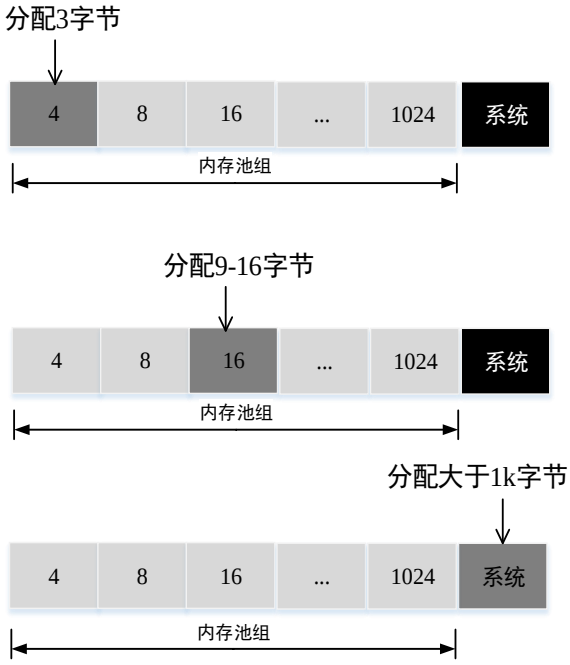


图 2-3 任意长度区间分配法

2.3 查找性能优化

基于内存池的分配器，其必然存在有限个（设为 N）固定长度的内存池组。其每次申请时的池查找动作，性能将直接影响分配器性能。

这里需要根据 N 大小，和固定长度的值分布

特点进行查找优化。因为为了尽可能减小N的值，以及业务申请内存的特点，一般内存池的固定长度的值分布是非均匀的，如（8,16,32,128,512,1024）。

非均匀分布的 key 值查找，最优的算法是哈希表查找(O(1))，其次是折半查找（O(log2n)），再次是插值查找（O（logn(logn)））。

考虑实际的业务场景 N 值一般不会太大（预先评估固定长度小内存，一般不会超过 1000 个元素，这里上限设置为 254 个），基础的哈希查找不会退化为红黑树查找（即折半查找 O（logn）），即其性能为 O(1)。

但是哈希查找无法实现区间查找（用于匹配任意长度分配），需要对折半查找进行优化实现区间查找，具体设计细节见 2.4 节，而内存器分配器的查找流程如下图 2-4：

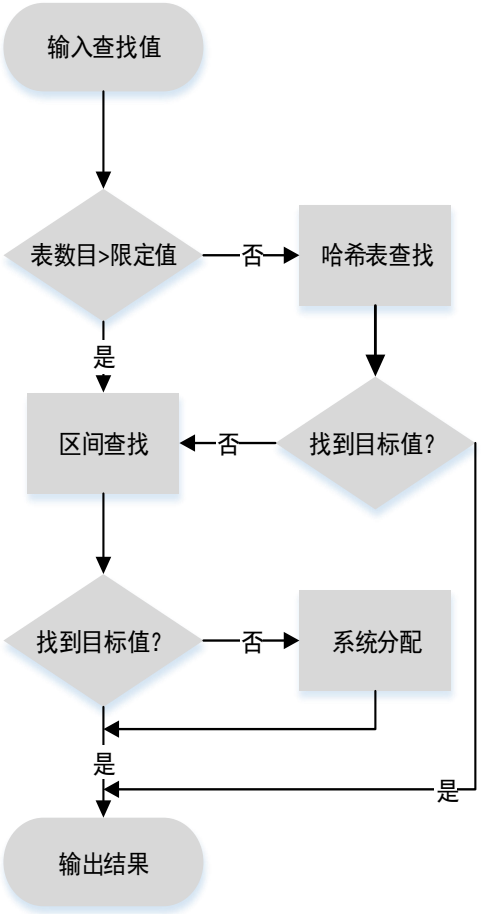


图 2-4 内存池查找流程图

2.4 折半插值混合区间查找

折半查找的特点是性能高（仅次于哈希），查找效率稳定。但是其缺点也是明显，对于 key 值跨度较大，查找 key 值处在两端边缘的场景下，查找效率下降。如查找范围 10-50 之间，若查找的 key 值大多数落在 40-50 范围内时，其折半查找性能将为 O(logn)效率，而插值查找可能可以达到 O（1）效率，如图 2-5，key 值均匀分布，每个值间隔为 10，查找值 49 时，其效率对比，显然折半查找需要 3 次，而插值查找只需要一次。

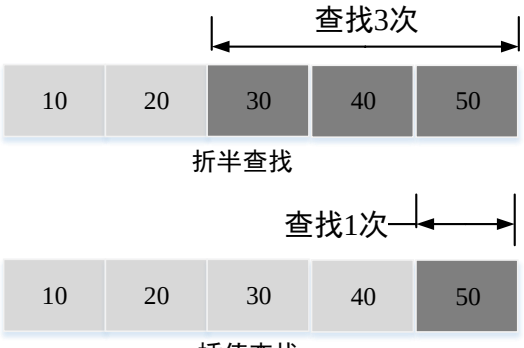


图 2-5 折半和插值查找效率对比

插值查找适用于 key 值均分分布索引表，其查找性能达到 O（1），但若不均匀分布的值，如图 2-6，当查找值为 9 时，由于该查找表的值离散比较严重，使用插值查找的化，其效率可能会退化成 O(n)，所以插值查找效率无法像折半查找一样稳定在 O(logn)。

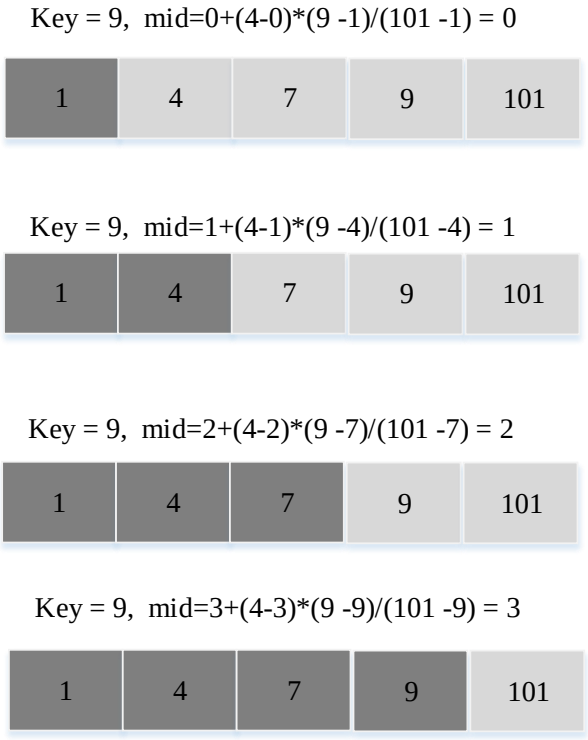


图 2-6 插值查找效率退化成 O（n）示意图

考虑基于内存池的分配器的内存申请长度特点（非均匀且 key 离散化），首先对任意长度查找要满足区间查找功能，即查找结果 y 必须是大于等于 key 值，且需要满足最逼近 key，即满足如下：

$$y = \min(\{Val(x) - key\}),\ 且 key \le Val(x)$$

结合折半和插值查找的特点，对折半和插值结合优化，可根据 key 动态地选择中值的收敛方向，既避免了折半查找边缘值存在的效率下降的问题，也避免了单纯插值查找退化成线性查找效率的不

足点。这里介绍混合区间查找的基本原理，混合区间查找算法如下：

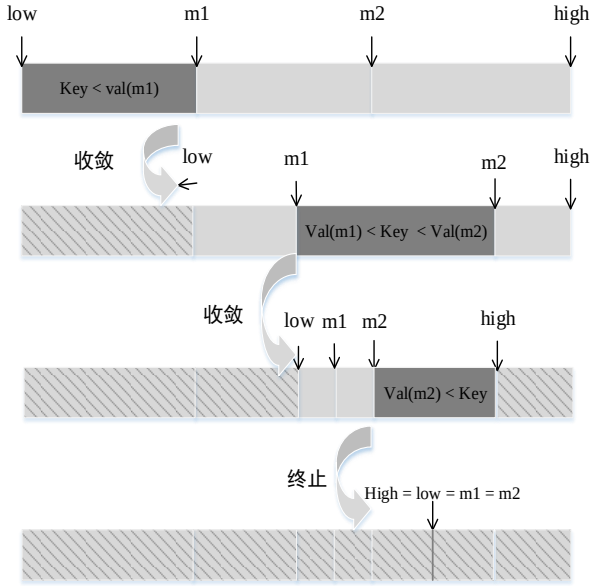


图 2-7 区间查找过程示意图

设一个有序表的最大值 Val（high）索引为 high，最小值 Val（low）的索引为 low，待查找值为 Val(x)，其中 x 为每轮迭代查找的目标索引值，同时设输入查找值为 key，目标是要找到索引值 x 对应的 Val（x），使其最逼近 key 值，即用函数关系表示如下：

$$key \le Val(x)$$

同时设插值和折半的中间值分别 Val (m1), Val (m2)，其中 m1，m2 是索引值，则有如下表达式：

$$m1 = low + \frac{(high - low) * (key - \mathrm{Val}(low))}{\mathrm{Val}(high) - \mathrm{Val}(low)}$$

$$m2 = \frac{high + low}{2}$$

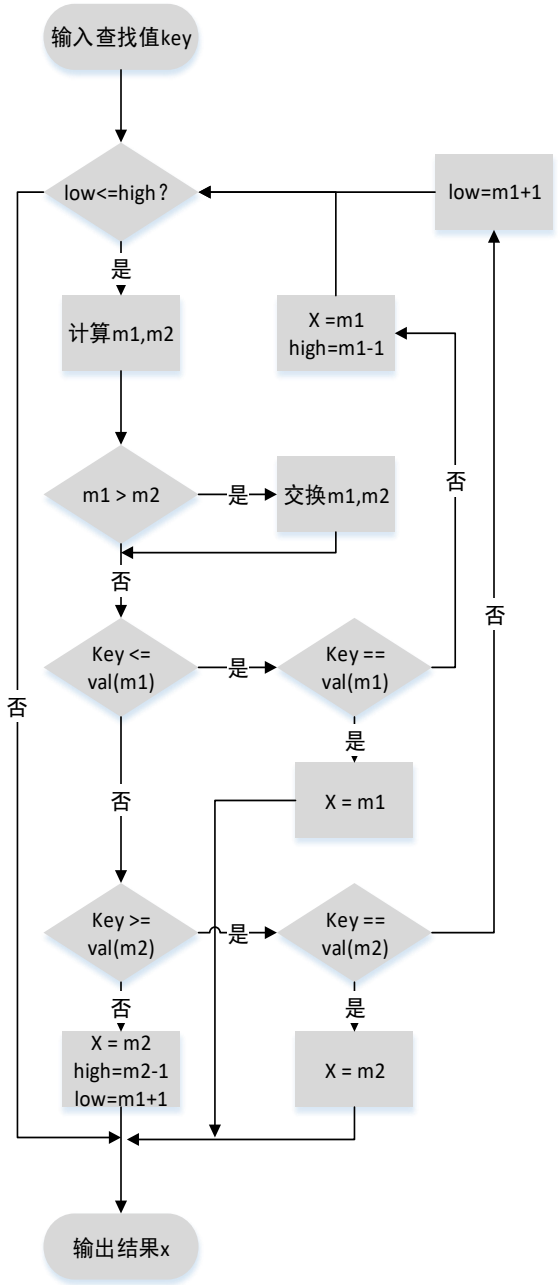
将 high、low、m1、m2 按照索引值大小划分区间为三个部分（low，m1）、（m1，m2）、（m2，high），如图 2-7 所示，存在如下关系：

$$\mathrm{Val}(low) \le \mathrm{Val}(m1) \le \mathrm{Val}(m2) \le \mathrm{Val}(high)$$

$$\mathrm{Val}(low) < key < \mathrm{Val}(high)$$

- 若 m1 > m2 则交换；
- 若 key < Val(m1)，则 x = m1, high = m1 - 1;
- 若 key > Val(m2)，则 x 保持不变，low = m2 + 1;
- 若 key = Val(m1) 或者 key = Val(m1)，则直接结束查找，x = m1 或者 x = m2;
- 若 m1 < key < m2，则 x = m2, high = m2 - 1，low = m1+1;

以上每轮迭代，会不断地让 Val(x)逼近或者等于目标值，直到上述关系式无法满足则终止，最后返回的 x 就是目标值对应的索引，算法基本流程图实现如下：



代码实现如下：

```
def range(val, key, high, low, x):
    m1 = low + ((high - low)*(key - val[low]))//(val[high] - val[low])
    m2 = (high + low)//2
    if m1 > m2:
        tmp = m1
        m1 = m2
        m2 = tmp
    if key < val[m1]:
        x = m1
        high = m1 - 1
    elif key == val[m1]:
        return True, high, low, m1
    elif key > val[m2]:
```

```
            low = m2 + 1
        elif key == val[m2]:
            return True, high, low, m2
        else:
            x = m2
            high = m2 -1
            low = m1 + 1
        return False, high, low, x
```

3 测试与验证

3.1 区间查找性能

区间查找性能测试，主要取折半查找、插值查找和混合区间查找进行对比。折半和插值通过返回区间上限值来实现区间查找功能，分别测试均匀分布和分均匀分布的性能对比。每次查询测试定为 50000 次，测试数据规模从 10 – 15210，查找值为遍历范围内所有索引点。

根据标 3-1 和 3-2 得出趋势图 3-1 可知，随着索引数量规模增大，混合查找的性能无论在均匀和非均匀分布的查找耗时接近或者是最优值。说明混合区间查找对离散化的查询性能比较有优势。考虑边缘查找（即查找值落在表的下上限附近）性能影响，使用混合区间查找整体效率优于单独使用折半或者插值：

表 3-1 均匀分布值性能测试对比

索引数	折半(us)	插值(us)	混合(us)
10	2568	2999	3716
90	3680	3021	3428
250	4235	2653	3211
490	4558	2447	3002
810	4738	2346	2815
1210	4861	2264	2710
1690	4894	2116	2574
2250	4854	2099	2318
2890	4766	1903	2313
3610	4770	1896	2224
4410	5119	1912	2431
5290	5283	1891	2282
6250	5786	1897	2416
7290	5329	1689	2096
8410	5344	1635	1973
9610	6206	1987	2253
10890	5964	1718	2066
12250	6776	1933	2323
13690	6140	1676	2112
15210	6854	1855	2365

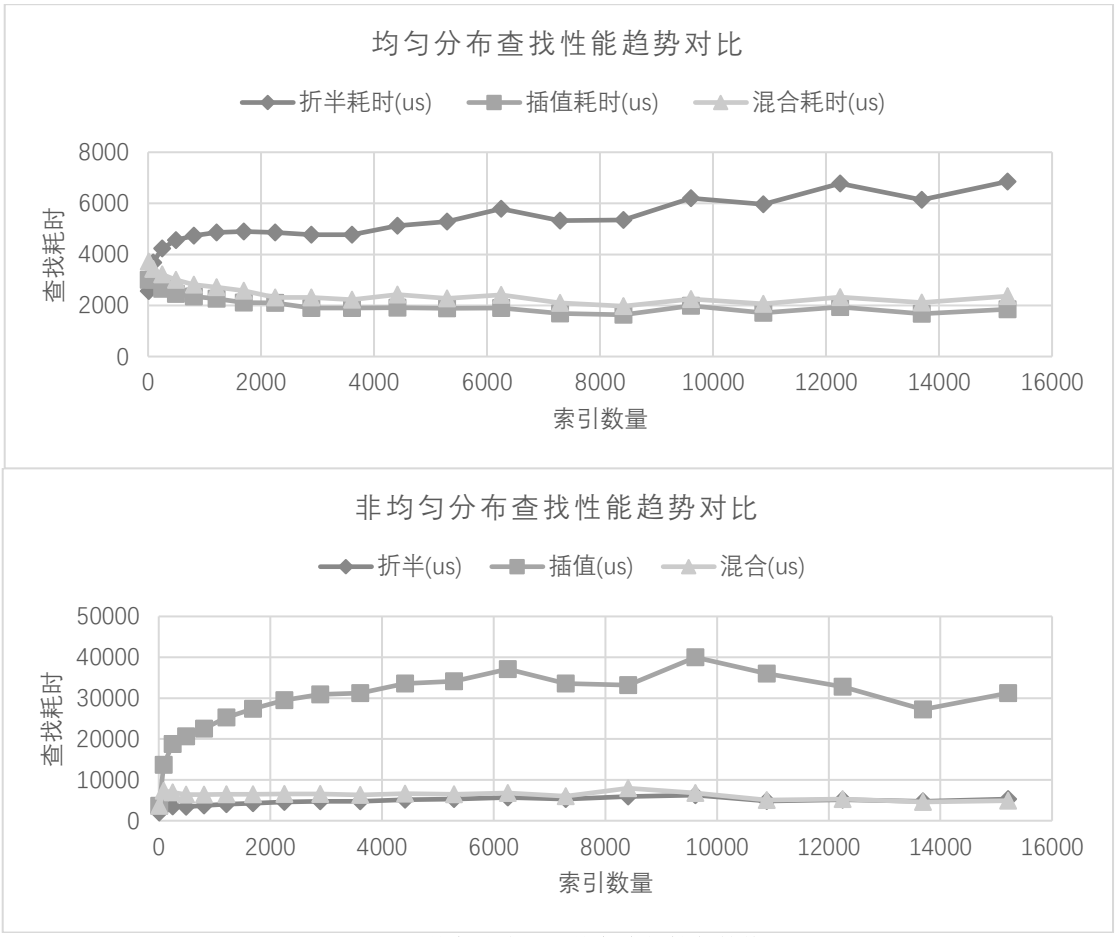


图 3-1 均匀分布和非均匀查找性能趋势

表 3-2 非均匀分布值性能测试对比

索引数	折半(us)	插值(us)	混合(us)
10	2021	3629	3643
90	3251	13673	7351
250	3461	18744	6965
490	3401	20651	6359
810	3744	22515	6423
1210	4071	25261	6488
1690	4261	27429	6447
2250	4568	29498	6567
2890	4737	30942	6555
3610	4751	31268	6285
4410	5103	33564	6637
5290	5280	34103	6442
6250	5713	37080	6779
7290	5315	33559	5999
8410	5933	33159	7953
9610	6264	39986	6771
10890	4806	35991	5050
12250	5108	32771	5315
13690	4787	27272	4583
15210	5327	31265	4924

3.2 分配器性能对比测试

考虑 glibc 的 ptmalloc 是最通用的内存分配器，而且测试用例所用的内存池也是基于 glibc 的 ptmalloc 实现池化，因此 mpmalloc 可以与 ptmalloc 进行性能测试对比。考虑内存分配的多线程场景，这里使用 pthread 创建 5 个独立线程，每个线程都执行对应长度的内存申请释放请求，每个线程各自申请 5000 次。对比的测试项主要包括：

- 小内存固定长度，即长度等于各个内存池的 size；
- 小内存随机长度非字节对齐；
- 小内存随机长度字节对齐；
- 大内存随机长度字节对齐；

同时为了对比固定组个动态申请对性能的影响，测试内容划分两大类，一个是 mpmalloc 固定组容量小于 5000，和动态组大于 5000。根据用例设计的测试结果，可得数据如下：

基于内存池的分配器（别名 mpmlloc），在固定组容量范围内申请时，其小内存申请和释放性能整体优于 ptmalloc 分配器。而大内存申请性能接近，这与 mpmalloc 的大内存申请是直接使用 ptmalloc 的事实相符。

动态组测试结果表明，mpmalloc 的动态组小内存固定长度分配性能会略低于 ptmalloc，说明读写锁对多线程存在一定影响。随机长度分配，mpmalloc 略高于 ptmalloc，说明区间查找性能消耗比内存碎片管理低。

表 3-3 固定组内存分配器性能测试

分配器	ptmalloc	mpmalloc
固定长度	16747us	8532us
随机长度(非对齐)	24208us	13148us
随机长度(对齐)	20947us	17523us
大内存(大于 1k)	64478us	64131us

表 3-4 动态组内存分配器性能测试

分配器	ptmalloc	mpmalloc
固定长度	15947us	16615us
随机长度(非对齐)	23667us	20191us
随机长度(对齐)	21931us	20086us
大内存(大于 1k)	67666us	64973us

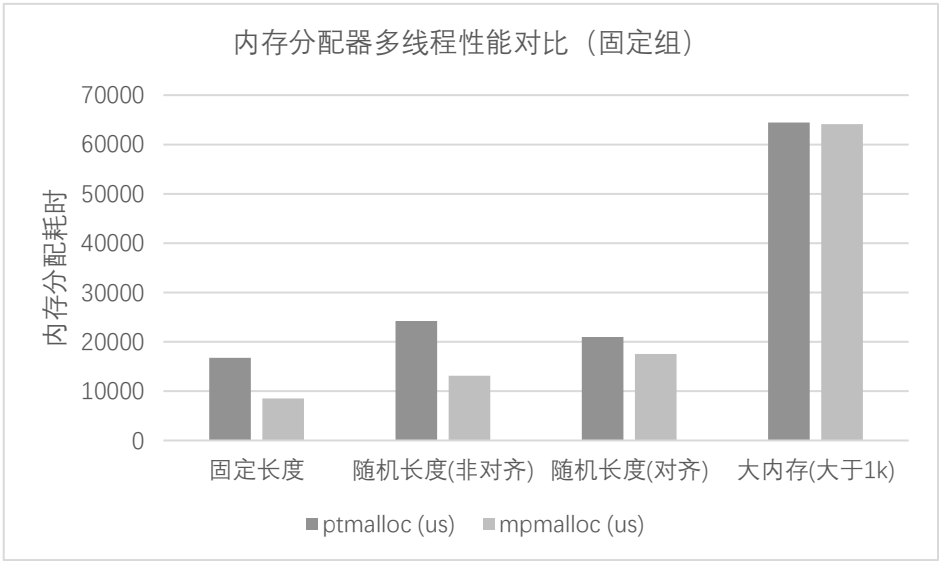


图 3-2 内存分配器固定组性能对比

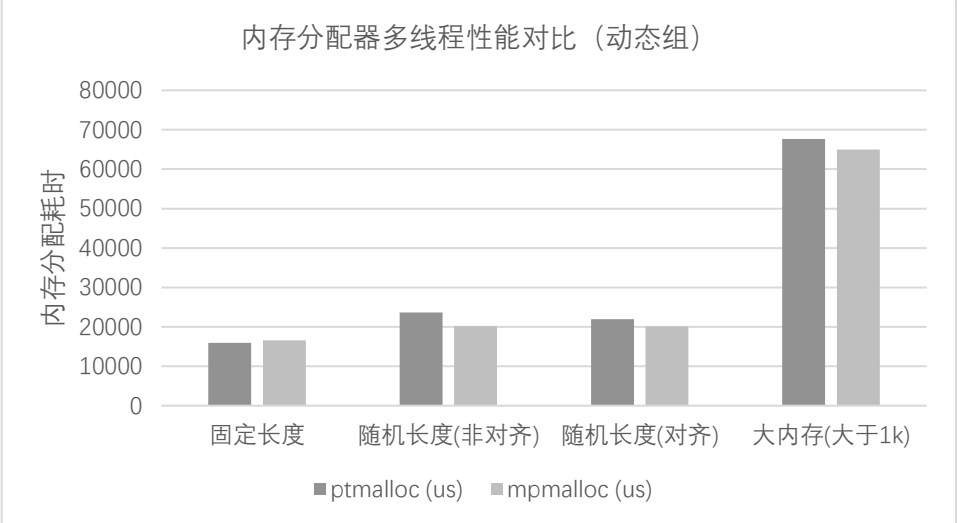


图 3-3 内存分配器固定组性能对比

3.3 p_malloc 分配器性能对比

考虑项目中使用 p_malloc 实现通用化内存分配，以及内存池默认采用 spdk 的无锁队列内存池。mpmalloc 的内存池适配使用 spdk 的内存池，同时 mpmalooc 的大内存申请适配为使用 p_malloc 实现。同样分配次数为 5000 次，测试线程使用 p_thread，线程数 1 个，按照固定长度，随机长度、大内存三项性能进行测试对比，其结果如下：

根据表 4-1 和图 4-1 可知，在同样的运行环境下，p_malloc 在小内存或者大内存分配的性能整体都比较低。Mpmalloc 性能优于 p_malloc，由于 mpmalloc 大内存使用 p_malloc，其性能相近。

表 4-1 pmalloc 内存分配器性能测试对比

分配器	ptmalloc	mpmalloc	p_malloc
固定长度	984	1566	2914
随机长度	2692	2385	4666
大内存	728	3180	2872

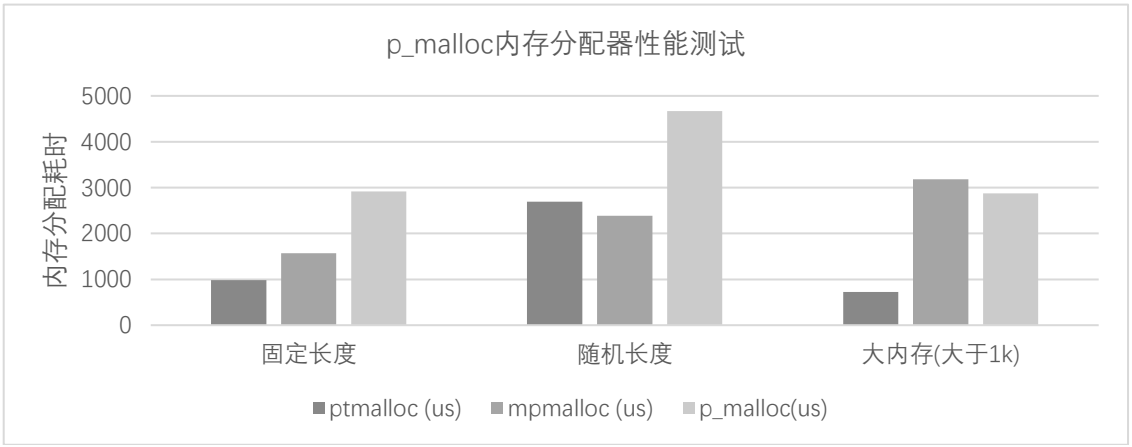


图 4-1 p_malloc 内存分配器性能对比

4 结论

综合上述，基于内存池的内存分配器适用于某些基本已确定所使用的内存块大小场景，其性能整体优于通用分配器。并且能同时减少碎片化管理的消耗。

文中所用到的混合区间查找算法，充分利用折半和插值各自优势，能明显提高离散化索引表的查找效率，测试数据表明，在索引数据规模越大以及离散程度越大的场景下，其性能将一直优于折半和插值查找性能。文中内存分配器结合哈希表查找，整体的内存分配查找效率可达 O(1)，提升 mpmlloc 分配器的整体性能。

参考文献：

[1] http://jemalloc.net/.
[2] https://github.com/google/tcmalloc
[3] https://github.com/microsoft/mimalloc