--[Table of contents

--[1 - 简介

Jemalloc最初是Jason Evans为FreeBSD开发的新一代内存分配器, 用来替代原来的phkmalloc, 最早投入使用是在2005年. 到目前为止, 除了原版Je, 还有很多变种

被用在各种项目里. Google在android5.0里将bionic中的默认分配器从Dl替换为Je,
也是看中了其强大的多核多线程分配能力.

同经典分配器, 如Dlmalloc相比, Je在基本思路和实现上存在明显的差别. 比如,
Dl在分配策略上倾向于先dss后mmap的方式, 为的是快速向前分配, 但Je则完全相反.
而实现上也放弃了经典的boundary tag. 这些设计牺牲了局部分配速度和回收效率,
但在更大的空间和时间范围内却获得更好的分配效果.

更重要的是, 相对经典分配器, Je最大的优势还是其强大的多核/多线程分配能力.
以现代计算机硬件架构来说, 最大的瓶颈已经不再是内存容量或cpu速度, 而是
多核/多线程下的lock contention. 因为无论核心数量如何多, 通常情况下内存
只有一份. 可以说, 如果内存足够大, CPU的核心数量越多, 程序线程数越多,
Je的分配速度越快. 而这一点是经典分配器所无法达到的.

这篇文章基于android5.x中的Jemalloc3.6.0. 最新的版本为4.x, 获取最新代码请至,

https://github.com/jemalloc/jemalloc/releases

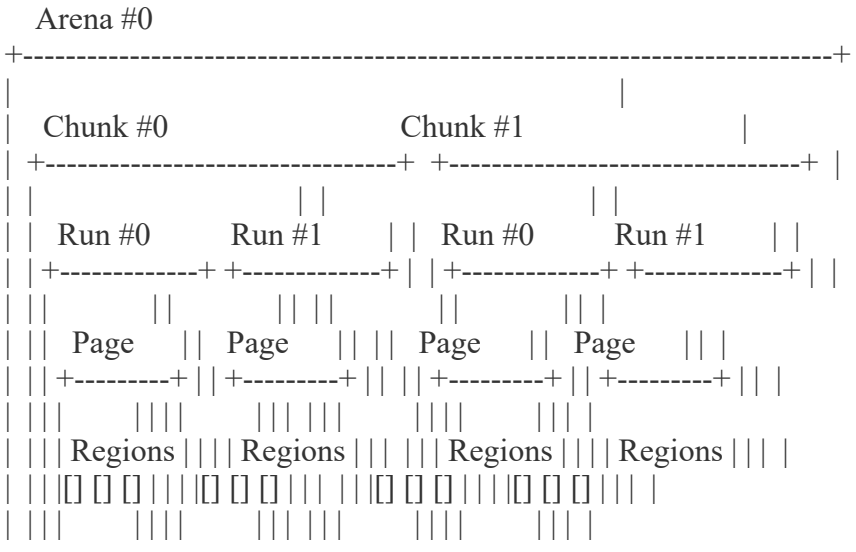
--[2 - Basic structures

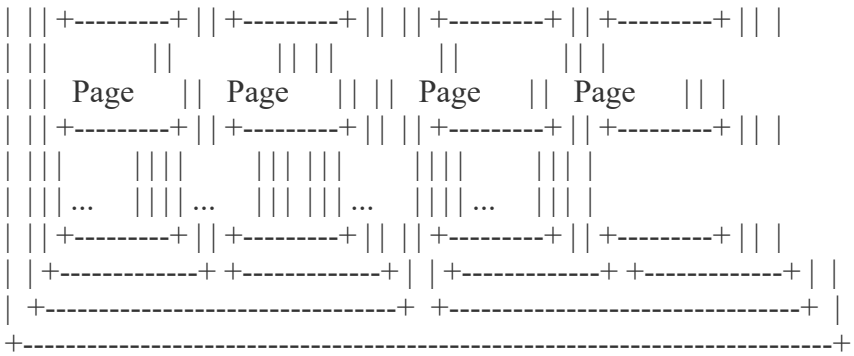
相对于Dl, Je引入了更多更复杂的分配结构, 如arena, chunk, bin, run, region,
tcache等等. 其中有些类似Dl, 但更多的具有不同含义, 本节将对它们做一一介绍.

----[2.1 - Overview

首先, 先给出一个整体的概念. Je对内存划分按照如下由高到低的顺序,

1. 内存是由一定数量的arenas进行管理.
2. 一个arena被分割成若干chunks, 后者主要负责记录bookkeeping.
3. chunk内部又包含着若干runs, 作为分配小块内存的基本单元.
4. run由pages组成, 最终被划分成一定数量的regions,
5. 对于small size的分配请求来说, 这些region就相当于user memory.





----[2.2 - Arena (arena_t)

如前所述, Arena是Je中最大或者说最顶层的基础结构. 这个概念其实上是针对"对称
多处理机(SMP)"产生的. 在SMP中, 导致性能劣化的一个重要原因在于"false sharing"
导致cache-line失效.

为了解决cache-line共享问题, 同时保证更少的内部碎片(internal fragmentation),
Je使用了arena.

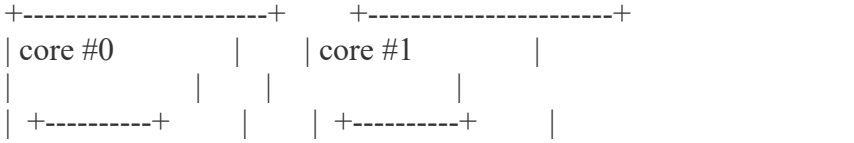
------[2.2.1 - CPU Cache-Line

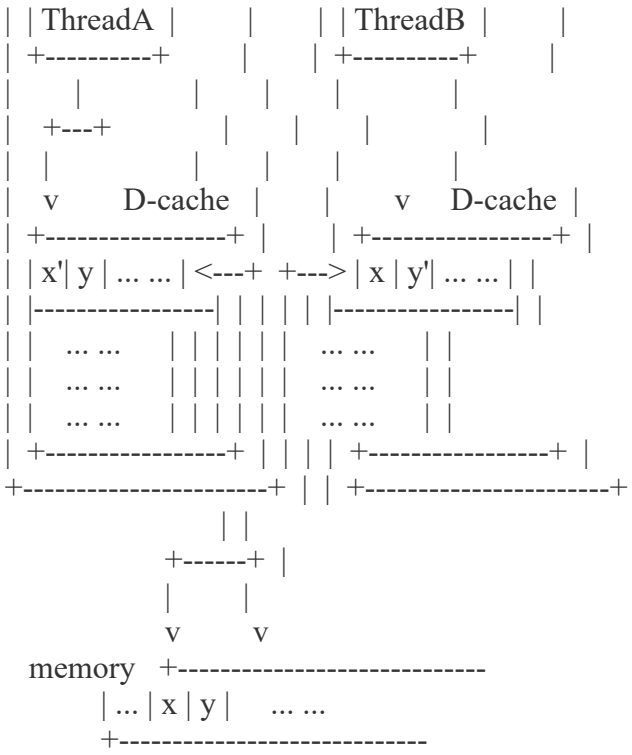
现代处理器为了解决内存总线吞吐量的瓶颈使用了内部cache技术. 尽管cache的
工作机制很复杂, 且对外透明, 但在编程上, 还是有必要了解cache的基本性质.

Cache相当于嵌入到cpu内部的一组内存单元, 速度是主存的N倍, 但造价很高, 因此
一般容量很小. 有些cpu设计了容量逐级逐渐增大的多级cache, 但速度逐级递减.
多级处理更复杂, 但原理类似, 为了简化, 仅讨论L1 data cache.

cache同主存进行数据交换有一个最小粒度, 称为cache-line, 通常这个值为64. 例如
在一个ILP32的机器上, 一次cache交换可以读写连续16个int型数据. 因此当访问数组
#0元素时, 后面15个元素也被同时"免费"读到了cache中, 这对于数组的连续访问是
非常有利的.

然而这种免费加载不总是会带来好处, 有时候甚至起到反效果, 所谓"false sharing".
试想两个线程A和B分别执行在不同的cpu核心中并分别操作各自上下文中的变量x和y.
如果因为某种原因(比如x, y可能位于同一个class内部, 或者分别是数组中的两个相邻
元素), 两者位于相同的cache-line中, 则在两个core的L1 cache里都存在x和y的副本.
倘若线程A修改了x的值, 就会导致在B中的x与A中看到的不一致. 尽管这个变量x对B可能
毫无用处, 但cpu为了保证前后的正确和一致性, 只能判定core #1的cache失效. 因此
core #0必须将cache-line回写到主存, 然后core #1再重新加载cache-line, 反之亦然.
如果恰好两个线程交替操作同一cache-line中的数据, 将对cache将造成极大的损害,
导致严重的性能退化.



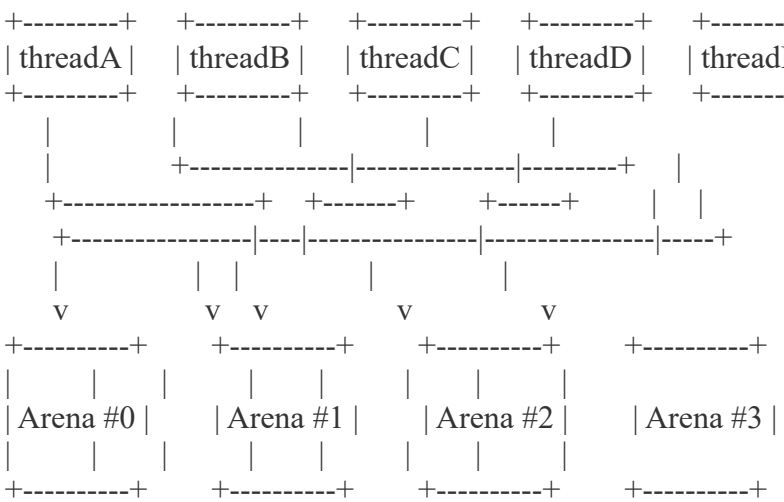


说到底, 从程序的角度看, 变量是独立的地址单元, 但在CPU看来则是以cache-line为整体的单元. 单独的变量竞争可以在代码中增加同步来解决, 而cache-line的竞争是透明的, 不可控的, 只能被动由CPU仲裁. 这种观察角度和处理方式的区别, 正是false sharing的根源.

------[2.2.2 - Arena原理

回到memory allocator的话题上. 对于一个多线程+多CPU核心的运行环境, 传统分配器中大量开销被浪费在lock contention和false sharing上, 随着线程数量和核心数量增多, 这种分配压力将越来越大.

针对多线程, 一种解决方法是将一把global lock分散成很多与线程相关的lock. 而针对多核心, 则要尽量把不同线程下分配的内存隔离开, 避免不同线程使用同一个cache-line的情况. 按照上面的思路, 一个较好的实现方式就是引入arena.



Je将内存划分成若干数量的arenas, 线程最终会与某一个arena绑定. 比如上图中的threadA和B就分别绑定到arena #1和#3上. 由于两个arena在地址空间上几乎不存在任何联系, 就可以在无锁的状态下完成分配. 同样由于空间不连续, 落到同一个cache-line中的几率也很小, 保证了各自独立.

由于arena的数量有限, 因此不能保证所有线程都能独占arena, 比如, 图中threadA和C就都绑定到了arena1上. 分享同一个arena的所有线程, 由该arena内部的lock保持同步.

Je将arena保存到一个数组中, 该数组全局记录了所有arenas,

```
arena_t            **arenas;
```

事实上, 该数组是动态分配的, arenas仅仅是一个数组指针. 默认情况下arenas数组的长度与如下变量相关,

```
unsigned    narenas_total;
unsigned    narenas_auto;
size_t        opt_narenas = 0;
```

而它们又与当前cpu核心数量相关. 核心数量记录在另外一个全局变量ncpus里,

```
unsigned    ncpus;
```

如果ncpus等于1, 则有且仅有一个arena, 如果大于1, 则默认arenas的数量为ncpus的四倍. 即双核下8个arena, 四核下16个arena, 依此类推.

```
(gdb) p ncpus
$20 = 4
(gdb) p narenas_total
$21 = 16
```

jemalloc变体很多, 不同变体对arenas的数量有所调整, 比如firefox中arena固定为1, 而android被限定为最大不超过2. 这个限制被写到android jemalloc的mk文件中.

------[2.2.3 - choose_arena

最早引入arena并非由Je首创, 但早期线程与arena绑定是通过hash线程id实现的, 相对来说随机性比较强. Je改进了绑定的算法, 使之更加科学合理.

Je中线程与arena绑定由函数choose_arena完成, 被绑定的arena记录在该线程的tls中,

```
JEMALLOC_INLINE arena_t *
choose_arena(arena_t *arena)
{
    ......
    // xf: 通常情况下线程所绑定的arena记录在arenas_tls中
    if ((ret = *arenas_tsd_get()) == NULL) {
        // xf: 如果当前thread未绑定arena, 则为其指定一个, 并保存到tls
        ret = choose_arena_hard();
```

```
    }

    return (ret);
}
```

初次搜索arenas_tsd_get可能找不到该函数在何处被定义. 实际上, Je使用了一组宏,
来生成一个函数族, 以达到类似函数模板的目的. tsd相关的函数族被定义在tsd.h中.

1. malloc_tsd_protos - 定义了函数声明, 包括初始化函数boot, get/set函数
2. malloc_tsd_externs - 定义变量声明, 包括tls, 初始化标志等等
3. malloc_tsd_data - tls变量定义
4. malloc_tsd_funcs - 定义了1中声明函数的实现.

与arena tsd相关的函数和变量声明如下,

```
malloc_tsd_protos(JEMALLOC_ATTR(unused), arenas, arena_t *)
malloc_tsd_externs(arenas, arena_t *)
malloc_tsd_data(, arenas, arena_t *, NULL)
malloc_tsd_funcs(JEMALLOC_ALWAYS_INLINE, arenas, arena_t *, NULL, arenas_cleanup)
```

当线程还未与任何arena绑定时, 会进一步通过choose_arena_hard寻找一个合适的arena
进行绑定. Je会遍历arenas数组, 并按照优先级由高到低的顺序挑选,

1. 如果找到当前线程绑定数为0的arena, 则优先使用它.
2. 如果当前已初始化arena中没有线程绑定数为0的, 则优先使用剩余空的数组位置
   构造一个新的arena. 需要说明的是, arenas数组遵循lazy create原则, 初始状态
   整个数组只有0号元素是被初始化的, 其他的slot位置都是null指针. 因此随着新的
   线程不断创造出来, arena数组也被逐渐填满.
3. 如果1,2两条都不满足, 则选择当前绑定数最小的, 且slot位置更靠前的一个arena.

```
arena_t * choose_arena_hard(void)
{
    ......
    if (narenas_auto > 1) {
        ......
        first_null = narenas_auto;
        // xf: 循环遍历所有arenas, 找到绑定thread数量最小的arena, 并记录
        // first_null索引值
        for (i = 1; i < narenas_auto; i++) {
            if (arenas[i] != NULL) {
                if (arenas[i]->nthreads <
                    arenas[choose]->nthreads)
                    choose = i;
            } else if (first_null == narenas_auto) {
                first_null = i;
            }
        }

        // xf: 若选定的arena绑定thread为0, 或者当前arena数组中已满, 则返回
        // 被选中的arena
        if (arenas[choose]->nthreads == 0
```

```
            || first_null == narenas_auto) {
            ret = arenas[choose];
        } else {
            // xf: 否则构造并初始化一个新的arena
            ret = arenas_extend(first_null);
        }
        ......
    } else {
        // xf: 若不存在多于一个arena(单核cpu或人为强制设定), 则返回唯一的
        // 0号arena
        ret = arenas[0];
        ......
    }

    // xf: 将已绑定的arena设置到tsd中
    arenas_tsd_set(&ret);

    return (ret);
}
```

对比早期的绑定方式, Je的算法显然更加公平, 尽可能的让各个cpu核心平分当前线程, 平衡负载.

------[2.2.4 - Arena结构

```
struct arena_s {
    unsigned        ind;
    unsigned        nthreads;
    malloc_mutex_t        lock;
    arena_stats_t        stats;
    ql_head(tcache_t)    tcache_ql;
    uint64_t        prof_accumbytes;
    dss_prec_t        dss_prec;
    arena_chunk_tree_t    chunks_dirty;
    arena_chunk_t        *spare;
    size_t            nactive;
    size_t            ndirty;
    size_t            npurgatory;
    arena_avail_tree_t    runs_avail;
    chunk_alloc_t        *chunk_alloc;
    chunk_dalloc_t        *chunk_dalloc;
    arena_bin_t        bins[NBINS];
};
```

ind: 在arenas数组中的索引值.

nthreads: 当前绑定的线程数.

lock: 局部arena lock, 取代传统分配器的global lock.
    一般地, 如下操作需要arena lock同步,
    1. 线程绑定, 需要修改nthreads

2. new chunk alloc
3. new run alloc

stats: 全局统计, 需要打开统计功能.

tcache_ql: ring queue, 注册所有绑定线程的tcache, 作为统计用途, 需要打开统计功能.

dss_prec: 代表当前chunk alloc时对系统内存的使用策略, 分为几种情况,

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

第一个代表禁止使用系统DSS, 后两种代表是否优先选用DSS. 如果使用primary, 则本着先dss->mmap的顺序, 否则按照先mmap->dss. 默认使用dss_prec_secondary.

chunks_dirty: rb tree, 代表所有包含dirty page的chunk集合. 后面在chunk中会详细介绍.

spare: 是一个缓存变量, 记录最近一次被释放的chunk. 当arena收到一个新的chunk alloc请求时, 会优先从spare中开始查找, 由此提高频繁分配释放时, 可能导致内部chunk利用率下降的情况.

runs_avail: rb tree, 记录所有未被分配的runs, 用来在分配new run时寻找合适的available run. 一般作为alloc run时的仓库.

chunk_alloc/chunk_dalloc: 用户可定制的chunk分配/释放函数, Je提供了默认的版本, chunk_alloc_default/chunk_dalloc_default

bins: bins数组, 记录不同class size可用free regions的分配信息, 后面会详细介绍.

----[2.3 - Chunk (arena_chunk_t)

chunk是仅次于arena的次级内存结构. 如果有了解过Dlmalloc, 就会知道在Dl中同样定义了名为'chunk'的基础结构. 但这个概念在两个分配器中含义完全不同, Dl中的chunk指代最低级分配单元, 而Je中则是一个较大的内存区域.

------[2.3.1 - overview

从前面arena的数据结构可以发现, 它是一个非常抽象的概念, 其大小也不代表实际的内存分配量. 原始的内存数据既非挂载在arena外部, 也并没有通过内部指针引用, 而是记录在chunk中. 按照一般的思路, chunk包含原始内存数据, 又从属于arena, 因此后者应该会有一个数组之类的结构以记录所有chunk信息. 但事实上同样找不到这样的记录. 那Je又如何获得chunk指针呢?

所谓的chunk结构, 只是整个chunk的一个header, bookkeeping以及user memory都挂在

header外面. 另外Je对chunk又做了规定, 默认每个chunk大小为4MB, 同时还必须对齐到4MB的边界上.

```
#define    LG_CHUNK_DEFAULT    22
```

这个宏定义了chunk的大小. 注意到前缀'LG_', 代表log即指数部分.
Je中所有该前缀的代码都是这个含义, 便于通过bit操作进行快速的运算.

有了上述规定, 获得chunk就变得几乎没有代价. 因为返回给user程序的内存地址肯定属于某个chunk, 而该chunk header对齐到4M边界上, 且不可能超过4M大小, 所以只需要对该地址做一个下对齐就得到chunk指针, 如下,

```
#define    CHUNK_ADDR2BASE(a)                    \
   ((void *)((uintptr_t)(a) & ~chunksize_mask))
```

计算相对于chunk header的偏移量,

```
#define    CHUNK_ADDR2OFFSET(a)                  \
   ((size_t)((uintptr_t)(a) & chunksize_mask))
```

以及上对齐到chunk边界的计算,

```
#define    CHUNK_CEILING(s)                      \
   (((s) + chunksize_mask) & ~chunksize_mask)
```

用图来表示如下,

```
  chunk_ptr(4M aligned)              memory for user
   |                                 |
   v                                 v
   +--------------+-------------------------------------------
   | chunk header |       ... ...     | region |   ... ...
   +--------------+-------------------------------------------
   |<------------- offset ------------>|
```

------[2.3.2 - Chunk结构

```
struct arena_chunk_s {
   arena_t           *arena;
   rb_node(arena_chunk_t)    dirty_link;
   size_t            ndirty;
   size_t            nruns_avail;
   size_t            nruns_adjac;
   arena_chunk_map_t    map[1];
}
```

arena: chunk属于哪个arena

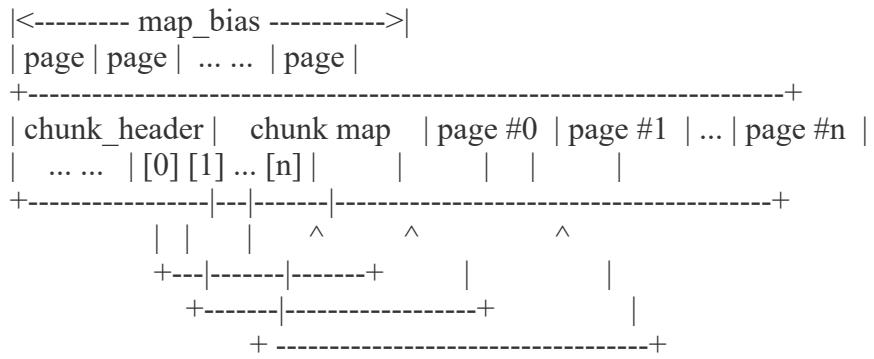
dirty_link: 用于rb tree的链接节点. 如果某个chunk内部含有任何dirty page,
就会被挂载到arena中的chunks_dirty tree上.

ndirty: 内部dirty page数量.

nruns_avail: 内部available runs数量.

nruns_adjac: available runs又分为dirty和clean两种, 相邻的两种run是无法合并的,
除非其中的dirty runs通过purge才可以. 该数值记录的就是可以通过
purge合并的run数量.

map: 动态数组, 每一项对应chunk中的一个page状态(不包含header即map本身的占用).
chunk(包括内部的run)都是由page组成的. page又分为unallocated, small,
large三种.
unallocated指的那些还未建立run的page.
small/large分别指代该page所属run的类型是small/large run.
这些page的分配状态, 属性, 偏移量, 及其他的标记信息等等, 都记录在
arena_chunk_map_t中.



至于由chunk header和chunk map占用的page数量, 保存在map_bias变量中.
该变量是Je在arena boot时通过迭代算法预先计算好的, 所有chunk都是相同的.
迭代方法如下,

1. 第一次迭代初始map_bias等于0, 计算最大可能大小, 即
   header_size + chunk_npages * map_size
   获得header+map需要的page数量, 结果肯定高于最终的值.
2. 第二次将之前计算的map_bias迭代回去, 将最大page数减去map_bias数, 重新计算
   header+map大小, 由于第一次迭代map_bias过大, 第二次迭代必定小于最终结果.
3. 第三次再将map_bias迭代回去, 得到最终大于第二次且小于第一次的计算结果.

相关代码如下,

```
void
arena_boot(void)
{
    ......
    map_bias = 0;
    for (i = 0; i < 3; i++) {
        header_size = offsetof(arena_chunk_t, map) +
            (sizeof(arena_chunk_map_t) * (chunk_npages-map_bias));
        map_bias = (header_size >> LG_PAGE) + ((header_size & PAGE_MASK)
            != 0);
    }
    ......
}
```

------[2.3.3 - chunk map (arena_chunk_map_t)

chunk记录page状态的结构为arena_chunk_map_t, 为了节省空间, 使用了bit压缩存储信息.

```
struct arena_chunk_map_s {
#ifndef JEMALLOC_PROF
   union {
#endif
   union {
      rb_node(arena_chunk_map_t)    rb_link;
      ql_elm(arena_chunk_map_t)    ql_link;
   }              u;
   prof_ctx_t           *prof_ctx;
#ifndef JEMALLOC_PROF
   };
#endif
   size_t               bits;
}
```

chunk map内部包含两个link node, 分别可以挂载到rb tree或环形队列上, 同时
为了节省空间又使用了union. 由于run本身也是由连续page组成的, 因此chunk map
除了记录page状态之外, 还负责run的基址检索.

举例来说, Je会把所有已分配run记录在内部rb tree上以快速检索, 实际地操作是
将该run中第一个page对应的chunk_map作为rb node挂载到tree上. 检索时也是先
找出将相应的chunk map, 再进行地址转换得到run的基址.

按照通常的设计思路, 我们可能会把run指针作为节点直接保存到rb tree中. 但Je中
的设计明显要更复杂. 究其原因, 如果把link node放到run中, 后果是bookkeeping和
user memory将混淆在一起, 这对于分配器的安全性是很不利的. 包括Dl在内的传统
分配器都具有这样的缺陷. 而如果单独用link node记录run, 又会造成空间浪费.
正因为Je中无论是chunk还是run都是连续page组成, 所以用首个page对应的chunk map
就能同时代表该run的基址.

Je中通常用mapelm换算出pageind, 再将pageind << LG_PAGE + chunk_base, 就能得到
run指针, 代码如下,

```
arena_chunk_t *run_chunk = CHUNK_ADDR2BASE(mapelm);
size_t pageind = arena_mapelm_to_pageind(mapelm);
run = (arena_run_t *)((uintptr_t)run_chunk + (pageind <<
   LG_PAGE));

JEMALLOC_INLINE_C size_t
arena_mapelm_to_pageind(arena_chunk_map_t *mapelm)
{
   uintptr_t map_offset =
      CHUNK_ADDR2OFFSET(mapelm) - offsetof(arena_chunk_t, map);

   return ((map_offset / sizeof(arena_chunk_map_t)) + map_bias);
}
```

chunk map对page状态描述都压缩记录到bits中, 由于内容较多, 直接引用Je代码中的注释,

下面是一个假想的ILP32系统下的bits layout,

```
???????? ???????? ????nnnn nnnndula
```

"?"的部分分三种情况, 分别对应unallocated, small和large.
? : Unallocated: 首尾page写入该run的地址, 而内部page则不做要求.
    Small: 全部是page的偏移量.
    Large: 首page是run size, 后续的page不做要求.
n : 对于small run指其所在bin的index, 对large run写入BININD_INVALID.
d : dirty?
u : unzeroed?
l : large?
a : allocated?

下面是对三种类型的run page做的举例,

p : run page offset
s : run size
n : binind for size class; large objects set these to BININD_INVALID
x : don't care
- : 0
+ : 1
[DULA] : bit set
[dula] : bit unset

Unallocated (clean):
```
  ssssssss ssssssss ssss++++ ++++du-a
  xxxxxxxx xxxxxxxx xxxxxxxx xxxx-Uxx
  ssssssss ssssssss ssss++++ ++++dU-a
```

Unallocated (dirty):
```
  ssssssss ssssssss ssss++++ ++++D--a
  xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
  ssssssss ssssssss ssss++++ ++++D--a
```

Small:
```
  pppppppp pppppppp ppppnnnn nnnnd--A
  pppppppp pppppppp ppppnnnn nnnn---A
  pppppppp pppppppp ppppnnnn nnnnd--A
```

  Small page需要注意的是, 这里代表的p并非是一个固定值, 而是该page相对于其所在run的第一个page的偏移量, 比如可能是这样,
```
  00000000 00000000 0000nnnn nnnnd--A
  00000000 00000000 0001nnnn nnnn---A
  00000000 00000000 0010nnnn nnnn---A
  00000000 00000000 0011nnnn nnnn---A
  ...
  00000000 00000001 1010nnnn nnnnd--A
```

```
  Large:
    ssssssss ssssssss ssss++++ ++++D-LA
    xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
    -------- -------- ----++++ ++++D-LA

  Large (sampled, size <= PAGE):
    ssssssss ssssssss ssssnnnn nnnnD-LA

  Large (not sampled, size == PAGE):
    ssssssss ssssssss ssss++++ ++++D-LA
```

为了提取/设置map bits内部的信息, Je提供了一组函数, 这里列举两个最基本的,
剩下的都是读取mapbits后做一些位运算而已,

读取mapbits,

```
JEMALLOC_ALWAYS_INLINE size_t
arena_mapbits_get(arena_chunk_t *chunk, size_t pageind)
{
    return (arena_mapbitsp_read(arena_mapbitsp_get(chunk, pageind)));
}
```

根据pageind获取对应的chunk map,

```
JEMALLOC_ALWAYS_INLINE arena_chunk_map_t *
arena_mapp_get(arena_chunk_t *chunk, size_t pageind)
{
    ......
    return (&chunk->map[pageind-map_bias]);
}
```

----[2.4 - Run (arena_run_t)

如同在2.1节所述, 在Je中run才是真正负责分配的主体(前提是对small region来说).
run的大小对齐到page size上, 并且在内部划分成大小相同的region. 当有外部分配
请求时, run就会从内部挑选一个free region返回. 可以认为run就是small region仓库.

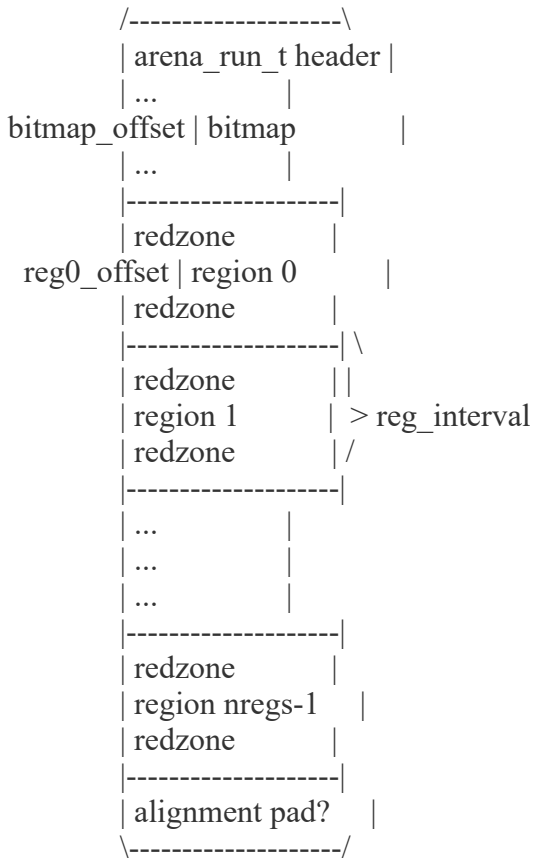
------[2.4.1 - Run结构

```
struct arena_run_s {
    arena_bin_t    *bin;
    uint32_t    nextind;
    unsigned    nfree;
};
```

run的结构非常简单, 但同chunk类似, 所谓的arena_run_t不过是整个run的header部分.

bin: 与该run相关联的bin. 每个run都有其所属的bin, 详细内容在之后介绍.
nextind: 记录下一个clean region的索引.
nfree: 记录当前空闲region数量.

除了header部分之外, run的真实layout如下,



正如chunk通过chunk map记录内部所有page状态一样, run通过在header后挂载bitmap来记录其内部的region状态. bitmap之后是regions区域. 内部region大小相等, 且在前后都有redzone保护(需要在设置里打开redzone选项).

简单来说, run就是通过查询bitmap来找到可用的region. 而传统分配器由于使用boundary tag, 空闲region一般是被双向链表管理的. 相比之下, 传统方式查找速度更快, 也更简单. 缺点之前也提到过, 安全和稳定性都存在缺陷. 从这一点可以看到, Je在设计思路上将bookkeeping和user memory分离是贯穿始终的原则, 更甚于对性能的影响(事实上这点影响在并发条件下被大大赚回来了).

------[2.4.2 - size classes

内存分配器对内部管理的region往往按照某种特殊规律来分配. 比如DI将内存划分成small和large两种类型. small类型从8字节开始每8个字节为一个分割直至256字节. 而large类型则从256字节开始, 挂载到dst上. 这种划分方式有助于分配器对内存进行有效的管理和控制, 让已分配的内存更加紧实(tightly packed), 以降低外部碎片率.

Je进一步优化了分配效率. 采用了类似于"二分伙伴(Binary Buddy)算法"的分配方式. 在Je中将不同大小的类型称为size class.

在Je源码的size_classes.h文件中, 定义了不同体系架构下的region size. 该文件实际是通过名为size_classes.sh的shell script自动生成的. script按照四种不同量纲定义来区分各个体系平台的区别, 然后将它们做排列组合, 就可以兼容各个体系. 这四种量纲分别是,

LG_SIZEOF_PTR: 代表指针长度, ILP32下是2, LP64则是3.

LG_QUANTUM: 量子, binary buddy分得的最小单位. 除了tiny size, 其他的size classes都是quantum的整数倍大小.

LG_TINY_MIN: 是比quantum更小的size class, 且必须对齐到2的指数倍上. 它是Je可分配的最小的size class.

LG_PAGE: 就是page大小

根据binary buddy算法, Je会将内存不断的二平分, 每一份称作一个group. 同一个group内又做四等分. 例如, 一个典型的ILP32, tiny等于8byte, quantum为16byte, page为4096byte的系统, 其size classes划分是这样的,

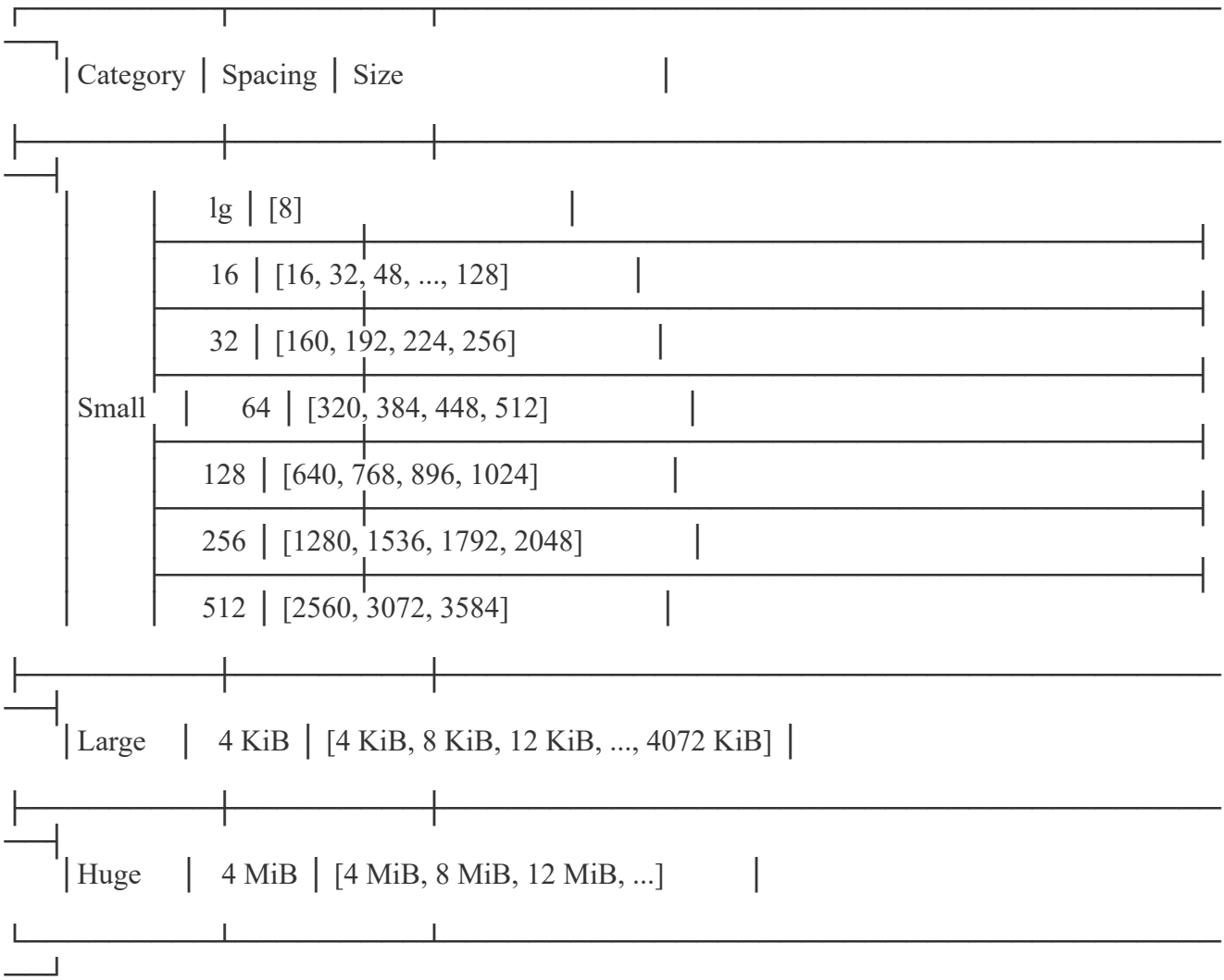
```
#if (LG_SIZEOF_PTR == 2 && LG_TINY_MIN == 3 && LG_QUANTUM == 4 && LG_PAGE == 12)
#define    SIZE_CLASSES \
     index, lg_grp, lg_delta, ndelta,  bin, lg_delta_lookup  \
   SC(  0,      3,       3,      0,   yes,       3) \
                                              \
   SC(  1,      3,       3,      1,   yes,       3) \
   SC(  2,      4,       4,      1,   yes,       4) \
   SC(  3,      4,       4,      2,   yes,       4) \
   SC(  4,      4,       4,      3,   yes,       4) \
                                              \
   SC(  5,      6,       4,      1,   yes,       4) \
   SC(  6,      6,       4,      2,   yes,       4) \
   SC(  7,      6,       4,      3,   yes,       4) \
   SC(  8,      6,       4,      4,   yes,       4) \
                                              \
   SC(  9,      7,       5,      1,   yes,       5) \
   SC( 10,      7,       5,      2,   yes,       5) \
   SC( 11,      7,       5,      3,   yes,       5) \
   SC( 12,      7,       5,      4,   yes,       5) \

   ... ...
```

宏SIZE_CLASSES主要功能就是可以生成几种类型的table. 而SC则根据不同的情况被定义成不同的含义. SC传入的6个参数的含义如下,

index: 在table中的位置
lg_grp: 所在group的指数
lg_delta: group内偏移量指数
ndelta: group内偏移数
bin: 是否由bin记录. small region是记录在bins中
lg_delta_lookup: 在lookup table中的调用S2B_#的尾数后缀

因此得到reg_size的计算公式, reg_size = 1 << lg_grp + ndelta << lg_delta
根据该公式, 可以得到region的范围,



除此之外, 在size_classes.h中还定义了一些常量,

tiny bins的数量
#define NTBINS 1

可以通过lookup table查询的bins数量
#define NLBINS 29

small bins的数量
#define NBINS 28

最大tiny size class的指数
#define LG_TINY_MAXCLASS 3

最大lookup size class, 也就是NLBINS - 1个bins
#define LOOKUP_MAXCLASS ((((size_t)1) << 11) + (((size_t)4) << 9))

最大small size class, 也就是NBINS - 1个bins
#define SMALL_MAXCLASS ((((size_t)1) << 11) + (((size_t)3) << 9))

------[2.4.3 - size2bin/bin2size

通过SIZE_CLASSES建立的table就是为了在O(1)的时间复杂度内快速进行size2bin或者bin2size切换. 同样的技术在Dl中有所体现, 来看Je是如何实现的.

size2bin切换提供了两种方式, 较快的是通过查询lookup table, 较慢的是计算得到.
从原理上, 只有small size class需要查找bins, 但可通过lookup查询的size class数量要小于整个small size class数量. 因此, 部分size class只能计算得到.
在原始Je中统一只采用查表法, 但在android版本中可能是考虑减小lookup table size, 而增加了直接计算法.

```
JEMALLOC_ALWAYS_INLINE size_t
small_size2bin(size_t size)
{
    ......
    if (size <= LOOKUP_MAXCLASS)
        return (small_size2bin_lookup(size));
    else
        return (small_size2bin_compute(size));
}
```

小于LOOKUP_MAXCLASS的请求通过small_size2bin_lookup直接查表.
查询的算法是这样的,

```
size_t ret = ((size_t)(small_size2bin_tab[(size-1) >> LG_TINY_MIN]));
```

也就是说, Je通过一个
    f(x) = (x - 1) / 2^LG_TINY_MIN
的变换将size映射到lookup table的相应区域. 这个table在gdb中可能是这样的,

```
(gdb) p  /d small_size2bin
$6 = {0, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 9, 9, 10, 10, 10, 10,
      11, 11, 11, 11, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14,
      14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15, 15, 16, 16, 16, 16, 16, 16,
      16, 16, 17 <repeats 16 times>, 18 <repeats 16 times>, 19 <repeats 16 times>,
      20 <repeats 16 times>, 21 <repeats 32 times>, 22 <repeats 32 times>,
      23 <repeats 32 times>, 24 <repeats 32 times>, 25 <repeats 64 times>,
      26 <repeats 64 times>, 27 <repeats 64 times>}
```

该数组的含义与binary buddy算法是一致的. 对应的bin index越高, 其在数组中占用的字节数就越多. 除了0号bin之外, 相邻的4个bin属于同一group, 两个group之间相差二倍, 因此在数组中占用的字节数也就相差2倍. 所以, 上面数组的group划分如下,

{0}, {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}, ...

以bin#9为例, 其所管辖的范围(128, 160], 由于其位于更高一级group, 因此相比bin#8在lookup table中多一倍的字节数, 假设我们需要查询132, 经过映射,

    (132 - 1) >> 3 = 16

这样可以快速得到其所在的bin #9. 如图,

```
      bin #1     bin #3          132 is HERE!
        |         |              |
        v         v              v
   +-------------------------------------------------------------
   | 0 | 1 | 2 2 | 3 3 | ... | 8 8 | 9 9 9 9 | ... | 16 ... 16 | ...
   +-------------------------------------------------------------
     ^       ^              ^      ^              ^
     |       |              |      |              |
  bin #0    bin #2          bin #8  bin #9         bin #16
```

Je巧妙的通过前面介绍CLASS_SIZE宏生成了这个lookup table, 代码如下,

```
JEMALLOC_ALIGNED(CACHELINE)
const uint8_t    small_size2bin_tab[] = {
#define    S2B_3(i)    i,
#define    S2B_4(i)    S2B_3(i) S2B_3(i)
#define    S2B_5(i)    S2B_4(i) S2B_4(i)
#define    S2B_6(i)    S2B_5(i) S2B_5(i)
#define    S2B_7(i)    S2B_6(i) S2B_6(i)
#define    S2B_8(i)    S2B_7(i) S2B_7(i)
#define    S2B_9(i)    S2B_8(i) S2B_8(i)
#define    S2B_no(i)
#define    SC(index, lg_grp, lg_delta, ndelta, bin, lg_delta_lookup) \
    S2B_##lg_delta_lookup(index)
    SIZE_CLASSES
#undef S2B_3
#undef S2B_4
#undef S2B_5
#undef S2B_6
#undef S2B_7
#undef S2B_8
#undef S2B_9
#undef S2B_no
#undef SC
};
```

这里的S2B_xx是一系列宏的嵌套展开, 最终对应的就是不同group在lookup table中占据的字节数以及bin索引. 相信看懂了前面的介绍就不难理解.

另一方面, 大于LOOKUP_MAXCLASS但小于SMALL_MAXCLASS的size class不能查表获得, 需要进行计算. 简言之, 一个bin number是三部分组成的,

```
bin_number = NTBIN + group_number << LG_SIZE_CLASS_GROUP + mod
```

即tiny bin数量加上其所在group再加上group中的偏移(0-2). 源码如下,

```
JEMALLOC_INLINE size_t
small_size2bin_compute(size_t size)
{
    ......
    {
        // xf: lg_floor相当于ffs
        size_t x = lg_floor((size<<1)-1);
```

```
        // xf: 计算size class所在group number
        size_t shift = (x < LG_SIZE_CLASS_GROUP + LG_QUANTUM) ? 0 :
            x - (LG_SIZE_CLASS_GROUP + LG_QUANTUM);
        size_t grp = shift << LG_SIZE_CLASS_GROUP;

        size_t lg_delta = (x < LG_SIZE_CLASS_GROUP + LG_QUANTUM + 1)
            ? LG_QUANTUM : x - LG_SIZE_CLASS_GROUP - 1;

        size_t delta_inverse_mask = ZI(-1) << lg_delta;
        // xf: 计算剩余mod部分
        size_t mod = ((((size-1) & delta_inverse_mask) >> lg_delta)) &
            ((ZU(1) << LG_SIZE_CLASS_GROUP) - 1);

        // xf: 组合计算bin number
        size_t bin = NTBINS + grp + mod;
        return (bin);
    }
}
```

其中LG_SIZE_CLASS_GROUP是size_classes.h中的定值, 代表一个group中包含的bin数量, 根据binary buddy算法, 该值通常情况下是2.
而要找到size class所在的group, 与其最高有效位相关. Je通过类似于ffs的函数首先获得size的最高有效位x,

```
size_t x = lg_floor((size<<1)-1);
```

至于group number, 则与quantum size有关. 因为除了tiny class, quantum size位于group #0的第一个. 因此不难推出,

```
group_number = 2^x / quantum_size / 2^LG_SIZE_CLASS_GROUP
```

对应代码就是,

```
size_t shift = (x < LG_SIZE_CLASS_GROUP + LG_QUANTUM) ? 0 :
        x - (LG_SIZE_CLASS_GROUP + LG_QUANTUM);
```

而对应group起始位置就是,

```
size_t grp = shift << LG_SIZE_CLASS_GROUP;
```

至于mod部分, 与之相关的是最高有效位之后的两个bit.
Je在这里经过了复杂的位变换,

```
size_t lg_delta = (x < LG_SIZE_CLASS_GROUP + LG_QUANTUM + 1)
    ? LG_QUANTUM : x - LG_SIZE_CLASS_GROUP - 1;
size_t delta_inverse_mask = ZI(-1) << lg_delta;
size_t mod = ((((size-1) & delta_inverse_mask) >> lg_delta)) &
    ((ZU(1) << LG_SIZE_CLASS_GROUP) - 1);
```

上面代码直白的翻译, 实际上就是要求得如下两个bit,

```
                    1 0000
                   10 0000
                   11 0000
group #0             100 0000
------------------------------------------------
                                +--+
                  101 0000 - 1 = 1|00| 1111
                  110 0000 - 1 = 1|01| 1111
                  111 0000 - 1 = 1|10| 1111
group #1            1000 0000 - 1 = 1|11| 1111
                                +--+
------------------------------------------------
                                +--+
                 1010 0000 - 1 = 1|00|1 1111
                 1100 0000 - 1 = 1|01|1 1111
                 1110 0000 - 1 = 1|10|1 1111
group #2           10000 0000 - 1 = 1|11|1 1111
                                +--+
------------------------------------------------
```

根据这个图示再去看Je的代码就不难理解了. mod的计算结果就是从0-3的数值.

而最终的结果是前面三部分的组合即,

size_t bin = NTBINS + grp + mod;

而bin2size查询就简单得多. 上一节介绍SIZE_CLASSES时提到过small region的计算公式, 只需要根据该公式提前计算出所有bin对应的region size, 直接查表即可.
这里不再赘述.

----[2.5 - bins (arena_bin_t)

run是分配的执行者, 而分配的调度者是bin. 这个概念同Dl中的bin是类似的, 但Je中bin要更复杂一些. 直白地说, 可以把bin看作non-full run的仓库, bin负责记录当前arena中某一个size class范围内所有non-full run的使用情况. 当有分配请求时, arena查找相应size class的bin, 找出可用于分配的run, 再由run分配region. 当然, 因为只有small region分配需要run, 所以bin也只对应small size class.

与bin相关的数据结构主要有两个, 分别是arena_bin_t和arena_bin_info_t.
在2.1.3中提到arena_t内部保存了一个bin数组, 其中的成员就是arena_bin_t.

其结构如下,

```
struct arena_bin_s {
    malloc_mutex_t       lock;
    arena_run_t          *runcur;
    arena_run_tree_t     runs;
    malloc_bin_stats_t   stats;
};
```

lock: 该lock同arena内部的lock不同, 主要负责保护current run. 而对于run本身的
      分配和释放还是需要依赖arena lock. 通常情况下, 获得bin lock的前提是获得

arena lock, 但反之不成立.

runcur: 当前可用于分配的run, 一般情况下指向地址最低的non-full run, 同一时间一个bin只有一个current run用于分配.

runs: rb tree, 记录当前arena中该bin对应size class的所有non-full runs. 因为分配是通过current run完成的, 所以也相当于current run的仓库.

stats: 统计信息.

另一个与bin相关的结构是arena_bin_info_t. 与前者不同, bin_info保存的是arena_bin_t的静态信息, 包括相对应size class run的bitmap offset, region size, region number, bitmap info等等(此类信息只要class size决定, 就固定下来). 所有上述信息在Je中由全局数组arena_bin_info记录. 因此与arena无关, 全局仅保留一份.

arena_bin_info_t的定义如下,

```
struct arena_bin_info_s {
    size_t        reg_size;
    size_t        redzone_size;
    size_t        reg_interval;
    size_t        run_size;
    uint32_t    nregs;
    uint32_t    bitmap_offset;
    bitmap_info_t    bitmap_info;
    uint32_t    reg0_offset;
};
```

reg_size: 与当前bin的size class相关联的region size.

reg_interval: reg_size+redzone_size

run_size: 当前bin的size class相关联的run size.

nregs: 当前bin的size class相关联的run中region数量.

bitmap_offset: 当前bin的size class相关联的run中bitmap偏移.

bitmap_info: 记录当前bin的size class相关联的run中bitmap信息.

reg0_offset: index为0的region在run中的偏移量.

以上记录的静态信息中尤为重要的是bitmap_info和bitmap_offset.

其中bitmap_info_t定义如下,

```
struct bitmap_info_s {
    size_t nbits;
    unsigned nlevels;
    bitmap_level_t levels[BITMAP_MAX_LEVELS+1];
};
```

nbits: bitmap中逻辑bit位数量(特指level#0的bit数)

nlevels: bitmap的level数量

levels: level偏移量数组, 每一项记录该级level在bitmap中的起始index

```
struct bitmap_level_s {
    size_t group_offset;
};
```

在2.3.1节中介绍arena_run_t时曾提到Je通过外挂bitmap将bookkeeping和user memory分离. 但bitmap查询速度要慢于boundary tag. 为了弥补这个缺陷, Je对此做了改进, 通过多级level缓冲以替代线性查找.

Je为bitmap增加了多级level, bottom level同普通bitmap一致, 每1bit代表一个region. 而高一级level中1bit代表前一级level中一个byte. 譬如说, 若我们在当前run中存在128个region, 则在ILP32系统上, 需要128/32 = 4byte来表示这128个region. Je将这4个byte看作level #0. 为了进一步表示这4个字节是否被占用, 又额外需要1byte以缓存这4byte的内容(仅使用了4bit), 此为level#1. 即整个bitmap, 一共有2级level, 共5byte来描述.

```
                +--------------+             +--------+
      +-----------|------------ +|   +----------|-------+|
      v           v             ||  v          v       ||
+-------------------------------------------------------------------------
| 1101 0010 | 0000 0000 | ... | 10?? ???? | ???? ???? | 1??? ????    | ...
+-------------------------------------------------------------------------
|<--------- level #0 -------->|<----- level #1 ------>|<- level #2 ->|
```

----[2.6 - Thread caches (tcache_t)

TLS/TSD是另一种针对多线程优化使用的分配技术, Je中称为tcache. tcache解决的是同一cpu core下不同线程对heap的竞争. 通过为每个线程指定专属分配区域, 来减小线程间的干扰. 但显然这种方法会增大整体内存消耗量. 为了减小副作用, je将tcache设计成一个bookkeeping结构, 在tcache中保存的仅仅是指向外部region的指针, region对象仍然位于各个run当中. 换句话说, 如果一个region被tcache记录了, 那么从run的角度看, 它就已经被分配了.

tcache的内容如下,

```
struct tcache_s {
    ql_elm(tcache_t) link;
    uint64_t         prof_accumbytes;
    arena_t            *arena;
    unsigned         ev_cnt;
    unsigned         next_gc_bin;
    tcache_bin_t     tbins[1];
};
```

link: 链接节点, 用于将同一个arena下的所有tcache链接起来.

prof_accumbytes: memory profile相关.

arena: 该tcache所属的arena指针.

ev_cnt: 是tcache内部的一个周期计数器. 每当tcache执行一次分配或释放时,
ev_cnt会记录一次. 直到周期到来, Je会执行一次incremental gc.
这里的gc会清理tcache中多余的region, 将它们释放掉. 尽管这不意味
着系统内存会获得释放, 但可以解放更多的region交给其他更饥饿的
线程以分配.

next_gc_bin: 指向下一次gc的binidx. tcache gc按照一周期清理一个bin执行.

tbins: tcache bin数组. 同样外挂在tcache后面.

同arena bin类似, tcache同样有tcache_bin_t和tcache_bin_info_t.
tcache_bin_t作用类似于arena bin, 但其结构要比后者更简单. 准确的说,
tcache bin并没有分配调度的功能, 而仅起到记录作用. 其内部通过一个stack
记录指向外部arena run中的region指针. 而一旦region被cache到tbins内,
就不能再被其他任何线程所使用, 尽管它可能甚至与其他线程tcache中记录的
region位于同一个arena run中.

tcache bin结构如下,
```
struct tcache_bin_s {
   tcache_bin_stats_t tstats;
   int     low_water;
   unsigned    lg_fill_div;
   unsigned    ncached;
   void        **avail;
}
```

tstats: tcache bin内部统计.

low_water: 记录两次gc间tcache内部使用的最低水线. 该数值与下一次gc时尝试
释放的region数量有关. 释放量相当于low water数值的3/4.

lg_fill_div: 用作tcache refill时作为除数. 当tcache耗尽时, 会请求arena run
进行refill. 但refill不会一次性灌满tcache, 而是依照其最大容量
缩小2^lg_fill_div的倍数. 该数值同low_water一样是动态的, 两者
互相配合确保tcache处于一个合理的充满度.

ncached: 指当前缓存的region数量, 同时也代表栈顶index.

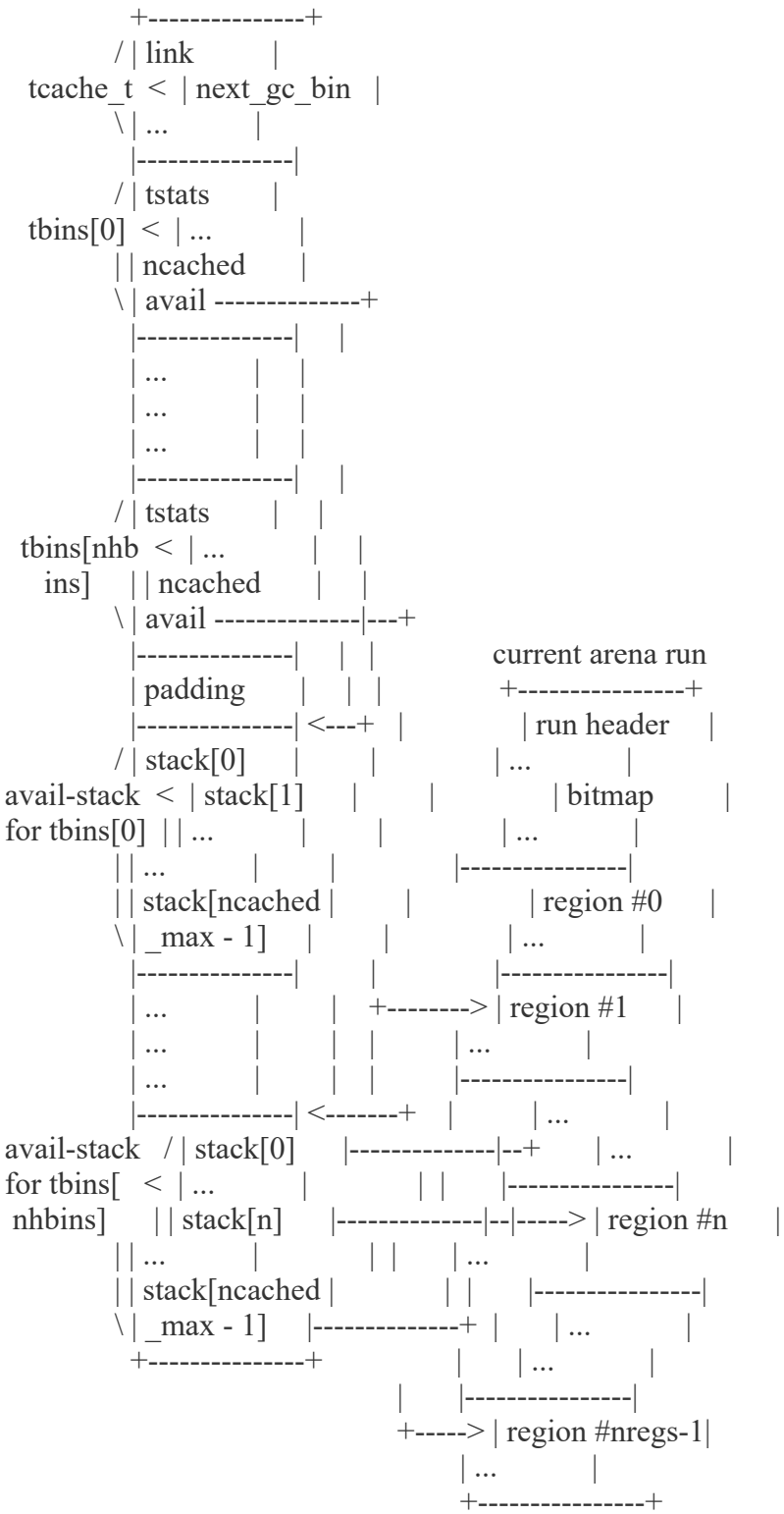
avail: 保存region指针的stack, 称为avail-stack.

tcache_bin_info_t保存tcache bin的静态信息. 其本身只保存了tcache max容量.
该数值是在tcache boot时根据相对应的arena bin的nregs决定的. 通常等于nregs
的二倍, 但不得超过TCACHE_NSLOTS_SMALL_MAX. 该数值默认为200, 但在android
中大大提升了该限制, small bins不得超过8, large bins则为16.

```
struct tcache_bin_info_s {
   unsigned    ncached_max;
```

};

tcache layout如下,



----[2.7 - Extent Node (extent_node_t)

extent node代表huge region, 即大于chunk大小的内存单元. 同arena run不同,
extent node并非是一个header构造, 而是外挂的. 因此其本身仍属small region.

只不过并不通过bin分配, 而由base_nodes直接动态创建.

Je中对所有huge region都是通过rb tree管理. 因此extent node同时也是tree node.
一个node节点被同时挂载到两棵rb tree上. 一棵采用szad的查询方式, 另一棵则采用
纯ad的方式. 作用是当执行chunk recycle时查询到可用region, 后面会详述.

```
struct extent_node_s {
    rb_node(extent_node_t)    link_szad;
    rb_node(extent_node_t)    link_ad;
    prof_ctx_t        *prof_ctx;
    void            *addr;
    size_t            size;
    arena_t            *arena;
    bool            zeroed;
};
```

link_szad: szad tree的link节点.

link_ad: ad tree的link节点.

prof_ctx: 用于memory profile.

addr: 指向huge region的指针.

size: region size.

arena: huge region所属arena.

zeroed: 代表是否zero-filled, chunk recycle时会用到.

----[2.8 - Base

base并不是数据类型, 而是一块特殊区域, 主要服务于内部meta data(例如arena_t,
tcache_t, extent_node_t等等)的分配. base区域管理与如下变量相关,

```
static malloc_mutex_t    base_mtx;
static void        *base_pages;
static void        *base_next_addr;
static void        *base_past_addr;
static extent_node_t    *base_nodes;
```

base_mtx: base lock
base_pages: base page指针, 代表整个区域的起始位置.
base_next_addr: 当前base指针, 类似于brk指针.
base_past_addr: base page的上限指针.
base_nodes: 指向extent_node_t链表的外挂头指针.

base page源于arena中的空闲chunk, 通常情况下, 大小相当于chunk. 当base耗尽时,
会以chunk alloc的名义重新申请新的base pages.

为了保证内部meta data的快速分配和访问. Je将内部请求大小都对齐到cache-line上, 以避免在SMP下的false sharing. 而分配方式上, 采用了快速移动base_next_addr 指针进行高速开采的方法.

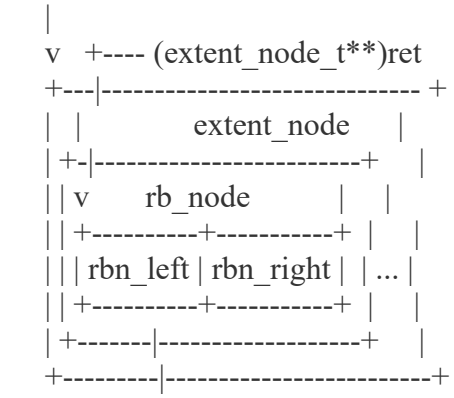
```
void *
base_alloc(size_t size)
{
    ......
    // xf: 将内部分配请求对齐的cache-line上, 阻止false sharing
    csize = CACHELINE_CEILING(size);

    malloc_mutex_lock(&base_mtx);
    // xf: 如果base耗尽, 则重新分配base_pages. 默认大小为chunksize.
    if ((uintptr_t)base_next_addr + csize > (uintptr_t)base_past_addr) {
        if (base_pages_alloc(csize)) {
            malloc_mutex_unlock(&base_mtx);
            return (NULL);
        }
    }
    // xf: 快速向前开采
    ret = base_next_addr;
    base_next_addr = (void *)((uintptr_t)base_next_addr + csize);
    malloc_mutex_unlock(&base_mtx);

    return (ret);
}
```

与通常的base alloc有所不同, 分配extent_node_t会优先从一个node链表中获取节点, 而base_nodes则为该链表的外挂头指针. 只有当其耗尽时, 才使用前面的分配方式. 这里区别对待extent_node_t与其他类型, 主要与chunk recycle机制有关, 后面会做详细说明.

有意思的是, 该链表实际上借用了extent node内部rb tree node的左子树节点指针作为其link指针. 如2.7节所述, extent_node_t结构的起始位置存放一个rb node. 但在这里, 当base_nodes赋值给ret后, 会强制将ret转型成(extent_node_t **), 实际上就是指向extent_node_t结构体的第一个field的指针, 并将其指向的node 指针记录到base_nodes里, 成为新的header节点. 这里需要仔细体会这个强制类型转换的巧妙之处.



```
            v
base_nodes---> +---------------+
            | extent_node   |
            +---------------+
```

```
extent_node_t *
base_node_alloc(void)
{
    extent_node_t *ret;

    malloc_mutex_lock(&base_mtx);
    if (base_nodes != NULL) {
        ret = base_nodes;
        // xf: 这里也可以理解为, base_nodes = (extent_node_t*)(*ret);
        base_nodes = *(extent_node_t **)ret;
        malloc_mutex_unlock(&base_mtx);
    } else {
        malloc_mutex_unlock(&base_mtx);
        ret = (extent_node_t *)base_alloc(sizeof(extent_node_t));
    }

    return (ret);
}
```

--[3 - Allocation

----[3.1 - Overview

在2.3.2节中得知, Je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, Je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Je是否初始化, 如果没有则初始化Je, 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 - Initialize

Je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Je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, Je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Je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,

1. 确认chunk_size和chunk_npages
2. 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.

3. 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Je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));
}
```

------[3.3.3 - arena_bin_nonfull_run_get

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Je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.

Je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Je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. Je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chunk alloc, 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Je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);
    }

    // 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Je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.
dss_max: 当前dss区域上限.

dss alloc算法如下,
1. 获取brk指针, 更新到dss_max.
2. 将dss_max对齐到chunksize边界上, 计算padding大小gap_size
3. 再将dss_max对齐到aligment边界上, 得到cpad_size
4. 计算需要分配的大小, 并尝试sbrk
    incr = gap_size + cpad_size + size
5. 分配成功, 检查cpad是否非0, 是则将这部分重新回收. 而gap_size部分因为
  不可用则被废弃.
6. 如果分配失败, 则检查dss是否耗尽, 如果没有则返回1重新尝试. 否则返回.

示意图,
```
chunk_base            cpad       ret       dss_next
   |                   |          |          |
   v                   v          v          v
   +--------+----------+-----------+------   ---+
   |  used  | gap_size | cpad_size | size ...   |
   +--------+----------+-----------+------   ---+
            |<------------- incr -------------->|
            ^          ^           ^
            |          |           |
        dss_max  chunk_base +     +-- chunk_base +
                    chunk_size          alignment
```

源码注释,
```
void *
chunk_alloc_dss(size_t size, size_t alignment, bool *zero)
{
   ......
   // xf: dss是否耗尽
   malloc_mutex_lock(&dss_mtx);
   if (dss_prev != (void *)-1) {
      ......
      do {
         // xf: 获取当前dss指针
         dss_max = chunk_dss_sbrk(0);

         // xf: 计算对齐到chunk size边界需要的padding大小
         gap_size = (chunksize - CHUNK_ADDR2OFFSET(dss_max)) &
            chunksize_mask;
         // xf: 对齐到chunk边界的chunk起始地址
         cpad = (void *)((uintptr_t)dss_max + gap_size);
         // xf: 对齐到alignment边界的起始地址
         ret = (void *)ALIGNMENT_CEILING((uintptr_t)dss_max,
            alignment);
```

```
            cpad_size = (uintptr_t)ret - (uintptr_t)cpad;
            // xf: dss_max分配后的更新地址
            dss_next = (void *)((uintptr_t)ret + size);
            ......
            incr = gap_size + cpad_size + size;
            // xf: 从dss分配
            dss_prev = chunk_dss_sbrk(incr);
            if (dss_prev == dss_max) {

                dss_max = dss_next;
                malloc_mutex_unlock(&dss_mtx);
                // xf: 如果ret和cpad对齐不在同一个位置, 则将cpad开始
                // cpad_size大小的内存回收到szad/ad tree中. 而以之前
                // dss起始的gap_size大小内存由于本身并非对齐到
                // chunk_size, 则被废弃.
                if (cpad_size != 0)
                    chunk_unmap(cpad, cpad_size);
                ......
                return (ret);
            }
        } while (dss_prev != (void *)-1);   // xf: 反复尝试直至dss耗尽
    }
    malloc_mutex_unlock(&dss_mtx);

    return (NULL);
}
```

最后介绍chunk_alloc_mmap. 同dss方式类似, mmap也存在对齐的问题. 由于系统mmap调用无法指定alignment, 因此Je实现了一个可以实现对齐但速度更慢的mmap slow方式. 作为弥补, 在chunk alloc mmap时先尝试依照普通方式mmap, 如果返回值恰好满足对齐要求则直接返回(多数情况下是可行的). 否则将返回值munmap, 再调用mmap slow.

```
void *
chunk_alloc_mmap(size_t size, size_t alignment, bool *zero)
{
    ......
    ret = pages_map(NULL, size);
    if (ret == NULL)
        return (NULL);
    offset = ALIGNMENT_ADDR2OFFSET(ret, alignment);
    if (offset != 0) {
        pages_unmap(ret, size);
        return (chunk_alloc_mmap_slow(size, alignment, zero));
    }
    ......

    return (ret);
}
```

mmap slow通过事先分配超量size, 对齐后再执行trim, 去掉前后余量实现mmap对齐. page trim通过两次munmap将leadsize和trailsize部分分别释放. 因此理论上, mmap slow需要最多三次munmap.

```
   |<-------------alloc_size---------->|
   +-----------+-----   --+------------+
   | lead_size | size...  | trail_size |
   +-----------+-----   --+------------+
   ^           ^
   |           |
   pages       ret(alignment)
```

```
static void *
chunk_alloc_mmap_slow(size_t size, size_t alignment, bool *zero)
{
   ......
   alloc_size = size + alignment - PAGE;
   if (alloc_size < size)
      return (NULL);
   do {
      pages = pages_map(NULL, alloc_size);
      if (pages == NULL)
         return (NULL);
      leadsize = ALIGNMENT_CEILING((uintptr_t)pages, alignment) -
         (uintptr_t)pages;
      ret = pages_trim(pages, alloc_size, leadsize, size);
   } while (ret == NULL);
   ......
   return (ret);
}

static void *
pages_trim(void *addr, size_t alloc_size, size_t leadsize, size_t size)
{
   void *ret = (void *)((uintptr_t)addr + leadsize);
   ......
   {
      size_t trailsize = alloc_size - leadsize - size;

      if (leadsize != 0)
         pages_unmap(addr, leadsize);
      if (trailsize != 0)
         pages_unmap((void *)((uintptr_t)ret + size), trailsize);
      return (ret);
   }
}
```

----[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Je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);
        return (NULL);
    }

    node->addr = ret;
    node->size = csize;
    node->arena = arena;

    // xf: 插入huge tree上
    malloc_mutex_lock(&huge_mtx);
    extent_tree_ad_insert(&huge, node);
    malloc_mutex_unlock(&huge_mtx);
    ......
    return (ret);
}
```

--[4 - Deallocation

----[4.1 - Overview

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 page机制却增加了整个算法的复杂程度. 原因就在于, Je使用了不同以往的内存释放方式.

在Dl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-most space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Je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Je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, Je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
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);
    }
}
```

Je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, Je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, Je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 dirty pages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,

```
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Je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.
    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. Je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, Je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Je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)
{
    ......
    malloc_mutex_lock(&huge_mtx);

    key.addr = ptr;
    node = extent_tree_ad_search(&huge, &key);
    assert(node != NULL);
    assert(node->addr == ptr);
    extent_tree_ad_remove(&huge, node);

    malloc_mutex_unlock(&huge_mtx);

    huge_dalloc_junk(node->addr, node->size);
    arena_chunk_dalloc_huge(node->arena, node->addr, node->size);
    base_node_dalloc(node);
}

void
arena_chunk_dalloc_huge(arena_t *arena, void *chunk, size_t size)
{
    chunk_dalloc_t *chunk_dalloc;

    malloc_mutex_lock(&arena->lock);
    chunk_dalloc = arena->chunk_dalloc;
    if (config_stats) {
        arena->stats.mapped -= size;
        arena->stats.allocated_huge -= size;
        arena->stats.ndalloc_huge++;
        stats_cactive_sub(size);
    }
    arena->nactive -= (size >> LG_PAGE);
    malloc_mutex_unlock(&arena->lock);
    chunk_dalloc(chunk, size, arena->ind);
}
```

前面已做了充分介绍, 这里不再赘述.

----[5 - 总结: 与Dl的对比

1. 单核单线程分配能力上两者不相上下, 甚至小块内存分配速度理论上Dl还略占优势. 原因是Dl利用双向链表组织free chunk可以做到O(1), 而尽管Je在bitmap上做了一定优化, 但不能做到常数时间.

2. 多核多线程下, Je可以秒杀Dl. arena的加入既可以避免false sharing, 又可以减少线程间lock contention. 另外, tcache也是可以大幅加快多线程分配速度的技术. 这些Dl完全不具备竞争力.

3. 系统内存交换效率上也是Je占明显优势. Je使用mmap/madvise的组合要比Dl使用sbrk/mmap/munmap灵活的多. 实际对系统的压力也更小. 另外, Dl使用dss->mmap, 追求的是速度, 而Je相反mmap->dss, 为的是灵活性.

4. 小块内存的碎片抑制上双方做的都不错, 但总体上个人觉得Je更好一些. 首先dalloc时, 两者对空闲内存都可以实时coalesce. alloc时Dl依靠dv约束外部碎片, Je更简单暴力, 直接在固定的small runs里分配.

   两相比较, dv的候选者是随机的, 大小不固定, 如果选择比较小的chunk, 效果其实有限. 更甚者, 当找不到dv时, Dl会随意切割top-most space, 通常这不利于heap trim.

   而small runs则是固定大小, 同时是页面的整数倍, 对外部碎片的约束力和规整度上都更好.

   但Dl的优势在算法更简单, 速度更快. 无论是coalesce还是split代价都很低. 在Je中有可能因为分配8byte的内存而实际去分配并初始化4k甚至4M的空间.

5. 大块内存分配能力上, Dl使用dst管理, 而Je采用rb tree. 原理上, 据说rb tree的cache亲和力较差, 不适合memory allocator. 我没有仔细研究Je的rb tree实现有何过人之处, 暂且认为各有千秋吧. 可以肯定的是Je的large/huge region具有比Dl更高的内部碎片, 皆因为其更规整的size class划分导致的.

6. 说到size class, 可以看到Je的划分明显比Dl更细致, tiny/small/large/huge四种分类能兼顾更多的内存使用模型. 且根据不同架构和配置, 可以灵活改变划分方式, 具有更好的兼容性. Dl划分的相对粗糙很多且比较固定. 一方面可能在当时256byte以上就可以算作大块的分配了吧. 另一方面某种程度是碍于算法的限制. 比如在boundary tag中为了容纳更多的信息, 就不能小于8byte(实际有效的最小chunk是16byte), bin数量不得多余31个也是基于位运算的方式.

7. bookkeeping占用上Dl因为算法简单, 本应该占用更少内存. 但由于boundary tag本身导致的占用, chunk数量越多, bookkeeping就越大. 再考虑到系统回收效率上的劣势, 应该说, 应用内存占用越大, 尤其是小内存使用量越多, 运行时间越长, Dl相对于Je内存使用量倾向越大.

8. 安全健壮性. 只说一点, boundary tag是原罪, 其他的可以免谈了.

--[附: 快速调试Jemalloc

一个简单的调试Je的方法是以静态库的方式将其编译到你的应用程序中.
先编译Je的静态库, 在源码目录下执行,

```
./configure
make
make install
```

就可以编译并安装Je到系统路径. 调试还必须打开一些选项, 例如,

```
./configure --enable-debug  --with-jemalloc-prefix=<prefix>
```

这些选项的意义可以参考INSTALL文档. 比如,

--disable-tcache 是否禁用tcache, 对调试非tcache流程有用.
--disable-prof 是否禁用heap profile.
--enable-debug 打开调试模式, 启动assert并关闭优化.
--with-jemalloc-prefix 将编译出的malloc加上设定的前缀, 以区别c库的调用.

之后就可以将其编译到你的代码中, 如,

```
gcc main.c /usr/local/lib/libjemalloc.a -std=c99 -O0 -g3 -pthread -o jhello
```