Mimalloc 内存分配

在 **Mimalloc** 页中进行的 **Free List** 的 **Sharding**

在 mimalloc 中，其通过将内存分页，每个页负责一个特定大小的内存的分配，每个页有一个 Free List，因此内存分配的空间局部性较好。

Local Free List

mimalloc 希望能够限制内存分配与内存释放在最差情况下的时间消耗，如果上层应用需要释放一个非常大的结构体，其可能需要递归的释放一些子结构体，因而可能带来非常大的时间消耗。因而在 koka 与 lean 语言中，其运行时系统使用引用计数来追踪各种数据，然后保留一个延迟释放的队列(deferred decrement list)，当每次释放的内存块超过一定数量后就将剩余需要释放的内存块加入该队列，在之后需要释放的时候再进行释放。那么下一个需要确定的问题是什么时候再去从该队列中释放内存。从延迟释放队列中继续进行释放的时机最好应当是内存分配器需要更多空间的时候，因此这需要内存分配器与上层应用的协作。

在 mimalloc 中，其提供一个回调函数，当进行了一定次数内存的分配与释放后会主动调用该回调函数来通知上层应用。mimalloc 在实现时检测当前进行内存分配的页的 Free List 是否为空，如果为空则调用该回调，但是为了避免用于一直不断的分配与释放内存，导致 Free List 一直不为空，而导致回调函数一直得不到回调。因此 mimalloc 将 Free List 第二次进行 Sharding，将其分为 Free List 与 Local Free List。

在 mimalloc 中，其提供一个回调函数，当进行了一定次数内存的分配与释放后会主动调用该回调函数来通知上层应用。mimalloc 在实现时检测当前进行内存分配的页的 Free List 是否为空，如果为空则调用该回调，但是为了避免用于一直不断的分配与释放内存，导致 Free List 一直不为空，而导致回调函数一直得不到回调。因此 mimalloc 将 Free List 第二次进行 Sharding，将其分为 Free List 与 Local Free List。

Thread Free List

在 mimalloc 中，其提供一个回调函数，当进行了一定次数内存的分配与释放后会主动调用该回调函数来通知上层应用。mimalloc 在实现时检测当前进行内存分配的页的 Free List 是否为空，如果为空则调用该回调，但是为了避免用于一直不断的分配与释放内存，导致 Free List 一直不为空，而导致回调函数一直得不到回调。因此 mimalloc 将 Free List 第二次进行 Sharding，将其分为 Free List 与 Local Free List。

Full List

由于在 mimalloc 中每个堆中都有一个数组 pages，该数组中每个元素都是一个由相应大小的页组成的队列；同时还有一个 pages_direct 的数组，该数组中每个元素对

应一个内存块的大小类型，每个元素均为指向负责对应大小内存块分配的页的指针。因此 mimalloc 在进行内存分配时会首先从该数组指向的页中尝试进行分配，如果分配失败则调用 malloc_generic，在该函数中会遍历 pages 数组中对应大小的队列，此时如果对应的队列中有很多页均是满的，且队列很长那么每次分配的时候都会进行队列的遍历，导致性能的损失。

因此 mimalloc 构建了一个 Full List，将所有已经没有空闲空间的页放入该队列中，仅当该页中有一些空闲空间被释放后才会将其放回 pages 对应的队列中。而在由于内存的释放可能由对应堆的拥有者线程进行也可能由其他线程进行，因此需要一定的方式提醒对应的堆该页已经有空闲块了，同时为了避免使用锁导致的开销，mimalloc 通过加入一个 Thread Delayed Free List，如果一个页处于 Full List 中，那么在释放时会将内存块加入 Thread Delayed Free List 中，该队列会在调用 malloc_generic 时进行检测与清除(由于时 Thread Local 的堆，因此仅可能是拥有者来进行)，因此此时仅需通过原子操作即可完成。那么还有一个问题是当释放内存的时候，其他线程如何知道是将内存块加入 Thread Free List 中还是 Thread Delayed Free List 中。mimalloc 通过设置 NORMAL、DELAYED、DELAYING 三种状态来完成该操作。

Mimalloc 源码

堆的定义

```
// A heap owns a set of pages.
struct mi_heap_s {
  mi_tld_t*             tld;
  _Atomic(mi_block_t*)  thread_delayed_free;
  mi_threadid_t         thread_id;                           // thread this
heap belongs too
  mi_arena_id_t         arena_id;                            // arena id if
the heap belongs to a specific arena (or 0)
  uintptr_t             cookie;                              // random cookie
to verify pointers (see `_mi_ptr_cookie`)
  uintptr_t             keys[2];                             // two random
keys used to encode the `thread_delayed_free` list
  mi_random_ctx_t       random;                              // random number
context used for secure allocation
  size_t                page_count;                          // total number
of pages in the `pages` queues.
  size_t                page_retired_min;                    // smallest
retired index (retired pages are fully free, but still in the page queues)
  size_t                page_retired_max;                    // largest
retired index into the `pages` array.
  mi_heap_t*            next;                                // list of heaps
per thread
  bool                  no_reclaim;                          // `true` if this
heap should not reclaim abandoned pages
  uint8_t               tag;                                 // custom tag,
```

```
can be used for separating heaps based on the object types
  mi_page_t*            pages_free_direct[MI_PAGES_DIRECT];  // optimize:
array where every entry points a page with possibly free blocks in the
corresponding queue for that size.
  mi_page_queue_t       pages[MI_BIN_FULL + 1];              // queue of pages
for each size class (or "bin")
};
```

在 mimalloc 中，每个线程都有一个 Thread Local 的堆，每个线程在进行内存的分配时均从该线程对应的堆上进行分配。在一个堆中会有一个或多个 segment，一个 segment 会对应一个或多个页，而内存的分配就是在这些页上进行。mimalloc 将页分为三类：

small 类型的 segment 的大小为 4M，其负责分配大小小于 MI_SMALL_SIZE_MAX 的内存块，该 segment 中一个页的大小均为 64KB，因此在一个 segment 中会包含多个页，每个页中会有多个块 large 类型的 segment 的大小为 4M，其负责分配大小处于 MI_SMALL_SIZE_MAX 与 MI_LARGE_SIZE_MAX 之间的内存块，该 segment 中仅会有一个页，该页占据该 segment 的剩余所有空间，该页中会有多个块 huge 类型的 segment，该类 segment 的负责分配大小大于 MI_LARGE_SIZE_MAX 的内存块，该类 segment 的大小取决于需要分配的内存的大小，该 segment 中也仅包含一个页，该页中仅会有一个块

根据 heap 的定义我们可以看到其有 pages_free_direct 数组、pages 数组、Thread Delayed Free List 以及一些元信息。其中 pages_free_direct 数组中每个元素对应一个内存块大小的类别，其内容为一个指针，指向一个负责分配对应大小内存块的页，mimalloc 在分配比较小的内存时可以通过该数组直接找到对应的页，然后试图从该页上分配内存，从而提升效率。pages 数组中每个元素为一个队列，该队列中所有的页大小均相同，这些页可能来自不同的 segment，其中数组的最后一个元素(即 pages[MI_BIN_FULL])就是前文提到的 Full List，倒数第二个元素(即 pages[MIN_BIN_HUGE])包含了所有的 huge 类型的页。thread_delayed_free 就是前文提到的 Thread Delayed Free List，用来让线程的拥有者能够将页面从 Full List 中移除。

```
// Thread local data
struct mi_tld_s {
  unsigned long long  heartbeat;     // monotonic heartbeat count
  bool                recurse;       // true if deferred was called; used to
prevent infinite recursion.
  mi_heap_t*          heap_backing;  // backing heap of this thread (cannot be
deleted)
  mi_heap_t*          heaps;         // list of heaps in this thread (so we
can abandon all when the thread terminates)
  mi_segments_tld_t   segments;      // segment tld
  mi_os_tld_t         os;            // os tld
  mi_stats_t          stats;         // statistics
};
```

在 heap 的定义中我们需要特别注意的一个成员是 tld(即 Thread Local Data)。其成员包括指向对应堆的 heap_backing，以及用于 segment 分配的 segment tld 以及 os tld。

_mi_malloc_generic 的流程可以归纳为：

如果需要的话进行全局数据/线程相关的数据/堆的初始化 调用回调函数(即实现前文所说的 deferred free) 找到或分配新的页 从页中分配内存

初始化
前面我们提到过每个线程都有一个 Thread Local 的堆，该堆默认被设为 _mi_heap_empty。如果调用_mi_malloc_generic 时发现该线程的堆为 _mi_heap_empty 则进行初始化。mi_thread_init 会首先调用 mi_process_init 来进行进程相关数据的初始化，之后初始化 Thread Local 的堆。

在 mimalloc 中，如果一个线程结束了，那么其对应的 Thread Local 的堆就可以释放了，但是在该堆中还可能存在有一些内存块正在被使用，且此时会将对应的 segment 设置为 ABANDON，之后由其他线程来获取该 segment，之后利用该 segment 进行对应的内存分配与释放(mimalloc 也有一个 no_reclaim 的选项，设置了该选项的堆不会主动获取其他线程 ABANDON 的 segment)。

Huge 类型页面的分配
由于 huge 类型的页面对应的 segment 中仅有一个页，且该页仅能分配一个块，因此其会重新分配一个 segment，从中建立新的页面。mi_huge_page_alloc 会调用 mi_page_fresh_alloc 分配一个页面，然后将其插入堆对应的 BIN 中(即 heap->pages[MI_BIN_HUGE])。由下图可以看到 Small 与 Large 类型页面分配时所调用的 mi_find_free_page 也会调用该函数来进行页面的分配，接下来我们就介绍一下 mi_page_fresh_alloc。