3. 分配及实现

本章节介绍dlmalloc的分配算法和实现.由于存在多mspace的情况, dlmalloc使用了两套API.一套对应默认的mspace,以dl前缀开头,如dlmalloc, dlrealloc等.如果创建了自定义的mspace,则使用mspace开头的API,如mspace_malloc, mspace_realloc等.但两套API在基础算法上是一致的.我们就以默认的API为主要对象介绍.

3.1 算法概览

事实上, dlmalloc虽然复杂,核心算法却非常简单,如果有前面章节的基础很容易就能看懂.

核心分配算法针对small request和large request概括起来各五句话(注意这里的分配请求大小都是经过align和padding处理后的大小),对应small request(<256字节),

1. 首先在分配请求对应大小的分箱以及更大一级分箱中查找, 如果有则返回,否则进入下一步.选择这两个分箱因为它们最接近分配目标大小,且剩余部分都无法单独成为一个chunk (原文中称之为remainderless chunk).

2. 如果dv大小足够满足,则切割dv chunk,否则进入下一步.

3. 在所有分箱范围内查找(包括small bins和tree bins),找到可以满足需求的最小的chunk,切割,将剩余部分指定为新的dv,否则进入下一步.

4. 如果top chunk满足需求,则切割top,否则进入下一步.

5. 从系统获取内存并使用它.

对应large request,

1. 从tree bins中查找最小可用的tchunk,如果其比dv更加适合(更接近目标大小),就使用该chunk.如果其剩余部分超过最小可分配chunk,则切割它.否则进入下一步.

2. 如果dv满足需求,且比任何分箱中的chunk更适合,则使用dv,否则进入下一步.

3. 如果top满足需求,则使用top,否则进入下一步.

4. 如果分配请求大于mmap_threshold阈值,则直接通过mmap分配,否则进入下一步.

5. 从系统获取内存并使用它.

从类型上, dlmalloc属于best-fit型分配器,只是Doug Lea在此基础上做了诸多优化.本质上都是本着物尽其用的思想来挑选合适的free chunk, 只有当不能首先满足时, dlmalloc会通过dv和top来做进一步的挑选,这就最大限度的减小了内部碎片产生.同时dv和top的存在也能比较有效的减少外部碎片.

而如果外部请求过大, dlmalloc不是优先获取系统内存后分配,反而倾向于直接通过mmap获取.原因在于位于top的free chunk有可能因为相邻高地址的alloced chunk而一直无法释放.如果dlmalloc向系统申请了大块内存,即便被应用程序free,

也可能因为auto trmming失败而导致它们长期驻留在top space中.而直接mmap的好处就是随时可以将这些huge chunk返回给系统,只要应用程序决定不再使用它们.

下面是更详细的代码分析,

```
04589: /* 如果定义USE_LOCKS, 则进行全局变量初始化 */
04590: #if USE_LOCKS
04591:   ensure_initialization();
04592: #endif
04593:
04594: /* 对当前内存池加锁 */
04595:   if (!PREACTION(gm)) {
04596:     void* mem;
04597:     size_t nb;
04598:     /* 对应small request */
04599:     if (bytes <= MAX_SMALL_REQUEST) {
04600:       bindex_t idx;
04601:       binmap_t smallbits;
04602:       /* 如果分配请求小于最小可分配内存, 就补齐为最小可分配chunk大小, 否则补齐为正常大小 */
04603:       nb = (bytes < MIN_REQUEST)? MIN_CHUNK_SIZE : pad_request(bytes);
04604:       /* 计算nb对应的索引, 转换成可用的small bins map */
04605:       idx = small_index(nb);
04606:       smallbits = gm->smallmap >> idx;
04607:
04608:       /* 首先在remainderless分箱中查找(对应small request算法1) */
04609:       if ((smallbits & 0x3U) != 0) {
04610:         mchunkptr b, p;
04611:         /* 如果nb所在分箱为空, 则使用更高一级分箱 */
04612:         idx += ~smallbits & 1;
04613:         /* 根据索引找到分箱地址 */
04614:         b = smallbin_at(gm, idx);
04615:         /* 第一个可用free chunk */
04616:         p = b->fd;
04617:         assert(chunksize(p) == small_index2size(idx));
04618:         /* 将free chunk从double list中摘除 */
04619:         unlink_first_small_chunk(gm, b, p, idx);
04620:         /* 设置该chunk的P位和C位 */
04621:         set_inuse_and_pinuse(gm, p, small_index2size(idx));
04622:         /* 返回payload指针, 结束 */
```

```
04623:         mem = chunk2mem(p);
04624:         check_malloced_chunk(gm, mem, nb);
04625:         goto ↓postaction;
04626:       }
04627:       /*
04628:        * 如果remainderless没有可用free chunk, 且dv也不满足,
04629:        * 使用剩余分箱查找(对应small request算法3)
04630:        */
04631:       else if (nb > gm->dvsize) {
04632:         /* 首先检查剩余的small bins */
04633:         if (smallbits != 0) {
04634:           mchunkptr b, p, r;
04635:           size_t rsize;
04636:           bindex_t i;
04637:           /* 计算剩余small bins map */
04638:           binmap_t leftbits = (smallbits << idx) & left_bits(idx2bit(idx));
04639:           /* 获取最接近nb的分箱掩码, 并计算分箱索引 */
04640:           binmap_t leastbit = least_bit(leftbits);
04641:           compute_bit2idx(leastbit, i);
04642:           /* 获取目标分箱, 以及第一个可用的free chunk */
04643:           b = smallbin_at(gm, i);
04644:           p = b->fd;
04645:           assert(chunksize(p) == small_index2size(i));
04646:           unlink_first_small_chunk(gm, b, p, i);
04647:           /* 多余的remainder size */
04648:           rsize = small_index2size(i) - nb;
04649:           /*
04650:            * 在size_t不等于4的系统上, remainder是不可用的, 直接标记C和P位,
04651:            * 多数系统下这里会切割free chunk.
04652:            */
04653:           if (SIZE_T_SIZE != 4 && rsize < MIN_CHUNK_SIZE)
04654:             set_inuse_and_pinuse(gm, p, small_index2size(i));
04655:           else {
04656:             /* 标记C和P位 */
04657:             set_size_and_pinuse_of_inuse_chunk(gm, p, nb);
04658:             /* 切割chunk p */
04659:             r = chunk_plus_offset(p, nb);
04660:             /* 设置remainder的大小和P位 */
04661:             set_size_and_pinuse_of_free_chunk(r, rsize);
04662:             /* 用remainder chunk替代称为新的dv */
04663:             replace_dv(gm, r, rsize);
04664:           }
04665:           /* 返回payload指针, 结束 */
04666:           mem = chunk2mem(p);
04667:           check_malloced_chunk(gm, mem, nb);
04668:           goto ↓postaction;
04669:         } ? end if smallbits!=0 ?
04670:         /* 剩余small bins中没有, 在tree bins中检查 */
04671:         else if (gm->treemap != 0 && (mem = tmalloc_small(gm, nb)) != 0) {
04672:           check_malloced_chunk(gm, mem, nb);
04673:           goto ↓postaction;
04674:         }
04675:       } ? end if nb>gm->dvsize ?
04676:     } ? end if bytes<=MAX_SMALL_REQUEST ?
04677:     /* 如果分配请求大于最高可分配大小, 则直接返回失败(在sys_alloc中) */
04678:     else if (bytes >= MAX_REQUEST)
04679:       nb = MAX_SIZE_T;
04680:     /* 对应large request */
04681:     else {
04682:       nb = pad_request(bytes);
04683:       /* 如果tree map中存在, 则在tree bins中寻找(对应large request算法1) */
04684:       if (gm->treemap != 0 && (mem = tmalloc_large(gm, nb)) != 0) {
04685:         check_malloced_chunk(gm, mem, nb);
04686:         goto ↓postaction;
04687:       }
04688:     }
04689:     /* 如果dv可用, 使用dv分配(对应small request和large request算法2) */
04690:     if (nb <= gm->dvsize) {
04691:       size_t rsize = gm->dvsize - nb;
```

```
04692:       mchunkptr p = gm->dv;
04693:       /* 如果dv有剩余, 则切割dv */
04694:       if (rsize >= MIN_CHUNK_SIZE) {
04695:         mchunkptr r = gm->dv = chunk_plus_offset(p, nb);
04696:         gm->dvsize = rsize;
04697:         set_size_and_pinuse_of_free_chunk(r, rsize);
04698:         set_size_and_pinuse_of_inuse_chunk(gm, p, nb);
04699:       }
04700:       /* 否则, 将耗尽该dv */
04701:       else {
04702:         size_t dvs = gm->dvsize;
04703:         gm->dvsize = 0;
04704:         gm->dv = 0;
04705:         set_inuse_and_pinuse(gm, p, dvs);
04706:       }
04707:       /* 返回payload, 结束 */
04708:       mem = chunk2mem(p);
04709:       check_malloced_chunk(gm, mem, nb);
04710:       goto ↓postaction;
04711:     } ? end if nb<=gm->dvsize ?
04712:     /* 如果top可用, 则使用top(对应small request算法4和large request算法3) */
04713:     else if (nb < gm->topsize) { /* Split top */
04714:       size_t rsize = gm->topsize -= nb;
04715:       /* 切割top */
04716:       mchunkptr p = gm->top;
04717:       mchunkptr r = gm->top = chunk_plus_offset(p, nb);
04718:       r->head = rsize | PINUSE_BIT;
04719:       set_size_and_pinuse_of_inuse_chunk(gm, p, nb);
04720:       /* 返回payload, 结束 */
04721:       mem = chunk2mem(p);
04722:       check_top_chunk(gm, gm->top);
04723:       check_malloced_chunk(gm, mem, nb);
04724:       goto ↓postaction;

04725:     }
04726:     /* 都不满足, 从获取系统内存(对应small request算法5和large request算法4,5) */
04727:     mem = sys_alloc(gm, nb);
04728:   /* 解开当前内存池锁 */
04729:   postaction:
04730:     POSTACTION(gm);
04731:     return mem;
04732:   } ? end if !PREACTION(gm) ?
04733:
04734:   return 0;
04735: } ? end dlmalloc ?
```

基本上还是比较好理解的, 下面对一些地方做展开说明,

1. Line5491, 这里如果使用lock, 会在开始确认一些全局参数是否初始化.这些参数保存在名为mparams的全局变量里,类型为malloc_params,包含交叉检查的magic,当前系统页面大小,设定的粒度大小, mmap的阈值, trimming阈值以及默认的mspace参数.并且以magic作为参数初始化的标志.

```
#define ensure_initialization() (void)(mparams.magic != 0 || init_mparams())

static struct malloc_params mparams;

struct malloc_params {
  size_t magic;
  size_t page_size;
  size_t granularity;
  size_t mmap_threshold;
  size_t trim_threshold;
  flag_t default_mflags;
};
```

2. Line4595, PREACTION和POSTACTION成对出现,就是加锁和解锁.因为是平台相关的,针对不同系统需要有具体的实现.从这里其实也可以看出dlmalloc对多线程条件下的分配设计的还是比较简陋的,关注的还是单线程下分配算法的实现.

```
#if USE_LOCKS
#define PREACTION(M)  ((use_lock(M))? ACQUIRE_LOCK(&(M)->mutex) : 0)
#define POSTACTION(M) { if (use_lock(M)) RELEASE_LOCK(&(M)->mutex); }
#else /* USE_LOCKS */
```

3. Line4619, 这里是一个对double link list首节点的删除操作,且如果list为空就更新small map.注意, dlmalloc为了提高list处理速度,是设计了头节点的,因此这个first chunk并不是头节点,而是其前一个节点.这个曾经在前面的章节提到过,可以通过这里的具体实现看到这些优势,其中B指代list头节点, P是需要删除的节点.

```
#define unlink_first_small_chunk(M, B, P, I) {\
  mchunkptr F = P->fd;\
  assert(P != B);\
  assert(P != F);\
  assert(chunksize(P) == small_index2size(I));\
  if (B == F) {\
    clear_smallmap(M, I);\
  }\
  else if (RTCHECK(ok_address(M, F) && F->bk == P)) {\
    F->bk = B;\
    B->fd = F;\
  }\
  else {\
    CORRUPTION_ERROR_ACTION(M);\
  }\
}
```

4. Line4663, 是替换dv的过程,旧dv如果还存在,会送回到分箱系统中管理,而新的chunk作为其替代. M指mstate, P是继任dv, S为继任dv大小.

```
#define replace_dv(M, P, S) {\
  size_t DVS = M->dvsize;\
  assert(is_small(DVS));\
  if (DVS != 0) {\
    mchunkptr DV = M->dv;\
    insert_small_chunk(M, DV, DVS);\
  }\
  M->dvsize = S;\
  M->dv = P;\
}
```

这里insert_small_chunk是前面删除的反向操作,实现如下,

```
#define insert_small_chunk(M, P, S) {\
  bindex_t I  = small_index(S);\
  mchunkptr B = smallbin_at(M, I);\
  mchunkptr F = B;\
  assert(S >= MIN_CHUNK_SIZE);\
  if (!smallmap_is_marked(M, I))\
    mark_smallmap(M, I);\
  else if (RTCHECK(ok_address(M, B->fd)))\
    F = B->fd;\
  else {\
    CORRUPTION_ERROR_ACTION(M);\
  }\
  B->fd = P;\
  F->bk = P;\
  P->fd = F;\
  P->bk = B;\
}
```


在插入的同时, 会对small map进行维护操作.

3.2 tmalloc_small

tmalloc_small是在tree bins中分配small chunk的子函数.用于small request的核心分配算法3,即当remainderless和dv都无法满足,且剩余small bins也没有free chunk时,从tree bins中搜索.

代码本身其实比较容易理解, 源码注释如下,

```
04526: static void* tmalloc_small(mstate m, size_t nb) {
04527:   tchunkptr t, v;
04528:   size_t rsize;
04529:   bindex_t i;
04530:   /* 从tree bins中找到最小的可用分箱, 并获取tchunk指针 */
04531:   binmap_t leastbit = least_bit(m->treemap);
04532:   compute_bit2idx(leastbit, i);
04533:   v = t = *treebin_at(m, i);
04534:   /* 计算该root节点的remainder size */
04535:   rsize = chunksize(t) - nb;
04536:   /* 遍历DST树, 找到best-fit chunk, 并记录在v中 */
04537:   while ((t = leftmost_child(t)) != 0) {
04538:     size_t trem = chunksize(t) - nb;
04539:     if (trem < rsize) {
04540:       rsize = trem;
04541:       v = t;
04542:     }
04543:   }
04544:   /* 对v进行地址检查 */
04545:   if (RTCHECK(ok_address(m, v))) {
04546:     /* 得到v的remainder chunk r*/
04547:     mchunkptr r = chunk_plus_offset(v, nb);
04548:     assert(chunksize(v) == rsize + nb);
04549:     if (RTCHECK(ok_next(v, r))) {
04550:       /* 将v从DST树上摘除 */
04551:       unlink_large_chunk(m, v);
04552:       /* 如果remainder size小于最小可用chunk size, 则不切割v */
04553:       if (rsize < MIN_CHUNK_SIZE)
04554:         set_inuse_and_pinuse(m, v, (rsize + nb));
04555:       else {
04556:         /* 否则切割v, 并将r指定为新的dv */
04557:         set_size_and_pinuse_of_inuse_chunk(m, v, nb);
04558:         set_size_and_pinuse_of_free_chunk(r, rsize);
04559:         replace_dv(m, r, rsize);
04560:       }
```

```
04561:        /* 返回payload, 结束 */
04562:        return chunk2mem(v);
04563:      }
04564:    } ? end if RTCHECK(ok_address(m,v)) ?
04565:    /* 分配失败, 执行corruption action */
04566:    CORRUPTION_ERROR_ACTION(m);
04567:    return 0;
04568: } ? end tmalloc_small ?
```

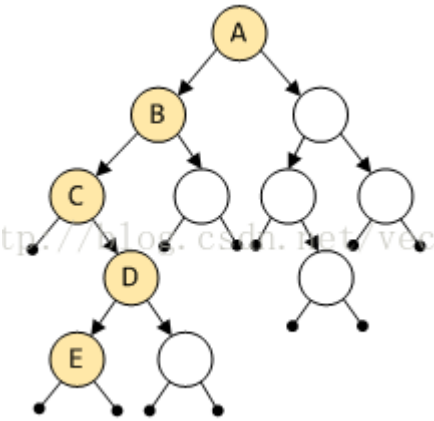
两点说明,

1. Line4537, 寻找DST最小节点通过宏leftmost_child完成,该宏的定义如下,

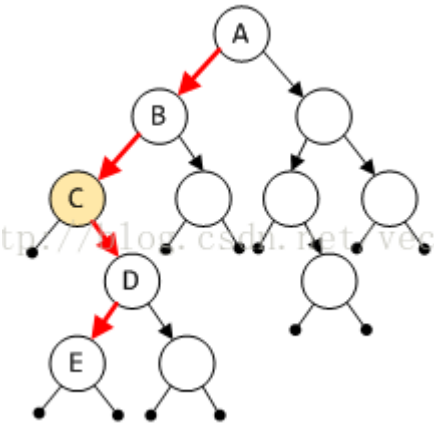
```
#define leftmost_child(t) ((t)->child[0] != 0? (t)->child[0] : (t)->child[1])
```

这里涉及到最小节点的遍历. 我们知道, 对于BST来说,根节点与左右子树有严格的排序关系,因此查找最小节点就是从根节点出发向左子树步进,一直遇到左子树为null停止的过程.

但如2.2.5小节中所述, DST本就不是一棵排序树,根节点同子树间不能确定大小关系,相比之下获取最小节点就更困难一些.但可以确定的一点是,同一级level中,越靠近左侧的子树节点就越小,因此我们可以大致圈定最小节点出现的范围,如下,



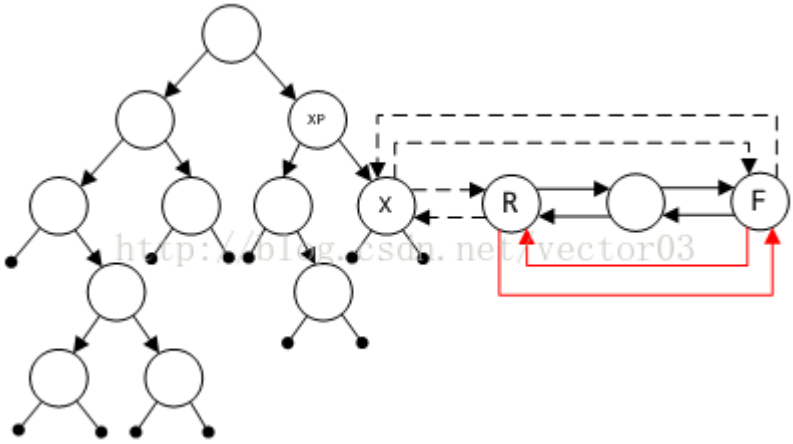
上图中用颜色标记了每一级level最左侧的节点(不限于左子树或右子树),尽管暂时还无法断定哪一个是最小节点,但它肯定出现在从A到E的路径上.所以DST的搜索路径为,从根节点出发,一路向左子树步进,若遇到左子树为空,就转头向右子树,一直遇到左右子树都为空停止.换句话说,沿着整棵DST的最左侧边缘走,如图所示



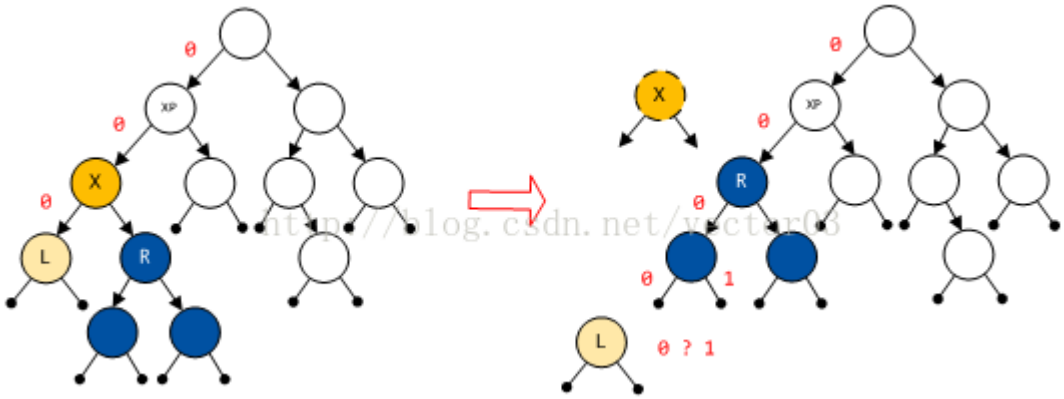
关于这一点, 我想应该是DST最大的缺陷,因为无论如何,遍历的次数是与树高相关的,上图中最小节点可能出现在位置C,但你需要完成每一次比较才能最后下决定.不过好在对于size_t等于4字节的系统,树高最多也只有32.无论如何这比线性查找还是要快得多了.

2. Line4551, 与DST的删除操作相关.由于unlink_large_chunk宏的代码比较长,还是先说明一下节点删除算法.基本上, Doug Lea的DST删除算法分为三个步骤,

第一步, 判断待删除节点X是否存在相同大小的兄弟节点.如果有,只要简单的将其从双链表上摘除再重新接好链表即可.如图,

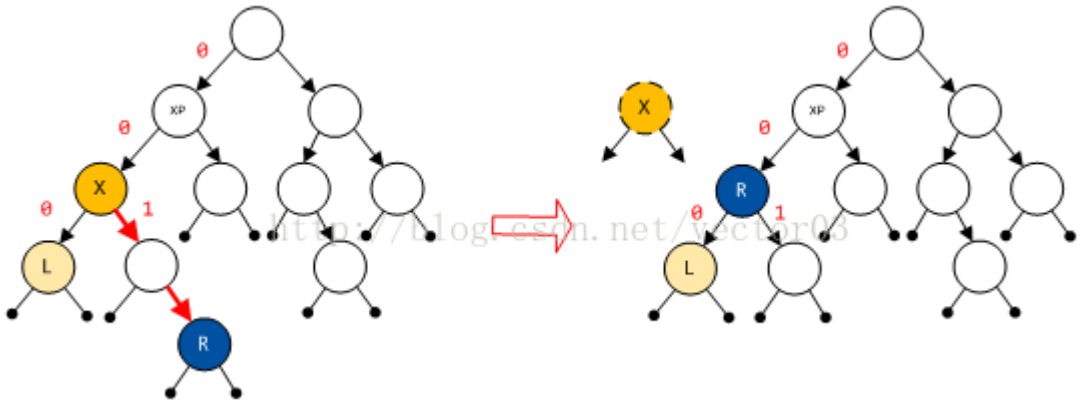


第二步, 如果节点X所在位置只有其一个节点,就需要选出一个继任节点R以替代X空缺的位置,同时还要保证DST的性质.由于DST也属于前缀树(prefix tree)的一种,因此子树节点提升level是很容易的,但降低level情况就相对复杂了.比如,子树节点前缀为0101,可以提升为010,但如果下降为0101x...x就必须参考其他子树的情况.如图,



这里如果我们选择R作为继任节点,则原节点X的左子树节点L就要改变其level.这时必须参考子树R的情况,为L寻找一个合适的插入点.如果R的内部很复杂,这个过程就会相对漫长.

因此Doug Lea取了个巧,他选择了right-most叶子节点作为X的继任节点.既然是叶节点,只需要简单的提升level即可,其他子树节点的level和位置都不会发生任何变化,于是就绕过了上述问题.选择right-most的原因还在于, dlmalloc在遍历best-fit节点时,会按照left-most的路径查找,导致多数情况下,左子树节点数少于右子树.为了平衡左右子树,同时削减子树高度,选择right-most相比更为合适,如图,



上图中查找到子树X的right-most节点R,用其替换X空缺的位置,可以看到L节点等子树节点位置没有任何变化.同时,改变R的位置平衡了左至右子树,让DST整体更均衡.

第三步, 这里就比较清楚了,只需要重新连接继任节点与原X的父节点和左右子树节点即可.

整个过程的源码注释如下,

```
03729: #define unlink_large_chunk(M, X) {\
03730:   /* 保存X的父节点XP */
03731:   tchunkptr XP = X->parent;\
03732:   tchunkptr R;\
03733:   /* 如果X存在相同size的兄弟节点 */
03734:   if (X->bk != X) {\
03735:     /* 从双链表中摘除X, 重新连接前后F,R节点 */
03736:     tchunkptr F = X->fd;\
03737:     R = X->bk;\
03738:     /* 地址和链接指针检查 */
03739:     if (RTCHECK(ok_address(M, F) && F->bk == X && R->fd == X)) {\
03740:       F->bk = R;\
03741:       R->fd = F;\
03742:     }\
03743:     else {\
03744:       CORRUPTION_ERROR_ACTION(M);\
03745:     }\
03746:   }\
03747:   /* X不存在相同size的节点 */
03748:   else {\
03749:     /* 保存继任节点的父节点链接指针RP */
03750:     tchunkptr* RP;\
03751:     if (((R = *(RP = &(X->child[1]))) != 0) ||\
03752:         ((R = *(RP = &(X->child[0]))) != 0)) {\
03753:       /* 查找right-most节点, 过程与left-most一致 */
03754:       tchunkptr* CP;\
03755:       while ((*(CP = &(R->child[1])) != 0) ||\
03756:              (*(CP = &(R->child[0])) != 0)) {\
03757:         R = *(RP = CP);\
03758:       }\
03759:       if (RTCHECK(ok_address(M, RP)))\
03760:         /* 摘除R节点 */
03761:         *RP = 0;\
03762:       else {\
03763:         CORRUPTION_ERROR_ACTION(M);\
03764:       }\
03765:     }\
03766:   }\
```

```
03767:   /* 如果X的父节点不为空, 则重新连接继任节点 */
03768:   if (XP != 0) {\
03769:     /* 获取X所在分箱指针 */
03770:     tbinptr* H = treebin_at(M, X->index);\
03771:     /* 如果X为分箱根节点, 且没有找到R节点, 则分箱耗尽, 清理bitmap */
03772:     if (X == *H) {\
03773:       if ((*H = R) == 0) \
03774:         clear_treemap(M, X->index);\
03775:     }\
03776:     else if (RTCHECK(ok_address(M, XP))) {\
03777:       /* X不为根节点, 将R节点与父节点XP重新连接 */
03778:       if (XP->child[0] == X) \
03779:         XP->child[0] = R;\
03780:       else \
03781:         XP->child[1] = R;\
03782:     }\
03783:     else\
03784:       CORRUPTION_ERROR_ACTION(M);\
03785:     /* 如果前面找到了R节点 */
03786:     if (R != 0) {\
03787:       if (RTCHECK(ok_address(M, R))) {\
03788:         tchunkptr C0, C1;\
03789:         /* R连接父节点XP */
03790:         R->parent = XP;\
03791:         /* 如果有, 则R连接原X的左右子树节点C0, C1 */
03792:         if ((C0 = X->child[0]) != 0) {\
03793:           if (RTCHECK(ok_address(M, C0))) {\
03794:             R->child[0] = C0;\
03795:             C0->parent = R;\
03796:           }\
03797:           else\
03798:             CORRUPTION_ERROR_ACTION(M);\
03799:         }\

03800:         if ((C1 = X->child[1]) != 0) {\
03801:           if (RTCHECK(ok_address(M, C1))) {\
03802:             R->child[1] = C1;\
03803:             C1->parent = R;\
03804:           }\
03805:           else\
03806:             CORRUPTION_ERROR_ACTION(M);\
03807:         }\
03808:       }\
03809:       else\
03810:         CORRUPTION_ERROR_ACTION(M);\
03811:     }\
03812:   }\
03813: }
```


3.3 tmalloc_large

该函数是在tree bins中分配large request的子函数.与tmalloc_small略有区别, large request并不是寻找最小节点,而是best-fit节点,即一个大于等于期望值的最小节点.

基本算法如下,

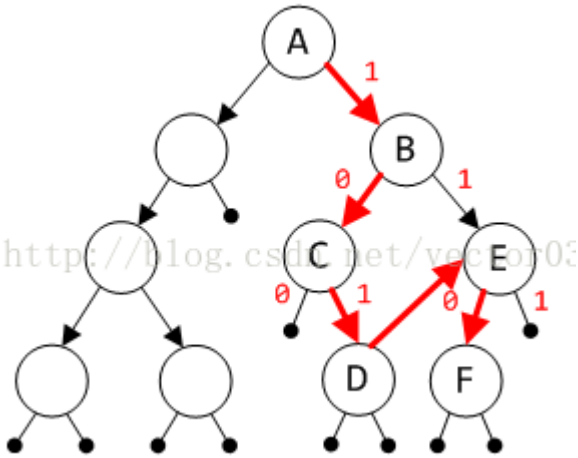
1. 以分配请求大小nb作为key值进行基值检索,并做两点记录.一是记录最接近的候选节点v,另一个记录当前最近的未被遍历的右子树节点rst (The deepest untaken right subtree).同时如果找到相同大小的chunk则立即返回.

2. 若已遍历到子树的最下层, 则返回记录的rst子树节点,从这个位置开始进行left-most遍历,这里同tmalloc_small中寻找最小子树节点是一致的.

3. 若找不到可用节点, 则从treemap中寻找最小可用分箱,从可用分箱中寻找.

4. 若dv比候选节点v更适合,则直接返回0,否则切割候选节点,并最终返回payload.

在同一分箱内的搜索过程如图所示,



在一个分箱内, 搜索best-fit节点按照从A到F的顺序执行.其中A-D属于基值检索,以nb的前缀为key值,而E-F则按照left-most检索,因为E子树是当前分箱中大于目标值的最小子树,只要找到最小节点即可.从这里也可以看出这个算法本质上很简单,就是先按照前缀寻找最接近的目标节点,如果没有则扩大范围在大于目标值的最小子树中搜索,还没有再到最接近的更大的分箱中查找,直到找到为止.

代码注释如下,

```
04455: static void* tmalloc_large(mstate m, size_t nb) {
04456:   tchunkptr v = 0;
04457:   /* 这里rsize为无符号表示 */
04458:   size_t rsize = -nb;
04459:   tchunkptr t;
04460:   bindex_t idx;
04461:   /* 根据请求大小nb计算分箱号 */
04462:   compute_tree_index(nb, idx);
04463:   if ((t = *treebin_at(m, idx)) != 0) {
04464:     /* 左移获得基值检索的掩码 */
04465:     size_t sizebits = nb << leftshift_for_tree_index(idx);
04466:     tchunkptr rst = 0;
04467:     for (;;) {
04468:       tchunkptr rt;
04469:       /* 计算并记录候选节点v */
04470:       size_t trem = chunksize(t) - nb;
04471:       if (trem < rsize) {
04472:         v = t;
04473:         /* 找到size正好符合的节点立即跳出循环 */
04474:         if ((rsize = trem) == 0)
04475:           break;
04476:       }
```

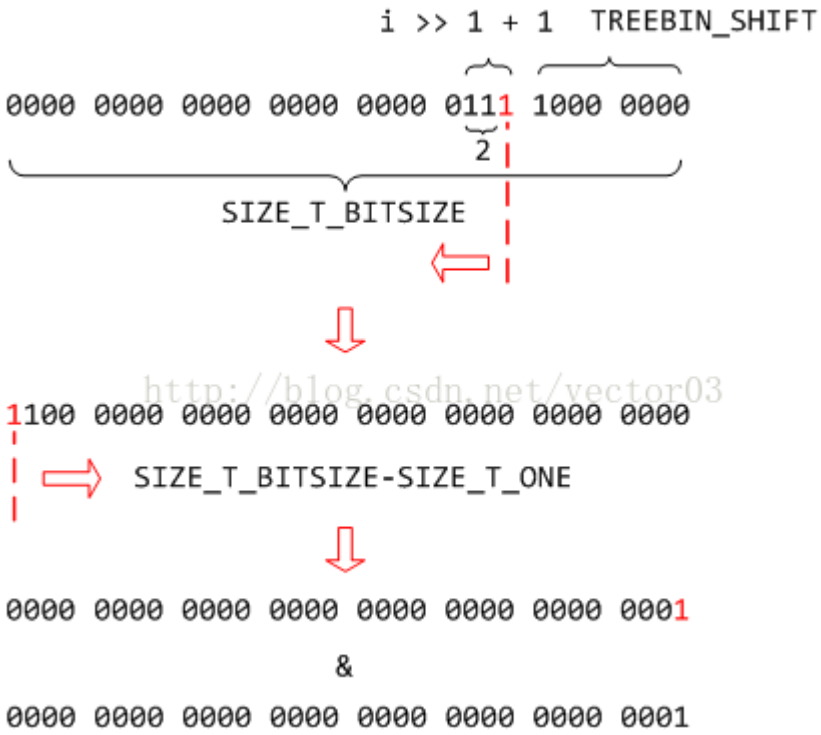
```
04477:       /* 保存右子树节点 */
04478:       rt = t->child[1];
04479:       /* 根据掩码步进到下一个子树节点 */
04480:       t = t->child[(sizebits >> (SIZE_T_BITSIZE-SIZE_T_ONE)) & 1];
04481:       /* 若步进节点t不是右子树节点，则保存rst，为后续的left-most检索准备 */
04482:       if (rt != 0 && rt != t)
04483:         rst = rt;
04484:       /* 若已步进到子树最下层，则返回rst节点，并退出第一阶段循环 */
04485:       if (t == 0) {
04486:         t = rst;
04487:         break;
04488:       }
04489:       /* 基值检索掩码右移 */
04490:       sizebits <<= 1;
04491:     } ? end for ;; ?
04492:   } ? end if (t=*treebin_at(m,idx)... ?
04493:   /* 若没有找到任何可用节点，则从treemap中找到最近可用的treebin开始 */
04494:   if (t == 0 && v == 0) {
04495:     binmap_t leftbits = left_bits(idx2bit(idx)) & m->treemap;
04496:     if (leftbits != 0) {
04497:       bindex_t i;
04498:       binmap_t leastbit = least_bit(leftbits);
04499:       compute_bit2idx(leastbit, i);
04500:       t = *treebin_at(m, i);
04501:     }
04502:   }
04503:   /* 开始二阶段left-most检索，包含rst和next treebin都在这里进行 */
04504:   while (t != 0) {
04505:     size_t trem = chunksize(t) - nb;
04506:     if (trem < rsize) {
04507:       rsize = trem;
04508:       v = t;
04509:     }

04510:     t = leftmost_child(t);
04511:   }
04512:   /* 若dv比候选节点v更适合，则直接返回0 */
04513:   if (v != 0 && rsize < (size_t)(m->dvsize - nb)) {
04514:     if (RTCHECK(ok_address(m, v))) {
04515:       /* 尝试切割节点v */
04516:       mchunkptr r = chunk_plus_offset(v, nb);
04517:       assert(chunksize(v) == rsize + nb);
04518:       if (RTCHECK(ok_next(v, r))) {
04519:         /* 将v从当前DST上摘除 */
04520:         unlink_large_chunk(m, v);
04521:         if (rsize < MIN_CHUNK_SIZE)
04522:           set_inuse_and_pinuse(m, v, (rsize + nb));
04523:         else {
04524:           /* 完成切割，将remainder chunk送回分箱系统中 */
04525:           set_size_and_pinuse_of_inuse_chunk(m, v, nb);
04526:           set_size_and_pinuse_of_free_chunk(r, rsize);
04527:           insert_chunk(m, r, rsize);
04528:         }
04529:         /* 返回payload，结束 */
04530:         return chunk2mem(v);
04531:       }
04532:     }
04533:     CORRUPTION_ERROR_ACTION(m);
04534:   } ? end if v!=0&&rsize<(size_t)(... ?
04535:   return 0;
04536: } ? end tmalloc_large ?
```

Line4465, Line4480, Line4490,这三处地方其实就是对nb掩码逐bit位的测试操作,以进行基值检索.

该宏展开如下,

```
#define leftshift_for_tree_index(i) \
   ((i == NTREEBINS-1)? 0 : \
    ((SIZE_T_BITSIZE-SIZE_T_ONE) - (((i) >> 1) + TREEBIN_SHIFT - 2)))
```

看上去有些复杂, 其实就是除最高有效位以及次最高有效位之外, 将后续bit位移动到msb端.之后每次循环就取出一位进行检测.如下图所示,



稍微不好理解的就是i >> 1的作用.回顾一下2.2.4小节中tree bins索引寻址的说明,这里就是computer_tree_index的逆运算.结果是不算末尾8bit,最高有效位的位号.这里减2的原因是最高有效和次高有效位用于计算分箱号,因此不计入key值.位移后获得的掩码在检测时会重新右移至最低位,并提取以决定是向左子树还是右子树步进.在整个循环中会不断左移掩码以保证遍历持续进行,直至达到最底层子树节点.

事实上这个宏Doug Lea搞得有点麻烦,这个运算用CLZ指令加2就能获得同样的结果.我猜Doug Lea不这样写的原因可能是尽量减少各个平台的区别,或者纯粹是他懒得再分别写四种实现.