Linux内存管理代码分析：伙伴系统

         Linux内存管理系统是对NUMA计算机内存模型的抽象。Linux内存管理系统的设计框架也是把内存分为不同的结点（pglist\_data），每个cpu对应一个本地结点，每个结点有几个的区域（zone），这是因为一个结点的不同区域可能有不同的访问属性，如在x86系统上，dma只能访问只能访问较低端的内存区域。区域是页（page）的集合，每个区域包含一个范围的页，每个页有一个页帧号，在系统里面唯一的，通常用pfn表示。页是内核内存管理的最基本单位，在内核中每页会有一个页面结构（page）来记录页的一些管理数据，一般mmu是以页为单位来进行内存映射的。调用函数getpagesize可以获得页的大小。

         页是伙伴系统内存管理的基本单位，但伙伴系统并不是直接基于页来进行内存管理的。在页之上，我提出块的概念，每个块包含连续的若干页面。在伙伴系统中每个块包含2^order块页面，我们把order叫做块的阶，块还有迁移类型属性，不同的块可能属于不同的迁移类型。伙伴系统把一些阶一样的块连接成一个链表，当要分配阶为n的块时，实质就是找到一个包含阶为n的空闲块的链表，在链表中摘除一项，返回这一项管理的物理地址。

每次内存分配都是在一个区域里面完成的，但在区域里面并不是直接一页一页来进行分配，而是每次分配一块。下面是这几个结构的详细说明

         typedefstruct pglist\_data {

         structzone node\_zones[MAX\_NR\_ZONES];            //节点包含的区域

         structzonelist node\_zonelists[MAX\_ZONELISTS];           //节点列表数组，在numa系统下有两个，第零个全局列表，包含系统所有区域，第一个包含本地区域。本地址列表的初始化顺序是ZONE\_DMA，ZONE\_NORMAL，ZONE\_HIGHMEM。全局列表的初始化是，对每个节点的初始化顺序和本地节点一样，对节点的初始化顺序是先初始化本节点，然后增加节点号，到底最大节点号后归零，包含每个节点的区域一次。

         intnr\_zones;             //本节点包含的区域数

#ifdef CONFIG\_FLAT\_NODE\_MEM\_MAP         /\* means !SPARSEMEM \*/

         structpage \*node\_mem\_map;

#ifdef CONFIG\_CGROUP\_MEM\_RES\_CTLR

         structpage\_cgroup \*node\_page\_cgroup;

#endif

#endif

#ifndef CONFIG\_NO\_BOOTMEM

         structbootmem\_data \*bdata;

#endif

#ifdef CONFIG\_MEMORY\_HOTPLUG

         spinlock\_tnode\_size\_lock;

#endif

         unsignedlong node\_start\_pfn;                 //节点的第一个页的页帧

         unsignedlong node\_present\_pages; /\* total number of physical pages \*/            //本节点物理内存总页数

         unsignedlong node\_spanned\_pages; /\* total size of physical page                //本节点可用的物理页面总数

                                                    range, including holes \*/

         intnode\_id;               //本节点id，全局唯一

         wait\_queue\_head\_tkswapd\_wait;

         structtask\_struct \*kswapd;   /\* Protected bylock\_memory\_hotplug() \*/

         intkswapd\_max\_order;

         enumzone\_type classzone\_idx;

} pg\_data\_t;

struct zone {

         unsignedlong watermark[NR\_WMARK];                  //水位线

         unsignedlong percpu\_drift\_mark;

         unsignedlong           lowmem\_reserve[MAX\_NR\_ZONES];               //每个区域会保留一些内存，在这里记录

         unsignedlong           dirty\_balance\_reserve;

#ifdef CONFIG\_NUMA

         intnode;  //所属的NUMA节点

   unsigned long     min\_unmapped\_pages;  //当

内存管理区中，用于slab的可回收页大于此值时，将回收slab中的缓存页。

         unsignedlong           min\_slab\_pages;

#endif

         structper\_cpu\_pageset \_\_percpu \*pageset;         //每CPU的页面缓存， 当分配单个页面时，首先从该缓存中分配页面，可提高效率

         spinlock\_t                  lock; //该锁用于保护伙伴系统数据结构。即保护free\_area相关数据。

         int                     all\_unreclaimable; /\* Allpages pinned \*/

#ifdef CONFIG\_MEMORY\_HOTPLUG

         /\*see spanned/present\_pages for more description \*/

         seqlock\_t          span\_seqlock;

#endif

         structfree\_area      free\_area[MAX\_ORDER]; //空闲块链表，这个数组定义了11个队列，每个队列中的元素都是大小为2^n的页面块。

#ifndef CONFIG\_SPARSEMEM

         unsignedlong           \*pageblock\_flags;

#endif /\* CONFIG\_SPARSEMEM \*/

#ifdef CONFIG\_COMPACTION

         unsignedint              compact\_considered;

         unsignedint              compact\_defer\_shift;

         int                       compact\_order\_failed;

#endif

         ZONE\_PADDING(\_pad1\_)

         /\*Fields commonly accessed by the page reclaim scanner \*/

         spinlock\_t                  lru\_lock;

         structlruvec              lruvec;

         structzone\_reclaim\_stat reclaim\_stat;

         unsignedlong           pages\_scanned;         /\* since last reclaim \*/

         unsignedlong           flags;                     /\* zone flags, see below \*/

         /\*Zone statistics \*/

         atomic\_long\_t          vm\_stat[NR\_VM\_ZONE\_STAT\_ITEMS];

         unsignedint inactive\_ratio;

         ZONE\_PADDING(\_pad2\_)

         wait\_queue\_head\_t         \* wait\_table;

         unsignedlong           wait\_table\_hash\_nr\_entries;

         unsignedlong           wait\_table\_bits;

         structpglist\_data    \*zone\_pgdat;           //指向本区域所在的节点结构

         unsignedlong           zone\_start\_pfn;                //本区域第一页的也帧

         unsignedlong           spanned\_pages;      /\* total size, including holes \*/        //本区域总页数，包含中间的空洞

         unsignedlong           present\_pages;        /\* amount of memory (excluding holes) \*/               //本区域可用的总页数

         constchar                  \*name;              //本区域名称，如Normal、DMA、Highmem等

} \_\_\_\_cacheline\_internodealigned\_in\_smp;

struct page {

         unsignedlong flags;

         structaddress\_space \*mapping;    //如果最低位为0，则指向inode 的address\_space，或则为NULL。如果页映射为匿名内存，最低位置位，而且该指针指向anon\_vma对象

         struct{

                   union{

                            pgoff\_tindex;           /\* Our offset withinmapping. \*/

                            void\*freelist;            /\* slub first freeobject \*/

                   };

                   union{

#if defined(CONFIG\_HAVE\_CMPXCHG\_DOUBLE)&& \

         defined(CONFIG\_HAVE\_ALIGNED\_STRUCT\_PAGE)

                            unsignedlong counters;

#else

                            unsignedcounters;

#endif

                            struct{

                                     union{

                                               atomic\_t\_mapcount; //表示在也表中有多少项指向该页

                                               struct{

                                                        unsignedinuse:16;

                                                        unsignedobjects:15;

                                                        unsignedfrozen:1;

                                               };       //用于slub页的使用计数

                                     };

                                     atomic\_t\_count;              /\* Usage count, seebelow. \*/

                            };

                   };

         };

         /\*Third double word block \*/

         union{

                   structlist\_head lru;                   //lru链表

                   struct{              /\* slub per cpu partialpages \*/

                            structpage \*next;   /\* Next partial slab \*/

#ifdef CONFIG\_64BIT

                            intpages;         /\* Nr of partial slabs left\*/

                            intpobjects;     /\* Approximate # of objects\*/

#else

                            shortint pages;

                            shortint pobjects;

#endif

                   };

         };

         //指向私有数据的指针，虚拟内存管理会忽略该数据。根据页的用途，可以用不同的方式使用该指针。

         union{

                   unsignedlong private;

#if USE\_SPLIT\_PTLOCKS

                   spinlock\_tptl;

#endif

                   structkmem\_cache \*slab;      //用于SLUB分配器，指向slab的指针

                   structpage \*first\_page; //用于复合页，指向首页

         };

#if defined(WANT\_PAGE\_VIRTUAL)

         void\*virtual;                      //用于高端内存区中的页，即无法直接映射到内核内存中的页。Virtual用于存储该页的虚拟地址。

#endif /\* WANT\_PAGE\_VIRTUAL \*/

#ifdef CONFIG\_WANT\_PAGE\_DEBUG\_FLAGS

         unsignedlong debug\_flags;     /\* Use atomic bitopson this \*/

#endif

#ifdef CONFIG\_KMEMCHECK

         void\*shadow;  //指向影子页面

#endif

}

为了更好的管理内存，还有几个辅助管理结构

1：区域索引（zoneref），根据区域所引可以定位区域和求得区域id 。

2：区域列表（zonelist），区域列表结构的目的是按一定的顺序把区域索引结构保存起来，在分配的时候对满足条件的区域索引索引的区域进行扫描，在扫描到的区域里分配内存。

3：区域缓存（zonelist\_cache），区域缓存主要目的是用来记录区域的内存紧张状况。

4：nodemask\_t，节点掩码，一个位图数组，根据不同情况，用来标记一个节点或一个区域的集合

5：free\_area空闲区域，包含一个链表数组，对每个迁移类型对应一个链表，每个链表的每一项都链接的是一个空闲块。

         以下伙伴系统内存管理系统的设计考虑到几个因素：

1：按阶分配内存页，即一次可以分配2^n页内存；

2：防止内存碎片，伙伴系统中采用伙伴算法来避免碎片，另外还采用迁移类型来避免内存碎片

3：页面交换，可以使程序使用到比实际物理内存更多的内存空间；

4：另外程序可以设置内存策略，不同的内存策略会有不同的内存分配结果；

5：水位线，用来在区域保留一定的内存，避免内存完全用尽的情形。

6：分配选项，如可等待选项等

7：在不同的结点上的内存分配，在不同的结点有不同的内存访问属性，访问本地节点的速度一般是快的

一次分配2^order页内存的实现方法是设置一个链表数组，数组里面项的索引对应order，分配的时候从链表里面摘除一项就可以了。

         防止内存碎片采用了伙伴系统和迁移类型两种方法。伙伴系统在内存回收的过程中把伙伴块合并成更大的块来避免内存碎片，后面这分析伙伴系统内存分配函数的时候会全面分析伙伴系统算法。伙伴系统从内存回收的角度去解决内存碎片，迁移类型则从内存分配的角度解决内存碎片，把区域分为不同的迁移类型，根据不同迁移类型来分配，这样在一个迁移类型里面有碎片，但在另外一个迁移类型里面没有碎片，这样可以在一定程度上避免内存碎片；另外对造成碎片的可迁移到页面，把它迁移到其他不引起碎片的页面，这样也可以消除碎片。

         下面先介绍几个影响伙伴系统内存管理的设计因素：

内存迁移

         这里介绍的内存迁移是和货币系统快速内存分配相关部分，而没有涉及到已经分配出去的页面的迁移，因为这一部分的内存比较多，需要单独一个章节才能介绍得完。

         在头文件include/linux/mmzone.h中定义了以下五中迁移类型

MIGRATE\_UNMOVABLE    这类页在内存当中有固定的位置，不能移动。内核的核心分配的内存大多属于这种类型

MIGRATE\_RECLAIMABLE  这类页不能直接移动，但可以删除，其内容页可以从其他地方重新生成，例如，映射自文件的数据属于这种类型，针对这种页，内核有专门的页面回收处理

MIGRATE\_MOVABLE可移动页:这类页可以随意移动，用户空间应用程序所用到的页属于该类别。它们通过页表来映射，如果他们复制到新的位置，页表项也会相应的更新，应用程序不会注意到任何改变。

MIGRATE\_PCPTYPES      是per\_cpu\_pageset，即用来表示每CPU页框高速缓存的数据结构中的链表的迁移类型数目

MIGRATE\_RESERVE       是在前三种的列表中都没用可满足分配的内存块时，就可以从MIGRATE\_RESERVE分配

MIGRATE\_ISOLATE       用于跨越NUMA节点移动物理内存页，在大型系统上，它有益于将物理内存页移动到接近于是用该页最频繁地CPU

MIGRATE\_TYPES         表示迁移类型的数目

         迁移会在两个方面进行，空闲内存和已分配内存，对每个页面会有个迁移类型，以位图数组的形式保存在zone结构的成员pageblock\_flags指向的内存中，在伙伴系统回收内存的时候会在这个位图数组中获得迁移类型，页面块释放到相应的空闲迁移链表中。对空闲内存，每个空闲区域（free\_area），包含一个空闲块链表，对每个迁移类型都对应一个空闲链表。

get\_pageblock\_migratetype函数

get\_pageblock\_migratetype函数用于获得页面的迁移类型。get\_pageblock\_migratetype函数在include/linux/mmzone.h中定义，代码如下：

52 static inlineint get\_pageblock\_migratetype(struct page \*page)

  53 {

  54         return get\_pageblock\_flags\_group(page,PB\_migrate, PB\_migrate\_end);

  55 }

         get\_pageblock\_migratetype函数调用get\_pageblock\_flags\_group函数来获得迁移类型，PB\_migrate和PB\_migrate\_end是枚举类型，在include/linux/pageblock-flags.h中定义

29 enum pageblock\_bits {

 30         PB\_migrate,

 31         PB\_migrate\_end = PB\_migrate + 3 - 1,

 32                         /\* 3 bits required formigrate types \*/

 33         NR\_PAGEBLOCK\_BITS

 34 };

get\_pageblock\_flags\_group函数中mm/page\_alloc.c中实现，代码如下：

5355 unsigned long get\_pageblock\_flags\_group(struct page \*page,

5356                                         intstart\_bitidx, int end\_bitidx)

5357 {

5358         struct zone\*zone;

5359         unsigned long\*bitmap;

5360         unsigned longpfn, bitidx;

5361         unsigned longflags = 0;

5362         unsigned long value = 1;

5363

5364         zone =page\_zone(page);

5365         pfn =page\_to\_pfn(page);

5366         bitmap =get\_pageblock\_bitmap(zone, pfn);

5367         bitidx =pfn\_to\_bitidx(zone, pfn);

5368

5369         for (;start\_bitidx <= end\_bitidx; start\_bitidx++, value <<= 1)

5370                 if(test\_bit(bitidx + start\_bitidx, bitmap))

5371                        flags |= value;

5372

5373         return flags;

5374 }

get\_pageblock\_flags\_group调用函数pfn\_to\_bitidx获得页面对应的第一页的索引保存在变量bitidx中，然后按位获取从bitidx+ start\_bitidx位到bitidx+ end\_bitidx-1位的值返回。

在zone结构中包含一个成员pageblock\_flags，pageblock\_flags指向一个位图数组，每个页面占用NR\_PAGEBLOCK\_BITS位，在这个位图数组中保存了页面的迁移类型。

         每个页面的第一位是通过页面的页帧号和页面所在的区域计算出的，计算的函数是：

         pfn\_to\_bitidx在mm/page\_alloc.c中实现，代码如下：

5337 static inline int pfn\_to\_bitidx(struct zone \*zone, unsignedlong pfn)

5338 {

5339 #ifdef CONFIG\_SPARSEMEM

5340         pfn &=(PAGES\_PER\_SECTION-1);

5341         return (pfn>> pageblock\_order) \* NR\_PAGEBLOCK\_BITS;

5342 #else

5343         pfn = pfn -zone->zone\_start\_pfn;

5344         return (pfn>> pageblock\_order) \* NR\_PAGEBLOCK\_BITS;

5345 #endif /\* CONFIG\_SPARSEMEM \*/

5346 }

我们考虑在没有定义CONFIG\_SPARSEMEM宏的情况，计算方法是页帧号右移pageblock\_order位乘以NR\_PAGEBLOCK\_BITS。这样代表这样的意思，就是迁移类型是按最大块来区分的，每个迁移块包pageblock\_order个页面，处于同一迁移块的页具有相同的迁移类型。

set\_pageblock\_migratetypeh函数

         上面的get\_pageblock\_migratetype 函数是用来获取页面迁移类型的，set\_pageblock\_migratetype函数是用来设置迁移类型的。get\_pageblock\_migratetype 函数在mm/page\_alloc.c中实现，代码如下：

221 static void set\_pageblock\_migratetype(struct page \*page, intmigratetype)

 222{

 223

 224        if (unlikely(page\_group\_by\_mobility\_disabled))

 225                 migratetype = MIGRATE\_UNMOVABLE;

 226

 227        set\_pageblock\_flags\_group(page, (unsigned long)migratetype,

 228                                        PB\_migrate, PB\_migrate\_end);

 229}

                   在page\_group\_by\_mobility\_disabled变量为真情况下，把迁移类型设置为MIGRATE\_UNMOVABLE，这样每次迁移都会把迁移类型迁移为MIGRATE\_UNMOVABLE类型。

                   set\_pageblock\_flags\_group函数只是在迁移类型位图数组中按位设置迁移类型的值。

move\_freepages\_block函数

上面的get\_pageblock\_migratetype函数是用来获取页面迁移类型的，move\_freepages\_block函数是用来设置迁移类型的，move\_freepages\_block函数在mm/page\_alloc.c中实现，代码如下：

932 static int move\_freepages\_block(structzone \*zone, struct page \*page,

 933                                 intmigratetype)

 934{

 935        unsigned long start\_pfn, end\_pfn;

 936        struct page \*start\_page, \*end\_page;

 937

 938        start\_pfn = page\_to\_pfn(page);

 939        start\_pfn = start\_pfn & ~(pageblock\_nr\_pages-1);

 940        start\_page = pfn\_to\_page(start\_pfn);

 941        end\_page = start\_page + pageblock\_nr\_pages - 1;

 942        end\_pfn = start\_pfn + pageblock\_nr\_pages - 1;

 943

 944        /\* Do not cross zone boundaries \*/

 945        if (start\_pfn < zone->zone\_start\_pfn)

 946                 start\_page = page;

 947        if (end\_pfn >= zone->zone\_start\_pfn + zone->spanned\_pages)

 948                 return 0;

 949

 950        return move\_freepages(zone, start\_page, end\_page, migratetype);

 951}

         move\_freepages\_block函数对一个范围内的页面进行迁移。这样范围的第一个页面由参数page传进来，页面数由宏pageblock\_nr\_pages定义。pageblock\_nr\_pages的值分两种情况，在配置在大页选项的情况下pageblock\_nr\_pages等于一个大页包含的页面数，否则等于最大空闲页包含的页面数。

         这个函数只是求出第一个要迁移的页面page结构指针，最后一个要迁移的页面page结构指针，调用move\_freepages函数进行迁移。

move\_freepages函数

         每个空闲区域（free\_area）都包含若一个空闲链表数组（free\_list），其实是对每个迁移类型为下标就得到对应的空闲链表。move\_freepages把一个范围的页面迁移到一个区域的迁移类型空闲链表中。参数zone是要进行迁移到区域。 start\_page 是要迁移到第一个页面结构地址，end\_page 是最后一个要迁移到页面结构地址。migratetype是要迁移到的迁移类型。move\_freepages在mm/page\_alloc.c中实现，代码如下：

         889static int move\_freepages(struct zone \*zone,

 890                           struct page\*start\_page, struct page \*end\_page,

 891                           int migratetype)

 892{

 893        struct page \*page;

 894        unsigned long order;

 895        int pages\_moved = 0;

 896

 897#ifndef CONFIG\_HOLES\_IN\_ZONE

 898        /\*

 899         \* page\_zone is not safe to call in this context when

 900         \* CONFIG\_HOLES\_IN\_ZONE is set. This bug check is probably redundant

 901         \* anyway as we check zone boundaries in move\_freepages\_block().

 902         \* Remove at a later date when no bug reports exist related to

 903         \* grouping pages by mobility

 904         \*/

 905        BUG\_ON(page\_zone(start\_page) != page\_zone(end\_page));

 906#endif

 907

 908        for (page = start\_page; page <= end\_page;) {

 909                 /\* Make sure we are notinadvertently changing nodes \*/

 910                 VM\_BUG\_ON(page\_to\_nid(page) !=zone\_to\_nid(zone));

 911

 912                 if(!pfn\_valid\_within(page\_to\_pfn(page))) {

 913                         page++;

 914                         continue;

 915                 }

 916

 917                 if (!PageBuddy(page)) {

 918                         page++;

 919                         continue;

 920                 }

 921

 922                 order = page\_order(page);

 923                 list\_move(&page->lru,

 924                          &zone->free\_area[order].free\_list[migratetype]);

 925                 page += 1 << order;

 926                 pages\_moved += 1 << order;

 927        }

 928

 929        return pages\_moved;

 930}

         895行定义实际迁移的页面数量并初始化为0。

         908行从第一个页面开始进行循环，值到最后一个页面。

         910行对节点号进行检查。

         912行对页帧号进行检查。

         917行检查页面是不是buddy系统的页面。

         922行获得阶。

         923-924行进行空闲链表迁移。

         925行对增加page地址。

         926行对迁移页面数加1 << order

         929返回迁移的页面数量。