内核启动早期初始化

start_kernel()->mm_init()->kmem_cache_init()

执行流程：

1，初始化静态 initkmem_list3 三链；

2，初始化 cache_cache 的 nodelists 字段为 1 中的三链；

3，根据内存情况初始化每个 slab 占用的页面数变量 slab_break_gfp_order；

4，将 cache_cache 加入 cache_chain 链表中，初始化 cache_cache；

5，创建 kmalloc 所用的 general cache：

1）cache 的名称和大小存放在两个数据结构对应的数组中，对应大小的 cache 可以从 size 数组中找到；

2）先创建 INDEX_AC 和 INDEX_L3 下标的 cache；

3）循环创建 size 数组中各个大小的 cache；

6，替换静态本地 cache 全局变量：

1) 替换 cache_cache 中的 arry_cache,本来指向静态变量 initarray_cache.cache；

2) 替换 malloc_sizes[INDEX_AC].cs_cachep 的 local cache，原本指向静态变量 initarray_generic.cache；

7，替换静态三链

1）替换 cache_cache 三链，原本指向静态变量 initkmem_list3；

2）替换 malloc_sizes[INDEX_AC].cs_cachep 三链，原本指向静态变量 initkmem_list3；

8，更新初始化进度

[cpp] view plaincopyprint?

```
1. /*
2.  * Initialisation.  Called after the page allocator have been initialised and
3.  * before smp_init().
4.  */
5. void __init kmem_cache_init(void)
```

```
6. {
7.     size_t left_over;
8.     struct cache_sizes *sizes;
9.     struct cache_names *names;
10.    int i;
11.    int order;
12.    int node;
13.    /* 在 slab 初始化好之前，无法通过 kmalloc 分配初始化过程中必要的一些对象
14.     ，只能使用静态的全局变量
15.     ，待 slab 初始化后期，再使用 kmalloc 动态分配的对象替换全局变量 */
16.
17.    /* 如前所述，先借用全局变量 initkmem_list3 表示的 slab 三链
18.     ，每个内存节点对应一组 slab 三链。initkmem_list3 是个 slab 三链数组，对于每个内存节点，
    包含三组
19.     ：struct kmem_cache 的 slab 三链、struct arraycache_init 的 slab 三链、struct kmem_list3
    的 slab 三链
20.    。这里循环初始化所有内存节点的所有 slab 三链 */
21.    if (num_possible_nodes() == 1)
22.        use_alien_caches = 0;
23.    /*初始化所有 node 的所有 slab 中的三个链表*/
24.    for (i = 0; i < NUM_INIT_LISTS; i++) {
25.        kmem_list3_init(&initkmem_list3[i]);
26.        /* 全局变量 cache_cache 指向的 slab cache 包含所有 struct kmem_cache 对象，不包含
    cache_cache 本身
27.        。这里初始化所有内存节点的 struct kmem_cache 的 slab 三链为空。*/
28.        if (i < MAX_NUMNODES)
29.            cache_cache.nodelists[i] = NULL;
30.    }
31.    /* 设置 struct kmem_cache 的 slab 三链指向 initkmem_list3 中的一组 slab 三链，
32.    CACHE_CACHE 为 cache 在内核 cache 链表中的索引，
33.    struct kmem_cache 对应的 cache 是内核中创建的第一个 cache
34.     ，故 CACHE_CACHE 为 0 */
35.    set_up_list3s(&cache_cache, CACHE_CACHE);
36.
37.    /*
38.     * Fragmentation resistance on low memory - only use bigger
39.     * page orders on machines with more than 32MB of memory.
40.     */
41.     /* 全局变量 slab_break_gfp_order 为每个 slab 最多占用几个页面
42.      ，用来抑制碎片，比如大小为 3360 的对象
43.      ，如果其 slab 只占一个页面，碎片为 736
44.      ，slab 占用两个页面，则碎片大小也翻倍
45.      。只有当对象很大
```

```
46.    ，以至于 slab 中连一个对象都放不下时
47.    ，才可以超过这个值
48.    。有两个可能的取值
49.    ：当可用内存大于 32MB 时
50.    ，BREAK_GFP_ORDER_HI 为 1
51.    ，即每个 slab 最多占用 2 个页面
52.    ，只有当对象大小大于 8192 时
53.    ，才可以突破 slab_break_gfp_order 的限制
54.    。小于等于 32MB 时 BREAK_GFP_ORDER_LO 为 0。*/
55.   if (totalram_pages > (32 << 20) >> PAGE_SHIFT)
56.       slab_break_gfp_order = BREAK_GFP_ORDER_HI;
57.
58.   /* Bootstrap is tricky, because several objects are allocated
59.    * from caches that do not exist yet:
60.    * 1) initialize the cache_cache cache: it contains the struct
61.    *    kmem_cache structures of all caches, except cache_cache itself:
62.    *    cache_cache is statically allocated.
63.    *    Initially an __init data area is used for the head array and the
64.    *    kmem_list3 structures, it's replaced with a kmalloc allocated
65.    *    array at the end of the bootstrap.
66.    * 2) Create the first kmalloc cache.
67.    *    The struct kmem_cache for the new cache is allocated normally.
68.    *    An __init data area is used for the head array.
69.    * 3) Create the remaining kmalloc caches, with minimally sized
70.    *    head arrays.
71.    * 4) Replace the __init data head arrays for cache_cache and the first
72.    *    kmalloc cache with kmalloc allocated arrays.
73.    * 5) Replace the __init data for kmem_list3 for cache_cache and
74.    *    the other cache's with kmalloc allocated memory.
75.    * 6) Resize the head arrays of the kmalloc caches to their final sizes.
76.    */
77.
78.   node = numa_node_id();
79.
80.   /* 1) create the cache_cache */
81.   /* 第一步，创建 struct kmem_cache 所在的 cache，由全局变量 cache_cache 指向
82.    ，这里只是初始化数据结构
83.    ，并未真正创建这些对象，要待分配时才创建。*/
84.   /* 全局变量 cache_chain 是内核 slab cache 链表的表头 */
85.   INIT_LIST_HEAD(&cache_chain);
86.
87.   /* 将 cache_cache 加入到 slab cache 链表 */
88.   list_add(&cache_cache.next, &cache_chain);
```

```
89.
90.   /* 设置 cache 着色基本单位为 cache line 的大小：32 字节 */
91.   cache_cache.colour_off = cache_line_size();
92.   /*  初始化 cache_cache 的 local cache，同样这里也不能使用 kmalloc
93.    ，需要使用静态分配的全局变量 initarray_cache */
94.   cache_cache.array[smp_processor_id()] = &initarray_cache.cache;
95.   /* 初始化 slab 链表 ,用全局变量*/
96.   cache_cache.nodelists[node] = &initkmem_list3[CACHE_CACHE + node];
97.
98.   /*
99.    * struct kmem_cache size depends on nr_node_ids, which
100.    * can be less than MAX_NUMNODES.
101.    */
102.    /* buffer_size 保存 slab 中对象的大小，这里是计算 struct kmem_cache 的大小
103.     ， nodelists 是最后一个成员
104.     ，nr_node_ids 保存内存节点个数，UMA 为 1
105.     ，所以 nodelists 偏移加上 1 个 struct kmem_list3 的大小即为 struct kmem_cache 的大小 *
    /
106.   cache_cache.buffer_size = offsetof(struct kmem_cache, nodelists) +
107.              nr_node_ids * sizeof(struct kmem_list3 *);
108.#if DEBUG
109.   cache_cache.obj_size = cache_cache.buffer_size;
110.#endif
111.   /* 将对象大小与 cache line 大小对齐 */
112.   cache_cache.buffer_size = ALIGN(cache_cache.buffer_size,
113.                cache_line_size());
114.   /* 计算对象大小的倒数，用于计算对象在 slab 中的索引 */
115.   cache_cache.reciprocal_buffer_size =
116.      reciprocal_value(cache_cache.buffer_size);
117.
118.   for (order = 0; order < MAX_ORDER; order++) {
119.      /* 计算 cache_cache 中的对象数目 */
120.      cache_estimate(order, cache_cache.buffer_size,
121.         cache_line_size(), 0, &left_over, &cache_cache.num);
122.      /* num 不为 0 意味着创建 struct kmem_cache 对象成功，退出 */
123.      if (cache_cache.num)
124.         break;
125.   }
126.   BUG_ON(!cache_cache.num);
127.    /* gfporder 表示本 slab 包含 2^gfporder 个页面 */
128.   cache_cache.gfporder = order;
129.    /* 着色区的大小，以 colour_off 为单位 */
130.   cache_cache.colour = left_over / cache_cache.colour_off;
```

```
131.  /* slab 管理对象的大小 */
132.  cache_cache.slab_size = ALIGN(cache_cache.num * sizeof(kmem_bufctl_t) +
133.                  sizeof(struct slab), cache_line_size());
134.
135.  /* 2+3) create the kmalloc caches */
136.  /* 第二步，创建 kmalloc 所用的 general cache
137.   ，kmalloc 所用的对象按大小分级
138.   ，malloc_sizes 保存大小，cache_names 保存 cache 名 */
139.  sizes = malloc_sizes;
140.  names = cache_names;
141.
142.  /*
143.   * Initialize the caches that provide memory for the array cache and the
144.   * kmem_list3 structures first.  Without this, further allocations will
145.   * bug.
146.   */
147.  /* 首先创建 struct array_cache 和 struct kmem_list3 所用的 general cache
148.   ，它们是后续初始化动作的基础 */
149.  /* INDEX_AC 是计算 local cache 所用的 struct arraycache_init 对象在 kmalloc size 中的
     索引
150.   ，即属于哪一级别大小的 general cache
151.   ，创建此大小级别的 cache 为 local cache 所用 */
152.  sizes[INDEX_AC].cs_cachep = kmem_cache_create(names[INDEX_AC].name,
153.              sizes[INDEX_AC].cs_size,
154.              ARCH_KMALLOC_MINALIGN,
155.              ARCH_KMALLOC_FLAGS|SLAB_PANIC,
156.              NULL);
157.  /* 如果 struct kmem_list3 和 struct arraycache_init 对应的 kmalloc size 索引不同
158.   ，即大小属于不同的级别
159.   ，则创建 struct kmem_list3 所用的 cache，否则共用一个 cache */
160.  if (INDEX_AC != INDEX_L3) {
161.     sizes[INDEX_L3].cs_cachep =
162.        kmem_cache_create(names[INDEX_L3].name,
163.           sizes[INDEX_L3].cs_size,
164.           ARCH_KMALLOC_MINALIGN,
165.           ARCH_KMALLOC_FLAGS|SLAB_PANIC,
166.           NULL);
167.  }
168.  /* 创建完上述两个 general cache 后，slab early init 阶段结束，在此之前
169.   ，不允许创建外置式 slab */
170.  slab_early_init = 0;
171.
172.  /* 循环创建 kmalloc 各级别的 general cache */
```

```
173.    while (sizes->cs_size != ULONG_MAX) {
174.        /*
175.         * For performance, all the general caches are L1 aligned.
176.         * This should be particularly beneficial on SMP boxes, as it
177.         * eliminates "false sharing".
178.         * Note for systems short on memory removing the alignment will
179.         * allow tighter packing of the smaller caches.
180.         */
181.        /* 某级别的 kmalloc cache 还未创建，创建之，struct kmem_list3 和
182.        struct arraycache_init 对应的 cache 已经创建过了 */
183.        if (!sizes->cs_cachep) {
184.            sizes->cs_cachep = kmem_cache_create(names->name,
185.                    sizes->cs_size,
186.                    ARCH_KMALLOC_MINALIGN,
187.                    ARCH_KMALLOC_FLAGS|SLAB_PANIC,
188.                    NULL);
189.        }
190.#ifdef CONFIG_ZONE_DMA
191.        sizes->cs_dmacachep = kmem_cache_create(
192.                    names->name_dma,
193.                    sizes->cs_size,
194.                    ARCH_KMALLOC_MINALIGN,
195.                    ARCH_KMALLOC_FLAGS|SLAB_CACHE_DMA|
196.                        SLAB_PANIC,
197.                    NULL);
198.#endif
199.        sizes++;
200.        names++;
201.    }
202.    /* 至此，kmalloc general cache 已经创建完毕，可以拿来使用了 */
203.    /* 4) Replace the bootstrap head arrays */
204.    /* 第四步，用 kmalloc 对象替换静态分配的全局变量
205.    。到目前为止一共使用了两个全局 local cache
206.    ，一个是 cache_cache 的 local cache 指向 initarray_cache.cache
207.    ，另一个是 malloc_sizes[INDEX_AC].cs_cachep 的 local cache 指向
    initarray_generic.cache
208.    ，参见 setup_cpu_cache 函数。这里替换它们。*/
209.    {
210.        struct array_cache *ptr;
211.        /* 申请 cache_cache 所用 local cache 的空间 */
212.        ptr = kmalloc(sizeof(struct arraycache_init), GFP_NOWAIT);
213.
214.        BUG_ON(cpu_cache_get(&cache_cache) != &initarray_cache.cache);
```

```
215.      /* 复制原 cache_cache 的 local cache，即 initarray_cache，到新的位置 */
216.      memcpy(ptr, cpu_cache_get(&cache_cache),
217.             sizeof(struct arraycache_init));
218.      /*
219.       * Do not assume that spinlocks can be initialized via memcpy:
220.       */
221.      spin_lock_init(&ptr->lock);
222.      /* cache_cache 的 local cache 指向新的位置 */
223.      cache_cache.array[smp_processor_id()] = ptr;
224.      /* 申请 malloc_sizes[INDEX_AC].cs_cachep 所用 local cache 的空间 */
225.      ptr = kmalloc(sizeof(struct arraycache_init), GFP_NOWAIT);
226.
227.      BUG_ON(cpu_cache_get(malloc_sizes[INDEX_AC].cs_cachep)
228.             != &initarray_generic.cache);
229.      /* 复制原 local cache 到新分配的位置，注意此时 local cache 的大小是固定的 */
230.      memcpy(ptr, cpu_cache_get(malloc_sizes[INDEX_AC].cs_cachep),
231.             sizeof(struct arraycache_init));
232.      /*
233.       * Do not assume that spinlocks can be initialized via memcpy:
234.       */
235.      spin_lock_init(&ptr->lock);
236.
237.      malloc_sizes[INDEX_AC].cs_cachep->array[smp_processor_id()] =
238.          ptr;
239.  }
240.  /* 5) Replace the bootstrap kmem_list3's */
241.  /* 第五步，与第四步类似，用 kmalloc 的空间替换静态分配的 slab 三链 */
242.  {
243.      int nid;
244.     /* UMA 只有一个节点 */
245.      for_each_online_node(nid) {
246.          /* 复制 struct kmem_cache 的 slab 三链 */
247.          init_list(&cache_cache, &initkmem_list3[CACHE_CACHE + nid], nid);
248.          /* 复制 struct arraycache_init 的 slab 三链 */
249.          init_list(malloc_sizes[INDEX_AC].cs_cachep,
250.                &initkmem_list3[SIZE_AC + nid], nid);
251.          /* 复制 struct kmem_list3 的 slab 三链 */
252.          if (INDEX_AC != INDEX_L3) {
253.              init_list(malloc_sizes[INDEX_L3].cs_cachep,
254.                    &initkmem_list3[SIZE_L3 + nid], nid);
255.          }
256.      }
257.  }
```

```
258.    /* 更新 slab 系统初始化进度 */
259.    g_cpucache_up = EARLY;
260.}
```

辅助操作

1，slab 三链初始化

[cpp] view plaincopyprint?

```
1.  static void kmem_list3_init(struct kmem_list3 *parent)
2.  {
3.      INIT_LIST_HEAD(&parent->slabs_full);
4.      INIT_LIST_HEAD(&parent->slabs_partial);
5.      INIT_LIST_HEAD(&parent->slabs_free);
6.      parent->shared = NULL;
7.      parent->alien = NULL;
8.      parent->colour_next = 0;
9.      spin_lock_init(&parent->list_lock);
10.     parent->free_objects = 0;
11.     parent->free_touched = 0;
12.}
```

2，slab 三链静态数据初始化

[cpp] view plaincopyprint?

```
1.   /*设置 cache 的 slab 三链指向静态分配的全局变量*/
2.  static void __init set_up_list3s(struct kmem_cache *cachep, int index)
3.  {
4.      int node;
5.      /* UMA 只有一个节点 */
6.      for_each_online_node(node) {
7.          /* 全局变量 initkmem_list3 是初始化阶段使用的 slab 三链 */
8.          cachep->nodelists[node] = &initkmem_list3[index + node];
9.          /* 设置回收时间 */
10.         cachep->nodelists[node]->next_reap = jiffies +
11.             REAPTIMEOUT_LIST3 +
12.             ((unsigned long)cachep) % REAPTIMEOUT_LIST3;
13.     }
14.}
```

3，计算每个 slab 中对象的数目

[cpp] view plaincopyprint?

```
1.  /*
2.   * Calculate the number of objects and left-over bytes for a given buffer size.
```

```
3.  */
4.  /*计算每个 slab 中对象的数目。*/
5.  /*
6.  1)      gfporder：slab 由 2gfporder 个页面组成。
7.  2)      buffer_size：对象的大小。
8.  3)      align：对象的对齐方式。
9.  4)      flags：内置式 slab 还是外置式 slab。
10.5)       left_over：slab 中浪费空间的大小。
11.6)       num：slab 中的对象数目。
12.*/
13.static void cache_estimate(unsigned long gfporder, size_t buffer_size,
14.             size_t align, int flags, size_t *left_over,
15.             unsigned int *num)
16.{
17.    int nr_objs;
18.    size_t mgmt_size;
19.    /* slab 大小为 1<<order 个页面 */
20.    size_t slab_size = PAGE_SIZE << gfporder;
21.
22.    /*
23.     * The slab management structure can be either off the slab or
24.     * on it. For the latter case, the memory allocated for a
25.     * slab is used for:
26.     *
27.     * - The struct slab
28.     * - One kmem_bufctl_t for each object
29.     * - Padding to respect alignment of @align
30.     * - @buffer_size bytes for each object
31.     *
32.     * If the slab management structure is off the slab, then the
33.     * alignment will already be calculated into the size. Because
34.     * the slabs are all pages aligned, the objects will be at the
35.     * correct alignment when allocated.
36.     */
37.    if (flags & CFLGS_OFF_SLAB) {
38.        /* 外置式 slab */
39.        mgmt_size = 0;
40.        /* slab 页面不含 slab 管理对象，全部用来存储 slab 对象 */
41.        nr_objs = slab_size / buffer_size;
42.        /* 对象数不能超过上限 */
43.        if (nr_objs > SLAB_LIMIT)
44.            nr_objs = SLAB_LIMIT;
45.    } else {
```

```
46.      /*
47.       * Ignore padding for the initial guess. The padding
48.       * is at most @align-1 bytes, and @buffer_size is at
49.       * least @align. In the worst case, this result will
50.       * be one greater than the number of objects that fit
51.       * into the memory allocation when taking the padding
52.       * into account.
53.       *//* 内置式 slab，slab 管理对象与 slab 对象在一起
54.        ，此时 slab 页面中包含：一个 struct slab 对象，一个 kmem_bufctl_t 数组，slab 对象。
55.       kmem_bufctl_t 数组大小与 slab 对象数目相同 */
56.      nr_objs = (slab_size - sizeof(struct slab)) /
57.          (buffer_size + sizeof(kmem_bufctl_t));
58.
59.      /*
60.       * This calculated number will be either the right
61.       * amount, or one greater than what we want.
62.       *//* 计算 cache line 对齐后的大小，如果超出了 slab 总的大小，则对象数减一 */
63.      if (slab_mgmt_size(nr_objs, align) + nr_objs*buffer_size
64.            > slab_size)
65.        nr_objs--;
66.
67.      if (nr_objs > SLAB_LIMIT)
68.        nr_objs = SLAB_LIMIT;
69.      /* 计算 cache line 对齐后 slab 管理对象的大小 */
70.      mgmt_size = slab_mgmt_size(nr_objs, align);
71.    }
72.    *num = nr_objs;/* 保存 slab 对象数目 */
73.    /* 计算浪费空间的大小 */
74.    *left_over = slab_size - nr_objs*buffer_size - mgmt_size;
75.}
```

辅助数据结构与变量

Linux 内核中将所有的通用 cache 以不同的大小存放在数组中，以方便查找。其中 malloc_sizes[]数组为 cache_sizes 类型的数组，存放各个 cache 的大小；cache_names[]数组为 cache_names 结构类型数组，存放各个 cache 大小的名称；malloc_sizes[]数组和 cache_names[]数组下标对应，也就是说 cache_names[i]名称的 cache 对应的大小为 malloc_sizes[i]。

[cpp] view plaincopyprint?

```
1.  /* Size description struct for general caches. */
2.  struct cache_sizes {
3.      size_t          cs_size;
```

```
4.     struct kmem_cache   *cs_cachep;
5.  #ifdef CONFIG_ZONE_DMA
6.     struct kmem_cache   *cs_dmacachep;
7.  #endif
8.  };
9.  /*
10. * These are the default caches for kmalloc. Custom caches can have other sizes.
11. */
12.struct cache_sizes malloc_sizes[] = {
13.#define CACHE(x) { .cs_size = (x) },
14.#include <linux/kmalloc_sizes.h>
15.    CACHE(ULONG_MAX)
16.#undef CACHE
17.};
```

[cpp] view plaincopyprint?

```
1.  /* Must match cache_sizes above. Out of line to keep cache footprint low. */
2.  struct cache_names {
3.     char *name;
4.     char *name_dma;
5.  };
6.
7.  static struct cache_names __initdata cache_names[] = {
8.  #define CACHE(x) { .name = "size-" #x, .name_dma = "size-" #x "(DMA)" },
9.  #include <linux/kmalloc_sizes.h>
10.    {NULL,}
11.#undef CACHE
12.};
```

[cpp] view plaincopyprint?

```
1.  #define INDEX_AC index_of(sizeof(struct arraycache_init))
2.  #define INDEX_L3 index_of(sizeof(struct kmem_list3))
```

从上面的初始化过程中我们看到，创建的 cache 与用途主要有：

1，cache_cache 用于 cache 管理结构空间申请，对象大小为 cache 管理结构大小；

2，sizes[INDEX_AC].cs_cachep 用于 local cache；

3，sizes[INDEX_L3].cs_cachep 用于三链；

4，其他的主要用于指定大小的通用数据 cache。

二、内核启动末期初始化

1，根据对象大小计算 local cache 中对象数目上限；

2，借助数据结构 ccupdate_struct 操作 cpu 本地 cache。为每个在线 cpu 分配 cpu 本地 cache；

3，用新分配的 cpu 本地 cache 替换原有的 cache；

4，更新 slab 三链以及 cpu 本地共享 cache。

第二阶段代码分析

Start_kernel()->kmem_cache_init_late()

[cpp] view plaincopyprint?

```
1. /*Slab 系统初始化分两个部分，先初始化一些基本的，待系统初始化工作进行的差不多时，再配置一些特殊功能。*/
2. void __init kmem_cache_init_late(void)
3. {
4.     struct kmem_cache *cachep;
5.     /* 初始化阶段 local cache 的大小是固定的，要根据对象大小重新计算 */
6.     /* 6) resize the head arrays to their final sizes */
7.     mutex_lock(&cache_chain_mutex);
8.     list_for_each_entry(cachep, &cache_chain, next)
9.         if (enable_cpucache(cachep, GFP_NOWAIT))
10.            BUG();
11.    mutex_unlock(&cache_chain_mutex);
12.
13.    /* Done! */
14.    /* 大功告成，general cache 终于全部建立起来了 */
15.    g_cpucache_up = FULL;
16.
17.    /* Annotate slab for lockdep -- annotate the malloc caches */
18.    init_lock_keys();
19.
20.    /*
21.     * Register a cpu startup notifier callback that initializes
22.     * cpu_cache_get for all new cpus
23.     */
24.     /* 注册 cpu up 回调函数，cpu up 时配置 local cache */
25.    register_cpu_notifier(&cpucache_notifier);
26.
27.    /*
28.     * The reap timers are started later, with a module init call: That part
29.     * of the kernel is not yet operational.
30.     */
31.}
```

[cpp] view plaincopyprint?

```
1. /* Called with cache_chain_mutex held always */
2. /*local cache 初始化*/
3. static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp)
4. {
5.     int err;
6.     int limit, shared;
7. 
8.     /*
9.      * The head array serves three purposes:
10.     * - create a LIFO ordering, i.e. return objects that are cache-warm
11.     * - reduce the number of spinlock operations.
12.     * - reduce the number of linked list operations on the slab and
13.     *   bufctl chains: array operations are cheaper.
14.     * The numbers are guessed, we should auto-tune as described by
15.     * Bonwick.
16.     */ /* 根据对象大小计算 local cache 中对象数目上限 */
17.    if (cachep->buffer_size > 131072)
18.        limit = 1;
19.    else if (cachep->buffer_size > PAGE_SIZE)
20.        limit = 8;
21.    else if (cachep->buffer_size > 1024)
22.        limit = 24;
23.    else if (cachep->buffer_size > 256)
24.        limit = 54;
25.    else
26.        limit = 120;
27. 
28.    /*
29.     * CPU bound tasks (e.g. network routing) can exhibit cpu bound
30.     * allocation behaviour: Most allocs on one cpu, most free operations
31.     * on another cpu. For these cases, an efficient object passing between
32.     * cpus is necessary. This is provided by a shared array. The array
33.     * replaces Bonwick's magazine layer.
34.     * On uniprocessor, it's functionally equivalent (but less efficient)
35.     * to a larger limit. Thus disabled by default.
36.     */
37.    shared = 0;
38.    /* 多核系统，设置 shared local cache 中对象数目 */
39.    if (cachep->buffer_size <= PAGE_SIZE && num_possible_cpus() > 1)
40.        shared = 8;
41. 
42.#if DEBUG
```

```
43.    /*
44.     * With debugging enabled, large batchcount lead to excessively long
45.     * periods with disabled local interrupts. Limit the batchcount
46.     */
47.    if (limit > 32)
48.        limit = 32;
49.#endif
50.    /* 配置 local cache */
51.    err = do_tune_cpucache(cachep, limit, (limit + 1) / 2, shared, gfp);
52.    if (err)
53.        printk(KERN_ERR "enable_cpucache failed for %s, error %d.\n",
54.               cachep->name, -err);
55.    return err;
56.}
```

[cpp] view plaincopyprint?

```
1.  /* Always called with the cache_chain_mutex held */
2.  /*配置 local cache、shared local cache 和 slab 三链*/
3.  static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
4.                 int batchcount, int shared, gfp_t gfp)
5.  {
6.      struct ccupdate_struct *new;
7.      int i;
8.
9.      new = kzalloc(sizeof(*new), gfp);
10.     if (!new)
11.         return -ENOMEM;
12.     /* 为每个 cpu 分配新的 struct array_cache 对象 */
13.     for_each_online_cpu(i) {
14.         new->new[i] = alloc_arraycache(cpu_to_node(i), limit,
15.                     batchcount, gfp);
16.         if (!new->new[i]) {
17.             for (i--; i >= 0; i--)
18.                 kfree(new->new[i]);
19.             kfree(new);
20.             return -ENOMEM;
21.         }
22.     }
23.     new->cachep = cachep;
24.     /* 用新的 struct array_cache 对象替换旧的 struct array_cache 对象
25.      ，在支持 cpu 热插拔的系统上，离线 cpu 可能没有释放 local cache
26.      ，使用的仍是旧 local cache，参见__kmem_cache_destroy 函数
27.      。虽然 cpu up 时要重新配置 local cache，也无济于事。考虑下面的情景
```

```
28.  ：共有 Cpu A 和 Cpu B，Cpu B down 后，destroy Cache X，由于此时 Cpu B 是 down 状
     态
29.  ，所以 Cache X 中 Cpu B 的 local cache 未释放，过一段时间 Cpu B 又 up 了
30.  ，更新 cache_chain 链中所有 cache 的 local cache，但此时 Cache X 对象已经释放回
31.  cache_cache 中了，其 Cpu B local cache 并未被更新。又过了一段时间
32.  ，系统需要创建新的 cache，将 Cache X 对象分配出去，其 Cpu B 仍然是旧的
33.  local cache，需要进行更新。
34.  */
35.  on_each_cpu(do_ccupdate_local, (void *)new, 1);
36.
37.  check_irq_on();
38.  cachep->batchcount = batchcount;
39.  cachep->limit = limit;
40.  cachep->shared = shared;
41.  /* 释放旧的 local cache */
42.  for_each_online_cpu(i) {
43.      struct array_cache *ccold = new->new[i];
44.      if (!ccold)
45.          continue;
46.      spin_lock_irq(&cachep->nodelists[cpu_to_node(i)]->list_lock);
47.      /* 释放旧 local cache 中的对象 */
48.      free_block(cachep, ccold->entry, ccold->avail, cpu_to_node(i));
49.      spin_unlock_irq(&cachep->nodelists[cpu_to_node(i)]->list_lock);
50.      /* 释放旧的 struct array_cache 对象 */
51.      kfree(ccold);
52.  }
53.  kfree(new);
54.  /* 初始化 shared local cache 和 slab 三链 */
55.  return alloc_kmemlist(cachep, gfp);
56.}
```

更新本地 cache

[cpp] view plaincopyprint?

```
1.  /*更新每个 cpu 的 struct array_cache 对象*/
2.  static void do_ccupdate_local(void *info)
3.  {
4.      struct ccupdate_struct *new = info;
5.      struct array_cache *old;
6.
7.      check_irq_off();
8.      old = cpu_cache_get(new->cachep);
9.       /* 指向新的 struct array_cache 对象 */
10.     new->cachep->array[smp_processor_id()] = new->new[smp_processor_id()];
```

```
11.      /* 保存旧的 struct array_cache 对象 */
12.    new->new[smp_processor_id()] = old;
13.}
```

[cpp] view plaincopyprint?

```
1. /*初始化 shared local cache 和 slab 三链，初始化完成后，slab 三链中没有任何 slab*/
2. static int alloc_kmemlist(struct kmem_cache *cachep, gfp_t gfp)
3. {
4.     int node;
5.     struct kmem_list3 *l3;
6.     struct array_cache *new_shared;
7.     struct array_cache **new_alien = NULL;
8.
9.     for_each_online_node(node) {
10.       /* NUMA 相关 */
11.            if (use_alien_caches) {
12.                  new_alien = alloc_alien_cache(node, cachep->limit, gfp);
13.                  if (!new_alien)
14.                        goto fail;
15.            }
16.
17.       new_shared = NULL;
18.       if (cachep->shared) {
19.          /* 分配 shared local cache */
20.          new_shared = alloc_arraycache(node,
21.             cachep->shared*cachep->batchcount,
22.                0xbaadf00d, gfp);
23.          if (!new_shared) {
24.             free_alien_cache(new_alien);
25.             goto fail;
26.          }
27.       }
28.       /* 获得旧的 slab 三链 */
29.       l3 = cachep->nodelists[node];
30.       if (l3) {
31.          /* 就 slab 三链指针不为空，需要先释放旧的资源 */
32.          struct array_cache *shared = l3->shared;
33.
34.          spin_lock_irq(&l3->list_lock);
35.          /* 释放旧的 shared local cache 中的对象 */
36.          if (shared)
37.             free_block(cachep, shared->entry,
38.                   shared->avail, node);
```

```
39.         /* 指向新的 shared local cache */
40.         l3->shared = new_shared;
41.         if (!l3->alien) {
42.             l3->alien = new_alien;
43.             new_alien = NULL;
44.         }/* 计算 cache 中空闲对象的上限 */
45.         l3->free_limit = (1 + nr_cpus_node(node)) *
46.                 cachep->batchcount + cachep->num;
47.         spin_unlock_irq(&l3->list_lock);
48.         /* 释放旧 shared local cache 的 struct array_cache 对象 */
49.         kfree(shared);
50.         free_alien_cache(new_alien);
51.         continue;/*访问下一个节点*/
52.     }
53.     /* 如果没有旧的 l3，分配新的 slab 三链 */
54.     l3 = kmalloc_node(sizeof(struct kmem_list3), gfp, node);
55.     if (!l3) {
56.         free_alien_cache(new_alien);
57.         kfree(new_shared);
58.         goto fail;
59.     }
60.     /* 初始化 slab 三链 */
61.     kmem_list3_init(l3);
62.     l3->next_reap = jiffies + REAPTIMEOUT_LIST3 +
63.         ((unsigned long)cachep) % REAPTIMEOUT_LIST3;
64.     l3->shared = new_shared;
65.     l3->alien = new_alien;
66.     l3->free_limit = (1 + nr_cpus_node(node)) *
67.             cachep->batchcount + cachep->num;
68.     cachep->nodelists[node] = l3;
69. }
70. return 0;
71.
72.fail:
73. if (!cachep->next.next) {
74.     /* Cache is not active yet. Roll back what we did */
75.     node--;
76.     while (node >= 0) {
77.         if (cachep->nodelists[node]) {
78.             l3 = cachep->nodelists[node];
79.
80.             kfree(l3->shared);
81.             free_alien_cache(l3->alien);
```

```
82.             kfree(l3);
83.             cachep->nodelists[node] = NULL;
84.         }
85.         node--;
86.     }
87. }
88. return -ENOMEM;
89.}
```

看一个辅助函数

[cpp] view plaincopyprint?

```
1. /*分配 struct array_cache 对象。*/
2. static struct array_cache *alloc_arraycache(int node, int entries,
3.                     int batchcount, gfp_t gfp)
4. {
5.     /* struct array_cache 后面紧接着的是 entry 数组，合在一起申请内存 */
6.     int memsize = sizeof(void *) * entries + sizeof(struct array_cache);
7.     struct array_cache *nc = NULL;
8.     /* 分配一个 local cache 对象，kmalloc 从 general cache 中分配 */
9.     nc = kmalloc_node(memsize, gfp, node);
10.    /*
11.     * The array_cache structures contain pointers to free object.
12.     * However, when such objects are allocated or transfered to another
13.     * cache the pointers are not cleared and they could be counted as
14.     * valid references during a kmemleak scan. Therefore, kmemleak must
15.     * not scan such objects.
16.     */
17.    kmemleak_no_scan(nc);
18.    /* 初始化 local cache */
19.    if (nc) {
20.        nc->avail = 0;
21.        nc->limit = entries;
22.        nc->batchcount = batchcount;
23.        nc->touched = 0;
24.        spin_lock_init(&nc->lock);
25.    }
26.    return nc;
27.}
```

源代码中涉及了 slab 的分配、释放等操作在后面分析中陆续总结。slab 相关数据结构、工作机制以及整体框架在分析完了 slab 的创建、释放工作后再做总结，这样可能会对 slab 机制有更好的了解。当然，从代码中看运行机制会更有说服了，

也是一种习惯。