# 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2order，例如：0阶表示连续页帧数为20=1，2阶表示连续页帧数为22=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（2MAX\_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2wait\_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2order。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2MAX\_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210=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2MAX\_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2×21=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2wait\_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：



#### 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2order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\_pag**e); /\*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2order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2order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2gfporder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2order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2order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。