Linux 内核中将对象释放到 slab 中上层所用函数为 kfree()或 kmem_cache_free()。两个函数都会调用__cache_free()函数。

代码执行流程：

1，当本地 CPU cache 中空闲对象数小于规定上限时，只需将对象放入本地 CPU cache 中；

2，当 local cache 中对象过多（大于等于规定上限），需要释放一批对象到 slab 三链中。由函数 cache_flusharray()实现。

1）如果三链中存在共享本地 cache，那么首先选择释放到共享本地 cache 中，能释放多少是多少；

2）如果没有 shared local cache，释放对象到 slab 三链中，实现函数为 free_block()。对于 free_block()函数，当三链中的空闲对象数过多时，销毁此 cache。不然，添加此 slab 到空闲链表。因为在分配的时候我们看到将 slab 结构从 cache 链表中脱离了，在这里，根据 page 描述符的 lru 找到 slab 并将它添加到三链的空闲链表中。

主实现

[cpp] view plaincopyprint?

```
1. /*
2.  * Release an obj back to its cache. If the obj has a constructed state, it must
3.  * be in this state _before_ it is released.  Called with disabled ints.
4.  */
5. static inline void __cache_free(struct kmem_cache *cachep, void *objp)
6. {
7.     /* 获得本 CPU 的 local cache */
8.     struct array_cache *ac = cpu_cache_get(cachep);
9.
10.    check_irq_off();
11.    kmemleak_free_recursive(objp, cachep->flags);
12.    objp = cache_free_debugcheck(cachep, objp, __builtin_return_address(0));
13.
14.    kmemcheck_slab_free(cachep, objp, obj_size(cachep));
15.
16.    /*
17.     * Skip calling cache_free_alien() when the platform is not numa.
18.     * This will avoid cache misses that happen while accessing slabp (which
19.     * is per page memory  reference) to get nodeid. Instead use a global
20.     * variable to skip the call, which is mostly likely to be present in
```

```
21.     * the cache.
22.     *//* NUMA 相关 */
23.    if (nr_online_nodes > 1 && cache_free_alien(cachep, objp))
24.        return;
25.
26.    if (likely(ac->avail < ac->limit)) {
27.        /* local cache 中的空闲对象数小于上限时
28.         ，只需将对象释放回 entry 数组中 */
29.        STATS_INC_FREEHIT(cachep);
30.        ac->entry[ac->avail++] = objp;
31.        return;
32.    } else {
33.        /* 大于等于上限时， */
34.        STATS_INC_FREEMISS(cachep);
35.        /* local cache 中对象过多，需要释放一批对象到 slab 三链中。*/
36.        cache_flusharray(cachep, ac);
37.        ac->entry[ac->avail++] = objp;
38.    }
39.}
```

释放对象到三链中

[cpp] view plaincopyprint?

```
1.  /*local cache 中对象过多，需要释放一批对象到 slab 三链中。*/
2.  static void cache_flusharray(struct kmem_cache *cachep, struct array_cache *ac)
3.  {
4.      int batchcount;
5.      struct kmem_list3 *l3;
6.      int node = numa_node_id();
7.      /* 每次释放多少个对象 */
8.      batchcount = ac->batchcount;
9.  #if DEBUG
10.     BUG_ON(!batchcount || batchcount > ac->avail);
11.#endif
12.     check_irq_off();
13.     /* 获得此 cache 的 slab 三链 */
14.     l3 = cachep->nodelists[node];
15.     spin_lock(&l3->list_lock);
16.     if (l3->shared) {
17.         /* 如果存在 shared local cache，将对象释放到其中 */
18.         struct array_cache *shared_array = l3->shared;
19.         /* 计算 shared local cache 中还有多少空位 */
20.         int max = shared_array->limit - shared_array->avail;
21.         if (max) {
```

```
22.         /* 空位数小于要释放的对象数时，释放数等于空位数 */
23.         if (batchcount > max)
24.             batchcount = max;
25.         /* 释放 local cache 前面的几个对象到 shared local cache 中
26.          ，前面的是最早不用的 */
27.         memcpy(&(shared_array->entry[shared_array->avail]),
28.                 ac->entry, sizeof(void *) * batchcount);
29.         /* 增加 shared local cache 可用对象数 */
30.         shared_array->avail += batchcount;
31.         goto free_done;
32.     }
33.   }
34.   /* 无 shared local cache，释放对象到 slab 三链中 */
35.   free_block(cachep, ac->entry, batchcount, node);
36.free_done:
37.#if STATS
38.   {
39.     int i = 0;
40.     struct list_head *p;
41.
42.     p = l3->slabs_free.next;
43.     while (p != &(l3->slabs_free)) {
44.         struct slab *slabp;
45.
46.         slabp = list_entry(p, struct slab, list);
47.         BUG_ON(slabp->inuse);
48.
49.         i++;
50.         p = p->next;
51.     }
52.     STATS_SET_FREEABLE(cachep, i);
53.   }
54.#endif
55.   spin_unlock(&l3->list_lock);
56.   /* 减少 local cache 可用对象数*/
57.   ac->avail -= batchcount;
58.   /* local cache 前面有 batchcount 个空位，将后面的对象依次前移 batchcount 位 */
59.   memmove(ac->entry, &(ac->entry[batchcount]), sizeof(void *)*ac->avail);
60.}
```

无 shared local cache，释放对象到 slab 三链中

[cpp] view plaincopyprint?

```
1.  /*
```

```
2.  * Caller needs to acquire correct kmem_list's list_lock
3.  */
4.  /*释放一定数目的对象*/
5.  static void free_block(struct kmem_cache *cachep, void **objpp, int nr_objects,
6.              int node)
7.  {
8.      int i;
9.      struct kmem_list3 *l3;
10.     /* 逐一释放对象到 slab 三链中 */
11.     for (i = 0; i < nr_objects; i++) {
12.         void *objp = objpp[i];
13.         struct slab *slabp;
14.         /* 通过虚拟地址得到 page，再通过 page 得到 slab */
15.         slabp = virt_to_slab(objp);
16.         /* 获得 slab 三链 */
17.         l3 = cachep->nodelists[node];
18.         /* 先将对象所在的 slab 从链表中摘除 */
19.         list_del(&slabp->list);
20.         check_spinlock_acquired_node(cachep, node);
21.         check_slabp(cachep, slabp);
22.         /* 将对象释放到其 slab 中 */
23.         slab_put_obj(cachep, slabp, objp, node);
24.         STATS_DEC_ACTIVE(cachep);
25.         /* 空闲对象数加一 */
26.         l3->free_objects++;
27.         check_slabp(cachep, slabp);
28.
29.         /* fixup slab chains */
30.         if (slabp->inuse == 0) {
31.             /* 如果 slab 中均为空闲对象 */
32.             if (l3->free_objects > l3->free_limit) {
33.                 /* 如果 slab 三链中空闲对象数超过上限
34.                  ，直接回收整个 slab 到内存
35.                  ，空闲对象数减去每个 slab 中对象数 */
36.                 l3->free_objects -= cachep->num;
37.                 /* No need to drop any previously held
38.                  * lock here, even if we have a off-slab slab
39.                  * descriptor it is guaranteed to come from
40.                  * a different cache, refer to comments before
41.                  * alloc_slabmgmt.
42.                  *//* 销毁 struct slab 对象 */
43.                 slab_destroy(cachep, slabp);
44.             } else {
```

```
45.             /* 将此 slab 添加到空 slab 链表中 */
46.             list_add(&slabp->list, &l3->slabs_free);
47.         }
48.     } else {
49.         /* Unconditionally move a slab to the end of the
50.          * partial list on free - maximum time for the
51.          * other objects to be freed, too.
52.          *//*将此 slab 添加到部分满 slab 链表中*/
53.         list_add_tail(&slabp->list, &l3->slabs_partial);
54.     }
55. }
56.}
```

将对象释放到其 slab 中

[cpp] view plaincopyprint?

```
1. static void slab_put_obj(struct kmem_cache *cachep, struct slab *slabp,
2.             void *objp, int nodeid)
3. {   /* 获得对象在 kmem_bufctl_t 数组中的索引 */
4.     unsigned int objnr = obj_to_index(cachep, slabp, objp);
5.
6. #if DEBUG
7.     /* Verify that the slab belongs to the intended node */
8.     WARN_ON(slabp->nodeid != nodeid);
9.
10.    if (slab_bufctl(slabp)[objnr] + 1 <= SLAB_LIMIT + 1) {
11.        printk(KERN_ERR "slab: double free detected in cache "
12.                "'%s', objp %p\n", cachep->name, objp);
13.        BUG();
14.    }
15.#endif
16.    /*这两步相当于静态链表的插入操作*/
17.    /* 指向 slab 中原来的第一个空闲对象 */
18.    slab_bufctl(slabp)[objnr] = slabp->free;
19.    /* 释放的对象作为第一个空闲对象 */
20.    slabp->free = objnr;
21.    /* 已分配对象数减一 */
22.    slabp->inuse--;
23.}
```

辅助函数

[cpp] view plaincopyprint?

```
1. /* 通过虚拟地址得到 page，再通过 page 得到 slab */
```

```
2. static inline struct slab *virt_to_slab(const void *obj)
3. {
4.     struct page *page = virt_to_head_page(obj);
5.     return page_get_slab(page);
6. }
```

[cpp] view plaincopyprint?

```
1. static inline struct slab *page_get_slab(struct page *page)
2. {
3.     BUG_ON(!PageSlab(page));
4.     return (struct slab *)page->lru.prev;
5. }
```

可见，用 page->lru.prev 得到 slab，和创建 slab 时相呼应。