内存池技术

内存池intro

为什么需要内存池

内存池是一种管理计算机内存的技术，它通过预先分配一块连续的内存块，并在需要时动态地将其分配给程序使用。以下是需要使用内存池的几个主要原因：

1. 提高内存分配效率：传统的内存分配操作（如malloc和free）需要频繁地向操作系统申请和释放内存，**系统调用涉及到较高的系统调用开销**。内存池通过事先分配好一块内存区域，避免了频繁的系统调用，提高了内存分配和释放的效率
2. 减少内存碎片：在使用传统的内存分配方式时，频繁的内存分配和释放可能导致内存碎片的产生
    - 内存碎片是指内存空间中被分割成小块而无法被充分利用的情况，它会降低内存的利用率并增加内存分配的复杂度。内存碎片的程度主要取决于内存对齐和MMU的分页机制
    - 内存池可以减少内存碎片的产生，因为它们分配的内存块大小是固定的，不会产生碎片化的问题
3. 控制内存分配的策略：使用内存池可以更好地控制内存分配的策略，例如可以选择预分配一定数量的内存块，避免内存不足的情况发生，或者可以根据需求动态地扩展内存池的大小。这种控制可以提高程序的性能和稳定性
4. 适用于特定的应用场景：某些应用程序或算法对内存的需求是连续且频繁的，传统的内存分配方式可能无法满足其性能要求。内存池可以在这些特定场景下提供更高效的内存管理，比如STL库的空间配置器就是内存池

总的来说，内存池的主要目的是提高内存分配和释放的效率，减少内存碎片，并在特定的应用场景下提供更好的性能和稳定性

内存池的层次

- 系统层/C风格：使用高性能内存管理组件 tcmalloc、jemalloc，具体来说就是用这两种动态链接库接管 glibc 的 ptmalloc 实现
- 应用层：根据应用自己定制，比如说Nginx定制的内存池，还有STL库的空间配置器（STL数据结构的内存池）也是封装了malloc和free

ptmalloc、jemalloc和tcmalloc对比

https://www.kandaoni.com/news/22268.html

	PtMalloc（glibc自带）	TcMalloc	JeMalloc
开发者	glibc 自带	Google 开源	Jason Evans（FreeBSD 著名开发人员）
性能(一次 malloc/free 操作)	300ns	50ns	<=50ns
弊端	锁机制降低性能，容易导致内存碎片	1%左右的额外内存开销	2%左右的额外内存开销
优点	传统，稳定	线程本地缓存，多线程分配效率高	线程本地缓存，多核多线程分配效率相当高
使用方式	glibc 编译	动态链接库	动态链接库
使用者	较普遍	safari、chrome等	facebook、firefox 等
适用场景	通用的内存管理，除特别追求高效内存分配以外的	多线程下高效内存分配	多线程下高效内存分配

ptmalloc

ptmalloc, Per-Thread Memory Allocator 是glibc默认的内存管理系统，是解决通用场景下默认的glibc内存管理系统。起源于Doug Lea的malloc。由 Wolfram Gloger 改进得到可以支持多线程

它在应对当今的多核高并发等压力环境下存在如下问题

- 高并发时较小内存块使用导致系统调用频繁，降低了系统的执行效率
- 频繁使用时增加了系统内存的碎片，降低内存使用效率
- 缺乏垃圾回收机制：容易造成内存泄漏，导致内存枯竭
- 内存分配与释放的逻辑在程序中相隔较远时，降低程序的稳定性

tcmalloc

ptmalloc的瓶颈在于锁竞争

tcmalloc实现了TLS级别的无锁

jemalloc

jemalloc 是由 Jason Evans 在 FreeBSD 项目中引入的新一代内存分配器。它是一个通用的 malloc 实现，侧重于减少内存碎片和提升高并发场景下内存的分配效率，其目标是能够替代 malloc。jemalloc 在 2005 年首次作为 FreeBSD libc 分配器使用，2010年，jemalloc 的功能延伸到如堆分析和监控/调优等。现代的 jemalloc 版本依然集成在 FreeBSD 中。jemalloc目前在firefox、facebook服务器各种组件中大量使用

为每个进程单独申请一个小的内存池，专门存放请求、解析的数据等。响应结束后统一删除。这个过程不需要很多的系统调用

协存池生存时间应该尽可能短，与请求或者连接具有相同的周期，减少小内存池的大量堆叠，减少碎片堆积和内存泄漏

Teaser：定长内存池

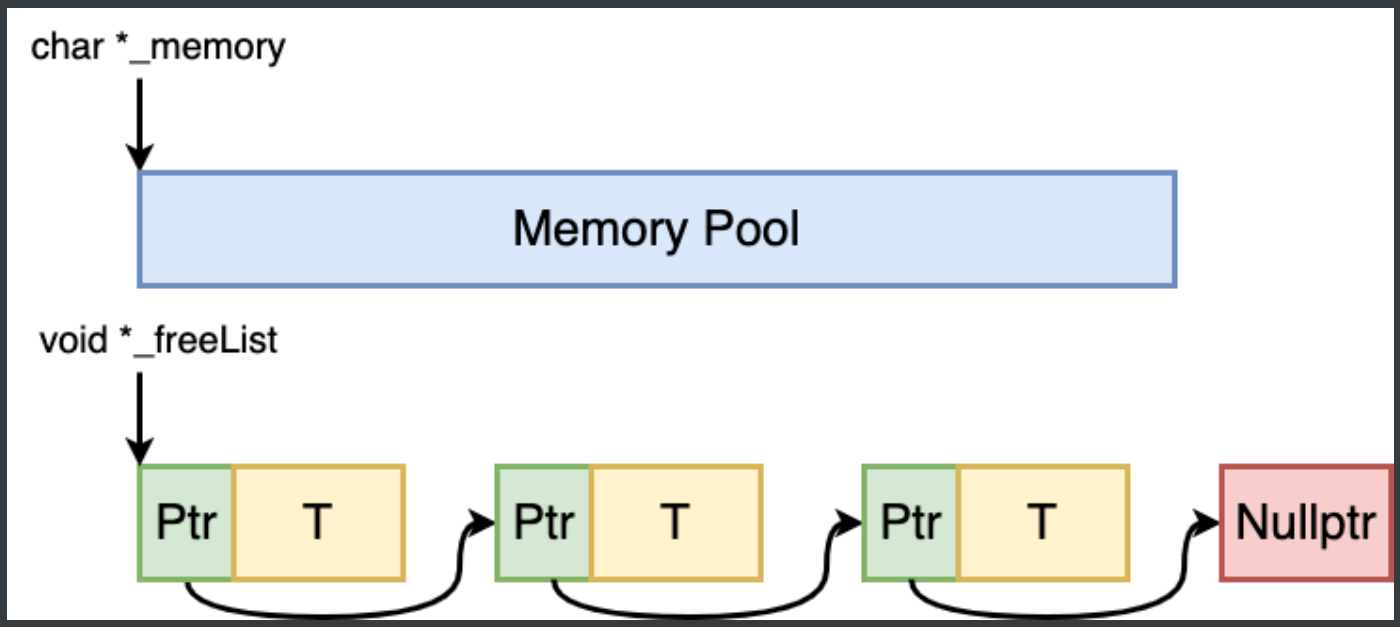
intro

定长对象的内存池，每次申请或者归还一个固定大小的内存对象T，定长内存池可以满足固定大小的内存申请释放需求，并且定长内存池在高并发内存池中可以被复用

定长内存池的特点是

- 性能达到极致
- 不考虑内存碎片等问题

实现



用自由链表 `void *_freeList` 来管理切好的小块内存，内存块头上存下一个内存块的地址，因此一个内存块至少要存4字节（32位）或8字节（64位）

```
// 获取内存对象中存储的头4 or 8字节值，即连接的下一个对象的地址
inline void*& NextObj(void* obj) {
    return *((void**)obj);
}
```

`void **` 在32位下是4 Byte，64位下是8 Byte。先将 `void *` 强转为 `void **` 对它再解引用一次的时候就可以得到 `void *` 的内容，这和对 `void *` 解引用得到 `void` 的效果是一样的。如果不这么做，就要通过32位用 `int *` 解引用和 64位用 `long *` 解引用来分别了，这很不灵活

可以用这种方式来判断当前系统是32位还是64位，然后强转

```
if (sizeof(int *) == 32) //32位
    *((int *)obj) = nullptr;
else //64位
    *((long long*)obj) = nullptr;
```

若剩余的空间不够分配一块内存了该怎么办？引入 `_remainByte` 管理

内存分配系统调用

Linux 的系统调用 brk、mmap、alloca 等查看 *系统编程.md*，这里介绍一下 Windows 下的内存分配系统调用

Windows 的 VirtualAlloc 和 VirtualFree 是用于动态内存管理的函数，常用于内存分配和释放的操作

https://blog.csdn.net/asce1885/article/details/5707155

VirtualAlloc

```
LPVOID WINAPI VirtualAlloc(
  __in_opt  LPVOID lpAddress, //分配的起始位置。
//如果要保留一段内存区域，函数会将其自动向最近的一个分配粒度对齐；
//如果要提交一段内存区域，函数将会向最近的一个页面对齐；
//如果为NULL，系统自行决定在什么地方分配
  __in      SIZE_T dwSize, //所需要分配的内存字节大小，
  __in      DWORD flAllocationType, //分配类型：MEM_COMMIT(提交)、MEM_RESERVED(保留)..
  __in      DWORD flProtect //内存保护属性：PAGE_READWRITE、PAGE_EXECUTE…
);
```

VirtualFree

```
BOOL WINAPI VirtualFree(
  __in  LPVOID lpAddress, //需要改变状态的内存区域的起始地址
  __in  SIZE_T dwSize, //需要改变状态的内存区域的字节大小
  __in  DWORD dwFreeType //MEM_DECOMMIT-将内存变为保留状态
                         //MEM_RELEASE-释放内存，将内存变为空闲状态
);
```

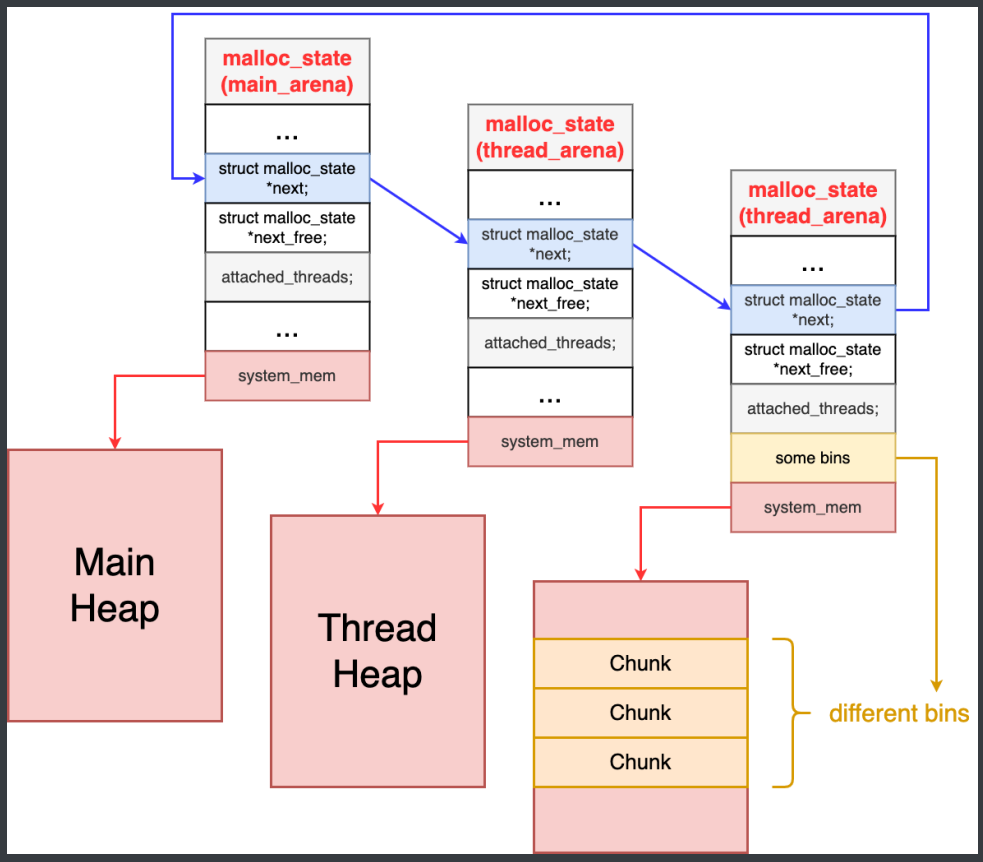
ptmalloc

https://blog.csdn.net/initphp/category_10542242.html

以下分析基于 glibc-2.31 中集成的 ptmalloc2，从https://ftp.gnu.org/gnu/glibc/这里下载

架构 & 核心数据结构

架构



- `malloc_state` (Arena header)：一个 thread arena 可以维护多个堆，这些堆共享同一个 arena header。Arena header 描述的信息包括：bins、top chunk、last remainder chunk 等
- `heap_info` (Heap Header)：每个堆都有自己的堆 Header（注：也即头部元数据）。当这个堆的空间耗尽时，新的堆（而非连续内存区域）就会被 mmap 当前堆的 arena 里
- `malloc_chunk` (Chunk header)：根据用户请求，每个堆被分为若干 chunk。每个 chunk 都有自己的 chunk header。内存管理使用malloc_chunk，把heap当作link list从一个内存块游走到下一个块

malloc_state

malloc_state是一个进程中全局的数据结构，每一个分配区都有一个（包括主分配区和非主分配区），用来组织、管理所有顶层的分配区

```
struct malloc_state {
  __libc_lock_define (, mutex);
  int flags;
  int have_fastchunks;
  mfastbinptr fastbinsY[NFASTBINS];
  mchunkptr top;
  mchunkptr last_remainder;
  mchunkptr bins[NBINS * 2 - 2];
  unsigned int binmap[BINMAPSIZE];
  struct malloc_state *next;
  struct malloc_state *next_free;
  INTERNAL_SIZE_T attached_threads;
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};
```

- mutex：线程锁，当多线程进行内存分配竞争的时候，需要首先拿到该锁才能进行分配区上的操作
- flags 位图：记录了分配区的一些标志的位图，比如 bit0 记录了分配区是否有 fast bin chunk ，bit1 标识分配区是否能返回连续的虚拟地址空间
- have_fastchunks：用于标记是否有fast bins
- fastbinsY：fast bins是bins的高速缓冲区，大约有10个定长队列。当用户释放一块不大于max_fast（默认值64）的chunk（一般小内存）的时候，会默认会被放到fast bins上
- top：指向分配区的 top chunk。top chunk相当于分配区的顶部空闲内存，当bins上都不能满足内存分配要求的时候，就会来top chunk上分配
- last_remainder：最新的 chunk 分割之后剩下的那部分
- bins 指针数组：用于存储 unstored bin，small bins 和 large bins 的 chunk 链表
- binmap：ptmalloc 用一个 bit 来标识某一个 bin 中是否包含空闲 chunk

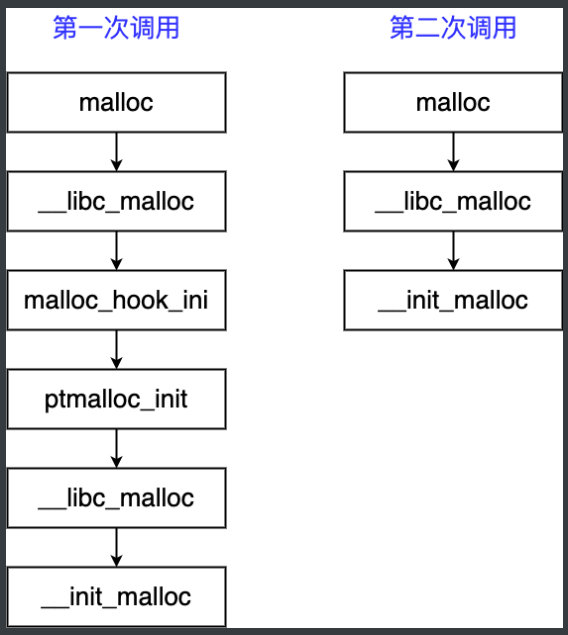
- next：分配区全局链表，主分配区放头部，新加入的分配区放main_arean.next 位置。
- next_free：空闲的分配区
- attached_threads：空闲链表的状态记录，0为空闲，n为正在使用中、关联的线程个数（一个分配区可以给多个线程使用）

初始化

ptmalloc的源码在 glibc/malloc 文件夹下，而malloc函数的入口在malloc/malloc.c文件中

```
1  strong_alias (__libc_malloc, __malloc) strong_alias (__libc_malloc,
   malloc)
```

没有直接的malloc函数，glibc通过 `__attribute__((alias))`，将 `__libc_malloc` 函数强绑定到malloc上



chunk

ptmalloc通过chunk 数据结构作为最小的内存单元，来进行内存管理

malloc_chunk

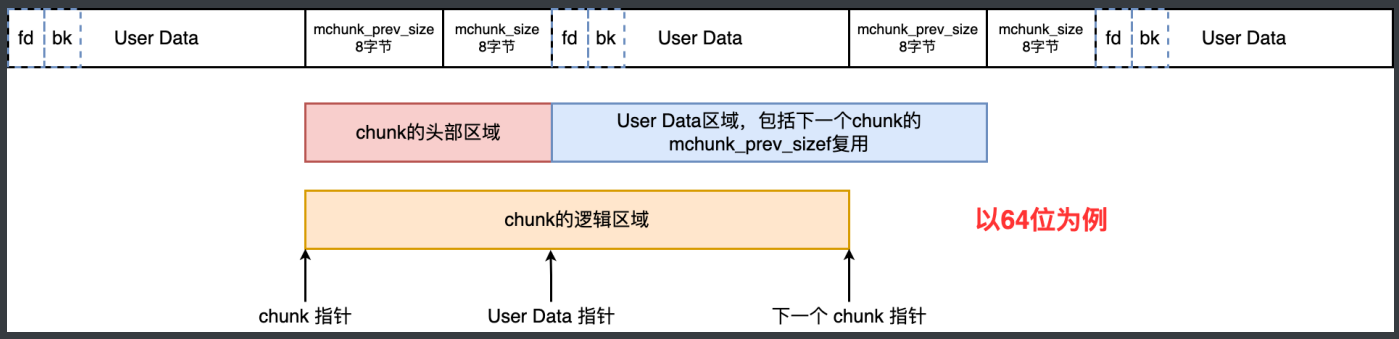
```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

  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;
};
```

- mchunk_prev_size：该字段记录物理相邻的前一个chunk的大小（低地址chunk）
  - 若前一个chunk处于空闲，则该字段记录前一个chunk大小
  - 如果前一个chunk已经被使用（allocated），则该字段空间可以被前一个chunk的用户数据空间复用
- mchunk_size：该字段是chunk的大小。该字段的低三个比特位对 chunk 的大小没有影响，所以被复用e为标志位
- fd和bk：当chunk空闲的时候，会放置到bins上双向链表管理。fd 指向下一个（非物理相邻）空闲的 chunk。bk 指向上一个（非物理相邻）空闲的 chunk。由于只有chunk空闲的时候，才会放置到bins上进行空闲管理，所以fd和bk占用的是用户数据区域user data
- fd_nextsize和bk_nextsize：用于管理large块的时候的空闲chunk双向链表的管理。一般空闲的 large chunk 在 fd 的遍历顺序中，按照由大到小的顺序排列。这样做可以避免在寻找合适 chunk 时挨个遍历，也是复用用户数据区域。large chunk的空间肯定装的下

使用中的状态

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             (size of chunk, but used for application data)    |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|1|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- 复用相邻下一个nextchunk（高地址chunk）的mchunk_prev_size字段的数据空间。所以该chunk的内存空间为当前内存 + mchunk_prev_size字段空间
- 由于chunk被使用中，所以不需要通过双向链表方式挂载到空闲bins上管理，fd和bk以及fd_nextsize和bk_nextsize不需要被使用

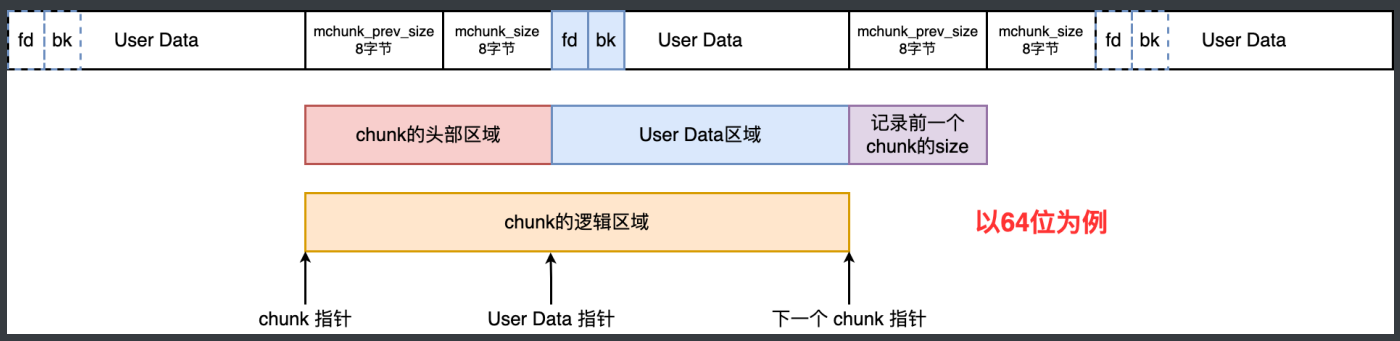


空闲时的状态

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `head:' |             Size of chunk, in bytes                     |A|0|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Forward pointer to next chunk in list             |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Back pointer to previous chunk in list            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Unused space (may be 0 bytes long)                .
            .                                                               .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
14      `foot:' |             Size of chunk, in bytes
          |
15          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    +-+-+
16          |             Size of next chunk, in bytes
    |A|0|0|
17          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    +-+-+
```

- fd/bk以及fd_nextsize/bk_nextsize的指针地址，可以直接侵占用户数据区域（userdata）的空间，因为这些用户数据已经无效了
- 下一个nextchunk的mchunk_prev_size值，记录了当前chunk的大小

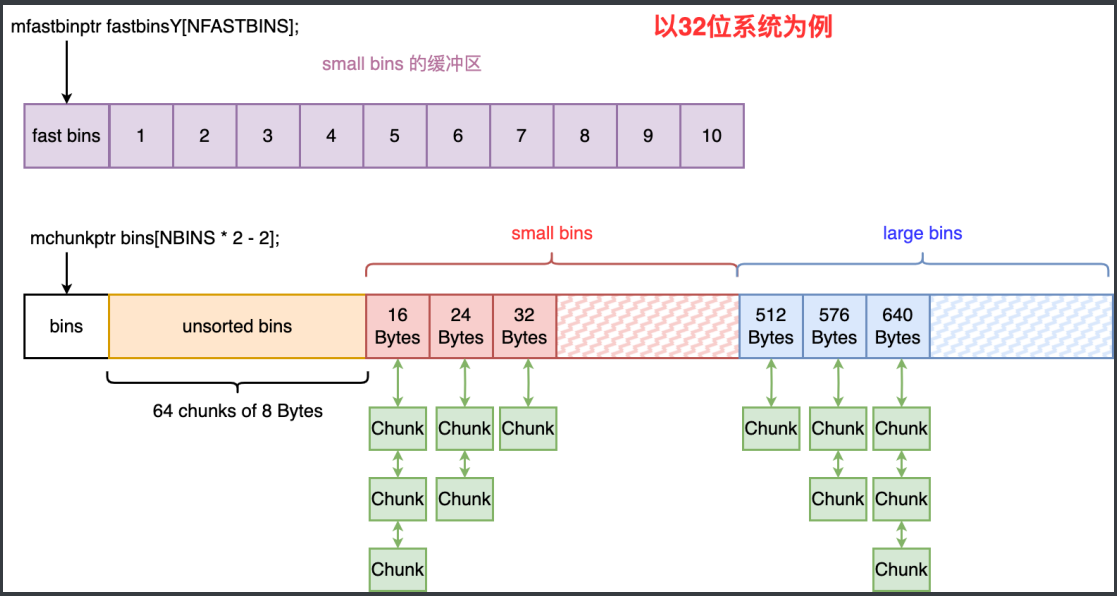


哈希桶free chunks管理

对于空闲的chunk，ptmalloc使用哈希桶来管理 free list，每一个内存分配区中维护着 bins 的列表数据结构，用于保存free chunks

根据空闲chunk的大小和其处于的状态将其放在四个不同的bin中，这四个管理空闲chunk的容器为fast bins、unsorted bin、small bins和large bins

不同bin的作用



1. fast bins是bins的高速缓存，大约有10个定长队列。当一些大小小于 max_fast（默认为64字节）的chunk被回收时，首先将其放入fast bins中，在分配小内存时，首先会查看fast bins中是否有合适的内存块，如果存在，则直接返回fast bins中的内存块，以加快分配速度

   fast bins可以看着是small bins的一小部分cache，主要是用于提高小内存的分配效率，虽然这可能会加剧内存碎片化，但也大大加速了内存释放的速度

2. unsorted bin只有一个，回收的chunk块必须先放到unsorted bin中，分配内存时会查看unsorted bin中是否有合适的chunk，如果找到满足条件的chunk，则直接返回给用户，否则将unsorted bin的所有chunk放入small bins或large bins中

   unsorted bin 可以重新使用最近 free 掉的 chunk，从而消除了寻找合适 bin 的时间开销，进而加速了内存分配及释放的效率

3. small bins用于存放固定大小的chunk，共64个bin，最小的chunk大小为16字节或32字节，每个bin的大小相差8字节或是16字节，当分配小内存块时，采用精确匹配的方式从small bins中查找合适的chunk

   small bins 相邻的 free chunk 将被合并，这减缓了内存碎片化，但是减慢了 free 的速度；

4. large bins用于存储大于等于512B或1024B的空闲chunk，这些chunk使用双向链表的形式按大小顺序排序，分配内存时按最近匹配方式从large bins中分配chunk

   large bin 中所有 chunk 大小不一定相同，各 chunk 大小递减保存。最大的 chunk 保存顶端，而最小的 chunk 保存在尾端；查找较慢，且释放时两个相邻的空闲 chunk 会被合并

其中fast bins保存在malloc_state结构的fastbinsY变量中，其他三者保存在malloc_state结构的bins变量中

关于哈希桶的步长，再强调一下

```
1     Bins for sizes < 512 bytes contain chunks of all the same size,
  spaced
2     8 bytes apart. Larger bins are approximately logarithmically
  spaced:
3
4     64 bins of size       8
5     32 bins of size      64
6     16 bins of size     512
7      8 bins of size    4096
8      4 bins of size   32768
9      2 bins of size  262144
10     1 bin  of size what's left
```

32位系统

组	bins数量	步长	
1	64	8字节	Small bins（62个）
2	32	64字节	Large bins（64个）
3	16	512字节	
4	8	4096字节	
5	4	32768字节	
6	2	262144字节	
7	1	不限制	

64位系统

组	bins数量	步长	
1	64	16字节	Small bin（62个）
2	32	128字节	Large bins（64个）
3	16	1024字节	
4	8	8192字节	
5	4	65536字节	
6	2	524288字节	
7	1	不限制	

说明：Small bins中实际个数为62个，Small bin从16字节/32字节作为起始bin，并且需要加上Large bins里面的起始bin512字节/1024字节

特殊的chunk

- top chunk
  - 一个 arena 中预留的最顶部的 chunk，位于已使用空间的上方。top chunk 不属于任何 bin 。当所有 bin 中都没有合适空闲内存时，就会使用 top chunk 来响应用户请求

    - 当 top chunk 的大小比用户请求的大小还要小的时候，top chunk 就通过 sbrk（main arena）或 mmap（ thread arena）系统调用扩容
- 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
- last remainder chunk
    - 是最后一次 small request 中因分割而得到的剩余部分，它有利于改进引用局部性，也即后续对 small chunk 的 malloc 请求可能最终被分配得彼此靠近
    - 当用户请求 small chunk 而无法从 small bin 和 unsorted bin 得到服务时，分配器就会通过扫描 binmaps 找到最小非空 bin。若这样的 bin 找到了，其中最合适的 chunk 就会分裂为两部分：返回给用户的 User chunk 以及添加到 unsorted bin 中的 Remainder chunk。这一 Remainder chunk 就将成为 last remainder chunk

heap管理

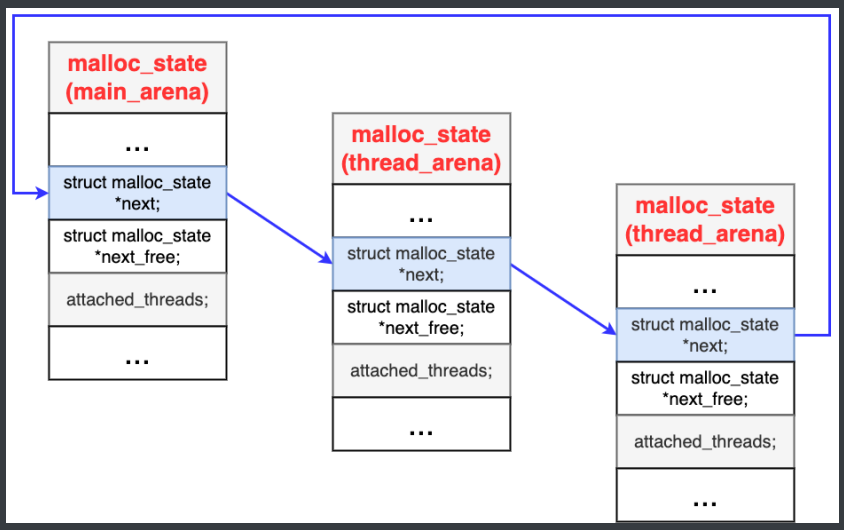
`_int_new_arena` 中主要调用 `new_heap` 来创建和初始化新的非主分配区

分配区 arena

ptmalloc对Doug Lea版本的改进

在Doug Lea实现的内存分配器中只有**一个主分配区（main arena），每次分配内存都必须对主分配区加锁**，分配完成后释放锁，在SMP的多线程环境下，对主分配区的锁的争用很激烈，严重影响了malloc的分配效率

为了解决多线程竞争锁的问题，ptmalloc增加了非主分配区thread_arena（或者叫动态分配区dynamic arena），主分配区与非主分配区 main_arena 用环形链表来进行管理。每一个分配区利用互斥锁（mutex）使线程对于该分配区的访问互斥




```
static struct malloc_state main_arena =
{
  .mutex = _LIBC_LOCK_INITIALIZER,
  .next = &main_arena,
  .attached_threads = 1
};
```

- 主分配区和非主分配区形成一个环形链表进行管理
- 每个分配区利用互斥锁使线程对于该分配区的访问互斥
- 每个进程只有一个主分配区，允许多个非主分配区
- ptmalloc根据系统对分配区的争用动态增加分配区的大小，分配区的数量一旦增加，则不会减少
- 主分配区可以使用brk和mmap来分配，非主分配区只能用mmap来映射
- 申请小内存是会产生很多内存碎片，ptmalloc在整理时也要对分配区做加锁操作

arena_get

arena_get2

分配区的管理

- 当一个线程需要malloc分配内存：先查看该线程私有变量中是否已经存在一个分配区
  - 若存在，尝试加锁
    - 若加锁成功，使用该分配区分配内存

        - 若失败，遍历循环链表，获取一个未加锁的分配区
    - 若没找到未加锁分配区，开辟新的分配区，加入全局链表并加锁，然后分配内存
- 当一个线程需要free释放内存
    - 先获取待释放内存块所在的分配区的锁
    - 若有其他线程持有该锁，必须等待其他线程释放该分配区互斥锁

其他malloc函数

free

realloc

tcmalloc

https://blog.csdn.net/ETalien_/article/details/88832703

https://jiajunhuang.com/articles/2020_10_10-tcmalloc.md.html

源代码版本

整体设计框架

三个主要考虑的点：**性能问题、内存碎片问题和多线程环境下的锁竞争问题**

tcmalloc 介绍

tcmalloc（Thread-Caching Malloc）是Google开发的一种用于多线程应用程序的内存分配器，在许多Google项目中得到广泛应用。tcmalloc旨在提供高效的内存分配和释放，**以减少多线程应用程序中的锁竞争和内存碎片化**

tcmalloc的设计目标是在**多线程环境下**最大限度地减少内存分配和释放的开销。它采用了许多优化策略来提高性能和可伸缩性

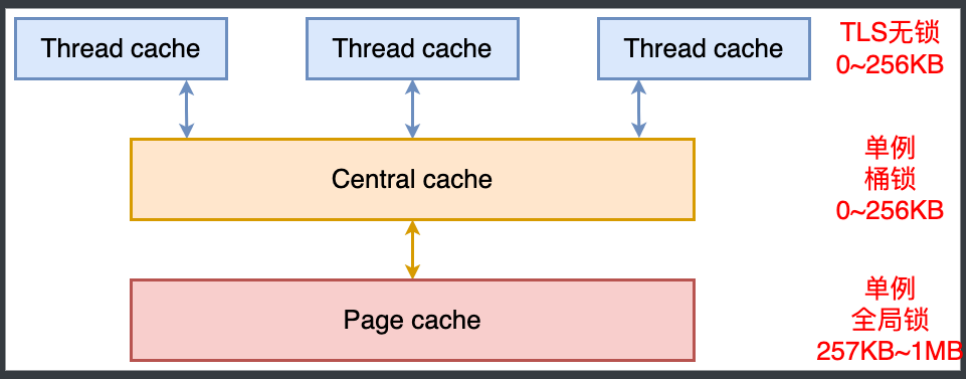
其中一个关键特性是线程本地缓存（Thread-Caching），它为每个线程维护了一个本地内存缓存，用于快速分配和释放内存。通过避免对全局数据结构的频繁访问，减少了锁竞争的情况，从而提高了性能

另一个重要的特性是分离的内存池（Central Cache），它用于处理大于某个阈值的内存分配请求。这些请求在被满足之前不会返回到操作系统，而是在内存池中进行高效的重用。这有助于减少对操作系统的系统调用次数，提高了性能

此外，tcmalloc还使用了一些其他的优化技术，如高效的内存块分配策略、精细的大小分类等，以提高内存分配的效率和内存利用率。

总的来说，tcmalloc是一种针对多线程应用程序的高性能内存分配器，通过利用线程本地缓存、分离的内存池和其他优化策略，提供了快速的内存分配和释放，并减少了锁竞争和内存碎片化的问题

三层设计

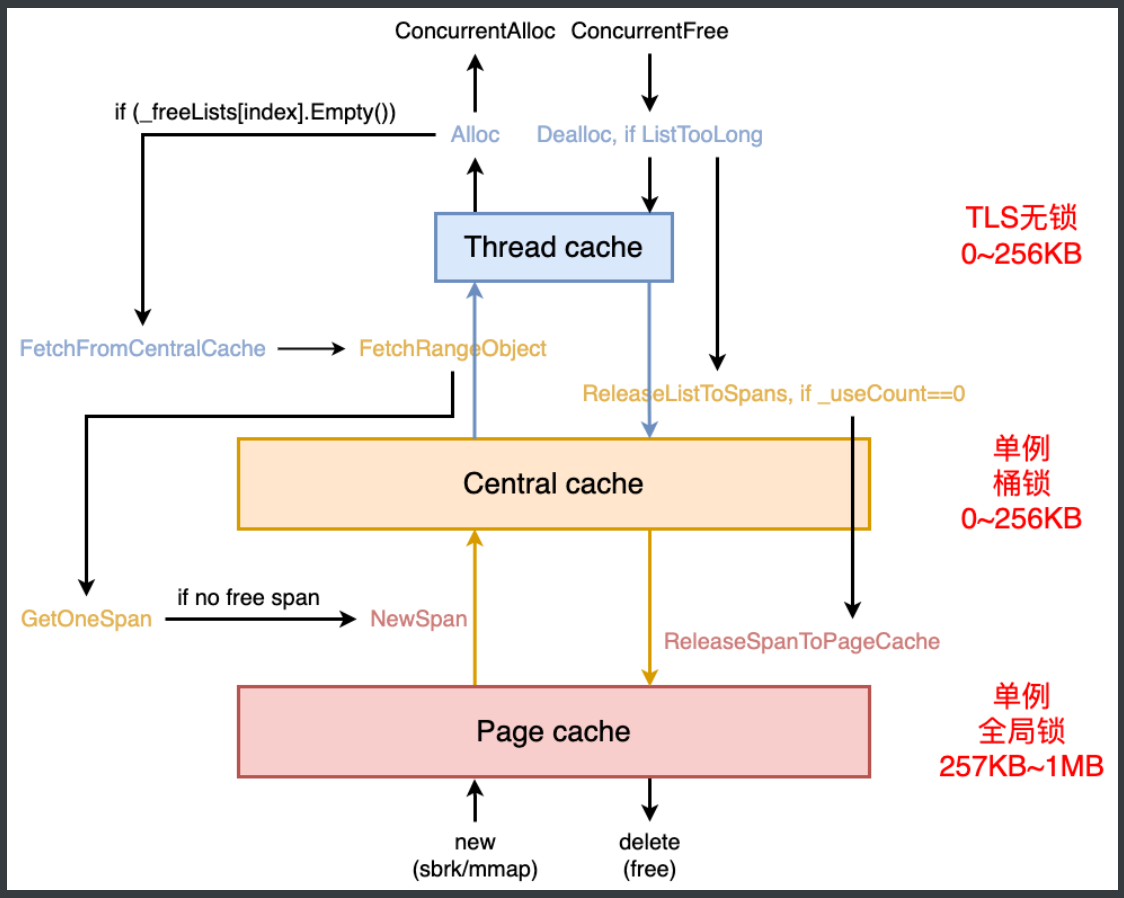


- Thread cache 解决锁竞争的问题：线程缓存是每个线程独有的，用于小于256KB的内存的分配，线程从这里申请内存不需要加锁，每个线程独享一个cache，相比于ptmalloc每个线程获取内存都要每次都要加锁，这就是这个并发线程池高效的地方
- Central cache 居中调度
  - central cache 是所有线程所共享的，thread cache按需从central cache中获取对象，central cache也会在合适的时机回收thread cache中的对象。central cache有负载均衡的作用，可以避免一个线程抢占过多内存
  - 因为central cache是共享的，所有存在竞争。但是通过哈希桶的设计，这里竞争不会很激烈。也就是所谓的桶锁。其次只有本身只有thread cache的

    没有内存对象时才会找central cache，所以更降低了竞争烈度

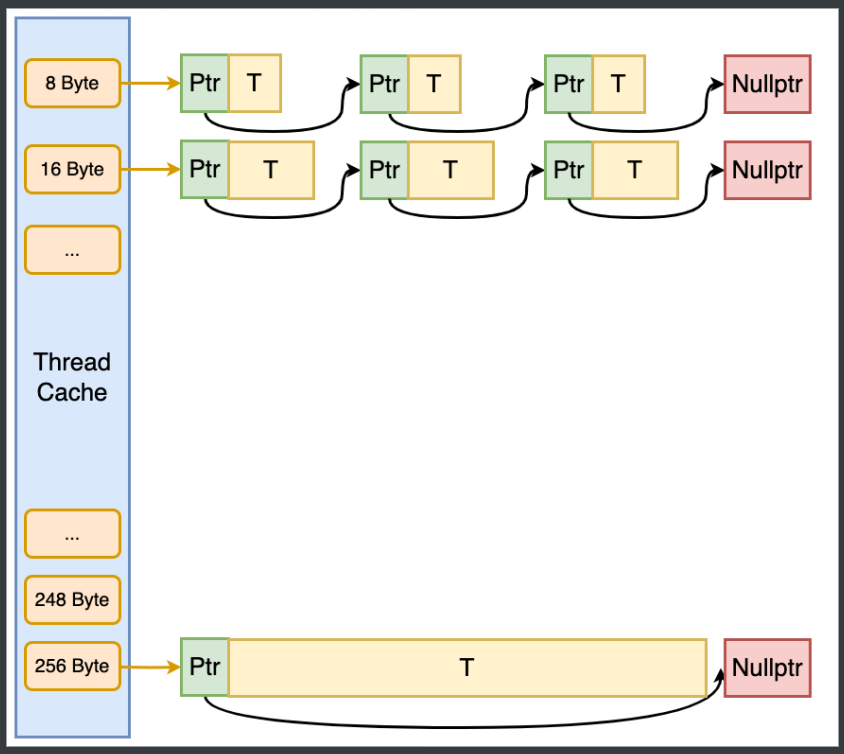
- Page cache 以页为单位管理大内存，用brk或mmap直接找OS的堆要内存

主要API结构大纲



Thread Cache

Thread Cache 的结构设计




```
class ThreadCache {
public:
  // 申请和释放空间
  void *Allocate(size_t size);
  void Deallocate(void *ptr, size_t size);
  // 从central cache获取对象
  void *FetchFromCentralCache(size_t index, size_t size);

private:
  FreeList _freeLists[NFREELISTS];
};
```

哈希桶映射与内存块对齐规则

定长内存池直接用一个自由链表就可以挂载所有的内存小块，但是 tcmalloc 是要管理不同大小的内存块该怎么办呢？很简单，分成不同的自由链表来管理不同大小的内存块就可以了！

Thread Cache是哈希桶结构，每个桶是一个按桶的位置映射大小的内存块对象的自由链表，相当于是**直接定址法**。但问题是要如何设计设计映射，或者说如何设计内存分配方式。最Naïve的想法是精确到每一个Byte都分配，但这样挂的自由链表将会非常多，即需要 `256 * 1024 = 262144` 个哈希桶来放自由链表

所以为了平衡效率，需要做出一些空间上的浪费（内碎片）。设计成以8 Byte为一个间隔作为哈希桶。若要1 Byte，给8 Byte；要2 Byte，也给8 Byte；要8 Byte，还给8 Byte。也就是说在这种情况下，内存全部对齐到8 Byte

为了存下64位指针，毋庸置疑最少值肯定是8字节。若按8字节递增对齐，那么到256KB的时候，总共需要 `256 * 1024 / 8 = 32768` 个哈希桶来放自由链表

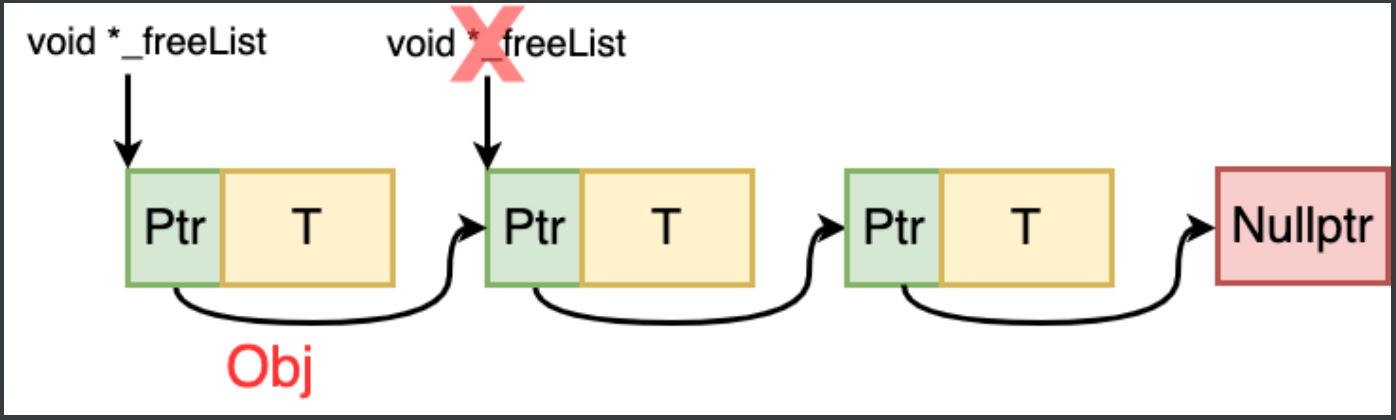
这感觉还是太多了，在tcmalloc的实现中实际采用的是下面这种，整体控制在最多10%左右的内碎片浪费。按下面的对齐方法，一共是有208个哈希桶来放自由链表

```
// 要的内存数                  实际对齐到某个大小    哈希桶保存的自由链表的index
// [1,128]                    8byte对齐           freelist[0,16)
// [128+1,1024]               16byte对齐          freelist[16,72)
// [1024+1,8*1024]            128byte对齐         freelist[72,128)
// [8*1024+1,64*1024]         1024byte对齐        freelist[128,184)
// [64*1024+1,256*1024]       8*1024byte对齐      freelist[184,208)
```

使用一个专门的类 `SizeClass` 来管理对象大小的对齐映射规则。在计算对齐的 `_RoundUp` 函数中有一个 align_num 对齐数（并不是对齐到的字节数）没有使用一般的取模操作，而是下面这么实现的

```
// 一种非常巧妙的算法
static inline size_t _RoundUp(size_t size, size_t align_num) {
    return ((size + align_num - 1) & ~(align_num - 1));
}
```

管理自由链表



```cpp
//获取内存对象中存储的头4或头8字节，即连接的下一个对象的地址
//void**强转后解引用可以适用于32和64位
static void *&NextObj(void *obj) { return *(void **)obj; }

// 管理切分好的小对象的自由列表
class FreeList {
public:
  // 自由链表头插
  void Push(void *obj) {
    assert(obj);
    // 头插
    //*(void **)obj = _freeList;
    NextObj(obj) = _freeList;
    _freeList = obj;
  };

  //支持范围内push多个对象
  void PushRange(void *start, void *end) {
    NextObj(end) = _freeList;
    _freeList = start;
  }

  // 自由链表头删
  void *Pop() {
    //头删
    assert(_freeList);
    void *obj = _freeList;
    _freeList = NextObj(obj);
    return obj;
  };

  bool Empty() { return _freeList == nullptr; }

  size_t &MaxSize() { return _maxSize; }

private:
```

```
37    void *_freeList = nullptr;
38    size_t _maxSize = 1;
39  };
```

TLS无锁访问

Thread-Local Storage 是一种线程级别的存储机制，它允许每个线程在共享的内存空间中拥有自己独立的变量副本。每个线程都可以访问自己的TLS变量副本，而不会干扰其他线程的副本

TLS的主要目的是提供一种线程隔离的机制，使得每个线程可以独立地使用一组变量，**而不需要使用全局变量或上锁**。这对于多线程应用程序非常有用，因为它可以避免并发访问共享变量所带来的竞争条件和同步开销

每一个Thread Cache都是TLS，可以不受影响的并发申请资源。我们把TLS的申请封装到ConcurrentAlloc.h中。和封装的malloc函数或者tcmalloc函数一样，申请内存的时候直接用 `static void *ConcurrentAlloc(size_t size);`

Thread Local Storage（线程局部存储）TLS - 一束灵光的文章 - 知乎 https://zhuanlan.zhihu.com/p/142418922

根据上文，TLS在Linux中有两种使用方式，API方式和语言级别的方式，在我们的实现中采用语言级别的实现，即用 `__thread` 来声明需要用TLS管理的资源

```
1  // 在tcmalloc中这个函数被命名为tcmalloc
2  static void *ConcurrentAlloc(size_t size) {
3    // 通过TLS每个线程无锁的获取自己专属的ThreadCache对象
4    if (pTLS_thread_cache == nullptr) {
5      pTLS_thread_cache = new ThreadCache;
6    }
7
8    // 获取线程号
9    cout << std::this_thread::get_id() << ": " << pTLS_thread_cache <<
   endl;
10
11   return pTLS_thread_cache->Allocate(size);
12 }
```

```
13
14 static void ConcurrentFree(void *ptr, size_t size) {
15   assert(pTLS_thread_cache);
16   pTLS_thread_cache->Deallocate(ptr, size);
17 }
```

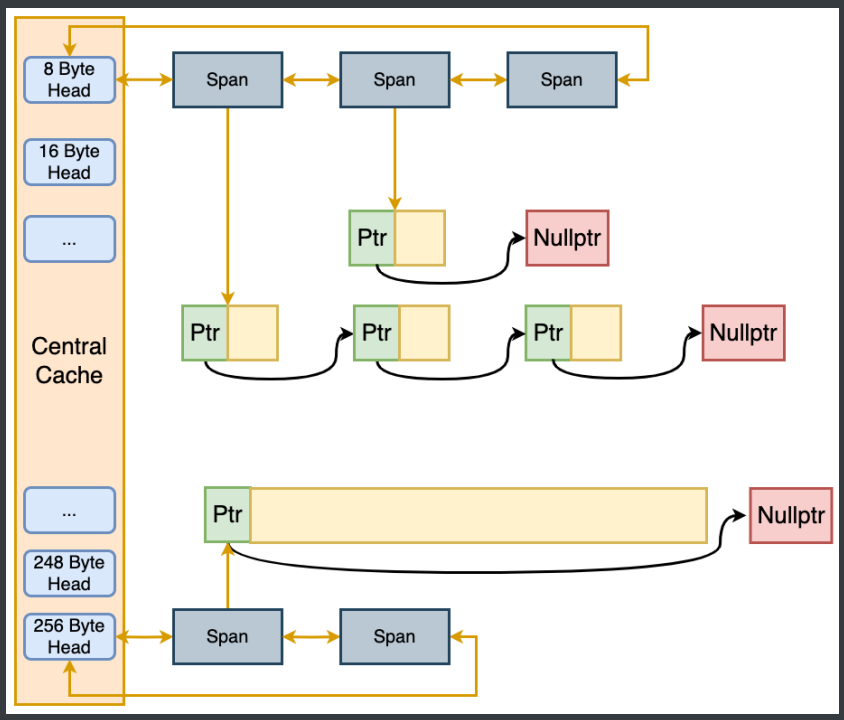
注意：在ThreadCache.h中为了避免头文件多次引入引发链接错误，所以将pTLS_thread_cache定义为static。这也意味着这意味着每个线程都会有自己独立的 `pTLS_thread_cache` 变量副本。每个线程在首次访问 `pTLS_thread_cache` 时，会进行初始化，并且每个线程的初始化操作都是独立的。因此，每个线程的 `pTLS_thread_cache` 变量在初始化之后都不会再为 `nullptr` ，并且每个线程都有自己独立的 `ThreadCache` 对象。并独立地使用它进行内存分配和释放操作

内存回收

Tcmalloc考虑了自由链表长度和总占用内存两个方面，我们这里简化一下，只考虑链表长度

Central Cache

Central Cache的结构设计



Central Cache和Thread Cache相似都采用了哈希桶的结构，并且它的哈希桶映射与内存块对齐规则也与Thread Cache一样。不同的是哈希桶挂的不是内存块的自由链表，而是SpanList

Span管理以页为单位的大块内存，每个span又会被切成对应的小块挂在span上，然后供Thread Cache取用或者回收并进行负载均衡。若所有Span都没了，那就再去找Page Cache要。Span及其管理结构SpanList既要给Central Cache用，也要给之后的Page Cache用，所以定义到Common.h中

首先要条件编译为不同的OS设置不同的page ID，若是在Win系统下，注意要先写宏 `_WIN64` 再写宏 `_WIN32` 的顺序https://blog.csdn.net/chunfangzhang/article/details/87895833

不同span里到底挂了多少个小的内存块是不知道的，因为时刻可能有新的内存块被Thread Cache还回来

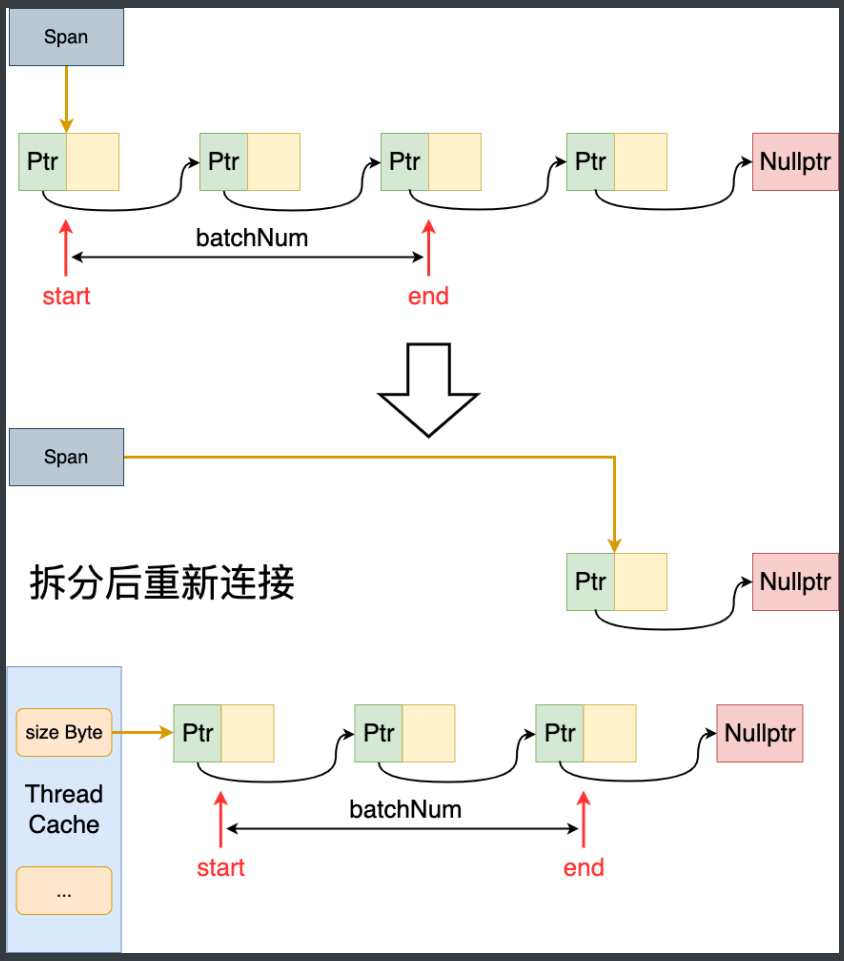
span设计为一个带头双向循环链表，方便当span的 `_useCount==0` 时，Central Cache把Span还给Page Cache的时候方便找到对应的Span，然后重新做被删除Span前后的Span的连接

Central Cache是管理多个Thread Cache的，因此它里面是有线程竞争的，但是它使用了**桶锁**来尽量降低竞争。所谓的桶锁就是不是一下子把整个 Central Cache 全部锁住，而是把单独的一个桶给锁住。如果两个线程找的是不同锁，那不构成线程竞争，只有当两个线程都映射到同一个桶的时候才会需要等待锁

申请内存

Central Cache只有一个，所以把Central Cache设计成饿汉单例（注意为了防止头文件重复包含，将 `_sInst` 定义到cc中）

桶锁：每个哈希桶有一个锁。所以只有在找的是同一个锁的时候才会有竞争，否则会去找不同的桶



一个Span每次给Thread Cache多少个切好的内存块合适呢？给固定数量是不合适的，一是因为不同的Thread要的频率不同，二是因为大的内存块给的数量跟小的一样可能会造成严重的浪费。采用**慢开始反馈调节算法**：最开始不会一次向 Central Cache 要太多，要太多了可能会用不完造成浪费。但是如果不断地往 Central Cache 要，每要一次maxSize++（maxSize是FreeList的属性），那么每次要的batchNum会越来越多，直到上限。而且内存块越小，上限就越高

一次给的是 start ~ end 那么多

慢开始反馈调节算法写在 `ThreadCache::FetchFromCentralCache` 里，规定不同大小内存块上限的逻辑写在Common.h里，如下

```
//一次thread cache 从 central cache 获取多少个对象
static size_t NumMoveSize(size_t size) {
    assert(size > 0);
    //[2, 512]，一次批量移动多少个对象的（慢启动）上限值
    // 小对象一次批量上限高
    int num = MAX_BYTES / size;
    if (num < 2)
        num = 2;
    // 测试得出的512
    if (num > 512)
        num = 512;
    return num;
}
```

以8字节为例，要取 `256*1024/8=32768` 个，此时又太多了，所以定了个上限500。若取一个256KB的，至少要取2个

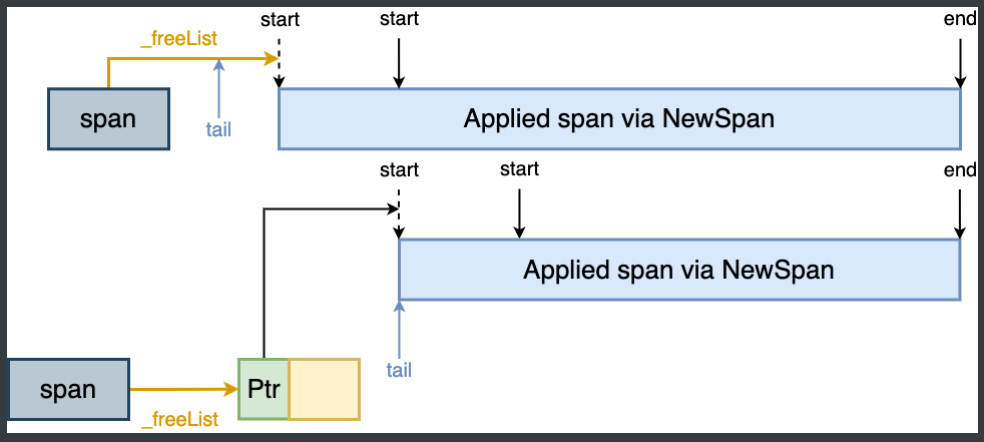
GetOneSpan：从Central Cache向Page Cache要一块span并切割内存

每次要多少页比较好，也设计成自适应的方法。size越小分配的page越少，size越大分配的page越大。 `num*size` 是总的字节数，PAGE_SHIFT是字节到页的转换。若定义一页为8KB，则 `PAGE_SHIFT=13` ；若定义一页为4KB，则 `PAGE_SHIFT=12`

```
static size_t NumMovePage(size_t size)
{
    size_t num = NumMoveSize(size);
    size_t nBytge = num*size;
    size_t nPage = nByte >> PAGE_SHIFT;
    if (nPage == 0)
        nPage = 1;
    return nPage;
}
```

上面的过程中是通过页号，计算页的起始地址，即 `PageNum << PAGE_SHIFT` 。假设页号为100，tcmalloc中默认一页为8K，可在configure时通过选项调整为32KB或64KB。注意⚠️：不要把用户层的 page 和虚拟内存中的 page 搞混了

```
100 * 8K
100 << PAGE_SHIFT
```



把大块span切成小块内存后以FreeList的方式连接起来，注意第一次连接是把span的 `_freeList` 跟内存块连接起来。尾插效果会比较好，头插的话SpanList中的地址会倒过来

内存回收

当申请了一个span，span上有多个切好的内存块，但是没人知道中间用户到底是怎么用了这些小内存块，出于**负载均衡**的目的，Thread Cache 是会在合适的时候主动向 Central Cache 归还内存块的，**但问题在于并不能确定从 Thread Cache 还回来的内存块是属于哪一个span的**

解决方法也很简单，就是建议好小块内存和切出来它的页中间的映射，所以tcmalloc的解决方案是通过 `_pageID` 来定位属于哪一个span。具体的实现有

1. 来一个内存块暴力遍历一遍span来确定属于哪个span是一个***O(N^2)***的算法，因此不采用这种方法
2. 采用 `unordered_map<PAGE_ID, Span*> _idSpanMap` 建立映射的***O(1)***算法，因为通过地址算PAGE_ID是很容易的，那么映射后直接可以找到对应的span

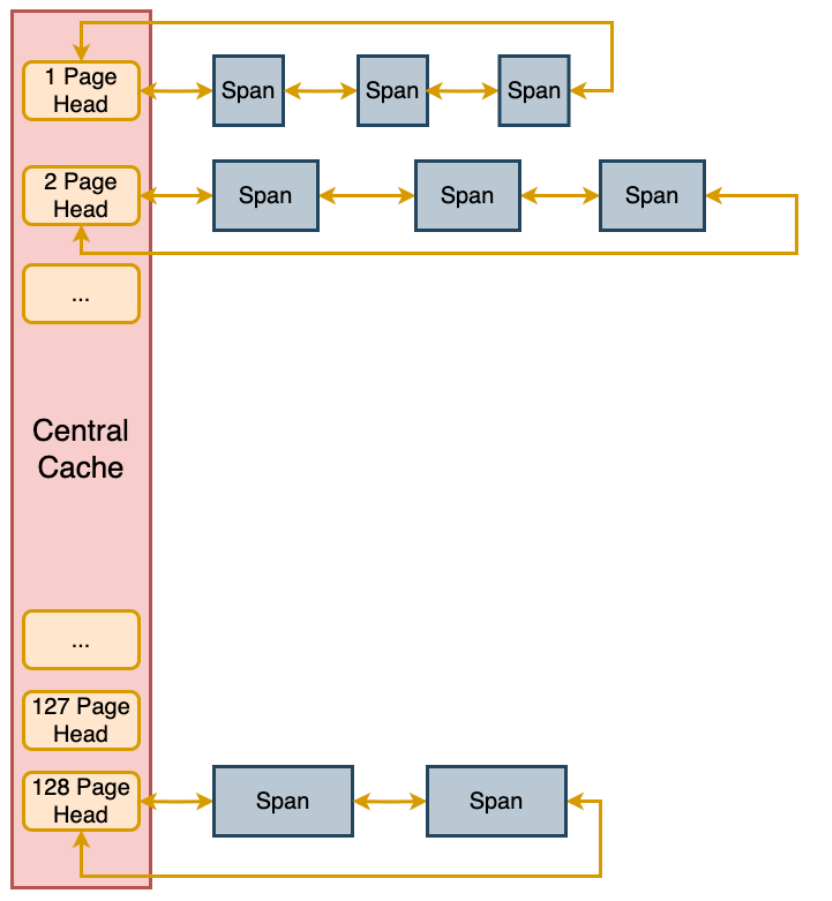
这个映射在向系统获取128页的NewSpan，或者是每次切割大小Span的时候建立

若Central Cache中的span的 `_useCount` 都等于0，说明切分给Thread Cache的小块内存都还回来了，那么 Central Cache 把这个 span 还给 Page Cache。Span 之间用双向链表来管理也是要便利 span 的归还。之后Page Cache会通过页号，查看前后的相邻页是否空闲，是的话就合并，合并出更大的页解决内存碎片问题

先把Central Cache的桶锁解掉，这样若其他线程释放内存对象回来，不会阻塞住

Page Cache

Page Cache的结构设计



Page Cache在全局中也只有一个，所以也一样设计成单例模式。它最多管理 128 页（`8KB*1024=1MB`）的内存

Page Cache也是哈希桶结构，但是它的映射规则跟Central Cache与Thread Cache不同。每个桶是一个按桶的位置映射页数的自由链表，一共有128个哈希桶来存放自由链表。Page Cache也采用SpanList进行管理，每个上挂的是span对象，不会切割，因为Page Cache服务的是Central Cache，切割的工作由Central Cache拿到span后自己完成

Page Cache中不能再实现成桶锁了，要实现成全局锁，因此Page Cache要做span的分裂与合并，桶锁只能制约不同哈希桶下的span取用，但是不能限制跨桶之间span的分裂与合并。之所以要垮桶操作span的原因见下。除此之外，当没有对应page的span时，每往后找一次大的span都要上锁、解锁，这种消耗可能会很大，干脆用一把大的全局锁

NewSpan：获取一个k页的span

直接去跟OS要的时候不要一直要小块的内存，为了减少外碎片，尽量每次要就要比较大块的内存。如果每次小的span没了都是去堆上要的话很容易就形成内存碎片了，而要一个大的span就是连续的一段空间会好很多。所以**没有合适的span时是先去找大的span然后进行分裂，而不是直接去向OS申请**。若真的一个大的都没有找到，就向系统申请一个128页的内存

块，并挂到128 page的哈希桶上

以申请一个2 page的内存块为例，若一个都没有找到，就去申请一个128 page的内存块，然后切一个2 page出来给Central Page，另外126 page的内存块挂到126 page的哈希桶上。其实刚开始的时候就是这种情况，一个span都没有

切分成一个k页的span和一个n-k页的span，k页的span返回给Central Cache，n-k页的span挂到第n-k桶上去

NewSpan的递归可能涉及到递归死锁的问题。可以通过递归互斥锁 `recursive_mutex` 或者分离一个调用函数来解决死锁问题

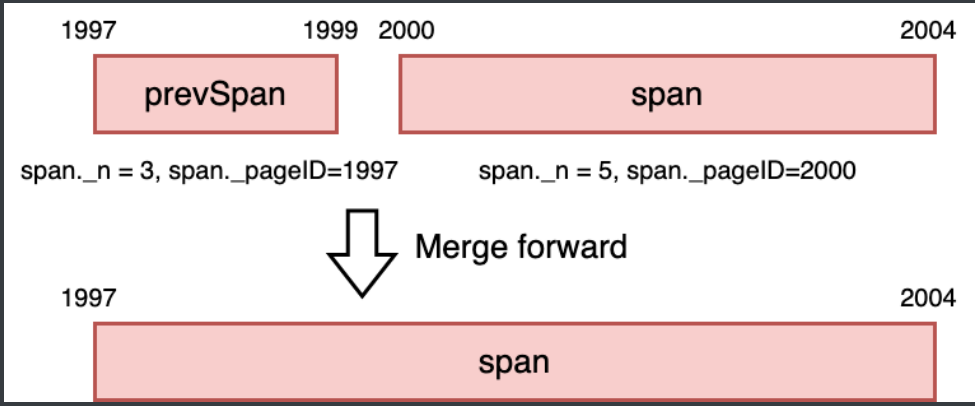
还涉及到一个锁的问题，当Central Cache调用Page Cache的内容时，因为Central Cache是桶锁，而Page Cache是全局锁，要不要先解开桶锁再加全局锁呢。这个问题比较有争议，但总的来说解了比较好。虽然本身是因为没有空闲的span了才会去找Page Cache要span，所以对于其他线程的申请没什么价值，因为反正也是申请不到span的，但是如果是其他线程想要归还span呢？把释放也给堵住了不太好

最后考虑到这两个问题，Page Cache的全局锁加再调用 `NewSpan` 的 `CentralCache::GetOneSpan` 里面，虽然粒度大一点，但可以避免递归死锁

有没有这样一种可能，当把桶锁解开后，有很多线程访问这个桶都发现没有空的span，所以它们都会去调Page Cache的NewSpan，但是因为Page Cache这时候已经上了全局锁，所以他们也申请不到。然后等到申请完后解开全局锁，大家都申请到了新的span要挂到Central Cache中对应的桶上，然后就有太多内存了？

内存回收：ReleaseSpanToPageCache

若直接把从Central Cache返回的切小的Span挂到对应Page桶上的话，会造成小Page的很多，而大Page的很少，即**外碎片问题**。所以也要对span前后的页尝试进行合并，最后也还给系统，缓解外碎片问题



借助 `_idSpanMap` 映射的帮助，可以根据 `PageID` 往前往后不断合并 `_n` 页，合并完了之后可能还可以继续往前往后合并，但如果是还在Central Cache 中的就不能合并了。**不过连续合并后超过 NPAGE（最大页数）之后就不可以继续合并了**（有可能系统分配内存的时候正好就将 NPAGE 的几段内存连续分配了），因为我们是要把合并好的页仍然加入 SpanList 管理，此时将 NPAGE 的 span 放入对应的 SpanList，然后如果下次又触发了相应的合并机制，就直接把 NPAGE 的span还给内存

但是如何判断是在Central Cache中还是在Page Cache中呢

- 不能使用 span 的 `_useCount` 来判断，因为存在线程安全问题，之前说过在NewSpan里，**从桶锁解开开始切分 NewSpan 到且分完重新上锁挂到 Central Cache 对应的 SpanList 那段时间间隔中，从Page Cache刚拿过来的正在切分的 `_useCount` 也为 0**。若此时把这段刚拿过来准过要分开的span给合并了那就完了
- 解决方法是在Span里增加一个 `bool _isUse=false;` 属性来表明是否在使用，只要分配给了Central Cache，那么就要变成 `true`

nSpan也要被合并：注意和Central Cache中的不同，kSpan是返回给Central Cache的，它之后会被切分为小块内存，所以每一个页都要建立映射，可以让它在在从 Thread Cache 返回给 Central Cache 的时候确认是属于哪一个Span的。而nSpan暂时是留在Page Cache里的，它暂时不需要被切分。但是我们也要通过它来进行合并以返回给系统，返回的时候确认id和span的映射只需要找首尾的页就行了，因为需要向前向后合并，既然还没有被喂给Central Cache，中间必然是连续的

注意⚠️：笔者当时漏掉的一个点是当merge完毕后要 `delete prevSpan` 。因为Span结构体是我们自己创造出来管理span对象的，当被合并了之后也就没有用了，所以delete掉。注意不要和申请的内存混淆，通过内存池申请的内存在程序运行期间不会被还给内存，就算不用了也都是集中到Page Cache手里。只有最后程序终止后，才会被还给OS

大于256KB的内存回收

因为Page Cache最高可以管理128页/1MB的内存块，所以当申请32~128页（假设一页为8KB，`256/8=32`）的内存时，还是找的Page Cache；若直接申请超过128页的那么就得去找堆要了

具体的那就是调用不同平台上的内存管理的系统调用就行了

替换定长内存池

用高效的定长内存池管理 Span & ThreadCache 对象

Page Cache 需要管理、操作 Span 对象（Central Cache只是使用已经存在的 Span 对象），一开始笔者的实现还是用了 `new/delete`，这就还是间接地在用 ptmalloc

实际上我们可以用上文写的高效的定长内存池 `ObjectPool<Span> _spanPool;` 来专门获取、管理所有的 Span 对象

ConcurrentAlloc.h和ObjectPool.h里获取ThreadCache同理

Free不传对象大小

每一个span里的管的小内存块都是切成同样大小的小块，所以干脆让span记录一个它管的小块内存的大小 `_objSize`，方便归还的时候找对应哪一个哈希桶

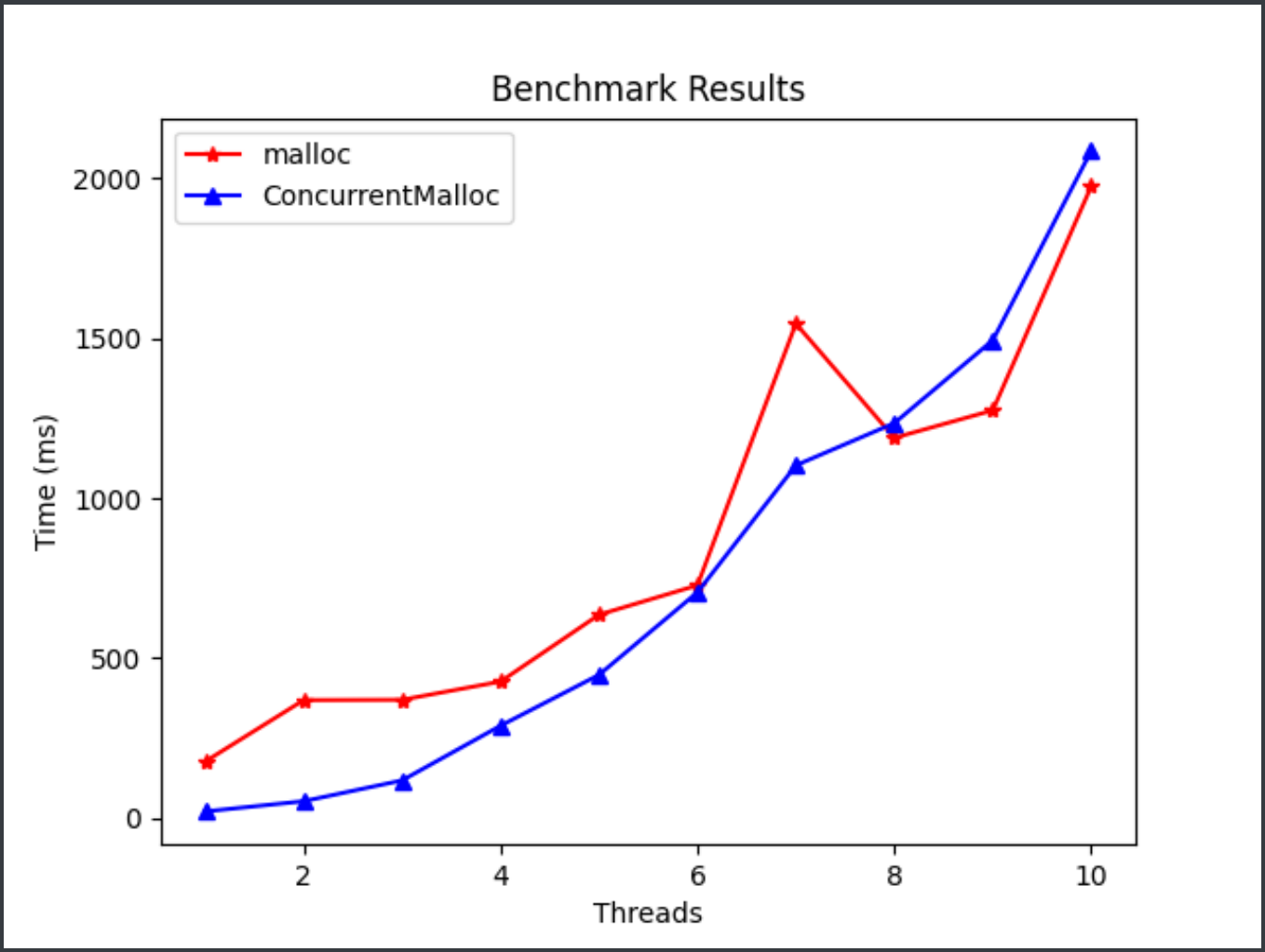
这也是为什么要用 Span 来统一管理大于1MB的大内存

Benchmark

多线程环境对比malloc测试

统一设定参数每轮申请10000次，一共进行10轮，观察不同线程的申请效果

单进程实验：可以看到，我自己实现的tcmalloc和glibc的ptmalloc2相比，尽管实际实现的tcmalloc代码量大概在十万行细节，我这里仅仅实现了核心的一两千行代码，很多细节都没有考虑在内，但性能已经相差不多了



性能瓶颈分析

函数名	调用数	已用非独占时间百分比	已用独占时间百分比	平均已用非独占时间	平均已用独占时间	模块名
_Mtx_lock	849.461	46,40 %	46,40 %	0,00	0,00	MSVCP140.dll
_Thrd_join	4	21,02 %	21,02 %	120,45	120,45	MSVCP140.dll
CentralCache::ReleaseListToSpans	13.916	33,20 %	10,13 %	0,05	0,02	MemoryPoolOnWin.exe
<lambda_2c48730f5763b0127a8e29b8a7fa27...	4	78,66 %	7,65 %	450,70	43,85	MemoryPoolOnWin.exe
CentralCache::FetchRangeObj	21.106	9,20 %	7,32 %	0,01	0,01	MemoryPoolOnWin.exe
_Mtx_unlock	849.461	5,89 %	5,89 %	0,00	0,00	MSVCP140.dll

直接利用vs提供的性能分析工具，其中第一项是Benchmark的加锁，但是可以发现第三项占用了大量时间

可以发现大量的时间浪费在了锁竞争上，这个锁竞争是Page Cache为了管理 Span（比如说发放 span、回收 span）去读写 `std::unordered_map<PAGE_ID, Span *>` 的时候产生的

之所以这时候要加锁解锁是因为其他线程可能会增删这个 unordered_map（底层的红黑树旋转平衡等会更改指针关系），而 STL 的 unordered_map 是线程不安全的，所以优化的方向是如何去减轻这里的锁竞争程度

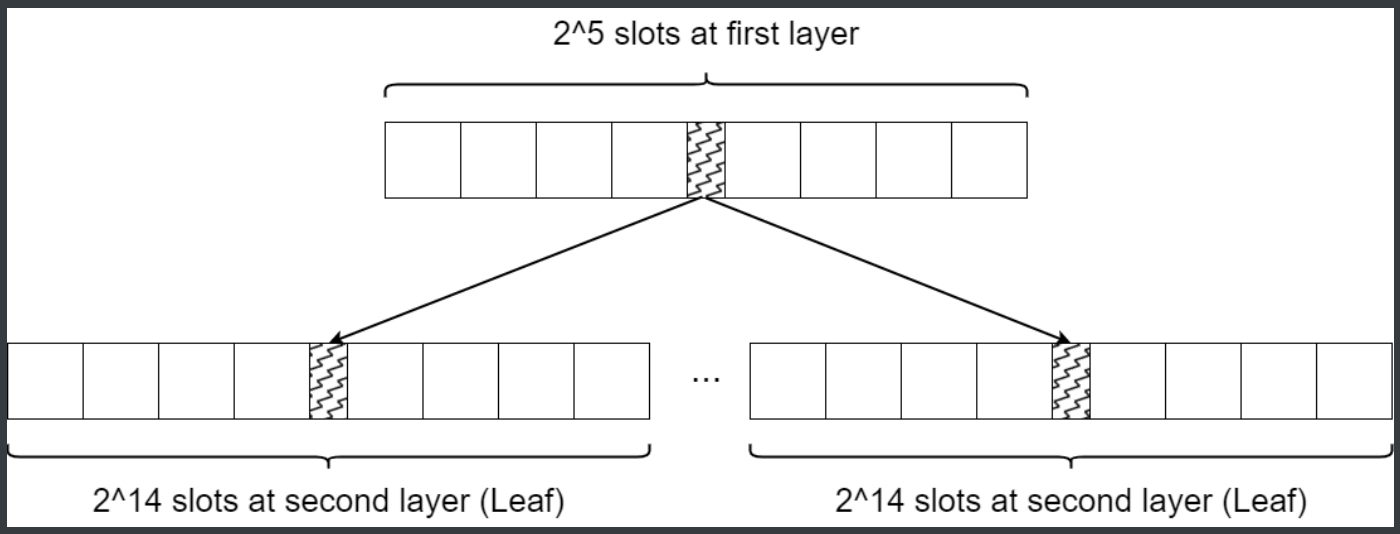
优化锁的消耗

基数树替换字典

`std::unordered_map<PAGE_ID, Span *>` 的 key 查找越慢，就越加剧锁竞争。PAGE_ID 是一个很大的值，所以可以考虑用 Radix Tree（压缩前缀树）来建立映射

假设一页是 `2^13=8KB`，那么一共有 `2^32/2^13=2^19=524,288` 页，所以也就需要 `2^13` 个 PAGE_ID 来标识它。占用内存为 `2^19*4 Byte=2^21=2MB`。基数树会建立页号 `_pageID` 和指针 `Span *` 的映射，`BITS = 32-PAGE_SHIFT` 或者 `BITS = 64-PAGE_SHIFT`，这里一层基数树的基为 `BITS=19`

tcmalloc 中分别设计了三种高度的基数树，即一层、两层和三层的，这三种基数树使用的空间没有变化的。32位下的时候一层或两层基数树就够了，但64位下是不够的 `2^64/2^13=2^51`，直接开一个连续的 `2^51` 的Vector 是不可能的，所以得用3层。一层的优势在于直接就开好了，访问非常简单，**一层本质上就是一个一次性开完的有 `2^19` 个元素的大哈希vector**。多层的是多层哈希，稍微麻烦了一点，实际上两层也是直接开好，但是三层就要按需开了



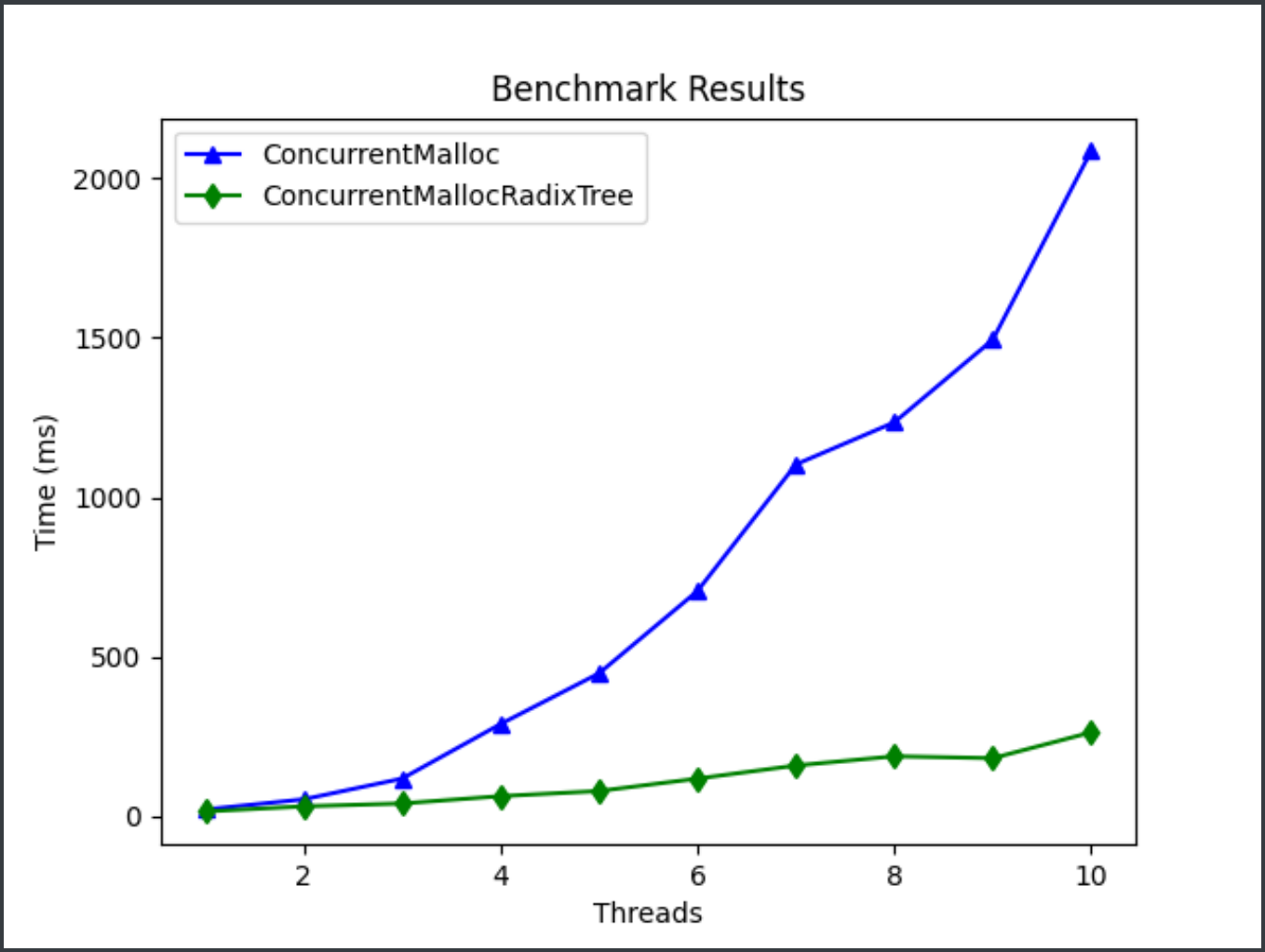
为什么此时去基数树里操作就不需要加锁解锁了？

1. 首先基数树是直接把整棵树给开出来，或者当三层的时候是提前通过ensure来开好内存的，写的时候也不会去改变基数树的结构，不像 unordered_map 需要修改其底层的红黑树

2. 设计成了读写分离的模式，首先可以分析一下只有两个地方需要写，即 `Span *PageCache::NewSpan(size_t k);` 和 `void PageCache::ReleaseSpanToPageCache(Span *span);` 的时候需要写（即建立 `PAGE_ID` 和 `Span *` 的映射关系），这时候是加了Central Cache的全局锁的。实际上不加锁都可以，因为不可能同时对一块 Span 既申请又释放

采用基数树后再次进行Benchmark

多线程实验，在实现了基数树管理Span之后效率提升十分明显，尤其是在多线程条件性能差异巨大



工程管理

Cmake编写

打包成库

jemalloc

传统分配器中大量开销被浪费在lock contention和false sharing上, 随着线程数量和核心数量增多, 这种分配压力将越来越大.**针对多线程, 一种解决方法是将一把global lock分散成很多与线程相关的lock.** 而针对多核心, 则要尽量把不同线程下分配的内存隔离开, 避免不同线程使用同一个cache-line的情况.按照上面的思路, 一个较好的实现方式就是引入arena.将内存划分成若干数量的arenas, 线程最终会与某一个arena绑定.由于两个arena在地址空间上几乎不存在任何联系, 就可以在无锁的状态下完成分配. 同样由于空间不连续, 落到同一个cache-line中的几率也很小, 保证了各自独立。由于arena的数量有限, 因此**不能保证所有线程都能独占arena**, 分享同一个arena的所有线程, 由该arena内部的lock保持同步

chunk是仅次于arena的次级内存结构，arena都有专属的chunks, 每个chunk的头部都记录了chunk的分配信息。chunk是具体进行内存分配的区域，目前的默认大小是4M。chunk以page（默认为4K)为单位进行管理，每个chunk的前几个page（默认是6个）用于存储chunk的元数据，后面跟着一个或多个page的runs。后面的runs可以是未分配区域， 多个小对象组合在一起组成run, 其元数据放在run的头部。 大对象构成的run, 其元数据放在chunk的头部。在使用某一个chunk的时候，会把它分割成很多个run，并记录到bin中。不同size的class对应着不同的bin，在bin里，都会有一个红黑树来维护空闲的run，并且在run里，使用了bitmap来记录了分配状态。此外，每个arena里面维护一组按地址排列的可获得的run的红黑树。

jemalloc 按照内存分配请求的尺寸，分了 small object (例如 1 – 57344B)、 large object (例如 57345 – 4MB)、 huge object (例如 4MB以上)。jemalloc同样有一层线程缓存的内存名字叫tcache，当分配的内存大小小于tcache_maxclass时，jemalloc会首先在tcache的small object以及large object中查找分配，tcache不中则从arena中申请run，并将剩余的区域缓存到tcache。若arena找不到合适大小的内存块， 则向系统申请内存。当申请大小大于tcache_maxclass且大小小于huge大小的内存块时，则直接从arena开始分配。而huge object

的内存不归arena管理， 直接采用mmap从system memory中申请，并由一棵与arena独立的红黑树进行管理。