第 3 章 物理内存管理

物理内存是内核及用户程序运行的基础，因为处理器需要从内存中取指、取数据，并将运算结果保存至内存，内核、用户程序可执行目标文件需要加载到内存中，才能被处理器执行。

内核在运行时，需要动态地分配、释放内存，这些内存主要用于创建、释放内核数据结构实例。用户进程由内核按页、按需地为其分配物理内存。

物理内存管理主要是对系统中空闲物理内存的管理，用于为内核和用户进程动态地分配和释放内存。物理内存管理主要包括伙伴系统、slab/slob/slub 分配器，伙伴系统是物理内存管理的基础，它按页管理系统中的内存，按（多）页为内核、用户进程分配和释放内存。slab/slob/slub 分配器建立在伙伴系统基础之上，用于为内核动态分配和释放数据结构实例，它从伙伴系统中按页申请内存，然后划分成小块，分配给内核使用（注意不能分配给用户进程使用）。

另外，在内核启动初期，在伙伴系统和 slab/slob/slub 分配器尚不可用时，内核实现了一个自举分配器，用于在启动初期按页为内核分配（一般不释放）内存。在伙伴系统和 slab/slob/slub 分配器初始化完成后，将废弃自举分配器，转而使用伙伴系统。

本章首先简要介绍处理器访问物理内存的机制，概述内核物理内存管理框架及相关数据结构，然后介绍自举分配器的实现，物理内存管理数据结构的初始化，最后重点介绍伙伴系统和 slab 分配器的实现。

3.1 概述

本节简要介绍处理器访问物理内存的机制，内核物理内存管理框架及相关的初始化函数，为后面理解物理内存管理子系统打下基础。

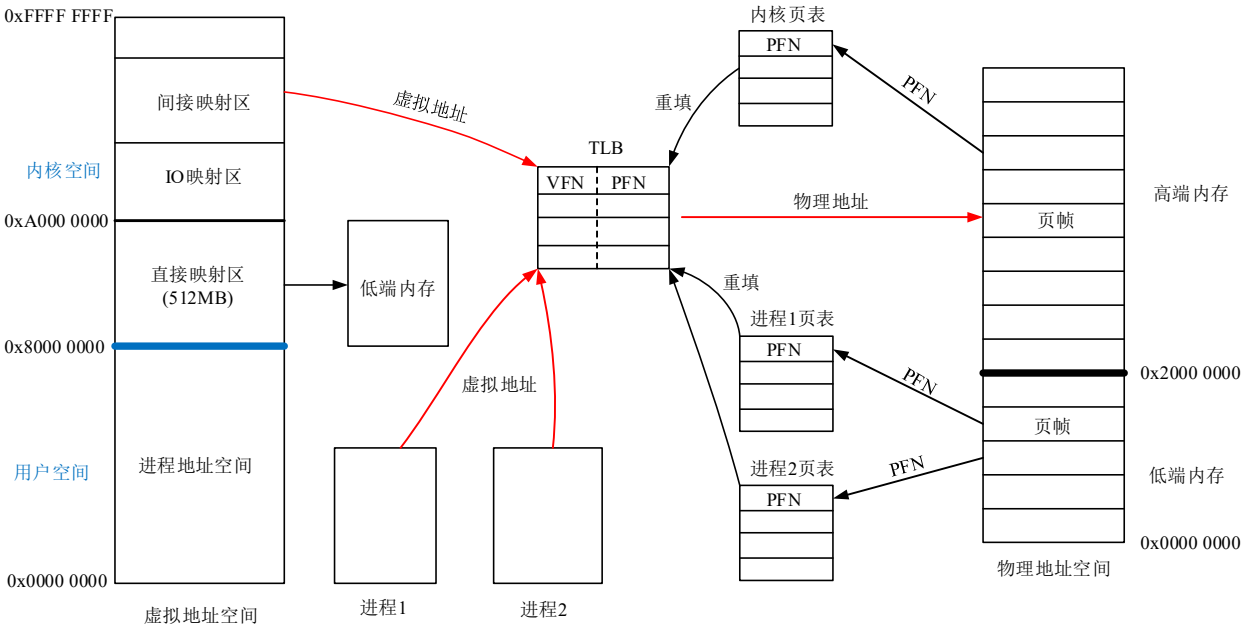
3.1.1 内存访问机制

MIPS32 处理器虚拟地址空间分为内核空间和用户空间（进程空间），如下图所示。内核空间位于上半部分（2GB），用户空间位于下半部分（2GB）。处理器处于内核态时可访问内核空间和用户空间，处于用户态时只能访问用户空间。内核被加载到内核地址空间，在处理器内核态下运行，用户程序加载到用户空间，在处理器用户态下运行。

内核地址空间又划分为直接映射区、IO 映射区、间接映射区等区域。直接映射区大小为 512MB，通过虚拟地址最高位清零直接将虚拟地址空间映射到物理内存低 512MB 空间，访问无需经过页表，地址转换由硬件完成，访问速度较快。物理内存低 512MB 空间称为低端内存，即可以直接映射到内核直接映射区的内存，而高于 512MB 空间的内存称为高端内存，内核需要通过页表映射间接访问。

内核空间中的 IO 映射区也是直接映射到物理内存低 512MB 空间，且不经过处理器缓存，表示的是外部设备寄存器空间。内核空间中的间接映射区，通过页表可映射到物理内存的任意位置。

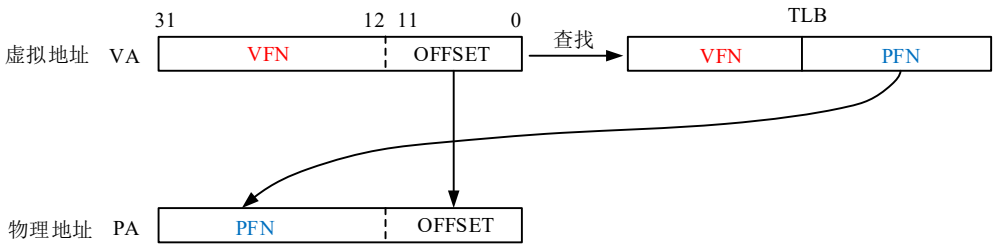
用户空间总是通过页表映射到物理内存，可映射到物理内存中任意位置。



内核按页对物理内存进行划分，通常一页为 4KB（或 8KB 等），物理内存页称为页帧（PF）。假设页帧大小为 4KB，内核保证每页的起始地址是 4KB 对齐的（低 12 位为 0），将页帧物理地址右移 12 位（高 30 位）所得整数称为页帧号（PFN，页帧编号）。物理内存是按页映射到虚拟内存的。

内核及每个进程具有一个页表，页表是一个保存页帧号的数组，数组的索引值是虚拟页帧号（VFN）。例如，假设进程的 0 号虚拟页映射到 2 号页帧，则页表第 0 个数组项保存页帧号 0x2（还有其它信息，详见下一章）。

处理器中包含一个被称为内存管理单元（MMU）的部件，该部件内主要包含一个 TLB 表，表中有若干个表项，表项的内容是 VFN 和 PFN，即指示虚拟页映射到哪个物理页。处理器通过虚拟地址访问内存时，先在 TLB 中查找与虚拟地址（VFN）匹配的项，若有匹配的项，则用表项中的 PFN 代替虚拟地址中的 VFN（地址低位不变），得到物理地址，访问物理内存，如下图所示。



如果 TLB 中没有与虚拟地址匹配的项，则从内核/进程页表中查找相应的表项，填入 TLB，然后再执行上面的地址转换。

由于所有进程的虚拟地址空间相同，所以在 TLB 是还需要标识表项适用于哪个进程的虚拟地址转换，相关的详细内容在下一章将做介绍。

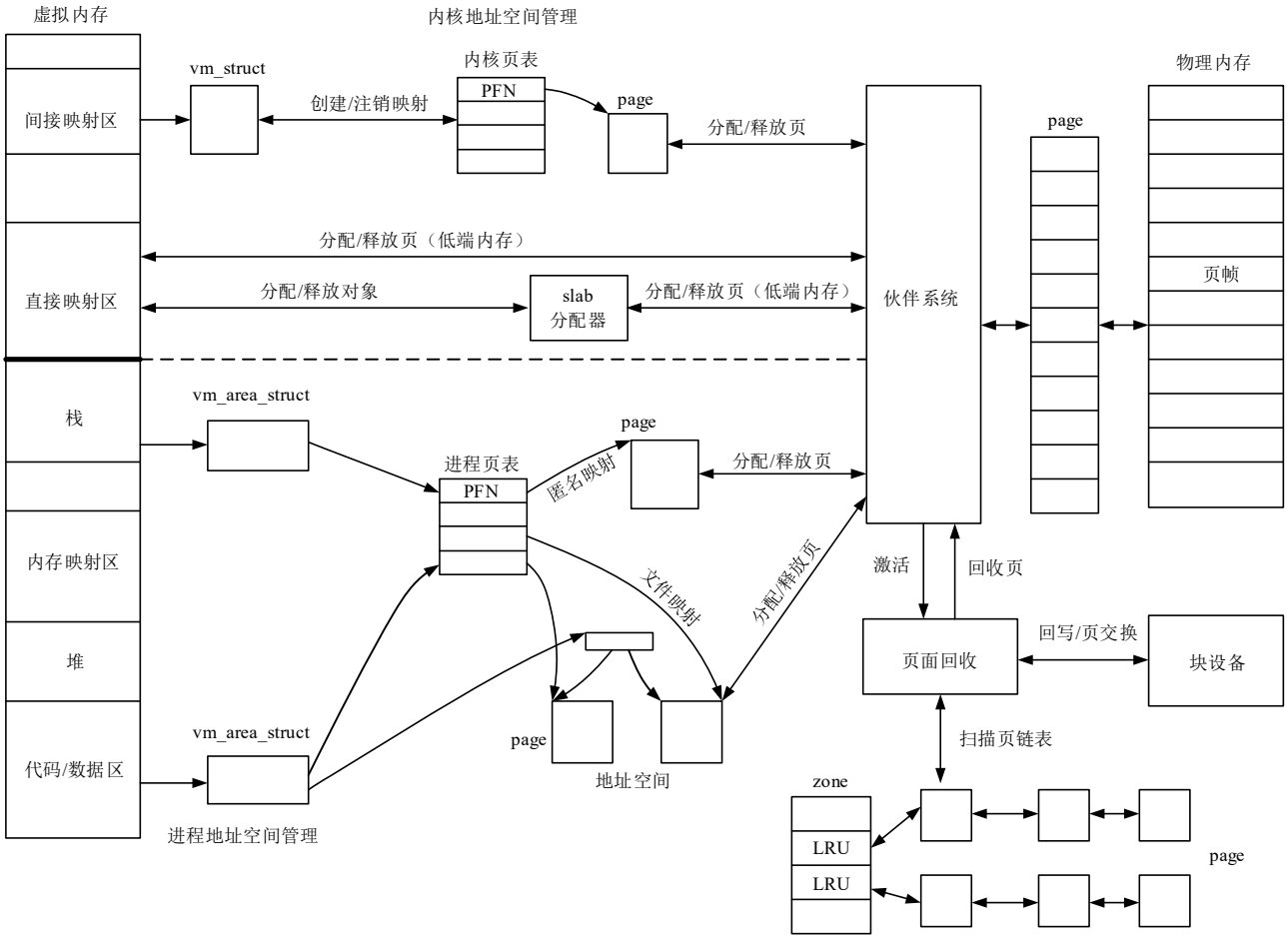
内核物理内存管理中的伙伴系统主要完成对物理内存页的划分，空闲页的分配和释放，虚拟内存管理主要是从伙伴系统中分配页，填充内核/进程页表中相应表项，以建立内核/进程虚拟内存到物理内存的映射，以及映射的解除。

3.1.2 内存管理框架

内核内存管理（含虚拟内存管理）框架如下图所示。内核在启动阶段从命令行参数（或环境变量）获取物理内存信息。内核将物理内存以页为单位进行划分，每个页面称为页帧，页帧在内核中由 page 结构体实例表示，内核建立 page 实例数组用于页帧管理。

伙伴系统用于管理页帧，主要是空闲页帧。伙伴系统将连续的空闲页帧按阶进行划分，阶 order 是一个正整数，表示连续的空闲页帧数量化 2^{order}，例如：0 阶表示连续页帧数为 2^0=1，2 阶表示连续页帧数为 2^2=4。伙伴系统为每个阶建立了双链表，用于管理连续的页帧，例如，2 阶对应的双链表管理的是连续 4 页空闲的内存区域。空闲页帧通过 page 实例，添加到伙伴系统中双链表。

内核可以按阶从伙伴系统中分配内存区域，假设指定分配阶为 2，则伙伴系统为内核分配连续 4 页的空闲内存。



内核直接映射区从伙伴系统分配页帧后，由 page 实例即可得知物理内存的物理地址（由于 page 数组与页帧一一对应），通过线性映射即可得到虚拟地址（物理地址最高位置 1）。

内核为间接映射区和用户空间从伙伴系统分配页帧后，由 page 实例获取物理地址，将页帧号填入对应页表项，即建立虚拟内存与物理内存的映射，这属于虚拟内存管理的内容。

内核间接映射区的虚拟内存区域由 vm_struct 结构体表示，用户空间虚拟内存区域由 vm_area_struct 结构体表示，这两个数据结构中包含映射信息，详见下一章。在这里读者只需要知道，处理器访问到虚拟内存域时，将从 TLB 中查找匹配表项（若不存在匹配表项则到页表中查找并填充至 TLB），由表项中 PFN 生成物理地址。若页表中对应表项为空（映射未建立），则从伙伴系统中分配页，填充页表项，建立映射。

内核在运行过程中要动态分配数据结构实例（小块内存），伙伴系统按页分配的内存对数据结构来说太大，太浪费了，因此内核实现了 slab/slob/slub 分配器，用于内核分配小块的内存。slob 分配器适用于资源有限的嵌入式等小型系统，slub 分配器适用于大型系统，slab 分配器比较适中。内核在配置时只能选择三者中的一个，因此本书后面将用 slab 分配器来指代 slab、slob 或 slub 分配器。

slab 分配器从伙伴系统中按页分配内存，然后划分成小块，分配给内核使用。slab 分配器只能从低端内存中分配页，分配的内存只能用于内核直接映射区，也就是说只能用于内核代码自身，不能用于间接映射区和用户空间。

伙伴系统分配给用户进程的页帧是可以回收的，分配给内核使用的页帧是不进行回收的（slab 缓存可收缩），除非内核主动释放。在伙伴系统分配函数中，当空闲页帧比较紧张时，会激活页面回收机制。页面回收机制扫描页帧 LRU 链表，分配给进程（包括页缓存）的页帧会添加到内存域的 LRU 链表。页面回收时，对于不常用的文件缓存页（映射文件内容），直接将页数据回写到块设备，释放页帧，下次需要时再从文件中读取。对于匿名映射页（不映射到文件）可启动页交换机制，将不常用的页帧数据写入交换区，在页表项中标注交换区位置，释放页帧，在下次需要时再从交换区中读入页帧数据。

本章主要介绍物理内存管理的数据结构及其初始化、伙伴系统、slab 分配器的实现，内核、进程虚拟内存管理（映射的建立和解除）将在第 4 章介绍，页面回收机制将在第 11 章再做介绍。

3.1.3 内存管理初始化

内核在启动阶段将完成内存管理的初始化工作，函数调用关系如下图所示：

以上各函数功能简述如下，本章及下一章将详细介绍下列初始化函数的实现：

●page_address_init()：初始化内核持久映射区地址管理数据结构。

●prom_init()：从引导加载程序传递的环境变量中获取内存大小信息，赋予全局变量，平台定义的函数。

●plat_mem_setup()：将物理内存段信息添加到 boot_mem_map 实例，平台定义的函数。

●bootmem_init()：初始化自举分配器，用位图的方式来实现内存的按页分配，执行完此函数后即可调用自举分配器的分配函数分配内存页，体系结构相关的函数。

●paging_init()：内存管理数据结构初始化，体系结构相关的函数。

▲pagetable_init()：初始化内核持久映射区和固定映射区页表项等，体系结构相关的函数。

▲kmap_init()：内核映射区初始化，体系结构相关的函数。

▲**free_area_init_nodes()**：物理内存管理数据结构初始化，通用函数。

●setup_per_cpu_areas()：percpu 变量早期初始化。

●build_all_zonelists()：初始化结点借用内存列表，通用函数。

●page_alloc_init()：添加 CPU 热插拔通知链事件。

●**mem_init()**：停用自举分配器，释放空闲页至伙伴系统，启用伙伴系统等。

●vmalloc_init(void)：内核地址空间 VMALLOC 映射区初始化。

●kmem_cache_init()、kmem_cache_init_late()：初始化 slab 分配器。

●setup_per_cpu_pageset()：为各内存域创建 per_cpu_pageset 结构体实例，并初始化。

●anon_vma_init()：为匿名映射的反向映射结构创建 slab 缓存。

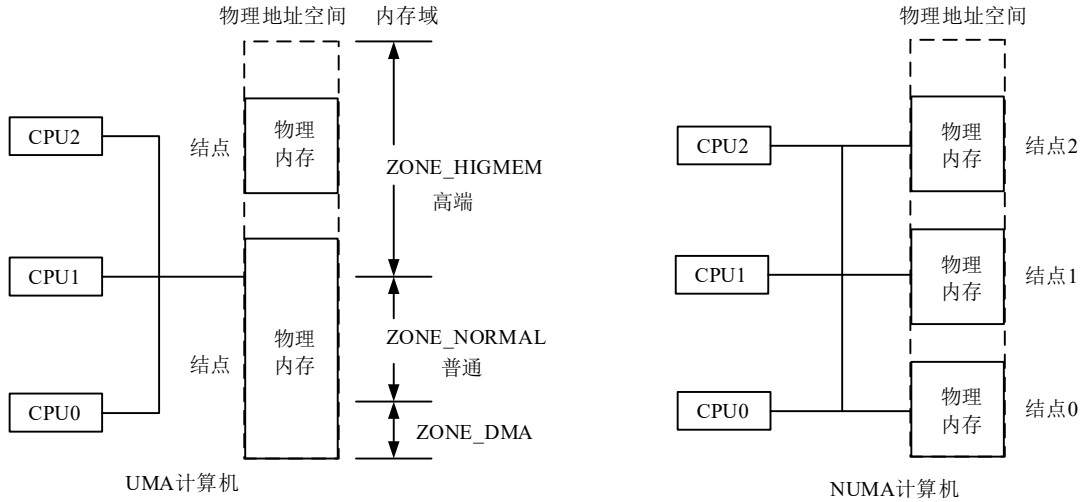
3.2 数据结构

本节介绍物理内存管理中的结点、内存域、页等数据结构。下面先来看一下内核对物理内存如何划分成结点和内存域。根据系统内存管理的方法不同，计算机可分为两种类型：

1、UMA（一致访问内存）计算机：即物理内存的质地是一样的，SMP（对称多处理器）系统各处理器核访问各段内存速度是一样的。

2、NUMA（非一致访问内存）计算机：它只存在于 SMP 系统内，即每个处理器核具有自身的物理内存，各内存以总线连接，统一编址。各 CPU 核访问本地内存速度较快，也可以通过总线访问其它内存，但访问速度较慢。

两种类型的计算机内存管理模型如下图所示：



系统中每个物理内存介质定义为结点，内核根据物理内存是否连续可将内存配置成三个类型：一是 FLATMEM（平坦内存模型，只有一个结点），二是 DISCONTIGMEM（不连续内存模型，多个结点），三是 SPARSEMEM（稀疏内存模型，多个结点）。其中后两者性质一样，只是实现代码有所不同。

UMA 系统一般配置为 FLATMEM 类型，内存存在较大空洞时也可配置成 DISCONTIGMEM 或 SPARSEMEM 类型。NUMA 系统内存只能是 DISCONTIGMEM 或 SPARSEMEM 类型，NUMA 计算机还

有专门的配置选项 NUMA，选择了此选项时会自动选择 DISCONTIGMEM 类型。

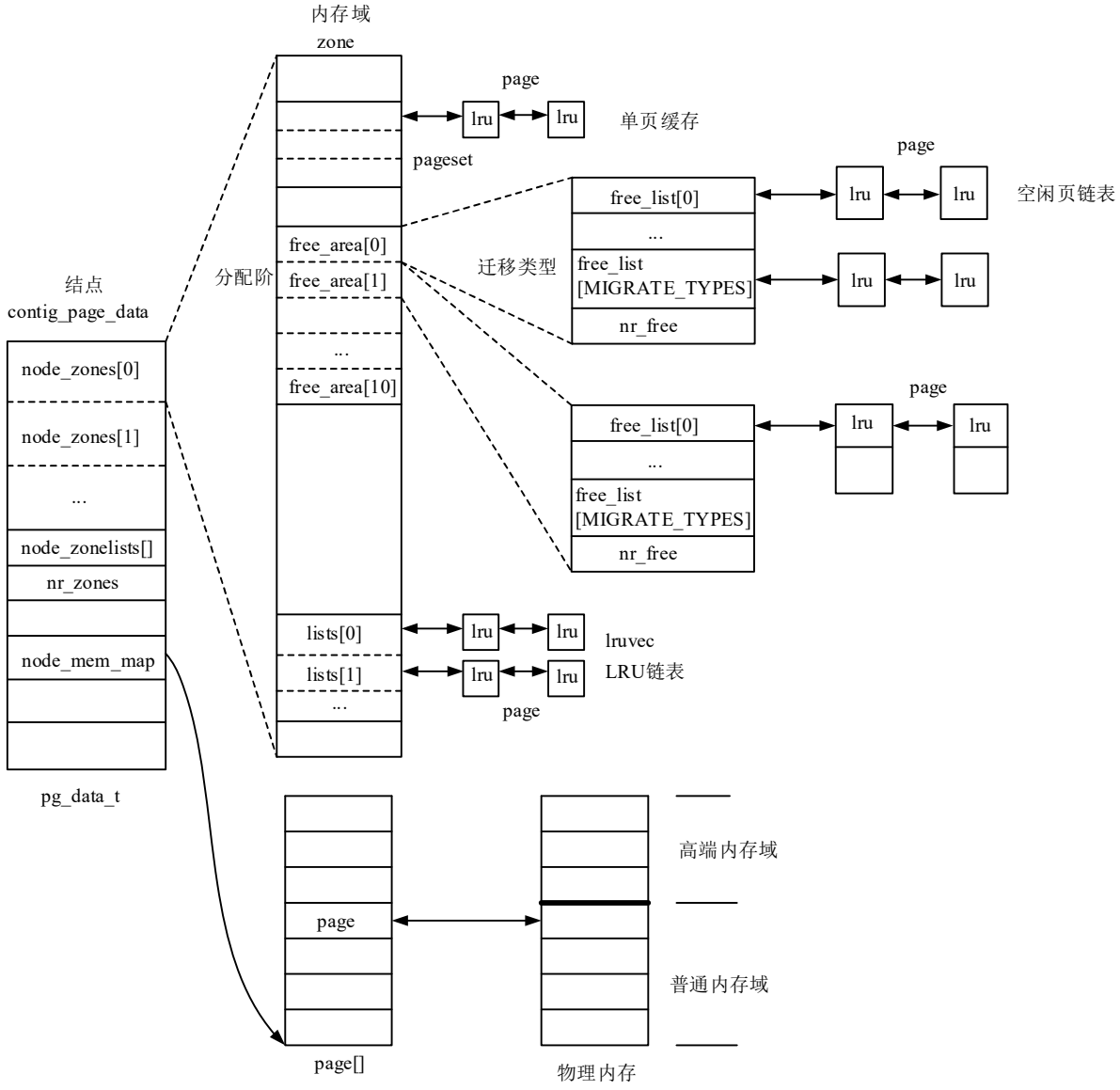
若 UMA 计算机中在多个内存结点，可选择/mm/Kconfig 配置文件内 DISCONTIGMEM 选项，此选项会选择 NEED_MULTIPLE_NODES 选项，NUMA 选项也会自动选择 NEED_MULTIPLE_NODES 选项，选择了 NMUA 或 DISCONTIGMEM 选项时表示系统具有多个内存结点。本书只讨论最常见的 FLATMEM 内存模型，这也是内核默认的配置类型，即系统只存在一个内存结点。

内核将物理内存按介质划分成结点，又按起止地址将物理内存划分成不同的内存域（普通内存域、高端内存域等），如上图左侧所示。某个内存结点可全部位于某一个内存域内也可以跨越多个内存域，某一内存域也可以没有对应的物理内存，如高端内存域，但是普通内存域是可以保证有与之对应的物理内存。内存域的划分标准由体系结构（处理器）访问物理内存的机制决定的，例如：MIPS32 体系结构低 512MB 物理内存被划分为普通内存域，因为内核可直接访问此区域而不需要经过页表转换，而高于 512MB 的物理内存划分为高端内存域。

DMA 内存域位于物理内存最底端，表示处理器 DMA 能访问的区域。某些体架结构（处理器）中 DMA 只能访问物理内存底端的限制大小的内存，如 x86_64 体系结构，DMA 只能访问物理内存的低 16MB 空间，此区域定义成 DMA 内存域。没有此限制的体架结构（处理器）不需要 DMA 内存域。

内存域中物理内存按页（4KB）划分成页帧，内核对物理内存的管理以页为单位进行。

在内核物理内存管理中，由 pg_data_t 结构体表示内存结点，zone 结构体表示内存域，page 结构体表示页帧，各结构体实例组织关系如下图所示：



内存结点结构体中包含指向表示页帧的 page 结构体实例数组的指针、内存域结构体实例等成员，内存域结构体内主要成员有：实现伙伴系统的 free_area 结构体实例数组，free_area 实例管理着本内存域的空闲内存区域（按迁移类型划分），这是分配内存页的主要数据结构；页结构 LRU 链表，主要用于页面回收机制。

3.2.1 结点

系统中每个物理内存介质称为一个结点，每个结点在内核中由 pglist_data（pg_data_t）结构体表示，结构体定义在/include/linux/mmzone.h 头文件内：

```
typedef struct pglist_data {
    struct zone    node_zones[MAX_NR_ZONES];              /*内存域实例数组，非指针*/
    struct zonelist    node_zonelists[MAX_ZONELISTS];     /*借用内存列表，伙伴系统使用*/
    int    nr_zones;                                      /*结点实际所含内存域数目*/
#ifdef CONFIG_FLAT_NODE_MEM_MAP                           /* FLATMEM 类型默认选择此选项 */
    struct page *node_mem_map;                            /*page 实例数组指针*/
  #ifdef CONFIG_PAGE_EXTENSION
     struct page_ext *node_page_ext;
  #endif
#endif
#ifndef   CONFIG_NO_BOOTMEM
     struct bootmem_data *bdata;              /*自举分配器数据结构指针*/
#endif
#ifdef CONFIG_MEMORY_HOTPLUG       /*内存热插拔*/
    spinlock_t node_size_lock;
#endif
    unsigned long    node_start_pfn;             /*结点第一个页帧逻辑编号，UMA 类型为 0*/
    unsigned long    node_present_pages;         /*结点页帧数量，不含空洞*/
    unsigned long    node_spanned_pages;         /*结点页帧数量，含空洞*/
    int    node_id;                              /*结点编号，UMA 系统为 0*/

    wait_queue_head_t    kswapd_wait;          /*交换守护进程的等待队列*/
    wait_queue_head_t    pfmemalloc_wait;      /*等待内存分配的进程队列*/

    struct task_struct    *kswapd;             /*指向页回收守护线程 task_struct 实例*/
    int    kswapd_max_order;                    /*页交换子系统定义需要释放区域的长度*/
    enum zone_type    classzone_idx;           /*执行页回收的内存域编号*/

#ifdef CONFIG_NUMA_BALANCING
    ...
#endif

#ifdef CONFIG_DEFERRED_STRUCT_PAGE_INIT
    unsigned long    first_deferred_pfn;
#endif
```

} **pg_data_t**;

pglist_data 结构体主要成员简介如下：

●**node_zones**[MAX_NR_ZONES]：内存域 zone 结构体实例数组，非指针。描述结点跨跃的内存域信息，结点结构体中包含表示所有内存域的实例，本结点没有跨越的内存域其数组项内容赋值为 0。

●node_zonelists[MAX_ZONELISTS]：借用内存域列表，在当前内存域中没有足够空闲页帧时，备用的借用内存域列表，见 3.5 节。

●nr_zones：本结点实际跨越的内存域数量。

●node_mem_map：指向结点物理页帧 page 实例数组的指针。

●bdata：自举分配器数据结构指针，详见 3.3 节。

●node_start_pfn：结点第一个页帧在系统内的逻辑编号，编号从 0 开始，UMA 系统中总是为 0。

●kswapd_wait：页回收守护线程的等待队列。

●kswapd：内存结点页回收守护线程 task_struct 实例指针，守护线程由 kswapd_run()函数在初始化子系统时创建，详见第 11 章。

●kswapd_max_order：页回收守护线程执行时设置的分配函数的分配阶。

●classzone_idx：执行页回收内存域编号，需要对 classzone_idx 及其下的内存域进行页回收。

●pfmemalloc_wait：等待内存分配的进程等待队列。

FLATMEM 系统中，唯一的结点实例定义在/mm/bootmem.c（没有选择 NO_BOOTEME 配置选项）文件内：

```
#ifndef CONFIG_NEED_MULTIPLE_NODES        /*不需要多个结点*/
  struct pglist_data __refdata contig_page_data = {
      .bdata = &bootmem_node_data[0]          /*自举分配器数据结构实例，/mm/bootmem.c*/
  };
  EXPORT_SYMBOL(contig_page_data);
#endif
```

NODE_DATA(nid)宏用于获取 pg_data_t 实例地址，定义在/include/linux/mmzone.h 头文件：

```
#define NODE_DATA(nid)      (&contig_page_data)
```

由于 FLATMEM 系统只有一个内存结点，因此 NODE_DATA(nid)宏始终返回 contig_page_data 实例地址，参数 nid 无意义。内核在内存管理初始化过程中将对 contig_page_data 实例各成员进行初始化。

3.2.2 物理内存域

内核根据处理器访问物理内存的方式，对物理内存按地址划分成内存域，内存域类型由枚举类型表示，定义在头文件/include/linux/mmzone.h：

```
enum zone_type {
  #ifdef CONFIG_ZONE_DMA
    ZONE_DMA,              /*DMA 内存域*/
  #endif
  #ifdef CONFIG_ZONE_DMA32
    ZONE_DMA32,            /*64 位系统才具有此内存域*/
  #endif
    ZONE_NORMAL,           /*普通内存域，低端内存*/
  #ifdef CONFIG_HIGHMEM    /*64 位系统中没有高端内存域，
```

```
                                *因为低端内存域空间足够大，能容下所有物理内存*/
    ZONE_HIGHMEM,          /*高端内存域，高端内存*/
  #endif
    ZONE_MOVABLE,          /*可移动内存域，伪内存域，用于防止内存碎片*/
    __MAX_NR_ZONES         /*内存域类型数量*/
};
```

各内存域类型定义如下：

●ZONE_DMA：适合 DMA 的内存域，位于物理内存最底端。某些体系结构（处理器）中 DMA 只能访问物理内存底端的限制大小的内存，如 x86_64 体系结构，DMA 只能访问物理内存的低 16MB 空间，此区域定义成 DMA 内存域。没有此限制的体系结构（处理器）不需要 DMA 内存域，MIPS 通常不需要此内存域。

●ZONE_DMA32：标记可以用 32 位地址寻址、适合 DMA 的内存域。显然只有在 64 位系统中才有此内存域，32 位系统中此内存域为空。

●**ZONE_NORMAL**：普通内存域，内核直接映射的物理内存区域（低端内存），不经过页表可直接访问（适用于 MIPS 架构，其它体系结构可能需要经过页表访问）。此区域是所有体系结构唯一保证会存在的区域，MIPS32 体系结构中此内存域包含物理内存低 512MB 空间。

●**ZONE_HIGHMEM**：高端内存域，超出内核直接映射区的物理内存区域（高于 512MB 的物理内存）。

●ZONE_MOVABLE：可移动内存域，伪内存域，从其它内存域中划出一部分归入此区域，用于防止内存碎片。

内核代码中经常用到的常数 MAX_NR_ZONES，在/kernel/bounds.c 内定义成与__MAX_NR_ZONES 相同，表示系统中内存域类型数量。

内核中描述内存域的 zone 结构体定义在/include/linux/mmzone.h 头文件：

```
struct zone {
    unsigned long    watermark[NR_WMARK];    /*内存域水印值，NR_WMARK=3*/
    long    lowmem_reserve[MAX_NR_ZONES]; /*为更高级内存域预留的页帧数，供紧急情况使用*/
                                          /*本内存域 lowmem_reserve[]数组项值为 0*/
  #ifdef CONFIG_NUMA
    int node;
  #endif
    unsigned int    inactive_ratio;       /*活跃匿名映射页 LRU 链表页数与不活跃链表页数最大比值*/
    struct pglist_data  *zone_pgdat;                /*指向所属结点数据结构*/
    struct per_cpu_pageset    __percpu    *pageset;    /*用于管理特定于 CPU 的单页缓存*/
    unsigned long    dirty_balance_reserve;         /*内存域中不可设置为脏的页数量*/

  #ifndef CONFIG_SPARSEMEM                  /*没有配置稀疏内存类型，存在此成员*/
    unsigned long  *pageblock_flags;        /*页块标记数组指针，标记页块迁移属性*/
  #endif

  #ifdef CONFIG_NUMA
    ...
  #endif
```

```
    unsigned long       zone_start_pfn;      /*内存域起始页帧号*/
    unsigned long       managed_pages;       /*伙伴系统可管理的实际页帧数量*/
    unsigned long       spanned_pages;       /*内存域跨越的页帧数量，包含空洞*/
    unsigned long       present_pages;       /*内存域实际存在的页帧数量，不含空洞*/
    const char          *name;               /*内存域名称*/

    int   nr_migrate_reserve_block;          /*设置为 MIGRATE_RESERVE 迁移类型的页块数*/

#ifdef   CONFIG_MEMORY_ISOLATION        /*支持内存隔离*/
    unsigned long       nr_isolate_pageblock;
#endif

#ifdef   CONFIG_MEMORY_HOTPLUG     /*支持内存热插拔，少量体系结构支持此特性*/
    seqlock_t       span_seqlock;
#endif

    wait_queue_head_t   *wait_table;             /*等待队列，供等待某一页为可用的进程使用*/
    unsigned long       wait_table_hash_nr_entries;
    unsigned long       wait_table_bits;

    ZONE_PADDING(_pad1_)             /*用于缓存行对齐*/
    struct free_area    free_area[MAX_ORDER];      /*伙伴链表，实现伙伴系统，后面再做介绍*/
    unsigned long       flags;               /*内存域标记*/
    spinlock_t          lock;                /*保护自旋锁*/

    ZONE_PADDING(_pad2_)
    spinlock_t          lru_lock;                /*保护 LRU 链表的自旋锁，锁竞争可能会很激烈*/
    struct lruvec       lruvec;                  /*页 LRU 链表数组，用于页面回收机制*/
    atomic_long_t       inactive_age;
    unsigned long       percpu_drift_mark;

#if defined CONFIG_COMPACTION || defined CONFIG_CMA     /*内存规整或 CMA*/
    unsigned long   compact_cached_free_pfn;
    unsigned long   compact_cached_migrate_pfn[2];
#endif

#ifdef CONFIG_COMPACTION             /*内存规整，用于降低系统内存碎片*/
    unsigned int    compact_considered;
    unsigned int    compact_defer_shift;
    int             compact_order_failed;
#endif

#if defined CONFIG_COMPACTION || defined CONFIG_CMA
```

```
        bool        compact_blockskip_flush;
    #endif

        ZONE_PADDING(_pad3_)
        atomic_long_t       vm_stat[NR_VM_ZONE_STAT_ITEMS];   /*内存域统计量*/
    } ____cacheline_internodealigned_in_smp;
```

内存域 zone 结构体被 ZONE_PADDING 宏分成四部分，主要是为了产生填充字节实现数据结构在缓存行中对齐，以加速访问。下面分别介绍结构体中比较重要的成员：

●**watermark[NR_WMARK]**：页回收与页交换中使用的水印值，数组项由枚举类型 zone_watermarks 表示，枚举类型定义在/include/linux/mmzone.h 头文件：

```
enum zone_watermarks {
    WMARK_MIN,
    WMARK_LOW,
    WMARK_HIGH,
    NR_WMARK     /*数组项数*/
};
```

各数组项语义如下：

watermark[WMARK_MIN]：表示如果内存域空闲页帧的数量低于此值，说明内存域中急需空闲页，页面回收的工作压力比较大，详见第 11 章。

watermark[WMARK_LOW]：如果空闲页帧数量低于此值，则启动页面回收机制。

watermark[WMARK_HIGH]：如果空闲页帧数量多于此值，则表示内存域状态是理想的。

●**lowmem_reserve[MAX_NR_ZONES]**：指示本内存域为本结点更高级内存域预留的页帧数，用于那些无论如何都不能失败的关键性分配。例如，在普通内存域中为高端内存域预留的页帧数。

●**inactive_ratio**：LRU_ACTIVE_ANON 链表与 LRU_INACTIVE_ANON 链表页数量比值，即活跃与不活跃匿名映射页链表中页数量的比值。如：比值为 3：1，表示四分之一的匿名映射页为不活跃的。

●**pageset**：percpu 变量指针，实际指向的是 per_cpu_pageset 结构体实例数组，处理器中每个 CPU 核对应数组中的一项，CPU 核只能访问与之关联的实例，不能访问其它数组项，这样避免了竞争，有利于提高系统性能。

per_cpu_pageset 结构体用于建立特定于 CPU 的空闲单页缓存，结构体定义在/include/linux/mmzone.h 头文件：

```
struct per_cpu_pageset {
    struct per_cpu_pages    pcp;        /*per_cpu_pages 结构体实例*/
  #ifdef CONFIG_NUMA
    ...
  #endif
  #ifdef CONFIG_SMP
    s8    stat_threshold;
    s8    vm_stat_diff[NR_VM_ZONE_STAT_ITEMS];
  #endif
};
```

per_cpu_pageset 结构体中内嵌 per_cpu_pages 结构体定义在/include/linux/mmzone.h 头文件内：

```
struct per_cpu_pages {
    int    count;              /*lists[]链表中页的数量*/
    int    high;               /*当链表中页的数量大于 high 时，则释放 batch 数量的页到伙伴系统*/
    int    batch;              /*释放到伙伴系统的页数量，或从伙伴系统分配 batch 数量单页至缓存的*/
    struct list_head    lists[MIGRATE_PCPTYPES];      /*单页缓存双链表（数组）*/
};
```

per_cpu_pages 结构体中 lists[]链表成员中链接的是空闲单页的 page 实例，用于为 CPU 创建空闲单页缓存。当内核释放单个页帧时，其实是将其释放到 per_cpu_pages 实例链表中，而不是直接释放到伙伴系统。当单页缓存中页的数量多于（等于）high 时，则将 batch 数量的页释放到伙伴系统。分配单页时从 per_cpu_pages 链表中分配，如果缓存中没有单页，则先从伙伴系统分配 batch 数量的单页添加到缓存，然后再分配。

per_cpu_pages 结构体中双链表数目由枚举类型定义，双链表按内存迁移类型分类。双链表数组项数为 MIGRATE_PCPTYPES，定义在/include/linux/mmzone.h 头文件：

```
enum {
    MIGRATE_UNMOVABLE,          /*不可移动页链表*/
    MIGRATE_RECLAIMABLE,        /*可回收页链表*/
    MIGRATE_MOVABLE,            /*可移动页链表*/
    MIGRATE_PCPTYPES,           /*per_cpu_pages 中双链表数量*/
    MIGRATE_RESERVE = MIGRATE_PCPTYPES,
#ifdef CONFIG_CMA
    MIGRATE_CMA,
#endif
#ifdef  CONFIG_MEMORY_ISOLATION
    MIGRATE_ISOLATE,        /* can't allocate from here */
#endif
    MIGRATE_TYPES           /*页迁移类型数量*/
};
```

●**pageblock_flags**：页块迁移类型数组指针。页块迁移类型用于防止内存碎片，内核在内存初始化时将内存域物理内存按块（$2^{MAX\_ORDER-1}$ 数量的连续页帧）进行划分，对每个页块定义迁移类型。pageblock_flags 指向的整数数组用于标记页块的迁移类型，详见 3.4 节。

●**spanned_pages**：内存域跨越的页帧数量，包含空洞。

●**present_pages**：内存域实际存在的页帧数量，不含空洞。

●**managed_pages**：伙伴系统可管理的页帧数量，除去被内核自身占用和 bootmem 分配器已经分配使用的页帧。

●**nr_migrate_reserve_block**：内存域中设置为 MIGRATE_RESERVE 迁移类型的页块数量。

●**wait_table、wait_table_bits、wait_table_hash_nr_entries**：用于实现在页上等待的等待进程队列。由于内存域中页数量较多，不可能为每个页帧创建等待队列，因此内存域创建固定大小的等待队列数组，在页上等待的进程通过散列的方式添加到某个数组项表示的等待队列中。wait_table 指向 wait_queue_head_t 等待队列数组基地址，wait_table_hash_nr_entries 表示等待队列数量，wait_table_bits 为队列数量的 2 的幂次表示，即 $2^{wait\_table\_bits-1}$=wait_table_hash_nr_entries。

●**free_area[MAX_ORDER]**：伙伴页双链表，实现伙伴系统的主要数据结构，后面将详细介绍。

●**flags**：内存域标记，取值定义在/include/linux/mmzone.h 头文件内：

```
enum zone_flags {
    ZONE_RECLAIM_LOCKED,    /*阻止当前页面回收*/
    ZONE_OOM_LOCKED,        /*zone is in OOM killer zonelist */
    ZONE_CONGESTED,         /*内存域被许多脏页阻塞*/
    ZONE_DIRTY,             /*页回收在 LRU 链表末尾扫描到许多文件页缓存脏页*/
    ZONE_WRITEBACK,         /*页回收扫描发现许多页正在回写*/
    ZONE_FAIR_DEPLETED,     /* fair zone policy batch depleted */
};
```

●**lruvec**：lruvec 结构体实例，主要包含页帧 LRU 链表。分配给进程的页帧（包括文件页缓存的页）会加入到 LRU 链表。页回收机制扫描 LRU 链表，对不常用的页进行回收。

lruvec 结构体定义在/include/linux/mmzone.h 头文件内：

```
struct lruvec {
    struct list_head    lists[NR_LRU_LISTS];    /*双链表数组*/
    struct zone_reclaim_stat    reclaim_stat;       /*页回收统计量*/
  #ifdef CONFIG_MEMCG
    struct zone *zone;
  #endif
};
```

lruvec 结构体内包含一个双链表数组和 zone_reclaim_stat 结构体实例。双链表数组项数 NR_LRU_LISTS 在枚举类型 lru_list 内定义：

```
#define   LRU_BASE      0
#define   LRU_ACTIVE    1
#define   LRU_FILE      2
enum lru_list {                                 /*/include/linux/mmzone.h*/
    LRU_INACTIVE_ANON = LRU_BASE,                   /*不活跃匿名映射页链表*/
    LRU_ACTIVE_ANON = LRU_BASE + LRU_ACTIVE,    /*活跃匿名映射页链表*/
    LRU_INACTIVE_FILE = LRU_BASE + LRU_FILE,        /*不活跃文件缓存页链表*/
    LRU_ACTIVE_FILE = LRU_BASE + LRU_FILE + LRU_ACTIVE,    /*活跃文件缓存页链表*/
    LRU_UNEVICTABLE,                /*不可回收页链表*/
    NR_LRU_LISTS            /*双链表数组项数*/
};
```

zone_reclaim_stat 结构体定义在/include/linux/mmzone.h 头文件，表示页回收状态：

```
struct zone_reclaim_stat {
    unsigned long       recent_rotated[2];
                            /*最近扫描链表页数量，[0]表示匿名映射页，[1]表示文件缓存页*/
    unsigned long       recent_scanned[2];
                /*扫描不活跃链表时，放回活跃链表的页数量，扫描活跃链表的引用页也加入其中*/
};
```

reclaim_stat 成员在页回收机制中使用。

●**inactive_age**：用于记录文件缓存页不活跃链表中的 eviction 和 activation 操作的计数。

●**vm_stat[NR_VM_ZONE_STAT_ITEMS]**：内存域统计量数组，每个数组项表示某一类型统计量的

数值。数组项数由 zone_stat_item 枚举类型确定，定义在/include/linux/mmzone.h 头文件：

```
enum zone_stat_item {           /*统计项目*/
    NR_FREE_PAGES,                  /*空闲页数量*/
    NR_ALLOC_BATCH,
    NR_LRU_BASE,                    /*页面 LRU 链表基值*/
    NR_INACTIVE_ANON = NR_LRU_BASE,   /*当前不活跃匿名映射 LRU 链表页数量*/
    NR_ACTIVE_ANON,         /*当前活跃匿名映射 LRU 链表页数量*/
    NR_INACTIVE_FILE,       /*当前不活跃文件缓存 LRU 链表页数量*/
    NR_ACTIVE_FILE,         /*当前活跃文件缓存 LRU 链表页数量*/
    NR_UNEVICTABLE,         /*当前不可回收 LRU 链表页数量*/
    NR_MLOCK,               /*被锁定页数量*/
    NR_ANON_PAGES,          /*匿名映射页数量*/
    NR_FILE_MAPPED,         /*文件缓存页数量*/
    NR_FILE_PAGES,          /*位于文件页缓存中页数量*/
    NR_FILE_DIRTY,          /*脏文件缓存页数量*/
    NR_WRITEBACK,           /*回写的页数量*/
    NR_SLAB_RECLAIMABLE,        /*slab 缓存中可回收页数量*/
    NR_SLAB_UNRECLAIMABLE,    /*slab 缓存中不可回收页数量*/
    NR_PAGETABLE,               /*用于页表的页数量*/
    NR_KERNEL_STACK,
    NR_UNSTABLE_NFS,            /* NFS unstable pages */
    NR_BOUNCE,
    NR_VMSCAN_WRITE,
    NR_VMSCAN_IMMEDIATE,     /* Prioritise for reclaim when writeback ends */
    NR_WRITEBACK_TEMP,        /* Writeback using temporary buffers */
    NR_ISOLATED_ANON,           /*当前正在处理的从匿名映射页 LRU 链表中分离的页数量*/
    NR_ISOLATED_FILE,           /*当前正在处理的从缓存页 LRU 链表中分离的页数量*/
    NR_SHMEM,                   /*共享内存页数量*/
    NR_DIRTIED,                 /*脏页数量*/
    NR_WRITTEN,                 /*被写的页数量*/
    NR_PAGES_SCANNED,       /*页回收中扫描 LRU 链表中页数量*/
#ifdef CONFIG_NUMA
    ...
#endif
    WORKINGSET_REFAULT,
    WORKINGSET_ACTIVATE,
    WORKINGSET_NODERECLAIM,
    NR_ANON_TRANSPARENT_HUGEPAGES,
    NR_FREE_CMA_PAGES,
    NR_VM_ZONE_STAT_ITEMS
};
```

内核在/mm/vmstat.c 文件内定义了统计所有内存域信息的静态全局数组：

```
atomic_long_t vm_stat[NR_VM_ZONE_STAT_ITEMS] __cacheline_aligned_in_smp;
```

内核在/include/linux/vmstat.h 头文件定义了读写内存域统计信息的函数，例如：

●unsigned long zone_page_state(struct zone *zone,enum zone_stat_item item)：读取指定内存域指定统计量的值；

●unsigned long global_page_state(enum zone_stat_item item)：读取统计量全局的统计量值。

●void zone_page_state_add(long x, struct zone *zone,enum zone_stat_item item)：指定内存域指定统计量增加 x 值.

●void inc_zone_state(struct zone *, enum zone_stat_item)：内存域指定统计量值加 1。

●void dec_zone_state(struct zone *, enum zone_stat_item)：内存域指定统计量值减 1。

3.2.3 伙伴系统链表

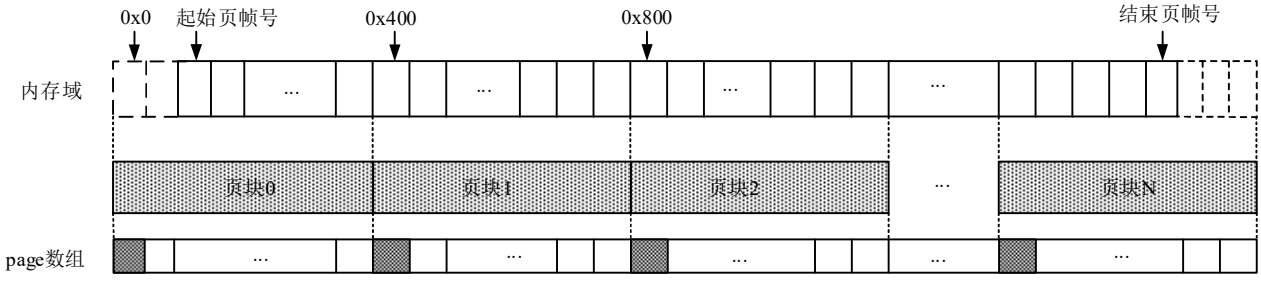
内存域 zone 结构体中 free_area[MAX_ORDER]成员用于管理内存域中空闲页帧，是实现伙伴系统的主要数据结构。free_area[MAX_ORDER]数组项数 MAX_ORDER 定义在/include/linux/mmzone.h 头文件内：

```
#ifndef CONFIG_FORCE_MAX_ZONEORDER
    #define   MAX_ORDER   11    /*free_area[]数组项数，支持的分配阶数量，从 0 开始，0-10*/
#else
    #define   MAX_ORDER   CONFIG_FORCE_MAX_ZONEORDER
#endif
#define   MAX_ORDER_NR_PAGES    (1 << (MAX_ORDER - 1))   /*分配的最大内存块页帧数量*/
```

分配阶（order）表示空闲内存块包含的连续页帧数量为 2 的 order 次幂，即页帧数量为 2^{order}。0 阶表示 1 个页帧的内存块，2 阶表示 4 个页帧的内存块，依此类推。伙伴系统按阶对空闲内存块进行管理，即内存块页帧数量必须是 2 的 N 次幂（N<MAX_ORDER），分配和释放函数只能按阶分配和释放内存块，也就是说并不能分配/释放任意大小的内存块，必须是 2 的 N 次幂个连续页帧。

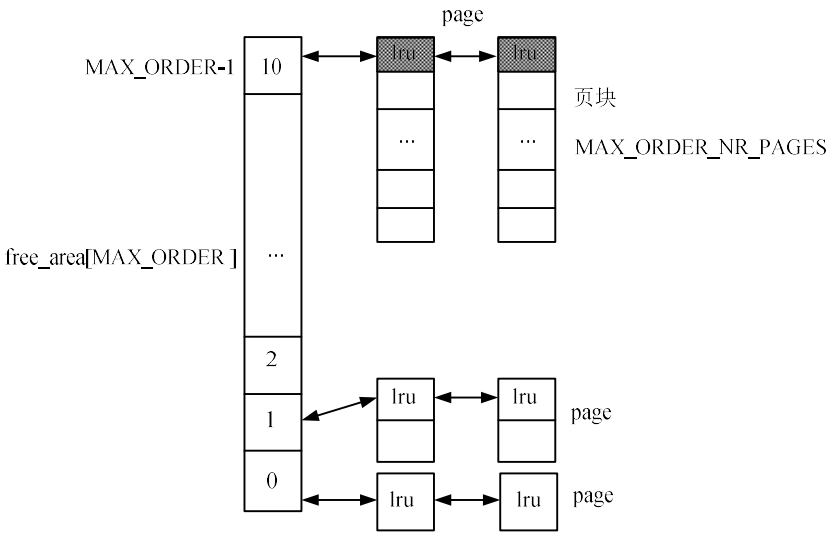
MAX_ORDER 为支持的分配阶数量，分配阶从 0 开始，因此最大分配阶为（MAX_ORDE-1），表示伙伴系统能分配/释放的最大内存块页帧数量为 $2^{MAX\_ORDER-1}$，即宏 MAX_ORDER_NR_PAGES 表示的页帧数量。MAX_ORDER 通常为 11，即伙伴系统能分配的最大内存块页帧数量为 2^{10}=1024（0x400）。

内核在内存管理初始化阶段对每个内存域内以 MAX_ORDER_NR_PAGES 个连续页帧为单位进行划分，划分的每个内存块称为页块（pageblock），页块即伙伴系统管理的最大连续内存块。在划分页块的过程中，还要对页帧号按 MAX_ORDER_NR_PAGES 对齐。下图示意了页块的划分方法：

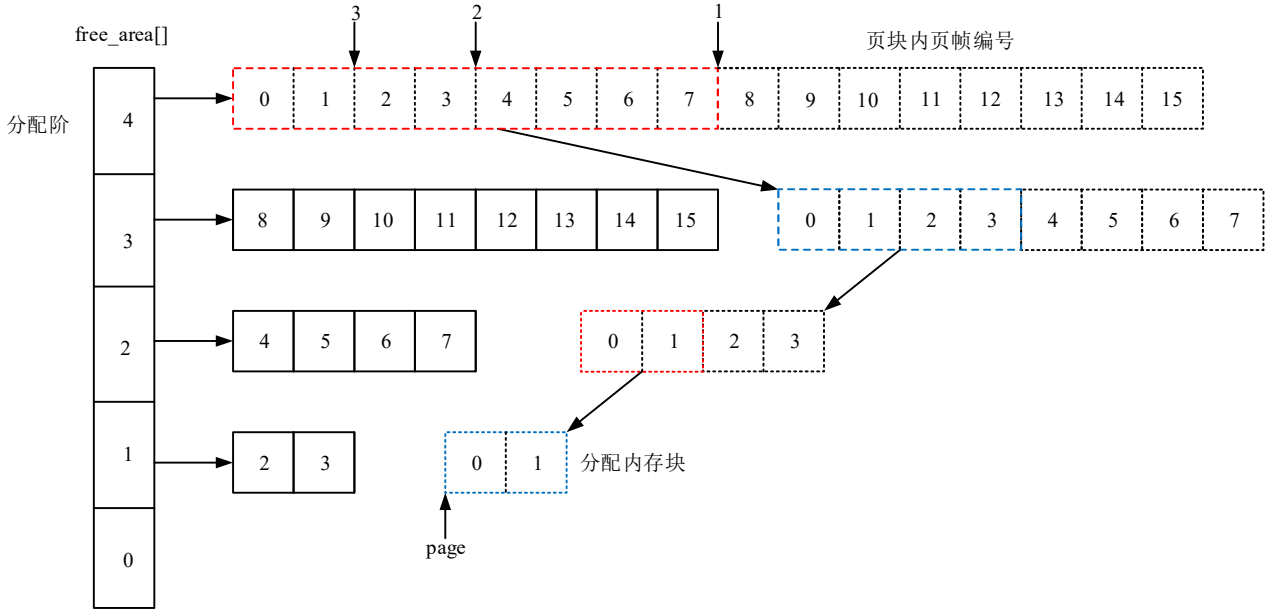


为不失一般性，假设物理内存不是从 0 号页帧开始（但起始页帧号小于 MAX_ORDER_NR_PAGES），划分页块时需要 MAX_ORDER_NR_PAGES 页帧数对齐，因此页块 0 从 0 号页帧开始，包含内存域起始页帧，内存域末尾不足一个页块的也视为一个页块（两头扩展）。

内存初始化过程中为页块中包含的页帧创建 page 实例数组，每个页帧对应一个 page 实例。zone 结构体中 free_area[]数组成员包含内存块链表，数组项索引值表示链接内存块的阶数，内存块通过首页 page 实例中的 lru 成员链接到对应的链表，如下图所示：



例如：free_area[MAX_ORDER-1]数组项链接的是空闲页块（内存块），通过页块首页的 page 实例 lru 成员链接到链表，内存块页帧数量为 $2^{MAX\_ORDER-1}$=MAX_ORDER_NR_PAGES，首页 page 实例 private 成员保存了本内存块的阶数。每降一阶，数组项链接的内存块页帧数量减半。链表中比页块小的内存块来自于对页块的进一步划分（分配过程中划分）。举个例子更容易理解，假设页块大小为 16 个页帧，下图示意了从页块中分配 1 阶（2 个连续页帧）内存块的过程：



上图中页帧编号不是真实编号，而是页帧在页块内的偏移量。分配函数比较目标阶数（1 阶）与当前页块阶数（4 阶）大小，页块阶数更大，则将当前页块对半分，后半部分（8-15 页帧）内存块添加到下一阶（3 阶）链表。前半部分（0-7 页帧）阶数还是比目际阶数更大，则再对半分，4-7 页帧内存块添加到 2 阶链表。0-3 页帧内存块仍然比目标阶更大，则再对其对半分，2-3 页帧内存块添加到 1 阶链表，而剩下的 0-1 页帧内存块阶数正好与目标阶相同，则 0-1 页帧内存块返回给分配函数调用者，分配结束。执行完分配操作后页块被划分成 2 的 N 次幂的更小的内存块，被添加到更低阶链表。

上面介绍的是从页块中分配内存块的操作，实际的分配函数先在目标阶对应的 free_area[order]链表中查找是否有空闲的内存块，如果有则直接从链表中取出内存块返回给分配函数调用者，分配结束。如果目标阶链表中没有空闲内存块，则需要逐级到高阶的链表中查找空闲内存块，然后将高阶的内存块进行拆分，得到所需阶的内存块返回给调用者。

将页帧（内存块）释放回伙伴系统是分配的逆过程。释放操作首先找到内存块阶数对应的 free_area[order]

链表，然后判断释放的内存块是否可以与链表中相邻的空闲内存块合并，以生成更高阶的内存块，如果可以则合并，并将合并后内存块添加到高阶链表。合并操作可以逐阶往上依次执行，直至不能合并为止或到达最高阶；如果不能合并，则直接将内存块添加到 free_area[order]链表。

这里需要注意的是，并不是只要相邻的空闲内存块就可以合并，合并操作要沿着拆分操作相反的路径进行。例如：假设释放上图中分配的 2-3 页帧，且 0-1 页帧和 4-5 页帧都在 1 阶链表中，这里 2-3 页帧只能和 0-1 页帧合并生成 2 阶内存块，而不能和 4-5 页帧合并生成 2 阶内存块（拆分时它们不在同一个 2 阶内存块中）。2-3 页帧和 0-1 页帧可以合并，它们称为伙伴，而与 4-5 页帧不是伙伴（不能合并生成高一阶的内存块）。但是 0-3 页帧可以和 4-7 页帧合并生成 3 阶内存块，0-3 页帧和 4-7 页帧是伙伴。总之，合并时需要遵守分配时拆分内存块的配对关系。

内存域 zone 结构体嵌入 free_area 结构体数组，用于管理内存域连续空闲内存块：

```
struct    zone{
    ...
    struct free_area     free_area[MAX_ORDER];   /*free_area 实例数组，数组项数 MAX_ORDER*/
    ...
}
```

free_area[]数组项是 free_area 结构体实例，每个分配阶对应数组中的一项，即一个 free_area 结构体实例，free_area 结构体管理某一阶的内存块。例如：free_area[0]管理的是 0 阶内存块（单页），free_area[2]管理的是 2 阶内存块。

free_area 结构体定义在/include/linux/mmzone.h 头文件内：

```
struct free_area {
    struct list_head    free_list[MIGRATE_TYPES];    /*伙伴内存块链表*/
    unsigned long       nr_free;      /*空闲内存区数量，即伙伴系统中本阶管理的内存块数量*/
};
```

free_area 结构体中包含双链表数组 free_list[]，每个双链表管理某一迁移类型的同阶内存块。nr_free 表示 free_list[]双链表数组中空闲内存块的总数量，注意这里不是页帧的数量，而是按阶划分的内存块的数量。例如：free_area[1].nr_free=2，表示 1 阶空闲链表中共有 2 个空闲内存块，页帧数量为 $2\times2^1=4$。

free_area 结构体中 free_list[]数组项数为 MIGRATE_TYPES，定义在/include/linux/mmzone.h 头文件内：

```
enum {
    MIGRATE_UNMOVABLE,          /*不可移动页*/
    MIGRATE_RECLAIMABLE,        /*可回收页*/
    MIGRATE_MOVABLE,            /*可移动页*/
    MIGRATE_PCPTYPES,           /*CPU 单页缓存链表数量*/
    MIGRATE_RESERVE = MIGRATE_PCPTYPES,    /*预留页链表*/
#ifdef CONFIG_CMA
    MIGRATE_CMA,
#endif
#ifdef CONFIG_MEMORY_ISOLATION
    MIGRATE_ISOLATE,            /* can't allocate from here */
#endif
    MIGRATE_TYPES               /*迁移类型数量，free_list[]数组项数*/
};
```

MIGRATE_TYPES 表示页面的迁移类型，内核对物理内存按页块指定迁移类型，用于防止内存碎片，

详见 3.4 节。

free_area 结构体内双链表数组项数与页迁移类型数量相同，同一阶的空闲内存块按迁移类型进行划分，添加到相同迁移类型的双链表中。free_list[]双链表数组链接的内存块大小是相同的，只不过迁移类型不同。

3.2.4 页

内核用 page 结构体代表页帧，内存管理中对页帧的管理和操作都是通过 page 实例进行的，因此 page 结构体是内存管理的基础，非常的重要。内核在内存初始化过程中，为连续页帧建立 page 实例数组，页帧与 page 实例一一对应，物理上连续的页帧其 page 实例也是连续的，因此在伙伴系统链表中只需要链接首页的 page 实例，由阶数即可确定内存区所有的 page 实例。

在介绍 page 结构体之前，我们先来归纳一下页帧在内核中可能处于的状态（使用状态）：

1、内核空间使用的页，主要包括以下几种情况：

（1）加载内核时被内核代码、数据段占用，伙伴系统不能管理（保留页面）；

（2）被动态分配使用，映射到直接映射区（普通内存域），如：用于表示进程结构实例等；

（3）被映射到间接映射区，内核通过页表访问，如：加载模块时占用的内存；

（4）被 slab 分配器使用。

2、用户空间使用的页，主要包括以下几种情况：

（1）用于文件映射的缓存页，保存的是文件内容；

（2）用于匿名映射，如：映射到进程堆、栈等区域；

（3）被页缓存/块缓存使用，用于缓存文件内容或块设备信息；

（4）位于交换缓存内，等待回写。

3、空闲页，由伙伴系统管理。

某一页帧只能处于以上状态之一，不能同时处于多种状态，因此 page 结构体中定义了大量的联合体，复用其中的成员。使用联合体的主要目的是为了减小数据结构大小，因为当系统物理内存较大时，page 实例数组占用的内存空间会比较大。

page 结构体定义在/include/linux/mm_types.h 头文件：

```
struct page {
    /*第一个双字*/
    unsigned long    flags;          /*第一个字，页标记，/include/linux/page-flages.h*/
    union {                          /*第二个字，联合体，指向地址空间或 slab 中第一个对象*/
      struct address_space *mapping;     /*映射页，指向 address_space 或 NULL，或 anon_vma*/
      void *s_mem;                       /*slab 首页，指向第一个对象地址*/
     };

    /*第二个双字*/
    struct {                             /*第二个双字，结构体开始*/
        union {                          /*第一个字，联合体，映射页索引或 freelist 指针*/
            pgoff_t index;       /*映射页，表示页在文件内偏移量或虚拟内存域偏移量*/
            void *freelist;      /*slab 首页指向 freelist 数组起始地址*/
        };

        union {                          /*第二个字，联合体*/
          #if defined(CONFIG_HAVE_CMPXCHG_DOUBLE)   \
```

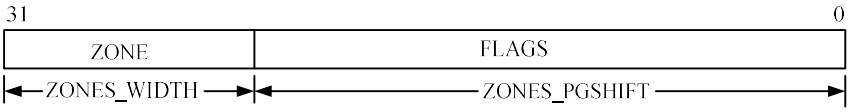
```
                                    &&defined(CONFIG_HAVE_ALIGNED_STRUCT_PAGE)
            unsigned long counters;
        #else
            unsigned counters;          /*联合体第一个成员*/
        #endif

        struct {                        /*联合体第二个成员，结构体开始*/
            union {
                atomic_t    _mapcount;      /*映射计数*/
                struct {            /* SLUB */
                    unsigned inuse:16;
                    unsigned objects:15;
                    unsigned frozen:1;
                };
                int units;      /* SLOB */
            };
            atomic_t    _count;         /*使用计数*/
        };    /*结构体结束*/

        unsigned int active;        /*联合体第三个成员，slab 中已使用对象数目*/
    };  /*第二个双字第二个字结束，联合体结束*/

};      /*第二个双字结束，结构体结束*/

/*第三个双字，联合体开始*/
union {
    struct list_head    lru;    /*联合体第一个成员，双链表成员，将 page 链接到双链表*/

    struct {                    /*联合体第二个成员，用于 slub 分配器*/
        struct page *next;
     #ifdef CONFIG_64BIT
        int pages;
        int pobjects;
     #else
        short int   pages;
        short int   pobjects;
     #endif
    };

    struct slab *slab_page;         /*联合体第三个成员*/
    struct rcu_head    rcu_head;    /*联合体第四个成员，用于 slab 分配器通过 RCU 注销 slab*/

    struct {                        /*联合体第五个成员，用于复合页，创建 slab 时会分配复合页*/
```

```
            compound_page_dtor *compound_dtor;     /*复合页释放页函数指针，page[1]*/
            unsigned long compound_order;          /*复合页阶数*/
        };

    #if defined(CONFIG_TRANSPARENT_HUGEPAGE) && USE_SPLIT_PMD_PTLOCKS
            pgtable_t pmd_huge_pte;       /*联合体第六个成员*/
    #endif
    };      /*第三个双字结束，联合体结束*/

    /*结构体剩余部分*/
    union {                               /*第七个字，联合体开始*/
        unsigned long private;            /*页帧私有数据*/
    #if USE_SPLIT_PTE_PTLOCKS
        #if ALLOC_SPLIT_PTLOCKS
            spinlock_t *ptl;
        #else
            spinlock_t ptl;
        #endif
    #endif
        struct kmem_cache    *slab_cache;        /*slab 首页指向 kmem_cache 实例*/
        struct page *first_page;                 /*复合页尾页，指向复合页首页*/
    };          /*第七个字结束，联合体结束*/

    #ifdef CONFIG_MEMCG          /*内存控制组*/
        struct mem_cgroup    *mem_cgroup;        /*结构体第八个字*/
    #endif

    #if defined(WANT_PAGE_VIRTUAL)
        void    *virtual;                        /*结构体第九个字*/
    #endif

    #ifdef CONFIG_KMEMCHECK
        void *shadow;                            /*结构体第十个字*/
    #endif

    #ifdef LAST_CPUPID_NOT_IN_PAGE_FLAGS
        int _last_cpupid;                        /*结构体第十一个字*/
    #endif
}       /*page 结构体结束*/
```

page 结构体内大量使用了联合体，联合体中各成员共用存储空间，某一实例只能使用联合体内其中一个成员。因为页处于不同的状态，因此在不同的状态可对联合体中对象有不同的解释。由于分配给内核使用的页帧分配后基本就固定不动了，除非内核主动释放，而分配给进程使用的页帧有较多的操作，因此 page 结构体内各成员主要用于用户空间使用的页。下面将详细介绍 page 结构体中比较常用的几个成员。

1 第一个字

page 结构体第一个字为 flags 标记成员，它不是联合体成员，因此对处于任何状态的页帧都有效。flags 成员是无符号整数类型，标记位定义在/include/linux/page-flags-layout.h 头文件。根据系统配置不同 flags 成员布局有所不同，这里以单结点 UMA 系统为例，flags 成员布局如下：



高位部分表示所处内存域类型，低位部分表示标记。内存域类型位宽在/include/linux/page-flags-layout.h 头内定义，例如：内存域类型数量小于 4 时，宽度为 2 个比特位。内核在/include/linux/mm.h 头文件内定义了 flags 标记成员的各位域的掩码，以及获取位域中数据值的函数，例如：

- set_page_zone(struct page *page, enum zone_type zone)：设置 page 内存域类型。
- int page_zone_id(struct page *page)：读取 page 内存域类型，枚举类型值。
- struct zone *page_zone(const struct page *page)：返回 page 所在内存域 zone 实例指针。

flags 标记成员低端表示页标记的位域中各比特位含义定义在/include/linux/page-flages.h 头文件：

```
enum pageflags {
    PG_locked,          /*置位表示页被锁定，不可操作，如正在回写或从外部读数据的页*/
    PG_error,           /*置位表示基于页的 IO 操作产生错误*/
    PG_referenced,      /*用于页回收机制，置位表示页被引用*/
    PG_uptodate,        /*置位表示页帧数据有效，页帧数据与块设备中数据同步*/
    PG_dirty,           /*脏标记，置位表示页中数据与块设备不同步，需要回写*/
    PG_lru,             /*页位于内存域可回收页 LRU 链表中*/
    PG_active,          /*活跃页*/
    PG_slab,            /*置位表示页帧被 slab（slub 或 slob）缓存使用*/
    PG_owner_priv_1,    /* Owner use. If pagecache, fs may use*/
    PG_arch_1,
    PG_reserved,        /*某些特殊页置位，表示不可交换出外部块设备*/
    PG_private,         /*置位表示 page 结构 private 成员不为空*/
    PG_private_2,       /* If pagecache, has fs aux data */
    PG_writeback,       /*页正在回写（缓存页），写出至块设备*/
#ifdef CONFIG_PAGEFLAGS_EXTENDED
    PG_head,            /*复合页首页*/
    PG_tail,            /*复合页尾页*/
#else
    PG_compound,        /*置位表示复合页，页组合成巨型页*/
#endif
    PG_swapcache,       /*页位于交换缓存中，表示位槽的 swp_entry_t 实例保存在 private 成员内*/
    PG_mappedtodisk,    /* Has blocks allocated on-disk */
    PG_reclaim,         /*正在回收页，页处于文件页缓存，具有后备存储设备，预读标记*/
    PG_swapbacked,
                /*可交换到交换区的页（匿名映射页或内存文件系统中页），非块设备中文件缓存页*/
    PG_unevictable,         /*页位于不可回收页 LRU 链表*/
```

```
#ifdef CONFIG_MMU
        PG_mlocked,             /*页帧映射到进程空间，置位表示页被进程锁定，不可回收*/
#endif
#ifdef CONFIG_ARCH_USES_PG_UNCACHED
        PG_uncached,            /*页被映射且是非缓存访问模式*/
#endif
#ifdef CONFIG_MEMORY_FAILURE
        PG_hwpoison,        /* hardware poisoned page. Don't touch */
#endif
#ifdef CONFIG_TRANSPARENT_HUGEPAGE
        PG_compound_lock,
#endif
        __NR_PAGEFLAGS,         /*有效标记位数量*/

        PG_checked = PG_owner_priv_1,       /*标记位别名*/
        PG_fscache = PG_private_2,          /* page backed by cache */
        PG_pinned = PG_owner_priv_1,
        PG_savepinned = PG_dirty,
        PG_foreign = PG_owner_priv_1,
        PG_slob_free = PG_private,
    };
```

内核代码中不能直接对标记位进行操作，而应该调用内核提供的形如 SetPageXXX(struct page *page)和 ClearPageXXX(struct page *page)的函数设置和清除标记位，以保证操作的原子性。PageXXX(const struct page *page)函数用于测试标记位的值，这些函数都定义在/include/linux/page-flages.h 头文件。

例如：SetPageUptodate(struct page *page)函数用于设置 PG_uptodate 标记位，ClearPageUptodate(struct page *page)函数用于清除 PG_uptodate 标记位。

2 第二个字

page 结构体内第二个字为联合体，定义如下：

```
union {
    struct address_space    *mapping;       /*映射页，指向地址空间*/
    void    *s_mem;                         /*被 slab 缓存使用时，指向 slab 第一个对象*/
};
```

如果页帧用于文件映射或匿名映射，联合体解释为 mapping 成员，用于文件映射时指向 address_space 地址空间实例或为 NULL；用于匿名映射时指向 anon_vma 结构体实例。

mapping 成员 bit[0]用于区分指向的是 address_space 还是 anon_vma 结构体实例，为 0 表示指向的是地址空间 address_space 实例，为 1 表示指向的是匿名映射 anon_vma 实例。

函数 int PageAnon(struct page *page)（/include/linux/page-flages.h）用于检测 bit[0]位是否为 1，是则返回真表示页帧为匿名映射页，否则表示不是匿名映射页。

如果页帧被 slab 缓存使用，第二个字解释为 s_mem 成员，如果是首页，则 s_mem 指向 slab 中第一个对象。

3 第三个字

page 结构体第三个字是一个联合体，定义如下：

```
union {                    /*第二个双字中第一个字，映射页索引值或 freelist 指针*/
    pgoff_t index;      /*映射页，表示页内容在文件内偏移量或虚拟内存域偏移量*/
    void *freelist;     /*slab 首页，保存 freelist 数组基地址*/
};
```

此联合体也是根据页帧是映射页还是被 slab 缓存使用有不同的解释。如果是文件映射页，则 index 表示页帧保存的内容在文件内的偏移量（基于文件开始的页偏移），如果是匿名映射页，则 index 表示映射页在虚拟内存域（vm_area_struct）内的偏移量。如果 page 为伙伴系统单页（含 CPU 单页缓存）index 表示页帧迁移属性。

如果页帧是由 slab 缓存使用，联合体解释为 freelist，如果 page 是首页，则 freelist 指向 slab 中 freelist 数组，详见 3.7 节。

4 第四个字

page 结构体第四个字也是一个联合体，联合体可解释成三个成员中的一个，主要用于计数，定义如下：

```
union {
        #if defined(CONFIG_HAVE_CMPXCHG_DOUBLE)  \
                                    &&defined(CONFIG_HAVE_ALIGNED_STRUCT_PAGE)
          unsigned long counters;
        #else
          unsigned counters;          /*联合体第一个成员*/
        #endif

          struct {                              /*联合体第二个成员*/
              union {
                  atomic_t  _mapcount;      /*页映射计数（映射到用户进程地址空间）*/
                  struct {            /* SLUB */
                      unsigned inuse:16;
                      unsigned objects:15;
                      unsigned frozen:1;
                  };
                  int units;        /* SLOB */
              };       /*结构体第一个成员*/
              atomic_t  _count;           /*结构体第二个成员，使用计数*/
          };                    /*联合体第二个成员结束*/

          unsigned int active;        /*联合体第三个成员，SLAB*/
};
```

_mapcount 表示页帧被映射的计数，即有多少个页表项指向它，_count 为页帧使用计数。页帧用于 slab 分配器时，首页 page 结构 active 成员表示 slab 中已使用对象的数目。

5 第三个双字

page 结构体第三个双字是一个联合体，联合体中包含六个成员，定义如下：

```
union {
    struct list_head    lru;   /*联合体第一个成员，双链表成员，用于将 page 链接到各种链表中*/

    struct {                        /*联合体第二个成员，用于 slub 分配器*/
        struct page *next;
    #ifdef CONFIG_64BIT
        int pages;              /* Nr of partial slabs left */
        int pobjects;           /* Approximate # of objects */
    #else
        short int    pages;       /*第三个双字中第二个字，缓存页帧数量及数据结构实例数量*/
        short int    pobjects;
    #endif
    };

    struct slab    *slab_page;         /*联合体第三个成员*/
    struct rcu_head    rcu_head;    /*联合体第四个成员，由 slab 分配器使用*/
    struct {                        /*联合体第五个成员，创建复合页时使用*/
        compound_page_dtor *compound_dtor;
        unsigned long compound_order;
    };

    #if defined(CONFIG_TRANSPARENT_HUGEPAGE) && USE_SPLIT_PMD_PTLOCKS
        pgtable_t pmd_huge_pte;       /*联合体第六个成员，protected by page->ptl */
    #endif
};
```

页帧处于空闲状态或分配给进程使用时，解释为 lru 成员，用于将页帧加入到伙伴链表或 LRU 链表。页帧用于 slab 分配器时，rcu_head 成员在注销 slab 时使用。创建复合页时联合体解释为第五个成员。

6 第七个字

page 结构体剩余部分主要与配置相关，这里主要看一下结构体第七个字的定义，它是一个联合体：

```
union {                     /*page 结构体第七个字*/
        unsigned long    private;         /*页帧私有数据*/
    #if USE_SPLIT_PTE_PTLOCKS
      #if ALLOC_SPLIT_PTLOCKS
            spinlock_t *ptl;
    #else
            spinlock_t ptl;
    #endif
  #endif
       struct kmem_cache    *slab_cache;       /*首页指向 kmem_cache 实例*/
```

```
        struct page *first_page;      /*如果是复合页尾页，指向复合页首页*/
};              /*page 结构体第七个字结束*/
```

private 成员表示页帧私有数据，页帧处于不同的状态时表示的信息不同，例如，页帧为伙伴系统空闲内存区首页时，private 表示阶数。若 private 不为空需要设置页标记位 PG_private。

如果页帧用于 slab 分配器且是首页，则第七个字解释为 slab_cache 成员，指向缓存描述符 kmem_cache 结构体实例。

如果页帧是复合页的尾页，则 first_page 指向复合页的首页 page 实例。复合页是内存区的一种附加属性，并不是一种新的内存区。

3.3 自举分配器

内核在启动阶段从命令行参数（或环境变量）中获取系统物理内存信息。在初始化内存管理数据结构之前，内核需要初始化自举分配器。自举分配器用于启动初期分配/释放内存，此时伙伴系统尚未启用。在启动后期内核弃用自举分配器，转用伙伴系统，由伙伴系统管理物理内存。自举分配器只能用于从低端内存中分配页帧，内核通过直接映射访问分配的页。

自举分配器在获取物理内存信息后，依然是将物理内存（低端内存）按页进行划分，并建立页帧位图，位图中的位代表相应页帧的使用情况，比特位清 0 表示页帧空闲可用，置 1 表示页帧已被使用。分配页帧时，查找位图找到标记为 0 的位对应的页帧，返回给调用函数，释放页帧只需将相应位图中的位清零即可。

内核启动函数 start_kernel()在体系结构相关的 setup_arch()函数内调用 arch_mem_init()函数，完成获取物理内存信息、初始化自举分配器及初始化物理内存管理数据结构等工作。本节介绍内核获取物理内存信息及初始化自举分配器的函数，物理内存管理数据结构的初始化函数在下一节再做介绍。

3.3.1 获取内存信息

处理器启动后运行引导加载程序，由其将物理内存信息传递给内核。比较常见的方法是以命令行参数的形式传递，参数为“mem=*size*”格式。内核在/arch/mips/kernel/setup.c 文件内定义了此参数的处理函数 early_parse_mem(char *p)。参数处理函数调用 add_memory_region()函数向内核添加物理内存段信息。

龙芯 1B 处理器引导加载程序（PMON）并没有采用命令行参数的形式来传递物理内存信息，而是通过环境变量 memsize 和 highmemsize（单位为 MB，只写数值）来传递内存和高端内存信息。龙芯 1B 平台相关代码在/arch/mips/loongson32/common/prom.c 文件内定义了全局变量 memsize 和 highmemsize，分别用于保存物理内存和高端内存大小。体系结构相关的 setup_arch()函数调用平台相关的 prom_init()函数从环境变量 memsize 和 highmemsize 中获取物理内存和高端内存的大小，赋予全局变量 memsize 和 highmemsize，若未定义环境变量则 memsize 赋予默认值（60MB），highmemsize 赋值 0。

内核在/arch/mips/include/asm/bootinfo.h 头文件内定义了 boot_mem_map 结构体用于保存物理内存信息，显然这是一个体系结构相关的结构体：

```
struct boot_mem_map {
    int nr_map;                         /*当前物理内存段数量*/
    struct boot_mem_map_entry {
        phys_addr_t   addr;             /*内存段起始地址 */
        phys_addr_t   size;             /*内存段大小 */
        long   type;                    /*内存段类型 */
    } map[BOOT_MEM_MAP_MAX];   /*保存内存段信息数组*/
};
```

内核在/arch/mips/kernel/setup.c 内定义了 boot_mem_map 结构体同名的结构体实例：

```
struct boot_mem_map    boot_mem_map;
```

bootinfo.h 头文件内定义了下列宏表示 boot_mem_map 结构体记录内存段的最大数量及内存段类型：

```
#define   BOOT_MEM_MAP_MAX  32      /*map[]数组最大可保存 32 段物理内存信息*/
#define   BOOT_MEM_RAM          1      /* RAM 内存段*/
#define   BOOT_MEM_ROM_DATA  2      /*只读内存段*/
#define   BOOT_MEM_RESERVED   3      /*保留内存段*/
#define   BOOT_MEM_INIT_RAM    4      /*初始化 RAM 段，保存内核初始化数据，后期释放*/
```

内核在/arch/mips/kernel/setup.c 文件内定义了 add_memory_region()函数，用于将内存段信息添加到 boot_mem_map 结构体实例中，函数定义如下：

```
void __init add_memory_region(phys_addr_t start, phys_addr_t size, long type)
/*start：起始物理地址，size：内存段大小，type：内存段类型*/
{
    int x = boot_mem_map.nr_map;
    int i;

    if (start + size < start) {       /*地址合法性检查*/
        pr_warn("Trying to add an invalid memory region, skipped\n");
        return;
    }
    /*添加内存段前判断是否可与现有内存段合并*/
    for (i = 0; i < boot_mem_map.nr_map; i++) {
        struct boot_mem_map_entry *entry = boot_mem_map.map + i;
        unsigned long top;

        if (entry->type != type)        /*必须内存段类型相同才能合并*/
            continue;
        if (start + size < entry->addr)
            continue;                   /*内存段没有重叠*/
        if (entry->addr + entry->size < start)
            continue;                   /*内存段没有重叠*/
        top = max(entry->addr + entry->size, start + size);      /*合并后内存段结束地址*/
        entry->addr = min(entry->addr, start);                   /*合并后内存段开始地址*/
        entry->size = top - entry->addr;                         /*合并后内存段大小*/
        return;
    }
    /*不能与现存内存段合并则填充新数组项*/
    if (boot_mem_map.nr_map == BOOT_MEM_MAP_MAX) {   /*内存段数量超过最大值，返回*/
        pr_err("Ooops! Too many entries in the memory map!\n");
        return;
    }
```

```
        boot_mem_map.map[x].addr = start;       /*填充新数组项*/
        boot_mem_map.map[x].size = size;
        boot_mem_map.map[x].type = type;
        boot_mem_map.nr_map++;
    }
```

添加内存段函数检查内存段是否能与 boot_mem_map 实例内已有内存段合并，能则合并，不能合并则填充新数组项。

平台相关的 plat_mem_setup()函数（/arch/mips/loongson32/common/setup.c）负责向 boot_mem_map 实例添加内存段信息，这里只添加了一个内存段信息：

```
    void __init plat_mem_setup(void)
    {
        add_memory_region(0x0, (memsize << 20), BOOT_MEM_RAM); /*BOOT_MEM_RAM 类型*/
    }
```

3.3.2 体系结构初始化

在体系结构相关的 arch_mem_init()函数中完成主要的物理内存管理初始化工作，对于 MIPS 架构此函数定义在/arch/mips/kernel/setup.c 文件内，函数调用关系为 start_kernel()->setup_arch()->arch_mem_init()。

```
    static void __init arch_mem_init(char **cmdline_p)
    {
        struct memblock_region *reg;
        extern void plat_mem_setup(void);

        plat_mem_setup(); /*向 boot_mem_map 实例添加内存段信息，/arch/mips/loong32/common/setup.c*/

        arch_mem_addpart(PFN_DOWN(__pa_symbol(&_text)) << PAGE_SHIFT,
                    PFN_UP(__pa_symbol(&_edata)) << PAGE_SHIFT, BOOT_MEM_RAM);
        arch_mem_addpart(PFN_UP(__pa_symbol(&__init_begin)) << PAGE_SHIFT,
                    PFN_DOWN(__pa_symbol(&__init_end)) << PAGE_SHIFT, BOOT_MEM_INIT_RAM);
            /*确保内核代码、数据段内存被包含在 boot_mem_map 实例内存段中，/arch/mips/kernel/setup.c*/

        pr_info("Determined physical RAM map:\n");
        print_memory_map();
        ...                 /*复制命令行参数*/
        *cmdline_p = command_line;          /*指向命令行参数*/
        parse_early_param();                /*处理早期命令行参数，见 2.5 节*/
        ...
        bootmem_init();         /*初始化自举分配器，/arch/mips/kernel/setup.c*/
    #ifdef CONFIG_PROC_VMCORE
        ...
    #endif
        mips_parse_crashkernel();   /*没有配置 KEXEC 时为空操作，/arch/mips/kernel/setup.c*/
    #ifdef CONFIG_KEXEC     /*是否支持 kexec()系统调用，用于关闭内核，并启动另一个内核*/
```

```
    if (crashk_res.start != crashk_res.end)
        reserve_bootmem(crashk_res.start,crashk_res.end - crashk_res.start + 1,BOOTMEM_DEFAULT);
#endif
    device_tree_init();          /*加载设备树目标文件，平台（板级）实现，否则为空*/
    sparse_init();               /*稀疏内存初始化，FLATMEM 系统为空操作*/
    plat_swiotlb_setup();        /*板级实现，龙芯 1B 为空操作*/
    paging_init();               /*物理内存管理数据结构初始化，下一节再做介绍，/arch/mips/mm/init.c */

    dma_contiguous_reserve(PFN_PHYS(max_low_pfn));
                /*需配置 DMA_CMA 选项，否则为空操作，/include/linux/dma-contiguous.h(L130)*/

    for_each_memblock(reserved, reg)
        if (reg->size != 0)
            reserve_bootmem(reg->base, reg->size, BOOTMEM_DEFAULT);
                /*将 memblock 实例中保留类型的物理内存，在自举分配器中标记为已使用*/
}
```

arch_mem_init()函数调用 plat_mem_setup()函数向 boot_mem_map 实例添加内存段信息，并需要确保存放内核代码和数据段的内存段被包含在 boot_mem_map 实例的内存段内。然后，调用 bootmem_init()函数初始化自举分配器，调用 paging_init()函数初始化物理内存管理数据结构实例。

这里再介绍一下内核目标文件的布局。内核源代码链接文件位于/arch/mips/kernel/vmlinux.lds 内：

```
ENTRY(kernel_entry)          /*内核入口地址*/
...
SECTIONS
{
#ifdef CONFIG_BOOT_ELF64
    ...
#endif
    . = VMLINUX_LOAD_ADDRESS;      /*内核起始虚拟地址*/
    _text = .;       /*代码段起始地址*/
    ...
    _etext = .;      /*代码段结束地址*/
    ...
    _sdata = .;      /*数据段起始地址*/
    ...
    _edata = .;      /*数据段结束地址*/

    . = ALIGN(PAGE_SIZE);
    __init_begin = .;     /*初始化段起始地址，页对齐，内核初始化完成后释放此段内存*/

    ...

    . = ALIGN(0x10000);
```

```
    __init_end = .;          /*初始化段结束地址*/

    BSS_SECTION(0, 0x10000, 0)      /*未初始化数据段*/

    _end = . ;               /*内核结束虚拟地址*/
    ...
}
```

内核目标文件各段起始结束地址的标号请读者留意，后面代码中还将用到。这里需要说明一下的是表示目标文件起始地址的 VMLINUX_LOAD_ADDRESS 宏，定义在/arch/mips/Makefile 文件内：

```
KBUILD_CPPFLAGS += -DVMLINUX_LOAD_ADDRESS=$(load-y)
...
bootvars-y = VMLINUX_LOAD_ADDRESS=$(load-y)   VMLINUX_ENTRY_ADDRESS=$(entry-y)
```

load-y 变量定义在板级文件/arch/mips/loongson32/Platform 内：

```
load-$(CONFIG_LOONGSON1_LS1B)          += 0xffffffff80100000
```

loongson32 处理器内核目标文件的起始虚拟地址（加载地址）为 0x80100000。

3.3.3 初始化自举分配器

本节介绍自举分配器的初始化，另外，内核还定义了 memblock 分配器，它与自举分配器具有相同名称的初始化和接口函数。若需要使用 memblock 分配器，需选择 HAVE_MEMBLOCK 配置选项（MIPS 默认选择），并在体系结构相关配置选项中选择 NO_BOOTMEM 选项（MIPS 默认没有选择）。

若选择了 NO_BOOTMEM 配置选项，内核将编译链接/mm/nobootmem.c 文件（实现 memblock 分配器，自举分配器的替代者），否则编译链接/mm/bootmem.c 文件（实现自举分配器）。MIPS 体系结构没有选择 NO_BOOTMEM 选项，因此本书暂时只介绍自举分配器。

1 数据结构

自举分配器通过 bootmem_data_t 结构体实现，物理内存每个结点对应一个 bootmem_data_t 结构体实例。bootmem_data_t 结构体定义在/include/linux/bootmem.h 头文件中：

```
#ifndef CONFIG_NO_BOOTMEM
  typedef struct bootmem_data {
    unsigned long     node_min_pfn;          /*结点最小页帧号*/
    unsigned long     node_low_pfn;          /*结点位于低端内存最大页帧号*/
    void *            node_bootmem_map;      /*位图指针，含空洞*/
    unsigned long     last_end_off;          /*最近一次分配的页帧编号*/
    unsigned long     hint_idx;
    struct list_head  list;          /*双链表节点，将 bootmem_data 实例添加到全局 bdata_list 双链表*/
  } bootmem_data_t;

  extern bootmem_data_t   bootmem_node_data[];   /*bootmem_data 实例数组，数组项数为结点数*/
#endif
```

bootmem_data 结构体主要成员简介如下：

●node_min_pfn：结点起始页帧号。

●node_low_pfn：结点在普通内存域内的最大页帧号，自举分配器只管理普通内存域内存（低端内存）。

●node_bootmem_map：指向物理页帧位图。

●last_end_off：上一次分配的页帧编号。

●list：双链表节点，将 bootmem_data_t 实例添加到全局 bdata_list 双链表。

内核在/mm/bootmem.c 文件内定义了静态 bootmem_data_t 实例数组，数组项数为内存结点数：

```
bootmem_data_t    bootmem_node_data[MAX_NUMNODES]    __initdata;
```

内核在/mm/bootmem.c 中定义了全局双链表用于管理 bootmem_data_t 实例：

```
static struct list_head    bdata_list    __initdata = LIST_HEAD_INIT(bdata_list);
```

内存结点数据结构 pg_data_t 内包含指向 bootmem_data_t 实例的指针成员 bdata。若为单结点系统，在/mm/bootmem.c 文件内定义了唯一的结点 pg_data_t 结构体实例 contig_page_data：

```
#ifndef CONFIG_NEED_MULTIPLE_NODES
   struct pglist_data __refdata    contig_page_data = {
         .bdata = &bootmem_node_data[0]          /*指向自举分配器结构实例，数组第一个成员*/
   };
   EXPORT_SYMBOL(contig_page_data);
#endif
```

在介绍自举分配器初始化函数之前，先了解几个在/mm/bootmem.c 文件内定义的全局变量：

```
unsigned long    max_low_pfn;        /*低端内存最大页帧号（实际存在的内存）*/
unsigned long    min_low_pfn;        /*低端内存最小页帧号*/
unsigned long    max_pfn;            /*物理内存最大页帧号，含高端内存，即实际内存上边界*/
```

max_low_pfn 表示低端内存最大页帧号，min_low_pfn 表示低端内存最小页帧号，max_pfn 表示最大页帧号。当物理内存中不存在高端内存时，max_pfn 与 max_low_pfn 相同。

内核在/arch/mips/mm/highmem.c 文件内定义了全局变量用于保存高端内存的起始和结束页帧号：

```
unsigned long highstart_pfn,    highend_pfn;        /*初始值为 0*/
```

在/include/linux/pfn.h 头文件中定义了下列宏，其中 x 表示物理地址：

```
#define    PFN_ALIGN(x)        (((unsigned long)(x) + (PAGE_SIZE - 1)) & PAGE_MASK)
#define    PFN_UP(x)           (((x) + PAGE_SIZE-1) >> PAGE_SHIFT)      /*x 上一页帧号*/
#define    PFN_DOWN(x)         ((x) >> PAGE_SHIFT)                      /*包含 x 的页帧号*/
#define    PFN_PHYS(x)         ((phys_addr_t)(x) << PAGE_SHIFT)   /*页帧号转为物理地址*/
```

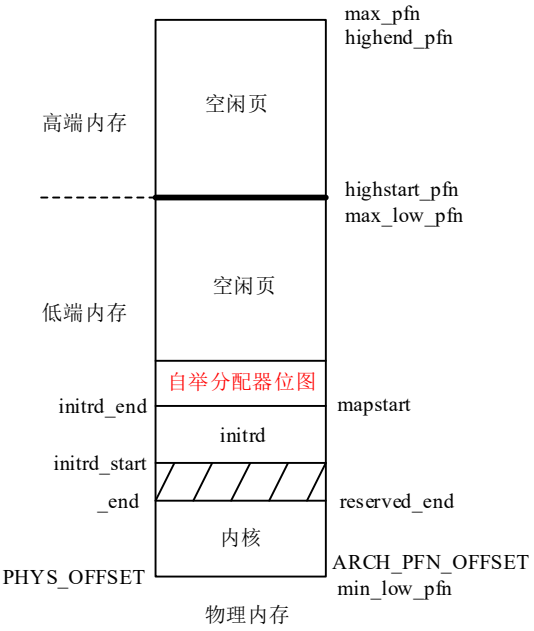
PFN_ALIGN(x)：返回地址 x 上一页帧起始地址（上对齐），若 x 正好是页对齐则返回 x（内存地址）。

PFN_UP(x)：返回地址 x 上一页帧号（上对齐），若 x 为页帧起始地址则返回本页帧号。

PFN_DOWN(x)：返回地址 x 所在页帧的编号，下对齐。

PFN_PHYS(x)：返回 x 页帧的起始物理地址。

自举分配器初始化后，将在内存中创建一个位图，位图标记低端内存页的使用情况，初始化后物理内存布局如下图所示：



物理内存的最底端是内核代码段和数据段，往上是启动阶段内存盘占用的空间，initrd_end、initrd_start 全局变量定义在/init/do_mounts_initrd.c 文件内，表示内存盘起止地址。reserved_end 变量赋值为内核镜像结束地址之后的第一个物理页帧号，_end 是链接器链接的内核数据段结束地址。min_low_pfn、max_low_pfn 分别表示低端内存最小和最大页帧号。max_pfn 表示物理内存最大页帧号。

如果内核配置选择了 HIGHMEM 选项，即支持高端内存，内核可管理和使用高端内存，并且物理内存存在高端内存（大于 512MB），则 highstart_pfn 表示高端内存起始页帧号，highend_pfn 表示高端内存结束页帧号，即物理内存最大页帧号。

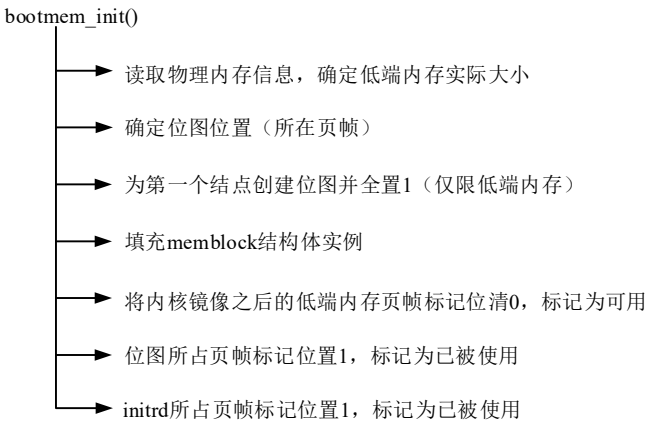
如果物理内存中不存在高端内存（小于 512MB）或没有选择 HIGHMEM 选项，highstart_pfn 和 highend_pfn 保持初始值 0。max_low_pfn 为物理内存最大页帧号（物理内存小于 512MB）或者低端内存最大页帧号（没有选择 HIGHMEM 选项，此时若存在高端内存，高端内存将不被使用）。

mapstart 表示自举分配器位图所在位置的起始页帧号，如果内核启动使用了内存盘则 mapstart 位在 initrd 盘之上，否则位于内核镜像之后。自举分配器位图标记所有低端内存的使用情况，即 min_low_pfn 至 max_low_pfn 的页帧，已使用的标记为 1，未使用的标记为 0。自举分配器不管高端内存，位图中没有高端内存对应的标记位。

2 初始化函数

自举分配器初始化函数最重要的工作就是为内存结点 bootmem_data_t 实例创建位图，并根据内存实际使用情况对位图进行标记，已使用页帧标记为 1，未使用页帧标记为 0。

自举分配器初始化函数 bootmem_init()在/arch/mips/kernel/setup.c 文件内实现，这是一个体系结构相关的函数。根据是否配置了 NUMA 等选项，函数具有不同的实现，这里仅以 UMA（单结点）系统为例介绍其实现，下图简要示意了初始化函数流程：



bootmem_init()函数首先扫描内存段信息，获取实际低端内存的起止页帧号，然后根据内核镜像结束地址以及 initrd 结束地址，确定自举分配器位图所在位置，随后为第一个结点创建位图（初始化全置 1），根据内存段信息填充 memblock 结构体实例，最后清 0 位图之上页帧对应的标记位，表示位图之上页帧可用，自举分配器初始化完成。

自举分配器初始化函数 bootmem_init()定义如下：

```
static void __init bootmem_init(void)
{
    unsigned long   reserved_end;
    unsigned long   mapstart = ~0UL;   /*取最大无符号整数，mapstart 用于记录位图起始页帧号*/
    unsigned long   bootmap_size;      /*位图大小*/
    int i;

    init_initrd();              /*初始化 initrd_end，initrd_start，/arch/mips/kernel/setup.c*/

   reserved_end = (unsigned long) PFN_UP(__pa_symbol(&_end));   /*内核镜像之后第一个页帧号*/
    min_low_pfn = ~0UL;     /*低端内存最小页帧号初始化为最大无符号整数*/
    max_low_pfn = 0;          /*低端内存最大页帧号初始化为 0*/

    /*扫描内存段信息，获取最小和最大页帧号*/
    for (i = 0; i < boot_mem_map.nr_map; i++)
    {
        unsigned long start, end;

        if (boot_mem_map.map[i].type != BOOT_MEM_RAM)   /*非BOOT_MEM_RAM类型则跳过*/
            continue;

        start = PFN_UP(boot_mem_map.map[i].addr);        /*内存段起始页帧号，上对齐*/
        end = PFN_DOWN(boot_mem_map.map[i].addr+ boot_mem_map.map[i].size);
                                                         /*内存段结束页帧号，下对齐*/
        if (end > max_low_pfn)
            max_low_pfn = end;
        if (start < min_low_pfn)
            min_low_pfn = start;                    /*修改低端内存最小、最大页帧号*/
```

```
        if (end <= reserved_end)
            continue;
    #ifdef CONFIG_BLK_DEV_INITRD        /*内存段在 initrd 下方，跳出，扫描下一个内存段*/
        if (initrd_end && end <= (unsigned long)PFN_UP(__pa(initrd_end)))
            continue;
    #endif
        if (start >= mapstart)
            continue;
        mapstart = max(reserved_end, start);   /*确定位图所在位置起始页帧号*/
}
...       /*若数据无效，输出错误信息*/
min_low_pfn = ARCH_PFN_OFFSET;
                    /*体系结构定义的最小页帧号，通常为 0，/arch/mips/include/asm/page.h*/

max_pfn = max_low_pfn;                      /*max_pfn 保存实际内存最大页帧号*/
if (max_low_pfn > PFN_DOWN(HIGHMEM_START))   /*存在高端内存（大于 0x20000000）*/
/*HIGHMEM_START 表示高端内存起始地址，/arch/mips/inlcude/asm/mach-generic/space.h*/
{
 #ifdef   CONFIG_HIGHMEM                                    /*配置支持高端内存*/
        highstart_pfn = PFN_DOWN(HIGHMEM_START);     /*设置高端内存起始页帧号*/
        highend_pfn = max_low_pfn;                          /*设置高端内存结束页帧号*/
 #endif
        max_low_pfn = PFN_DOWN(HIGHMEM_START);
                                    /*max_low_pfn 设为高端内存起始页帧号，低端内存结束页帧号*/
}
#ifdef CONFIG_BLK_DEV_INITRD
   if (initrd_end)
            mapstart = max(mapstart, (unsigned long)PFN_UP(__pa(initrd_end)));
                                                        /*mapstart 需位于 initrd 之上*/
 #endif

/*初始化自举分配器位图，标记 min_low_pfn 至 max_low_pfn 页帧（低端内存）已使用，*/
/*位图从 max_pfn 页帧开始，初始值全标记为 1，bootmem_data_t 实例添加到全局双链表，
/*设置 bootmem_data_t 实例，函数返回位图所占空间字节数。*/
bootmap_size = init_bootmem_node(NODE_DATA(0), mapstart,min_low_pfn, max_low_pfn);

for (i = 0; i < boot_mem_map.nr_map; i++) {        /*扫描内存段，填充 memblock 结构体实例*/
        unsigned long start, end;

        start = PFN_UP(boot_mem_map.map[i].addr);
        end = PFN_DOWN(boot_mem_map.map[i].addr+ boot_mem_map.map[i].size);

        if (start <= min_low_pfn)
```

```
            start = min_low_pfn;
        if (start >= end)
            continue;

#ifndef CONFIG_HIGHMEM
        if (end > max_low_pfn)
            end = max_low_pfn;
        if (end <= start)
            continue;
#endif
        memblock_add_node(PFN_PHYS(start), PFN_PHYS(end - start), 0);
                        /*将内存段信息保存到 memblock 结构体实例，见下文，/mm/memblock.c*/
    }

    /*扫描内存段，标记内核镜像之后的页帧可用，标记位清 0*/
    for (i = 0; i < boot_mem_map.nr_map; i++) {
        unsigned long start, end, size;

        start = PFN_UP(boot_mem_map.map[i].addr);
        end = PFN_DOWN(boot_mem_map.map[i].addr+ boot_mem_map.map[i].size);

        switch (boot_mem_map.map[i].type) {
        case BOOT_MEM_RAM:      /*默认的内存类型*/
            break;
        case BOOT_MEM_INIT_RAM:
            memory_present(0, start, end);        /*/include/linux/mmzone.h*/
                                /*需选择 HAVE_MEMORY_PRESENT 选项，否则为空操作*/
            continue;
        default:
            /*不可用内存*/
            continue;
        }

        if (start >= max_low_pfn)
            continue;
        if (start < reserved_end)      /*起始页帧号不小于 reserved_end*/
            start = reserved_end;
        if (end > max_low_pfn)
            end = max_low_pfn;     /*结束页帧号不超过低端内存最大页帧号*/

        if (end <= start)
            continue;
        size = end - start;        /*内存段中空闲页帧数量*/
```

```
        /*标记 start 页帧开始的 size 个页帧可用，标记位清 0*/
        free_bootmem(PFN_PHYS(start), size << PAGE_SHIFT);      /*/mm/bootmem.c*/
        memory_present(0, start, end);
    }

    /*将位图所占页帧标记为 1，已使用，/mm/bootmem.c*/
    reserve_bootmem(PFN_PHYS(mapstart), bootmap_size, BOOTMEM_DEFAULT);

    /*initrd 占用的页帧标记为 1，已使用*/
    finalize_initrd();      /*arch/mips/kernel/setup.c*/
}
```

通过前面的分析，bootmem_init()函数已不难理解了。bootmem_init()函数的主要功能是为第一个结点在内存中分配实际低端内存对应的位图，将内核镜像、initrd 以及位图自身占用的页帧标记位置 1，表示已被使用，将位图之上的低端内存页帧标记位清 0，表示可以使用，自举分配器初始化完成。

下面介绍一下 bootmem_init()函数中调用的初始化位图 init_bootmem_node()函数，以及填充 memblock 结构体实例的 memblock_add_node()函数的实现。

■初始化位图

bootmem_init()函数中调用 init_bootmem_node()函数为第一个结点分配位图，并初始化，函数调用如下：

```
bootmap_size = init_bootmem_node(NODE_DATA(0), mapstart,min_low_pfn, max_low_pfn);
```

init_bootmem_node()函数在/mm/bootmem.c 文件内实现，参数 NODE_DATA(0)是物理内存第一个结点 pg_data_t 实例指针，mapstart 为位图起始页帧号，后两个参数分别为实际低端内存最小和最大页帧号。

init_bootmem_node()函数定义如下：

```
unsigned long __init    init_bootmem_node(pg_data_t *pgdat, unsigned long freepfn,   \
                                          unsigned long startpfn, unsigned long endpfn)
{
    return init_bootmem_core(pgdat->bdata, freepfn, startpfn, endpfn);   /*/mm/bootmem.c*/
}
```

init_bootmem_node()函数直接将工作委托给 init_bootmem_core()函数，函数定义在/mm/bootmem.c 文件内，代码如下：

```
static unsigned long __init init_bootmem_core(bootmem_data_t *bdata,   \
                                    unsigned long mapstart, unsigned long start, unsigned long end)
{
    unsigned long mapsize;

    mminit_validate_memmodel_limits(&start, &end);
        /*配置了 CONFIG_SPARSEMEM 才有定义，否则为空操作，/mm/internal.h*/
    bdata->node_bootmem_map = phys_to_virt(PFN_PHYS(mapstart));     /*位图起始地址*/
```

```
    bdata->node_min_pfn = start;     /*结点位于低端内存的起始页帧号*/
    bdata->node_low_pfn = end;       /*结点位于低端内存的最大页帧号*/
    link_bootmem(bdata);             /*将 bootmem_data_t 实例加入全局双链表，/mm/bootmem.c*/
                                     /*实例在双链表中按起始页帧号从小到大排列*/
    mapsize = bootmap_bytes(end - start);          /*计算位图大小（字节数），/mm/bootmem.c*/
    memset(bdata->node_bootmem_map, 0xff, mapsize); /*位图全标记为 1，标记已使用*/
    ...          /*输出信息*/
    return mapsize;     /*返回位图大小，字节数*/
}
```

init_bootmem_core()函数从 mapstart 表示的页帧起始地址开始为结点具有的低端内存分配位图空间，并初始化为全 1，表示低端内存全部已被使用，返回位图所占空间字节数。

init_bootmem_core()函数中调用的 phys_to_virt(address)函数定义在/arch/mips/include/asm/io.h 头文件，函数返回物理地址 address 在内核直接映射区的虚拟地址：

```
static inline void * phys_to_virt(unsigned long address)
{
    return (void *) (address + PAGE_OFFSET - PHYS_OFFSET);      /*线性映射*/
                                    /*PAGE_OFFSET 为 0x80000000，PHYS_OFFSET 为 0*/
                                    /*/arch/mips/include/asm/mach-generic/spaces.h*/
}
```

■填充 memblock 实例

在特定于体系结构的代码中，内核使用 boot_mem_map 实例保存内存段信息。在体系结构无关的代码中采用 memblock 结构体（逻辑内存块）保存内存段地址信息。

在 bootmem_init()函数中将扫描 boot_mem_map 实例中的内存段信息，调用 memblock_add_node()函数将各内存段信息添加到 memblock 结构体实例，以便体系结构无关的代码获取内存段信息。

逻辑内存块 memblock 结构体定义在/include/linux/memblock.h 头文件内，内核若支持逻辑内存块需选择 HAVE_MEMBLOCK 配置选项，MIPS 默认选择了此选项（/arch/mips/Kconfig）。

```
struct memblock {
    bool    bottom_up;                  /*内存块是向下还是向上生长*/
    phys_addr_t    current_limit;       /*内存块地址最大值限制*/
    struct memblock_type    memory;     /*可正常使用的内存块，内存块类型*/
    struct memblock_type    reserved;   /*保留的内存块*/
  #ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
    struct memblock_type    physmem;
  #endif
};
```

memblock 结构体中包含 memblock_type 结构体成员 memory 和 reserved，表示两种类型的内存块，memblock_type 结构体定义如下（/include/linux/memblock.h）：

```
struct memblock_type {
    unsigned long    cnt;               /*实际内存域数量（内存域数组中的有效项）*/
    unsigned long    max;               /*内存域数组最大项数*/
    phys_addr_t    total_size;          /*所有内存域大小*/
```

```
    struct memblock_region    *regions;     /*指向内存域数组*/
};
```

memblock_type 结构体表示物理内存中某一类型的内存，如可正常使用的内存，需保留的内存。每种类型的内存又划分成内存域（实际的内存段），如果物理内存是连续的就只需一个内存域实例。

内存域 memblock_region 结构体定义如下（/include/linux/memblock.h）：

```
struct memblock_region {
        phys_addr_t    base;          /*起始地址*/
        phys_addr_t    size;          /*大小*/
        unsigned long    flags;       /*内存域标记*/
 #ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP       /*MIPS 默认选择此选项，/arch/mips/kconfig*/
        int nid;                    /*结点编号，平台代码通过指定内存域的结点号，向内核传递结点信息*/
 #endif
};
```

memblock_region 结构体中标记成员取值定义在/include/linux/memblock.h 文件内：

```
enum {
    MEMBLOCK_NONE          = 0x0,    /*没有特殊要求的内存域*/
    MEMBLOCK_HOTPLUG       = 0x1,    /*可热插拔内存域*/
    MEMBLOCK_MIRROR        = 0x2,    /*镜像内存域*/
};
#define   INIT_MEMBLOCK_REGIONS          128      /*正常使用内存内存域最大数量*/
#define   INIT_PHYSMEM_REGIONS           4
```

内核在/mm/memblock.c 文件内静态定义了 memblock_region 实例数组（128 项）：

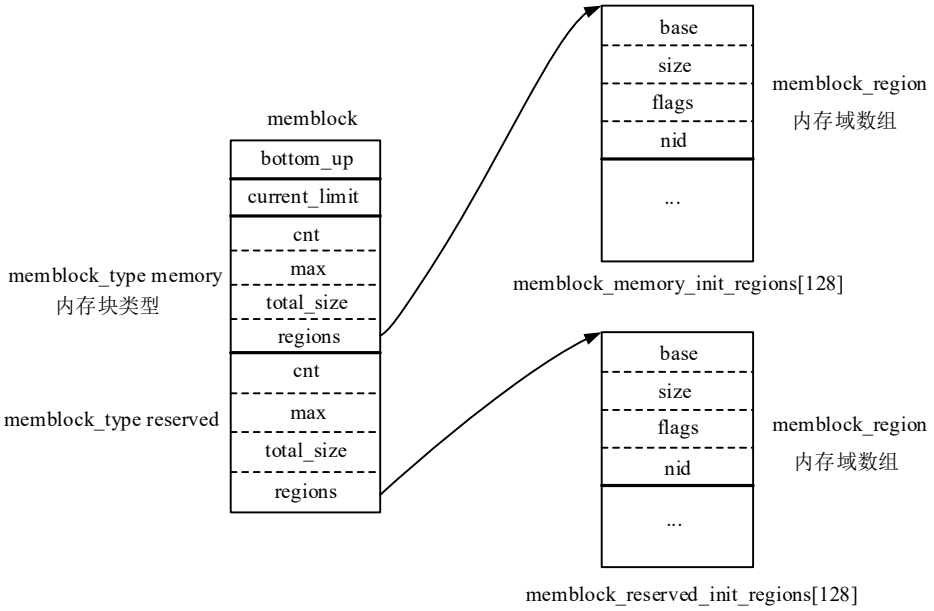
```
static struct memblock_region    memblock_memory_init_regions[INIT_MEMBLOCK_REGIONS]
                                                                __initdata_memblock;
static struct memblock_region    memblock_reserved_init_regions[INIT_MEMBLOCK_REGIONS]
                                                                __initdata_memblock;
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
   static struct memblock_region    memblock_physmem_init_regions[INIT_PHYSMEM_REGIONS]
                                                                __initdata_memblock;
#endif
```

在同一文件内定义了描述物理内存的 memblock 结构体实例 memblock：

```
struct memblock memblock __initdata_memblock = {
        .memory.regions    = memblock_memory_init_regions,     /*指向内存域数组*/
        .memory.cnt        = 1,                                /*当前内存域数量*/
        .memory.max        = INIT_MEMBLOCK_REGIONS,    /*最大限制为 128*/

        .reserved.regions    = memblock_reserved_init_regions,   /*指向内存域数组*/
        .reserved.cnt        = 1,                               /*当前内存域数量*/
        .reserved.max        = INIT_MEMBLOCK_REGIONS,    /*最大限制为 128*/
```

```
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
    .physmem.regions  = memblock_physmem_init_regions,
    .physmem.cnt      = 1, /* empty dummy entry */
    .physmem.max      = INIT_PHYSMEM_REGIONS,
#endif
    .bottom_up        = false,      /*内存块向上生长*/
    .current_limit    = MEMBLOCK_ALLOC_ANYWHERE,
                                    /*处理器可寻址最大值，/include/linux/memblock.h*/
};
```

下图示意了以上各数据结构实例的组织关系：



在 bootmem_init()函数内对每个内存段将调用 memblock_add_node(PFN_PHYS(start), PFN_PHYS(end - start), **0**)函数将物理内存段添加到 memblock 实例中，函数前 2 个参数分别为内存段起始地址和长度，最后一个参数为 0，表示添加内存段都位于第一个结点。

memblock_add_node()函数声明在/include/linux/memblock.h 头文件，在/mm/memblock.c 文件内实现：

```
int __init_memblock memblock_add_node(phys_addr_t base, phys_addr_t size, int nid)
{
    return memblock_add_range(&memblock.memory, base, size, nid, 0);     /*/mm/memblock.c*/
}
```

memblock_add_node()函数调用 memblock_add_range()函数将内存段信息添加到 memory 类型的内存域数组中。注意，内存域数组项按照物理地址从小到大排序。memblock_add_range()函数在/mm/memblock.c 文件内实现：

```
int __init_memblock memblock_add_range(struct memblock_type *type,phys_addr_t base, \
                                       phys_addr_t size,int nid, unsigned long flags)
/*type：内存块类型 memblock_type 实例指针，base：起始地址，size：大小，nid、flags：都为 0*/
{
```

```
    bool    insert = false;
    phys_addr_t    obase = base;
    phys_addr_t    end = base + memblock_cap_size(base, &size);     /*确保 end 不超过寻址最大值*/
    int i, nr_new;

    if (!size)
        return 0;

    if (type->regions[0].size == 0) {           /*内存域数组第一项为空，则直接填充至此数组项*/
        WARN_ON(type->cnt != 1 || type->total_size);
        type->regions[0].base = base;
        type->regions[0].size = size;
        type->regions[0].flags = flags;
        memblock_set_region_node(&type->regions[0], nid);
        type->total_size = size;
        return 0;                           /*填入第一个内存域后直接返回*/
    }
repeat:                                     /*repeat 以下代码要执行两遍*/
    base = obase;
    nr_new = 0;

    for (i = 0; i < type->cnt; i++) {
        struct memblock_region    *rgn = &type->regions[i];
        phys_addr_t    rbase = rgn->base;
        phys_addr_t    rend = rbase + rgn->size;       /*已填充内存域的起始、结束地址*/

        if (rbase >= end)               /*新内存域在当前内存域之下则跳出循环，当前位置为插入点*/
            break;
        if (rend <= base)               /*新内存域在当前内存域之上则跳出本次循环，扫描下一内存域*/
            continue;

        /*以下是处理有重叠的情况*/
        if (rbase > base) {             /*新内存域起始地址在当前内存域之下*/
            nr_new++;                   /*插入操作次数加 1（新内存域数量）*/
            if (insert)
                /*将新内存域位于当前内存域下方的部分作为单独内存域插入数组，
                *插入点后面的数组项下移。*/
                memblock_insert_region(type, i++, base,rbase - base, nid,flags);
        }
        base = min(rend, end);      /*超出当前内存域部分的起始地址*/
    }       /*for 循环结束*/

    /*
```

```
        *地址有重叠时，将新内存域位于当前内存域上方的部分作为单独内存域插入数组；
        *地址没有重叠时，用于插入新内存域。
        */
        if (base < end) {
            nr_new++;
            if (insert)
                memblock_insert_region(type, i, base, end - base, nid, flags);
        }
        if (!insert) {
            while (type->cnt + nr_new > type->max)       /*确定插入次数不超标*/
                if (memblock_double_array(type, obase, size) < 0)
                    return -ENOMEM;
            insert = true;
            goto repeat;    /*再次返回 repeat 处执行*/
        }
      else {      /*执行完插入操作后，执行合并操作*/
            memblock_merge_regions(type);     /*合并内存域*/
            return 0;
        }
    }
```

memblock_insert_region()函数执行内存域插入操作，base 和 size 分别表示新添加内存段的起始地址和大小。

memblock_add_range()函数执行流程：

（1）判断内存域数组第一项是否为空，如果是则直接将新内存段信息填入数组第一项即可返回，如果不为空，继续往下执行。

（2）函数执行两遍扫描现有内存域操作，第一遍用于确定插入操作的次数（填充新内存域的数量），如果新内存段与现有内存域不存在重叠，则插入操作次数（nr_new）为 1。第二遍执行插入操作，内存域按起止地址从小到大，在内存域数组中排列。执行插入操作时，将插入点（数组项）及其之后的内存域数据后移一个数组项，新内存段信息填充至插入点数组项。

（3）如果有重叠，则新内存段超出现有内存域上、下边界的内存都视为新内存段，执行插入操作，而重叠的部分不需要处理。新内存段可能同时超出现有内存域上、下边界，这时需要执行 2 次插入操作，只超出上边界或下边界，只需要执行一次插入操作。

（3）执行完内存段插入操作后，调用 memblock_merge_regions(type)函数合并相邻连续的内存域。

在后续执行的内存管理数据结构初始化函数中，体系结构无关的函数将会从 memblock 结构体实例读取物理内存信息，以初始化结点及内存域实例。

3.3.4 分配与释放函数

自举分配器初始化完成后，内核就可以调用其分配和释放函数分配和释放内存了，下面列出自举分配器提供的接口函数。

分配函数声明在/include/linux/bootmem.h 头文件内，函数实现在/mm/bootmem.c 文件内，主要的有：

```
#define alloc_bootmem(x) \
        __alloc_bootmem(x, SMP_CACHE_BYTES, BOOTMEM_LOW_LIMIT)
            /*从 NORMAL 内存域开始查找分配大小为 x 的内存，SMP_CACHE_BYTES 对齐*/
```

```
#define alloc_bootmem_pages(x) \
        __alloc_bootmem(x, PAGE_SIZE, BOOTMEM_LOW_LIMIT)
                        /*从 NORMAL 内存域开始查找分配大小为 x 的内存，PAGE_SIZE 对齐*/
#define alloc_bootmem_low(x) \
         __alloc_bootmem_low(x, SMP_CACHE_BYTES, 0)
                /*从 DMA 内存域开始查找分配大小为 x 的内存，SMP_CACHE_BYTES 对齐*/

#define alloc_bootmem_low_pages(x) \
        __alloc_bootmem_low(x, PAGE_SIZE, 0)
                        /*从 DMA 内存域开始查找分配大小为 x 的内存，PAGE_SIZE 对齐*/
```

以上分配函数最终都是调用 alloc_bootmem_bdata()函数实现内存的分配。分配函数参数主要是分配内存大小、数据对齐方式及分配内存起始查找的内存域。函数内部都是将内存大小按页对齐，按页进行分配，分配出去的页帧在位图中相应位置 1，返回分配物理内存起始虚拟地址（内核直接映射区地址）。

自举分配器释放内存的函数为：

```
void free_bootmem(unsigned long physaddr, unsigned long size);
```

free_bootmem()函数参数为释放内存起始地址和大小，函数内部将释放内存转换成页帧，清 0 页帧在位图中的相应位即可。实际上内核启动早期调用自举分配器分配的内存一般都不会释放。

自举分配器还定义了 free_all_bootmem(void)函数负责将自举分配器管理的空闲页帧全部释放到伙伴系统，详见 3.6 节。

3.4 数据结构初始化

在体系结构相关的 arch_mem_init()函数完成自举分配器初始化后，将调用 paging_init()函数完成物理内存管理数据结构的初始化工作，这也是体系结构相关的函数，主要是结点、内存域、页及伙伴系统数据结构的初始化。

3.4.1 防止碎片

在介绍内存管理数据结构的初始化之前，先来了解一下内核防止内存碎片的措施。内核启动完成后通过伙伴系统来管理物理内存，在内核长时间运行后，经过不断地分配释放内存，内存中的空闲页帧将会散布在物理内存的各个区域，变的支离破碎，很难获得连续较大块的物理内存。这对用户进程影响不大，因为进程总是通过页表访问物理内存，不过启用了巨型页的系统除外（巨型页由连续的大块内存组成）。内存碎片对内核影响就比较大了，我们知道内核自身的代码和数据，特别是动态分配的数据结构实例，主要位于直接映射区。直接映射区是线性映射到低端内存的，也就是说直接映射区虚拟地址连续的内存对应的物理内存也必须是连续的。因此，内存碎片对内核不利，使内核获取较大内存（连续多页的空闲内存）变得困难。

内核防止碎片的措施主要由设置页帧迁移属性和使用可移动内存域，前者默认编译入内核，后者需要用户启用。

1 页块迁移属性

前面介绍伙伴系统伙伴链表结构时讲到，内核对内存域页帧按页块（最大分配阶）进行划分，内核对

每个页块赋予一个迁移属性，同一页块内的页帧迁移属性相同。迁移属性类型主要分为下面 3 种：

（1）不可移动页（MIGRATE_UNMOVABLE）：页中数据在内存中位置固定不可移动，内核自身使用的页帧属于此类型。因为内核空间直接映射区不经过页表直接线性映射到物理内存，如果页数据移动位置，内核将无法访问。

（2）可回收面（MIGRATE_RECLAIMABLE）：页中保存的数据具有后备存储器，数据丢失后可从后备存储器中重新读取，如：文件地址空间页缓存中的页。

（3）可移动页（MIGRATE_MOVABLE）：页中数据可任意移动位置，如：用户进程地址空间中的页，因为进程总是通过页表访问内存，页中数据移动后，只需修改进程页表项即可。

页迁移属性类型定义/include/linux/mmzone.h 头文件内：

```
enum {
    MIGRATE_UNMOVABLE,          /*不可移动页*/
    MIGRATE_RECLAIMABLE,        /*可回收页*/
    MIGRATE_MOVABLE,            /*可移动页*/
    MIGRATE_PCPTYPES,           /*CPU 单页缓存中页链表数*/
    MIGRATE_RESERVE = MIGRATE_PCPTYPES,    /*预留页*/
#ifdef CONFIG_CMA
    MIGRATE_CMA,                /*CMA 连续内存区*/
#endif
#ifdef CONFIG_MEMORY_ISOLATION
    MIGRATE_ISOLATE,            /* can't allocate from here ，隔离内存，不可分配*/
#endif
    MIGRATE_TYPES               /*迁移类型数量*/
};
```

内存域 zone 结构体中 pageblock_flags 成员指向的整型数组用于标记页块的迁移属性：

```
struct zone {
    ...
    unsigned long   *pageblock_flags;
    ...
}
```

页块中页帧的数量为 pageblock_nr_pages，定义在/include/linux/pageblock-flags.h 头文件，如果没有选择巨型页（HUGETLB_PAGE）配置选项，则定义如下：

```
#define   pageblock_order          (MAX_ORDER-1)
#define   pageblock_nr_pages       (1UL << pageblock_order)
```

MAX_ORDER-1 表示伙伴系统链表中管理的最大空闲内存区的阶数，pageblock_nr_pages 表示的物理页帧数量与伙伴系统管理的最大分配阶内存区所含页帧数相同。

pageblock_flags 成员指向的整型数（实际为数组），用于标记页块的迁移属性。pageblock_flags 指向的整型数组并不是一个整数表示一个页块的迁移属性，而是将其视为位图，用若干个比特位表示一个页块的迁移属性。表示迁移属性的比特位数定义在/include/linux/pageblock-flags.h 头文件：

```
enum pageblock_bits {
    PB_migrate,
    PB_migrate_end = PB_migrate + 3 - 1,      /*需要 3 比特位表示页块迁移属性，bit[0...2]*/
```

```
    PB_migrate_skip,     /* If set the block is skipped by compaction */
    NR_PAGEBLOCK_BITS        /*比特位数，为 4，以保证在一个字中对齐*/
};
```

NR_PAGEBLOCK_BITS 为表示页块迁移属性所占的比特位数，这里为 4，以保证在一个字中对齐。

内核在/mm/page_alloc.c 文件内定义了 set_pageblock_migratetype(struct page *page, int migratetype)函数用于在 pageblock_flags 指向的数组中设置以 page 为首页的页块迁移属性类型为 migratetype。

```
void set_pageblock_migratetype(struct page *page, int migratetype)
{
    if (unlikely(page_group_by_mobility_disabled &&migratetype < MIGRATE_PCPTYPES))
        migratetype = MIGRATE_UNMOVABLE;   /*若停用迁移属性，则设为不可移动*/

    set_pageblock_flags_group(page, (unsigned long)migratetype,PB_migrate, PB_migrate_end);
                                        /*在数组中设置迁移属性，/include/linux/pageblock-flags.h*/
}
```

set_pageblock_migratetype()函数根据全局变量 page_group_by_mobility_disabled 的值（初始值为 0，数据结构初始化时可能设置为 1）确定设置的迁移类型，如果全局变量为非零，表示禁用页迁移属性，则页迁移类型默认设置为不可移动。若全局变量为零，则设置页迁移类型为 migratetype 参数值。

set_pageblock_flags_group()函数实际调用的是 set_pfnblock_flags_mask()（/mm/page_alloc.c）函数，函数内由 page 结构体实例获取所属内存域 zone 实例，由 page 实例计算出页帧物理地址，从而确定所属页块在位图中位置，最后设置迁移属性位域。

get_pageblock_migratetype(page)和 get_pfnblock_migratetype(struct page *page, unsigned long pfn)函数定义在/include/linux/mmzone.h 头文件，用于获取页帧迁移属性类型。

2 可移动内存域

内核防止内存碎片的另一个措施是启用可移动内存域。在内存域类型中定义了 ZONE_MOVABLE 可移动内存域。ZONE_MOVABLE 是一个伪内存域，内核从其它内存域中分配一定数量页帧视归入可移动内存域，伙伴系统分配页帧时可通过分配掩码指定从可移动内存域中分配页帧。

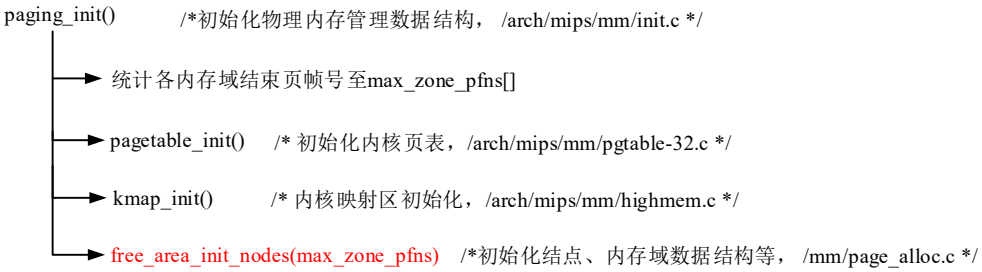
内核在/mm/page_alloc.c 文件内定义了全局变量 required_kernelcore 和 required_movablecore，前者表示不可回收或不可移动页帧数量，由命令行参数 kernelcore 设置。后者表示可回收或可移动页帧数量，由命令行参数 movablecore 设置。这两个参数的处理函数定义在/mm/page_alloc.c 文件内，负责将参数值复制到内核全局变量。

find_zone_movable_pfns_for_nodes()用于计算每个结点添加到 ZONE_MOVABLE 内存域的起止页帧号，并将页帧号填充至 zone_movable_pfn[MAX_NUMNODES]全局数组。

如果全局变量 required_kernelcore 和 required_movablecore 都为 0，则 ZONE_MOVABLE 内存域为空，该机制失效。本书暂不考虑启用可移动内存域的情形，有兴趣的读者可自行阅读相关代码。

3.4.2 初始化函数

物理内存管理数据结构初始化工作主要由 paging_init()函数完成，函数定义在/arch/mips/mm/init.c 文件内，这是一个体系结构相关的函数，函数调用关系如下图所示：



paging_init()函数先根据物理内存信息统计各内存域实际的结束页帧号，保存至 max_zone_pfns[]数组（局部变量），然后调用 pagetable_init()和 kmap_init()函数初始化内核页表及映射区数据结构（第 4 章介绍），最后调用体系结构无关的 free_area_init_nodes(max_zone_pfns)函数初始化结点和内存域数据结构，注意参数为 max_zone_pfns[]数组指针。

paging_init()函数定义在/arch/mips/mm/init.c 文件内：

```
void __init paging_init(void)
{
    unsigned long   max_zone_pfns[MAX_NR_ZONES];      /*各内存域实际的最大页帧号*/
    unsigned long   lastpfn   __maybe_unused;         /*物理内存结束页帧号*/

    pagetable_init();       /*初始化内核页表，第 4 章再做介绍*/

  #ifdef CONFIG_HIGHMEM
    kmap_init();            /*内核固定映射区初始化，第 4 章再做介绍*/
  #endif
  #ifdef CONFIG_ZONE_DMA
    max_zone_pfns[ZONE_DMA] = MAX_DMA_PFN;      /*体系结构定义的内存域结束页帧号*/
  #endif
  #ifdef CONFIG_ZONE_DMA32
    max_zone_pfns[ZONE_DMA32] = MAX_DMA32_PFN;
  #endif

    max_zone_pfns[ZONE_NORMAL] = max_low_pfn;     /*普通内存域最大页帧号*/
    lastpfn = max_low_pfn;                /*下一内存域起始页帧号*/

  #ifdef CONFIG_HIGHMEM                   /*如果支持高端内存*/
    max_zone_pfns[ZONE_HIGHMEM] = highend_pfn;    /*如果实际不存在高端内存，则为 0*/
    lastpfn = highend_pfn;

    if (cpu_has_dc_aliases && max_low_pfn != highend_pfn) {     /*如果处理器不支持高端内存*/
        ...       /*输出信息*/
        max_zone_pfns[ZONE_HIGHMEM] = max_low_pfn;
        lastpfn = max_low_pfn;
    }
  #endif
```

```
    free_area_init_nodes(max_zone_pfns);  /*初始化结点、内存域结构等，/mm/page_alloc.c*/
}
```

paging_init()函数内定义了数组 max_zone_pfn[MAX_NR_ZONE]，用于保存每个内存域的最大页帧号。如果内核配置支持高端内存，但实际物理内存中不存在高端内存，则 max_zone_pfns[ZONE_HIGHMEM]数组项值为 0。

paging_init()函数最后调用体系结构无关的 free_area_init_nodes()函数完成内存管理数据结构的初始化，调用该函数需要选择 HAVE_MEMBLOCK_NODE_MAP 配置选项。MIPS 体系结构在/arch/mips/Kconfig 配置文件内默认选择此选项。

3.4.3 初始化结点

在介绍 free_area_init_nodes()函数前，先看几个定义在/mm/page_alloc.c 文件内的数组变量：

```
static unsigned long __meminitdata   arch_zone_lowest_possible_pfn[MAX_NR_ZONES];
                                                        /*各内存域最小页帧号*/
static unsigned long __meminitdata   arch_zone_highest_possible_pfn[MAX_NR_ZONES];
                                                        /*各内存域最大页帧号*/
static unsigned long __meminitdata   zone_movable_pfn[MAX_NUMNODES];
                                                        /*可移动内存域起始页帧号*/
```

以上应用于体系结构无关的代码，保存内存域最小和最大页帧号，用于后面计算结点各内存域的长度。

free_area_init_nodes(unsigned long *max_zone_pfn)函数在/mm/page_alloc.c 文件内实现，体系结构相关的代码调用此函数前需构建 max_zone_pfn[]数组作为参数，数组项保存各内存域实际结束页帧号，函数代码如下：

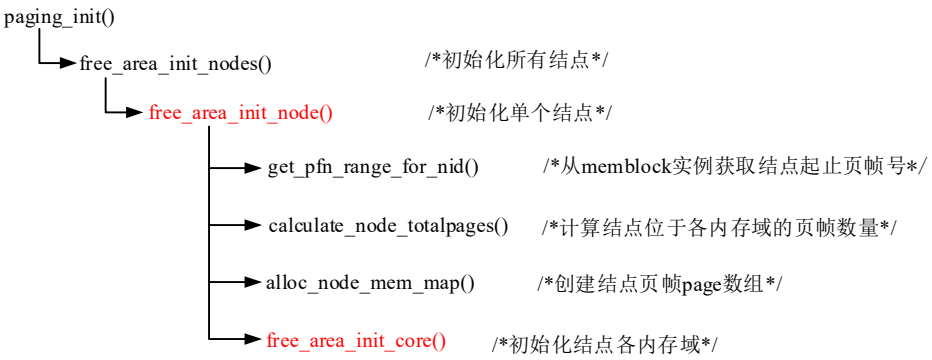
```
void __init free_area_init_nodes(unsigned long *max_zone_pfn)
{
    unsigned long start_pfn, end_pfn;
    int i, nid;
    memset(arch_zone_lowest_possible_pfn, 0,sizeof(arch_zone_lowest_possible_pfn));    /*数组清零*/
    memset(arch_zone_highest_possible_pfn, 0,sizeof(arch_zone_highest_possible_pfn));  /*数组清零*/
    arch_zone_lowest_possible_pfn[0] = find_min_pfn_with_active_regions();
                                /*从 memblock 实例中查找物理内存最小页帧号，/mm/page_alloc.c*/
    arch_zone_highest_possible_pfn[0] = max_zone_pfn[0];       /*第一个内存域结束页帧号*/
    for (i = 1; i < MAX_NR_ZONES; i++)          /*填充数组项*/
    {
        if (i == ZONE_MOVABLE)       /*跳过可移动内存域*/
                continue;
        arch_zone_lowest_possible_pfn[i] =arch_zone_highest_possible_pfn[i-1];
        arch_zone_highest_possible_pfn[i] =max(max_zone_pfn[i], arch_zone_lowest_possible_pfn[i]);
        /*假设不存在高端内存，则高端内存域最低最高页帧号都为最大内存页帧号*/
    }
    arch_zone_lowest_possible_pfn[ZONE_MOVABLE] = 0;       /*可移动内存域起止页帧号赋 0*/
    arch_zone_highest_possible_pfn[ZONE_MOVABLE] = 0;
    memset(zone_movable_pfn, 0, sizeof(zone_movable_pfn));
    find_zone_movable_pfns_for_nodes();    /*设置可移动内存域起始页帧号，/mm/page_alloc.c*/
```

```
    ...             /*输出内存域信息*/
    mminit_verify_pageflags_layout();    /*输出 page 标记成员布局，/mm/mm_init.c*/
    setup_nr_node_ids();                 /*单结点为空操作*/
    for_each_online_node(nid)            /*遍历每个结点，初始化结点数据结构*/
    {
        pg_data_t *pgdat = NODE_DATA(nid);
        free_area_init_node(nid, NULL,find_min_pfn_for_node(nid), NULL);
                /*初始化结点数据结构，结点起始页帧号来自 memblock 实例，/mm/page_alloc.c*/
        if (pgdat->node_present_pages)          /*设置结点状态，单结点系统为空操作*/
            node_set_state(nid, N_MEMORY);   /*/include/linux/nodemask.h*/
        check_for_memory(pgdat, nid);          /*设置结点位图，单结点没有位图*/
    }
}
```

free_area_init_nodes()函数根据 max_zone_pfn[]数组及 memblock 实例信息，填充各内存域起止页帧号数组，arch_zone_lowest_possible_pfn[]和 arch_zone_highest_possible_pfn[]。对于实际不存在的内存域起止页帧号都为物理内存最大页帧号。例如：假设物理内存最大页帧号为 lastpfn，并且位于低端内存，则以上两个数组中表示高端内存域起止页帧号的数组项值都为 lastpfn。

free_area_init_nodes()函数遍历每个内存结点对其调用 free_area_init_node()函数初始化结点数据结构，结点起始页帧号来自 memblock 实例（内存域 memblock_region 中记录了结点号）。

free_area_init_node()函数执行流程简如下图所示：



free_area_init_node()函数定义在/mm/page_alloc.c 文件内，代码如下：

```
void __paginginit free_area_init_node(int nid, unsigned long *zones_size,     \
                                      unsigned long node_start_pfn, unsigned long *zholes_size)
/*nid：结点编号，node_start_pfn：结点起始页帧号，
* *zones_size：保存内存域大小，*zholes_size：保存空洞大小，这里以上 2 个参数都为 NULL。*/
{
    pg_data_t   *pgdat = NODE_DATA(nid);        /*结点 pg_data_t 实例指针*/
    unsigned long   start_pfn = 0;              /*结点起始页帧号*/
    unsigned long   end_pfn = 0;                /*结点结束页帧号*/

    WARN_ON(pgdat->nr_zones || pgdat->classzone_idx);

    /*需配置 DEFERRED_STRUCT_PAGE_INIT 选项，设置结点结构 first_deferred_pfn 成员*/
```

```
    reset_deferred_meminit(pgdat);   /*/mm/page_alloc.c*/

    pgdat->node_id = nid;                    /*结点编号*/
    pgdat->node_start_pfn = node_start_pfn;          /*结点起始页帧号*/
#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
    get_pfn_range_for_nid(nid, &start_pfn, &end_pfn);
                                  /*从 memblock 实例获取结点起始/结束页帧号，/mm/page_alloc.c*/
    ...                           /*输出信息*/
#endif
    calculate_node_totalpages(pgdat, start_pfn, end_pfn, zones_size, zholes_size);
                          /*计算结点各内存域真实页帧数、空洞页帧数等，见下文，/mm/page_alloc.c*/
    alloc_node_mem_map(pgdat);      /*创建结点 page 结构体数组，见下文，/mm/page_alloc.c*/
#ifdef CONFIG_FLAT_NODE_MEM_MAP
    ...                           /*输出信息*/
#endif
    free_area_init_core(pgdat, start_pfn, end_pfn);  /*初始化结点中各内存域，/mm/page_alloc.c*/
}
```

free_area_init_node()函数中调用 get_pfn_range_for_nid()函数从 memblock 实例获取结点起止页帧号，调用 calculate_node_totalpages()函数计算结点位于各内存域的页帧数量（含空洞、不含空洞），调用函数 alloc_node_mem_map()为结点页帧创建 page 实例数组，调用 free_area_init_core()遍历结点实例中内存域 zone 结构体成员并对其进行初始化。下先看一下内存域页帧的计算和 page 实例数组的创建，内存域初始化函数下一小节再做介绍。

1 计算页帧数

初始化结点 free_area_init_node()函数调用 calculate_node_totalpages()函数计算结点总页帧数及各内存域的页帧数、真实页帧数及空洞页帧数，在计算内存域页帧数量时需要考虑可移动内域占用的页帧。

calculate_node_totalpages()函数代码如下（/mm/page_alloc.c）：

```
static void __meminit calculate_node_totalpages(struct pglist_data *pgdat,unsigned long node_start_pfn,  \
                        unsigned long node_end_pfn,unsigned long *zones_size,unsigned long *zholes_size)
/*pgdat：指向结点结构，node_start_pfn：起始页帧号，node_end_pfn：结束页帧号*/
{
    unsigned long realtotalpages = 0, totalpages = 0;
    enum zone_type i;

    for (i = 0; i < MAX_NR_ZONES; i++) {      /*遍历内存域*/
        struct zone *zone = pgdat->node_zones + i;
        unsigned long size, real_size;

        size = zone_spanned_pages_in_node(pgdat->node_id, i,node_start_pfn,node_end_pfn,zones_size);
                          /*计算结点内存位于此内存域的页帧数（含空洞），/mm/page_alloc.c*/
        real_size = size - zone_absent_pages_in_node(pgdat->node_id, i,node_start_pfn, node_end_pfn, \
                                                                                    zholes_size);
                          /*计算结点内存位于此内存域的真实页帧数（不含空洞），/mm/page_alloc.c*/
```

```
                        /*zone_absent_pages_in_node()返回空洞页帧数，由 memblock 实例计算空洞*/
        zone->spanned_pages = size;              /*内存域跨越的页帧数量，含空洞*/
        zone->present_pages = real_size;       /*内存域实际跨越的页帧数，不含空洞*/

        totalpages += size;                      /*累加结点页帧数*/
        realtotalpages += real_size;
    }       /*遍历内存域结束*/

    pgdat->node_spanned_pages = totalpages;          /*结点跨越的页帧数*/
    pgdat->node_present_pages = realtotalpages;        /*结点跨越的真实页帧数*/
    printk(KERN_DEBUG "On node %d totalpages: %lu\n", pgdat->node_id,realtotalpages);
}
```

calculate_node_totalpages()函数遍历结点各内存域实例，计算结点内存位于各内存域含空洞和不含空洞情况下的页帧数量，空洞长度由 memblock 实例计算得来。

2 创建 page 实例数组

初始化结点 free_area_init_node()函数调用 alloc_node_mem_map(pgdat)函数为结点创建 page 实例数组，函数代码如下（/mm/page_alloc.c）：

```
static void __init_refok    alloc_node_mem_map(struct pglist_data *pgdat)
{
    if (!pgdat->node_spanned_pages)        /*跳过空结点*/
        return;

  #ifdef    CONFIG_FLAT_NODE_MEM_MAP       /*默认选择此配置选项，/mm/Kconfig*/
    if (!pgdat->node_mem_map) {                    /*如果尚未创建 page 数组*/
        unsigned long    size, start, end;
        struct page *map;
        start = pgdat->node_start_pfn & ~(MAX_ORDER_NR_PAGES - 1);
                                        /*起始页帧号 MAX_ORDER_NR_PAGES 对齐，下对齐*/
        end = pgdat_end_pfn(pgdat);     /*结点结束页帧号，/include/linux/mmzone.h*/
        end = ALIGN(end, MAX_ORDER_NR_PAGES);
                                        /*结束页帧号 MAX_ORDER_NR_PAGES 对齐，上对齐*/
        size = (end - start) * sizeof(struct page);              /*page 数组大小*/
        map = alloc_remap(pgdat->node_id, size);            /*/include/linux/bootmem.h*/
          /*若配置了 HAVE_ARCH_ALLOC_REMAP 由体系结构代码实现，否则直接返回 NULL*/
        if (!map)
            map = memblock_virt_alloc_node_nopanic(size,pgdat->node_id);
                                    /*由自举分配器为 page 数组分配空间，/include/linux/bootmem.h*/
        pgdat->node_mem_map = map + (pgdat->node_start_pfn - start);
                                                /*指向结点实际起始页帧 page 实例*/
    }
  #ifndef CONFIG_NEED_MULTIPLE_NODES     /*全局变量 mem_map 赋值，单结点系统*/
    if (pgdat == NODE_DATA(0)) {       /*第一个结点 pg_data_t 实例*/
```

```
            mem_map = NODE_DATA(0)->node_mem_map;
        #ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
            if (page_to_pfn(mem_map) != pgdat->node_start_pfn)
                                        /*如果结点起始页帧号不是 ARCH_PFN_OFFSET*/
                mem_map -= (pgdat->node_start_pfn - ARCH_PFN_OFFSET);
        #endif
        }
    #endif
#endif    /* CONFIG_FLAT_NODE_MEM_MAP */
}
```

alloc_node_mem_map()函数对结点起止页帧号按页块（MAX_ORDER_NR_PAGES）进行下、上对齐，对页块所含页帧创建 page 实例数组，最后将实际起始页帧对应 page 实例地址赋予 pgdat->node_mem_map 成员，并设置 mem_map 全局变量值。

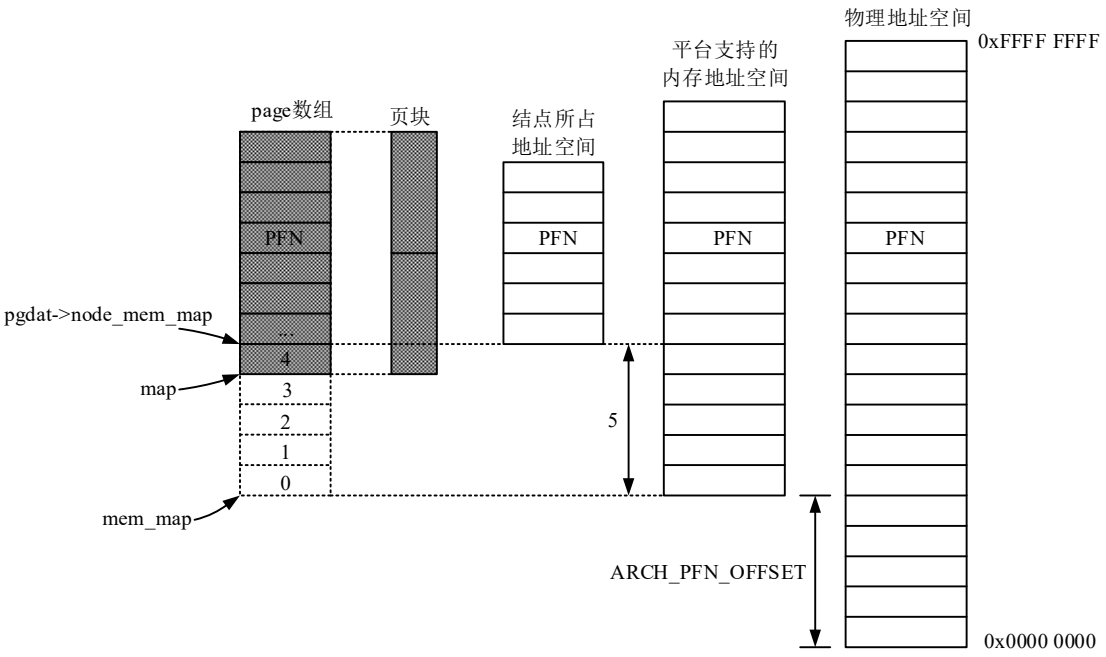
下面通过一个例子来说明，如何给结点创建 page 实例数组。对于 32 位处理器，其可寻址的物理地址空间范围是 0x0000 0000 至 0xFFFF FFFF。对于某个平台或处理器来说，可能并不是所有物理地址空间都是内存空间，因为物理地址空间还要映射外设寄存器、ROM 等，因此实际可寻址的物理内存是处理器可寻址空间的一个子集。

如下图所示，为不失一般性，我们假设处理器可寻址物理内存地址并不是从 0 开始，而是有一个偏移量，为 ARCH_PFN_OFFSET 个页帧（ARCH_PFN_OFFSET 通常为 0，如 MIPS）。假设内存结点所占空间又是平台寻址物理内存空间的一个子集，相对于 ARCH_PFN_OFFSET 又有一个偏移量，如下图中所示为偏移 5 个页帧。

假设内核设置的最大分配阶为 2，即页块中包含 4 个页帧，则对内存结点页帧按页块下、上对齐，对上下不满一个页块的页帧视为一个页块，如下图阴影部分所示。

实际创建的 page 实例数组是按页块对齐后的页帧数创建的，map 局部变量指向实例 page 数组的起始位置，而结点 node_mem_map 成员指向实际的第一个物理页帧对应的 page 实例，page 数组包含空洞页帧。

全局变量 mem_map 表示的是平台（处理器）可寻址物理内存地址空间起始页对应的 page 实例（虚指，实际不存在），即 ARCH_PFN_OFFSET 页帧对应的 page 实例。



内核在/include/asm-generic/memory_model.h 头文件定义了 page 实例与页帧号之间转换的宏（UMA 系统）：

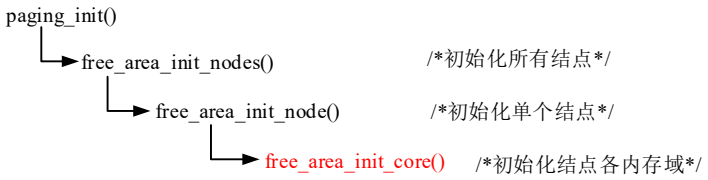
```
#define   page_to_pfn    __page_to_pfn        /*page 转页帧号，FLATMEM 内存模型，只有一个结点*/

#define   __page_to_pfn(page)    ((unsigned long)((page) - mem_map) + ARCH_PFN_OFFSET)
                                                    /*返回 page 实例对应的页帧号*/

#define   pfn_to_page    __pfn_to_page                /*页帧号转 page 指针*/
#define   __pfn_to_page(pfn)      (mem_map + ((pfn) - ARCH_PFN_OFFSET))
                                                    /*页帧号对应的 page 实例地址*/
```

通过上面的例子，读者应该不难理解以上的宏了。

读者需要注意的是page_to_pfn(page)是基于mem_map变量计算的，mem_map初始设为node_mem_map成员值，在 alloc_node_mem_map()函数中调用了 page_to_pfn(mem_map)函数，由上面的计算式可知，此函数返回值为 ARCH_PFN_OFFSET，if 语句的判断条件是结点起始页帧号不是 ARCH_PFN_OFFSET，就需要修改 mem_map 变量值，使其指向虚拟的 ARCH_PFN_OFFSET 页帧对应的 page 实例。

3.4.4 初始化内存域

初始化单个结点函数 free_area_init_node()在最后将调用 free_area_init_core()函数完成结点内存域实例的初始化，函数调用关系如下图所示：



由于内存域初始化函数 free_area_init_core()比较重要且复杂许多，因此这里将其单独列出，函数定义在/mm/page_alloc.c 文件内，代码如下：

```
static void __paginginit free_area_init_core(struct pglist_data *pgdat,        \
                                    unsigned long node_start_pfn, unsigned long node_end_pfn)
/*pgdat：指向结点 pglist_data 实例，node_start_pfn、node_end_pfn：结点起止页帧号*/
{
    enum zone_type j;
    int nid = pgdat->node_id;        /*结点编号，UMA 系统为 0*/
    unsigned long   zone_start_pfn = pgdat->node_start_pfn;     /*结点起始页帧号*/
    int ret;

    pgdat_resize_init(pgdat);
            /*若没有选择 MEMORY_HOTPLUG 则为空操作，/include/linux/memory_hotplug.h*/
  #ifdef CONFIG_NUMA_BALANCING
    ...
  #endif
    init_waitqueue_head(&pgdat->kswapd_wait);          /*初始化页面回收守护线程等待队列*/
    init_waitqueue_head(&pgdat->pfmemalloc_wait);       /*初始化等待内存分配的进程等待队列*/
    pgdat_page_ext_init(pgdat);
```

```
            /*若没有选择 PAGE_EXTENSION 则为空操作，/include/linux/page_ext.h*/

for (j = 0; j < MAX_NR_ZONES; j++) {              /*遍历结点中内存域，初始化各内存域*/
    struct zone *zone = pgdat->node_zones + j;
    unsigned long size, realsize, freesize, memmap_pages;    /*freesize 影响水印值的计算*/

    size = zone->spanned_pages;                   /*内存域跨越的页帧数*/
    realsize = freesize = zone->present_pages;    /*内存域实际页帧数*/

    memmap_pages = calc_memmap_size(size, realsize);
                /*计算内存域对应的 page 数组所占的页帧数，/mm/page_alloc.c*/
    if (!is_highmem_idx(j)) {                     /*非高端内存域*/
        if (freesize >= memmap_pages) {           /*内存域实际页帧数不小于 page 数组所占页帧数*/
            freesize -= memmap_pages;             /*freesize 减去 page 数组占页帧数*/
            if (memmap_pages)
                ...     /*输出信息*/
        } else
            ...         /*输出信息*/
    }

    if (j == 0 && freesize > dma_reserve) {   /*没有 DMA 内存域，dma_reserve 为 0*/
        freesize -= dma_reserve;              /*若存在 DMA 内存域，预留 DMA 空间*/
        ...                 /*输出信息*/
    }

    if (!is_highmem_idx(j))             /*非高端内存域*/
        nr_kernel_pages += freesize;
            /*nr_kernel_pages 全局变量统计内核直接映射区可用内存，/mm/page_alloc.c*/
    else if (nr_kernel_pages > memmap_pages * 2)      /*高端内存域*/
        nr_kernel_pages -= memmap_pages;              /*减去高端内存域 page 数组占用页帧数*/

    nr_all_pages += freesize;                 /*总的可用内存页帧数*/

    zone->managed_pages = is_highmem_idx(j) ? realsize : freesize;
                /*估算内存域伙伴系统可管理的页帧数，区分高端内存域和非高端内存域*/
#ifdef CONFIG_NUMA
    ...
#endif
    zone->name = zone_names[j];       /*zone_names[]为静态全局数组，/mm/page_alloc.c*/
    spin_lock_init(&zone->lock);
    spin_lock_init(&zone->lru_lock);
    zone_seqlock_init(zone);
    zone->zone_pgdat = pgdat;     /*指向结点*/
```

```
            zone_pcp_init(zone);
                    /*初始化 CPU 单页缓存，zone->pageset = &boot_pageset，/mm/page_alloc.c*/

            mod_zone_page_state(zone, NR_ALLOC_BATCH, zone->managed_pages);
                                    /*修改内存域 NR_ALLOC_BATCH 统计量，/mm/vmstat.c*/
            lruvec_init(&zone->lruvec);      /*初始化 LRU 链表为空，/mm/mmzone.c*/
            if (!size)                       /*内存域大小为 0，跳出此次循环*/
                continue;

            set_pageblock_order();  /*没有配置 HUGETLB_PAGE_SIZE_VARIABLE，则为空操作*/
            setup_usemap(pgdat, zone, zone_start_pfn, size);      /*为 pageblock_flags 分配空间*/
            ret = init_currently_empty_zone(zone, zone_start_pfn,size, MEMMAP_EARLY);
                                    /*初始化等待队列，伙伴系统链表初始化为空，/mm/page_alloc.c*/
            BUG_ON(ret);
            memmap_init(size, nid, j, zone_start_pfn);    /*初始化 page 数组，/mm/page_alloc.c*/
            zone_start_pfn += size;      /*下一内存域起始页帧号*/
        }     /*遍历结点中内存域结束*/
    }
```

free_area_init_core()函数遍历结点中各内存域，对每个内存域执行以下操作：

（1）估算内存域中空闲页帧数量（可被伙伴系统管理的页帧数），并累加到全局变量。

（2）调用 zone_pcp_init()函数，设置 zone->pageset = &boot_pageset，boot_pageset 为 per_cpu_pageset 实例（percpu 变量）。

（3）调用 lruvec_init()函数初始化内存域 LRU 链表数组，初始设为空。

（4）调用 setup_usemap()函数为页块迁移属性标记位图分配空间。

（5）调用 init_currently_empty_zone()函数初始化等待队列，以及伙伴系统链表。

（6）调用 memmap_init()函数初始化内存域页帧对应的 page 实例（数组）。

下面分别介绍以上主要函数代码及其实现的功能。

1 初始化 CPU 单页缓存

在 free_area_init_core()函数内，各内存域 pageset 成员都指向 per_cpu_pageset 结构体实例 boot_pageset（percpu 变量）。在内核启动函数 start_kernel()内将调用 setup_per_cpu_pageset()函数（伙伴系统和 slab 分配器初始化后）为每个内存域创建 per_cpu_pageset 结构体实例（percpu 变量，每个 CPU 对应一个实例）。

setup_per_cpu_pageset()函数定义在/mm/page_alloc.c 文件内：

```
void __init setup_per_cpu_pageset(void)
{
    struct zone *zone;

    for_each_populated_zone(zone)            /*遍历非空的内存域*/
        setup_zone_pageset(zone);            /*分配并初始化 per_cpu_pageset 实例，/mm/page_alloc.c*/
}
```

setup_per_cpu_pageset()函数遍历所有结点非空的内存域，对每个内存域调用 setup_zone_pageset(zone)函数，用于创建和初始化 per_cpu_pageset 实例（percpu 变量），函数代码如下：

```
static void __meminit setup_zone_pageset(struct zone *zone)
```

```
{
    int cpu;
    zone->pageset = alloc_percpu(struct per_cpu_pageset);   /*创建 percpu 变量 per_cpu_pageset 实例*/
    for_each_possible_cpu(cpu)
        zone_pageset_init(zone, cpu);        /*初始化每个 per_cpu_pageset 实例，/mm/page_alloc.c*/
}
```

初始化 per_cpu_pageset 实例的函数为 zone_pageset_init(zone, cpu)，定义如下：

```
static void __meminit zone_pageset_init(struct zone *zone, int cpu)
{
    struct per_cpu_pageset *pcp = per_cpu_ptr(zone->pageset, cpu);

    pageset_init(pcp);      /*单页缓存数量清零，初始化链表等，/mm/page_alloc.c*/
    pageset_set_high_and_batch(zone, pcp);      /*/mm/page_alloc.c*/
                                /*根据 zone->managed_pages 值，计算并设置 high、batch 成员值*/
}
```

2 页迁移属性标记分配空间

free_area_init_core()函数调用 setup_usemap()函数为内存域所含页块，分配标记迁移属性的位图空间，函数代码如下（/mm/page_alloc.c）：

```
static void __init setup_usemap(struct pglist_data *pgdat,struct zone *zone,unsigned long zone_start_pfn, \
                                                                    unsigned long zonesize)
/*zone_start_pfn：内存域起始页帧号，zonesize：内存域跨越的页帧数量*/
{
    unsigned long usemapsize = usemap_size(zone_start_pfn, zonesize);
                                                        /*页块迁移属性标记位图所占空间，字节数*/
    zone->pageblock_flags = NULL;
    if (usemapsize)
        zone->pageblock_flags =memblock_virt_alloc_node_nopanic(usemapsize,pgdat->node_id);
                                        /*从自举分配器分配空间，赋予 pageblock_flags 成员*/
}
```

setup_usemap()函数将内存域按页块对齐（pageblock_nr_pages），每个页块由 NR_PAGEBLOCK_BITS 数量的比特位表示迁移属性。从自举分配器为内存域页块迁移属性标记分配空间，最后将分配空间基地址赋予内存域结构 pageblock_flags 成员。

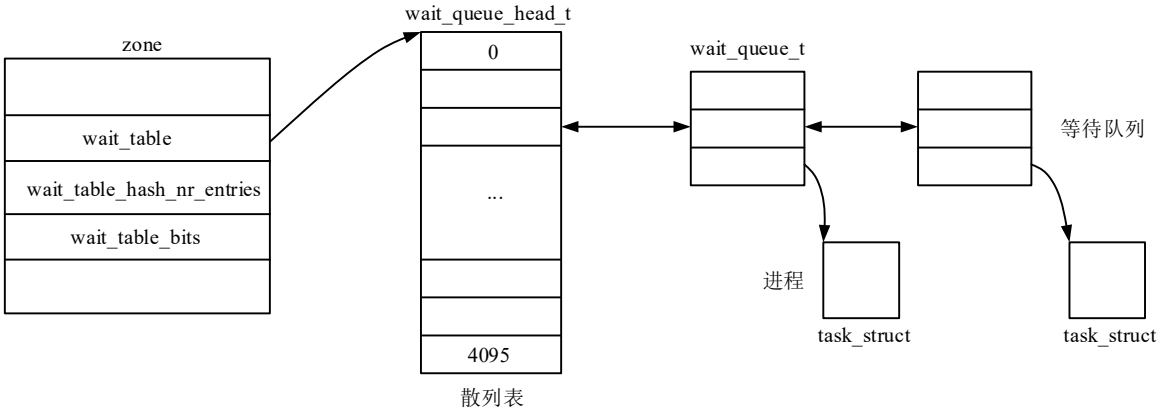
3 初始化等待队列及伙伴系统链表

free_area_init_core()函数调用 init_currently_empty_zone()函数完成内存域进程等待队列及伙伴系统链表的初始化，函数定义在/mm/page_alloc.c 文件内：

```
int __meminit init_currently_empty_zone(struct zone *zone,unsigned long zone_start_pfn,  \
                                                    unsigned long size,enum memmap_context context)
 /*zone_start_pfn：起始页帧号，size：内存域跨越的页帧数量，context：未使用参数*/
{
```

```
    struct pglist_data *pgdat = zone->zone_pgdat;       /*结点结构*/
    int ret;
    ret = zone_wait_table_init(zone, size);     /*内存域等待队列初始化，/mm/page_alloc.c*/
    if (ret)
        return ret;
    pgdat->nr_zones = zone_idx(zone) + 1;           /*结点实际内存域数量加 1*/
    zone->zone_start_pfn = zone_start_pfn;          /*内存域起始页帧号*/
    ...     /*输出信息*/
    zone_init_free_lists(zone);         /*初始化伙伴系统页链表为空，/mm/page_alloc.c*/
    return 0;
}
```

init_currently_empty_zone()函数中调用 zone_wait_table_init()函数用于创建内存域页等待队列，用于管理在页上等待的进程。由于内存域中页数量较多，不可能为每个页帧创建等待队列，因此内存域创建了固定大小的等待队列列表，在页上等待的进程通过散列的方式添加到等待队列列表中的某一个等待队列。内存域等待队列结构如下图所示：



wait_table 指向 wait_queue_head_t 等待队列列表起始位置，wait_table_hash_nr_entries 表示等待队列列表成员数量，wait_table_bits 为队列数量的 2 的幂次表示，即 $2^{wait\_table\_bits-1}$=wait_table_hash_nr_entries。

zone_wait_table_init()函数定义在/mm/page_alloc.c 文件内，代码如下：

```
static noinline __init_refok int    zone_wait_table_init(struct zone *zone, unsigned long zone_size_pages)
/*zone_size_pages：内存域跨越的页帧数量*/
{
    int i;
    size_t alloc_size;
    zone->wait_table_hash_nr_entries =wait_table_hash_nr_entries(zone_size_pages);
                                        /*等待队列头列表项数（或为 4096），/mm/page_alloc.c*/
    zone->wait_table_bits =wait_table_bits(zone->wait_table_hash_nr_entries);
    alloc_size = zone->wait_table_hash_nr_entries* sizeof(wait_queue_head_t);
                                        /*等待队列散列表占用的内存空间*/
    if (!slab_is_available()) {         /*此时 slab 分配器尚不可用*/
        zone->wait_table = (wait_queue_head_t *)memblock_virt_alloc_node_nopanic(alloc_size,\
                                                zone->zone_pgdat->node_id);
                                        /*由自举分配器为等待队列头列表*/
```

```
    } else {                        /*slab 分配器可用*/
        zone->wait_table = vmalloc(alloc_size);          /*内核映射区分配空间*/
    }
    if (!zone->wait_table)
        return -ENOMEM;

    for (i = 0; i < zone->wait_table_hash_nr_entries; ++i)    /*初始化等待队列表头*/
        init_waitqueue_head(zone->wait_table + i);
    return 0;
}
```

散列表由 wait_queue_head_t 结构体实例组成，它是进程等待队列表头，等待进程添加到队列中，详见第 5 章。

init_currently_empty_zone()函数中调用 zone_init_free_lists()函数初始化内存域伙伴系统链表，函数定义在/mm/page_alloc.c 文件内：

```
static void __meminit zone_init_free_lists(struct zone *zone)
{
    unsigned int order, t;
    for_each_migratetype_order(order, t) {
        INIT_LIST_HEAD(&zone->free_area[order].free_list[t]);
        zone->free_area[order].nr_free = 0;
    }
}
```

zone_init_free_lists()函数将内存域伙伴系统各链表初始化为空，空闲页帧数量设为 0。在停用自举分配器时，会将空闲页帧释放到伙伴系统链表，而后就要以使用伙伴系统分配内存了。

4 初始化 page 数组

free_area_init_core()函数最后调用 memmap_init()函数对内存域页帧对应 page 实例进行初始化，函数定义在/mm/page_alloc.c 文件内：

```
#ifndef   __HAVE_ARCH_MEMMAP_INIT
  #define   memmap_init(size, nid, zone, start_pfn) \
        memmap_init_zone((size), (nid), (zone), (start_pfn), MEMMAP_EARLY)
#endif
```

MEMMAP_EARLY 参数为 memmap_context 枚举类型变量（/include/linux/mmzone.h）：

```
enum memmap_context {
    MEMMAP_EARLY,
    MEMMAP_HOTPLUG,        /*热插拔内存*/
};
```

memmap_init_zone()函数定义在/mm/page_alloc.c 文件内，代码如下：

```
void __meminit memmap_init_zone(unsigned long size, int nid, unsigned long zone,    \
                                unsigned long start_pfn, enum memmap_context context)
```

```
/*size：内存域跨越页帧数，nid：结点编号，zone：内存域编号，start_pfn：起始页帧号*/
{
    pg_data_t *pgdat = NODE_DATA(nid);
    unsigned long end_pfn = start_pfn + size;        /*内存域最大页帧号*/
    unsigned long pfn;
    struct zone *z;
    unsigned long nr_initialised = 0;

    if (highest_memmap_pfn < end_pfn - 1)       /*highest_memmap_pfn 定义在/mm/memory.c*/
        highest_memmap_pfn = end_pfn - 1;     /*page 数组最大映射页帧号*/

    z = &pgdat->node_zones[zone];                    /*zone 实例指针*/

    for (pfn = start_pfn; pfn < end_pfn; pfn++) {     /*遍历内存域页帧对应的 page 实例*/
        if (context == MEMMAP_EARLY) {        /*默认值*/
            if (!early_pfn_valid(pfn))                /*页帧号有效性判断*/
                continue;
            if (!early_pfn_in_nid(pfn, nid))
                continue;
            if (!update_defer_init(pgdat, pfn, end_pfn,&nr_initialised))
                break;
        }
        if (!(pfn & (pageblock_nr_pages - 1))) {       /*页块首页*/
            struct page *page = pfn_to_page(pfn);
            __init_single_page(page, pfn, zone, nid);   /*初始化 page 实例*/
            set_pageblock_migratetype(page, MIGRATE_MOVABLE); /*设置页块迁移属性*/
                /*此时全局变量 page_group_by_mobility_disabled 值为 0，
                *因此设置的页块迁移属性应为 MIGRATE_UNMOVABLE（不可移动）。*/
        } else {                        /*非页块首页*/
            __init_single_pfn(pfn, zone, nid);
                                /*调用__init_single_page(pfn_to_page(pfn), pfn, zone, nid)函数*/
        }
    }    /*遍历 page 实例结束*/
}
```

前面介绍过内核为防止碎片，将物理内存以 pageblock_nr_pages 个页帧为单位划分成页块，页块赋予迁移属性类型。memmap_init_zone()函数遍历内存域页帧对应的 page 实例，若 page 表示页帧为页块首页，则调用 set_pageblock_migratetype()函数设置页块迁移属性（设为 MIGRATE_UNMOVABLE），并初始化 page 实例；若 page 不是页块首页，则直接初始化 page 实例。

page 实例初始化最终由__init_single_page()函数完成，函数定义在/mm/page_alloc.c 文件内，代码如下：

```
static void __meminit    __init_single_page(struct page *page, unsigned long pfn,unsigned long zone, int nid)
{
    set_page_links(page, zone, nid, pfn);   /*设置 page 所在内存域、结点标记位，/include/linux/mm.h*/
    init_page_count(page);                  /*page->_count 置 1，/include/linux/mm.h*/
```

```
        page_mapcount_reset(page);              /*page->_mapcount 置-1，/include/linux/mm.h*/
        page_cpupid_reset_last(page);           /*UMA 空操作，/include/linux/mm.h*/

        INIT_LIST_HEAD(&page->lru);         /*初始化 lru 链表成员*/
    #ifdef WANT_PAGE_VIRTUAL
        if (!is_highmem_idx(zone))
            set_page_address(page, __va(pfn << PAGE_SHIFT));
    #endif
    }
```

内存域页帧对应 page 实例在 memmap_init()函数中完成初始化，此时 page 实例并没有添加到伙伴系统链表中，因此还不能由伙伴系统分配内存。在停用自举分配器时，会将空闲页帧（注意只有空闲页帧）对应的 page 实例添加到伙伴系统链表中，而后就可以从伙伴系统分配内存。

至此，物理内存结点和内存域数据结构实例初始化基本完成。初始化函数主要完成的工作有：设置结点、内存域起止页帧号；为内存域创建页块迁移属性位图，并初始化所有页块为不可移动迁移类型；初始化内存域等待进程队列，初始化伙伴系统链表为空；为结点创建 page 实例数组并初始化等。

3.4.5 设置内存域水印值

物理内存结点、内存域数据结构实例初始化中还有一项重要的工作，没有在初始化函数中完成，那就是设置内存域的水印值，水印值主要用于页面回收机制。内核在启动阶段后期初始化子系统时完成内存域水印值的设置。

zone 结构体中水印值相关成员如下：

```
struct zone {
    unsigned long    watermark[NR_WMARK];      /*内存域水印值，NR_WMARK=3*/
    long    lowmem_reserve[MAX_NR_ZONES];     /*预留的页帧数，供紧急情况使用*/
    unsigned int    inactive_ratio;          /*LRU 链表活跃匿名页与不活跃匿名页比值*/
    ...
    atomic_long_t       inactive_age;              /*活跃与不活跃匿名映射页比值*/
    ...
}
```

水印值数组 watermark[]成员，其数组项数由枚举类型确定，定义在/include/linux/mmzone.h 头文件：

```
enum zone_watermarks {
    WMARK_MIN,
    WMARK_LOW,
    WMARK_HIGH,
    NR_WMARK     /*数组项数*/
};
```

内存域水印值主要用于页面回收机制，水印值各数组项语义如下：

watermark[WMARK_MIN]：如果内存域空闲页帧数量低于它，说明内存域中急需空闲页，页面回收机制的工作压力比较大。

watermark[WMARK_LOW]：如果内存域空闲页帧数量低于它，则启动页面回收机制。

watermark[WMARK_HIGH]：如果内存域空闲页帧数量多于它，则表示内存域状态是理想的。

内核在/mm/page_alloc.c 文件内定义了以下全局变量：

```
unsigned long    totalram_pages    __read_mostly;          /*RAM 内存页帧总数*/
unsigned long    totalreserve_pages    __read_mostly;      /*预留内存总数*/

int    min_free_kbytes = 1024;          /*最小空闲内存大小，单位 KB*/
int    user_min_free_kbytes = -1;
```

min_free_kbytes 表示最小空闲内存字节数（KB）。

内存域水印值初始化函数为 init_per_zone_wmark_min()，此函数在内核启动阶段后期调用，即在伙伴系统、slab 分配器初始化完成后，并且内核已经通过伙伴系统、slab 分配器分配了一些内存。

init_per_zone_wmark_min()函数定义在/mm/page_alloc.c 文件内，代码如下：

```
int __meminit    init_per_zone_wmark_min(void)
{
    unsigned long    lowmem_kbytes;      /*低端内存域空闲内存大小（单位 KB）*/
    int    new_min_free_kbytes;

    lowmem_kbytes = nr_free_buffer_pages() * (PAGE_SIZE >> 10);
                /*DMA 和普通内存域中减去 watermark[WMARK_HIGH]页帧后的空间大小，单位 KB*/
    new_min_free_kbytes = int_sqrt(lowmem_kbytes * 16);
                                        /*新最小空闲内存大小，字节数（KB）*/
    if (new_min_free_kbytes > user_min_free_kbytes) {
        min_free_kbytes = new_min_free_kbytes;      /*全局变量赋值*/
        if (min_free_kbytes < 128)
            min_free_kbytes = 128;
        if (min_free_kbytes > 65536)
            min_free_kbytes = 65536;      /*初始化 min_free_kbytes  数值*/
     } else {
      ...          /*输出信息*/
     }
     setup_per_zone_wmarks();        /*设置所有内存域水印值，/mm/page_alloc.c*/
     refresh_zone_stat_thresholds();    /*设置 pageset.stat_threshold 成员值，SMP，/mm/vmstat.c*/
     setup_per_zone_lowmem_reserve();    /*设置 lowmem_reserve[]数组值，/mm/page_alloc.c*/
     setup_per_zone_inactive_ratio();      /*设置活跃与不活跃匿名映射页比值，/mm/page_alloc.c*/
     return 0;
}
module_init(init_per_zone_wmark_min)    /*内核启动后期初始化子系统时调用此函数*/
```

init_per_zone_wmark_min()函数根据普通内存域（含 DMA 内存域）空闲页帧数量确定系统最小空闲页帧数，赋予全局变量 min_free_kbytes。随后，调用 setup_per_zone_wmarks()函数设置各内存域 watermark[]数组表示的水印值，调用 setup_per_zone_lowmem_reserve()函数设置各内存域 lowmem_reserve[]数组项值，调用 setup_per_zone_inactive_ratio()函数设置 LRU 链表中活跃与不活跃匿名映射页的比值。

1 设置水印值

setup_per_zone_wmarks()函数用于设置所有内存域水印值，函数定义在/mm/page_alloc.c 文件内：

```
void setup_per_zone_wmarks(void)
{
    mutex_lock(&zonelists_mutex);
    __setup_per_zone_wmarks();          /*/mm/page_alloc.c*/
    mutex_unlock(&zonelists_mutex);
}
```

setup_per_zone_wmarks()函数在互斥量的保护下将工作委托给__setup_per_zone_wmarks()函数，代码如下：

```
static void   __setup_per_zone_wmarks(void)
{
    unsigned long pages_min = min_free_kbytes >> (PAGE_SHIFT - 10);    /*最小空闲页帧数*/
    unsigned long lowmem_pages = 0;        /*低端内存页帧数*/
    struct zone *zone;
    unsigned long flags;

    for_each_zone(zone) {          /*遍历各结点中各内存域，/include/linux/mmzone.h*/
        if (!is_highmem(zone))        /*不是高端内存域*/
            lowmem_pages += zone->managed_pages;    /*计算除高端内存域外内存域页帧数之和*/
    }

    for_each_zone(zone) {        /*遍历所有内存域*/
        u64 tmp;

        spin_lock_irqsave(&zone->lock, flags);
        tmp = (u64)pages_min * zone->managed_pages;      /*内存域实际页帧数*/
        do_div(tmp, lowmem_pages);

        if (is_highmem(zone)) {      /*设置高端内存域 watermark[WMARK_MIN]数组*/
            unsigned long min_pages;
            min_pages = zone->managed_pages / 1024;
            min_pages = clamp(min_pages, SWAP_CLUSTER_MAX, 128UL);
            zone->watermark[WMARK_MIN] = min_pages;
        } else {                      /*设置非高端内存域 watermark[WMARK_MIN]*/
            zone->watermark[WMARK_MIN] = tmp;
        }
                    /*#define min_wmark_pages(z) (z->watermark[WMARK_MIN])*/
        zone->watermark[WMARK_LOW] = min_wmark_pages(zone) + (tmp >> 2);
        zone->watermark[WMARK_HIGH] = min_wmark_pages(zone) + (tmp >> 1);
         /*设置内存域 watermark[WMARK_MIN]和 watermark[WMARK_HIGH]水印值*/
```

```
            __mod_zone_page_state(zone, NR_ALLOC_BATCH,
                high_wmark_pages(zone) - low_wmark_pages(zone) -
                atomic_long_read(&zone->vm_stat[NR_ALLOC_BATCH]));
                                        /*修改内存域统计量*/
            setup_zone_migrate_reserve(zone);   /*/mm/page_alloc.c*/
                                    /*设置一定数量页块迁移类型为 MIGRATE_RESERVE*/
            spin_unlock_irqrestore(&zone->lock, flags);
        }   /*遍历所有内存域结束*/

        calculate_totalreserve_pages(); /*计算保留页帧数 totalreserve_pages，/mm/page_alloc.c*/
    }
```

__setup_per_zone_wmarks()函数遍历所有内存域设置其水印值，调用 setup_zone_migrate_reserve(zone)函数设置一定数量页块的迁移属性为 MIGRATE_RESERVE。

2 设置预留页帧数

setup_per_zone_lowmem_reserve()函数用于设置各内存域 lowmem_reserve[MAX_NR_ZONES]数组成员值，函数定义在/mm/page_alloc.c 文件内：

```
static void setup_per_zone_lowmem_reserve(void)
{
    struct pglist_data *pgdat;
    enum zone_type j, idx;

    for_each_online_pgdat(pgdat) {      /*遍历各结点*/
        for (j = 0; j < MAX_NR_ZONES; j++) {        /*遍历结点中各内存域*/
            struct zone *zone = pgdat->node_zones + j;
            unsigned long managed_pages = zone->managed_pages;      /*内存域管理的页帧数*/
            zone->lowmem_reserve[j] = 0;      /*本内存域对应的数组项值设为 0*/
            idx = j;
            while (idx) {       /*遍历更低级内存域中对应本内存域的 lowmem_reserve[]数组项*/
              struct zone *lower_zone;
              idx--;
              if (sysctl_lowmem_reserve_ratio[idx] < 1)     /*预留比例数组，/mm/page_alloc.c*/
                  sysctl_lowmem_reserve_ratio[idx] = 1;

              lower_zone = pgdat->node_zones + idx;     /*低级内存域指针*/
              lower_zone->lowmem_reserve[j] = managed_pages /sysctl_lowmem_reserve_ratio[idx];
                                                        /*设置预留页帧数*/
              managed_pages += lower_zone->managed_pages;     /*累加内存域管理页帧数*/
            }
        }     /*遍历结点中各内存域结束*/
    }       /*遍历各结点结束*/
    calculate_totalreserve_pages();     /*计算总的预留内存页帧数量*/
}
```

每个内存域中 lowmem_reserve[**MAX_NR_ZONES**]数组中每个数组项对应一个内存域（含所有内存域），表示的是在本内存域中为本结点高于本内存域的内存域预留的页帧数，本内存域对应的数组项值设为 0。例如，假设系统中只有一个内存结点，只具有普通内存域（编号 0）和高端内存域（编号 1），则在普通内存域中 lowmem_reserve[0]数组项值为 0（对应普通内存域的数组项），而 lowmem_reserve[1]数组项的值表示在普通内存域中为高端内存域预留的页帧数。在高端内存域中 lowmem_reserve[]数组项全为 0，因为没有比高端内存域更高级的内存域了。

内核定义了 sysctl_lowmem_reserve_ratio[]数组，表示各内存域预留页帧数的比例，页帧数基数是本内存域上一级内存域至目地内存域，管理页帧数的总和。例如，计算 DMA 内存域为高端内存域预留的页帧数，计算基数为普通内存域和高端内存域管理页帧数的总和。

3 设置活跃/不活跃匿名映射页比值

setup_per_zone_inactive_ratio()函数用于设置内存域结构体中 inactive_ratio 成员值，即内存域 LRU 链表中活跃匿名映射页与不活跃匿名映射页之间的最大比值，函数定义在/mm/page_alloc.c 文件内：

```
static void __meminit    setup_per_zone_inactive_ratio(void)
{
    struct zone *zone;
    for_each_zone(zone)       /*遍历各内存域*/
        calculate_zone_inactive_ratio(zone);       /*/mm/page_alloc.c*/
}
```

setup_per_zone_inactive_ratio()函数遍历各个内存域，调用 calculate_zone_inactive_ratio(zone)函数计算比值，函数代码如下：

```
static void __meminit calculate_zone_inactive_ratio(struct zone *zone)
{
    unsigned int gb, ratio;

    /*内存域大小，单位 GB*/
    gb = zone->managed_pages >> (30 - PAGE_SHIFT);
    if (gb)
        ratio = int_sqrt(10 * gb);        /*大约等于 10*gb 开平方值*/
    else
        ratio = 1;
    zone->inactive_ratio = ratio;
}
```

calculate_zone_inactive_ratio()函数计算的比值结果如下：

内存域管理内存大小	比值（inactive_ratio）	不活跃匿名映射内存大小
10MB	1	5MB
100MB	1	50MB
1GB	3	250MB
10GB	10	0.9GB
100GB	31	3GB

1TB	101	10GB
10TB	320	32GB

zone->**inactive_ratio** 是物理内存域中活跃匿名映射页 LRU 链表中页数量与不活跃 LRU 链表中页数量比值的最大值，当实际比值比 inactive_ratio 大时，页回收机制将收缩活跃 LRU 链表，移动页至不活跃 LRU 链表，详见第 11 章。

3.5 借用内存初始化

伙伴系统在分配页帧时，根据分配函数参数从指定的内存域中分配，如果指定内存域没有足够的空闲页帧，则可以从其它内存域中借用。借用内存是有限制的，即只能从指定内存域更低端的内存域中借用，而不能从更高级的内存域中借用。例如：假设物理内存包括 DMA、NORMAL、HIGHMEM 三个内存域，HIGHMEM 内存域内存紧张时可从 NORMAL 和 DMA 内存域借用内存，NORMAL 内存域只能从 DMA 内存域借用内存，而不能从 HIGHMEM 内存域借用。

内存结点结构体中包含借用内存的内存域列表，在内核初始化过程中初始化各结点中借用内存域列表。伙伴系统分配页帧时，先从指定内存域分配，当指定内存域内存紧张时，分配函数将搜索借用内存列表查找合适的替代内存域，并从中分配内存。

3.5.1 借用内存列表

内存结点结构体 pg_data_t 中借用内存列表成员定义如下：

```
typedef struct pglist_data {
    ...
    struct zonelist    node_zonelists[MAX_ZONELISTS];      /*zonelist 实例数组*/
    ...
 }pg_data_t;
```

借用内存列表为 zonelist 结构体数组，数组项数为 MAX_ZONELISTS。MAX_ZONELISTS 宏定义在头文件/include/linux/mmzone.h。若选项了 NUMA 配置选项，则 MAX_ZONELISTS=2，否则为 1。因此，在 UMA 系统内，借用内存列表为一个 zonelist 结构体实例。

zonelist 结构体定义在/include/linux/mmzone.h 头文件：

```
struct zonelist {
    struct zonelist_cache    *zlcache_ptr;                  / *UMA 系统为空结构*/
    struct zoneref    _zonerefs[MAX_ZONES_PER_ZONELIST + 1];     /*最后一项为空*/
  #ifdef CONFIG_NUMA
    ...
  #endif
};

#define    MAX_ZONES_PER_ZONELIST    (MAX_NUMNODES * MAX_NR_ZONES)
                                                    /*最大结点数乘最大内存域数*/
```

zonelist 结构体内主要包含一个 zoneref 结构体数组，项数为 MAX_ZONES_PER_ZONELIST+1。MAX_ZONES_PER_ZONELIST 为最大结点数量与最大内存域数量之积，也就是说列表关联所有结点的所

有内存域，数组最后一项为空，表示列表结束。

zoneref 结构体定义在/include/linux/mmzone.h 头文件：

```
struct zoneref {
    struct zone    *zone;       /*内存域实例指针*/
    int    zone_idx;            /*内存域编号*/
};

#define zone_idx(zone)        ((zone) - (zone)->zone_pgdat->node_zones)      /*计算内存域编号*/
```

由以上数据结构可知，在 UMA 系统内，借用内存列表实际上是一个 zoneref 结构体数组，数组项包含指向各内存域实例指针和内存域编号成员。借用内存列表实际就是系统内所有内存域实例的列表，表示借用内存的优先顺序，最后一项为空，表示列表结束。

借用内存初始化函数的主要工作就是将所有内存域实例以某种排序规则添加到借用内存列表。伙伴系统分配物理内存时，若指定内存域没有足够的空闲内存则依次搜索借用内存列表，找到可借用内存的内存域并从中分配。

各内存域在借用内存列表中的排序方式有三种，定义在/mm/page_alloc.c 文件内：：

```
#define    ZONELIST_ORDER_DEFAULT     0     /*自动选择最优排序方式法*/
#define    ZONELIST_ORDER_NODE          1    /*结点（距离）优先*/
#define    ZONELIST_ORDER_ZONE          2    /*内存域类型优先*/

static int current_zonelist_order = ZONELIST_ORDER_DEFAULT;      /*内核默认排序方式*/
```

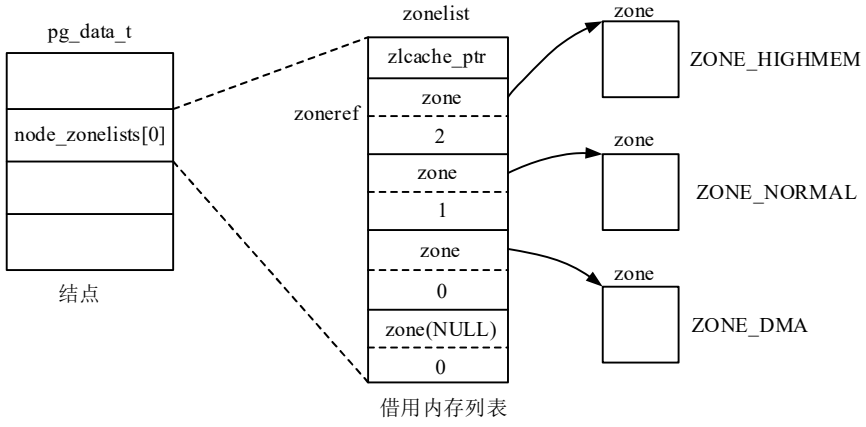
ZONELIST_ORDER_DEFAULT 表示内核根据探测到的信息自动决定最优排序方法，后面两个分别表示以结点（距离）和内存域类型优先排序。如果是 UMA 系统，后面两种方式所得列表是一样的。

set_zonelist_order()函数用于设置借用内存列表排序方式（UMA 系统），定义在/mm/page_alloc.c 文件内：

```
static void set_zonelist_order(void)
{
    current_zonelist_order = ZONELIST_ORDER_ZONE;      /*内存域优先排序方式*/
}
```

由函数可知，UMA 系统采用内存域优先的排序方式（只有一个结点）。

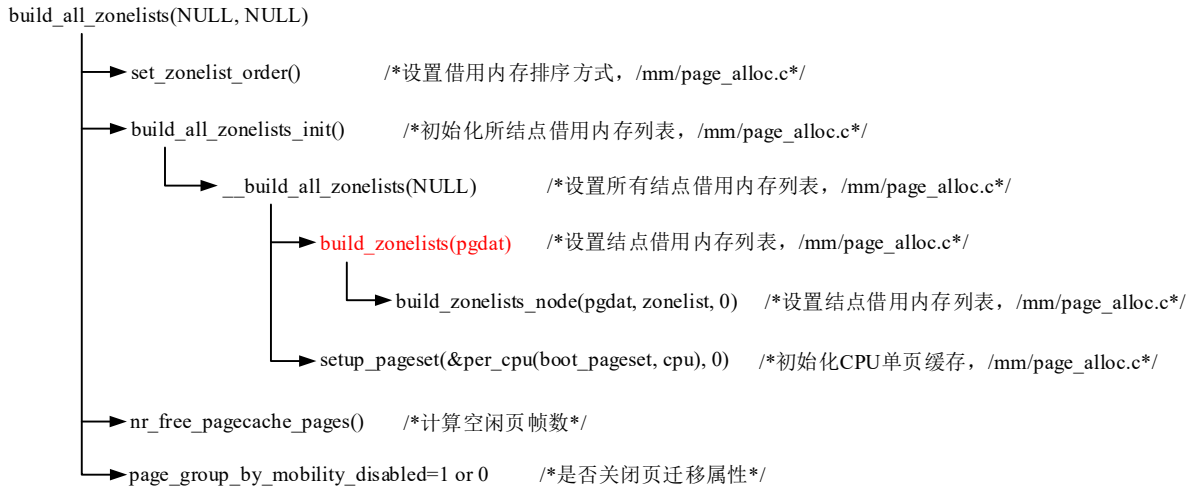
对于只有一个结点的 UMA 系统，初始化的借用内存列表如下图所示：



内存域实例在列表中从高到低排序，假设分配函数申请的是高端内存域内存，则先从列表第一项关联的高端内存域分配，若不能分配则依次选择列表中后面的普通内存域、DMA 内存域进行分配。

3.5.2 初始化函数

内核启动函数 start_kernel()调用 build_all_zonelists(NULL, NULL)函数完成各结点借用内存列表的初始化，函数定义/mm/page_alloc.c 文件内，函数调用关系如下：



build_all_zonelists()函数定义在/mm/page_alloc.c 文件内，代码如下：

```
void __ref build_all_zonelists(pg_data_t *pgdat, struct zone *zone)
{
    set_zonelist_order();                    /*设置列表排序方法，见上文，/mm/page_alloc.c*/

    if (system_state == SYSTEM_BOOTING) {        /*系统启动阶段*/
        build_all_zonelists_init();              /*完成所有结点借用内存列表初始化，/mm/page_alloc.c*/
    }
    else {        /*非系统启动阶段*/
      #ifdef CONFIG_MEMORY_HOTPLUG
        if (zone)
            setup_zone_pageset(zone);
      #endif
        stop_machine(__build_all_zonelists, pgdat, NULL);
    }

    vm_total_pages = nr_free_pagecache_pages();
                    /*计算超过 watermark[WMARK_HIGH]水印值的空闲页帧数，/mm/page_alloc.c*/

    if (vm_total_pages < (pageblock_nr_pages * MIGRATE_TYPES))  /*是否关闭页块迁移属性*/
        page_group_by_mobility_disabled = 1;      /*关闭页块迁移属性*/
    else
        page_group_by_mobility_disabled = 0;      /*启用页块迁移属性*/

    ...          /*输出信息*/
```

```
    #ifdef CONFIG_NUMA
        ...         /*输出信息*/
    #endif
}
```

build_all_zonelists()函数首先调用 set_zonelist_order()函数设置借用内存列表排序方式，然后调用函数 build_all_zonelists_init()完成各结点借用内存列表的初始化，最后，调用 nr_free_pagecache_pages()函数累加各结点内存域超过 watermark[WMARK_HIGH]水印值的空闲页帧数量，赋予全局变量 vm_total_pages，并确定是否关闭页块迁移属性。

1 初始化所有结点借用列表

build_all_zonelists()函数被调用时，若系统处于启动阶段，则调用 build_all_zonelists_init()函数完成各结点借用内存列表的初始化。全局变量 system_state 表示当前系统的状态，定义在/init/main.c 文件内。变量类型为同名的枚举类型，定义在/include/linux/kernel.h 头文件内：

```
extern enum system_states {
    SYSTEM_BOOTING,
    SYSTEM_RUNNING,
    SYSTEM_HALT,
    SYSTEM_POWER_OFF,
    SYSTEM_RESTART,
} system_state;
```

启动初期内核处于 SYSTEM_BOOTING 状态，因此执行 build_all_zonelists_init()函数初始化所有结点借用内存列表，函数在/mm/page_alloc.c 文件内实现：

```
static noinline void __init build_all_zonelists_init(void)
{
    __build_all_zonelists(NULL);      /*/mm/page_alloc.c*/
    mminit_verify_zonelist();         /*输出信息*/
    cpuset_init_current_mems_allowed();
                /*需选择 CPUSETS 选项（/init/Kconfig），/kernel/cpuset.c*/
}
```

若选择了 CONFIG_CPUSETS 配置选项，cpuset_init_current_mems_allowed(void)函数（/kernel/cpuset.c）调用 nodes_setall(current->mems_allowed)函数，在当前进程结构实例（task_struct）的 mems_allowed 位图成员中将所有结点设置为 1。

__build_all_zonelists(NULL)完成所有结点借用内存列表初始化，函数在/mm/page_alloc.c 文件内实现：

```
static int __build_all_zonelists(void *data)
/*data：此处为 NULL*/
{
    int nid;
    int cpu;
    pg_data_t *self = data;
```

```
    #ifdef CONFIG_NUMA
      ...
    #endif

      if (self && !node_online(self->node_id)) {      /*self==NULL，不进入循环*/
          build_zonelists(self);
          build_zonelist_cache(self);
      }

      for_each_online_node(nid) {       /*遍历内存结点*/
          pg_data_t *pgdat = NODE_DATA(nid);

          build_zonelists(pgdat);          /*初始化结点借用内存列表，/mm/page_alloc.c*/
          build_zonelist_cache(pgdat);   /*UMA 系统 pgdat->node_zonelists[0].zlcache_ptr = NULL*/
      }

      for_each_possible_cpu(cpu) {
          setup_pageset(&per_cpu(boot_pageset, cpu), 0);
                                  /*初始化 boot_pageset 实例，源代码请读者自行阅读，/mm/page_alloc.c*/
      #ifdef CONFIG_HAVE_MEMORYLESS_NODES
          if (cpu_online(cpu))
              set_cpu_numa_mem(cpu, local_memory_node(cpu_to_node(cpu)));
      #endif
      }
      return 0;
  }
```

__build_all_zonelists()函数内遍历系统内存结点，调用 build_zonelists(pgdat)函数初始化各结点的借用内存列表，调用 setup_pageset()函数初始化 CPU 单页缓存 boot_pageset 实例。

build_zonelists()函数对 UMA 和 NUMA 系统具有不同的实现，这里以 UMA 为例，函数代码如下：

```
static void build_zonelists(pg_data_t *pgdat)
{
    int node, local_node;
    enum zone_type j;
    struct zonelist *zonelist;

    local_node = pgdat->node_id;      /*当前结点编号*/

    zonelist = &pgdat->node_zonelists[0];          /*借用内存列表指针*/
    j = build_zonelists_node(pgdat, zonelist, 0);    /*将当前结点内存域添加到借用内存列表*/
                                                    /*返回借用内存列表可用项数加 1*/

    for (node = local_node + 1; node < MAX_NUMNODES; node++) { /*遍历当前结点之后的结点*/
```

```
        ...
        j = build_zonelists_node(NODE_DATA(node), zonelist, j);        }
    for (node = 0; node < local_node; node++) {                    /*遍历当前结点之前的结点*/
        ...
        j = build_zonelists_node(NODE_DATA(node), zonelist, j);
    }

    zonelist->_zonerefs[j].zone = NULL;        /*列表最后项为空*/
    zonelist->_zonerefs[j].zone_idx = 0;
}
```

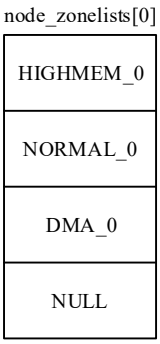
build_zonelists()函数依次对当前结点、之后的结点、之前的结点调用函数 build_zonelists_node()，将各结点内存域实例添加到本结点借用内存列表。对于单结点系统，只调用 build_zonelists_node(pgdat, zonelist, 0)函数。

build_zonelists_node()函数定义如下：

```
static int build_zonelists_node(pg_data_t *pgdat, struct zonelist *zonelist,int nr_zones)
/*pgdat、zonelist：将 pgdat 结点中内存域添加到 zonelist 借用内存列表，
*nr_zones：表示本次新添加内存域在借用列表项中的起始位置（数组项）*/
{
    struct zone *zone;
    enum zone_type zone_type = MAX_NR_ZONES;       /*最大内存域数量*/

    do {                            /*将内存域倒序排列添加到借用内存列表*/
        zone_type--;
        zone = pgdat->node_zones + zone_type;          /*内存域指针*/
        if (populated_zone(zone)) {                    /*内存域不为空，空内存域不添加到列表*/
            zoneref_set_zone(zone,&zonelist->_zonerefs[nr_zones++]);
                                        /*将内存域关联到列表中指定项，/mm/page_alloc.c*/
            check_highest_zone(zone_type); /*UMA 系统为空操作，/include/linux/mempolicy.h*/
        }
    } while (zone_type);

    return nr_zones;     /*列表中下一个可用项索引值*/
}
```

build_zonelists_node()函数将 pgdat 结点中内存域按倒序（内存域编号从大到小）添加到 zonelist 借用内存列表，起始列表项为 nr_zones，函数返回列表中下一个可用项索引值。

下图示意了在单结点系统中的借用内存列表，内存域在列表中从高至低倒序排列：



结点0（单结点）

2 计算空闲页帧数量

build_all_zonelists()随后还要调用 nr_free_pagecache_pages()函数累加超过 watermark[WMARK_HIGH]水印值的空闲页帧数量，函数定义在/mm/page_alloc.c 文件内：

```
unsigned long nr_free_pagecache_pages(void)
{
    return    nr_free_zone_pages(gfp_zone(GFP_HIGHUSER_MOVABLE));     /*/mm/page_alloc.c*/
}
```

nr_free_zone_pages()函数定义在/mm/page_alloc.c，参数为最高内存域的索引值，函数代码如下：

```
static unsigned long nr_free_zone_pages(int offset)
{
    struct zoneref *z;
    struct zone *zone;

    /* Just pick one node, since fallback list is circular */
    unsigned long sum = 0;

    struct zonelist *zonelist = node_zonelist(numa_node_id(), GFP_KERNEL);
                                        /*当前结点列表，/include/linux/gfp.h*/
    for_each_zone_zonelist(zone, z, zonelist, offset) {
                                        /*遍历借用内存列表（包含所有结点中所有内存域）*/
        unsigned long size = zone->managed_pages;        /*内存域管理的页帧数*/
        unsigned long high = high_wmark_pages(zone);
                            /*返回 z->watermark[WMARK_HIGH]值，/include/linux/mmzone.h*/
        if (size > high)
            sum += size - high;        /*累加超过 watermark[WMARK_HIGH]值的页帧数*/
    }
    return sum;
}
```

nr_free_pagecache_pages()函数累加所有结点各内存域管理的页帧中超过 watermark[WMARK_HIGH]水印值的页帧数量。

在启动阶段调用 build_all_zonelists()函数时，还没有调用前面介绍的 init_per_zone_wmark_min()函数设置各内存域的水印值，也就是说此时各内存域 watermark[WMARK_HIGH]水印值应为 0，因此此处累加的

是所有结点各内存域实际管理的页帧数。

build_all_zonelists()函数根据 nr_free_pagecache_pages()函数累加的页帧数（vm_total_pages），确定是否关闭页块迁移属性（设置 page_group_by_mobility_disabled 变量值）。由 build_all_zonelists()函数可知当所有结点各内存域累加的空闲页块数量大于迁移类型数量时，将启用页块迁移属性。

3.6 伙伴系统

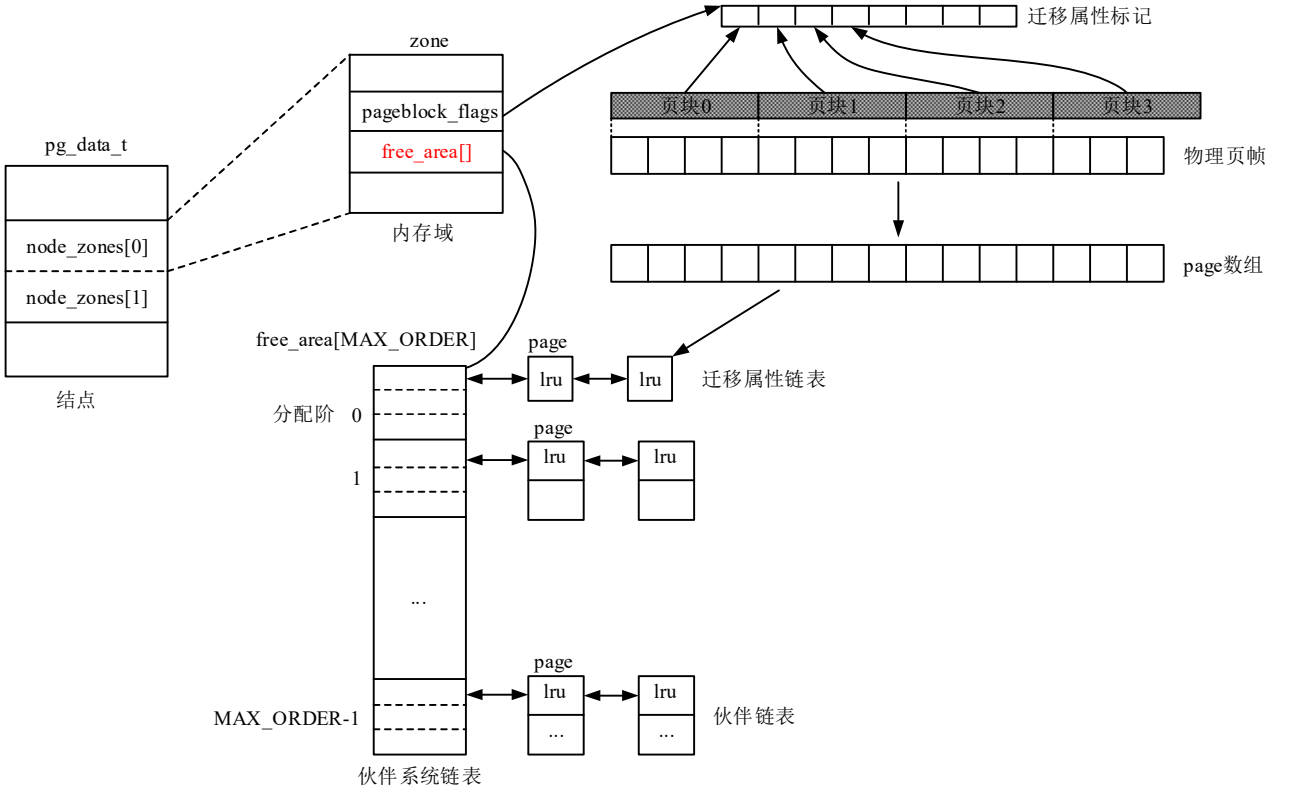
伙伴系统是内核物理内存管理的核心，它按页（多页）管理和分配物理内存，前面介绍的所有内容其实都是在为伙伴系统做准备。伙伴系统通过链表管理各内存域空闲的物理内存，每个链表管理固定大小的连续空闲内存。空闲内存按分配阶 order 进行划分，order 分配阶表示连续 2^{order} 页的空闲物理内存。伙伴系统只能按分配阶分配/释放物理内存。

在内核启动初始阶段，低端内存由自举分配器管理（高端内存暂未管理），伙伴系统链表为空，伙伴系统不可用。随后，自举分配器管理的空闲内存（页帧）及高端内存释放到伙伴系统，停用自举分配器，启用伙伴系统，至此内核开始使用伙伴系统管理物理内存。

本节将介绍释放页至伙伴系统的操作，停用自举分配器启用伙伴系统的过程，重点介绍伙伴系统分配函数的实现。

3.6.1 引言

在前面介绍的物理内存管理数据结构中，物理内存被划分成结点，结点中包含内存域，创建了与结点其所含物理页帧对应的 page 数组，如下图所示。内存域中页帧按页块进行划分，每个页块赋予一个迁移类型属性。



内存域结构体中 free_area[]数组成员，用于按阶（2 的 N 次幂数量的页帧）链接内存域中的空闲内存块（首页）的 page 实例，最大数组项链接的是页块首页 page 实例，其它数组项链接的内存块来源于对页块的划分，内存区不能跨越页块。由于页块定义了迁移属性，free_area[]数组项中对内存块又按迁移属性进

行区分，每种迁移属性对应一个链表成员。

分配内存时，分配函数需要指定分配内存的阶数（多少个内存页）和分配掩码，分配函数根据分配掩码确定从哪个内存域分配内存，如果指定内存域具有足够空闲内存则根据阶数及迁移属性查询空闲内存块链表，从链表中取下内存块即可（page 实例）。如果指定阶数链表中没有空闲内存块，则查找更高阶链表，对更高阶链表中空闲内存块进行拆分，以获取所需的内存块。

如果指定内存域空闲内存不足则搜索借用内存列表，从其它内存域分配内存。如果仍不成功则激活页面回收机制后，再进行分配。

释放函数相对简单点，释放函数将内存块添加到相应阶数的链表中，如果有可以合并的伙伴则合并，合并后内存块添加到更高一阶的链表中，依此向上进行，直至不能再合并或达到最高分配阶为止。

内核启动初期，各内存域伙伴系统链表是空的，低端物理内存由自举分配器管理。内核在启动后期调用 mem_init()函数将自举分配器管理的空闲页帧和高端内存域页帧释放到伙伴系统，从而停用自举分配器，启用伙伴系统，物理内存交由伙伴系统管理。

3.6.2 释放/分配页

在介绍伙伴系统的分配与释放函数前，我们先直接切入主题，看看如何从伙伴系统链表中获取内存块及如何将内存块释放到伙伴系统链表。伙伴系统的管理机制其实是非常简单和高效的，只要确定了从哪个链表分配，分配操作是比较简单的。分配函数的复杂之处在于对分配流程的控制，以及内存紧张时的处置，也就是确定从结点、内存域、伙伴链表比较麻烦。

释放的内存块都是 n 阶大小的内存块，释放内存块中首页的 page 实例中保存了内存块所在的结点、内存域等信息，释放函数将首页 page 实例添加到内存域伙伴链表中，如果能与链表中现有的空闲内存块合并，则合并生成高一阶的空闲内存块，添加到高一阶的链表中，如此逐级往上进行，直到不能合并为止。

这里我们先不考虑如何选择伙伴链表，先讨论在确定内存域及伙伴链表的情况下，如何从中分配和释放内存块。由于伙伴系统链表初始状态为空，在内核启动阶段需要把自举分配器管理的空闲内存及高端内存释放到伙伴系统，而后才能通过伙伴系统进行分配操作。因此下面先介绍释放操作，然后再介绍分配操作。

1 释放页函数

初始化内存域实例时，伙伴系统链表初始化为空，内核在启动阶段将自举分配器管理的空闲页帧及高端内存页帧释放到伙伴系统，从而使其可用，下面介绍释放（多）页内存块函数的实现。释放内存块必须按阶进行，释放内存块的步骤如下：

（1）查找释放内存块的伙伴内存块。查找伙伴内存块的函数为：

```
static inline unsigned long    __find_buddy_index(unsigned long page_idx, unsigned int order)
{
    return page_idx ^ (1 << order);       /*异或操作*/
}
```

page_idx 表示释放内存块首页在页块中的编号（偏移量），order 表示释放内存块的阶数，函数返回值为伙伴内存块首页在页块中的编号（偏移量）。

例如，假设释放 1 阶内存块，页编号（页块内编号）为 2-3，则__find_buddy_index(2,1)函数返回值为 0，表示伙伴内存块的起始页编号为 0，即 0-1 内存块。

（2）判断伙伴内存块是否在伙伴链表中，是则执行第（3）步，否则将内存块添加到当前阶伙伴链表，释放操作结束。

位于伙伴链表中的空闲内存块其首页的 page 实例 private 成员保存内存块的阶数 order，_mapcount 成

员赋值为 PAGE_BUDDY_MAPCOUNT_VALUE（-128，/include/linux/page-flags.h)。内存块移出伙伴链表时首页 page 实例_mapcount 成员赋值-1。内核通过 **PageBuddy**(struct page *page)函数检查内存块是否在伙伴链表中，此函数依据_mapcount 值确定内存块是否在伙伴链表中。

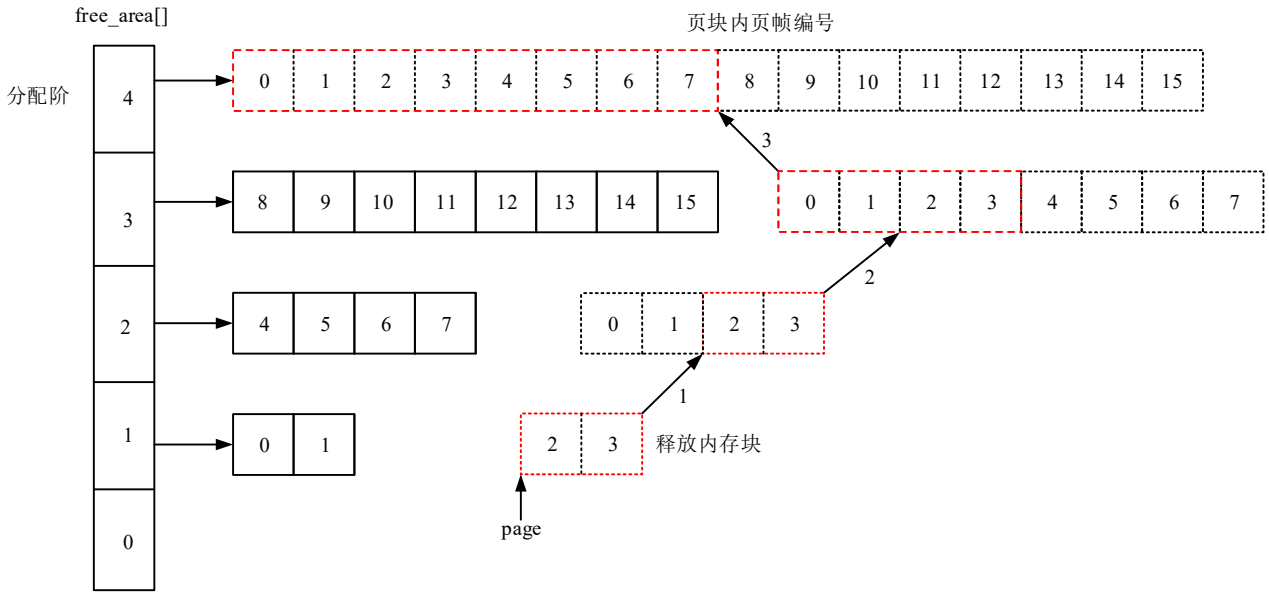
由于所有页帧的 page 实例是一个事先分配好的数组，数组项与页帧一一对应，因此查找伙伴内存块首页的 page 实例并不需要到伙伴链表中去查找。只需要通过前面介绍的__find_buddy_index()函数确定伙伴内存块首页的编号，通过对应关系就能在 page 数组中找到对应的 page 实例，见下面释放函数。

set_page_order(struct page *page, unsigned int order)函数用于标记内存块位于伙伴链表中，page 指向首页的 page 实例，其 private 成员保存内存块的阶数 order，PAGE_BUDDY_MAPCOUNT_VALUE 值赋予_mapcount 成员。

rmv_page_order(struct page *page)函数用于标记内存块已不在伙伴链表中，首页 page 实例 private 成员值清零，_mapcount 成员赋值为-1。

（3）如果伙伴内存块在伙伴链表中，则合并成高一阶内存块后跳至步骤（1）继续执行（逐阶往上合并）。

下面我们来看一个例子，如下图所示，我们依然假设页块大小为 16 个页帧（4 阶），释放 2-3 页帧（1 阶内存块）至伙伴系统：



假设 2-3 页帧所在页块其它内存块都在伙伴链表随，如图中实线框所示，8-15 页帧内存块位于 3 阶链表，4-7 页帧内存块位于 2 阶链表，0-1 页帧内存块位于 1 阶链表。

此时释放 2-3 页帧内存块，将起始页帧编号 2 和阶数 1 代入__find_buddy_index()函数，返回值为 0，表示伙伴内存块起始页帧号为 0（页块内偏移量）。前面介绍过，并不是所有相邻的内存块都是伙伴，只有符合配对关系的内存块才能成为伙伴。由于伙伴内存块在伙伴链表中，则合并成 2 阶内存块，此时起始页帧号为 0，阶数为 2。0-3 页帧内存块的伙伴内存块首页编号为 4，由于其在伙伴链表中，继续合并添加到 3 阶链表。合并生成的 3 阶内存块还可以继续合并生成 4 阶内存块（页块），添加到 4 阶链表中，释放操作结束。图中虚线框表示合并操作步骤。

下面来看一下释放页函数__free_one_page()的实现（/mm/page_alloc.c）：

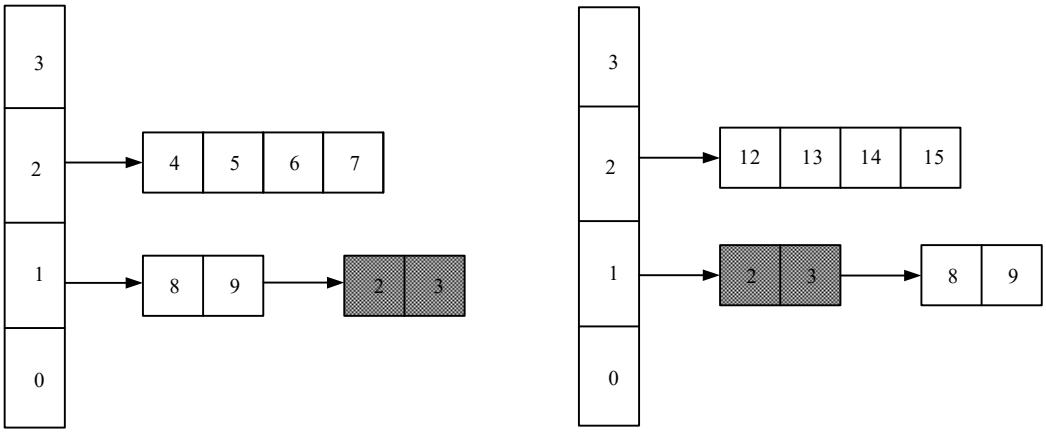
```
static inline void __free_one_page(struct page *page,unsigned long pfn,struct zone *zone,   \
                                                    unsigned int order,    int migratetype)
/*
```

```
  *page：释放内存块首页 page 实例指针，pfn：首页页帧号，zone：所在内存域实例指针，
  *order：内存块阶数，migratetype：迁移属性类型。
  */
{
    unsigned long page_idx;                 /*内存块首页在页块内偏移量（页帧编号）*/
    unsigned long combined_idx;             /*合并后内存块首页页帧编号*/
    unsigned long uninitialized_var(buddy_idx);    /*伙伴内存块首页编号*/
    struct page *buddy;                      /*伙伴内存块首页 page 实例指针*/
    int max_order = MAX_ORDER;          /*内核最大分配阶（加 1）*/

    VM_BUG_ON(!zone_is_initialized(zone));
    VM_BUG_ON_PAGE(page->flags & PAGE_FLAGS_CHECK_AT_PREP, page);

    VM_BUG_ON(migratetype == -1);
    if (is_migrate_isolate(migratetype)) {
        max_order = min(MAX_ORDER, pageblock_order + 1);
    } else {
        __mod_zone_freepage_state(zone, 1 << order, migratetype);
            /*修改内存域空闲页帧数量统计量，/include/linux/vmstat.h*/
    }

    page_idx = pfn & ((1 << max_order) - 1);    /*pfn 转成页块内页帧编号（偏移量）*/

    VM_BUG_ON_PAGE(page_idx & ((1 << order) - 1), page);
    VM_BUG_ON_PAGE(bad_range(zone, page), page);

    while (order < max_order - 1) {          /*逐阶合并内存块*/
        buddy_idx = __find_buddy_index(page_idx, order);
                                        /*查找伙伴内存块首页（编号），/mm/internal.h*/
        buddy = page + (buddy_idx - page_idx);    /*伙伴内存块首页 page 实例指针*/
        if (!page_is_buddy(page, buddy, order))    /*伙伴内存块是否在伙伴链表中*/
            break;                      /*不在伙伴链表中则跳出 while 循环*/

        /*以下为伙伴内存块在伙伴链表中，执行合并操作*/
        if (page_is_guard(buddy)) {
            clear_page_guard(zone, buddy, order, migratetype);
        } else {
            list_del(&buddy->lru);              /*将伙伴内存块从链表移除*/
            zone->free_area[order].nr_free--;
            rmv_page_order(buddy);          /*清除伙伴内存块 buddy 属性，/mm/page_alloc.c*/
        }
        combined_idx = buddy_idx & page_idx;      /*合并后内存块首页编号*/
        page = page + (combined_idx - page_idx);  /*合并后内存块首页 page 实例指针*/
```

```
        page_idx = combined_idx;
        order++;                        /*将阶数加 1，继续执行 while 循环*/
    }   /*while 循环结束*/

    /*将不能再合并的内存块添加到伙伴链表，page 指向首面 page 实例，order 为分配阶*/
    set_page_order(page, order); /*分配阶保存至 private 成员，并设置 buddy 属性，/mm/page_alloc.c*/

    /*判断内存块上一阶的伙伴是否在伙伴链表中，是则添加到链表末尾，否则添加到链表头部*/
    if ((order < MAX_ORDER-2) && pfn_valid_within(page_to_pfn(buddy))) {
        struct page *higher_page, *higher_buddy;
        combined_idx = buddy_idx & page_idx;
        higher_page = page + (combined_idx - page_idx);
        buddy_idx = __find_buddy_index(combined_idx, order + 1);
        higher_buddy = higher_page + (buddy_idx - combined_idx);
        if (page_is_buddy(higher_page, higher_buddy, order + 1)) {
            list_add_tail(&page->lru,&zone->free_area[order].free_list[migratetype]);
                                                    /*内存区添加到链表末尾*/
            goto out;
        }
    }

    list_add(&page->lru, &zone->free_area[order].free_list[migratetype]);
                                                    /*内存块添加到链表头部*/
out:
    zone->free_area[order].nr_free++;           /*增加可用内存块数量计数值*/
}
```

__free_one_page()函数执行如前所述的步骤，查找释放内存块本阶链表的伙伴内存块是否在链表中，是则取下伙伴内存块与释放内存块合并，形成高一阶的内存块，继续往上一阶执行释放操作，直至不能合并或达到最高分配阶为止。然后，将合并形成的内存块（不能再合并的内存块）添加到对应阶的伙伴链表。在添加到链表的过程中进行了一个小小的处理，如果内存块上一阶的伙伴在链表中，则将内存块添加到链表末尾。因为在分配函数中，是从链表的头部取内存块的，添加到链表末尾的内存块被分配的机会减小，当本阶的伙伴内存块被释放后更容易与上一阶的伙伴合并形成更高阶的内存块。如果内存块上一阶的伙伴不在链表中，则直接将内存块添加到链表头部。

举个例子，如下图所示：



如图中所示，释放函数释放 2-3 页帧组成的 1 阶内存块，2-3 页帧本阶的伙伴 0-1 页帧内存块不在链表中，因此不能合并，释放内存块添加到 1 阶链表。左图中 2-3 页帧（合并后的 0-3 页帧内存块）上一阶（2 阶）的伙伴，4-7 页帧内存块在伙伴链表中，2-3 页帧释放到 1 阶链表的末尾，右图中 4-7 页帧不在伙伴链表中，2-3 页帧释放到 1 阶链表的头部。

注意：__free_one_page()函数字面上意思好像是释放单页，实际上是释放 order 阶内存块。

2 分配页函数

从伙伴系统链表中分配内存块的操作要简单一些，这里假设内核已经确定了从哪个内存域哪种迁移属性某阶内存块链表中分配内存，链表中内存块阶数可以比需要分配的内存块阶数更高（需拆分内存块）。例如，分配 2 阶内存块，可能是直接从 2 阶内存块链表中分配，也可能是由 3 阶、4 阶或其它更高阶内存块拆分而得，拆分后剩余的内存块释放回伙伴系统相应阶的链表中。

按阶分配内存页的函数定义如下（/mm/page_alloc.c）：

```
static inline void expand(struct zone *zone, struct page *page, \
                                        int low, int high, struct free_area *area,int migratetype)
/*
 *zone：内存域实例指针，page：调用前指向源内存块首页 page 实例，调用后指向分配得内存块首页，
 *low：分配（申请）内存块阶数，high：从 high 阶内存块中分配，
 *area：high 分配阶对应的 free_area 实例指针，migratetype：迁移属性类型。
 */
{
    unsigned long size = 1 << high;     /*high 阶内存块大小，页帧数*/

    while (high > low) {            /*源内存块阶数比目标内存块分配阶大，需要拆分*/
        area--;             /*降一阶*/
        high--;
        size >>= 1;         /*源内存块对半分后的页帧数*/
        VM_BUG_ON_PAGE(bad_range(zone, &page[size]), &page[size]);

        if (IS_ENABLED(CONFIG_DEBUG_PAGEALLOC) &&
            debug_guardpage_enabled() &&high < debug_guardpage_minorder()) {
            set_page_guard(zone, &page[size], high, migratetype);
            continue;
        }
```

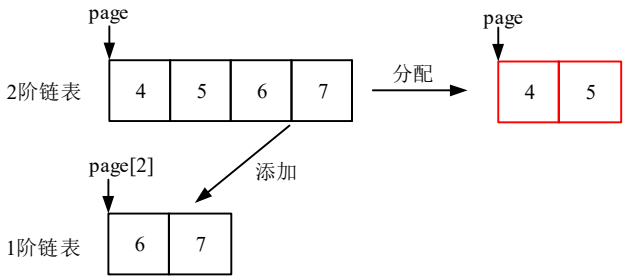
```
            list_add(&page[size].lru, &area->free_list[migratetype]);
                                                    /*拆分后，后半部分内存块添加到低一阶链表中*/
            area->nr_free++;          /*低一阶 free_area 可用内存块数量加 1*/
            set_page_order(&page[size], high);     /*设置后半部分内存块阶数等，/mm/page_alloc.c*/
        }     /*while 循环结束，high=low*/
    }
```

在调用 expand()函数时，以 page 指向 page 实例为首页的内存块已经从伙伴链表中取出，内存块为 high 阶内存块，目标分配阶为 low。

如果 high 等于 low，则不需要拆分，expand()函数不需要执行什么工作，page 为首页的内存块即是分配得内存块。

如果 high 大于 low，则 expand()函数需要对 page 为首页的内存块进行拆分，拆分操作对半对半地进行，每次拆分后的后半部分内存块放回低一阶的伙伴链表，直至拆分出内存块阶数与目标内存块阶数相同。page 始终指向源内存块首页的 page 实例，也就是说分配操作始终把源内存块开头一部分分配出去。

如下图所示，假设从 2 阶链表中（已取出）的 4-7 编号的内存块中分配 1 阶内存块，expand()函数将此内存块对半分，前半部分为分配出去的内存块（4-5 编号），而后半部分内存块（6-7 编号）添加到 1 阶伙伴链表中。



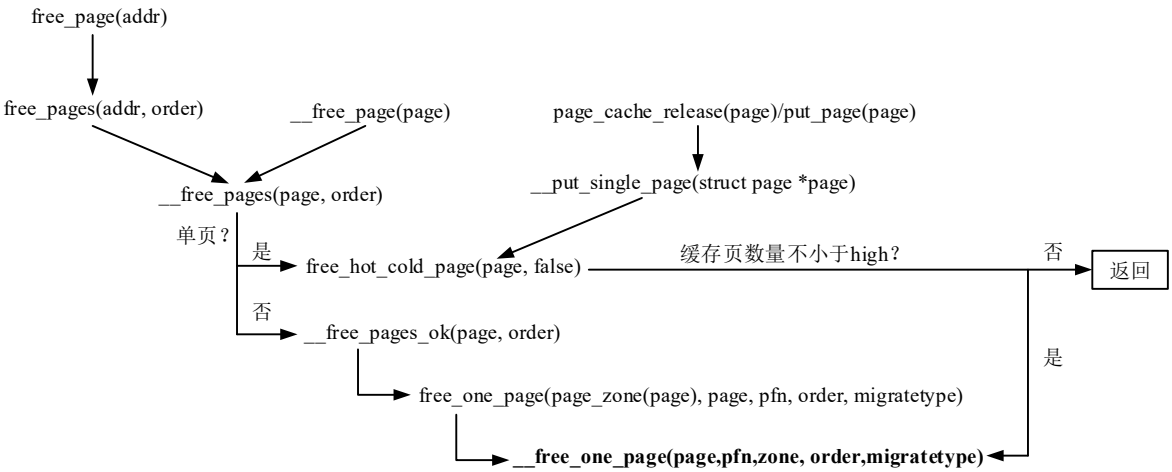
3.6.3 释放函数

前一小节介绍了将指定阶及迁移属性的内存块释放到其所在内存域的__free_one_page()函数，这个函数需要多个参数。内核释放内存块的接口函数是__free_one_page()函数的包装器，它们不需要指定这么多参数，而是在接口函数内确定这些参数值，然后调用__free_one_page()函数。

本小节介绍内核释放内存块接口函数的实现。

1 接口函数

内核释放内存块接口函数调用关系如下图所示：



释放内存区接口函数声明在/include/linux/gfp.h 头文件内：

●**free_page(addr)/free_pages(addr,order)**：用于将起始地址为 addr（虚拟地址）的单页或 order 阶内存区释放到伙伴系统。注意这两个函数只能释放从普通内存域分配给内核使用的内存（映射到内核直接映射区），因为只有映射到内核直接映射区的内存才能由虚拟地址线性映射确定物理地址，进而确定其 page 实例。free_pages()函数定义在/mm/page_alloc.c 文件内。

●**__free_page(page)**：释放 page 实例指定的单页。

●**__free_pages(page,order)**：执行释放操作的核心函数，参数 page 为内存块首页 page 实例指针，order 为内存块阶数，函数定义在/mm/page_alloc.c 文件内。

●**page_cache_release(page)/put_page(page)**：用于释放分配给用户进程的匿名映射页或文件缓存页，函数内确定 page 实例已从 LRU 链表中移除，最后调用 free_hot_cold_page()函数将页释放回伙伴系统。这两个函数声明在头文件/include/linux/pagemap.h，函数定义在/mm/swap.c 文件内。

由以上函数调用关系可知，__free_pages(page, order)函数是伙伴系统所有释放内存块接口函数的执行函数，下面主要介绍此函数的实现，函数定义在/mm/page_alloc.c 文件内：

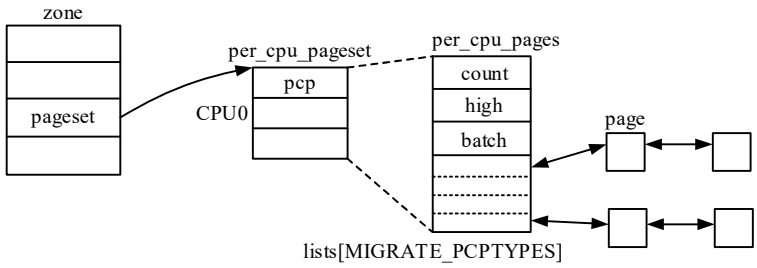
```
void __free_pages(struct page *page, unsigned int order)
{
    if (put_page_testzero(page)) {    /*引用计数_count 减 1 后为 0 的页才能释放，/include/linux/mm.h*/
        if (order == 0)
            free_hot_cold_page(page, false);   /*释放单页到 CPU 缓存（热页）*/
        else
            __free_pages_ok(page, order);     /*释放多页内存块*/
    }
}
```

__free_pages()函数内调用 put_page_testzero(page)函数对 page 实例引用计数_count 值减 1，如果_count 值为 0 则表示页可以被释放。随后判断释放的是单页还是多页，分别调用不同的释放函数，单页优先释放到 CPU 单页缓存中，而多页内存块直接释放到伙伴系统链表，下面将分别介绍释放单页和多页函数的实现。

2 释放单页

内存域 zone 结构 pageset 成员指向 per_cpu_pageset 实例（percpu 变量），如下图所示，用于缓存特定于 CPU 的单页（空闲页，没有添加到伙伴系统链表）。释放单页时并没有直接将其释放到伙伴系统，而

是释放到 CPU 单页缓存，释放后如果缓存中单页数量超过（或等于）high 时，则将释放缓存中 batch 数量的单页到伙伴系统。per_cpu_pageset 结构体中包含部分内存迁移类型的单页链表。



释放单页的函数 free_hot_cold_page()定义在/mm/page_alloc.c 文件内，第一个参数为 page 实例指针，第二个参数表示是否为冷页（真表示冷页，假表示热页）：

```
void free_hot_cold_page(struct page *page, bool cold)
{
    struct zone *zone = page_zone(page);     /*从 page 标记成员获取内存域指针，/include/linux/mm.h*/
    struct per_cpu_pages *pcp;
    unsigned long flags;
    unsigned long pfn = page_to_pfn(page);     /*page 转页帧号*/
    int migratetype;

    if (!free_pages_prepare(page, 0))     /*内存块是否可释放，/mm/page_alloc.c*/
        return;

    migratetype = get_pfnblock_migratetype(page, pfn);   /*从页块位图获取页迁移类型*/
    set_freepage_migratetype(page, migratetype);
                        /*设置迁移类型，page->index = migratetype，/include/linux/mm.h*/
    local_irq_save(flags);
    __count_vm_event(PGFREE);

    /*单缓存中不包含的迁移类型，直接释放到伙伴系统*/
    if (migratetype >= MIGRATE_PCPTYPES) {
        if (unlikely(is_migrate_isolate(migratetype))) {
            free_one_page(zone, page, pfn, 0, migratetype);     /*内部调用__free_one_page()函数*/
            goto out;
        }
        migratetype = MIGRATE_MOVABLE;
    }

    pcp = &this_cpu_ptr(zone->pageset)->pcp;       /*pageset.pcp*/
    if (!cold)                    /*热页添加到链表头部，冷页添加到链表末尾*/
        list_add(&page->lru, &pcp->lists[migratetype]);
    else
        list_add_tail(&page->lru, &pcp->lists[migratetype]);
    pcp->count++;          /*缓存中单页数量加 1*/
```

```
        if (pcp->count >= pcp->high) {      /*链表页数量不小于 high 时，释放 batch 数量页到伙伴系统*/
            unsigned long batch = READ_ONCE(pcp->batch);
            free_pcppages_bulk(zone, batch, pcp);      /*调用__free_one_page()函数，/mm/page_alloc.c*/
                                                       /*释放 batch 数量单页至伙伴系统，注意迁移类型链表的选择*/
            pcp->count -= batch;
        }
    out:
        local_irq_restore(flags);
    }
```

释放单页的 free_hot_cold_page()函数首先调用 free_pages_prepare()函数判断页（内存块）是否可释放，然后获取页迁移类型，如果是 CPU 单页缓存中不包含的迁移类型，则直接将页释放到伙伴系统。如果是缓存包含的迁移类型，则将页释放到相应链表，热页添加到链表头部，冷页添加到链表末尾。添加完页之后判断缓存中单页数量（包含所有迁移类型）是否不小于 high，是则调用 free_pcppages_bulk(zone, batch, pcp)函数释放缓存中 batch 数量的单页到伙伴系统，否则函数返回。

free_pcppages_bulk()函数在/mm/page_alloc.c 文件内实现，其最终调用__free_one_page()函数将选择释放的页释放到伙伴系统，需要注意的是选择页时还需要考虑从哪种迁移类型链表中选择页，源代码请读者自行阅读。

前面提到的 free_pages_prepare()函数用于判断单页（或内存块）是否可释放，定义在/mm/page_alloc.c 文件内：

```
static bool free_pages_prepare(struct page *page, unsigned int order)
{
  bool compound = PageCompound(page);   /*是否为复合页（设置 PG_head 或 PG_tail 标记位）*/
  int i, bad = 0;

  VM_BUG_ON_PAGE(PageTail(page), page);
  VM_BUG_ON_PAGE(compound && compound_order(page) != order, page);

  trace_mm_page_free(page, order);
  kmemcheck_free_shadow(page, order);
  kasan_free_pages(page, order);

  if (PageAnon(page))            /*是否为匿名映射页*/
      page->mapping = NULL;
  bad += free_pages_check(page);
  for (i = 1; i < (1 << order); i++) {   /*检查复合页*/
      if (compound)
          bad += free_tail_pages_check(page, page + i);      /*尾页是否可释放*/
      bad += free_pages_check(page + i);
  }
  if (bad)            /*具有不可释放的页*/
      return false;

  reset_page_owner(page, order);     /*/include/linux/page_owner.h*/
```

```
    if (!PageHighMem(page)) {
        debug_check_no_locks_freed(page_address(page), PAGE_SIZE << order);
        debug_check_no_obj_freed(page_address(page), PAGE_SIZE << order);
    }
    arch_free_page(page, order);
    kernel_map_pages(page, 1 << order, 0);              /*空操作，/include/linux/mm.h(L2133)*/

    return true;
}
```

free_pages_prepare()函数主要是检查 page 指示的内存块是否具有不可释放的页，函数返回真表示内存块所有页帧可释放，否则返回假。

释放单页时优先将其释放到单页缓存中，而不是伙伴系统，分配页时将优先从单页缓存中分配。因为内核对用户进程总是按单页分配内存的，这样做可以提高分配的效率。

3 释放多页

在释放函数__free_pages()中，若释放内存块的阶数大于 0（多页），__free_pages()函数内将调用函数__free_pages_ok(page, order)释放内存块，函数定义在/mm/page_alloc.c 文件内：

```
static void __free_pages_ok(struct page *page, unsigned int order)
/*page：指向内存块首页 page 实例，order：阶数*/
{
    unsigned long    flags;
    int    migratetype;
    unsigned long pfn = page_to_pfn(page);       /*page 转页帧号*/

    if (!free_pages_prepare(page, order))       /*内存块是否可释放*/
       return;

    migratetype = get_pfnblock_migratetype(page, pfn);       /*获取迁移类型*/
    local_irq_save(flags);
    __count_vm_events(PGFREE, 1 << order);
    set_freepage_migratetype(page, migratetype);     /*page->index = migratetype，/include/linux/mm.h*/
    free_one_page(page_zone(page), page, pfn, order, migratetype);   /*/mm/page_alloc.c*/
    local_irq_restore(flags);
}
```

__free_pages_ok()函数内调用 free_one_page()函数完成释放操作（/mm/page_alloc.c）：

```
static void free_one_page(struct zone *zone,struct page *page, unsigned long pfn,unsigned int order, \
                                                                    int migratetype)
{
    unsigned long nr_scanned;
    spin_lock(&zone->lock);
    nr_scanned = zone_page_state(zone, NR_PAGES_SCANNED);
```

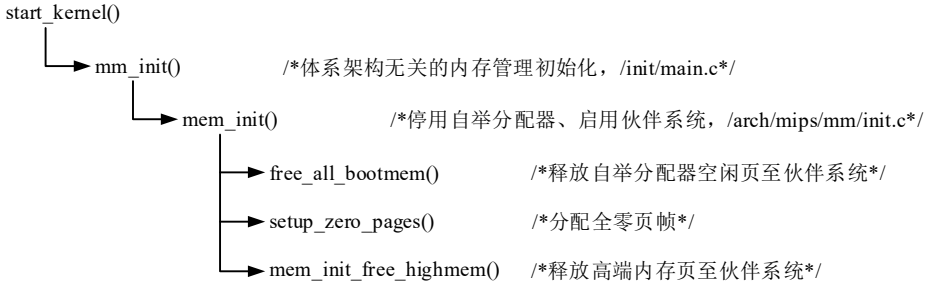
```
    if (nr_scanned)                    /*更新统计量*/
        __mod_zone_page_state(zone, NR_PAGES_SCANNED, -nr_scanned);

    if (unlikely(has_isolate_pageblock(zone) ||is_migrate_isolate(migratetype))) {
        migratetype = get_pfnblock_migratetype(page, pfn);
    }
    __free_one_page(page, pfn, zone, order, migratetype);      /*释放多页至伙伴系统*/
    spin_unlock(&zone->lock);
}
```

free_one_page()函数内调用前面介绍的__free_one_page()函数直接将内存块释放到伙伴系统链表。

3.6.4 启用伙伴系统

内核启动初期物理内存由自举分配器管理，在物理内存管理数据结构初始化完成后，将释放自举分配器管理的空闲页以及高端内存页至伙伴系统，停用自举分配器，启用伙伴系统，随后物理内存将由伙伴系统管理。

停用自举分配器、启用伙伴系统由 mem_init()函数实现，函数调用关系如下图所示：



mm_init()函数在/init/main.c 文件内实现，代码如下：

```
static void __init mm_init(void)
{
    page_ext_init_flatmem();
    mem_init();                /*停用自举分配器、启用伙伴系统，/arch/mips/mm/init.c*/
    kmem_cache_init();         /*初始化 slab 分配器，下一节介绍*/
    percpu_init_late();        /*percpu 变量后期初始化，见 3.8 节*/
    pgtable_init();            /*页表初始化，见第 4 章，/include/linux/mm.h*/
    vmalloc_init();            /*初始化内核 VMALLOC 虚拟内存区，见第 4 章*/
    ioremap_huge_init();
}
```

本节主要介绍其中的 mem_init()函数，它负责停用自举分配器、启用伙伴系统，这是一个体系结构相关的函数，MIPS 架构函数定义在/arch/mips/mm/init.c 文件内，代码如下：

```
void __init mem_init(void)
{
  #ifdef CONFIG_HIGHMEM          /*支持高端内存域*/
    ...
    max_mapnr = highend_pfn ? highend_pfn : max_low_pfn;   /*max_mapnr 表示内存最大页帧号*/
  #else
    max_mapnr = max_low_pfn;      /*若不支持高端内存域，max_mapnr 为低端内存域最大页帧号*/
```

```
    #endif

    high_memory = (void *) __va(max_low_pfn << PAGE_SHIFT);     /*high_memory 为全局变量*/
                        /*max_low_pfn 映射到内核直接映射区的虚拟地址，/arch/mips/include/asm/page.h*/
    maar_init();                /*完成处理器 MAAR 寄存器操作，龙芯 1B 为空操作，/arch/mips/mm/init.c*/
    free_all_bootmem();         /*释放自举分配器空闲页，/mm/bootmem.c*/
    setup_zero_pages();         /*分配全零页帧，赋予全局变量，/arch/mips/mm/init.c*/
    mem_init_free_highmem();    /*释放高端内存页至伙伴系统，/arch/mips/mm/init.c*/
    mem_init_print_info(NULL);
#ifdef CONFIG_64BIT
    ...
#endif
}
```

内核在/arch/mips/include/asm/page.h 头文件定义了以下两个宏：

```
#define    __pa(x)     ((unsigned long)(x) - PAGE_OFFSET + PHYS_OFFSET)
#define    __va(x)    ((void *)((unsigned long)(x) + PAGE_OFFSET - PHYS_OFFSET))
```

这两个宏用于低端内存物理地址与其映射到内核直接映射区的虚拟地址之间的转换。物理地址与虚拟地址之间是线性映射的关系，虚拟地址相对于物理地址的偏移量为 PAGE_OFFSET -PHYS_OFFSET。

__pa(x)宏返回虚拟地址 x 对应的物理地址，__va(x)宏返回物理地址 x 对应的虚拟地址，注意这种转换仅限于低端内存中映射到内核直接映射区的内存。

mem_init()函数调用 free_all_bootmem()函数释放自举分配器中空闲页至伙伴系统，setup_zero_pages()函数用于申请全零的空闲页帧并将其 page 实例赋予全局变量，供内核使用，mem_init_free_highmem()函数用于释放高端内存至伙伴系统。下面将分别介绍以上 3 个主要函数的实现。

1 释放低端内存

free_all_bootmem()函数释放自举分配器管理的空闲低端内存页至伙伴系统，函数在/mm/bootmem.c 文件内实现，返回释放的空闲页帧数，函数代码如下：

```
unsigned long __init free_all_bootmem(void)
{
    unsigned long total_pages = 0;
    bootmem_data_t *bdata;
    reset_all_zones_managed_pages();   /*所有 zone 实例 managed_pages 成员清 0，/mm/bootmem.c*/

    list_for_each_entry(bdata, &bdata_list, list)          /*遍历所有结点*/
        total_pages += free_all_bootmem_core(bdata); /*释放自举分配器空闲页，/mm/bootmem.c*/

    totalram_pages += total_pages;
    return total_pages;         /*返回释放页数量*/
}
```

free_all_bootmem()函数将所有结点中所有内存域 zone 实例 managed_pages 成员清 0，对各结点调用函数 free_all_bootmem_core()释放结点自举分配器管理的空闲页至伙伴系统，最后返回所有结点释放空闲页

总数量。

free_all_bootmem_core()函数定义在/mm/bootmem.c 文件内，代码如下：

```
static unsigned long __init    free_all_bootmem_core(bootmem_data_t *bdata)
{
    struct page *page;
    unsigned long *map, start, end, pages, cur, count = 0;

    if (!bdata->node_bootmem_map)
        return 0;

    map = bdata->node_bootmem_map;        /*自举分配器位图指针（整型数组）*/
    start = bdata->node_min_pfn;          /*结点位于低端内存最小页帧号*/
    end = bdata->node_low_pfn;            /*结点位于低端内存最大页帧号*/
    ...
    /*以下 while 循环用于释放自举分配器标记的空闲页*/
    while (start < end) {          /*start 表示每次释放时的起始页帧号*/
        unsigned long idx, vec;
        unsigned shift;

        idx = start - bdata->node_min_pfn;        /*释放页偏移量，相对于位图的起始页的偏移量*/
        shift = idx & (BITS_PER_LONG - 1);
                        于          /*释放页偏移量，相对于所在整型数最低位表示页的偏移量*/
        vec = ~map[idx / BITS_PER_LONG];      /*释放页标记位所在整型数，按位取反*/

        if (shift) {        /*释放页不是整型数最低位表示的页*/
            vec >>= shift;
            if (end - start >= BITS_PER_LONG)
                vec |= ~map[idx / BITS_PER_LONG + 1] <<(BITS_PER_LONG - shift);
          /*vec 是一个整型数，其值为从 start 页开始连续 BITS_PER_LONG 页的位图值按位取反*/
        }

        /*起始页帧号为 BITS_PER_LONG 对齐，且具有连续的 BITS_PER_LONG 数量的空闲页帧*/
        if (IS_ALIGNED(start, BITS_PER_LONG) && vec == ~0UL) {
            int order = ilog2(BITS_PER_LONG);          /*BITS_PER_LONG 数量页帧内存块的阶数*/

            __free_pages_bootmem(pfn_to_page(start), start, order);   /*释放 order 阶内存块*/
            count += BITS_PER_LONG;
            start += BITS_PER_LONG;
        } else {
          /*起始页帧号不是 BITS_PER_LONG 对齐，或不具有连续的 BITS_PER_LONG
          *数量的空闲页帧，逐页释放页。*/
            cur = start;
            start = ALIGN(start + 1, BITS_PER_LONG);
```

```
                    /*下次扫描起始页帧号 BITS_PER_LONG 对齐（上对齐），不是偏移量*/
            while (vec && cur != start) {       /*逐页释放*/
                if (vec & 1) {
                    page = pfn_to_page(cur);
                    __free_pages_bootmem(page, cur, 0);
                    count++;
                }
                vec >>= 1;
                ++cur;
            }
        }
    }

    /*以下是释放自举分配器位图占用的页*/
    cur = bdata->node_min_pfn;
    page = virt_to_page(bdata->node_bootmem_map);
    pages = bdata->node_low_pfn - bdata->node_min_pfn;
    pages = bootmem_bootmap_pages(pages);
    count += pages;
    while (pages--)
        __free_pages_bootmem(page++, cur++, 0);       /*逐页释放*/

    bdebug("nid=%td released=%lx\n", bdata - bootmem_node_data, count);

    return count;
}
```

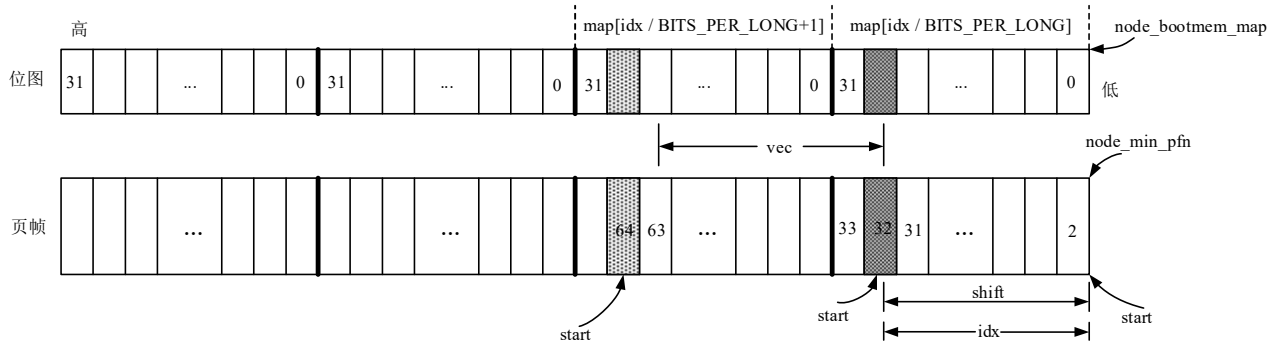
free_all_bootmem_core()函数主要分两步，第一步是释放自举分配器中标记为 0 的空闲页，第二步是释放自举分配器位图占用的页。

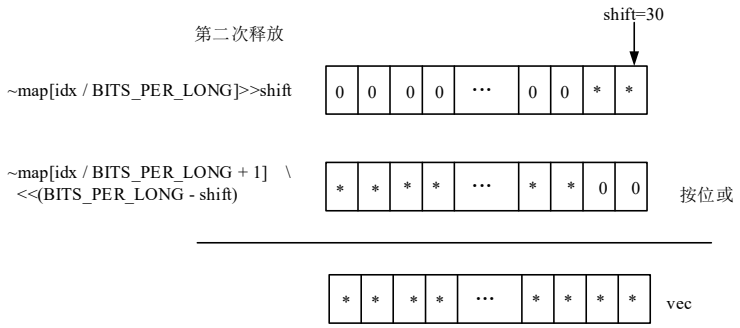
先介绍第一步，自举分配器位图实际上是整型数组，每个整型数代表的页帧数量为 BITS_PER_LONG（整型数位宽），第一步中扫描位图，逐段释放空闲页帧，每段的页帧数量为 BITS_PER_LONG，并且每段的起始页帧号 BITS_PER_LONG 对齐（除第一段外）。

例如，如下图所示，假设起始页帧号为 2，其对应位图中的最低位，BITS_PER_LONG 为 32。第一次扫描时释放 2-31 页帧，第二次扫描时释放 32-63 页帧，第三次时释放 64-95 页帧，依此类推。

注意，每次释放的 32 个页帧并不一定是与位图中的整数一一对应的，例如下图中就错开了 2 位。除第一次和最后一次释放的页帧外，其它每次释放的页帧标记位会跨越 2 个整数。

在 free_all_bootmem_core()函数中 idx 表示每次释放的起始页帧相对于位图开始的偏移量，shift 表示起始页帧相对于其标记位所在整数最低位的偏移量。如下图所示，第一次释放时 idx、shift 都为 0，第二次释放时 idx、shift 都为 30，第三次释放时 idx 为 62，shift 为 30。





vec 值为从释放的起始页帧开始，BITS_PER_LONG 数量页帧的标记位值，它由位图中两个相邻的整数拼凑而成。即起始页帧标记位所在位图中整数按位取反后右移 shift 位与下一个整数按位取反后左移(BITS_PER_LONG - shift)位，按位或所得的结果。vec 中为 1 的位表示页帧空闲，为 0 的位表示页帧已被使用。

如果起始页帧号 BITS_PER_LONG 对齐，且 vec 各位值为全 1，表示连续的 BITS_PER_LONG 数量的空闲页帧，则按其阶数释放内存块。如果不是连续的 BITS_PER_LONG 数量空闲页帧，或起始页帧号不是 BITS_PER_LONG 对齐，则逐页释放。下一次释放时，起始页帧号增加 BITS_PER_LONG（对齐）。

第二步中对位图占用的页帧逐页进行释放，两步中调用的释放函数都是__free_pages_bootmem()，函数在/mm/page_alloc.c 文件内定义，代码如下：

```
void __init    __free_pages_bootmem(struct page *page, unsigned long pfn,unsigned int order)
/*page：首页 page 实例指针，pfn：页帧号，order：阶数*/
{
    if (early_page_uninitialised(pfn))     /*需配置 DEFERRED_STRUCT_PAGE_INIT，否则返回 false*/
        return;
    return    __free_pages_boot_core(page, pfn, order);     /*/mm/page_alloc.c*/
}
```

__free_pages_boot_core()函数定义如下：

```
static void __init    __free_pages_boot_core(struct page *page,unsigned long pfn, unsigned int order)
{
    unsigned int nr_pages = 1 << order;     /*内存块页帧数*/
    struct page *p = page;
    unsigned int loop;
```

```
        prefetchw(p);
        for (loop = 0; loop < (nr_pages - 1); loop++, p++) {     /*逐页初始化*/
            prefetchw(p + 1);
            __ClearPageReserved(p);     /*清除 PG_reserved 标记*/
            set_page_count(p, 0);          /*设置 page 实例_count 成员值为 0，/mm/internal.h*/
        }
        __ClearPageReserved(p);     /*初始化最后页*/
        set_page_count(p, 0);

        page_zone(page)->managed_pages += nr_pages;         /*增加伙伴系统管理的页帧数量*/
        set_page_refcounted(page);       /*设置 page 实例_count 成员为 1，/mm/internal.h*/
        __free_pages(page, order);         /*释放页接口函数，见上一小节*/
    }
```

__free_pages_boot_core()函数逐页清除 page 实例 PG_reserved 标记，设置_count 成员为 1，最后调用__free_pages(page, order)释放函数将空闲内存块释放到伙伴系统链表。

2 分配全零页帧

mem_init()函数内调用 setup_zero_pages()函数为内核分配全局的全零内存块，以供内核使用，函数定义在/arch/mips/mm/init.c 文件内：

```
    unsigned long empty_zero_page, zero_page_mask;
    EXPORT_SYMBOL_GPL(empty_zero_page);         /*全零内存块起始虚拟地址*/
    EXPORT_SYMBOL(zero_page_mask);                  /*全零内存块大小掩码*/

    void setup_zero_pages(void)
    {
        unsigned int order, i;
        struct page *page;

        if (cpu_has_vce)         /*确定分配阶*/
            order = 3;
        else
            order = 0;

        empty_zero_page = __get_free_pages(GFP_KERNEL | __GFP_ZERO, order);
                                                        /*分配全零页帧，分配函数见下一小节*/
        if (!empty_zero_page)
            panic("Oh boy, that early out of memory?");

        page = virt_to_page((void *)empty_zero_page);              /*指向首页 page 实例*/
        split_page(page, order);       /*设置 page 引用计数，所有者等，/mm/page_alloc.c*/
        for (i = 0; i < (1 << order); i++, page++)
            mark_page_reserved(page);       /*设置为预留页，/include/linux/mm.h*/
```

```
    zero_page_mask = ((PAGE_SIZE << order) - 1) & PAGE_MASK;      /*全零内存块大小掩码*/
}
```

3 释放高端内存

mem_init()函数已经将自举分配器空闲页帧释放到伙伴系统，自举分配器只管理低端内存，因此还需要将高端内存释放到伙伴系统。mem_init_free_highmem()函数定义在/arch/mips/mm/init.c 文件内，完成高端内存的释放，代码如下：

```
static inline void mem_init_free_highmem(void)
{
 #ifdef CONFIG_HIGHMEM        /*配置支持高端内存*/
    unsigned long tmp;
    for (tmp = highstart_pfn; tmp < highend_pfn; tmp++) {     /*逐页释放高端内存*/
        struct page *page = pfn_to_page(tmp);        /*页帧号转换 page 实例指针*/

        if (!page_is_ram(tmp))
            SetPageReserved(page);        /*非 RAM 内存设置为保留*/
        else
            free_highmem_page(page);      /*逐页释放高端内存，体系结构无关函数*/
    }
 #endif
}
```

free_highmem_page(page)函数在/mm/page_alloc.c 文件内实现，用于逐页释放高端内存到伙伴系统：

```
#ifdef CONFIG_HIGHMEM
void free_highmem_page(struct page *page)
{
    __free_reserved_page(page);            /*/include/linux/mm.h*/
    totalram_pages++;                      /*更新管理页帧数量*/
    page_zone(page)->managed_pages++;
    totalhigh_pages++;
}
#endif
```

__free_reserved_page(page)函数定义在/include/linux/mm.h 头文件：

```
static inline void __free_reserved_page(struct page *page)
{
    ClearPageReserved(page);   /*清除页保留标记*/
    init_page_count(page);      /*_count 成员置 1，/include/linux/mm.h*/
    __free_page(page);          /*释放页接口函数*/
}
```

__free_page(page)函数完成单页的释放，函数定义在/include/linux/gfp.h 头文件：

```
#define   __free_page(page)   __free_pages((page), 0)
```

至此系统内所有空闲页帧都释放到伙伴系统，自举分配器位图也被释放，自举分配器被抛弃不可使用，启用伙伴系统。

4 释放初始化段

内核只在初始化阶段使用的函数和变量被链接到指定的段中（初始化段），段起始地址为__init_begin，结束地址为__init_end，链接操作保证起止地址都是页对齐的。

内核在启动阶段后期将释放初始化段占用的内存至伙伴系统，因为这些函数和变量在后面将不再使用了。在内核创建的 kernel_init 内核线程中将调用 free_initmem()函数释放初始化段占用的内存。

free_initmem()函数定义在/arch/mips/mm/init.c 文件内，代码如下：

```
void __init_refok free_initmem(void)
{
    prom_free_prom_memory();   /*空操作，/arch/mips/loongson32/common/prom.c*/
    if (free_init_pages_eva)          /*空指针，/arch/mips/mm/init.c*/
        free_init_pages_eva((void *)&__init_begin, (void *)&__init_end);
    else
        free_initmem_default(POISON_FREE_INITMEM);   /*POISON_FREE_INITMEM=0xCC*/
                                        /*默认的释放函数，/include/linux/mm.h*/
}
```

如果体系结构没有定义 free_init_pages_eva()函数，则调用默认的释放函数 free_initmem_default()，参数 POISON_FREE_INITMEM 定义在/include/linux/poison.h 头文件。free_initmem_default()函数定义如下：

```
static inline unsigned long free_initmem_default(int poison)
{
    extern char __init_begin[], __init_end[];    /*初始化段起止地址，虚拟地址*/

    return free_reserved_area(&__init_begin, &__init_end,poison, "unused kernel");
                                                        /*/mm/page_alloc.c*/
}
```

free_reserved_area()函数定义在/mm/page_alloc.c 文件内：

```
unsigned long free_reserved_area(void *start, void *end, int poison, char *s)
{
    void *pos;
    unsigned long pages = 0;

    start = (void *)PAGE_ALIGN((unsigned long)start);    /*地址页对齐*/
    end = (void *)((unsigned long)end & PAGE_MASK);
    for (pos = start; pos < end; pos += PAGE_SIZE, pages++) {   /*逐页释放*/
        if ((unsigned int)poison <= 0xFF)
            memset(pos, poison, PAGE_SIZE);    /*页内容填充 0xCC*/
        free_reserved_page(virt_to_page(pos));  /*虚拟地址转 page，/include/linux/mm.h*/
    }

    if (pages && s)
```

```
            pr_info("Freeing %s memory: %ldK (%p - %p)\n",s, pages << (PAGE_SHIFT - 10), start, end);

        return pages;   /*返回释放的页帧数量*/
    }
```

free_reserved_area()函数逐页对页帧写入 0xCC（POISON_FREE_INITMEM）值，然后调用释放函数 free_reserved_page()释放页帧，函数定义在/include/linux/mm.h 头文件：

```
    static inline void free_reserved_page(struct page *page)
    {
        __free_reserved_page(page);      /*同释放高端内存页帧函数*/
        adjust_managed_page_count(page, 1);      /*增加内存域管理页帧数，/mm/page_alloc.c*/
    }
```

free_reserved_page()函数内调用前面释放高端内存页帧时使用的__free_reserved_page()函数释放页帧，然后对 page 所在内存域 zone 实例中 managed_pages 成员加 1，全局变量 totalram_pages 加 1，表示系统多了一个页帧可用。

3.6.5 内存分配

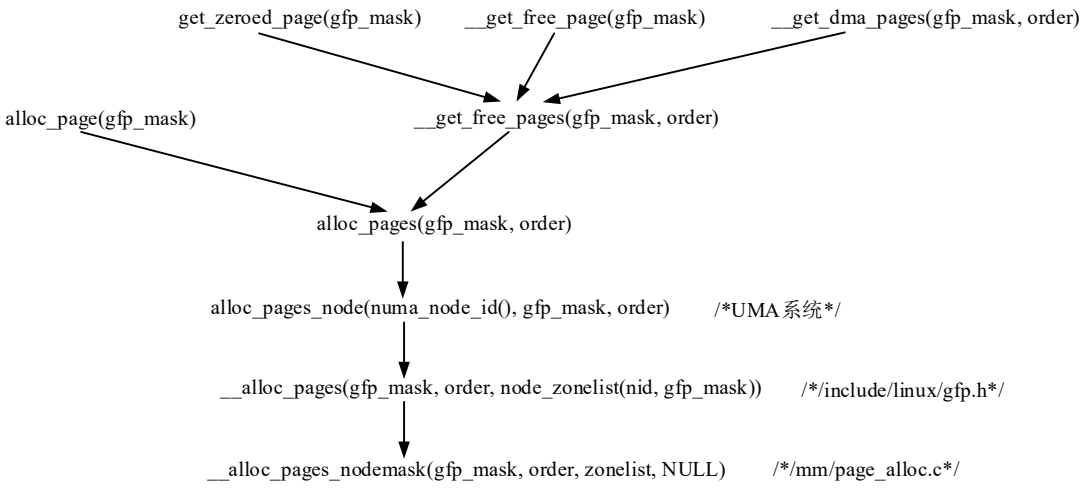
伙伴系统最重要的工作就是内存分配。内存分配是一项复杂而艰巨的任务，分配操作的难点不在于从伙伴链表中摘取内存块，而在于分配流程的控制，如何确定从哪个伙伴链表分配内存，尤其空闲内存紧张和不足时的处置。本节详细介绍伙伴系统分配函数的实现。

1 前言

下面先介绍内核提供的从伙伴系统分配内存的接口函数，以及控制分配流程的分配掩码，后面将详细介绍分配函数的实现。

■接口函数

下图列出了内核提供的内存分配接口函数调用关系：



以下各分配函数定义在/include/linux/gfp.h 头文件或/mm/page_alloc.c 文件内：

●**alloc_pages(mask,order)**：分配 2^{order} 页帧并返回首页 page 实例指针，此函数是各分配函数的核心。

●**alloc_page(mask)**：只分配一个页帧，调用 alloc_pages(mask,0)函数实现，返回 page 实例指针。

●**get_zeroed_page(mask)**：分配一个全零页帧，返回页帧在内核直接映射区虚拟地址。

●**__get_free_page(mask)**：分配一个页帧，返回页帧在内核直接映射区虚拟地址。

●**__get_free_pages(mask,order)**：分配 2^{order} 页，不清零，返回首页在内核直接映射区虚拟地址，函数定义在/mm/page_alloc.c 文件内。

●**__alloc_pages_nodemask(gfp_mask, order,zonelist, nodemask)**：函数定义在/mm/page_alloc.c 文件内，它是伙伴系统分配内存的核心实现函数。

注意：__get_free_pages(mask,order)函数及调用它的接口函数，只能用来分配低端内存，函数返回分配页帧在内核直接映射区的虚拟地址，不能用来分配高端内存。

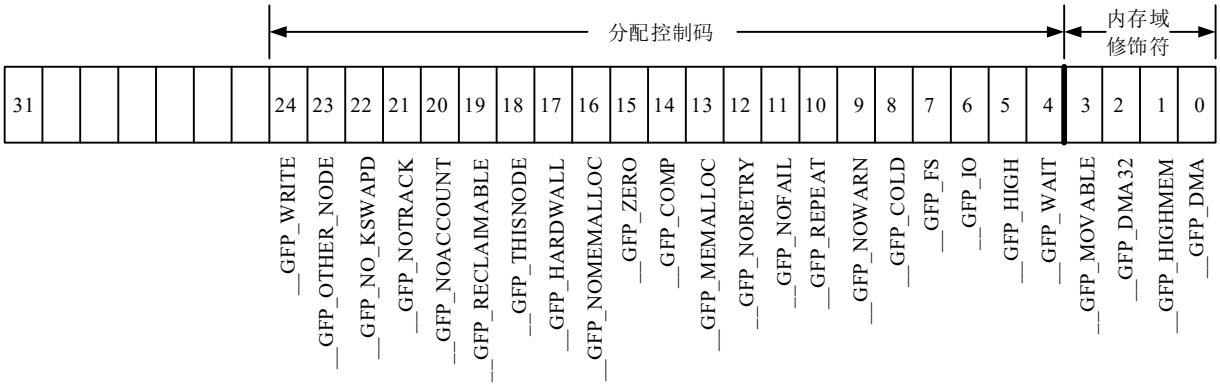
__get_free_pages(mask,order)函数在调用 alloc_pages(mask,order)函数之后调用 page_address(page)函数将 page 实例转换成其映射到内核空间的起始虚拟地址。page_address(page)函数还可用来处理内核间接映射区 page 实例与虚拟地址之间的转换，下一章再做介绍。

■分配掩码

在分配函数中通过分配掩码 gfp_mask 参数来控制分配操作的流程。分配掩码参数类型定义在头文件/include/linux/types.h：

typedef unsigned __bitwise__ gfp_t;

分配掩码各比特位布局如下图所示：



分配掩码低 4 位用于选择内存域，其余位用于控制分配流程，各位定义在/include/linux/gfp.h 头文件内：

```
__GFP_DMA          /*从 DMA 内存域分配*/
__GFP_HIGHMEM      /*从高端内存域分配*/
__GFP_DMA32        /*从 DMA32 内存域分配*/
__GFP_MOVABLE      /*从可移动内存域分配*/
                   /*低 4 位为 0 表示从普通内存域分配*/

__GFP_WAIT         /*分配过程中可以等待和重调度 */
__GFP_HIGH         /*分配等级高，可访问紧急分配池*/
__GFP_IO           /*可以启动物理 IO*/
__GFP_FS           /*可以进入底层文件系统*/
__GFP_COLD         /*需要缓存的冷页*/
```

```
__GFP_NOWARN          /*禁止分配失败警告*/
__GFP_REPEAT          /*努力实现内存分配，可能失败*/
__GFP_NOFAIL          /*一直重试，不能失败*/
__GFP_NORETRY         /*不重试，可能失败*/
__GFP_MEMALLOC        /*允许忽略水印值进行分配，可从预留内存中分配*/
__GFP_COMP            /*增加复合页元数据*/
__GFP_ZERO            /*清零分配的内存块，返回全 0 页*/
__GFP_NOMEMALLOC      /*不允许忽略水印值分配，不可访问预留的内存*/
__GFP_HARDWALL        /*只在允许进程运行的 CPU 关联的结点分配*/
__GFP_THISNODE        /*没有备用结点，没有策略*/
__GFP_RECLAIMABLE     /*分配页可被回收*/
__GFP_NOACCOUNT       /* Don't account to kmemcg */
__GFP_NOTRACK         /* Don't track with kmemcheck */
__GFP_NO_KSWAPD       /*不激活页回收（页交换）机制*/
__GFP_OTHER_NODE      /*从其它结点分配*/
__GFP_WRITE           /*分配器试图更改分配页内容，使其变脏页*/
```

分配函数一般不直接使用以上标记位而是使用以上标记位的组合，/include/linux/gfp.h 头文件内定义了内核常用的标记组合：

```
#define   GFP_NOWAIT   (GFP_ATOMIC & ~__GFP_HIGH)        /*常数 0*/
#define   GFP_ATOMIC   (__GFP_HIGH)        /*分配等级较高，使用紧急内存池*/
#define   GFP_NOIO     (__GFP_WAIT)
#define   GFP_NOFS     (__GFP_WAIT | __GFP_IO)
#define   GFP_KERNEL   (__GFP_WAIT | __GFP_IO | __GFP_FS)
                       /*内核常用分配掩码，只能从普通内存域（或低级内存域）分配*/
#define   GFP_TEMPORARY    (__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_RECLAIMABLE)
#define   GFP_USER     (__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HARDWALL)
                       /*为用户进程分配页帧掩码，从普通（或更低）内存域分配*/
#define   GFP_HIGHUSER     (GFP_USER | __GFP_HIGHMEM)      /*为用户进程分配高端内存*/
#define   GFP_HIGHUSER_MOVABLE    (GFP_HIGHUSER | __GFP_MOVABLE)
#define   GFP_IOFS     (__GFP_IO | __GFP_FS)
#define   GFP_MOVABLE_MASK      (__GFP_RECLAIMABLE|__GFP_MOVABLE)
#define   GFP_DMA          __GFP_DMA
...
```

这里再介绍两个函数，一是由分配掩码确定分配内存的迁移属性类型的函数，另一个是由分配掩码确定从哪个内存域分配内存的函数。

由分配掩码确定分配内存迁移属性的函数为 gfpflags_to_migratetype(gfp_flags)，函数定义如下：

```
static inline int    gfpflags_to_migratetype(const gfp_t gfp_flags)              /*/include/linux/gfp.h*/
{
    WARN_ON((gfp_flags & GFP_MOVABLE_MASK) == GFP_MOVABLE_MASK);

    if (unlikely(page_group_by_mobility_disabled))    /*如果不支持迁移属性，返回不可迁移类型*/
```

```
        return    MIGRATE_UNMOVABLE;

    return (((gfp_flags & __GFP_MOVABLE) != 0) << 1) |((gfp_flags & __GFP_RECLAIMABLE) != 0);
}
```

这里再次列出内存迁移属性的定义：

```
enum {
    MIGRATE_UNMOVABLE,          /*不可移动页，0*/
    MIGRATE_RECLAIMABLE,        /*可回收页，1*/
    MIGRATE_MOVABLE,            /*可移动页，2*/
    MIGRATE_PCPTYPES,           /*CPU 单页缓存中页链表数，3*/
    MIGRATE_RESERVE = MIGRATE_PCPTYPES,    /*预留页链表，3*/
    ...
    MIGRATE_TYPES               /*迁移类型数量*/
};
```

在前面介绍的内存域初始化函数中，所有页块被初始化成 MIGRATE_UNMOVABLE（不可移动）迁移类型。如果在借用内存初始化函数中确定不支持页迁移属性，则 gfpflags_to_migratetype()函数始终返回 MIGRATE_UNMOVABLE 迁移类型，即分配页迁移类型为不可移动。如果支持迁移属性，内核可通过 set_pageblock_migratetype()函数重新设置页块的迁移类型。

gfpflags_to_migratetype()函数分配掩码参数中__GFP_MOVABLE 和__GFP_RECLAIMABLE 标记位用来确定目标页的迁移类型。如果只设置了__GFP_MOVABLE 标记位，则返回 MIGRATE_MOVABLE 迁移类型，如果只设置了__GFP_RECLAIMABLE 标记位，则返回 MIGRATE_RECLAIMABLE 迁移类型。如果两个标记位都设置了，则返回 MIGRATE_RESERVE 迁移类型。如果两个标记位都没有设置，则函数返回 MIGRATE_UNMOVABLE 迁移类型。

分配掩码的低 4 位用于确定分配内存域类型，gfp_zone(gfp_t flags)函数用来完成这项工作，函数返回目标内存域类型，函数定义在/include/linux/gfp.h 头文件内：

```
static inline enum    zone_type    gfp_zone(gfp_t flags)
{
    enum zone_type z;
    int bit = (__force int) (flags & GFP_ZONEMASK);    /*取分配掩码低 4 位*/

    z = (GFP_ZONE_TABLE >> (bit * ZONES_SHIFT)) & ((1 << ZONES_SHIFT) - 1);
    VM_BUG_ON((GFP_ZONE_BAD >> bit) & 1);
    return z;
}
```

GFP_ZONE_TABLE 和 ZONES_SHIFT 宏分别定义在/include/linux/gfp.h 和 page-flags-layout.h 头文件内，函数通过位运算确定内存域类型。具体位运算请读者阅读源代码，这里只讲解运算的结果。

分配掩码低 4 位用于选择内存域，其中低三位中最多只能有一位置位，否则为错误，三位分别表示选择 DMA，DMA32 或 HIGHMEM 内存域，全 0 时表示选择 NORMAL 内存域。第 4 位既能表示选择 MOVABLE 内存域，也能表示分配页的迁移类型。因此，低 4 位 16 种组合选择的结果如下：

位　域	内存域类型
0x0（0000）	NORMAL

0x1（0001）	DMA 或 NORMAL
0x2（0010）	HIGHMEM 或 NORMAL
0x3（0011）	错误
0x4（0100）	DMA32 或 DMA 或 NORMAL
0x5（0101）	错误
0x6（0110）	错误
0x7（0111）	错误
0x8（1000）	NORMAL
0x9（1001）	DMA 或 NORMAL
0xA（1010）	HIGHMEM
0xB（1011）	错误
0xC（1100）	DMA32
0xD（1101）	错误
0xE（1110）	错误
0xF（1111）	错误

注：迁移类型由前面介绍的__GFP_MOVABLE 和__GFP_RECLAIMABLE 标记位确定。

2 分配函数

分配掩码是调用者传递给分配函数的修饰符，分配函数内部有其自身的表示方式。alloc_context 结构体用于在分配函数内部表示分配流程的控制信息，结构体定义在/mm/internal.h 头文件内：

```
struct alloc_context {
    struct zonelist     *zonelist;          /*借用内存列表，快速路径中确定，可能在慢速路径中修改*/
    nodemask_t    *nodemask;                /*结点掩码，初始化后不再改变*/
    struct zone    *preferred_zone;         /*目标内存域，快速路径中确定，可能在慢速路径中修改*/
    int    classzone_idx;                   /*preferred_zone 内存域索引值*/
    int    migratetype;                     /*分配掩码确定的迁移类型，初始化后不再改变*/
    enum zone_type    high_zoneidx;         /*分配掩码确定的内存域类型，初始化后不再改变*/
};
```

分配函数最初由分配掩码参数初始化 alloc_context 实例，其中 nodemask、migratetype 和 high_zoneidx 成员值设置之后不会再改变，其余成员值在分配过程中（慢速路径中）可能会根据情况的变化而改变。

内核在/mm/internal.h 头文件内定义了下列宏，用于表示分配函数内部的分配标记：

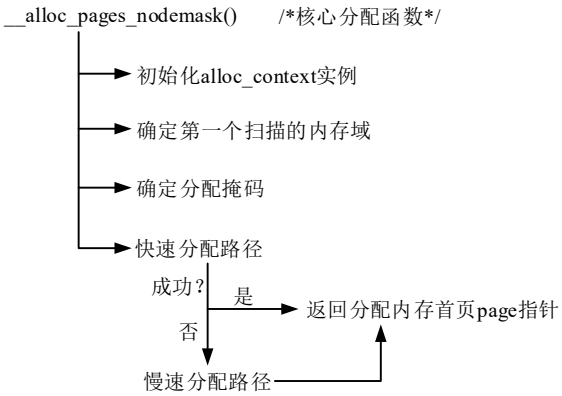
```
#define    ALLOC_WMARK_MIN          WMARK_MIN        /*水印值类型*/
#define    ALLOC_WMARK_LOW          WMARK_LOW
#define    ALLOC_WMARK_HIGH         WMARK_HIGH
#define    ALLOC_NO_WATERMARKS      0x04                /*不检查水印值*/

#define    ALLOC_WMARK_MASK    (ALLOC_NO_WATERMARKS-1)    /*水印值掩码*/

#define    ALLOC_HARDER        0x10    /*更努力地分配*/
#define    ALLOC_HIGH          0x20    /*分配掩码设置了__GFP_HIGH 标记位*/
```

```
#define   ALLOC_CPUSET        0x40    /*核查 cpuset */
#define   ALLOC_CMA           0x80    /*允许从 CMA 内存域分配*/
#define   ALLOC_FAIR          0x100   /*内存域之间公平分配*/
```

函数 gfp_to_alloc_flags(gfp_t gfp_mask)通过分配掩码设置分配标记，函数定义如下（/mm/page_alloc.c）：

```
static inline int gfp_to_alloc_flags(gfp_t gfp_mask)
{
    int alloc_flags = ALLOC_WMARK_MIN | ALLOC_CPUSET;
    const bool atomic = !(gfp_mask & (__GFP_WAIT | __GFP_NO_KSWAPD));
                            /*是否同时没有设置__GFP_WAIT 和__GFP_NO_KSWAPD 标记位*/

    BUILD_BUG_ON(__GFP_HIGH != (__force gfp_t) ALLOC_HIGH);

    alloc_flags |= (__force int) (gfp_mask & __GFP_HIGH);
                            /*分配掩码设置了__GFP_HIGH，分配标记设置 ALLOC_HIGH*/
    if (atomic) {
        if (!(gfp_mask & __GFP_NOMEMALLOC))
                                    /*分配掩码没有设置__GFP_NOMEMALLOC 标记位*/
            alloc_flags |= ALLOC_HARDER;

        alloc_flags &= ~ALLOC_CPUSET;    /*忽略 ALLOC_CPUSET 标记*/
    } else if (unlikely(rt_task(current)) && !in_interrupt())    /*实时进程，不处于中断处理中*/
        alloc_flags |= ALLOC_HARDER;

    if (likely(!(gfp_mask & __GFP_NOMEMALLOC))) {
                                    /*分配掩码没有设置__GFP_NOMEMALLOC 标记位*/
        if (gfp_mask & __GFP_MEMALLOC)
            alloc_flags |= ALLOC_NO_WATERMARKS;
        else if (in_serving_softirq() && (current->flags & PF_MEMALLOC))
            alloc_flags |= ALLOC_NO_WATERMARKS;
        else if (!in_interrupt() &&
                ((current->flags & PF_MEMALLOC) ||unlikely(test_thread_flag(TIF_MEMDIE))))
            alloc_flags |= ALLOC_NO_WATERMARKS;
    }
#ifdef CONFIG_CMA
    ...
#endif
    return alloc_flags;      /*分配标记*/
}
```

分配函数各接口函数最终调用__alloc_pages_nodemask()函数执行分配操作，函数在/mm/page_alloc.c 文件内实现，它是伙伴系统分配内存的核心函数。函数执行流程简列如下图所示：



__alloc_pages_nodemask()函数根据分配掩码参数设置 alloc_context 结构体实例，在借用内存列表中确定第一个扫描的内存域（可从当前或更低端的内存域分配内存），确定实际使用的分配掩码，随后先进入快速分配路径，如果分配成功则函数返回，否则进入慢速分配路径，分配成功后返回。

所谓快速分配路径就是从指定内存域开始扫描借用内存列表，从第一个空闲内存充足的内存域中分配内存。如果快速路径分配失败，则说明系统中空闲内存比较紧张了，将进入慢速分配路径。

__alloc_pages_nodemask()函数代码如下：

```
struct page *__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order, struct zonelist *zonelist,  \
                                                                        nodemask_t *nodemask)
/ *gfp_mask：分配掩码，order：分配阶，zonelist：结点借用内存列表，nodemask：结点掩码。*/
{
    struct zoneref *preferred_zoneref;          /*借用内存域指针*/
    struct page *page = NULL;
    unsigned int    cpuset_mems_cookie;
    int alloc_flags = ALLOC_WMARK_LOW|ALLOC_CPUSET|ALLOC_FAIR;   /*默认分配标记*/
                        /*分配标记，注意 ALLOC_WMARK_LOW 标记，用于检查的水印值*/
    gfp_t alloc_mask;           /*分配函数实际使用的分配掩码*/
    struct alloc_context ac = {                 /*初始化 alloc_context 实例*/
        .high_zoneidx = gfp_zone(gfp_mask),                 /*分配掩码参数确定的目标内存域类型*/
        .nodemask = nodemask,                               /*结点掩码，通常为 NULL*/
        .migratetype = gfpflags_to_migratetype(gfp_mask),   /*分配掩码参数确定的迁移类型*/
    };

    /*在启动阶段 gfp_allowed_mask=GFP_BOOT_MASK，屏蔽__GFP_WAIT，__GFP_IO，__GFP_FS
    *标记位（/mm/page_alloc.c），开中断后 gfp_allowed_mask = __GFP_BITS_MASK，不屏蔽任何
    *标记位（见 kernel_init_freeable()函数，/init/main.c）。*/

    gfp_mask &= gfp_allowed_mask;
    lockdep_trace_alloc(gfp_mask);

    might_sleep_if(gfp_mask & __GFP_WAIT);
                        /*如果设置了__GFP_WAIT 标记位，可能执行重调度，进入睡眠*/
    if (should_fail_alloc_page(gfp_mask, order))   /*需配置 FAIL_PAGE_ALLOC 选项，否则返回 false*/
        return NULL;
```

```
    if (unlikely(!zonelist->_zonerefs->zone))     /*至少需要存在一个有效内存域*/
        return NULL;

    if (IS_ENABLED(CONFIG_CMA) && ac.migratetype == MIGRATE_MOVABLE)
        alloc_flags |= ALLOC_CMA;              /*CMA 迁移类型*/

retry_cpuset:
    cpuset_mems_cookie = read_mems_allowed_begin();
                                   /*需选择配置 CPUSETS 选项，否则返回 0，/include/linux/cpuset.h*/
    ac.zonelist = zonelist;            /*借用内存域列表*/
    preferred_zoneref = first_zones_zonelist(ac.zonelist, ac.high_zoneidx,        \
                        ac.nodemask ? : &cpuset_current_mems_allowed,&ac.preferred_zone);
                  /*返回借用内存列表中第一个编号等于或小于 high_zoneidx 的内存域对应 zoneref 实例，
                   *可从编号为 high_zoneidx 或更小的内存域分配内存，/include/linux/mmzone.h*/
    if (!ac.preferred_zone)
        goto out;
    ac.classzone_idx = zonelist_zone_idx(preferred_zoneref);     /*zoneref->zone_idx，mmzone.h*/

    alloc_mask = gfp_mask|__GFP_HARDWALL;
                                       /*实际使用的分配掩码，增加__GFP_HARDWALL 标记位*/
    page = get_page_from_freelist(alloc_mask, order, alloc_flags, &ac);
                                 /*扫描借用内存列表从中分配内存，快速分配路径，/mm/page_alloc.c*/
    if (unlikely(!page)) {      /*如果快速路径分配不成功，进入慢速分配路径*/
        alloc_mask = memalloc_noio_flags(gfp_mask);
                                                        /*重置分配掩码，/include/linux/sched.h*/
        page = __alloc_pages_slowpath(alloc_mask, order, &ac); /*慢速分配路径，/mm/page_alloc.c*/
    }

    if (kmemcheck_enabled && page)
        kmemcheck_pagealloc_alloc(page, order, gfp_mask);

    trace_mm_page_alloc(page, order, alloc_mask, ac.migratetype);

out:
    if (unlikely(!page && read_mems_allowed_retry(cpuset_mems_cookie)))   /*/include/linux/cpuset.h*/
        goto retry_cpuset;

    return page;     /*返回分配内存块首页 page 指针*/
}
```

__alloc_pages_nodemask()函数中使用的 gfp_allowed_mask 是一全局变量，它是分配掩码的掩码。在内核启动初期 gfp_allowed_mask 用于屏蔽分配掩码参数中的__GFP_WAIT、__GFP_IO 和__GFP_FS 标记位，在启动后期（开中断后），gfp_allowed_mask 将设为__GFP_BITS_MASK，表示不屏蔽任何标记位。

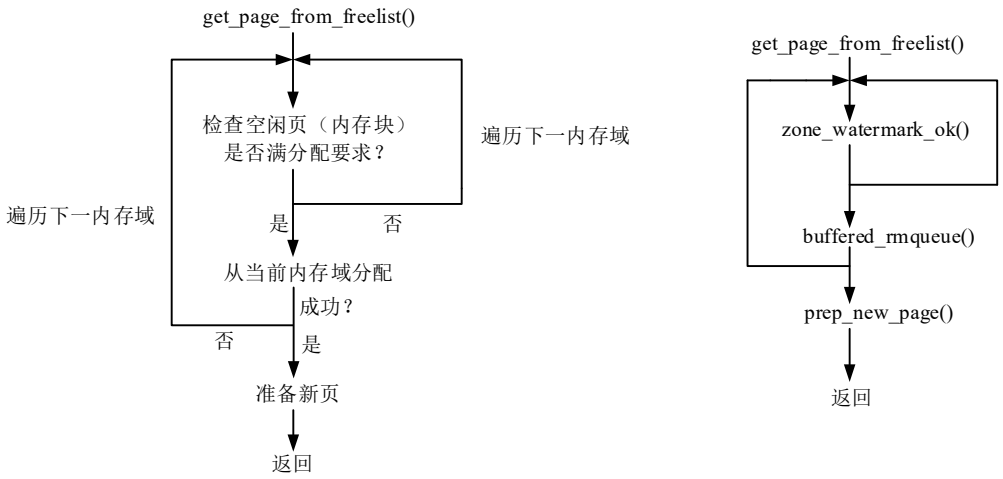
first_zones_zonelist()函数用于确定在借用内存列表中第一个扫描的内存域，内存域编号可等于或小于high_zoneidx，表示可以从 high_zoneidx 指定的内存域或更低端的内存域分配内存。

__alloc_pages_nodemask()函数在确定最终使用的分配掩码 alloc_mask 后，调用 get_page_from_freelist()函数进入快速分配路径，如果分配成功则返回分配内存块首页 page 指针。如果快速分配路径失败，则重置分配掩码后，调用__alloc_pages_slowpath()函数进入慢速分配路径，分配成功后返回。

下面将分别介绍快速和慢速分配路径的实现。

3 快速分配路径

分配函数中的快速分配路径由 get_page_from_freelist()函数实现，函数从 ac->high_zoneidx 指示的内存域开始在借用内存列表中往下遍历更低端的内存域，尝试从内存域中分配内存，成功则函数返回，否则遍历下一内存域，直至最底端的内存域。分配成功函数返回分配内存块首页 page 指针，否则返回 NULL。函数执行流程及函数调用关系简列如下图所示：



前面介绍过，借用内存只能从比指定内存域类型低级的内存域借用，例如，如果指定分配高端内存，在高端内存不足时，可从普通内存域、DMA 内存域借用，而从普通内存域分配时，不能借用高端内存。

分配函数由分配掩码确定起始的分配内存域。get_page_from_freelist()函数依次遍历借用内存列表中分配掩码指示的内存域和更低级的内存域，对每个内存域检查其空闲页帧（内存块）数量是否满足分配要求，如果是则尝试从本内存域分配，否则遍历下一内存域。在内存域空闲页帧数量足够，但分配仍然失败的情况下，也将遍历下一内存域。注意，在慢速路径中也将调用 get_page_from_freelist()函数。

get_page_from_freelist()函数定义在/mm/page_alloc.c 文件内，代码如下：

```
static struct page *get_page_from_freelist(gfp_t gfp_mask, unsigned int order, int alloc_flags, \
                                                    const struct alloc_context *ac)
/*gfp_mask：分配掩码，alloc_flags：分配标记，order：阶数*/
{
    struct zonelist *zonelist = ac->zonelist;          /*借用内存列表*/
    struct zoneref *z;
    struct page *page = NULL;
    struct zone *zone;
    nodemask_t *allowednodes = NULL;
    int zlc_active = 0;
    int did_zlc_setup = 0;
    bool consider_zone_dirty = (alloc_flags & ALLOC_WMARK_LOW) && \
```

```
                                                (gfp_mask & __GFP_WRITE);
    int nr_fair_skipped = 0;
    bool zonelist_rescan;

zonelist_scan:
    zonelist_rescan = false;

    /*遍历借用内存列表内存域，检查是否满足分配条件，/include/linux/mmzone.h*/
    for_each_zone_zonelist_nodemask(zone, z, zonelist, ac->high_zoneidx,ac->nodemask) {
        unsigned long mark;

        if (IS_ENABLED(CONFIG_NUMA) && zlc_active &&   \
                                !zlc_zone_worth_trying(zonelist, z, allowednodes))
            continue;
        if (cpusets_enabled() &&(alloc_flags & ALLOC_CPUSET) &&     \
                                        !cpuset_zone_allowed(zone, gfp_mask))
            continue;
        if (alloc_flags & ALLOC_FAIR) {       /*默认设置 ALLOC_FAIR 分配标记*/
            if (!zone_local(ac->preferred_zone, zone))     /*两个内存域是否属于同一结点*/
                break;
            if (test_bit(ZONE_FAIR_DEPLETED, &zone->flags)) {
                nr_fair_skipped++;
                continue;
            }
        }
        if (consider_zone_dirty && !zone_dirty_ok(zone))     /*内存域脏页数若超过限制值，跳过*/
            continue;

        mark = zone->watermark[alloc_flags & ALLOC_WMARK_MASK];
                                                /*mark 保存内存域 WMARK_LOW 水印值*/
        if (!zone_watermark_ok(zone, order, mark,ac->classzone_idx, alloc_flags)) {
            /*检查空闲页帧数量，/mm/page_alloc.c*/
            /*空闲页帧不足时，执行以下代码*/
            int ret;
            BUILD_BUG_ON(ALLOC_NO_WATERMARKS < NR_WMARK);
            if (alloc_flags & ALLOC_NO_WATERMARKS)
                goto try_this_zone;          /*设置了忽略水印值，仍然尝试从当前内存域分配*/

            if (IS_ENABLED(CONFIG_NUMA) &&!did_zlc_setup && nr_online_nodes > 1) {
                allowednodes = zlc_setup(zonelist, alloc_flags);
                zlc_active = 1;
                did_zlc_setup = 1;
            }
```

```
                if (zone_reclaim_mode == 0 || !zone_allows_reclaim(ac->preferred_zone, zone))
                        goto this_zone_full;

                if (IS_ENABLED(CONFIG_NUMA) && zlc_active &&     \
                                !zlc_zone_worth_trying(zonelist, z, allowednodes))
                        continue;

                ret = zone_reclaim(zone, gfp_mask, order);
                        /*NUMA 系统回收部分页面，UMA 系统直接返回 0，/include/linux/swap.h*/
                switch (ret) {
                    case    ZONE_RECLAIM_NOSCAN:
                            continue;
                    case    ZONE_RECLAIM_FULL:
                            continue;
                    default:
                            if (zone_watermark_ok(zone, order, mark,ac->classzone_idx, alloc_flags))
                                    goto try_this_zone;             /*再次检查水印值，合格则尝试分配*/
                            if (((alloc_flags & ALLOC_WMARK_MASK) == ALLOC_WMARK_MIN) ||    \
                                    ret == ZONE_RECLAIM_SOME)
                                    goto this_zone_full;

                            continue;
                }
            }       /*if 结束*/

        /*尝试从当前内存域分配内存*/
try_this_zone:
        page = buffered_rmqueue(ac->preferred_zone, zone, order,gfp_mask, ac->migratetype);
                                        /*从当前内存域分配内存，/mm/page_alloc.c*/
        if (page) {
            if (prep_new_page(page, order, gfp_mask, alloc_flags))
                            /*检查准备 page 实例，合格返回 0，否则返回 1，/mm/page_alloc.c*/
                goto try_this_zone;
            return page;            /*函数成功返回首页 page 指针*/
        }
    this_zone_full:
        if (IS_ENABLED(CONFIG_NUMA) && zlc_active)
            zlc_mark_zone_full(zonelist, z);
    }   /*遍历借用内存列表内存域结束*/

    /*分配不成功，执行以下代码*/
    if (alloc_flags & ALLOC_FAIR) {
```

```
            alloc_flags &= ~ALLOC_FAIR;
            if (nr_fair_skipped) {
                zonelist_rescan = true;
                reset_alloc_batches(ac->preferred_zone);
            }
            if (nr_online_nodes > 1)
                zonelist_rescan = true;
        }

        if (unlikely(IS_ENABLED(CONFIG_NUMA) && zlc_active)) {
            zlc_active = 0;
            zonelist_rescan = true;
    }

    if (zonelist_rescan)
        goto zonelist_scan;      /*重新扫描*/

    return NULL;        /*分配不成功返回 NULL*/
}
```

函数对每个内存域调用 zone_watermark_ok()函数检查空闲页数量是否充足，是否满足分配条件，是则调用 buffered_rmqueue()函数尝试从本内存域分配内存。若空闲页数量不足，不满足分配条件，但分配标记设置了忽略水印值，也将尝试从本内存域分配，否则遍历下一内存域。

若调用 buffered_rmqueue()函数从本内存域分配成功，则调用 prep_new_page()函数对内存区进行检查和准备后返回给分配函数调用者，否则遍历下一内存域。遍历完内存域后，若分配仍不成功，则返回 NULL。

下面再介绍一下快速分配路径函数 get_page_from_freelist()调用的三个重要的函数。

■检查空闲页数量

zone_watermark_ok()函数用于检查内存域空闲页数量是否充足，是否满足分配 order 阶内存块的需求，函数返回 true 表示可以从当前内存域分配，否则返回 false，函数定义在/mm/page_alloc.c 文件内：

```
bool zone_watermark_ok(struct zone *z, unsigned int order, unsigned long mark, \
                                                    int classzone_idx, int alloc_flags)
/*order：阶数，mark：需要满足的空闲页数量（WMARK_LOW 水印值），
*classzone_idx：原目标内存域编号，alloc_flags：分配标记，z：内存域*/
{
  return   __zone_watermark_ok(z, order, mark, classzone_idx, alloc_flags, \
                                            zone_page_state(z, NR_FREE_PAGES));
}
```

__zone_watermark_ok()函数定义如下（/mm/page_alloc.c）：

```
static bool __zone_watermark_ok(struct zone *z, unsigned int order,     \
                        unsigned long mark, int classzone_idx, int alloc_flags,long free_pages)
/*free_pages：内存域当前空闲页帧数量*/
{
```

```
    long min = mark;          /*分配内存后内存域空闲页帧需要满足的数量*/
    int o;
    long free_cma = 0;

    free_pages -= (1 << order) - 1;        /*空闲页数量减去本次分配出去的页帧数量*/
    if (alloc_flags & ALLOC_HIGH)   /*如果分配操作级别比较高，要求空闲页帧数量减半*/
        min -= min / 2;
    if (alloc_flags & ALLOC_HARDER)     /*尽大努力分配，空闲页帧数再减 1/4*/
        min -= min / 4;
#ifdef CONFIG_CMA
    if (!(alloc_flags & ALLOC_CMA))
        free_cma = zone_page_state(z, NR_FREE_CMA_PAGES);
#endif

    if (free_pages - free_cma <= min + z->lowmem_reserve[classzone_idx])
        return false;
         /*空闲页数量比 min 值与本内存域为原目标内存域预留页数量之和还小，返回 false*/

    /*执行到这里表示内存域空闲页帧数量满足要求，下面还要检查大于等于 order 链表中是否有
    *充足的内存块，如果内存域空闲页帧数量足够，但都是 order 阶以下的内存块，也不能分配。
    *下面循环从 0 阶开始至 order-1 阶，每次循环 free_pages 减去本阶链表中的空闲页数量，同时
    *min 值减半，最后如果 free_pages 小于等于 min 值，将返回 false，不能进行分配。*/

    for (o = 0; o < order; o++) {
        free_pages -= z->free_area[o].nr_free << o;
        min >>= 1;
        if (free_pages <= min)
            return false;
    }
    return true;      /*内存域空闲页数量充足（满足阶数要求）返回 true，表示可以分配*/
}
```

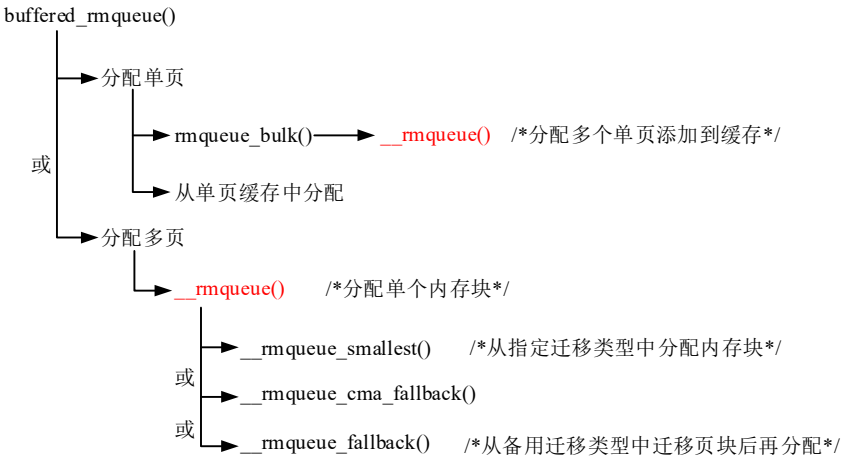
zone_watermark_ok()函数参数 mark 用于计算分配内存后，内存域空闲页帧数量需要满足的要求，这里 mark 值为内存域 WMARK_LOW 水印值。free_pages 表示内存域空闲页数量（减去了本次分配的页数量）。分配后内存域空闲页数量最小值由 min 表示，它通过 mark 值计算得来，如果是等级较高或尽最大努力分配，将缩减 min 值。分配后内存域空闲页帧数量需要大于 min 与 lowmem_reserve[classzone_idx]定义的预留页数量之和，否则函数返回 false。

除了考虑空闲页帧数量外，还需要考虑内存域伙伴链表中空闲内存块的阶数，如果阶数都小于 order，也将不能分配。函数最后的 for 循环用于检查内存域中大于等于 order 阶的空闲内存块数量是否满足一定的要求，满足则表示可以分配，否则表示不能分配。

zone_watermark_ok()函数最后返回真，表示可以尝试从当前内存域分配 order 阶内存块，否则表示不能分配，快速分配路径需要扫描下一个内存域。

■内存域分配函数

快速分配路径在判断了内存域空闲页满足分配条件后，将调用 buffered_rmqueue()函数尝试从当前内存域分配内存，函数调用关系如下图所示：



buffered_rmqueue()函数判断分配的是单页（order=0）还是多页（order>0），如果是单页则从 CPU 单页缓存中分配，如果单页缓存为空则先调用 rmqueue_bulk()函数从伙伴系统中分配一定数量的单页添加到单页缓存中，然后再从单页缓存中分配；如果是分配多页则调用__rmqueue()函数直接从伙伴系统中分配。rmqueue_bulk()函数内也是调用__rmqueue()函数从伙伴系统分配单页，后面将介绍。

buffered_rmqueue()函数定义如下（/mm/page_alloc.c）：

```
static inline struct page *buffered_rmqueue(struct zone *preferred_zone,    \
                                            struct zone *zone, unsigned int order,gfp_t gfp_flags, int migratetype)
/*preferred_zone：快速路径中优先扫描的内存域，zone：当前扫描的内存域，order：分配阶
*gfp_flags：分配掩码（不是分配标记），migratetype：迁移类型。*/
{
    unsigned long flags;
    struct page *page;
    bool cold = ((gfp_flags & __GFP_COLD) != 0);       /*分配冷页还是热页*/

    if (likely(order == 0)) {       /*分配 0 阶内存块，单页*/
        struct per_cpu_pages *pcp;
        struct list_head *list;

        local_irq_save(flags);
        pcp = &this_cpu_ptr(zone->pageset)->pcp;
        list = &pcp->lists[migratetype];            /*CPU 单页缓存链表*/
        if (list_empty(list)) {                     /*单缓存页链表为空，从伙伴系统链表分配*/
            pcp->count += rmqueue_bulk(zone, 0,pcp->batch, list,migratetype, cold);
          /*调用__rmqueue()函数从伙伴系统获取 batch 数量单页添加到单页缓存，/mm/page_alloc.c*/
            if (unlikely(list_empty(list)))
                goto failed;
        }
        /*从单页缓存中取页*/
        if (cold)
```

```
            page = list_entry(list->prev, struct page, lru);     /*冷页取链表末尾页*/
        else
            page = list_entry(list->next, struct page, lru);     /*热页取链表首页*/

        list_del(&page->lru);       /*从缓存链表移除*/
        pcp->count--;

    } else {          /*分配大于 0 阶的内存块*/
        if (unlikely(gfp_flags & __GFP_NOFAIL)) {
            WARN_ON_ONCE(order > 1);
        }
        spin_lock_irqsave(&zone->lock, flags);
        page = __rmqueue(zone, order, migratetype);
                                /*尝试从内存域伙伴链表中分配 order 阶内存块，/mm/page_alloc.c*/
        spin_unlock(&zone->lock);
        if (!page)
            goto    failed;
        __mod_zone_freepage_state(zone, -(1 << order), get_freepage_migratetype(page));
    }    /*分配操作结束，后面是一些更新统计量的操作*/

    __mod_zone_page_state(zone, NR_ALLOC_BATCH, -(1 << order));
    if (atomic_long_read(&zone->vm_stat[NR_ALLOC_BATCH]) <= 0 &&
    !test_bit(ZONE_FAIR_DEPLETED, &zone->flags))
        set_bit(ZONE_FAIR_DEPLETED, &zone->flags);

    __count_zone_vm_events(PGALLOC, zone, 1 << order);
    zone_statistics(preferred_zone, zone, gfp_flags);
    local_irq_restore(flags);

    VM_BUG_ON_PAGE(bad_range(zone, page), page);
    return page;          /*成功返回分配内存块首页 page 实例指针*/

failed:
    local_irq_restore(flags);
    return NULL;        /*失败返回 NULL*/
}
```

buffered_rmqueue()函数对分配 0 阶和非 0 阶内存块采用了不同的方法。如果是分配单页，则从内存域 zone 实例中特定于 CPU 的单页缓存链表中分配，如果链表为空则调用 rmqueue_bulk()函数从伙伴系统中获取 batch 数量的单页添加至链表，再从单页缓存中分配。如果需要分配的是冷页则从链表末尾取页，如果是热页则从链表头部取页。

如果是分配大于 0 阶的内存块，则调用__rmqueue()函数，尝试从伙伴系统中分配内存块。下面将分别介绍 rmqueue_bulk()和__rmqueue()函数的实现。

●批发内存块

rmqueue_bulk()函数用于从伙伴系统中分配指定数量的内存块（order 阶）并将内存块添加到指定链表中，也就是从伙伴系统中成批地分配内存块，函数定义如下（/mm/page_alloc.c）：

```
static int rmqueue_bulk(struct zone *zone, unsigned int order,unsigned long count, struct list_head *list,
                                                                    int migratetype, bool cold)
/*zone：内存域，order：分配阶，count：分配内存块数量，list：保存内存块的双链表，
*migratetype：内存块迁移类型，cold：冷页则内存块添加到 list 链表末尾，热页则添加到链表头部*/
{
    int i;
    spin_lock(&zone->lock);
    for (i = 0; i < count; ++i) {      /*每次循环分配一个 order 阶内存块，并添加到双链表*/
        struct page *page = __rmqueue(zone, order, migratetype);      /*分配单个内存块，见下文*/
        if (unlikely(page == NULL))
            break;

        if (likely(!cold))
            list_add(&page->lru, list);        /*热页添加到链表头*/
        else
            list_add_tail(&page->lru, list);     /*冷页添加到链表末尾*/
        list = &page->lru;
        if (is_migrate_cma(get_freepage_migratetype(page)))
            __mod_zone_page_state(zone, NR_FREE_CMA_PAGES,-(1 << order));
    }
    __mod_zone_page_state(zone, NR_FREE_PAGES, -(i << order));
    spin_unlock(&zone->lock);
    return i;       /*返回实际分配的 order 阶内存块数量*/
}
```

rmqueue_bulk()函数中每次循环调用__rmqueue()函数从内存域中分配一个 order 阶内存块，并添加到双链表中，下面将介绍__rmqueue()函数实现。

●分配单个内存块

分配大于 0 阶的内存块时，buffered_rmqueue()函数将调用**__rmqueue()**函数从伙伴系统中分配内存块，前面介绍的 rmqueue_bulk()函数中也调用了__rmqueue()函数分配一定数量的单页。__rmqueue()函数用于从内存域中分配一个任意阶（小于最大分配阶）的内存块，函数定义如下（/mm/page_alloc.c）：

```
static struct page *__rmqueue(struct zone *zone, unsigned int order,int migratetype)
{
    struct page *page;

  retry_reserve:
    page = __rmqueue_smallest(zone, order, migratetype);
                                        /*从指定迁移类型链表中分配内存，/mm/page_alloc.c*/

    /*指定迁移类型内存分配不成功，则需要执行以下代码*/
```

```
        if (unlikely(!page) && migratetype != MIGRATE_RESERVE) {
            if (migratetype == MIGRATE_MOVABLE)                    /*如果是可移动迁移类型*/
                page = __rmqueue_cma_fallback(zone, order);        /*备用分配函数，/mm/page_alloc.c*/
            if (!page)
                page = __rmqueue_fallback(zone, order, migratetype);   /*备用迁移类型分配函数*/
            if (!page) {
                migratetype = MIGRATE_RESERVE;      /*仍不成功，则尝试从保留类型中分配*/
                goto retry_reserve;                 /*再次执行分配*/
            }
        }
        trace_mm_page_alloc_zone_locked(page, order, migratetype);
        return page;
    }
```

在伙伴系统链表中，每一阶的内存块还按迁移类型进行了划分，相同迁移类型的内存块在同一链表中。分配函数中指定了分配内存块的迁移类型，但前面检查内存域中的空闲内存块时，并没有考虑内存块的迁移类型，只考虑了空闲内存块的阶数和数量。

__rmqueue()函数首先调用__rmqueue_smallest()函数从内存域伙伴系统中指定迁移类型的链表中分配内存块，如果分配成功则返回内存块首页 page 实例指针，否则继续往下执行。

如果指定迁移类型不是 MIGRATE_RESERVE，且迁移类型是 MIGRATE_MOVABLE 则先调用函数__rmqueue_cma_fallback()进行分配，若不成功则再调用__rmqueue_fallback()函数从备用迁移类型链表中分配。

如果指定迁移类型不是 MIGRATE_RESERVE 和 MIGRATE_MOVABLE，调用__rmqueue_fallback()函数从备用迁移类型链表中迁移页块至指定迁移类型，然后再进行分配。

如果执行完上面操作后仍然分配不成功，则将迁移类型设为 MIGRATE_RESERVE，再执行上面的分配操作。

如果没有选择 CMA 配置选项，__rmqueue_cma_fallback()函数直接返回 NULL，这里暂且不介绍此函数。下面分别介绍一下__rmqueue_smallest()和__rmqueue_fallback()函数的实现。

（1）指定迁移类型分配函数

__rmqueue_smallest()函数用于从指定迁移类型链表中分配内存块，链表阶数可以是 order 或大于 order，函数定义如下（/mm/page_alloc.c）：

```
static inline struct page *__rmqueue_smallest(struct zone *zone, unsigned int order,int migratetype)
{
    unsigned int current_order;
    struct free_area *area;
    struct page *page;

    /*从 order 阶链表开始往上搜索，查找并分配内存块*/
    for (current_order = order; current_order < MAX_ORDER; ++current_order) {
        area = &(zone->free_area[current_order]);
        if (list_empty(&area->free_list[migratetype]))     /*当前阶链表为空，则执行下一个循环*/
```

```
            continue;

        /*链表不为空，可以分配*/
        page = list_entry(area->free_list[migratetype].next,struct page, lru);
        list_del(&page->lru);          /*取下链表头部内存块*/
        rmv_page_order(page);          /*清阶内存块 buddy 属性*/
        area->nr_free--;               /*空闲内存块数量减 1*/
        expand(zone, page, order, current_order, area, migratetype);   /*前面介绍过的拆分函数*/
        set_freepage_migratetype(page, migratetype);    /*设置分配内存块迁移属性，index 成员值*/
        return page;          /*返回分配内存块首页 page 实例指针*/
    }
    return NULL;    /*没有找到可供拆分的内存块，返回 NULL*/
}
```

__rmqueue_smallest()函数依次扫描当前内存域 order 阶及更高阶的伙伴链表，如果 migratetype 迁移类型链表不为空，则取下链表头部内存块，调用久违的 expand()函数对其进行拆分，拆分后剩余内存块放回伙伴链表，函数返回分配内存块首页 page 实例指针。如果没有找到合适的可供拆分的内存块，函数返回 NULL。

（2）备用迁移类型分配函数

如果从指定迁移类型链表中分配内存块不成功，将调用__rmqueue_fallback()函数从备用迁移类型链表中迁移页块至指定迁移类型链表，然后再分配内存块。内核在/mm/page_alloc.c 文件内定义了迁移类型的备用关系数组：

```
static int fallbacks[MIGRATE_TYPES][4] = {        /*MIGRATE_TYPES 迁移类型*/
    [MIGRATE_UNMOVABLE] = { MIGRATE_RECLAIMABLE, MIGRATE_MOVABLE,
                                                    MIGRATE_RESERVE },
    [MIGRATE_RECLAIMABLE] = { MIGRATE_UNMOVABLE,    MIGRATE_MOVABLE,
                                                    MIGRATE_RESERVE },
    [MIGRATE_MOVABLE] = { MIGRATE_RECLAIMABLE, MIGRATE_UNMOVABLE,
                                                      MIGRATE_RESERVE },
#ifdef CONFIG_CMA
    [MIGRATE_CMA]   = { MIGRATE_RESERVE }, /* Never used */
#endif
    [MIGRATE_RESERVE] = { MIGRATE_RESERVE }, /* Never used */
#ifdef CONFIG_MEMORY_ISOLATION
    [MIGRATE_ISOLATE] = { MIGRATE_RESERVE }, /* Never used */
#endif
};
```

每种迁移类型最多包含 4 种备用的迁移类型。

__rmqueue_fallback()函数根据 fallbacks[MIGRATE_TYPES][]数组，从备用迁移类型链表中分配内存块，函数定义如下（/mm/page_alloc.c）：

```
static inline struct page *__rmqueue_fallback(struct zone *zone, unsigned int order, int start_migratetype)
```

```
{
    struct free_area *area;
    unsigned int current_order;
    struct page *page;
    int fallback_mt;
    bool can_steal;        /*是否可以从备用迁移类型中迁移页块*/

    /*从最高分配阶往下遍历 free_area[]数组，*/
    for (current_order = MAX_ORDER-1;current_order >= order && current_order <= MAX_ORDER-1; \
                                                                    --current_order) {
        area = &(zone->free_area[current_order]);
        fallback_mt = find_suitable_fallback(area, current_order,start_migratetype, false, \
                                                                    &can_steal);
                    /*检查备用迁移类型链表，判断是否具有可以迁移的页块，/mm/page_alloc.c*/
        if (fallback_mt == -1)
            continue;

        page = list_entry(area->free_list[fallback_mt].next,struct page, lru);
                                                    /*备用迁移类型链表中首个内存块*/
        if (can_steal)
            steal_suitable_fallback(zone, page, start_migratetype);
                        /*将页块迁移到 start_migratetype 迁移类型链表，/mm/page_alloc.c*/

        /*下面就是熟悉的拆分内存块操作了*/
        area->nr_free--;
        list_del(&page->lru);
        rmv_page_order(page);
        expand(zone, page, order, current_order, area,start_migratetype);
        set_freepage_migratetype(page, start_migratetype);
        trace_mm_page_alloc_extfrag(page, order, current_order,start_migratetype, fallback_mt);
        return page;
    }
    return NULL;
}
```

__rmqueue_fallback()函数根据 fallbacks[]数组定义的备用迁移类型列表，从最高阶开始往下遍历伙伴系统链表。对每一阶的 free_area[]实例，根据备用迁移类型列表的顺序，检查是否具有可以修改迁移类型的页块，如果有则迁移页块（链表中首个内存块所在页块）至 start_migratetype 迁移类型，对原备用链表中首个内存块进行拆分，返回 order 阶内存块给调用者。

__rmqueue_fallback()函数中调用的 find_suitable_fallback()函数用于查找具有可迁移页块的备用迁移类型，steal_suitable_fallback()函数用于修改页块的迁移类型，这两个函数源代码请读者自行阅读。

■准备新页

最后，快速分配路径分配内存块后，需要对其进行检查和准备再返回给调用者，prep_new_page()函数

用于完成此项工作，函数定义在/mm/page_alloc.c 文件内：

```
static int prep_new_page(struct page *page, unsigned int order, gfp_t gfp_flags,int alloc_flags)
/*page：内存块首页 page 实例，order：阶数，gfp_flags：分配掩码，alloc_flags：分配标记*/
{
    int i;

    for (i = 0; i < (1 << order); i++) {     /*遍历内存块各页*/
        struct page *p = page + i;
        if (unlikely(check_new_page(p)))     /*逐页检查，无错误返回 0，否则返回 1，/mm/page_alloc.c*/
                return 1;
    }

    set_page_private(page, 0);          /*首页 private 成员置 0*/
    set_page_refcounted(page);          /*首页_count 置 1*/
    arch_alloc_page(page, order);
    kernel_map_pages(page, 1 << order, 1);
    kasan_alloc_pages(page, order);

    if (gfp_flags & __GFP_ZERO)         /*如果需清零页*/
        for (i = 0; i < (1 << order); i++)
            clear_highpage(page + i);   /*逐页清零，/include/linux/highmem.h*/

    if (order && (gfp_flags & __GFP_COMP))      /*处理复合页*/
            prep_compound_page(page, order);        /*/mm/page_alloc.c*/

    set_page_owner(page, order, gfp_flags);
            /*设置页所有者，需选择 PAGE_OWNER 选项，/include/linux/page_owner.h*/

    if (alloc_flags & ALLOC_NO_WATERMARKS)      /*分配内存时忽略水印值*/
        set_page_pfmemalloc(page);      /*page->index = -1UL，/include/linux/mm.h*/
    else
        clear_page_pfmemalloc(page);    /*page->index =0，/include/linux/mm.h*/
    return 0;       /*内存块没问题返回 0*/
}
```

prep_new_page()函数主要是对页对应的 page 实例进行检查和设置，需要时清零页帧。这里需要说明一下的是对复合页的处理。page 结构体第三个双字是一个联合体，其中包含复合页的信息，定义如下：

```
struct page{
    ...
    union {         /*联合体开始，第三个双字*/
        struct list_head    lru;
        ...
        struct {                            /*用于复合页成员*/
            compound_page_dtor *compound_dtor;      /*释放复合页的函数指针*/
```

```
            unsigned long compound_order;          /*复合页阶数*/
        };
        ...
    }   /*联合体结束*/
    ...
    union {                                 /*第七个字*/
        unsigned long private;
        ...
        struct page *first_page;    /*若是复合页第 1 页之后的页，指向复合页首页 page 实例*/
    };          /*第七个字结束*/
    ...
}
```

compound_page_dtor 类型是一个函数指针，定义如下（/include/linux/mm_types.h）：

```
typedef void compound_page_dtor(struct page *);     /*释放复合页时调用的函数*/
```

compound_order 成员表示复合页的阶数。

处理复合页的 prep_compound_page()函数定义如下（/mm/page_alloc.c）：

```
void prep_compound_page(struct page *page, unsigned long order)
{
    int i;
    int nr_pages = 1 << order;
    /*处理第 2 页（源代码说明里说的是首页，但代码中是 page[1]，应该是第 2 页）*/
    set_compound_page_dtor(page, free_compound_page);
            /*page[1].compound_dtor =free_compound_page()，/include/linux/mm.h*/
    set_compound_order(page, order);
            /*page[1].compound_order = order，/include/linux/mm.h*/

    __SetPageHead(page);          /*设置首页 PG_head 标记位*/

     /*处理首页之后的页*/
    for (i = 1; i < nr_pages; i++) {
        struct page *p = page + i;
        set_page_count(p, 0);          /*_count 置 0*/
        p->first_page = page;          /*指向首页*/
        smp_wmb();
        __SetPageTail(p);       /*设置 PG_tail 标记位*/
    }
}
```

复合页首页 page 实例第三个双字中联合体解释为 lru 成员，用于将复合页添加到各种链表中，第二页中联合体则解释成 compound_dtor 和 compound_order 成员，前者指向释放复合页的函数指针，这里赋值为 free_compound_page()函数指针，compound_order 成员表示复合页的阶数。free_compound_page()函数调用

伙伴系统提供的__free_pages_ok()函数释放复合页。首页之后的所有页 first_page 成员指向首页 page 实例。首页 page 实例设置 PG_head 标记位，后面的所有页设置 PG_tail 标记位。

4 慢速分配路径

若快速路径分配不成功，说明指定内存域（及其下内存域）没有足够的空闲页或没有需要大小的连续内存块，分配函数将进入慢速分配路径，以使出更大的力气来完成内存分配工作。在进入慢速路径前，分配函数会对分配掩码进行修改：

```
alloc_mask = memalloc_noio_flags(gfp_mask);
```

memalloc_noio_flags()函数定义在/include/linux/sched.h 头文件：

```
static inline gfp_t memalloc_noio_flags(gfp_t flags)
{
    if (unlikely(current->flags & PF_MEMALLOC_NOIO))
                                        /*如果进程设置了 PF_MEMALLOC_NOIO 标记位*/
        flags &= ~(__GFP_IO | __GFP_FS);        /*清除掩码__GFP_IO 和__GFP_FS 标记位*/
    return flags;
}
```

如果当前进程标记设置了 PF_MEMALLOC_NOIO 标记位，则清除分配掩码的__GFP_IO 和 __GFP_FS 标记位。

慢速分配路径实现函数定义如下（/mm/page_alloc.c）：

```
static inline struct page *__alloc_pages_slowpath(gfp_t gfp_mask, unsigned int order,    \
                                                  struct alloc_context *ac)
 /*gfp_mask：修改过的分配掩码*/
{
    const gfp_t    wait = gfp_mask & __GFP_WAIT;        /*分配函数是否可以进入睡眠*/
    struct page *page = NULL;
    int alloc_flags;          /*分配标记*/
    unsigned long pages_reclaimed = 0;
    unsigned long did_some_progress;
    enum migrate_mode migration_mode = MIGRATE_ASYNC;     /*/include/linux/migrate_mode.h*/
    bool deferred_compaction = false;
    int contended_compaction = COMPACT_CONTENDED_NONE;

    if (order >= MAX_ORDER) {            /*分配阶检查*/
        WARN_ON_ONCE(!(gfp_mask & __GFP_NOWARN));
        return NULL;
    }

    if (IS_ENABLED(CONFIG_NUMA) && (gfp_mask & __GFP_THISNODE) && !wait)
        goto nopage;          /*指定本结点，不能睡眠，直接返回*/

retry:
```

```
if (!(gfp_mask & __GFP_NO_KSWAPD))     /*没有设置__GFP_NO_KSWAPD 标记*/
    wake_all_kswapds(order, ac);
                            /*唤醒页回收守护线程，详见 11 章，/mm/page_alloc.c*/
/*
* 设置分配标记，主要是根据情况设置 ALLOC_HARDER 或 ALLOC_NO_WATERMARKS 等
* 标记位，如：实时进程设置 ALLOC_HARDER 标记。
*/
alloc_flags = gfp_to_alloc_flags(gfp_mask);   /*设置分配标记，/mm/page_alloc.c*/

/*确定优先分配内存域（分配掩码确定的内存域）*/
if (!(alloc_flags & ALLOC_CPUSET) && !ac->nodemask) {
    struct zoneref *preferred_zoneref;
    preferred_zoneref = first_zones_zonelist(ac->zonelist,ac->high_zoneidx, NULL, \
                                                        &ac->preferred_zone);
    ac->classzone_idx = zonelist_zone_idx(preferred_zoneref);
}

/*激活页回收守护线程后，再次尝试分配，仍然考虑水印值*/
page = get_page_from_freelist(gfp_mask, order,alloc_flags & ~ALLOC_NO_WATERMARKS, \
                                                                    ac);
if (page)
    goto   got_pg;     /*分配成功跳至 got_pg*/

/*仍然不成功，则检查 alloc_flags 是否设置了忽略水印值标记*/
if (alloc_flags & ALLOC_NO_WATERMARKS) {          /*设置了忽略水印值*/
    ac->zonelist = node_zonelist(numa_node_id(), gfp_mask);
    page = __alloc_pages_high_priority(gfp_mask, order, ac);     /*/mm/page_alloc.c*/
        /*忽略水印值尝试分配，若不成功且设置了__GFP_NOFAIL，将不停尝试*/
    if (page) {
        goto got_pg;
    }
}

/*以下是没有设置 ALLOC_NO_WATERMARKS，或设置了但掩码没有设置__GFP_NOFAIL*/
if (!wait) {
    WARN_ON_ONCE(gfp_mask & __GFP_NOFAIL);
    goto nopage;            /*分配失败*/
}

/*进程设置了 PF_MEMALLOC 标记，分配失败*/
if (current->flags & PF_MEMALLOC)
    goto nopage;
```

```
/*避免在忽略水印的情况下，陷入无限循环*/
if (test_thread_flag(TIF_MEMDIE) && !(gfp_mask & __GFP_NOFAIL))
    goto nopage;

/*配置选择了 COMPACTION 选项，激活内存规整，整理内存碎片后再分配，/mm/page_alloc.c*/
page = __alloc_pages_direct_compact(gfp_mask, order, alloc_flags, ac,migration_mode,    \
                                    &contended_compaction,&deferred_compaction);
if (page)
    goto got_pg;

/*巨型页高阶（内存块）分配*/
if ((gfp_mask & GFP_TRANSHUGE) == GFP_TRANSHUGE) {
    if (deferred_compaction)
        goto nopage;

    if (contended_compaction == COMPACT_CONTENDED_LOCK)
        goto nopage;

    if (contended_compaction == COMPACT_CONTENDED_SCHED    \
                                && !(current->flags & PF_KTHREAD))
        goto nopage;
}

/*启动直接页回收机制*/
if ((gfp_mask & GFP_TRANSHUGE) != GFP_TRANSHUGE ||(current->flags & PF_KTHREAD))
     migration_mode = MIGRATE_SYNC_LIGHT;

page = __alloc_pages_direct_reclaim(gfp_mask, order, alloc_flags, ac,&did_some_progress);
            /*调用 try_to_free_pages()函数执行直接页回收后再分配，/mm/page_alloc.c*/
if (page)
     goto got_pg;

if (gfp_mask & __GFP_NORETRY)       /*不重试*/
     goto   noretry;          /*直接页面回收后，跳转至 noretry，再次进行内存规整后分配*/

pages_reclaimed += did_some_progress;
if ((did_some_progress && order <= PAGE_ALLOC_COSTLY_ORDER) ||
    ((gfp_mask & __GFP_REPEAT) && pages_reclaimed < (1 << order))) {
    wait_iff_congested(ac->preferred_zone, BLK_RW_ASYNC, HZ/50);
    goto retry;         /*等待写请求完成后，再尝试分配*/
}

/*启用终极武器，OOM 机制，杀死一个进程（先尝试分配，不成功再启用 OOM），/mm/page_alloc.c*/
```

```
        page = __alloc_pages_may_oom(gfp_mask, order, ac, &did_some_progress);
        if (page)       /*分配成功，返回（没有启用 OOM）*/
            goto got_pg;

        /*分配不成功，但启用了 OOM，再次尝试分配*/
        if (did_some_progress)
            goto    retry;

    noretry:
        /*执行直接页回收后，再次执行内存规整，尝试分配*/
        page = __alloc_pages_direct_compact(gfp_mask, order, alloc_flags,ac, migration_mode,   \
                                                &contended_compaction, &deferred_compaction);
        if (page)
            goto got_pg;
    nopage:     /*无能为力了，分配不成功*/
        warn_alloc_failed(gfp_mask, order, NULL);     /*分配失败，输出信息，/mm/page_alloc.c*/
    got_pg:          /*分配成功*/
        return page;      /*返回分配内存块首页 page 实例指针*/
    }
```

慢速分配路径主要是通过激活页回收与页交换机制、内存归整、直接页面回收、OOM 机制（out of memory）等以释放出内存，然后再进行分配。

内存规整本书暂不介绍，页回收与页交换机制在第 11 章再做介绍，激活这些机制后，最终的分配还是调用快速路径的 get_page_from_freelist()函数来分配内存。

这里先看一下 OOM 机制的激活，OOM 机制简单地说就是寻找一个“倒霉”的进程，将它杀死以释放内存。激活 OOM 机制的函数为__alloc_pages_may_oom()，定义在/mm/page_alloc.c 文件内，代码如下：

```
static inline struct page *__alloc_pages_may_oom(gfp_t gfp_mask, unsigned int order,      \
                                    const struct alloc_context *ac, unsigned long *did_some_progress)
{
    struct page *page;
    *did_some_progress = 0;

     /*获取 OOM 锁，若其它进程正在执行，则不能再执行*/
    if (!mutex_trylock(&oom_lock)) {
        *did_some_progress = 1;
        schedule_timeout_uninterruptible(1);
        return NULL;
    }

    /*再次尝试分配，注意检查水印值为 WMARK_HIGH*/
    page = get_page_from_freelist(gfp_mask | __GFP_HARDWALL, order,     \
                                        ALLOC_WMARK_HIGH|ALLOC_CPUSET, ac);
    if (page)
```

```
            goto out;     /*若分配成功，则可以返回*/

        if (!(gfp_mask & __GFP_NOFAIL)) {
            /*分配掩码没有设置__GFP_NOFAIL 标记，执行 if 内代码*/
            /*以下是判断不执行 OOM 机制的条件*/
            if (current->flags & PF_DUMPCORE)
                goto out;
            if (order > PAGE_ALLOC_COSTLY_ORDER)
                goto out;
            if (ac->high_zoneidx < ZONE_NORMAL)
                goto out;
            if (!(gfp_mask & __GFP_FS)) {
                *did_some_progress = 1;
                goto out;
            }

            if (pm_suspended_storage())
                goto out;
            if (gfp_mask & __GFP_THISNODE)
                goto out;
        }
        /*分配掩码设置了__GFP_NOFAIL 标记，激活 OOM 机制，/mm/oom_kill.c*/
        if (out_of_memory(ac->zonelist, gfp_mask, order, ac->nodemask, false)||   \
                                                WARN_ON_ONCE(gfp_mask & __GFP_NOFAIL))
            *did_some_progress = 1;
    out:
        mutex_unlock(&oom_lock);
        return page;
    }
```

out_of_memory()函数用于激活 OOM 机制，第 5 章介绍进程管理时我们再具体讨论此机制的实现。至此伙伴系统分配函数就介绍完了。

3.7 slab 分配器

前面介绍的伙伴系统是以页为单位对物理内存进行管理和分配的，而内核代码中需要大量分配大小比页更小的数据结构实例或者是小块的内存。如果仍然采用伙伴系统为其分配一页的空间，势必造成巨大的浪费，因此内核开发者创建了 slab（slub,slob）分配器，用于分配小块的内存。

内核提供了 slab、slub 和 slob 三个分配器，三个分配器具有相同的 API 函数，配置内核时选择其中之一。内核其它部分并不需要关心具体选用的是哪种分配器，只需要调用通用接口函数创建缓存，分配和释放对象即可。三个分配器共用的接口函数定义在/mm/slab_common.c 文件内，具体分配器实现函数分别定义在/mm/slab.c、slub.c 和 slob.c 文件内。slub 分配器主要用于具有大量物理内存的计算机，而 slob 分配器是一个比较简化的分配器，适用于微小嵌入式系统，slab 分配器代码量和性能比较适中。本节以 slab 分配器为例，介绍分配器的实现，本书后面也用 slab 分配器来指代用户在 slab、slub 和 slob 三个分配器中所选的分配器。

slab 分配器从伙伴系统按页获取内存（可理解成批发），再将其划分成更小的内存块供内核使用（可理解成零售）。用户空间 malloc()库函数与此类似，它将按页获取的用户内存划分成小块供用户进程使用。

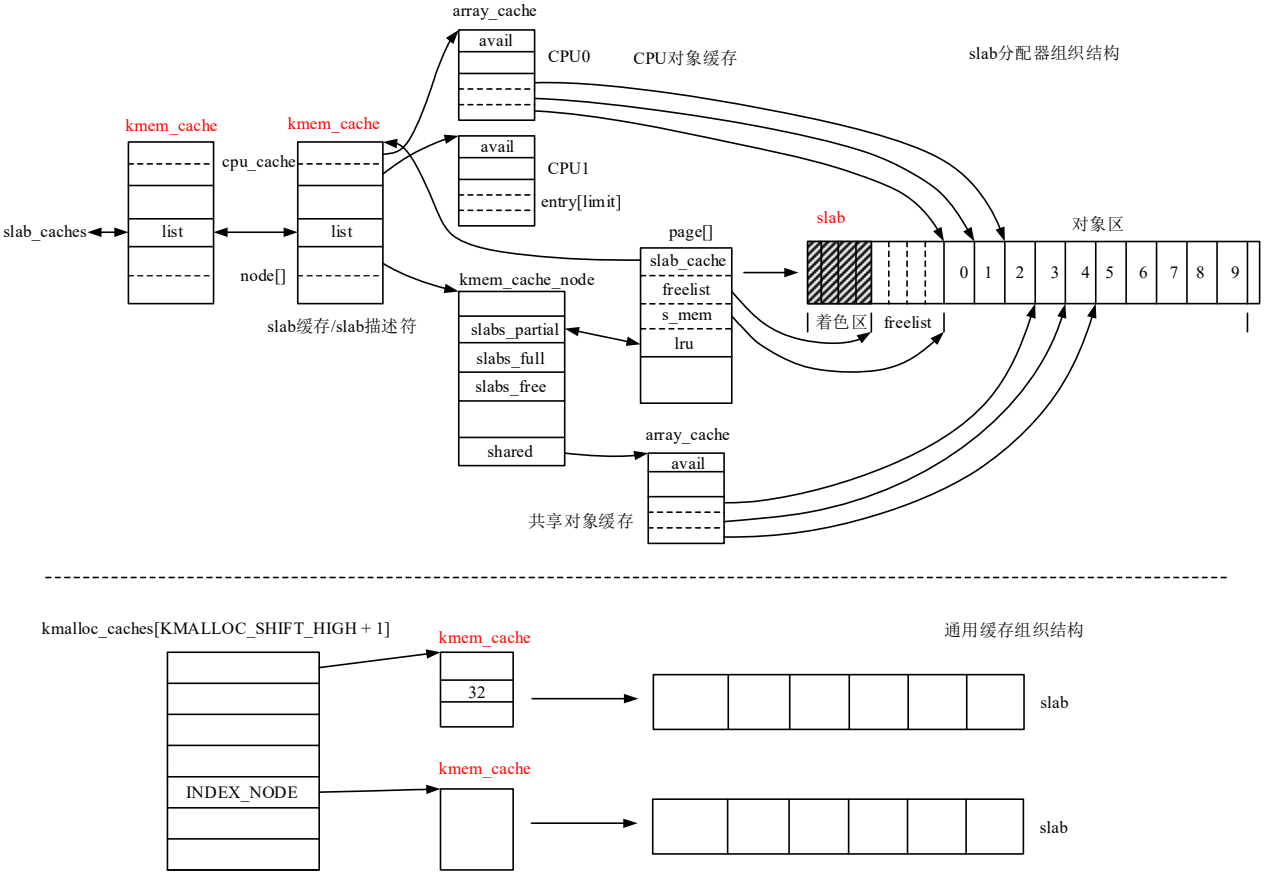
3.7.1 分配器结构

本小节概述 slab 分配器框架及主要的数据结构。

1 组织结构

slab 分配器用于管理、分配小块的内存，主要用于分配内核各数据结构实例。分配器从伙伴系统中按页批发大块的内存，再将其划分成小块的内存供内核使用。slab 分配器中管理着许多使用和未使用的小块内存（数据结构实例），因此又称为 slab 缓存。每个 slab 缓存管理着相同大小的内存块，用于分配同类型数据结构实例，因此内核需要为不同类型数据结构创建不同的 slab 缓存。

slab 分配器组织结构如下图所示：



每个 slab 缓存由 kmem_cache 结构体表示，称为 slab 描述符，一个描述符管理着某一固定大小的内存块缓存（对象），缓存对象保存在 slab 中。内核中所有 kmem_cache 实例由全局双链表 slab_caches 管理。

对象（内存块）保存在 slab 中，每个 slab 由 2^gfporder 个连续内存页组成（从伙伴系统中分配，只限低端内存），gfporder 称为阶数，如 gfporder=2，则 slab 大小为 4 个页帧。每个 slab 分成三个部分，开头为着色区，用于设置对象区起始地址的偏移量，以防止对象导入到相同的 CPU 缓存行中而降低 CPU 缓存效率，freelist 区是一个字符或短整型数数组，用于保存 slab 中空闲对象编号（空闲对象链表），对象区即连续保存着对象。

在从伙伴系统中为 slab 分配页帧时，设置了页帧组合成复合页，首页 page 实例中 **slab_cache** 成员指

向描述符 kmem_cache 实例，freelist 成员是指向 freelist 区的指针，s_mem 成员保存第一个对象地址。

每个 slab 缓存中包含三个 slab 链表，一个是所含对象部分被使用的 slab 链表，一个是所含对象全部被使用的 slab 链表，另一个是所含对象全部空闲的 slab 链表（未被使用），上图中只画出部分对象被使用的 slab 链表。这三个链表由 kmem_cache_node 结构体管理，每个 slab 通过首页 page 实例的 lru 成员加入到管理双链表中。

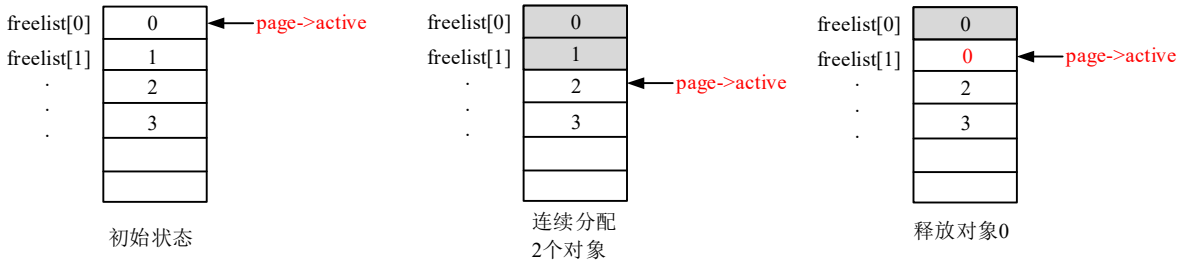
array_cache 结构体通过对象指针数组管理关联各 CPU 的对象缓存（空闲对象），指针数组项指向 slab 中的对象(缓存中对象在 slab 中标记为已分配)。kmem_cache 结构体中的 cpu_cache 成员指向的 array_cache 实例是一个 percpu 变量，kmem_cache_node 结构体 shared 成员指向的 array_cache 实例缓存的是各 CPU 之间共享的对象。

分配对象时始终从本地 CPU 对象缓存中查找，若有缓存对象则直接返回对象指针；若没有则从共享缓存中转移对象至本地 CPU 缓存后再分配；若共享缓存中也没有对象，则从 slab 中查找或创建新 slab（从伙伴系统中分配内存块），并从中分配对象关联到本地 CPU 缓存，然后再从本地 CPU 缓存中分配。

释放对象与分配对象正好相反，释放的对象始终释放到本地 CPU 缓存，若释放前本地 CPU 缓存已满，则先将本地 CPU 缓存中对象释放到共享缓存或 slab 中（可能还需要释放 slab），再将对象释放到本地 CPU 缓存。

slab 中 freelist 区（也可能在 slab 之外）是一个数组，它记录了 slab 中空闲对象的编号。分配对象时取出第一个空闲对象的编号分配出去，释放对象时将对象编号写入数组。

freelist 区相当于一个栈，用于保存空闲对象编号，栈指针是 slab 首页 page 实例 active 成员值。初始状态 slab 中所有对象空闲，对象编号依次写入 freelist 数组项，page->active 指向数组首项。



从 page->active 指向的数组项开始往后，记录的是空闲对象编号，之前的数组项记录的是已分配的对象编号，或部分已分配对象的编号。分配释放操作中不用管 page->active 之前的数组项。

分配对象时，取出 page->active 数组项保存的对象编号值，分配出去，page->active 往下移一项（出栈）。释放对象时，page->active 先往上移一项（入栈），然后将释放对象编号写入 page->active 指向的数组项。

内核中通常为常用的各数据结构类型创建各自的 slab 缓存，同时也按内存块大小创建了一组通用的 slab 缓存，即在申请内存块时无需指定从哪个缓存中分配，只需要指定大小，内核会自动从最合适的通用缓存中分配对象。

内核中 slab 分配器是动态的，kmem_cache 结构体实例也由 slab 缓存管理，而 kmem_cache_node 实例来源于通用缓存，关联各 CPU 的 array_cache 结构体实例为 percpu 变量。

slab 分配器初始化时需要为 kmem_cache 结构体创建 slab 缓存以及创建通用缓存。

slab 分配器通用接口函数声明如下（/include/linux/slab.h）：

●struct kmem_cache ***kmem_cache_create**(const char *name, size_t size, size_t align,unsigned long flags, void (*ctor)(void *))：创建 slab 缓存，name 表示名称，size 表示对象实际大小的字节数，align 表示 slab 中对象的对齐字节数，flags 表示 slab 标记，ctor 表示构造函数指针，在分配 slab 创建对象时调用。

●void **kmem_cache_destroy**(struct kmem_cache *s)：销毁 slab 缓存函数。

●void ***kmem_cache_alloc**(struct kmem_cache *cachep, gfp_t flags)：从指定 slab 缓存中分配对象，返回

对象指针。

●void **kmem_cache_free**(struct kmem_cache *, void *)：释放从指定 slab 缓存中分配的对象。

●void ***kmem_cache_zalloc**(struct kmem_cache *k, gfp_t flags)：从指定 slab 缓存分配对象，并且对象清零，返回对象指针。

●void ***kmalloc**(size_t size, gfp_t flags)：从通用缓存中分配指定大小内存块，返回起始地址。

●void ***kzalloc**(size_t size, gfp_t flags)：从通用缓存中分配指定大小内存块，内存块清零，返回起始地址。

●void **kfree**(const void *)：释放从通用缓存分配的内存块。

2 kmem_cache

在对 slab 分配器有了整体认识后，下面我们来详细看一下 slab 分配器主要的数据结构，即 kmem_cache、array_cache 和 kmem_cache_node 结构体的定义。

kmem_cache 结构体称为 slab 缓存描述符，表示一个 slab 缓存，结构体中主要包含各 CPU 本地对象缓存、slab 布局的信息以及 kmem_cache_node 实例指针数组等。

kmem_cache 结构体定义在/include/linux/slab_def.h 头文件内：

```
struct kmem_cache {
    struct array_cache __percpu *cpu_cache;      /*各 CPU 本地对象缓存，array_cache 实例*/

    unsigned int batchcount;   /*本地缓存对象为空时，从共享缓存或 slab 中成批获取对象的数量*/
    unsigned int limit;        /*本地缓存对象数量大于等于 limit 时，释放 batchcount 个对象*/
    unsigned int shared;     /*共享缓存中对象数量（8），用于多核处理器，单核处理器为 0*/

    unsigned int size;        /*对象大小（加上对齐字节数）*/
    struct reciprocal_value    reciprocal_buffer_size;     /*由对象地址计算编号时使用*/

    unsigned int flags;        /*slab 缓存标记*/
    unsigned int num;          /*每个 slab 中对象的数量*/
    unsigned int gfporder;   /*分配阶，每个 slab 占用的页帧数量为 2gfporder*/
    gfp_t allocflags;          /*为 slab 分配页帧额外的分配掩码，如__GFP_COMP*/

    size_t colour;                  /*最大颜色数量*/
    unsigned int colour_off;      /*颜色偏移量*/
    struct kmem_cache *freelist_cache;     /*从通用缓存中分配 freelist 内存时，指向通用缓存描述符*/
    unsigned int freelist_size;           /*freelist 区大小*/

    void (*ctor)(void *obj);          /*构造函数，在创建对象时调用*/

    const char *name;               /*描述符名称*/
    struct list_head list;          /*双链表成员，将 kmem_cache 实例添加到全局双链表*/
    int refcount;                  /*引用计数*/
    int object_size;                /*对象的实际大小，不加对齐字节数*/
    int align;                     /*对齐字节数*/
```

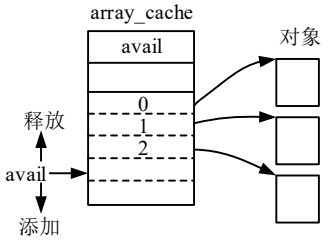
```
#ifdef CONFIG_DEBUG_SLAB
    ...         /*统计量*/
#endif
#ifdef CONFIG_MEMCG_KMEM
    ...
#endif
    struct kmem_cache_node *node[MAX_NUMNODES];     /*kmem_cache_node 指针数组*/
};
```

kmem_cache 结构体主要成员简介如下：

●**cpu_cache**：array_cache 结构体指针，percpu 变量，每个 CPU 对应一个指针成员，指向 array_cache 实例。array_cache 结构体用于管理各 CPU 本地的对象缓存，结构体定义在/mm/slab.c 文件内：

```
struct array_cache {
   unsigned int    avail;        /*array_cache 缓存中当前可用对象的数目*/
   unsigned int    limit;        /*空闲对象数目大于 limit 时，主动释放 batchcount 个对象*/
   unsigned int    batchcount;   /*缓存为空时成批获取对象的数量，缓存对象较多时释放对象的数量*/
   unsigned int    touched;
   void *entry[];        /*缓存对象指针，数组项数为 limit*/
};
```

array_cache 结构体如下图所示，空闲对象在指针数组 entry[]中从低到高（数组项）依次排列，中间没有空缺，avail 表示空闲对象的数量。指针数组是一个后进先出的队列，当分配对象时，avail 值先减 1，分配 entry[avail]指向的对象。向缓存增加对象时，对象指针赋予 entry[avail]成员，而后 avail 值加 1。



●**batchcount**：本地 CPU 对象缓存（array_cache）中对象为空时，从共享缓存或 slab 链表中获取 batchcount 个对象关联到本地 CPU 对象缓存，释放对象时当本地缓存中对象数大于等于 limit 时，释放 batchcount 个对象到共享缓存或 slab 链表。

●**limit**：释放对象时，将对象释放到本地 CPU 缓存，当本地 CPU 缓存中对象数大于等于 limit 时，释放 batchcount 个对象到共享缓存或 slab 链表中。

●**reciprocal_buffer_size**：reciprocal_value 结构体成员，定义如下（/include/linux/reciprocal_div.h）：

```
struct reciprocal_value {
    u32 m;
    u8 sh1, sh2;
};
```

在由对象地址计算对象在 slab 中编号时使用，用于将除法转换成乘法，以提高运算速度。

●**flags**：slab 缓存标记，创建 slab 缓存时由参数传递进来，标记位简列如下（/include/linux/slab.h）：

```
#define   SLAB_DEBUG_FREE  0x00000100UL     /**/
#define   SLAB_RED_ZONE    0x00000400UL     /*对象前后设置额外内存区，用于检测未知访问*/
#define   SLAB_POISON      0x00000800UL     /*分配释放对象时用预定义模式填充*/
```

```
#define   SLAB_HWCACHE_ALIGN     0x00002000UL     /*对象硬件缓存行对齐*/
#define   SLAB_CACHE_DMA         0x00004000UL     /*从 DMA 内存域为 slab 分配内存*/
#define   SLAB_STORE_USER        0x00010000UL     /*保存最后使用者*/
#define   SLAB_PANIC             0x00040000UL     /*创建缓存失败引发内核恐慌*/

#define   SLAB_DESTROY_BY_RCU    0x00080000UL     /* Defer freeing slabs to RCU */
#define   SLAB_MEM_SPREAD        0x00100000UL     /* Spread some memory over cpuset */
#define   SLAB_TRACE             0x00200000UL     /* Trace allocations and frees */
#define   SLAB_NOLEAKTRACE       0x00800000UL     /* Avoid kmemleak tracing */
...
#define   SLAB_RECLAIM_ACCOUNT   0x00020000UL     /*对象是可回收的*/
#define   SLAB_TEMPORARY         SLAB_RECLAIM_ACCOUNT   /*对象是暂时存在*/
```

内核在/mm/slab.c 内还定义了 CFLGS_OFF_SLAB 标记：

```
#define   CFLGS_OFF_SLAB      (0x80000000UL)     /*用于标记 freelist 区是在 slab 外部还是内部*/
```

由于 CFLGS_OFF_SLAB 标记位不是定义在头文件中，因此不能由函数参数传递，而是在创建 slab 缓存时根据实际情况由内核设置。

●**colour、colour_off**：着色区是为了将 slab 中对象相对于 slab 起始地址有个偏移量，以防止同一缓存中各 slab 中相对位置相同的对象共用 CPU 内相同的缓存行。例如：假设同一缓存两个 slab 中的第一个对象都位于 slab 开始位置，而页帧都是 4KB 对齐的，因此两个对象很可能映射到相同的 CPU 缓存行中，引起缓存颠簸。slab 中加入着色区就是为了将对象映射到不同的缓存行中。

每个 slab 中的着色区长度是以 colour_off 为单位的，colour_off 一般为 CPU 一级缓存中缓存行的长度。slab 中着色区最大长度为(colour-1)*colour_off，colour 称为颜色数。假设 colour_off 为 32 字节，colour 值为 3，则在创建 slab 时，着色区长度依次为 0*32=0，1*32=32，2*32=64 字节，当 slab 数量超过颜色数量时颜色值又 0 开始往上递增，创建 slab 时，下一个 slab 的颜色值保存在 kmem_cache_node 实例中。

●**node[MAX_NUMNODES]**：kmem_cache_node 结构体指针数组，每个内存结点对应一个指针数组项。

kmem_cache_node 结构体主要用于管理 slab 双链表和共享对象缓存，kmem_cache 实例中对每个内存结点创建一个 kmem_cache_node 实例，结构体定义在/mm/slab.h 头文件内：

```
struct kmem_cache_node {
    spinlock_t list_lock;          /*自旋锁*/

#ifdef CONFIG_SLAB
    struct list_head    slabs_partial;   /*部分对象空闲的 slab 链表*/
    struct list_head    slabs_full;      /*全部对象使用的 slab 链表*/
    struct list_head    slabs_free;      /*全部对象空闲的 slab 链表*/
    unsigned long    free_objects;       /*所有空闲对象总数*/
    unsigned int    free_limit;          /*空闲对象阀值，大于此阀值则释放全部空闲 slab*/
    unsigned int    colour_next;         /*下一个颜色值*/
    struct array_cache    *shared;       /*共享对象缓存，指向 array_cache 实例*/
    struct alien_cache    **alien;       /*其它结点上的缓存，本书只考虑单结点*/
    unsigned long    next_reap;          /*下一次收缩缓存间隔时间*/
    int    free_touched;                 /* updated without locking */
```

```
#endif

#ifdef CONFIG_SLUB
    ...
#endif
};
```

创建 slab 缓存时 kmem_cache_node 实例将从通用缓存中分配。

3.7.2 初始化

slab 分配器使用的主要数据结构有 kmem_cache、array_cache、kmem_cache_node 等，kmem_cache 和 kmem_cache_node 实例也来自于 slab 缓存（通用缓存），array_cache 实例为 percpu 变量或来自于通用缓存。

分配器在初始化阶段首先要初始化 kmem_cache 结构体 slab 缓存，因为创建其它 slab 缓存时需要分配 kmem_cache 实例。然后，是要初始化通用缓存，kmem_cache_node 实例等来自于通用缓存，最后还需要完成其它一些初始化工作。

内核定义了枚举类型 slab_state 用于表示 slab 分配器的状态，这里是指内核 slab 分配器子系统的状态，而不是某个 slab 缓存的状态，枚举类型定义在/mm/slab.h 头文件：

```
enum slab_state {
    DOWN,               /*分配器不可使用*/
    PARTIAL,            /*SLUB: kmem_cache_node available */
    PARTIAL_NODE,       /*SLAB:  分配 kmem_cache_node 实例的通用缓存可用*/
    UP,                 /*slab 分配器可用，通用缓存初始化完成*/
    FULL                /*slab 分配器完全初始化完毕*/
};
```

内核在/mm/slab_common.c 内定义了以下全局变量：

```
enum slab_state    slab_state;          /*表示 slab 分配器的状态*/
LIST_HEAD(slab_caches);                 /*全局 kmem_cache 实例双链表头*/
struct kmem_cache *kmem_cache;          /*指向 kmem_cache 结构体 slab 缓存描述符的指针*/
```

slab_state 表示 slab 分配器状态，slab_caches 为全局 kmem_cache 实例双链表头。

内核在/mm/slab.c 文件内定义了 kmem_cache 结构体 slab 缓存描述符，kmem_cache 指针指向此描述符：

```
static struct kmem_cache kmem_cache_boot = {
    .batchcount = 1,
    .limit = BOOT_CPUCACHE_ENTRIES,
    .shared = 1,
    .size = sizeof(struct kmem_cache),      /*kmem_cache 结构体大小*/
    .name = "kmem_cache",
};
```

内核在/mm/slab.c 文件内静态定义了关联的 kmem_cache_node 实例（数组）：

```
#define   NUM_INIT_LISTS   (2 * MAX_NUMNODES)
```

```
static struct kmem_cache_node __initdata    init_kmem_cache_node[NUM_INIT_LISTS];
#define    CACHE_CACHE    0
#define    SIZE_NODE    (MAX_NUMNODES)
```

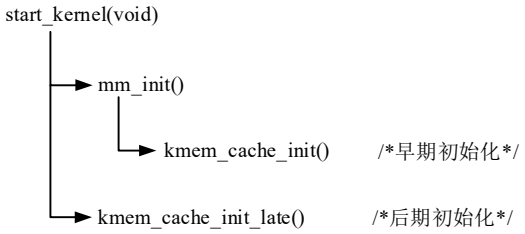
init_kmem_cache_node[]数组项数为 2* MAX_NUMNODES，其中前一半用于构建 kmem_cache 结构体 slab 缓存，后一半用于构建 kmem_cache_node 结构体对应的通用缓存（临时性地使用）。

内核在/mm/slab_common.c 文件内定义了通用缓存 kmem_cache 实例指针数组：

```
struct kmem_cache *kmalloc_caches[KMALLOC_SHIFT_HIGH + 1];
```

KMALLOC_SHIFT_HIGH 宏定义在/include/linux/slab.h 头文件，取值一般为 22。kmalloc_caches[]数组项指向各通用缓存的 kmem_cache 实例（描述符）。

slab 分配器初始化分为两步，第一步称它为早期初始化，第二步称它为后期初始化，这两步函数调用关系如下图所示：



kmem_cache_init()函数完成早期初始化，由 mm_init()函数调用，完成 kmem_cache 和 kmem_cache_node 结构体 slab 缓存的创建（包含在通用缓存中），创建通用缓存等。

kmem_cache_init_late()函数完成后期初始化工作，主要是为在 slab 分配器初始化阶段中创建的 slab 缓存，创建完整的 array_cache 实例和 kmem_cache_node 实例。

1 早期初始化

早期初始化函数 kmem_cache_init()定义在/mm/slab.c 文件内，代码如下：

```
void __init kmem_cache_init(void)
{
    int i;

    BUILD_BUG_ON(sizeof(((struct page *)NULL)->lru) <sizeof(struct rcu_head));
    kmem_cache = &kmem_cache_boot;     /*指向 kmem_cache 结构体 slab 缓存描述符，/mm/slab.c*/

    if (num_possible_nodes() == 1)
        use_alien_caches = 0;

    for (i = 0; i < NUM_INIT_LISTS; i++)     /*初始化静态定义 kmem_cache_node 数组，/mm/slab.c*/
        kmem_cache_node_init(&init_kmem_cache_node[i]);     /*数组项数 2 * MAX_NUMNODES*/

    /*slab_max_order_set 初始值为 0，若传递了"slab_max_order="命令行参数则设为 true。*/
```

```
    if (!slab_max_order_set && totalram_pages > (32 << 20) >> PAGE_SHIFT)
        slab_max_order = SLAB_MAX_ORDER_HI;
                        /*内存大于 32MB，SLAB_MAX_ORDER_HI 取值 1，/mm/slab.c*/

    create_boot_cache(kmem_cache, "kmem_cache",
            offsetof(struct kmem_cache, node) +nr_node_ids * sizeof(struct kmem_cache_node *),
            SLAB_HWCACHE_ALIGN);            /*此时分配器状态为 DOWN*/
                /*初始化 kmem_cache 结构体 slab 缓存描述符，/mm/slab_common.c*/
    list_add(&kmem_cache->list, &slab_caches);          /*将 kmem_cache 实例添加到全局双链表*/
    slab_state = PARTIAL;       /*分配器状态设为 PARTIAL*/

    kmalloc_caches[INDEX_NODE] = create_kmalloc_cache("kmalloc-node",
            kmalloc_size(INDEX_NODE), ARCH_KMALLOC_FLAGS); /*/mm/slab_common.c*/

   /*创建分配 kmem_cache_node 实例的通用缓存，create_kmalloc_cache()函数内最终对通用
    *slab 缓存调用 create_kmalloc_cache()函数从 kmem_cache 结构体 slab 缓存中分配 kmem_cache
    *实例，调用 create_boot_cache()函数初始化描述符，并将其添加到全局双链表。*/

    slab_state = PARTIAL_NODE;      /*slab 分配器状态设为 PARTIAL_NODE*/
    setup_kmalloc_cache_index_table();
            /*修改 size_index[24]数组（对齐要求高时），一般不需要修改，/mm/slab_common.c*/

    slab_early_init = 0;      /*早期初始化完成*/

    /*从通用缓存中分配 kmem_cache_node 实例代替静态定义的数组*/
    {
        int nid;
        for_each_online_node(nid) {
            init_list(kmem_cache, &init_kmem_cache_node[CACHE_CACHE + nid], nid);
            init_list(kmalloc_caches[INDEX_NODE],&init_kmem_cache_node[SIZE_NODE + nid], nid);
        }
    }
    create_kmalloc_caches(ARCH_KMALLOC_FLAGS); /*创建通用缓存，/mm/slab_common.c*/
}
```

内核静态定义了 kmem_cache 结构体 slab 缓存描述符实例，kmem_cache_init()调用 **create_boot_cache()** 函数对此描述符进行初始化，主要是创建/初始化关联的 array_cache 和 kmem_cache_node 实例。然后，将 kmem_cache 实例添加到全局双链表，设置 slab 分配器状态为 PARTIAL。

kmem_cache_init()函数随后调用 create_kmalloc_cache()函数创建分配 kmem_cache_node 实例的通用缓存，设置 slab 分配器状态为 PARTIAL_NODE，然后从通用缓存中分配 kmem_cache_node 实例，代替静态定义的 kmem_cache_node 实例数组。

kmem_cache_init()函数最后调用 create_kmalloc_caches()函数创建其它的通用缓存。

下面介绍初始化 slab 缓存描述符的 **create_boot_cache()**函数和创建通用缓存的 **create_kmalloc_caches()** 函数的实现。

■初始化 kmem_cache 实例

内核静态定义了 kmem_cache 结构体的 slab 缓存描述符 kmem_cache 实例，要使此缓存可用，可用来为其它 slab 缓存分配 kmem_cache 实例，还需要对此 kmem_cache 实例进行初始化。初始化的主要工作是确定 slab 布局，创建关联的 array_cache 和 kmem_cache_node 实例等。后面在为其它数据结构创建 slab 缓存时，分配 kmem_cache 实例后，也需要进行同样的初始化。

create_boot_cache()函数（/mm/slab_common.c）用于在早期初始化阶段，初始化 kmem_cache 实例，函数代码如下：

```
void __init create_boot_cache(struct kmem_cache *s, const char *name, size_t size,unsigned long flags)
/*s：kmem_cache 实例指针，name：slab 缓存名称，size：缓存对象大小，
*flags：slab 标记，如 SLAB_POISON、SLAB_RED_ZONE、SLAB_HWCACHE_ALIGN 等。*/
{
    int err;

    s->name = name;             /*参数值赋予 kmem_cache 实例*/
    s->size = s->object_size = size;        /*对象实际大小*/
    s->align = calculate_alignment(flags, ARCH_KMALLOC_MINALIGN, size);     /*对齐字节数*/

    slab_init_memcg_params(s);

    err = __kmem_cache_create(s, flags);    /*kmem_cache 实例初始化，详见下文，/mm/slab.c*/
    if (err)
        panic("Creation of kmalloc slab %s size=%zu failed. Reason %d\n",name, size, err);

    s->refcount = -1;           /* Exempt from merging for now */
}
```

create_boot_cache()函数对 kmem_cache 实例部分成员赋值后，调用__kmem_cache_create()函数主要完成 slab 布局的确定以及创建关联的 array_cache 和 kmem_cache_node 实例等工作。

在 slab 分配器初始化完成以前及完成后，创建 slab 缓存的操作中都将调用__kmem_cache_create()函数完成新创建 slab 缓存（kmem_cache 实例）的初始化，因此在函数内需要检测当前 slab 分配器的状态，不同的状态创建 array_cache 和 kmem_cache_node 实例的操作会有所不同。

__kmem_cache_create()函数定义如下（/mm/slab.c）：

```
int __kmem_cache_create (struct kmem_cache *cachep, unsigned long flags)
/*flags：slab 缓存标记*/
{
    size_t left_over, freelist_size;            /*着色区长度，freelist 区长度*/
    size_t ralign = BYTES_PER_WORD;     /*对齐字节数，初始值设为字对齐*/
    gfp_t gfp;              /*slab 初始化时使用的分配掩码，主要用于 kmem_cache_node 实例的分配*/
    int err;
    size_t size = cachep->size;     /*管理对象对齐后大小，初始值为对象实际大小*/

  #if DEBUG
    ...
  #endif
```

```
    if (size & (BYTES_PER_WORD - 1)) {      /*确保对象字对齐，最小值为一个字*/
        size += (BYTES_PER_WORD - 1);
        size &= ~(BYTES_PER_WORD - 1);
    }

    if (flags & SLAB_RED_ZONE) {    /*设置危险区，在对象前后增加额外的空间*/
        ralign = REDZONE_ALIGN;      /*长整型数字节数，/mm/slab.c*/
        size += REDZONE_ALIGN - 1;
        size &= ~(REDZONE_ALIGN - 1);
    }

    if (ralign < cachep->align) {           /*若描述符定义的对齐字节数更大，则采用它*/
        ralign = cachep->align;
    }

    if (ralign > __alignof__(unsigned long long))
        flags &= ~(SLAB_RED_ZONE | SLAB_STORE_USER);      /*清除标记位*/

    cachep->align = ralign;          /*保存对齐字节数*/

    if (slab_is_available())          /*slab 分配器状态大于等于 UP，slab 分配器初始化已经完成*/
        gfp = GFP_KERNEL;   /*分配 kmem_cache_node 实例使用掩码，从 NORMAL 内存域分配*/
    else
        gfp = GFP_NOWAIT;    /*slab 分配器初始化未完全完成，现在就是这种状态*/

#if DEBUG
    ...
#endif
    if ((size >= (PAGE_SIZE >> 5)) && !slab_early_init && !(flags & SLAB_NOLEAKTRACE))
            flags |= CFLGS_OFF_SLAB;        /*此标记表示 freelist 在 slab 外部*/
    /*对象大小大于 128 字节，且 slab 分配器通用缓存初始化完成后，freelist 放在 slab 外部。*/
    /*分配器初始化阶段 slab_early_init 为 1，此时表示 freelist 在 slab 内部。*/

    size = ALIGN(size, cachep->align);    /*对象大小对齐*/
    if (FREELIST_BYTE_INDEX && size < SLAB_OBJ_MIN_SIZE)
        size = ALIGN(SLAB_OBJ_MIN_SIZE, cachep->align);        /*限制对象大小最小值*/

    left_over = calculate_slab_order(cachep, size, cachep->align, flags);
            /*确定 slab 布局，返回 slab 减 freelist 区和对象区剩下的区域长度（保留区长度），/mm/slab.c*/
    if (!cachep->num)
        return -E2BIG;
```

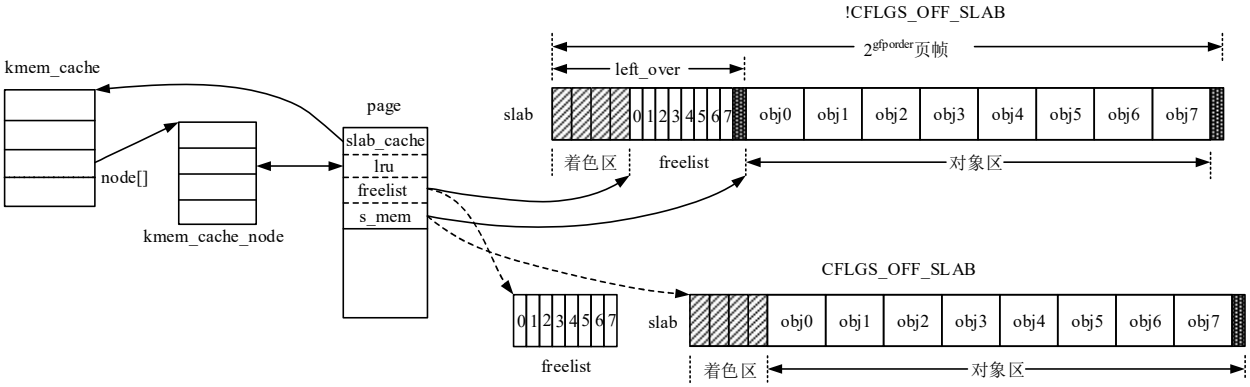
```
    freelist_size = calculate_freelist_size(cachep->num, cachep->align);
                        /*计算 freelist 区大小，数组项数为对象数量，加上对齐要求，/mm/slab.c*/
    if (flags & CFLGS_OFF_SLAB && left_over >= freelist_size) {
        flags &= ~CFLGS_OFF_SLAB;          /*left_over 大于 freelist_size，则将其放在 slab 内部*/
        left_over -= freelist_size;
    }

    if (flags & CFLGS_OFF_SLAB) {       /*freelist 在 slab 外部*/
        freelist_size = calculate_freelist_size(cachep->num, 0);   /*计算 freelist 大小，没有对齐要求*/

      #ifdef CONFIG_PAGE_POISONING
        ...
      #endif
    }

    cachep->colour_off = cache_line_size();     /*L1_CACHE_BYTES，/include/linux/cache.h*/
    if (cachep->colour_off < cachep->align)       /*确定每个颜色占用空间大小*/
        cachep->colour_off = cachep->align;
    cachep->colour = left_over / cachep->colour_off;          /*最大颜色数*/
    cachep->freelist_size = freelist_size;              /*freelist 区大小*/
    cachep->flags = flags;                              /*slab 缓存标记*/
    cachep->allocflags = __GFP_COMP;                    /*分配掩码，设置复合页标记位*/
    if (CONFIG_ZONE_DMA_FLAG && (flags & SLAB_CACHE_DMA))
        cachep->allocflags |= GFP_DMA;
    cachep->size = size;                                /*对齐后对象大小*/
    cachep->reciprocal_buffer_size = reciprocal_value(size);
                                /*由对象地址计算索引值时使用，/lib/reciprocal_div.c*/

    if (flags & CFLGS_OFF_SLAB) {           /*freelist 在 slab 外部*/
        cachep->freelist_cache = kmalloc_slab(freelist_size, 0u);   /*从通用缓存为 freelist 分配空间*/
        BUG_ON(ZERO_OR_NULL_PTR(cachep->freelist_cache));
    }

    err = setup_cpu_cache(cachep, gfp);     /*创建 array_cache 实例等，不同状态操作不同，/mm/slab.c*/
    ...
    return 0;       /*成功返回 0*/
}
```

__kmem_cache_create()函数首先确定对象的大小及对齐要求，设置分配掩码，确定 freelist 区是在 slab 内部还是外部，然后调用 calculate_slab_order()函数确定 slab 合适的分配阶（对象数量，保留区长度等），调用 calculate_freelist_size()函数计算 freelist 区长度，并将以上数值赋予 kmem_cache 实例。最后，调用函数 setup_cpu_cache()为 slab 缓存描述符创建 array_cache 和 kmem_cache 结构体实例。

下面将介绍 calculate_slab_order()和 setup_cpu_cache()函数的实现。

●**确定 slab 布局**

在介绍 calculate_slab_order()函数代码前，先来看一下 slab 缓存中的布局，如下图所示：



每个 slab 由 $2^{gfporder}$ 个物理页帧组成，slab 头部是着色区，接着是 freelist 区，最后是保存对象实例的区。

●**着色区**：着色区跟颜色并没有关系，它是为了防止同一缓存不同 slab 中相对位置相同的对象被导入到 CPU 缓存的相同行中，从而降低缓存的效率。着色区用于错开对象的起始地址。

着色区的概念在前面介绍过了，着色区的长度以 colour_off 为单位，最大长度为(colour-1)*colour_off。在创建 slab 时从 kmem_cache_node 结构体实例 colour_next 成员获取下一个 slab 的颜色数，下一个 slab 着色区长度为 colour_next*colour_off，colour_next 成员值在 0 至(colour-1)间循环。

●**freelist**：freelist 区意思为空闲对象链表，它实际上是一个字符或短整型数数组，数组项数与对象数量相同，但与对象并不是一一对应的关系，数组项保存的是空闲对象编号，后面讲解分配对象时将介绍此数组的实际使用。

●**对象区**：满足对齐要求的对象实例数组。

__kmem_cache_create()函数调用 calculate_slab_order()函数确定 slab 布局，即 slab 所占的页帧数（分配阶）、对象数量及保留区长度等信息。

在介绍函数实现前，先介绍一下 KMALLOC_MAX_ORDER 宏，此宏定义在/include/linux/slab.h 头文件，其值为页块大小（2^(MAX_ORDER-1)个页帧）和 32MB 之间的最小值。

calculate_slab_order()函数定义在/mm/slab.c 文件内，源代码如下：

```
static size_t calculate_slab_order(struct kmem_cache *cachep,size_t size, size_t align, unsigned long flags)
{
    unsigned long offslab_limit;
    size_t left_over = 0;
    int gfporder;

    for (gfporder = 0; gfporder <= KMALLOC_MAX_ORDER; gfporder++) {      /*遍历每个分配阶*/
        unsigned int num;          /*对象数量*/
        size_t remainder;          /*保留区长度（含着色区）*/

        cache_estimate(gfporder, size, align, flags, &remainder, &num);
                                              /*确定当前阶情况下的 slab 布局，/mm/slab.c*/
        if (!num)                    /*对象数为 0，遍历下一阶*/
            continue;
```

```
            if (num > SLAB_OBJ_MAX_NUM)     /*对象数超过最大值，跳出循环*/
                break;

            if (flags & CFLGS_OFF_SLAB) {         /*freelist 在 slab 外部*/
                size_t freelist_size_per_obj = sizeof(freelist_idx_t);         /*freelist 数组项大小*/

                if (IS_ENABLED(CONFIG_DEBUG_SLAB_LEAK))
                    freelist_size_per_obj += sizeof(char);
                offslab_limit = size;         /*对象大小*/
                offslab_limit /= freelist_size_per_obj;          /*对象大小除 freelist 数组项大小*/

                if (num > offslab_limit)          /*freelist 区大于一个对象大小，跳出循环*/
                    break;
            }

            cachep->num = num;                /*slab 中对象数量*/
            cachep->gfporder = gfporder;      /*slab 分配阶*/
            left_over = remainder;            /*保留区长度，slab 中除 freelist 区和对象区外的长度*/

            if (flags & SLAB_RECLAIM_ACCOUNT)     /*设置 SLAB_RECLAIM_ACCOUNT，0 阶*/
                break;

            if (gfporder >= slab_max_order)     /*分配阶达到最大值，跳出循环*/
                break;

            if (left_over * 8 <= (PAGE_SIZE << gfporder))   /*保留区长度小于 slab 长度 1/8，跳出循环*/
                break;
        }
        /*跳出循环后表示找到了合适的 slab 分配阶（slab 大小）*/
        return left_over;       /*返回 slab 中除 freelist 区和对象区外的长度*/
    }
```

calculate_slab_order()函数从 0 阶开始遍历各分配阶，对每个阶调用 cache_estimate()函数计算当前阶内存下 slab 的布局，当达到设置的某一条件时将跳出循环，表示将当前阶作为 slab 的分配阶，返回 slab 中保留区长度。

cache_estimate()函数用于指定阶时，计算 slab 中对象数量，保留区长度等，函数定义如下（/mm/slab.c）：

```
static void cache_estimate(unsigned long gfporder, size_t buffer_size,       \
                                        size_t align, int flags, size_t *left_over,unsigned int *num)
/* buffer_size：对象大小，*left_over：记录保留区长度，*num：slab 中对象数量*/
{
    int nr_objs;                    /*对象数量*/
    size_t mgmt_size;               /*freelist 区大小*/
    size_t slab_size = PAGE_SIZE << gfporder;       /*slab 大小，字节数*/
```

```
    if (flags & CFLGS_OFF_SLAB) {          /*freelist 区在 slab 外部*/
        mgmt_size = 0;
        nr_objs = slab_size / buffer_size;      /*对象数量*/

    } else {                               /*freelist 区在 slab 内部*/
        nr_objs = calculate_nr_objs(slab_size, buffer_size,sizeof(freelist_idx_t), align);
                                                        /*计算对象数量，/mm/slab.c*/
        mgmt_size = calculate_freelist_size(nr_objs, align);  /*计算 freelist 区大小，/mm/slab.c*/

    }
    *num = nr_objs;          /*对象数量*/
    *left_over = slab_size - nr_objs*buffer_size - mgmt_size; /*slab 中除 freelist 区和对象区外的大小*/
}
```

cache_estimate()函数首先判断 freelist 区是在 slab 内部还是外部，以确定对象的数量，当 freelist 在 slab 内部时，由对象大小加上 freelist 数组成员大小（字节）确定对象数量，由对象数量可确定 freelist 区大小（需对齐），最后 slab 大小减对象区和 freelist 区占用的大小就是保留区大小。

如果 freelist 区在 slab 外部，则此时 freelist 区大小暂时记为 0，保留区大小即 slab 大小减对象区大小。

calculate_slab_order()函数调用 cache_estimate()函数找到第一个合适的 slab 分配阶，确定其布局后，返回保留区大小。

●创建 array_cache 和 kmem_cache_node 实例

__kmem_cache_create()函数在确定了 slab 缓存分配阶及各参数后，将计算 freelist 区大小，并将以上各参数值赋予 kmem_cache 实例，最后调用 **setup_cpu_cache()**函数为 kmem_cache 实例创建各 CPU 核关联的 array_cache 实例（缓存对象只有 1 个）和在线内存结点对应的 kmem_cache_node 实例。

setup_cpu_cache()函数根据当前 slab 分配器的状态，确定 array_cache 和 kmem_cache_node 实例的创建方式，函数定义如下（/mm/slab.c）：

```
static int __init_refok setup_cpu_cache(struct kmem_cache *cachep, gfp_t gfp)
/*gfp：分配掩码，用于 kmem_cache_node 实例分配*/
{
    if (slab_state >= FULL)                 /*slab 分配器初始化完成后的情形*/
        return enable_cpucache(cachep, gfp);        /*此函数在后面讲解创建 slab 缓存时再介绍*/

    /*slab 分配器未完成初始化时，执行以下代码*/
    cachep->cpu_cache = alloc_kmem_cache_cpus(cachep, 1, 1);     /*只含 1 个缓存对象*/
                        /*分配 array_cache 实例，调用分配 percpu 变量实例的函数，/mm/slab.c*/
    if (!cachep->cpu_cache)
        return 1;

    /*下面是分配（设置）kmem_cache_node 实例*/
    if (slab_state == DOWN) {       /*kmem_cache 结构体 slab 缓存尚未初始化*/
        set_up_node(kmem_cache, CACHE_CACHE);
                        /*kmem_cache 结构体 slab 缓存关联静态定义的 kmem_cache_node 实例*/
```

```
    } else if (slab_state == PARTIAL) {       /*kmem_cache_node 结构体 slab 缓存尚未初始化*/
        set_up_node(cachep, SIZE_NODE);
                    /*kmem_cache_node 结构体 slab 缓存关联静态定义的 kmem_cache_node 实例*/
    } else {        /*kmem_cache 和 kmem_cache_node 结构体 slab 缓存已初始化完成，可用*/
        int node;           /*从 kmem_cache_node 结构体 slab 缓存（通用缓存）中分配*/

        for_each_online_node(node) {
            cachep->node[node] = kmalloc_node(sizeof(struct kmem_cache_node), gfp, node);
            BUG_ON(!cachep->node[node]);
            kmem_cache_node_init(cachep->node[node]);     /*初始化 kmem_cache_node 实例*/
        }
    }

    cachep->node[numa_mem_id()]->next_reap =jiffies + REAPTIMEOUT_NODE +     \
                                    ((unsigned long)cachep) % REAPTIMEOUT_NODE;
    /*初始化当前 CPU 关联 array_cache 实例*/
    cpu_cache_get(cachep)->avail = 0;
    cpu_cache_get(cachep)->limit = BOOT_CPUCACHE_ENTRIES;
    cpu_cache_get(cachep)->batchcount = 1;
    cpu_cache_get(cachep)->touched = 0;
    cachep->batchcount = 1;
    cachep->limit = BOOT_CPUCACHE_ENTRIES;
    return 0;       /*成功返回 0*/
}
```

slab 分配器最开始处于 DOWN 状态，当 kmem_cache 结构体 slab 缓存初始化完成后转为 PARTIAL 状态；当 kmem_cache_node 结构体 slab 缓存初始化完成后转为 PARTIAL 状态；通用缓存初始化完成后转为 UP 状态；当为所有在此之前创建的各 slab 缓存描述符创建完各 CPU 核关联的 array_cache 实例和各内存结点关联的 kmem_cache_node 实例后，slab 状态转为 FULL（后期初始化内容），表示可以正常调用接口函数创建 slab 缓存了。

setup_cpu_cache()函数根据当前 slab 分配器状态，采用不同的方式为 slab 缓存描述符创建 array_cache 和 kmem_cache_node 实例。当状态为 FULL 时，调用 enable_cpucache()函数实现，这个函数后面再介绍。

当 slab 分配器状态不为 FULL 时，调用分配 percpu 变量实例的函数创建 array_cache 实例（percpu 变量后面将介绍）。内核静态定义了 kmem_cache_node 实例数组 init_kmem_cache_node[]，其前半部分用于 kmem_cache 结构体 slab 缓存，后半部分用于 kmem_cache_node 结构体 slab 缓存。

在早期初始化 kmem_cache_init()函数中，调用 create_kmalloc_cache()函数创建了 kmem_cache_node 结构体 slab 缓存（通用缓存中的一项）。create_kmalloc_cache()函数最终从 kmem_cache 结构体 slab 缓存中分配 kmem_cache 实例，调用 create_boot_cache()函数初始化 kmem_cache 实例，并将其添加到全局双链表，函数源代码请读者自行阅读。

kmem_cache_init()函数最后调用 create_kmalloc_caches()函数创建通用缓存，函数实现见下文。

■创建通用缓存

通用缓存是一组不固定用于哪种数据结构，而只按大小分配内存块的 slab 缓存。从通用缓存中分配对象时不需要指明从哪个 slab 缓存分配，只需要指明对象大小，分配器会从最合适的通用 slab 缓存中分配对

象。

内核在/mm/slab_common.c 文件内定义了通用缓存 kmem_cache 实例指针数组：

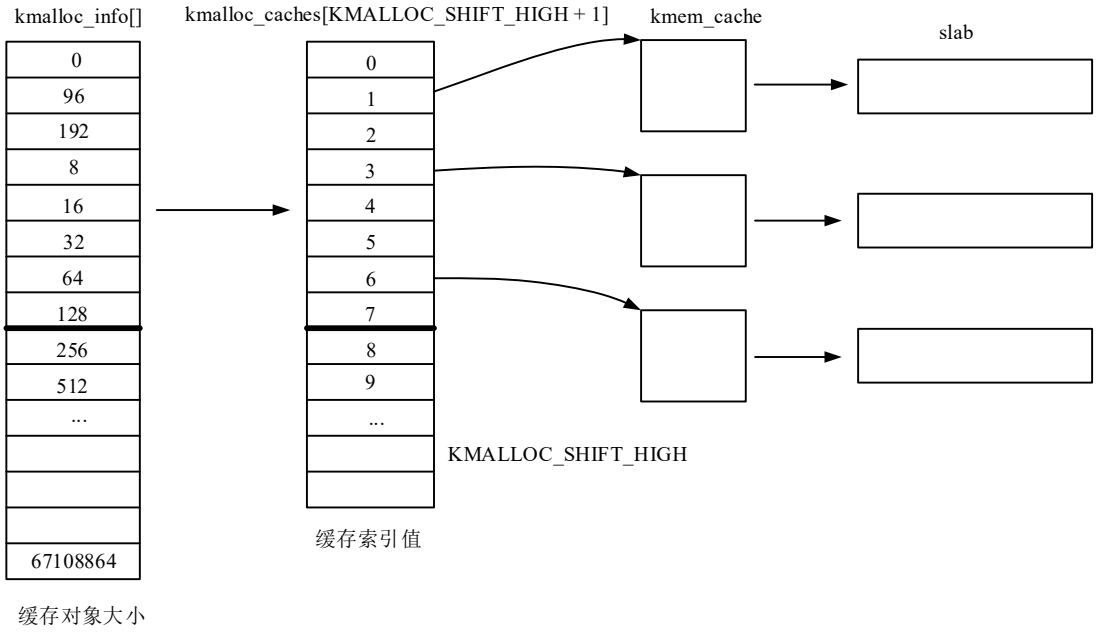
struct kmem_cache ***kmalloc_caches**[KMALLOC_SHIFT_HIGH + 1];

数组项数为 KMALLOC_SHIFT_HIGH + 1，最后一项为空指针，KMALLOC_SHIFT_HIGH 宏定义在头文件/include/linux/slab.h，其值与 MAX_ORDER 和 PAGE_SHIFT 相关，一般为 22。

内核在/mm/slab_common.c 文件内定义了 kmalloc_info[]数组，它与 kmalloc_caches[]数组对应，但其数组项数多于 kmalloc_caches[]数组。kmalloc_info[]数组项表示对应 kmalloc_caches[]数组项关联的 slab 缓存管理对象的大小及缓存名称。

```
static struct {
    const char *name;           /*slab 缓存名称*/
    unsigned long size;         /*缓存对象大小*/
} const kmalloc_info[] __initconst = {          /*第 1、2 项比较特殊*/
    {NULL,                      0},     {"kmalloc-96",          96},
    {"kmalloc-192",           192},     {"kmalloc-8",            8},
    {"kmalloc-16",             16},     {"kmalloc-32",          32},
    {"kmalloc-64",             64},     {"kmalloc-128",        128},
    {"kmalloc-256",           256},     {"kmalloc-512",        512},
    {"kmalloc-1024",         1024},     {"kmalloc-2048",      2048},
    {"kmalloc-4096",         4096},     {"kmalloc-8192",      8192},
    {"kmalloc-16384",       16384},     {"kmalloc-32768",    32768},
    {"kmalloc-65536",       65536},     {"kmalloc-131072",  131072},
    {"kmalloc-262144",     262144},     {"kmalloc-524288",  524288},
    {"kmalloc-1048576",   1048576},     {"kmalloc-2097152", 2097152},
    {"kmalloc-4194304",   4194304},     {"kmalloc-8388608", 8388608},
    {"kmalloc-16777216", 16777216},     {"kmalloc-33554432", 33554432},
    {"kmalloc-67108864", 67108864}      /*支持的最大对象为 64MB*/
};
```

通用缓存结构如下图所示，前 8 项用于小对象（小于 192 字节），注意前 8 项 slab 缓存并不是按对象大小排序的。



内核在/mm/slab_common.c 文件内定义了 size_index[]数组，用于指示分配小内存块（小于 192 字节）使用的通用缓存索引值：

```
static s8 size_index[24] = {
    3,    /*8，分配 8 字节内存，使用通用缓存索引为 3，下同*/
    4,    /* 16 */
    5,    /* 24 */
    5,    /* 32 */
    6,    /* 40 */
    6,    /* 48 */
    6,    /* 56 */
    6,    /* 64 */
    1,    /* 72 */
    1,    /* 80 */
    1,    /* 88 */
    1,    /* 96 */
    7,    /* 104 */
    7,    /* 112 */
    7,    /* 120 */
    7,    /* 128 */
    2,    /* 136 */
    2,    /* 144 */
    2,    /* 152 */
    2,    /* 160 */
    2,    /* 168 */
    2,    /* 176 */
    2,    /* 184 */
    2     /* 192 */
};
```

kmem_cache_init()函数中调用 setup_kmalloc_cache_index_table()函数根据 KMALLOC_MIN_SIZE 宏定义修改部分 size_index[24]数组项，这是为了满足较高对齐要求的处理器架构，一般不需要修改。

kmem_cache_init()函数最后调用 create_kmalloc_caches()函数（/mm/slab_common.c），用于创建通用缓存，函数定义如下：

```
void __init create_kmalloc_caches(unsigned long flags)
/*flags：slab 缓存标记*/
{
    int i;
    /*i 表示 kmalloc_caches[]数组项索引值，起始值不是 0，KMALLOC_SHIFT_LOW 值为 5*/
    for (i = KMALLOC_SHIFT_LOW; i <= KMALLOC_SHIFT_HIGH; i++) {
        if (!kmalloc_caches[i])
            new_kmalloc_cache(i, flags);
                    /*创建 slab 缓存，调用 create_kmalloc_cache()函数，/mm/slab_common.c*/

        if (KMALLOC_MIN_SIZE <= 32 && !kmalloc_caches[1] && i == 6)
            new_kmalloc_cache(1, flags);        /*创建 kmalloc_caches[1]*/
        if (KMALLOC_MIN_SIZE <= 64 && !kmalloc_caches[2] && i == 7)
            new_kmalloc_cache(2, flags);        /*创建 kmalloc_caches[2]*/
    }

    slab_state = UP;       /*通用缓存创建好了*/

  #ifdef CONFIG_ZONE_DMA
    ...
  #endif
}
```

在 create_kmalloc_caches()函数中并不是从 kmalloc_caches[1]数组项（kmalloc_caches[0]数组项为空）开始创建 slab 缓存的。kmalloc_info[]数组从 kmalloc_info[3]数组项开始，管理对象的大小才是从小到大排序的，由于 kmalloc_caches[]数组需要适用 slab、slub 和 slob 三个分配器，因此前几项看起来有点怪怪的。对于 slab 分配器，KMALLOC_SHIFT_LOW 为 5，即从 kmalloc_caches[5]开始创建通用缓存。而后还将根据 KMALLOC_MIN_SIZE 值大小，确定是否创建 kmalloc_caches[1]和 kmalloc_caches[2]关联的 slab 缓存。

在分配内存块时，分配函数会调用 kmalloc_index(size_t size)函数（size 为分配内存大小）确定从数组 kmalloc_caches[]中哪一项关联的 slab 缓存中分配，函数定义在/include/linux/slab.h 头文件，源代码请读者自行阅读。

3 后期初始化

slab 分配器在早期初始化中创建了 kmem_cache 结构体 slab 缓存和通用缓存，kmem_cache_init_late()函数用于完成后期初始化工作。

在早期初始化函数中我们知道，当 slab 分配器没有完成初始化时，初始化 kmem_cache 实例的函数中调用的 setup_cpu_cache()函数为各 CPU 核创建的 array_cache 实例中只有一个缓存对象。

在后期初始化函数中，将为在 slab 分配器初始化阶段创建的 slab 缓存重新创建/设置 array_cache 实例和 kmem_cache_node 实例（含共享对象缓存）。

slab 分配器后期初始化函数为 kmem_cache_init_late()，定义在/mm/slab.c 文件内：

```
void __init kmem_cache_init_late(void)
{
    struct kmem_cache *cachep;

    slab_state = UP;

    mutex_lock(&slab_mutex);
    list_for_each_entry(cachep, &slab_caches, list)          /*遍历所有 kmem_cache 实例*/
        if (enable_cpucache(cachep, GFP_NOWAIT)) /*创建 array_cache 实例等，/mm/slab.c*/
            BUG();
    mutex_unlock(&slab_mutex);

    slab_state = FULL;     /*slab 初始化全部完成*/

    register_cpu_notifier(&cpucache_notifier);     /*向 CPU 通知链注册通知*/

  #ifdef CONFIG_NUMA
    ...
  #endif

}
```

kmem_cache_init_late()函数扫描 kmem_cache 实例链表，对每个实例调用 enable_cpucache()函数，其主要工作是确定 array_cache 实例中的 limit 和 batchcount 参数，然后为各 CPU 核创建关联的 array_cache 实例，并创建/设置 kmem_cache_node 实例（含共享对象缓存）。下一小节讲解创建 slab 分配器函数时，将介绍此函数的实现。至此，slab 分配器初始化全部完成。

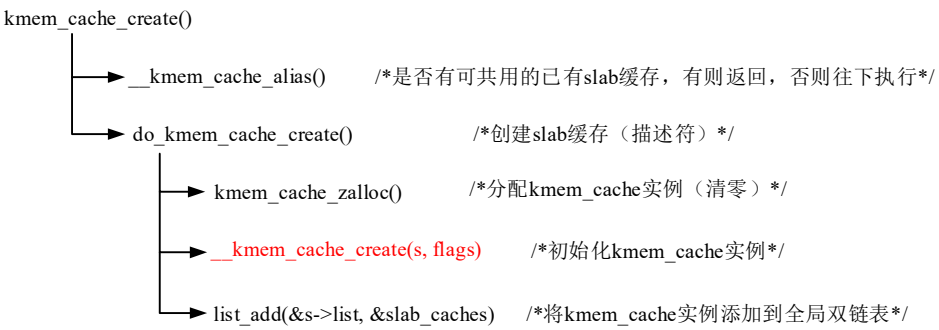
3.7.3 创建 slab 缓存

在 slab 分配器初始化完成之后，内核代码中就可以调用接口函数为某个数据结构创建 slab 缓存了。本小节介绍创建 slab 缓存的 **kmem_cache_create**()接口函数的实现。

1 接口函数

kmem_cache_create()接口函数用于创建 slab 缓存，函数内主要完成 slab 缓存数据结构的创建，但并没有分配保存对象的 slab，在分配函数中当检测到 slab 缓存中没有空闲对象时会触发 slab 的创建（分配）。

kmem_cache_create()接口函数定义在/mm/slab_common.c 文件内，成功则返回 kmem_cache 实例指针，否则返回 NULL，函数调用关系简列如下图所示：



kmem_cache_create()接口函数源代码如下（/mm/slab_common.c）：

```
struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align,unsigned long flags, \
                                     void (*ctor)(void *))
/*name：名称，size：对象实际大小，align：对齐字节数，flags：slab 标记，
*ctor：为 slab 分配页帧时，创建对象时调用的构造函数。*/
{
    struct kmem_cache *s;
    const char *cache_name;
    int err;

    get_online_cpus();
    get_online_mems();
    memcg_get_cache_ids();

    mutex_lock(&slab_mutex);

    err = kmem_cache_sanity_check(name, size);
                    /*需选择 DEBUG_VM 配置选项，否则直接返回 0，/mm/slab_common.c*/
    ...
    flags &= CACHE_CREATE_MASK;     /*保留有效标记位，/mm/slab.h*/

    s = __kmem_cache_alias(name, size, align, flags, ctor);
                                    /*查找是否有现成的 kmem_cache 可共用，/mm/slab.c*/
    if (s)                      /*有可共用的描述符则返回*/
        goto out_unlock;

    cache_name = kstrdup_const(name, GFP_KERNEL);    /*复制名称字符串*/
    ...

    s = do_kmem_cache_create(cache_name, size, size,calculate_alignment(flags, align, size),  \
                                                    flags, ctor, NULL, NULL);
                                            /*创建 kmem_cache 实例，/mm/slab_common.c*/
    ...

out_unlock:
```

```
        mutex_unlock(&slab_mutex);
        memcg_put_cache_ids();
        put_online_mems();
        put_online_cpus();
        ...
        return s;       /*返回 kmem_cache 实例指针*/
    }
```

kmem_cache_create()函数调用__kmem_cache_alias()函数搜索 kmem_cache 实例双链表，查找是否有现成的描述符可共用，如果有可共用描述符则直接返回搜索得 kmem_cache 实例指针。如果没有可共用的描述符则调用函数 **do_kmem_cache_create()**函数创建描述符。do_kmem_cache_create()函数参数中调用的函数 calculate_alignment(flags, align, size)用于计算对象的对齐字节，函数定义在/mm/slab_common.c 文件内，一般为硬件缓存行字节数（设置 SLAB_HWCACHE_ALIGN 标记位）或整型数的字节数。

do_kmem_cache_create()函数代码如下（/mm/slab_common.c）：

```
    static struct kmem_cache *do_kmem_cache_create(const char *name, size_t object_size, size_t size,   \
                    size_t align, unsigned long flags, void (*ctor)(void *),      \
                    struct mem_cgroup *memcg, struct kmem_cache *root_cache)
    /*name：名称字符串，object_size：对象实际大小，size：对象对齐后大小，align：对齐数值，
    *flags：标记，ctor：对象构造函数指针，最后两个参数值为 NULL。*/
    {
        struct kmem_cache *s;
        int err;

        err = -ENOMEM;
        s = kmem_cache_zalloc(kmem_cache, GFP_KERNEL);   /*/include/linux/slab.h*/
                                                /*从 kmem_cache 结构体 slab 缓存中分配实例，并清零*/
        if (!s)
            goto out;

        s->name = name;                  /*赋值名称字符串*/
        s->object_size = object_size;    /*对象实际大小*/
        s->size = size;                  /*对齐后的对象大小，初始值与实际大小相同*/
        s->align = align;                /*对齐字节数*/
        s->ctor = ctor;                  /*构造函数*/

        err = init_memcg_params(s, memcg, root_cache);    /*初始化内存控制组参数*/
        ...
        err = __kmem_cache_create(s, flags);    /*kmem_cache 实例初始化，前面介绍过，/mm/slab.c*/
        ...
        s->refcount = 1;                        /*引用计数设为 1*/
        list_add(&s->list, &slab_caches);       /*将实例添加到全局双链表*/
    out:
        ...
        return s;       /*返回 kmem_cache 实例指针*/
```

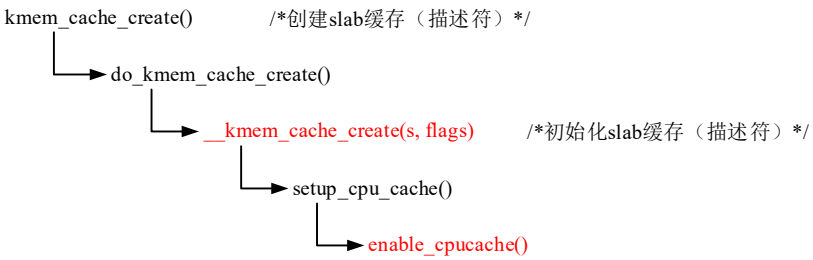
```
    ...
}
```

do_kmem_cache_create()函数首先调用 kmem_cache_zalloc()函数从 slab 缓存中分配 kmem_cache 实例并清零，然后对实例进行设置并调用__kmem_cache_create(s, flags)函数进一步初始化，主要是创建并设置关联的 array_cache 和 kmem_cache_node 实例（前面已介绍过此函数），将 kmem_cache 实例添加到全局双链表，最后返回 kmem_cache 实例指针。

在调用 kmem_cache_create()接口函数时，slab 分配器已经初始化完成了，__kmem_cache_create()函数内将最终调用 enable_cpucache()函数创建/设置 array_cache 和 kmem_cache_node 实例，详见下文。

3 初始化对象缓存

do_kmem_cache_create()函数在分配 kmem_cache 实例后，调用__kmem_cache_create()函数初始化 slab 缓存，主要是确定 slab 大小、布局，以及创建关联的 array_cache 和 kmem_cache_node 实例等，函数调用关系如下图所示：



__kmem_cache_create()函数在前面介绍过了，主要看一下其中的 setup_cpu_cache()函数，因为此时 slab 分配器已经初始化完成了，setup_cpu_cache()函数将调用 enable_cpucache()函数创建关联的 array_cache 和 kmem_cache_node 实例。

setup_cpu_cache()函数代码简列如下（/mm/slab.c）：

```
static int __init_refok setup_cpu_cache(struct kmem_cache *cachep, gfp_t gfp)
/*gfp：分配掩码，用于从通用缓存中分配 kmem_cache_node 实例时使用*/
{
    if (slab_state >= FULL)                 /*slab 分配器已完全初始化*/
            return enable_cpucache(cachep, gfp);        /*/mm/slab.c*/

    ...         /*slab 分配器未完成初始化时执行的操作，前面介绍过*/
}
```

enable_cpucache()函数定义如下（/mm/slab.c）：

```
static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp)
{
    int err;
    int limit = 0;              /*CPU 缓存中对象阀值（最大值）*/
    int shared = 0;             /*共享对象数量*/
    int batchcount = 0;         /*批处理对象数量*/

    if (!is_root_cache(cachep)) {       /*若没有配置 MEMCG_KMEM 选项，函数返回 true，/mm/slab.h*/
        struct kmem_cache *root = memcg_root_cache(cachep);
```

```
            limit = root->limit;
            shared = root->shared;
            batchcount = root->batchcount;
        }

        if (limit && shared && batchcount)
            goto skip_setup;

        if (cachep->size > 131072)          /*确定 array_cache 缓存中空闲对象阀值*/
            limit = 1;
        else if (cachep->size > PAGE_SIZE)
            limit = 8;
        else if (cachep->size > 1024)
            limit = 24;
        else if (cachep->size > 256)
            limit = 54;
        else
            limit = 120;

        shared = 0;
        if (cachep->size <= PAGE_SIZE && num_possible_cpus() > 1)       /*CPU 数量需大于 1*/
            shared = 8;       /*设置共享缓存对象数量，单 CPU 时共享对象数量为 0*/

        #if DEBUG
            ...
        #endif

        batchcount = (limit + 1) / 2;         /*批处理对象数量为空闲对象阀值的一半*/
    skip_setup:
        err = do_tune_cpucache(cachep, limit, batchcount, shared, gfp);     /*/mm/slab.c*/
        if (err)
            ...
        return err;
    }
```

enable_cpucache()函数确定 array_cache 缓存中对象数量阀值、共享缓存对象数量及批处理对象数量后，调用 do_tune_cpucache()函数为 slab 缓存创建 array_cache 和 kmem_cache_node 实例，函数定义如下。

```
    static int do_tune_cpucache(struct kmem_cache *cachep, int limit,int batchcount, int shared, gfp_t gfp)
    {
        int ret;
        struct kmem_cache *c;

        ret = __do_tune_cpucache(cachep, limit, batchcount, shared, gfp);       /*/mm/slab.c*/
```

```
    if (slab_state < FULL)
        return ret;

    if ((ret < 0) || !is_root_cache(cachep))
        return ret;

    lockdep_assert_held(&slab_mutex);
    for_each_memcg_cache(c, cachep) {
        __do_tune_cpucache(c, limit, batchcount, shared, gfp);
    }
    return ret;
}
```

do_tune_cpucache()函数内调用__do_tune_cpucache()函数完成实例创建工作，函数定义如下：

```
static int __do_tune_cpucache(struct kmem_cache *cachep, int limit,int batchcount, int shared, gfp_t gfp)
{
    struct array_cache __percpu *cpu_cache, *prev;
    int cpu;

    cpu_cache = alloc_kmem_cache_cpus(cachep, limit, batchcount);       /*分配 percpu 变量*/
        /*分配各 CPU 关联 array_cache 实例并初始化，对象指针数组项数为 limit，/mm/slab.c*/
    if (!cpu_cache)
        return -ENOMEM;

    prev = cachep->cpu_cache;       /*原先关联的 array_cache 实例*/
    cachep->cpu_cache = cpu_cache;
    kick_all_cpus_sync();

    check_irq_on();
    cachep->batchcount = batchcount;     /*参数赋予 kmem_cache 实例*/
    cachep->limit = limit;
    cachep->shared = shared;

    if (!prev)        /*跳转至创建 kmem_cache_node 实例，不需要释放原 CPU 对象缓存*/
        goto alloc_node;

    for_each_online_cpu(cpu) {       /*释放原 CPU 对象缓存中对象*/
        LIST_HEAD(list);
        int node;
        struct kmem_cache_node *n;
        struct array_cache *ac = per_cpu_ptr(prev, cpu);
```

```
            node = cpu_to_mem(cpu);
            n = get_node(cachep, node);
            spin_lock_irq(&n->list_lock);
            free_block(cachep, ac->entry, ac->avail, node, &list);   /*将 CPU 缓存对象释放到 slab*/
            spin_unlock_irq(&n->list_lock);
            slabs_destroy(cachep, &list);      /*释放空闲的 slab（若空闲对象数超过阀值）*/
        }
        free_percpu(prev);

    alloc_node:
        return alloc_kmem_cache_node(cachep, gfp);   /*创建 kmem_cache_node 实例，/mm/slab.c*/
    }
```

__do_tune_cpucache()函数内调用 alloc_kmem_cache_cpus()函数创建 array_cache 实例（分配 percpu 变量），实例中缓存对象指针数为 limit，即各 CPU 缓存中包含 limit 数量的对象。如果原先关联有 array_cache 实例，则释放缓存的对象至 slab。最后调用 alloc_kmem_cache_node()函数为 slab 缓存创建 kmem_cache_node 实例（数组）。

alloc_kmem_cache_node()函数定义如下（/mm/slab.c）：

```
static int alloc_kmem_cache_node(struct kmem_cache *cachep, gfp_t gfp)
{
    int node;
    struct kmem_cache_node *n;
    struct array_cache *new_shared;
    struct alien_cache **new_alien = NULL;

    for_each_online_node(node) {        /*遍历所有内存结点，这里只考虑单结点*/
        if (use_alien_caches) {     /*use_alien_caches 初始为 1，通过"noaliencache"命令行参数清零*/
            new_alien = alloc_alien_cache(node, cachep->limit, gfp);
                                        /*返回 BAD_ALIEN_MAGIC 常数，UMA 系统*/
            if (!new_alien)
                goto fail;
        }

        new_shared = NULL;
        if (cachep->shared) {     /*共享缓存中对象数量，创建管理共享缓存对象的 array_cache 实例*/
            new_shared = alloc_arraycache(node,cachep->shared*cachep->batchcount,   \
                                          0xbaadf00d, gfp);
            /*从通用缓存中分配 array_cache 实例并初始化，用于缓存共享对象，
            *0xbaadf00d 为 batchcount 成员初始值，/mm/slab.c*/
            ...
        }

        n = get_node(cachep, node);        /*原 kmem_cache_node 实例指针*/
```

```
        if (n) {        /*原 kmem_cache_node 实例非空，释放共享缓存对象*/
            struct array_cache *shared = n->shared;
            LIST_HEAD(list);

            spin_lock_irq(&n->list_lock);

            if (shared)
                free_block(cachep, shared->entry,shared->avail, node, &list);   /*释放共享缓存对象*/

            n->shared = new_shared;
            if (!n->alien) {
                n->alien = new_alien;
                new_alien = NULL;
            }
            n->free_limit = (1 + nr_cpus_node(node)) *cachep->batchcount + cachep->num;
            spin_unlock_irq(&n->list_lock);
            slabs_destroy(cachep, &list);       /*释放全部空闲 slab*/
            kfree(shared);
            free_alien_cache(new_alien);
            continue;       /*遍历下一节点*/
        }
        /*原 kmem_cache_node 实例为空，重新创建*/
        n = kmalloc_node(sizeof(struct kmem_cache_node), gfp, node);
                                        /*从通用缓存中重新分配 kmem_cache_node 实例*/
        ...
        kmem_cache_node_init(n);        /*初始化 kmem_cache_node 实例，/mm/slab.c*/
        n->next_reap = jiffies + REAPTIMEOUT_NODE +     \
                ((unsigned long)cachep) % REAPTIMEOUT_NODE;       /*缓存收缩间隔时间*/
        n->shared = new_shared;
        n->alien = new_alien;
        n->free_limit = (1 + nr_cpus_node(node)) *cachep->batchcount + cachep->num;
                                                                  /*空闲对象阀值*/
        cachep->node[node] = n;       /*kmem_cache 关联 kmem_cache_node 实例*/
    }   /*遍历内存结点结束*/
    return 0;
    ...
}
```

至此，kmem_cache_create()函数创建 slab 缓存的工作结束，主要是创建并初始化 slab 缓存描述符数据结构实例，而真正保存对象的 slab 并没有分配。在分配对象时，当 slab 缓存中没有空闲对象时，将从伙伴系统中为 slab 缓存分配内存块，用于创建 slab，然后再分配对象。下一小节将介绍 slab 分配函数。

3.7.4 分配对象

内核代码调用 kmem_cache_create()函数创建 slab 缓存后，随后就可以调用 kmem_cache_alloc()分配函

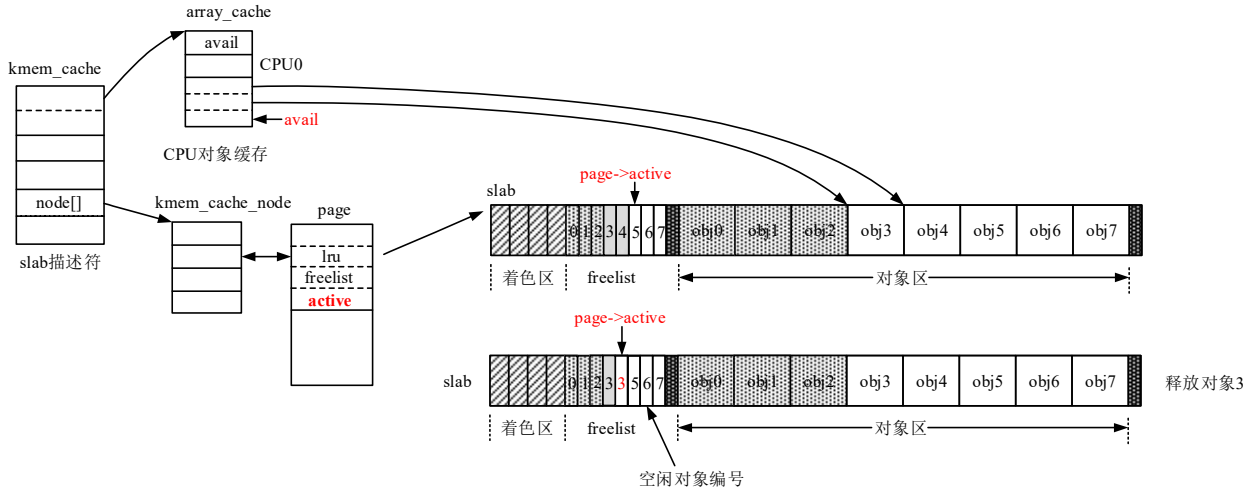
数分配对象了。前面介绍过新创建的 slab 缓存其实是空的，并没有缓存对象，因此分配函数需要先从伙伴系统中分配内存块创建 slab，然后才能分配对象，这称它为缓存扩展。当 slab 缓存中没有空闲对象时，如果再需要分配的话也需要先执行缓存扩展。

下面将介绍 slab 分配函数的实现，下一小节将介绍释放对象函数的实现。

1 对象管理

在介绍分配和释放对象函数之前，先介绍一下从 slab 中分配与释放对象的操作原理，slab 对象管理如下图所示。在分配和释放对象时，都直接操作于本地 CPU 对象缓存。分配对象时，从 CPU 对象缓存中获取对象，若 CPU 对象缓存为空，则将共享对象缓存（下图中未画出）转移至 CPU 对象缓存后再分配；若没有共享缓存或共享缓存为空，则从 slab 中获取对象转移至 CPU 本地缓存，然后再分配；若 slab 中空闲对象不足（或没有 slab），则先扩展缓存，创建 slab 后，再移动对象至 CPU 对象缓存，最后再分配对象。

释放对象时，直接将对象释放至本地 CPU 对象缓存。若本地 CPU 缓存已满，则先将本地 CPU 对象缓存中部分对象转移至共享缓存，若共享缓存也满了或不存在，则将对象放回 slab 中，然后再释放对象到本地 CPU 对象缓存。



在 slab 中 freelist 区记录的是空闲对象编号，位于 CPU 对象缓存及共享对象缓存中的对象，在 freelist 区中视为已经分配出去的对象，而不是空闲对象。如上图中上侧 slab 所示，假设 0、1、2 对象真实地分配出去了，而 3、4 对象位于 CPU 对象缓存，则 0-4 对象在 freelist 区都视为已分配出去（非空闲），空闲对象从对象 5 开始。

当 CPU 对象缓存中的对象 3 释放回 slab 时，freelist[page->active]数组项先前移一项，编号 3 将写入当前数组项。

表示 CPU 对象缓存的 array_cache 实例中，avail 成员记录的是当前缓存对象数量，limit 成员记录的是缓存对象最大数量，batchcount 成员记录的是批处理缓存对象的数量。分配对象操作中，当 CPU 缓存对象为空时，将从共享对象缓存或 slab 中转移最多 batchcount 数量的对象至 CPU 对象缓存，然后再分配。释放对象时，若 CPU 对象缓存已满（数量为 limit），则先将 batchcount 数量的缓存对象转移至共享对象缓存或 slab 中，然后再释放对象至 CPU 对象缓存。

2 分配函数

从 slab 缓存中分配对象的接口函数为 kmem_cache_alloc()，函数定义如下（/mm/slab.c）：

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
/*cachep：指向 kmem_cache 实例，flags：从伙伴系统为 slab 分配内存块的分配掩码*/
```

{

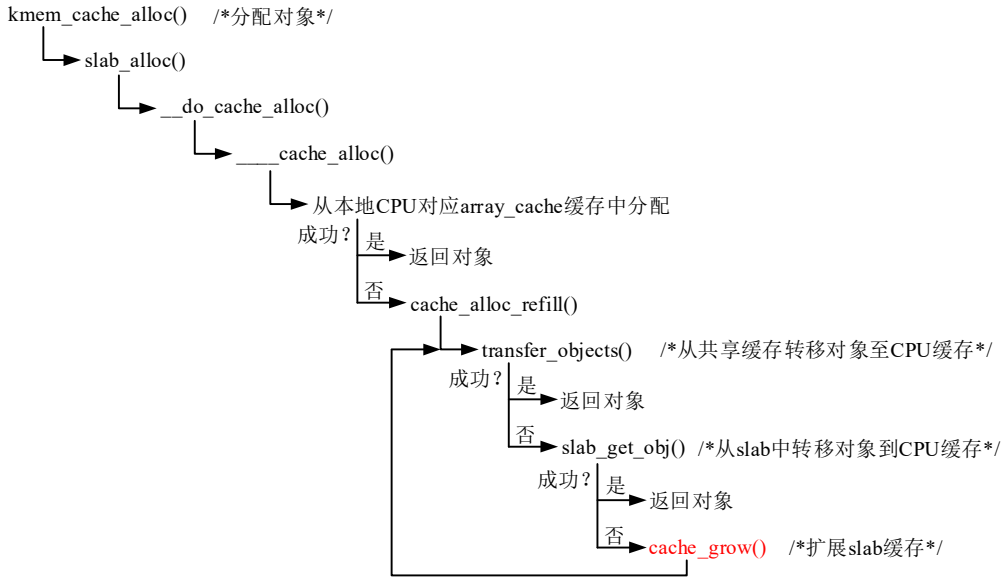
void *ret = slab_alloc(cachep, flags, _RET_IP_); /*分配对象，_RET_IP_为 0，/mm/slab.c*/

trace_kmem_cache_alloc(_RET_IP_, ret,cachep->object_size, cachep->size, flags);

return ret;

}

kmem_cache_alloc()函数从指定 slab 缓存中分配对象，返回对象指针，函数内直接调用 slab_alloc()函数。分配对象函数调用关系如下图所示：



分配对象操作先从本地 CPU 关联的对象缓存 array_cache 中分配，如果分配成功则返回对象指针，如果不成功继续往下执行；调用 cache_alloc_refill()函数，如果存在共享对象缓存，则将共享缓存对象转移到 CPU 对象缓存，然后分配，成功则返回对象指针，不成功继续往下执行；调用 slab_get_obj()函数从现有 slab 中获取对象转移到本地 CPU 对象缓存，而后分配，成功则返回对象指针，不成功继续往下执行；调用函数 cache_grow()函数，扩展 slab 缓存，即从伙伴系统分配内存块，创建 slab，然后再重复以上分配操作。

下面从 slab_alloc()函数开始介绍以上各函数的实现，slab_alloc()函数定义如下（/mm/slab.c）：

static __always_inline void *slab_alloc(struct kmem_cache *cachep, gfp_t flags, unsigned long caller)

/*flags：伙伴系统分配掩码*/

{

unsigned long save_flags;

void *objp;

flags &= gfp_allowed_mask; /*不屏蔽任何分配掩码标记位*/

lockdep_trace_alloc(flags);

if (slab_should_failslab(cachep, flags))

return NULL;

cachep = memcg_kmem_get_cache(cachep, flags);

```
    cache_alloc_debugcheck_before(cachep, flags);

    local_irq_save(save_flags);
    objp = __do_cache_alloc(cachep, flags);      /*分配操作，/mm/slab.c*/
    local_irq_restore(save_flags);

    objp = cache_alloc_debugcheck_after(cachep, flags, objp, caller);
    kmemleak_alloc_recursive(objp, cachep->object_size, 1, cachep->flags,flags);
    prefetchw(objp);

    if (likely(objp)) {
        kmemcheck_slab_alloc(cachep, flags, objp, cachep->object_size);
        if (unlikely(flags & __GFP_ZERO))
            memset(objp, 0, cachep->object_size);     /*对象清零*/
    }
    memcg_kmem_put_cache(cachep);
    return objp;      /*返回对象指针*/
}
```

slab_alloc()函数调用__do_cache_alloc(cachep, flags)函数完成分配工作，此函数内又直接调用函数____cache_alloc(cachep, flags)（!NUMA），定义如下（/mm/slab.c）：

```
static inline void *____cache_alloc(struct kmem_cache *cachep, gfp_t flags)
{
    void *objp;
    struct array_cache *ac;
    bool force_refill = false;

    check_irq_off();

    ac = cpu_cache_get(cachep);     /*本地 CPU 对应 array_cache 缓存*/
    if (likely(ac->avail)) {              /*本地 CPU 对象缓存不为空，则从中分配*/
        ac->touched = 1;
        objp = ac_get_obj(cachep, ac, flags, false);
        /*返回 array_cache 实例中 entry[--ac->avail]指向的对象，avail 为缓存对象数量，/mm/slab.c*/
        if (objp) {
            STATS_INC_ALLOCHIT(cachep);
            goto out;      /*跳至函数末尾*/
        }
        force_refill = true;
    }

    /*本地 CPU 对象缓存为空*/
```

```
        STATS_INC_ALLOCMISS(cachep);
        objp = cache_alloc_refill(cachep, flags, force_refill);
                                                /*填充本地 CPU 对象缓存并分配，/mm/slab.c*/
        ac = cpu_cache_get(cachep);

    out:
        if (objp)
            kmemleak_erase(&ac->entry[ac->avail]);
        return objp;
    }
```

____cache_alloc()函数先判断本地 CPU 对象缓存是否为空，如果不为空则直接返回 entry[]对象数组中最后一个缓存对象 entry[--ac->avail]指针即可，如果本地 CPU 对象缓存为空则调用 cache_alloc_refill()函数填充本地 CPU 对象缓存后再分配。

cache_alloc_refill()函数定义如下（/mm/slab.c）：

```
static void *cache_alloc_refill(struct kmem_cache *cachep, gfp_t flags,bool force_refill)
/*flags：分配掩码*/
{
    int batchcount;
    struct kmem_cache_node *n;
    struct array_cache *ac;
    int node;

    check_irq_off();
    node = numa_mem_id();
    if (unlikely(force_refill))
        goto force_grow;
retry:
    ac = cpu_cache_get(cachep);      /*本地 CPU 缓存 array_cache 实例*/
    batchcount = ac->batchcount;
    if (!ac->touched && batchcount > BATCHREFILL_LIMIT) {
        batchcount = BATCHREFILL_LIMIT;
    }
    n = get_node(cachep, node);

    BUG_ON(ac->avail > 0 || !n);
    spin_lock(&n->list_lock);

    /*将共享对象缓存转移到 CPU 对象缓存*/
    if (n->shared && transfer_objects(ac, n->shared, batchcount)) {
        n->shared->touched = 1;
        goto alloc_done;
    }
```

```
    /*没有共享对象缓存*/
    while (batchcount > 0) {        /*从 slab 链表中获取 batchcount 数量对象，添加到本地 CPU 缓存*/
        struct list_head *entry;
        struct page *page;
        entry = n->slabs_partial.next;     /*部分空闲 slab 链表*/
        if (entry == &n->slabs_partial) {     /*部分空闲 slab 链表为空*/
            n->free_touched = 1;
            entry = n->slabs_free.next;        /*搜索全部空闲 slab 链表*/
            if (entry == &n->slabs_free)        /*如果全部空闲 slab 链表为空，扩展缓存*/
                goto must_grow;                 /*跳至缓存扩展处*/
        }

        /*从 slab 链表中，获取对象转移到本地 CPU 对象缓存*/
        page = list_entry(entry, struct page, lru);       /*取出链表中首个 slab*/
        check_spinlock_acquired(cachep);

        BUG_ON(page->active >= cachep->num);

        while (page->active < cachep->num && batchcount--) {     /*转移 slab 中对象至 CPU 缓存*/
            STATS_INC_ALLOCED(cachep);
            STATS_INC_ACTIVE(cachep);
            STATS_SET_HIGH(cachep);
            ac_put_obj(cachep, ac, slab_get_obj(cachep, page,node));
                                                /*从 slab 中获取对象添加到本地 CPU 对象缓存*/
        }

        list_del(&page->lru);
        if (page->active == cachep->num)
            list_add(&page->lru, &n->slabs_full);   /*slab 中对象全部使用完则添加到全部使用链表*/
        else
            list_add(&page->lru, &n->slabs_partial);   /*slab 中还有空闲对象则添加到部分空闲链表*/
    }    /*while 循环结束，从 slab 中获取对象结束*/

    /*从 CPU 对象缓存中分对象，可能需要扩展 slab 缓存*/
must_grow:
    n->free_objects -= ac->avail;     /*减少 slab 中空闲对象数量*/
alloc_done:
    spin_unlock(&n->list_lock);

    if (unlikely(!ac->avail)) {       /*若本地 CPU 对象缓存为空*/
        int x;
force_grow:
        x = cache_grow(cachep, gfp_exact_node(flags), node, NULL);     /*扩展 slab 缓存*/
```

```
        ac = cpu_cache_get(cachep);
        node = numa_mem_id();

        if (!x && (ac->avail == 0 || force_refill))
            return NULL;

        if (!ac->avail)        /*分配 slab 后重新分配对象*/
            goto retry;        /*跳转至 retry 处*/
    }
    ac->touched = 1;

    return ac_get_obj(cachep, ac, flags, force_refill);
                                /*返回 array_cache 实例中 entry[--ac->avail]指向的缓存对象*/
}
```

由以上分析可知，分配对象始终是从 CPU 关联的 array_cache 实例中分配。如果 array_cache 实例中缓存对象不为空，则直接分配，否则按以下顺序获取对象填充 array_cache 实例中对象缓存后再分配。

（1）如果存在共享对象缓存，则将共享对象缓存转移到本地 CPU 对象缓存。

（2）扫描 slab 链表，从 slab 中获取最多 batchcount 个空闲对象填充至本地 CPU 对象缓存，如果 slab 中没有空闲对象了则先扩展 slab 缓存。

下面介绍一下对象在共享对象缓存、slab 链表中如何转移至本地 CPU 对象缓存，以及扩展 slab 缓存的实现。

■转移对象

在 cache_alloc_refill()函数中，调用 transfer_objects()函数将共享对象缓存转移到本地 CPU 对象缓存，函数内只是将 kmem_cache_node 实例 shared 成员指向的 array_cache 实例中对象指针复制到本地 CPU 关联的 array_cache 实例中，并修改两个缓存中对象数量，函数源代码如下所示（/mm/slab.c）：

```
static int transfer_objects(struct array_cache *to,struct array_cache *from, unsigned int max)
{
    int nr = min3(from->avail, max, to->limit - to->avail);       /*确定转移对象数量*/

    if (!nr)
        return 0;

    memcpy(to->entry + to->avail, from->entry + from->avail -nr,sizeof(void *) *nr); /*复制对象指针*/

    from->avail -= nr;      /*修改缓存中对象数量*/
    to->avail += nr;
    return nr;       /*转移对象数量*/
}
```

如果共享对象缓存为空，则需要扫描 slab 链表，从中获取最多 batchcount 数量对象，移动到本地 CPU 对象缓存（array_cache 实例）。ac_put_obj()函数用于将对象关联到本地 CPU 对象缓存，slab_get_obj()函

数用于从 slab 中获取一个空闲对象。

slab_get_obj()函数定义如下（/mm/slab.c）：

```
static void *slab_get_obj(struct kmem_cache *cachep, struct page *page,int nodeid)
/*page：slab 首页 page 实例指针，nodeid：内存结点*/
{
    void *objp;

    objp = index_to_obj(cachep, page, get_free_obj(page, page->active));   /*编号到对象指针*/
    page->active++;        /*freelist 数组项后移一位，指向空闲对象数组*/
  #if DEBUG
    ...
  #endif
    return objp;     /*返回对象指针*/
}
```

slab_get_obj()函数调用 index_to_obj()函数获取 slab 中编号为 idx 对象的指针，函数定义如下：

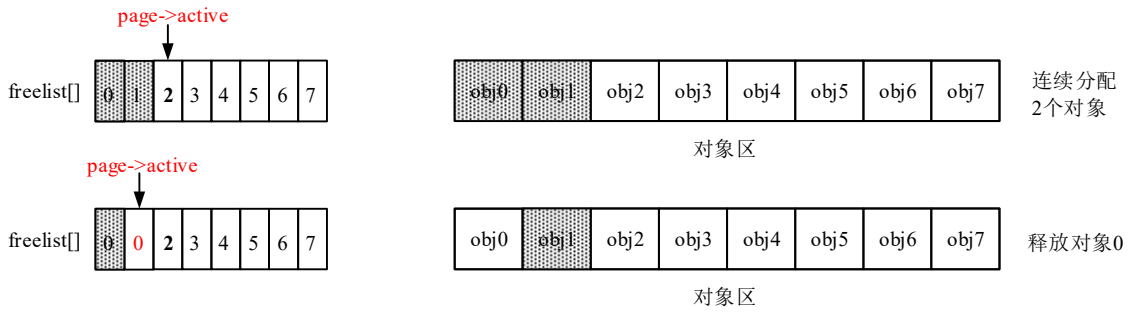
```
static inline void *index_to_obj(struct kmem_cache *cache, struct page *page,unsigned int idx)
/*idx：空闲对象编号*/
{
    return page->s_mem + cache->size * idx;     /*编号转对象指针*/
}
```

空闲对象编号在 slab_get_obj()函数中由 get_free_obj()函数获取，get_free_obj()函数也很简单，定义如下（/mm/slab.c）：

```
static inline freelist_idx_t get_free_obj(struct page *page, unsigned int idx)
/*idx：freelist 数组项索引值，此处为 page->active*/
{
    return ((freelist_idx_t *)page->freelist)[idx];   /*返回 freelist[page->active]内保存的空闲对象编号*/
}
```

freelist[]数组用来记录 slab 中空闲对象的编号，如下图所示。起始状态 freelist[]数组中依次保存对象的编号，page->active 为 0。分配一个对象时，返回 freelist[page->active]数组项中保存的对象编号，page->active 值加 1（右移一个数组项）。下图中上半部分所示是连续分配了 2 个对象后的结果。

释放对象时，page->active 值先减 1（左移一个数组项），然后将释放对象的编号写入 freelist[page->active]数组项。下图中下半部分是释放对象 0 后的结果。



由以上分析可知，freelist[]数组中 page->active 及以后的数组项中保存的是空闲对象的编号（最后释放的在最前面），page->active 之前的数组项保存的是全部或部分已分配对象的编号。在分配对象时，则分配 page->active 数组项保存编号对应的对象。

■缓存扩展

在分配函数中，如果要从 slab 中转移对象至本地 CPU 对象缓存 array_cache 实例中，当 slab 中没有空闲时，将调用 cache_grow()函数扩展 slab 缓存。

cache_grow()函数的主要工作是从伙伴系统中分配内存块创建并设置 slab，并添加到 kmem_cache_node 实例的空闲 slab 双链表中，函数成功返回 1，否则返回 0。

cache_grow()函数定义在/mm/slab.c 文件内，代码如下：

```
static int cache_grow(struct kmem_cache *cachep,gfp_t flags, int nodeid, struct page *page)
/*cachep：指向 kmem_cache 实例，flags：分配掩码，nodeid：内存结点，page：指向首页 page 实例*/
{
    void *freelist;
    size_t offset;
    gfp_t local_flags;
    struct kmem_cache_node *n;

    ...
    local_flags = flags & (GFP_CONSTRAINT_MASK|GFP_RECLAIM_MASK);
                                        /*分配掩码，只能从本内存结点低端内存中分配*/
    check_irq_off();
    n = get_node(cachep, nodeid);       /*kmem_cache_node 实例*/
    spin_lock(&n->list_lock);

    offset = n->colour_next;      /*下一颜色值*/
    n->colour_next++;             /*下一颜色值加*/
    if (n->colour_next >= cachep->colour)      /*下一颜色值达到最大值，重置为 0*/
        n->colour_next = 0;
    spin_unlock(&n->list_lock);

    offset *= cachep->colour_off;       /*计算 slab 着色区长度*/

    if (local_flags & __GFP_WAIT)       /*分配函数允许睡眠*/
        local_irq_enable();             /*开中断*/

    kmem_flagcheck(cachep, flags);     /*主要检查 DMA 域分配掩码，/mm/slab.c*/

    if (!page)
        page = kmem_getpages(cachep, local_flags, nodeid);      /*/mm/slab.c*/
          /*从伙伴系统分配内存块，分配掩码与 cachep->allocflags 或，即设置复合页标记位，
          *函数返回首页 page 实例指针并设置 PG_slab 标记位。*/
    if (!page)
```

```
        goto failed;

    freelist = alloc_slabmgmt(cachep, page, offset,local_flags & ~GFP_CONSTRAINT_MASK, nodeid);
                                                        /*创建 freelist 数组，/mm/slab.c*/
    if (!freelist)
        goto opps1;

    slab_map_pages(cachep, page, freelist);     /*设置首页 page 实例，/mm/slab.c*/

    cache_init_objs(cachep, page);              /*对象初始化，/mm/slab.c*/

    if (local_flags & __GFP_WAIT)
        local_irq_disable();
    check_irq_off();
    spin_lock(&n->list_lock);

    list_add_tail(&page->lru, &(n->slabs_free));      /*将 slab 添加到空闲 slab 链表末尾*/
    STATS_INC_GROWN(cachep);
    n->free_objects += cachep->num;           /*增加空闲对象数量*/
    spin_unlock(&n->list_lock);
    return 1;       /*成功返回 1*/
    ...
}
```

cache_grow()函数主要工作如下：

●确定分配掩码和着色区长度。

●调用 kmem_getpages()从伙伴系统中分配内存块，用于创建 slab，内存块阶数保存在 kmem_cache 实例中，参数传递的分配掩码与 cachep->allocflags 进行或操作，即设置复合页标记位，函数返回首页 page 实例指针，函数定义在/mm/slab.c 文件内，源代码请读者自行阅读。

●调用 alloc_slabmgmt()函数为 freelist 区分配空间或从 slab 中为 freelist 区划分空间，源代码见下文。

●调用 slab_map_pages()函数设置 slab 首页 page 实例，源代码见下文。

●调用 cache_init_objs()函数初始化 slab 及其中对象，设置所有对象为空闲，源代码见下文。

●将 slab 添加到 kmem_cache_node 实例的空闲 slab 链表末尾，增加空闲对象数量统计值。

下面介绍 alloc_slabmgmt()、slab_map_pages()和 cache_init_objs()函数的实现。

（1）创建 freelist 区

alloc_slabmgmt()函数为 freelist 区分配空间或从 slab 中为 freelist 区划分空间，源代码如下（/mm/slab.c）：

```
static void *alloc_slabmgmt(struct kmem_cache *cachep,struct page *page, int colour_off,  \
                                                        gfp_t local_flags, int nodeid)
/*colour_off：着色区长度，local_flags：分配掩码*/
{
    void *freelist;
```

```
    void *addr = page_address(page);      /*页帧起始虚拟地址（内核直接映射区）*/

    if (OFF_SLAB(cachep)) {      /*freelist 区在 slab 外部，则从通用缓存中分配*/
        freelist = kmem_cache_alloc_node(cachep->freelist_cache,local_flags, nodeid);
        if (!freelist)
            return NULL;
    } else {                  /* freelist 区在 slab 内部，则在着色区后划出 freelist 区*/
        freelist = addr + colour_off;      /*跳过着色区*/
        colour_off += cachep->freelist_size;
    }
    page->active = 0;           /*slab 中活动对象（已使用对象）数量初始化为 0*/
    page->s_mem = addr + colour_off;      /*指向第一个对象*/
    return freelist;
}
```

（2）设置首页 page 实例

slab_map_pages()函数用于设置 slab 首页 page 实例中的相关成员，定义如下（/mm/slab.c）：

```
static void slab_map_pages(struct kmem_cache *cache, struct page *page,void *freelist)
{
    page->slab_cache = cache;      /*指向 kmem_cache 实例*/
    page->freelist = freelist;          /*指向 freelist 区*/
}
```

（3）对象初始化

cache_init_objs()函数用于初始化 slab 中对象，函数定义如下（/mm/slab.c）：

```
static void cache_init_objs(struct kmem_cache *cachep,struct page *page)
{
    int i;

    for (i = 0; i < cachep->num; i++) {            /*遍历 slab 中所有对象*/
        void *objp = index_to_obj(cachep, page, i);      /*编号转对象指针*/
    #if DEBUG
        ...
    #else
        if (cachep->ctor)           /*调用构造函数*/
            cachep->ctor(objp);
    #endif
        set_obj_status(page, i, OBJECT_FREE); /*设置 slab 状态，需选配 DEBUG_SLAB_LEAK*/
        set_free_obj(page, i, i);     /*设置对象空闲，/mm/slab.c*/
    }  /*遍历对象结束*/
}
```

cache_init_objs()函数遍历 slab 中的每个对象并调用构造函数，并调用 set_free_obj()函数设置对象空闲，

函数定义如下（/mm/slab.c）：

```
static inline void set_free_obj(struct page *page,unsigned int idx, freelist_idx_t val)
{
    ((freelist_idx_t *)(page->freelist))[idx] = val;     /*将对象编号写入 freelist[]数组*/
}
```

3 通用缓存分配函数

kmem_cache_alloc()函数用于从指定的 slab 缓存中分配对象，在 slab 分配器初始化阶段还创建了一组通用缓存。通用缓存只按大小来管理对象，而不是按对象表示的数据类型管理对象。

从通用缓存中分配对象（内存块），只需要指明分配对象的大小，分配函数会自动查找合适的通用缓存，从中分配对象，实际分配的对象可能比申请的要大。

从通用缓存中分配对象的接口函数是 **kmalloc()**，定义如下（/include/linux/slab.h）：

```
static __always_inline void *kmalloc(size_t size, gfp_t flags)
/*size：分配对象大小，flags：创建 slab 时的分配掩码*/
{
    if (__builtin_constant_p(size)) {
        if (size > KMALLOC_MAX_CACHE_SIZE)
            return kmalloc_large(size, flags);       /*从伙伴系统分配*/
    #ifndef CONFIG_SLOB
        ...
    #endif
    }
    return __kmalloc(size, flags);       /*/mm/slab.c*/
}
```

kmalloc()函数内调用__kmalloc()函数分配对象，函数定义在/mm/slab.c 文件内：

```
void *__kmalloc(size_t size, gfp_t flags)
{
    return __do_kmalloc(size, flags, _RET_IP_);       /*/mm/slab.c*/
}
```

__do_kmalloc()函数定义在/mm/slab.c 文件内，代码如下：

```
static __always_inline void *__do_kmalloc(size_t size, gfp_t flags,unsigned long caller)
{
    struct kmem_cache *cachep;
    void *ret;

    cachep = kmalloc_slab(size, flags);
        /*选择合适的通用缓存 kmem_cache 实例，源代码请读者自行阅读，/mm/slab_common.c*/
    if (unlikely(ZERO_OR_NULL_PTR(cachep)))
        return cachep;
    ret = slab_alloc(cachep, flags, caller);        /*分配函数，见上文*/
```

```
    trace_kmalloc(caller, ret,size, cachep->size, flags);

    return ret;
}
```

通用缓存的分配函数主要是根据对象大小，调用 kmalloc_slab()函数选择 kmalloc_caches[]数组中合适的项（通用缓存），从数组项关联的 kmem_cache 描述符表示的缓存中分配对象，分配对象操作与普通的 slab 分配对象操作相同。

内核还定义了其它的通用缓存分配接口函数，例如：**kzalloc**(size_t size, gfp_t flags)函数分配全零的对象，其函数实现与 kmalloc()类似，源代码请读者自行阅读。

3.7.5 释放对象

前面介绍了 slab 对象的分配，本小节介绍 slab 对象的释放。在分配对象时，始终从本地 CPU 对象缓存中分配，如果本地缓存为空，则先从共享缓存、slab 中获取对象，然后再分配。由此可推断，释放对象时也是释放到 CPU 对象缓存，如果 CPU 对象缓存满了，则先成批将 CPU 对象缓存中对象转移到共享缓存或 slab，然后再将对象释放到 CPU 对象缓存。

1 释放函数

释放 slab 对象的接口函数为 kmem_cache_free()，定义如下（/mm/slab.c）：

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp)
/*cachep：对象所在 slab 缓存描述符，objp：释放对象指针*/
{
    unsigned long flags;
    cachep = cache_from_obj(cachep, objp);      /*查找对象所属 kmem_cache 实例，/mm/slab.h*/
    if (!cachep)
        return;

    local_irq_save(flags);
    debug_check_no_locks_freed(objp, cachep->object_size);
    if (!(cachep->flags & SLAB_DEBUG_OBJECTS))
        debug_check_no_obj_freed(objp, cachep->object_size);
    __cache_free(cachep, objp, _RET_IP_);     /*释放对象，/mm/slab.c*/
    local_irq_restore(flags);

    trace_kmem_cache_free(_RET_IP_, objp);
}
```

kmem_cache_free()函数调用 cache_from_obj()函数获取 slab 缓存 kmem_cache 实例指针，然后调用函数__cache_free()函数执行释放对象操作。

cache_from_obj(cachep, objp)函数定义在/mm/slab.h 头文件，函数根据对象虚拟地址获得所在 slab 首页对应的 page 实例，其 page->slab_cache 成员指向 slab 缓存 kmem_cache 实例，源代码请读者自行阅读。

__cache_free(cachep, objp, _RET_IP_)定义在/mm/slab.c 文件内，用于释放对象：

```
static inline void __cache_free(struct kmem_cache *cachep, void *objp,unsigned long caller)
{
```

```
    struct array_cache *ac = cpu_cache_get(cachep);

    check_irq_off();
    kmemleak_free_recursive(objp, cachep->flags);
    objp = cache_free_debugcheck(cachep, objp, caller);

    kmemcheck_slab_free(cachep, objp, cachep->object_size);

    if (nr_online_nodes > 1 && cache_free_alien(cachep, objp))
        return;

    if (ac->avail < ac->limit) {      /*本地 CPU 对象缓存中对象数量小于阀值*/
        STATS_INC_FREEHIT(cachep);    /*增加统计量值*/
    } else {                          /*本地 CPU 对象缓存中对象数大于等于阀值*/
        STATS_INC_FREEMISS(cachep);       /*增加统计量值*/
        cache_flusharray(cachep, ac);     /*释放缓存对象到共享缓存或 slab 中，/mm/slab.c*/
    }

    ac_put_obj(cachep, ac, objp);         /*将对象释放到本地 CPU 对象缓存*/
}
```

__cache_free()函数判断本地 CPU 对象缓存中对象数量是否小于 limit（阀值），是则直接将对象关联到本地 CPU 对象缓存（ac->entry[avail]），并将 avail 值加 1 即可，这由 ac_put_obj()函数完成。如果对象缓存中对象数量不小于 limit，则先将 batchcount 个缓存对象释放到共享缓存或 slab 中，然后再将要释放的对象关联到本地 CPU 对象缓存。

cache_flusharray()函数用于将本地 CPU 对象缓存中对象转移到共享缓存或放回 slab 中（在 slab 中标记对象空闲），下面将介绍此函数的实现。

2 刷新缓存

在释放对象函数中，当本地 CPU 对象缓存中对象数量不小于 limit 时，则将 batchcount 个对象转移到共享缓存或放回 slab 中，此工作由 cache_flusharray(cachep, ac)函数完成，称它为刷新缓存，函数定义如下（/mm/slab.c）：

```
static void cache_flusharray(struct kmem_cache *cachep, struct array_cache *ac)
{
    int batchcount;
    struct kmem_cache_node *n;
    int node = numa_mem_id();
    LIST_HEAD(list);        /*临时双链表，缓存全部空闲的 slab*/

    batchcount = ac->batchcount;          /*释放对象数量*/
#if DEBUG
    BUG_ON(!batchcount || batchcount > ac->avail);
#endif
    check_irq_off();
```

```
        n = get_node(cachep, node);
        spin_lock(&n->list_lock);
        if (n->shared) {                /*存在共享缓存且缓存未满，则填满共享缓存*/
            struct array_cache *shared_array = n->shared;
            int max = shared_array->limit - shared_array->avail;   /*共享缓存中可容纳对象数量*/
            if (max) {
                if (batchcount > max)
                    batchcount = max;
                memcpy(&(shared_array->entry[shared_array->avail]),ac->entry,\
                                                    sizeof(void *) * batchcount);   /*复制对象指针*/
                shared_array->avail += batchcount;          /*增加共享缓存中对象数量*/
                goto free_done;                 /*释放对象完成*/
            }
        }
        /*没有共享缓存或共享缓存已满，则调用以下函数，将缓存对象放回 slab 中*/
        free_block(cachep, ac->entry, batchcount, node, &list);          /*/mm/slab.c*/
    free_done:
      #if STATS
        ...
      #endif
        spin_unlock(&n->list_lock);
        slabs_destroy(cachep, &list);   /*释放全部空闲的 slab，释放回伙伴系统，/mm/slab.c*/
        ac->avail -= batchcount;        /*减小本地 CPU 缓存对象数量*/
        memmove(ac->entry, &(ac->entry[batchcount]), sizeof(void *)*ac->avail);   /*后面对象往前移*/
    }
```

cache_flusharray()函数判断 slab 缓存是否存在共享缓存且共享缓存是否已满，如果存在且共享缓存没有满，则将本地 CPU 缓存中前 min(batchcount,max)个对象转移至共享缓存，本地缓存后面的对象需要往前移。

如果没有共享缓存或共享缓存已满，则调用 free_block()函数将本地 CPU 对象缓存中前 batchcount 个对象释放回 slab（后面对象前移），即在 slab 中标记对象空闲，并据此改变 slab 所在链表或释放 slab。如果 slab 缓存中空闲对象数量超过阀值（kmem_cache_node->free_limit），则将全部对象空闲的 slab 移至 list 链表等待释放。

slabs_destroy()函数用于释放 list 临链表中的 slab，即将 slab 所占内存释放回伙伴系统，源代码请读者自行阅读。

下面看一下 free_block()函数的定义，如下所示（/mm/slab.c）：

```
static void free_block(struct kmem_cache *cachep, void **objpp,int nr_objects, int node,   \
                                                                    struct list_head *list)
/*list：缓存需要释放的 slab*/
{
    int i;
    struct kmem_cache_node *n = get_node(cachep, node);
```

```
    for (i = 0; i < nr_objects; i++) {      /*遍历本地 CPU 对象缓存，数量为 batchcount*/
        void *objp;
        struct page *page;

        clear_obj_pfmemalloc(&objpp[i]);      /*清零地址最低位，/mm/slab.c*/
        objp = objpp[i];      /*对象虚拟地址*/

        page = virt_to_head_page(objp);      /*对象所在 slab 首页 page 实例*/
        list_del(&page->lru);
        check_spinlock_acquired_node(cachep, node);
        slab_put_obj(cachep, page, objp, node);
                                /*释放一个对象，在 freelist 中写入对象编号等，/mm/slab.c*/
        STATS_DEC_ACTIVE(cachep);
        n->free_objects++;      /*空闲对象数量加 1*/

        /*更改对象所在 slab 所处链表*/
        if (page->active == 0) {          /*slab 中对象全部空闲*/
            if (n->free_objects > n->free_limit) {      /*slab 缓存中空闲对象超过阀值*/
                n->free_objects -= cachep->num;
                list_add_tail(&page->lru, list);     /*将 slab 移入 list 链表等待释放*/
            } else {
                list_add(&page->lru, &n->slabs_free);     /*slab 加入全部空闲 slab 链表*/
            }
        } else {                          /*slab 中部分对象空闲*/
            list_add_tail(&page->lru, &n->slabs_partial);      /*slab 加入部分空闲链表*/
        }
    }
}
```

free_block()函数扫描本地 CPU 对象缓存（数量为 batchcount），对每个对象获取其所在 slab 首页 page 实例，调用 **slab_put_obj()**函数释放对象后，检查对象所在 slab 是全部对象空闲还是部分对象空闲，并将 slab 移动到相应的链表中。如果空闲对象太多超过阀值，则需要注销全部对象空闲的 slab，即将其移动至 list 临时双链表，在后面调用的 slabs_destroy()函数中释放。

slab_put_obj()函数用于释放一个对象，函数定义如下（/mm/slab.c）：

```
static void slab_put_obj(struct kmem_cache *cachep, struct page *page,void *objp, int nodeid)
{
    unsigned int objnr = obj_to_index(cachep, page, objp);     /*对象指针转化成对象编号，/mm/slab.c*/
  #if DEBUG
    ...
  #endif
    page->active--;        /*page->active 前移一项*/
    set_free_obj(page, page->active, objnr);     /*将释放对象编号写入 freelist[page->active]数组项*/
}
```

set_free_obj()函数将释放对象的编号写入 freelist[page->active]数组项，如下所示：

```
static inline void set_free_obj(struct page *page,unsigned int idx, freelist_idx_t val)
{
    ((freelist_idx_t *)(page->freelist))[idx] = val;
}
```

释放 slab 对象的函数到此介绍完毕。另外，通用缓存的释放对象函数为 kfree(const void *objp)，函数内通过对象 objp 虚拟地址最终获取其所在 slab 的首页 page 实例，由 page->slab_cache 成员获取通用缓存 kmem_cache 实例，然后调用__cache_free()函数释放对象，源代码请读者自行阅读。

3.7.6 内存池

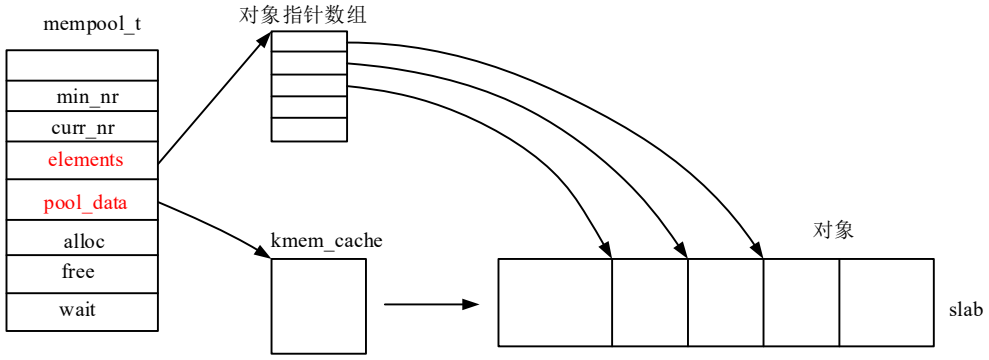
slab 分配器管理着小块的内存，或者说数据结构实例，内核源代码可以从中分配和释放对象。另外，内核还实现了管理内存块（数据结构实例）更一般（更抽象）的方式，即内存池。内存池与 slab 分配器一样也是管理着一批缓存对象，但是其缓存的对象不仅可以来自 slab 分配器，还可以来自伙伴系统或其它地方。

每个内存池中包含一个缓存对象指针数组，定义了分配和释放对象的函数指针。一个内存池中的对象可以来自 slab 缓存，或伙伴系统，或其它地方。

内存池数据结构定义在/include/linux/mempool.h：

```
typedef struct mempool_s {
    spinlock_t lock;       /*保护自旋锁*/
    int min_nr;            /*最小对象数*/
    int curr_nr;           /*当前对象指针*/
    void **elements;       /*对象指针数组*/

    void *pool_data;       /*若对象来自 slab 缓存，则指向 kmem_cache 实例*/
    mempool_alloc_t *alloc;     /*分配对象函数*/
    mempool_free_t *free;       /*释放对象函数*/
    wait_queue_head_t wait;     /*等待队列头*/
} mempool_t;

typedef void * (mempool_alloc_t)(gfp_t gfp_mask, void *pool_data);   /*从内存池分配对象的函数原型*/
typedef void (mempool_free_t)(void *element, void *pool_data);       /*从内存池释放对象的函数原型*/
```

内存池数据结构如下图所示（对象来自 slab 缓存）：



elements 成员指向的是缓存对象指针数组，对象实例来源于 slab 缓存、或伙伴系统等。若是从 slab 缓存中分配对象，则 pool_data 成员指向 slab 缓存 kmem_cache 实例。alloc()和 free()分别表示分配和释放对象的函数指针，从 slab 缓存或伙伴系统获取对象时，分配释放函数调用 slab 分配器或伙伴系统的分配释放函数。

内存池主要接口函数如下（/mm/mempool.c、/include/linux/mempool.h）：

●mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn,mempool_free_t *free_fn, void ***pool_data**)：创建内存池，手工赋予分配和释放对象函数。

●mempool_t ***mempool_create_slab_pool**(int min_nr, struct kmem_cache *kc)：创建内存池，从指定 slab 缓存分配对象，min_nr 表示最小对象数。分配释放函数默认设为 slab 分配器的分配释放对象函数。

●void * mempool_alloc(mempool_t *pool, gfp_t gfp_mask)：从内存池分配对象函数；

●void mempool_free(void *element, mempool_t *pool)：释放对象函数。

3.8 percpu 变量

在内核代码中我们经常能看到由**__percpu**（/include/linux/compiler.h）修饰的指针变量，表示指针指向的是一个 percpu 变量。percpu 变量本质上与一般的变量是一样的，只不过一般的变量只有一个实例，而 percpu 变量对每个 CPU 核都定义一个实例（副本），每个 CPU 核只访问属于自己的变量实例，以避免 CPU 核之间对变量的竞态访问。但是，如果启用了内核抢占（PREEMPT），在同一 CPU 核运行的进程依然可能产生对同一 percpu 变量的竞态访问，此时访问变量时需要关闭内核抢占，访问结束后打开内核抢占。percpu 变量适用于 SMP 处理器，如果是单核处理器，percpu 变量就是普通变量。

内核中分配了专门的内存用于保存 percpu 变量实例，percpu 变量实例按 CPU 核划分，属于同一 CPU 核的 percpu 变量放在一起。内核定义或分配一个 percpu 变量实例后，返回的是一个基准地址（指针），不能通过这个基准地址直接访问 percpu 变量，而应该通过专用的函数依据基准地址获取特定于 CPU 核的变量实例地址，以此地址访问变量实例。

访问 percpu 变量的专用函数在变量基准地址的基础上，依当前运行 CPU 核，累加上一个固定的偏移量即得到实际变量实例的地址。

本节先介绍 percpu 变量实现的原理，然后介绍静态定义 percpu 变量的处理，以及动态分配/释放 percpu 变量的实现，percpu 变量相关代码位于/mm/percpu.c 文件内。

3.8.1 概述

本小节概述 percpu 变量的管理，访问 percpu 变量的接口函数，定义、分配/释放 percpu 变量的接口函数等。

1 percpu 变量管理

percpu 变量可以静态定义，也可以动态分配，我们先看静态定义的 percpu 变量。

内核代码中可以利用 **DEFINE_PER_CPU(type, name)**宏等静态定义 type 数据类型的 percpu 变量，宏定义在/include/linux/percpu-defs.h 头文件：

```
#define DEFINE_PER_CPU(type, name)                    \    /*type：数据类型，name：变量名称*/
    DEFINE_PER_CPU_SECTION(type, name, "")

#define DEFINE_PER_CPU_SECTION(type, name, sec)              \
    __PCPU_ATTRS(sec)   PER_CPU_DEF_ATTRIBUTES       \
    __typeof__(type) name                                    /*定义变量，标记链接段*/
```

```
#define __PCPU_ATTRS(sec)                                       \
    __percpu __attribute__((section(PER_CPU_BASE_SECTION sec)))     \
    PER_CPU_ATTRIBUTES
```

__PCPU_ATTRS(sec)宏指示变量 name 链接段的名称，PER_CPU_BASE_SECTION 宏定义在头文件/include/asm-generic/percpu.h：

```
#define PER_CPU_BASE_SECTION ".data..percpu"        /*段名前缀*/
```

__PCPU_ATTRS(sec)宏表示将变量链接到段名为.data..percpu*sec* 的段中（后接 sec 表示的字符串）。

DEFINE_PER_CPU(type, name)宏的实际效果是定义了一个名称为 name 的 type 类型变量实例，并将其链接到.data..percpu*sec* 段中，也就是说内核静态定义的 percpu 变量会链接到同一个段中，放在一起。

内核在/include/asm-generic/vmlinux.lds.h 头文件内定义了 percpu 变量链接到的段（嵌入到链接文件）：

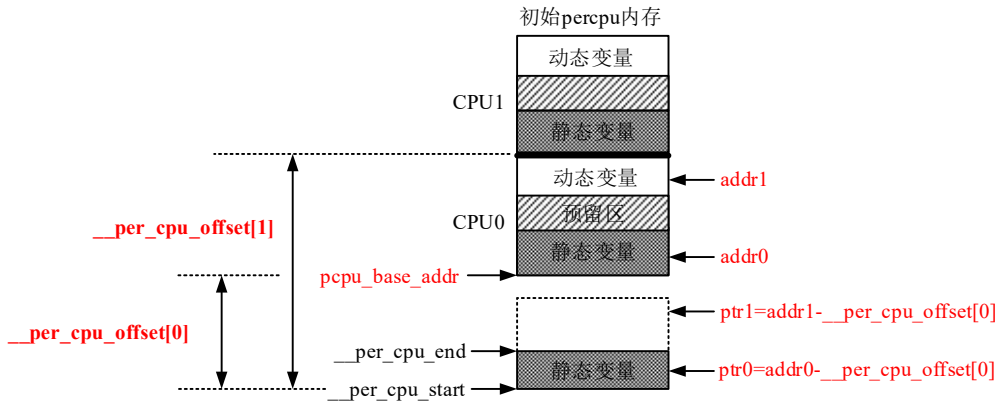
```
#define PERCPU_INPUT(cacheline)                                 \
    VMLINUX_SYMBOL(__per_cpu_start) = .;                    \
    *(.data..percpu..first)                 \   /*DEFINE_PER_CPU_FIRST()*/
    . = ALIGN(PAGE_SIZE);                   \
    *(.data..percpu..page_aligned)          \
    . = ALIGN(cacheline);                   \
    *(.data..percpu..read_mostly)           \
    . = ALIGN(cacheline);                   \
    *(.data..percpu)                        \   /*DEFINE_PER_CPU()*/
    *(.data..percpu..shared_aligned)        \
    VMLINUX_SYMBOL(__per_cpu_end) = .;
```

全局变量__per_cpu_start 和__per_cpu_end 由链接器导出，表示段的起始和结束地址，段内又分成多个小段（由 sec 参数指示），分别表示由不同名称的变量实例。

在/include/linux/percpu-defs.h 头文件中还定义了其它定义 percpu 变量的宏，主要区别是 sec 参数不同，请读者自行阅读源代码。

下图下半部分示意了保存静态定义 percpu 变量的内存段。在内核初始化阶段，将为 percpu 变量分配初始的内存区，这里称它为初始 percpu 内存，专门用于保存 percpu 变量实例。percpu 内存按 CPU 核进行划分，每个 CPU 核对应区域的布局是一样的，因为每个 CPU 核的 percpu 变量实例是一样的，只不过变量值不同。内核初始化阶段分配初始 percpu 内存后，会将静态定义的 percpu 变量复制到各个 CPU 核对应的内存区域中。

全局变量 pcpu_base_addr 记录了初始 percpu 内存的起始地址，全局数组__per_cpu_offset[NR_CPUS]记录了初始 percpu 内存中各 CPU 核内存区域相对于保存静态定义 percpu 变量内存段的偏移量，如下图所示。例如，__per_cpu_offset[0]=pcpu_base_addr - __per_cpu_start。

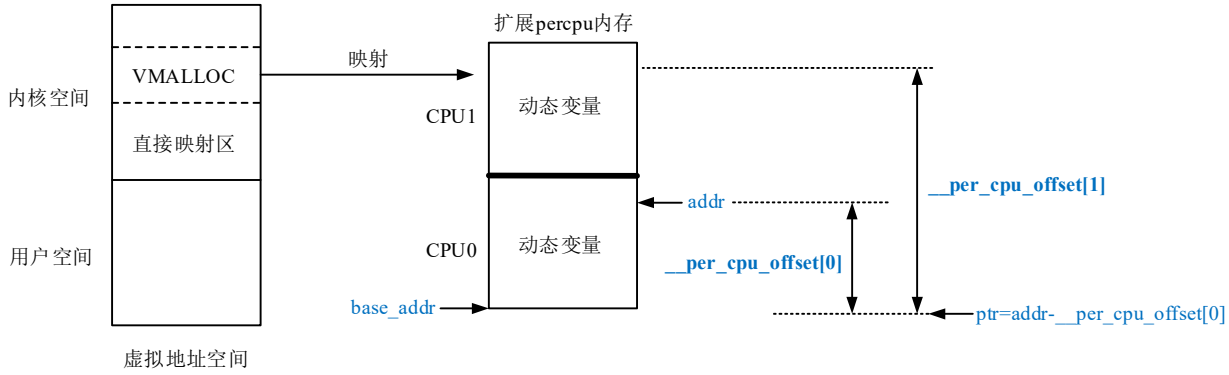


内核静态定义 percpu 变量时，获得的是在.data..percpu*sec* 链接段中的实例地址，这里用 ptr 表示，各 CPU 核要访问属于自己的变量副本时，需要在此基础上加上一个偏移量，即__per_cpu_offset[cpu]。例如，CPU0 要访问属于自己的变量实例时，其实例地址为 ptr+__per_cpu_offset[0]。

percpu 变量还可以动态创建。在上图中的初始 percpu 内存中，除保存静态定义变量的区域和预留区域外，还有动态变量区，可从中动态分配 percpu 变量实例。由于各 CPU 核 percpu 变量区域布局是一样的，因此内核只管理 CPU0 核（也可能是其它 CPU 核）对应的 percpu 变量区。

假设在 CPU0 内存区域中动态分配的 percpu 变量地址为 addr1，为了统一静态定义和动态分配 percpu 变量的访问（使用相同的接口函数），动态分配 percpu 变量的基准地址设为 addr1-__per_cpu_offset[0]。当访问 percpu 变量时会加上偏移量__per_cpu_offset[0]，正好是变量实例的实际地址。

内核在启动阶段分配的初始 percpu 内存是固定的，随着 percpu 变量的分配，初始 percpu 内存可能不够用，这时就需要创建新的 percpu 内存，称它为扩展 percpu 内存，如下图所示。新建的 percpu 内存映射到内核地址空间 VMALLOC 区，而初始 percpu 内存位于内核地址空间直接映射区。



扩展 percpu 内存映射到内核空间的间接映射区，后面第 4 章将会介绍相关内容。上图中 base_addr 表示扩展 percpu 内存的基地址。在扩展 percpu 内存中就没有静态定义的变量了，只有动态分配的变量。假设动态分配变量的实际地址为 addr，则动态分配变量的基准地址设为 addr-__per_cpu_offset[0]，这是分配函数的返回地址。在访问变量时，会累加上相应的偏移量，正好获得变量实例的实际地址。

2 接口函数

内核提供的静态和动态创建 percpu 变量的接口函数如下：

（1）静态定义 percpu 变量，由 **DEFINE_PER_CPU()**等宏实现，返回变量基准地址。

（2）动态创建 percpu 变量，由 **alloc_percpu()/__alloc_percpu()**等函数实现，返回变量基准地址。

不管是静态定义的还是动态创建的 percpu 变量，都采用相同的接口函数对其进行访问，即获取当前 CPU 核对应变量实例的基地址（指针）。

内核通过接口函数 smp_processor_id()判断当前程序在哪个 CPU 核上运行，函数返回 CPU 核编号（ID），单核处理器总是返回 0。

per_cpu_ptr(ptr, cpu)宏用于获取 cpu（编号）核对应基准地址为 ptr 的 percpu 变量实例：

```
#define per_cpu_ptr(ptr, cpu)                  \      /*ptr 为地址值，/include/linux/percpu-defs.h*/
({                                             \
    __verify_pcpu_ptr(ptr);                    \      /*指针有效性判断*/
    SHIFT_PERCPU_PTR((ptr), per_cpu_offset((cpu))); \
                                                    /* per_cpu_offset(x) =__per_cpu_offset[x]*/
})

#define SHIFT_PERCPU_PTR(__p, __offset)                                 \
    RELOC_HIDE((typeof(*(__p)) __kernel __force *)(__p), (__offset))    /*__p+__offset*/
```

RELOC_HIDE()宏计算__p 和__offset 参数之和并转为__p 指向数据类型的指针。

由前面的分析可知，percpu 变量基准地址加上__per_cpu_offset[cpu]正好是 cpu 核对应变量实例的地址。

内核在/include/linux/percpu-defs.h 头文件还定义了其它获取 percpu 变量实例指针的接口函数，例如：

- **per_cpu(var, cpu)**：获取变量 var 在 cpu 编号的处理器核上副本的地址，var 为变量名称。
- **get_cpu_var(var)**：获取变量 var 在当前 CPU 核上的副本地址，函数内将关闭内核抢占，访问变量结束后要调用 put_cpu_var(var)函数使能内核抢占，var 为变量名称。

将内核静态定义的 percpu 变量导出的宏定义如下（/include/linux/percpu-defs.h）：

```
EXPORT_PER_CPU_SYMBOL(per_cpu_var);          /*per_cpu_var 为变量名称*/
EXPORT_PER_CPU_SYMBOL_GPL(per_cpu_var);
```

在模块中访问导出的 percpu 变量，应该这样做声明（/include/linux/percpu-defs.h）：

```
DECLARE_PER_CPU(type, name);
```

3 percpu 分配器

前面介绍了 percpu 变量的管理，定义、分配以及访问变量的接口函数。还有一个重要的问题没有介绍，那就是 percpu 变量在 percpu 内存中如何进行分配和释放。

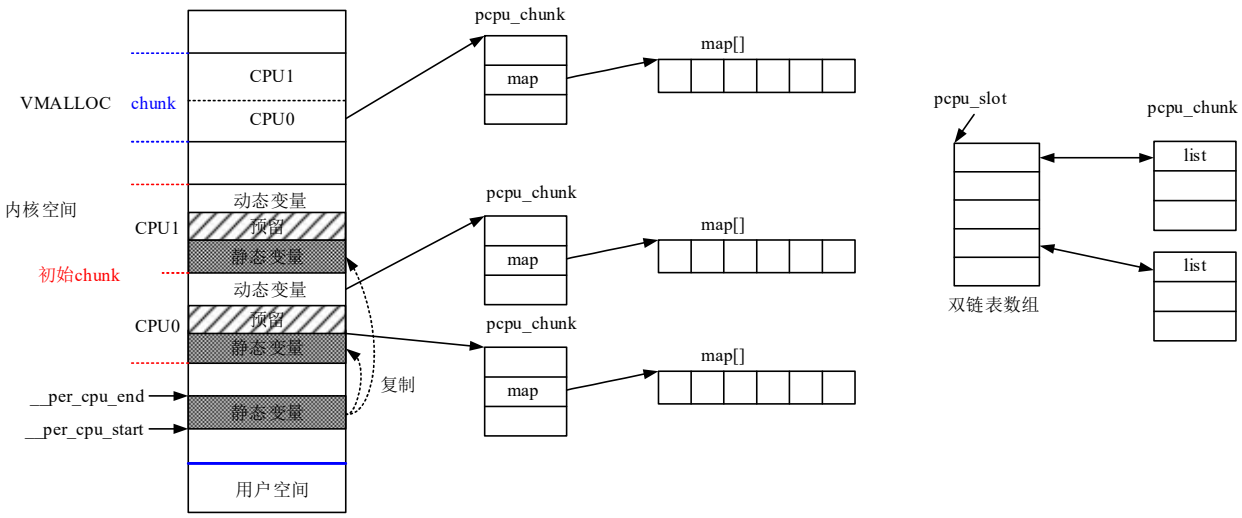
percpu 内存中包含静态变量区、预留区、动态变量区，或只有动态变量区。分配变量即从动态变量区（或预留区）划分小块内存用于保存变量实例，这有点类似于 slab 分配器，都是从大块内存中划分出小块内存，因此这里暂且称之为 percpu 分配器。

percpu 内存中可动态分配变量的区域称之为 chunk，例如，初始 percpu 内存中的静态变量区加预留区是一个 chunk，动态变量区是一个 chunk，后面动态创建的 percpu 内存中没有静态变量区和预留区，因此只含一个 chunk。

内核通过 pcpu_chunk 结构体实例来管理 chunk，如下图所示。由于各 CPU 核对应的 percpu 内存区域

布局是一样的，因此内核只需管理一个 CPU（一般是 CPU0）核对应的内存区域，为其创建 pcpu_chunk 实例即可。

在内核初始化阶段，将为初始 percpu 内存中的静态变量区（含预留区）和动态变量区分别创建 pcpu_chunk 实例。如果初始 percpu 内存使用完了，将在内核虚拟地址空间 VMALLOC 区创建新的 percpu 内存，并为其创建 pcpu_chunk 实例。内核中除静态变量区（含预留区）对应的 pcpu_chunk 实例外，都将按空闲区域大小，动态添加到全局双链表数组中。

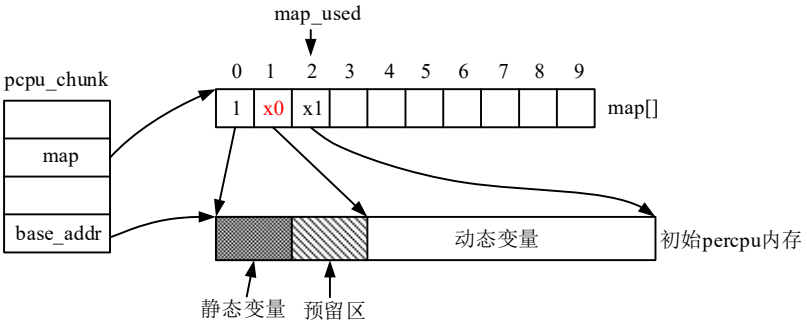


pcpu_chunk 结构体定义在/mm/percpu.c 文件内：

```
struct pcpu_chunk {
    struct list_head    list;       /*双链表成员，将实例插入到 pcpu_slot 散列表*/
    int         free_size;      /*空闲区域总大小，字节数*/
    int         contig_hint;    /*最大可用连续空闲内存块大小*/
    void        *base_addr;     /*percpu 内存基地址*/

    int   map_used;     /*map 数组项数（已用的）*/
    int   map_alloc;    /*map 数组项数，初值为 128，可扩展*/
    int   *map;         /*map 数组，记录区域中空间使用情况，用于分配和释放变量*/
    struct work_struct  map_extend_work;   /*异步的扩展 map[]数组的工作*/

    void        *data;          /*私有数据结构，指向 vm_struct 实例*/
    int         first_free;   /*第一个空闲内存块编号（对应 map[]数组项），其下没有空闲内存块了*/
    bool        immutable;          /* no [de]population allowed */
    int         nr_populated;       /*percpu 内存建立了物理内存映射页的数量*/
    unsigned long       populated[];   /*位图，标记 percpu 内存页是否映射到了物理页面*/
};
```

pcpu_chunk 结构体中最重要的成员就是 map[]整数数组，它用于标记区域中内存空间使用情况，表示哪些区域已分配出去（已使用），哪些区域未被使用可供分配。pcpu_chunk 结构体中最后的位图用于标记 percpu 内存是否映射到了物理内存。在 VMALLOC 区中创建的 percpu 内存时，虚拟内存（地址）并没有立即映射到物理内存（只是划分出虚拟内存空间），在分配变量时需要动态地创建虚拟地址空间到物理内存的映射，位图表示虚拟内存是否映射到了物理内存。

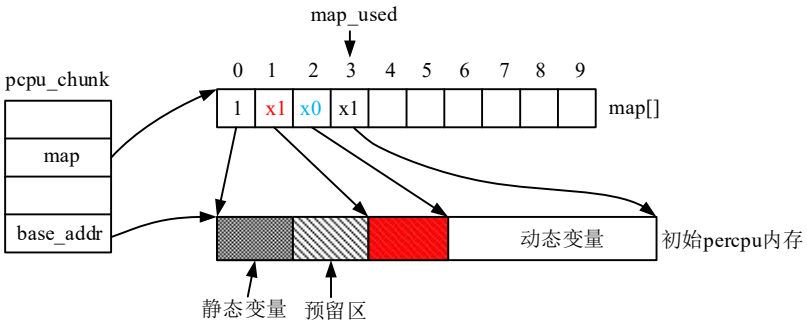
下面看一下如何通过 map[]数组来管理 percpu 内存区域中的空闲内存块，如下图所示：



map[]数组项依次对应 percpu 内存中的内存块，数组项值表示对应内存块相对于 percpu 内存起始地址的偏移量。数组项最低比特位表示对应内存块是否空闲（可用），1 表示已使用，不能再分配，0 表示可以使用，可以分配。

上图中的是初始 percpu 内存中动态变量区对应的 pcpu_chunk 实例。map[0]对应的是静态变量区加预留区，表示不能从这个区域分配变量。map[1]对应动态变量区，map[2]表示结束地址。

分配变量时，即在 map[]数组中查找空闲内存块，划分出需要的大小，增加 map[]数组项（也可能不需要增加项）。下图示意了从动态变量区分配一个变量后的结果。



上图中 map[1]对应内存块表示分配的变量，原 map[2]数组项需要修改数组项值，并将其及之后的数组项后移一项（增加了一个数组项）。

释放变量的操作要简单一些，如果释放变量前后的内存块都已使用，则只需要将对应数组项最低位清零即可，表示内存块可用。如果前后内存块空闲，则可以合并内存块生成更大的空闲内存块，并修改相应的 map[]数组项。

3.8.2 初始 percpu 内存

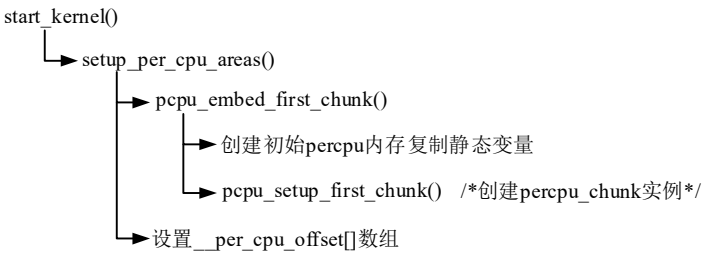
在内核代码中由 DEFINE_PER_CPU()等宏静态定义的 percpu 变量，将链接到内核目标文件的指定段中，段起止地址为__per_cpu_start 和__per_cpu_end。此段中只包含定义变量的一个实例，在内核启动阶段将分配初始的 percpu 内存，并将静态 percpu 变量复制到每个 CPU 核对应的区域。

在初始化阶段分配了初始 percpu 内存，复制静态变量后，还需要为初始 percpu 内存创建 pcpu_chunk 实例，这些工作主要由 **setup_per_cpu_areas()**函数完成，本小节主要介绍此函数的实现。

1 早期初始化函数

setup_per_cpu_areas()函数也可视为 percpu 变量的初始化函数，主要是创建初始 percpu 内存，复制静态变量，以及创建相关的数据结构实例，此后就可以动态创建 percpu 变量了。

setup_per_cpu_areas()函数调用关系简列如下图所示：



内核在启动函数 start_kernel()中调用 **setup_per_cpu_areas()**函数（伙伴系统和 slab 分配器尚不可用），函数内调用 pcpu_embed_first_chunk()函数创建初始 percpu 内存、复制静态变量、创建 pcpu_chunk 实例，最后依据创建的初始 percpu 内存设置__per_cpu_offset[]数组。

setup_per_cpu_areas()函数定义如下（选择 SMP 配置选项，/mm/percpu.c）：

```
void __init setup_per_cpu_areas(void)
{
    unsigned long delta;
    unsigned int cpu;
    int rc;

    /*分配初始 percpu 内存，复制静态定义变量并创建 pcpu_chunk 实例*/
    rc = pcpu_embed_first_chunk(PERCPU_MODULE_RESERVE,
                PERCPU_DYNAMIC_RESERVE, PAGE_SIZE, NULL,
                pcpu_dfl_fc_alloc, pcpu_dfl_fc_free);     /*/mm/percpu.c*/
    if (rc < 0)
        panic("Failed to initialize percpu areas.");

    delta = (unsigned long)pcpu_base_addr - (unsigned long)__per_cpu_start;      /*基地址偏移量*/
    for_each_possible_cpu(cpu)
        __per_cpu_offset[cpu] = delta + pcpu_unit_offsets[cpu];
                    /*设置各 CPU 核 percpu 内存区域基地址相对于__per_cpu_start 的偏移量*/
}
```

setup_per_cpu_areas()函数调用 pcpu_embed_first_chunk()函数创建初始 percpu 内存，复制静态定义变量至每个 CPU 核的 percpu 内存，并为 CPU0 核 percpu 内存区域创建并设置 pcpu_chunk 实例，用于动态分配/释放 percpu 变量。

pcpu_embed_first_chunk()函数中还将设置全局变量 pcpu_base_addr 和 pcpu_unit_offsets[]数组，分别表示分配的 percpu 内存基地址和各 CPU 核 percpu 内存区域相对于基地址 pcpu_base_addr 的偏移量。

delta 变量表示创建的初始 percpu 内存基地址相对于静态 percpu 变量段起始地址__per_cpu_start 的偏移量，全局数组**__per_cpu_offset[]**表示每个 CPU 核 percpu 内存区域基地址相对于__per_cpu_start 的偏移量，用于 CPU 核访问自己的变量实例。

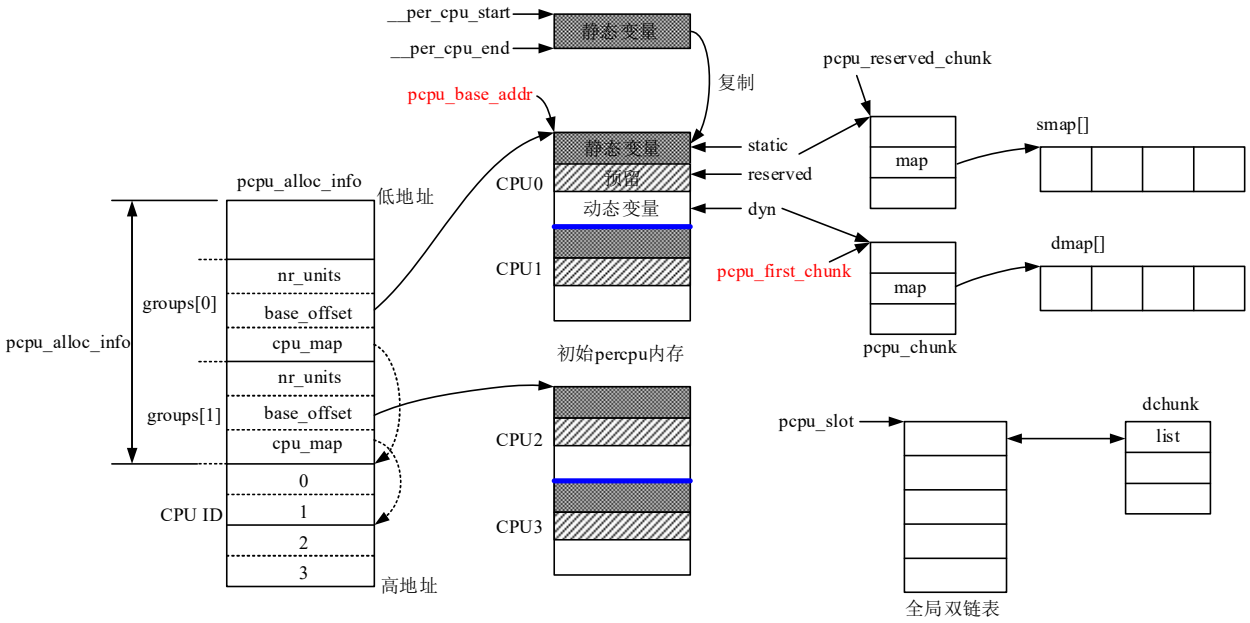
下面将介绍创建初始 percpu 内存的 pcpu_embed_first_chunk()函数的实现。

2 创建初始 percpu 内存

到目前为至，我们都假设各 CPU 核对应的变量区域都在同一个连续的物理内存中，其实它们也可以不在同一块连续的内存中。

内核可以对 CPU 核进行分组，同组 CPU 核的变量区域在同一个内存块中，不同组 CPU 核的变量区域可以在不同的内存块中。如下图所示，假设处理器具有 4 个处理器核，CPU0 与 CPU1 为一组，CPU2 与

CPU3 为一组。在创建 percpu 内存时，按 CPU 组创建，即同一组的 CPU 核 percpu 内存在同一连续内存中。每个 CPU 核对应的初始 percpu 内存区域中将包含三个区域，分别是保存静态 percpu 变量的区域、预留区域和用于动态分配 percpu 变量的区域。



初始 percpu 内存创建，复制完静态变量后，需要为 CPU0 对应的区域创建 pcpu_chunk 实例，其中静态变量区域和预留区域共用一个 pcpu_chunk 实例，动态变量区域对应一个 pcpu_chunk 实例。

内核中的所有 pcpu_chunk 实例将由全局双链表管理，这其实是一个双链表数组，pcpu_chunk 实例按所代表区域中空闲内存的大小插入到相应的双链表中，以便在分配 percpu 变量时查找适合的 pcpu_chunk 实例。

创建初始 percpu 内存的工作由 pcpu_embed_first_chunk()函数完成，函数内将定义/设置 pcpu_alloc_info 结构体实例，用于传递创建初始 percpu 内存的信息。下面我们先看一下 pcpu_alloc_info 结构体的定义，然后再介绍 pcpu_embed_first_chunk()函数的实现。

pcpu_alloc_info 结构体定义如下（/include/linux/percpu.h）：

```
struct pcpu_alloc_info {
    size_t          static_size;      /*静态变量区大小*/
    size_t          reserved_size;    /*预留区大小，用于模块中静态定义的 percpu 变量*/
    size_t          dyn_size;         /*动态变量区大小*/
    size_t          unit_size;        /*处理器核对应 percpu 内存大小*/
    size_t          atom_size;        /*分配 percpu 内存的最小单位，PAGE_SIZE，按页分配*/
    size_t          alloc_size;       /*CPU 组对应的 percpu 内存大小*/
    size_t          __ai_size;        /*pcpu_alloc_info 结构体大小，内部成员*/
    int             nr_groups;        /*处理器核分组数量，通常为 1*/
    struct pcpu_group_info  groups[];    /*CPU 核分组信息，若只有一个分组则只有一个实例*/
};
```

CPU 核分组信息 pcpu_group_info 结构体定义如下（/include/linux/percpu.h）：

```
struct pcpu_group_info {
    int         nr_units;       /*组中 CPU 核数量*/
    unsigned long       base_offset;   /*组 percpu 内存相对于基址（最低 percpu 内存地址）的偏移量*/
```

```
    unsigned int        *cpu_map;    /*指向整数组，表示组内 CPU 编号，空项写入 NR_CPUS */
};
```

下面来看一下 **pcpu_embed_first_chunk()**函数的实现，代码如下（/mm/percpu.c）：

```
int __init pcpu_embed_first_chunk(size_t reserved_size, size_t dyn_size,size_t atom_size,
                pcpu_fc_cpu_distance_fn_t cpu_distance_fn,pcpu_fc_alloc_fn_t  alloc_fn,
                pcpu_fc_free_fn_t  free_fn)
/*
*reserved_size：预留区大小，PERCPU_MODULE_RESERVE 值大约为 8KB，
*dyn_size：动态变量区大小，PERCPU_DYNAMIC_RESERVE 值大约为 20KB，
*atom_size：分配 percpu 内存的单位，PAGE_SIZE 表示按页分配，
*cpu_distance_fn：计算 CPU 核间距离函数，用于 CPU 分组，这里为 NULL，
*alloc_fn：分配内存的函数，这里为 pcpu_dfl_fc_alloc()，表示从自举分配器分配内存，
*free_fn：释放内存的函数，这里为 pcpu_dfl_fc_free()，表示内存释放到自举分配器。
*/
{
    void *base = (void *)ULONG_MAX;
    void **areas = NULL;          /*areas 指向指针数组保存各 CPU 组 percpu 内存基地址*/
    struct pcpu_alloc_info *ai;      /*指向 pcpu_alloc_info 实例*/
    size_t size_sum, areas_size, max_distance;
    int group, i, rc;

    /*创建并初始化 pcpu_alloc_info 实例，源代码请读者自行阅读*/
    ai = pcpu_build_alloc_info(reserved_size, dyn_size, atom_size,cpu_distance_fn);   /*/mm/percpu.c*/
    ...
    size_sum = ai->static_size + ai->reserved_size + ai->dyn_size;
                                                    /*每个 CPU 核对应 percpu 内存大小*/
    areas_size = PFN_ALIGN(ai->nr_groups * sizeof(void *));    /*areas 指针数组大小*/

    areas = memblock_virt_alloc_nopanic(areas_size, 0);    /*为 areas 指针数组分配空间*/
    ...
    /*遍历每个 CPU 组，为每个组分配 percpu 内存*/
    for (group = 0; group < ai->nr_groups; group++) {
        struct pcpu_group_info *gi = &ai->groups[group];
        unsigned int cpu = NR_CPUS;
        void *ptr;

        for (i = 0; i < gi->nr_units && cpu == NR_CPUS; i++)    /*查找组中最大 CPU 编号*/
            cpu = gi->cpu_map[i];
        BUG_ON(cpu == NR_CPUS);

        ptr = alloc_fn(cpu, gi->nr_units * ai->unit_size, atom_size);   /*为 CPU 组分配 percpu 内存*/
        ...
```

```
            kmemleak_free(ptr);
            areas[group] = ptr;       /*记录 CPU 核组 percpu 内存基地址*/

            base = min(ptr, base);     /*CPU 核组 percpu 内存最低地址*/
        }

        /*遍历每个组，复制静态变量至各 CPU 核 percpu 内存*/
        for (group = 0; group < ai->nr_groups; group++) {
            struct pcpu_group_info *gi = &ai->groups[group];
            void *ptr = areas[group];          /*CPU 核组 percpu 内存基地址*/

            for (i = 0; i < gi->nr_units; i++, ptr += ai->unit_size) {     /*遍历组内 CPU 核 percpu 内存*/
                if (gi->cpu_map[i] == NR_CPUS) {
                    /*释放未使用的 cpu 核对应的内存*/
                    free_fn(ptr, ai->unit_size);
                    continue;
                }
                memcpy(ptr, __per_cpu_load, ai->static_size);   /*复制静态变量至 CPU 核 percpu 内存*/
                free_fn(ptr + size_sum, ai->unit_size - size_sum);     /*释放未使用内存*/
            }
        }

        max_distance = 0;
        for (group = 0; group < ai->nr_groups; group++) {
            ai->groups[group].base_offset = areas[group] - base;
                                                    /*各组 percpu 内存相对于基地址 base 的偏移量*/
            max_distance = max_t(size_t, max_distance,ai->groups[group].base_offset);
        }
        max_distance += ai->unit_size;   /*所有组 percpu 内存中最高地址与最低地址距离*/

        /*max_distance 超过 vmalloc 空间 75%报错*/
        if (max_distance > VMALLOC_TOTAL * 3 / 4) {
            ...
        }
        ...
        rc = pcpu_setup_first_chunk(ai, base);     /*创建并设置 pcpu_chunk 实例，/mm/percpu.c*/
        goto out_free;
            ...
    out_free:
        pcpu_free_alloc_info(ai);     /*释放 pcpu_alloc_info 实例*/
        if (areas)
            memblock_free_early(__pa(areas), areas_size);        /*释放指针数组*/
        return rc;        /*成功返回 0，否则返回错误码*/
```

}

pcpu_embed_first_chunk()函数首先调用 pcpu_build_alloc_info()函数创建并设置 pcpu_alloc_info 实例，然后依据 pcpu_alloc_info 实例为各 CPU 核组分配 percpu 内存，复制静态 percpu 变量至各 CPU 核对应的 percpu 内存，随后调用 **pcpu_setup_first_chunk(ai, base)**函数为初始 percpu 内存创建并设置 pcpu_chunk 实例，最后释放 pcpu_alloc_info 实例及 areas 指针数组。

■创建 pcpu_chunk 实例

在分配完 percpu 内存，复制静态定义的变量后，将调用 **pcpu_setup_first_chunk()**函数为初始 percpu 内存创建并设置 pcpu_chunk 实例，以便从中动态分配 percpu 变量。

pcpu_setup_first_chunk()函数根据参数传递的 pcpu_alloc_info 实例和 percpu 内存基地址创建并设置 pcpu_chunk 实例，函数定义如下（/mm/percpu.c）。

```
int __init pcpu_setup_first_chunk(const struct pcpu_alloc_info *ai,void *base_addr)
/*ai：pcpu_alloc_info 实例指针，base_addr：percpu 内存基地址（最低地址）*/
{
    static int smap[PERCPU_DYNAMIC_EARLY_SLOTS] __initdata;     /*数组项数为 128*/
    static int dmap[PERCPU_DYNAMIC_EARLY_SLOTS] __initdata;     /*map 数组*/
    size_t dyn_size = ai->dyn_size;         /*动态变量区大小*/
    size_t size_sum = ai->static_size + ai->reserved_size + dyn_size;
                                                    /*每个 CPU 核对应 percpu 内存大小*/
    struct pcpu_chunk *schunk, *dchunk = NULL;
                        /*静态（预留）和动态变量 percpu 内存区对应的 pcpu_chunk 实例指针*/
    unsigned long *group_offsets;
    size_t *group_sizes;
    unsigned long *unit_off;
    unsigned int cpu;
    int *unit_map;
    int group, unit, i;
    ...         /*安全性检查*/
    group_offsets = memblock_virt_alloc(ai->nr_groups *sizeof(group_offsets[0]), 0);
                                    /*保存各 CPU 组 percpu 内存基址相对于 base_addr 的偏移量数组*/
    group_sizes = memblock_virt_alloc(ai->nr_groups *sizeof(group_sizes[0]), 0);
                                                    /*保存各 CPU 组 percpu 内存长度数组*/
    unit_map = memblock_virt_alloc(nr_cpu_ids * sizeof(unit_map[0]), 0);
                                                    /*保存 CPU 核编号数组*/
    unit_off = memblock_virt_alloc(nr_cpu_ids * sizeof(unit_off[0]), 0);
                                    /*保存各 CPU 核 percpu 内存基址相对于 base_addr 的偏移量*/
    for (cpu = 0; cpu < nr_cpu_ids; cpu++)
        unit_map[cpu] = UINT_MAX;

    pcpu_low_unit_cpu = NR_CPUS;     /*最小和最大 CPU 核编号*/
    pcpu_high_unit_cpu = NR_CPUS;

    /*遍历各组中的 CPU 核，获取各 CPU 核 percpu 内存区域相对于 base_addr 的偏移量*/
```

```
for (group = 0, unit = 0; group < ai->nr_groups; group++, unit += i) {
        const struct pcpu_group_info *gi = &ai->groups[group];

        group_offsets[group] = gi->base_offset;     /*CPU 组 percpu 内存相对于基地址的偏移量*/
        group_sizes[group] = gi->nr_units * ai->unit_size;   /*CPU 组 percpu 内存长度*/

        for (i = 0; i < gi->nr_units; i++) {       /*遍历组内 CPU 核*/
                cpu = gi->cpu_map[i];
                if (cpu == NR_CPUS)
                        continue;
                ...
                unit_map[cpu] = unit + i;     /*CPU 核编号 ID*/
                unit_off[cpu] = gi->base_offset + i * ai->unit_size;   /*CPU 核 percpu 内存偏移量*/
                if (pcpu_low_unit_cpu == NR_CPUS ||unit_off[cpu] < unit_off[pcpu_low_unit_cpu])
                        pcpu_low_unit_cpu = cpu;
                if (pcpu_high_unit_cpu == NR_CPUS ||unit_off[cpu] > unit_off[pcpu_high_unit_cpu])
                        pcpu_high_unit_cpu = cpu;
        }
}    /*遍历各组各 CPU 核结束*/

pcpu_nr_units = unit;     /*CPU 核数量*/
...
/*设置全局变量*/
pcpu_nr_groups = ai->nr_groups;       /*CPU 核组数量*/
pcpu_group_offsets = group_offsets;     /*CPU 组 percpu 内存偏移量数组*/
pcpu_group_sizes = group_sizes;         /*CPU 组 percpu 内存长度数组*/
pcpu_unit_map = unit_map;               /*CPU 核编号数组*/
pcpu_unit_offsets = unit_off;      /*全局变量，用于设置__per_cpu_offset[cpu]数组*/
                        /*各 CPU 核 percpu 内存起始地址相对于 base_addr 的偏移量*/

pcpu_unit_pages = ai->unit_size >> PAGE_SHIFT;       /*CPU 核 percpu 内存所含页数*/
pcpu_unit_size = pcpu_unit_pages << PAGE_SHIFT;     /*CPU 核 percpu 内存长度，页对齐*/
pcpu_atom_size = ai->atom_size;       /*分配 percpu 内存最小单位*/
pcpu_chunk_struct_size = sizeof(struct pcpu_chunk) +
                        BITS_TO_LONGS(pcpu_unit_pages) * sizeof(unsigned long);
                            /*pcpu_chunk 实例大小，后接位图标记 percpu 内存区中页状态*/

pcpu_nr_slots = __pcpu_size_to_slot(pcpu_unit_size) + 2;
                                /*根据 CPU 核 percpu 内存大小确定 pcpu_slot 指向双链表数组项数*/
pcpu_slot = memblock_virt_alloc(pcpu_nr_slots * sizeof(pcpu_slot[0]), 0); /*创建全局双链表数组*/
for (i = 0; i < pcpu_nr_slots; i++)
        INIT_LIST_HEAD(&pcpu_slot[i]);       /*初始化双链表数组*/
```

```
schunk = memblock_virt_alloc(pcpu_chunk_struct_size, 0);
                                    /*分配 pcpu_chunk 实例（后接位图），管理静态和预留区*/
INIT_LIST_HEAD(&schunk->list);
INIT_WORK(&schunk->map_extend_work, pcpu_map_extend_workfn);
                                                /*异步扩展 map[]数组的工作*/
schunk->base_addr = base_addr;      /*percpu 内存基地址*/
schunk->map = smap;          /*静态 map[]数组，项数为 128*/
schunk->map_alloc = ARRAY_SIZE(smap);
schunk->immutable = true;
bitmap_fill(schunk->populated, pcpu_unit_pages);    /*位图全部置位，表示映射到物理内存*/
schunk->nr_populated = pcpu_unit_pages;    /*每个 CPU 核 percpu 内存所含页数量*/

if (ai->reserved_size) {      /*具有预留区*/
    schunk->free_size = ai->reserved_size;
    pcpu_reserved_chunk = schunk;    /*与静态变量区共用 pcpu_chunk 实例*/
    pcpu_reserved_chunk_limit = ai->static_size + ai->reserved_size; /*静态变量和预留区大小*/
} else {    /*没有设置预留区，则将动态变量区并入静态变量区*/
    schunk->free_size = dyn_size;    /*动态变量区大小*/
    dyn_size = 0;                /*动态区清零*/
}
schunk->contig_hint = schunk->free_size;    /*最大空闲区域大小，预留区或动态变量区大小*/

schunk->map[0] = 1;      /*初始化 map 数组*/
schunk->map[1] = ai->static_size;  /*map 数组第一项值为静态变量区大小*/
schunk->map_used = 1;  /*已使用 map 数组项数最大值*/
if (schunk->free_size)
    schunk->map[++schunk->map_used] = 1 | (ai->static_size + schunk->free_size);  /*map[2]*/
else
    schunk->map[1] |= 1;

/*如果设置了预留区，则具有单独的动态变量区，为其创建 pcpu_chunk 实例*/
if (dyn_size) {
    dchunk = memblock_virt_alloc(pcpu_chunk_struct_size, 0);      /*分配 pcpu_chunk 实例*/
    INIT_LIST_HEAD(&dchunk->list);
    INIT_WORK(&dchunk->map_extend_work, pcpu_map_extend_workfn);
    dchunk->base_addr = base_addr;
    dchunk->map = dmap;
    dchunk->map_alloc = ARRAY_SIZE(dmap);
    dchunk->immutable = true;
    bitmap_fill(dchunk->populated, pcpu_unit_pages);
    dchunk->nr_populated = pcpu_unit_pages;

    dchunk->contig_hint = dchunk->free_size = dyn_size;
```

```
        dchunk->map[0] = 1;
        dchunk->map[1] = pcpu_reserved_chunk_limit;   /*静态区和预留区大小之和*/
        dchunk->map[2] = (pcpu_reserved_chunk_limit + dchunk->free_size) | 1;
        dchunk->map_used = 2;
    }

    pcpu_first_chunk = dchunk ?: schunk;    /*设置了预留区等于 dchunk，否则等于 schunk*/
    pcpu_nr_empty_pop_pages +=pcpu_count_occupied_pages(pcpu_first_chunk, 1);
    pcpu_chunk_relocate(pcpu_first_chunk, -1);    /*由 chunk->free_size 确定双链表*/
                                                  /*将 pcpu_first_chunk 指向实例添加到全局双链表*/

    pcpu_base_addr = base_addr;      /*percpu 内存基地址*/
    return 0;
}
```

pcpu_setup_first_chunk()函数根据参数传递的 pcpu_alloc_info 实例和已分配的 percpu 内存基地址（最低地址），为 percpu 内存创建并设置 pcpu_chunk 实例。

如果设置有预留区域，将创建两个 pcpu_chunk 实例，即静态变量区和预留区共用的 schunk 实例和动态变量区使用的 dchunk 实例。全局变量 **pcpu_reserved_chunk** 与 schunk 指向相同实例，并且 schunk 实例添加到全局双链表，只有分配函数中指定了从预留区分配，才会从此实例中分配变量，而表示动态变量区的 dchunk 实例将插入到全局双链表，用于普通动态变量的分配。

如果没有设置预留区域，将只创建一个 pcpu_chunk 实例，即静态变量区和动态变量区共用的 schunk 实例，并将其插入到全局双链表，可从中分配普通的动态变量。

全局变量 pcpu_first_chunk 指向 schunk 指向的实例（没有设置预留区域）或 dchunk 指向的实例（设置了预留区）。

在设置 pcpu_chunk 实例时，最重要的一项工作就是初始化 map[]数组，它用于动态分配和释放 percpu 变量。每个 map[]数组项对应 percpu 内存区中的一个内存块（变量实例），数组项值表示对应内存块起始地址相对于 percpu 内存起始地址的偏移量，即内存块之前的内存区域大小。下一小节将详细介绍 map[]数组，以及据此实现的变量分配与释放操作。

全局变量 **pcpu_base_addr** 保存了初始 percpu 内存的基地址，**pcpu_unit_offsets** 指向的整数数组保存了各 CPU 核 percpu 内存起始地址相对于基地址 pcpu_base_addr 的偏移量。在 setup_per_cpu_areas()函数的最后将依据这些参数设置全局的**__per_cpu_offset[]**数组，在访问 percpu 变量时需要用到__per_cpu_offset[]数组项值。

3 后期初始化

内核在启动阶段调用的 mm_init()函数中将调用 **percpu_init_late()**函数完成 percpu 分配器的后期初始化，此时伙伴系统和 slab 分配器已经初始化完成。

percpu_init_late()函数定义如下（/mm/percpu.c）：

```
void __init percpu_init_late(void)
{
    struct pcpu_chunk *target_chunks[] ={ pcpu_first_chunk, pcpu_reserved_chunk, NULL };
    struct pcpu_chunk *chunk;
    unsigned long flags;
    int i;
```

```
    for (i = 0; (chunk = target_chunks[i]); i++) {
        int *map;
        const size_t size = PERCPU_DYNAMIC_EARLY_SLOTS * sizeof(map[0]);

        BUILD_BUG_ON(size > PAGE_SIZE);

        map = pcpu_mem_zalloc(size);     /*在内核 vmalloc 区分配空间*/
        BUG_ON(!map);

        spin_lock_irqsave(&pcpu_lock, flags);
        memcpy(map, chunk->map, size);     /*复制 map 数组数据*/
        chunk->map = map;     /*指向新的数组*/
        spin_unlock_irqrestore(&pcpu_lock, flags);
    }
}
```

percpu_init_late()函数的主要工作是为 pcpu_first_chunk 和 pcpu_reserved_chunk 指向 pcpu_chunk 实例在内核空间 vmalloc 区重新分配 map[]数组空间，并复制原数组数据，使 pcpu_chunk 实例使用新分配的 map[]数组。

在前面介绍的 pcpu_setup_first_chunk()函数中静态定义了 smap[]和 dmap[]数组，它们位于内核初始化数据段，在内核启动阶段后期将会被释放。因此此处为 pcpu_first_chunk 和 pcpu_reserved_chunk 指向的实例重新分配并使用 map[]数组，以便在后续能正常分配 percpu 变量。

3.8.3 动态变量

内核在启动阶段创建了初始 percpu 内存，并为其创建了 pcpu_chunk 实例，用于动态分配 percpu 变量，此后内核代码就可以动态创建和释放 percpu 变量了。

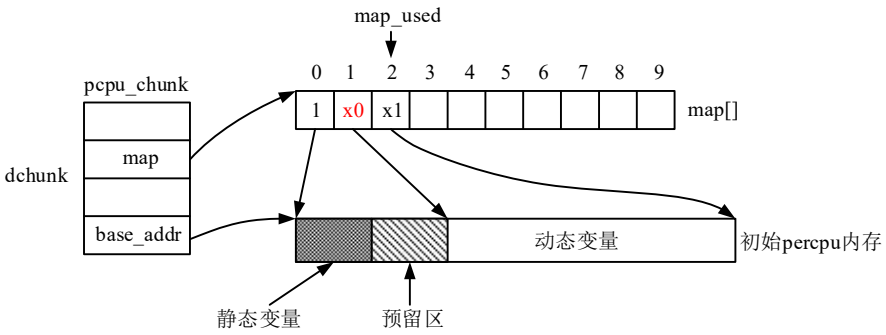
分配变量将通过 pcpu_chunk 实例中的 map[]数组查找到合适的空闲内存块，并分配出去，修改 map[]数组相应项，返回修改后得到的 percpu 变量基准地址。释放 percpu 变量将修改 map[]数组中相应项，标记变量对应的内存空闲。

分配动态变量将从初始 percpu 内存的预留区或动态变量区分配空间，用于保存变量实例。当初始 percpu 内存中预留区或动态变量区使用完了，没有足够的空间时，分配函数将在内核地址空间 VMALLOC 区创建新的 percpu 内存（内核地址空间映射详见下一章），并为其创建 pcpu_chunk 实例，分配函数随后从中分配 percpu 变量。

本小节先介绍分配/释放动态 percpu 变量的原理，然后介绍分配/释放变量函数的实现。

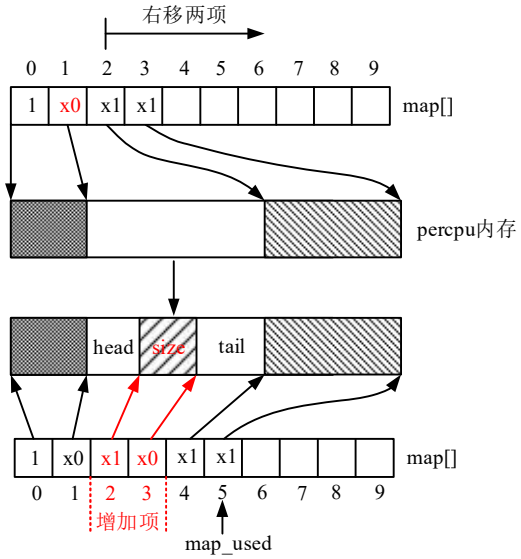
1 分配释放原理

pcpu_chunk 实例中的 map[]各数组项，记录的是各内存块相对于 percpu 内存起始地址的偏移量。如下图所示是初始 percpu 内存中动态变量区（假设存在预留区）对应的 pcpu_chunk 实例的 map[]数组初始值。



map[]数组中每个数组项对应 percpu 内存中的一个内存块，记录的是内存块起始地址相对于 percpu 内存起始地址的偏移量，其中最低位记录的是对应内存块是否空闲，1 表示已使用（已分配），0 表示空闲（可分配）。初始状态 map[]数组使用了前 3 项，map[0]值为 1 表示起始偏移量为 0，对应内存块已被使用，即将静态变量区和预留区视为一个内存块，不能从中分配变量。map[1]值为静态变量区和预留区大小值，最低位为 0，表示内存块空闲。map[2]值为静态变量区、预留区和动态变量区大小之和，最低位为 1，表示后面的内存块不能使用了。

下面来看一下如何从 map[]数组中分配 percpu 变量。如下图所示，假设 map[1]表示一个空闲的内存块，要分配的 percpu 变量的大小为 size，且 map[1]表示的空闲内存块大小大于 size。在 map[1]内存块中分配变量时，为满足变量的对齐要求，可能要舍弃前面 head 字节的空间，如果 head 字节太小将并入 map[0]，如果 head 字节还可利用，则用 map[1]数组项代表此内存块。同理，尾部剩余的字节 tail 如有利用价值则需创建新 map[]数组项，也就是说此时要增加 2 个数组项。一个用于表示分配出去的内存块，一个表示尾部剩余的内存块，分别为数组项 map[2]、map[3]，而原来 map[2]及其后的数组项整体后移 2 项。



map[2]代表的内存块地址转换后，将作为 percpu 变量的基准地址，返回给分配函数的调用者。

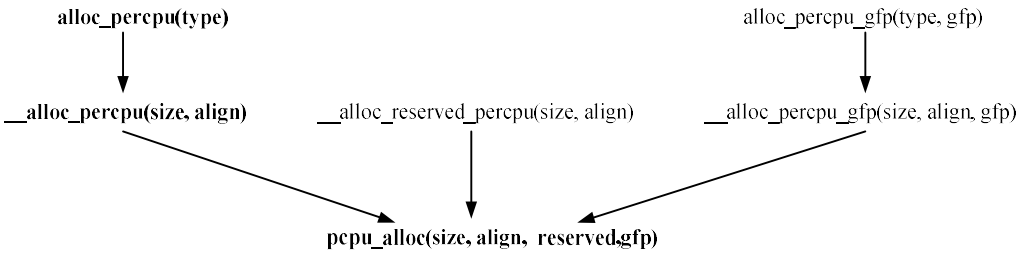
释放 percpu 变量的操作要简单一些，释放内存块时，主要是清零对应 map[]数组项最低位。如果内存块可以与前后的空闲内存块合并，则执行合并操作，生成更大的空闲内存块。

2 分配变量

下面介绍分配 percpu 变量函数的实现。

■分配函数

内核在/include/linux/percpu.h 头文件声明了分配 percpu 变量的接口函数，函数调用关系如下图所示：



所有接口函数最终都是调用 **pcpu_alloc()**函数实现 percpu 变量的分配，__alloc_reserved_percpu()函数表示从初始 percpu 内存中的预留区分配变量。

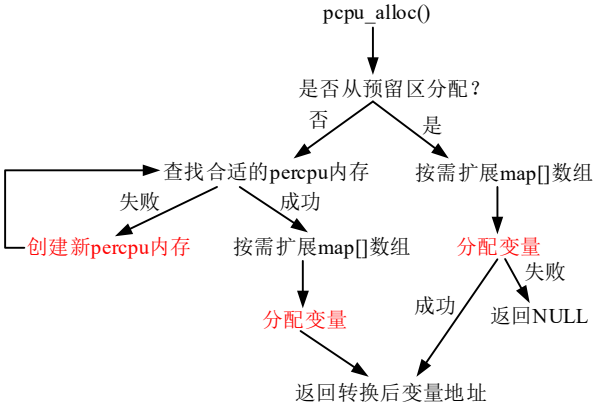
下面我们以常见的 **alloc_percpu()**函数为例，介绍分配 percpu 变量的实现。

```
#define alloc_percpu(type)                                  \
    (typeof(type) __percpu *)__alloc_percpu(sizeof(type),      \
                                            __alignof__(type))          /*变量对齐要求*/
```

__alloc_percpu()函数定义在/mm/percpu.c 文件内，函数代码如下：

```
void __percpu *__alloc_percpu(size_t size, size_t align)
{
    return pcpu_alloc(size, align, false, GFP_KERNEL);     /*/mm/percpu.c*/
}
```

最终的执行函数 pcpu_alloc()执行流程简列如下图所示：



pcpu_alloc()函数判断是否是从初始 percpu 内存的预留区分配变量，如果是则按需扩展 map[]数组，然后进行分配操作，最后成功则返回转换后的变量地址。如果不是从初始 percpu 内存的预留区分配变量，则查找合适的 percpu 内存，找到了合适的则执行同样的分配操作，如果没有找到合适的，则创建新 percpu 内存后，再进行分配操作。

pcpu_alloc()函数代码如下（/mm/percpu.c）：

```
static void __percpu *pcpu_alloc(size_t size, size_t align, bool reserved,gfp_t gfp)
/*size：变量大小，align：对齐字节数，不能超过一页，reserved：是否从预留区分配变量，
*gfp：分配掩码，若设置了 GFP_KERNEL 表示分配不是原子的，没有设置表示分配操作是原子的。
```

```
*/
{
    static int warn_limit = 10;
    struct pcpu_chunk *chunk;
    const char *err;
    bool is_atomic = (gfp & GFP_KERNEL) != GFP_KERNEL;   /*设置 GFP_KERNEL 时为 0*/
                        /*原子分配将不扩展 map 数组，不创建新 percpu 内存，可能失败*/
    int occ_pages = 0;
    int slot, off, new_alloc, cpu, ret;
    unsigned long flags;
    void __percpu *ptr;

    /*对齐要求，最少要 2 字节对齐，因为 map[]数组项值最低位有特殊用途*/
    if (unlikely(align < 2))
       align = 2;

    size = ALIGN(size, 2);

    if (unlikely(!size || size > PCPU_MIN_UNIT_SIZE || align > PAGE_SIZE)) {
          ...              /*变量大小、对齐字节数合法性检查*/
    }
    spin_lock_irqsave(&pcpu_lock, flags);

    if (reserved && pcpu_reserved_chunk) {    /*指示从预留区分配，且存在预留区*/
        chunk = pcpu_reserved_chunk;     /*初始 percpu 内存预留区 pcpu_chunk 实例*/
        if (size > chunk->contig_hint) {       /*变量大小检查*/
           ...
        }

        while ((new_alloc = pcpu_need_to_extend(chunk, is_atomic))) {
                                                    /*检查是否需要扩展 map[]数组，/mm/percpu.c*/
            spin_unlock_irqrestore(&pcpu_lock, flags);
            if (is_atomic || pcpu_extend_area_map(chunk, new_alloc) < 0) {
                ...                  /*扩展 map[]数组，重新分配内存，复制数组数据*/
            }
            spin_lock_irqsave(&pcpu_lock, flags);
        }

        off = pcpu_alloc_area(chunk, size, align, is_atomic,&occ_pages);
                        /*分配变量，返回分配变量在 percpu 内存中的偏移量，/mm/percpu.c*/
        if (off >= 0)
            goto area_found;          /*从预留区分配成功*/
        err = "alloc from reserved chunk failed";
```

```
        goto fail_unlock;           /*从预留区分配失败*/
    }    /*从预留区分配结束*/

    /*以下是从动态变量区分配变量的操作*/
restart:
    /*在全局链表中查找合适的 pcpu_chunk 实例，并从中分配变量*/
    for (slot = pcpu_size_to_slot(size); slot < pcpu_nr_slots; slot++) {
        list_for_each_entry(chunk, &pcpu_slot[slot], list) {
            if (size > chunk->contig_hint)
                continue;

            new_alloc = pcpu_need_to_extend(chunk, is_atomic);   /*是否需要扩展 map[]数组*/
            if (new_alloc) {
                if (is_atomic)
                    continue;
                spin_unlock_irqrestore(&pcpu_lock, flags);
                if (pcpu_extend_area_map(chunk,new_alloc) < 0) {    /*扩展 map[]数组*/
                   ...
                }
                spin_lock_irqsave(&pcpu_lock, flags);
                goto restart;
            }

            off = pcpu_alloc_area(chunk, size, align, is_atomic,&occ_pages);   /*分配变量*/
            if (off >= 0)
               goto area_found;    /*分配成功，跳转至 area_found*/
        }
    }    /*for 循环结束*/

    /*从现有 pcpu_chunk 实例中分配不成功，执行以下代码，创建新 percpu 内存后再分配*/
    spin_unlock_irqrestore(&pcpu_lock, flags);

    if (is_atomic)    /*如果是原子操作，函数返回*/
        goto fail;

    mutex_lock(&pcpu_alloc_mutex);

    if (list_empty(&pcpu_slot[pcpu_nr_slots - 1])) {   /*双链表数组中最后项链表为空则创建 chunk*/
        chunk = pcpu_create_chunk();      /*在 VMALLOC 中创建新 chunk，/mm/percpu-vm.c*/
        ...
        spin_lock_irqsave(&pcpu_lock, flags);
        pcpu_chunk_relocate(chunk, -1);    /*将新创建的 pcpu_chunk 实例插入全局双链表*/
    } else {
```

```
            spin_lock_irqsave(&pcpu_lock, flags);
        }

        mutex_unlock(&pcpu_alloc_mutex);
        goto restart;       /*创建新 percpu 内存后，跳转至 restart 处重新分配*/

        /*以下是 percpu 变量分配成功后执行的代码*/
area_found:
        spin_unlock_irqrestore(&pcpu_lock, flags);
        if (!is_atomic) {       /*不是原子操作，将为分配的没有映射到物理内存的 percpu 变量创建映射*/
            int page_start, page_end, rs, re;          /*适用于 VMALLOC 区中 percpu 内存*/

            mutex_lock(&pcpu_alloc_mutex);

            page_start = PFN_DOWN(off);     /*percpu 变量占用的页*/
            page_end = PFN_UP(off + size);

            pcpu_for_each_unpop_region(chunk, rs, re, page_start, page_end) {   /*扫描未映射区域*/
                WARN_ON(chunk->immutable);
                ret = pcpu_populate_chunk(chunk, rs, re);
                                        /*建立映射，page->index=chunk，/mm/percpu-vm.c*/
                spin_lock_irqsave(&pcpu_lock, flags);
                if (ret) {
                  ...
                }
                pcpu_chunk_populated(chunk, rs, re);   /*设置 pcpu_chunk 实例中位图，/mm/percpu.c*/
                spin_unlock_irqrestore(&pcpu_lock, flags);
            }
            mutex_unlock(&pcpu_alloc_mutex);
        }

        if (chunk != pcpu_reserved_chunk)
            pcpu_nr_empty_pop_pages -= occ_pages;          /*减少未建立映射的页数量*/

        if (pcpu_nr_empty_pop_pages < PCPU_EMPTY_POP_PAGES_LOW)   /*空闲页少时创建映射*/
            pcpu_schedule_balance_work();       /*工作中为 percpu 内存创建物理映射*/

        /*清零刚分配 percpu 变量在各 CPU 核内存区中对应的空间*/
        for_each_possible_cpu(cpu)
            memset((void *)pcpu_chunk_addr(chunk, cpu, 0) + off, 0, size);

        ptr = __addr_to_pcpu_ptr(chunk->base_addr + off);     /*变量地址转为基准地址，/mm/percpu.c*/
        kmemleak_alloc_percpu(ptr, size, gfp);
```

```
    return ptr;     /*返回变量基准地址*/
    ...
}
```

pcpu_alloc()函数如果参数 reserved 不为 0，则从 **pcpu_reserved_chunk** 指示的预留区 pcpu_chunk 实例中分配变量，否则从全局双链表中查找合适的 pcpu_chunk 实例并从中分配变量，如果没有合适的实例，则创建新 pcpu_chunk 实例后再分配。

分配变量的操作首先是检查是否需要扩展 map[]数组，如果需要则为 map[]数组重新分配空间，复制原 map[]数组值（释放原数组），然后再分配变量。

分配变量后，要检查 pcpu_chunk 实例位图，看变量所在虚拟内存有没有映射到物理内存，如果没有则需要建立映射，映射的建立在第 4 章再做介绍。前面介绍的初始 percpu 内存，它是直接线性映射到物理内存，因此其位图全置为 1，表示已经建立了映射。后面新创建的 percpu 内存位于内核地址空间的间接映射区，分配变量后需要检查是否映射到了物理内存，如果没有则需要建立映射。

最后，需要将分配变量的地址转换成 percpu 变量的基准地址，返回给分配函数的调用者。

__addr_to_pcpu_ptr()宏定义在/mm/percpu.c 文件内：

```
#define __addr_to_pcpu_ptr(addr)                    \
    (void __percpu *)((unsigned long)(addr) -       \
                (unsigned long)pcpu_base_addr +     \
                (unsigned long)__per_cpu_start)
                                    /*__per_cpu_offset[0]=pcpu_base_addr-__per_cpu_start*/
```

返回基准地址为变量地址 addr-__per_cpu_offset[0]，在访问变量时会在基准地址上加上偏移量，正好获得变量的实际地址。

pcpu_alloc()函数中调用 pcpu_alloc_area()函数在指定 pcpu_chunk 实例中分配变量，pcpu_create_chunk()函数用于创建新 percpu 内存及 pcpu_chunk 实例。下面分别介绍这两个函数的实现，pcpu_alloc()函数中调用的其它函数请读者自行阅读源代码。

■分配操作

pcpu_alloc_area()函数代码如下（/mm/percpu.c）：

```
static int pcpu_alloc_area(struct pcpu_chunk *chunk, int size, int align,bool pop_only, int *occ_pages_p)
/*chunk：指向 pcpu_chunk 实例，size：分配变量大小，align：对齐字节数，
*pop_only：只在建立映射的区域分配，occ_pages_p：变量占用页数。*/
{
    int oslot = pcpu_chunk_slot(chunk);
    int max_contig = 0;
    int i, off;
    bool seen_free = false;     /*是否已扫描到了空闲内存块（太小不能用于分配变量）*/
    int *p;     /*map[]数组项指针*/

    for (i = chunk->first_free, p = chunk->map + i; i < chunk->map_used; i++, p++) {
                                        /*从第一个空闲内存块对应 map[]数组项开始扫描*/
        int head, tail;
        int this_size;

        off = *p;           /*map 数组项值，*/
```

```
if (off & 1)       /*本数组项对应内存块已使用，遍历下一个内存块*/
    continue;
/*本内存块空闲*/
this_size = (p[1] & ~1) - off;
                    /*p[1]表示下一内存块起始偏移量， this_size 表示本内存块大小*/

head = pcpu_fit_in_area(chunk, off, this_size, size, align,pop_only);
                /*确定 head 大小，如果空闲内存块太小不能分配，haed<0*/
if (head < 0) {
    if (!seen_free) {
        chunk->first_free = i;   /*第一个空闲内存块编号（map[]数组项数）*/
        seen_free = true;        /*扫描到了不可用的空闲内存块*/
    }
    max_contig = max(this_size, max_contig);   /*最大空闲内存块大小*/
    continue;    /*扫描下一项*/
}
/*找到了可分配变量的空闲内存块*/
if (head && (head < sizeof(int) || !(p[-1] & 1))) {
                        /*head 太小不可用或前项空闲，并入前一项*/
    *p = off += head;
    if (p[-1] & 1)
        chunk->free_size -= head;
    else
        max_contig = max(*p - p[-1], max_contig);
    this_size -= head;
    head = 0;
}

tail = this_size - head - size;     /*tail 表示剩余空间大小*/
if (tail < sizeof(int)) {
    tail = 0;
    size = this_size - head;       /*size 表示分配变量大小，可能比申请的大*/
}

/*拆分空闲内存块*/
if (head || tail) {
    int nr_extra = !!head + !!tail;   /*增加的 map[]数组项数*/
    memmove(p + nr_extra + 1, p + 1,sizeof(chunk->map[0]) * (chunk->map_used - i));
                                                /*右移 map[]数组*/
    chunk->map_used += nr_extra;     /*已用数组项数*/

    if (head) {     /*需要为 head 增加 map[]数组项*/
        if (!seen_free) {
```

```
                    chunk->first_free = i;
                    seen_free = true;
                }
                *++p = off += head;   /*p 自加，p 指向分配变量内存对应的数组项*/
                ++i;
                max_contig = max(head, max_contig);
            }
            if (tail) {     /*需要为 tail 增加 map[]数组项*/
                p[1] = off + size;     /*p[1]指向 size 项（分配变量）*/
                max_contig = max(tail, max_contig);
            }
        }

        if (!seen_free)
            chunk->first_free = i + 1;       /*第 1 个空闲内存块编号*/

        if (i + 1 == chunk->map_used)     /*更新最大空闲内存块大小值*/
            chunk->contig_hint = max_contig;
        else
            chunk->contig_hint = max(chunk->contig_hint,max_contig);

        chunk->free_size -= size;
        *p |= 1;     /*分配变量对应 map[]数组项值最低位置 1，表示已使用*/

        *occ_pages_p = pcpu_count_occupied_pages(chunk, i);   /*变量占用页数量*/
        pcpu_chunk_relocate(chunk, oslot);     /*pcpu_chunk 实例重新插入全局双链表*/
        return off;       /*返回变量起始地址的偏移量*/
    }   /*扫描 map[]数组结束*/

    /*分配不成功，执行以下代码*/
    chunk->contig_hint = max_contig;/* fully scanned */
    pcpu_chunk_relocate(chunk, oslot);
    return -1;
}
```

pcpu_alloc_area()函数用于实现本小节开始介绍的分配变量操作，请读者结合本小节开始内容，理解函数源代码。

■创建 percpu 内存

pcpu_alloc()函数中调用 pcpu_create_chunk()函数用于分配新 percpu 内存，并为其创建 pcpu_chunk 实例，函数定义如下（/mm/percpu-vm.c）：

```
static struct pcpu_chunk *pcpu_create_chunk(void)
{
    struct pcpu_chunk *chunk;
```

```
    struct vm_struct **vms;

    chunk = pcpu_alloc_chunk();        /*创建并设置 pcpu_chunk 实例，见下文，/mm/percpu.c*/
    if (!chunk)
        return NULL;

    vms = pcpu_get_vm_areas(pcpu_group_offsets, pcpu_group_sizes,
                                        pcpu_nr_groups, pcpu_atom_size);
                                            /*在内核空间创建映射内存区，/mm/vmalloc.c*/
    ...
    chunk->data = vms;
    chunk->base_addr = vms[0]->addr - pcpu_group_offsets[0];      /*percpu 内存基地址*/
    return chunk;      /*返回 pcpu_chunk 实例指针*/
}
```

pcpu_create_chunk()函数主要分两步，一是创建并设置 pcpu_chunk 实例，二是在内核间接映射区分配 percpu 内存。pcpu_alloc_chunk()函数用于创建并设置 pcpu_chunk 实例，pcpu_get_vm_areas()函数用于在内核 VMALLOC 区分配一个虚拟内存域，用于映射 percpu 内存，相关内容第 4 章再做介绍。

下面看一下 pcpu_alloc_chunk()函数的定义，代码如下（/mm/percpu.c）：

```
static struct pcpu_chunk *pcpu_alloc_chunk(void)
{
    struct pcpu_chunk *chunk;

    chunk = pcpu_mem_zalloc(pcpu_chunk_struct_size);     /*分配全零 pcpu_chunk 实例*/
    if (!chunk)
        return NULL;

    chunk->map = pcpu_mem_zalloc(PCPU_DFL_MAP_ALLOC *sizeof(chunk->map[0]));
                                                        /*为 map[]数组分配空间/
    ...
    chunk->map_alloc = PCPU_DFL_MAP_ALLOC;
    chunk->map[0] = 0;
    chunk->map[1] = pcpu_unit_size | 1;          /*pcpu_unit_size 为 percpu 内存大小*/
    chunk->map_used = 1;

    INIT_LIST_HEAD(&chunk->list);
    INIT_WORK(&chunk->map_extend_work, pcpu_map_extend_workfn);
    chunk->free_size = pcpu_unit_size;
    chunk->contig_hint = pcpu_unit_size;

    return chunk;      /*返回 pcpu_chunk 实例指针*/
}
```

3 释放变量

动态分配的 percpu 变量（不含静态定义的变量），不使用时可将其释放，释放函数为 **free_percpu()**，函数定义如下（/mm/percpu.c）：

```
void free_percpu(void __percpu *ptr)
 /*ptr：释放变量指针（基准地址）*/
{
    void *addr;
    struct pcpu_chunk *chunk;
    unsigned long flags;
    int off, occ_pages;

    if (!ptr)
        return;

    kmemleak_free_percpu(ptr);
    addr = __pcpu_ptr_to_addr(ptr);        /*ptr 指针转为实际的变量地址*/
    spin_lock_irqsave(&pcpu_lock, flags);
    chunk = pcpu_chunk_addr_search(addr);
                /*由变量地址查找 pcpu_chunk 实例，page->index=chunk，/mm/percpu.c*/
    off = addr - chunk->base_addr;     /*变量在 percpu 内存中的偏移量*/

    pcpu_free_area(chunk, off, &occ_pages);   /*释放变量占用内存块，/mm/percpu.c*/

    if (chunk != pcpu_reserved_chunk)
        pcpu_nr_empty_pop_pages += occ_pages;   /*增加 percpu 内存中空闲页数量*/

    if (chunk->free_size == pcpu_unit_size) {   /*percpu 变量全部释放完，释放 percpu 内存*/
        struct pcpu_chunk *pos;

        list_for_each_entry(pos, &pcpu_slot[pcpu_nr_slots - 1], list)
            if (pos != chunk) {
                pcpu_schedule_balance_work();   /*销毁 chunk*/
                break;
            }
    }
    spin_unlock_irqrestore(&pcpu_lock, flags);
}
```

释放函数首先将参数传递的变量基地址 ptr 转为实际变量的地址，pcpu_chunk_addr_search(addr)函数由变量地址 addr 查找 pcpu_chunk 实例时，如果变量地址位于初始 percpu 内存区，则 pcpu_chunk 实例为初始 pcpu_chunk 实例。如果 addr 位于内核 VMALLOC 区，则将 addr 转为映射页的 page 实例，其 page->index 成员保存了 pcpu_chunk 实例地址。

然后，调用 pcpu_chunk_addr_search()函数查找变量所属的 pcpu_chunk 实例，调用 pcpu_free_area()函数将变量所占的内存块释放，如果释放内存块可与前后内存块合并成更大的空闲内存块，则合并，并将之

后的 map[]数组项前移。

最后，如果 pcpu_chunk 实例所在的 percpu 内存中所有 percpu 变量都释放了，则调度执行平衡工作线程异步锁毁 pcpu_chunk 实例（释放映射物理内存等）。

3.9 小结

内核按页（通常是 4KB 或 8KB）对物理内存进行管理。每个内存页称为页帧，内核定义了 page 实例数组用于表示页帧，并对页帧进行管理。

物理内存管理主要是物理内存的分配与释放。在启动阶段通过自举分配器等管理物理内存，随后将启用伙伴系统，用于管理物理内存。

伙伴系统用于按页（多页）分配释放物理内存，分配的粒度较大。为满足小块内存（数据结构实例）的分配，内核实现了 slab/slub/slob 分配器。内核物理内存管理的主要内容就是伙伴系统和 slab/slub/slob 分配器。

伙伴系统接口函数：

struct page *alloc_pages(mask,order)：分配 2^{order} 数量连续页帧，返回首面 page 实例指针。

struct page *alloc_page(mask)：分配一个页帧，返回 page 实例指针。

unsigned long __get_free_pages(mask,order)：分配 2^{order} 数量连续页帧，返回分配内存块的起始虚拟地址，只能分配低端内存。

unsigned long get_zeroed_page(mask)：分配一个页帧并返回起始虚拟地址，页帧内容清 0，只能分配低端内存。

void free_page(addr)、void free_pages(addr,order)：释放指定起始虚拟地址内存块。

void __free_pages(page,order)、void __free_page(page)：释放 page 实例表示起始页帧的内存块。

sl[a,u,o]b 分配器接口函数：

struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align,unsigned long flags, void (*ctor)(void *))：创建指定大小对象的 slab 缓存，返回 kmem_cache 实例指针。

void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)：从指定 slab 缓存分配对象函数，返回对象指针。

void kmem_cache_free(struct kmem_cache *, void *)：释放从指定 slab 缓存中分配的对象。

void *kmem_cache_zalloc(struct kmem_cache *k, gfp_t flags)：从指定 slab 缓存分配对象函数，对象清零，返回对象指针。

void *kmalloc(size_t size, gfp_t flags)：从通用缓存中分配指定大小内存块，返回起始虚拟地址。

void *kzalloc(size_t size, gfp_t flags)：从通用缓存中分配指定大小内存块，内存块清零，返回起始虚拟地址。

void kmem_cache_free(struct kmem_cache *, void *)：释放从指定 slab 缓存中分配的对象。

void kfree(const void *)：释放从通用缓存分配的内存块。