glibc内存管理精髓

本文档来自公众号：高性能架构探索

微信扫码关注

前言

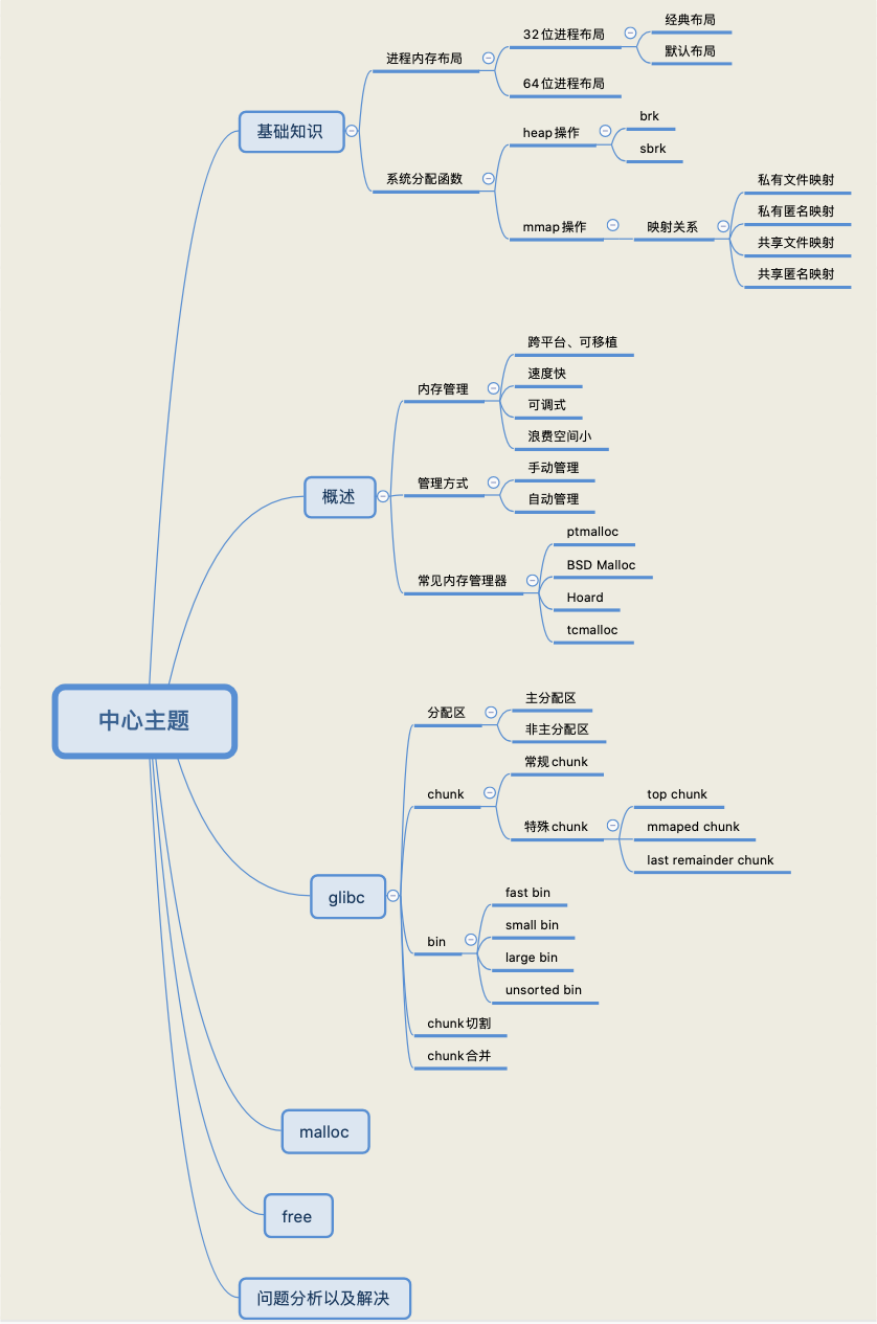
大家好，我是雨乐。

5年前，在上家公司的时候，因为进程OOM造成了上千万的损失，当时用了一个月的时间来分析glibc源码，最终将问题彻底解决。

最近在逛知乎的时候，发现不少人有对malloc/free有类似的疑惑，恰好自己有阅读过这方面的源码，所以将之前的源码阅读笔记整理了下，用了大概3周的时间写了这篇文章，分析glibc的内存管理精髓，相信对c/c++从业者都会有用。

> 由于本文涉及知识点较多，因此为了方便阅读，提供了PDF版本，关注本公众号**[高性能架构探索]，公众号留言、私信**获取；另外还有**批量免费计算机电子书，后台回复[pdf]免费获取**。

提纲


中心主题
基础知识
进程内存布局
32位进程布局
经典布局
默认布局
64位进程布局
系统分配函数
heap操作
brk
sbrk
mmap操作
映射关系
私有文件映射
私有匿名映射
共享文件映射
共享匿名映射
概述
内存管理
跨平台、可移植
速度快
可调式
浪费空间小
管理方式
手动管理
自动管理
常见内存管理器
ptmalloc
BSD Malloc
Hoard
tcmalloc
glibc
分配区
主分配区
非主分配区
chunk
常规chunk
特殊chunk
top chunk
mmaped chunk
last remainder chunk
bin
fast bin
small bin
large bin
unsorted bin
chunk切割
chunk合并
malloc
free
问题分析以及解决

1 写在前面

源码分析本身就很枯燥乏味，尤其是要将其写成通俗易懂的文章，更是难上加难。

本文尽可能的从读者角度去进行分析，重点写大家关心的点，必要的时候，会贴出部分源码，以加深大家的理解，尽可能的通过本文，让大家理解内存分配释放的本质原理。

接下来的内容，干货满满，对于你我都是一次收获的过程。主要从内存布局、glibc内存管理、malloc实现以及free实现几个点来带你遨游glibc内存管理的实质。最后，针对项目中的问题，指出了解决方案。

2 背景

几年前，在上家公司做了一个项目，暂且称之为SeedService吧。SeedService从kafka获取feed流的转、评、赞信息，加载到内存。因为每天数据不一样，所以kafka的topic也是按天来切分，整个server内部，类似于双指针实现，当天0点释放头一天的数据。

项目上线，一切运行正常。

但是几天之后，进程开始无缘无故的消失。开始定位问题，最终发现是因为内存暴增导致OOM，最终被操作系统kill掉。

弄清楚了进程消失的原因之后，开始着手分析内存泄漏。在解决了几个内存泄露的潜在问题以后，发现系统在高压力高并发环境下长时间运行仍然会发生内存暴增的现象，最终进程因OOM被操作系统杀掉。

由于内存管理不外乎三个层面，用户管理层，C 运行时库层，操作系统层，在操作系统层发现进程的内存暴增，同时又确认了用户管理层没有内存泄露，因此怀疑是 C 运行时库的问题，也就是Glibc 的内存管理方式导致了进程的内存暴增。

问题缩小到glibc的内存管理方面，把下面几个问题弄清楚，才能解决SeedService进程消失的问题：

- glibc 在什么情况下不会将内存归还给操作系统?
- glibc 的内存管理方式有哪些约束?适合什么样的内存分配场景?
- 我们的系统中的内存管理方式是与glibc 的内存管理的约束相悖的?
- glibc 是如何管理内存的?

带着上面这些问题，大概用了将近一个月的时间分析了glibc运行时库的内存管理代码，今天将当时的笔记整理了出来，希望能够对大家有用。

3 基础

Linux 系统在装载 elf 格式的程序文件时，会调用 loader 把可执行文件中的各个段依次载入到从某一地址开始的空间中。

用户程序可以直接使用系统调用来管理 heap 和mmap 映射区域，但更多的时候程序都是使用 C 语言提供的 malloc()和 free()函数来动态的分配和释放内存。stack区域是唯一不需要映射，用户却可以访问的内存区域，这也是利用堆栈溢出进行攻击的基础。

3.1 进程内存布局

计算机系统分为32位和64位，而32位和64位的进程布局是不一样的，即使是同为32位系统，其布局依赖于内核版本，也是不同的。

在介绍详细的内存布局之前，我们先描述几个概念：

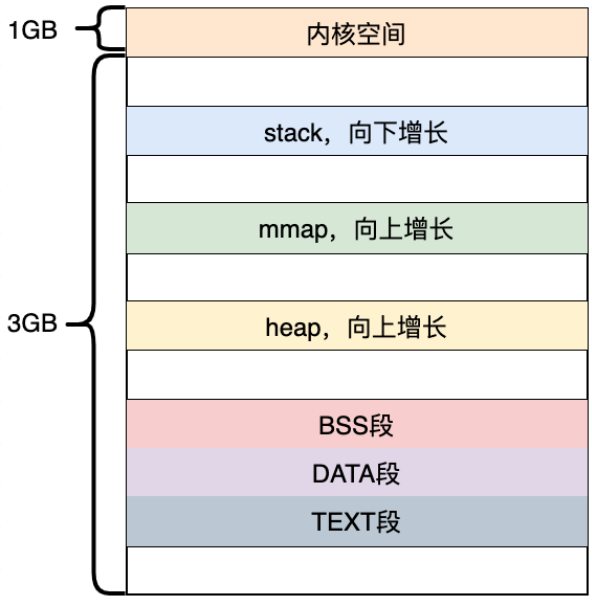
- 栈区（Stack）– 存储程序执行期间的本地变量和函数的参数，从高地址向低地址生长
- 堆区（Heap）动态内存分配区域，通过 malloc、new、free 和 delete 等函数管理
- 未初始化变量区（BSS）– 存储未被初始化的全局变量和静态变量
- 数据区（Data）– 存储在源代码中有预定义值的全局变量和静态变量
- 代码区（Text）– 存储只读的程序执行代码，即机器指令

3.1.1 32位进程内存布局

基于内核版本的不同，在32位系统中，进程内的布局也不一样。

3.1.1.1 经典布局

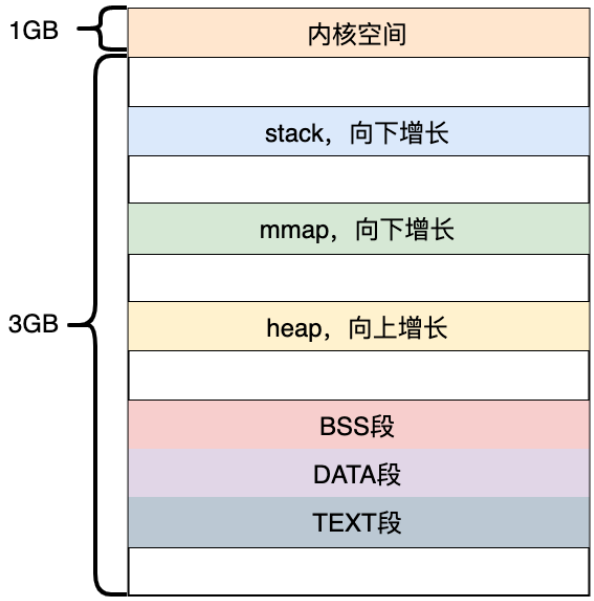
在Linux内核2.6.7以前，进程内存布局如下图所示。



在该内存布局示例图中，mmap 区域与栈区域相对增长，这意味着堆只有 1GB 的虚拟地址空间可以使用，继续增长就会进入 mmap 映射区域， 这显然不是我们想要的。这是由于 32 模式地址空间限制造成的，所以内核引入了另一种虚拟地址空间的布局形式。但对于 64 位系统，因为提供了巨大的虚拟地址空间，所以64位系统就采用的这种布局方式。

3.1.1.2 默认布局

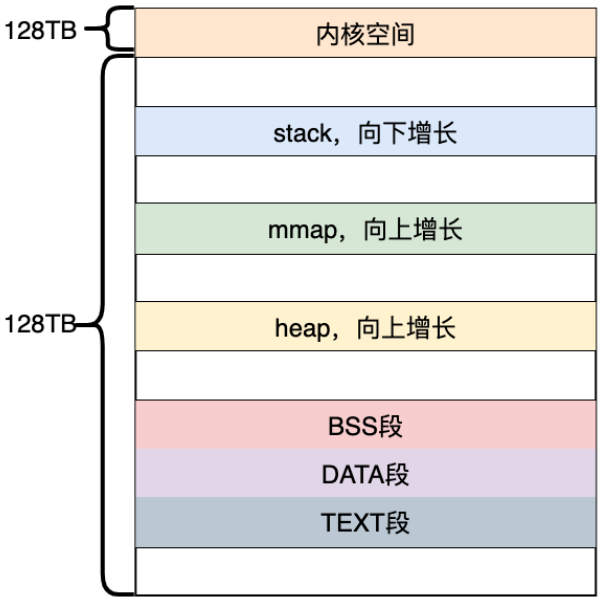
如上所示，由于经典内存布局具有空间局限性，因此在内核2.6.7以后，就引入了下图这种默认进程布局方式。



从上图可以看到，栈至顶向下扩展，并且栈是有界的。堆至底向上扩展，mmap 映射区域至顶向下扩展，mmap 映射区域和堆相对扩展，直至耗尽虚拟地址空间中的剩余区域，这种结构便于C运行时库使用 mmap 映射区域和堆进行内存分配。

3.1.2 64位进程内存布局

如之前所述，64位进程内存布局方式由于其地址空间足够，且实现方便，所以采用的与32位经典内存布局的方式一致，如下图所示：



3.2 操作系统内存分配函数

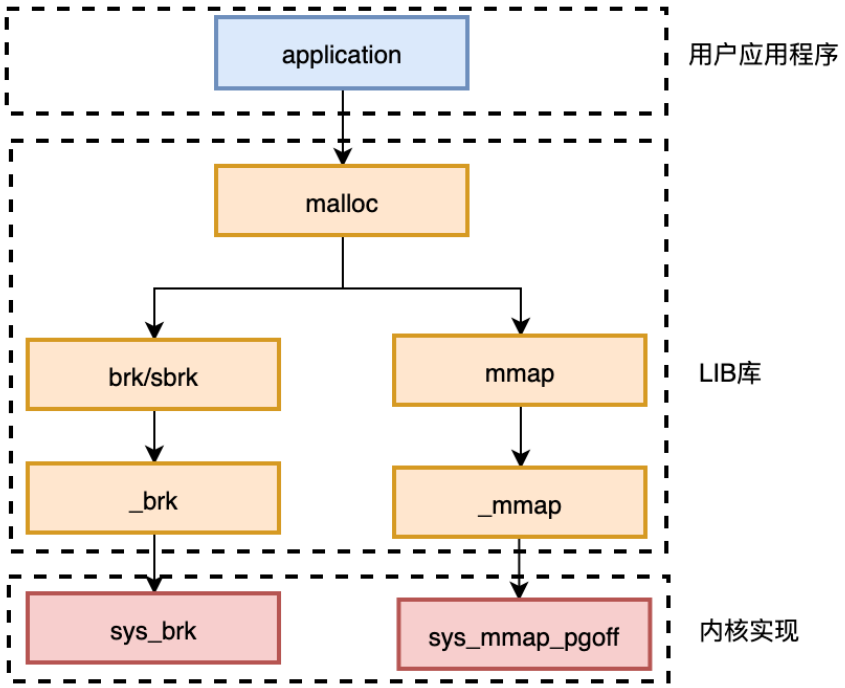
在之前介绍内存布局的时候，有提到过，heap 和mmap 映射区域是可以提供给用户程序使用的虚拟内存空间。那么我们该如何获得该区域的内存呢?

操作系统提供了相关的系统调用来完成内存分配工作。

- 对于heap的操作，操作系统提供了brk()函数，c运行时库提供了sbrk()函数。
- 对于mmap映射区域的操作，操作系统提供了mmap()和munmap()函数。

sbrk()，brk() 或者 mmap() 都可以用来向我们的进程添加额外的虚拟内存。而glibc就是使用这些函数来向操作系统申请虚拟内存，以完成内存分配的。

> 这里要提到一个很重要的概念，内存的延迟分配，只有在真正访问一个地址的时候才建立这个地址的物理映射，这是 Linux 内存管理的基本思想之一。Linux 内核在用户申请内存的时候，只是给它分配了一个线性区（也就是虚拟内存），并没有分配实际物理内存；只有当用户使用这块内存的时候，内核才会分配具体的物理页面给用户，这时候才占用宝贵的物理内存。内核释放物理页面是通过释放线性区，找到其所对应的物理页面，将其全部释放的过程。



进程的内存结构，在内核中，是用mm_struct来表示的，其定义如下：

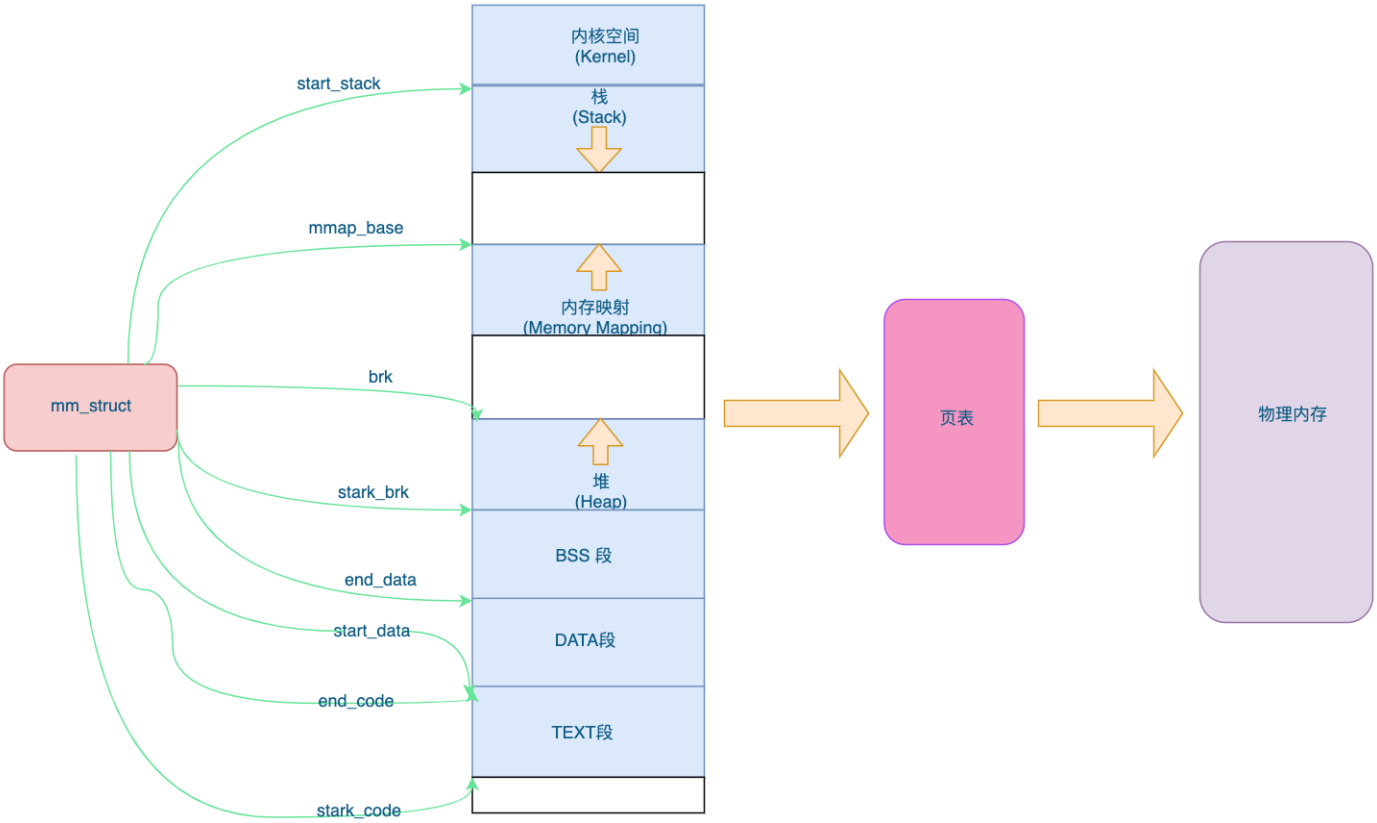
```
struct mm_struct {
 ...
 unsigned long (*get_unmapped_area) (struct file *filp,
 unsigned long addr, unsigned long len,
 unsigned long pgoff, unsigned long flags);
 ...
 unsigned long mmap_base; /* base of mmap area */
 unsigned long task_size; /* size of task vm space */
 ...
 unsigned long start_code, end_code, start_data, end_data;
 unsigned long start_brk, brk, start_stack;
 unsigned long arg_start, arg_end, env_start, env_end;
 ...
```

```
14 }
```

在上述mm_struct结构中：

- [start_code,end_code)表示代码段的地址空间范围。
- [start_data,end_start)表示数据段的地址空间范围。
- [start_brk,brk)分别表示heap段的起始空间和当前的heap指针。
- [start_stack,end_stack)表示stack段的地址空间范围。
- mmap_base表示memory mapping段的起始地址。

C语言的动态内存分配基本函数是 malloc()，在 Linux 上的实现是通过内核的 brk 系统调用。brk()是一个非常简单的系统调用， 只是简单地改变mm_struct结构的成员变量 brk 的值。



3.2.1 Heap操作

在前面有提过，有两个函数可以直接从堆(Heap)申请内存，brk()函数为系统调用，sbrk()为c库函数。

系统调用通常提过一种最小的功能，而库函数相比系统调用，则提供了更复杂的功能。在glibc中，malloc就是调用sbrk()函数将数据段的下界移动以来代表内存的分配和释放。sbrk()函数在内核的管理下，将虚拟地址空间映射到内存，供malloc()函数使用。

下面为brk()函数和sbrk()函数的声明。

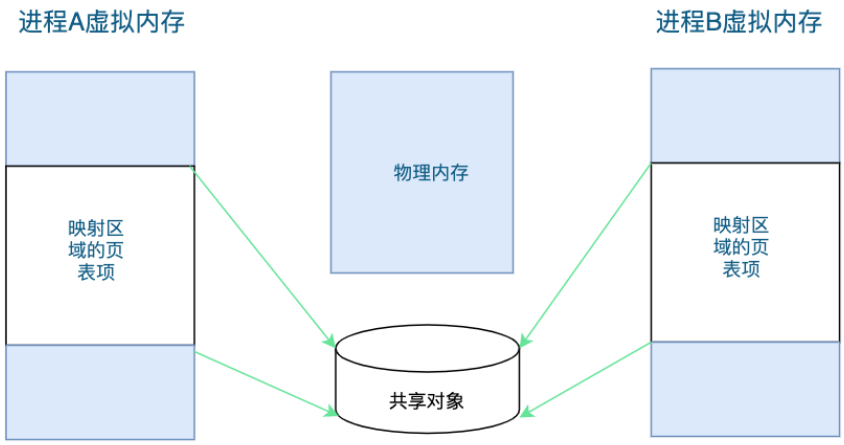
```
#include <unistd.h>
int brk(void *addr);

void *sbrk(intptr_t increment);
```

> 需要说明的是，当sbrk()的参数increment为0时候，sbrk()返回的是进程当前brk值。increment 为正数时扩展 brk 值，当 increment 为负值时收缩 brk 值。

3.2.2 MMap操作

在Linux系统中我们可以使用mmap用来在进程虚拟内存地址空间中分配地址空间，创建和物理内存的映射关系。



mmap()函数将一个文件或者其它对象映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会清零。

munmap 执行相反的操作，删除特定地址区域的对象映射。

函数的定义如下：

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

int munmap(void *addr, size_t length);
```

.

映射关系分为以下两种：

- 文件映射：磁盘文件映射进程的虚拟地址空间，使用文件内容初始化物理内存。
- 匿名映射：初始化全为0的内存空间

映射关系是否共享，可以分为：

- 私有映射(MAP_PRIVATE)
  - 多进程间数据共享，修改不反应到磁盘实际文件，是一个copy-on-write（写时复制）的映射方式。
- 共享映射(MAP_SHARED)
  - 多进程间数据共享，修改反应到磁盘实际文件中。

因此，整个映射关系总结起来分为以下四种：

- 私有文件映射
  多个进程使用同样的物理内存页进行初始化，但是各个进程对内存文件的修改不会共享，也不会反应到物理文件中
- 私有匿名映射
  - mmap会创建一个新的映射，各个进程不共享，这种使用主要用于分配内存(malloc分配大内存会调用mmap)。例如开辟新进程时，会为每个进程分配虚拟的地址空间，这些虚拟地址映射的物理内存空间各个进程间读的时候共享，写的时候会copy-on-write。
- 共享文件映射
  - 多个进程通过虚拟内存技术共享同样的物理内存空间，对内存文件的修改会反应到实际物理文件中，也是进程间通信(IPC)的一种机制。
- 共享匿名映射
  - 这种机制在进行fork的时候不会采用写时复制，父子进程完全共享同样的物理内存

页，这也就实现了父子进程通信(IPC)。

这里值得注意的是，mmap只是在虚拟内存分配了地址空间，只有在第一次访问虚拟内存的时候才分配物理内存。

在mmap之后，并没有在将文件内容加载到物理页上，只有在虚拟内存中分配了地址空间。当进程在访问这段地址时，通过查找页表，发现虚拟内存对应的页没有在物理内存中缓存，则产生"缺页"，由内核的缺页异常处理程序处理，将文件对应内容，以页为单位(4096)加载到物理内存，注意是只加载缺页，但也会受操作系统一些调度策略影响，加载的比所需的多。

> 下面的内容将是本文的重点中的重点，对于了解内存布局以及后面glibc的内存分配原理至关重要，必要的话，可以多阅读几次。

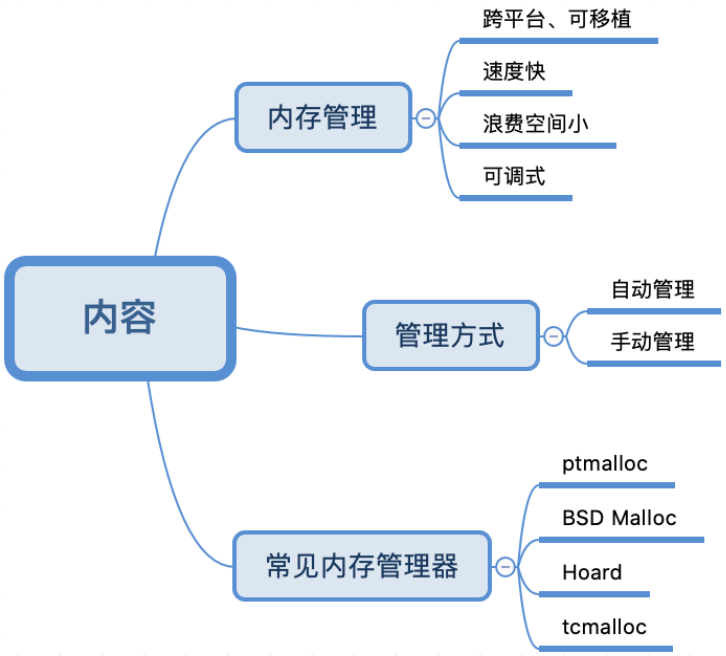
4 概述

在前面，我们有提到在堆上分配内存有两个函数，分别为brk()系统调用和sbrk()c运行时库函数，在内存映射区分配内存有mmap函数。

现在我们假设一种情况，如果每次分配，都直接使用brk(),sbrk()或者mmap()函数进行多次内存分配。如果程序频繁的进行内存分配和释放，都是和操作系统直接打交道，那么性能可想而知。

这就引入了一个概念，***内存管理*** 。

本节大纲如下：



4.1 内存管理

内存管理是指软件运行时对计算机内存资源的分配和使用的技术。其最主要的目的是如何高效，快速的分配，并且在适当的时候释放和回收内存资源。

一个好的内存管理器，需要具有以下特点：

1、跨平台、可移植

通常情况下，内存管理器向操作系统申请内存，然后进行再次分配。所以，针对不同的操作系统，内存管理器就需要支持操作系统兼容，让使用者在跨平台的操作上没有区别。

2、浪费空间小
内存管理器管理内存，如果内存浪费比较大，那么显然这就不是一个优秀的内存管理器。通常说的内存碎片，就是浪费空间的罪魁祸首，若内存管理器中有大量的内存碎片，它们是一些不连续的小块内存，它们总量可能很大，但无法使用，这显然也不是一个优秀的内存管理器。

3、速度快
之所以使用内存管理器，根本原因就是为了分配/释放快。

4、调试功能
作为一个 C/C++程序员，内存错误可以说是我们的噩梦，上一次的内存错误一定还让你记忆犹新。内存管理器提供的调试功能，强大易用，特别对于嵌入式环境来说，内存错误检测工具缺乏，内存管理器提供的调试功能就更是不可或缺了。

4.2 管理方式

内存管理的管理方式，分为 *手动管理* 和 *自动管理* 两种。

所谓的手动管理，就是使用者在申请内存的时候使用malloc等函数进行申请，在需要释放的时候，需要调用free函数进行释放。一旦用过的内存没有释放，就会造成内存泄漏，占用更多的系统内存；如果在使用结束前释放，会导致危险的悬挂指针，其他对象指向的内存已经被系统回收或者重新使用。

自动管理内存由编程语言的内存管理系统自动管理，在大多数情况下不需要使用者的参与，能够自动释放不再使用的内存。

4.2.1 手动管理

手动管理内存是一种比较传统的内存管理方式，C/C++ 这类系统级的编程语言不包含狭义上的自动内存管理机制，使用者需要主动申请或者释放内存。
经验丰富的工程师能够精准的确定内存的分配和释放时机，人肉的内存管理策略只要做到足够精准，使用手动管理内存的方式可以提高程序的运行性能，也不会造成内存安全问题。

但是，毕竟这种经验丰富且能精准确定内存和分配释放实际的使用者还是比较少的，只要是人工处理，总会带来一些错误，内存泄漏和悬挂指针基本是 C/C++ 这类语言中最常出现的错误，手动的内存管理也会占用工程师的大量精力，很多时候都需要思考对象应该分配到栈上还是堆上以及堆上的内存应该何时释放，维护成本相对来说还是比较高的，这也是必然要做的权衡。

4.2.2 自动管理

自动管理内存基本是现代编程语言的标配，因为内存管理模块的功能非常确定，所以我们可以在编程语言的编译期或者运行时中引入自动的内存管理方式，最常见的自动内存管理机制就是垃圾回收，不过除了垃圾回收之外，一些编程语言也会使用自动引用计数辅助内存的管理。

自动的内存管理机制可以帮助工程师节省大量的与内存打交道的时间，让使用者将全部的精力都放在核心的业务逻辑上，提高开发的效率；在一般情况下，这种自动的内存管理机制都可以很好地解决内存泄漏和悬挂指针的问题，但是这也会带来额外开销并影响语言的运行时性能。

4.1 常见的内存管理器

1、ptmalloc
ptmalloc是隶属于glibc(GNU Libc)的一款内存分配器，现在在Linux环境上，我们使用的运行库的内存分配(malloc/new)和释放(free/delete)就是由其提供。

2、 BSD Malloc：BSD Malloc 是随 4.2 BSD 发行的实现，包含在 FreeBSD 之中，这个分配程序可以从预先确实大小的对象构成的池中分配对象。它有一些用于对象大小的size 类，这些对象的大小为 2 的若干次幂减去某一常数。所以，如果您请求给定大小的一个对象，它就简单地分配一个与之匹配的 size 类。这样就提供了一个快速的实现，但是可能会浪费内存。

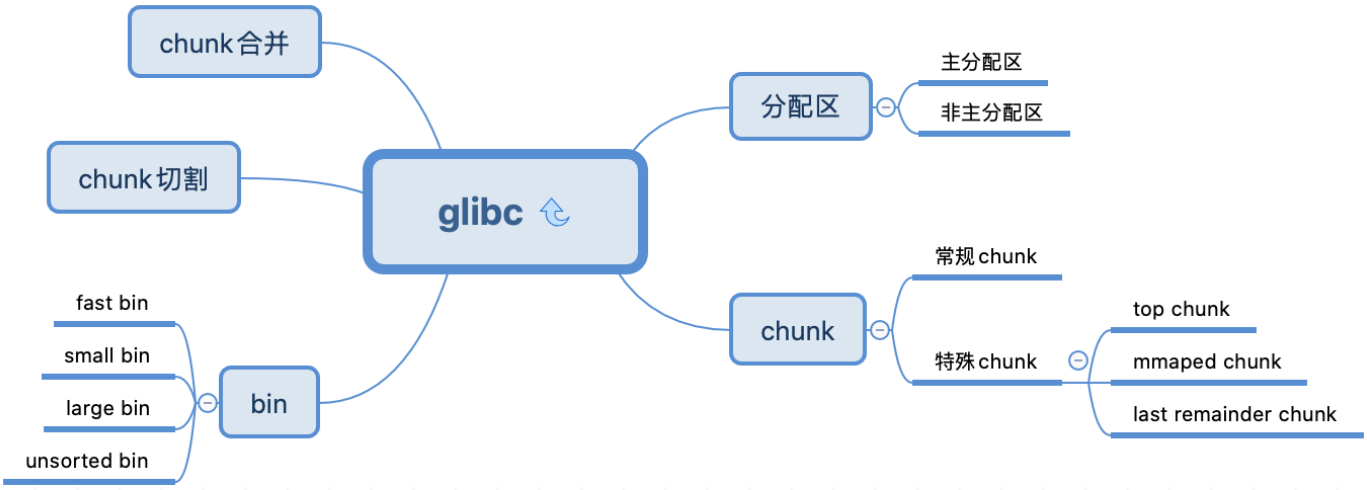
3、Hoard：编写 Hoard 的目标是使内存分配在多线程环境中进行得非常快。因此，它的构造以锁的使用为中心，从而使所有进程不必等待分配内存。它可以显著地加快那些进行很多分配和回收的多线程进程的速度。

4、TCMalloc：Google 开发的内存分配器，在不少项目中都有使用，例如在 Golang 中就使用了类似的算法进行内存分配。它具有现代化内存分配器的基本特征：对抗内存碎片、在多核处理器能够 scale。据称，它的内存分配速度是 glibc2.3 中实现的 malloc的数倍。

5 glibc之内存管理(ptmalloc)

因为本次事故就是用的运行库函数new/delete进行的内存分配和释放，所以本文将着重分析glibc下的内存分配库ptmalloc。

本节大纲如下：



在c/c++中，我们分配内存是在堆上进行分配，那么这个堆，在glibc中是怎么表示的呢？我们先看下堆的结构声明：

```
typedef struct _heap_info
{
  mstate ar_ptr;            /* Arena for this heap. */
  struct _heap_info *prev;  /* Previous heap. */
  size_t size;              /* Current size in bytes. */
  size_t mprotect_size;     /* Size in bytes that has been
mprotected
                               PROT_READ|PROT_WRITE.  */
  /* Make sure the following data is properly aligned,
particularly
     that sizeof (heap_info) + 2 * SIZE_SZ is a multiple of
     MALLOC_ALIGNMENT. */
  char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
```

在堆的上述定义中，ar_ptr是指向分配区的指针，堆之间是以链表方式进行连接，后面我会详细讲述进程布局下，堆的结构表示图。

在开始这部分之前，我们先了解下一些概念。

5.1 分配区(arena)

> ptmalloc对进程内存是通过一个个Arena来进行管理的。

在ptmalloc中，分配区分为主分配区(arena)和非主分配区(narena)，分配区用struct malloc_state来表示。主分配区和非主分配区的区别是 *主分配区可以使用sbrk和mmap向os申请内存，而非分配区只能通过mmap向os申请内存* 。

当一个线程调用malloc申请内存时，该线程先查看线程私有变量中是否已经存在一个分配区。如果存在，则对该分配区加锁，加锁成功的话就用该分配区进行内存分配；失败的话则搜索环形链表找一个未加锁的分配区。如果所有分配区都已经加锁，那么malloc会开辟一个新的分配区加入环形链表并加锁，用它来分配内存。释放操作同样需要获得锁才能进行。

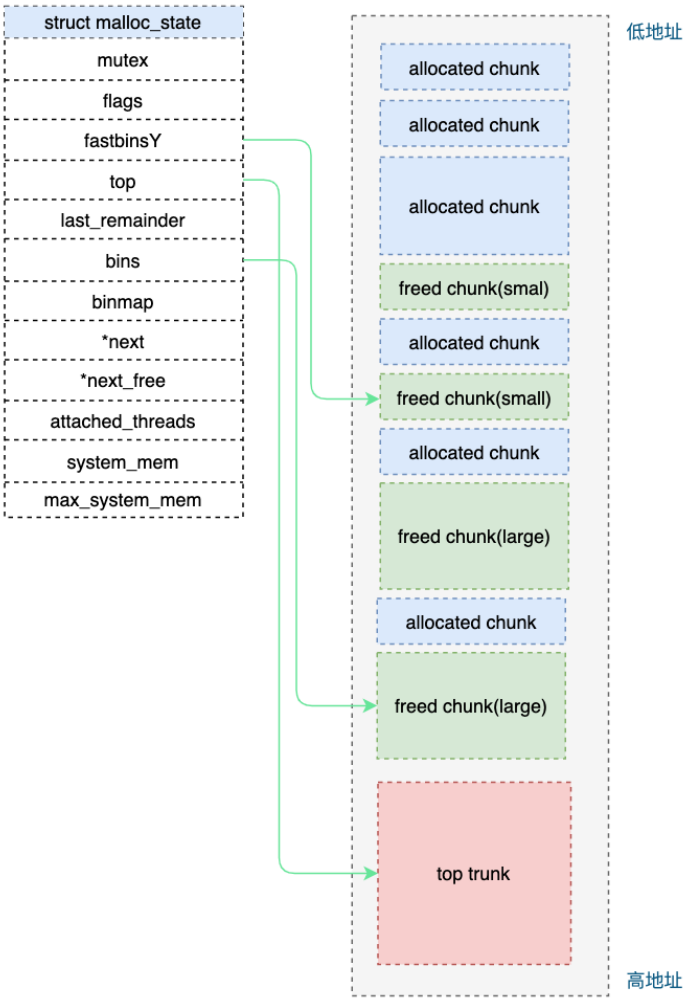
需要注意的是，非主分配区是通过mmap向os申请内存，一次申请64MB，一旦申请了，该分配区就不会被释放，为了避免资源浪费，ptmalloc对分配区是有个数限制的。

> 对于32位系统，分配区最大个数 = 2 * CPU核数 + 1
> 对于64位系统，分配区最大个数 = 8 * CPU核数 + 1

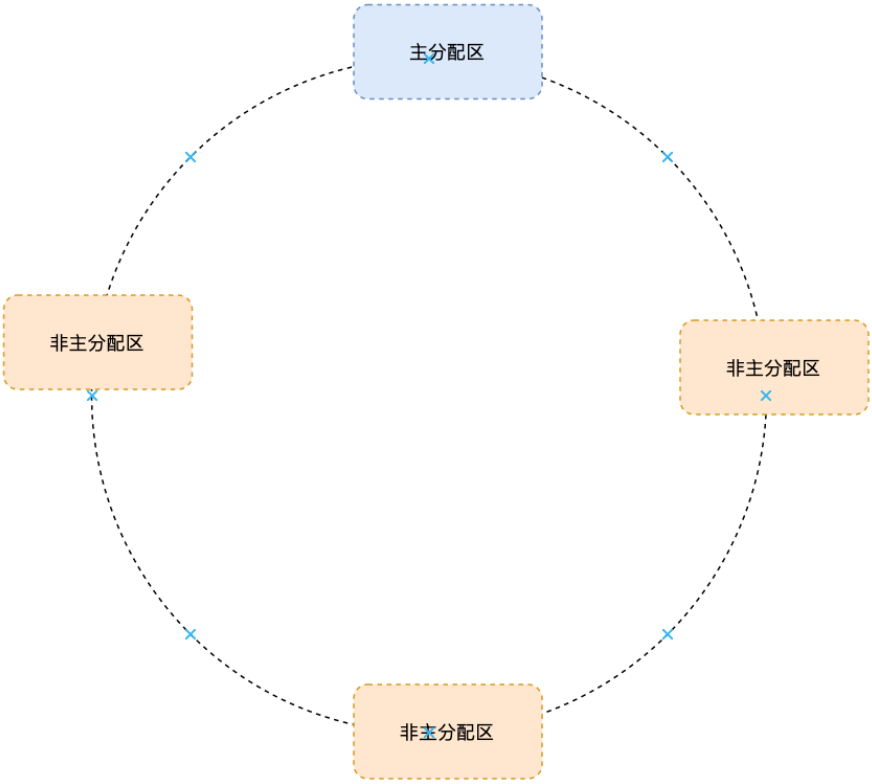
堆管理结构：

```
struct malloc_state {
 mutex_t mutex;                  /* Serialize access. */
 int flags;                        /* Flags (formerly in
max_fast). */
 #if THREAD_STATS
 /* Statistics for locking. Only used if THREAD_STATS is
defined. */
```

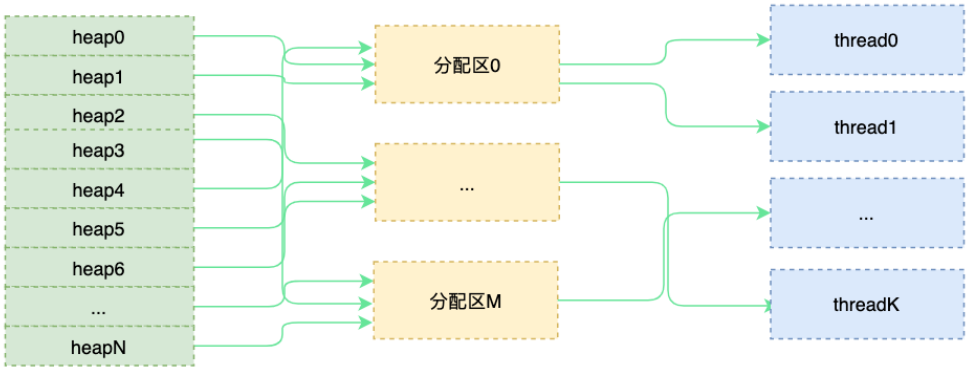
```
long stat_lock_direct, stat_lock_loop, stat_lock_wait;
#endif
mfastbinptr fastbins[NFASTBINS];    /* Fastbins */
mchunkptr top;
mchunkptr last_remainder;
mchunkptr bins[NBINS * 2];
unsigned int binmap[BINMAPSIZE];   /* Bitmap of bins */
struct malloc_state *next;           /* Linked list */
INTERNAL_SIZE_T system_mem;
INTERNAL_SIZE_T max_system_mem;
};
```



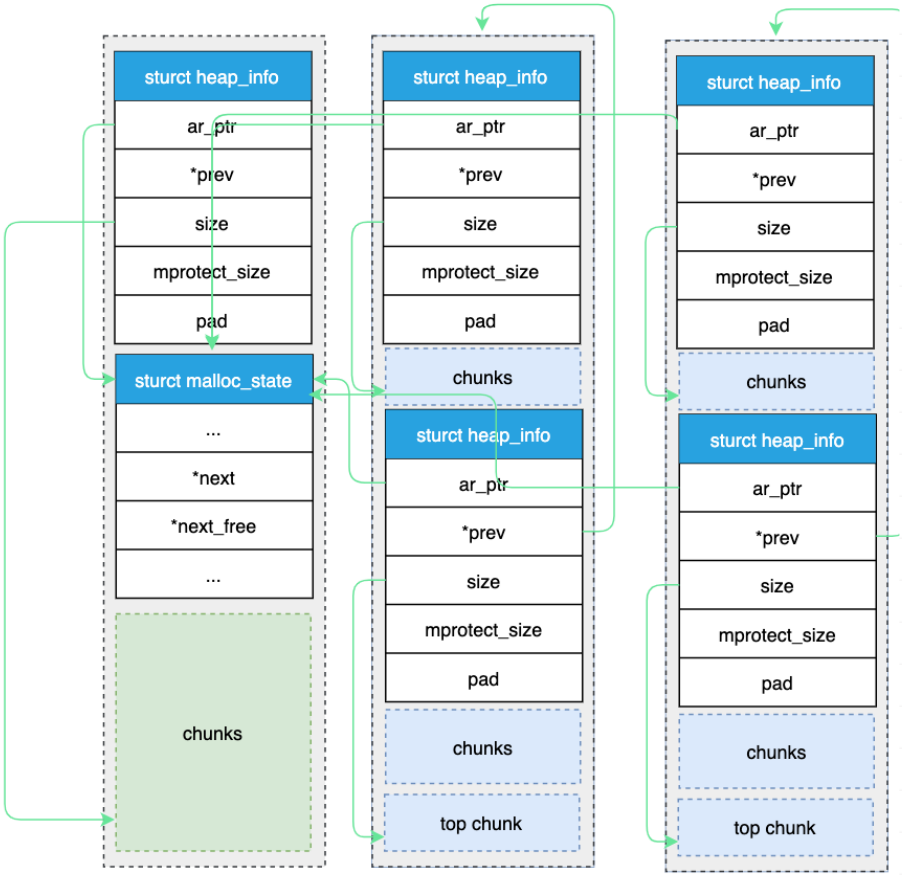
每一个进程只有一个主分配区和若干个非主分配区。主分配区由main线程或者第一个线程来创建持有。主分配区和非主分配区用环形链表连接起来。分配区内有一个变量mutex以支持多线程访问。



在前面有提到，在每个分配区中都有一个变量mutex来支持多线程访问。每个线程一定对应一个分配区，但是一个分配区可以给多个线程使用，同时一个分配区可以由一个或者多个的堆组成，同一个分配区下的堆以链表方式进行连接，它们之间的关系如下图：



一个进程的动态内存，由分配区管理，一个进程内有多个分配区，一个分配区有多个堆，这就组成了复杂的进程内存管理结构。



需要注意几个点：

- 主分配区通过brk进行分配，非主分配区通过mmap进行分配
- 非主分配区虽然是mmap分配，但是和大于128K直接使用mmap分配没有任何联系。大于128K的内存使用mmap分配，使用完之后直接用ummap还给系统
- 每个线程在malloc会先获取一个area，使用area内存池分配自己的内存，这里存在竞争问题
- 为了避免竞争，我们可以使用线程局部存储，thread cache（tcmalloc中的tc正是此意），线程局部存储对area的改进原理如下：
- 如果需要在一个线程内部的各个函数调用都能访问、但其它线程不能访问的变量（被称为static memory local to a thread 线程局部静态变量），就需要新的机制来实现。这就是TLS。
- thread cache本质上是在static区为每一个thread开辟一个独有的空间，因为独有，不再有竞争
- 每次malloc时，先去线程局部存储空间中找area，用thread cache中的area分配存在thread area中的chunk。当不够时才去找堆区的area

5.2 chunk

ptmalloc通过malloc_chunk来管理内存，给User data前存储了一些信息，使用边界标记区分各个chunk。

chunk定义如下：

```
struct malloc_chunk {
  INTERNAL_SIZE_T      prev_size;    /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;         /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;           /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize;      /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

- prev_size: 如果前一个chunk是空闲的，则该域表示前一个chunk的大小，如果前一个chunk不空闲，该域无意义。

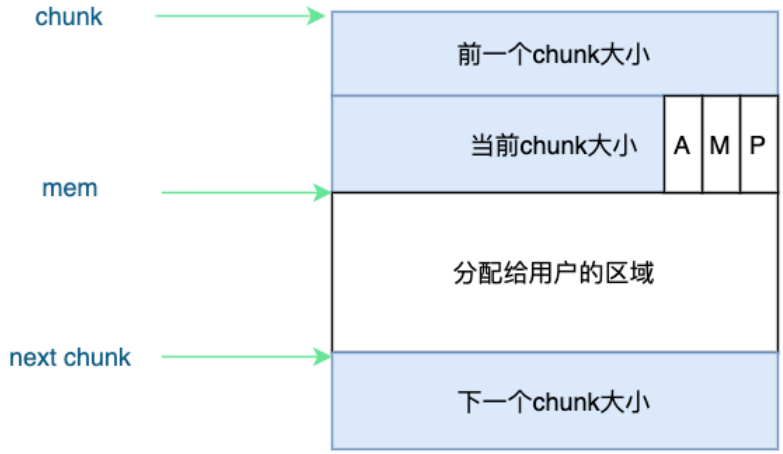
一段连续的内存被分成多个chunk，prev_size记录的就是相邻的前一个chunk的size，知道当前chunk的地址，减去prev_size便是前一个chunk的地址。prev_size主要用于相邻空闲chunk的合并 。

- size ： 当前 chunk 的大小，并且记录了当前 chunk 和前一个 chunk 的一些属性，包括前一个 chunk 是否在使用中，当前 chunk 是否是通过 mmap 获得的内存，当前 chunk 是否属于非主分配区。
- fd 和 bk ： 指针 fd 和 bk 只有当该 chunk 块空闲时才存在，其作用是用于将对应的空闲 chunk 块加入到空闲chunk 块链表中统一管理，如果该 chunk 块被分配给应用程序使用，那么这两个指针也就没有用（该 chunk 块已经从空闲链中拆出）

了，所以也当作应用程序的使用空间，而不至于浪费。

- fd_nextsize 和 bk_nextsize: 当前的 chunk 存在于 large bins 中时，large bins 中的空闲 chunk 是按照大小排序的，但同一个大小的 chunk 可能有多个，增加了这两个字段可以加快遍历空闲 chunk ，并查找满足需要的空闲 chunk ， fd_nextsize 指向下一个比当前 chunk 大小大的第一个空闲 chunk ，bk_nextszie 指向前一个比当前 chunk 大小小的第一个空闲 chunk 。（同一大小的chunk可能有多块，在总体大小有序的情况下，要想找到下一个比自己大或小的chunk，需要遍历所有相同的chunk，所以才有fd_nextsize和bk_nextsize这种设计） 如果该 chunk 块被分配给应用程序使用，那么这两个指针也就没有用（该 chunk 块已经从 size 链中拆出）了，所以也当作应用程序的使用空间，而不至于浪费。

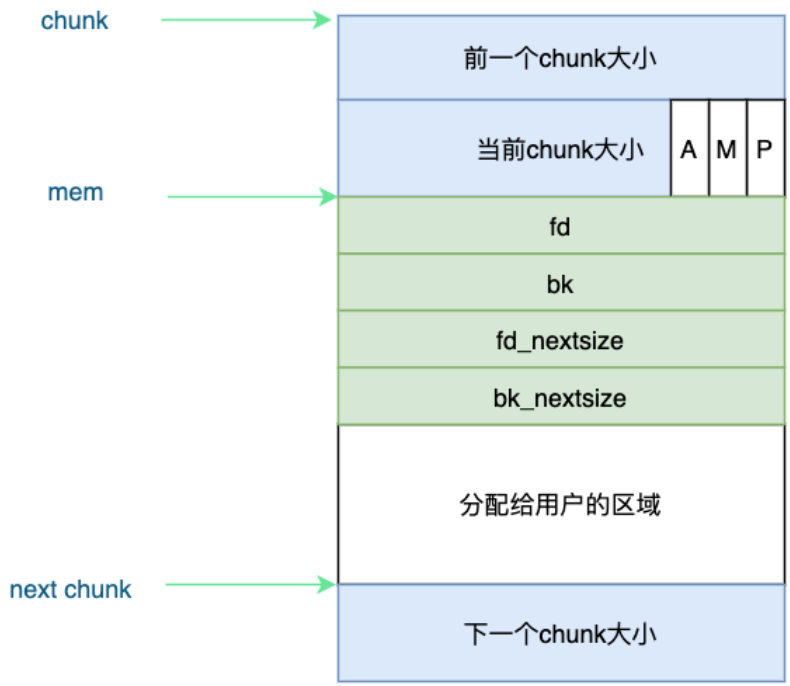
正如上面所描述，在ptmalloc中，为了尽可能的节省内存，使用中的chunk和未使用的chunk在结构上是不一样的。



在上图中：

- chunk指针指向chunk开始的地址
- mem指针指向用户内存块开始的地址。
- p=0时，表示前一个chunk为空闲，prev_size才有效
- p=1时，表示前一个chunk正在使用，prev_size无效 p主要用于内存块的合并操作；ptmalloc 分配的第一个块总是将p设为1，以防止程序引用到不存在的区域
- M=1 为mmap映射区域分配；M=0为heap区域分配
- A=0 为主分配区分配；A=1 为非主分配区分配。

与非空闲chunk相比，空闲chunk在用户区域多了四个指针，分别为fd,bk,fd_nextsize,bk_nextsize，这几个指针的含义在上面已经有解释，在此不再赘述。



5.3 空闲链表(bins)

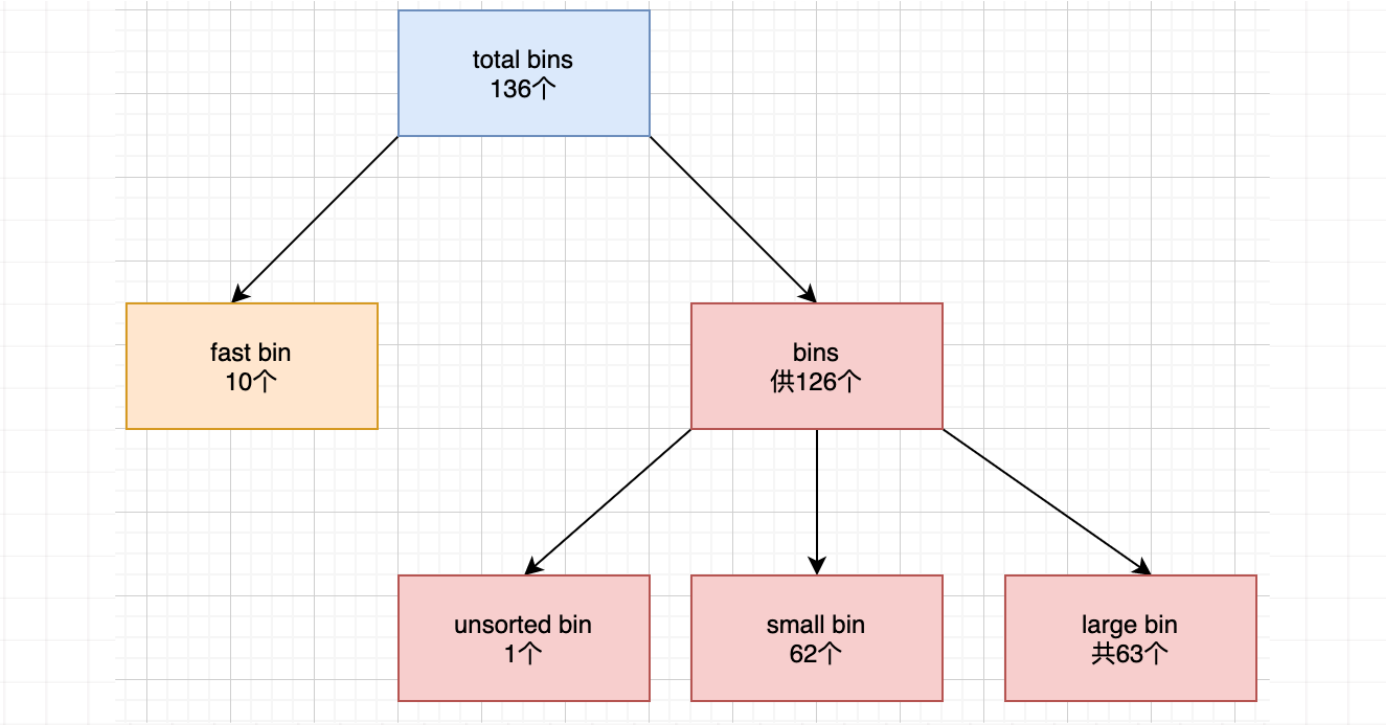
用户调用free函数释放内存的时候，ptmalloc并不会立即将其归还操作系统，而是将其放入空闲链表(bins)中，这样下次再调用malloc函数申请内存的时候，就会从bins中取出一块返回，这样就避免了频繁调用系统调用函数，从而降低内存分配的开销。

在ptmalloc中，会将大小相似的chunk链接起来，叫做bin。总共有128个bin供ptmalloc使用。
根据chunk的大小，ptmalloc将bin分为以下几种：

- fast bin
- unsorted bin
- small bin
- large bin

从前面malloc_state结构定义，对bin进行分类，可以分为fast bin和bins，其中unsorted bin、small bin 以及 large bin属于bins。

在glibc中，上述4中bin的个数都不等，如下图所示：



5.3.1 fast bin

程序在运行时会经常需要申请和释放一些较小的内存空间。当分配器合并了相邻的几个小的 chunk 之后,也许马上就会有另一个小块内存的请求,这样分配器又需要从大的空闲内存中切分出一块,这样无疑是比较低效的,故而,malloc 中在分配过程中引入了 fast bins。

在前面malloc_state定义中

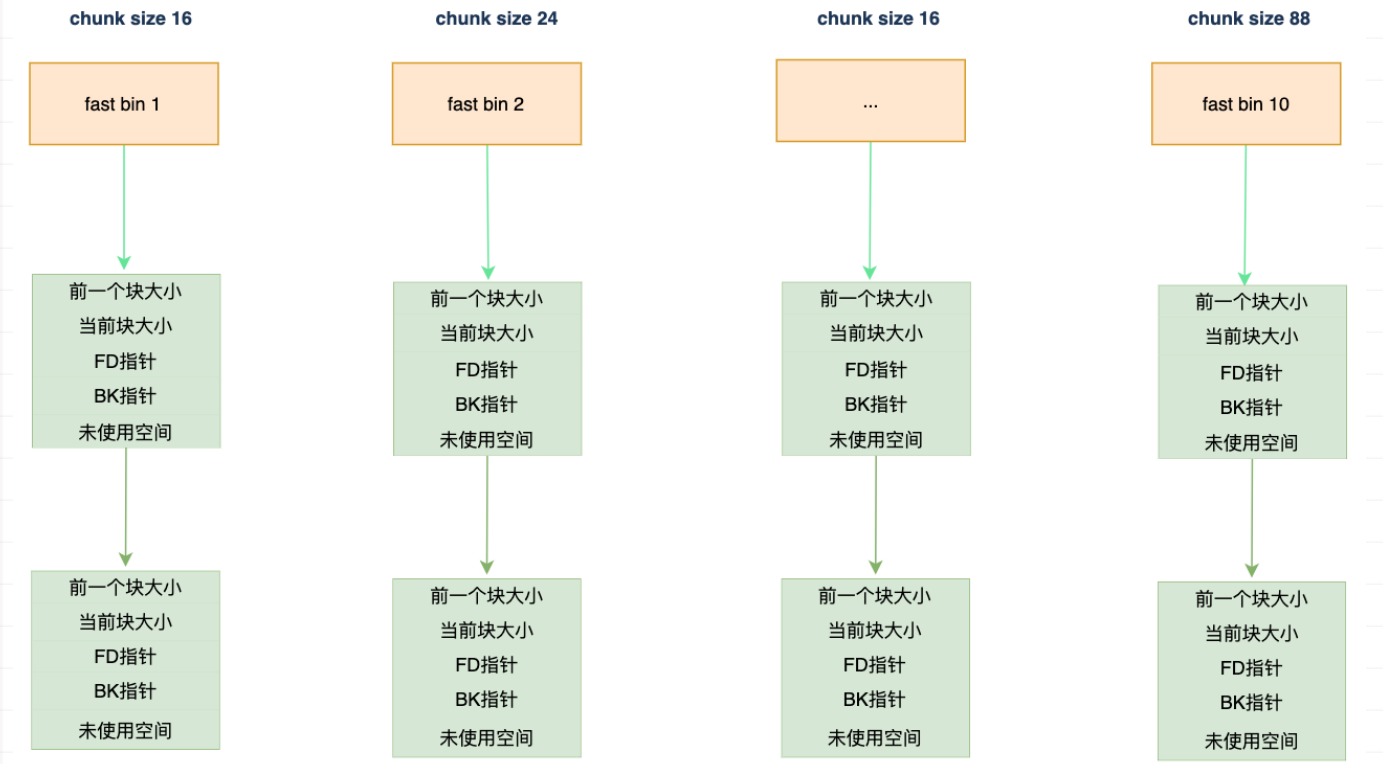
```
mfastbinptr fastbins[NFASTBINS]; // NFASTBINS  = 10
```

1. fast bin的个数是10个
2. 每个fast bin都是一个单链表(只使用fd指针)。这是因为fast bin无论是添加还是移除chunk都是在链表尾进行操作，也就是说，对fast bin中chunk的操作，采用的是LIFO(后入先出)算法：添加操作(free内存)就是将新的fast chunk加入链表尾，删除操作(malloc内存)就是将链表尾部的fast chunk删除。
3. chunk size：10个fast bin中所包含的chunk size以8个字节逐渐递增，即第一个fast bin中chunk size均为16个字节，第二个fast bin的chunk size为24字节，以此类推，最后一个fast bin的chunk size为80字节。
4. 不会对free chunk进行合并操作。这是因为fast bin设计的初衷就是小内存的快速分配和释放，因此系统将属于fast bin的chunk的P(未使用标志位)总是设置为1，这样即使当fast bin中有某个chunk同一个free chunk相邻的时候，系统也不会进行自动

合并操作，而是保留两者。

5. malloc操作：在malloc的时候，如果申请的内存大小范围在fast bin的范围内，则先在fast bin中查找，如果找到了，则返回。否则则从small bin、unsorted bin以及large bin中查找。
6. free操作：先通过chunksize函数根据传入的地址指针获取该指针对应的chunk的大小；然后根据这个chunk大小获取该chunk所属的fast bin，然后再将此chunk添加到该fast bin的链尾即可。

下面是fastbin结构图：



5.3.2 unsorted bin

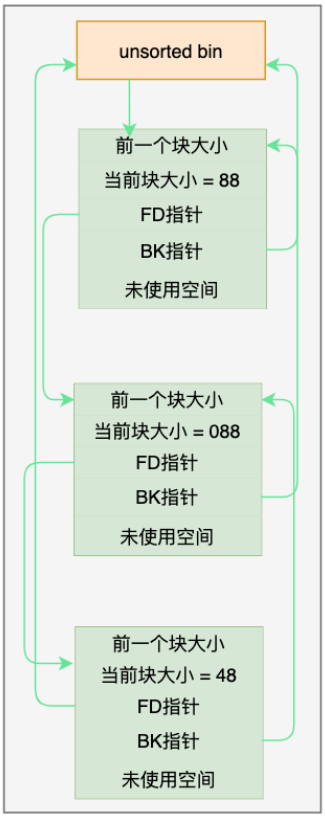
unsorted bin 的队列使用 bins 数组的第一个，是bins的一个缓冲区，加快分配的速度。当用户释放的内存大于max_fast或者fast bins合并后的chunk都会首先进入unsorted bin上。

在unsorted bin中，chunk的size 没有限制，也就是说任何大小chunk都可以放进unsorted bin中。这主要是为了让“glibc malloc机制”能够有第二次机会重新利用最近释放的chunk(第一次机会就是fast bin机制)。利用unsorted bin，可以加快内存的分配和释放操作，因为整个操作都不再需要花费额外的时间去查找合适的bin了。

用户malloc时，如果在 fast bins 中没有找到合适的 chunk,则malloc 会先在 unsorted bin 中查找合适的空闲 chunk，如果没有合适的bin，ptmalloc会将 unsorted bin上的chunk放入bins上，然后到bins上查找合适的空闲chunk。

与fast bin所不同的是，unsortedbin采用的遍历顺序是FIFO。

unsorted bin结构图如下：



5.3.3 small bin

大小小于512字节的chunk被称为small chunk，而保存small chunks的bin被称为 small bin。数组从2开始编号，前62个bin为small bins，small bin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。

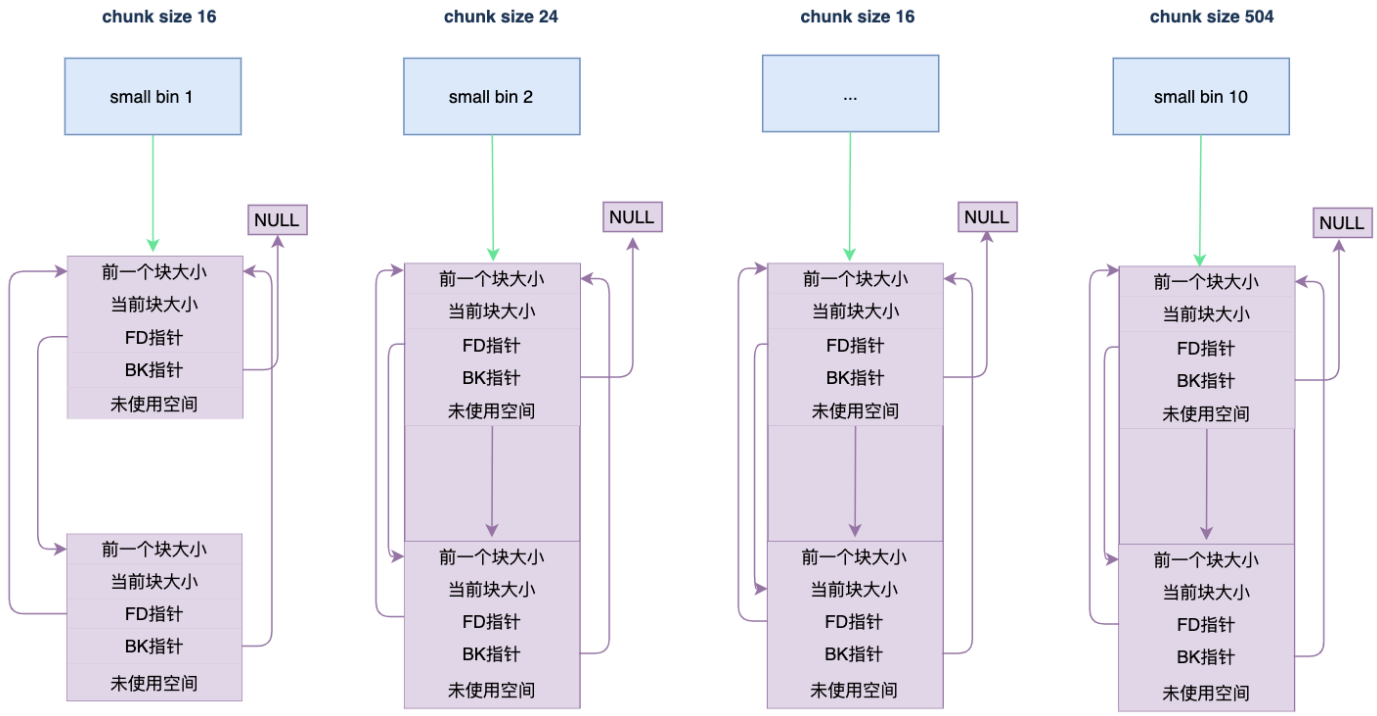
每个small bin都包括一个空闲区块的双向循环链表（也称binlist）。free掉的chunk添加在链表的前端，而所需chunk则从链表后端摘除。

两个毗连的空闲chunk会被合并成一个空闲chunk。合并消除了碎片化的影响但是减慢了free的速度。

分配时，当samll bin非空后，相应的bin会摘除binlist中最后一个chunk并返回给用户。

在free一个chunk的时候，检查其前或其后的chunk是否空闲，若是则合并，也即把它们从所属的链表中摘除并合并成一个新的chunk，新chunk会添加在unsorted bin链表的前端。

small bin也采用的是FIFO算法，即内存释放操作就将新释放的chunk添加到链表的front end(前端)，分配操作就从链表的rear end(尾端)中获取chunk。



5.3.4 large bin

大小大于等于512字节的chunk被称为large chunk，而保存large chunks的bin被称为large bin，位于small bins后面。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按大小递减排序，大小相同则按照最近使用时间排列。

两个毗连的空闲chunk会被合并成一个空闲chunk。

small bins 的策略非常适合小分配，但我们不能为每个可能的块大小都有一个 bin。对于超过 512 字节（64 位为 1024 字节）的块，堆管理器改为使用“large bin”。

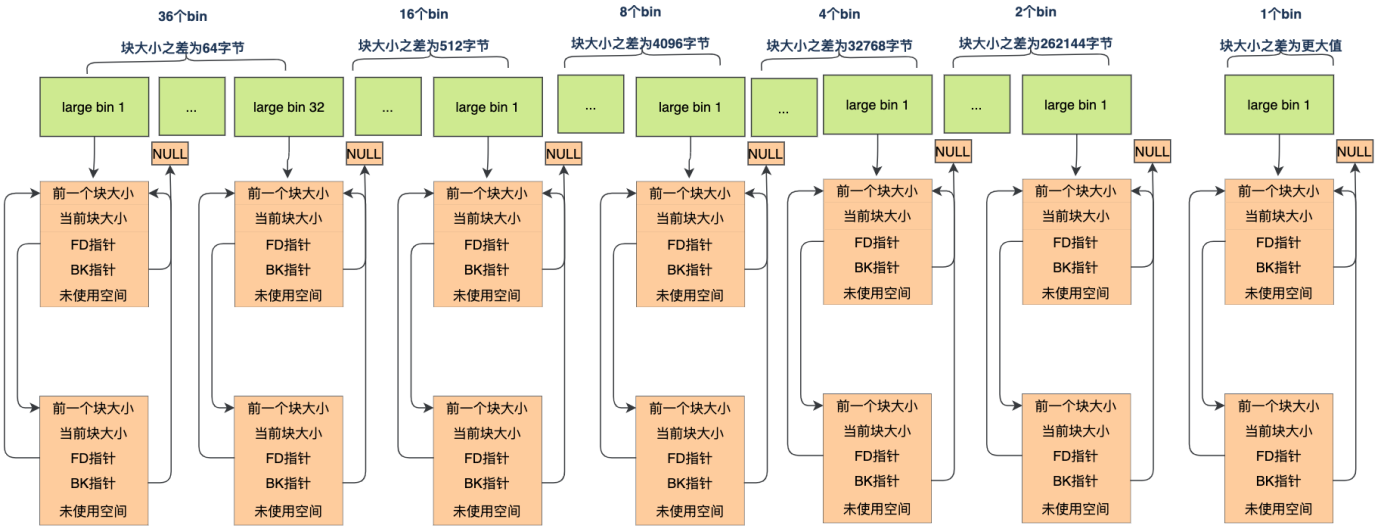
63个large bin中的每一个都与small bin的操作方式大致相同，但不是存储固定大小的块，而是存储大小范围内的块。每个large bin 的大小范围都设计为不与small bin 的块大小或其他large bin 的范围重叠。换句话说，给定一个块的大小，这个大小对应的正好是一个small bin或large bin。

在这63个largebins中：第一组的32个largebin链依次以64字节步长为间隔，即第一个largebin链中chunksize为1024-1087字节，第二个large bin中chunk size为1088~1151字节。第二组的16个largebin链依次以512字节步长为间隔；第三组的8个largebin链以步长4096为间隔；第四组的4个largebin链以32768字节为间隔；第五组的2个largebin链以262144字节为间隔；最后一组的largebin链中的chunk大小无限制。

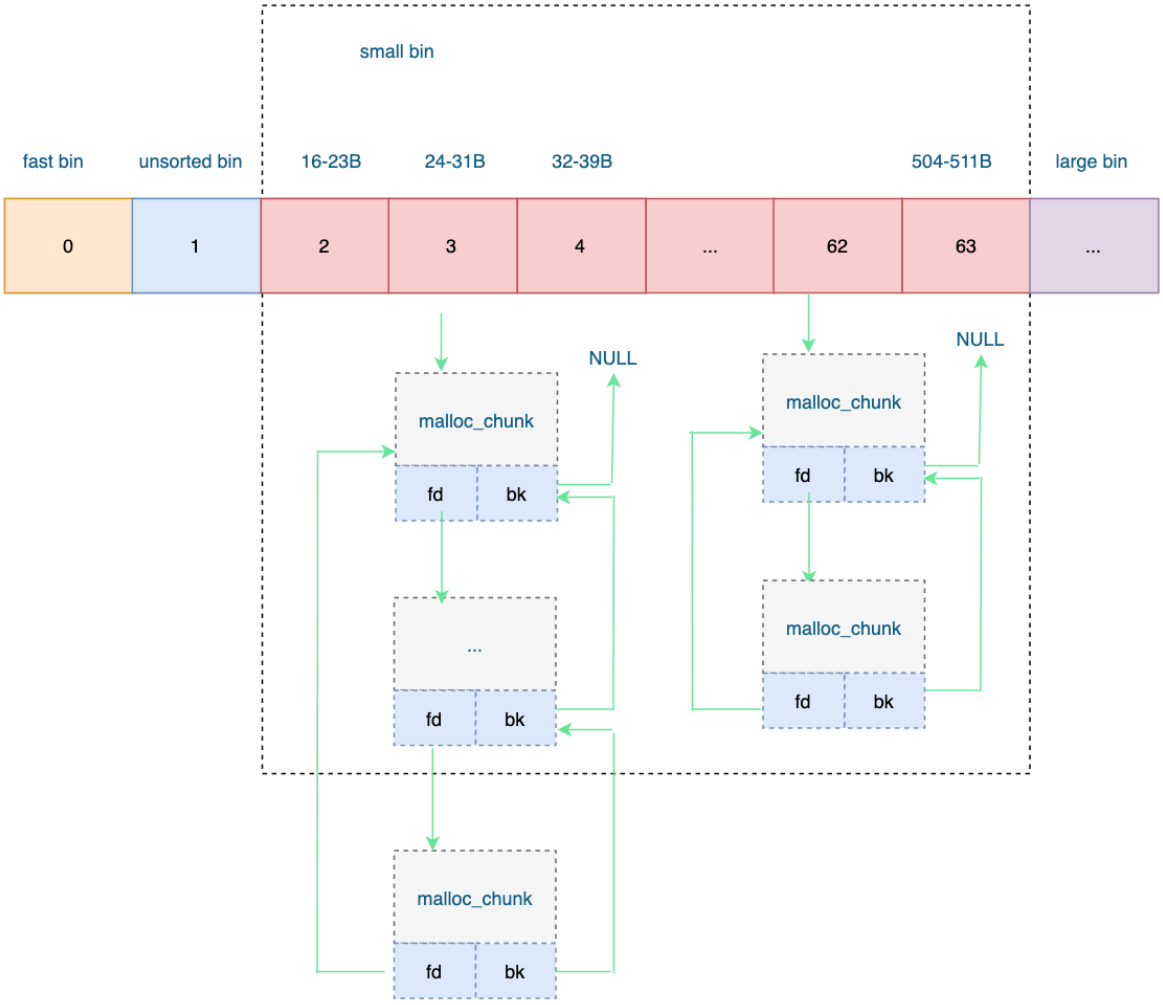
在进行malloc操作的时候，首先确定用户请求的大小属于哪一个large bin，然后判断该large bin中最大的chunk的size是否大于用户请求的size。如果大于，就从尾开始遍历该large bin，找到第一个size相等或接近的chunk，分配给用户。如果该chunk大于用户请求的size的话，就将该chunk拆分为两个chunk：前者返回给用户，且size等同于用户请求的size；剩余的部分做为一个新的chunk添加到unsorted bin中。

如果该large bin中最大的chunk的size小于用户请求的size的话，那么就依次查看后续的large bin中是否有满足需求的chunk，不过需要注意的是鉴于bin的个数较多(不同bin中的chunk极有可能在不同的内存页中)，如果按照上一段中介绍的方法进行遍历的话(即遍历每个bin中的chunk)，就可能会发生多次内存页中断操作，进而严重影响检索速度，所以glibc malloc设计了Binmap结构体来帮助提高bin-by-bin检索的速度。Binmap记录了各个bin中是否为空，通过bitmap可以避免检索一些空的bin。如果通过binmap找到了下一个非空的large bin的话，就按照上一段中的方法分配chunk，否则就使用top chunk（在后面有讲）来分配合适的内存。

large bin的free 操作与small bin一致，此处不再赘述。



上述几种bin，组成了进程中最核心的分配部分：bins，如下图所示：



;" />

5.4 特殊chunk

上节内容讲述了几种bin以及各种bin内存的分配和释放特点，但是，仅仅上面几种bin还不能够满足，比如假如上述bins不能满足分配条件的时候，glibc提出了另外几种特殊的chunk供分配和释放，分别为top chunk，mmaped chunk 和last remainder chunk。

5.4.1 top trunk

top chunk是堆最上面的一段空间，它不属于任何bin，当所有的bin都无法满足分配要求时，就要从这块区域里来分配，分配的空间返给用户，剩余部分形成新的top chunk，如果top chunk的空间也不满足用户的请求，就要使用brk或者mmap来向系统申请更多的堆空间（主分配区使用brk、sbrk，非主分配区使用mmap）。

在free chunk的时候，如果chunk size不属于fastbin的范围，就要考虑是不是和top chunk挨着，如果挨着，就要merge到top chunk中。

5.4.2 mmaped chunk

当分配的内存非常大（大于分配阀值，默认128K）的时候，需要被mmap映射，则会放到mmaped chunk上，当释放mmaped chunk上的内存的时候会直接交还给操作系统。（chunk中的M标志位置1）

5.4.3 last remainder chunk

Last remainder chunk是另外一种特殊的chunk，这个特殊chunk是被维护在unsorted bin中的。
如果用户申请的size属于small bin的，但是又不能精确匹配的情况下，这时候采用最佳匹配（比如申请128字节，但是对应的bin是空，只有256字节的bin非空，这时候就要从256字节的bin上分配），这样会split chunk成两部分，一部分返给用户，另一部分形成last remainder chunk，插入到unsorted bin中。

当需要分配一个small chunk,但在small bins中找不到合适的chunk，如果last remainder chunk的大小大于所需要的small chunk大小，last remainder chunk被分裂成两个chunk，其中一个chunk返回给用户，另一个chunk变成新的last remainder chunk。

last remainder chunk主要通过提高内存分配的局部性来提高连续malloc（产生大量small chunk）的效率。

5.5 chunk 切分

chunk释放时，其长度不属于fastbins的范围，则合并前后相邻的chunk。

首次分配的长度在large bin的范围，并且fast bins中有空闲chunk，则将fastbins中的chunk与相邻空闲的chunk进行合并，然后将合并后的chunk放到unsorted bin中，如果fastbin中的chunk相邻的chunk并非空闲无法合并，仍旧将该chunk放到unsorted bin中，即能合并的话就进行合并，但最终都会放到unsorted bin中。

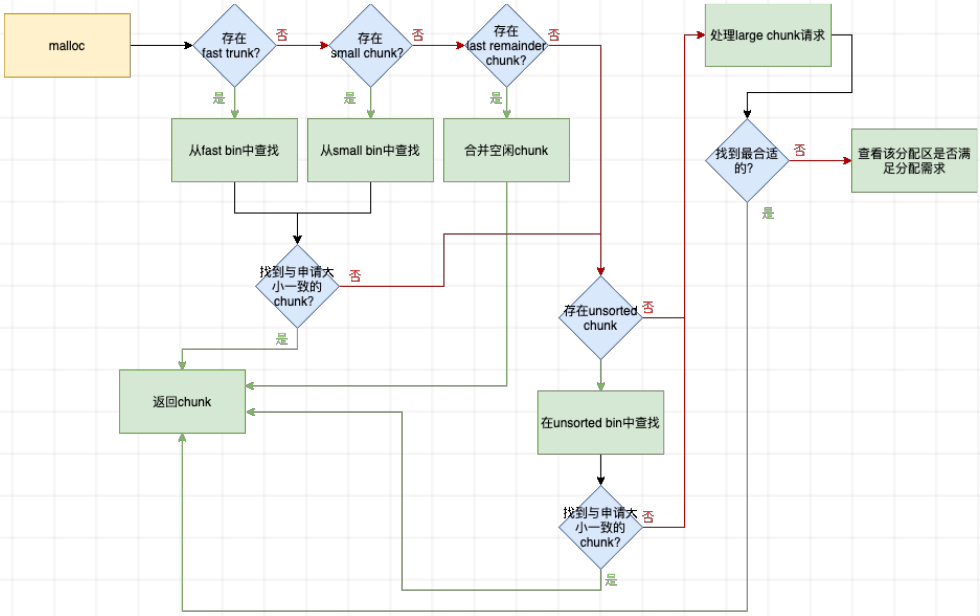
fastbins，small bin中都没有合适的chunk，top chunk的长度也不能满足需要，则对fast bin中的chunk进行合并。

5.6 chunk 合并

前面讲了相邻的chunk可以合并成一个大的chunk，反过来，一个大的chunk也可以分裂成两个小的chunk。chunk的分裂与从top chunk中分配新的chunk是一样的。需要注意的一点是：分裂后的两个chunk其长度必须均大于chunk的最小长度（对于64位系统是32字节），即保证分裂后的两个chunk仍旧是可以被分配使用的，否则则不进行分裂，而是将整个chunk返回给用户。

6 内存分配(malloc)

glibc运行时库分配动态内存，底层用的是malloc来实现(new 最终也是调用malloc)，下面是malloc函数调用流程图：



在此，将上述流程图以文字形式表示出来，以方便大家理解：

1、获取分配区的锁，为了防止多个线程同时访问同一个分配区，在进行分配之前需要取得分配区域的锁。线程先查看线程私有实例中是否已经存在一个分配区，如果存在尝试对该分配区加锁，如果加锁成功，使用该分配区分配内存，否则，该线程搜索分配区循环链表试图获得一个空闲（没有加锁）的分配区。如果所有的分配区都已经加锁，那么 ptmalloc 会开辟一个新的分配区，把该分配区加入到全局分配区循环链表和线程的私有实例中并加锁，然后使用该分配区进行分配操作。开辟出来的新分配区一定为非主分配区，因为主分配区是从父进程那里继承来的。开辟非主分配区时会调用 mmap()创建一个 sub-heap，并设置好 top chunk。

2、将用户的请求大小转换为实际需要分配的 chunk 空间大小。

3、判断所需分配chunk 的大小是否满足chunk_size <= max_fast (max_fast 默认为 64B)， 如果是的话，则转下一步，否则跳到第 5 步。

4、首先尝试在 fast bins 中取一个所需大小的 chunk 分配给用户。如果可以找到，则分配结束。否则转到下一步。

5、判断所需大小是否处在 small bins 中，即判断 chunk_size < 512B 是否成立。如果chunk 大小处在 small bins 中，则转下一步，否则转到第 6 步。

6、根据所需分配的 chunk 的大小，找到具体所在的某个 small bin，从该 bin 的尾部摘取一个恰好满足大小的 chunk。若成功，则分配结束，否则，转到下一步。

7、到了这一步，说明需要分配的是一块大的内存，或者 small bins 中找不到合适的 chunk。于是，ptmalloc 首先会遍历 fast bins 中的 chunk，将相邻的 chunk 进行合并，并链接到 unsorted bin 中，然后遍历 unsorted bin 中的 chunk，如果 unsorted bin 只有一个 chunk，并且这个 chunk 在上次分配时被使用过，并且所需分配的 chunk 大小属于 small bins，并且 chunk 的大小大于等于需要分配的大小，这种情况下就直接将该 chunk 进行切割，分配结束，否则将根据 chunk 的空间大小将其放入 small bins 或是 large bins 中，遍历完成后，转入下一步。

8、到了这一步，说明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，并且 fast bins 和 unsorted bin 中所有的 chunk 都清除干净了。从 large bins 中按照“smallest-first，best-fit”原则，找一个合适的 chunk，从中划分一块所需大小的 chunk，并将剩下的部分链接回到 bins 中。若操作成功，则分配结束，否则转到下一步。

9、如果搜索 fast bins 和 bins 都没有找到合适的 chunk，那么就需要操作 top chunk 来进行分配了。判断 top chunk 大小是否满足所需 chunk 的大小，如果是，则从 top chunk 中分出一块来。否则转到下一步。

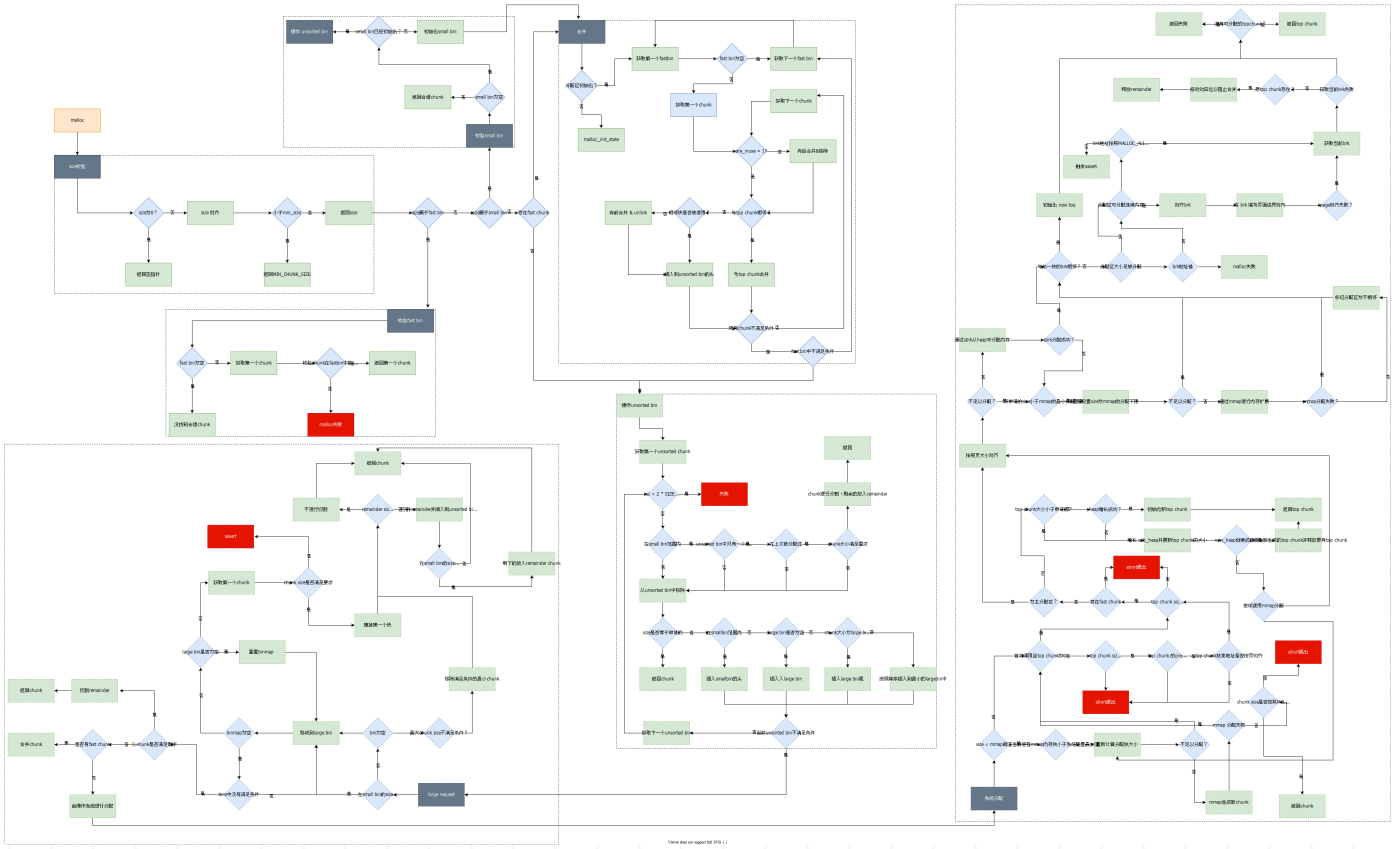
10、到了这一步，说明 top chunk 也不能满足分配要求，所以，于是就有了两个选择: 如果是主分配区，调用 sbrk()，增加 top chunk 大小；如果是非主分配区，调用 mmap 来分配一个新的 sub-heap，增加 top chunk 大小；或者使用 mmap()来直接分配。在这里，需要依靠 chunk 的大小来决定到底使用哪种方法。判断所需分配的 chunk 大小是否大于等于 mmap 分配阈值，如果是的话，则转下一步，调用 mmap 分配， 否则跳到第 12 步，增加 top chunk 的大小。

11、使用 mmap 系统调用为程序的内存空间映射一块 chunk_size align 4kB 大小的空间。然后将内存指针返回给用户。

12、判断是否为第一次调用 malloc，若是主分配区，则需要进行一次初始化工作，分配一块大小为(chunk_size + 128KB) align 4KB 大小的空间作为初始的 heap。若已经初始化过了，主分配区则调用 sbrk()增加 heap 空间，分主分配区则在 top chunk 中切割出一个 chunk，使之满足分配需求，并将内存指针返回给用户。
将上面流程串起来就是：

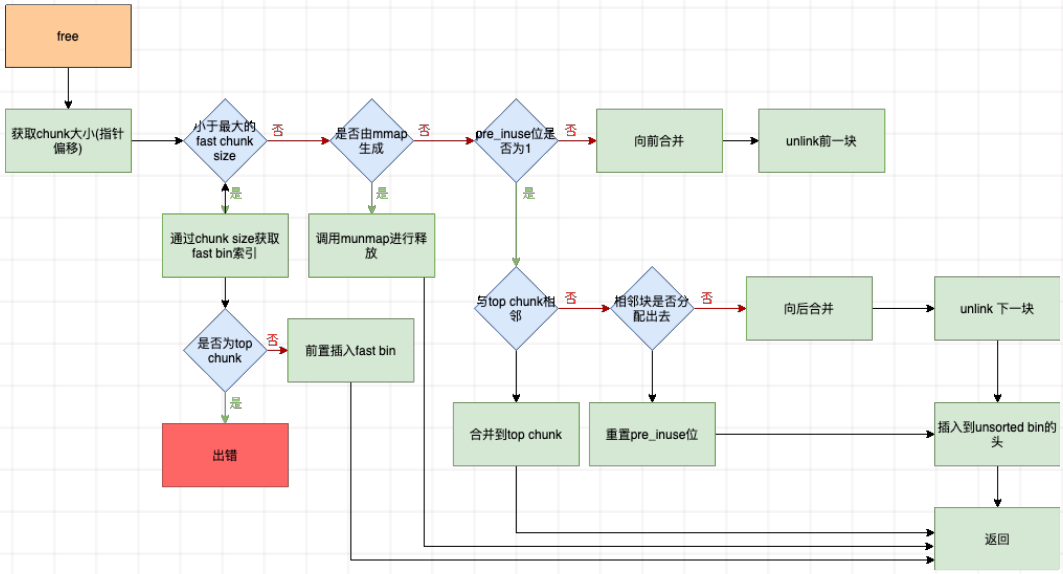
> 根据用户请求分配的内存的大小，ptmalloc有可能会在两个地方为用户分配内存空间。在第一次分配内存时，一般情况下只存在一个主分配区，但也有可能从父进程那里继承来了多个非主分配区，在这里主要讨论主分配区的情况，brk值等于start_brk，所以实际上heap大小为0，top chunk 大小也是0。这时，如果不增加heap大小，就不能满足任何分配要求。所以，若用户的请求的内存大小小于mmap分配阈值， 则ptmalloc会初始heap。然后在heap中分配空间给用户，以后的分配就基于这个heap进行。若第一次用户的请求就大于mmap分配阈值，则ptmalloc直接使用mmap()分配一块内存给用户，而heap也就没有被初始化，直到用户第一次请求小于mmap分配阈值的内存分配。第一次以后的分配就比较复杂了，简单说来，ptmalloc首先会查找fast bins，如果不能找到匹配的chunk，则查找small bins。若仍然不满足要求，则合并fast bins，把chunk加入unsorted bin，在unsorted bin中查找，若仍然不满足要求，把unsorted bin 中的chunk全加入large bins 中，并查找large bins。在fast bins 和small bins中的查找都需要精确匹配， 而在large bins 中查找时，则遵循“smallest-first，best-fit”的原则，不需要精确匹配。若以上方法都失败了，则ptmalloc会考虑使用top chunk。若top chunk也不能满足分配要求。而且所需chunk大小大于mmap分配阈值，则使用mmap进行分配。否则增加heap，增大top chunk。以满足分配要求。

当然了，glibc中malloc的分配远比上面的要复杂的多，要考虑到各种情况，比如指针异常越界等，将这些判断条件也加入到流程图中，如下图所示：



7 内存释放(free)

malloc进行内存分配，那么与malloc相对的就是free，进行内存释放，下面是free函数的基本流程图：



对上述流程图进行描述，如下：

1、 获取分配区的锁，保证线程安全。

2、如果free的是空指针，则返回，什么都不做。

3、判断当前chunk是否是mmap映射区域映射的内存，如果是，则直接munmap()释放这块内存。前面的已使用chunk的数据结构中，我们可以看到有M来标识是否是mmap映射的内存。

4、 判断chunk是否与top chunk相邻，如果相邻，则直接和top chunk合并（和top chunk相邻相当于和分配区中的空闲内存块相邻）。否则，转到步骤8

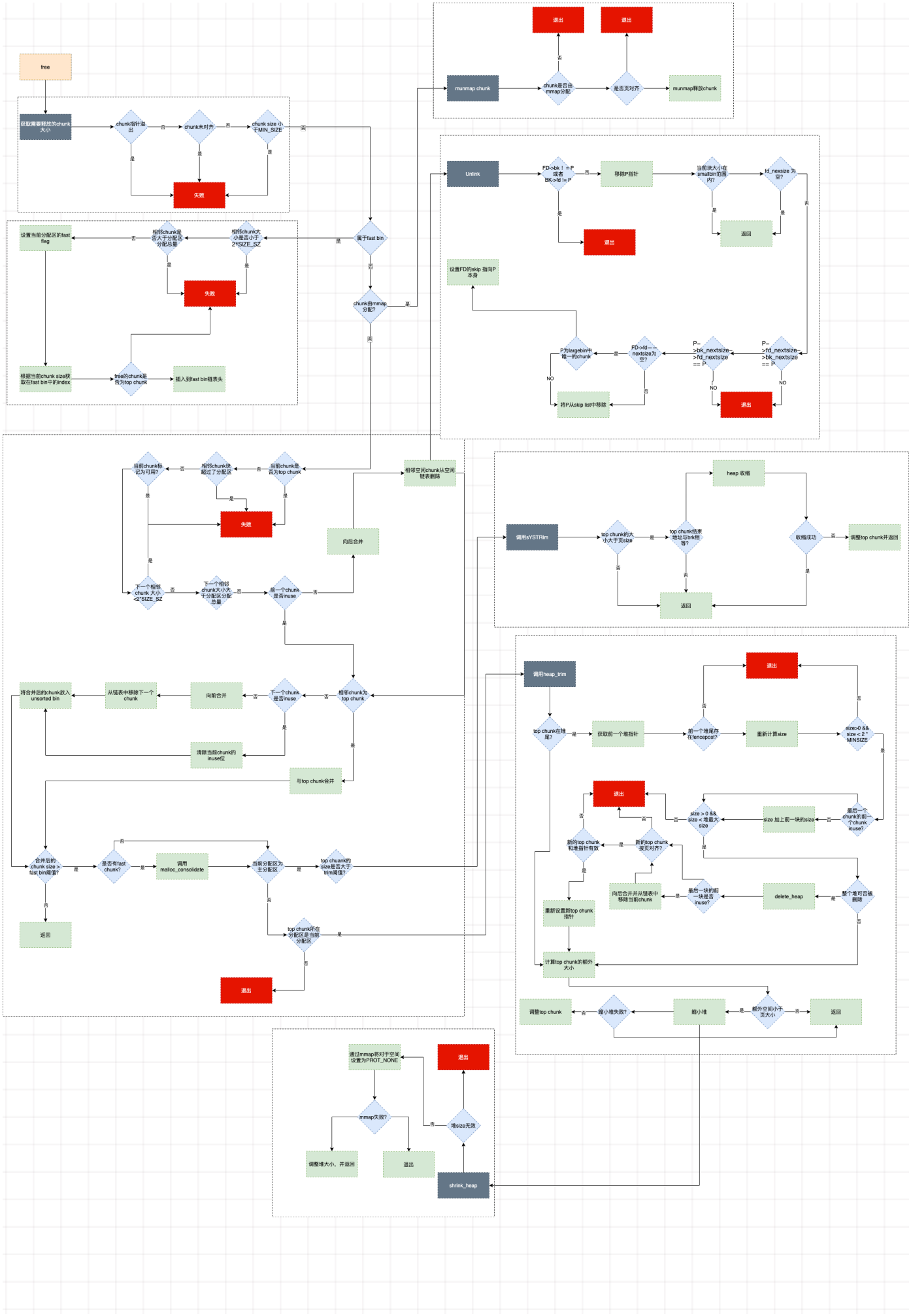
5、如果chunk的大小大于max_fast（64b），则放入unsorted bin，并且检查是否有合并，有合并情况并且和top chunk相邻，则转到步骤8；没有合并情况则free。

6、如果chunk的大小小于 max_fast（64b），则直接放入fast bin，fast bin并没有改变chunk的状态。没有合并情况，则free；有合并情况，转到步骤7

7、在fast bin，如果当前chunk的下一个chunk也是空闲的，则将这两个chunk合并，放入unsorted bin上面。合并后的大小如果大于64B，会触发进行fast bins的合并操作，fast bins中的chunk将被遍历，并与相邻的空闲chunk进行合并，合并后的chunk会被放到unsorted bin中，fast bin会变为空。合并后的chunk和topchunk相邻，则会合并到topchunk中。转到步骤8

8、判断top chunk的大小是否大于mmap收缩阈值（默认为128KB），如果是的话，对于主分配区，则会试图归还top chunk中的一部分给操作系统。free结束。

如果将free函数内部各种条件加入进去，那么free调用的详细流程图如下：


free
获取需要释放的chunk大小
chunk指针溢出
chunk未对齐
chunk size 小于MIN_SIZE
失败
munmap chunk
chunk是否由mmap分配
是否页对齐
munmap释放chunk
退出
退出
Unlink
FD->bk ！= P 或者 BK->fd != P
移除P指针
当前块大小在smallbin范围内?
fd_nexsize 为空?
返回
退出
设置FD的skip 指向P本身
P为largebin中唯一的chunk
FD->fd->nextsize为空?
P->bk_nextsize->fd_nextsize == P
P->fd_nextsize->bk_nextsize == P
将P从skip list中移除
退出
属于fast bin
相邻chunk大小是否小于2*SIZE_SZ
相邻chunk是否大于分配区分配总量
失败
设置当前分配区的fast flag
根据当前chunk size获取在fast bin中的index
tree的chunk是否为top chunk
插入到fast bin链表头
chunk由mmap分配?
相邻空闲chunk从空闲链表删除
当前chunk标记为可用?
相邻chunk块超过了分配区
当前chunk是否为top chunk
失败
向后合并
下一个相邻chunk 大小 <2*SIZE_SZ
下一个相邻chunk大小大于分配区分配总量
前一个chunk是否inuse
调用sYSTRIm
top chunk的大小大于页size
top chunk结束地址与brk相等?
heap 收缩
收缩成功
调整top chunk并返回
返回
调用heap_trim
top chunk在堆尾?
获取前一个堆指针
前一个堆尾存在fencepost?
重新计算size
size>0 && size < 2 * MINSIZE
退出
size > 0 && size < 堆最大size
size 加上前一块的size
最后一个chunk的前一个chunk inuse?
新的top chunk和堆指针有效
新的top chunk按页对齐?
向后合并并从链表中移除当前chunk
最后一块的前一块是否inuse?
delete_heap
整个堆可否被删除
重新设置新top chunk指针
计算top chunk的额外大小
调整top chunk
缩小堆失败?
缩小堆
额外空间小于页大小
返回
相邻chunk为top chunk
从链表中移除下一个chunk
向前合并
下一个chunk是否inuse
将合并后的chunk放入unsorted bin
清除当前chunk的inuse位
与top chunk合并
合并后的chunk size > fast bin阈值?
是否有fast chunk?
调用malloc_consolidate
当前分配区为主分配区
top chuank的size是否大于trim阈值?
返回
top chunk所在分配区是当前分配区
退出
通过mmap将对于空间设置为PROT_NONE
mmap失败?
堆size无效
退出
调整堆大小，并返回
退出
shrink_heap

8 问题分析以及解决

通过前面对glibc运行时库的分析，基本就能定位出原因，是因为我们调用了free进行释放，但仅仅是将内存返还给了glibc库，而glibc库却没有将内存归还操作系统，最终导致系统内存耗尽，程序因为 OOM 被系统杀掉。

有以下两种方案：

- 禁用 ptmalloc 的 mmap 分配阈值动态调整机制。通过 mallopt() 设置M_TRIM_THRESHOLD，M_MMAP_THRESHOLD，M_TOP_PAD 和 M_MMAP_MAX 中的任意一个，关闭 mmap 分配阈值动态调整机制，同时需要将 mmap 分配阈值设置为64K，大于 64K 的内存分配都使用mmap 向系统分配，释放大于 64K 的内存将调用 munmap 释放回系统。但是，这种方案的 *缺点是每次内存分配和申请，都是直接向操作系统申请，效率低* 。
- 预估程序可以使用的最大物理内存大小，配置系统的 /proc/sys/vm/overcommit_memory，/proc/sys/vm/overcommit_ratio，以及使用 ulimt –v限制程序能使用虚拟内存空间大小，防止程序因 OOM 被杀掉。这种方案的 *缺点是如果预估的内存小于进程实际占用，那么仍然会出现OOM，导致进程被杀掉* 。
- tcmalloc

最终采用tcmalloc来解决了问题，后面有机会的话，会写一篇tcmalloc内存管理的相关文章。

9 结语

业界语句说法，是否了解内存管理机制，是辨别C/C++程序员和其他的高级语言程序员的重要区别。作为C/C++中的最重要的特性，指针及动态内存管理在给编程带来极大的灵活性的同时，也给开发人员带来了许多困扰。

了解底层内存实现，有时候会有意想不到的效果哦。

先写到这里吧。

> 本文提供了PDF版本，关注本公众号(高性能架构探索)，后台回复[malloc]获取，另外还有批量电子书，后台回复[pdf]免费获取。

10 参考

1、https://sourceware.org/glibc/wiki/MallocInternals

2、https://titanwolf.org/Network/Articles/Article?AID=99524c69-cb90-4d61-bb28-01c0864d0ccc

3、https://blog.fearcat.in/a?ID=00100-535db575-0d98-4287-92b6-4d7d9604b216

4、https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

5、https://sploitfun.wordpress.com/2015/02/11/syscalls-used-by-malloc

---