# 3 分配

## 3.1 概览

在2.3.2节中得知, jemalloc将size class划分成small, large, huge三种类型. 分配时这三种类型分别按照不同的算法执行. 后面的章节也将按照这个类型顺序描述.

总体来说, jemalloc分配函数从je_malloc入口开始, 经过,

```
je_malloc -> imalloc_body -> imalloc -> imalloct ---> arena_malloc
                                                  |
                                                  +-> huge_malloc
```

实际执行分配的分别是对应small/large的arena malloc 和对应huge的huge malloc.
分配算法可以概括如下,

1. 首先检查jemalloc是否初始化, 如果没有则初始化jemalloc, 并标记全局malloc_initialized标记.
2. 检查请求size是否大于huge, 如果是则执行8, 否则进入下一步.
3. 执行arena_malloc, 首先检查size是否小于等于small maxclass, 如果是则下一步, 否则执行6.
4. 如果允许且当前线程已绑定tcache, 则从tcache分配small, 并返回. 否则下一步.
5. choose arena, 并执行arena malloc small, 返回.
6. 如果允许且当前线程已绑定tcache, 则从tcache分配large, 并返回. 否则下一步.
7. choose arena, 并执行arena malloc large, 返回.
8. 执行huge malloc, 并返回.

## 3.2 初始化

jemalloc通过全局标记malloc_initialized指代是否初始化. 在每次分配时, 需要检查该标记,如果没有则执行malloc_init.

但通常条件下, malloc_init是在jemalloc库被载入之前就调用的. 通过gcc的编译扩展属性"constructor"实现,

```
JEMALLOC_ATTR(constructor)
static void
jemalloc_constructor(void)
{
    malloc_init();
}
```

接下来由malloc_init_hard执行各项初始化工作. 这里首先需要考虑的是多线程初始化导致的重入, jemalloc通过malloc_initialized和malloc_initializer两个标记来识别.

```
malloc_mutex_lock(&init_lock);
// xf: 如果在获得init_lock前已经有其他线程完成malloc_init,
// 或者当前线程在初始化过程中执行了malloc, 导致递归初始化, 则立即退出.
if (malloc_initialized || IS_INITIALIZER) {
    malloc_mutex_unlock(&init_lock);
    return (false);
}
// xf: 如果开启多线程初始化, 需要执行busy wait直到malloc_init在另外线程中
// 执行完毕后返回.
#ifdef JEMALLOC_THREADED_INIT
if (malloc_initializer != NO_INITIALIZER && IS_INITIALIZER == false) {
    do {
        malloc_mutex_unlock(&init_lock);
        CPU_SPINWAIT;
        malloc_mutex_lock(&init_lock);
    } while (malloc_initialized == false);
```

```
    malloc_mutex_unlock(&init_lock);
    return (false);
}
#endif
// xf: 将当前线程注册为initializer
malloc_initializer = INITIALIZER;
```

初始化工作由各个xxx_boot函数完成. 注意的是, boot函数返回false代表成功, 否则代表失败.

- **tsd boot:** Thread specific data初始化, 主要负责tsd析构函数数组长度初始化.
- **base boot:** base是jemalloc内部用于meta data分配的保留区域, 使用内部独立的分配方式. base boot负责base node和base mutex的初始化.
- **chunk boot:** 主要有三件工作,
    - 确认chunk_size和chunk_npages.
    - chunk_dss_boot, chunk dss指chunk分配的dss(Data Storage Segment方式. 其中涉及dss_base, dss_prev指针的初始化工作.
    - chunk tree的初始化, 在chunk recycle时要用到.
- **arena boot:** 主要是确认arena_maxclass, 这个size代表arena管理的最大region,超过该值被认为huge region.在2.2.2小节中有过介绍, 先通过多次迭代计算出map_bias, 再用chunksize - (map_bias << LG_PAGE)即可得到.另外还对另一个重要的静态数组arena_bin_info执行了初始化. 可参考2.3.2介绍class size的部分.
- **tcache boot:** 分为tcache_boot0和tcache_boot1两个部分执行.前者负责tcache所有静态信息, 包含tcache_bin_info, stack_nelms, nhbins等的初始化.后者负责tcache tsd数据的初始化(tcache保存到线程tsd中).
- **huge boot:** 负责huge mutex和huge tree的初始化.

除此之外, 其他重要的初始化还包括分配arenas数组. 注意arenas是一个指向指针数组的指针, 因此各个arena还需要动态创建. 这里**jemalloc采取了lazy create的方式, 只有当choose_arena时才可能由choose_arena_hard创建真实的arena实例. 但在malloc_init中, 首个arena还是会在此时创建, 以保证基本的分配.**

相关代码如下,

```
arena_t *init_arenas[1];
......

// xf: 此时narenas_total只有1
narenas_total = narenas_auto = 1;
arenas = init_arenas;
memset(arenas, 0, sizeof(arena_t *) * narenas_auto);

// xf: 创建首个arena实例, 保存到临时数组init_arenas中
arenas_extend(0);
......

// xf: 获得当前系统核心数量
ncpus = malloc_ncpus();
......

// xf: 默认的narenas为核心数量的4倍
if (opt_narenas == 0) {
    if (ncpus > 1)
        opt_narenas = ncpus << 2;
    else
        opt_narenas = 1;
}

// xf: android中max arenas限制为2, 参考mk文件
#if defined(ANDROID_MAX_ARENAS)
if (opt_narenas > ANDROID_MAX_ARENAS)
    opt_narenas = ANDROID_MAX_ARENAS;
```

```
#endif
narenas_auto = opt_narenas;
......

// xf: 修正narenas_total
narenas_total = narenas_auto;

// xf: 根据total数量, 构造arenas数组, 并置空
arenas = (arena_t **)base_alloc(sizeof(arena_t *) * narenas_total);
......
memset(arenas, 0, sizeof(arena_t *) * narenas_total);

// xf: 将之前的首个arena实例指针保存到新构造的arenas数组中
arenas[0] = init_arenas[0];
```

### 3.3 Small allocation (Arena)

先介绍最复杂的arena malloc small.

1. 先通过small_size2bin查到bin index(2.4.3节有述).
2. 若对应bin中current run可用则进入下一步, 否则执行4.
3. 由arena_run_reg_alloc在current run中直接分配, 并返回.
4. current run耗尽或不存在, 尝试从bin中获得可用run以填充current run, 成功则执行9, 否则进入下一步.
5. 当前bin的run tree中没有可用run, 转而从arena的avail-tree上尝试切割一个可用run, 成功则执行9, 否则进入下一步.
6. 当前arena没有可用的空闲run, 构造一个新的chunk以分配new run. 成功则执行9, 否则进入下一步.
7. chunk分配失败, 再次查询arena的avail-tree, 查找可用run. 成功则执行9,否则进入下一步.
8. alloc run尝试彻底失败, 则再次查询当前bin的run-tree, 尝试获取run.
9. 在使用新获得run之前, 重新检查当前bin的current run, 如果可用(这里有两种可能, 其一是其他线程可能通过free释放了多余的region或run, 另一种可能是抢在当前线程之前已经分配了新run), 则使用其分配, 并返回.另外, 如果当前手中的new run是空的, 则将其释放掉. 否则若其地址比current run更低, 则交换二者, 将旧的current run插回avail-tree.
10. 在new run中分配region, 并返回.

```
void *arena_malloc_small(arena_t *arena, size_t size, bool zero)
{
    ......
    // xf: 根据size计算bin index
    binind = small_size2bin(size);
    assert(binind < NBINS);
    bin = &arena->bins[binind];
    size = small_bin2size(binind);

    malloc_mutex_lock(&bin->lock);
    // xf: 如果bin中current run不为空, 且存在空闲region, 则在current
    // run中分配. 否则在其他run中分配.
    if ((run = bin->runcur) != NULL && run->nfree > 0)
        ret = arena_run_reg_alloc(run, &arena_bin_info[binind]);
    else
        ret = arena_bin_malloc_hard(arena, bin);

    // xf: 若返回null, 则分配失败.
    if (ret == NULL) {
        malloc_mutex_unlock(&bin->lock);
        return (NULL);
    }
    ......

    return (ret);
}
```

#### 3.3.1 arena_run_reg_alloc

1. 首先根据bin_info中的静态信息bitmap_offset计算bitmap基址.
2. 扫描当前run的bitmap, 获得第一个free region所在的位置.
3. region地址 = run基址 + 第一个region的偏移量 + free region索引 * region内部size.

```
static inline void *
arena_run_reg_alloc(arena_run_t *run, arena_bin_info_t *bin_info)
{
    ......
    // xf: 计算bitmap基址
    bitmap_t *bitmap = (bitmap_t *)((uintptr_t)run +
        (uintptr_t)bin_info->bitmap_offset);
    ......

    // xf: 获得当前run中第一个free region所在bitmap中的位置
    regind = bitmap_sfu(bitmap, &bin_info->bitmap_info);
    // xf: 计算返回值
    ret = (void *)((uintptr_t)run + (uintptr_t)bin_info->reg0_offset +
        (uintptr_t)(bin_info->reg_interval * regind));
    // xf: free减1
    run->nfree--;
    ......

    return (ret);
}
```

其中bitmap_sfu是执行bitmap遍历并设置第一个unset bit. 如2.5节所述, bitmap由多级组成, 遍历由top level开始循环迭代, 直至bottom level.

```
JEMALLOC_INLINE size_t
bitmap_sfu(bitmap_t *bitmap, const bitmap_info_t *binfo)
{
    ......
    // xf: 找到最高级level, 并计算ffs
    i = binfo->nlevels - 1;
    g = bitmap[binfo->levels[i].group_offset];
    bit = jemalloc_ffsl(g) - 1;
    // xf: 循环迭代, 直到level0
    while (i > 0) {
        i--;
        // xf: 根据上一级level的结果, 计算当前level的group
        g = bitmap[binfo->levels[i].group_offset + bit];
        // xf: 根据当前level group, 计算下一级需要的bit
        bit = (bit << LG_BITMAP_GROUP_NBITS) + (jemalloc_ffsl(g) - 1);
    }

    // xf: 得到level0的bit, 设置bitmap
    bitmap_set(bitmap, binfo, bit);
    return (bit);
}
```

bitmap_set同普通bitmap操作没有什么区别, 只是在set/unset之后需要反向迭代更新各个高等级level对应的bit位.

```
JEMALLOC_INLINE void
bitmap_set(bitmap_t *bitmap, const bitmap_info_t *binfo, size_t bit)
{
    ......
    // xf: 计算该bit所在level0中的group
    goff = bit >> LG_BITMAP_GROUP_NBITS;
    // xf: 得到目标group的值g
    gp = &bitmap[goff];
```

```
    g = *gp;
    // xf: 根据remainder, 找到target bit, 并反转
    g ^= 1LU << (bit & BITMAP_GROUP_NBITS_MASK);
    *gp = g;
    ......
    // xf: 若target bit所在group为0, 则需要更新highlevel的相应bit,
    // 是bitmap_sfu的反向操作.
    if (g == 0) {
        unsigned i;
        for (i = 1; i < binfo->nlevels; i++) {
            bit = goff;
            goff = bit >> LG_BITMAP_GROUP_NBITS;
            gp = &bitmap[binfo->levels[i].group_offset + goff];
            g = *gp;
            assert(g & (1LU << (bit & BITMAP_GROUP_NBITS_MASK)));
            g ^= 1LU << (bit & BITMAP_GROUP_NBITS_MASK);
            *gp = g;
            if (g != 0)
                break;
        }
    }
}
```

### 3.3.2 arena_bin_malloc_hard

1. 从bin中获得可用的nonfull run, 这个过程中bin->lock有可能被解锁.
2. 暂不使用new run, 返回检查bin->runcur是否重新可用. 如果是, 则直接在其中分配region(其他线程在bin lock解锁期间可能提前修改了runcur). 否则, 执行4.
3. 重新检查1中得到的new run, 如果为空, 则释放该run.否则与当前runcur作比较, 若地址低于runcur, 则与其做交换. 将旧的runcur插回run tree. 并返回new rigion.
4. 用new run填充runcur, 并在其中分配region, 返回.

```
static void *
arena_bin_malloc_hard(arena_t *arena, arena_bin_t *bin)
{
    ......
    // xf: 获得bin对应的arena_bin_info, 并将current run置空
    binind = arena_bin_index(arena, bin);
    bin_info = &arena_bin_info[binind];
    bin->runcur = NULL;

    // xf: 从指定bin中获得一个可用的run
    run = arena_bin_nonfull_run_get(arena, bin);

    // 对bin->runcur做重新检查. 如果可用且未耗尽, 则直接分配.
    if (bin->runcur != NULL && bin->runcur->nfree > 0) {
        ret = arena_run_reg_alloc(bin->runcur, bin_info);

        // xf: 若new run为空, 则将其释放. 否则重新插入run tree.
        if (run != NULL) {
            arena_chunk_t *chunk;
            chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(run);
            if (run->nfree == bin_info->nregs)
                arena_dalloc_bin_run(arena, chunk, run, bin);
            else
                arena_bin_lower_run(arena, chunk, run, bin);
        }
        return (ret);
    }

    if (run == NULL)
        return (NULL);
```

```
    // xf: 到这里在bin->runcur中分配失败, 用当前新获得的run填充current run
    bin->runcur = run;

    // xf: 在new run中分配region
    return (arena_run_reg_alloc(bin->runcur, bin_info));
}
```

1. 尝试在当前run tree中寻找可用run, 成功则返回, 否则进入下一步.
2. 解锁bin lock, 并加锁arena lock, 尝试在当前arena中分配new run.之后重新解锁arena lock, 并加锁bin lock. 如果成功则返回, 否则进入下一步.
3. 分配失败, 重新在当前run tree中寻找一遍可用run.

```
static arena_run_t *
arena_bin_nonfull_run_get(arena_t *arena, arena_bin_t *bin)
{
    ......
    // xf: 尝试从当前run tree中寻找一个可用run, 如果存在就返回
    run = arena_bin_nonfull_run_tryget(bin);
    if (run != NULL)
        return (run);
    ......

    // xf: 打开bin lock, 让其他线程可以操作当前的bin tree
    malloc_mutex_unlock(&bin->lock);
    // xf: 锁住arena lock, 以分配new run
    malloc_mutex_lock(&arena->lock);

    // xf: 尝试分配new run
    run = arena_run_alloc_small(arena, bin_info->run_size, binind);
    if (run != NULL) {
        // 初始化new run和bitmap
        bitmap_t *bitmap = (bitmap_t *)((uintptr_t)run +
            (uintptr_t)bin_info->bitmap_offset);

        run->bin = bin;
```

### 3.3.2 arena_bin_malloc_hard

1. 从bin中获得可用的nonfull run, 这个过程中bin->lock有可能被解锁.
2. 暂不使用new run, 返回检查bin->runcur是否重新可用. 如果是, 则直接在其中分配region(其他线程在bin lock解锁期间可能提前修改了runcur). 否则, 执行4.
3. 重新检查1中得到的new run, 如果为空, 则释放该run.否则与当前runcur作比较, 若地址低于runcur, 则与其做交换. 将旧的runcur插回run tree. 并返回new rigion.
4. 用new run填充runcur, 并在其中分配region, 返回.

```
static void *
arena_bin_malloc_hard(arena_t *arena, arena_bin_t *bin)
{
    ......
    // xf: 获得bin对应的arena_bin_info, 并将current run置空
    binind = arena_bin_index(arena, bin);
    bin_info = &arena_bin_info[binind];
    bin->runcur = NULL;

    // xf: 从指定bin中获得一个可用的run
    run = arena_bin_nonfull_run_get(arena, bin);

    // 对bin->runcur做重新检查. 如果可用且未耗尽, 则直接分配.
    if (bin->runcur != NULL && bin->runcur->nfree > 0) {
```

```
        ret = arena_run_reg_alloc(bin->runcur, bin_info);

        // xf: 若new run为空, 则将其释放. 否则重新插入run tree.
        if (run != NULL) {
            arena_chunk_t *chunk;
            chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(run);
            if (run->nfree == bin_info->nregs)
                arena_dalloc_bin_run(arena, chunk, run, bin);
            else
                arena_bin_lower_run(arena, chunk, run, bin);
        }
        return (ret);
    }

    if (run == NULL)
        return (NULL);

    // xf: 到这里在bin->runcur中分配失败, 用当前新获得的run填充current run
    bin->runcur = run;

    // xf: 在new run中分配region
    return (arena_run_reg_alloc(bin->runcur, bin_info));
}
```

1. 尝试在当前run tree中寻找可用run, 成功则返回, 否则进入下一步.
2. 解锁bin lock, 并加锁arena lock, 尝试在当前arena中分配new run.之后重新解锁arena lock, 并加锁bin lock. 如果成功则返回, 否则进入下一步.
3. 分配失败, 重新在当前run tree中寻找一遍可用run.

```
static arena_run_t *
arena_bin_nonfull_run_get(arena_t *arena, arena_bin_t *bin)
{
    ......
    // xf: 尝试从当前run tree中寻找一个可用run, 如果存在就返回
    run = arena_bin_nonfull_run_tryget(bin);
    if (run != NULL)
        return (run);
    ......

    // xf: 打开bin lock, 让其他线程可以操作当前的bin tree
    malloc_mutex_unlock(&bin->lock);
    // xf: 锁住arena lock, 以分配new run
    malloc_mutex_lock(&arena->lock);

    // xf: 尝试分配new run
    run = arena_run_alloc_small(arena, bin_info->run_size, binind);
    if (run != NULL) {
        // 初始化new run和bitmap
        bitmap_t *bitmap = (bitmap_t *)((uintptr_t)run +
            (uintptr_t)bin_info->bitmap_offset);

        run->bin = bin;
        run->nextind = 0;
        run->nfree = bin_info->nregs;
        bitmap_init(bitmap, &bin_info->bitmap_info);
    }

    // xf: 解锁arena lock
    malloc_mutex_unlock(&arena->lock);
    // xf: 重新加锁bin lock
    malloc_mutex_lock(&bin->lock);

    if (run != NULL) {
        ......
```

```
        return (run);
    }

    // xf: 如果run alloc失败, 则回过头重新try get一次(前面解锁bin lock
    // 给了其他线程机会).
    run = arena_bin_nonfull_run_tryget(bin);
    if (run != NULL)
        return (run);

    return (NULL);
}
```

### 3.3.4 Small Run Alloc

1. 从arena avail tree上获得一个可用run, 并对其切割. 失败进入下一步.
2. 尝试给arena分配新的chunk, 以构造new run. 此过程可能会解锁arena lock. 失败进入下一步.
3. 其他线程可能在此过程中释放了某些run, 重新检查avail-tree, 尝试获取run.

```
static arena_run_t *
arena_run_alloc_small(arena_t *arena, size_t size, size_t binind)
{
    ......
    // xf: 从available tree上尝试寻找并切割一个合适的run, 并对其初始化
    run = arena_run_alloc_small_helper(arena, size, binind);
    if (run != NULL)
        return (run);

    // xf: 当前arena内没有可用的空闲run, 构造一个新的chunk以分配new run.
    chunk = arena_chunk_alloc(arena);
    if (chunk != NULL) {
        run = (arena_run_t *)((uintptr_t)chunk + (map_bias << LG_PAGE));
        arena_run_split_small(arena, run, size, binind);
        return (run);
    }

    // xf: 重新检查arena avail-tree.
    return (arena_run_alloc_small_helper(arena, size, binind));
}

static arena_run_t *
arena_run_alloc_small_helper(arena_t *arena, size_t size, size_t binind)
{
    ......
    // xf: 在arena的available tree中寻找一个大于等于size大小的最小run
    key = (arena_chunk_map_t *)(size | CHUNK_MAP_KEY);
    mapelm = arena_avail_tree_nsearch(&arena->runs_avail, key);
    if (mapelm != NULL) {
        arena_chunk_t *run_chunk = CHUNK_ADDR2BASE(mapelm);
        size_t pageind = arena_mapelm_to_pageind(mapelm);

        // xf: 计算候选run的地址
        run = (arena_run_t *)((uintptr_t)run_chunk + (pageind <<
            LG_PAGE));
        // xf: 根据分配需求, 切割候选run
        arena_run_split_small(arena, run, size, binind);
        return (run);
    }

    return (NULL);
}
```

切割small run主要分为4步,

1. 将候选run的arena_chunk_map_t节点从avail-tree上摘除.
2. 根据节点储存的原始page信息, 以及need pages信息, 切割该run.
3. 更新remainder节点信息(只需更新首尾page), 重新插入avail-tree.
4. 设置切割后new run所有page对应的map节点信息(根据2.3.3节所述).

```
static void
arena_run_split_small(arena_t *arena, arena_run_t *run, size_t size,
    size_t binind)
{
    ......
    // xf: 获取目标run的dirty flag
    chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(run);
    run_ind = (unsigned)(((uintptr_t)run - (uintptr_t)chunk) >> LG_PAGE);
    flag_dirty = arena_mapbits_dirty_get(chunk, run_ind);
    need_pages = (size >> LG_PAGE);

    // xf: 1. 将候选run从available tree上摘除
    //     2. 根据need pages对候选run进行切割
    //     3. 将remainder重新插入available tree
    arena_run_split_remove(arena, chunk, run_ind, flag_dirty, need_pages);

    // xf: 设置刚刚被split后的run的第一个page
    arena_mapbits_small_set(chunk, run_ind, 0, binind, flag_dirty);
    ......

    // xf: 依次设置run中的其他page, run index依次递增
    for (i = 1; i < need_pages - 1; i++) {
        arena_mapbits_small_set(chunk, run_ind+i, i, binind, 0);
        .......
    }

    // xf: 设置run中的最后一个page
    arena_mapbits_small_set(chunk, run_ind+need_pages-1, need_pages-1,
        binind, flag_dirty);
    ......
}
```

### 3.3.5 Chunk Alloc

arena获取chunk一般有两个途径. 其一是通过内部的spare指针. 该指针缓存了最近一次chunk被释放的记录. 因此该方式速度很快. 另一种更加常规, 通过内部分配函数分配, 最终将由chunk_alloc_core执行. 但在jemalloc的设计中, 执行arena chunk的分配器是可定制的, 你可以替换任何第三方chunk分配器. 这里仅讨论默认情况.

jemalloc在chunk_alloc_core中同传统分配器如Dl有较大区别. 通常情况下, 从系统获取内存无非是morecore或mmap两种方式. Dl中按照先morecore->mmap的顺序, 而jemalloc更为灵活, 具体的顺序由dss_prec_t决定.

该类型是一个枚举, 定义如下,

```
typedef enum {
    dss_prec_disabled  = 0,
    dss_prec_primary   = 1,
    dss_prec_secondary = 2,
    dss_prec_limit     = 3
} dss_prec_t;
```

这里dss和morecore含义是相同的. primary表示优先dss, secondary则优先mmap. jemalloc默认使用后者.

实际分配时, 无论采用哪种策略, 都会优先执行chunk_recycle, 再执行chunkalloc, 如下,

```
static void *
chunk_alloc_core(size_t size, size_t alignment, bool base, bool *zero,
    dss_prec_t dss_prec)
{
```

```
    void *ret;

    if (have_dss && dss_prec == dss_prec_primary) {
        if ((ret = chunk_recycle(&chunks_szad_dss, &chunks_ad_dss, size,
            alignment, base, zero)) != NULL)
            return (ret);
        if ((ret = chunk_alloc_dss(size, alignment, zero)) != NULL)
            return (ret);
    }

    if ((ret = chunk_recycle(&chunks_szad_mmap, &chunks_ad_mmap, size,
        alignment, base, zero)) != NULL)
        return (ret);
    if ((ret = chunk_alloc_mmap(size, alignment, zero)) != NULL)
        return (ret);

    if (have_dss && dss_prec == dss_prec_secondary) {
        if ((ret = chunk_recycle(&chunks_szad_dss, &chunks_ad_dss, size,
            alignment, base, zero)) != NULL)
            return (ret);
        if ((ret = chunk_alloc_dss(size, alignment, zero)) != NULL)
            return (ret);
    }

    return (NULL);
}
```

所谓chunk recycle是在alloc chunk之前, 优先在废弃的chunk tree上搜索可用chunk, 并分配base node以储存meta data的过程. 好处是其一可以加快分配速度, 其二是使空间分配更加紧凑, 并节省内存.

在jemalloc中存在4棵全局的rb tree, 分别为,

```
static extent_tree_t    chunks_szad_mmap;
static extent_tree_t    chunks_ad_mmap;
static extent_tree_t    chunks_szad_dss;
static extent_tree_t    chunks_ad_dss;
```

它们分别对应mmap和dss方式. 当一个chunk或huge region被释放后, 将收集到这4棵tree中. szad和ad在内容上并无本质区别, 只是检索方式不一样. 前者采用先size后address的方式, 后者则是纯address的检索.

recycle算法概括如下,

1. 检查base标志, 如果为真则直接返回, 否则进入下一步. 开始的检查是必要的, 因为recycle过程中可能会创建新的extent node, 要求调用base allocator分配. 另一方面, base alloc可能因为耗尽的原因而反过来调用chunk alloc. 如此将导致dead loop.
2. 根据alignment计算分配大小, 并在szad tree(mmap还是dss需要上一级决定)上寻找一个大于等于alloc size的最小node.
3. chunk tree上的node未必对齐到alignment上, 将地址对齐, 之后将得到leadsize和trailsize.
4. 将原node从chunk tree上remove. 若leadsize不为0, 则将其作为新的chunk重新insert回chunk tree. trailsize不为0的情况亦然. 若leadsize和trailsize同时不为0, 则通过base_node_alloc为trailsize生成新的node并插入. 若base alloc失败, 则整个新分配的region都要销毁.
5. 若leadsize和trailsize都为0, 则将node(注意仅仅是节点)释放. 返回对齐后的chunk地址.

```
static void *
chunk_recycle(extent_tree_t *chunks_szad, extent_tree_t *chunks_ad, size_t size,
    size_t alignment, bool base, bool *zero)
{
    ......
    // xf: 由于构造extent_node时可能因为内存不足的原因, 同样需要构造chunk,
    // 这样就导致recursively dead loop. 因此依靠base标志, 区分普通alloc和
    // base node alloc. 如果是base alloc, 则立即返回.
    if (base) {
        return (NULL);
```

```
        return (NULL);
    }

    // xf: 计算分配大小
    alloc_size = size + alignment - chunksize;
    ......
    key.addr = NULL;
    key.size = alloc_size;

    // xf: 在指定的szad tree上寻找大于等于alloc size的最小可用node
    malloc_mutex_lock(&chunks_mtx);
    node = extent_tree_szad_nsearch(chunks_szad, &key);
    ......

    // xf: 将候选节点基址对齐到分配边界上, 并计算leadsize, trailsize
    // 以及返回地址.
    leadsize = ALIGNMENT_CEILING((uintptr_t)node->addr, alignment) -
        (uintptr_t)node->addr;
    trailsize = node->size - leadsize - size;
    ret = (void *)((uintptr_t)node->addr + leadsize);
    ......

    // xf: 将原node从szad/ad tree上移除
    extent_tree_szad_remove(chunks_szad, node);
    extent_tree_ad_remove(chunks_ad, node);

    // xf: 如果存在leadsize, 则将前面多余部分作为一个chunk重新插入
    // szad/ad tree上.
    if (leadsize != 0) {
        node->size = leadsize;
        extent_tree_szad_insert(chunks_szad, node);
        extent_tree_ad_insert(chunks_ad, node);
        node = NULL;
    }

    // xf: 同样如果存在trailsize, 也将后面的多余部分插入.
    if (trailsize != 0) {
        // xf: 如果leadsize不为0, 这时原来的extent_node已经被用过了,
        // 则必须为trailsize部分重新分配新的extent_node
        if (node == NULL) {
            malloc_mutex_unlock(&chunks_mtx);
            node = base_node_alloc();
            ......
        }
        // xf: 计算trail chunk, 并插入
        node->addr = (void *)((uintptr_t)(ret) + size);
        node->size = trailsize;
        node->zeroed = zeroed;
        extent_tree_szad_insert(chunks_szad, node);
        extent_tree_ad_insert(chunks_ad, node);
        node = NULL;
    }
    malloc_mutex_unlock(&chunks_mtx);

    // xf: leadsize & basesize都不存在, 将node释放.
    if (node != NULL)
        base_node_dalloc(node);
    ......

    return (ret);
}
```

常规分配方式先来看dss. 由于dss是与当前进程的brk指针相关的, 任何线程(包括可能不通过jemalloc执行分配的线程)都有权修改该指针值. 因此, 首先要把dss指针调整到对齐在chunksize边界的位置, 否则很多与chunk相关的计算都会失效. 接下来, 还要做第二次调整对齐到外界请求的alignment边界. 在此基础上再进行分配.

与dss分配相关的变量如下,

```
static malloc_mutex_t    dss_mtx;
static void        *dss_base;
static void        *dss_prev;
static void        *dss_max;
```

- **dss_mtx:** dss lock. 注意其并不能起到保护dss指针的作用, 因为brk是一个系统资源.该lock保护的是dss_prev, dss_max指针.
- **dss_base:** 只在chunk_dss_boot时更新一次. 主要用作识别chunk在线性地址空间中所处的位置, 与mmap作出区别.
- **dss_prev:** 当前dss指针, 是系统brk指针的副本, 值等于-1代表dss耗尽.

### 3.4 Small allocation (tcache)

tcache内分配按照先easy后hard的方式. easy方式直接从tcache bin的avail-stack中获得可用region. 如果tbin耗尽, 使用hard方式, 先refill avail-stack, 再执行easy分配.

```
JEMALLOC_ALWAYS_INLINE void *
tcache_alloc_small(tcache_t *tcache, size_t size, bool zero)
{
    ......
    // xf: 先从tcache bin尝试分配
    ret = tcache_alloc_easy(tbin);
    // xf: 如果尝试失败, 则refill tcache bin, 并尝试分配
    if (ret == NULL) {
        ret = tcache_alloc_small_hard(tcache, tbin, binind);
        if (ret == NULL)
            return (NULL);
    }
    ......

    // xf: 执行tcache event
    tcache_event(tcache);
    return (ret);
}

JEMALLOC_ALWAYS_INLINE void *
tcache_alloc_easy(tcache_bin_t *tbin)
{
    void *ret;

    // xf: 如果tcache bin耗尽, 更新水线为-1
    if (tbin->ncached == 0) {
        tbin->low_water = -1;
        return (NULL);
    }
    // xf: pop栈顶的region, 如果需要更新水线
    tbin->ncached--;
    if ((int)tbin->ncached < tbin->low_water)
        tbin->low_water = tbin->ncached;
    ret = tbin->avail[tbin->ncached];
    return (ret);
}

void *
tcache_alloc_small_hard(tcache_t *tcache, tcache_bin_t *tbin, size_t binind)
{
    void *ret;

    arena_tcache_fill_small(tcache->arena, tbin, binind,
        config_prof ? tcache->prof_accumbytes : 0);
    if (config_prof)
        tcache->prof_accumbytes = 0;
    ret = tcache_alloc_easy(tbin);

    return (ret);
}
```

tcache fill同普通的arena bin分配类似. 首先, 获得与tbin相同index的arena bin.之后确定fill值, 该数值与2.7节介绍的lg_fill_div有关. 如果arena run的runcur可用则直接分配并push stack, 否则arena_bin_malloc_hard分配region. push后的顺序按照从低到高排列, 低地址的region更靠近栈顶位置.

```
void
arena_tcache_fill_small(arena_t *arena, tcache_bin_t *tbin, size_t binind,
    uint64_t prof_accumbytes)
{
    ......
    // xf: 得到与tbin同index的arena bin
    bin = &arena->bins[binind];
    malloc_mutex_lock(&bin->lock);
    // xf: tbin的充满度与lg_fill_div相关
    for (i = 0, nfill = (tcache_bin_info[binind].ncached_max >>
        tbin->lg_fill_div); i < nfill; i++) {
        // xf: 如果current run可用, 则从中分配
        if ((run = bin->runcur) != NULL && run->nfree > 0)
            ptr = arena_run_reg_alloc(run, &arena_bin_info[binind]);
        else    // xf: current run耗尽, 则从bin中查找其他run分配
            ptr = arena_bin_malloc_hard(arena, bin);
        if (ptr == NULL)
            break;
        ......
        // xf: 低地址region优先放入栈顶
        tbin->avail[nfill - 1 - i] = ptr;
    }
    ......
    malloc_mutex_unlock(&bin->lock);
    // xf: 更新ncached
    tbin->ncached = i;
}
```

另外, 如2.7节所述, tcache在每次分配和释放后都会更新ev_cnt计数器. 当计数周期达到TCACHE_GC_INCR时, 就会启动tcache gc. gc过程中会清理相当于low_water 3/4数量的region, 并根据当前的low_water和lg_fill_div动态调整下一次refill时, tbin的充满度.

```
void
tcache_bin_flush_small(tcache_bin_t *tbin, size_t binind, unsigned rem,
    tcache_t *tcache)
{
    ......
    // xf: 循环scan, 直到nflush为空.
    // 因为avail-stack中的region可能来自不同arena, 因此需要多次scan.
    // 每次scan将不同arena的region移动到栈顶, 留到下一轮scan时清理.
    for (nflush = tbin->ncached - rem; nflush > 0; nflush = ndeferred) {
        // xf: 获得栈顶region所属的arena和arena bin
        arena_chunk_t *chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(
            tbin->avail[0]);
        arena_t *arena = chunk->arena;
        arena_bin_t *bin = &arena->bins[binind];
        ......
        // xf: 锁住栈顶region的arena bin
        malloc_mutex_lock(&bin->lock);
        ......
        // xf: ndefered代表所属不同arena的region被搬移的位置, 默认从0开始.
        // 本意是随着scan进行, nflush逐渐递增, nflush之前的位置空缺出来.
        // 当scan到不同arena region时, 将其指针移动到nflush前面的空缺中,
        // 留到下一轮scan, nflush重新开始. 直到ndefered和nflush重新为0.
        ndeferred = 0;
        for (i = 0; i < nflush; i++) {
            ptr = tbin->avail[i];
            chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(ptr);
            // xf: 如果scan的region与栈顶region位于同一arena, 则释放,
            // 否则移动到ndefered标注的位置, 留到后面scan.
            if (chunk->arena == arena) {
                size_t pageind = ((uintptr_t)ptr -
```

```
                    (uintptr_t)chunk) >> LG_PAGE;
                arena_chunk_map_t *mapelm =
                    arena_mapp_get(chunk, pageind);
                ......
                // xf: 释放多余region
                arena_dalloc_bin_locked(arena, chunk, ptr,
                    mapelm);
            } else {
                tbin->avail[ndeferred] = ptr;
                ndeferred++;
            }
        }
        malloc_mutex_unlock(&bin->lock);
    }
    ......
    // xf: 将remainder regions指针移动到栈顶位置, 完成gc过程
    memmove(tbin->avail, &tbin->avail[tbin->ncached - rem],
        rem * sizeof(void *));
    // xf: 修正ncached以及low_water
    tbin->ncached = rem;
    if ((int)tbin->ncached < tbin->low_water)
        tbin->low_water = tbin->ncached;
}
```

### 3.5 Large allocation

Arena上的large alloc同small相比除了省去arena bin的部分之外, 并无本质区别. 基本算法如下,

1. 把请求大小对齐到page size上, 直接从avail-tree上寻找first-best-fit runs. 如果成功, 则根据请求大小切割内存. 切割过程也同切割small run类似, 区别在之后对chunk map的初始化不同. chunk map细节可回顾2.3.3. 如果失败, 则进入下一步.
2. 没有可用runs, 尝试创建new chunk, 成功同样切割run, 失败进入下一步.
3. 再次尝试从avail-tree上寻找可用runs, 并返回.

同上面的过程可以看出, 所谓large region分配相当于small run的分配. 区别仅在于chunk map信息不同.

Tcache上的large alloc同样按照先easy后hard的顺序. 尽管常规arena上的分配不存在large bin, 但在tcache中却存在large tbin, 因此仍然是先查找avail-stack.如果tbin中找不到, 就会向arena申请large runs. 这里与small alloc的区别在不执行tbin refill, 因为考虑到过多large region的占用量问题. large tbin仅在tcache_dalloc_large的时候才负责收集region. 当tcache已满或GC周期到时执行tcache gc.

### 3.6 Huge allocation

Huge alloc相对于前面就更加简单. 因为对于jemalloc而言, huge region和chunk是等同的,这在前面有过叙述. Huge alloc就是调用chunk alloc, 并将extent_node记录在huge tree上.

```
void *
huge_palloc(arena_t *arena, size_t size, size_t alignment, bool zero)
{
    void *ret;
    size_t csize;
    extent_node_t *node;
    bool is_zeroed;

    // xf: huge alloc对齐到chunksize
    csize = CHUNK_CEILING(size);
    ......
    // xf: create extent node以记录huge region
    node = base_node_alloc();
    ......
    arena = choose_arena(arena);
    // xf: 调用chunk alloc分配
    ret = arena_chunk_alloc_huge(arena, csize, alignment, &is_zeroed);
    // xf: 失败则清除extent node
    if (ret == NULL) {
        base_node_dalloc(node);
```

```
21          return (NULL);
22      }
23
24      node->addr = ret;
25      node->size = csize;
26      node->arena = arena;
27
28      // xf: 插入huge tree上
29      malloc_mutex_lock(&huge_mtx);
30      extent_tree_ad_insert(&huge, node);
31      malloc_mutex_unlock(&huge_mtx);
32      ......
33      return (ret);
34  }
```