Jemalloc 深入分析

Evers Chen

Revision History

Revision	Date	Author	Description
0.1	2018-10-30	Evers.chen	Create
1.0	2019-01-09	Evers.chen	Initial draft
1.1	2019-01-25	Evers.chen	

Table of Contents

Revision History 2
Table of Contents 1
1. Introduction 1
1.1. 主要内容 1
1.2. 介绍 1
2. Jemalloc 的数据结构 2
2.1. 配对堆 Pairing Heap 2
2.1.1. phn_merge_ordered 2
2.1.2. ph_merge_siblings 3
2.1.3. ph_merge_aux 5
2.1.4. ph_merge_children 5
2.1.5. a_prefix##first 5
2.1.6. a_prefix##insert 5
2.1.7. a_prefix##remove_first 5
2.1.8. a_prefix##remove 5
2.1.9. 配对堆在 jemalloc 中的使用 5
2.2. bitmap 的两种实现 9
2.2.1. 什么时候使用 bitmap TREE 9
2.2.2. bitmap TREE 设计 11
2.2.3. bitmap_sfu 的处理过程 11
2.2.4. bitmap 初始化过程 12
2.2.5. bitmap_info 初始化过程 14
2.2.6. bitmap_full 检测 14
2.2.7. bitmap_set 15
2.2.8. bitmap_sfu 15
2.3. radix tree 基数树 15
2.3.1. radix tree 的 level 和对应的 subtree 结构 16
2.3.2. radix tree 头文件定义 16
2.3.3. radix tree 在 jemalloc 的实现 18
2.3.4. radix tree gdb 数据 22
2.4. red-black tree 红黑树 24
2.4.1. AVL 失去平衡后的 4 种姿态 25
2.4.2. AVL 旋转 26

2.4.3. RB TREE 定义 28
2.4.4. RB TREE 插入过程 28
2.4.5. RB TREE 删除过程 34
2.5. Ring definitions 双向循环链表 47
2.5.1. Ring 在 jemalloc 中的使用 47
2.6. List definitions 双向链表 47
2.6.1. List definitions 在 jemalloc 中的使用 48
2.7. PRNG 线性同余伪随机数生成器和原子操作的使用 49
2.7.1. 原子操作的使用 49
2.7.2. 线性同余伪随机数生成器 49
3. Tcache 实现原理 53
3.1. TSD:thread specific data 线程局部存储 53
3.2. Tcache 和 arena 的关系 55
3.3. Tcache 的定义 56
3.4. Tcache 的结构 58
3.5. Tcache boot 与初始化 60
3.6. Tcache fill 过程 61
3.7. Tcache 的分配过程 61
3.8. Tcache 的回收 flush 过程 62
3.9. Android 对 TCACHE_NSLOTS_SMALL_MAX 配置问题 65
4. Jemalloc 的参数和调试相关 66
4.1. Jemalloc 的调试开关 67
4.1.1. JEMALLOC_DEBUG 67
4.1.2. JEMALLOC_FILL 67
4.1.3. JEMALLOC_TCACHE 67
4.2. Android 里 Jemalloc 的初始化参数 68
4.2.1. Arena 数量 68
4.2.2. LG_SIZEOF_PTR 68
4.2.3. NBINS 69
4.2.4. Chunk size 大小 69
4.2.5. Tcache 相关配置 70
4.3. Jemalloc 参数初始化 71
4.3.1. 以 LG_开始的宏常量定义 71
4.3.2. map_bias 的初始化 72
4.4. Jemalloc 的统计输出 73
5. Jemalloc 多核/多线程分配和互斥锁 77

5.1. 锁的分类 77
5.2. Tcache 分配的过程 77
5.3. 从 bin 的 runcur 中的分配过程 78
5.4. 从 bin 的 runs 中的分配过程 78
5.5. arena->runs_avail[i] 中的分配过程 78
5.6. new chunk 的分配过程 78
6. Region size 设计以及和 index 对应关系 78
6.1. Region size 步长的设计 78
6.2. Region 相关的参数定义（anrdoid 64bits） 79
6.3. SIZE_CLASSES 定义 80
6.4. size2index_tab （用于 reg_size 小于 4096 的快速查找） 84
6.5. size2index_compute 的计算过程 86
6.6. index2size_compute 的计算过程 89
6.7. 步长增加规律分析-浪费率控制 91
6.8. index2size_tab 91
7. Jemalloc 内存分配释放过程 93
7.1. 从 arena 中分配 small size 内存的过程 93
7.1.1. tcache 中的分配过程 93
7.1.2. 从 bin 的 runcur 中的分配过程 94
7.1.3. 从 bin 的 runs，arena->runs_avail[i],new chunk 的分配过程 95
7.1.4. 从 arena bin 的分配 debug 过程 96
7.2. small size 内存的释放过程 97
7.2.1. 释放内存到 tcache 的过程 97
7.2.2. 如果 tcache 已满，释放一半 tcache 回各自的 run 97
7.2.3. 如果 run 已满，释放回 arena->runs_avail[i] 97
7.2.4. 如果 chunk 已满，释放 chunk 回 arena->spare 97
7.2.5. 如果 arena->spare 已保存前面释放的 chunk，则释放 spare 的 chunk .. 98
7.3. 从 arena 中分配 large size 内存的过程 98
7.3.1. tcache 分配过程 98
7.3.2. 非 tcache 的分配过程 98
7.4. large size 内存的释放过程 100
7.4.1. 释放 large 内存到 tcache 的过程 100
7.4.2. 如果 tcache 已满，释放一半 tcache 回 arena->runs_avail[i] 100
7.4.3. 如果 chunk 已满，释放 chunk 回 arena->spare 100
7.4.4. 如果 arena->spare 已保存前面释放的 chunk，则释放 spare 的 chunk 100
7.5. 从 arena 中分配 huge size 内存的过程 101

7.6. huge size 内存的释放过程 101
8. Chunk 102
8.1. Chunk 结构 （9832E 64 位系统） 102
8.2. Chunk 分配过程 102
8.3. 根据内存地址对 chunk 头的进一步分析 105
8.4. map_bits 107
8.5. map_misc_t 110
8.6. 从 arena 找到 chunk 地址 112
8.7. Chunk 的 register 过程 113
9. jemalloc 存储块(region、run、chunk) 113
9.1. 存储块(region、run、chunk) 113
9.2. arena_bin_info 114
9.3. arena_bin_info 中的 bitmap_info 115
10. Jemalloc 的初始化过程 116
11. Jemalloc 的异常行为？ 119
11.1. Free(P+n)释放部分空间的问题 119
12. Jemalloc unit test 120
12.1. Jemalloc 的 unit test 设计 120
12.2. RB tree 的 unit test 设计 122
12.3. 和 unit test 配套的强壮的 assert 设计 126
13. Jemalloc 不相关问题 126
13.1. 32/64 位系统数据类型对应字节数 126
13.2. Assert 的使用 127
13.3. 0xFFLU 和 0xFFU 128
13.4. 堆排序 128
13.5. typedef struct 的使用 130
13.6. big-endian 和 little-endian 格式 130
14. 参考资料 130

1. Introduction

1.1. 主要内容

1. bitmap 查找算法的优化-32 路查找
2. paring heap，配对堆的使用，提高排序效率
3. RB tree 红黑树
4. Tcache 机制
5. 支持原子操作的线性同余伪随机数生成器
6. 动态头长度的计算过程 map_bias
7. Region size 设计以及和 index 对应关系
8. radix tree 基数树
9. 高可靠性编程讨论
10. small/large/huge 内存的分配和释放过程

1.2. 介绍

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 jemalloc, 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 dlmalloc 替换为 jemalloc, 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, jemalloc 在基本思路和实现上存在明显的差别. 比如, dlmalloc 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 jemalloc 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, **jemalloc 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(锁竞争). 因为无论 CPU 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, jemalloc 的分配速度越快. 而这一点是经典分配器所无法达到的.

本文档基于 jemalloc4.4.0 版本。

2. Jemalloc 的数据结构

2.1. 配对堆 Pairing Heap

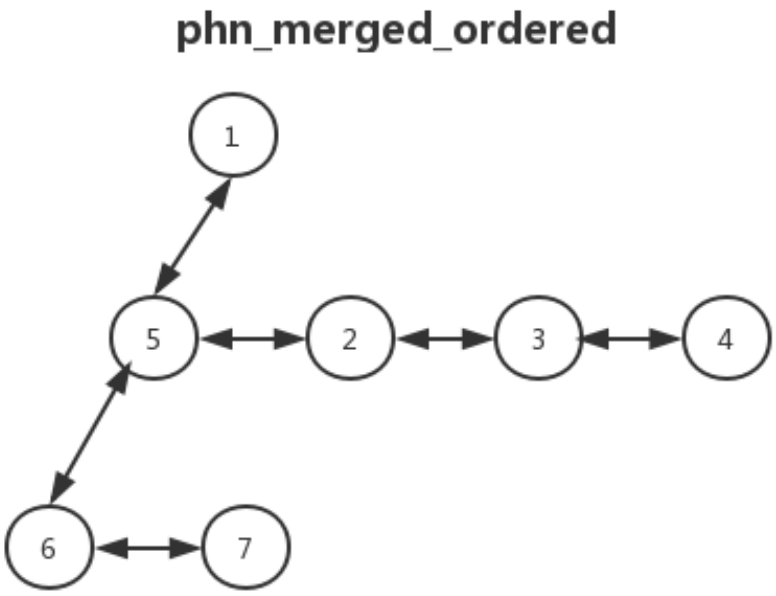
斐波那契堆主要有两个缺点：编程实现难度较大和实际效率没有理论的那么快（由于它的存储结构和四个指针）。Pairing Heap 的提出就是弥补斐波那契堆的两个缺点——编程简单操作的时间复杂度和斐波那契堆一样。

Pairing Heap 其实就是一个具有堆（最大堆或最小堆）性质的树，它的特性不是由它的结构决定的，而是由于它的操作（插入，合并，减小一个关键字等）决定的。

一个指针指向该节点的第一个孩子 lchild，一个指向它的下个兄弟 next，一个指向它的上个兄弟 prev（对于最左边的兄弟则指向它的父亲），对于第一个孩子，prev 属性表示该孩子的父结点；对于其他结点，prev 属性表示该结点的左兄弟。

2.1.1. phn_merge_ordered

把大的作为小的树的最左孩子，插入配对树，小的树如果有孩子作为大的树的根节点的兄弟节点。

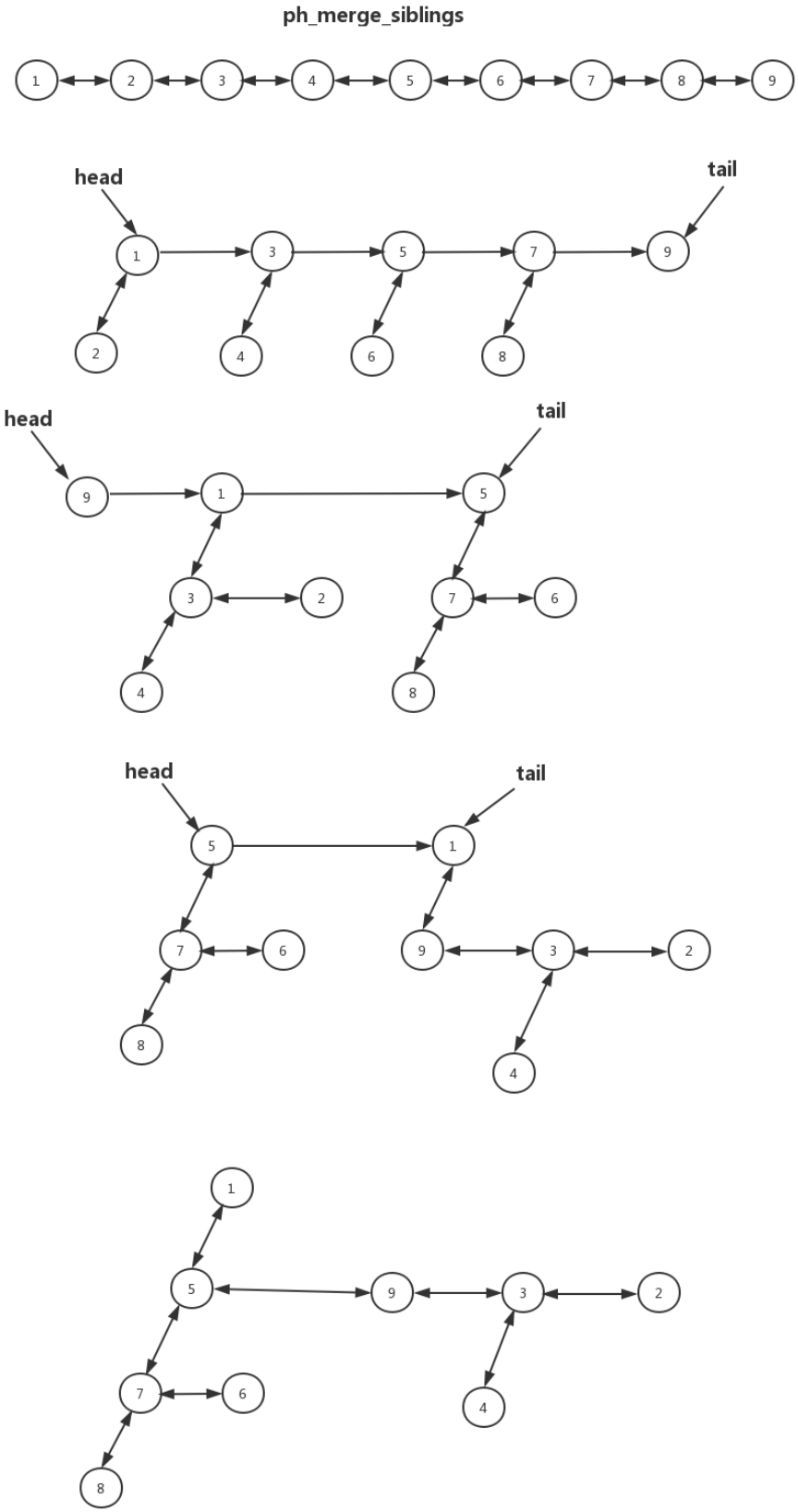


2.1.2. ph_merge_siblings

对兄弟节点两两进行 merge，merge 完后插入 FIFO 队尾（FIFO 是单链表的），再进行后续节点的操作，直到只剩下一个元素为止。

由于一开始没有队尾节点，所以先进行一轮的 merge 操作，找到队尾和头，然后再进行循环的 merge 操作，直到只剩下一个节点为止。

排序后返回一个最小值的节点。


ph_merge_siblings
1
2
3
4
5
6
7
8
9
head
tail
head
tail
head
tail
1
2
3
4
5
6
7
8
9

2.1.3. ph_merge_aux

对一个树作排序，找到一个最小值，作为树的 root。

2.1.4. ph_merge_children

对孩子节点做排序操作，找到一个最小值。

2.1.5. a_prefix##first

对这个树作排序，找到一个最小值并返回。调用 ph_merge_aux。

2.1.6. a_prefix##insert

如果树不为空，作为第一个兄弟节点。如果树为空作为 root 节点。

```
/*                                                      \
 * Treat the root as an aux list during insertion, and lazily       \
 * merge during a_prefix##remove_first().  For elements that \
 * are inserted, then removed via a_prefix##remove() before the      \
 * aux list is ever processed, this makes insert/remove          \
 * constant-time, whereas eager merging would make insert  \
 * O(log n).                                                \
```

2.1.7. a_prefix##remove_first

对这个树作排序，找到一个最小值作为根节点，并且调用 ph_merge_children，对孩子节点做排序操作，找到一个最小值作为树的新的根节点。返回原树的根节点。

2.1.8. a_prefix##remove

1）如果被删除的根节点，需要对树做 merge。Merge 后如果还是根节点，则需要对孩子节点做 merge 找到一个新的根节点。这样删除后树还是保持最小值在根节点。
2）如果被删除节点是最左孩子节点，则存在父节点。要不不存在父节点。
3）如果被删除节点是一个树，则可以对被删除节点所在的树的孩子节点进行 merge，merge 后从被删除节点所在的树找新的根节点，如果新的根节点存在，可以用这个节点代替被删除节点，如果该节点不存在，则直接删除这个节点。

2.1.9. 配对堆在 jemalloc 中的使用

主要用来管理 arena_t 的 runs_avail，和 arena_bin_t 的 runs。

```
/* Generate pairing heap functions. */
ph_gen(static UNUSED, arena_run_heap_, arena_run_heap_t,
arena_chunk_map_misc_t, ph_link, arena_snad_comp)

#define ph_gen(a_attr, a_prefix, a_ph_type, a_type, a_field, a_cmp)
a_attr  void a_prefix##new(a_ph_type *ph)
a_attr bool a_prefix##empty(a_ph_type *ph)
a_attr a_type * a_prefix##first(a_ph_type *ph)
a_attr void a_prefix##insert(a_ph_type *ph, a_type *phn)
a_attr a_type * a_prefix##remove_first(a_ph_type *ph)
a_attr void a_prefix##remove(a_ph_type *ph, a_type *phn)
```

实际的定义函数如下：

```
static void arena_run_heap_new(arena_run_heap_t *ph)
static bool arena_run_heap_empty(arena_run_heap_t *ph)
static a_type * arena_run_heap_first(arena_run_heap_t *ph)
static void arena_run_heap_insert(arena_run_heap_t *ph, arena_chunk_map_misc_t
*phn)
static a_type * arena_run_heap_remove_first(arena_run_heap_t *ph)
static void arena_run_heap_remove(arena_run_heap_t *ph, arena_chunk_map_misc_t
*phn)

/* Node structure. */
#define phn(a_type) \
struct { \
    a_type *phn_prev; \
    a_type *phn_next; \
    a_type *phn_lchild; \
}

/* Root structure. */
#define ph(a_type) \
struct { \
    a_type *ph_root; \
}
```

```
typedef ph(arena_chunk_map_misc_t) arena_run_heap_t;
struct {                                                \
        arena_chunk_map_misc_t *ph_root;                \
};


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

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

        /*
         * Heap of non-full runs.  This heap is used when looking for an
         * existing run when runcur is no longer usable.  We choose the
         * non-full run that is lowest in memory; this policy tends to keep
         * objects packed well, and it can also help reduce the number of
         * almost-empty chunks.
         */
        arena_run_heap_t   runs;

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

struct arena_s {
```

```
        /* This arena's index within the arenas array. */
        unsigned                ind;
        ……
        /*
         * Size-segregated address-ordered heaps of this arena's available runs,
         * used for first-best-fit run allocation.  Runs are quantized, i.e.
         * they reside in the last heap which corresponds to a size class less
         * than or equal to the run size.
         */
        arena_run_heap_t   runs_avail[NPSIZES];
};


phn(arena_chunk_map_misc_t)             ph_link;
struct {                                                \
        arena_chunk_map_misc_t *phn_prev;                       \
        arena_chunk_map_misc_t *phn_next;                       \
        arena_chunk_map_misc_t *phn_lchild;                     \
}

/*
 * Each arena_chunk_map_misc_t corresponds to one page within the chunk, just
 * like arena_chunk_map_bits_t.  Two separate arrays are stored within each
 * chunk header in order to improve cache locality.
 */
struct arena_chunk_map_misc_s {
        /*
         * Linkage for run heaps.  There are two disjoint uses:
         *
         * 1) arena_t's runs_avail heaps.
         * 2) arena_bin_t conceptually uses this linkage for in-use non-full
         *    runs, rather than directly embedding linkage.
         */
        phn(arena_chunk_map_misc_t)             ph_link;

        union {
```

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

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

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

比较函数：arena_snad_comp，先比较 arena_sn_comp，如果两个 arena_chunk_map_misc_t 来自同一个 chunk，则 arena_sn_comp 结果一样，再用 arena_ad_comp 直接比较两个地址。

2.2. bitmap 的两种实现

2.2.1. 什么时候使用 bitmap TREE

```
/*
 * Do some analysis on how big the bitmap is before we use a tree.  For a brute
 * force linear search, if we would have to call ffs_lu() more than 2^3 times,
 * use a tree instead.
 */

#if LG_BITMAP_MAXBITS - LG_BITMAP_GROUP_NBITS > 3
#  define USE_TREE
#endif
```

LG_BITMAP_MAXBITS= LG_RUN_MAXREGS= (LG_PAGE - LG_TINY_MIN)=12-3=9

LG_BITMAP_GROUP_NBITS= (LG_SIZEOF_BITMAP + 3)= LG_SIZEOF_LONG+3

=6（64bit）或者 5（32bit）。

```
/* sizeof(long) == 2^LG_SIZEOF_LONG. */
#ifdef __LP64__
#define LG_SIZEOF_LONG 3
#else
#define LG_SIZEOF_LONG 2
#endif
```

LG_BITMAP_MAXBITS - LG_BITMAP_GROUP_NBITS=9-6/5，所以在 64bit 的时候不定义 USE_TREE，在 32bit 的时候定义了 USE_TREE。

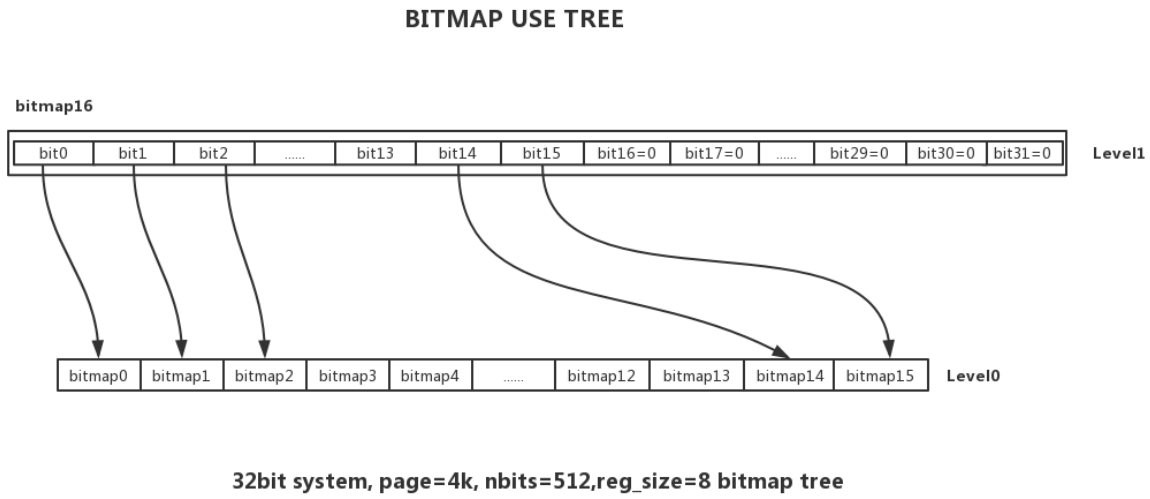
64bit 的时候，ffs_lu 调用 2^3=8 次，超过 8 次就用 TREE。32bit 的时候 ffs_lu 调用 2^4=16 次，所以使用了 TREE，看看效果如何？

```
/* Number of groups required to store a given number of bits. */
#define        BITMAP_BITS2GROUPS(nbits)                                \
   ((nbits + BITMAP_GROUP_NBITS_MASK=31) >> LG_BITMAP_GROUP_NBITS=5)

#elif LG_BITMAP_MAXBITS <= LG_BITMAP_GROUP_NBITS * 2
#define BITMAP_GROUPS_MAX BITMAP_GROUPS_2_LEVEL(BITMAP_MAXBITS)
LG_BITMAP_MAXBITS=9 <= LG_BITMAP_GROUP_NBITS * 2=5*5=10
BITMAP_MAXBITS=2^9
BITMAP_GROUPS_2_LEVEL(BITMAP_MAXBITS)=

#define        BITMAP_GROUPS_L0(nbits)                                  \
   BITMAP_BITS2GROUPS(nbits)
#define        BITMAP_GROUPS_L1(nbits)                                  \
   BITMAP_BITS2GROUPS(BITMAP_BITS2GROUPS(nbits))
#define        BITMAP_GROUPS_1_LEVEL(nbits)                             \
   BITMAP_GROUPS_L0(nbits)
#define        BITMAP_GROUPS_2_LEVEL(nbits)                             \
   (BITMAP_GROUPS_1_LEVEL(nbits) + BITMAP_GROUPS_L1(nbits))

BITMAP_GROUPS_1_LEVEL(512)=BITMAP_GROUPS_L0(512)=
BITMAP_BITS2GROUPS(512)=(512+31)>>5=16
BITMAP_GROUPS_L1(512)= BITMAP_BITS2GROUPS(16)=1
BITMAP_GROUPS_2_LEVEL(512)=16+1=17
```

BITMAP_GROUPS_MAX=17

2.2.2. bitmap TREE 设计

USE_TREE 时，level[0]还是放置所有的 groups，然后每往上一层，该层的每一位对应下层的一个 bitmap。如此往上，直到最高层只有一个 bitmap 为止。



512/32=16，所以这里共需要 16 个 bitmap，每一个 bitmap 是 32 位的字节。

2.2.3. bitmap_sfu 的处理过程

```
bitmap_sfu(bitmap_t *bitmap, const bitmap_info_t *binfo)
{
        size_t bit;
        bitmap_t g;
        unsigned i;

        assert(!bitmap_full(bitmap, binfo));

#ifdef USE_TREE
        i = binfo->nlevels - 1;//得到顶层 level
        g = bitmap[binfo->levels[i].group_offset];//得到顶层 bitmap 值
        bit = ffs_lu(g) - 1;
```

```
        while (i > 0) {
                i--;
                g = bitmap[binfo->levels[i].group_offset + bit];//得到下一层 bit 位对应的bitmap 值
                bit = (bit << LG_BITMAP_GROUP_NBITS) + (ffs_lu(g) - 1);//得到下一层的bit 所在的值，bit << LG_BITMAP_GROUP_NBITS 表示上一层的每一个单位值需要乘以32，表示下层的一个字节跨度。
        }
#else
        i = 0;
        g = bitmap[0];
        while ((bit = ffs_lu(g)) == 0) {
                i++;
                g = bitmap[i];
        }
        bit = (i << LG_BITMAP_GROUP_NBITS) + (bit - 1);
#endif
        bitmap_set(bitmap, binfo, bit);
        return (bit);
}
```

— Built-in Function: int __builtin_ffsl (unsigned long)
Returns one plus the index of the least significant 1-bit of x, or if x is zero, returns zero.
返回右起第一个‘1’的位置。
如果是 TREE 方式，需要从 root group 开始往下查找，相比非 TREE 的方式也会更快。按 16 个 bitmaps 为例，只需要调用 ffs_lu() 2 次。而非 TREE 方式，平均会在 4 次。

LG_BITMAP_GROUP_NBITS=6，i<<6=i*64，一个 bitmap 是标识 64 个 regions。

2.2.4. bitmap 初始化过程

```
void
bitmap_init(bitmap_t *bitmap, const bitmap_info_t *binfo)
{
        size_t extra;
```

```
        memset(bitmap, 0xffU, bitmap_size(binfo));
        extra=(BITMAP_GROUP_NBITS-(binfo-
>nbits&BITMAP_GROUP_NBITS_MASK))
            & BITMAP_GROUP_NBITS_MASK;
        if (extra != 0)
                bitmap[binfo->ngroups - 1] >>= extra;
}
```

extra 是多余的标志位，对于 64 位系统，extra 的计算过程如下：

```
extra=(64-(binfo->nbits&63))& 63;
```

所以当 nbits（extra 是 2 的次幂形式）大于等于 64 的，extra=0，当 nbits 小于 64 时，extra=64-nbits。然后把最后一个 bitmap 的高 extra 位置 0，即右移 extra 位。

对于使用 TREE 时的 bitmap_init，除了 level0 的 extra 置 0，还需要往上把其他的查找 bitmap 的 extra 位置 0。

```
bitmap_init(bitmap_t *bitmap, const bitmap_info_t *binfo)
{
        size_t extra;
        unsigned i;

        /*
         * Bits are actually inverted with regard to the external bitmap
         * interface, so the bitmap starts out with all 1 bits, except for
         * trailing unused bits (if any).  Note that each group uses bit 0 to
         * correspond to the first logical bit in the group, so extra bits
         * are the most significant bits of the last group.
         */
        memset(bitmap, 0xffU, bitmap_size(binfo));
        extra = (BITMAP_GROUP_NBITS - (binfo->nbits &
BITMAP_GROUP_NBITS_MASK))
            & BITMAP_GROUP_NBITS_MASK;
        if (extra != 0)
                bitmap[binfo->levels[1].group_offset - 1] >>= extra;
        for (i = 1; i < binfo->nlevels; i++) {
                size_t group_count = binfo->levels[i].group_offset -
                    binfo->levels[i-1].group_offset;
```

```
                extra = (BITMAP_GROUP_NBITS - (group_count &
                    BITMAP_GROUP_NBITS_MASK)) & BITMAP_GROUP_NBITS_MASK;
                if (extra != 0)
                        bitmap[binfo->levels[i+1].group_offset - 1] >>= extra;
        }
}
```

2.2.5. bitmap_info 初始化过程

1）对于非 TREE 的模式，只需要计算 ngroups，这里是 64bit 的系统。

```
binfo->ngroups = BITMAP_BITS2GROUPS(nbits);
/* Number of groups required to store a given number of bits. */
#define        BITMAP_BITS2GROUPS(nbits)                                \
    ((nbits + BITMAP_GROUP_NBITS_MASK) >> LG_BITMAP_GROUP_NBITS)
```

也就是：binfo->ngroups =(nbits+63)/64

2）对于 TREE 的模式，这里是 32bit 的系统。

首先置 level0 的 group 偏移从第 0 个开始，binfo->levels[0].group_offset = 0;

然后下一个 level 的 group 偏移，是上一个 level 的偏移+group 数量。

```
binfo->levels[i].group_offset = binfo->levels[i-1].group_offset+ group_count;
```

然后再把这一级的 group 数量作为 nbits 来计算下一级的 group 数量，

```
group_count = BITMAP_BITS2GROUPS(group_count);
```

这样计算，直到 group_count 为 1 为止。

为 1 后，把最后一个偏移保存在最高一级 level 中，实际这一级不作为查询的 bitmap，只是作为 group 数量的返回值，因为 group 计算从 0 开始，这里 group_count=1，加上去后刚好是 group 的个数。这个 i 会保存为 binfo->nlevels，后续所有的处理都是 i < binfo->nlevels 也说明这层实际不存 bitmap。

```
binfo->levels[i].group_offset = binfo->levels[i-1].group_offset+ group_count;
```

2.2.6. bitmap_full 检测

对于 TREE，只需要检查 root group 是否为 0，也就是最后一个 bitmap。

对于非 TREE，需要逐个检查，最坏的情况下是检查 8 个 bitmaps。

2.2.7. bitmap_set

对于非 TREE，直接把对应 bitmap 的对应位置 0。
对于 TREE，首先也需要把对应 bitmap 的对应位置 0，然后如果这个 bitmap 值为 0 后，需要把 level1 的对应的 bitmap 位置 0，如果这个时候这个 bitmap 值也为 0，则需要继续往上对查找 bitmap 置 0 的动作，直到 i == binfo->nlevels 或者当前的 bitmap 不为 0 为止。

2.2.8. bitmap_sfu

对于非 TREE，直接通过循环查找 bitmap 不为 0 的 bit。
对于 TREE，需要从上往下通过查找 bitmap 来找，对于 32bit 系统，理论上是 32 分查找，而且最大查找次数为 BITMAP_MAX_LEVELS=5，实际一般 2 次就完成，远远小于非 TREE 的最大 8 次 ffs_lu 操作。

2.3. radix tree 基数树

* This radix tree implementation is tailored to the singular purpose of
 * associating metadata with chunks that are currently owned by jemalloc.

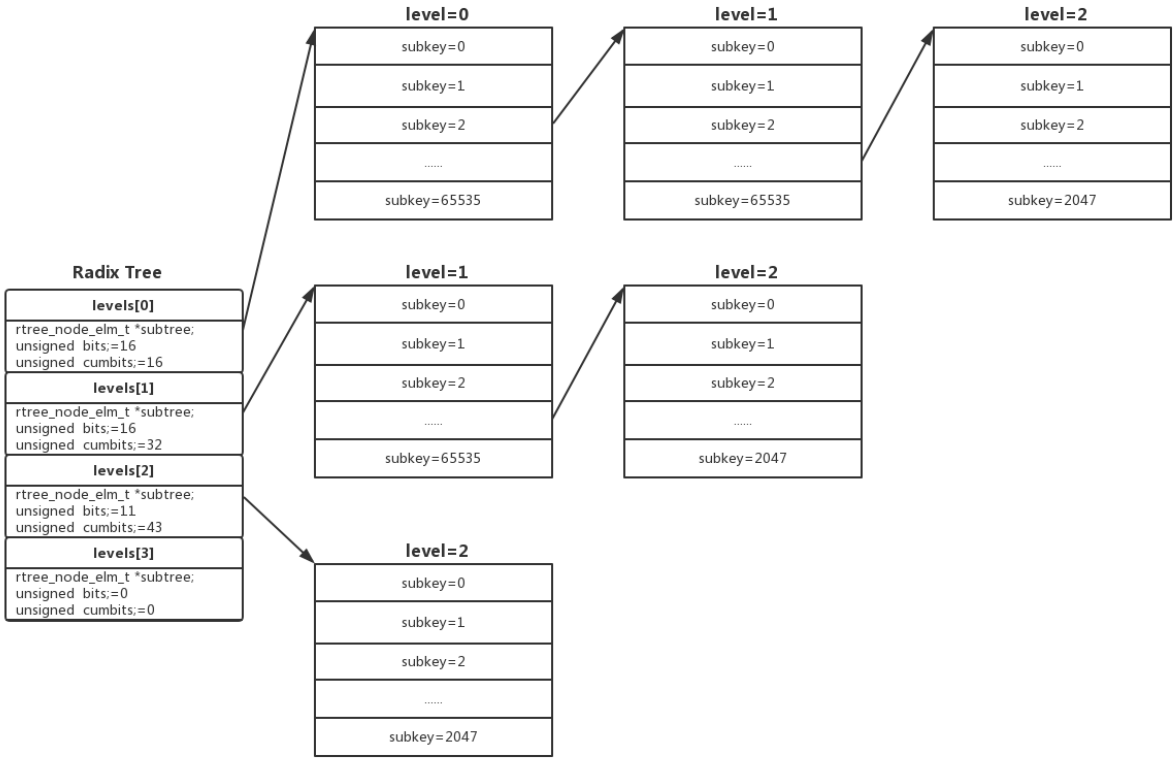
对于长整型数据的映射。怎样解决 Hash 冲突和 Hash 表大小的设计是一个非常头疼的问题。radix 树就是针对这样的稀疏的长整型数据查找，能高速且节省空间地完毕映射。借助于 radix 树，我们能够实现对于长整型数据类型的路由。

利用 radix 树能够依据一个长整型（比方一个长 ID）高速查找到其相应的对象指针。这比用 hash 映射来的简单，也更节省空间，使用 hash 映射 hash 函数难以设计，不恰当的 hash 函数可能增大冲突，或浪费空间。

radix tree 是一种多叉搜索树。树的叶子结点是实际的数据条目。每一个结点有一个固定的、2^n 指针指向子结点（每一个指针称为槽 slot，n 为划分的基的大小）（如果 n=2，就有 4 个指针指向子节点）。

radix 树为稀疏树提供了有效的存储，取代固定尺寸数组提供了键值到指针的高速查找。

2.3.1. radix tree 的 level 和对应的 subtree 结构



2.3.2. radix tree 头文件定义

```
struct rtree_node_elm_s {
        union {
                void                    *pun;
                rtree_node_elm_t        *child;
                extent_node_t           *val;
        };
};

struct rtree_level_s {
        /*
         * A non-NULL subtree points to a subtree rooted along the hypothetical
         * path to the leaf node corresponding to key 0.  Depending on what keys
         * have been used to store to the tree, an arbitrary combination of
```

```
         * subtree pointers may remain NULL.
         *
         * Suppose keys comprise 48 bits, and LG_RTREE_BITS_PER_LEVEL is 4.
         * This results in a 3-level tree, and the leftmost leaf can be directly
         * accessed via subtrees[2], the subtree prefixed by 0x0000 (excluding
         * 0x00000000) can be accessed via subtrees[1], and the remainder of the
         * tree can be accessed via subtrees[0].
         *
         *   levels[0] : [<unused> | 0x0001******** | 0x0002******** | ...]
         *
         *   levels[1] : [<unused> | 0x00000001**** | 0x00000002**** | ... ]
         *
         *   levels[2] : [val(0x000000000000) | val(0x000000000001) | ...]
         *
         * This has practical implications on x64, which currently uses only the
         * lower 47 bits of virtual address space in userland, thus leaving
         * subtrees[0] unused and avoiding a level of tree traversal.
         */
        union {
                void                    *subtree_pun;
                rtree_node_elm_t        *subtree;
        };
        /* Number of key bits distinguished by this level. */
        unsigned                bits;
        /*
         * Cumulative number of key bits distinguished by traversing to
         * corresponding tree level.
         */
        unsigned                cumbits;
};

struct rtree_s {
        rtree_node_alloc_t      *alloc;
        rtree_node_dalloc_t     *dalloc;
        unsigned                height;
        /*
```

```
        * Precomputed table used to convert from the number of leading 0 key
        * bits to which subtree level to start at.
        */
        unsigned            start_level[RTREE_HEIGHT_MAX];
        rtree_level_t       levels[RTREE_HEIGHT_MAX];
};
```

2.3.3. radix tree 在 jemalloc 的实现

```
JEMALLOC_ALWAYS_INLINE unsigned
rtree_start_level(rtree_t *rtree, uintptr_t key)
{
        unsigned start_level;
        if (unlikely(key == 0))
                return (rtree->height - 1);
        start_level = rtree->start_level[lg_floor(key) >>
            LG_RTREE_BITS_PER_LEVEL];
        assert(start_level < rtree->height);
        return (start_level);
}
```

对于 key 最高位 1 的位置（从 0 开始计数）如果

key 范围：0 - 2^16-1, lg_floor(key) =0->15 ， i=0， start_level=2,

key 范围：2^16 - 2^32-1, lg_floor(key) =16->31, i=1, start_level=2,

key 范围：2^32 - 2^48-1, lg_floor(key) =32->47, i=2, start_level=1,

key 范围：2^48 - 2^64-1, lg_floor(key) =48->63, i=3, start_level=0,

这个 start_level，对于可以 2^32 - 2^48-1, lg_floor(key) =32->47，因为高 16 位为 0，所以 start_level 直接从 1 开始。

```
(gdb) p je_chunks_rtree
$162 = {alloc = 0x7247636b60 <chunks_rtree_node_alloc>, dalloc = 0x0,
  height = 3, start_level = {2, 2, 1, 0}, levels = {
{{subtree_pun = 0x0, subtree = 0x0}, bits = 16, cumbits = 16}, //
{{subtree_pun = 0x7246e0bc00, subtree = 0x7246e0bc00}, bits = 16,cumbits = 32},
{{subtree_pun = 0x0, subtree = 0x0}, bits = 11,cumbits = 43}, //
{{subtree_pun = 0x0, subtree = 0x0}, bits = 0,cumbits = 0}//
}}
```

从这里可以看出，当前只有 level=1 的，key 范围 2^32 - 2^48-1, lg_floor(key) =32->47, 的 chunk 存在。Small 和 large 的 chunk 也是存这里吗？是的，后面通过 gdb 调试可以查看到所有的 chunk 地址。

```
rtree_subkey(rtree_t *rtree, uintptr_t key, unsigned level)
{
        return ((key >> ((ZU(1) << (LG_SIZEOF_PTR+3)) -
            rtree->levels[level].cumbits)) & ((ZU(1) <<
            rtree->levels[level].bits) - 1));
}
```

如果 level=0，rtree_subkey 返回 key 前 16bits 的值，值范围[0,65535]
如果 level=1，rtree_subkey 返回 key 高位第二个 16bits 的值，值范围[0,65535]
如果 level=2，rtree_subkey 返回 key 后续 11bits 的值，值范围[0,2047]

```
rtree_new(&chunks_rtree, (unsigned)((ZU(1) << (LG_SIZEOF_PTR+3)) -
            opt_lg_chunk), chunks_rtree_node_alloc, NULL)
```

((ZU(1) << (LG_SIZEOF_PTR+3)) - opt_lg_chunk)=2^6-21=43，21 是 chunk 大小 2M 的后 21 位全 0，所以不用作为 key，真正作为 key 的是前 64-21=43 位 chunk 的地址。

```
/* Only the most significant bits of keys passed to rtree_[gs]et() are used. */
bool
rtree_new(rtree_t *rtree, unsigned bits, rtree_node_alloc_t *alloc,
    rtree_node_dalloc_t *dalloc)
{
        unsigned bits_in_leaf, height, i;

        assert(RTREE_HEIGHT_MAX == ((ZU(1) << (LG_SIZEOF_PTR+3)) /
            RTREE_BITS_PER_LEVEL));
        assert(bits > 0 && bits <= (sizeof(uintptr_t) << 3));

        bits_in_leaf = (bits % RTREE_BITS_PER_LEVEL) == 0 ?
RTREE_BITS_PER_LEVEL: (bits % RTREE_BITS_PER_LEVEL);
bits_in_leaf=43%16==0?16:(43%16)  bits_in_leaf =11
        if (bits > bits_in_leaf) {
```

```
        height = 1 + (bits - bits_in_leaf) / RTREE_BITS_PER_LEVEL;
        height=1+(43-11)/16=3
        if ((height-1) * RTREE_BITS_PER_LEVEL + bits_in_leaf != bits)
                height++;
} else
        height = 1;
assert((height-1) * RTREE_BITS_PER_LEVEL + bits_in_leaf == bits);

rtree->alloc = alloc;
rtree->dalloc = dalloc;
rtree->height = height;

/* Root level. */
rtree->levels[0].subtree = NULL;
rtree->levels[0].bits = (height > 1) ? RTREE_BITS_PER_LEVEL :
   bits_in_leaf;//=16
rtree->levels[0].cumbits = rtree->levels[0].bits;
/* Interior levels. */
for (i = 1; i < height-1; i++) {
        rtree->levels[i].subtree = NULL;
        rtree->levels[i].bits = RTREE_BITS_PER_LEVEL;//=16
        rtree->levels[i].cumbits = rtree->levels[i-1].cumbits +
           RTREE_BITS_PER_LEVEL;//=32
}
/* Leaf level. */
if (height > 1) {
        rtree->levels[height-1].subtree = NULL;
        rtree->levels[height-1].bits = bits_in_leaf;//=11
        rtree->levels[height-1].cumbits = bits;//=43
}

/* Compute lookup table to be used by rtree_start_level(). */
for (i = 0; i < RTREE_HEIGHT_MAX; i++) {
        rtree->start_level[i] = hmin(RTREE_HEIGHT_MAX - 1 - i, height -
           1);//2,2,1,0
}
```

```
	return (false);
}

JEMALLOC_ALWAYS_INLINE rtree_node_elm_t *
rtree_subtree_read(rtree_t *rtree, unsigned level, bool dependent)
{
	rtree_node_elm_t *subtree;

	subtree = rtree_subtree_tryread(rtree, level, dependent);
	if (!dependent && unlikely(!rtree_node_valid(subtree)))
		subtree = rtree_subtree_read_hard(rtree, level);
	assert(!dependent || subtree != NULL);
	return (subtree);
}
```

得到 start_level 的 subtree，如果没有，则初始化 2^bits 个 subtree 地址为首地址的 rtree_node_elm_t *指针。

```
JEMALLOC_INLINE bool
rtree_set(rtree_t *rtree, uintptr_t key, const extent_node_t *val)
{
	uintptr_t subkey;
	unsigned i, start_level;
	rtree_node_elm_t *node, *child;

	start_level = rtree_start_level(rtree, key);
```

//由 key 的最高位 1 的位置，得到下面开始的 start_level，再由这个得到 node 的地址

```
	node = rtree_subtree_read(rtree, start_level, false);
	if (node == NULL)
		return (true);
	for (i = start_level; /**/; i++, node = child) {
		subkey = rtree_subkey(rtree, key, i);
		if (i == rtree->height - 1) {
			/*
			 * node is a leaf, so it contains values rather than
			 * child pointers.
```

```
                        */
                        //如果已经是叶子节点，则把 val 值存入对应的 subkey 的位置
                        rtree_val_write(rtree, &node[subkey], val);
                        return (false);
                }
                assert(i + 1 < rtree->height);
//如果还不是叶子节点，由 subkey 得到下一 level 的 node 地址
                child = rtree_child_read(rtree, &node[subkey], i, false);
                if (child == NULL)
                        return (true);
        }
        not_reached();
}


JEMALLOC_ALWAYS_INLINE rtree_node_elm_t *
rtree_child_read(rtree_t *rtree, rtree_node_elm_t *elm, unsigned level,
    bool dependent)
{
        rtree_node_elm_t *child;
        child = rtree_child_tryread(elm, dependent);
        if (!dependent && unlikely(!rtree_node_valid(child)))
                child = rtree_child_read_hard(rtree, elm, level);
        assert(!dependent || child != NULL);
        return (child);
}
```

读取 child 值，如果 child 为空，则创建该 level 对应数量的 rtree_node_elm_t *指针。

2.3.4. radix tree gdb 数据

```
(gdb) p je_chunks_rtree
$294 = {alloc = 0x6f89b7950c <chunks_rtree_node_alloc>, dalloc = 0x0,
  height = 3, start_level = {2, 2, 1, 0}, levels = {
{{subtree_pun = 0x0,subtree = 0x0}, bits = 16, cumbits = 16},
{{subtree_pun = 0x6f8940bc00, subtree = 0x6f8940bc00}, bits = 16,cumbits = 32},
{{subtree_pun = 0x0, subtree = 0x0}, bits = 11,cumbits = 43},
```

{{subtree_pun = 0x0, subtree = 0x0}, bits = 0,cumbits = 0}
}}

查看 level=1 下的所有的 subtree 值，这里只有一个，在 subkey=111 的位置：

```
(gdb) p *je_chunks_rtree->levels[1]->subtree@65536
$300 = {{{pun = 0x0, child = 0x0, val = 0x0}} <repeats 111 times>, {{
    pun = 0x6f8948bc00, child = 0x6f8948bc00, val = 0x6f8948bc00}}, {{
    pun = 0x0, child = 0x0, val = 0x0}} <repeats 65424 times>}
```

查看 subkey=111 的指向的 child 的元素，总共有 2048 个，有效的实际存储为 7 个。

```
(gdb) p *je_chunks_rtree->levels[1]->subtree[111]->child@2048
$313 = {{{pun = 0x0, child = 0x0, val = 0x0}} <repeats 1060 times>, {{
    pun = 0x6f88328000, child = 0x6f88328000, val = 0x6f88328000}}, {{
    pun = 0x0, child = 0x0, val = 0x0}}, {{pun = 0x0, child = 0x0,
    val = 0x0}}, {{pun = 0x0, child = 0x0, val = 0x0}}, {{pun = 0x0,
    child = 0x0, val = 0x0}}, {{pun = 0x6f8829e380, child = 0x6f8829e380,
    val = 0x6f8829e380}}, {{pun = 0x0, child = 0x0, val = 0x0}}, {{
    pun = 0x0, child = 0x0, val = 0x0}}, {{pun = 0x0, child = 0x0,
    val = 0x0}}, {{pun = 0x6f88234500, child = 0x6f88234500,
    val = 0x6f88234500}}, {{pun = 0x0, child = 0x0, val = 0x0}}, {{
    pun = 0x0, child = 0x0, val = 0x0}}, {{pun = 0x0, child = 0x0,
    val = 0x0}}, {{pun = 0x6f8829e680, child = 0x6f8829e680,
    val = 0x6f8829e680}}, {{pun = 0x0, child = 0x0,
    val = 0x0}} <repeats 11 times>, {{pun = 0x6f87a00000,
    child = 0x6f87a00000, val = 0x6f87a00000}}, {{pun = 0x0, child = 0x0,
    val = 0x0}}, {{pun = 0x0, child = 0x0, val = 0x0}}, {{pun = 0x0,
    child = 0x0, val = 0x0}}, {{pun = 0x6f88200000, child = 0x6f88200000,
    val = 0x6f88200000}}, {{pun = 0x0, child = 0x0, val = 0x0}}, {{
    pun = 0x0, child = 0x0, val = 0x0}}, {{pun = 0x0, child = 0x0,
    val = 0x0}}, {{pun = 0x0, child = 0x0, val = 0x0}}, {{pun = 0x0,
    child = 0x0, val = 0x0}}, {{pun = 0x0, child = 0x0, val = 0x0}}, {{
    pun = 0x0, child = 0x0, val = 0x0}}, {{pun = 0x6f89200000,
    child = 0x6f89200000, val = 0x6f89200000}}, {{pun = 0x0, child = 0x0,
    val = 0x0}} <repeats 950 times>}
```

这个时候查看系统的 chunk 数量和地址：

(gdb) p (*je_arenas[0])->achunks
$314 = {qlh_first = 0x6f89200000}
(gdb) p (*je_arenas[0])->achunks->qlh_first
$315 = (extent_node_t *) 0x6f89200000
(gdb) p (*je_arenas[0])->achunks->qlh_first->ql_link.qre_next
$316 = (extent_node_t *) 0x6f87a00000
<chunks->qlh_first->ql_link.qre_next->ql_link.qre_next
$317 = (extent_node_t *) 0x6f89200000

(gdb) p (*je_arenas[1])->achunks
$319 = {qlh_first = 0x6f88200000}
(gdb) p (*je_arenas[1])->achunks->qlh_first->ql_link.qre_next
$320 = (extent_node_t *) 0x6f88200000

再查看 huge 的 chunk 数量和地址：
(gdb) p (*je_arenas[1])->huge->qlh_first
$333 = (extent_node_t *) 0x6f8829e680
(gdb) p (*je_arenas[1])->huge->qlh_first->ql_link.qre_nex
There is no member named qre_nex.
(gdb) p (*je_arenas[1])->huge->qlh_first->ql_link.qre_next
$334 = (extent_node_t *) 0x6f88234500
(gdb) p (*je_arenas[1])->huge->qlh_first->ql_link.qre_next->ql_link.qre_next
$335 = (extent_node_t *) 0x6f8829e380
<uge->qlh_first->ql_link.qre_next->ql_link.qre_next->ql_link.qre_next
$336 = (extent_node_t *) 0x6f88328000
<l_link.qre_next->ql_link.qre_next->ql_link.qre_next->ql_link.qre_next
$337 = (extent_node_t *) 0x6f8829e680
刚好是上面看到的对应的 7 个 chunk 地址，另外还可以看出来，huge 的 extent_node_t * 地址因为不能复用 chunk 的起始地址，所以不是 5 个 0 结尾的，而系统分配的其他 chunk 地址的 extent_node_t *则是 5 个 0 结尾的，也就是 chunk 的首地址。

2.4. red-black tree 红黑树

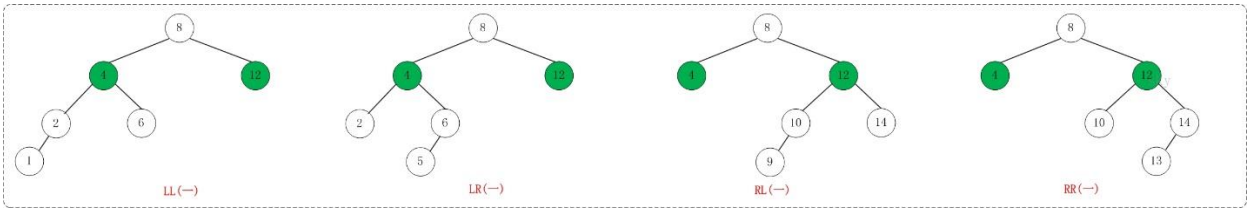
* cpp macro implementation of left-leaning 2-3 red-black trees. Parent
 * pointers are not used, and color bits are stored in the least significant

* bit of right-child pointers (if RB_COMPACT is defined), thus making node
* linkage as compact as is possible for red-black trees.

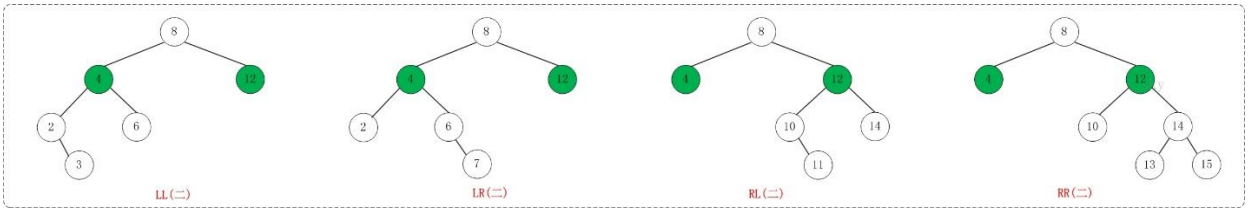
在 AVL 树中任何节点的两个儿子子树的高度最大差别为一，查找、插入和删除在平均和最坏情况下都是 O（lg n）。但因为增加和删除节点可能会破坏“平衡状态”，所以大多数情况下需要通过多次树旋转来重新平衡这个树。所以简单地说，如果你的应用查找次数远远多于增删操作，那么 AVL 是最好的，但是如果增删次数和查找次数不相上下时，RBT 因为相比 AVL 没有过多的旋转操作，效率要比 AVL 高。并且在是实际情况中，RBT 的应用也更为广泛。

2.4.1. AVL 失去平衡后的 4 种姿态

如果在 AVL 树中进行插入或删除节点后，可能导致 AVL 树失去平衡。这种失去平衡的可以概括为 4 种姿态：LL(左左)，LR(左右)，RR(右右)和 RL(右左)。下面给出它们的示意图：



上图中的 4 棵树都是"失去平衡的 AVL 树"，从左往右的情况依次是：LL、LR、RL、RR。除了上面的情况之外，还有其它的失去平衡的 AVL 树，如下图：



上面的两张图都是为了便于理解，而列举的关于"失去平衡的 AVL 树"的例子。总的来说，AVL 树失去平衡时的情况一定是 LL、LR、RL、RR 这 4 种之一，它们都由各自的定义：

(1) **LL**：LeftLeft，也称为"左左"。插入或删除一个节点后，根节点的左子树的左子树还有非空子节点，导致"根的左子树的高度"比"根的右子树的高度"大 2，导致 AVL 树失去了平衡。

例如，在上面 LL 情况中，由于"根节点(8)的左子树(4)的左子树(2)还有非空子节点",

而"根节点(8)的右子树(12)没有子节点"；导致"根节点(8)的左子树(4)高度"比"根节点(8)的右子树(12)"高2。

(2) **LR**：LeftRight，也称为"左右"。插入或删除一个节点后，根节点的左子树的右子树还有非空子节点，导致"根的左子树的高度"比"根的右子树的高度"大 2，导致 AVL 树失去了平衡。

例如，在上面LR 情况中，由于"根节点(8)的左子树(4)的左子树(6)还有非空子节点"，而"根节点(8)的右子树(12)没有子节点"；导致"根节点(8)的左子树(4)高度"比"根节点(8)的右子树(12)"高2。

(3) **RL**：RightLeft，称为"右左"。插入或删除一个节点后，根节点的右子树的左子树还有非空子节点，导致"根的右子树的高度"比"根的左子树的高度"大 2，导致 AVL 树失去了平衡。

例如，在上面RL 情况中，由于"根节点(8)的右子树(12)的左子树(10)还有非空子节点"，而"根节点(8)的左子树(4)没有子节点"；导致"根节点(8)的右子树(12)高度"比"根节点(8)的左子树(4)"高2。

(4) **RR**：RightRight，称为"右右"。插入或删除一个节点后，根节点的右子树的右子树还有非空子节点，导致"根的右子树的高度"比"根的左子树的高度"大 2，导致 AVL 树失去了平衡。

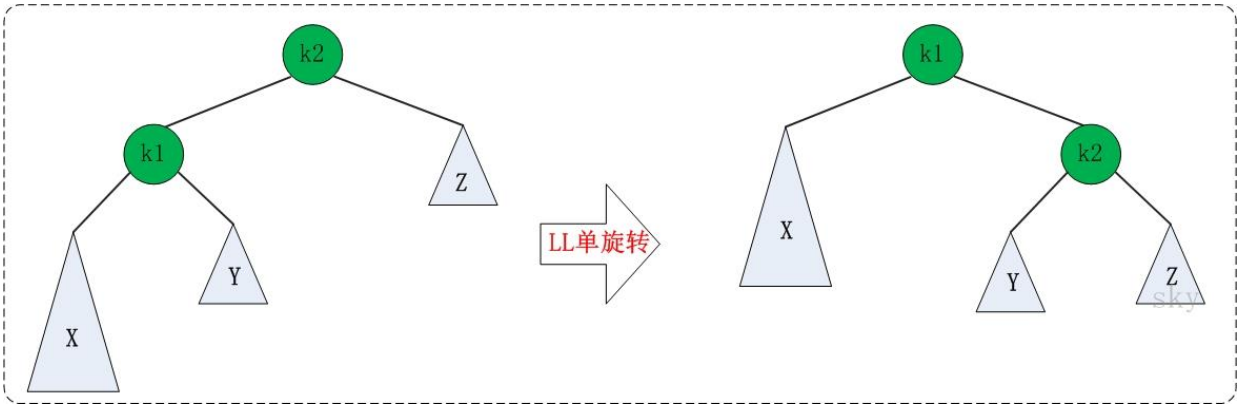
例如，在上面RR 情况中，由于"根节点(8)的右子树(12)的右子树(14)还有非空子节点"，而"根节点(8)的左子树(4)没有子节点"；导致"根节点(8)的右子树(12)高度"比"根节点(8)的左子树(4)"高2。

前面说过，如果在 AVL 树中进行插入或删除节点后，可能导致 AVL 树失去平衡。AVL 失去平衡之后，可以通过旋转使其恢复平衡，下面分别介绍"LL(左左)，LR(左右)，RR(右右)和 RL(右左)"这 4 种情况对应的旋转方法。

2.4.2. AVL 旋转

LL 的旋转

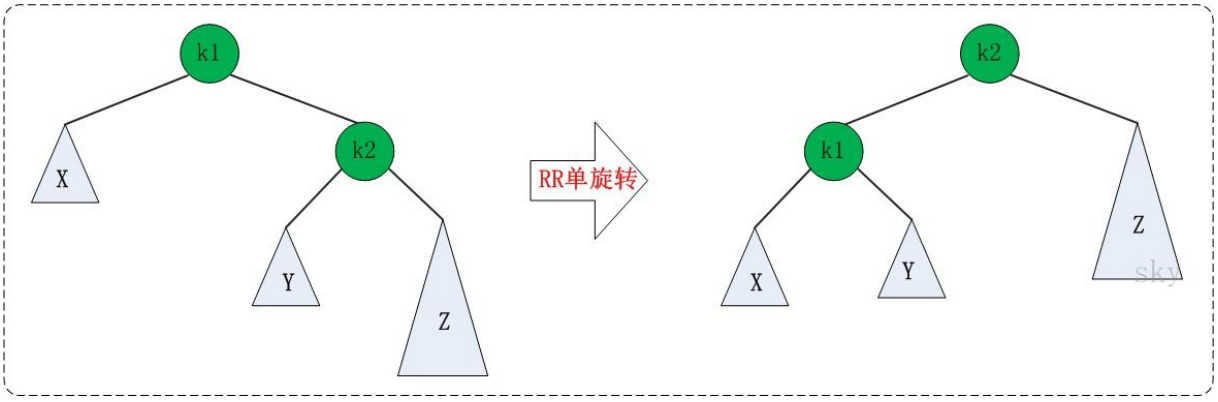
LL 失去平衡的情况，可以通过一次旋转让 AVL 树恢复平衡。如下图：



图中左边是旋转之前的树，右边是旋转之后的树。从中可以发现，旋转之后的树又变成了 AVL 树，而且该旋转只需要一次即可完成。
对于 LL 旋转，你可以这样理解为：LL 旋转是围绕"失去平衡的 AVL 根节点"进行的，也就是节点 k2；而且由于是 LL 情况，即左左情况，就用手抓着"左孩子，即 k1"使劲摇。将 k1 变成根节点，k2 变成 k1 的右子树，"k1 的右子树"变成"k2 的左子树"。

RR 的旋转

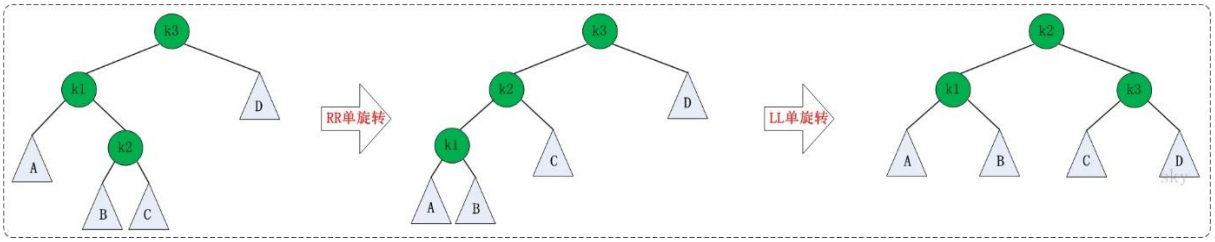
理解了 LL 之后，RR 就相当容易理解了。RR 是与 LL 对称的情况！RR 恢复平衡的旋转方法如下：



图中左边是旋转之前的树，右边是旋转之后的树。RR 旋转也只需要一次即可完成。

LR 的旋转

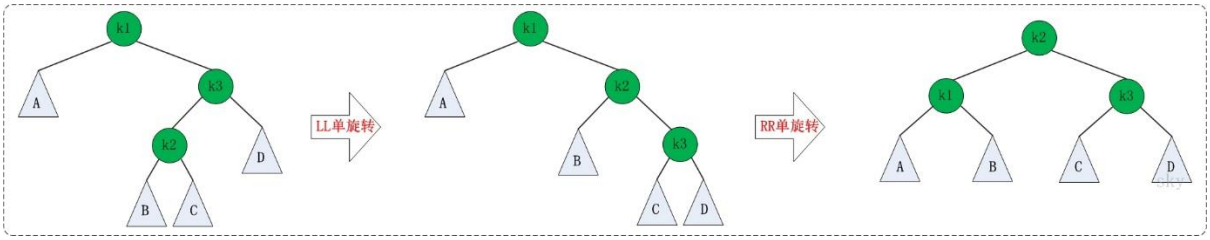
LR 失去平衡的情况，需要经过两次旋转才能让 AVL 树恢复平衡。如下图：



第一次旋转是围绕"k1"进行的"RR 旋转"，第二次是围绕"k3"进行的"LL 旋转"。

RL 的旋转

RL 是与 LR 的对称情况！RL 恢复平衡的旋转方法如下：



第一次旋转是围绕"k3"进行的"LL 旋转"，第二次是围绕"k1"进行的"RR 旋转"。

2.4.3. RB TREE 定义

和 AVL 树一样，红黑树也是一种自平衡二叉排序树，其定义如下：

（1）节点有且只有两种颜色，红色和黑色。

（2）根节点和叶子节点必须是黑色，其中，叶子节点是虚拟存在的空节点（NULL）。

（3）红色节点的两个子节点必须是黑色。

（4）任意节点到叶子节点的路径上，必须包含相同数目的黑色节点。

从红黑树的定义可以发现，任意节点左右子树的高度差在一倍之内（最长路径为节点红黑相间，最短路径为节点全黑）。

由于红黑树对平衡性的要求没有 AVL 树高，因此频繁插入和删除节点时，触发平衡调整的次数更少，平衡调整的过程也更易收敛。

2.4.4. RB TREE 插入过程

新插入节点为红色，当往上回溯，如果碰到一个黑色的父辈节点，则完成。

```
a_attr void                                                    \
a_prefix##insert(a_rbt_type *rbtree, a_type *node) {                \
    struct {                                                   \
```

```
        a_type *node;                                                   \
        int cmp;                                                        \
    } path[sizeof(void *) << 4], *pathp;                                \
    rbt_node_new(a_type, a_field, rbtree, node);                        \
    /* Wind. */                                                         \
    path->node = rbtree->rbt_root;                                      \
    //这里把插入点的路径节点放入 path 数组，类似于一个栈的数组，第一个是树的根节点，最后一个是需要插入的 node 自己，并且记录当前节点和 node 节点大小比较值，放入 cmp 中，为后续使用做准备。
    for (pathp = path; pathp->node != NULL; pathp++) {                  \
        int cmp = pathp->cmp = a_cmp(node, pathp->node);                \
        assert(cmp != 0);                                               \
        if (cmp < 0) {                                                  \
            pathp[1].node = rbtn_left_get(a_type, a_field,              \
              pathp->node);                                             \
        } else {                                                        \
            pathp[1].node = rbtn_right_get(a_type, a_field,             \
              pathp->node);                                             \
        }                                                               \
    }                                                                   \
    pathp->node = node;                                                 \
    /* Unwind. */                                                       \
    //放入完成后，从最后面开始，处理放入路径的节点，每次取栈顶两个节点进行处理。
    for (pathp--; (uintptr_t)pathp >= (uintptr_t)path; pathp--) {       \
        a_type *cnode = pathp->node;//当前栈顶第二个节点                \
        if (pathp->cmp < 0) {                                           \
            a_type *left = pathp[1].node;//路径经过左侧，当前栈顶第一个节点 \
            rbtn_left_set(a_type, a_field, cnode, left);//重新连接，可能有新的子树根节点生成 \
            if (rbtn_red_get(a_type, a_field, left)) {//如果当前 top/left 为红 \
                a_type *leftleft = rbtn_left_get(a_type, a_field, left);\
                if (leftleft != NULL && rbtn_red_get(a_type, a_field,   \
                  leftleft)) {                                          \
                    /* Fix up 4-node. */                                \
                    a_type *tnode;                                      \
                    rbtn_black_set(a_type, a_field, leftleft);//置 leftleft 为黑 \
```

```
                rbtn_rotate_right(a_type, a_field, cnode, tnode);//右旋        \
                cnode = tnode;//把新的子树 root，即 left/top 作为下一轮栈顶        \
//这里调整解决的是 leftleft 和 left 都是红色的问题，把 leftleft 置为黑色，然后把 left 右旋，
这样 left 成为了新的子树的根，也就是新的下一次待处理的栈顶。处理前 cnode 到 leftleft
有一个黑色节点，它自己，处理后 leftleft 变为了黑色的节点，cnode 跑到新子树的右边，
所以对比新子树和原子树，黑色节点个数没有变化，同时调整了两个红色节点相连的问题。
变化如下图所示：
        |
     cnode(b)
    /       \
  left(r)     a
  /
leftleft(r)

         |
       left(r)
     /         \
leftleft(b)   cnode(b)
                \
                 a

                 }                                                  \
            } else {                                                \
                 return;//如果栈顶为黑色，结束返回                  \
            }                                                       \
          } else {                                                  \
            a_type *right = pathp[1].node; //路径经过右侧，当前栈顶第一个节点       \
            rbtn_right_set(a_type, a_field, cnode, right); //重新连接，可能有新的子树根节
点生成                    \
            if (rbtn_red_get(a_type, a_field, right)) {//如果当前 top/right 为红   \
                 a_type *left = rbtn_left_get(a_type, a_field, cnode);      \
                 if (left != NULL && rbtn_red_get(a_type, a_field,          \
                   left)) {                    \
                   /* Split 4-node. */                                  \
                   //把左右两个节点都置黑色，父节点置红色
                   rbtn_black_set(a_type, a_field, left);               \
```

```
                rbtn_black_set(a_type, a_field, right);          \
                rbtn_red_set(a_type, a_field, cnode);            \
//该算法的终结条件是栈顶元素为黑色，这个分支是当前栈顶是红色，它的左兄弟节点也是红色，这里直接把两个兄弟节点置黑色，把 cnode 置红色，继续处理下一个栈元素。
            } else {                                             \
                /* Lean left. */                                 \
                a_type *tnode;                                   \
                bool tred = rbtn_red_get(a_type, a_field, cnode);//保持当前节点颜色\
                rbtn_rotate_left(a_type, a_field, cnode, tnode);//左旋        \
                rbtn_color_set(a_type, a_field, tnode, tred);//置新的根节点颜色为原根节点 cnode 的颜色     \
                rbtn_red_set(a_type, a_field, cnode);//cnode 作为左孩子，置为红 \
                cnode = tnode;//用新的根节点作为下一次的栈顶节点        \
//这个分支是当前栈顶红色，它的左兄弟不存在或者为黑色，把 cnode 左旋，栈顶元素成为新的子树根节点，所以最后需要设 cnode=tnode，把新的子树根节点置为原来的 conde 的颜色，conde 置为红色，也就是原来栈顶的颜色。这样处理可能为下一次的调整做准备，同时又往根节点方向推进了一步。
            }                                                    \
        } else {                                                 \
            return;//如果栈顶为黑色，结束返回                    \
        }                                                        \
    }                                                            \
    pathp->node = cnode;//保存新的栈顶                           \
}                                                                \
/* Set root, and make it black. */                               \
rbtree->rbt_root = path->node;                                   \
rbtn_black_set(a_type, a_field, rbtree->rbt_root);               \
}
```

根据当前插入节点在父节点的左侧或者右侧分为两种情况讨论：

插入的红色节点在左边时：

1） 如果父节点为黑色，直接完成；

2） 如果父节点为红色，在下一轮循环时就出现了左和左左都是红色的情况，这个时候左左变黑，左成为新的根节点，左的父节点一定是黑，要不然原来的子树不符合 rb 树规则。调整后左变为新的根节点，还是红色，左左变为新的左，为黑色。原

来的黑色的根节点成为新的右节点，黑色。因为新的根节点变为了红色，所以继续向上回溯，直到找到黑色的节点为止。

插入的红色节点在右边时：

3） 如果父节点的左孩子不存在或者为黑色，且父节点为黑色，左旋，使得新节点成为新的父节点，设为原父节点的颜色黑色。原来的父节点成为左孩子，设为红色，因为新的父节点为黑色，下一轮循环时就会退出。

4） 如果父节点的左孩子不存在或者为黑色，且父节点为红色，左旋，使得新节点成为新的父节点，设为原父节点的颜色红色。原来的父节点成为左孩子，设为红色，进入下一个循环，下一个循环的时候就会出现左是红色，左左也是红色的情况，和 2）的下一轮循环一样处理。

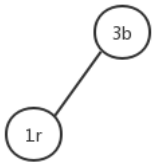
5） 如果父节点的左孩子存在，且为红色，这个时候设置左右孩子为黑色，父节点为红色，因为父节点颜色发生了变化，所以需要继续往根回溯，直到遇到黑色节点为止。（这里需要继续回溯的原因是，担心因为父节点变为红色后，出现红-红的情况。）

以下是 key 为 3，1，5，4，2 的插入过程的例子：

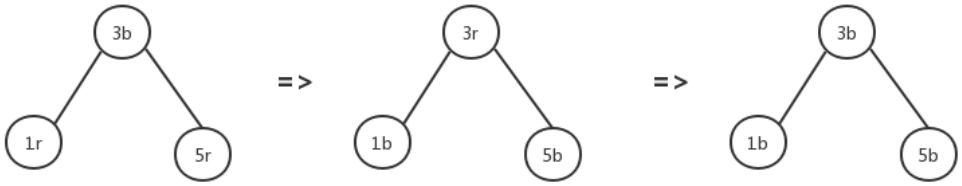
RB tree insert

1)

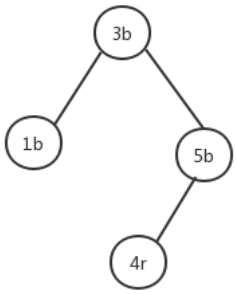
2)



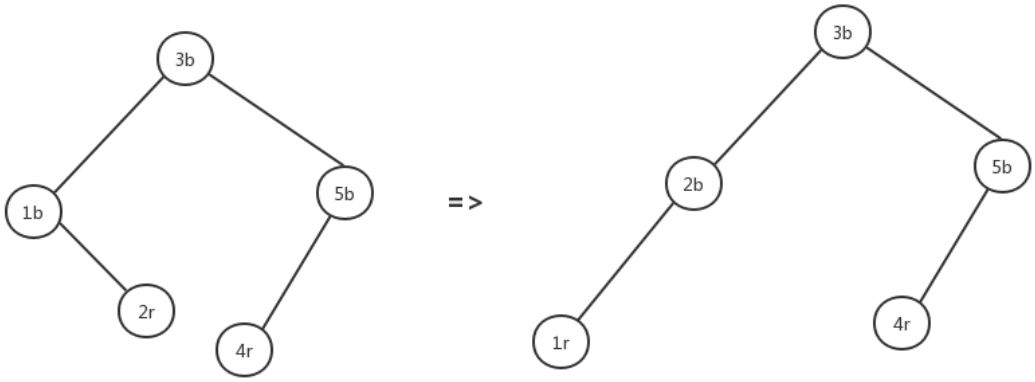
3)



4)



5)



2.4.5. RB TREE 删除过程

删除过程和插入相反，如果删除的是红色节点，因为不影响平衡，所以直接完成。如果是黑色节点，那么必须回溯恢复平衡。如果子树平衡恢复，可以直接终止回溯，如果子树还是小一个黑色节点，需要继续向上回溯。

```
a_attr void                                                     \
a_prefix##remove(a_rbt_type *rbtree, a_type *node) {                \
    struct {                                                    \
        a_type *node;                                           \
        int cmp;                                                \
    } *pathp, *nodep, path[sizeof(void *) << 4];                \
    /* Wind. */                                                 \
    nodep = NULL; /* Silence compiler warning. */               \
    path->node = rbtree->rbt_root;                              \
    for (pathp = path; pathp->node != NULL; pathp++) {          \
        int cmp = pathp->cmp = a_cmp(node, pathp->node);        \
        if (cmp < 0) {                                          \
            pathp[1].node = rbtn_left_get(a_type, a_field,      \
              pathp->node);                                     \
        } else {                                                \
            pathp[1].node = rbtn_right_get(a_type, a_field,     \
              pathp->node);                                     \
            if (cmp == 0) {                                     \
                /* Find node's successor, in preparation for swap. */   \
//把 node 节点的后继保存起来，为后续处理做准备
                pathp->cmp = 1;                                 \
                nodep = pathp;                                  \
                for (pathp++; pathp->node != NULL;              \
                  pathp++) {                                    \
                    pathp->cmp = -1;                            \
                    pathp[1].node = rbtn_left_get(a_type, a_field,      \
                      pathp->node);                             \
                }                                               \
                break;                                          \
            }                                                   \
        }                                                       \
```

```
    }                                                                   \
    assert(nodep->node == node);                                        \
    pathp--;                                                            \
    if (pathp->node != node) {                                              \
        /* Swap node with its successor. */                             \
//用最后一个最靠近的 node 的后继节点代替 node
//保存替换节点的颜色
        bool tred = rbtn_red_get(a_type, a_field, pathp->node);             \
//把替换节点着色为要删除节点的颜色
        rbtn_color_set(a_type, a_field, pathp->node,                    \
          rbtn_red_get(a_type, a_field, node));                         \
//设置替换节点的左孩子为删除节点的左孩子
        rbtn_left_set(a_type, a_field, pathp->node,                     \
          rbtn_left_get(a_type, a_field, node));                        \
        /* If node's successor is its right child, the following code */\
        /* will do the wrong thing for the right child pointer.       */\
        /* However, it doesn't matter, because the pointer will be    */\
        /* properly set when the successor is pruned.                 */\
//设置替换节点的右孩子为删除节点的右孩子，如果替换节点就是删除节点的右孩子，这会造成替换节点的右孩子其实是指向自己，等后续删除节点删除后不会有影响。
        rbtn_right_set(a_type, a_field, pathp->node,                    \
          rbtn_right_get(a_type, a_field, node));                       \
//把删除节点设置为替换节点的颜色
        rbtn_color_set(a_type, a_field, node, tred);                    \
        /* The pruned leaf node's child pointers are never accessed   */\
        /* again, so don't bother setting them to nil.                */\
//交换 path 栈的替换节点和删除节点
        nodep->node = pathp->node;                                      \
        pathp->node = node;                                             \
//如果删除节点是根节点，需要置新的根节点
        if (nodep == path) {                                            \
            rbtree->rbt_root = nodep->node;                             \
        } else {                                                        \
//设置父节点到替换节点（原来的删除节点被替换了）的链接
            if (nodep[-1].cmp < 0) {                                    \
                rbtn_left_set(a_type, a_field, nodep[-1].node,          \
```

```
                nodep->node);                                   \
            } else {                                            \
                rbtn_right_set(a_type, a_field, nodep[-1].node, \
                  nodep->node);                                 \
            }                                                   \
        }                                                       \
    } else {                                                    \
    //如果删除节点就是叶子节点
        a_type *left = rbtn_left_get(a_type, a_field, node);    \
        if (left != NULL) {                                     \
            /* node has no successor, but it has a left child.       */\
            /* Splice node out, without losing the left child.       */\
            //没有右孩子，所以一定是黑色的
            assert(!rbtn_red_get(a_type, a_field, node));       \
            //没有右孩子，如果 left 也是黑色，那么 node 节点左右子树的黑色节点数不相等，所以 left 一定红色的。
            assert(rbtn_red_get(a_type, a_field, left));        \
//直接把左孩子者黑色，用左孩子替换删除节点
            rbtn_black_set(a_type, a_field, left);              \
            if (pathp == path) {                                \
                rbtree->rbt_root = left;                        \
            } else {                                            \
//设置父节点到替换节点（原来的删除节点被替换了）的链接
                if (pathp[-1].cmp < 0) {                        \
                    rbtn_left_set(a_type, a_field, pathp[-1].node, \
                      left);                                    \
                } else {                                        \
                    rbtn_right_set(a_type, a_field, pathp[-1].node, \
                      left);                                    \
                }                                               \
            }                                                   \
            return;                                             \
        } else if (pathp == path) {                             \
//如果删除节点是根节点，而且这个节点没有后继，左孩子也没有，所以是唯一的节点。
            /* The tree only contained one node. */             \
            rbtree->rbt_root = NULL;                            \
```

```
            return;                                                 \
        }                                                           \
    }                                                               \
    if (rbtn_red_get(a_type, a_field, pathp->node)) {                   \
        /* Prune red node, which requires no fixup. */              \
//这里 pathp->node 就是需要删除的节点，但是他的颜色已经替换为替换它的节点的颜色，他在 RB Tree 里的位置也被移除，在 path 栈的位置被替换到了最顶端。
//这里 assert 是指没有右节点为红色的 RB 树，只有左节点为红色的 RB 树，如果右节点为红色，插入的时候如果左节点为红色，则需要同时变为黑色；如果左节点为黑色或者为空，则需要左旋，所以这里确保红色的节点只出现在左孩子这边。所以它的父节点的cmp<0。
        assert(pathp[-1].cmp < 0);                                  \
        rbtn_left_set(a_type, a_field, pathp[-1].node, NULL);           \
        return;                                                 \
    }                                                               \
    /* The node to be pruned is black, so unwind until balance is     */\
    /* restored.                                                  */\
    //开始回溯，先把栈顶节点置空，表示后续子树为空，为回溯做准备。
    pathp->node = NULL;                                             \
    for (pathp--; (uintptr_t)pathp >= (uintptr_t)path; pathp--) {   \
        assert(pathp->cmp != 0);                                    \
        if (pathp->cmp < 0) {                                           \
//删除黑色节点子树出现在左子树时
//重置左子树，左子树根节点可能有调整。
            rbtn_left_set(a_type, a_field, pathp->node,                 \
              pathp[1].node);                                       \
            if (rbtn_red_get(a_type, a_field, pathp->node)) {           \
//如果当前节点为红色时
                a_type *right = rbtn_right_get(a_type, a_field,         \
                  pathp->node);                                     \
                a_type *rightleft = rbtn_left_get(a_type, a_field, \
                  right);                                           \
                a_type *tnode;                                      \
                if (rightleft != NULL && rbtn_red_get(a_type, a_field,  \
                  rightleft)) {                                     \
//如果当前节点为红色时，right 应该是黑色，如果 rightleft 也是红色时
```

```
                /* In the following diagrams, ||, //, and \\      */\
                /* indicate the path to the removed node.         */\
                /*                                                */\
                /*      ||                                        */\
                /*    pathp(r)                                    */\
                /*  //       \                                    */\
                /* (b)        (b)                                 */\
                /*           /                                    */\
                /*          (r)                                   */\
                /*                                                */\
```

//通过右旋和左旋，使得新的左子树黑色节点增加一，右子树黑色节点保持不变，达到平衡。

```
                rbtn_black_set(a_type, a_field, pathp->node);          \
                rbtn_rotate_right(a_type, a_field, right, tnode);      \
                rbtn_right_set(a_type, a_field, pathp->node, tnode);\
                rbtn_rotate_left(a_type, a_field, pathp->node,         \
                  tnode);                                              \
              } else {                                                 \
```

//如果当前节点为红色时，right 应该是黑色，如果 rightleft 是黑色时

```
                /*      ||                                        */\
                /*    pathp(r)                                    */\
                /*  //       \                                    */\
                /* (b)        (b)                                 */\
                /*           /                                    */\
                /*          (b)                                   */\
                /*                                                */\
```

//直接左旋，使得新子树的左子树黑色节点数增加一，新子树的右子树黑色节点数保持不变。达到平衡。新子树的根节点发生变化。

```
                rbtn_rotate_left(a_type, a_field, pathp->node,         \
                  tnode);                                              \
              }                                                        \
              /* Balance restored, but rotation modified subtree    */\
              /* root.                                              */\
```

//因为当前节点为红色，所以肯定不是树的根节点

```
              assert((uintptr_t)pathp > (uintptr_t)path);              \
              if (pathp[-1].cmp < 0) {                                 \
```

```
            rbtn_left_set(a_type, a_field, pathp[-1].node, \
              tnode);                                      \
            } else {                                       \
              rbtn_right_set(a_type, a_field, pathp[-1].node,      \
              tnode);                                      \
            }                                              \
            return;                                        \
          } else {                                         \
//如果当前节点为黑色时
            a_type *right = rbtn_right_get(a_type, a_field,      \
              pathp->node);                                \
            a_type *rightleft = rbtn_left_get(a_type, a_field, \
              right);                                      \
            if (rightleft != NULL && rbtn_red_get(a_type, a_field,   \
              rightleft)) {                                \
//如果当前节点为黑色时，right 应该是黑色，如果 rightleft 是红色时
              /*      ||                                */\
              /*    pathp(b)                              */\
              /*  //       \                             */\
              /* (b)        (b)                           */\
              /*           /                              */\
              /*          (r)                             */\
              a_type *tnode;                               \
//把 rightleft 着黑色，通过右旋 right 和左旋当前节点，使得新子树的左子树黑色节点数增加一，新子树的右子树黑色节点数不变，达到平衡。
              rbtn_black_set(a_type, a_field, rightleft);          \
              rbtn_rotate_right(a_type, a_field, right, tnode);    \
              rbtn_right_set(a_type, a_field, pathp->node, tnode);\
              rbtn_rotate_left(a_type, a_field, pathp->node,       \
                tnode);                                    \
              /* Balance restored, but rotation modified       */\
              /* subtree root, which may actually be the tree   */\
              /* root.                                         */\
              if (pathp == path) {                         \
//如果当前节点是根节点，重置新的根节点
                /* Set root. */                            \
```

```
                    rbtree->rbt_root = tnode;                   \
                } else {                                        \
//重置父节点到新子树的根节点的连接
                    if (pathp[-1].cmp < 0) {                    \
                        rbtn_left_set(a_type, a_field,          \
                          pathp[-1].node, tnode);               \
                    } else {                                    \
                        rbtn_right_set(a_type, a_field,         \
                          pathp[-1].node, tnode);               \
                    }                                           \
                }                                               \
                return;                                         \
            } else {                                            \
//如果当前节点为黑色时，right 应该是黑色，如果 rightleft 是黑色时
                /*      ||                                  */\
                /*    pathp(b)                                */\
                /*  //      \                                 */\
                /* (b)       (b)                              */\
                /*          /                                 */\
                /*         (b)                                */\
                a_type *tnode;                                  \
//通过着当前节点为红色，左旋当前节点，使得新的右子树的黑色节点数也减少一，新的左子树的黑色节点数不变。这样新子树的黑色节点数量还是少一，所以这种情况平衡没调整完成，需要继续往上回溯。
                rbtn_red_set(a_type, a_field, pathp->node);     \
                rbtn_rotate_left(a_type, a_field, pathp->node,  \
                  tnode);                                       \
                pathp->node = tnode;                            \
            }                                                   \
        }                                                       \
    } else {                                                    \
//if(pathp->cmp>=0) 删除黑色节点的子树出现在右子树时
        a_type *left;                                           \
//重置右子树，右子树根节点可能有调整；第一次循环的时候解决了前面提到的指向自己的问题，这里会把右子树置空。
        rbtn_right_set(a_type, a_field, pathp->node,            \
```

```
            pathp[1].node);                                         \
//因为右子树有删除黑色节点，所以原来是平衡的 RB Tree 的话，左子树必不为空，并且也有相应数量的黑色节点存在。
            left = rbtn_left_get(a_type, a_field, pathp->node);     \
            if (rbtn_red_get(a_type, a_field, left)) {              \
//当前节点的左孩子为红色时
                a_type *tnode;                                      \
                a_type *leftright = rbtn_right_get(a_type, a_field, \
                  left);                                            \
                a_type *leftrightleft = rbtn_left_get(a_type, a_field, \
                  leftright);                                       \
                if (leftrightleft != NULL && rbtn_red_get(a_type, \
                  a_field, leftrightleft)) {                        \
//当前节点的左孩子为红色时，左右为黑色，左右左为红色时
                    /*      ||                                  */\
                    /*    pathp(b)                                */\
                    /*   /       \\                               */\
                    /* (r)        (b)                             */\
                    /*   \                                        */\
                    /*   (b)                                      */\
                    /*   /                                        */\
                    /* (r)                                        */\
                    a_type *unode;                                  \
//经过右右左三次旋转，使得新子树的右子树增加一个黑色节点，新子树的左子树保持黑色节点数目不变。达到平衡。
                    rbtn_black_set(a_type, a_field, leftrightleft); \
                    rbtn_rotate_right(a_type, a_field, pathp->node, \
                      unode);                                       \
                    rbtn_rotate_right(a_type, a_field, pathp->node, \
                      tnode);                                       \
                    rbtn_right_set(a_type, a_field, unode, tnode);  \
                    rbtn_rotate_left(a_type, a_field, unode, tnode); \
                } else {                                            \
//当前节点的左孩子为红色时，左右为黑色，左右左为黑色时
                    /*      ||                                  */\
                    /*    pathp(b)                                */\
```

```
                /*   /       \\                    */\
                /* (r)       (b)                    */\
                /*   \                              */\
                /*   (b)                            */\
                /*   /                              */\
                /* (b)                              */\
//通过右旋，并且给新的根节点着黑色，使得右子树黑色节点增加一，同时给 leftright 着红色，使得原子树根节点右旋后的左子树的黑色节点数也是少一个，和右子树平衡。
                    assert(leftright != NULL);                          \
                    rbtn_red_set(a_type, a_field, leftright);           \
                    rbtn_rotate_right(a_type, a_field, pathp->node,     \
                      tnode);                                           \
                    rbtn_black_set(a_type, a_field, tnode);             \
                }                                                       \
                /* Balance restored, but rotation modified subtree    */\
                /* root, which may actually be the tree root.         */\
                if (pathp == path) {                                    \
//如果是根节点，设置新的子树根节点为树的根节点
                    /* Set root. */                                     \
                    rbtree->rbt_root = tnode;                           \
                } else {                                                \
//重新置父节点到新子树的连接
                    if (pathp[-1].cmp < 0) {                            \
                        rbtn_left_set(a_type, a_field, pathp[-1].node,  \
                          tnode);                                       \
                    } else {                                            \
                        rbtn_right_set(a_type, a_field, pathp[-1].node, \
                          tnode);                                       \
                    }                                                   \
                }                                                       \
                return;                                                 \
            } else if (rbtn_red_get(a_type, a_field, pathp->node)) {    \
//当前节点为红色，左孩子为黑色时；
                a_type *leftleft = rbtn_left_get(a_type, a_field, left);\
                if (leftleft != NULL && rbtn_red_get(a_type, a_field,   \
                  leftleft)) {                                          \
```

```
//当前节点为红色，左孩子为黑色时，且当前节点的左左为红色时
                /*       ||                          */\
                /*     pathp(r)                        */\
                /*    /       \\                       */\
                /*  (b)       (b)                      */\
                /*  /                                  */\
                /* (r)                                 */\
                a_type *tnode;                                  \
//着当前节点为黑色
                rbtn_black_set(a_type, a_field, pathp->node);   \
//着左节点为红色
                rbtn_red_set(a_type, a_field, left);            \
//着左左节点为黑色
                rbtn_black_set(a_type, a_field, leftleft);      \
//右旋当前节点为根的子树，右旋使得右子树黑色节点数量增加一，恢复平衡
                rbtn_rotate_right(a_type, a_field, pathp->node, \
                  tnode);                                       \
                /* Balance restored, but rotation modified      */\
                /* subtree root.                        */\
//平衡恢复，但是根节点发生了变化，需要调整根节点，因为当前节点为红色，所以肯定不是根节点。
                assert((uintptr_t)pathp > (uintptr_t)path);      \
                if (pathp[-1].cmp < 0) {                        \
                        rbtn_left_set(a_type, a_field, pathp[-1].node,  \
                          tnode);                               \
                } else {                                        \
                        rbtn_right_set(a_type, a_field, pathp[-1].node, \
                          tnode);                               \
                }                                               \
                return;                                         \
        } else {                                                \
//当前节点为红色，左孩子为黑色时，且当前节点的左左为黑色，或者为空时
                /*       ||                          */\
                /*     pathp(r)                        */\
                /*    /       \\                       */\
                /*  (b)       (b)                      */\
```

```
                /*   /                                  */\
                /* (b)                                   */\
//把左孩子着红色，把当前节点着黑色。这样当前节点的左子树黑色也减一，保持了平衡，
当前节点所在的子树因为根节点变黑，所以黑色节点数量不变。恢复了平衡，直接 return。
                rbtn_red_set(a_type, a_field, left);                \
                rbtn_black_set(a_type, a_field, pathp->node);       \
                /* Balance restored. */                             \
                return;                                             \
            }                                                       \
        } else {                                                    \
//当前节点为黑色，左孩子为黑色时
            a_type *leftleft = rbtn_left_get(a_type, a_field, left);\
            if (leftleft != NULL && rbtn_red_get(a_type, a_field,   \
              leftleft)) {                                          \
//当前节点为黑色，左孩子为黑色时，且当前节点的左左为红色时

                /*               ||                               */\
                /*             pathp(b)                           */\
                /*            /        \\                         */\
                /*          (b)        (b)                        */\
                /*          /                                     */\
                /*        (r)                                     */\
                a_type *tnode;                                      \
//左左着黑色，为右旋增加一个黑色节点做好准备
                rbtn_black_set(a_type, a_field, leftleft);          \
//右旋当前节点为根的子树，右旋使得右子树黑色节点数量增加一，恢复平衡，子树的根
节点发生了变化
                rbtn_rotate_right(a_type, a_field, pathp->node,     \
                  tnode);                                           \
                /* Balance restored, but rotation modified        */\
                /* subtree root, which may actually be the tree   */\
                /* root.                                          */\
//因为当前节点是黑色，所以有可能是根节点
                if (pathp == path) {                                \
                    /* Set root. */                                 \
//如果是根节点，设置新的子树根节点为树的根节点
```

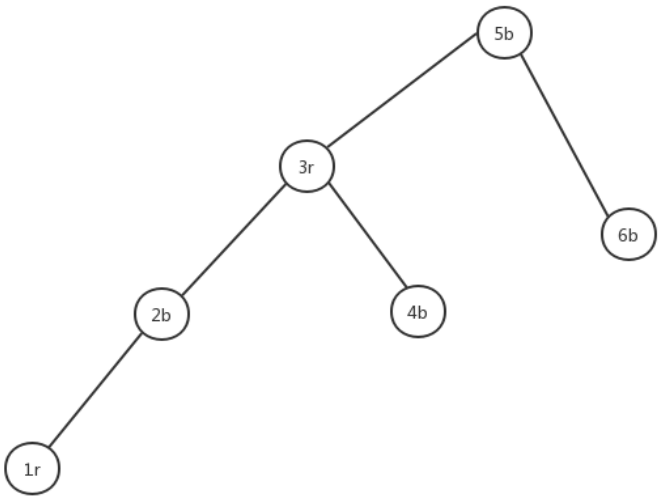
```
                rbtree->rbt_root = tnode;                       \
            } else {                                            \
//重置父节点到新的子树根节点的连接，因为当前节点不再是子树根节点。
                if (pathp[-1].cmp < 0) {                        \
                    rbtn_left_set(a_type, a_field,              \
                      pathp[-1].node, tnode);                   \
                } else {                                        \
                    rbtn_right_set(a_type, a_field,             \
                      pathp[-1].node, tnode);                   \
                }                                               \
            }                                                   \
            return;                                             \
        } else {                                                \
//当前节点为黑色，左孩子为黑色时，且当前节点的左左为黑色时
            /*            ||                        */\
            /*          pathp(b)                      */\
            /*          /      \\                     */\
            /*        (b)      (b)                    */\
            /*        /                               */\
            /*      (b)                               */\
//直接把左孩子着红色，使得左子树也减少一个黑色节点。这种情况需要向上继续回溯，
因为子树的黑色节点还是减少了一。
            rbtn_red_set(a_type, a_field, left);            \
        }                                                   \
    }                                                       \
}                                                           \
/* Set root. */                                             \
rbtree->rbt_root = path->node;                              \
assert(!rbtn_red_get(a_type, a_field, rbtree->rbt_root));           \
}
```

画图举例具体的删除过程。

RB tree delete

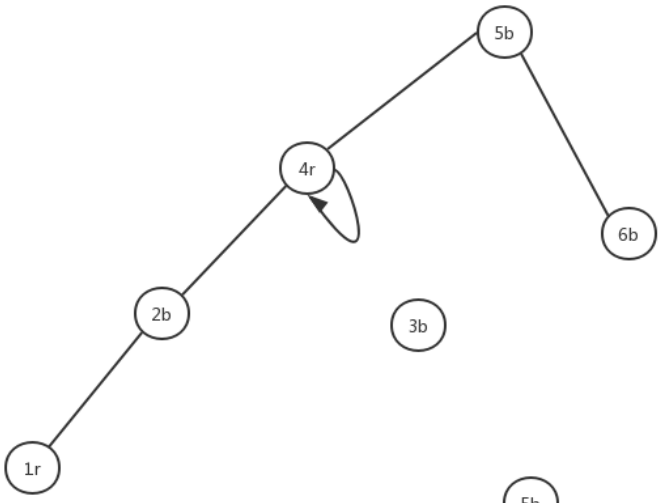
delete node 3 3

1)



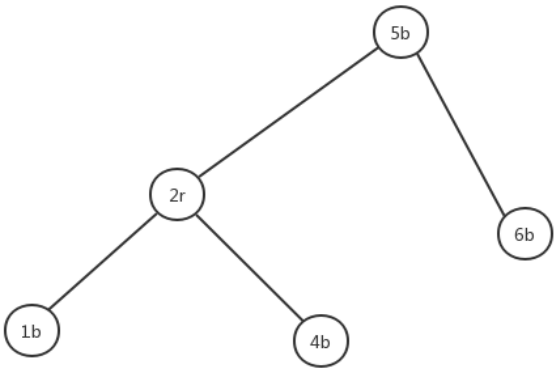
	NULL	
pathp	4	-1
nodep	3	1
	5	-1

2)



NULL	
3	-1
4	1
5	-1

3)



NULL	-1
4	1
5	-1

2.5. Ring definitions 双向循环链表

实现在 qr.h。

```
#define	qr(a_type)
struct {                                                \
	a_type	*qre_next;                              \
	a_type	*qre_prev;                              \
}

#define	qr_new(a_qr, a_field)
#define	qr_next(a_qr, a_field)
#define	qr_prev(a_qr, a_field)
#define	qr_before_insert(a_qrelm, a_qr, a_field)
#define	qr_after_insert(a_qrelm, a_qr, a_field)
#define	qr_meld(a_qr_a, a_qr_b, a_field)
#define	qr_split(a_qr_a, a_qr_b, a_field)= qr_meld
#define	qr_remove(a_qr, a_field)
#define	qr_foreach(var, a_qr, a_field)
#define	qr_reverse_foreach(var, a_qr, a_field)
```

2.5.1. Ring 在 jemalloc 中的使用

```
qr(extent_node_t)	cc_link;
struct {                                                \
	extent_node_t	*qre_next;                      \
	extent_node_t	*qre_prev;                      \
} cc_link;
```

2.6. List definitions 双向链表

List 基于 Ring，定义了一个头结点，所以也有了相应的尾结点。实现在 ql.h。

```
#define	ql_head(a_type)                                 \
struct {                                                \
```

```
        a_type *qlh_first;                                      \
}

#define     ql_elm(a_type)      qr(a_type)
#define     ql_new(a_head)
#define     ql_elm_new(a_elm, a_field) qr_new((a_elm), a_field)
#define     ql_first(a_head) ((a_head)->qlh_first)
#define     ql_last(a_head, a_field)
#define     ql_next(a_head, a_elm, a_field)
#define     ql_prev(a_head, a_elm, a_field)
#define     ql_before_insert(a_head, a_qlelm, a_elm, a_field)
#define     ql_after_insert(a_qlelm, a_elm, a_field)
#define     ql_head_insert(a_head, a_elm, a_field)
#define     ql_tail_insert(a_head, a_elm, a_field)
#define     ql_remove(a_head, a_elm, a_field)
#define     ql_head_remove(a_head, a_type, a_field)
#define     ql_tail_remove(a_head, a_type, a_field)
#define     ql_foreach(a_var, a_head, a_field)
#define     ql_reverse_foreach(a_var, a_head, a_field)
```

2.6.1. List definitions 在 jemalloc 中的使用

```
/* Extant arena chunks. */
ql_head(extent_node_t)     achunks;

#define     ql_head(a_type)                                     \
struct {                                                        \
        a_type *qlh_first;                                      \
}

struct {                                                        \
        extent_node_t *qlh_first;                               \
} achunks;
```

这种方法并没有定义一个结构，而是定义了一个 achunks 的结构变量，编译器会为 achunks 分配内存。

```
/* Linkage for arena's achunks, huge, and node_cache lists. */
ql_elm(extent_node_t)       ql_link;

struct {                                                    \
        extent_node_t       *qre_next;                      \
        extent_node_t       *qre_prev;                      \
} ql_link;
```

2.7. PRNG 线性同余伪随机数生成器和原子操作的使用

2.7.1. 原子操作的使用

所谓的原子操作，取的就是“原子是最小的、不可分割的最小个体”的意义，它表示在多个线程访问同一个全局资源的时候，能够确保所有其他的线程都不在同一时间内访问相同的资源。也就是他确保了在同一时刻只有唯一的线程对这个资源进行访问。这有点类似互斥对象对共享资源的访问的保护，但是原子操作更加接近底层，因而效率更高。

在以往的 C++标准中并没有对原子操作进行规定，我们往往是使用汇编语言，或者是借助第三方的线程库，例如 intel 的 pthread 来实现。在新标准 C++11，引入了原子操作的概念，并通过这个新的头文件提供了多种原子操作数据类型，例如，atomic_bool,atomic_int 等等，如果我们在多个线程中对这些类型的共享资源进行操作，编译器将保证这些操作都是原子性的，也就是说，确保任意时刻只有一个线程对这个资源进行访问，编译器将保证，多个线程访问这个共享资源的正确性。从而避免了锁的使用，提高了效率。

2.7.2. 线性同余伪随机数生成器

```
/*
 * Simple linear congruential pseudo-random number generator:
 *
 *   prng(y) = (a*x + c) % m
 *
```

```
* where the following constants ensure maximal period:
*
*   a == Odd number (relatively prime to 2^n), and (a-1) is a multiple of 4.
*   c == Odd number (relatively prime to 2^n).
*   m == 2^32
*
* See Knuth's TAOCP 3rd Ed., Vol. 2, pg. 17 for details on these constraints.
*
* This choice of m has the disadvantage that the quality of the bits is
* proportional to bit position.  For example, the lowest bit has a cycle of 2,
* the next has a cycle of 4, etc.  For this reason, we prefer to use the upper
* bits.
*/
```

低位的重复周期远远高于高位，所以这里取高位的值作为随机数值。

c≠0

When c≠0, correctly chosen parameters allow a period equal to m, for all seed values. This will occur if and only if:

m and the offset，c are relatively prime,
a-1 is divisible by all prime factors of m,
a-1 is divisible by 4 if m is divisible by 4.
These three requirements are referred to as the Hull–Dobell Theorem.

This form may be used with any m. A power‐of‐2 modulus equal to a computer's word size is convenient and achieves the longest possible period (for any odd c and a≡1 mod 4), but has the same problem mentioned above: only the most significant bit achieves the full period. Lower–order bit k repeats with a period of length 2^{k+1}.

The following table lists the parameters of LCGs in common use, including built–in rand() functions in runtime libraries of various compilers.

Source	modulus *m*	multiplier *a*	increment *c*	output bits of seed in *rand()* or *Random(L)*

Numerical Recipes	2^{32}	1664525	1013904223	
Borland C/C++	2^{32}	22695477	1	bits 30..16 in *rand()*, 30..0 in *lrand()*
glibc (used by GCC)[9]	2^{31}	1103515245	12345	bits 30..0
ANSI C: Watcom, Digital Mars, CodeWarrior, IBM VisualAge C/C++ [10] C99, C11: Suggestion in the ISO/IEC 9899 [11]	2^{31}	1103515245	12345	bits 30..16
Borland Delphi, Virtual Pascal	2^{32}	134775813	1	bits 63..32 of *(seed * L)*
Turbo Pascal	2^{32}	134775813 ($0x8088405_{16}$)	1	
Microsoft Visual/Quick C/C++	2^{32}	214013 ($343FD_{16}$)	2531011 ($269EC3_{16}$)	bits 30..16
Microsoft Visual Basic (6 and earlier)[12]	2^{24}	1140671485 ($43FD43FD_{16}$)	12820163 ($C39EC3_{16}$)	
RtlUniform from Native API[13]	$2^{31} - 1$	2147483629 ($7FFFFFED_{16}$)	2147483587 ($7FFFFFC3_{16}$)	
Apple CarbonLib, C++11's `minstd_rand0` [14]	$2^{31} - 1$	16807	0	see MINSTD
C++11's `minstd_rand` [14]	$2^{31} - 1$	48271	0	see MINSTD

MMIX by Donald Knuth	2^{64}	6364136223846793005	1442695040888963407	
Newlib, Musl	2^{64}	6364136223846793005	1	bits 63...32
VMS's **MTH$RANDOM**,[15] old versions of glibc	2^{32}	69069 ($10DCD_{16}$)	1	
Java's java.util.Random, POSIX [ln]rand48, glibc [ln]rand48[_r]	2^{48}	25214903917 ($5DEECE66D_{16}$)	11	bits 47...16
`random0` [16][17][18][19][20] If X is even then $X+1$ will be odd, and vice versa—the lowest bit oscillates at each step.	$134456 = 2^{37} \cdot 5$	8121	28411	
POSIX[21] [jm]rand48, glibc [mj]rand48[_r]	2^{48}	25214903917 ($5DEECE66D_{16}$)	11	bits 47...15
POSIX [de]rand48, glibc [de]rand48[_r]	2^{48}	25214903917 ($5DEECE66D_{16}$)	11	bits 47...0
cc65[22]	2^{23}	65793 (10101_{16})	4282663 (415927_{16})	bits 22...8
cc65	2^{32}	16843009 (1010101_{16})	826366247 (31415927_{16})	bits 31...16
Formerly common: RANDU [4]	2^{31}	65539	0	

```
#define     PRNG_A_32 UINT32_C(1103515241)
#define     PRNG_C_32 UINT32_C(12347)

#define     PRNG_A_64 UINT64_C(6364136223846793005)
#define     PRNG_C_64 UINT64_C(1442695040888963407)
```

这里 32 位的值修正过，64 位的直接使用的。只有在 arena_malloc_large 一个地方使用。

prng_lg_range_zu(size_t *state, unsigned lg_range, bool atomic)
返回高 lg_range 位

3. Tcache 实现原理

3.1. TSD:thread specific data 线程局部存储

```
pthread_setspecific(a_name##tsd_tsd,   (void *)wrapper))
a_name##tsd_wrapper_t *wrapper = (a_name##tsd_wrapper_t *)    \
        pthread_getspecific(a_name##tsd_tsd);
pthread_key_create(&a_name##tsd_tsd, a_name##tsd_cleanup_wrapper)

a_name##tsd_wrapper_t *wrapper;                                    \
wrapper=(a_name##tsd_wrapper_t                                                *)
malloc_tsd_malloc(sizeof(a_name##tsd_wrapper_t));
```

tcache_get(tsd_t *tsd, bool create)，
会先查找 tcache，如果不存在，绑定一个 arena，再创建 tcache_create(tsd_tsdn(tsd), arena) tcache。

```
tsd_t *tsd,
typedef struct tsd_s tsd_t;
struct tsd_s {
        tsd_state_t    state;
#define        O(n, t)                                    \
        t              n;
MALLOC_TSD
#undef O
};

#define        O(n, t)                                    \
t       *tsd_##n##p_get(tsd_t *tsd);                      \
t       tsd_##n##_get(tsd_t *tsd);                 \
void    tsd_##n##_set(tsd_t *tsd, t n);
```

MALLOC_TSD
#undef O
这里定义了如下两个函数：
tcache_t * tsd_tcache_get(tsd_t *tsd);
void tsd_tcache_set(tsd_t *tsd, tcache_t * tcache);

```
#define MALLOC_TSD                                  \
/*  O(name,                 type) */                \
    O(tcache,               tcache_t *)             \
    O(thread_allocated,             uint64_t)       \
    O(thread_deallocated,   uint64_t)               \
    O(prof_tdata,           prof_tdata_t *)         \
    O(iarena,               arena_t *)              \
    O(arena,                arena_t *)              \
    O(arenas_tdata,         arena_t *)              \
    O(narenas_tdata,        unsigned)               \
    O(arenas_tdata_bypass,          bool)           \
    O(tcache_enabled,               tcache_enabled_t)\
    O(quarantine,           quarantine_t *)         \
    O(witnesses,            witness_list_t)         \
    O(witness_fork,         bool)                   \

#define TSD_INITIALIZER {                           \
    tsd_state_uninitialized,                        \
    NULL,                                           \
    0,                                              \
    0,                                              \
    NULL,                                           \
    NULL,                                           \
    NULL,                                           \
    NULL,                                           \
    0,                                              \
    false,                                          \
    tcache_enabled_default,                         \
    NULL,                                           \
```

```
    ql_head_initializer(witnesses),                                 \
    false                                                           \
}

typedef enum {
        tsd_state_uninitialized,
        tsd_state_nominal,
        tsd_state_purgatory,
        tsd_state_reincarnated
} tsd_state_t;

static const tsd_t tsd_initializer = TSD_INITIALIZER;
malloc_tsd_types(, tsd_t)
malloc_tsd_funcs(JEMALLOC_ALWAYS_INLINE, , tsd_t, tsd_initializer, tsd_cleanup)

#define         malloc_tsd_types(a_name, a_type)                                \
typedef struct {                                                        \
        bool    initialized;                                            \
        a_type          val;                                                    \
} a_name##tsd_wrapper_t;
#endif


wrapper->val = a_initializer; // TSD_INITIALIZER;
```

3.2. Tcache 和 arena 的关系

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

arena 中的每个 thread 有一个 tcache。

```
(gdb) p *je_arenas@3
$53 = {0x7ce5c00140, 0x7ce5c8fc00, 0x0}

(gdb) p *je_arenas@3
$7 = {0x7145a00140, 0x7145a8fc00, 0x0}
(gdb) p (*je_arenas[0])->tcache_ql
$8 = {qlh_first = 0x714580e000}
(gdb)
$9 = {qlh_first = 0x714580e000}
(gdb) p (*je_arenas[0])->nthreads
$10 = {8, 8}

(gdb) p ((tcache_t *)0x714580e000)->link.qre_next
$11 = (tcache_t *) 0x7145811800
(gdb) p ((tcache_t *)0x7145811800)->link.qre_next
$12 = (tcache_t *) 0x7144079800
(gdb) p ((tcache_t *)0x7144079800)->link.qre_next
$13 = (tcache_t *) 0x7144076000
(gdb) p ((tcache_t *)0x7144076000)->link.qre_next
$14 = (tcache_t *) 0x7144188000
(gdb) p ((tcache_t *)0x7144188000)->link.qre_next
$15 = (tcache_t *) 0x714418b800
(gdb) p ((tcache_t *)0x714418b800)->link.qre_next
$16 = (tcache_t *) 0x71441d9000
(gdb) p ((tcache_t *)0x71441d9000)->link.qre_next
$17 = (tcache_t *) 0x71441dc800
(gdb) p ((tcache_t *)0x71441dc800)->link.qre_next
$18 = (tcache_t *) 0x714580e000
```

3.3. Tcache 的定义

```
/*
 * Number of tcache bins.  There are NBINS=36 small-object bins, plus 0 or more
```

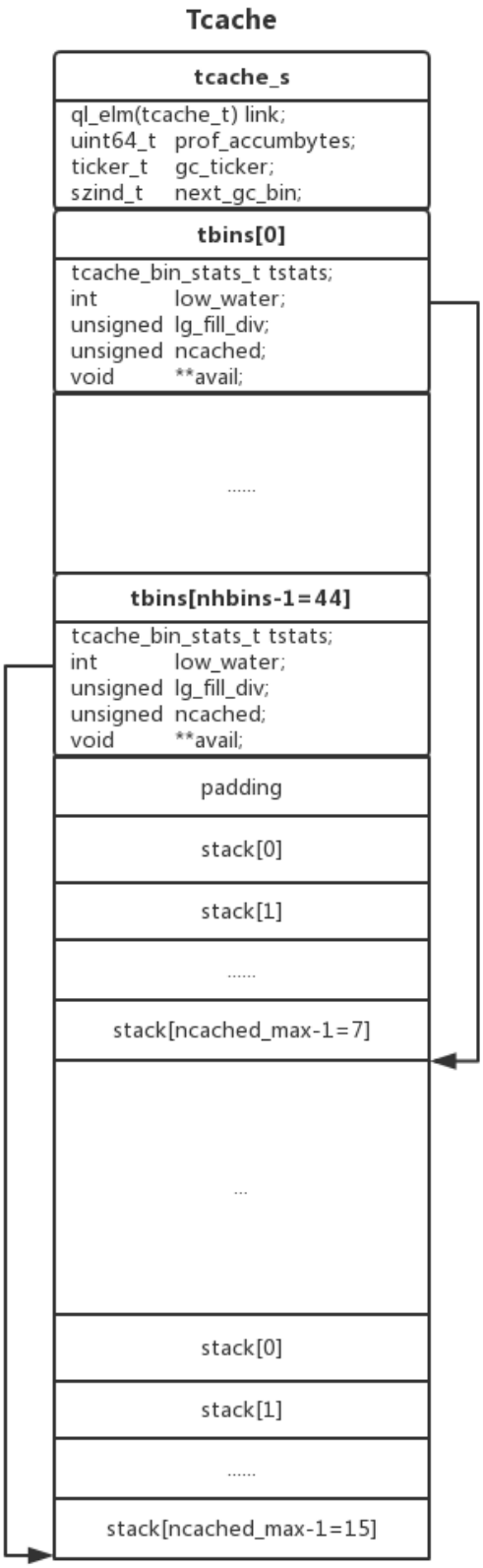
```
 * large-object bins.
 */
extern unsigned     nhbins;//总共有 45 个 bins，small bin 用掉 36 个，还有 9 个是 large 用的。
```

这个值的计算如下，首先 android 定义了这个值，DANDROID_LG_TCACHE_MAXCLASS_DEFAULT=16，默认是 15，android 定义的是 16，16 时，算得的 tcache_maxclass=64k，如果 15，tcache_maxclass=32k，然后根据这个值来计算 nhbins = size2index(tcache_maxclass) + 1=41;得到 tcache 的 bin 数量。所以默认有 5 个 large 被 cache。详细 size2index 的计算参考 size2index 的计算过程。

```
struct tcache_bin_s {
        tcache_bin_stats_t tstats;
        int             low_water;    /* Min # cached since last GC. */
        unsigned        lg_fill_div;  /* Fill (ncached_max >> lg_fill_div). */
        unsigned        ncached;      /* # of cached objects. */
        /*
         * To make use of adjacent cacheline prefetch, the items in the avail
         * stack goes to higher address for newer allocations.  avail points
         * just above the available space, which means that
         * avail[-ncached, ... -1] are available items and the lowest item will
         * be allocated first.
         */
        void            **avail;      /* Stack of available objects. */
};

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

```
        */
};
```

3.4. Tcache 的结构


Tcache
tcache_s
ql_elm(tcache_t) link;
uint64_t prof_accumbytes;
ticker_t gc_ticker;
szind_t next_gc_bin;
tbins[0]
tcache_bin_stats_t tstats;
int low_water;
unsigned lg_fill_div;
unsigned ncached;
void **avail;
.....
tbins[nhbins-1=44]
tcache_bin_stats_t tstats;
int low_water;
unsigned lg_fill_div;
unsigned ncached;
void **avail;
padding
stack[0]
stack[1]
.....
stack[ncached_max-1=7]
...
stack[0]
stack[1]
.....
stack[ncached_max-1=15]

avail 指向下一个 stack 的开始位置，也就是 stack[ncached_max-1]的下一个元素。所以后续插入的时候用*(tbin->avail - nfill + i) = ptr，如果 nfill=4，则从 avail[-4]，avail[-3]，avail[-2]，avail[-1]顺序放置，-1 刚好是第一个位置。

```
/*
* To make use of adjacent cacheline prefetch, the items in the avail
* stack goes to higher address for newer allocations.  avail points
* just above the available space, which means that
* avail[-ncached, ... -1] are available items and the lowest item will
* be allocated first.
*/

/*
 * avail points past the available space.  Allocations will
 * access the slots toward higher addresses (for the benefit of
 * prefetch).
*/
tcache->tbins[i].avail = (void **)((uintptr_t)tcache +(uintptr_t)stack_offset);
```

avail 越过了可用的空间，指到了下一个 stack 的开始位置，分配是从低往高地址进行，主要是考虑了 prefetch 的好处。

3.5. Tcache boot 与初始化

当前 45 个 tbin 的 ncached_max 值：

```
(gdb) p *je_tcache_bin_info@45
$7 = {
  {ncached_max = 8} <repeats 16 times>, {ncached_max = 20}, {
   ncached_max = 8}, {ncached_max = 8}, {ncached_max = 8}, {
   ncached_max = 20}, {ncached_max = 8}, {ncached_max = 20}, {
   ncached_max = 8}, {ncached_max = 20} <repeats 12 times>, {

   ncached_max = 16}, {ncached_max = 16}, {ncached_max = 16}, {
   ncached_max = 16}, {ncached_max = 16}, {ncached_max = 16}, {
   ncached_max = 16}, {ncached_max = 16}, {ncached_max = 16}}
```

Tcache 的栈元素总共为：stack_nelms=21×8+20×15+16×9=612

#define SMALL_MAXCLASS ((((size_t)1) << 13) + (((size_t)3) << 11))
SMALL_MAXCLASS=14k=14336
当 size 区间在 SMALL_MAXCLASS=14336<size <= tcache_maxclass=65536，

3.6. Tcache fill 过程

填充函数：arena_tcache_fill_small，对于 large 没有 fill 的过程。
lg_fill_div 初始化是 1，在填充 cache 时，会填 tcache_bin_info[binind].ncached_max >> tbin->lg_fill_div 个，也就是说填充 ncached_max/2 的个数。

如果当前 bin 的 runcur 有可用的 region，直接调用 arena_run_reg_alloc 分配内存；如果不存在 runcur，或者当前 run 没有可用的 region，则调用 arena_bin_malloc_hard 分配一个新的可用 run 到 runcur，再调用 arena_run_reg_alloc 进行分配内存，下一次循环的时候就可以直接用 arena_run_reg_alloc 分配了。

填充的时候从栈顶开始，往下填充，如果填充到一半，没有把整个 nfill 填满，这个时候需要移动前面的填充信息，确保填充的位置是从栈底开始的。

3.7. Tcache 的分配过程

tcache_alloc_small
如果当前 cache 里有可用的缓存，直接调用 tcache_alloc_easy 分配，要不调用 tcache_alloc_small_hard，先进行填充，再进行缓存分配。
tcache_alloc_easy 不用加锁，因为是线程内。如果 tcache 分配完了，需要 arena_tcache_fill_small 去填充 tcache 时，这个时候需要 arena lock 的加锁保护。

```
tcache_alloc_easy(tcache_bin_t *tbin, bool *tcache_success)
{
        void *ret;
        if (unlikely(tbin->ncached == 0)) {
                tbin->low_water = -1;
                *tcache_success = false;
                return (NULL);
        }
        *tcache_success = true;
        ret = *(tbin->avail - tbin->ncached);
        tbin->ncached--;
```

```
        if (unlikely((int)tbin->ncached < tbin->low_water))
                tbin->low_water = tbin->ncached;

        return (ret);
}
```

如果 tbin->ncached =0，返回 tcache_success = false，会调用 tcache_alloc_small_hard 先进行填充，然后再进行 tcache_alloc_easy 分配。
如果 tbin->ncached！=0，直接取栈顶元素返回，tbin->ncached--，检查是否需要调整 tbin->low_water。

3.8. Tcache 的回收 flush 过程

tcache_bin_flush_small，在 GC 过程中可能会触发 flush 操作，还有在释放过程中，如果 cache 的数量达到了 ncached_max 值，也需要进行 flush 回收。
在释放过程中，每次总是挑选栈底 region 所在的 arena 的 region 先进行释放。如果非当前 arena 的 region，则先保存在栈底，在下一个循环中释放。
arena_decay_ticks 的作用？这个是用来清理 arena 层面的 arena->ndirty 的数量，也是一种回收机制。

```
(gdb) p (*je_arenas[0])->tcache_ql
$33 = {qlh_first = 0x6f8200e000}
(gdb) p ((tcache_t *)0x6f8200e000)
$34 = (tcache_t *) 0x6f8200e000
(gdb) p *((tcache_t *)0x6f8200e000)
$35 = {link = {qre_next = 0x6f82011800, qre_prev = 0x6f80b85c00},
  prof_accumbytes = 0, gc_ticker = {tick = 9, nticks = 228},
  next_gc_bin = 14, tbins = {{tstats = {nrequests = 6}, low_water = 0,
      lg_fill_div = 1, ncached = 2, avail = 0x6f8200e608}}}
(gdb) p ((tcache_t *)0x6f8200e000)->gc_ticker
$36 = {tick = 9, nticks = 228}
```

每一次 tcache_dalloc_large/tcache_dalloc_small/tcache_alloc_large/tcache_alloc_small 都会调用 tcache_event，然后做 tick-1 动作，直到 228 个初始值被减完，再恢复初始值 228，然后触发 tcache_event_hard(tsd, tcache);

tcache_event->tcache_event_hard

```
struct ticker_s {
        int32_t         tick;
        int32_t         nticks;
};
ticker_t        gc_ticker;      /* Drives incremental GC. */
szind_t                 next_gc_bin; /* Next bin to GC. */

/* Number of tcache allocation/deallocation events between incremental GCs. */
#define         TCACHE_GC_INCR                                                  \
   ((TCACHE_GC_SWEEP / NBINS) + ((TCACHE_GC_SWEEP / NBINS == 0) ? 0 : 1))
```

TCACHE_GC_SWEEP=8192，
NBINS=36
TCACHE_GC_INCR=228

```
tcache_event(tsd_t *tsd, tcache_t *tcache)
{
        if (TCACHE_GC_INCR == 0)
                return;
        if (unlikely(ticker_tick(&tcache->gc_ticker)))
                tcache_event_hard(tsd, tcache);
}
```

tcache_event_hard

```
int     low_water;    /* Min # cached since last GC. */
```

自上次 GC 以来该 bin 最低 cache 的数量。

```
unsigned      lg_fill_div;      /* Fill (ncached_max >> lg_fill_div). */
unsigned      ncached;      /* # of cached objects. */
```

1）如果 low_water > 0，则保留 3/4 low_water 的 cache。

```
/*
* Reduce fill count by 2X.  Limit lg_fill_div such that the
* fill count is always at least 1.
```

```
*/
if ((tbin_info->ncached_max >> (tbin->lg_fill_div+1)) >= 1)
        tbin->lg_fill_div++;
```

2）如果 low_water < 0，增加 cache 填充数为 2 倍。

```
/*
* Increase fill count by 2X.  Make sure lg_fill_div stays
* greater than 0.
*/
if (tbin->lg_fill_div > 1)
        tbin->lg_fill_div--;
```

设置 tbin->low_water = tbin->ncached; （另外只有在当 ncached 小于 low_water 时做调整。）所以要使 low_water>0，必须是一个 GC 周期内该 bin 还是没有分配完。也就是说这个 reg_size 的 bin 在当前线程中使用较少。所以会降低缓存数量。

再调整到下一个 gc bin，指向 tcache->next_gc_bin++;

lg_fill_div 初始化为 1，所以 low_water 在第一次 tcache_alloc_easy 分配的时候被初始化为-1，-1 不可能大于任何 tbin->ncached，所以直到第一次触发 GC，low_water 一直为-1，因为 lg_fill_div=1，所以第一次触发 GC 的时候只是修改了 low_water 值，为当前的 tbin->ncached。

当 tbin->ncached 达到 0 后，也就是分配完成后，会设置 tbin->low_water = -1; 然后才会有可能触发 tbin->lg_fill_div--;

所以这里的分配思想是如果在一个周期内使用的少，则只保留 3/4 的 low_water 的 cache，然后提高 lg_fill_div 的值，使得下次分配数量减少。相反，如果在一个周期内使用的较多，则会导致 ncached 达到 0，这样 low_water 会被置为-1，然后减少 lg_fill_div 的值，使得下次分配的量增加。（lg_fill_div 的最小值为 1）

如果 lg_fill_div 最小值为 1，那么每次填充只能填最大值 ncached_max 的一半，这样另一半空间是不是浪费？不浪费，因为可能释放回收，需要空间填充，如果达到 ncached_max 后，就需要从 tcache 释放了。

tcaches_create 这个没有调用到。
tcache 四个分配和回收函数全部做 inline 处理，减少函数调用开销。

3.9. Android 对 TCACHE_NSLOTS_SMALL_MAX 配置问题

对于 Android 的配置，

```
#define        TCACHE_NSLOTS_SMALL_MIN          20
#define        TCACHE_NSLOTS_SMALL_MAX
        ANDROID_TCACHE_NSLOTS_SMALL_MAX=8

        for (i = 0; i < NBINS; i++) {
                if ((arena_bin_info[i].nregs << 1) <= TCACHE_NSLOTS_SMALL_MIN) {
                        tcache_bin_info[i].ncached_max =
                           TCACHE_NSLOTS_SMALL_MIN;
                } else if ((arena_bin_info[i].nregs << 1) <=
                   TCACHE_NSLOTS_SMALL_MAX) {
                        tcache_bin_info[i].ncached_max =
                           (arena_bin_info[i].nregs << 1);
                } else {
                        tcache_bin_info[i].ncached_max =
                           TCACHE_NSLOTS_SMALL_MAX;
                }
                stack_nelms += tcache_bin_info[i].ncached_max;
        }
        for (; i < nhbins; i++) {
                tcache_bin_info[i].ncached_max = TCACHE_NSLOTS_LARGE;
                stack_nelms += tcache_bin_info[i].ncached_max;
        }
```

```
(gdb) p *je_tcache_bin_info@45
$7 = {
  {ncached_max = 8} <repeats 16 times>, {ncached_max = 20}, {
   ncached_max = 8}, {ncached_max = 8}, {ncached_max = 8}, {
   ncached_max = 20}, {ncached_max = 8}, {ncached_max = 20}, {
   ncached_max = 8}, {ncached_max = 20} <repeats 12 times>, {

   ncached_max = 16}, {ncached_max = 16}, {ncached_max = 16}, {
   ncached_max = 16}, {ncached_max = 16}, {ncached_max = 16}, {
```

ncached_max = 16}, {ncached_max = 16}, {ncached_max = 16}}

对于 Android 的配置，TCACHE_NSLOTS_SMALL_MIN> TCACHE_NSLOTS_SMALL_MAX 可能会造成不是期望的值的问题；其实中间的 else if ((arena_bin_info[i].nregs << 1) <= TCACHE_NSLOTS_SMALL_MAX)永远得不到执行，因为 TCACHE_NSLOTS_SMALL_MAX< TCACHE_NSLOTS_SMALL_MIN。所以造成越是 nregs 值很大的反而只有 8，nregs 值比 10 小的反而 ncached_max=20，这个应该是一个问题。

jemalloc 的默认 TCACHE_NSLOTS_SMALL_MIN=20，TCACHE_NSLOTS_SMALL_MAX=200，如果 2nregs<TCACHE_NSLOTS_SMALL_MIN=20, ncached_max=20，如果 2nregs< TCACHE_NSLOTS_SMALL_MAX=200，ncached_max=2nregs，要不，ncached_max=200，上限。Jemalloc 的设计的目的是对于 Tcache 的数量和一个 run 中的 2 倍 region 数量建立联系，也就是说如果一个 run 中的 region 越多，相应的 Tcache 的最大的 cache 数量也相应也大一点。

这样会导致 nregs>10 时，ncached_max=8，实际分配的 cache 数量只有 4 个，因为 lg_fill_div 的最小值为 1，而如果没有 android 的这个配置文件，这个值是在 20-200 之间，实际分配的数量是 10-100 之间，大大降低了缓冲数量，而这些主要集中在 reg_size 比较小的 bin 中，8，16, 32, 48, 64, 80, 96, 112,128,160,192,224,256,320,384,448…，这些 region 在实际分配中一般有大量的分配，所以造成了一定的不合理性。

Android 这么配置的目的是？是因为线程很多，而且分配内存可能没有这么高的需求？所以限制了 ncached_max 值比较小。经过测试吗？

4. Jemalloc 的参数和调试相关

4.1. Jemalloc 的调试开关

4.1.1. JEMALLOC_DEBUG

If possible, really consider using the excellent valgrind tool for memory debugging. If for some reason valgrind won't work for you (e.g. it is too slow or uses too much memory), there are some basic features in jemalloc that may be of some help.

If jemalloc is configured with --enable-debug specified, various assertions are compiled in that can detect double frees, misaligned pointers, etc. These checks are unnecessary for correctly functioning applications, but they can be helpful during development. One caveat is that many of the assertions relate to arena data structures, but thread caching can prevent arena-related assertions from running in a timely fashion. Disabling thread caching (MALLOC_CONF=tcache:false) tends to make jemalloc's internal assertions much more effective at catching application errors, especially double frees.

4.1.2. JEMALLOC_FILL

If jemalloc is configured with --enable-fill specified, you can set MALLOC_CONF=junk:true in the environment to tell malloc() to fill objects with 0xa5 bytes, and free() to fill freed memory with 0x5a bytes. It is quite common for an application to accidentally get zeroed memory from malloc(), which can hide bugs due to missing initialization code. Another common mistake is to access an object after it has been freed. If your application behaves differently with the junk:true option specified, there's a bug in the code somewhere. Similarly, if the problem is intermittent, and MALLOC_CONF=zero:true makes the problem go away, your application probably depends on getting zeroed memory from malloc().

4.1.3. JEMALLOC_TCACHE

4.2. Android 里 Jemalloc 的初始化参数

4.2.1. Arena 数量

```
//   ANDROID_MAX_ARENAS=XX
//     The total number of arenas will be less than or equal to this number.
//     The number of arenas will be calculated as 2 * the number of cpus
//     but no larger than XX.
common_product_variables = {
    // Only enable the tcache on non-svelte configurations, to save PSS.
    malloc_not_svelte: {
        cflags: [
            "-UANDROID_MAX_ARENAS",
            "-DANDROID_MAX_ARENAS=2",
            "-DJEMALLOC_TCACHE",
            "-DANDROID_TCACHE_NSLOTS_SMALL_MAX=8",
            "-DANDROID_TCACHE_NSLOTS_LARGE=16",
        ],
    },
}
```

Android 最大使用两个 arena，

4.2.2. LG_SIZEOF_PTR

LG_SIZEOF_PTR 这个是什么意思？

```
/* sizeof(void *) == 2^LG_SIZEOF_PTR. */
#ifdef __LP64__
#define        LG_SIZEOF_PTR 3
#else
#define        LG_SIZEOF_PTR 2
#endif
```

```
(gdb) p sizeof(void *)
$55 = 8
```

LG_SIZEOF_PTR=3

4.2.3. NBINS

NBINS 为 36，

#if (LG_SIZEOF_PTR == 2 && LG_TINY_MIN == 3 && LG_QUANTUM == 4 && LG_PAGE == 12)

#if (LG_SIZEOF_PTR == 3 && LG_TINY_MIN == 3 && LG_QUANTUM == 4 && LG_PAGE == 12)

这两种情况 NBINS 都是 36，所以 LG_TINY_MIN=3，LG_QUANTUM=4。

```
#define     SIZE_CLASSES_DEFINED
#define     NTBINS                  1
#define     NLBINS                  29
#define     NBINS                   36
#define     NSIZES                  232
#define     NPSIZES                 199
#define     LG_TINY_MAXCLASS        3
#define     LOOKUP_MAXCLASS             ((((size_t)1) << 11) + (((size_t)4) << 9))
#define     SMALL_MAXCLASS              ((((size_t)1) << 13) + (((size_t)3) << 11))
#define     LG_LARGE_MINCLASS       14
#define     HUGE_MAXCLASS           ((((size_t)1) << 62) + (((size_t)3) << 60))
```

4.2.4. Chunk size 大小

```
//   ANDROID_LG_CHUNK_DEFAULT=XX
//     1 << XX is the default chunk size used by the system. Decreasing this
//     usually decreases the amount of PSS used, but can increase
//     fragmentation.

   multilib: {
      lib32: {
         // Use a 512K chunk size on 32 bit systems.
         // This keeps the total amount of virtual address space consumed
         // by jemalloc lower.
         cflags: [
            "-DANDROID_LG_CHUNK_DEFAULT=19",
         ],
      },
```

```
        lib64: {
            // Use a 2MB chunk size on 64 bit systems.
            // This is the default currently used by 4.0.0
            cflags: [
                "-DANDROID_LG_CHUNK_DEFAULT=21",
            ],
        },
    },
```

对于 32 位 chunk size 是 512K，对于 64 位是 2M。

4.2.5. Tcache 相关配置

```
//   ANDROID_TCACHE_NSLOTS_SMALL_MAX=XX
//     The number of small slots held in the tcache. The higher this number
//     is, the higher amount of PSS consumed. If this number is set too low
//     then small allocations will take longer to complete.
//   ANDROID_TCACHE_NSLOTS_LARGE=XX
//     The number of large slots held in the tcache. The higher this number
//     is, the higher amount of PSS consumed. If this number is set too low
//     then large allocations will take longer to complete.
//   ANDROID_LG_TCACHE_MAXCLASS_DEFAULT=XX
//     1 << XX is the maximum sized allocation that will be in the tcache.

common_cflags += [
    "-DANDROID_MAX_ARENAS=1",
    "-DANDROID_LG_TCACHE_MAXCLASS_DEFAULT=16",
]

common_product_variables = {
    // Only enable the tcache on non-svelte configurations, to save PSS.
    malloc_not_svelte: {
        cflags: [
            "-UANDROID_MAX_ARENAS",
            "-DANDROID_MAX_ARENAS=2",
            "-DJEMALLOC_TCACHE",
            "-DANDROID_TCACHE_NSLOTS_SMALL_MAX=8",
            "-DANDROID_TCACHE_NSLOTS_LARGE=16",
```

```
        ],
    },
}
```

Tcache 中最大的分配是 1<<16, = 64K

```
for (; i < nhbins; i++) { //i=[36,44]
                tcache_bin_info[i].ncached_max      =      TCACHE_NSLOTS_LARGE;//
ANDROID_TCACHE_NSLOTS_LARGE=16
                stack_nelms += tcache_bin_info[i].ncached_max;
}

(gdb) p *je_tcache_bin_info@45
$7 = {
  {ncached_max = 8} <repeats 16 times>, {ncached_max = 20}, {
   ncached_max = 8}, {ncached_max = 8}, {ncached_max = 8}, {
   ncached_max = 20}, {ncached_max = 8}, {ncached_max = 20}, {
   ncached_max = 8}, {ncached_max = 20} <repeats 12 times>, {
//上面是前 36 个，下面是后 9 个 large 的。
   ncached_max = 16}, {ncached_max = 16}, {ncached_max = 16}, {
   ncached_max = 16}, {ncached_max = 16}, {ncached_max = 16}, {
   ncached_max = 16}, {ncached_max = 16}, {ncached_max = 16}}
```

4.3. Jemalloc 参数初始化

4.3.1. 以 LG_开始的宏常量定义

所有 jemalloc LG_开始的符号定义，其值为 log base 2 的意思。LG_SIZEOF_LONG 在 64 位的系统为 3，因为 SIZEOF_LONG 为 8，所以 LG_SIZEOF_LONG=3， 2^3=8。

```
/* Maximum number of regions in one run. */
#define        LG_RUN_MAXREGS                (LG_PAGE - LG_TINY_MIN)
#define        RUN_MAXREGS             (1U << LG_RUN_MAXREGS)

LG_PAGE=12  //4K  One page is 2^LG_PAGE bytes.
```

LG_TINY_MIN=3 //8 Minimum size class to support is 2^LG_TINY_MIN bytes.
LG_RUN_MAXREGS=9 // RUN_MAXREGS=2^9
LG_BITMAP_MAXBITS= LG_RUN_MAXREGS=9

LG_SIZEOF_BITMAP= LG_SIZEOF_LONG=2 or 3 // 每个 BITMAP 字节数 log
/* Number of bits per group. */
#define LG_BITMAP_GROUP_NBITS (LG_SIZEOF_BITMAP + 3)
#define BITMAP_GROUP_NBITS (ZU(1) << LG_BITMAP_GROUP_NBITS)
#define BITMAP_GROUP_NBITS_MASK (BITMAP_GROUP_NBITS-1)

LG_BITMAP_GROUP_NBITS= LG_SIZEOF_BITMAP+3=5 or 6 //3 是每字节的 bits 数的 log

4.3.2. map_bias 的初始化

map_bias：chunk 中 header 所占的 page 数量。

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

分三次进行，第一次把 map_bias 初始化为 0，然后计算出 header_size，这个 header_size 值比实际偏大的，再计算出 map_bias，这个值相比 0 应该很接近真实值了。第二次计算的时候，把第一次的 map_bias 计算结果代入计算，相对于第一次的 0 应该和实际是比较接近的，这个时候计算出来的 header_size 比第一次是小的，所以再次计算出来的 map_bias 可能比第一次的结果小 1。因为第二次的 map_bias 可能会发生了变化，所以进行第三次的迭代，这次计算的结果可能会比第二次的结果大 1。

sizeof(arena_chunk_t)=128
offsetof(arena_chunk_t, map_bits)=120
sizeof(arena_chunk_map_bits_t)=8
sizeof(arena_chunk_map_misc_t)=96
chunk_npages=512
PAGE_MASK= PAGE – 1=4095， LG_PAGE=12

第一次用 map_bias=0 计算得到 header_size=53368, map_bias=14
第二次用 map_bias=14 计算得到 header_size=51912, map_bias=13
第三次用 map_bias=13 计算得到 header_size=52016, map_bias=13

第一次的 header_size=53368，应该是理论最大值，实际不可能。所以 map_bias=14 也是最大值。第二次的计算是根据 map_bias 的最大值 14 计算的 header_size=51912 的最小值, map_bias=13 也是最小值。第三次是用修正后的值 map_bias=13 再次计算得到 map_bias=13 。

4.4. Jemalloc 的统计输出

```
>trace
trace

___ Begin jemalloc statistics ___
Version: 4.4.0-0-gf1f76357313e7dcad7262f17a48ff0a2e005fcdc
Assertions disabled
config.malloc_conf: ""
Run-time option settings:
  opt.abort: false
  opt.lg_chunk: 19
```

```
  opt.dss: "secondary"
  opt.narenas: 1
  opt.purge: "decay"
  opt.decay_time: 0 (arenas.decay_time: 0)
  opt.junk: "false"
  opt.quarantine: 0
  opt.redzone: false
  opt.zero: false
  opt.stats_print: false
Arenas: 1
Unused dirty page decay time: 0
Quantum size: 8
Page size: 4096
Allocated: 1446088, active: 1912832, metadata: 129832, resident: 2031616, mapped:
3145728, retained: 0
Current active ceiling: 2097152

arenas[0]:
assigned threads: 13
dss allocation precedence: disabled
decay time: 0
purging: dirty: 0, sweeps: 117, madvises: 117, purged: 317
                     allocated      nmalloc      ndalloc    nrequests
small:                  463048         6030         4892         6030
large:                  983040           20            5           20
huge:                        0            0            0            0
total:                  1446088         6050         4897         6050
active:                 1912832
mapped:                 2621440
retained:                     0
metadata: mapped: 61440, allocated: 11752
bins:               size ind     allocated        nmalloc        ndalloc     nrequests        curregs
curruns regs pgs  util      newruns       reruns
                       8   0         152            47             28            47             19             1  512   1 0.037
1          0
```

	16	1	544	162	128	162	34	1	256	1	
0.132	4	0									
	24	2	2280	366	271	366	95	1	512	3	
0.185	1	0									
	32	3	416	140	127	140	13	1	128	1	
0.101	1	0									
	40	4	840	521	500	521	21	1	512	5	
0.041	1	0									
	48	5	11232	1418	1184	1418	234	2	256	3	
0.457	17	38									
	56	6	6608	776	658	776	118	1	512	7	
0.230	1	0									
	64	7	11776	1224	1040	1224	184	3	64	1	
0.958	36	41									
	80	8	2320	62	33	62	29	1	256	5	
0.113	1	0									
	96	9	2016	49	28	49	21	1	128	3	
0.164	1	0									
	112	10	1120	47	37	47	10	1	256	7	
0.039	1	0									
	128	11	14208	275	164	275	111	5	32	1	
0.693	5	18									
	160	12	1280	147	139	147	8	1	128	5	
0.062	1	0									
	192	13	1152	87	81	87	6	1	64	3	
0.093	1	0									
	224	14	5824	49	23	49	26	1	128	7	
0.203	1	0									
	256	15	4096	60	44	60	16	1	16	1	1
28	1										
	320	16	5120	46	30	46	16	1	64	5	
0.250	1	0									
	384	17	5760	59	44	59	15	1	32	3	
0.468	1	0									
	448	18	2688	19	13	19	6	1	64	7	
0.093	2	0									

```
                512  19       5120         85         75         85         10          2    8   1
0.625        2          2
                640  20       7680         13          1         13         12          1   32   5
0.375        1          0
                768  21       3840         10          5         10          5          1   16   3 0.312
1            0
                896  22       4480          8          3          8          5          1   32   7 0.156
1            0
               1024  23       3072         12          9         12          3          1    4   1 0.750
1            0
               1280  24      23040         33         15         33         18          2   16   5
0.562        2          2
               1536  25       1536          1          0          1          1          1    8   3 0.125
1            0
               1792  26      10752          7          1          7          6          1   16   7 0.375
1            0
               2048  27       4096          3          1          3          2          1    2   1 1
1            0
               2560  28     186880        170         97        170         73         10    8   5
0.912       18          1
               3072  29       3072          3          2          3          1          1    4   3 0.250
2            0
                 ---
               4096  31      12288         11          8         11          3          3    1   1 1
11           0
               5120  32      46080        109        100        109          9          4    4   5
0.562       18          9
               6144  33      18432          5          2          5          3          2    2   3 0.750
3            0
                 ---
              10240  36      40960          5          1          5          4          2    2   5 1
3            0
              12288  37      12288          1          0          1          1          1    1   3 1
1            0
                 ---
large:          size ind    allocated      nmalloc      ndalloc    nrequests      curruns
```

```
            16384  39      49152          3         0         3         3
                ---
            40960  44          0          2         2         2         0
            49152  45      49152          1         0         1         1
                ---
            65536  47     458752         10         3        10         7
                ---
            98304  49     196608          2         0         2         2
           114688  50     229376          2         0         2         2
                ---
huge:           size ind    allocated      nmalloc     ndalloc    nrequests   curhchunks
                ---
--- End jemalloc statistics ---
return = 0x00000000
```

5. Jemalloc 多核/多线程分配和互斥锁

相对经典分配器, jemalloc 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(锁竞争). 因为无论 CPU 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, jemalloc 的分配速度越快. 而这一点是经典分配器所无法达到的。

5.1. 锁的分类

malloc_mutex_lock
有几种互斥锁，arena->lock，bin->lock，arena->node_cache_mtx，arena->huge_mtx，base_mtx，arena->chunks_mtx，ctl_mtx，init_lock，gctx->lock，

5.2. Tcache 分配的过程

Tcache 是线程拥有的内存的快速分配，所以不需要加锁。这也就是 jemalloc 在线程越多分配越快的原因。

5.3. 从 bin 的 runcur 中的分配过程

这个时候需要加锁 bin->lock，因为需要从 bin 分配内存到线程 Tcache 或者给应用程序，malloc_mutex_lock(tsdn, &bin->lock);

5.4. 从 bin 的 runs 中的分配过程

这个时候需要加锁 bin->lock，因为需要从 bin 分配内存到线程 Tcache 或者给应用程序，malloc_mutex_lock(tsdn, &bin->lock);

5.5. arena->runs_avail[i] 中的分配过程

这个时候需要加锁 arena->lock，因为需要从 arena 分配新的 run 到线程 Tcache 或者给应用程序， malloc_mutex_lock(tsdn, &arena->lock);

5.6. new chunk 的分配过程

1） 从 chunk = arena->spare;分配的时候不需要加锁；
2） 从&arena->chunks_szsnad_cached, &arena->chunks_ad_cached 分配的时候，需要加锁 arena->chunks_mtx ，malloc_mutex_lock(tsdn, &arena->chunks_mtx);
3） 其他方式的时候会释放 malloc_mutex_unlock(tsdn, &arena->lock);这个锁再尝试分配。

6. Region size 设计以及和 index 对应关系

6.1. Region size 步长的设计

下图是对于从 reg_size=64 开始的增加规则，可以看出步长增加规则是每次在相邻幂次数位加 1，得到下一个 reg_size 值。

1024	512	256	128	64	32	16	8	4	2	1	
				1	0	0					reg_size = 64
				1	0	1					reg_size = 80
				1	1	0					reg_size = 96
				1	1	1					reg_size = 112
			1	0	0						reg_size = 128
			1	0	1						reg_size = 160
			1	1	0						reg_size = 192
			1	1	1						reg_size = 224
		1	0	0							reg_size = 256
		1	0	1							reg_size = 320
		1	1	0							reg_size = 384
		1	1	1							reg_size = 448
	1	0	0								reg_size = 512
	1	0	1								reg_size = 640
	1	1	0								reg_size = 768
	1	1	1								reg_size = 896
1	0	0									reg_size = 1024
1	0	1									reg_size = 1280
1	1	0									reg_size = 1536
1	1	1									reg_size = 1792
											reg_size = 2048
											reg_size = 2560
											reg_size = 3072
											reg_size = 3584
											reg_size = 4096
											reg_size = 5120
											reg_size = 6144
											reg_size = 7168
											reg_size = 8192
											reg_size = 10240
											reg_size = 12288
											reg_size = 14336

6.2. Region 相关的参数定义（anrdoid 64bits）

以下配置是对应于如下的系统参数： (LG_SIZEOF_PTR == 3 && LG_TINY_MIN == 3 && LG_QUANTUM == 4 && LG_PAGE == 12)

```
#define     NTBINS                  1
#define     NLBINS                  29
#define     NBINS                   36
#define     NSIZES                  232
#define     NPSIZES                 199
#define     LG_TINY_MAXCLASS        3
#define     LOOKUP_MAXCLASS             ((((size_t)1) << 11) + (((size_t)4) << 9))
#define     SMALL_MAXCLASS              ((((size_t)1) << 13) + (((size_t)3) << 11))
#define     LG_LARGE_MINCLASS       14
```

```
#define         HUGE_MAXCLASS           ((((size_t)1) << 62) + (((size_t)3) << 60))
```

6.3. SIZE_CLASSES 定义

```
#define         SIZE_CLASSES \
  /* index, lg_grp, lg_delta, ndelta, psz, bin, lg_delta_lookup */ \
  SC(  0,     3,      3,     0,  no, yes,  3) \
                                             \
  SC(  1,     3,      3,     1,  no, yes,  3) \
  SC(  2,     4,      4,     1,  no, yes,  4) \
  SC(  3,     4,      4,     2,  no, yes,  4) \
  SC(  4,     4,      4,     3,  no, yes,  4) \
                                             \
                                  \
  SC(  5,     6,      4,     1,  no, yes,  4) \
  SC(  6,     6,      4,     2,  no, yes,  4) \
  SC(  7,     6,      4,     3,  no, yes,  4) \
  SC(  8,     6,      4,     4,  no, yes,  4) \
                                             \
  SC(  9,     7,      5,     1,  no, yes,  5) \
  SC( 10,     7,      5,     2,  no, yes,  5) \
  SC( 11,     7,      5,     3,  no, yes,  5) \
  SC( 12,     7,      5,     4,  no, yes,  5) \
                                             \
  SC( 13,     8,      6,     1,  no, yes,  6) \
  SC( 14,     8,      6,     2,  no, yes,  6) \
  SC( 15,     8,      6,     3,  no, yes,  6) \
  SC( 16,     8,      6,     4,  no, yes,  6) \
                                             \
  SC( 17,     9,      7,     1,  no, yes,  7) \
  SC( 18,     9,      7,     2,  no, yes,  7) \
  SC( 19,     9,      7,     3,  no, yes,  7) \
  SC( 20,     9,      7,     4,  no, yes,  7) \
                                             \
  SC( 21,    10,      8,     1,  no, yes,  8) \
  SC( 22,    10,      8,     2,  no, yes,  8) \
  SC( 23,    10,      8,     3,  no, yes,  8) \
  SC( 24,    10,      8,     4,  no, yes,  8) \
                                             \
  SC( 25,    11,      9,     1,  no, yes,  9) \
  SC( 26,    11,      9,     2,  no, yes,  9) \
  SC( 27,    11,      9,     3,  no, yes,  9) \
  SC( 28,    11,      9,     4, yes, yes,  9) \
                                             \
  SC( 29,    12,     10,     1,  no, yes, no) \
  SC( 30,    12,     10,     2,  no, yes, no) \
  SC( 31,    12,     10,     3,  no, yes, no) \
  SC( 32,    12,     10,     4, yes, yes, no) \
                                             \
  SC( 33,    13,     11,     1,  no, yes, no) \
  SC( 34,    13,     11,     2, yes, yes, no) \
  SC( 35,    13,     11,     3,  no, yes, no) \   //small max
  SC( 36,    13,     11,     4, yes,  no, no) \   //large start = 2^14
                                             \
  SC( 37,    14,     12,     1, yes,  no, no) \
  SC( 38,    14,     12,     2, yes,  no, no) \
  SC( 39,    14,     12,     3, yes,  no, no) \
  SC( 40,    14,     12,     4, yes,  no, no) \
                                             \
  SC( 41,    15,     13,     1, yes,  no, no) \
  SC( 42,    15,     13,     2, yes,  no, no) \
  SC( 43,    15,     13,     3, yes,  no, no) \
  SC( 44,    15,     13,     4, yes,  no, no) \
                                             \
  SC( 45,    16,     14,     1, yes,  no, no) \
```

```
    SC( 46,    16,     14,     2, yes,  no, no) \
    SC( 47,    16,     14,     3, yes,  no, no) \
    SC( 48,    16,     14,     4, yes,  no, no) \
                                               \
    SC( 49,    17,     15,     1, yes,  no, no) \
    SC( 50,    17,     15,     2, yes,  no, no) \
    SC( 51,    17,     15,     3, yes,  no, no) \
    SC( 52,    17,     15,     4, yes,  no, no) \
                                               \
    SC( 53,    18,     16,     1, yes,  no, no) \
    SC( 54,    18,     16,     2, yes,  no, no) \
    SC( 55,    18,     16,     3, yes,  no, no) \
    SC( 56,    18,     16,     4, yes,  no, no) \
                                               \
    SC( 57,    19,     17,     1, yes,  no, no) \
    SC( 58,    19,     17,     2, yes,  no, no) \
    SC( 59,    19,     17,     3, yes,  no, no) \
    SC( 60,    19,     17,     4, yes,  no, no) \
                                               \
    SC( 61,    20,     18,     1, yes,  no, no) \
    SC( 62,    20,     18,     2, yes,  no, no) \
    SC( 63,    20,     18,     3, yes,  no, no) \
    SC( 64,    20,     18,     4, yes,  no, no) \
                                               \
    SC( 65,    21,     19,     1, yes,  no, no) \
    SC( 66,    21,     19,     2, yes,  no, no) \
    SC( 67,    21,     19,     3, yes,  no, no) \
    SC( 68,    21,     19,     4, yes,  no, no) \
                                               \
    SC( 69,    22,     20,     1, yes,  no, no) \
    SC( 70,    22,     20,     2, yes,  no, no) \
    SC( 71,    22,     20,     3, yes,  no, no) \
    SC( 72,    22,     20,     4, yes,  no, no) \
                                               \
    SC( 73,    23,     21,     1, yes,  no, no) \
    SC( 74,    23,     21,     2, yes,  no, no) \
    SC( 75,    23,     21,     3, yes,  no, no) \
    SC( 76,    23,     21,     4, yes,  no, no) \
                                               \
    SC( 77,    24,     22,     1, yes,  no, no) \
    SC( 78,    24,     22,     2, yes,  no, no) \
    SC( 79,    24,     22,     3, yes,  no, no) \
    SC( 80,    24,     22,     4, yes,  no, no) \
                                               \
    SC( 81,    25,     23,     1, yes,  no, no) \
    SC( 82,    25,     23,     2, yes,  no, no) \
    SC( 83,    25,     23,     3, yes,  no, no) \
    SC( 84,    25,     23,     4, yes,  no, no) \
                                               \
    SC( 85,    26,     24,     1, yes,  no, no) \
    SC( 86,    26,     24,     2, yes,  no, no) \
    SC( 87,    26,     24,     3, yes,  no, no) \
    SC( 88,    26,     24,     4, yes,  no, no) \
                                               \
    SC( 89,    27,     25,     1, yes,  no, no) \
    SC( 90,    27,     25,     2, yes,  no, no) \
    SC( 91,    27,     25,     3, yes,  no, no) \
    SC( 92,    27,     25,     4, yes,  no, no) \
                                               \
    SC( 93,    28,     26,     1, yes,  no, no) \
    SC( 94,    28,     26,     2, yes,  no, no) \
    SC( 95,    28,     26,     3, yes,  no, no) \
    SC( 96,    28,     26,     4, yes,  no, no) \
                                               \
    SC( 97,    29,     27,     1, yes,  no, no) \
    SC( 98,    29,     27,     2, yes,  no, no) \
    SC( 99,    29,     27,     3, yes,  no, no) \
    SC(100,    29,     27,     4, yes,  no, no) \
                                               \
    SC(101,    30,     28,     1, yes,  no, no) \
    SC(102,    30,     28,     2, yes,  no, no) \
    SC(103,    30,     28,     3, yes,  no, no) \
    SC(104,    30,     28,     4, yes,  no, no) \
                                               \
    SC(105,    31,     29,     1, yes,  no, no) \
```

```
SC(106,    31,     29,     2, yes,  no, no) \
SC(107,    31,     29,     3, yes,  no, no) \
SC(108,    31,     29,     4, yes,  no, no) \
                                            \
SC(109,    32,     30,     1, yes,  no, no) \
SC(110,    32,     30,     2, yes,  no, no) \
SC(111,    32,     30,     3, yes,  no, no) \
SC(112,    32,     30,     4, yes,  no, no) \
                                            \
SC(113,    33,     31,     1, yes,  no, no) \
SC(114,    33,     31,     2, yes,  no, no) \
SC(115,    33,     31,     3, yes,  no, no) \
SC(116,    33,     31,     4, yes,  no, no) \
                                            \
SC(117,    34,     32,     1, yes,  no, no) \
SC(118,    34,     32,     2, yes,  no, no) \
SC(119,    34,     32,     3, yes,  no, no) \
SC(120,    34,     32,     4, yes,  no, no) \
                                            \
SC(121,    35,     33,     1, yes,  no, no) \
SC(122,    35,     33,     2, yes,  no, no) \
SC(123,    35,     33,     3, yes,  no, no) \
SC(124,    35,     33,     4, yes,  no, no) \
                                            \
SC(125,    36,     34,     1, yes,  no, no) \
SC(126,    36,     34,     2, yes,  no, no) \
SC(127,    36,     34,     3, yes,  no, no) \
SC(128,    36,     34,     4, yes,  no, no) \
                                            \
SC(129,    37,     35,     1, yes,  no, no) \
SC(130,    37,     35,     2, yes,  no, no) \
SC(131,    37,     35,     3, yes,  no, no) \
SC(132,    37,     35,     4, yes,  no, no) \
                                            \
SC(133,    38,     36,     1, yes,  no, no) \
SC(134,    38,     36,     2, yes,  no, no) \
SC(135,    38,     36,     3, yes,  no, no) \
SC(136,    38,     36,     4, yes,  no, no) \
                                            \
SC(137,    39,     37,     1, yes,  no, no) \
SC(138,    39,     37,     2, yes,  no, no) \
SC(139,    39,     37,     3, yes,  no, no) \
SC(140,    39,     37,     4, yes,  no, no) \
                                            \
SC(141,    40,     38,     1, yes,  no, no) \
SC(142,    40,     38,     2, yes,  no, no) \
SC(143,    40,     38,     3, yes,  no, no) \
SC(144,    40,     38,     4, yes,  no, no) \
                                            \
SC(145,    41,     39,     1, yes,  no, no) \
SC(146,    41,     39,     2, yes,  no, no) \
SC(147,    41,     39,     3, yes,  no, no) \
SC(148,    41,     39,     4, yes,  no, no) \
                                            \
SC(149,    42,     40,     1, yes,  no, no) \
SC(150,    42,     40,     2, yes,  no, no) \
SC(151,    42,     40,     3, yes,  no, no) \
SC(152,    42,     40,     4, yes,  no, no) \
                                            \
SC(153,    43,     41,     1, yes,  no, no) \
SC(154,    43,     41,     2, yes,  no, no) \
SC(155,    43,     41,     3, yes,  no, no) \
SC(156,    43,     41,     4, yes,  no, no) \
                                            \
SC(157,    44,     42,     1, yes,  no, no) \
SC(158,    44,     42,     2, yes,  no, no) \
SC(159,    44,     42,     3, yes,  no, no) \
SC(160,    44,     42,     4, yes,  no, no) \
                                            \
SC(161,    45,     43,     1, yes,  no, no) \
SC(162,    45,     43,     2, yes,  no, no) \
SC(163,    45,     43,     3, yes,  no, no) \
SC(164,    45,     43,     4, yes,  no, no) \
                                            \
SC(165,    46,     44,     1, yes,  no, no) \
```

```
SC(166,    46,    44,    2, yes,  no, no) \
SC(167,    46,    44,    3, yes,  no, no) \
SC(168,    46,    44,    4, yes,  no, no) \
                                          \
SC(169,    47,    45,    1, yes,  no, no) \
SC(170,    47,    45,    2, yes,  no, no) \
SC(171,    47,    45,    3, yes,  no, no) \
SC(172,    47,    45,    4, yes,  no, no) \
                                          \
SC(173,    48,    46,    1, yes,  no, no) \
SC(174,    48,    46,    2, yes,  no, no) \
SC(175,    48,    46,    3, yes,  no, no) \
SC(176,    48,    46,    4, yes,  no, no) \
                                          \
SC(177,    49,    47,    1, yes,  no, no) \
SC(178,    49,    47,    2, yes,  no, no) \
SC(179,    49,    47,    3, yes,  no, no) \
SC(180,    49,    47,    4, yes,  no, no) \
                                          \
SC(181,    50,    48,    1, yes,  no, no) \
SC(182,    50,    48,    2, yes,  no, no) \
SC(183,    50,    48,    3, yes,  no, no) \
SC(184,    50,    48,    4, yes,  no, no) \
                                          \
SC(185,    51,    49,    1, yes,  no, no) \
SC(186,    51,    49,    2, yes,  no, no) \
SC(187,    51,    49,    3, yes,  no, no) \
SC(188,    51,    49,    4, yes,  no, no) \
                                          \
SC(189,    52,    50,    1, yes,  no, no) \
SC(190,    52,    50,    2, yes,  no, no) \
SC(191,    52,    50,    3, yes,  no, no) \
SC(192,    52,    50,    4, yes,  no, no) \
                                          \
SC(193,    53,    51,    1, yes,  no, no) \
SC(194,    53,    51,    2, yes,  no, no) \
SC(195,    53,    51,    3, yes,  no, no) \
SC(196,    53,    51,    4, yes,  no, no) \
                                          \
SC(197,    54,    52,    1, yes,  no, no) \
SC(198,    54,    52,    2, yes,  no, no) \
SC(199,    54,    52,    3, yes,  no, no) \
SC(200,    54,    52,    4, yes,  no, no) \
                                          \
SC(201,    55,    53,    1, yes,  no, no) \
SC(202,    55,    53,    2, yes,  no, no) \
SC(203,    55,    53,    3, yes,  no, no) \
SC(204,    55,    53,    4, yes,  no, no) \
                                          \
SC(205,    56,    54,    1, yes,  no, no) \
SC(206,    56,    54,    2, yes,  no, no) \
SC(207,    56,    54,    3, yes,  no, no) \
SC(208,    56,    54,    4, yes,  no, no) \
                                          \
SC(209,    57,    55,    1, yes,  no, no) \
SC(210,    57,    55,    2, yes,  no, no) \
SC(211,    57,    55,    3, yes,  no, no) \
SC(212,    57,    55,    4, yes,  no, no) \
                                          \
SC(213,    58,    56,    1, yes,  no, no) \
SC(214,    58,    56,    2, yes,  no, no) \
SC(215,    58,    56,    3, yes,  no, no) \
SC(216,    58,    56,    4, yes,  no, no) \
                                          \
SC(217,    59,    57,    1, yes,  no, no) \
SC(218,    59,    57,    2, yes,  no, no) \
SC(219,    59,    57,    3, yes,  no, no) \
SC(220,    59,    57,    4, yes,  no, no) \
                                          \
SC(221,    60,    58,    1, yes,  no, no) \
SC(222,    60,    58,    2, yes,  no, no) \
SC(223,    60,    58,    3, yes,  no, no) \
SC(224,    60,    58,    4, yes,  no, no) \
                                          \
SC(225,    61,    59,    1, yes,  no, no) \
```

```
    SC(226,    61,     59,     2, yes,  no, no) \
    SC(227,    61,     59,     3, yes,  no, no) \
    SC(228,    61,     59,     4, yes,  no, no) \
                                    \
    SC(229,    62,     60,     1, yes,  no, no) \
    SC(230,    62,     60,     2, yes,  no, no) \
    SC(231,    62,     60,     3, yes,  no, no) \
```

6.4. size2index_tab （用于 reg_size 小于 4096 的快速查找）

```
/*
 * size2index_tab is a compact lookup table that rounds request sizes up to
 * size classes.  In order to reduce cache footprint, the table is compressed,
 * and all accesses are via size2index().
 */
```

对于 LG_TINY_MIN=3 化简后的 size2index_tab 定义如下：

```
const uint8_t size2index_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, psz, bin, lg_delta_lookup) \
        S2B_##lg_delta_lookup(index)
        SIZE_CLASSES
};
```

下面是 SIZE_CLASSES 定义，根据这个生成常量数组。

```
    SC(  0,     3,      3,      0,  no, yes,  3) \
                                        \
    SC(  1,     3,      3,      1,  no, yes,  3) \
    SC(  2,     4,      4,      1,  no, yes,  4) \
    SC(  3,     4,      4,      2,  no, yes,  4) \
    SC(  4,     4,      4,      3,  no, yes,  4) \
                                        \
```

```
    SC(  5,     6,      4,     1,  no, yes,  4) \
    SC(  6,     6,      4,     2,  no, yes,  4) \
    SC(  7,     6,      4,     3,  no, yes,  4) \
    SC(  8,     6,      4,     4,  no, yes,  4) \
                                               \
    SC(  9,     7,      5,     1,  no, yes,  5) \
    SC( 10,     7,      5,     2,  no, yes,  5) \
    SC( 11,     7,      5,     3,  no, yes,  5) \
    SC( 12,     7,      5,     4,  no, yes,  5) \
                                               \
    SC( 13,     8,      6,     1,  no, yes,  6) \
    SC( 14,     8,      6,     2,  no, yes,  6) \
    SC( 15,     8,      6,     3,  no, yes,  6) \
    SC( 16,     8,      6,     4,  no, yes,  6) \
                                               \
    SC( 17,     9,      7,     1,  no, yes,  7) \
    SC( 18,     9,      7,     2,  no, yes,  7) \
    SC( 19,     9,      7,     3,  no, yes,  7) \
    SC( 20,     9,      7,     4,  no, yes,  7) \
                                               \
    SC( 21,    10,      8,     1,  no, yes,  8) \
    SC( 22,    10,      8,     2,  no, yes,  8) \
    SC( 23,    10,      8,     3,  no, yes,  8) \
    SC( 24,    10,      8,     4,  no, yes,  8) \
                                               \
    SC( 25,    11,      9,     1,  no, yes,  9) \
    SC( 26,    11,      9,     2,  no, yes,  9) \
    SC( 27,    11,      9,     3,  no, yes,  9) \
    SC( 28,    11,      9,     4, yes, yes,  9) \

const uint8_t size2index_tab[] = {
S2B_3(0)  S2B_3(1)  S2B_4(2)  S2B_4(3)  S2B_4(4)  S2B_4(5)  S2B_4(6)  S2B_4(7)
S2B_4(8) S2B_5(9) S2B_5(10) S2B_5(11) S2B_5(12) S2B_6(13) S2B_6(14) S2B_6(15)
S2B_6(16)  S2B_7(17)  S2B_7(18)  S2B_7(19)  S2B_7(20)  S2B_8(21)  S2B_8(22)
S2B_8(23) S2B_8(24) S2B_9(25) S2B_9(26) S2B_9(27) S2B_9(28)
}
```

```
const uint8_t size2index_tab[] = {
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, …
}
```

因为后续步长拉大，而 size2index_tab 按 8 个字节设置一个索引。所以越后面重复的 index 会越多。最大索引是 4096，4096/8=512。所以 size2index_tab 数组个数为 512。

```
 (gdb) p /d je_size2index_tab
$87 = {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>,
  28 <repeats 64 times>}
```

6.5. size2index_compute 的计算过程

```
lg_floor(size_t x)
{
        size_t ret;
        assert(x != 0);
        asm ("bsr %1, %0"
            : "=r"(ret) // Outputs.
            : "r"(x)    // Inputs.
            );
        assert(ret < UINT_MAX);
        return ((unsigned)ret);
}
```

BSR scans the bits in the second word or doubleword operand from the most significant bit to the least significant bit. The ZF flag is cleared if the bits are all 0; otherwise, ZF is set and the destination register is loaded with the bit index of the first set bit found when scanning in the reverse direction.

lg_floor 找到 x 的二进制表示，最高位 1 是第几个数，从右开始，从 0 开始计数。

```
JEMALLOC_INLINE szind_t
size2index_compute(size_t size)
{

	if (unlikely(size > HUGE_MAXCLASS))
		return (NSIZES);
#if (NTBINS != 0)
	if (size <= (ZU(1) << LG_TINY_MAXCLASS)) {
		szind_t lg_tmin = LG_TINY_MAXCLASS - NTBINS + 1;
		szind_t lg_ceil = lg_floor(pow2_ceil_zu(size));
		return (lg_ceil < lg_tmin ? 0 : lg_ceil - lg_tmin);
	}
#endif
	{
		szind_t x = lg_floor((size<<1)-1);
		szind_t shift = (x < LG_SIZE_CLASS_GROUP + LG_QUANTUM) ? 0 :
		    x - (LG_SIZE_CLASS_GROUP + LG_QUANTUM);
		szind_t grp = shift << LG_SIZE_CLASS_GROUP;

		szind_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;
		szind_t mod = ((((size-1) & delta_inverse_mask) >> lg_delta)) &
		    ((ZU(1) << LG_SIZE_CLASS_GROUP) - 1);

		szind_t index = NTBINS + grp + mod;
		return (index);
	}
}
```

size2index_compute 计算过程如下：

```
	szind_t x = lg_floor((size<<1)-1);
//lg_floor 找到输入参数的二进制表示，最高位 1 是第几个数，从右开始，从 0 开始计数。
```

// lg_floor((size<<1)-1);和 size 的关系？如果 size 是 2 的幂次，则 x 就是 size 中 1 的最高位，如果 size 不是 2 的幂次，那么 x 是 size 中 1 的最高位+1。所以这里把 2 的幂次的 size 放到了上一个 group，然后通过 mod 的计算，2 的幂次的 size 的 mod 为 3，比其他的上一个 group 的其他 size 的 index 都要大。（为什么这么处理呢？因为 2 的幂次的 size 的递增的步长和上一个 group 一样，所以需要这么处理。）如下图所示：

1024	512	256	128	64	32	16	mod	
				1	0	0		reg_size = 64
				1	0	1	0	reg_size = 80
				1	1	0	1	reg_size = 96
				1	1	1	2	reg_size = 112
			1	0	0		3	reg_size = 128
				1	0	1	0	reg_size = 160
				1	1	0	1	reg_size = 192
				1	1	1	2	reg_size = 224
		1	0	0			3	reg_size = 256
			1	0	1			reg_size = 320
			1	1	0			reg_size = 384
			1	1	1			reg_size = 448
	1		0	0				reg_size = 512
	1		0	1				reg_size = 640
	1		1	0				reg_size = 768
	1		1	1				reg_size = 896
1	0		0					reg_size = 1024
1	0		1					reg_size = 1280
1	1		0					reg_size = 1536
1	1		1					reg_size = 1792
								reg_size = 2048
								reg_size = 2560
								reg_size = 3072
								reg_size = 3584
								reg_size = 4096
								reg_size = 5120
								reg_size = 6144
								reg_size = 7168
								reg_size = 8192
								reg_size = 10240
								reg_size = 12288
								reg_size = 14336

```
szind_t shift = (x < LG_SIZE_CLASS_GROUP=2 + LG_QUANTUM=4) ? 0 :
        x - (LG_SIZE_CLASS_GROUP=2 + LG_QUANTUM=4);
```

shift=x<6?0:x-6

```
szind_t grp = shift << LG_SIZE_CLASS_GROUP=2;
```

grp=shift*4

```
szind_t lg_delta = (x < LG_SIZE_CLASS_GROUP=2 + LG_QUANTUM=4 + 1)
        ? LG_QUANTUM=4 : x - LG_SIZE_CLASS_GROUP - 1;
```

lg_delta=(x<7)?4:x-3

```
size_t delta_inverse_mask = ZI(-1) << lg_delta;
szind_t mod = ((((size-1) & delta_inverse_mask) >> lg_delta)) &
        ((ZU(1) << LG_SIZE_CLASS_GROUP) - 1);
```

Mod=((((size-1) & delta_inverse_mask) >> lg_delta)) & 0x11

如果 size 是 2 的幂次，则 size-1 是低位全 1 的一个数，从 reg_size 从 80 开始（size 从 128 开始）lg_delta=3，delta_inverse_mask=0xFFFFFFFFFFFFFFF8，这个时候 size-1 至少是 7 位，除去低 3 位，还有 4 位全 1，所以和 0x11 取与后为 3。

如果 size 是非 2 的幂次，mod 计算结果是 0，1，2 中的一种。

```
szind_t index = NTBINS=1 + grp + mod;
return (index);
```

6.6. index2size_compute 的计算过程

```
index2size_compute(szind_t index)
{

#if (NTBINS > 0)
        if (index < NTBINS)
                return (ZU(1) << (LG_TINY_MAXCLASS - NTBINS + 1 + index));
#endif
        {
                size_t reduced_index = index - NTBINS;
                size_t grp = reduced_index >> LG_SIZE_CLASS_GROUP;
                size_t mod = reduced_index & ((ZU(1) << LG_SIZE_CLASS_GROUP) -
                    1);

                size_t grp_size_mask = ~((!!grp)-1);
                size_t grp_size = ((ZU(1) << (LG_QUANTUM +
                    (LG_SIZE_CLASS_GROUP-1))) << grp) & grp_size_mask;

                size_t shift = (grp == 0) ? 1 : grp;
                size_t lg_delta = shift + (LG_QUANTUM-1);
```

```
                size_t mod_size = (mod+1) << lg_delta;

                size_t usize = grp_size + mod_size;
                return (usize);
        }
}
```

index2size_compute 计算过程如下：

```
                size_t reduced_index = index – NTBINS=1;
                size_t grp = reduced_index >> LG_SIZE_CLASS_GROUP=2;
                size_t mod = reduced_index & ((ZU(1) << LG_SIZE_CLASS_GROUP=2) -
                   1);

                size_t grp_size_mask = ~((!!grp)-1);
                size_t grp_size = ((ZU(1) << (LG_QUANTUM +
                   (LG_SIZE_CLASS_GROUP-1))) << grp) & grp_size_mask;

                size_t shift = (grp == 0) ? 1 : grp;
                size_t lg_delta = shift + (LG_QUANTUM-1);
                size_t mod_size = (mod+1) << lg_delta;

                size_t usize = grp_size + mod_size;
                return (usize);
```

index=0,由第一个 if (index < NTBINS) 这里算出，因为 NTBINS=1，1<<3=8

index=1-4 时，reduced_index=0-3，grp=0，mod= reduced_index & 3 = 0-3
grp_size_mask=0x0，grp_size=(1<<5)&0x0=0, shift=1, lg_delta=4,
mod_size=(1-4)<<4=16,32,48,64 usize=0+ mod_size= mod_size=16,32,48,64

index>=5 时，reduced_index>=4, grp= reduced_index/4,
mod= reduced_index&0x3=0-3 grp_size_mask=0xffffffff，
grp_size=((1<<5)<<grp)& 0xffffffff=(32<<grp) & 0xffffffff
shift=grp，lg_delta=grp+3，mod_size=(mod+1) << lg_delta=(1-4)<<(grp+3)
usize = grp_size + mod_size=(32<<grp) & 0xffffffff+(1-4)<<(grp+3)

=(1<<(grp+5))+(1-4)<<(grp+3) (grp=(index-1)/4)

=(1<<lg_grp + ndelta<<lg_delta)

lg_grp=grp+5, lg_delta=grp+3 这个就是 grp 和 index2size_tab 中的 lg_grp 和 lg_delta 的对应关系，也说明了这两个值为什么一直相差 2。

```
   index  lg_grp lg_delta
SC(  5,     6,       4,      1,  no, yes,  4) \ //grp=1
SC(  6,     6,       4,      2,  no, yes,  4) \ //grp=1
SC(  7,     6,       4,      3,  no, yes,  4) \ //grp=1
SC(  8,     6,       4,      4,  no, yes,  4) \ //grp=1
                                              \
SC(  9,     7,       5,      1,  no, yes,  5) \ //grp=2
SC( 10,     7,       5,      2,  no, yes,  5) \ //grp=2
SC( 11,     7,       5,      3,  no, yes,  5) \ //grp=2
SC( 12,     7,       5,      4,  no, yes,  5) \ //grp=2
```

6.7. 步长增加规律分析-浪费率控制

步长=2^(lg_delta)=2^(grp+3)

区间上限=2^(grp+5)+4*2^(grp+3)= 2^(grp+5)+2^(grp+5)=2^(grp+6)

区间下限=2^(grp+5)+1*2^(grp+3)= 2^(grp+5)+2^(grp+3)

所以可以得到步长和区间上限的比值为：2^(grp+3)/ 2^(grp+6)=1/8

步长和区间下限的比值为：2^(grp+3)/ (2^(grp+5)+2^(grp+3))=1/5

主要原则就是保证内存浪费率控制 1/5 内。

6.8. index2size_tab

```
/*
 * index2size_tab encodes the same information as could be computed (at
 * unacceptable cost in some code paths) by index2size_compute().
 */
extern size_t const   index2size_tab[NSIZES];
```

```
const size_t   index2size_tab[NSIZES] = {
#define        SC(index, lg_grp, lg_delta, ndelta, psz, bin, lg_delta_lookup) \
        ((ZU(1)<<lg_grp) + (ZU(ndelta)<<lg_delta)),
        SIZE_CLASSES
#undef SC
};
```

```
(gdb) p je_index2size_tab
$57 = {8, 16, 32, 48, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320, 384,
  448, 512, 640, 768, 896, 1024, 1280, 1536, 1792, 2048, 2560, 3072, 3584,
  4096, 5120, 6144, 7168, 8192, 10240, 12288, 14336, 16384, 20480, 24576,
  28672, 32768, 40960, 49152, 57344, 65536, 81920, 98304, 114688, 131072,
  163840, 196608, 229376, 262144, 327680, 393216, 458752, 524288, 655360,
  786432, 917504, 1048576, 1310720, 1572864, 1835008, 2097152, 2621440,
  3145728, 3670016, 4194304, 5242880, 6291456, 7340032, 8388608, 10485760,
  12582912, 14680064, 16777216, 20971520, 25165824, 29360128, 33554432,
  41943040, 50331648, 58720256, 67108864, 83886080, 100663296, 117440512,
  134217728, 167772160, 201326592, 234881024, 268435456, 335544320,
  402653184, 469762048, 536870912, 671088640, 805306368, 939524096,
  1073741824, 1342177280, 1610612736, 1879048192, 2147483648, 2684354560,
  3221225472, 3758096384, 4294967296, 5368709120, 6442450944, 7516192768,
  8589934592, 10737418240, 12884901888, 15032385536, 17179869184,
  21474836480, 25769803776, 30064771072, 34359738368, 42949672960,
  51539607552, 60129542144, 68719476736, 85899345920, 103079215104,
  120259084288, 137438953472, 171798691840, 206158430208, 240518168576,
  274877906944, 343597383680, 412316860416, 481036337152, 549755813888,
  687194767360, 824633720832, 962072674304, 1099511627776, 1374389534720,
  1649267441664, 1924145348608, 2199023255552, 2748779069440, 3298534883328,
  3848290697216, 4398046511104, 5497558138880, 6597069766656, 7696581394432,
  8796093022208, 10995116277760, 13194139533312, 15393162788864,
  17592186044416, 21990232555520, 26388279066624, 30786325577728,
  35184372088832, 43980465111040, 52776558133248, 61572651155456,
  70368744177664, 87960930222080, 105553116266496, 123145302310912,
  140737488355328, 175921860444160, 211106232532992, 246290604621824,
  281474976710656, 351843720888320, 422212465065984, 492581209243648,
```

562949953421312, 703687441776640, 844424930131968, 985162418487296,
1125899906842624, 1407374883553280, 1688849860263936, 1970324836974592,
2251799813685248, 2814749767106560, 3377699720527872, 3940649673949184,
4503599627370496, 5629499534213120, 6755399441055744, 7881299347898368,
9007199254740992, 11258999068426240, 13510798882111488,
15762598695796736,
18014398509481984, 22517998136852480, 27021597764222976,
31525197391593472...}
(gdb)

7. Jemalloc 内存分配释放过程

7.1. 从 arena 中分配 small size 内存的过程

small region size 区间为： 8 <= size <= 14336 = SMALL_MAXCLASS。

函数 arena_malloc 中，
#define SMALL_MAXCLASS ((((size_t)1) << 13) + (((size_t)3) << 11))
SMALL_MAXCLASS=14k=14336
对于 small allocation，size <= SMALL_MAXCLASS=14k=14336，需要确定请求大小对应到哪一 bin 上，确定的公式如下： size2index(usize)，该公式通过查数组或计算来确定 bin 的 index。

7.1.1. tcache 中的分配过程

当 size 区间在 size <= SMALL_MAXCLASS=14k=14336 时，如果也 enable 了 tcache，使用 tcache_alloc_small 进行分配。tcache_alloc_small 调用 tcache_alloc_easy 进行快速分配，

```
tbin = &tcache->tbins[binind];
*tcache_success = true;
ret = *(tbin->avail - tbin->ncached);
tbin->ncached--;
```

如上面代码所示，直接取得栈顶元素返回。如果 tcache_bin 用完，则调用 arena_tcache_fill_small 从 arena 分配分配内存填充 tcache_bin，然后再通过 tcache_alloc_easy 进行分配。arena_tcache_fill_small 的内存填充和非 tcache 的分配方式一样。

7.1.2. 从 bin 的 runcur 中的分配过程

如果不满足 tcache 分配条件，则用 arena_malloc_hard 再调用 arena_malloc_small 进行分配。在 arena_malloc_small 中，如果当前 bin 存在 runcur，直接调用 arena_run_reg_alloc 进行分配，

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

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

        regind = (unsigned)bitmap_sfu(run->bitmap, &bin_info->bitmap_info);
        miscelm = arena_run_to_miscelm(run);
        rpages = arena_miscelm_to_rpages(miscelm);
        ret = (void *)((uintptr_t)rpages + (uintptr_t)bin_info->reg0_offset +
           (uintptr_t)(bin_info->reg_interval * regind));
        run->nfree--;
        return (ret);
}
```

其中 bitmap_sfu()返回 bitmap 中第一个 1 的位置，并且将该位置 0。

接下来就是通过 run 找对应的 miscelm，再通过 miscelm 找到 run 对应的 page，它的起始位置 rpages。run 找对应的 miscelm，只需要减去 run 在 miscelm 的偏移就可以了。

```
size_t pageind = ((uintptr_t)miscelm - ((uintptr_t)chunk +
        map_misc_offset)) / sizeof(arena_chunk_map_misc_t) + map_bias;
```

通过 miscelm 的地址减去 map_misc 的开始地址（chunk + map_misc_offset）得到该 page 的偏移地址，然后除以 sizeof(arena_chunk_map_misc_t)就得到当前存储 page 的序号，再加上 map_bias，就得到该 page 在当前 chunk 中的绝对 page 号码，然后根据这个绝对 page 号很容易得到该 page 的首地址 rpages。

rpages 的值+regind*reg_interval（同 reg_size），就能得出这个空闲的 region 的实际地址了。
最后再将 run 的 nfree 减一，整个内存申请过程就结束了。

7.1.3. 从 bin 的 runs，arena->runs_avail[i],new chunk 的分配过程

如果 runcur 已分配完成，则是 arena_bin_malloc_hard 的分配过程。选择 run 的顺序是 bin 的 runs，arena->runs_avail[i],new chunk。
如果当前 bin runcur 已满，则调用 arena_bin_malloc_hard 选择一个 run 作为 runcur，再调用 arena_run_reg_alloc 进行分配。

1) 选择 run 的过程，从 bin->runs 中选择地址最低的 run。函数为 arena_bin_nonfull_run_tryget。arena_bin_nonfull_run_tryget 通过 ph 配对堆找到地址最小的 run，比较方式是先按 run 所在 chunk 的 en_sn 比较，如果两个 run 在同一个 chunk，则按 run 的地址比较。

2) 若 bin->runs 为空，则从 arena->runs_avail 中找空间，分配新的 run。函数路径为 arena_run_alloc_small_helper/ arena_run_first_best_fit。

pind2sz_tab 包含了从 4096 开始的每一个 4096 的倍数的 region size 都有保存。

index	size= pind2sz(index)	psz2ind(size-4096+1)
0	4096	0
1	8192	1
2	12288	2
3	16384	3
4	20480	4
5	24576	5
6	28672	6
7	32768	7
8	40960	8
9	49152	9
10	57344	10
…	…	…

run_quantize_ceil 对于 small runsize 和 large 的 size 来说，这个函数的处理结果输入和输出都是相同的。
psz2ind 找到这个 run size 所在的 pind，然后从 arena->runs_avail[pind]开始找 run。
如果找到的 run 的 size 大于需要的值，需要用 arena_run_split_small 把后面多余的 pages 分开，重新插入回 arena->runs_avail，这次插入回的 pind 是根据剩余的 pages 大小找到对应的 pind。所以 arena->runs_avail[]是随时都根据 run 的 size 有序排列的。

3) 如果 arena->runs_avail 也空间不够了，只好重新弄个 chunk，分出所需空间，剩余部分放入 arena->runs_avail 。chunk 的分配过程参考 chunk 分配过程。

7.1.4. 从 arena bin 的分配 debug 过程

arena 是 jemalloc 的总的管理块，一个进程中可以有多个 arena，arena 的最大个可以通过静态变量 narenas_auto。

```
(gdb) p je_narenas_auto
$52 = 2
```

可通过静态数组 arenas 获取进程中所有 arena 的指针：

```
(gdb) p *je_arenas@3
$53 = {0x7ce5c00140, 0x7ce5c8fc00, 0x0}
```

```
/* bins is used to store trees of free regions. */
(gdb) p (*je_arenas[0])->bins[5]
$30 = {lock = {lock = {__private = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}},
    witness = {name = 0x0, rank = 0, comp = 0x0, link = {qre_next = 0x0,
        qre_prev = 0x0}}}, runcur = 0x7dc4606788, runs = {ph_root = 0x0},
  stats = {nmalloc = 552, ndalloc = 223, nrequests = 841, curregs = 329,
    nfills = 138, nflushes = 47, nruns = 2, reruns = 6, curruns = 2}}
```

可以通过变量 arena_bin_info 获取对应 bin 的其他信息：

```
(gdb) p je_arena_bin_info[5]
$32 = {reg_size = 80, redzone_size = 0, reg_interval = 80, run_size = 20480,
  nregs = 256, bitmap_info = {nbits = 256, ngroups = 4}, reg0_offset = 0}
```

```
/*
 * Current run being used to service allocations of this bin's size
 * class.
 */
(gdb) p /x *(*je_arenas[0])->bins[5].runcur
$31 = {binind = 0x5, nfree = 0xb7, bitmap = {0x8300001000000000,
    0xfffffffffffb000, 0xffffffffffffffff, 0xffffffffffffffff, 0x0, 0x0,
    0x0, 0x0}}
```

其中，nfree 表示的是当前 run 中空闲的 region 个数。ngroups=4 表示 bitmap 里有 4 个 bitmap 是有效的。

7.2. small size 内存的释放过程

7.2.1. 释放内存到 tcache 的过程

如果 tcache 打开，并且 tbin->ncached < tbin_info->ncached_max，也就是 tcache 没有满时，直接保存在 tcache 中。

tbin->ncached++;

*(tbin->avail - tbin->ncached) = ptr;

7.2.2. 如果 tcache 已满，释放一半 tcache 回各自的 run

如果 cache 的数量达到了 ncached_max 值，需要进行 flush 回收。释放从栈底开始的一半的 tcache。在释放过程中，每次总是挑选栈底 region 所在的 arena 的 region 先进行释放。如果非当前 arena 的 region，则先保存在栈底，在下一个循环中释放。

当原来全用完的 run，现在有一个 region 可分配了，将其插入所属 bin 中，供分配。如果该 run 比 runcur 小时，需要调整 runcur，要不直接插入 bin->runs 就行。

7.2.3. 如果 run 已满，释放回 arena->runs_avail[i]

如果 bin_info->nregs！=1 时，把 run 从 bin->runs 中移除。然后再释放 run。
如果 bin_info->nregs ==1 时，为什么不移除呢？因为如果 run 只有一个 run 时，永远不会插入非满 runs 树中。

```
/*
 * The following block's conditional is necessary because if the
 * run only contains one region, then it never gets inserted
 * into the non-full runs tree.
 */
```

释放 run 时，首先尝试合并前后未分配 run，合并后再重新插入 arena->runs_avail[]。

7.2.4. 如果 chunk 已满，释放 chunk 回 arena->spare

释放 chunk 的过程，首先从 arena->runs_avail[]移除这个 chunk 的 run，然后把 chunk 从 arena->achunks 中删除，再把这个 chunk 保存在 arena->spare 中。

7.2.5. 如果 arena->spare 已保存前面释放的 chunk，则释放 spare 的 chunk

首先进行 chunk 和 node 的去注册，也就是从 chunks_rtree 删除它们的索引关系。
arena->chunks_szsnad_cached， arena->chunks_ad_cached，arena_maybe_purge

7.3. 从 arena 中分配 large size 内存的过程

Large region size 区间为：LARGE_MINCLASS= 16384 = 4pages <= size <= 448pages = 1835008 = large_maxclass。
Large 的分配 run 和 map_misc 只作为地址使用，arena->runs_avail[]里保存的是 map_misc，而不像 small，真的有 arena_run_s，因为这个只是给 small region 用的。

7.3.1. tcache 分配过程

当 size 区间在 LARGE_MINCLASS=16384<=size <= tcache_maxclass=65536 时，如果 enable 了 tcache，使用 tcache_alloc_large 进行分配。

首先从 tcache 中满足。函数为 arena_malloc/tcache_alloc_large，两点说明：
tcache 只涵盖一部分的 large allocation 请求（size <= tcache_maxclass=65536）对应的 tcache_bin 为空时，不进行填充，而是走非 tcache 分配。这点与 small allocation 的情形是不同的，因为 large 的内存比较大，如果申请了不使用有点浪费。对于 large，只有在释放 large 内存的时候会被缓存到 tcache 中。

7.3.2. 非 tcache 的分配过程

如果没有 enable tcache，或者 1835008>=size>65536 则用 arena_malloc_hard/arena_malloc_large 进行分配。所以从这里的大小可以看出，large 的最大空间是小于一个 chunk 的最大可用空间 2043904=499pages=499*4K。
large_maxclass = index2size(size2index(chunksize)-1);

1) Cache oblivious 时的伪随机数生成

```
/*
 * Compute a uniformly distributed offset within the first page
 * that is a multiple of the cacheline size, e.g. [0 .. 63) * 64
 * for 4 KiB pages and 64-byte cachelines.
*/
r = prng_lg_range_zu(&arena->offset_state, LG_PAGE -
```

LG_CACHELINE, false);
random_offset = ((uintptr_t)r) << LG_CACHELINE;
这里 r 取的是高 6 位的伪随机数，取高位值是为了保证更长的循环周期，这是 PRNG 的实现算法。然后左移 6 位，所以 random_offset 的值是小于 4K 的一个伪随机值。

```
/*
 * If defined, explicitly attempt to more uniformly distribute large allocation
 * pointer alignments across all cache indices.
 */
#define JEMALLOC_CACHE_OBLIVIOUS
```

In computing, a cache-oblivious algorithm (or cache-transcendent algorithm) is an algorithm designed to take advantage of a CPU cache without having the size of the cache (or the length of the cache lines, etc.) as an explicit parameter.
cache-oblivious 算法是对于 CPU cache 没有明确参数的一种优化。因为这个 offset 的存在，所以在分配 large 的时候需要增加 large_pad 的内存分配，这个大小是一个 page，4K。

2) 调用 arena_run_alloc_large 从 arena->runs_avail 中找空间，分配新的 run。函数路径为 arena_run_alloc_large_helper/ arena_run_first_best_fit。这个和 small 的分配方式类似，arena_run_first_best_fit 函数是一样的，只是参数 size 大小不一样。

pind2sz_tab 包含了从 4096 开始的每一个 4096 的倍数的 region size 都有保存。

index	size= pind2sz(index)	psz2ind(size-4096+1)
0	4096	0
1	8192	1
2	12288	2
3	16384	3
4	20480	4
5	24576	5
6	28672	6
7	32768	7
8	40960	8
9	49152	9
10	57344	10
…	…	…

run_quantize_ceil 对于 small runsize 和 large 的 size 来说，这个函数的处理结果输入和输出都是相同的。
psz2ind 找到这个 run size 所在的 pind，然后从 arena->runs_avail[pind]开始找 run。

如果找到的 run 的 size 大于需要的值，需要用 arena_run_split_large 把后面多余的 pages 分开，重新插入回 arena->runs_avail，这次插入回的 pind 是根据剩余的 pages 大小找到对应的 pind。所以 arena->runs_avail[]是随时都根据 run 的 size 有序排列的。

3) 如果 arena->runs_avail 也空间不够了，只好重新分配一个 chunk，chunk 的分配过程参考 chunk 分配过程。然后调用 arena_run_split_large/ arena_run_split_large_helper 分出所需空间，arena_run_split_remove 先在 arena->runs_avail 中删除原来的 run，然后写入剩余部分的第一页和最后一页的 map_bits 值，再调用 arena_avail_insert 把剩余部分放入 arena->runs_avail[]，然后再设置分出来分配的 run 的 map_bits，因为大小减小了 。

7.4. large size 内存的释放过程

large 内存和 small 有点不同的是，small 内存的一个 run 可能包含多个 region，large 内存的 run 都只包含一块内存。其实没有 run 的概念，run 只是用来定位地址用。

7.4.1. 释放 large 内存到 tcache 的过程

如果 tcache 打开，并且 tbin->ncached < tbin_info->ncached_max，也就是 tcache 没有满时，直接保存在 tcache 中。

```
tbin->ncached++;
*(tbin->avail - tbin->ncached) = ptr;
```

7.4.2. 如果 tcache 已满，释放一半 tcache 回 arena->runs_avail[i]

如果 cache 的数量达到了 ncached_max 值，需要进行 flush 回收。释放从栈底开始的一半的 tcache。在释放过程中，每次总是挑选栈底 large 内存所在的 arena 先进行释放。如果非当前 arena 的 run，则先保存在栈底，在下一个循环中释放。
释放 run 时，首先尝试合并前后未分配 run，合并后再重新插入 arena->runs_avail[]。

7.4.3. 如果 chunk 已满，释放 chunk 回 arena->spare

释放 chunk 的过程，首先从 arena->runs_avail[]移除这个 chunk 的 run，然后把 chunk 从 arena->achunks 中删除，再把这个 chunk 保存在 arena->spare 中。

7.4.4. 如果 arena->spare 已保存前面释放的 chunk，则释放 spare 的 chunk

首先进行 chunk 和 node 的去注册，也就是从 chunks_rtree 删除它们的索引关系。

arena->chunks_szsnad_cached， arena->chunks_ad_cached，arena_maybe_purge

7.5. 从 arena 中分配 huge size 内存的过程

huge size 区间为： HUGE_MAXCLASS > size > 448pages = 1835008 = large_maxclass。

#define HUGE_MAXCLASS ((((size_t)1) << 62) + (((size_t)3) << 60))

huge_malloc/arena_chunk_alloc_huge 首先会尝试 cache 分配 chunk 地址，如果没有可复用的 chunk 地址则用 arena_chunk_alloc_huge_hard 分配新的地址。

chunk_alloc_cache/chunk_recycle 在 chunks_szsnad_cached 和 chunks_ad_cached 中重复利用先前分配的 chunk 空间。

```
/*
 * Trees of chunks that were previously allocated (trees differ only in
 * node ordering).  These are used when allocating chunks, in an attempt
 * to re-use address space.  Depending on function, different tree
 * orderings are needed, which is why there are two trees with the same
 * contents.
 */
extent_tree_t        chunks_szsnad_cached;
extent_tree_t        chunks_ad_cached;
extent_tree_t        chunks_szsnad_retained;
extent_tree_t        chunks_ad_retained;
```

huge chunk 的 extent_node_s 会保存在 arena->huge 中。

extent_node_s 和 chunk 在一起的吗？不在一起，没有办法在一起，因为 chunk 是 2M 对齐的，实际 huge 的 chunk 大小都超过 2M，大小 2M 取整，所以没有空间存放 extent_node_s。所以这里通过 register 的方式，把它们两个值关联，方便以后通过 chunk 地址找到 extent_node_s 地址。实际通过 radix tree 查找。 参考 radix tree 基数树。

7.6. huge size 内存的释放过程

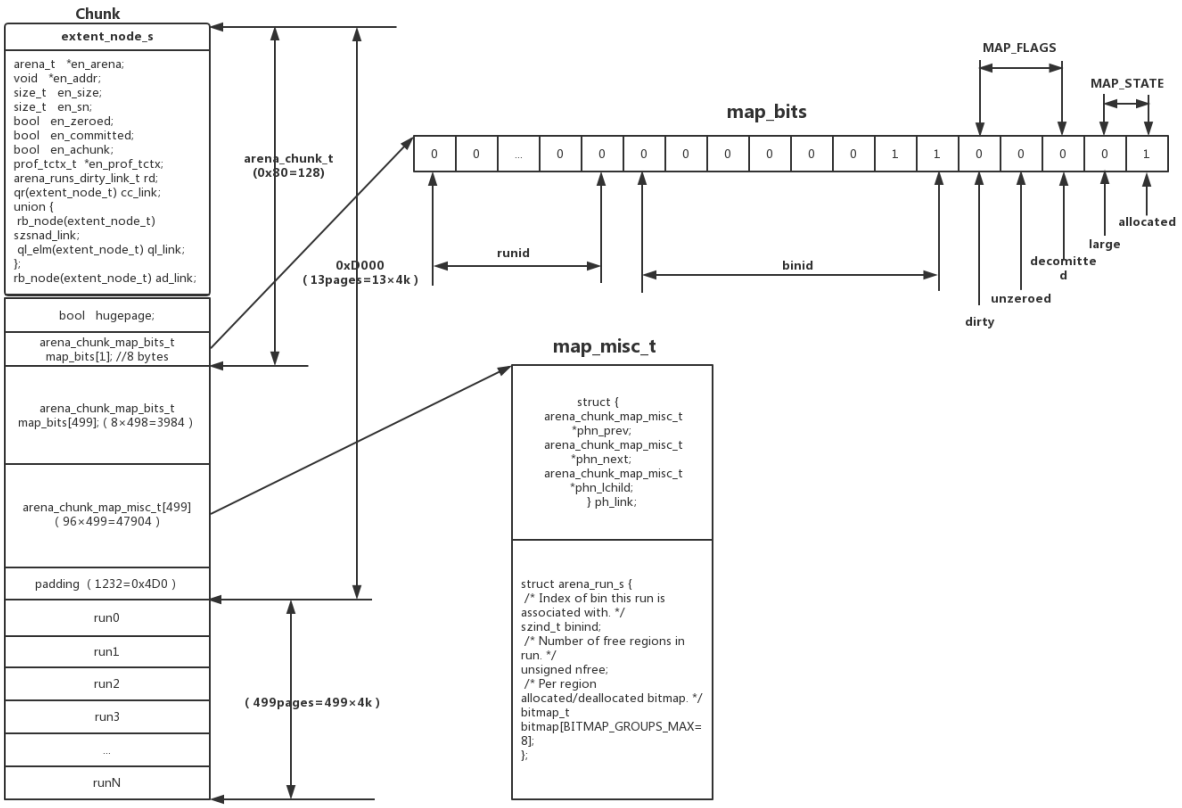
huge_dalloc(tsdn, ptr);

首先进行 chunk 和 node 的去注册，也就是从 chunks_rtree 删除它们的索引关系。然后从 arena->huge 删除 node。

arena->chunks_szsnad_cached， arena->chunks_ad_cached，arena_maybe_purge

8. Chunk

8.1. Chunk 结构 （9832E 64 位系统）



8.2. Chunk 分配过程

```
static arena_chunk_t *
arena_chunk_alloc(tsdn_t *tsdn, arena_t *arena)
{
        arena_chunk_t *chunk;
```

```
        if (arena->spare != NULL)
                chunk = arena_chunk_init_spare(arena);
        else {
                chunk = arena_chunk_init_hard(tsdn, arena);
                if (chunk == NULL)
                        return (NULL);
        }

        ql_elm_new(&chunk->node, ql_link);
        ql_tail_insert(&arena->achunks, &chunk->node, ql_link);
        arena_avail_insert(arena, chunk, map_bias, chunk_npages-map_bias);

        return (chunk);
}
```

如果没有 spare，调用 arena_chunk_init_hard 分配 chunk，并且初始化 map_bits，chunk 被初始化为最大的一个未用的 run。arena_maxrun=2043904=499×4K。
然后把 chunk 存入当前 arena 的 arena->achunks 链表中。
最后把 chunk 插入 pind=31 的 arena->runs_avail[31]中。

```
(gdb) p je_pind2sz(31)
$147 = 2097152
(gdb) p je_pind2sz(30)
$148 = 1835008
```

2043904 间与 1835008 和 2097152 之间，这里取高一位的 pind=31。也就是说 pind=31 保存的 run 是小于或者等于 2097152 的 run。但要大于 1835008。
下面是 arena->runs_avail 的说明，也是这个意思。

```
        /*
         * Size-segregated address-ordered heaps of this arena's available runs,
         * used for first-best-fit run allocation.  Runs are quantized, i.e.
         * they reside in the last heap which corresponds to a size class less
         * than or equal to the run size.
         */
```

arena_chunk_init_hard 初始化过程：

```
assert((mapbits & CHUNK_MAP_DECOMMITTED) == 0 || (mapbits &
            (CHUNK_MAP_DIRTY|CHUNK_MAP_UNZEROED)) == 0);
```

如果是 decommitted，那么 dirty=0，unzeroed=0，也就是说回收后，一定要清 0 和 dirty 去标志。

```
JEMALLOC_ALWAYS_INLINE size_t
arena_mapbits_size_encode(size_t size)
{
        size_t mapbits;

#if CHUNK_MAP_SIZE_SHIFT > 0
        mapbits = size << CHUNK_MAP_SIZE_SHIFT;
#elif CHUNK_MAP_SIZE_SHIFT == 0
        mapbits = size;
#else
        mapbits = size >> -CHUNK_MAP_SIZE_SHIFT;
#endif

        assert((mapbits & ~CHUNK_MAP_SIZE_MASK) == 0);
        return (mapbits);
}
```

CHUNK_MAP_SIZE_SHIFT=1，所以 size 还需要左移 1 位，因为 map_bits 后面 13 位是用来表示 binid 和一些 flag 标志位的，而 size 只有后 12 位为 0，左移 1 位后确保后 13 位为 0，不影响 map_bits 的 flag 标志位。

```
arena_mapbits_unallocated_set(chunk, map_bias, arena_maxrun,
            flag_unzeroed | flag_decommitted);
```

这个会把第 13 页的 map_bits 的 runid 域写上最大的 run 的 size，arena_maxrun=2043904=499×4K，还把第 13 页的 map_bits 的 bindid 域写上 BININD_INVALID，也就是 0xFFU。然后再置当前的 flag_unzeroed 和 flag_decommitted 标志。

```
arena_mapbits_unallocated_set(chunk, chunk_npages-1, arena_maxrun,
            flag_unzeroed);
```

这个会把第 511 页的 map_bits 的 runid 域写上最大的 run 的 size，arena_maxrun=2043904=499×4K，还把第 511 页的 map_bits 的 bindid 域写上 BININD_INVALID，也就是 0xFFU。然后再置当前的 flag_unzeroed 标志。

8.3. 根据内存地址对 chunk 头的进一步分析

(gdb) p (*je_arenas[0])->achunks
$174 = {qlh_first = 0x75de200000}
(gdb) p ((extent_node_t *)0x75de200000)->ql_link.qre_next
$175 = (extent_node_t *) 0x75dcc00000
(gdb) p ((extent_node_t *)0x75dcc00000)->ql_link.qre_next
$176 = (extent_node_t *) 0x75db800000
(gdb) p ((extent_node_t *)0x75db800000)->ql_link.qre_next
$177 = (extent_node_t *) 0x75de200000

(gdb) p (*je_arenas[1])->achunks
$166 = {qlh_first = 0x75dd400000}
(gdb) p ((extent_node_t *)0x75dd400000)->ql_link.qre_next
$168 = (extent_node_t *) 0x75dd400000
当前 arena 里只有一个 chunk。

通过下面调试得到的内存地址为：

```
void *p=Zos_SysStrAlloc("1234567890abcdefghijklmnopqrst");
ZOS_LOG_DBG(ZOS_LOGID, "evers debug=%s", p);
Zos_Free(p);
```

(gdb) p p
$172 = (void *) 0x75dd413670

所以这个是 chunk 地址为 0x75dd400000 的内存，通过它的偏移计算它所在的 page 号如下：

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

pageind=19，

(gdb) p /x ((arena_chunk_t *)0x75dd400000)->map_bits[19-13]->bits

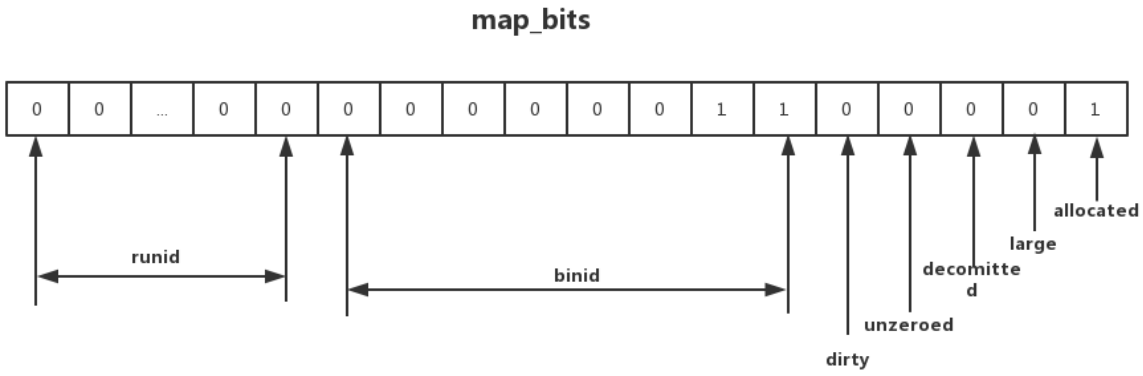
$182 = 0x61

chunk 的 mapbits 的各个位的含义如下：

#define CHUNK_MAP_ALLOCATED ((size_t)0x01U)

```
#define    CHUNK_MAP_LARGE          ((size_t)0x02U)
#define    CHUNK_MAP_STATE_MASK     ((size_t)0x3U)
#define    CHUNK_MAP_DECOMMITTED    ((size_t)0x04U)
#define    CHUNK_MAP_UNZEROED       ((size_t)0x08U)
#define    CHUNK_MAP_DIRTY          ((size_t)0x10U)
#define    CHUNK_MAP_FLAGS_MASK     ((size_t)0x1cU)
#define    CHUNK_MAP_BININD_SHIFT   5
#define    BININD_INVALID           ((size_t)0xffU)
#define    CHUNK_MAP_BININD_MASK    (BININD_INVALID <<
CHUNK_MAP_BININD_SHIFT)
#define    CHUNK_MAP_BININD_INVALID  CHUNK_MAP_BININD_MASK
#define    CHUNK_MAP_RUNIND_SHIFT   (CHUNK_MAP_BININD_SHIFT + 8)
#define    CHUNK_MAP_SIZE_SHIFT     (CHUNK_MAP_RUNIND_SHIFT - LG_PAGE)
#define    CHUNK_MAP_SIZE_MASK      (~(CHUNK_MAP_BININD_MASK |
CHUNK_MAP_FLAGS_MASK | CHUNK_MAP_STATE_MASK))
```

对应 0x61：



runind=0, binind=3，由 binind=3 得到该 bin_info 如下：reg_size = 48, redzone_size = 0, reg_interval = 48, run_size = 12288, nregs = 256, bitmap_info = {nbits = 256, ngroups = 4}, reg0_offset = 0

reg_size=48，和实际分配的大小 39 匹配。前面讲过 run 是由多个 page 组成，这里的 runind 就是当前 page 在其所属 run 中的索引。runind = 0 表示当前 page 是其所属 run 中的第一个 page。由 run_size = 12288，所以该 run 是由 3 个 pages 组成的。我们再来看看下面两个 page 信息，验证一下：

(gdb) p /x ((arena_chunk_t *)0x75dd400000)->map_bits[20-13]->bits

$183 = 0x2061，得到 runid=1

(gdb) p /x ((arena_chunk_t *)0x75dd400000)->map_bits[21-13]->bits

$184 = 0x4061，得到 runid=2

(gdb) p /x ((arena_chunk_t *)0x75dd400000)->map_bits[22-13]->bits

$185 = 0x81，为 binind=4 的 run 了。

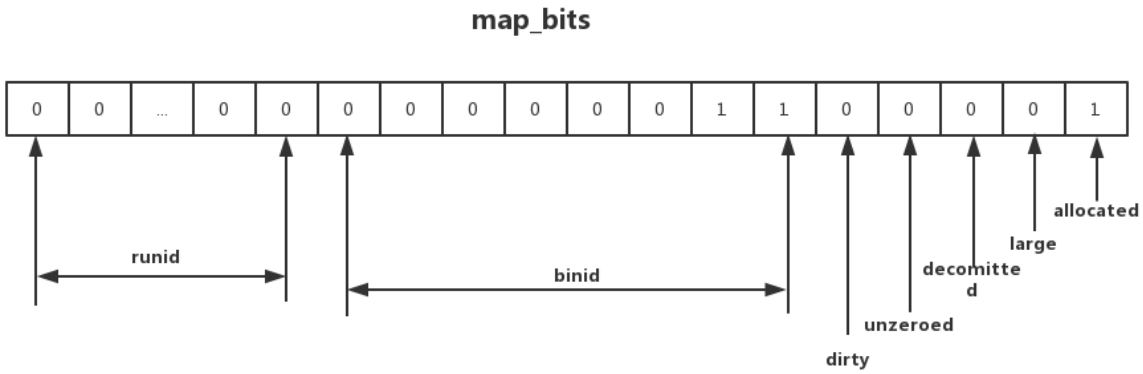
8.4. map_bits

```
/* Each element of the chunk map corresponds to one page within the chunk. */
struct arena_chunk_map_bits_s {
        /*
         * Run address (or size) and various flags are stored together.  The bit
         * layout looks like (assuming 32-bit system):
         *
         *   ???????? ???????? ???nnnnn nnndumla
         *
         * ? : Unallocated: Run address for first/last pages, unset for internal
         *                  pages.
         *     Small: Run page offset.
```

```
*     Large: Run page count for first page, unset for trailing pages.
* n : binind for small size class, BININD_INVALID for large size class.
* d : dirty?
* u : unzeroed?
* m : decommitted?
* l : large?
* a : allocated?
*
* Following are example bit patterns for the three types of runs.
*
* p : run page offset
* s : run size
* n : binind for size class; large objects set these to BININD_INVALID
* x : don't care
* - : 0
* + : 1
* [DUMLA] : bit set
* [dumla] : bit unset
*
*   Unallocated (clean):
*     ssssssss ssssssss sss+++++ +++dum-a
*     xxxxxxxx xxxxxxxx xxxxxxxx xxx-Uxxx
*     ssssssss ssssssss sss+++++ +++dUm-a
*
*   Unallocated (dirty):
*     ssssssss ssssssss sss+++++ +++D-m-a
*     xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
*     ssssssss ssssssss sss+++++ +++D-m-a
*
*   Small:
*     pppppppp pppppppp pppnnnnn nnnd---A
*     pppppppp pppppppp pppnnnnn nnn----A
*     pppppppp pppppppp pppnnnnn nnnd---A
*
*   Large:
*     ssssssss ssssssss sss+++++ +++D--LA
```

```
         *     xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
         *     -------- -------- ---+++++ +++D--LA
         *
         *   Large (sampled, size <= LARGE_MINCLASS):
         *     ssssssss ssssssss sssnnnnn nnnD--LA
         *     xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
         *     -------- -------- ---+++++ +++D--LA
         *
         *   Large (not sampled, size == LARGE_MINCLASS):
         *     ssssssss ssssssss sss+++++ +++D--LA
         *     xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
         *     -------- -------- ---+++++ +++D--LA
         */
        size_t                          bits;
};
```

Map_bits 为 0x61 的结构图如下：



runind=0, binind=3，由 binind=3 得到该 bin_info 如下：reg_size = 48, redzone_size = 0, reg_interval = 48, run_size = 12288, nregs = 256, bitmap_info = {nbits = 256, ngroups = 4}, reg0_offset = 0

前面讲过 run 是由多个 page 组成，这里的 runind 就是当前 page 在其所属 run 中的索引。runind = 0 表示当前 page 是其所属 run 中的第一个 page。由 run_size = 12288，所以该 run 是由 3 个 pages 组成的。我们再来看看下面两个 page 信息，验证一下：

(gdb) p /x ((arena_chunk_t *)0x75dd400000)->map_bits[20-13]->bits

$183 = 0x2061，得到 runid=1
(gdb) p /x ((arena_chunk_t *)0x75dd400000)->map_bits[21-13]->bits
$184 = 0x4061，得到 runid=2
(gdb) p /x ((arena_chunk_t *)0x75dd400000)->map_bits[22-13]->bits
$185 = 0x81，为 binind=4 的 run 了。

8.5. map_misc_t

```
struct arena_chunk_map_misc_s {
        phn(arena_chunk_map_misc_t)             ph_link;
        union {
                /* Linkage for list of dirty runs. */
                arena_runs_dirty_link_t         rd;
                /* Profile counters, used for large object runs. */
                union {
                        void                    *prof_tctx_pun;
                        prof_tctx_t             *prof_tctx;
                };
                /* Small region run metadata. */
                arena_run_t                     run;
        };
};

struct {
        arena_chunk_map_misc_t *phn_prev;
        arena_chunk_map_misc_t *phn_next;
        arena_chunk_map_misc_t *phn_lchild;
} ph_link;
(gdb) p sizeof(arena_chunk_map_misc_t)
$102 = 96
(gdb) p sizeof(arena_run_t)
$103 = 72
```

所以这里 arena_chunk_map_misc_t 的大小就是 arena_run_t 的大小 72 加上 3 个 phn 指针共 24 字节。
所以这里 misc 对于 small region 来说就是存了 run 信息和 run 前后的链接指针。

```
struct arena_run_s {
        /* Index of bin this run is associated with. */
        szind_t         binind;
        /* Number of free regions in run. */
        unsigned        nfree;
        /* Per region allocated/deallocated bitmap. */
        bitmap_t        bitmap[BITMAP_GROUPS_MAX=8];
};
```

misc = (arena_chunk_map_misc_t *)((uintptr_t)chunk +(uintptr_t)map_misc_offset) + (pageind - runind)- map_bias)

```
JEMALLOC_ALWAYS_INLINE arena_chunk_map_misc_t *
arena_miscelm_get_mutable(arena_chunk_t *chunk, size_t pageind)
{
        assert(pageind >= map_bias);
        assert(pageind < chunk_npages);
        return ((arena_chunk_map_misc_t *)((uintptr_t)chunk +
           (uintptr_t)map_misc_offset) + pageind-map_bias);
}
```

在 small region 的情况下，misc 是由用于连接 arena 中的 ph 树的 ph_link 和 arena_run_t 结构的 run 组成。

arena 的 bins 中 runcur 就是指向某个 chunk 中的某个 arena_chunk_map_misc_t[]中的某个 run。

运行时的数据如下：

```
(gdb) p /x *((arena_chunk_map_misc_t *)0x75dd401010 + 19 - 13)
$187 = {ph_link = {phn_prev = 0x0, phn_next = 0x0, phn_lchild = 0x0}, {rd = {
    rd_link = {qre_next = 0xc100000003, qre_prev = 0x8153400000000000}}, {
    prof_tctx_pun = 0xc100000003, prof_tctx = 0xc100000003}, run = {
    binind = 0x3, nfree = 0xc1, bitmap = {0x8153400000000000,
      0xfffffffffffff8ce, 0xffffffffffffffff, 0xffffffffffffffff, 0x0, 0x0,
      0x0, 0x0}}}}
(gdb) p /x *((arena_chunk_map_misc_t *)0x75dd401010 + 20 - 13)
$188 = {ph_link = {phn_prev = 0x0, phn_next = 0x0, phn_lchild = 0x0}, {rd = {
    rd_link = {qre_next = 0x0, qre_prev = 0x0}}, {prof_tctx_pun = 0x0,
```

```
    prof_tctx = 0x0}, run = {binind = 0x0, nfree = 0x0, bitmap = {0x0, 0x0,
      0x0, 0x0, 0x0, 0x0, 0x0, 0x0}}}}
(gdb) p /x *((arena_chunk_map_misc_t *)0x75dd401010 + 21 - 13)
$189 = {ph_link = {phn_prev = 0x0, phn_next = 0x0, phn_lchild = 0x0}, {rd = {
    rd_link = {qre_next = 0x0, qre_prev = 0x0}}, {prof_tctx_pun = 0x0,
    prof_tctx = 0x0}, run = {binind = 0x0, nfree = 0x0, bitmap = {0x0, 0x0,
      0x0, 0x0, 0x0, 0x0, 0x0, 0x0}}}}
```

可以看到第 20，21 个 page 没有 misc 信息，因为它们的 runid 不为 0，也就是不是 run 的第一个 page，所以 run 只把 misc 信息放在它的第一个 page 对应的 misc 位置，所以前面查找 misc 的时候，需要减去 runind：pageind – runind，这样就会直接去找到第一个 page 对应的 misc。所以也说明实际空闲的 misc 大量存在，只要 run_size 超过 4K，就会有空闲的 misc。这里也可以看到 binind = 0x3。

因此，arena_chunk_map_bits_t[499] 是描述 chunk 内的 499 个 page 的，而 arena_chunk_map_misc_t[499]是描述 run 的，一般多个 page 组成一个 run，也就是说一个 chunk 中 run 的个数远远小于 499 个，所以 arena_chunk_map_misc_t[499]中很多数据都是空的。

8.6. 从 arena 找到 chunk 地址

有 arena 指针地址，怎么找到 chunk 地址？根据 stats 统计，应该是可以找到的，需要分析一下，通过具体内存然后进行 2M 对其是一种找法，希望直接找到系统当前所有的 chunk 地址。

```
/* Extant arena chunks. */
ql_head(extent_node_t)     achunks;
```

```
(gdb) p (*je_arenas[0])->achunks
$174 = {qlh_first = 0x75de200000}
(gdb) p ((extent_node_t *)0x75de200000)->ql_link.qre_next
$175 = (extent_node_t *) 0x75dcc00000
(gdb) p ((extent_node_t *)0x75dcc00000)->ql_link.qre_next
$176 = (extent_node_t *) 0x75db800000
(gdb) p ((extent_node_t *)0x75db800000)->ql_link.qre_next
$177 = (extent_node_t *) 0x75de200000
```

所以当前 arena 中有三个 chunk。

(gdb) p (*je_arenas[1])->achunks
$166 = {qlh_first = 0x75dd400000}
(gdb) p ((extent_node_t *)0x75dd400000)->ql_link.qre_next
$168 = (extent_node_t *) 0x75dd400000
当前 arena 里只有一个 chunk。

8.7. Chunk 的 register 过程

把 chunk 的 node 地址存入 chunks_rtree，chunks_rtree 是 radix tree，chunk 是索引值，node 是 value，为以后通过 chunk 地址查找 node 做准备。详细过程参考 radix tree。

9. jemalloc 存储块(region、run、chunk)

9.1. 存储块(region、run、chunk)

按存储单元的块大小分，有 region、run、chunk 三种存储块。最小的单元是 region，它的大小是 8 字节~14KB=14336，1 个或多个相同大小的 region 组成一个 run。run 的大小必须是页(4kB)的整数倍，相同大小 region 对应的 run 的大小总是相同的，一个 bin 对应的 region 一样，所以一个 bin 可能有多个 run，多个 run 组成一个 chunk。

chunk 的大小固定是 2M（或 4M，可配置），并且是 2M 字节对齐的。一个 chunk 的大小是 2M = 512*4K，也就是 512 个 page。其中，前 map_bias 个 page 用于管理块 Header，后 512- map_bias 个 page 是实际可使用的内存。(map_bias 大小的详细计算过程参考 map_bias 的初始化，在 64 位的系统中 4.4.0 版本的这个值是 13。)这 512-map_bias 个 page 又被划分为若干个 run，每个 run 又由若干的 page 组成。

每个 bin 的各自组成 run 的 region 数量是 arena_bin_info 中的 nregs。如第二个 small bin 的 arena_bin_info 信息如下：
{reg_size = 16, redzone_size = 0, reg_interval = 16, run_size = 4096, nregs = 256, bitmap_info = {nbits = 256, ngroups = 4}, reg0_offset = 0},
第二个 small bin 的 region size 是 16 字节，对应的 run size 是 4096，所以包含的 region 个数是 nregs =256 个。

9.2. arena_bin_info

```
arena_bin_info_t    arena_bin_info[NBINS];
```

bin 有 36 个，给 small 用的

```
static void
bin_info_init(void)
{
    arena_bin_info_t *bin_info;
#define    BIN_INFO_INIT_bin_yes(index, size)                \
    bin_info = &arena_bin_info[index];                \
    bin_info->reg_size = size;                    \
    bin_info_run_size_calc(bin_info);                \
    bitmap_info_init(&bin_info->bitmap_info, bin_info->nregs);
#define    BIN_INFO_INIT_bin_no(index, size)
#define    SC(index, lg_grp, lg_delta, ndelta, bin, lg_delta_lookup)    \
    BIN_INFO_INIT_bin_##bin(index, (ZU(1)<<lg_grp) + (ZU(ndelta)<<lg_delta))
    SIZE_CLASSES
#undef BIN_INFO_INIT_bin_yes
#undef BIN_INFO_INIT_bin_no
#undef SC
}
```

bin index 和 size 的对应关系及计算过程参考 index2size 计算过程。

```
(gdb) p je_arena_bin_info
$8 = {{
  reg_size = 8, redzone_size = 0, reg_interval = 8, run_size = 4096, nregs = 512, bitmap_info = {nbits = 512, ngroups = 8}, reg0_offset = 0}, {
  reg_size = 16, redzone_size = 0, reg_interval = 16, run_size = 4096, nregs = 256, bitmap_info = {nbits = 256, ngroups = 4}, reg0_offset = 0}, {
  reg_size = 32, redzone_size = 0, reg_interval = 32, run_size = 4096, nregs = 128, bitmap_info = {nbits = 128, ngroups = 2}, reg0_offset = 0}, {
  reg_size = 48, redzone_size = 0, reg_interval = 48, run_size = 12288,    nregs = 256, bitmap_info = {nbits = 256, ngroups = 4}, reg0_offset = 0}, {
  reg_size = 64, redzone_size = 0, reg_interval = 64, run_size = 4096,    nregs = 64, bitmap_info = {nbits = 64, ngroups = 1}, reg0_offset = 0}, {
  reg_size = 80, redzone_size = 0, reg_interval = 80, run_size = 20480,    nregs = 256, bitmap_info = {nbits = 256, ngroups = 4}, reg0_offset = 0}, {
  reg_size = 96, redzone_size = 0, reg_interval = 96, run_size = 12288,    nregs = 128, bitmap_info = {nbits = 128, ngroups = 2}, reg0_offset = 0}, {
  reg_size = 112, redzone_size = 0, reg_interval = 112, run_size = 28672,    nregs = 256, bitmap_info = {nbits = 256, ngroups = 4}, reg0_offset = 0}, {
  reg_size = 128, redzone_size = 0, reg_interval = 128, run_size = 4096,    nregs = 32, bitmap_info = {nbits = 32, ngroups = 1}, reg0_offset = 0}, {
```

reg_size = 160, redzone_size = 0, reg_interval = 160, run_size = 20480, nregs = 128, bitmap_info = {nbits = 128, ngroups = 2}, reg0_offset = 0}, {

reg_size = 192, redzone_size = 0, reg_interval = 192, run_size = 12288, nregs = 64, bitmap_info = {nbits = 64, ngroups = 1}, reg0_offset = 0}, {

reg_size = 224, redzone_size = 0, reg_interval = 224, run_size = 28672, nregs = 128, bitmap_info = {nbits = 128, ngroups = 2}, reg0_offset = 0}, {

reg_size = 256, redzone_size = 0, reg_interval = 256, run_size = 4096, nregs = 16, bitmap_info = {nbits = 16, ngroups = 1}, reg0_offset = 0}, {

reg_size = 320, redzone_size = 0, reg_interval = 320, run_size = 20480, nregs = 64, bitmap_info = {nbits = 64, ngroups = 1}, reg0_offset = 0}, {

reg_size = 384, redzone_size = 0, reg_interval = 384, run_size = 12288, nregs = 32, bitmap_info = {nbits = 32, ngroups = 1}, reg0_offset = 0}, {

reg_size = 448, redzone_size = 0, reg_interval = 448, run_size = 28672, nregs = 64, bitmap_info = {nbits = 64, ngroups = 1}, reg0_offset = 0}, {

reg_size = 512, redzone_size = 0, reg_interval = 512, run_size = 4096, nregs = 8, bitmap_info = {nbits = 8, ngroups = 1}, reg0_offset = 0}, {

reg_size = 640, redzone_size = 0, reg_interval = 640, run_size = 20480, nregs = 32, bitmap_info = {nbits = 32, ngroups = 1}, reg0_offset = 0}, {

reg_size = 768, redzone_size = 0, reg_interval = 768, run_size = 12288, nregs = 16, bitmap_info = {nbits = 16, ngroups = 1}, reg0_offset = 0}, {

reg_size = 896, redzone_size = 0, reg_interval = 896, run_size = 28672, nregs = 32, bitmap_info = {nbits = 32, ngroups = 1}, reg0_offset = 0}, {

reg_size = 1024, redzone_size = 0, reg_interval = 1024, run_size = 4096, nregs = 4, bitmap_info = {nbits = 4, ngroups = 1}, reg0_offset = 0}, {

reg_size = 1280, redzone_size = 0, reg_interval = 1280, run_size = 20480, nregs = 16, bitmap_info = {nbits = 16, ngroups = 1}, reg0_offset = 0}, {

reg_size = 1536, redzone_size = 0, reg_interval = 1536, run_size = 12288, nregs = 8, bitmap_info = {nbits = 8, ngroups = 1}, reg0_offset = 0}, {

reg_size = 1792, redzone_size = 0, reg_interval = 1792, run_size = 28672, nregs = 16, bitmap_info = {nbits = 16, ngroups = 1}, reg0_offset = 0}, {

reg_size = 2048, redzone_size = 0, reg_interval = 2048, run_size = 4096, nregs = 2, bitmap_info = {nbits = 2, ngroups = 1}, reg0_offset = 0}, {

reg_size = 2560, redzone_size = 0, reg_interval = 2560, run_size = 20480, nregs = 8, bitmap_info = {nbits = 8, ngroups = 1}, reg0_offset = 0}, {

reg_size = 3072, redzone_size = 0, reg_interval = 3072, run_size = 12288, nregs = 4, bitmap_info = {nbits = 4, ngroups = 1}, reg0_offset = 0}, {

reg_size = 3584, redzone_size = 0, reg_interval = 3584, run_size = 28672, nregs = 8, bitmap_info = {nbits = 8, ngroups = 1}, reg0_offset = 0}, {

reg_size = 4096, redzone_size = 0, reg_interval = 4096, run_size = 4096, nregs = 1, bitmap_info = {nbits = 1, ngroups = 1}, reg0_offset = 0}, {

reg_size = 5120, redzone_size = 0, reg_interval = 5120, run_size = 20480, nregs = 4, bitmap_info = {nbits = 4, ngroups = 1}, reg0_offset = 0}, {

reg_size = 6144, redzone_size = 0, reg_interval = 6144, run_size = 12288, nregs = 2, bitmap_info = {nbits = 2, ngroups = 1}, reg0_offset = 0}, {

reg_size = 7168, redzone_size = 0, reg_interval = 7168, run_size = 28672, nregs = 4, bitmap_info = {nbits = 4, ngroups = 1}, reg0_offset = 0}, {

reg_size = 8192, redzone_size = 0, reg_interval = 8192, run_size = 8192, nregs = 1, bitmap_info = {nbits = 1, ngroups = 1}, reg0_offset = 0}, {

reg_size = 10240, redzone_size = 0, reg_interval = 10240, run_size = 20480, nregs = 2, bitmap_info = {nbits = 2, ngroups = 1}, reg0_offset = 0}, {

reg_size = 12288, redzone_size = 0, reg_interval = 12288, run_size = 12288, nregs = 1, bitmap_info = {nbits = 1, ngroups = 1}, reg0_offset = 0}, {

reg_size = 14336, redzone_size = 0, reg_interval = 14336, run_size = 28672, nregs = 2, bitmap_info = {nbits = 2, ngroups = 1}, reg0_offset = 0}}

(gdb)

9.3. arena_bin_info 中的 bitmap_info

/* Logical number of bits in bitmap (stored at bottom level). */

size_t nbits;

```
/* Number of groups necessary for nbits. */
size_t ngroups;


arenas[0]->bins[5].runcur
struct arena_run_s {
        /* Index of bin this run is associated with. */
        szind_t                 binind;
        /* Number of free regions in run. */
        unsigned        nfree;
        /* Per region allocated/deallocated bitmap. */
        bitmap_t        bitmap[BITMAP_GROUPS_MAX];
};
typedef unsigned long bitmap_t;
```

所以一个 bitmap 是 8 个字节，有 64 个 bits。所以 ngroups= nbits/64，最小为 1，ngroups 相当于 bitmap 的个数。nbits= nregs，也就是说一个 bit 表示一个 reg，也就是 region 是内存分配的最小单位。

10. Jemalloc 的初始化过程

gcc 允许为函数设置__attribute__ ((constructor))和__attribute__ ((destructor))两种属性，顾名思义，就是将被修饰的函数作为构造函数或析构函数。程序员可以通过类似下面的方式为函数设置这些属性：

```
void funcBeforeMain() __attribute__ ((constructor));
void funcAfterMain() __attribute__ ((destructor));
```

也可以放在函数名之前：

```
void __attribute__ ((constructor)) funcBeforeMain();
void __attribute__ ((destructor)) funcAfterMain();
```

带有(constructor)属性的函数将在 main()函数之前被执行，而带有(destructor)属性的函数将在 main()退出时执行。

```
JEMALLOC_ATTR(constructor)
static void
jemalloc_constructor(void)
{

        malloc_init();
}
```

malloc_init->malloc_init_hard-> malloc_tsd_boot0/ malloc_tsd_boot1

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,

```
static bool
malloc_init_hard(void)
{
        tsd_t *tsd;

#if defined(_WIN32) && _WIN32_WINNT < 0x0600
        _init_init_lock();
#endif
        malloc_mutex_lock(TSDN_NULL, &init_lock);
        if (!malloc_init_hard_needed()) {//防止多线程重入函数
                malloc_mutex_unlock(TSDN_NULL, &init_lock);
                return (false);
        }
//pages_boot, base_boot, chunk_boot, ctl_boot, arena_boot, tcache_boot, arena_init 启动和初始化工作
        if (malloc_init_state != malloc_init_a0_initialized &&
            malloc_init_hard_a0_locked()) {
                malloc_mutex_unlock(TSDN_NULL, &init_lock);
                return (true);
        }
```

```
        malloc_mutex_unlock(TSDN_NULL, &init_lock);
        /* Recursive allocation relies on functional tsd. */
        tsd = malloc_tsd_boot0();//boot0 在全局空间创建 TSD
        if (tsd == NULL)
                return (true);
        if (malloc_init_hard_recursible())//set malloc_init_state = malloc_init_recursible
                return (true);
        malloc_mutex_lock(tsd_tsdn(tsd), &init_lock);

        if (config_prof && prof_boot2(tsd)) {
                malloc_mutex_unlock(tsd_tsdn(tsd), &init_lock);
                return (true);
        }
//malloc_init_initialized，arena 数量的初始化和内存的分配
        if (malloc_init_hard_finish(tsd_tsdn(tsd))) {
                malloc_mutex_unlock(tsd_tsdn(tsd), &init_lock);
                return (true);
        }

        malloc_mutex_unlock(tsd_tsdn(tsd), &init_lock);
        malloc_tsd_boot1();//boot1 在线程局部空间创建 TSD
        return (false);
}

typedef enum {
        malloc_init_uninitialized     = 3,
        malloc_init_a0_initialized    = 2,
        malloc_init_recursible                = 1,
        malloc_init_initialized               = 0 /* Common case --> jnz. */
} malloc_init_t;
```

11. Jemalloc 的异常行为？

11.1. Free(P+n)释放部分空间的问题

当调用 je_free 的时候用，不是拿分配的时候的地址，而且有一定的往后偏移的时候，这个时候 jemalloc 会只释放后半部分，前半部分保留。

```
(gdb) list
167             void *p=Zos_SysStrAlloc("1234567890abcdefghijklmnopqrst");
168             ZOS_LOG_DBG(ZOS_LOGID, "evers debug=%s", p);
169             Zos_Free(p);
170
171
(gdb) n
168             ZOS_LOG_DBG(ZOS_LOGID, "evers debug=%s", p);
(gdb) p /x *(char*)(p-8)@39
$139 = {0xea, 0xde, 0x23, 0x1, 0x1e, 0x0, 0x0, 0x0, 0x31, 0x32, 0x33, 0x34,
  0x35, 0x36, 0x37, 0x38, 0x39, 0x30, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66,
  0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x72,
  0x73, 0x74, 0x0}
(gdb) p  *(char*)(p)@30
$140 = "1234567890abcdefghijklmnopqrst"
(gdb) n
169             Zos_Free(p);
(gdb) n
184             fflush(stdout);
(gdb) p  *(char*)(p)@30
$141 = 'Z' <repeats 30 times>
(gdb) p /x *(char*)(p-8)@39
$142 = {0xea, 0xde, 0x23, 0x1, 0x1e, 0x0, 0x0, 0x0, 0x5a <repeats 31 times>}
(gdb)
```

Zos_SysStrAlloc 会分配内存给字符串，并且在前面 8 个字节存在头信息，返回的地址是字符串的首地址，这个时候如果调用 Zos_Free，Zos_Free 会直接调用 je_free，这样只会释放字符串的占有内存空间，前面 8 个字节的内存不会得到释放。如上面调试结果可见，后续被用 0x5a 填充了，前 8 个字节则保持不变。

12. Jemalloc unit test

12.1. Jemalloc 的 unit test 设计

```
#define	TEST_BEGIN(f)				\
static void					\
f(void)					\
{					\
	p_test_init(#f);

#define	TEST_END				\
	goto label_test_end;			\
label_test_end:				\
	p_test_fini();				\
}
```

通过 TEST_BEGIN 和 TEST_END 增加了每个测试 case 的初始化和结束的操作。所以设计 case 的时候都以 TEST_BEGIN 开始，以 TEST_END 结尾。

比如原来的测试用例定义如下：

```
TEST_BEGIN(test_rb_empty)
{
	tree_t tree;
	node_t key;
	……
	key.key = 0;
	key.magic = NODE_MAGIC;
	assert_ptr_null(tree_psearch(&tree, &key), "Unexpected node");
}
TEST_END
```

在宏解开后如下所示：

```
static void
```

```
test_rb_empty(void)
{
        p_test_init(test_rb_empty);

{
        tree_t tree;
        node_t key;
        ……
        key.key = 0;
        key.magic = NODE_MAGIC;
        assert_ptr_null(tree_psearch(&tree, &key), "Unexpected node");
}

        goto label_test_end;
label_test_end:
        p_test_fini();
}
```

具体测试 case 只需要直接调用相关的测试函数就可以了，如下：

```
return (test(
            test_rb_empty,
            test_rb_random));
```

```
test -> p_test ->p_test_impl

static test_status_t
p_test_impl(bool do_malloc_init, test_t *t, va_list ap)
{
        test_status_t ret;

        if (do_malloc_init) {
                /*
                 * Make sure initialization occurs prior to running tests.
                 * Tests are special because they may use internal facilities
                 * prior to triggering initialization as a side effect of
                 * calling into the public API.
```

```
                 */
                if (nallocx(1, 0) == 0) {
                        malloc_printf("Initialization error");
                        return (test_status_fail);
                }
        }

        ret = test_status_pass;
        for (; t != NULL; t = va_arg(ap, test_t *)) {
                t(); //test 传递的函数名参数在这里被调用
                if (test_status > ret)
                        ret = test_status;
        }

        malloc_printf("--- %s: %u/%u, %s: %u/%u, %s: %u/%u ---\n",
           test_status_string(test_status_pass),
           test_counts[test_status_pass], test_count,
           test_status_string(test_status_skip),
           test_counts[test_status_skip], test_count,
           test_status_string(test_status_fail),
           test_counts[test_status_fail], test_count);

        return (ret);
}
```

12.2. RB tree 的 unit test 设计

```
TEST_BEGIN(test_rb_random)
{
#define        NNODES 25
#define        NBAGS 250
#define        SEED 42
        sfmt_t *sfmt;
        uint64_t bag[NNODES];
        tree_t tree;
```

```
node_t nodes[NNODES];
unsigned i, j, k, black_height, imbalances;

sfmt = init_gen_rand(SEED);
for (i = 0; i < NBAGS; i++) {//250 次循环测试
        switch (i) {
        case 0:
                /* Insert in order. *///key 值为正序的测试
                for (j = 0; j < NNODES; j++)
                        bag[j] = j;
                break;
        case 1:
                /* Insert in reverse order. *///key 值为逆序的测试
                for (j = 0; j < NNODES; j++)
                        bag[j] = NNODES - j - 1;
                break;
        default://key 值随机测试
                for (j = 0; j < NNODES; j++)
                        bag[j] = gen_rand64_range(sfmt, NNODES);
        }

        for (j = 1; j <= NNODES; j++) {//1-25 个节点的树的测试
                /* Initialize tree and nodes. */
                tree_new(&tree);
                for (k = 0; k < j; k++) {
                        nodes[k].magic = NODE_MAGIC;
                        nodes[k].key = bag[k];
                }

                /* Insert nodes. */
                for (k = 0; k < j; k++) {//每插入一个节点做一次检测
                        tree_insert(&tree, &nodes[k]);

                        rbtn_black_height(node_t, link, &tree,
                            black_height);
                        imbalances = tree_recurse(tree.rbt_root,
```

```
            black_height, 0);
        assert_u_eq(imbalances, 0,
            "Tree is unbalanced");

        assert_u_eq(tree_iterate(&tree), k+1,
            "Unexpected node iteration count");
        assert_u_eq(tree_iterate_reverse(&tree), k+1,
            "Unexpected node iteration count");

        assert_false(tree_empty(&tree),
            "Tree should not be empty");
        assert_ptr_not_null(tree_first(&tree),
            "Tree should not be empty");
        assert_ptr_not_null(tree_last(&tree),
            "Tree should not be empty");

        tree_next(&tree, &nodes[k]);
        tree_prev(&tree, &nodes[k]);
}

/* Remove nodes. *///删除检测
switch (i % 5) {
case 0:
        for (k = 0; k < j; k++)
                node_remove(&tree, &nodes[k], j - k);
        break;
case 1:
        for (k = j; k > 0; k--)
                node_remove(&tree, &nodes[k-1], k);
        break;
case 2: {
        node_t *start;
        unsigned nnodes = j;

        start = NULL;
        do {
```

```
                                start = tree_iter(&tree, start,
                                    remove_iterate_cb, (void *)&nnodes);
                                nnodes--;
                            } while (start != NULL);
                            assert_u_eq(nnodes, 0,
                                "Removal terminated early");
                            break;
                        } case 3: {
                            node_t *start;
                            unsigned nnodes = j;

                            start = NULL;
                            do {
                                start = tree_reverse_iter(&tree, start,
                                    remove_reverse_iterate_cb,
                                    (void *)&nnodes);
                                nnodes--;
                            } while (start != NULL);
                            assert_u_eq(nnodes, 0,
                                "Removal terminated early");
                            break;
                        } case 4: {
                            unsigned nnodes = j;
                            tree_destroy(&tree, destroy_cb, &nnodes);
                            assert_u_eq(nnodes, 0,
                                "Destruction terminated early");
                            break;
                        } default:
                            not_reached();
                        }
                    }
                }
                fini_gen_rand(sfmt);
#undef NNODES
#undef NBAGS
#undef SEED
```

```
}
```

RB-tree 的 unit test 会测试 250×25=6250 次的树的插入和删除动作，如果当前这次测试树的节点为 10，那么每插入一个节点，都会经过平衡测试，以及前序后序的查找过程。

12.3. 和 unit test 配套的强壮的 assert 设计

```
static arena_chunk_t *
arena_chunk_init_spare(arena_t *arena)
{
        arena_chunk_t *chunk;
        assert(arena->spare != NULL);

        chunk = arena->spare;
        arena->spare = NULL;

        assert(arena_mapbits_allocated_get(chunk, map_bias) == 0);
        assert(arena_mapbits_allocated_get(chunk, chunk_npages-1) == 0);
        assert(arena_mapbits_unallocated_size_get(chunk, map_bias) ==
            arena_maxrun);
        assert(arena_mapbits_unallocated_size_get(chunk, chunk_npages-1) ==
            arena_maxrun);
        assert(arena_mapbits_dirty_get(chunk, map_bias) ==
            arena_mapbits_dirty_get(chunk, chunk_npages-1));

        return (chunk);
}
```

13. Jemalloc 不相关问题

13.1. 32/64 位系统数据类型对应字节数

理论上来讲 我觉得数据类型的字节数应该是由 CPU 决定的，但是实际上主要由编译器决定(占多少位由编译器在编译期间说了算)。

常用数据类型对应字节数
可用如 sizeof（char),sizeof(char*)等得出

32 位编译器：

char ：1 个字节
char*（即指针变量）: 4 个字节（32 位的寻址空间是 2^32, 即 32 个 bit，也就是 4 个字节。同理 64 位编译器）
short int : 2 个字节
int： 4 个字节
unsigned int : 4 个字节
float: 4 个字节
double: 8 个字节
long: 4 个字节
long long: 8 个字节
unsigned long: 4 个字节

64 位编译器：

char ：1 个字节
char*(即指针变量): 8 个字节
short int : 2 个字节
int： 4 个字节
unsigned int : 4 个字节
float: 4 个字节
double: 8 个字节
long: 8 个字节
long long: 8 个字节
unsigned long: 8 个字节

13.2. Assert 的使用

如何定义使用的，这个对于软件的开发很有帮助的。看看 jemalloc 里是怎么使用的。

一共有 614 个 assert。

13.3. 0xFFLU 和 0xFFU

U 是 unsigned 的简写
代表前面的 FF 是无符号数
L 是 long 的简写
代表前面的 FF 是无符号长整形数

13.4. 堆排序

通常堆是通过一维数组来实现的。在数组起始位置为 0 的情形中：
父节点 i 的左子节点在位置 (2i+1);
父节点 i 的右子节点在位置 (2i+2);
子节点 i 的父节点在位置 floor((i-1)/2);

最坏时间复杂度	O(n\log n)
最优时间复杂度	O(n\log n)
平均时间复杂度	O(n\log n)
最坏空间复杂度	O(1) auxiliary

```
#include <stdio.h>
#include <stdlib.h>

void swap(int* a, int* b) {
	int temp = *b;
	*b = *a;
	*a = temp;
}

void max_heapify(int arr[], int start, int end) {
	//建立父節點指標和子節點指標
	int dad = start;
```

```
	int son = dad * 2 + 1;
	while (son <= end) { //若子節點指標在範圍內才做比較
		if (son + 1 <= end && arr[son] < arr[son + 1]) //先比較兩個子節點大小，選擇最大的
			son++;
		if (arr[dad] > arr[son]) //如果父節點大於子節點代表調整完畢，直接跳出函數
			return;
		else { //否則交換父子內容再繼續子節點和孫節點比較
			swap(&arr[dad], &arr[son]);
			dad = son;
			son = dad * 2 + 1;
		}
	}
}

void heap_sort(int arr[], int len) {
	int i;
	//初始化，i 從最後一個父節點開始調整
	for (i = len / 2 - 1; i >= 0; i--)
		max_heapify(arr, i, len - 1);
	//先將第一個元素和已排好元素前一位做交換，再重新調整，直到排序完畢
	for (i = len - 1; i > 0; i--) {
		swap(&arr[0], &arr[i]);
		max_heapify(arr, 0, i - 1);
	}
}

int main() {
	int arr[] = { 3, 5, 3, 0, 8, 6, 1, 5, 8, 6, 2, 4, 9, 4, 7, 0, 1, 8, 9, 7, 3, 1, 2, 5, 9, 7, 4, 0, 2, 6 };
	int len = (int) sizeof(arr) / sizeof(*arr);
	heap_sort(arr, len);
	int i;
	for (i = 0; i < len; i++)
		printf("%d ", arr[i]);
```

```
        printf("\n");
        return 0;
}
```

13.5. typedef struct 的使用

typedef struct arena_chunk_map_misc_s arena_chunk_map_misc_t;
这里其实只是为 struct arena_chunk_map_misc_s 起了一个别名 arena_chunk_map_misc_t，以后需要定义 arena_chunk_map_misc_s，只需要用 arena_chunk_map_misc_t 就可以了。

13.6. big-endian 和 little-endian 格式

在小端模式中，低位字节放在低地址，高位字节放在高地址；在大端模式中，低位字节放在高地址，高位字节放在低地址。

Android 是小端模式。

```
 (gdb) p /x *(char*)(p-8)@40
$91 = {0xea, 0xde, 0x23, 0x1, 0x1e, 0x0, 0x0, 0x0, 0x31, 0x32, 0x33, 0x34,
  0x35, 0x36, 0x37, 0x38, 0x39, 0x30, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66,
  0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x72,
  0x73, 0x74, 0x0, 0x4}
(gdb) p *(char*)p@30
$83 = "1234567890abcdefghijklmnopqrst"
```

这里前四个字节是 magic 标志为：0x0123deea，后面四个字节是长度 0x1e=30，表示 30 个字符长度。可以看出是低位字节放在低地址。

14. 参考资料

1. "The Pairing Heap: A New Form of Self-Adjusting Heap" https://www.cs.cmu.edu/~sleator/papers/pairing-heaps.pdf
2. https://github.com/jemalloc/jemalloc/wiki
3. http://jemalloc.net/jemalloc.3.html

4. https://en.wikipedia.org/wiki/Linear_congruential_generator
5. https://blog.csdn.net/koozxcv/article/details/50973217
6. https://www.cnblogs.com/gaoxing/p/4253833.html
7. http://www.cnblogs.com/YYPapa/p/6914199.html
8. http://www.cnblogs.com/YYPapa/p/6913768.html
9. https://blog.csdn.net/romandion/article/details/9063841
10. https://blog.csdn.net/romandion/article/details/8926252
11. https://www.cnblogs.com/skywang12345/p/3576969.html