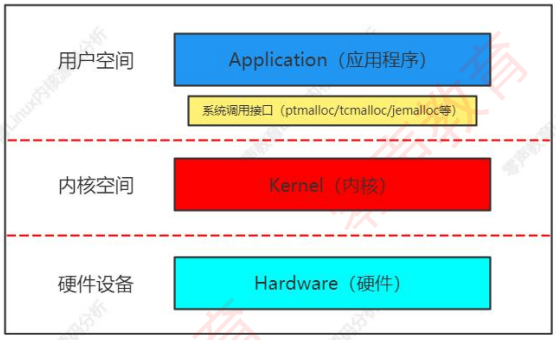
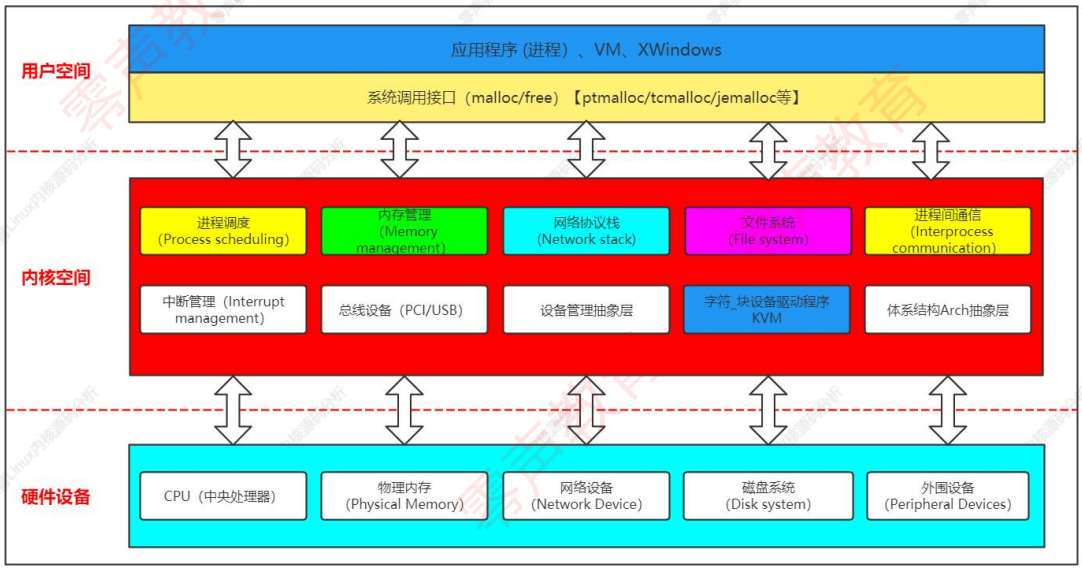
第 002 讲 Linux 内核《物理与虚拟内存管理》

Linux 内核源码分析架构图





一、虚拟地址空间布局及内存映射

1、用户空间

应用程序使用 malloc()申请内存，使用 free()释放内存，malloc()/free()是 glibc 库的内存分配器 ptmalloc 提供的接口，ptmalloc 使用系统调用 brk/mmap 向内核以页为单位申请内存，然后划分成小内存块分配给用户应用程序。用户空间的内存分配器，除 glibc 库的 ptmalloc，google 的 tcmalloc/FreeBSD 的 jemalloc。

2、内核空间

内核空间的基本功能：虚拟内存管理负责从进程的虚拟地址空间分配虚拟页，sys_brk 用来扩大或收缩堆，sys_mmap 用来在内存映射区域分配虚拟页，sys_munmap 用来释放虚拟页。

页分配器负责分配物理页，当前使用的页分配器是伙伴分配器。内核空间提供把页划分成小内存块分配的块分配器，提供分配内存的接口 kmalloc()和释放内存接口 kfree()。块分配器：SLAB/SLUB/SLOB。

内核空间的扩展功能：不连续页分配器提供了分配内存的接口 vmalloc 和释放内存接口 vfree，在内存碎片化时，申请连续物理页的成功率很低，可申请不连续的物理页，映射到连续的虚拟页，即虚拟地址连续页物理地址不连续。

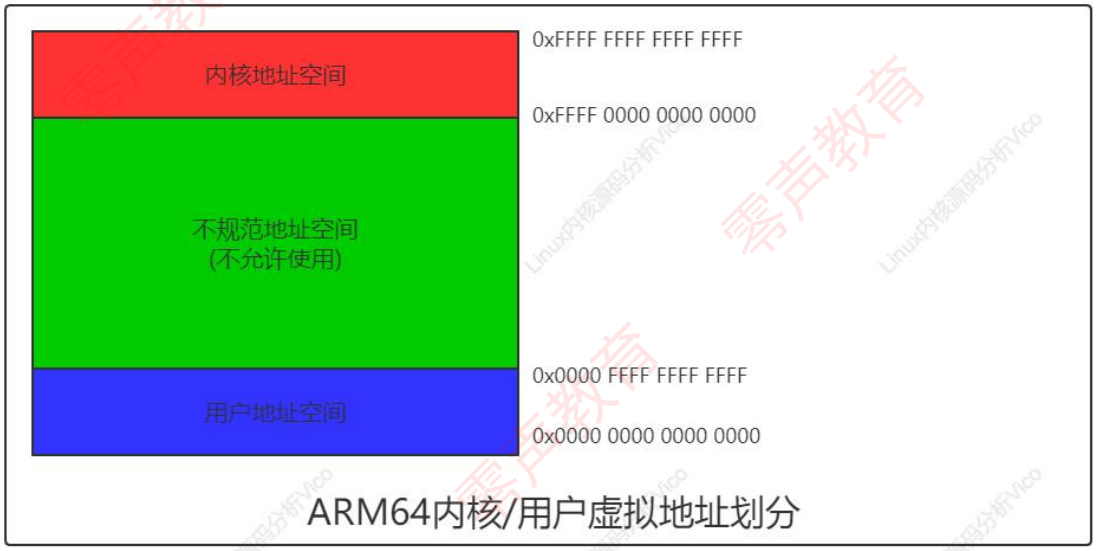
连续内存分配器（contiguous memory allocator，CMA）用来给驱动程序预留一段连续的内存，当驱动程序不用的时候，可以给进程使用；当驱动程序需要使用的时候，把进程占用的内存通过回收或迁移的方式让出来，给驱动程序使用。

3、硬件层面

处理器包含一个称为内存管理单元（Memory Management Unit，MMU）的部件，负责把虚拟地址转换成物理地址。内存管理单元包含一个称为页表缓存（Translation Lookaside Buffer，TLB）的部件，保存最近使用的页表映射，避免每次把虚拟地址转换物理地址都需要查询内存中的页表。

4、虚拟地址空间布局

a、虚拟地址空间划分



以 ARM64 处理器为例：虚拟地址 的最大宽度是 48 位。内核虚拟地址在 64 位地址空间顶部，高 16 位全是 1，范围是[0xFFFF 0000 0000 0000,0xFFFF FFFF FFFF FFFF] 。用户虚拟地址 在 64 位地址 空间的底部，高 16 位全是 0，范围是[0x0000 0000 0000 0000,0x0000 FFFF FFFF FFFF]。

在编译 ARM64 架构的 Linux 内核时，可以选择虚拟地址宽度：

- 选择页长度 4KB，默认虚拟地址宽度为 39 位；
- 选择页长度 16KB，默认虚拟地址宽度为 47 位；
- 选择页长度 64KB，默认虚拟地址宽度为 42 位；
- 选择 48 位虚拟地址。

在 ARM64 架构 linux 内核中，内核虚拟地址用户虚拟地址宽度相同。所有进程共享内核虚拟地址空间，每个进程有独立的用户虚拟地址 空间，同一个线程组的用户线程共享用户虚拟地址空间，内核线程没有用户虚拟地址 空间。

b、用户虚拟地址空间布局

进程的用户虚拟地址空间的起始地址是 0，长度是 TASK_SIZE，由每种处理器架构定义自己的宏 TASK_SIZE。ARM64 架构定义宏 TASK_SIZE 如下所示：

- 32 位用户空间程序：TASK_SIZE 的值是 TASK_SIZE_32，即 0x10000000，等于 4GB。

- 64 位用户空间程序：TASK_SIZE 的值是 TASK_SIZE_64，即 2 的 VA_BITS 次方字节，VA_BITS 是编译内核时选择的虚拟地址位数。

arch > arm64 > include > asm > C memory.h > ...

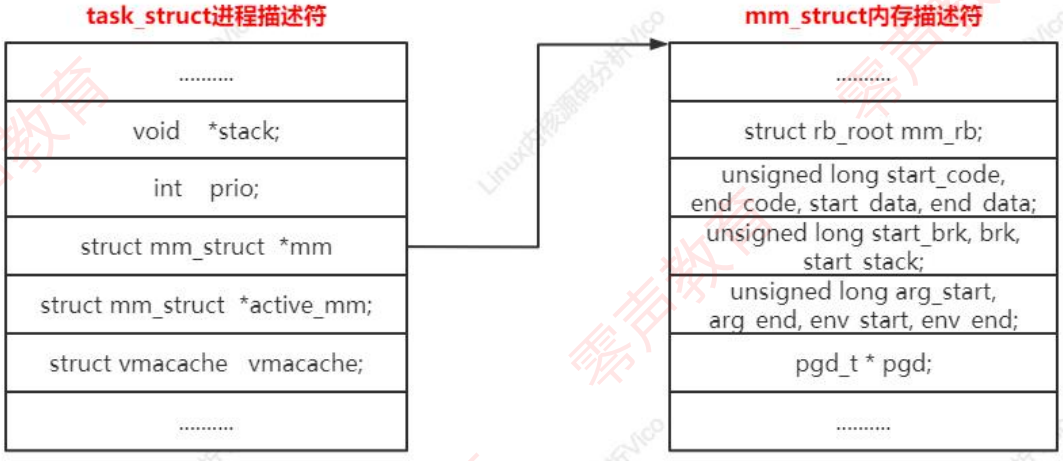
```
68  #define VA_BITS         (CONFIG_ARM64_VA_BITS)
69  #define VA_START        (UL(0xffffffffffffffff) << VA_BITS)
70  #define PAGE_OFFSET     (UL(0xffffffffffffffff) << (VA_BITS - 1))
71  #define KIMAGE_VADDR        (MODULES_END)
72  #define MODULES_END     (MODULES_VADDR + MODULES_VSIZE)
73  #define MODULES_VADDR       (VA_START + KASAN_SHADOW_SIZE)
74  #define MODULES_VSIZE       (SZ_128M)
75  #define VMEMMAP_START       (PAGE_OFFSET - VMEMMAP_SIZE)
76  #define PCI_IO_END      (VMEMMAP_START - SZ_2M)
77  #define PCI_IO_START        (PCI_IO_END - PCI_IO_SIZE)
78  #define FIXADDR_TOP     (PCI_IO_START - SZ_2M)
79  #define TASK_SIZE_64        (UL(1) << VA_BITS)
80
81  #ifdef CONFIG_COMPAT
82  #define TASK_SIZE_32        UL(0x100000000)
83  #define TASK_SIZE       (test_thread_flag(TIF_32BIT) ? \
84                  TASK_SIZE_32 : TASK_SIZE_64)
85  #define TASK_SIZE_OF(tsk)   (test_tsk_thread_flag(tsk, TIF_32BIT) ? \
86                  TASK_SIZE_32 : TASK_SIZE_64)
87  #else
88  #define TASK_SIZE       TASK_SIZE_64
89  #endif /* CONFIG_COMPAT */
```

Linux 内核使用内存描述符 mm_struct 描述进程的用户虚拟地址空间，主要核心成员如下：

include > linux > C mm_types.h > kioctx_table

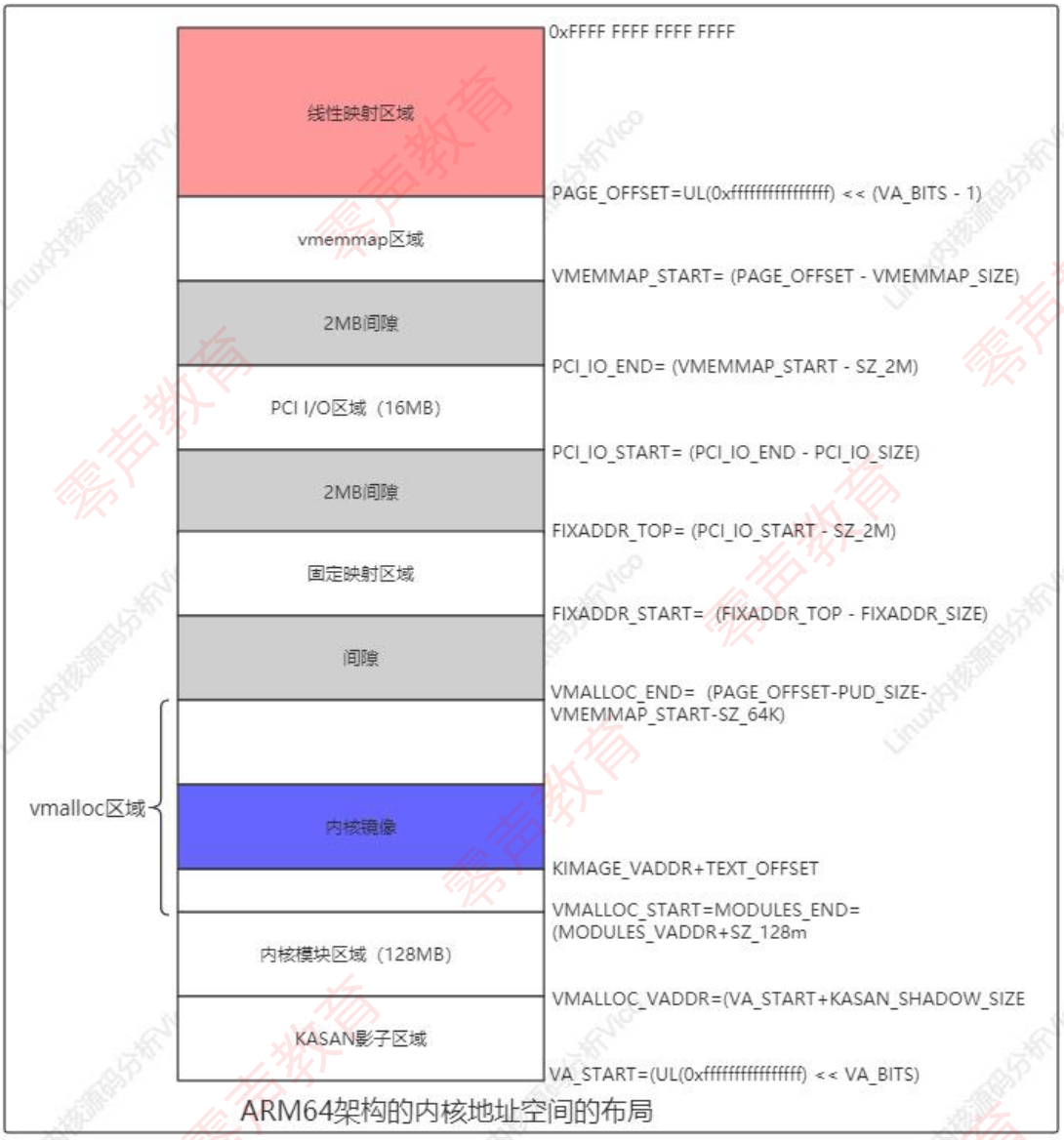
```
361
362 struct mm_struct {
363     struct vm_area_struct *mmap;        /* list of VMAs */
364     struct rb_root mm_rb;
365     u32 vmacache_seqnum;                   /* per-thread vmacache */
```

c、进程的进程描述和内存描述符关系如下图所示：



c、内核地址空间布局

ARM64 处理器架构的内核地址空间布局如下：



ARM64架构的内核地址空间的布局

二、伙伴分配器及算法

内核初始化完毕后，使用页分配器管理物理页，当前使用的页分配器就是伙伴分配器，伙伴分配器的特点是管理算法简单且高效。

1、基本伙伴分配器

连续的物理页称为页块（page block），阶（order）是页的数量单位，2 的 n 次方个连续页称为 n 阶页块，满足如下条件的两个 n 阶页块称为伙伴（buddy）。

- 1）两个页块是相邻的，即物理地址是连续的；
- 2）页块的第一页的物理面页号必须是 2 的 n 次方的整数倍；
- 3）如果合并（n+1）阶页块，第一页的物理页号必须是 2 的括号(n+1)次方的整数倍。

伙伴分配器分配和释放物理页的数量单位也为阶（order）。

以单页为说明，0 号页和 1 号页是伙伴，2 号页和 3 号页是伙伴。1 号页和 2 号页不是伙伴？因为 1 号页和 2 号页合并组成一阶页块，第一页的物理页号不是 2 的整数倍。

2、分区伙伴分配器

内存区域的结构体成员 free_area 用来维护空闲页块，数组下标对应页块的除数。结构体 free_area 的成员 free_list 是空闲页块的链表，nr_free 是空闲页块的数量。内存区域的结构体成员 managed_pages 是伙伴分配器管理的物理页的数量。

2.1）内存区域数据结构分析如下：

include > linux > C mmzone.h > zone

```
384  struct zone {  // 内存区域数据结构
385      /* Read-mostly fields */
386
387      /* zone watermarks, access with *_wmark_pages(zone) macros */
388      unsigned long _watermark[NR_WMARK]; // 页分配器使用的水线
389
390      unsigned long watermark_boost;
391
392      unsigned long nr_reserved_highatomic;
393
394      /*
395       * We don't know if the memory that we're going to allocate will be
396       * freeable or/and it will be released eventually, so to avoid totally
397       * wasting several GB of ram we must reserve some of the lower zone
398       * memory (otherwise we risk to run OOM on the lower zones despite
399       * there being tons of freeable ram on the higher zones).  This array is
400       * recalculated at runtime if the sysctl_lowmem_reserve_ratio sysctl
401       * changes.
402       */
403      long lowmem_reserve[MAX_NR_ZONES]; // 页分配器使用，当前区域保留多少页不能借给高的区域类型
404
```

include > linux > C mmzone.h > ...

```
109  struct free_area {
110      struct list_head    free_list[MIGRATE_TYPES];
111      unsigned long       nr_free;
112  };
113
```

2.2）区域水线数据结构分析

首选的内存区域在什么情况下从备用区域借用物理页？此问题从区域水线讲解深入理解，每个内存区域有 3 个水线。

- 高水线（HIGH）：如果内存区域的空闲页数大于高水线，说明该内存区域的内存充足；
- 低水线（LOW）：如果内存区域的空闲页数小于低水线，说明该内存区域的内存轻微不足；
- c.最低水线（MIN）：如果内存区域空闲页数小于最低水线，说明该内存区域的内存严重不足。

include > linux > C mmzone.h > zone_watermarks

```
278  enum zone_watermarks {
279      WMARK_MIN,
280      WMARK_LOW,
281      WMARK_HIGH,
282      NR_WMARK
283  };
284
```

三、slab/slub/slob 块分配器

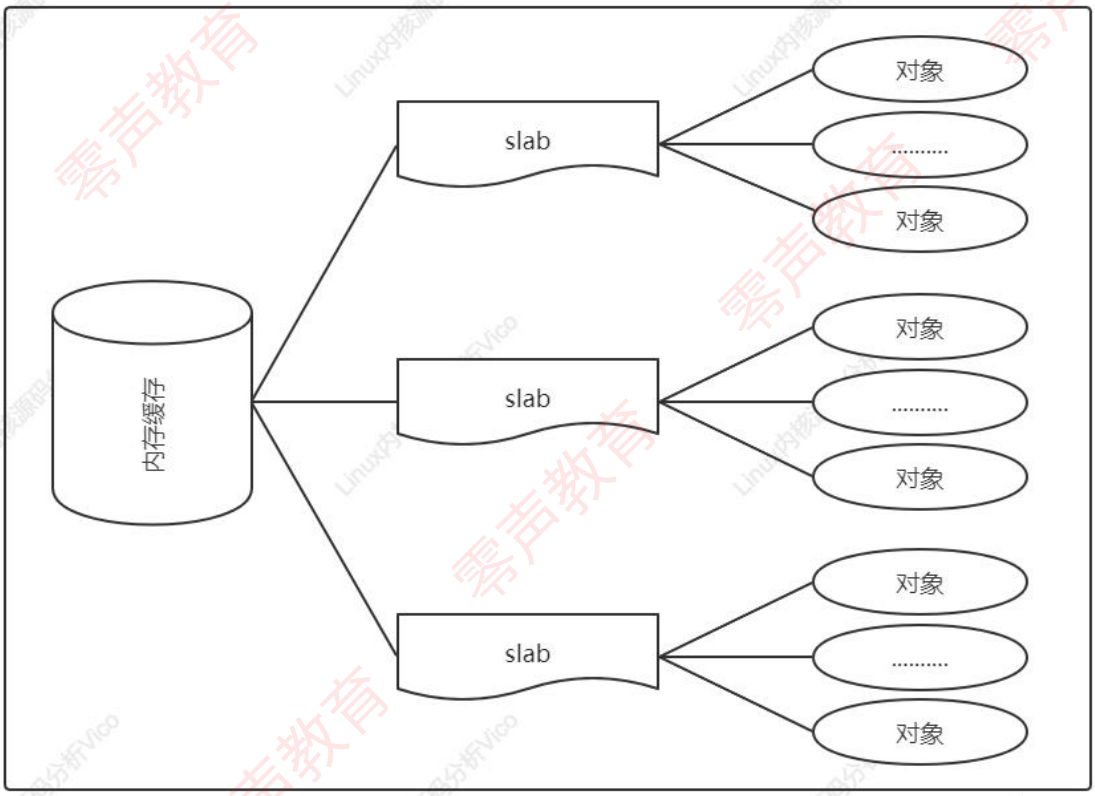
1、基本概念

Buddy 提供以 page 为单位的内存分配接口，这对内核来说颗粒度还太大，所以需要一种新的机制，将 page 拆分为更小的单位来管理。

Linux 中支持的主要有：slab、slub、slob。其中 slob 分配器的总代码量比较少，但分配速度不是最高效的，所以不是为大型系统设计，适合内存紧张的嵌入式系统。

2、slab 块分配器原理

slab 分配器的作用不仅仅是分配小块内存，更重要的作用是针对经常分配和释放的对象充当缓存。slab 分配器的核心思路是：为每种对象类型创建一个内存缓存，每个内存缓存由多个大块组成，一个大块是由一个或多个连续的物理页，每个大块包含多个对象。slab 采用面向对象的思想，基于对象类型管理内存，每种对象被划分为一类，比如进程描述符 task_struct 是一个类，每个进程描述符实例是一个对象。如下图所示为内存缓存的组成结构：



slab 分配器在某些情况下表现不太优先，所以 Linux 内核提供两个改进的块分配器。

- 在配备大量物理内存的大型计算机上，slab 分配器的管理数据结构的内存开销比较大，所以设计了 slub 分配器；
- 在小内存的嵌入式设备上，slab 分配器的代码过多、相当复杂，所以设计一个精简 slob 分配器。

目前 slub 分配器已成为默认的块分配器。

3、系统编程接口

通用的内存缓存的编程接口如下：

- a. 分配内存 kmalloc；
- kmalloc（size_t size,gfp_t flags）
- b. 重新分配内存 krealloc；
- krealloc(const void *p,size_t new_size,gpf_t flags)
- c. 释放内存 kfree；
- kfree(const void *objp)

创建专用的内存缓存编程接口如下：

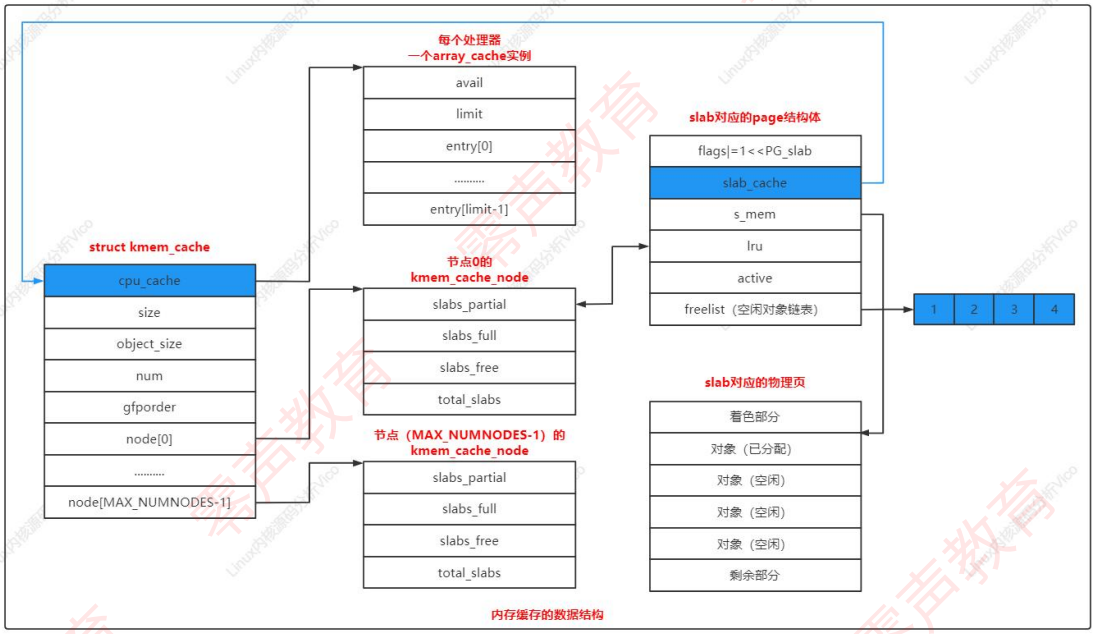
- a. 创建内存缓存 kmem_cache_create
- b. 指定内存缓存分配 kmem_cache_alloc
- c. 释放对象 kmem_cache_free
- d. 销毁内存缓存 keme_cache_destroy

4、内存缓存的数据结构

include > linux > C slub_def.h

```
82
83  /*
84   * Slab cache management.
85   */
86  struct kmem_cache {
87      struct kmem_cache_cpu __percpu *cpu_slab;
88      /* Used for retriving partial slabs etc */
89      slab_flags_t flags;
90      unsigned long min_partial;
91      unsigned int size;  /* The size of an object including meta data */
92      unsigned int object_size;/* The size of an object without meta data */
93      unsigned int offset;    /* Free pointer offset. */
94  #ifdef CONFIG_SLUB_CPU_PARTIAL
95      /* Number of per cpu partial objects to keep around */
96      unsigned int cpu_partial;
97  #endif
```

内存缓存的数据结构如下图所示：



5、计算 slab 长度及着色

a. 计算 slab

函数 calculate_slab_order 负责计算 slab 长度，从 0 阶到 kmalloc()函数支持最大除数 KMALLOC_MAX_ORDER。

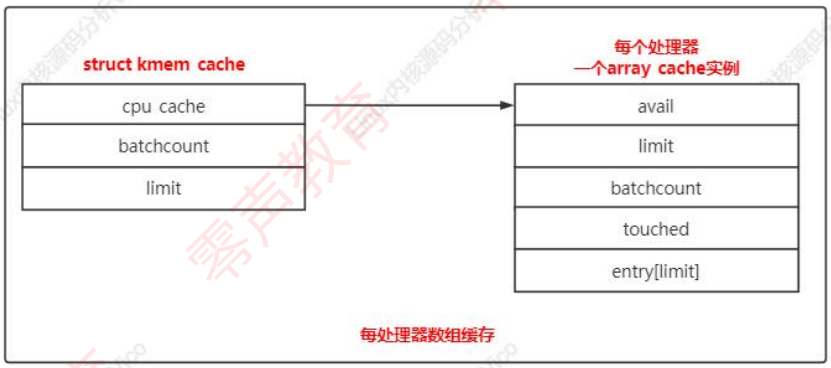
```
mm > C slab.c
1735  static size_t calculate_slab_order(struct kmem_cache *cachep,
1736                          size_t size, slab_flags_t flags)
1737  {
1738      size_t left_over = 0;
1739      int gfporder;
1740
1741      for (gfporder = 0; gfporder <= KMALLOC_MAX_ORDER; gfporder++) {
1742          unsigned int num;
1743          size_t remainder;
1744
1745          num = cache_estimate(gfporder, size, flags, &remainder);
1746          if (!num)
1747              continue;
1748
```

b. 着色

slab 是一个或多个连续的物理页，起始地址总是页长度的整数倍，不同 slab 中相同偏移的位置在处理器一级缓存中的索引相同。如果 slab 的剩余部分的长度超过一级缓存行的长度，剩余部分对应的一级缓存行没有被利用；如果对象的填充字节的长度超过一级缓存行的长度，填充字节对应的一级缓存行没有被利用。这两种情况导致处理器的某些缓存行被过度使用，另一些缓存行很少使用。

6、每处理器数组缓存

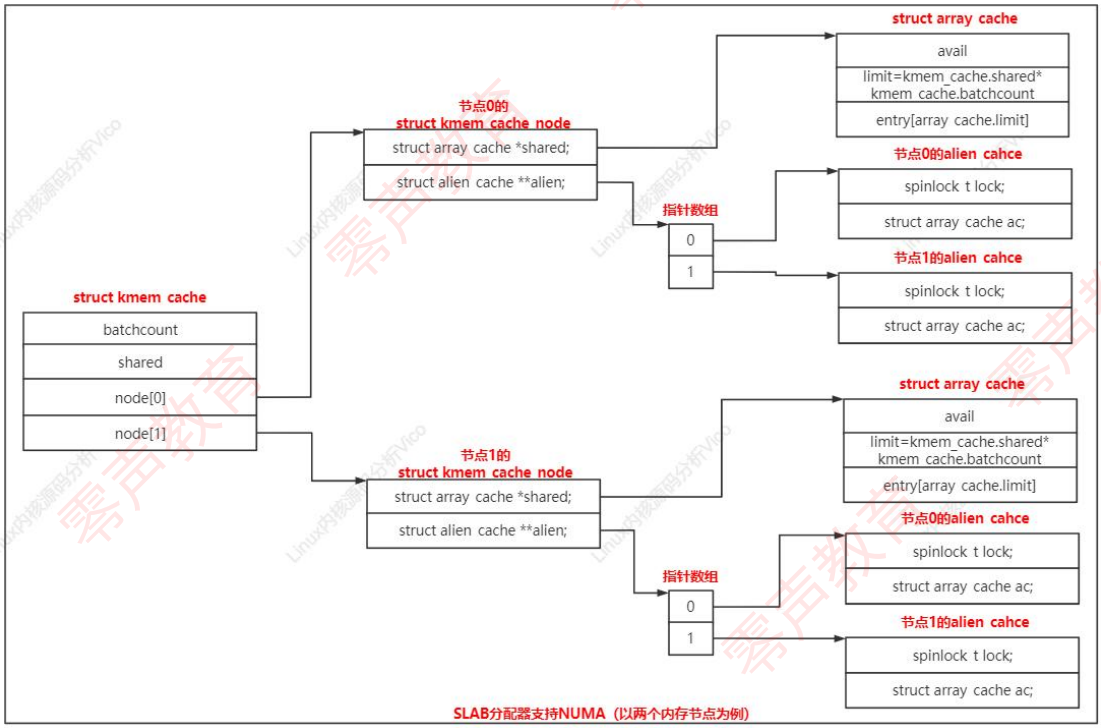
内存缓存为每个处理器创建一个数组缓存（结构体 array_cahce）。释放对象时，把对象存放到当前处理器对应的数组缓存中；分配对象的时候，先从当前处理器的数组缓存分配对象，采用后进先出（Last In First Out，LIFO）的原则，这种做可以提高性能。




```
include > linux > C slab_def.h > kmem_cache > cpu_cache
5
6   /*
7    * Definitions unique to the original Linux SLAB allocator.
8    */
9   struct kmem_cache {
10      struct array_cache __percpu *cpu_cache;
11
12  /* 1) Cache tunables. Protected by slab_mutex */
13      unsigned int batchcount;
14      unsigned int limit;
15      unsigned int shared;
16
17      unsigned int size;
18      struct reciprocal_value reciprocal_buffer_size;
19  /* 2) touched by every alloc & free from the backend */
```

7、slab 分配器支持 NUMA 体系结构

内存缓存针对每个内存节点创建一个 kmem_cache_node 实例。

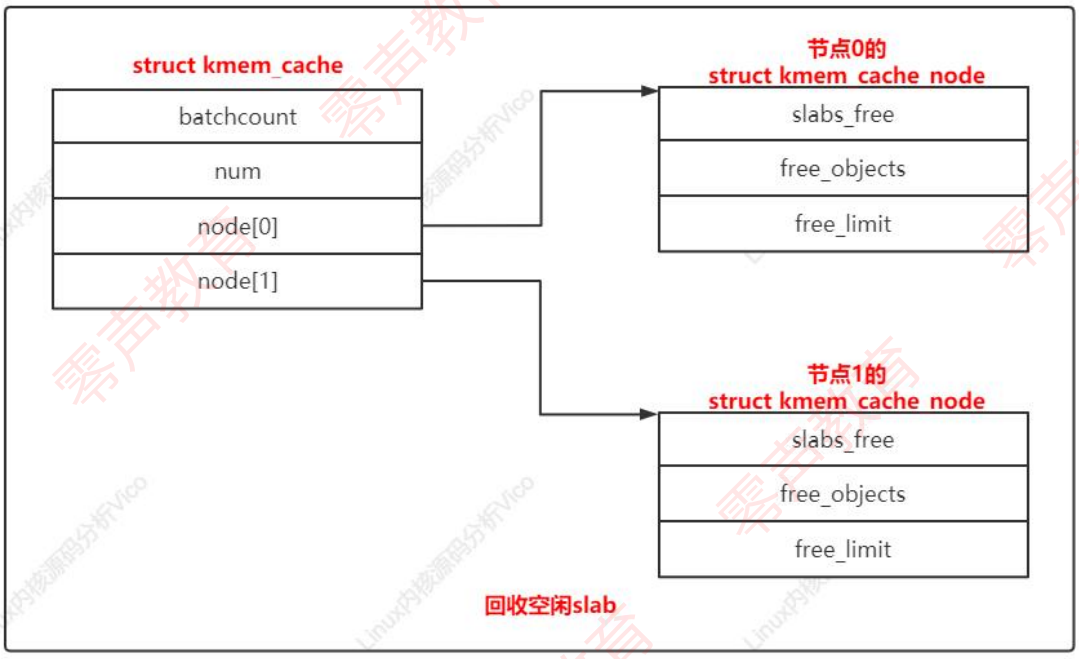


include > linux > C slab_def.h > kmem_cache > node

```
6  /*
7   * Definitions unique to the original Linux SLAB allocator.
8   */
9  struct kmem_cache {
10     struct array_cache __percpu *cpu_cache;
11
12 /* 1) Cache tunables. Protected by slab_mutex */
13     unsigned int batchcount;
14     unsigned int limit;
15     unsigned int shared;
16
17     unsigned int size;
18     struct reciprocal_value reciprocal_buffer_size;
19 /* 2) touched by every alloc & free from the backend */
```

8、回收内存

对于所有对象空闲的 slab，没有立即释放，而是放在空闲 slab 链表中。只有内存节点上空闲对象的数量超过限制，才开始回收空闲 slab，直到空闲对象的数量小于或等于限制。结构体 kmem_cache_node 的成员 slabs_free 是空闲 slab 链表的头节点，成员 free_objects 是空闲对象的数量，成员 free_limit 是空闲对象的数量限制。



节点 x 的空闲对象的数量限制=（1+节点的处理器数量）*keme_cache.batchcount+kmem_cache.num

四、vmalloc/vfree/slab 内核实现

当设备长时间运行后，内存碎片化，很难找到连续的物理页。在这种情况下，如果需要分配长度超过一页的内存块，可以使用不连续页分配器，分配虚拟地址连续但是物理地址不连续的内存块。在 32 位系统中不连分配器还有一个好处：优先从高端内存区域分配页，保留稀缺的低端内存区域。

1、系统编程接口

a. 不连续页分配器提供的编程接口：

- vmalloc:分配不连续的物理页并且把物理页映射到连续的虚拟地址空间
- vfree:释放 vmalloc 分配的物理页和虚拟地址空间
- vmap:把已经分配的不连续物理页映射到连续的虚拟地址空间
- vunmap:释放使用 vmap 分配的虚拟地址空间

b. 内核提供函数接口：

kvmalloc：首先尝试使用 kmalloc 分配内存块，如果失败，那么使用 vmalloc 函数分配不连续的物理页。

kvfree：如果内存块是使用 vmalloc 分配的，那么使用 vfree 释放，否则使用 kfree 释放。

2、数据结构

include > linux > C vmalloc.h > vm_struct > pages

```
31
32  struct vm_struct {
33      struct vm_struct    *next;
34      void                *addr;
35      unsigned long       size;
36      unsigned long       flags;
37      struct page         **pages;
38      unsigned int        nr_pages;
39      phys_addr_t         phys_addr;
40      const void          *caller;
41  };
42
43  struct vmap_area {
44      unsigned long va_start;
45      unsigned long va_end;
46      unsigned long flags;
47      struct rb_node rb_node;         /* address sorted rbtree */
48      struct list_head list;          /* address sorted list */
49      struct llist_node purge_list;    /* "lazy purge" list */
50      struct vm_struct *vm;
51      struct rcu_head rcu_head;
52  };
```

每个虚拟内存区域对应一个 vmap_area 实例；
每个 vmap_area 实例关联一个 vm_struct 实例；

vmalloc 函数执行过程：

- a. 分配虚拟内存区域
- b. 分配物理页
- c. 在内核的页表中把虚拟页映射到物理页

【Linux 内核技术常见面试题】：

1、为什么自旋锁的临界区不允许发生抢占？
2、自述 MCS 锁机制的实现原理？
3、PG_locked 常见使用方法？
4、softlockup 和 hardlockup ？
5、问 C++ lamda 怎么递包，怎么判断内存溢出？

1、RCU 实现的基本原理？
2、乐观自旋等待的判断条件是什么？
3、什么是中断现场？中断现场需要保存哪些内容？中断现场保存在什么地方？
4、软中断上下文包括哪些几种情况？
5、当发生硬件中断后，ARM64 处理器做哪些工作？