


jemalloc源码解析-内存管理

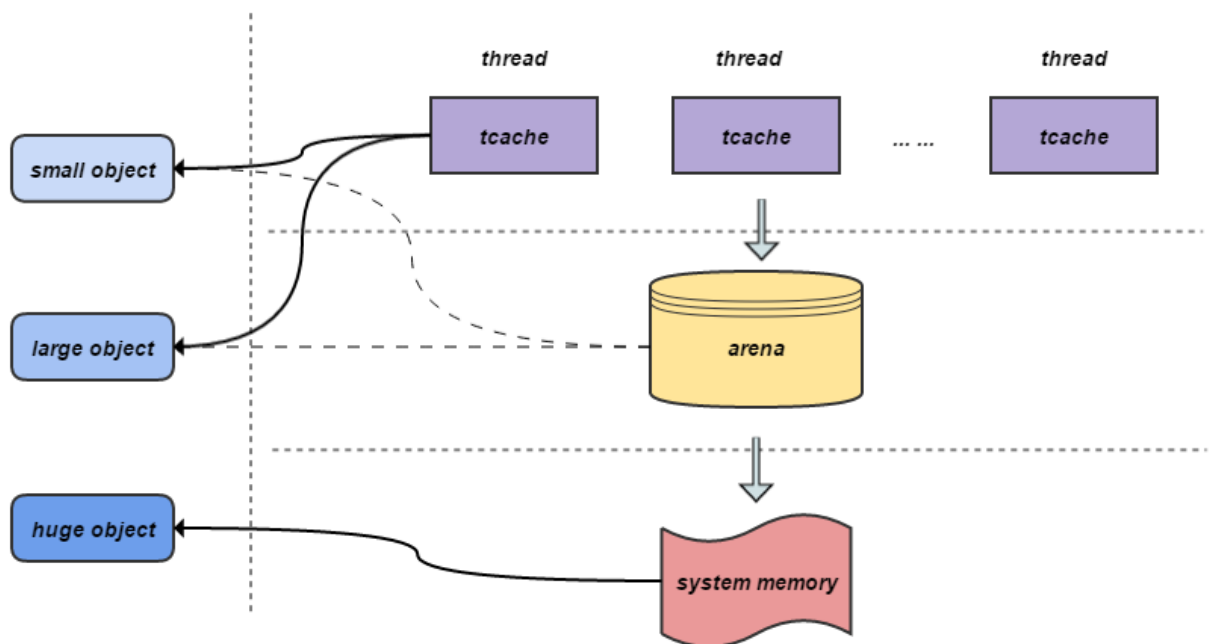
 brionas.github.io/2015/01/31/jemalloc源码解析-内存管理

发布者: [Renjian Qiu](#)

前文对jemalloc的核心架构做了详细介绍, 本文重点分析jemalloc的内存管理。jemalloc采用多级内存分配, 引入线程缓存tcache, 分配区arena来减少线程间锁的争用, 提高申请释放的效率和线程并发扩展性。前面提到, jemalloc根据内存对象的大小将其分为small object, large object和huge object, 本文将分别介绍这些内存对象的申请和释放流程。

结构分析:

先介绍一下内存管理的层级结构:



如图所示, jemalloc的内存管理采用层级架构, 分别是线程缓存tcache, 分配区arena和系统内存memory, 不同大小的内存块对应不同的分配区。每个线程对应一个tcache, 负责当前线程使用内存块的快速申请和释放, 避免线程间锁的竞争和同步。分配arena的具体结构在前文已经提到, 采用内存池的思想对内存区域进行合理的管理, 在有效保证低内存碎片的情况下实现不同大小内存块的高效管理。system memory是系统的内存区域。

- small object: 当jemalloc支持tcache时, small object的分配从tcache开始, tcache不中则从arena申请并将剩余区域缓存到tcache, 若从arena中不能分配再从system memory中申请chunk加入arena进行管理, 不支持tcache时, 则直接从arena中申请。
- large object: 当jemalloc支持tcache时, 如果large object的size小于tcache_maxclass, 则从tcache开始分配, tcache不中则从arena申请, 只申请需要的内存块, 不做多余cache, 若从arena中不能分配则从system memory中申请。当large object的size大于tcache_maxclass或者jemalloc不支持tcache时, 直接从arena中申请。
- huge object: 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。

接下来我们基于代码进行详细介绍。

内存分配分析：

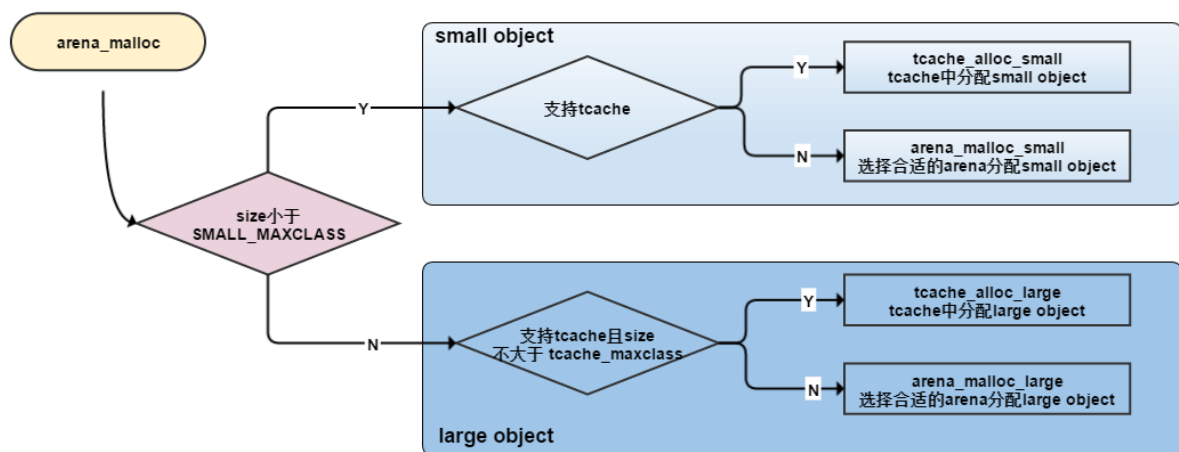
内存分配函数的入口位于：je_malloc/MALLOC_BODY/imalloc/imalloct

```

1. JEMALLOC_ALWAYS_INLINE void *
2. imalloc(size_t size, bool try_tcache, arena_t *arena)
3. {
4.     assert(size != 0);
5.     if (size <= arena_maxclass)
6.         /*这里进行small object或large object的分配*/
7.         return (arena_malloc(arena, size, false, try_tcache));
8.     else
9.         /*这里进行huge object的分配*/
10.        return (huge_malloc(size, false, huge_dss_prec_get(arena)));
11. }

```

这里把huge object的申请和其他对象进行的分流，条件是arena_maxclass，其中small object和large object的分配需要通过tcache和arena进行分配，如图：



下面我们将分成以下几个部分介绍jemalloc内存分配：

- tache中分配small object
- arena中分配small object
- tache中分配large object
- arena中分配large object
- 分配huge object

这里我们先关注tcache中small object的分配，这里主要要注意的是当tcache不中时，需要调用arena_tcache_fill_small从arena中申请整块run进行分配，run剩余部分挂靠在tcache中使用。

```

1. JEMALLOC_ALWAYS_INLINE void *
2. tcache_alloc_small(tcache_t *tcache, size_t size, bool zero)
3. {
4.     ...
5.     binind = SMALL_SIZE2BIN(size);
6.     assert(binind < NBINS);
7.     /*定位到对应的tbin*/
8.     tbin = &tcache->tbins[binind];
9.     size = arena_bin_info[binind].reg_size;
10.    /如果tbin中有cache对象, 直接返回/
11.    ret = tcache_alloc_easy(tbin);
12.    if (ret == NULL) {
13.        /*如果tbin对应cache为空, 则从对应arena中提取run填充cache并返回, 填充操作通过
        arena_tcache_fill_small完成*/
14.        ret = tcache_alloc_small_hard(tcache, tbin, binind);
15.        if (ret == NULL)
16.            return (NULL);
17.    }
18.    ...
19. }

```

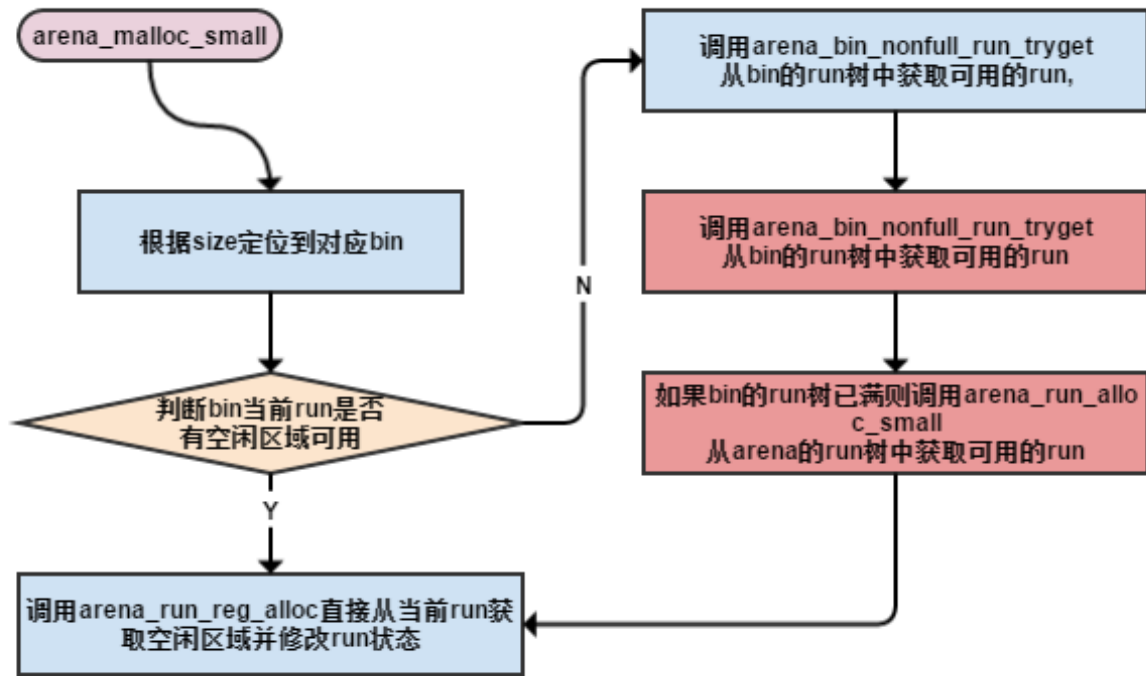
arena中small object的分配:

```

1. void * arena_malloc_small(arena_t *arena, size_t size, bool zero)
2. {
3.     binind = SMALL_SIZE2BIN(size);
4.     assert(binind < NBINS);
5.     /*定位到arena中对应的bin*/
6.     bin = &arena->bins[binind];
7.     size = arena_bin_info[binind].reg_size;
8.     malloc_mutex_lock(&bin->lock);
9.     /*从bin取出一个可用的run进行内存块的分配中*/
10.    if ((run = bin->runcur) != NULL && run->nfree > 0)
11.        /*若bin中当前run中有空闲内存块可使用, 直接返回*/
12.        ret = arena_run_reg_alloc(run, &arena_bin_info[binind]);
13.    else
14.        /*若runcur已满, 则从当前bin的runs (按地址排序的红黑树) 中选择地址最低的可用run*/
15.        ret = arena_bin_malloc_hard(arena, bin);
16.    ...
17.    return (ret);
18. }

```

arena中small object的分配涉及到run的管理和分配, 空闲的run红黑树顺序的调整已经run中bitmap状态的改变, 见下图:



tcache中large object的分配与small object的分配流程类似，不同的是定位的tbin和处理cache不中的方法不同：

- 分配small object时, $\text{binind} = \text{small_size2bin}[(s-1) \gg \text{LG_TINY_MIN}]$, 在cache不命中时, 会从arena得到多个small objects进行冗余cache。
- 分配large object时, $\text{binind} = \text{NBINS} + (\text{size} \gg \text{LG_PAGE}) - 1$, 在tcache不命中时, 考虑到large object内存较大, 做冗余备份过于浪费, 只从arena申请一个对象。

```

1. JEMALLOC_ALWAYS_INLINE void *
2. tcache_alloc_large(tcache_t *tcache, size_t size, bool zero)
3. {
4.     ...
5.     size = PAGE_CEILING(size);
6.     assert(size <= tcache_maxclass);
7.     binind = NBINS + (size >> LG_PAGE) - 1;
8.     assert(binind < nhbins);
9.     /*定位到对应的tbin*/
10.    tbin = &tcache->tbins[binind];
11.    /*如果对应的tbin中有cache对象, 直接返回*/
12.    ret = tcache_alloc_easy(tbin);
13.    if (ret == NULL) {
14.        /* 对应的tbin中无cache时, 直接从arena中申请一个large object, 不像small
15.        object一次申请多个做cache */
16.        ret = arena_malloc_large(tcache->arena, size, zero);
17.        if (ret == NULL)
18.            return (NULL);
19.    }
20.    return (ret);
21. }

```

arena中large object的分配较为简单, 不通过bin, 直接从对应chunk中申请相应大小的对象并通过run管理返回。

```

1. void * arena_malloc_large(arena_t *arena, size_t size, bool zero)
2. {
3.     void *ret;
4.     UNUSED bool idump;
5.     /* Large allocation. */
6.     size = PAGE_CEILING(size);
7.     malloc_mutex_lock(&arena->lock);
8.     /*arena中large object不通过bin进行管理, 直接从对应chunk中申请相应大小的对象并通过
run管理返回.*/
9.     ret = (void *)arena_run_alloc_large(arena, size, zero);
10.    ...
11.    return (ret);
12. }

```

所有的huge object通过一棵与arena独立的红黑树进行管理:

```

1. void *
2. huge_palloc(size_t size, size_t alignment, bool zero, dss_prec_t dss_prec)
3. {
4.     ...
5.     /*构造一个红黑树节点来管理相应的large object*/
6.     node = base_node_alloc();
7.     if (node == NULL)
8.         return (NULL);
9.     /*申请足够的chunk并挂靠到前面的红黑树节点下面
10.    */
11.     is_zeroed = zero;
12.     ret = chunk_alloc(csize, alignment, false, &is_zeroed, dss_prec);
13.     if (ret == NULL) {
14.         base_node_dealloc(node);
15.         return (NULL);
16.     }
17.     /* Insert node into huge. */
18.     node->addr = ret;
19.     node->size = csizer;
20.     malloc_mutex_lock(&huge_mtx);
21.     /*把对应的红黑树节点插入红黑树*/
22.     extent_tree_ad_insert(&huge, node);
23.     ...
24.     return (ret);
25. }

```

内存释放分析:

接下来介绍内存释放的过程, 刚好与内存申请对应, 内存释放函数的入口位于:

je_free/ifree/iqualloc/iqualloct/idalloct,

```

1. JEMALLOC_ALWAYS_INLINE void
2. idalloct(void *ptr, bool try_tcache)
3. {
4.     arena_chunk_t *chunk;
5.     assert(ptr != NULL);
6.     /*基于small object和large object的大小小于chunk,判断对象指针指向对类型 */
7.     chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(ptr);
8.     if (chunk != ptr)
9.         /*释放small object或large object*/
10.        arena_dalloc(chunk->arena, chunk, ptr, try_tcache);
11.     else
12.         /*释放huge object*/
13.        huge_dalloc(ptr, true);
14. }

1. JEMALLOC_ALWAYS_INLINE void
2. arena_dalloc(arena_t *arena, arena_chunk_t *chunk, void *ptr, bool try_tcache)
3. {
4.     ...
5.     if ((mapbits & CHUNK_MAP_LARGE) == 0) {
6.         /* Small allocation. */
7.         if (try_tcache && (tcache = tcache_get(false)) != NULL) {
8.             size_t binind;
9.             binind = arena_ptr_small_binind_get(ptr, mapbits);
10.            /*tcache中释放small object*/
11.            tcache_dalloc_small(tcache, ptr, binind);
12.        } else
13.            /*arena中释放small object*/
14.            arena_dalloc_small(arena, chunk, ptr, pageind);
15.    } else {
16.        size_t size = arena_mapbits_large_size_get(chunk, pageind);
17.        assert(((uintptr_t)ptr & PAGE_MASK) == 0);
18.        if (try_tcache && size <= tcache_maxclass && (tcache =
19.            tcache_get(false)) != NULL) {
20.            /*tcache中释放large object*/
21.            tcache_dalloc_large(tcache, ptr, size);
22.        } else
23.            /*arena中释放large object*/
24.            arena_dalloc_large(arena, chunk, ptr);
25.    }
26. }

```

内存释放的流程刚好与内存分配对应起来，根据对象的类型和来源的不同考虑不同的释放方法，这里将分为以下几个部分介绍：

- tcache中释放对象
- arena中释放small object
- arena中释放large object
- 释放huge object

tcache中释放small object和large object的流程大致相同，首先定位到对应的tbin, 如果当前tbin的缓存已满，则调用tcache_bin_flush_xxx清除部分缓存，然后把当前释放的object挂靠到缓存列表下面。

```

1. JEMALLOC_ALWAYS_INLINE void
2. tcache_dalloc_xxx(tcache_t *tcache, void *ptr, size_t size)
3. {
4.     ...
5.     /*定位到tbin*/
6.     tbin = &tcache->tbins[binind];
7.     tbin_info = &tcache_bin_info[binind];
8.     /*若cache满了则flush掉一半cache对象*/
9.     if (tbin->ncached == tbin_info->ncached_max) {
10.         tcache_bin_flush_xxx(tbin, binind, (tbin_info->ncached_max >>
11.             1), tcache);
12.     }
13.     assert(tbin->ncached < tbin_info->ncached_max);
14.     /*把要释放的对象插入bin对应cache数组, 并更新数据*/
15.     tbin->avail[tbin->ncached] = ptr;
16.     tbin->ncached++;
17. }

```

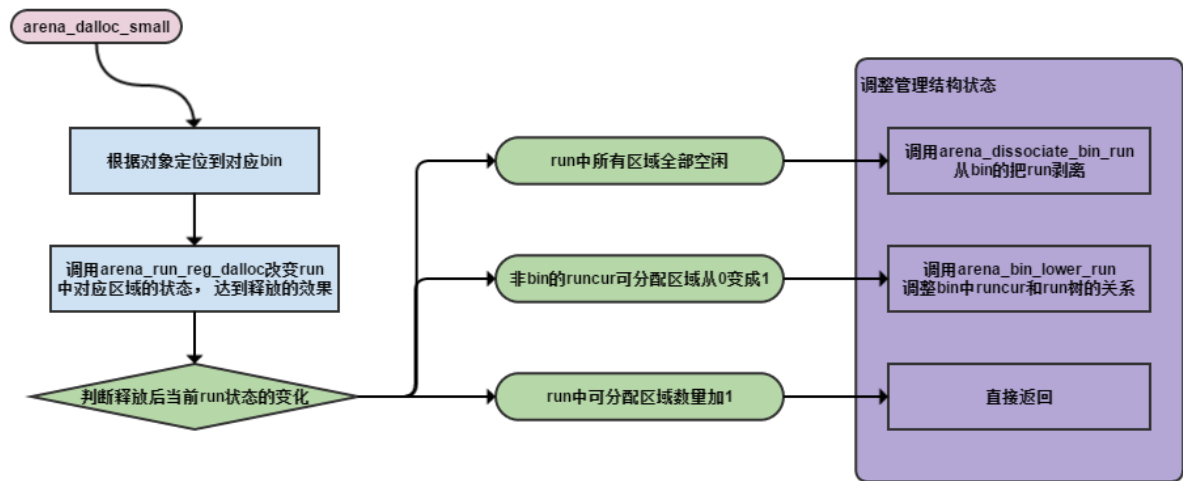
arena中small object的释放: arena_dalloc_small/arena_dalloc_bin/arena_dalloc_bin_locked:

```

1. void
2. arena_dalloc_bin_locked(arena_t *arena, arena_chunk_t *chunk, void *ptr,
3.     arena_chunk_map_t *mapelm)
4. {
5.     ...
6.     pageind = ((uintptr_t)ptr - (uintptr_t)chunk) >> LG_PAGE;
7.     /*定位到对应的run*/
8.     run = (arena_run_t *)((uintptr_t)chunk + (uintptr_t)((pageind -
9.         arena_mapbits_small_runind_get(chunk, pageind)) << LG_PAGE));
10.    bin = run->bin;
11.    binind = arena_ptr_small_binind_get(ptr, mapelm->bits);
12.    bin_info = &arena_bin_info[binind];
13.    ...
14.    /*改变run内部对应分配区域的状态*/
15.    arena_run_reg_dalloc(run, ptr);
16.    /*针对run状态的变化对bin进行调整*/
17.    if (run->nfree == bin_info->nregs) {
18.        /*run有只是用一个到变空的状态变化*/
19.        arena_dissociate_bin_run(chunk, run, bin);
20.        arena_dalloc_bin_run(arena, chunk, run, bin);
21.    } else if (run->nfree == 1 && run != bin->runcur)
22.        /*run由full变成空闲一个的状态变化*/
23.        arena_bin_lower_run(arena, chunk, run, bin);
24.    ...
25. }

```

arena中small object的释放和申请一样会造成所在run空闲状态的变化, 进而导致对应bin和arena结构的调整, 稍微复杂一些, 如图:



arena的large object的释放跟bin没有关系，只需要把占用的内存页还给arena并更新arena的状态就可以:arena_dalloc_large/arena_dalloc_large_locked:

```

1. void arena_dalloc_large_locked(arena_t *arena, arena_chunk_t *chunk, void *ptr)
2. {
3.     /*更新arena的run的状态*/
4.     arena_run_dalloc(arena, (arena_run_t *)ptr, true, false);
5. }
  
```

huge object的释放就是红黑树的删除节点的操作并释放相关内存。

```

1. void huge_dalloc(void *ptr, bool unmap)
2. {
3.     ...
4.     /* Extract from tree of huge allocations. */
5.     key.addr = ptr;
6.     /*找到对应红黑树节点*/
7.     node = extent_tree_ad_search(&huge, &key);
8.     assert(node != NULL);
9.     assert(node->addr == ptr);
10.    /*从红黑树中删除节点*/
11.    extent_tree_ad_remove(&huge, node);
12.    ...
13.    /*释放相关内存*/
14.    chunk_dealloc(node->addr, node->size, unmap);
15.    base_node_dealloc(node);
16. }
  
```

小结:

jemalloc的内存分配就介绍完毕，这里做一个简单的总结：

- jemalloc引入线程缓存tcache, 分配区arena来减少线程间锁的争用, 保证线程并发扩展性的同时实现了内存的快速申请释放
- 采用arena管理不同大小的内存对象在保证内存高效管理的同时减少了内存碎片
- 引入红黑树管理空闲run和chunk, 相比链表具有了更高的效率
- 在run中采用bitmap管理可分配区域来实现管理和数据分离, 且能够更快地定位到空闲区域
- 引入了多层cache, 基于内存池的思想, 虽然增加了内存占用, 但实现了内存的快速申请释放, 除了tcache, 还有bin中的runcur, arena中的spare等。

