Linux 内核从 slab 中分配内存空间上层函数由 kmalloc()或 kmem_cache_alloc()函数实现。

kmalloc()->__kmalloc()->__do_kmalloc()

**[cpp]** view plaincopyprint?

```
1. /**
2.  * __do_kmalloc - allocate memory
3.  * @size: how many bytes of memory are required.
4.  * @flags: the type of memory to allocate (see kmalloc).
5.  * @caller: function caller for debug tracking of the caller
6.  */
7. static __always_inline void *__do_kmalloc(size_t size, gfp_t flags,
8.                                            void *caller)
9. {
10.    struct kmem_cache *cachep;
11.    void *ret;
12.
13.    /* If you want to save a few bytes .text space: replace
14.     * __ with kmem_.
15.     * Then kmalloc uses the uninlined functions instead of the inline
16.     * functions.
17.     */
18.    /*查找指定大小的通用 cache，关于 sizes[]数组，在前面
19.    的初始化中就已经分析过了*/
20.    cachep = __find_general_cachep(size, flags);
21.    if (unlikely(ZERO_OR_NULL_PTR(cachep)))
22.        return cachep;
23.    /*实际的分配工作*/
24.    ret = __cache_alloc(cachep, flags, caller);
25.
26.    trace_kmalloc((unsigned long) caller, ret,
27.                  size, cachep->buffer_size, flags);
28.
29.    return ret;
30.}
```

实际的分配工作：__do_cache_alloc()->__cache_alloc()->____cache_alloc()

**[cpp]** view plaincopyprint?

```
1. /*从指定 cache 中分配对象*/
2. static inline void *____cache_alloc(struct kmem_cache *cachep, gfp_t flags)
3. {
4.     void *objp;
```

```
5.    struct array_cache *ac;
6.
7.    check_irq_off();
8.    /* 获得本 CPU 的 local cache */
9.    ac = cpu_cache_get(cachep);
10.   /* 如果 local cache 中有可用的空闲对象 */
11.   if (likely(ac->avail)) {
12.      /* 更新 local cache 命中计数 */
13.      STATS_INC_ALLOCHIT(cachep);
14.      /* touched 置 1 表示最近使用了 local cache，这会影响填充
15.      local cache 时的数目，最近使用的填充较多的对象 */
16.      ac->touched = 1;
17.       /* 从 local cache 的 entry 数组中提取最后面的空闲对象 */
18.      objp = ac->entry[--ac->avail];
19.   } else {
20.       /* local cache 中没有空闲对象，更新未命中计数 */
21.      STATS_INC_ALLOCMISS(cachep);
22.      /* 从 slab 三链中提取空闲对象填充到 local cache 中 */
23.      objp = cache_alloc_refill(cachep, flags);
24.#if 0/*这里是我新加的，这里可能是这个版本的一个 bug,在最新的内核里面这块已经加上了*/
25.      /*
26.                * the 'ac' may be updated by cache_alloc_refill(),
27.                * and kmemleak_erase() requires its correct value.
28.                */
29.           /* cache_alloc_refill 的 cache_grow 打开了中断，local cache 指针可能发生了变化，需
     要重新获得 */
30.               ac = cpu_cache_get(cachep);
31.#endif
32.   }
33.   /*
34.    * To avoid a false negative, if an object that is in one of the
35.    * per-CPU caches is leaked, we need to make sure kmemleak doesn't
36.    * treat the array pointers as a reference to the object.
37.    */ /* 分配出去的对象，其 entry 指针指向空 */
38.   kmemleak_erase(&ac->entry[ac->avail]);
39.   return objp;
40.}
```

**该函数的执行流程：**

1，从本地 CPU cache 中查找是否有空闲的对象；

2，如果本地 CPU cache 中没有空闲对象，从 slab 三链中提取空闲对象，此操作由函数 cache_alloc_refill()实现

1）如果存在共享本地 cache，那么将共享本地 cache 中的对象批量复制到本地 cache。

2）如果没有 shared local cache，或是其中没有空闲的对象，从 slab 链表中分配，其中，从 slab 中分配时，先查看部分空余链表，然后再查看空余链表。将 slab 链表中的数据先放到本地 CPU cache 中。

3）如果本地 CPU cache 中任然没有数据，那么只有重新创建一个 slab，然后再试。

**[cpp]** view plaincopyprint?

```
1. /*从 slab 三链中提取一部分空闲对象填充到 local cache 中*/
2. static void *cache_alloc_refill(struct kmem_cache *cachep, gfp_t flags)
3. {
4.     int batchcount;
5.     struct kmem_list3 *l3;
6.     struct array_cache *ac;
7.     int node;
8.
9. retry:
10.    check_irq_off();
11.    /* 获得本内存节点，UMA 只有一个节点 */
12.    node = numa_node_id();
13.    /* 获得本 CPU 的 local cache */
14.    ac = cpu_cache_get(cachep);
15.    /* 批量填充的数目，local cache 是按批填充的 */
16.    batchcount = ac->batchcount;
17.    if (!ac->touched && batchcount > BATCHREFILL_LIMIT) {
18.        /*
19.         * If there was little recent activity on this cache, then
20.         * perform only a partial refill.  Otherwise we could generate
21.         * refill bouncing.
22.         */
23.        /* 最近未使用过此 local cache，没有必要添加过多的对象
24.          ，添加的数目为默认的限定值 */
25.        batchcount = BATCHREFILL_LIMIT;
26.    }
27.    /* 获得本内存节点、本 cache 的 slab 三链 */
28.    l3 = cachep->nodelists[node];
29.
30.    BUG_ON(ac->avail > 0 || !l3);
31.    spin_lock(&l3->list_lock);
32.
33.    /* See if we can refill from the shared array */
```

```
34.  /* shared local cache 用于多核系统中，为所有 cpu 共享
35.   ，如果 slab cache 包含一个这样的结构
36.   ，那么首先从 shared local cache 中批量搬运空闲对象到 local cache 中
37.   。通过 shared local cache 使填充工作变得简单。*/
38.  if (l3->shared && transfer_objects(ac, l3->shared, batchcount))
39.      goto alloc_done;
40.
41.  /* 如果没有 shared local cache，或是其中没有空闲的对象
42.   ，从 slab 链表中分配 */
43.  while (batchcount > 0) {
44.      struct list_head *entry;
45.      struct slab *slabp;
46.      /* Get slab alloc is to come from. */
47.
48.      /* 先从部分满 slab 链表中分配 */
49.      entry = l3->slabs_partial.next;
50.      /* next 指向头节点本身，说明部分满 slab 链表为空 */
51.      if (entry == &l3->slabs_partial) {
52.          /* 表示刚刚访问了 slab 空链表 */
53.          l3->free_touched = 1;
54.          /* 检查空 slab 链表 */
55.          entry = l3->slabs_free.next;
56.          /* 空 slab 链表也为空，必须增加 slab 了 */
57.          if (entry == &l3->slabs_free)
58.              goto must_grow;
59.      }
60.      /* 获得链表节点所在的 slab */
61.      slabp = list_entry(entry, struct slab, list);
62.      /*调试用*/
63.      check_slabp(cachep, slabp);
64.      check_spinlock_acquired(cachep);
65.
66.      /*
67.       * The slab was either on partial or free list so
68.       * there must be at least one object available for
69.       * allocation.
70.       */
71.      BUG_ON(slabp->inuse >= cachep->num);
72.
73.      while (slabp->inuse < cachep->num && batchcount--) {
74.          /* 更新调试用的计数器 */
75.          STATS_INC_ALLOCED(cachep);
76.          STATS_INC_ACTIVE(cachep);
```

```
77.             STATS_SET_HIGH(cachep);
78.             /* 从 slab 中提取一个空闲对象，将其虚拟地址插入到 local cache 中 */
79.             ac->entry[ac->avail++] = slab_get_obj(cachep, slabp,
80.                                     node);
81.         }
82.         check_slabp(cachep, slabp);
83.
84.         /* move slabp to correct slabp list: */
85.         /* 从原链表中删除此 slab 节点，list 表示此
86.         slab 位于哪个链表（满、部分满、空）中 */
87.         list_del(&slabp->list);
88.         /*因为从中删除了一个 slab，需要从新检查*/
89.         if (slabp->free == BUFCTL_END)
90.             /* 此 slab 中已经没有空闲对象，添加到“full”slab 链表中 */
91.             list_add(&slabp->list, &l3->slabs_full);
92.         else
93.             /* 还有空闲对象，添加到“partial”slab 链表中 */
94.             list_add(&slabp->list, &l3->slabs_partial);
95.     }
96.
97.must_grow:
98.     /* 前面从 slab 链表中添加 avail 个空闲对象到 local cache 中
99.      ，更新 slab 链表的空闲对象数 */
100.     l3->free_objects -= ac->avail;
101.alloc_done:
102.     spin_unlock(&l3->list_lock);
103.     /* local cache 中仍没有可用的空闲对象，说明 slab
104.     三链中也没有空闲对象，需要创建新的空 slab 了 */
105.     if (unlikely(!ac->avail)) {
106.         int x;
107.         /* 创建一个空 slab */
108.         x = cache_grow(cachep, flags | GFP_THISNODE, node, NULL);
109.
110.         /* cache_grow can reenable interrupts, then ac could change. */
111.         /* 上面的操作使能了中断，此期间 local cache 指针可能发生了变化，需要重新获得 */
112.         ac = cpu_cache_get(cachep);
113.         /* 无法新增空 slab，local cache 中也没有空闲对象，表明系统已经无法分配新的空闲对象
    了 */
114.         if (!x && ac->avail == 0)    /* no objects in sight? abort */
115.             return NULL;
116.         /* 走到这有两种可能，第一种是无论新增空 slab 成功或失败，只要 avail 不为 0
117.          ，表明是其他进程重填了 local cache，本进程就不需要重填了
118.          ，不执行 retry 流程。第二种是 avail 为 0，并且新增空 slab 成功
```

```
119.        ，则进入 retry 流程，利用新分配的空 slab 填充 local cache */
120.       if (!ac->avail)      /* objects refilled by interrupt? */
121.           goto retry;
122.    }
123.    /* 重填了 local cache，设置近期访问标志 */
124.    ac->touched = 1;
125.    /* 返回 local cache 中最后一个空闲对象的虚拟地址 */
126.    return ac->entry[--ac->avail];
127.}
```

## 几个涉及到的辅助函数

**[cpp]** view plaincopyprint?

```
1.  /*
2.   * Transfer objects in one arraycache to another.
3.   * Locking must be handled by the caller.
4.   *
5.   * Return the number of entries transferred.
6.   */
7.  static int transfer_objects(struct array_cache *to,
8.          struct array_cache *from, unsigned int max)
9.  {
10.    /* Figure out how many entries to transfer */
11.    int nr = min(min(from->avail, max), to->limit - to->avail);
12.
13.    if (!nr)
14.        return 0;
15.    /*拷贝*/
16.    memcpy(to->entry + to->avail, from->entry + from->avail -nr,
17.           sizeof(void *) *nr);
18.    /*两边数据更新*/
19.    from->avail -= nr;
20.    to->avail += nr;
21.    to->touched = 1;
22.    return nr;
23.}
```

**[cpp]** view plaincopyprint?

```
1.  /*从 slab 中提取一个空闲对象*/
2.  static void *slab_get_obj(struct kmem_cache *cachep, struct slab *slabp,
3.                int nodeid)
4.  {
5.      /* 获得一个空闲的对象，free 是本 slab 中第一个空闲对象的索引 */
6.      void *objp = index_to_obj(cachep, slabp, slabp->free);
```

```
7.      kmem_bufctl_t next;
8.      /* 更新在用对象计数 */
9.      slabp->inuse++;
10.     /* 获得下一个空闲对象的索引 */
11.     next = slab_bufctl(slabp)[slabp->free];
12.#if DEBUG
13.     slab_bufctl(slabp)[slabp->free] = BUFCTL_FREE;
14.     WARN_ON(slabp->nodeid != nodeid);
15.#endif
16.     /* free 指向下一个空闲的对象 */
17.     slabp->free = next;
18.
19.     return objp;
20.}
```

**[cpp]** view plaincopyprint?

```
1. static inline void *index_to_obj(struct kmem_cache *cache, struct slab *slab,
2.                 unsigned int idx)
3. {   /* s_mem 是 slab 中第一个对象的起始地址，buffer_size 是每个对象的大小
4.      ，这里根据对象索引计算对象的地址 */
5.     return slab->s_mem + cache->buffer_size * idx;
6. }
```

**[cpp]** view plaincopyprint?

```
1. static inline kmem_bufctl_t *slab_bufctl(struct slab *slabp)
2. {
3.     return (kmem_bufctl_t *) (slabp + 1);
4. }
```

**[cpp]** view plaincopyprint?

```
1. static inline struct array_cache *cpu_cache_get(struct kmem_cache *cachep)
2. {
3.     return cachep->array[smp_processor_id()];
4. }
```

总结：从 slab 分配器中分配空间实际工作很简单，先查看本地 CPU cache，然后是本地共享 CPU cache，最后是三链。前面三个都没有空间时，需要从新分配 slab。可以看出，从 slab 分配器中分配内存空间一般不会申请不到空间，也就是说返回空 的可能性很小。