Ptmalloc 内存申请

- Ptmalloc 内存申请
  - 内存申请
    - 内存分配之单线程
  - 内存分配之 _int_malloc
    - _int_malloc 之 small bins
    - _int_malloc 之 malloc_consolidate

> 如有错误以及图片不清晰等问题请提交 issue，谢谢 ~
>
> 源路径：[https://github.com/HATTER-LONG/glibc_Learnging]

内存申请

前几篇文章介绍了 Ptmalloc 初始化、Tcache 缓存，本文将继续 malloc 源码的分析：

```
void *
__libc_malloc (size_t bytes)
{
  mstate ar_ptr;
  void *victim;

  ...........

  if (SINGLE_THREAD_P)
    {
      victim = _int_malloc (&main_arena, bytes);
      assert (!victim || chunk_is_mmapped (mem2chunk (victim)) ||
          &main_arena == arena_for_chunk (mem2chunk (victim)));
      return victim;
    }

  arena_get (ar_ptr, bytes);

  victim = _int_malloc (ar_ptr, bytes);
  /* Retry with another arena only if we were able to find a usable arena
     before.  */
  if (!victim && ar_ptr != NULL)
    {
      LIBC_PROBE (memory_malloc_retry, 1, bytes);
      ar_ptr = arena_get_retry (ar_ptr, bytes);
      victim = _int_malloc (ar_ptr, bytes);
    }

  if (ar_ptr != NULL)
    __libc_lock_unlock (ar_ptr->mutex);

  assert (!victim || chunk_is_mmapped (mem2chunk (victim)) ||
          ar_ptr == arena_for_chunk (mem2chunk (victim)));
  return victim;
}
```

内存分配之单线程

- 参考文章--堆漏洞挖掘中的 bins 分类
- 参考文章--Linux 堆内存管理深入分析

```
  if (SINGLE_THREAD_P) // 判断进程是否为单线程
    {
      victim = _int_malloc (&main_arena, bytes);
      assert (!victim || chunk_is_mmapped (mem2chunk (victim)) ||
          &main_arena == arena_for_chunk (mem2chunk (victim)));
      return victim;
    }
```

SINGLE_THREAD_P 判断当前进程是否为单线程，如果为真则调用 _int_malloc 方法并传入 main_arena 主分配区进行内存内存，这个接口也是 malloc 的核心代码：

> main_arena 主分配区在 Ptmalloc 初始化分析中介绍过，当进程中第一次调用 malloc 申请内存的线程会进行 ptmalloc 初始化，会构建一个主分配区。

内存分配之 _int_malloc

1. _int_malloc 入口部分代码，定义了一些局部变量：

```
static void *
_int_malloc (mstate av, size_t bytes)
{
    INTERNAL_SIZE_T nb;               /* normalized request size */
    unsigned int idx;                 /* associated bin index */
    mbinptr bin;                      /* associated bin */

    mchunkptr victim;                 /* inspected/selected chunk */
    INTERNAL_SIZE_T size;             /* its size */
    int victim_index;                 /* its bin index */

    mchunkptr remainder;              /* remainder from a split */
    unsigned long remainder_size;     /* its size */

    unsigned int block;               /* bit map traverser */
    unsigned int bit;                 /* bit map traverser */
    unsigned int map;                 /* current word of binmap */

    mchunkptr fwd;                    /* misc temp for linking */
    mchunkptr bck;                    /* misc temp for linking */

    #if USE_TCACHE
    size_t tcache_unsorted_count;      /* count of unsorted chunks processed */
    #endif

    .......

}
```

2. 将申请的内存字节转化为对应 chunk 且对其后的大小，使用的也是之前分析过的 `checked_request2size`，接下来判断传入的分配区是否存在，如果为空则使用 sysmalloc 创建一个分配区，并使用其申请内存。：

```
/*
    Convert request size to internal form by adding SIZE_SZ bytes
    overhead plus possibly more to obtain necessary alignment and/or
    to obtain a size of at least MINSIZE, the smallest allocatable
    size. Also, checked_request2size returns false for request sizes
    that are so large that they wrap around zero when padded and
    aligned.
*/

if (!checked_request2size (bytes, &nb))
    {
    __set_errno (ENOMEM);
    return NULL;
    }

/* There are no usable arenas.  Fall back to sysmalloc to get a chunk from
    mmap.  */
if (__glibc_unlikely (av == NULL))
    {
    void *p = sysmalloc (nb, av);
    if (p != NULL)
    alloc_perturb (p, bytes);
    return p;
    }
```

3. 首先判断申请内存 fastbin 是否能满足，并使用 fastbin 进行分配内存：

    - chunk 的大小在 32 字节~128 字节（0x20~0x80）的 chunk 称为 “fast chunk”，注意是指 struct malloc_chunk 结构的大小。
    - 不会对 free chunk 进行合并：鉴于设计 fast bin 的初衷就是进行快速的小内存分配和释放，因此系统将属于 fast bin 的 chunk 的 PREV_INUSE 位总是设置为 1，这样即使当 fast bin 中有某个 chunk 同一个 free chunk 相邻的时候，系统也不会进行自动合并操作，而是保留两者。虽然这样做可能会造成额外的碎片化问题。

```
/*
    If the size qualifies as a fastbin, first check corresponding bin.
    This code is safe to execute even if av is not yet initialized, so we
    can try it without checking, which saves some time on this fast path.
*/
// 多线程安全的从 fastbin 里面移除一个 chunk.
#define REMOVE_FB(fb, victim, pp)            \
do                                \
    {                                \
    victim = pp;                    \
    if (victim == NULL)                \
    break;                        \
    }                                \
while ((pp = catomic_compare_and_exchange_val_acq (fb, victim->fd, victim)) \
    != victim);                    \

if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
    {
    idx = fastbin_index (nb); // 判断某一个 fastchunk 属于哪一个 fastbin 链表。
    mfastbinptr *fb = &fastbin (av, idx); // 获得 bin 链表上的 chunk
    mchunkptr pp;
    victim = *fb;

    if (victim != NULL)
    {
    if (SINGLE_THREAD_P)
        *fb = victim->fd; // 如果为单线程则直接删除对应 chunk 节点即可
    else
        REMOVE_FB (fb, pp, victim); // 多线程需要使用 原子操作 进行删除
    if (__glibc_likely (victim != NULL))
        {
        size_t victim_idx = fastbin_index (chunksize (victim)); // 检查分配的 chunk 内存
        if (__builtin_expect (victim_idx != idx, 0))
        malloc_printerr ("malloc(): memory corruption (fast)");
        check_remalloced_chunk (av, victim, nb);
#if USE_TCACHE
        /* While we're here, if we see other chunks of the same size,
        stash them in the tcache.
        发现其他相同大小的块，将它们存放在 tcache 中，直到 tcache 对应链长度表达到 tcache_count = 7
        */
        size_t tc_idx = csize2tidx (nb);
        if (tcache && tc_idx < mp_.tcache_bins)
        {
        mchunkptr tc_victim;

        /* While bin not empty and tcache not full, copy chunks.  */
        while (tcache->counts[tc_idx] < mp_.tcache_count
            && (tc_victim = *fb) != NULL)
            {
            if (SINGLE_THREAD_P)
            *fb = tc_victim->fd;
            else
            {
            REMOVE_FB (fb, pp, tc_victim);
            if (__glibc_unlikely (tc_victim == NULL))
                break;
            }
            tcache_put (tc_victim, tc_idx);
            }
        }
#endif
        void *p = chunk2mem (victim); // 定位到 chunk 中用户内存地址
        alloc_perturb (p, bytes); //对用户使用的内存进行初始化
        return p;
        }
    }
```

    - REMOVE_FB 中不好理解的宏便是 catomic_compare_and_exchange_val_acq，其中传入的参数 mem 与 oldval 一般都是相等的，因此其作用便是将 mem 指向 newval 并返回 oldval，达成在 mem 链表中取出 oldval 的目的：

```
 /* Atomically store NEWVAL in *MEM if *MEM is equal to OLDVAL.
    Return the old *MEM value.
    如果 *MEM 等于 OLDVAL，则将 NEWVAL 原子地存储在 *MEM 中。
```

```
   返回旧的 *MEM 值。
*/
#ifndef catomic_compare_and_exchange_val_acq
# ifdef __arch_c_compare_and_exchange_val_32_acq
#  define catomic_compare_and_exchange_val_acq(mem, newval, oldval) \
__atomic_val_bysize (__arch_c_compare_and_exchange_val,acq,         \
            mem, newval, oldval)
# else
#  define catomic_compare_and_exchange_val_acq(mem, newval, oldval) \
atomic_compare_and_exchange_val_acq (mem, newval, oldval)
# endif
#endif
```

- fastbin_index 与 fastbin 可以看出其组合使用取出目标大小的 chunk 链表节点，fastbin_index 判断 64 位系统则除以 8 再减去 2，64 位系统举例： sz 为 32 时，32 >> 4 = 2 - 2 = 0; 得出 fastbin 数字的对应 chunk 链表下标，因此可以反推 fastbin 是 32 字节开始，最大的请求为 160 字节（注意这并不是指 fastbin 中 chunk 的大小，其是通过 set_max_fast 设置，后文会分析）：

```
typedef struct malloc_chunk *mfastbinptr;
#define fastbin(ar_ptr, idx) ((ar_ptr)->fastbinsY[idx])

/* offset 2 to use otherwise unindexable first 2 bins */
#define fastbin_index(sz) \
((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)

/* The maximum fastbin request size we support */
#define MAX_FAST_SIZE     (80 * SIZE_SZ / 4)

#define NFASTBINS  (fastbin_index (request2size (MAX_FAST_SIZE)) + 1)
struct malloc_state
{
    ......
    /* Fastbins */
    mfastbinptr fastbinsY[NFASTBINS];
    .......
};
```

- 当获取到正确大小的 chunk 后，会有进行一个检查操作：

```
size_t victim_idx = fastbin_index (chunksize (victim)); // 检查分配的 chunk 内存
if (__builtin_expect (victim_idx != idx, 0))            // 应该与申请时的下表保持一致，否则可能 chunk 内存被破坏
malloc_printerr ("malloc(): memory corruption (fast)");
check_remalloced_chunk (av, victim, nb);  // 这一部分用于 debug 模式

.....

/* size field is or'ed with PREV_INUSE when previous adjacent chunk in use */
#define PREV_INUSE 0x1
/* size field is or'ed with IS_MMAPPED if the chunk was obtained with mmap() */
#define IS_MMAPPED 0x2
/* size field is or'ed with NON_MAIN_ARENA if the chunk was obtained
from a non-main arena.  This is only set immediately before handing
the chunk to the user, if necessary.  */
#define NON_MAIN_ARENA 0x4

/*
Bits to mask off when extracting size

Note: IS_MMAPPED is intentionally not masked off from size field in
macros for which mmapped chunks should never be seen. This should
cause helpful core dumps to occur if it is tried by accident by
people extending or adapting this malloc.
*/
#define SIZE_BITS (PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)

/* Get size, ignoring use bits */
#define chunksize(p) (chunksize_nomask (p) & ~(SIZE_BITS))
/* Like chunksize, but do not mask SIZE_BITS.  */
#define chunksize_nomask(p)         ((p)->mchunk_size)
```

- 终于 fastbin 内存分配就分析完了，但是有一点需要注意，我们默认 get_max_fast 是有效的，但是从源码可以看出 global_max_fast 是需要设置的，其就是在 malloc_init_state 在 ptmalloc 初始化过程中进行设置。

```
/*
Set value of max_fast.
Use impossibly small value if 0.
Precondition: there are no existing fastbin chunks in the main arena.
Since do_check_malloc_state () checks this, we call malloc_consolidate ()
before changing max_fast.  Note other arenas will leak their fast bin
entries if max_fast is reduced.
    */

#define set_max_fast(s) \
global_max_fast = (((s) == 0)    \
                   ? MIN_CHUNK_SIZE / 2 : ((s + SIZE_SZ) & ~MALLOC_ALIGN_MASK))

static inline INTERNAL_SIZE_T
get_max_fast (void)
{
/* Tell the GCC optimizers that global_max_fast is never larger
    than MAX_FAST_SIZE.  This avoids out-of-bounds array accesses in
    _int_malloc after constant propagation of the size parameter.
    (The code never executes because malloc preserves the
    global_max_fast invariant, but the optimizers may not recognize
    this.)  */
if (global_max_fast > MAX_FAST_SIZE)
    __builtin_unreachable ();
return global_max_fast;
}
```

_int_malloc 之 small bins

当 fast bins 未初始化或者不能满足申请的内存时，流程就会尝试使用 small bins 来完成内存申请。

```
  /*
     If a small request, check regular bin.  Since these "smallbins"
     hold one size each, no searching within bins is necessary.
```

```
      (For a large request, we need to wait until unsorted chunks are
      processed to find best fit. But for small ones, fits are exact
      anyway, so we can check now, which is faster.)
    */

  if (in_smallbin_range (nb))
    {
      idx = smallbin_index (nb);
      bin = bin_at (av, idx);

      // 判断当前 chunk 不是此链表最后一个 chunk，并进行正确性链表检查
      if ((victim = last (bin)) != bin)
        {
          bck = victim->bk;
      if (__glibc_unlikely (bck->fd != victim))
        malloc_printerr ("malloc(): smallbin double linked list corrupted");

          set_inuse_bit_at_offset (victim, nb);
          bin->bk = bck;
          bck->fd = bin;

          if (av != &main_arena)
        set_non_main_arena (victim);
          check_malloced_chunk (av, victim, nb);
#if USE_TCACHE
      /* While we're here, if we see other chunks of the same size,
         stash them in the tcache.  */
      //类似 fastbins 一样，会将相同类型的 chunk 填充到 tcache 中
      size_t tc_idx = csize2tidx (nb);
      if (tcache && tc_idx < mp_.tcache_bins)
        {
          mchunkptr tc_victim;

          /* While bin not empty and tcache not full, copy chunks over.  */
          while (tcache->counts[tc_idx] < mp_.tcache_count
             && (tc_victim = last (bin)) != bin)
        {
          if (tc_victim != 0)
            {
              bck = tc_victim->bk;
              set_inuse_bit_at_offset (tc_victim, nb);
              if (av != &main_arena)
            set_non_main_arena (tc_victim);
              bin->bk = bck;
              bck->fd = bin;

              tcache_put (tc_victim, tc_idx);
                }
        }
#endif
          void *p = chunk2mem (victim);
          alloc_perturb (p, bytes);
          return p;
        }
    }
    /*
      If this is a large request, consolidate fastbins before continuing.
      While it might look excessive to kill all fastbins before
      even seeing if there is space available, this avoids
      fragmentation problems normally associated with fastbins.
      Also, in practice, programs tend to have runs of either small or
      large requests, but less often mixtures, so consolidation is not
      invoked all that often in most programs. And the programs that
      it is called frequently in otherwise tend to fragment.
    */

  else
    {
      idx = largebin_index (nb);
      if (atomic_load_relaxed (&av->have_fastchunks))
        malloc_consolidate (av);
    }
```

参考文章--glibc-malloc 源码分析

1. 首先判断申请的内存大小是否符合 `small bins` 范围，然后使用 `smallbin_index` 寻找对应链表。 small bins 中每个 chunk 的大小与其所在的 bin 的 index 的关系为 `chunk_size = 2 * SIZE_SZ *index`：

```
/*
Indexing

    Bins for sizes < 512 bytes contain chunks of all the same size, spaced
    8 bytes apart. Larger bins are approximately logarithmically spaced:

    64 bins of size       8
    32 bins of size      64
    16 bins of size     512
    8 bins of size    4096
    4 bins of size   32768
    2 bins of size  262144
    1 bin  of size what's left

    There is actually a little bit of slop in the numbers in bin_index
    for the sake of speed. This makes no difference elsewhere.

    The bins top out around 1MB because we expect to service large
    requests via mmap.

    Bin 0 does not exist.  Bin 1 is the unordered list; if that would be
    a valid chunk size the small bins are bumped up one.
*/
//bins 总的数量
#define NBINS             128
//Small bin 大小
#define NSMALLBINS         64
// 2*size_t
#define SMALLBIN_WIDTH    MALLOC_ALIGNMENT
//是否需要对 small bin 的下标进行纠正
#define SMALLBIN_CORRECTION (MALLOC_ALIGNMENT > 2 * SIZE_SZ)
#define MIN_LARGE_SIZE    ((NSMALLBINS - SMALLBIN_CORRECTION) * SMALLBIN_WIDTH)

#define in_smallbin_range(sz)  \
```

```
((unsigned long) (sz) < (unsigned long) MIN_LARGE_SIZE)
//根据 sz 得到 smallbin 的序号，例如 sz = 32， 32>>4 = 2 + SMALLBIN_CORRECTION (0) = 2
// 对应 bins 数组，第 1 位是无序链表，小于 512 字节 的是 smalle bins，8、64、512、。因此返回 下标 2（64）
#define smallbin_index(sz) \
((SMALLBIN_WIDTH == 16 ? (((unsigned) (sz)) >> 4) : (((unsigned) (sz)) >> 3))\
+ SMALLBIN_CORRECTION)
```

2. 使用 `bin_at` 得到 `small bin` 的链表头地址：

```
/* addressing -- note that bin_at(0) does not exist */
#define bin_at(m, i) \
(mbinptr) (((char *) &((m)->bins[((i) - 1) * 2]))     \
           - offsetof (struct malloc_chunk, fd))

/* analog of ++bin */
#define next_bin(b)  ((mbinptr) ((char *) (b) + (sizeof (mchunkptr) << 1)))

/* Reminders about list directionality within bins */
#define first(b)     ((b)->fd)
#define last(b)      ((b)->bk)

// 由于 分配区的 bins 管理是双向链表 fd 与 bk 进行管理
// 当拿到 fd 地址时需要减去对应偏移就是当前 chunk 的头部地址
struct malloc_chunk {

INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

struct malloc_chunk* fd;         /* double links -- used only if free. */
struct malloc_chunk* bk;

/* Only used for large blocks: pointer to next larger size.  */
struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
struct malloc_chunk* bk_nextsize;
};
```

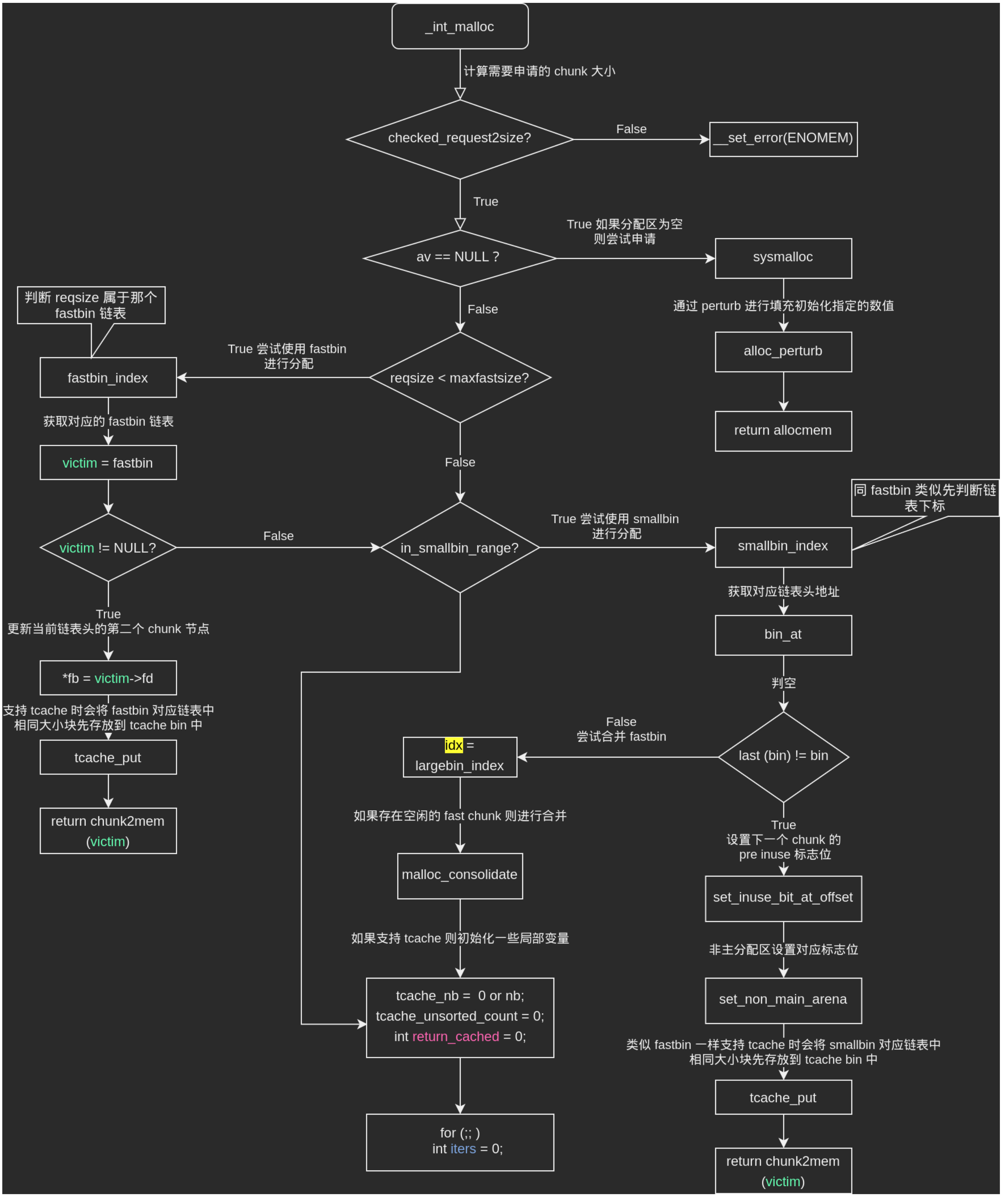
3. 设置取出的 chunk 使用标志位，然后将其从链表中取出，非主分配区设置此 chunk 对应标志位：

```
set_inuse_bit_at_offset (victim, nb);
bin->bk = bck;
bck->fd = bin;
if (av != &main_arena)
    set_non_main_arena (victim);
check_malloced_chunk (av, victim, nb);// debug 模式使用
.................

#define set_inuse_bit_at_offset(p, s)     \
    (((mchunkptr) (((char *) (p)) + (s)))->mchunk_size |= PREV_INUSE)

/* Check for chunk from main arena.  */
#define chunk_main_arena(p) (((p)->mchunk_size & NON_MAIN_ARENA) == 0)

/* Mark a chunk as not being on the main arena.  */
#define set_non_main_arena(p) ((p)->mchunk_size |= NON_MAIN_ARENA)
```

4. else 分支则表示当 small bins 无法满足必须使用 large bins 之前，先进行 fast bins 的合并工作：

   - 参考文章--浅析 largebin attack
   - 参考文章--largebin 学习从源码到做题
   - 大于 1024（512）字节的 chunk 称之为 large chunk，large bin 就是用于管理这些 large chunk 的。一共包括 63 个 bin，index 为 64~126，每个 bin 中的 chunk 的大小不一致，而是处于一定区间范围内。
   - 例如分配 1024 字节内存，largebin_index_64：1024 >> 16 = 16 < 48 = 48 + 16 = 64。

```
  idx = largebin_index (nb); //获取申请内存所需的 largebin 的 下标
  if (atomic_load_relaxed (&av->have_fastchunks)) // 原子获取分配区中是否有空闲的 fast chunks 接下来进行合并操作
    malloc_consolidate (av);

........

//  It remains to be seen whether it is good to keep the widths of
//  the buckets the same or whether it should be scaled by a factor
//  of two as well.
#define largebin_index_64(sz)                                                \
(((((unsigned long) (sz)) >> 6) <= 48) ?  48 + (((unsigned long) (sz)) >> 6) :\
((((unsigned long) (sz)) >> 9) <= 20) ?  91 + (((unsigned long) (sz)) >> 9) :\
((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12) :\
((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15) :\
((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18) :\
126)

#define largebin_index(sz) \
(SIZE_SZ == 8 ? largebin_index_64 (sz)                                     \
: MALLOC_ALIGNMENT == 16 ? largebin_index_32_big (sz)                     \
: largebin_index_32 (sz))

# define atomic_load_relaxed(mem) \
({ __atomic_check_size_ls((mem));    \
    __atomic_load_n ((mem), __ATOMIC_RELAXED); })
```

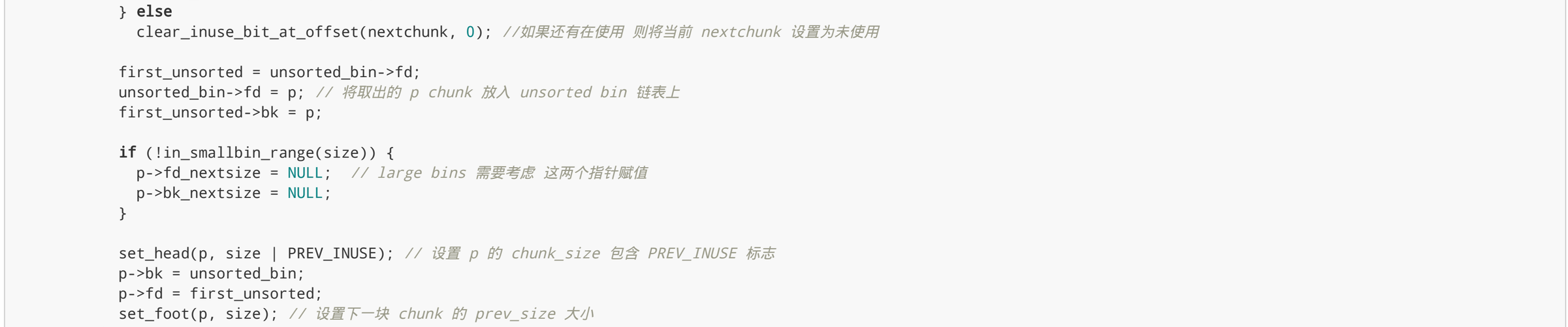


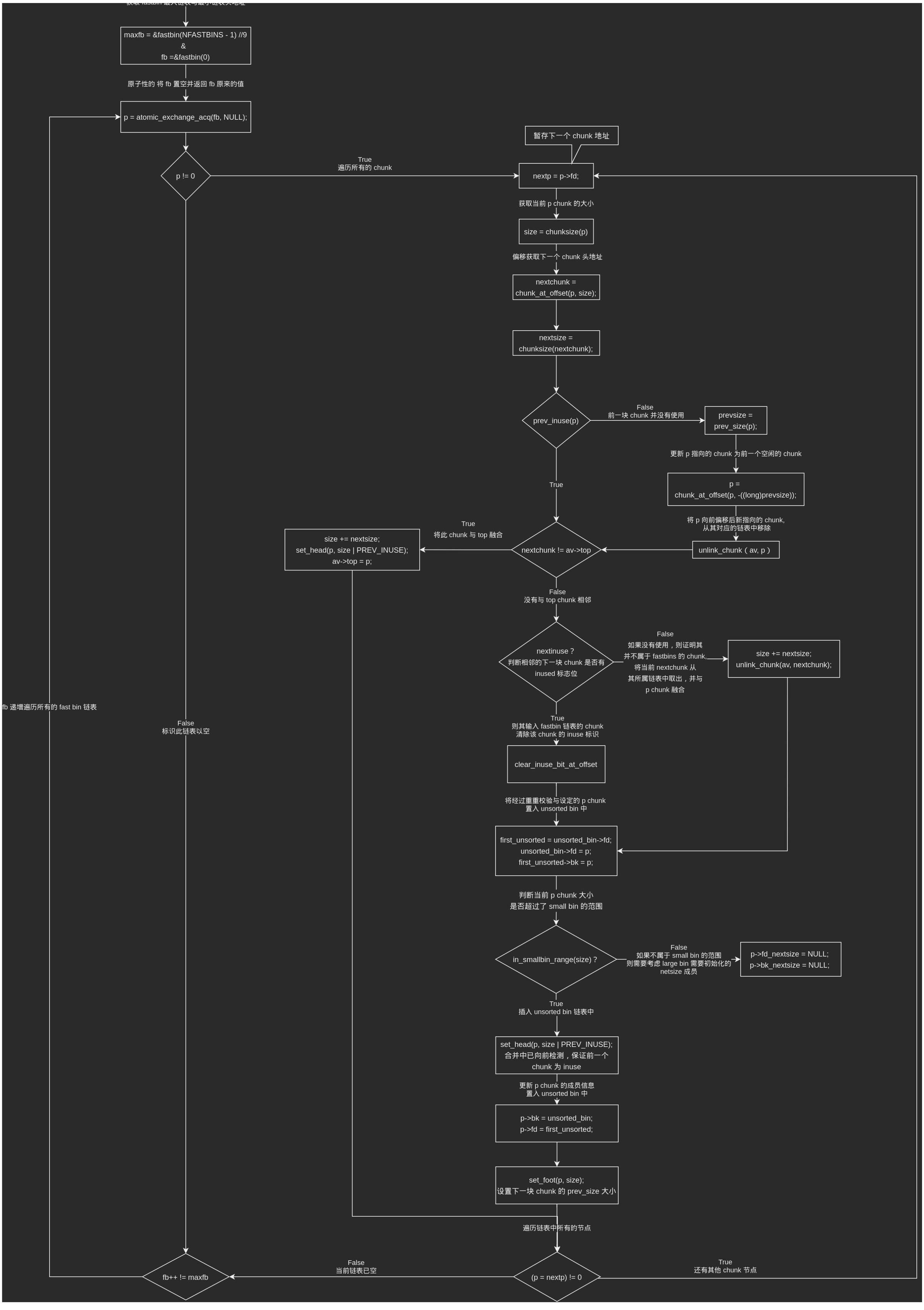
_int_malloc 之 malloc_consolidate

- 参考文章--堆漏洞挖掘中的 malloc_consolidate 与 FASTBIN_CONSOLIDATION_THRESHOLD
- consolidate 的目的对堆中的碎片 chunk 进行合并整理，减少堆中的碎片。主要将 fastbin 管理的所有 chunk 空闲的进行释放，将其添加到 usorted bin 上。

```
/*
  ------------------------- malloc_consolidate -------------------------

  malloc_consolidate is a specialized version of free() that tears
  down chunks held in fastbins.  Free itself cannot be used for this
  purpose since, among other things, it might place chunks back onto
  fastbins.  So, instead, we need to use a minor variant of the same
  code.
*/

static void malloc_consolidate(mstate av) {
  mfastbinptr *fb;          /* current fastbin being consolidated */
```

```c
    mfastbinptr *maxfb;       /* last fastbin (for loop control) */
    mchunkptr p;              /* current chunk being consolidated */
    mchunkptr nextp;          /* next chunk to consolidate */
    mchunkptr unsorted_bin;   /* bin header */
    mchunkptr first_unsorted; /* chunk to link to */

    /* These have same use as in free() */
    mchunkptr nextchunk;
    INTERNAL_SIZE_T size;
    INTERNAL_SIZE_T nextsize;
    INTERNAL_SIZE_T prevsize;
    int nextinuse;

    atomic_store_relaxed(&av->have_fastchunks,
                         false); //首先原子操作将分配区 have_fastchunks 置为 false

    unsorted_bin =
        unsorted_chunks(av); //取出 bin[1] 也就是 unsorted bins 的第一个 chunk

    /*
      Remove each chunk from fast bin and consolidate it, placing it
      then in unsorted bin. Among other reasons for doing this,
      placing in unsorted bin avoids needing to calculate actual bins
      until malloc is sure that chunks aren't immediately going to be
      reused anyway.
    */
    /*
        从 fast bin 中删除每个块并合并它，然后将其放入 unsorted bin 中。
        这样做的其他原因之一是，放入 unsorted bin 避免了需要计算实际 bin，
        直到 malloc 确定无论如何都不会立即重用块。
    */
    maxfb = &fastbin(av, NFASTBINS - 1); // 取出 fastbin 中最后一个链表 chunk
    fb = &fastbin(av, 0);                // fastbin 第一个链表的 chunk
    do {
      p = atomic_exchange_acq(fb, NULL); //原子性的 将 fb 置空并返回 fb 原来的值
      if (p != 0) {
        do {
          {
            unsigned int idx =
                fastbin_index(chunksize(p)); // 通过 chunksize 重新获取 index
            if ((&fastbin(av, idx)) != fb) // 并重新获取一次 chunk 判断链表布局是否正常
              malloc_printerr("malloc_consolidate(): invalid chunk size");
          }

          check_inuse_chunk(av, p); // debug 模式使用
          nextp = p->fd;

          /* Slightly streamlined version of consolidation code in free() */
          size = chunksize(p);                  // 获取当前 p chunk 的大小
          nextchunk = chunk_at_offset(p, size); // 偏移获取下一个 chunk 头地址
          nextsize = chunksize(nextchunk);

          if (!prev_inuse(p)) {      // 如果前一块 chunk 没有使用，
            prevsize = prev_size(p); // 则获取前一块 chunk 偏移
            size += prevsize;
            p = chunk_at_offset(
                p, -((long)prevsize)); // 并将 p 向前推移到前一块 chunk 的首地址
            if (__glibc_unlikely(chunksize(p) != prevsize))
              malloc_printerr("corrupted size vs. prev_size in fastbins");
            unlink_chunk(av, p); // 将这块空闲的 chunk 从对应的 bins 链表取出
          }

          if (nextchunk != av->top) { // 判断当前 chunk 是否与 top chunk 相邻
            nextinuse = inuse_bit_at_offset(nextchunk, nextsize); // 判断 nextchunk 是否 inuse

            if (!nextinuse) {
              size += nextsize;
              unlink_chunk(av, nextchunk); // 如果没有使用，则证明其并不属于 fastbins 的 chunk, 将当前 nextchunk 从其所属链表中取出
            } else
              clear_inuse_bit_at_offset(nextchunk, 0); //如果还有在使用 则将当前 nextchunk 设置为未使用

            first_unsorted = unsorted_bin->fd;
            unsorted_bin->fd = p; // 将取出的 p chunk 放入 unsorted bin 链表上
            first_unsorted->bk = p;

            if (!in_smallbin_range(size)) {
              p->fd_nextsize = NULL;  // large bins 需要考虑 这两个指针赋值
              p->bk_nextsize = NULL;
            }

            set_head(p, size | PREV_INUSE); // 设置 p 的 chunk_size 包含 PREV_INUSE 标志
            p->bk = unsorted_bin;
            p->fd = first_unsorted;
            set_foot(p, size); // 设置下一块 chunk 的 prev_size 大小
          }

          else { // 将此 chunk 与 top 融合
            size += nextsize;
            set_head(p, size | PREV_INUSE);
            av->top = p;
          }

        } while ((p = nextp) != 0); // 遍历链表所有节点
      }
    } while (fb++ != maxfb); // 便利所有链表
  }
```




maxfb = &fastbin(NFASTBINS - 1) //9
&
fb =&fastbin(0)
原子性的 将 fb 置空并返回 fb 原来的值
p = atomic_exchange_acq(fb, NULL);
p != 0
True
遍历所有的 chunk
暂存下一个 chunk 地址
nextp = p->fd;
获取当前 p chunk 的大小
size = chunksize(p)
偏移获取下一个 chunk 头地址
nextchunk =
chunk_at_offset(p, size);
nextsize =
chunksize(nextchunk);
prev_inuse(p)
False
前一块 chunk 并没有使用
prevsize =
prev_size(p);
更新 p 指向的 chunk 为前一个空闲的 chunk
p =
chunk_at_offset(p, -((long)prevsize));
将 p 向前偏移后新指向的 chunk,
从其对应的链表中移除
unlink_chunk (av, p)
True
将此 chunk 与 top 融合
size += nextsize;
set_head(p, size | PREV_INUSE);
av->top = p;
nextchunk != av->top
False
没有与 top chunk 相邻
nextinuse ?
判断相邻的下一块 chunk 是否有
inused 标志位
False
如果没有使用，则证明其
并不属于 fastbins 的 chunk,
将当前 nextchunk 从
其所属链表中取出，并与
p chunk 融合
size += nextsize;
unlink_chunk(av, nextchunk);
fb 递增遍历所有的 fast bin 链表
False
标识此链表以空
True
则其输入 fastbin 链表的 chunk
清除该 chunk 的 inuse 标识
clear_inuse_bit_at_offset
将经过重重校验与设定的 p chunk
置入 unsorted bin 中
first_unsorted = unsorted_bin->fd;
unsorted_bin->fd = p;
first_unsorted->bk = p;
判断当前 p chunk 大小
是否超过了 small bin 的范围
in_smallbin_range(size) ?
False
如果不属于 small bin 的范围
则需要考虑 large bin 需要初始化的
netsize 成员
p->fd_nextsize = NULL;
p->bk_nextsize = NULL;
True
插入 unsorted bin 链表中
set_head(p, size | PREV_INUSE);
合并中已向前检测，保证前一个
chunk 为 inuse
更新 p chunk 的成员信息
置入 unsorted bin 中
p->bk = unsorted_bin;
p->fd = first_unsorted;
set_foot(p, size);
设置下一块 chunk 的 prev_size 大小
遍历链表中所有的节点
(p = nextp) != 0
True
还有其他 chunk 节点
False
当前链表已空
fb++ != maxfb