4 释放

4.1 概览

释放同分配过程相反, 按照一个从ptr -> run -> bin -> chunk -> arena的路径. 但因为涉及page合并和purge, 实现更为复杂. dalloc的入口从je_free -> ifree -> iqalloc -> iqalloct -> idalloct. 对dalloc的分析从idalloct开始. 代码如下,

```
JEMALLOC_ALWAYS_INLINE void
idalloct(void *ptr, bool try_tcache)
{
    ......
    // xf: 获得被释放地址所在的chunk
    chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(ptr);
    if (chunk != ptr)
        arena_dalloc(chunk, ptr, try_tcache);
    else
        huge_dalloc(ptr);
}
```

首先会检测被释放指针ptr所在chunk的首地址与ptr是否一致, 如果是, 则一定为huge region, 否则为small/large. 从这里分为arena和huge两条线.再看一下arena_dalloc,

```
JEMALLOC_ALWAYS_INLINE void
arena_dalloc(arena_chunk_t *chunk, void *ptr, bool try_tcache)
{
    ......
    // xf: 得到页面mapbits
    mapbits = arena_mapbits_get(chunk, pageind);

    if ((mapbits & CHUNK_MAP_LARGE) == 0) {
        if (try_tcache && (tcache = tcache_get(false)) != NULL) {
            // xf: ptr所在tcache的index
            binind = arena_ptr_small_binind_get(ptr, mapbits);
            tcache_dalloc_small(tcache, ptr, binind);
        } else
            arena_dalloc_small(chunk->arena, chunk, ptr, pageind);
    } else {
        size_t size = arena_mapbits_large_size_get(chunk, pageind);
        if (try_tcache && size <= tcache_maxclass && (tcache =
            tcache_get(false)) != NULL) {
            tcache_dalloc_large(tcache, ptr, size);
        } else
            arena_dalloc_large(chunk->arena, chunk, ptr);
    }
}
```

这里通过得到ptr所在page的mapbits, 判断其来自于small还是large. 然后再分别作处理.

因此, 在dalloc一开始基本上分成了small/large/huge三条路线执行. 事实上, 结合前面的知识, large/huge可以看作run和chunk的特例. 所以, 这三条dalloc路线最终会汇到一起, 只需要搞清楚其中最为复杂的small region dalloc就可以了.

无论small/large region, 都会先尝试释放回tcache, 不管其是否从tache中分配而来. 所谓tcache dalloc只不过是将region记录在tbin中, 并不算真正的释放. 除非两种情况, 一是如果当前线程tbin已满, 会直接执行一次tbin flush, 释放出部分tbin空间. 二是如果tcache_event触发发了tache gc, 也会执行flush. 两者的区别在于, 前者会回收指定tbin 1/2的空间, 而后者则释放next_gc_bin相当于3/4 low water数量的空间.

```
JEMALLOC_ALWAYS_INLINE void
tcache_dalloc_small(tcache_t *tcache, void *ptr, size_t binind)
{
    ......
    tbin = &tcache->tbins[binind];
    tbin_info = &tcache_bin_info[binind];
    // xf: 如果当前tbin已满, 则执行flush清理tbin
    if (tbin->ncached == tbin_info->ncached_max) {
        tcache_bin_flush_small(tbin, binind, (tbin_info->ncached_max >>
```

```
            1), tcache);
    }
    // xf: 将被释放的ptr重新push进tbin
    tbin->avail[tbin->ncached] = ptr;
    tbin->ncached++;

    tcache_event(tcache);
}
```

tcache gc和tcache flush在2.7和3.4节中已经介绍, 不再赘述.

4.2 arena_dalloc_bin

small region dalloc的第一步是尝试将region返还给所属的bin. 首要的步骤就是根据用户传入的ptr推算出其所在run的地址.

```
run addr = chunk base + run page offset << LG_PAGE
```

而run page offset根据2.3.3小节的说明, 可以通过ptr所在page的mapbits获得.

```
run page offset = ptr page index - ptr page offset
```

得到run后就进一步拿到所属的bin, 接着对bin加锁并回收, 如下,

```
void
arena_dalloc_bin(arena_t *arena, arena_chunk_t *chunk, void *ptr,
    size_t pageind, arena_chunk_map_t *mapelm)
{
    ......
    // xf: 计算ptr所在run地址.
    run = (arena_run_t *)((uintptr_t)chunk + (uintptr_t)((pageind -
        arena_mapbits_small_runind_get(chunk, pageind)) << LG_PAGE));
    bin = run->bin;

    malloc_mutex_lock(&bin->lock);
    arena_dalloc_bin_locked(arena, chunk, ptr, mapelm);
    malloc_mutex_unlock(&bin->lock);
}
```

lock的内容无非是将region在run内部的bitmap上标记为可用. bitmap unset的过程此处省略, 请参考3.3.1小节中分配算法的解释. 与tcache dalloc类似, 通常情况下region并不会真正释放. 但如果run内部全部为空闲region, 则会进一步触发run的释放.

```
void
arena_dalloc_bin_locked(arena_t *arena, arena_chunk_t *chunk, void *ptr,
    arena_chunk_map_t *mapelm)
{
    ......
    // xf: 通过run回收region, 在bitmap上重新标记region可用.
    arena_run_reg_dalloc(run, ptr);

    // xf: 如果其所在run完全free, 则尝试释放该run.
    // 如果所在run处在将满状态(因为刚刚的释放腾出一个region的空间),
    // 则根据地址高低优先将其交换到current run的位置(MRU).
    if (run->nfree == bin_info->nregs) {
        arena_dissociate_bin_run(chunk, run, bin);
        arena_dalloc_bin_run(arena, chunk, run, bin);
    } else if (run->nfree == 1 && run != bin->runcur)
        arena_bin_lower_run(arena, chunk, run, bin);
    ......
}
```

此外还有一种情况是, 如果原先run本来是满的, 因为前面的释放多出一个空闲位置,就会尝试与current run交换位置. 若当前run比current run地址更低, 会替代后者并成为新的current run, 这样的好处显然可以保证低地址的内存更紧实.

```
static void
arena_bin_lower_run(arena_t *arena, arena_chunk_t *chunk, arena_run_t *run,
```

```
    arena_bin_t *bin)
{
    if ((uintptr_t)run < (uintptr_t)bin->runcur) {
        if (bin->runcur->nfree > 0)
            arena_bin_runs_insert(bin, bin->runcur);
        bin->runcur = run;
        if (config_stats)
            bin->stats.reruns++;
    } else
        arena_bin_runs_insert(bin, run);
}
```

通常情况下, 至此一个small region就释放完毕了, 准确的说是回收了. 但如前面所说, 若整个run都为空闲region, 则进入run dalloc. 这是一个比较复杂的过程.

4.3 small run dalloc

一个non-full的small run被记录在bin内的run tree上, 因此要移除它, 首先要移除其在run tree中的信息, 即arena_dissociate_bin_run.

```
static void
arena_dissociate_bin_run(arena_chunk_t *chunk, arena_run_t *run,
    arena_bin_t *bin)
{
    // xf: 如果当前run为current run, 清除runcur. 否则, 从run tree上remove.
    if (run == bin->runcur)
        bin->runcur = NULL;
    else {
        ......
        if (bin_info->nregs != 1) {
            arena_bin_runs_remove(bin, run);
        }
    }
}
```

接下来要通过arena_dalloc_bin_run()正式释放run, 由于过程稍复杂, 这里先给出整个算法的梗概,

1. 计算nextind region所在page的index. 所谓nextind是run内部clean-dirty region的边界. 如果内部存在clean pages则执行下一步, 否则执行3.
2. 将原始的small run转化成large run, 之后根据上一步得到的nextind将run切割成dirty和clean两部分, 且单独释放掉clean部分.
3. 将待remove的run pages标记为unalloc. 且根据传入的dirty和cleaned两个hint决定标记后的page mapbits的dirty flag.
4. 检查unalloc后的run pages是否可以前后合并. 合并的标准是,
   1) 不超过chunk范围
   2) 前后毗邻的page同样为unalloc
   3) 前后毗邻page的dirty flag与run pages相同.
5. 将合并后(也可能没合并)的unalloc run插入avail-tree.
6. 检查如果unalloc run的大小等于chunk size, 则将chunk释放掉.
7. 如果之前释放run pages为dirty, 则检查当前arena内部的dirty-active pages比例. 若dirty数量超过了active的1/8(Android这里的标准有所不同), 则启动arena purge.否则直接返回.
8. 计算当前arena可以清理的dirty pages数量npurgatory.
9. 从dirty tree上依次取出dirty chunk, 并检查内部的unalloc dirty pages, 将其重新分配为large pages, 并插入到临时的queue中.
10. 对临时队列中的dirty pages执行purge, 返回值为unzeroed标记. 再将purged pages的unzeroed标记设置一遍.
11. 最后对所有purged pages重新执行一遍dalloc run操作, 将其重新释放回avail-tree.

可以看到, 释放run本质上是将其回收至avail-tree, 但额外的dirty pages机制却增加了整个算法的复杂程度. 原因就在于, jemalloc使用了不同

可以看到, 释放run本质上是将其回收至avail-tree. 但额外的dirty page机制却增加了整个算法的复杂程度. 原因就在于, jemalloc使用了不同以往的内存释放方式.

在dlmalloc这样的经典分配器中, 系统内存回收方式更加"古板". 比如在heap区需要top-mostspace存在大于某个threshold的连续free空间时才能进行auto-trimming. 而mmap区则更要等到某个segment全部空闲才能执行munmap. 这对于回收系统内存是极为不利的, 因为条件过于严格.

而jemalloc使用了更为聪明的方式, 并不会直接交还系统内存, 而是通过madvise暂时释放掉页面与物理页面之间的映射. 本质上这同sbrk/munmap之类的调用要达到的目的是类似的, 只不过从进程内部的角度看, 该地址仍然被占用. 但jemalloc对这些使用过的地址都详细做了记录, 因此再分配时可以recycle, 并不会导致对线性地址无休止的开采.

另外, 为了提高对已释放page的利用率, jemalloc将unalloc pages用dirty flag(注意, 这里同page replacement中的含义不同)做了标记(参考2.3.3节中chunkmapbits). 所有pages被分成active, dirty和clean三种. dirty pages表示曾经使用过, 且仍可能关联着物理页面, recycle速度较快. 而clean则代表尚未使用, 或已经通过purge释放了物理页面, 较前者速度慢. 显然, 需要一种内置算法来保持三种page的动态平衡, 以兼顾分配速度和内存占用量. 如果当前dirty pages数量超过了active pages数量的1/2^opt_lg_dirty_mult, 就会启动arena_purge(). 这个值默认是1/8, 如下,

```
static inline void
arena_maybe_purge(arena_t *arena)
{
    ......
    // xf: 如果当前dirty pages全部在执行purging, 则直接返回.
    if (arena->ndirty <= arena->npurgatory)
        return;

    // xf: 检查purageable pages是否超出active-dirty比率, 超出则
    // 执行purge. google在这里增加了ANDROID_ALWAYS_PURGE开关,
    // 打开则总会执行arena_purge(默认是打开的).
#if !defined(ANDROID_ALWAYS_PURGE)
    npurgeable = arena->ndirty - arena->npurgatory;
    threshold = (arena->nactive >> opt_lg_dirty_mult);
    if (npurgeable <= threshold)
        return;
#endif

    // xf: 执行purge
    arena_purge(arena, false);
}
```

但google显然希望对dirty pages管理更严格一些, 以适应移动设备上内存偏小的问题. 这里增加了一个ALWAYS_PURGE的开关, 打开后会强制每次释放时都执行arena_purge.

arena_run_dalloc代码如下,

```
static void
arena_run_dalloc(arena_t *arena, arena_run_t *run, bool dirty, bool cleaned)
{
    ......
    // xf: 如果run pages的dirty flag实际读取为true, 且cleaned不为true,
    // 则同样认为该pages在dalloc后是dirty的, 否则被视为clean(该情况适用于
    // chunk purge后, 重新dalloc时, 此时的run pages虽然dirty flag可能为ture,
    // 但经过purge后应该修改为clean).
    if (cleaned == false && arena_mapbits_dirty_get(chunk, run_ind) != 0)
        dirty = true;
    flag_dirty = dirty ? CHUNK_MAP_DIRTY : 0;

    // xf: 将被remove的run标记为unalloc pages. 前面的判断如果是dirty, 则pages
    // mapbits将带有dirty flag, 否则将不带有dirty flag.
    if (dirty) {
        arena_mapbits_unallocated_set(chunk, run_ind, size,
            CHUNK_MAP_DIRTY);
        arena_mapbits_unallocated_set(chunk, run_ind+run_pages-1, size,
            CHUNK_MAP_DIRTY);
    } else {
        arena_mapbits_unallocated_set(chunk, run_ind, size,
            arena_mapbits_unzeroed_get(chunk, run_ind));
```

```
        arena_mapbits_unallocated_set(chunk, run_ind+run_pages-1, size,
            arena_mapbits_unzeroed_get(chunk, run_ind+run_pages-1));
    }

    // xf: 尝试将被remove run与前后unalloc pages 合并.
    arena_run_coalesce(arena, chunk, &size, &run_ind, &run_pages,
        flag_dirty);
    ......

    // xf: 将执行过合并后的run重新insert到avail-tree
    arena_avail_insert(arena, chunk, run_ind, run_pages, true, true);

    // xf: 检查如果合并后的size已经完全unallocated, 则dalloc整个chunk
    if (size == arena_maxclass) {
        ......
        arena_chunk_dalloc(arena, chunk);
    }
    if (dirty)
        arena_maybe_purge(arena);
}
```

coalesce代码如下,

```
static void
arena_run_coalesce(arena_t *arena, arena_chunk_t *chunk, size_t *p_size,
    size_t *p_run_ind, size_t *p_run_pages, size_t flag_dirty)
{
    ......
    // xf: 尝试与后面的pages合并
    if (run_ind + run_pages < chunk_npages &&
        arena_mapbits_allocated_get(chunk, run_ind+run_pages) == 0 &&
        arena_mapbits_dirty_get(chunk, run_ind+run_pages) == flag_dirty) {
        size_t nrun_size = arena_mapbits_unallocated_size_get(chunk,
            run_ind+run_pages);
        size_t nrun_pages = nrun_size >> LG_PAGE;
        ......
        // xf: 如果与后面的unalloc pages合并, remove page时后方的adjacent
        // hint应为true
        arena_avail_remove(arena, chunk, run_ind+run_pages, nrun_pages,
            false, true);

        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);
    }

    // xf: 尝试与前面的pages合并
    if (run_ind > map_bias && arena_mapbits_allocated_get(chunk,
        run_ind-1) == 0 && arena_mapbits_dirty_get(chunk, run_ind-1) ==
        flag_dirty) {
        ......
    }

    *p_size = size;
    *p_run_ind = run_ind;
    *p_run_pages = run_pages;
}
```

avail-tree remove代码如下,

```
static void
arena_avail_remove(arena_t *arena, arena_chunk_t *chunk, size_t pageind,
    size_t npages, bool maybe_adjac_pred, bool maybe_adjac_succ)
{
```

```
    ......
    // xf: 该调用可能将导致chunk内部的碎片化率改变, 从而影响其在dirty tree
    // 中的排序. 因此, 在正式remove之前需要将chunk首先从dirty tree中remove,
    // 待更新内部ndirty后, 再将其重新insert回dirty tree.
    if (chunk->ndirty != 0)
        arena_chunk_dirty_remove(&arena->chunks_dirty, chunk);

    // xf: maybe_adjac_pred/succ是外界传入的hint, 根据该值检查前后是否存在
    // clean-dirty边界. 若存在边界, 则remove avail pages后边界将减1.
    if (maybe_adjac_pred && arena_avail_adjac_pred(chunk, pageind))
        chunk->nruns_adjac--;
    if (maybe_adjac_succ && arena_avail_adjac_succ(chunk, pageind, npages))
        chunk->nruns_adjac--;
    chunk->nruns_avail--;
    ......

    // xf: 更新arena及chunk中dirty pages统计.
    if (arena_mapbits_dirty_get(chunk, pageind) != 0) {
        arena->ndirty -= npages;
        chunk->ndirty -= npages;
    }
    // xf: 如果chunk内部dirty不为0, 将其重新insert到arena dirty tree.
    if (chunk->ndirty != 0)
        arena_chunk_dirty_insert(&arena->chunks_dirty, chunk);

    // xf: 从chunk avail-tree中remove掉unalloc pages.
    arena_avail_tree_remove(&arena->runs_avail, arena_mapp_get(chunk,
        pageind));
}
```

从avail-tree上remove pages可能会改变当前chunk内部clean-dirty碎片率, 因此一开始要将其所在chunk从dirty tree上remove, 再从avail-tree上remove pages.另外, arena_avail_insert()的算法同remove是一样的, 只是方向相反, 不再赘述.

4.4 arena purge

清理arena的方式是按照从小到大的顺序遍历一棵dirty tree, 直到将dirty pages降低到threshold以下. dirty tree挂载所有dirty chunks, 同其他tree的区别在于它的cmp函数较特殊, 决定了最终的purging order, 如下,

```
static inline int
```

4.4 arena purge

清理arena的方式是按照从小到大的顺序遍历一棵dirty tree, 直到将dirty pages降低到threshold以下. dirty tree挂载所有dirty chunks, 同其他tree的区别在于它的cmp函数较特殊, 决定了最终的purging order, 如下,

```
static inline int
arena_chunk_dirty_comp(arena_chunk_t *a, arena_chunk_t *b)
{
    ......
    if (a == b)
        return (0);

    {
        size_t a_val = (a->nruns_avail - a->nruns_adjac) *
            b->nruns_avail;
        size_t b_val = (b->nruns_avail - b->nruns_adjac) *
            a->nruns_avail;

        if (a_val < b_val)
            return (1);
        if (a_val > b_val)
            return (-1);
    }
    {
        uintptr_t a_chunk = (uintptr_t)a;
        uintptr_t b_chunk = (uintptr_t)b;
        int ret = ((a_chunk > b_chunk) - (a_chunk < b_chunk));
```

```
        if (a->nruns_adjac == 0) {
            assert(b->nruns_adjac == 0);
            ret = -ret;
        }
        return (ret);
    }
}
```

jemalloc在这里给出的算法是这样的,

1. 首先排除short cut, 即a和b相同的特例.
2. 计算a, b的fragmentation, 该数值越高, 相应的在dirty tree上就越靠前.其计算方法为,

   ```
   当前平均avail run大小  所有avail run数量 - 边界数量
   --------------------- = ----------------------------
   去碎片后的平均大小  所有avail run数量
   ```

   注意, 这个fragment不是通常意义理解的碎片. 这里指由于clean-dirty 边界形成的所谓碎片, 并且是可以通过purge清除掉的, 如图,

   ```
   nruns_adjac = 2
   +--------+----------+--------+-------+---------+----------+--------+-----
   | dirty | clean | | clean | dirty | | dirty | ...
   +--------+----------+--------+-------+---------+----------+--------+-----
   ^ ^
   | |
   +--adjac #0 +--adjac #1
   ```

3. 当a, b的fragmentation相同时, 同通常的方法类似, 按地址大小排序. 但若nruns_adjac为0, 即不存在clean-dirty边界时, 反而会将低地址chunk排到后面. 因为adjac为0的chunk再利用价值是比较高的, 所以放到后面可以增加其在purge中的幸存几率, 从而提升recycle效率.

这里需要说明的是, jemalloc这个cmp函数个人觉得似乎有问题, 实际跟踪代码也发现其并不能更优先purge高碎片率的chunk. 但与其本人证实并未得到信服的说明. 但这套算法仅仅在3.x版本中有效, 在最新的4.x中则完全抛弃了现有的回收算法.

purge代码如下,

```
static void
arena_purge(arena_t *arena, bool all)
{
    ......
    // xf: 计算purgeable pages, 结果加入到npurgatory信息中.
    npurgatory = arena_compute_npurgatory(arena, all);
    arena->npurgatory += npurgatory;

    // xf: 从dirty chunk tree上逐chunk执行purge, 直到期望值npurgatory为0
    while (npurgatory > 0) {
        ......
        chunk = arena_chunk_dirty_first(&arena->chunks_dirty);
        // xf: traversal结束, 当前线程无法完成purge任务, 返回.
        if (chunk == NULL) {
            arena->npurgatory -= npurgatory;
            return;
        }
        npurgeable = chunk->ndirty;
        ......
        // xf:  如果当前chunk中purgeable大于前期计算的purgatory,
        // 且其clean-dirty碎片为0, 则让当前线程负责purge所有prgeable pages.
        // 原因是为了尽可能避免避免多个线程对该chunk的purge竞争.
        if (npurgeable > npurgatory && chunk->nruns_adjac == 0) {
            arena->npurgatory += npurgeable - npurgatory;
            npurgatory = npurgeable;
        }
        arena->npurgatory -= npurgeable;
        npurgatory -= npurgeable;
        npurged = arena_chunk_purge(arena, chunk, all);
        // xf: 计算purge期望值npurgatory和实际purge值npurged差值
```

```
        nunpurged = npurgeable - npurged;
        arena->npurgatory += nunpurged;
        npurgatory += nunpurged;
    }
}
```

chunk purge如下,

```
static inline size_t
arena_chunk_purge(arena_t *arena, arena_chunk_t *chunk, bool all)
{
    ......
    if (chunk == arena->spare) {
        ......
        arena_chunk_alloc(arena);
    }
    ......
    // xf: 为了减小arena purge时arena lock的暂停时间, 先将所有满足
    // 需求的unalloc dirty pages重新"alloc"并保存, 待purge结束再重新
    // 释放回avail-tree.
    arena_chunk_stash_dirty(arena, chunk, all, &mapelms);
    npurged = arena_chunk_purge_stashed(arena, chunk, &mapelms);
    arena_chunk_unstash_purged(arena, chunk, &mapelms);

    return (npurged);
}
```

chunk purge重点在于这是一个线性查找dirty pages过程, jemalloc在这里会导致性能下降. 更糟糕的是, 之前和之后都是在arena lock被锁定的条件下被执行, 绑定同一arena的线程不得不停下工作. 因此, 在正式purge前需要先把unalloc dirtypages全部临时分配出来, 当purging时解锁arena lock, 而结束后再一次将它们全部释放.

stash dirty代码,

```
static void
arena_chunk_stash_dirty(arena_t *arena, arena_chunk_t *chunk, bool all,
    arena_chunk_mapelms_t *mapelms)
{
    ......
    for (pageind = map_bias; pageind < chunk_npages; pageind += npages) {
        arena_chunk_map_t *mapelm = arena_mapp_get(chunk, pageind);
        if (arena_mapbits_allocated_get(chunk, pageind) == 0) {
            ......
            if (arena_mapbits_dirty_get(chunk, pageind) != 0 &&
                (all || arena_avail_adjac(chunk, pageind,
                npages))) {
                arena_run_t *run = (arena_run_t *)((uintptr_t)
                    chunk + (uintptr_t)(pageind << LG_PAGE));
                // xf: 暂时将这些unalloc dirty pages通过split large
                // 重新分配出来.
                arena_run_split_large(arena, run, run_size,
                    false);
                // 加入临时列表, 留待后用.
                ql_elm_new(mapelm, u.ql_link);
                ql_tail_insert(mapelms, mapelm, u.ql_link);
            }
        } else {
            //xf: 跳过allocated pages
            ......
        }
    }
    ......
}
```

stash时会根据传入的hint all判断, 如果为false, 只会stash存在clean-dirty adjac的pages, 否则会全部加入列表.

purge stashed pages代码如下

purge stashed pages代码如下,

```
static size_t
arena_chunk_purge_stashed(arena_t *arena, arena_chunk_t *chunk,
    arena_chunk_mapelms_t *mapelms)
{
    ......
    // xf: 暂时解锁arena lock, 前面已经realloc过, 这里不考虑contention问题.
    malloc_mutex_unlock(&arena->lock);
    ......
    ql_foreach(mapelm, mapelms, u.ql_link) {
        ......
        // xf: 逐个purge dirty page, 返回pages是否unzeroed.
        unzeroed = pages_purge((void *)((uintptr_t)chunk + (pageind <<
            LG_PAGE)), (npages << LG_PAGE));
        flag_unzeroed = unzeroed ? CHUNK_MAP_UNZEROED : 0;

        // xf: 逐pages设置unzeroed标志.
        for (i = 0; i < npages; i++) {
            arena_mapbits_unzeroed_set(chunk, pageind+i,
                flag_unzeroed);
        }
        ......
    }
    // xf: purging结束重新lock arena
    malloc_mutex_lock(&arena->lock);
    ......
    return (npurged);
}
```

这里要注意的是, 在page purge过后, 会逐一设置unzero flag. 这是因为有些操作系统在demand page后会有一步zero-fill-on-demand. 因此, 被purge过的clean page当再一次申请到物理页面时会全部填充为0.

unstash代码,

```
static void
arena_chunk_unstash_purged(arena_t *arena, arena_chunk_t *chunk,
    arena_chunk_mapelms_t *mapelms)
{
    ......
    for (mapelm = ql_first(mapelms); mapelm != NULL;
        mapelm = ql_first(mapelms)) {
        ......
        run = (arena_run_t *)((uintptr_t)chunk + (uintptr_t)(pageind <<
            LG_PAGE));
        ql_remove(mapelms, mapelm, u.ql_link);
        arena_run_dalloc(arena, run, false, true);
    }
}
```

unstash需要再一次调用arena_run_dalloc()以释放临时分配的pages. 要注意此时我们已经位于arena_run_dalloc调用栈中, 而避免无限递归重入依靠参数cleaned flag.

4.5 arena chunk dalloc

当free chunk被jemalloc释放时, 根据局部性原理, 会成为下一个spare chunk而保存起来, 其真身并未消散. 而原先的spare则会根据内部dalloc方法被处理掉.

```
static void
arena_chunk_dalloc(arena_t *arena, arena_chunk_t *chunk)
{
    ......
    // xf: 将chunk从avail-tree上remove
    arena_avail_remove(arena, chunk, map_bias, chunk_npages-map_bias,
        false, false);

    // xf: 如果spare不为空, 则将被释放的chunk替换原spare chunk
```

```
    // xf: 如果spare不为空, 则将被释放的chunk替换原spare chunk.
    if (arena->spare != NULL) {
        arena_chunk_t *spare = arena->spare;

        arena->spare = chunk;
        arena_chunk_dalloc_internal(arena, spare);
    } else
        arena->spare = chunk;
}
```

同chunk alloc一样, chunk dalloc算法也是可定制的. jemalloc提供的默认算法chunk_dalloc_default最终会调用chunk_unmap, 如下,

```
void
chunk_unmap(void *chunk, size_t size)
{
    ......
    // xf: 如果启用dss, 且当前chunk在dss内, 将其record在dss tree上.
    // 否则如果就记录在mmap tree上, 或者直接munmap释放掉.
    if (have_dss && chunk_in_dss(chunk))
        chunk_record(&chunks_szad_dss, &chunks_ad_dss, chunk, size);
    else if (chunk_dalloc_mmap(chunk, size))
        chunk_record(&chunks_szad_mmap, &chunks_ad_mmap, chunk, size);
}
```

在3.3.5小节中alloc时会根据dss和mmap优先执行recycle. 源自在dalloc时record在四棵chunk tree上的记录. 但同spare记录的不同, 这里的记录仅仅只剩下躯壳, record时会强行释放物理页面, 因此recycle速度相比spare较慢.

chunk record算法如下,

1. 先purge chunk内部所有pages
2. 预分配base node, 以记录释放后的chunk. 这里分配的node到后面可能没有用,提前分配是因为接下来要加锁chunks_mtx. 而如果在临界段内再分配base node, 则可能因为base pages不足而申请新的chunk, 这样一来就会导致dead lock.
3. 寻找与要插入chunk的毗邻地址. 首先尝试与后面的地址合并, 成功则用后者的base node记录, 之后执行5.
4. 合并失败, 用预分配的base node记录chunk.
5. 尝试与前面的地址合并.
6. 如果预分配的base node没有使用, 释放掉.

代码如下,

```
static void
chunk_record(extent_tree_t *chunks_szad, extent_tree_t *chunks_ad, void *chunk,
    size_t size)
{
    ......
    // xf: purge all chunk pages
    unzeroed = pages_purge(chunk, size);

    // xf: 预先分配extent_node以记录chunk. 如果该chunk可以进行合并, 该node
    // 可能并不会使用. 这里预先分配主要是避免dead lock. 因为某些情况
    // base_node_alloc同样可能会alloc base chunk, 由于后面chunk mutex被lock,
    // 那样将导致dead lock.
    xnode = base_node_alloc();
    xprev = NULL;

    malloc_mutex_lock(&chunks_mtx);
    // xf: 首先尝试与后面的chunk合并.
    key.addr = (void *)((uintptr_t)chunk + size);
    node = extent_tree_ad_nsearch(chunks_ad, &key);

    if (node != NULL && node->addr == key.addr) {
        extent_tree_szad_remove(chunks_szad, node);
```

```
        node->addr = chunk;
        node->size += size;
        node->zeroed = (node->zeroed && (unzeroed == false));
        extent_tree_szad_insert(chunks_szad, node);
    } else {
        // xf: 合并失败, 用提前分配好的xnode保存当前chunk信息.
        if (xnode == NULL) {
            goto label_return;
        }
        node = xnode;
        xnode = NULL;
        node->addr = chunk;
        node->size = size;
        node->zeroed = (unzeroed == false);
        extent_tree_ad_insert(chunks_ad, node);
        extent_tree_szad_insert(chunks_szad, node);
    }

    // xf: 再尝试与前面的chunk合并
    prev = extent_tree_ad_prev(chunks_ad, node);
    if (prev != NULL && (void *)((uintptr_t)prev->addr + prev->size) ==
        chunk) {
        ......
    }

label_return:
    malloc_mutex_unlock(&chunks_mtx);
    // xf: 如果预先分配的node没有使用, 则在此将之销毁
    if (xnode != NULL)
        base_node_dalloc(xnode);
    if (xprev != NULL)
        base_node_dalloc(xprev);
}
```

最后顺带一提, 对于mmap区的pages, jemalloc也可以直接munmap, 前提是需要在jemalloc_internal_defs.h中开启JEMALLOC_MUNMAP, 这样就不会执行pages purge.默认该选项是不开启的. 但源自dss区中的分配则不存在反向释放一说, 默认jemalloc也不会优先选择dss就是了.

```
bool
chunk_dalloc_mmap(void *chunk, size_t size)
{

    if (config_munmap)
        pages_unmap(chunk, size);

    return (config_munmap == false);
}
```

4.6 large/huge dalloc

前面说过large/huge相当于以run和chunk为粒度的特例. 因此对于arena dalloc large来说, 最终就是arena_run_dalloc,

```
void
arena_dalloc_large_locked(arena_t *arena, arena_chunk_t *chunk, void *ptr)
{

    if (config_fill || config_stats) {
        size_t pageind = ((uintptr_t)ptr - (uintptr_t)chunk) >> LG_PAGE;
        size_t usize = arena_mapbits_large_size_get(chunk, pageind);

        arena_dalloc_junk_large(ptr, usize);
        if (config_stats) {
            arena->stats.ndalloc_large++;
            arena->stats.allocated_large -= usize;
            arena->stats.lstats[(usize >> LG_PAGE) - 1].ndalloc++;
            arena->stats.lstats[(usize >> LG_PAGE) - 1].curruns--;
```

```
        }
    }

    arena_run_dalloc(arena, (arena_run_t *)ptr, true, false);
}
```

而huge dalloc, 则是在huge tree上搜寻, 最终执行chunk_dalloc,

```
void
huge_dalloc(void *ptr)
{
```