内存分配奥义·jemalloc(二)

tinylab.org/memory-allocation-mystery-·-jemalloc-b

by Chen Jie of TinyLab.org 2014/12/11

Chen Jie 创作于 2014/12/30

打赏

1 前情回顾

在上篇 jemalloc 文中，我们看到了一个类似电商物流体系的内存分配体系，其中 **tcache** 是本线程内存仓库； **arena** 是几个线程共享的区域内存仓库；并且存在所有线程共用的内存仓库。

另一方面，jemalloc 按照内存分配请求的尺寸，分了 **small allocation** (例如 1 – 57344B)、 **large allocation** (例如 57345 – 4MB)、 **huge allocation** (例如 4MB以上)。

jemalloc 以 *chunk* （例如标准大小为 4MB）为单位批发内存，并再分成 *run(s)* 来满足实际的需求。对于 small allocation ，进一步将 run 分成 *region(s)* 。

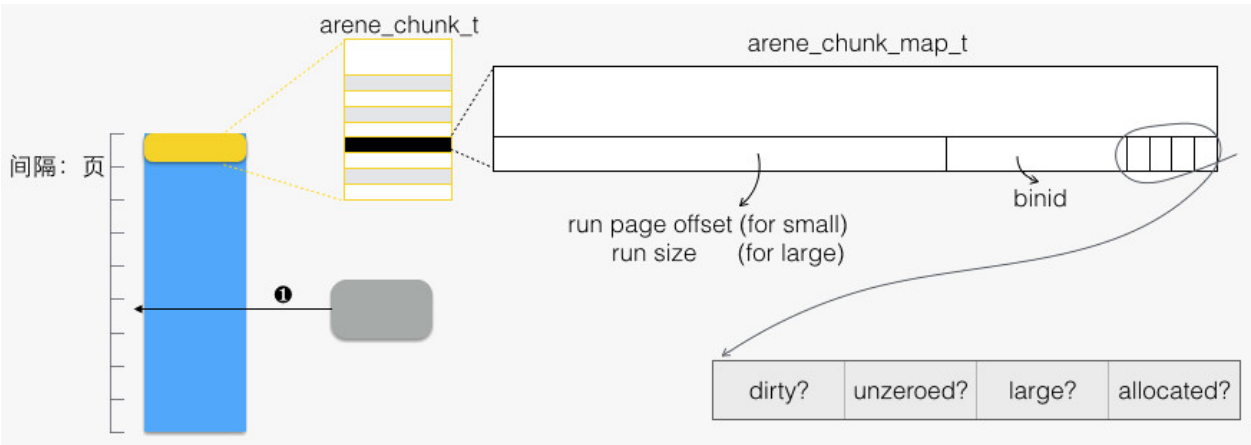
最后，疑惑 jemalloc 的层层缓冲，会造成过多的内存占用，这对实时性要求较高，内存较为紧张的移动设备影响较大。对此，jemalloc 如何应对呢？还有，是否存在系统内存紧张时，减少缓冲的联动机制呢?

2 How to free()?

在内存分配时，jemalloc 按照 small/large/huge allocation 来特殊处理。因此，释放时，需要由地址来判断为何种分配类型。

我们知道分配出去的空间，都属于某个 chunk，首先通过将地址对齐到 *标准 chunk 大小*，找到所属 chunk（还记得 chunk 是按照 标准 chunk 大小对齐的么）。函数路径: `je_free/ifree/iqalloc/iqalloct/idalloct` :

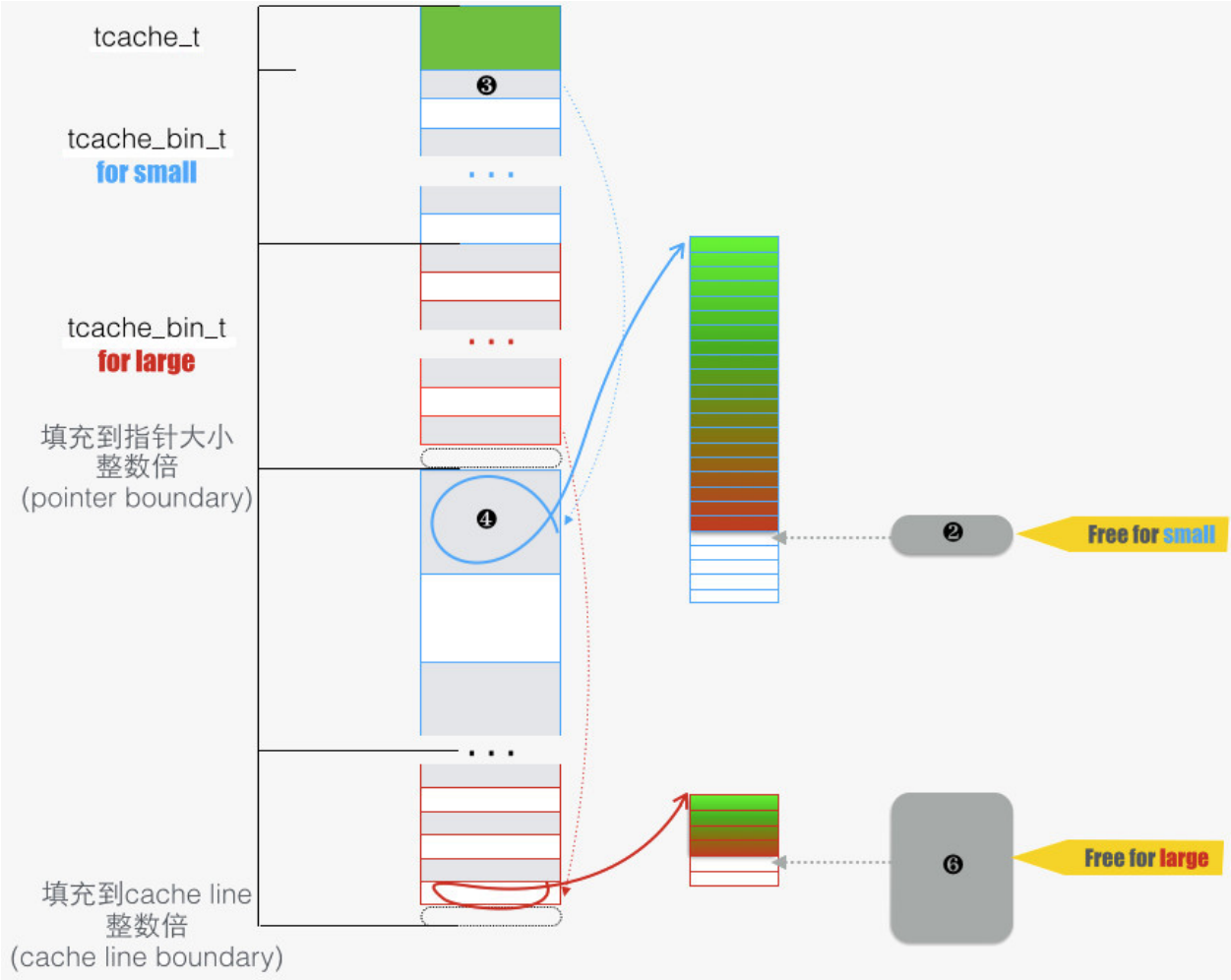
1. 对于 huge allocation，free 的地址本身在 chunk 边界上。搜索全局的 **huge** 树来获得本次分配的长度。函数路径： `idalloct/huge_dalloc` 。
2. 对于 small/large allocation，根据 free 的地址所在页在 chunk 内的相对页号，访问 chunk 头部的 arena_chunk_map_t 数组，获得进一步的信息，函数路径为 `idalloct/arena_dalloc` 。下图为其示意:



对于 small allocation，large 位为 0。下面以 small allocation 为主线，开展一场内存回归之旅。

2.1 free to tcache

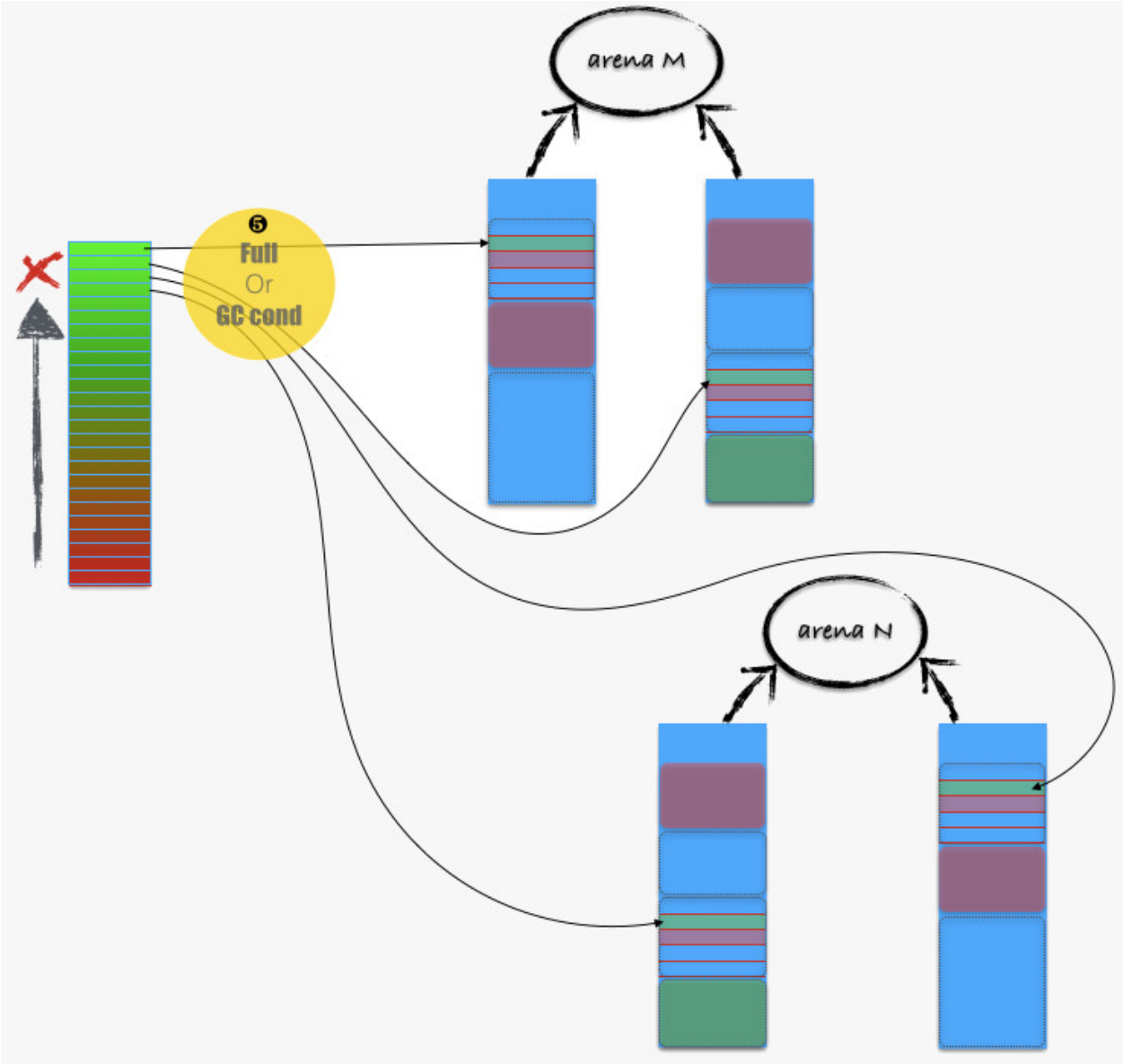
下图为 tcache 的示意，tcache 用于屏蔽常用的内存分配和释放（所有 small 和部分 large 类型的），免得下代码走的太远。函数路径： `arena_dalloc/tcache_dalloc_small`



- 2 → 3：释放一段 region，首先要知道所属 bin（tcache_bin_t）。Q：如何找到所属 bin？可以从对应 arena_chunk_map_t 成员中的 binid 域。
- 3 → 4：tcache_bin_t 用指针数组来收纳释放的内存，这是一个栈的结构。最近释放的内存放在栈顶，使得下次被先分配出去。如此能保持 cache 热度。

2.2 缩减 tcache

当 tcache_bin_t 满了以后，或者 GC 事件被触发，则降低 tcache_bin_t 中缓冲内存的数量，如下图：



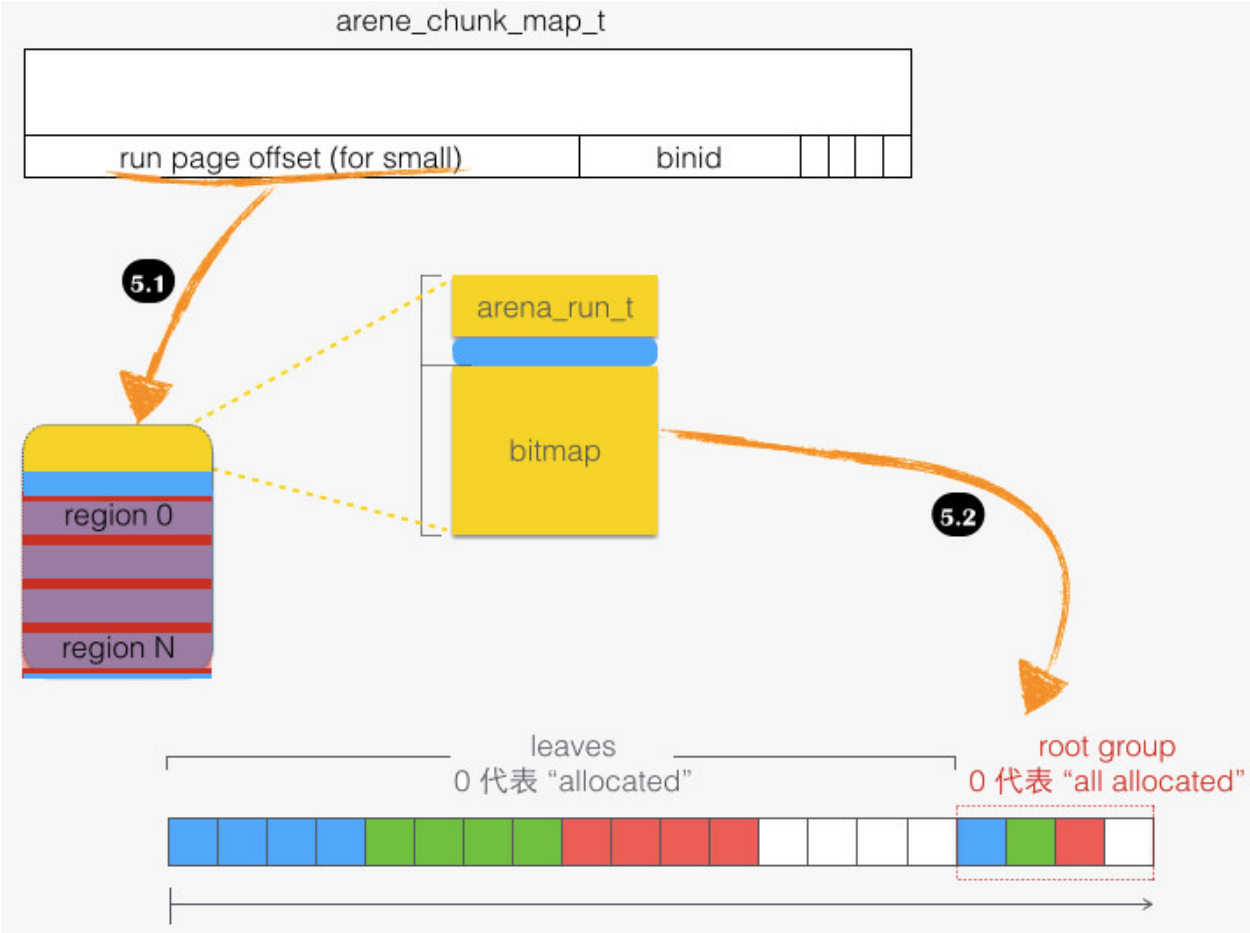
基本上的过程就是让 tcache_bin_t 中部分缓冲内存“各回各家” —— 回到所在 arena 中的 chunk 中的 run 内。

注意，刷的过程是将 栈底 N 个内存刷回，然后将其它部分整体移到栈底，这么做同样是保持 cache 热度。

啰嗦下 flush 触发的两个条件：

- tcache_bin_t 满了。函数路径为 `tcache_dalloc_small/tcache_bin_flush_small` 。
- GC 条件满足。函数路径为 `tcache_dalloc_small/tcache_event/tcache_event_hard/tcache_bin_flush_small` 。注意：此时 flush 的 bin，由 tcache->next_gc_bin 指出。

再来看看具体的回归过程，即从 tcache_bin_t 到 run，函数路径为 `tcache_bin_flush_small/arena_dalloc_bin_locked` 。如下图所示：

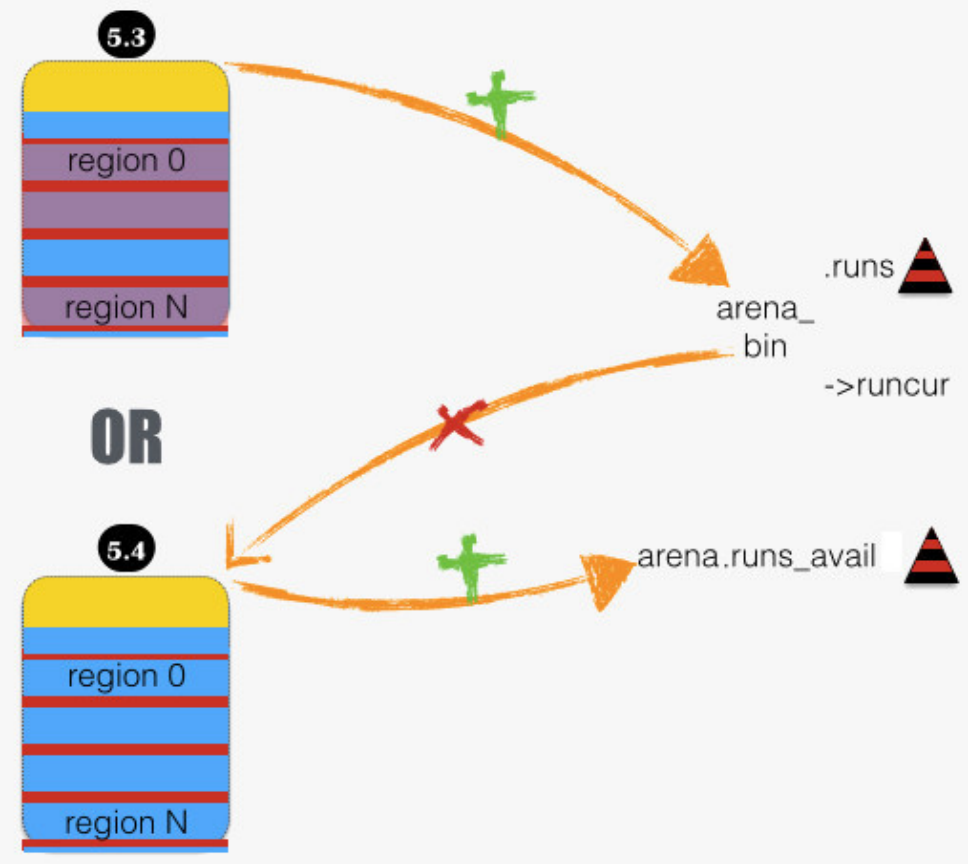


- 5 → 5.1：首先找到 run 结构体，然后才能塞回去。run 结构体在整个 run 块头部（图中黄色部分），我们知道 run 块本身占有多个页面，待“回归”的 region 本身可能在 run 中间某个页，这需要通过 “ *run page offset* ”来定位 run 块的起始位置，从而找到 run 结构体。
- 5.1 → 5.2：释放具体的 region，主要是将 region 在 bitmap 中对应位置 1。两个细节：一是 region 索引通过函数 `arena_run_regind` 获得，该函数试图避免整数除法，来减少计算开销；二是 bitmap 这个数据结构，为 sfu 操作 (set first unallocated)) 优化，而 sfu 正是为了实现 jemalloc 先分配出低地址的设计。bitmap 背后围绕“扫描最先设置位”的指令实现，该指令被封装到函数 ffsl 中。

对应函数路径为 `arena_dalloc_bin_locked/arena_run_reg_dalloc` 。

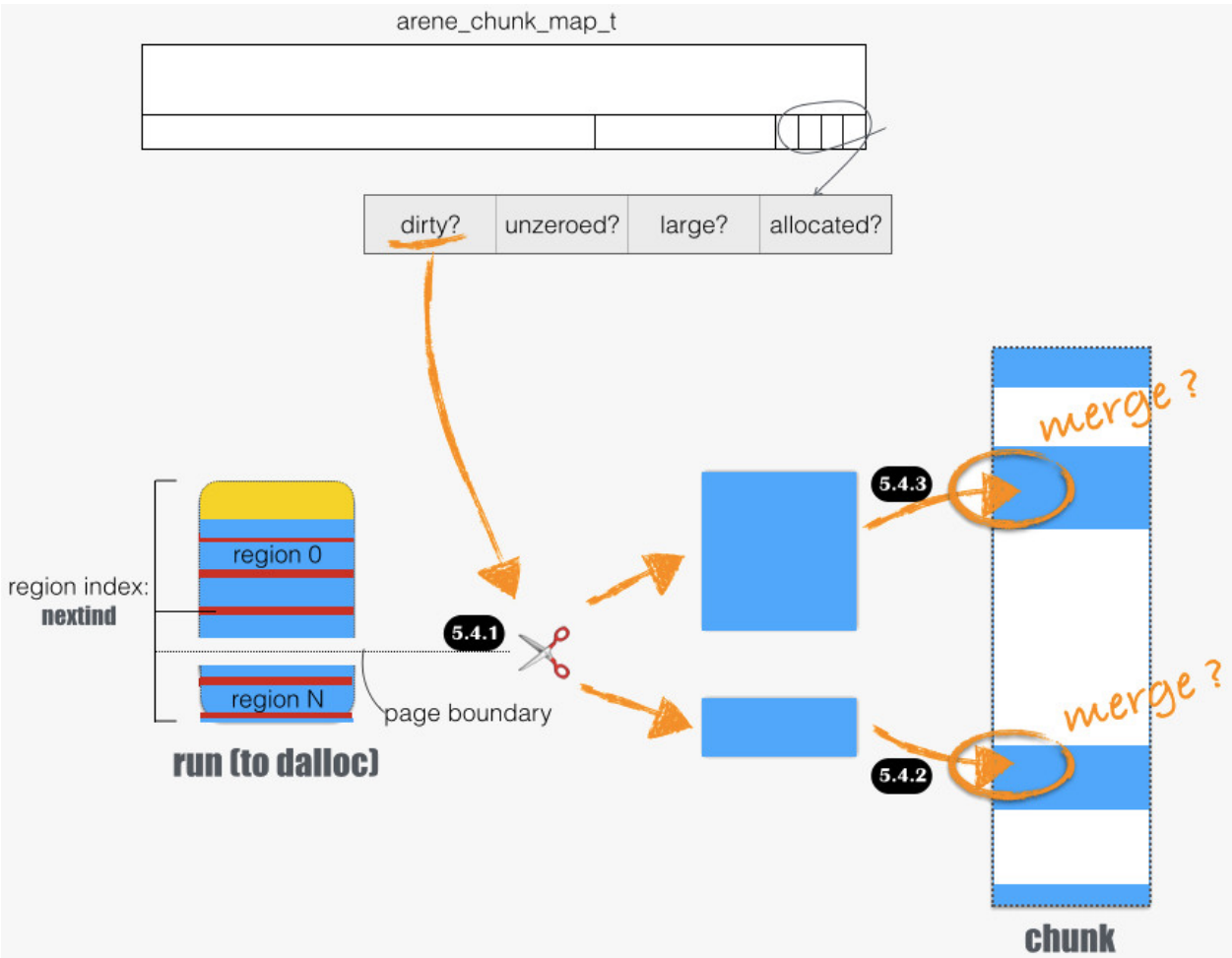
2.3 释放 run

当 region 刷回 run，可能产生两种“深远”影响，如下图：



- 5.3: 当原来全用完的 run，现在有一个 region 可分配了，将其插入所属 bin 中，供分配。
- 5.4：当 run 全空了，则释放之：从所属 bin 中移除，并将空间交还给 arena.runs_avail。

下图展开 5.4 部分细节，函数路径为 `arena_dalloc_bin_locked/arena_dalloc_bin_run`：



在 jemalloc 中，定义了内存区域的额外属性：

- dirty? 一个内存区域被分配出去以后，就是“脏”的。脏的意思是，虚拟地址背后的物理页被分配了。jemalloc 有定期的清理过程，通过 “madvise(addr, length, MADV_DONTNEED)” 释放背后的物理内存。
- unzeored? 某些情况下，从 OS 批发来的内存区域已经初始化成全零了，例如 “madvise(addr, length, MADV_DONTNEED) 以后” 。如此标记后可免除某些清零操作。

dirty 页面一定是 unzeroed。当一个分配出去的、clean 的 run 被释放，并回归 arena.runs_avail，就会标记为 dirty。一个例外是 run for small allocation，通过 nextind 成员我们可以知道，哪些页面没有事实上被分配出。5.4.1 展示了这种情形：

- run 后半部分未被分配，且原 run 是干净的。如 5.4.2：这部分塞回所在 chunk 时，会与之后相邻的、 *干净* 的空闲内存区域合并（如果存在的话），函数路径为 `arena_dalloc_bin_run/arena_run_trim_tail/arena_run_dalloc/arena_run_coalesce`
- run 前半部分被分配了，故是脏的。如 5.4.3：这部分塞回所在 chunk 时，会与之前相邻的、*脏* 的空闲内存区域尝试合并（如果存在的话），函数路径为 `arena_dalloc_bin_run/arena_run_dalloc/arena_run_coalesce`

这里细节再啰嗦下：arena_run_dalloc 函数中会尝试合并。为方便合并，需要在 run 首尾页面的两项 arena_chunk_map_t 赋值为有意义值。

2.4 释放 chunk

释放 run 以后，进一步可能导致 chunk 被释放。函数路径为：`arena_run_dalloc/arena_chunk_dealloc` ：

- `arena_avail_remove` 把 chunk 从 arena.runs_avail 和 arena.chunks_dirty 中移除。
- 进入 arena->spare，并“挤走（chunk_dealloc）”先前的spare chunk。这又体现了维持 cache 热度的原则。

唠叨下 chunk_dealloc，对于走 mmap 从 os 批发的内存（which is mo ren fang shi），该函数会通过 mnumap 进行释放，而不是放到 **chunks_szad** 全局红黑树中。

对于走 sbrk 从 OS 批发的内存，则在放入 **chunks_szad** 全局红黑树前，通过 madvise 通知 OS 来释放背后的物理页面。

所以 chunk 的释放，实际上释放了 chunk 的物理内存。

2.5 垃圾回收

最后，来看下 jemalloc 是如何运功逼走多余内存缓冲的。jemalloc 中，有两个层面的回收，一是 tcache 中多余缓冲赶到 arena 中；二是将 arena.runs_avail 中多余的物理内存释放掉一些。分别是下表的左右两部分：

	tcache_event_hard	arena_maybe_purge
作用对象	tcache	arena

阀值条件	tbin->low_water > 0	npurgeable = ndirty – npurgatory threshold = nactive >> lg_dirty_mult npurgeable > threshold
GC 对象	tcache->next_gc_bin (bin 索引号)	arena->chunks_dirty （红黑树）
GC数量	3/4 * low_water	npurgeable – threshold
反馈1	调整填充率 (lg_fill_div)	ndirty 减少
反馈2	low_water = ncached (GC 完成后的 ncached 另：low_water <= ncached)	npurgatory 减少

来啰嗦下右边过程，函数路径为 `arena_run_dalloc/arena_maybe_purge/arena_purge`：

- 以 chunk 为单位进行。chunk 从 arena.chunks_dirty 中取，实际上是先处理含有分离的脏区域多（ndirty），且因此导致较多碎片（nruns_adjac）的情况。
- 对每个 chunk 用函数 `arena_chunk_purge` 来洗净。该函数先让脏块 “出列”，再进行 “清洗”，最后再 “入列”：
  - “出列”：挑选碎片区域，分配出来（ `arena_run_split_large` ），并放入一个临时链表。所谓碎片是指，两块空闲的区域，地址相接邻，但由于其干净程度不一样，成了碎掉的两片。
  - “清洗”：遍历链表，用 madvise 逐一洗干净。
  - “入列”：释放回去（ `arena_run_dalloc` ），前述释放时会进行合并（ `arena_run_coalesce` ），由此减少了碎片，并可能导致整个chunk 被释放。

3 小结

至此，终于写完了这篇长的要死的文章。忽想起张无忌学完了太极剑，啥招都忘了，只会了剑（贱?）意。那么忘了 jemalloc 那些细招，剩下了哪些贱意呢？随便说说有

1. 循环利用最近扔掉的内存，因为 cache 还有印象，对这块内存比较熟悉。
2. 总是从最低地址分配，创造局部性环境。毕竟 cache 记性不好，跨度太大的内存人家反应慢。
3. 打通任督二脉，及时逼走多余的缓冲内存。

回头看看之前的疑问，层层缓冲会不会使进程过多占用内存呢?

1. tcache 过多的内存缓冲 —— GC 会处理的。
2. arena.bins —— for small allocation，无处理。
3. arena.avail_runs —— GC 会处理的。
4. arena->spare —— 如果是 dirty 的，无处理。保持直到有新释放的 chunk 进入 spare，挤走本座。
5. 内部使用的缓冲 —— 无处理。

6. chunks_szad —— 无物理内存占用。

OK，除了第 2 和 第 4 点需要改进下以外，其他看起来都有应对机制了。其中第 2 点可以通过调整 run 大小 和 档位来缓解。

第 4 点，是一个可优化的地方。

最后，系统物理内存紧张时，能否进行联动，即缩减应用额外的内存缓冲呢？也许我们可以引入一种特殊的 madvise 指示，告诉内核这段地址空间是作缓冲的，在紧张时可以优先释放，且不用交换。
