作者 | daydreamer

前篇《探秘C++内存管理（理论篇）》主要介绍了Linux C++程序内存管理的理论基础，本文作为系列文章《探秘C++内存管理》的第二篇，将会探讨经典内存管理器ptmalloc如何管理C++程序的内存。借助剖析ptmalloc解决问题的着重点和设计实现成本的权衡，更具体的呈现c++内存管理面临的问题和工程落地中的巧思。

# 一、概述

ptmalloc是开源GNU C Library(glibc)默认的内存管理器，当前大部分Linux服务端程序使用的是ptmalloc提供的malloc/free系列函数，而它在性能上远差于Meta的jemalloc和Google的tcmalloc。服务端程序调用ptmalloc提供的malloc/free函数申请和释放内存，ptmalloc提供对内存的集中管理，以尽可能达到：

- 用户申请和释放内存更加高效，避免多线程申请内存并发和加锁
- 寻求与操作系统交互过程中内存占用和malloc/free性能消耗的平衡点，降低内存碎片化，不频繁调用系统调用函数

简单概括ptmalloc的内存管理策略：

- 预先向操作系统申请并持有一块内存供用户malloc，同时管理已使用和空闲的内存
- 用户执行free，会将回收的内存管理起来，并执行管理策略决定是否交还给操作系统

接下来，将从ptmalloc数据结构、内存分配及优缺点介绍最经典的c++内存管理器的实现和使用（以32位机为例）。

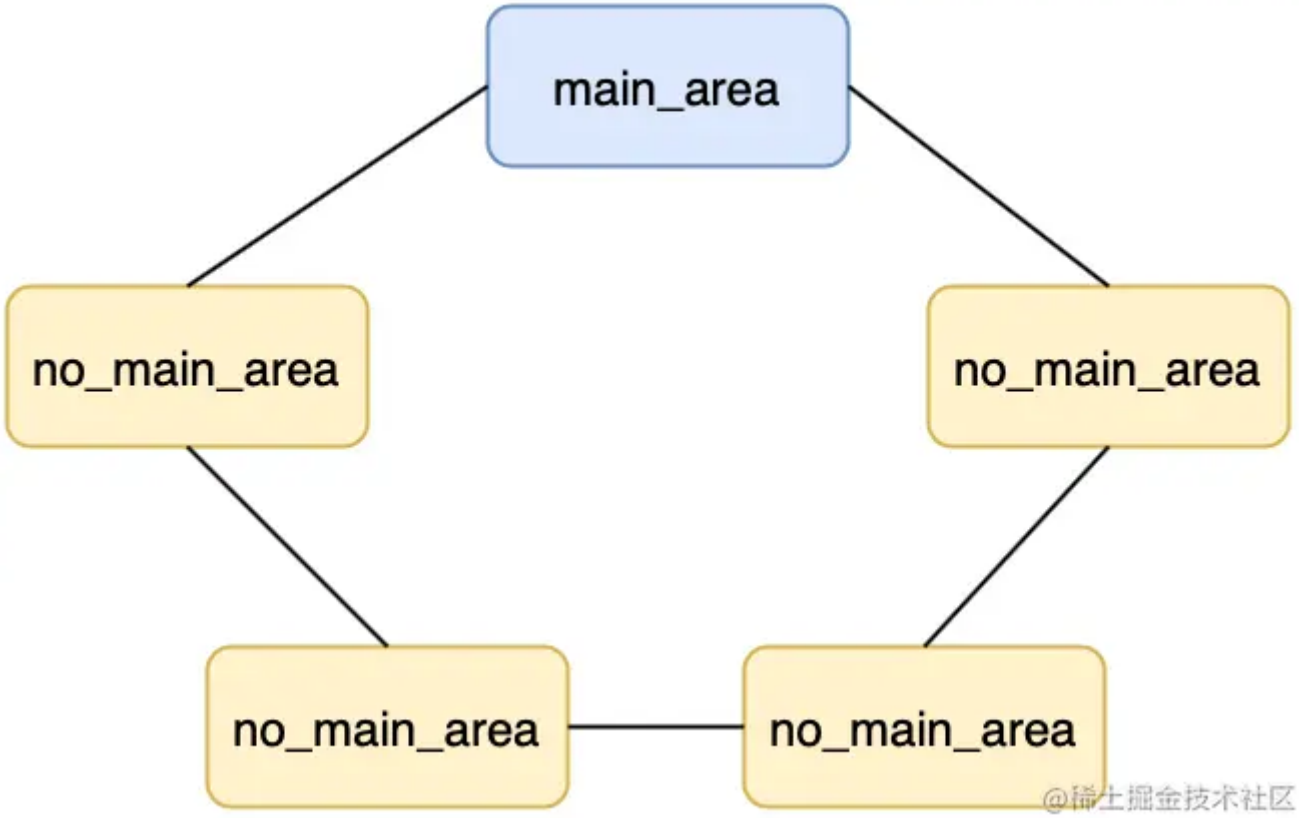
# 二、内存管理

## 2.1 数据结构

为了解决多线程锁争夺问题，将内存分配区分为主分配区(main_area)和非主分配区(no_main_area)。同时，为了便于管理内存，对预申请的内存采用边界标记法划分成很多块(chunk)；ptmalloc内存分配器中，malloc_chunk是基本组织单元，用于管理不同类型的chunk，功能和大小相近的chunk串联成链表，被称为一个bin。

### main_arena与non_main_arena

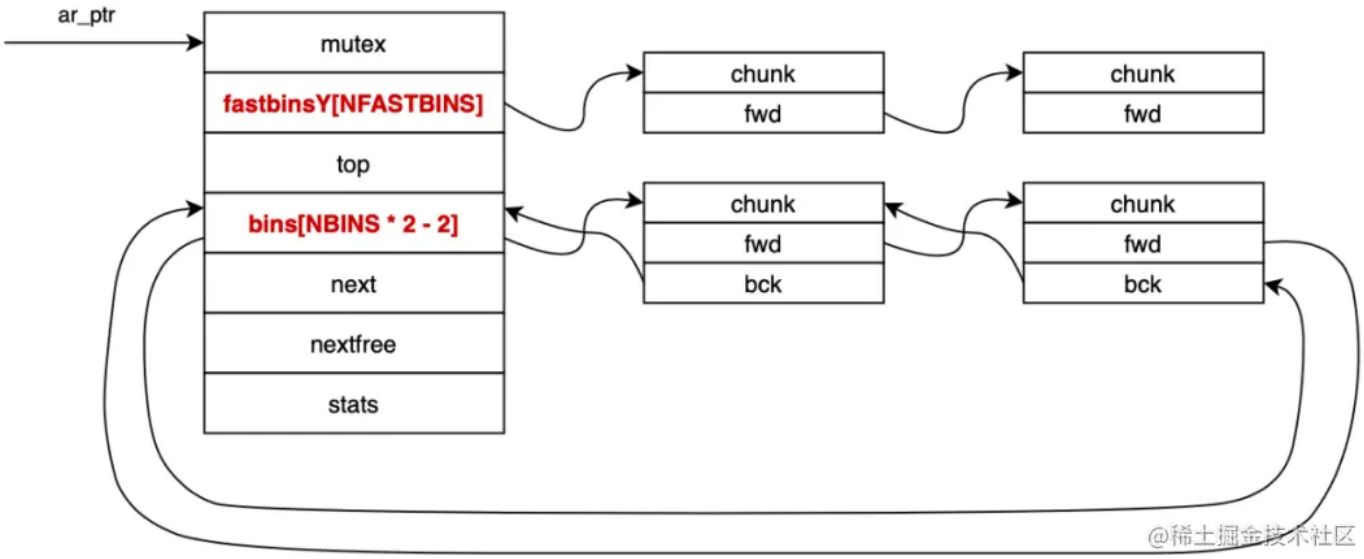
主分配区和非主分配区形成一个环形链表进行管理， 每一个分配区利用互斥锁实现线程对该分配区的访问互斥。每个进程只有一个主分配区，但允许有多个非主分配区，且非主分配区的数量只增加不减少。主分配区可以访问进程的heap区域和mmap映射区域，即主分配区可以使用sbrk()和mmap()分配内存；非主分配区只能使用mmap()分配内存。




对于不同arena的管理策略大致如下：

- **分配内存**
- 查看该线程的私有变量中是否已经存在一个分配区并对其进行加锁操作，如果加锁成功，则使用该分配区分配内存；如果未找到该分区或加锁失败，遍历环形链表中获取一个未加锁的分配区
- 如果整个环形链表中没有未加锁的分配区，开辟一个新的分配区，将其加入循环链表并加锁，使用该分配区满足当前线程的内存分配
- **释放内存**
- 先获取待释放内存块所在的分配区的锁，如果有其他线程正在使用该分配区，等待其他线程释放该分配区互斥锁后，再释放内存

主分配区和非主分配区的结构如下：

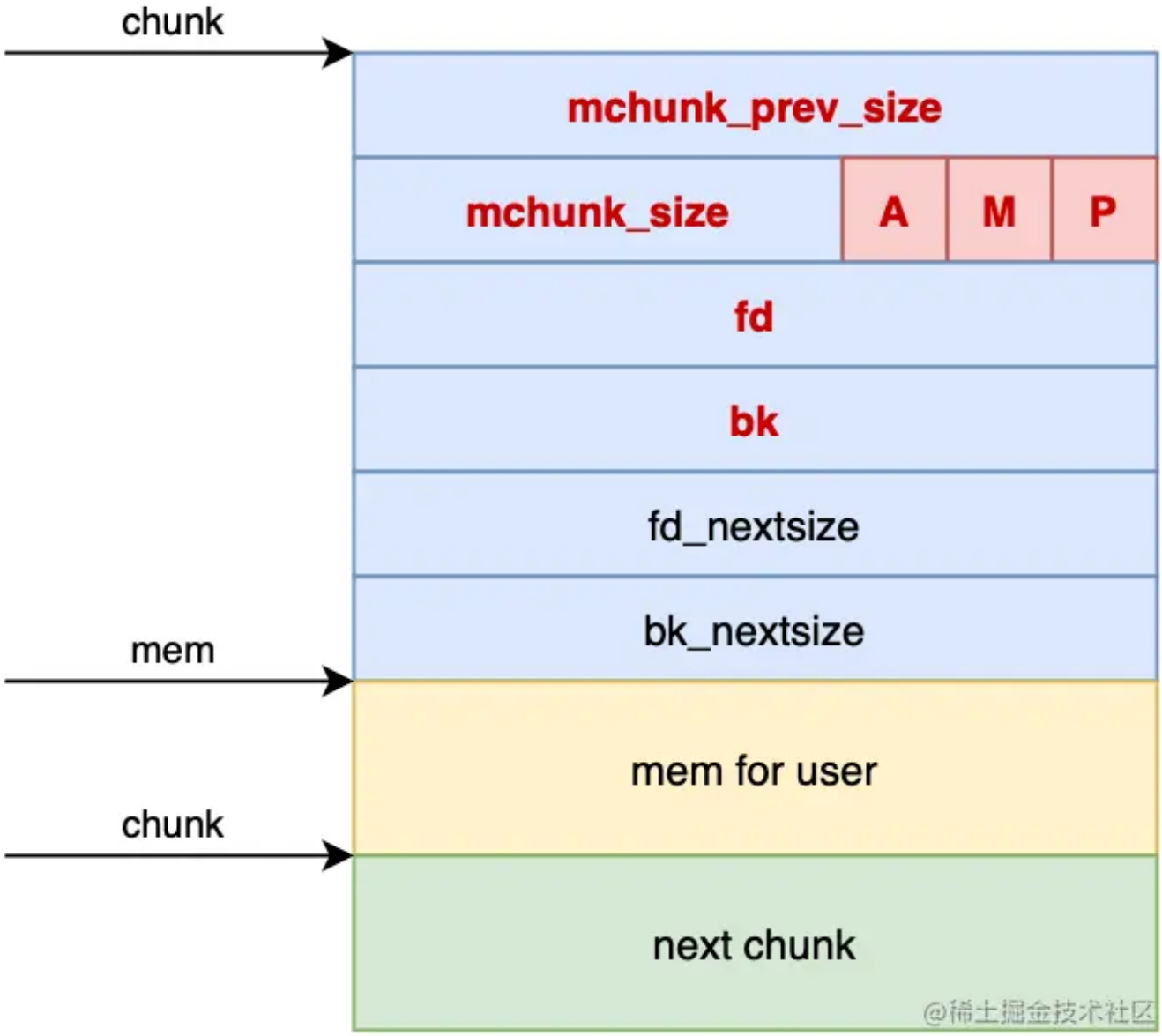



其中fastbinsY和bins是对实际内存块的管理和操作结构：

- fastbinsY: 用以保存fast bins
- bins[NBINS * 2 - 2]: unsorted bin（1个，bin[1]）、small bins（62 个，bin[2]~bin[63]）、large bins（63 个，bin[64]~bin[126]）的集合，一共有 126 个表项(NBINS = 128)，bin[0] 和 bin[127] 没有被使用

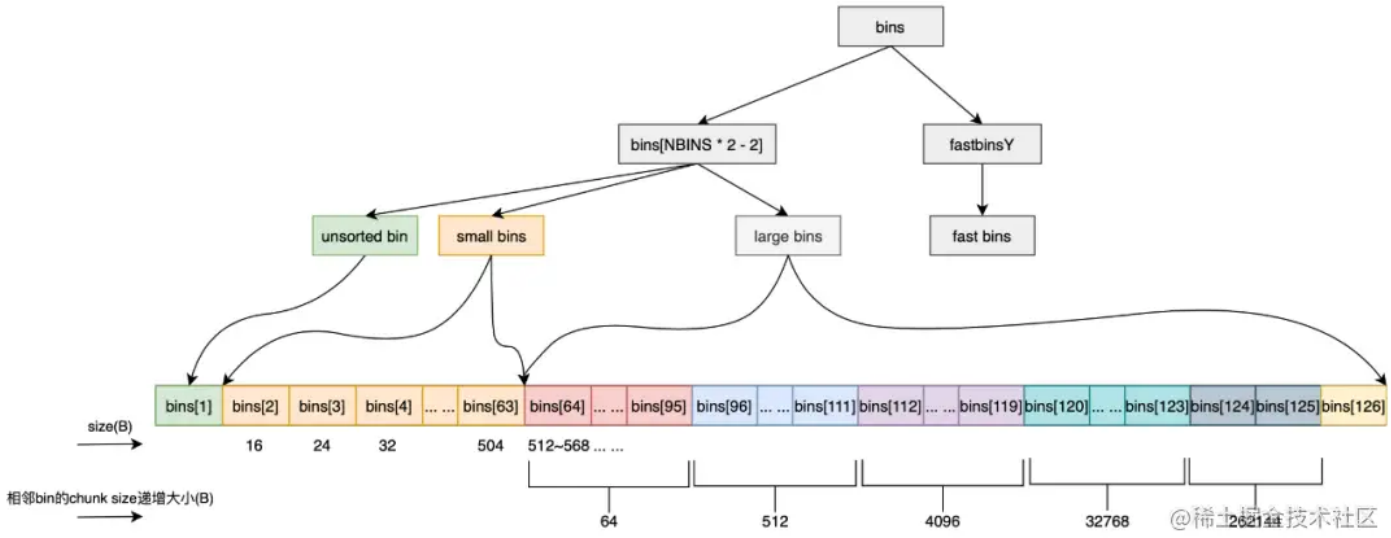
## malloc_chunk与bins

ptmalloc统一管理heap和mmap映射区域中空闲的chunk，当用户进行分配请求时，会先试图在空闲的chunk中查找和分割，从而避免频繁的系统调用，降低内存分配的开销。为了更好的管理和查找空闲chunk，在预分配的空间的前后添加了必要的控制信息，内存管理结构malloc_chunk的成员及作用如下：




- mchunk_prev_size: 前一个空闲chunk的大小
- mchunk_size: 当前chunk的大小
- 必要的属性标志位：
- 前一个chunk在使用中(P = 1)
- 当前chunk是mmap映射区域分配(M = 1)或是heap区域分配(M = 0)
- 当前chunk属于非主分配区(A = 0)或非主分配区(A = 1)
- fd和bk: chunk块空闲时存在，用于将空闲chunk块加入到空闲chunk块链表中统一管理

基于chunk的大小和使用方法，划分出以下几种bins：




- **fast bins**

  fast bins仅保存很小的堆，采用单链表串联，增删chunk都发生在链表的头部，进一步提高小内存的分配效率。fast bins记录着大小以8字节递增的bin链表，一般不会和其他堆块合并。

- **unsorted bin**

  small bins和large bins的缓冲区，用于加快分配的速度，chunk大小无尺寸限制，用户释放的堆块，会先进入unsorted bin。分配堆块时，会优先检查unsorted bin链表中是否存在合适的堆块，并进行切割并返回。

- **small bins**

  保存大小 < 512B的chunk的bin被称为small bins。small bins每个bin之间相差8个字节，同一个small bin中的chunk具有相同大小，采用双向循环链表串联。

- **large bins**

  保存大小 >= 512B的chunk的bin被称为large bins。large bins中的每一个bin分别包含了一个给定范围内的chunk，其中的chunk按大小降序，相同大小按时间降序。

当然，并不是所有chunk都按上述的方式来组织，其他常用的chunk，如：

- top chunk: 分配区的顶部空闲内存，当bins不能满足内存分配要求的时候，会尝试在top chunk分配。
- 当top chunk > 用户请求大小，top chunk会分为两个部分：用户请求大小(user chunk)和剩余top chunk大小(remainder chunk)
- 当top chunk < 用户所请求大小，top chunk就通过sbrk（main_arena）或mmap（non_main_arena）系统调用来扩容

## 2.2 内存分配与释放

概括内存malloc和free的流程大致如下：

## 内存分配malloc流程

1、获取分配区的锁

2、计算出需要分配的内存的chunk实际大小

3、如果chunk的大小 < max_fast，在fast bins上查找适合的chunk；如果不存在，转到5

4、如果chunk大小 < 512B，从small bins上去查找chunk，如果存在，分配结束

5、需要分配的是一块大的内存，或者small bins中找不到chunk：

- a.遍历fast bins，合并相邻的chunk，并链接到unsorted bin中
- b.遍历unsorted bin中的chunk：
- ①能够切割chunk直接分配，分配结束
- ②根据chunk的空间大小将其放入small bins或是large bins中，遍历完成后，转到6

6、需要分配的是一块大的内存，或者small bins和unsorted bin中都找不到合适的 chunk，且fast bins和unsorted bin中所有的chunk已清除：

- 从large bins中查找，反向遍历链表，直到找到第一个大小大于待分配的chunk进行切割，余下放入unsorted bin，分配结束

7、检索fast bins和bins没有找到合适的chunk，判断top chunk大小是否满足所需chunk的大小，从top chunk中分配

8、top chunk不能满足需求，需要扩大top chunk：

- a.主分区上，如果分配的内存 < 分配阈值（默认128KB），使用brk()分配；如果分配的内存 > 分配阈值，使用mmap分配
- b.非主分区上，使用mmap来分配一块内存

## 内存释放free流程

1、获取分配区的锁

2、如果free的是空指针，返回

3、如果当前chunk是mmap映射区域映射的内存，调用munmap()释放内存

4、如果chunk与top chunk相邻，直接与top chunk合并，转到8

5、如果chunk的大小 > max_fast，放入unsorted bin，并且检查是否有合并：

- a.没有合并情况则free
- b.有合并情况并且和top chunk相邻，转到8

6、如果chunk的大小 < max_fast，放入fast bin，并且检查是否有合并：

- a.fast bin并没有改变chunk的状态，没有合并情况则free
- b.有合并情况，转到7

7、在fast bin，如果相邻chunk空闲，则将这两个chunk合并，放入unsorted bin。如果合并后的大小 > 64KB，会触发进行fast bins的合并操作，fast bins中的chunk将被遍历合并，合并后的chunk会被放到unsorted bin中。合并后的chunk和top chunk相邻，则会合并到top chunk中，转到8

8、如果top chunk的大小 > mmap收缩阈值（默认为128KB），对于主分配区，会试图归还top chunk中的一部分给操作系统

# 三、优缺点

ptmalloc作为glibc默认的内存管理器，已经广泛的满足大多数大型项目的内存管理，同时它的实现思路也对后来的内存管理器提供了借鉴。

| | 系统调用函数 | ptmalloc |
|---|---|---|
| 优点 | • 无平台兼容问题，无使用管理库学习使用成本<br>• 单次内存操作性能好，直接操作内存，不存在多层管理、调用带来的额外损耗 | • 具有良好的兼容性、可移植性、稳定性，且兼具效率，广泛应用于各种大型项目中<br>• 解决了部分内存管理中的难题，如：降低多线程并发产生冲突、降低内存碎片化、降低内存malloc带来的性能损耗 |
| 缺点 | • 难以满足大型项目中高并发、高性能的要求：<br>◦ 单次申请和释放内存的操作耗时长<br>◦ 容易出现多线程并发冲突和加锁，内存分配效率低<br>◦ 根据系统设计合理的内存管理方案，开发难度大、成本高 | • 额外的损耗：性能损耗&&内存碎片化<br>◦ fast bins不合并可能会导致碎片化问题，但是可以加速释放的过程；small bins合并消除了碎片化的影响，但是减慢了free的速度<br>◦ 整理内存碎片需要对分配区加锁，这也增加了额外的性能消耗和线程冲突<br>• 存在内存暴增的风险：<br>◦ 线程数增加和频繁的内存分配，会导致锁的竞争，最终导致非主分配区增加，而非主分配区开辟后不会释放<br>◦ 当用于管理长生命周期的内存时，这个问题会更加明显：<br>▪ 后分配的内存先释放，而ptmalloc收缩内存是从top chunk开始，当与top chunk相邻的chunk不释放时，top chunk以下的chunk都无法释放；<br>▪ 当部分chunk长时间无法释放时，top chunk被迫升到更高的内存空间，缓存的内存块也会随之增加<br>• 其他风险<br>◦ 优化性能带来的安全性不高 |



ptmalloc的介绍暂告一段落，接下来的几篇文章将继续探讨高性能内存管理库的集大成者——jemalloc、tcmalloc内存管理库。

---------- END ----------

**参考资料**：

[1] sourceware.org/glibc/wiki/…

[2] sploitfun.wordpress.com/tag/ptmallo…

[3] www.cnblogs.com/biterror/p/…