jemalloc 源码分析

jemalloc (https://github.com/jemalloc/jemalloc) 是一个通用的 `malloc(3)` 实现，着重于减少内存碎片和提高并发性能，在许多项目中都有用到，比如 `Rust` 和 `Redis`。因为在做 `Redis` 相关的工作，就看了一下源码，记录一下。

背景知识

内存的来源

关于 `Linux` 中虚拟内存管理和进程地址空间就不再赘述了，可以查看我之前的笔记 (https://youjiali1995.github.io/study/study/#%E8%BF%9B%E7%A8%8B%E8%99%9A%E6%8B%9F%E5%9C%B0%E5%9D%80%E7%A9%BA%E9%97%B4)。

`Linux` 提供了几个系统调用用于分配内存:

- `brk()`: 调整 `program break`，改变 `data segment` 的大小。
- `mmap()`: 在进程的虚拟地址空间中创建新的内存映射。内存分配器一般使用该系统调用创建**私有匿名映射**分配内存，内核会以 `page size` 大小倍数来分配，`page size` 一般为 `4096` 字节。

对应的释放内存也有几种方式:

- `brk()`: 既可以增大也可以缩小。
- `munmap()`: 解除映射。
- `madvise()`: 该系统调用会告诉操作系统这块内存之后会如何使用，由操作系统进行处理。释放时，会使用 `MADV_DONTNEED` 或 `MADV_FREE`:

```
MADV_DONTNEED
          Do not expect access in the near future.  (For the time being,
          the application is finished with the given range, so the
          kernel can free resources associated with it.)

          After a successful MADV_DONTNEED operation, the semantics of
          memory access in the specified region are changed: subsequent
          accesses of pages in the range will succeed, but will result
          in either repopulating the memory contents from the up-to-date
          contents of the underlying mapped file (for shared file
          mappings, shared anonymous mappings, and shmem-based
          techniques such as System V shared memory segments) or zero-
          fill-on-demand pages for anonymous private mappings.

          Note that, when applied to shared mappings, MADV_DONTNEED
          might not lead to immediate freeing of the pages in the range.
          The kernel is free to delay freeing the pages until an
          appropriate moment.  The resident set size (RSS) of the
          calling process will be immediately reduced however.

          MADV_DONTNEED cannot be applied to locked pages, Huge TLB
          pages, or VM_PFNMAP pages.  (Pages marked with the kernel-
          internal VM_PFNMAP flag are special memory areas that are not
          managed by the virtual memory subsystem.  Such pages are
          typically created by device drivers that map the pages into
          user space.)
```

可以看出来，对于**私有匿名映射**会立即释放，而在之后再次访问这块内存时，不需要重新分配。

False cache line sharing

为了减少存储器访问延迟，`CPU` 中会有本地 `Cache`，`Cache` 被划分为 `cache line`，大小一般为 `64B`。`CPU` 访问内存时，会首先将内存缓存在 `cache line` 中。 在多处理器系统中，每个 `CPU` 都有自己的本地 `Cache`，会导致数据多副本，也就带来了一致性问题：多个 `CPU` 的 `cache line` 中有相同地址的内存。需要实现 `Cache Coherence Protocol` 来解决 这个问题。现代处理器一般使用 `MESI` 协议实现 `Cache Coherence`，这会带来通讯耗时、总线压力、导致 `cache line` 的抖动，影响性能。

避免这个问题主要有下面几个方法：

1. `__declspec (align(64))`：变量起始地址按 `cache line` 对齐
2. 当使用数组或结构体时，不仅需要起始地址对齐，还需要 `padding`，使得数组元素或结构体大小为 `cache line` 倍数
3. 避免多线程使用相近地址的内存，多使用局部变量

内存着色

现代 `CPU` 的 `Cache` 映射策略有很多，如组相联、全相联、直接相联。不同地址的内存有可能映射到相同的 `cache line` (主要发生在地址对齐的情况，如不同对象的地址按照 `page size` 对齐)，如果频繁交替访问映射到相同 `cache line` 的内存，就会造成 `cache line` 的颠簸。 内存着色通过给对象地址增加 `cache line` 大小倍数的偏移，从而映射到不同的 `cache line`，来避免上面的问题。

为什么需要内存分配器

因为 `mmap()` 按照 `page size` 进行分配，一般是 `4096` 字节，若每次分配时都调用一次会造成极大的内存浪费，并且性能不好。若由程序员自己管理 `page`，容易出错且性能不好，所以 `glibc` 中提供了标准 `malloc(3)` 供 程序员使用。

内存分配器的目标

内存分配器的目标主要有2个：

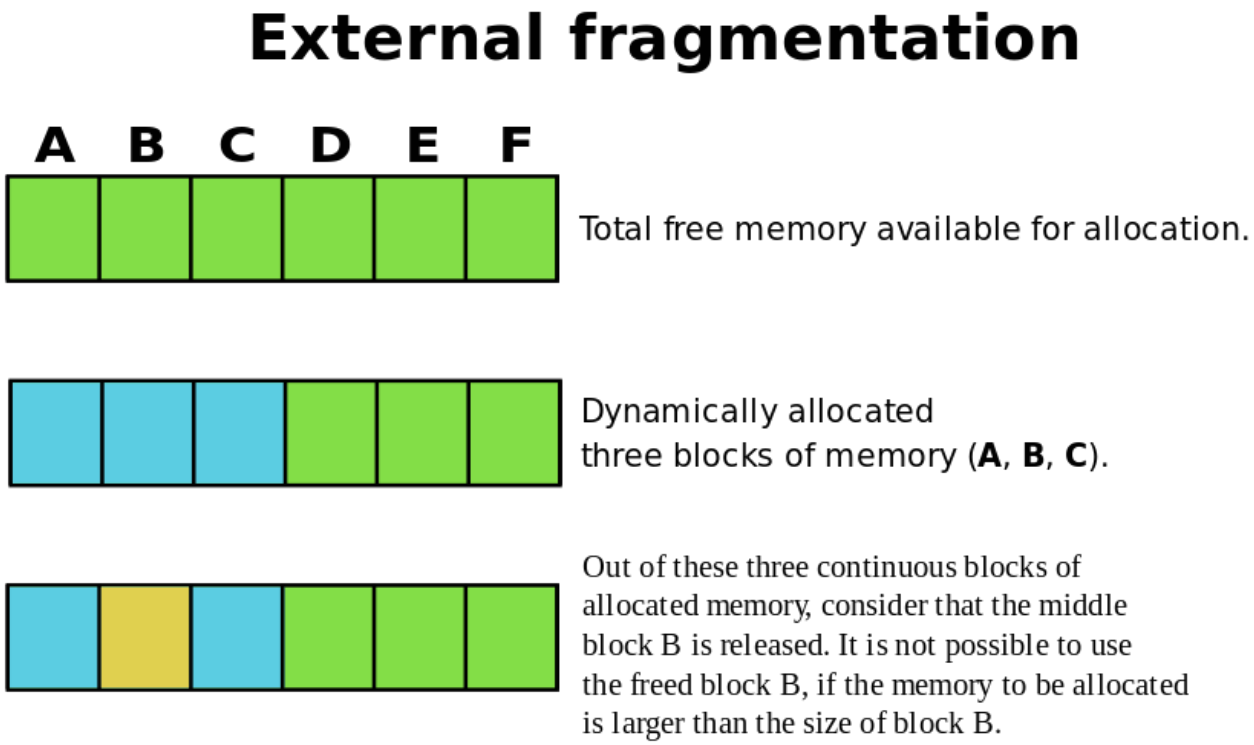
1. 减少内存碎片，包括内部碎片和外部碎片：
    - 内部碎片：分配出去的但没有使用到的内存，比如需要 `32` 字节，分配了 `40` 字节，多余的 `8` 字节就是内部碎片。
    - 外部碎片：大小不合适导致无法分配出去的内存，比如一直申请 `16` 字节的内存，但是内存分配器中保存着部分 `8` 字节的内存，一直分配不出去。
2. 提高性能:
    - 单线程性能
    - 多线程性能

`jemalloc` 和 `tcmalloc` 都是对 `glibc` 中的优化，目的也是为了减少内存碎片和提高性能。

常用内存分配器算法

Dynamic memory allocation

首先分配一整块内存，然后按需从这块内存中分配。一般会在分配出的内存前面保存 `metadata`，还会维护 `freelist` 用于查找空闲内存。但这会导致比较严重的外部碎片：



Buddy memory allocation

以 `Binary buddy algorithm` 为例: 同样从一块内存中分配，但此时不是按需分配大小，而是这块内存不断分成一半，直到到达目标大小或者下界。在释放的时候，会和之前分裂的且空闲的进行合并。 一般会用有序结构如红黑树，来存储不同大小的 `buddy block`，这样分配和合并时可以快速查找合适的内存。

这种算法能够有效减少外部碎片，但内部碎片很严重， `Binary buddy algorithm` 最严重会带来 `50%` 的内部碎片。

Slab allocation

对象的初始化和释放往往比内存的分配和释放代价大，基于此发明了 `slab` 。 `slab` 会提前分配一块内存，然后将这块连续内存划分为大小相同的 `slots`，使用相应的数据结构记录每个 `slots` 的分配状况，如 `bitmap` 。 当需要分配时，就查找对应大小的 `slab`，分配出一个空闲 `slot`，而释放时就是把这个 `slot` 标记为空闲。

`slab` 的 `size classes` 影响碎片的产生，需要精心选择:

- `size classes` 太稀疏会导致内部碎片
- `size classes` 太密集又会导致外部碎片

jemalloc 源码分析

`Redis` 一般不使用 `glibc` 中默认的内存分配器，在编译时可以指定使用自带的 `jemalloc`，版本为 `4.0.3`，编译参数如下:

```
./configure --with-lg-quantum=3 --with-jemalloc-prefix=je_ --enable-cc-silence CFLAGS="-std=gnu99 -Wall -
pipe -g3 -O3 -funroll-loops " LDFLAGS=""

1. --with-lg-quantum=<lg-quantum>: Base 2 log of minimum allocation alignment. 8字节对齐
2. --with-jemalloc-prefix=<prefix>: Prefix to prepend to all public APIs.
3. --disable-cc-silence: Do not silence irrelevant compiler warnings.
```

`jemalloc` 可以在编译时配置也支持运行时配置，配置项可以查看文档，可配置的有 `page size`、`chunksize`、`quantum` 等。配置支持 4 种方式:

- `/etc/malloc.conf` 符号链接
- `MALLOC_OPTIONS` 环境变量
- `_malloc_options` 全局变量
- `je_mallctl()` 在代码里进行配置

这次看的 `jemalloc` 源码是 `Redis` 附带的 `4.0.3` 版本，使用编译时的默认配置，只关注内存分配释放相关的逻辑，忽略其他功能，如 `profile`、`valgrind` 等。会按照我 理解的方式一层层进行分析。

page

最底层是从操作系统申请内存，由 `pages.h/pages.c` 封装了跨平台实现， `Linux` 中使用 `mmap(2)`。主要关注下面几个函数:

1. `pages_map()`: 调用 `mmap()` 分配可读可写、私有匿名映射。
2. `pages_unmap()`: 调用 `mummap()` 删除指定范围的映射。
3. `pages_trim()`: `trim` 头尾部分的内存映射，用于内存对齐。
4. `pages_purge()`: 调用 `madvise()` 清除(purge)部分内存页，也就是释放。

`mmap()` 会以 `page size` 的倍数分配内存，匿名映射会初始化为0，私有映射采用 `COW` 策略。

chunk

每当内存不够用了， `jemalloc` 会以 `chunk` 为单位从操作系统申请内存，大小为 `page size` 倍数，默认为 `2 MiB`，分配的函数为 `chunk_alloc_mmap()`。 `chunk_alloc_map()` 会调用 `pages_map()` 分配地址按 `chunk_size` 对齐的内存，既可以避免 `false cache line sharing`，也可以在常量时间内得到起始地址。 但是 `pages_map()` 不能保证对齐，首先会调用 `pages_map()` 分配一块内存查看是否对齐，若没对齐，会重新多分配一些内存，然后调用 `pages_trim()` 截取两端使内存对齐，所以可能会有多次 `mmap()` 和 `munmap()` 的过程。

`chunk` 分配出来需要进行管理，每个 `chunk` 会分配一个头部 `extent_node_t` 记录其中的信息，如:

- `en_arena`: 负责该 `chunk` 的 `arena` (后面介绍)。
- `en_addr`: 该 `chunk` 的起始地址。

`chunk` 分配出来会插入到 `chunks_rtree` (`radix tree`) 中，保存 `chunk` 地址到 `extent_node_t` 的映射，以便能快速从地址找到 `node`，方便后面 `huge object` 的释放。

base

`jemalloc` 不可能只使用栈空间或全局变量，内部也需要动态分配一些内存。`base.h/base.c` 实现了内部使用的内存分配器。

`base` 通过地址对齐和 `padding` 避免 `false cache line sharing`： `chunk` 会按照 `chunksize` 地址对齐，且分配的大小会 `padding` 到 `cache line` 大小倍数。

`base` 以 `chunk` 为单位申请内存，记录 `chunk` 信息的 `extent_node_t` 使用 `chunk` 的起始内存:

```
static extent_node_t *
base_chunk_alloc(size_t minsize)
{
        extent_node_t *node;
        size_t csize, nsize;
        void *addr;

        assert(minsize != 0);
        /* 尝试从 base_nodes 中复用 node */
        node = base_node_try_alloc();
        /* Allocate enough space to also carve a node out if necessary. */
        // 需要分配的 node 的内存
        nsize = (node == NULL) ? CACHELINE_CEILING(sizeof(extent_node_t)) : 0;
        // 多分配 node size，也按照 chunk size 对齐
        csize = CHUNK_CEILING(minsize + nsize);
        // 内部调用 chunk_alloc_mmap()
        addr = chunk_alloc_base(csize);
        if (addr == NULL) {
                if (node != NULL)
                        base_node_dalloc(node);
                return (NULL);
        }
        base_mapped += csize;
        if (node == NULL) {
            // 使用 chunk 的起始内存
                node = (extent_node_t *)addr;
                addr = (void *)((uintptr_t)addr + nsize);
                csize -= nsize;
                if (config_stats) {
                        base_allocated += nsize;
                        base_resident += PAGE_CEILING(nsize);
                }
        }
        extent_node_init(node, NULL, addr, csize, true, true);
        return (node);
}
```

`base` 使用 `extent_node_t` 组成的红黑树 `base_avail_szad` 管理 `chunk`。每次需要分配时，会从红黑树中查找内存大小相同或略大的、地址最低的 `node`， 然后从 `node` 负责的 `chunk` 中分配内存，剩下的内存会继续由该 `node` 负责，修改大小和地址后再次插入到红黑树中；若该 `node` 负责的内存全部分配完了，会将该 `node` 添加到链表头 `base_nodes`，留待后续分配时复用。当没有合适的 `node` 时，会新分配 `chunk` 大小倍数的内存，由 `node` 负责，这个 `node` 优先从链表 `base_nodes` 中分配，也可能是新分配的连续内存的起始位置构成的 `node`。

`base_alloc()`：从 `base_avail_szad` 中查找大小相同或略大的、地址最低的 `extent_node_t`，再从 `chunk` 里分配内存。如果没有合适的内存，会先调用 `base_chunk_alloc()` 分配 `chunk` 大小倍数的内存，返回负责这块内存的 `node`，然后进行分配。

```
ret = extent_node_addr_get(node); /* node 中用于分配内存的起始地址 */
if (extent_node_size_get(node) > csize) {
    extent_node_addr_set(node, (void *)((uintptr_t)ret + csize)); /* 起始地址增加 csize，表明之前的内存被分配出去
*/
    extent_node_size_set(node, extent_node_size_get(node) - csize); /* 内存大小减少 */
    extent_tree_szad_insert(&base_avail_szad, node); /* 按照大小、地址顺序插入到红黑树 */
} else
    /* 这种情况只发生在 extent_node_size_get(node) == csize 这种情况。
     * 此时该 node 负责的内存已经全部分配了，会将该 node 插入到一个链表中去，备用。
     * 该链表用嵌入式实现，在 node 的起始内存存放下一个 node 的地址，节省空间 */
    base_node_dalloc(node);
```

为了减少内存浪费，`base_nodes` 链表缓存了之前分配的 `extent_node_t`，`base_nodes` 指向链表头，`base_node_dalloc()` 将 `node` 添加到表头，而 `base_node_try_alloc()` 移除表头。采用嵌入式实现，比较晦涩:

```
static extent_node_t *
base_node_try_alloc(void)
{
        extent_node_t *node;

        if (base_nodes == NULL)
                return (NULL);
        // 返回链表头
        node = base_nodes;
        // base_nodes 指向下一个 node
        base_nodes = *(extent_node_t **)node;
        JEMALLOC_VALGRIND_MAKE_MEM_UNDEFINED(node, sizeof(extent_node_t));
        return (node);
}

/* base_mtx must be held. */
static void
base_node_dalloc(extent_node_t *node)
{

        JEMALLOC_VALGRIND_MAKE_MEM_UNDEFINED(node, sizeof(extent_node_t));
        // 将 base_nodes 指向的地址保存在 node 的指向的内存起始处
        // 形成一个 node 的链表，base_nodes 指向链表头，内存起始处为
        // 下一个 node 的地址
        *(extent_node_t **)node = base_nodes;
        base_nodes = node;
}
```

arena

`arena` 是 `jemalloc` 中最重要的部分，内存大多数由 `arena` 管理，分配算法是 `Buddy allocation` 和 `Slab allocation` 的组合:

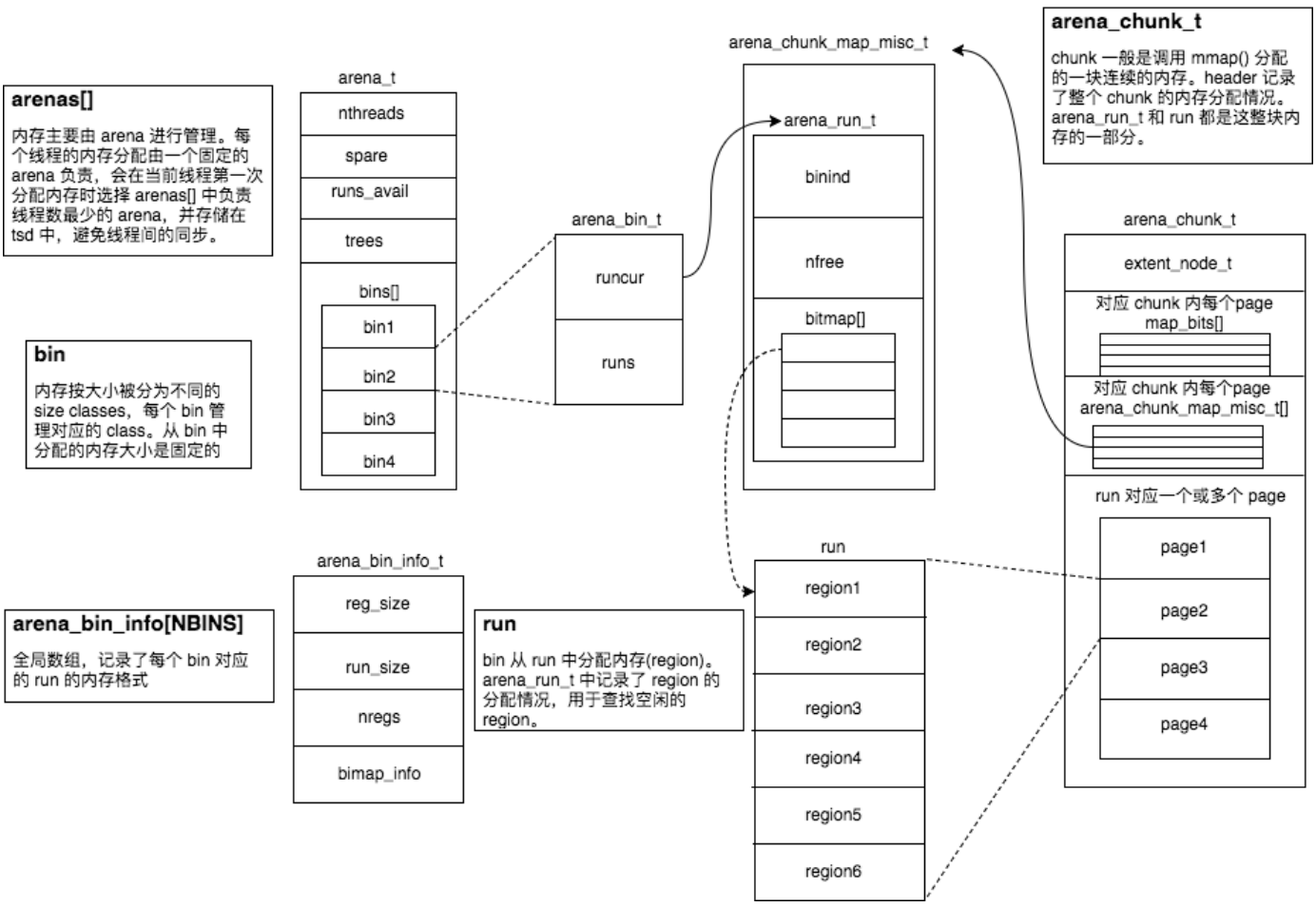
1. `chunk` 使用 `Buddy allocation` 划分为不同大小的 `run`。
2. `run` 使用 `Slab allocation` 划分为固定大小的 `region`，大部分内存分配直接查找对应的 `run`，从中分配空闲的 `region`，释放就是标记 `region` 为空闲。
3. `run` 被释放会和空闲的、相邻的 `run` 进行合并。当合并为整个 `chunk` 时，若发现有相邻的空闲 `chunk`，也会进行合并。

```
struct arena_s {
        /* This arena's index within the arenas array. */
        unsigned                ind;

        /*
         * Number of threads currently assigned to this arena.  This field is
         * protected by arenas_lock.
         */
        unsigned                nthreads;

        /*
         * There are three classes of arena operations from a locking
         * perspective:
         * 1) Thread assignment (modifies nthreads) is protected by arenas_lock.
         * 2) Bin-related operations are protected by bin locks.
         * 3) Chunk- and run-related operations are protected by this mutex.
         */
        malloc_mutex_t    lock;

        arena_stats_t      stats;
        /*
         * List of tcaches for extant threads associated with this arena.
         * Stats from these are merged incrementally, and at exit if
         * opt_stats_print is enabled.
         */
        ql_head(tcache_t)       tcache_ql;

        uint64_t                prof_accumbytes;

        /*
         * PRNG state for cache index randomization of large allocation base
         * pointers.
         */
        uint64_t                offset_state;

        dss_prec_t            dss_prec;

        /*
         * In order to avoid rapid chunk allocation/deallocation when an arena
         * oscillates right on the cusp of needing a new chunk, cache the most
         * recently freed chunk.  The spare is left in the arena's chunk trees
         * until it is deleted.
         *
         * There is one spare chunk per arena, rather than one spare total, in
         * order to avoid interactions between multiple threads that could make
         * a single spare inadequate.
         */
        arena_chunk_t      *spare;

        /* Minimum ratio (log base 2) of nactive:ndirty. */
        ssize_t          lg_dirty_mult;

        /* True if a thread is currently executing arena_purge(). */
        bool                purging;

        /* Number of pages in active runs and huge regions. */
        // 已经分配出的 page 个数
```

```
    size_t           nactive;

    /*
     * Current count of pages within unused runs that are potentially
     * dirty, and for which madvise(... MADV_DONTNEED) has not been called.
     * By tracking this, we can institute a limit on how much dirty unused
     * memory is mapped for each arena.
     */
    // runs_dirty 中的page数目(包含 chunk)
    size_t           ndirty;

    /*
     * Size/address-ordered tree of this arena's available runs.  The tree
     * is used for first-best-fit run allocation.
     */
    // 红黑树
    arena_avail_tree_t      runs_avail;

    /*
     * Unused dirty memory this arena manages.  Dirty memory is conceptually
     * tracked as an arbitrarily interleaved LRU of dirty runs and cached
     * chunks, but the list linkage is actually semi-duplicated in order to
     * avoid extra arena_chunk_map_misc_t space overhead.
     *
     *   LRU-----------------------------------------------------------MRU
     *
     *        /-- arena ---\
     *        |            |
     *        |            |
     *        |------------|                             /- chunk -\
     *   ...->|chunks_cache|<--------------------------->|  /----\ |<--...
     *        |------------|                             |  |node| |
     *        |            |                             |  |    | |
     *        |            |    /- run -\    /- run -\   |  |    | |
     *        |            |    |       |    |       |   |  |    | |
     *        |            |    |       |    |       |   |  |    | |
     *        |------------|    |-------|    |-------|   |  |----| |
     *   ...->|runs_dirty  |<-->|rd     |<-->|rd     |<----->|rd  |<----...
     *        |------------|    |-------|    |-------|   |  |----| |
     *        |            |    |       |    |       |   |  |    | |
     *        |            |    |       |    |       |   |  \----/ |
     *        |            |    \-------/    \-------/   |         |
     *        |            |                             |         |
     *        |            |                             |         |
     *        \------------/                             \---------/
     */
    // 空闲的 dirty run 会存在这，用于 purge
    arena_runs_dirty_link_t runs_dirty;
    // 都是 runs_dirty 中存在的，是为了保存脏的 chunk
    extent_node_t       chunks_cache;

    /* Extant huge allocations. */
    ql_head(extent_node_t)  huge;
    /* Synchronizes all huge allocation/update/deallocation. */
    malloc_mutex_t    huge_mtx;

    /*
     * Trees of chunks that were previously allocated (trees differ only in
```

```
         * node ordering).  These are used when allocating chunks, in an attempt
         * to re-use address space.  Depending on function, different tree
         * orderings are needed, which is why there are two trees with the same
         * contents.
         */
        // 用于复用 chunk
        // 2种树的内容一样，order 不同
        extent_tree_t      chunks_szad_cached;
        extent_tree_t      chunks_ad_cached;
        extent_tree_t      chunks_szad_retained;
        extent_tree_t      chunks_ad_retained;

        malloc_mutex_t    chunks_mtx;
        /* Cache of nodes that were allocated via base_alloc(). */
        ql_head(extent_node_t)  node_cache;
        malloc_mutex_t    node_cache_mtx;

        /* User-configurable chunk hook functions. */
    // chunk_hooks_default
        chunk_hooks_t      chunk_hooks;

        /* bins is used to store trees of free regions. */
        arena_bin_t          bins[NBINS];
};
```

整体的结构图如下，忽略了很多细节：



run

`small classes` 从 `run` 中使用 `slab` 算法分配，每个 `run` 对应一块连续的内存，大小为 `page size` 倍数，划分为等大小的 `region`，分配时就从 `run` 中分配一个空闲 `region`，释放时就标记该 `region` 为空闲， 留待之后分配。

`arena_run_t` 记录了 `run` 的分配情况:

```
struct arena_run_s {
        /* Index of bin this run is associated with. */
        szind_t  binind;

        /* Number of free regions in run. */
        unsigned        nfree;

        /* Per region allocated/deallocated bitmap. */
        // 记录 run 中 region 的分配情况，每 bit 对应1个 region
        bitmap_t        bitmap[BITMAP_GROUPS_MAX];
};
```

现在看一下如何从 `run` 中分配:

1. 首先设置 `bitmap` 中第一个未设置的并返回，也就是要分配的 `region id`
2. 返回对应的 `region`，具体的地址计算后面再来看
3. `nfree--`

```
JEMALLOC_INLINE_C void *
arena_run_reg_alloc(arena_run_t *run, arena_bin_info_t *bin_info)
{
        void *ret;
        unsigned regind;
        arena_chunk_map_misc_t *miscelm;
        void *rpages;

        assert(run->nfree > 0);
        assert(!bitmap_full(run->bitmap, &bin_info->bitmap_info));

        // set first unset 并 返回
        regind = bitmap_sfu(run->bitmap, &bin_info->bitmap_info);
        miscelm = arena_run_to_miscelm(run);
        rpages = arena_miscelm_to_rpages(miscelm);
        // 获取 run 中对应的 region，返回
        ret = (void *)((uintptr_t)rpages + (uintptr_t)bin_info->reg0_offset +
            (uintptr_t)(bin_info->reg_interval * regind));
        run->nfree--;
        return (ret);
}
```

释放是相反的过程:

1. 首先获取该 `ptr` 的 `region id`
2. `unset` 对应的 `bitmap`
3. `nfree++`

```
JEMALLOC_INLINE_C void
arena_run_reg_dalloc(arena_run_t *run, void *ptr)
{
        arena_chunk_t *chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(run);
        size_t pageind = ((uintptr_t)ptr - (uintptr_t)chunk) >> LG_PAGE;
        size_t mapbits = arena_mapbits_get(chunk, pageind);
        szind_t binind = arena_ptr_small_binind_get(ptr, mapbits);
        arena_bin_info_t *bin_info = &arena_bin_info[binind];
        unsigned regind = arena_run_regind(run, bin_info, ptr);

        assert(run->nfree < bin_info->nregs);
        /* Freeing an interior pointer can cause assertion failure. */
        assert(((uintptr_t)ptr -
            ((uintptr_t)arena_miscelm_to_rpages(arena_run_to_miscelm(run)) +
            (uintptr_t)bin_info->reg0_offset)) %
            (uintptr_t)bin_info->reg_interval == 0);
        assert((uintptr_t)ptr >=
            (uintptr_t)arena_miscelm_to_rpages(arena_run_to_miscelm(run)) +
            (uintptr_t)bin_info->reg0_offset);
        /* Freeing an unallocated pointer can cause assertion failure. */
        assert(bitmap_get(run->bitmap, &bin_info->bitmap_info, regind));

        bitmap_unset(run->bitmap, &bin_info->bitmap_info, regind);
        run->nfree++;
}
```

bin

`jemalloc` 中 `small size classes` 都使用 `slab` 算法分配，所以会有多种不同的 `run`。`bin` 管理相同类型的 `run`，`bin_info` 记录了对应的 `run` 的内存格式。

`bin_info_init()` 根据 `size classes` 初始化 `small class bins` 的信息 `arena_bin_info[NBINS]`。数组中每个元素记录了 `bin` 对应的 `run` 的信息:

- `reg_size`：每个 `region` 的大小，对应着 `small size classes` 大小
- `run_size`：`bin` 对应的整个 `run` 的大小，`page_size` 的倍数，一般为 `reg_size` 和 `page_size` 的最小公倍数，但是不能超过 `arena_maxrun`。
- `nregs`：该 `run` 中 `region` 的个数
- `reg0_offset`：第一个 `region` 距离 `run` 起始地址的偏移
- 还有一些其他的信息，主要用于 `debug`

```
/*
 * Read-only information associated with each element of arena_t's bins array
 * is stored separately, partly to reduce memory usage (only one copy, rather
 * than one per arena), but mainly to avoid false cacheline sharing.
 *
 * Each run has the following layout:
 *
 *               /--------------------\
 *               | pad?               |
 *               |--------------------|
 *               | redzone            |
 *   reg0_offset | region 0           |
 *               | redzone            |
 *               |--------------------| \
 *               | redzone            | |
 *               | region 1           |  > reg_interval
 *               | redzone            | /
 *               |--------------------|
 *               | ...                |
 *               | ...                |
 *               | ...                |
 *               |--------------------|
 *               | redzone            |
 *               | region nregs-1     |
 *               | redzone            |
 *               |--------------------|
 *               | alignment pad?     |
 *               \--------------------/
 *
 * reg_interval has at least the same minimum alignment as reg_size; this
 * preserves the alignment constraint that sa2u() depends on.  Alignment pad is
 * either 0 or redzone_size; it is present only if needed to align reg0_offset.
 */
```

`bin` 的结构如下:

- `runcur`：指向有空闲 `region` 且地址最低的 `run`
- `runs`：红黑树，管理有空闲 `region` 的 `run`，按照 `run` 的地址排序

```
struct arena_bin_s {
        /*
         * All operations on runcur, runs, and stats require that lock be
         * locked.  Run allocation/deallocation are protected by the arena lock,
         * which may be acquired while holding one or more bin locks, but not
         * vise versa.
         */
        malloc_mutex_t  lock;

        /*
         * Current run being used to service allocations of this bin's size
         * class.
         */
        arena_run_t     *runcur;

        /*
         * Tree of non-full runs.  This tree is used when looking for an
         * existing run when runcur is no longer usable.  We choose the
         * non-full run that is lowest in memory; this policy tends to keep
         * objects packed well, and it can also help reduce the number of
         * almost-empty chunks.
         */
        // 红黑树 non-full runs, 按照地址排序
        arena_run_tree_t runs;

        /* Bin statistics. */
        malloc_bin_stats_t stats;
};
```

来看下如何从 `bin` 中分配 `run`：

1. 若 `runcur != NULL`，则从该 `run` 分配
2. 从 `runs` 中查找地址最低的 `run`，分配

当从 `run` 中释放 `region` 时，根据 `run` 的状态会有不同的操作:

1. 若该 `run` 原先已满，则会调用 `arena_bin_lower_run()` 设置为 `runcur` 或者插入到 `runs` 中
2. 若该 `run` 之前有空闲空间，说明是 `runcur` 或已经在 `runs` 中，此时无特殊处理
3. 若该 `run` 释放 `region` 后已空，则会将该 `run` 与 `bin` 解除关系，返回到 `arena` 中，后面再来看这种情况

`bin->runcur` 指向的永远是地址最低的 `run`，目的是减少 `active pages`。

chunk

`chunk` 是 `jemalloc` 中申请内存的基本单位。 `arena` 中有如下元素管理 `chunk`：

- `spare`：缓存最近空闲的 `chunk`，为了避免频繁的 `chunk` 分配和释放
- `chunks_szad_cached` / `chunks_ad_cached`：`extent_node_t` 的红黑树，缓存之前分配的、空闲的 `chunk`，数据一样，只是顺序不同:
  - `szad`：按照 `size`、`address` 排序
  - `ad`：按照 `address` 排序
- `chunks_szad_retained` / `chunks_ad_retained`：`extent_node_t` 的红黑树，缓存已经被释放的、空闲的 `chunk`，在后面 `purge` 阶段会看到

现在来看一下 `chunk` 的申请过程:

1. 若 `spare != NULL`，则返回 `spare`
2. 从 `cached` 中查找
3. 从 `retained` 中查找
4. 调用 `chunk_alloc_mmap()` 新分配一个 `chunk`

第2、3步会调用 `chunk_recycle()` 实施伙伴算法的分裂过程: 从对应的树中进行分配指定大小的 `chunk`， `chunk` 起始地址会按 `chunk_size` 对齐。因为需要对齐且大小不一定相等，所以前后需要进行裁剪， `leadsize` 和 `trailsize` 也会重新插入树中，留待之后的分配使用。

相对应的， `chunk` 释放过程如下:

1. 若 `spare == NULL`，则设置为 `spare`
2. 将原先的 `spare` 插入到 `cached` 中，设置为 `spare`

第2步会调用 `chunk_record()` 实施伙伴算法的合并过程: 会查找连续地址空间的前后的 `chunk` 在不在树中，如果在的话会进行合并，然后再插入到树中。

arena_chunk_t

`run` 从 `chunk` 中分配，同样采用伙伴算法。一整个 `chunk` 的内存分为4个部分:

1. `extent_node_t`: 记录 `chunk` 的状态，用于之后管理 `chunk`
2. `arena_chunk_map_bits_t`: 一一对应 `chunk` 内每个 `page`，记录从 `chunk` 分配出去的 `run` 的大小和信息、记录 `page` 的分配状态。
3. `arena_chunk_map_misc_t`: 一一对应 `chunk` 内每个 `page` 对应的 `run`
4. `page`: 大小为 `4096B`

这些记录 `chunk` 信息的 `header` 存放在每个 `chunk` 起始地址处，所以会占用掉部分内存。这些 `header` 和 `chunk` 中的 `page` 个数有关，而 `chunk` 中减去 `header` 的内存又和 `page` 的个数有关， 所以 `arena_boot()` 中使用循环计算 `header` 占用的 `page` 个数(`map_bias`):

```
/*
* Compute the header size such that it is large enough to contain the
* page map.  The page map is biased to omit entries for the header
* itself, so some iteration is necessary to compute the map bias.
*
* 1) Compute safe header_size and map_bias values that include enough
*    space for an unbiased page map.
* 2) Refine map_bias based on (1) to omit the header pages in the page
*    map.  The resulting map_bias may be one too small.
* 3) Refine map_bias based on (2).  The result will be >= the result
*    from (2), and will always be correct.
*/
map_bias = 0;
for (i = 0; i < 3; i++) {
    size_t header_size = offsetof(arena_chunk_t, map_bits) +
        ((sizeof(arena_chunk_map_bits_t) +
        sizeof(arena_chunk_map_misc_t)) * (chunk_npages-map_bias));
    map_bias = (header_size + PAGE_MASK) >> LG_PAGE;
}
```

`header` 使用连续内存存放而不是每个 `page` 头部存放有如下好处:

1. 提高 `header` 的缓存局部性
2. 提高 `page` 中分配出去的缓存局部性
3. 可以减少 `rss` 占用，因为操作系统按照 `page` 管理虚拟地址，若每个空闲 `page` 都有些 `header` 占用，会使一整个 `page` 驻留在内存中

`arena_chunk_map_bits_t` 在64位系统上，共有 `64bits`，记录了 `chunk` 内每个 `page` 的分配情况，这些信息用于快速的查找 `metadata`。对于不同状态的 `page` 有不同的格式:

1. 未分配 `page`：连续、未分配的 `page` 会作为一个整体，由起始 `page` 对应的 `run` 进行管理。首尾 `page` 对应的 `arena_chunk_map_bits_t` 中会设置连续的空闲 `page` 的数量， 中间的 `page` 不设置。同时，管理这些空闲 `page` 的 `run` 会插入到 `runs_avail` 中，该 `run` 的大小就是整个空闲 `page` 的大小(从 `arena_chunk_map_bits_t` 中获取)
2. 已分配的 `run` 对应的 `page`：每个 `page` 会设置该 `page` 是 `run` 中第几个 `page(run page offset)`，并且设置 `run` 对应的 `bin id`

```
/*
* Run address (or size) and various flags are stored together.  The bit
* layout looks like (assuming 32-bit system):
*
*   ???????? ???????? ???nnnnn nnndumla
*
* ? : Unallocated: Run address for first/last pages, unset for internal
*                  pages.
*     Small: Run page offset.
*     Large: Run page count for first page, unset for trailing pages.
* n : binind for small size class, BININD_INVALID for large size class.
* d : dirty?
* u : unzeroed?
* m : decommitted?
* l : large?
* a : allocated?
*
* Following are example bit patterns for the three types of runs.
*
* p : run page offset (这个page是run中第几个page (offset))
* s : run size (连续的空闲 page 个数)
* n : binind for size class; large objects set these to BININD_INVALID(该 page 对应的 bin Id)
* x : don't care
* - : 0
* + : 1
* [DUMLA] : bit set
* [dumla] : bit unset
*
*   Unallocated (clean):
*     ssssssss ssssssss sss+++++ +++dum-a
*     xxxxxxxx xxxxxxxx xxxxxxxx xxx-Uxxx
*     ssssssss ssssssss sss+++++ +++dUm-a
*
*   Unallocated (dirty):
*     ssssssss ssssssss sss+++++ +++D-m-a
*     xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*     ssssssss ssssssss sss+++++ +++D-m-a
*
*   Small:
*     pppppppp pppppppp pppnnnnn nnnd---A
*     pppppppp pppppppp pppnnnnn nnn----A
*     pppppppp pppppppp pppnnnnn nnnd---A
*
*   Large:
*     ssssssss ssssssss sss+++++ +++D--LA
*     xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*     -------- -------- ---+++++ +++D--LA
*
*   Large (sampled, size <= LARGE_MINCLASS):
*     ssssssss ssssssss sssnnnnn nnnD--LA
*     xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*     -------- -------- ---+++++ +++D--LA
*
*   Large (not sampled, size == LARGE_MINCLASS):
*     ssssssss ssssssss sss+++++ +++D--LA
```

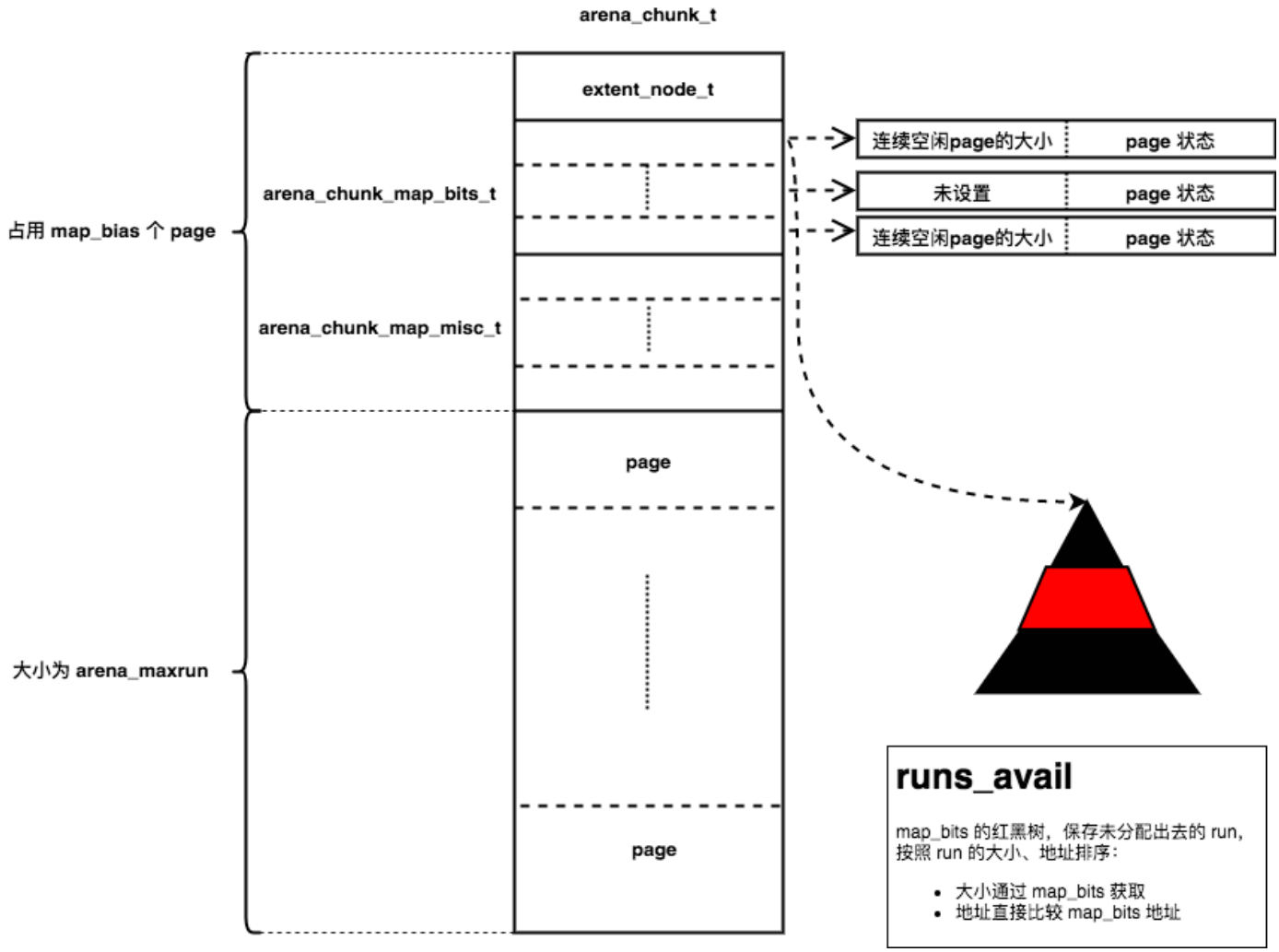
```
 *     xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
 *     -------- -------- ---+++++ +++D--LA
 */
```

`arena_chunk_map_misc_t` 顾名思义，有很多用途，主要用于记录 `run` 的 `metadata`。`run` 大小是 `page size` 倍数，每个 `run` 会由起始 `page` 对应的 `arena_chunk_map_misc_t` 中的 `run` 管理。

```
struct arena_chunk_map_misc_s {
        /*
         * Linkage for run trees.  There are two disjoint uses:
         *
         * 1) arena_t's runs_avail tree.
         * 2) arena_run_t conceptually uses this linkage for in-use non-full
         *    runs, rather than directly embedding linkage.
         */
        rb_node(arena_chunk_map_misc_t)  rb_link;

        union {
                /* Linkage for list of dirty runs. */
                arena_runs_dirty_link_t  rd;

                /* Profile counters, used for large object runs. */
                union {
                        void                    *prof_tctx_pun;
                        prof_tctx_t              *prof_tctx;
                };

                /* Small region run metadata. */
                arena_run_t              run;
        };
};
```

接下来看一下如何从 `chunk` 中分配 `run`。

第一个 `chunk` 是调用 `chunk_alloc_mmap()` 分配的，然后调用 `arena_mapbits_unallocated_set()` 设置首尾 `page` 对应的 `arena_chunk_map_bits_t`，然后将整个空闲 `chunk` 作为大小为 `arena_maxrun` 的空闲 `run` 插入到 `runs_avail` 中:

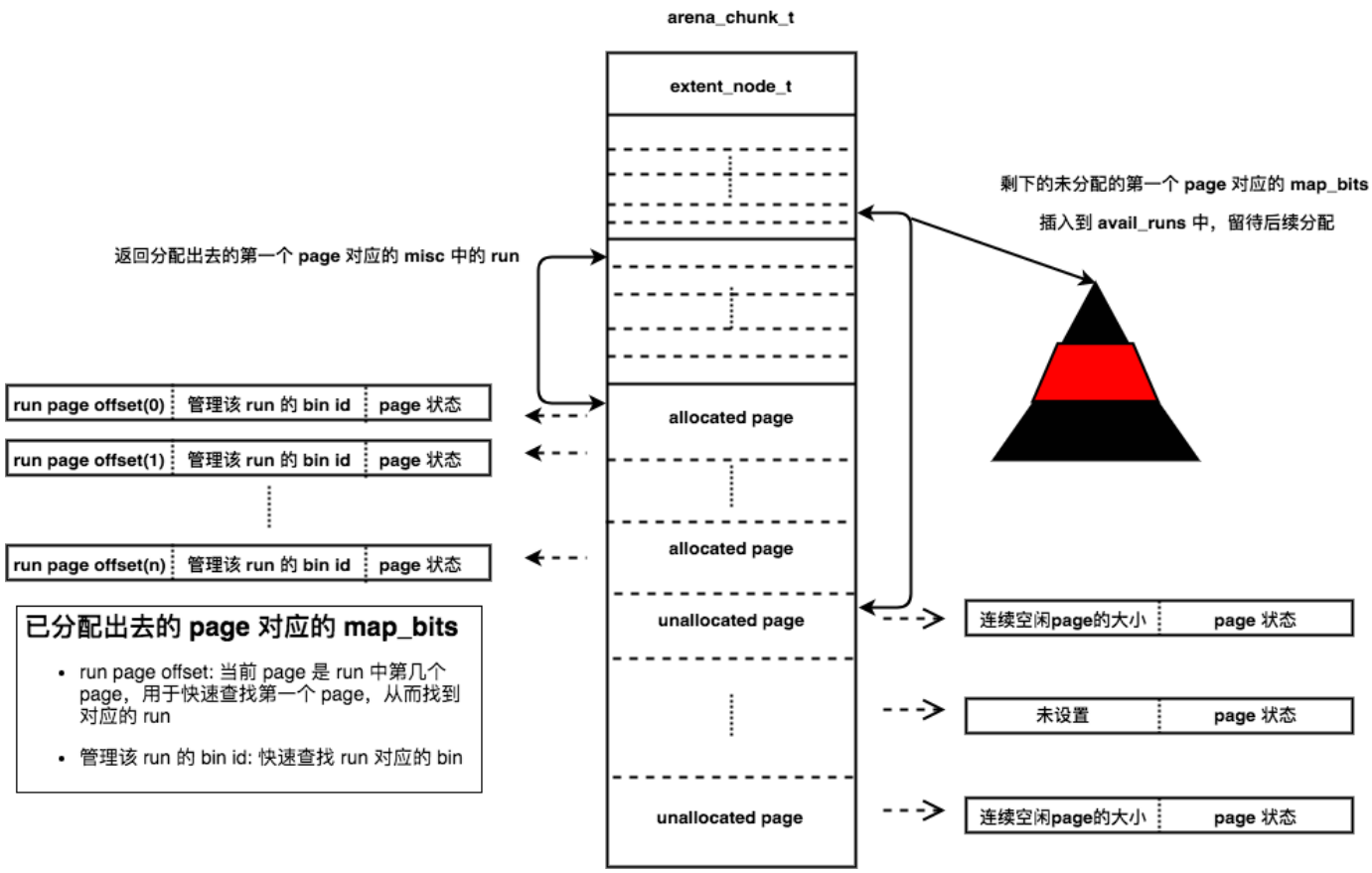


然后调用 `arena_run_split_small()` 将该 `run` 分解为对应的 `bin` 管理的 `run`：

1. 从 `run` 中分配出需要的 `page`，多余的 `page` 会设置首尾 `page` 对应的 `map_bits`，再次插入到 `avail_runs` 中留待后续分配
2. 设置分配出去的 `run` 对应的 `map_bits`，返回分配出去的第一个 `page` 对应的 `misc` 中的 `run`
3. 之后 `run` 就会有对应的 `bin` 进行管理

```
static bool
arena_run_split_small(arena_t *arena, arena_run_t *run, size_t size,
    szind_t binind)
{
        arena_chunk_t *chunk;
        arena_chunk_map_misc_t *miscelm;
        size_t flag_dirty, flag_decommitted, run_ind, need_pages, i;

        assert(binind != BININD_INVALID);

        chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(run);
        miscelm = arena_run_to_miscelm(run);
        run_ind = arena_miscelm_to_pageind(miscelm);
        flag_dirty = arena_mapbits_dirty_get(chunk, run_ind);
        flag_decommitted = arena_mapbits_decommitted_get(chunk, run_ind);
        // size 是 bin 对应的 run_size
        need_pages = (size >> LG_PAGE);
        assert(need_pages > 0);

        if (flag_decommitted != 0 && arena->chunk_hooks.commit(chunk, chunksize,
            run_ind << LG_PAGE, size, arena->ind))
                return (true);

        arena_run_split_remove(arena, chunk, run_ind, flag_dirty,
            flag_decommitted, need_pages);

        // 设置分配出去的 page 对应的 map_bits
        for (i = 0; i < need_pages; i++) {
                size_t flag_unzeroed = arena_mapbits_unzeroed_get(chunk,
                    run_ind+i);
                arena_mapbits_small_set(chunk, run_ind+i, i, binind,
                    flag_unzeroed);
                if (config_debug && flag_dirty == 0 && flag_unzeroed == 0)
                        arena_run_page_validate_zeroed(chunk, run_ind+i);
        }
        JEMALLOC_VALGRIND_MAKE_MEM_UNDEFINED((void *)((uintptr_t)chunk +
            (run_ind << LG_PAGE)), (need_pages << LG_PAGE));
        return (false);
}

// 从 run_ind 对应 run 中分配出 need_pages，剩余的再次插入到 avail_runs 中
static void
arena_run_split_remove(arena_t *arena, arena_chunk_t *chunk, size_t run_ind,
    size_t flag_dirty, size_t flag_decommitted, size_t need_pages)
{
        size_t total_pages, rem_pages;

        assert(flag_dirty == 0 || flag_decommitted == 0);

        total_pages = arena_mapbits_unallocated_size_get(chunk, run_ind) >>
            LG_PAGE;
        assert(arena_mapbits_dirty_get(chunk, run_ind+total_pages-1) ==
            flag_dirty);
        assert(need_pages <= total_pages);
        rem_pages = total_pages - need_pages;

        arena_avail_remove(arena, chunk, run_ind, total_pages);
```

```
        if (flag_dirty != 0)
                arena_run_dirty_remove(arena, chunk, run_ind, total_pages);
        arena_cactive_update(arena, need_pages, 0);
        arena->nactive += need_pages;

        /* Keep track of trailing unused pages for later use. */
        if (rem_pages > 0) {
                size_t flags = flag_dirty | flag_decommitted;
                size_t flag_unzeroed_mask = (flags == 0) ?  CHUNK_MAP_UNZEROED :
                    0;

            // 设置 run 对应的 page 的信息，设置开头和结尾的 page 对应的 map_bits 的
            // 未分配内存大小
                arena_mapbits_unallocated_set(chunk, run_ind+need_pages,
                    (rem_pages << LG_PAGE), flags |
                    (arena_mapbits_unzeroed_get(chunk, run_ind+need_pages) &
                    flag_unzeroed_mask));
                arena_mapbits_unallocated_set(chunk, run_ind+total_pages-1,
                    (rem_pages << LG_PAGE), flags |
                    (arena_mapbits_unzeroed_get(chunk, run_ind+total_pages-1) &
                    flag_unzeroed_mask));
                if (flag_dirty != 0) {
                        arena_run_dirty_insert(arena, chunk, run_ind+need_pages,
                            rem_pages);
                }
                arena_avail_insert(arena, chunk, run_ind+need_pages, rem_pages);
        }
}
```



当 `bin` 中有完全空闲的 `run` 时，会返回给 `arena` 管理:

1. 调用 `arena_dissociate_bin_run()` 解除该 `run` 和 `bin` 的关系:
    - 若该 `run` 为 `bin->runcur`，设置 `bin->runcur = NULL`
    - 从 `bin->runs` 中移除
2. 调用 `arena_run_coalesce()` 尝试合并相邻的空闲 `run`
3. 将 `run` 插入到 `avail_runs` 中
4. 若该 `run` 大小已经达到 `arena_maxrun`，表明整个 `chunk` 都是空闲的，调用 `arena_chunk_dalloc()` 释放 `run`

常量时间获取 metadata

有了上面的铺垫，现在可以来看一下 `jemalloc` 中的地址运算操作及如何在常量时间获取 `metadata`。

从 run 到 region

`arena_run_t` 中只记录了 `run` 的分配情况，并没有地址，需要快速的获取到需要分配的 `region` 的地址:

1. 先获取到 `misc` 的地址:

```
arena_chunk_map_misc_t *miscelm = (arena_chunk_map_misc_t
 *)((uintptr_t)run - offsetof(arena_chunk_map_misc_t, run));
```

2. 获取包含该 `misc` 的 `chunk` 起始地址:

```
/* 因为内存申请以 `chunk` 为单位，且按照 `chunk size` 对齐，所以只要将低位置零即可获取 `chunk` 起始地址 */
#define CHUNK_ADDR2BASE(a)                                     \
 ((void *)((uintptr_t)(a) & ~chunksize_mask))
arena_chunk_t *chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(miscelm);
```

3. 获取该 `misc` 的 `page id`:

```
arena_chunk_t *chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(miscelm);
/* (该 misc 在数组中的地址偏移 / misc 大小) 即可获取是数组中第几个元素 */
size_t pageind = ((uintptr_t)miscelm - ((uintptr_t)chunk +
 map_misc_offset)) / sizeof(arena_chunk_map_misc_t) + map_bias;
```

4. 获取 `misc` 对应的 `page` 地址:

```
return ((void *)((uintptr_t)chunk + (pageind << LG_PAGE)));
```

5. 获取对应的 `region`:

```
/* page 起始地址 + region0 的偏移 + (region id * region size) */
ret = (void *)((uintptr_t)rpages + (uintptr_t)bin_info->reg0_offset +
 (uintptr_t)(bin_info->reg_interval * regind));
```

从 region 到 run

当释放 `region` 时，需要快速查找 `region` 对应的 `run` 及 `region id`:

1. 先获取到 `chunk` 起始地址:

```
chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(ptr);
```

2. 获取 `region` 的 `page id`:

```
pageind = ((uintptr_t)ptr - (uintptr_t)chunk) >> LG_PAGE;
```

3. 获取 `page` 对应的 `map_bits`:

```
return (&chunk->map_bits[pageind-map_bias]);
```

4. 根据 `map_bits` 中设置的 `run page offset` 获取 `run` 的起始 `page id`:

```
rpages_ind = pageind - arena_mapbits_small_runind_get(chunk, pageind);

JEMALLOC_ALWAYS_INLINE size_t
arena_mapbits_small_runind_get(arena_chunk_t *chunk, size_t pageind)
{
    size_t mapbits;
    mapbits = arena_mapbits_get(chunk, pageind);
    assert((mapbits & (CHUNK_MAP_LARGE|CHUNK_MAP_ALLOCATED)) ==
        CHUNK_MAP_ALLOCATED);
    return (mapbits >> CHUNK_MAP_RUNIND_SHIFT);
}
```

5. 获取管理该 `region` 的 `run`:

```
run = &arena_miscelm_get(chunk, rpages_ind)->run;
```

6. 从 `run` 中得到 `bin_info`，再根据 `bin_info` 获取 `region id`:

```
/* diff 为 region 在 run 中的偏移 */
diff = (unsigned)((uintptr_t)ptr - (uintptr_t)rpages -
    bin_info->reg0_offset);
/* region id 可以通过 diff / bin_info->reg_interval 得到，但是 jemalloc 使用了复杂的运算为了提高性能，下面是它的注释 */

/*
 * Avoid doing division with a variable divisor if possible.  Using
 * actual division here can reduce allocator throughput by over 20%!
 */
```

7. 之后就可以设置 `region` 对应的 `bitmap` 进行释放了

run 的合并

前面看到 `run` 释放时会前后进行合并:

1. 查看 `run` 相邻的后面的 `page` 是不是空闲的:

```
/* 根据后面 page 的 map_bits 获取分配状态
arena_mapbits_allocated_get(chunk, run_ind+run_pages) == 0
```

2. 根据 `map_bits` 获取空闲 `page` 的大小:

```
size_t nrun_size = arena_mapbits_unallocated_size_get(chunk,
 run_ind+run_pages);
size_t nrun_pages = nrun_size >> LG_PAGE;
```

3. 然后将大小合并，在设置首尾 `page` 的 `map_bits`:

```
 size += nrun_size;
 run_pages += nrun_pages;

 arena_mapbits_unallocated_size_set(chunk, run_ind, size);
 arena_mapbits_unallocated_size_set(chunk, run_ind+run_pages-1,
     size);
```

4. 查看 `run` 相邻的前面的 `page` 是不是空闲的:

```
arena_mapbits_allocated_get(chunk, run_ind-1) == 0
```

5. 根据 `map_bits` 获取空闲 `page` 的大小:

```
size_t prun_size = arena_mapbits_unallocated_size_get(chunk,
 run_ind-1);
size_t prun_pages = prun_size >> LG_PAGE;
```

6. 然后将大小合并，再设置首尾 `page` 的 `map_bits`:

```
 size += prun_size;
 run_pages += prun_pages;

 arena_mapbits_unallocated_size_set(chunk, run_ind, size);
 arena_mapbits_unallocated_size_set(chunk, run_ind+run_pages-1,
     size);
```

由此得知，因为前后都需要进行合并，所以首尾 `page` 对应的 `map_bits` 都会设置大小。

size classes

`jemalloc` 将对象按大小分为3类，不同大小类别的分配算法不同:

- `small`: 从对应 `bin` 管理的 `run` 中返回一个 `region`
- `large`: 大小比 `chunk` 小，比 `page` 大，会单独返回一个 `run`
- `huge`: 大小为 `chunk` 倍数，会分配 `chunk`

在 `2MiB chunk`，`4KiB page` 的64位系统上，`size classes` 如下:

```
+---------+---------+--------------------------------------+
|Category | Spacing | Size                                 |
+---------+---------+--------------------------------------+
|         |      lg | [8]                                  |
|         +---------+--------------------------------------+
|         |      16 | [16, 32, 48, 64, 80, 96, 112, 128]   |
|         +---------+--------------------------------------+
|         |      32 | [160, 192, 224, 256]                 |
|         +---------+--------------------------------------+
|         |      64 | [320, 384, 448, 512]                 |
|         +---------+--------------------------------------+
|Small    |     128 | [640, 768, 896, 1024]                |
|         +---------+--------------------------------------+
|         |     256 | [1280, 1536, 1792, 2048]             |
|         +---------+--------------------------------------+
|         |     512 | [2560, 3072, 3584, 4096]             |
|         +---------+--------------------------------------+
|         |   1 KiB | [5 KiB, 6 KiB, 7 KiB, 8 KiB]         |
|         +---------+--------------------------------------+
|         |   2 KiB | [10 KiB, 12 KiB, 14 KiB]             |
+---------+---------+--------------------------------------+
|         |   2 KiB | [16 KiB]                             |
|         +---------+--------------------------------------+
|         |   4 KiB | [20 KiB, 24 KiB, 28 KiB, 32 KiB]     |
|         +---------+--------------------------------------+
|         |   8 KiB | [40 KiB, 48 KiB, 54 KiB, 64 KiB]     |
|         +---------+--------------------------------------+
|         |  16 KiB | [80 KiB, 96 KiB, 112 KiB, 128 KiB]   |
|Large    +---------+--------------------------------------+
|         |  32 KiB | [160 KiB, 192 KiB, 224 KiB, 256 KiB] |
|         +---------+--------------------------------------+
|         |  64 KiB | [320 KiB, 384 KiB, 448 KiB, 512 KiB] |
|         +---------+--------------------------------------+
|         | 128 KiB | [640 KiB, 768 KiB, 896 KiB, 1 MiB]   |
|         +---------+--------------------------------------+
|         | 256 KiB | [1280 KiB, 1536 KiB, 1792 KiB]       |
+---------+---------+--------------------------------------+
|         | 256 KiB | [2 MiB]                              |
|         +---------+--------------------------------------+
|         | 512 KiB | [2560 KiB, 3 MiB, 3584 KiB, 4 MiB]   |
|         +---------+--------------------------------------+
|         |   1 MiB | [5 MiB, 6 MiB, 7 MiB, 8 MiB]         |
|         +---------+--------------------------------------+
|Huge     |   2 MiB | [10 MiB, 12 MiB, 14 MiB, 16 MiB]     |
|         +---------+--------------------------------------+
|         |   4 MiB | [20 MiB, 24 MiB, 28 MiB, 32 MiB]     |
|         +---------+--------------------------------------+
|         |   8 MiB | [40 MiB, 48 MiB, 56 MiB, 64 MiB]     |
|         +---------+--------------------------------------+
|         |     ... | ...                                  |
+---------+---------+--------------------------------------+
```

small

`small` 的分配流程如下：

1. 查找对应 `size classes` 的 `bin`
2. 从 `bin` 中获取 `run`：
    1. `bin->runcur`
    2. 从 `bin->runs` 查找未满的 `run`
3. 从 `arena` 中获取 `run`：
    1. 从 `arena->avail_runs` 中查找空闲 `run`
    2. 当没有合适 `run` 时，从 `chunk` 中分配 `run`：
        1. `arena->spare`
        2. `arena->cached_tree`
        3. `arena->retained_tree`
        4. 调用 `mmap()` 新分配一块 `chunk`
4. 从 `run` 中返回一个空闲 `region`

`small` 的释放流程如下：

1. 将该 `region` 返回给对应的 `run`，即设置 `bitmap` 为空闲，增加 `nfree`
2. 将 `run` 还给 `bin`：
    1. 如果 `run->nfree == 1`，则设置为 `bin->runcur` 或者插入到 `bin->runs` 中
3. 如果 `run->nfree == bin_info->nregs`，则将该 `run` 与 `bin` 分离，再将 `run` 还给 `arena`：
    1. 尝试与相同 `chunk` 中前后相邻的空闲 `run` 进行合并，然后插入到 `arena->avail_runs` 中
    2. 若合并完后，整个 `chunk` 为空，则尝试与连续地址空间的空闲 `chunk` 进行合并，然后插入到 `arena->cached_tree` 中

large

分配 `large` 和分配 `small` 类似：

1. 先从 `arena->avail_runs` 中查找，因为 `large object` 不由 `bin` 管理，所以与 `small object` 相比，少了从 `bin->runs` 中查找的一步
2. 分配 `chunk`，步骤和 `small` 一样，然后从 `chunk` 中分配需要的 `run` 大小，此时 `run` 的大小为单个 `object` 的大小，而 `small run` 的大小会从 `bin_info[]` 中获取

因为 `large` 大小是 `page` 的倍数，且会按照 `page size` 地址对齐，有可能造成 `cache line` 颠簸， 所以会根据配置多分配一个 `page`，用于内存着色，防止 `cache line` 的颠簸

```
if (config_cache_oblivious) {
    uint64_t r;

    /*
     * Compute a uniformly distributed offset within the first page
     * that is a multiple of the cacheline size, e.g. [0 .. 63) * 64
     * for 4 KiB pages and 64-byte cachelines.
     */
    prng64(r, LG_PAGE - LG_CACHELINE, arena->offset_state,
        UINT64_C(6364136223846793009),
        UINT64_C(1442695040888963409));
    random_offset = ((uintptr_t)r) << LG_CACHELINE;
}
```

`large` 和 `small` 的 `arena_chunk_map_misc_t` 格式也不同，`large` 只在首个 `page` 设置 `run` 的大小。释放流程和 `small` 一样，只是缺少了 `run` 在 `bin` 中的处理，直接将 `run` 还给 `arena`。

huge

`huge object` 大小比 `chunk` 大。分配策略和上面分配 `chunk` 一样:

1. 从 `arena` 中分配 `extent_node_t`
2. 从 `arena` 中分配 `chunk`：
    1. 从 `arena->cached_tree` 中分配 `chunk`
    2. 从 `arena->retained_tree` 中分配
    3. 调用 `mmap()` 新分配一块 `chunk`
3. 将 `chunk` 和 `node` 插入到 `chunks_rtree` 中
4. 插入到 `arena->huge` 链表中

释放和分配过程相反:

1. 从 `chunks_rtree` 中获取 `chunk` 对应的 `node`，从而获取对应的 `arena`
2. 移出 `arena->huge`
3. 释放 `chunk`，插入到 `arena->cached_tree` 中
4. 释放 `node`

`huge` 使用了线程间共享的 `chunks_rtree` 来保存信息，这会导致锁的竞争，但是应用程序很少会分配如此大的内存，所以带来的影响很小。

purge

前面的释放只是将之前分配的缓存起来，备用，现在来看一下真正的释放操作。

`arena` 中会统计 `dirty` 和 `active` 的数目：

- `nactive`：已经分配出去的 `page` 数目
- `ndirty`：分配出去又被释放的 `page` 数目

`arena` 中会保存最多 `nactive >> lg_dirty_mult` 的 `dirty pages` 暂存使用，当超出时，就会释放掉多余的部分。

`purge` 按照 `page` 维度进行回收。`arena` 中 `runs_dirty` 和 `chunks_cache` 存放着 `dirty pages`，当 `run` 和 `chunk` 被释放时，会插入到这里(`chunk` 也会插入到 `runs_dirty` 中，同时也插入到 `chunks_cache`):

```
 *
 * Unused dirty memory this arena manages.  Dirty memory is conceptually
 * tracked as an arbitrarily interleaved LRU of dirty runs and cached
 * chunks, but the list linkage is actually semi-duplicated in order to
 * avoid extra arena_chunk_map_misc_t space overhead.
 *
 *   LRU-----------------------------------------------------------MRU
 *
 *        /-- arena ---\
 *        |            |
 *        |            |
 *        |------------|                             /- chunk -\
 *   ...->|chunks_cache|<--------------------------->|  /----\ |<--...
 *        |------------|                             |  |node| |
 *        |            |                             |  |    | |
 *        |            |    /- run -\    /- run -\   |  |    | |
 *        |            |    |       |    |       |   |  |    | |
 *        |            |    |       |    |       |   |  |    | |
 *        |------------|    |-------|    |-------|   |  |----| |
 *   ...->|runs_dirty  |<-->|rd     |<-->|rd     |<----->|rd  |<----...
 *        |------------|    |-------|    |-------|   |  |----| |
 *        |            |    |       |    |       |   |  |    | |
 *        |            |    |       |    |       |   |  \----/ |
 *        |            |    \-------/    \-------/   |         |
 *        |            |                             |         |
 *        |            |                             |         |
 *        \------------/                             \---------/
 *
```

在每次 `dalloc run/chunk` 时都会调用 `arena_maybe_purge()` 尝试 `purge` 。`arena` 根据 `lg_dirty_mult` 判断是否需要 `purge`，当 `(nactive >> lg_dirty_mult) <= ndirty` 时进行 `purge`，默认配置为 `8 : 1` 。

`purge` 分为4步:

1. `arena_compute_npurge()`: 返回需要 `purge` 的 `page` 数目，为超出 `nactive >> lg_dirty_mult` 的 `page` 数。
2. `arena_stash_dirty()`: 将需要 `purge` 的部分从 `arena->cached_tree` 或 `arena->avail_runs` 中移除，防止 `purge` 过程中被其他线程分配出去，并插入到需要 `purge` 的循环链表中。
3. `arena_purge_stashed()`: 将循环链表中的 `run` 进行 `purge`
4. `arena_unstash_purged()`: 将 `chunk` 进行 `purge` 。将 `purged` 插入到 `arena->cached_tree` 或 `arena->avail_runs`，留待后面分配。

对 `chunk` 和 `run` 采取不同的 `purge`:

- 对于 `run` 而言，并不是真正的释放，根据操作系统的不同，会使用不同的方式，在 `linux` 中会调用 `madvise(addr, size, MADV_DONTNEED)` 。
- `jemalloc` 以 `chunk` 为单位向操作系统申请内存，在释放 `chunk` 时，会尽量调用 `munmap()` (因为根据操作系统和配置的不同，`chunk` 的来源也不同)，否则会类似 `run`，调用 `madvise()` 然后再插入到 `chunk_retained_tree` 中，留待后续分配。

`jemalloc` 中单线程的部分就到此结束了，下面开始看 `jemalloc` 是如何提升多线程性能的。

多线程

`jemalloc` 的一个目标就是提高多线程的性能，多线程的分配思路和单线程是一样的，每个线程还是从 `arena` 中分配内存，不过会多了线程间的同步和竞争。想要提高多线程性能，主要通过下面 2 个方式:

- 减少锁的竞争：缩小临界区，更细粒度的锁
- 避免锁的竞争：线程间不共享数据，使用局部变量、线程特有数据(`tsd`)、线程局部存储(`tls`)等

arena

`jemalloc` 会创建多个 `arena`，每个线程由一个 `arena` 负责。在 `malloc_init_hard_finish()` 中会设置 `arena` 的相关配置， `narenas_auto` 和 `narenas_total` 都设置为 `cpu` 核数*4，默认最多创建那么多 `arena`。 `arena->nthreads` 记录负责的线程数量。

每个线程分配时会首先调用 `arena_choose()` 选择一个 `arena` 来负责该线程的分配。选择 `arena` 的逻辑如下:

1. 若有空闲的(`nthreads==0`)已创建的 `arena`，则选择该 `arena`
2. 若还有未创建的 `arena`，则选择新创建一个 `arena`
3. 选择负载最低的 `arena` (`nthreads` 最小)

锁

`mutex` 尽量使用 `spinlock`，减少线程间的上下文切换:

```
JEMALLOC_INLINE void
malloc_mutex_lock(malloc_mutex_t *mutex)
{

        if (isthreaded) {
#ifdef _WIN32
#  if _WIN32_WINNT >= 0x0600
                AcquireSRWLockExclusive(&mutex->lock);
#  else
                EnterCriticalSection(&mutex->lock);
#  endif
#elif (defined(JEMALLOC_OSSPIN))
                OSSpinLockLock(&mutex->lock);
#else
                pthread_mutex_lock(&mutex->lock);
#endif
        }
}
```

为了缩小临界区， `arena` 中有多个锁管理不同的部分:

- `arenas_lock` : `arena` 的初始化、分配等
- `arena->lock` : `run` 和 `chunk` 的管理
- `arena->huge_mtx` : `huge object` 的管理
- `bin->lock` : `bin` 中的操作

tsd

当选择完 `arena` 后，会将 `arena` 绑定到 `tsd` 中，之后会直接从 `tsd` 中获取 `arena`。

`tsd` 用于保存每个线程特有的数据，主要是 `arena` 和 `tcache`，避免锁的竞争。 `tsd_t` 中的数据会在第一次访问时延迟初始化(调用相应的 `get_hard()`)， `tsd` 中各元素使用宏生成对应的 `get/set` 函数来获取/设置， 在线程退出时，会调用相应的 `cleanup` 函数清理。下面只介绍 `linux` 平台中的实现。

在 `linux` 中会使用 `tls(__thread)` 和 `tsd(pthread_key_create(), pthread_setspecific())` 来实现:

```
#elif (defined(JEMALLOC_TLS))
#define malloc_tsd_data(a_attr, a_name, a_type, a_initializer)          \
a_attr __thread a_type JEMALLOC_TLS_MODEL                               \
    a_name##tsd_tls = a_initializer;                                    \
a_attr pthread_key_t    a_name##tsd_tsd;                                \
a_attr bool             a_name##tsd_booted = false;
```

- `__thread` 保存需要线程局部存储的数据 `tsd_t`
- `pthread_key_t` 将 `key` 与 `__thread` 联系起来， 用于注册 `destructor`， 在线程退出时清理 `tsd_t`
  其实可以只用 `pthread_key_t` 来实现，但使用 `__thread` 可以直接获取数据，不用再调用 `pthread_getspecific()`。

tcache

`tcache` 用于 `small` 和 `large` 的分配，避免多线程的同步。

```
"opt.tcache" (bool) r- [--enable-tcache]
        Thread-specific caching (tcache) enabled/disabled. When there are multiple threads, each thread
uses a tcache for objects up to a certain size. Thread-specific caching allows many allocations to be
satisfied without performing any
        thread synchronization, at the cost of increased memory use. See the "opt.lg_tcache_max" option for
related tuning information. This option is enabled by default unless running inside Valgrind[2], in which
case it is forcefully
        disabled.

"opt.lg_tcache_max" (size_t) r- [--enable-tcache]
    Maximum size class (log base 2) to cache in the thread-specific cache (tcache). At a minimum, all small
size classes are cached, and at a maximum all large size classes are cached. The default maximum is 32 KiB
(2^15).
```

```
struct tcache_s {
        ql_elm(tcache_t) link;    /* Used for aggregating stats. */
        uint64_t        prof_accumbytes;/* Cleared after arena_prof_accum(). */
        unsigned        ev_cnt;  /* Event count since incremental GC. */
        szind_t  next_gc_bin;   /* Next bin to GC. */
        tcache_bin_t    tbins[1];       /* Dynamically sized. */
        /*
         * The pointer stacks associated with tbins follow as a contiguous
         * array.  During tcache initialization, the avail pointer in each
         * element of tbins is initialized to point to the proper offset within
         * this array.
         */
};

struct tcache_bin_s {
        tcache_bin_stats_t tstats;
        int          low_water; /* Min # cached since last GC. */
        unsigned        lg_fill_div;    /* Fill (ncached_max >> lg_fill_div). */
        unsigned        ncached;        /* # of cached objects. */
        void        **avail;    /* Stack of available objects. */
};
```

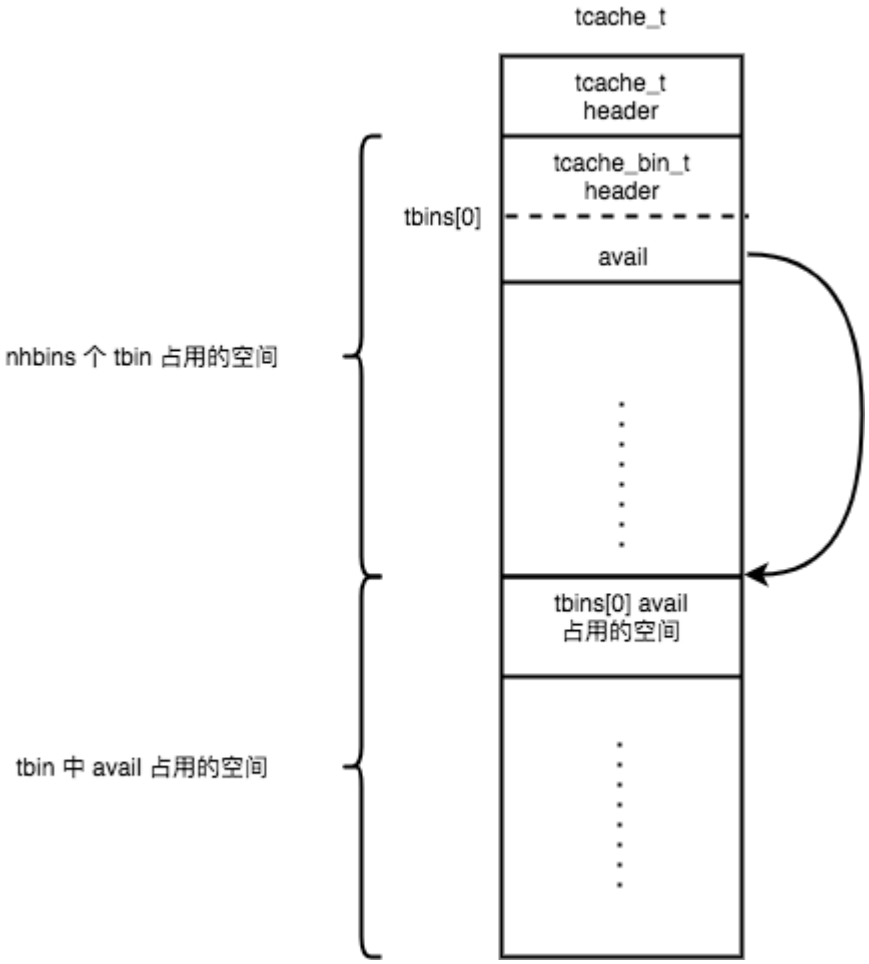
`tcache` 同样使用 `slab` 算法分配:

1. `tcache` 中有多种 `bin`，每个 `bin` 管理一个 `size class`
2. 当分配时，从对应 `bin` 中返回一个 `cache slot`
3. 当释放时，将 `cache slot` 返回给对应的 `bin`

`tcache` 中 `avail` 是指针数组，每个数组元素指向对应的 `cache slot`， `cache slot` 是从 `arena` 中分配的，缓存在 `tcache` 中。

`tcache_boot()`：根据配置 `opt.lg_tcache_max` 设置 `tcache` 中 `bin` 的范围(`nhbins`)。 设置 `tcache_bin_info`，保存每种 `bin` 的 `cache slots` 个数(类似 `arena_bin_info` 中 `nregs`)， `small` 在 `TCACHE_NSLOTS_SMALL_MIN` 到 `TCACHE_NSLOTS_SMALL_MAX` 间， `large` 固定为 `TCACHE_NSLOTS_LARGE` 。

`tcache_create()`： `tcache_t` 中保存着 `tbins[]` 信息， `tcache_bin_t` 中 `avail` 指向每一个 `cache slot` (类似 `arena->bin` 中 `region`)， `tcache_create()` 根据 `tcache_boot()` 设置的配置分配 `tcache_t` 和 `tcache_bin_t` 的内存， `tcache_t` 和 `tbins[]` 为连续内存， `tbins[]` 中 `avail` 使用后面连续空间的内存。



small

`small` 分配流程如下:

1. `tcache_alloc_small()`：先获取对应的 `tbin`，调用 `tcache_alloc_easy()`，若 `tbin` 中还有剩余的元素，返回 `tbin->avail[tbin->ncached]` (从后往前分配， `ncached` 既是剩余数量也是索引)， `tbin->low_water` 保存着 `tbin->ncached` 的最小值。
2. `tcache_alloc_small_hard()`： `tbin` 已空，先调用 `arena_tcache_fill_small()` 重新装载 `tbin`，再调用 `tcache_alloc_easy()` 分配。
3. `arena_tcache_fill_small()`：从 `arena` 中对应的 `bin` 分配 `region` 保存在 `tbin->avail` 中，只会填充 `ncached_max >> lg_fill_div` 个。

`small` 释放流程如下:

1. `tcache_dalloc_small()`：通过 `ptr` 对应的 `map_bits` 获取 `binind`，然后将 `ptr` 释放(保存在 `tbin->avail[tbin->ncached]`，同时 `tbin->ncached++`)。若该 `tbin` 已满(`tbin->ncached == tbin_info->ncached_max`)，会调用 `tcache_bin_flush_small()`，释放一半 `cache slots` 给 `arena`。
2. `tcache_bin_flush_small()`：会释放 `tbin` 中部分 `avail` 返回给 `arena` 中对应的 `bin`，这里为了减少锁的调用，会在一次加锁中，释放所有对应该锁(`bin`)的 `cache slot`。

large

分配和 `small` 类似，先调用 `tcache_alloc_easy()`，不过若 `tbin` 为空时，不会像 `small` 一样分配所有的 `avail`， 而是调用 `arena_malloc_large()` 从 `arena` 中分配一个 `run`。因为创建多个 `large object` 太过昂贵，并且有可能会用不到，浪费空间。

释放和 `small` 类似，先释放到 `tbin->avail[tbin->ncached]` 中，备用。若该 `tbin` 已满，调用 `arena_bin_flush_large()` 释放一半到 `arena` 中。

gc

前面注意到，当从 `arena` 分配 `small` 时， 会分配 `ncached_max >> lg_fill_div` 个，若每次均分配固定数目，有可能会造成内存浪费， `jemalloc` 对 `tcache` 中的 `bin` 采用渐进式 `GC`，动态的调整分配数目。有 `2` 个宏控制着 `GC` 的进行:

- `TCACHE_GC_SWEEP`：可以近似认为每发生该数量的分配或释放操作，所有的 `bin` 都被 `GC`
- `TCACHE_GC_INCR`：每发生该数量的分配或释放操作，单个 `bin` 进行一次 `GC`

`tcache` 中每个 `bin` 会有如下2个字段:

- `low_water`：保存着一次 `GC` 时间间隔内， `ncached` 的最小值，也就意味着在这之下的 `avail` 都没被分配
- `lg_fill_div`：用于控制每次分配的数量(`ncached_max >> lg_fill_div`)，初始为 `1`

在每次分配和释放时，都会调用 `tcache_event()`，增加 `tcache->ev_cnt`，若和 `TCACHE_GC_INCR` 相等，则调用 `tcache_event_hard()` 对单个 `bin` 进行 `GC`(只对 `small object` 有效)。 `tcache_event_hard()`：对单个 `bin(next_gc_bin)` 进行 `GC`：

1. 若 `tbin->low_water > 0`：说明 `tbin->avail` 中有些未被用到，可以尝试减少分配。对应的操作就是释放掉 `3/4 low_water`， `lg_fill_div++` (下次分配时会减少一半)
2. 若 `tbin->low_water < 0`：只有在该 `tbin->avail` 全部分配完才会置 `low_water = -1`，说明不够用，所以会 `lg_fill_div--` (下次分配时加倍)
   `tcache` 中的 `tbin` 分配数量就会一直动态调整。

线程退出

线程退出时，会调用 `tsd_cleanup()` 对 `tsd` 中数据进行清理:

- `arena`：降低 `arena` 负载(`arena->nthreads--`)
- `tcache`：调用 `tcache_bin_flush_small/large()` 释放 `tcache->tbins[]` 所有元素，释放 `tcache` 本身

当从一个线程分配的内存由另一个线程释放时，该内存还是由原先的 `arena` 来管理，通过 `chunk` 的 `extent_node_t` 来获取对应的 `arena`。

总结

`jemalloc` 中大量使用了宏生成代码，比较晦涩，不过其他部分还是比较清楚的，只要理解了它的思路就容易看懂，一层一层的。现在来总结一下 `jemalloc` 的思路：

- 通过避免 `false cache line sharing`，使用内存着色等，提高 `cache line` 效率
- 使用 `slab` 分配不同大小的对象，精心选择 `size classes`，减少内存碎片
- 使用多层缓存，内存的释放和分配会经历很多阶段，提升速度
- `metadata` 存放在连续内存，降低 `metadata` 的 `overhead`，同时能减少 `active pages`
- 地址对齐从而在常量时间内获取 `metadata`
- 首先复用低地址的内存，减少 `active pages`
- 使用多个 `arena` 管理、更细粒度的锁、 `tsd` 、`tcache` 等，最小化锁竞争

看懂 `jemalloc` 大概花了 2 周时间，但是写这篇博客也花了 2 周的时间，写的很难受，写博客真的不容易。

分类: Allocator (https://youjiali1995.github.io/categories/#allocator)

更新时间: February 28, 2018