第 11 章 内存与块设备交互

第 10 章介绍的块设备驱动程序用于实现对块设备的读写操作和控制，这称它为访问块设备的机制。本章要介绍的是 VFS 层访问块设备的策略问题，简单地说就是在什么场合、什么时机、访问块设备的哪些数据块，访问操作由块设备驱动程序执行。

本章内容主要包括四部分，分别是页缓存与块缓存、通用读写文件函数、数据回写和页回收与页交换。

（1）页缓存与块缓存

内核打开块设备文件或分区文件系统中普通文件时，会在内存中建立此文件内容的缓存（由地址空间管理），即页缓存，以提高文件的访问效率。因为块设备访问速度较慢，访问内存较快，因此在内存中建立文件内容的缓存。

页缓存由（文件）地址空间管理，缓存单位为一页，地址空间通过基数树管理缓存页。用户对文件内容的读写通常是对页缓存内容的读写，由地址空间操作结构中函数实现页缓存与块设备之间的数据传输。

块缓存寄生于页缓存，块缓存是对页缓存的划分，将缓存页划分成更小的缓存块，即更小的缓存单位。块缓存可对块设备的读写进行更细粒度的控制，以提高效率。例如：如果只修改了缓存页中一小部分内容，则可以只对修改过的缓存块执行写操作，而其它缓存块不需要写出到块设备。

（2）通读写文件函数

进程对普通文件的读写通常都是对页缓存的读写，这与文件所处的文件系统类型无关。因此内核定义了通用的读写文件内容函数，通用读写函数实现用户内存与文件页缓存之间的数据传输，这是内存到内存的数据传输。

页缓存与块设备之间的数据传输由地址空间操作结构中的函数实现，因此各文件系统类型需要定义的是地址空间操作结构实例。

内核也支持不经过页缓存，直接实现用户内存与块设备之间的传输，不过这种方式效率低，不常用，只在特殊场景下使用。

（3）数据回写

VFS 对块设备中普通文件（包括块设备文件）进行访问时通常会在内存中建立缓存，写文件操作就是将数据写入文件内容页缓存。内核需要在适当的时机将页缓存中修改过的数据写入块设备，这称为数据回写。数据回写内容不仅包括文件内容，还包括文件元数据，超级块数据等。

内核触发数据回写的时机有：周期回写，脏页平衡回写（脏页较多了），分配内存（内存不足）时。另外，用户进程也可以通过系统调用对文件、文件系统等执行数据回写。

（4）页回收与页交换

系统中的物理内存总是不够用的，内核需要对有限的物理内存进行管理调度。在空闲内存紧张时，内核会对分配给进程使用的内存，按照最近最少使用原则进行回收，即将其释放回伙伴系统，成为空闲页。

分配给内核使用的内存一般不回收，除非内核自己把它释放，不过页回收机制可能会收缩 slab 缓存。

对用户内存的回收主要包括匿名映射页和缓存页（可能映射到进程地址空间）的回收。

对缓存页的回收比较简单，主要是将缓存页中数据（脏页）写回块设备即可释放页，如果缓存页映射到了用户进程地址空间，还需要断开映射。缓存页下次需要时再分配页，从文件中读回数据即可。

对匿名映射页（保存进程运行数据）的回收稍微复杂一些，因为页中数据不是来自块设备，而是进程运行时产生的，如果要回收匿名映射页，得为其在块设备中找个地方暂存数据，下次需要时再读回来，这称为页交换。页交换其实是页回收的一个副产品，只用于回收匿名映射页。

用户可设置一个或数个文件或分区，用于暂存进程匿名映射页数据，这称为交换区。回收匿名映射页就是将页数据写出至交换区，并将位置信息写入原映射页表项，然后释放页。下次需要时再分配页，从交换区中读回数据，恢复映射关系。

11.1 页缓存与块缓存

页缓存是以页为单位建立的文件内容（含块设备文件）在内存中的缓存。因为块设备读写较慢，因此

缓存其内容以提高效率，如同 CPU 核内的 cache。页缓存由地址空间管理，地址空间实现页缓存与块设备的数据传输。

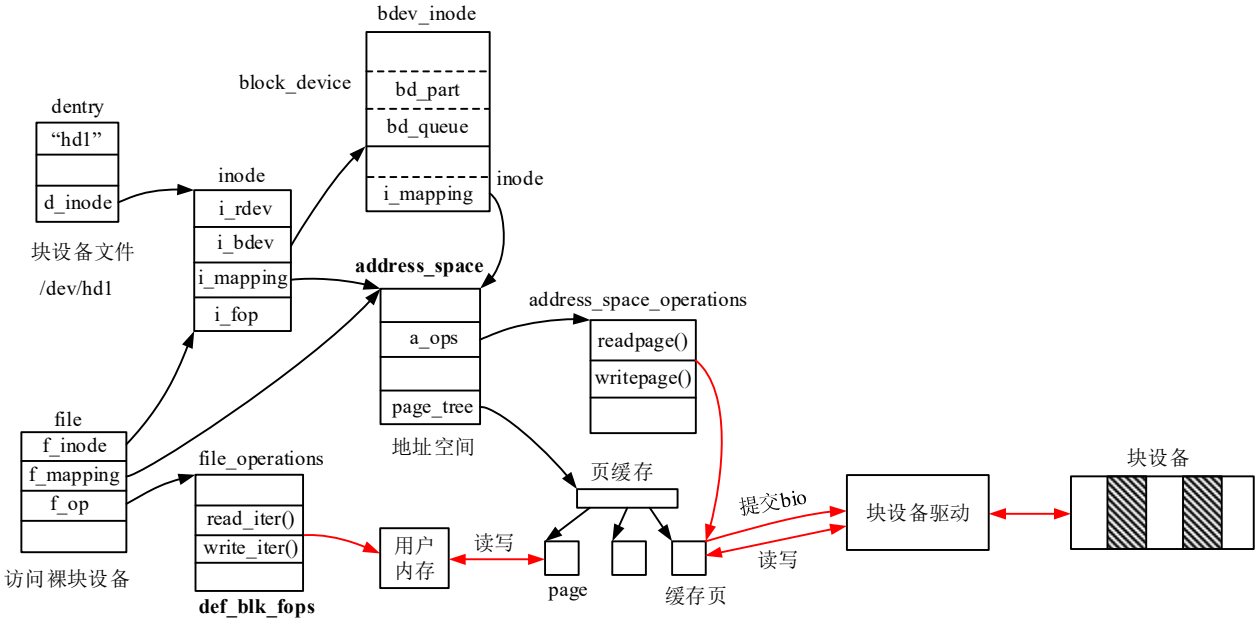
块缓存是以块为单位建立的块设备数据在内存中的缓存，这里的块是比页小的单位，一个页可划分成若干个块。块缓存可寄存在页缓存中，也可以单独存在。

11.1.1 前言

下面先简要介绍一下 VFS 访问文件内容的流程，以及文件内容与块设备数据块之间的映射，然后再介绍页缓存和块缓存。

1 VFS 访问块设备

VFS 层访问文件内容的流程如下图所示：



地址空间由 address_space 结构体表示，这里的地址空间指的是文件内容从头至尾的空间，而不是 CPU 访问内存的地址空间。

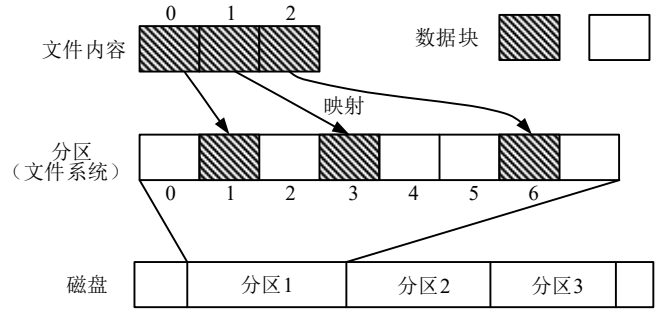
地址空间通过基数树管理着缓存页（page），因为文件内容在逻辑上是连续的，因此用基数树管理比较适合，管理缓存页的索引值按顺序编号，代表了连续的文件内容。

地址空间 address_space 关联地址空间操作结构 address_space_operations，其中包含读写缓存页的函数，即实现缓存页与块设备数据传输的函数。

用户访问文件内容时，实际是对缓存页内容的访问。读操作中，当缓存页中没有数据或无效时，将调用地址空间的读函数实现缓存页与块设备之间的数据传输。写操作中，先写入缓存页，而后调用地址空间的写函数将缓存页中数据写入块设备。

2 磁盘映射

文件内容在逻辑上是连续的，而文件内容在块设备中是离散保存的，如下图所示：



文件内容保存在块设备中哪些数据块，这是由具体文件系统类型决定的，也由文件系统类型实现文件内容逻辑数据块与块设备中数据块之间的映射。

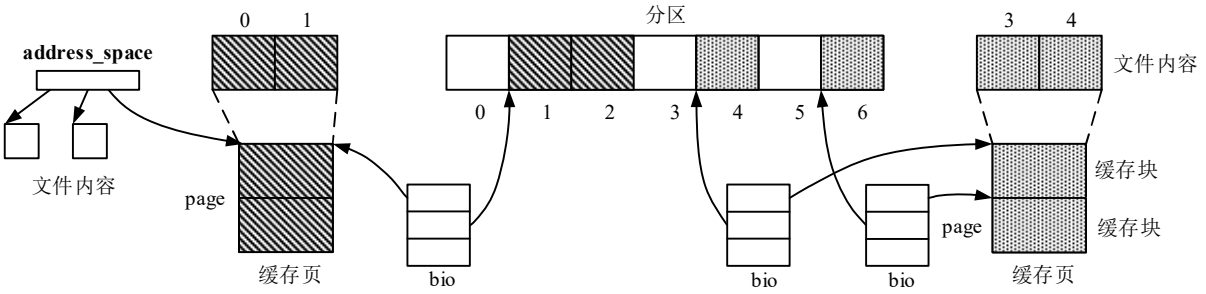
在地址空间中是以页为单位缓存文件内容的，页大小通常为 4KB、8KB、16KB 等，而文件系统中数据块的大小可能与页大小不同。如下图所示，文件系统中数据块大小为页大小的一半，当然也可能比页大。例如，格式化分区为 FAT 文件系统时，可选择数据块大小为 2KB、4KB、8KB 等。

缓存页映射的数据块在块设备可能是连续的，也可能是不连续的。当连续时，访问缓存页映射的数据块时，只需要构建一个 bio 实例即可，当数据块不连续时，则需要多个 bio 实例。

当缓存页对应内容在块设备中不连续时，需要将缓存页划分成多个缓存块，这就是块缓存，一个缓存页中包含若干个缓存块。那么缓存块的大小如何确定呢？

在挂载文件系统时，文件系统类型代码会设置 super_block 实例 **s_blocksize_bits** 成员，这表示内核看到的文件系统中数据块的大小，这个值最大为 PAGE_SIZE，也就是缓存页的大小，而不是格式化文件系统时设置的数据块大小。也就是说真实的文件系统中数据块大小可能大于一页，但是文件系统反映给内核时数据块大小不能超过一页。

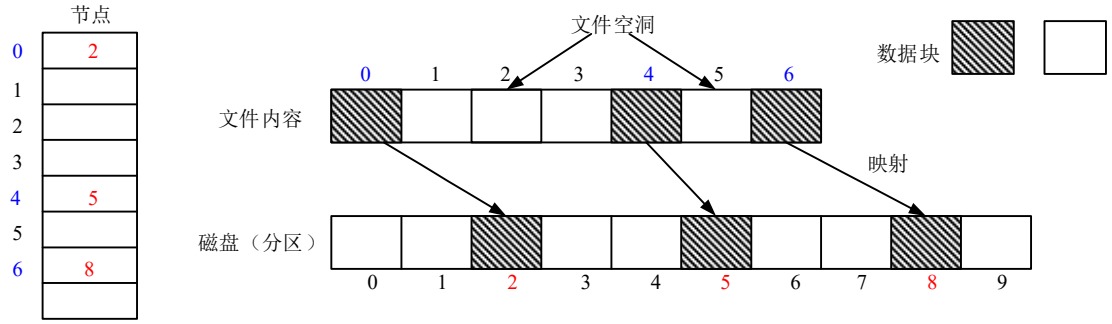
下图中文件系统数据块大小为缓存页的一半，示意了按缓存页和按缓存块访问块设备的情形：



下面来讨论一下复杂一点的文件内容与块设备中数据块之间的映射关系。

Linux 系统允许文件空洞，即在写文件时可定位到比现有文件内容大的位置，对其进行读写，即文件的开头结尾有数据而中间没有数据。如下图所示，文件内包含两个空洞区，空洞区可映射到块设备中数据块（数据清零），也可以不映射到块设备中数据块，这由具体文件系统类型决定。

例如，第 7 章介绍的 ext2 文件系统类型，其文件节点结构中包含一个数组，数组项的内容即映射的数据块号，如下图所示，可以只在建立了映射的数据块对应的数组项中填入映射的数据块号，而空洞区不用填入数据块号（不占用磁盘空间）。内核为空洞区在地址空间中创建缓存页后将用 0 字节填充，如果是写操作，对缓存页写入数据后将为其分配块设备数据块并建立映射，实现数据的写入块设备。



文件空洞是有用的，例如在下载文件时，可对文件多点进行下载，各点下载文件不同位置的内容，最后拼接成完整的文件。内核在读写文件为其创建缓存页，建立到块设备数据块之间的映射时需要考虑文件空洞的情况，未映射的空洞区在缓存页中用 0 填充。

11.1.2 页缓存

文件内容的页缓存由地址空间 address_space 结构体，通过基数树管理。内核中表示文件的 inode 结构体中内嵌了 address_space 结构体成员 **i_data**，**i_mapping** 成员指向 address_space 结构体。通常情况下，i_mapping 成员指向 i_data 成员，但也可以指向其它 address_space 实例，即指向其它文件的地址空间。

address_space 结构体中包含一个基数树用于管理缓存页 page 实例，还包含 **address_space_operations** 结构体指针成员 a_ops，address_space_operations 结构体表示地址空间操作，主要是对缓存页的读写操作，结构体实例需由具体文件系统类型定义。

下面介绍地址空间、地址空间操作结构的定义，以及地址空间中缓存页的操作。

1 数据结构

这里介绍的数据结构主要有 address_space 和 address_space_operations。

■地址空间

地址空间 address_space 结构体定义在/include/linux/fs.h 头文件：

```
struct address_space {
    struct inode            *host;          /*文件或块设备文件 inode 实例指针*/
    struct radix_tree_root  page_tree;      /*基数树根节点，管理缓存页 page 实例*/
    spinlock_t              tree_lock;      /*保护基数树的自旋锁*/
    atomic_t                i_mmap_writable;  /*虚拟内存空间共享映射次数 VM_SHARED */
    struct rb_root          i_mmap;         /*红黑树根节点，管理映射到本文件的虚拟内存域*/
    struct rw_semaphore     i_mmap_rwsem;   /* protect tree, count, list */

    /*以下成员由 tree_lock 提供保护*/
    unsigned long           nrpages;        /*地址空间内缓存页数量*/
    unsigned long           nrshadows;      /* number of shadow entries */
    pgoff_t                 writeback_index;  /*页缓存回写起始页索引值*/
    const struct address_space_operations  *a_ops;   /*地址空间操作结构指针*/
    unsigned long           flags;          /*标记成员*/
    spinlock_t              private_lock;   /* for use by the address_space */
    struct list_head        private_list;   /* ditto */
    void                    *private_data;  /* ditto */
```

} __attribute__((aligned(sizeof(long))));

address_space 结构体中主要成员简介如下：

●**host**：文件或块设备文件的 inode 实例指针。

●**page_tree**：基数树根节点，管理缓存页 page 实例，基数树结构详见第 1 章。

●**i_mmap**：红黑树根节点，管理映射到本文件内容的虚拟内存域 vm_area_struct 实例，见第 4 章。

●**a_ops**：地址空间操作结构 address_space_operations 指针，结构体中包含读写缓存页等函数指针，见下文。

●**flags**：标记成员，高位表示错误标记，低位表示分配缓存页的分配掩码，即伙伴系统中的分配掩码。错误码标记位定义在/include/linux/pagemap.h 头文件：

```
enum mapping_flags {                    /*__GFP_BITS_SHIFT 表示分配掩码最高位*/
    AS_EIO        = __GFP_BITS_SHIFT + 0,    /* IO error on async write  异步写 IO 错误*/
    AS_ENOSPC = __GFP_BITS_SHIFT + 1,    /* ENOSPC on async write */
    AS_MM_ALL_LOCKS = __GFP_BITS_SHIFT + 2,    /* under mm_take_all_locks() */
    AS_UNEVICTABLE    = __GFP_BITS_SHIFT + 3,    /* e.g., ramdisk, SHM_LOCK */
    AS_EXITING = __GFP_BITS_SHIFT + 4,    /* final truncate in progress */
};
```

■地址空间操作结构

地址空间操作结构 address_space_operations 定义在/include/linux/fs.h 头文件：

```
struct address_space_operations {
    int    (*writepage)    (struct page *page, struct writeback_control *wbc);    /*写缓存页内容至块设备*/
    int    (*readpage)     (struct file *, struct page *);                        /*从块设备读数据至缓存页*/

    int    (*writepages)    (struct address_space *, struct writeback_control *); /*回写地址空间脏页*/
    int    (*set_page_dirty)    (struct page *page);          /*设置缓存页脏标记，设置返回 true*/

    int (*readpages)(struct file *filp, struct address_space *mapping,struct list_head *pages, \
                                                    unsigned nr_pages);     /*从块设备读多页*/
    int (*write_begin)(struct file *, struct address_space *mapping,loff_t pos, unsigned len, unsigned flags,
                                                        struct page **pagep, void **fsdata);
    int (*write_end) (struct file *, struct address_space *mapping,loff_t pos, unsigned len, unsigned copied,
                                                        struct page *page, void *fsdata);
                                                /*用户内存数据开始/结束写入缓存页时调用的函数*/
    /* Unfortunately this kludge is needed for FIBMAP. Don't use it */
    sector_t    (*bmap)    (struct address_space *, sector_t);      /*返回数据块的扇区号，用于交换文件*/
    void    (*invalidatepage)    (struct page *, unsigned int, unsigned int);
    int    (*releasepage)    (struct page *, gfp_t);       /*释放缓存页*/
    void    (*freepage)    (struct page *);
    ssize_t    (*direct_IO)    (struct kiocb *, struct iov_iter *iter, loff_t offset);
                                                        /*直接读写函数，跳过页缓存*/

    int    (*migratepage)    (struct address_space *,struct page *, struct page *, enum migrate_mode);
                                                        /*移动缓存页中数据*/
    int    (*launder_page)    (struct page *);
    int    (*is_partially_uptodate)    (struct page *, unsigned long,unsigned long);
```

```
    void    (*is_dirty_writeback)    (struct page *, bool *, bool *);
    int    (*error_remove_page)    (struct address_space *, struct page *);

    /*交换文件需要的函数*/
    int    (*swap_activate)    (struct swap_info_struct *sis, struct file *file,sector_t *span); /*启用交换区*/
    void    (*swap_deactivate)    (struct file *file);
};
```

address_space_operations 结构体中主要函数指针成员说明如下：

●**writepage**：将单个缓存页数据写入到块设备，数据回写（同步）时调用，通常是按缓存块进行回写。

●**readpage**：从块设备中读入数据至缓存页，读文件时调用，通常是按缓存块进行读取。

●**writepages**：地址空间（缓存页）同步函数，通过 writeback_control 参数可控制同步地址空间中某段区域或整个地址空间（缓存页）。

●**readpages**：从块设备中读数据至多个缓存页（读多页数据），通常在文件预读操作中调用。

●**write_begin**：用户内存数据复制到缓存页之前调用此函数，并不是将缓存页写入块设备前调用，用于创建/获取缓存页等，见后面写缓存页函数。

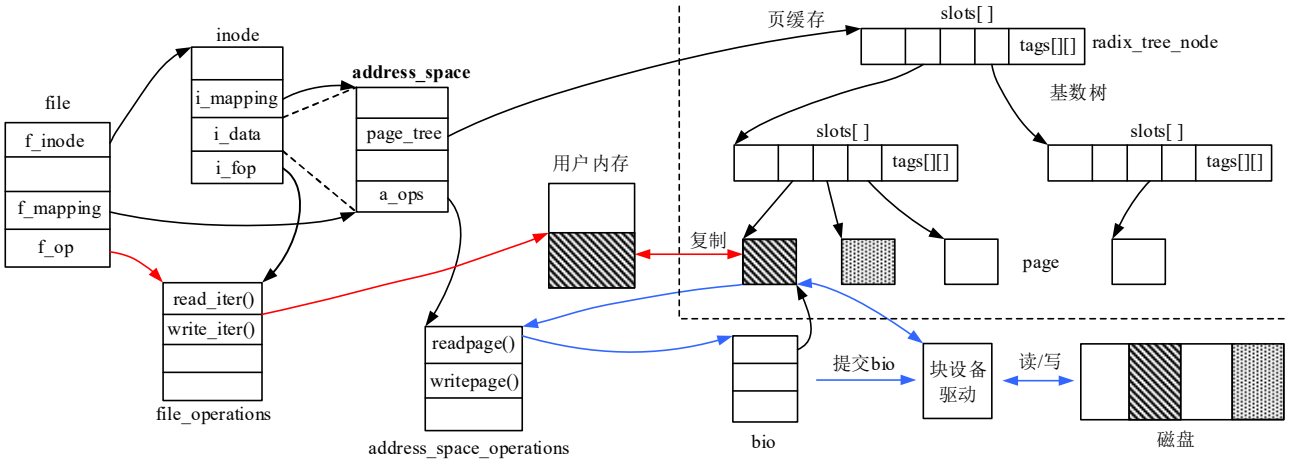
●**write_end**：用户内存数据复制到缓存页之后调用此函数，并不是将缓存页写入块设备后调用，见后面写缓存页函数。

●**direct_IO**：直接读写操作函数，不经过页缓存，直接从块设备中读入数据至用户内存，或将用户内存数据写入块设备。在读写文件内容的系统调用中调用（打开文件需设置 O_DIRECT 标记位）。

address_space_operations 实例需由具体文件系统类型的实现，内核在打开文件创建 inode 实例时，由文件系统类型相关代码对 inode 指向地址空间实例中的 **a_ops** 指针成员赋值。

用户读写文件内容时，是对页缓存的读写，需要时调用 address_space_operations 实例中的读写缓存页函数，实现缓存页与块设备之间的数据传输，如下图所示。

address_space_operations 实例中的读写缓存页函数，需要将缓存页转换成块设备中的数据块，从而构建 bio 实例，提交到块设备驱动程序，由其实现数据传输。



2 缓存页操作

缓存页在地址空间中由基数树管理，基数树叶节点为缓存页的 page 实例。叶节点的索引值即缓存页在文件内容中的页偏移量。例如，索引值为 0 的缓存页保存文件最开始一页（第 0 页）的内容，索引值为 1 的缓存页保存文件内容第 1 页的内容，依此类推。

页缓存中一般不会缓存文件的全部内容，而只是一部分，否则再多的内存也不够用。在用户对文件进行读写操作时，按需缓存适量的文件内容。

地址空间对缓存页的管理包括查找缓存页、查找/分配缓存页、插入缓存页、修改基数树标签标记值等。

■缓存页标记

管理页缓存的基数树定义了三种类型的检签，如下所示（/include/linux/fs.h）：

```
#define   PAGECACHE_TAG_DIRTY        0    /*缓存页脏标记类型*/
#define   PAGECACHE_TAG_WRITEBACK    1    /*缓存页正在回写标记类型*/
#define   PAGECACHE_TAG_TOWRITE      2    /*缓存页即将写出标记类型*/
```

下列函数分别用于设置或清除指定索引值缓存页某一类型标签的标记值：

●radix_tree_tag_set(struct radix_tree_root *root,unsigned long index, unsigned int tag)：设置 index 索引值缓存页的 tag 类型标签标记值。

●radix_tree_tag_clear(struct radix_tree_root *root,unsigned long index, unsigned int tag)：清除 index 索引值缓存页的 tag 类型标签标记值。

■查找/分配缓存页

对文件内容进行读写操作前，需要查找文件内容对应的缓存页是否已经在基数树中，如果在则可直接对缓存页进行读写，如果不在则需要从伙伴系统分配缓存页，并将其插入到页缓存基数树和物理内存域文件缓存页 LRU 链表后，才能对缓存页进行读写。

查找和分配缓存页的接口函数如下，以下分配函数分配的缓存页并没有插入基数树和 LRU 链表：

●struct page ***find_get_page**(struct address_space *mapping,pgoff_t offset)：从页缓存中查找 offset 索引值的缓存页，查找成功返回其 page 实例指针，失败返回 NULL。（/include/linux/pagemap.h）

●struct page *find_get_entry(struct address_space *mapping, pgoff_t offset)：从页缓存中查找 offset 索引值的缓存页，查找成功返回其 page 实例指针，失败返回 NULL。（/mm/filemap.c）

●struct page *__page_cache_alloc(gfp_t gfp)：从伙伴系统分配缓存页的函数。

●struct page ***page_cache_alloc_readahead**(struct address_space *x)：预读操作中分配缓存页的接口函数，它是__page_cache_alloc(gfp_t gfp)函数的包装器。（/include/linux/pagemap.h）

●page_cache_release(page)：释放缓存页，等同 put_page(page)。（/include/linux/pagemap.h）

■插入缓存页

在分配缓存页之后需要将其插入到地址空间基数树和文件缓存页 LRU 链表，插入缓存页的接口函数如下（/mm/filemap.c）：

```
int add_to_page_cache_lru(struct page *page, struct address_space *mapping,   \
                                                        pgoff_t offset, gfp_t gfp_mask)
/*
*page：指向插入页面 page 实例，mapping：指向地址空间实例，
*offset：缓存页在文件内容中的页偏移量，gfp_mask：分配掩码。
*/
{
    void *shadow = NULL;
    int ret;

    __set_page_locked(page);        /*设置页面锁定标记位*/
    ret = __add_to_page_cache_locked(page, mapping, offset,gfp_mask, &shadow);
                    /*将 page 添加到基数树，成功返回 0，否则返回错误码，/mm/filemap.c*/
```

```
    if (unlikely(ret))      /*添加失败，清零锁定标记位*/
        __clear_page_locked(page);
    else {                  /*缓存页添加成功*/
        if (shadow && workingset_refault(shadow)) {
            SetPageActive(page);        /*清除 page 实例 PG_active 标记位*/
            workingset_activation(page);
        } else
            ClearPageActive(page);      /*清除 page 实例 PG_active 标记*/
        lru_cache_add(page);   /*将缓存页 page 添加到不活跃页 LRU 链表，/mm/swap.c*/
    }
    return ret;
}
```

add_to_page_cache_lru()函数内调用__add_to_page_cache_locked()函数将缓存页 page 实例添加到地址空间基数树中，并设置 **page->mapping = mapping** 和 **page->index = offset**，随后调用 lru_cache_add(page)函数将 page 实例添加到物理内存域中不活跃页 LRU 链表，便于对页面进行回收。lru_cache_add(page)函数详见本章下文。

■查找或创建缓存页

内核还定义了同时完成查找、分配和插入缓存页的接口函数 **find_or_create_page()**，函数主要由文件系统类型代码调用。此函数首先在页缓存中查找指定索引值的缓存页，如果查找成功则返回缓存页 page 实例指针，如果不成功则分配并插入缓存页，函数返回 page 实例指针。

find_or_create_page()函数定义如下（/include/linux/pagemap.h）：

```
static inline struct page *find_or_create_page(struct address_space *mapping,pgoff_t offset, gfp_t gfp_mask)
/*offset：缓存页索引值*/
{
    return pagecache_get_page(mapping, offset,    \
                FGP_LOCK|FGP_ACCESSED|FGP_CREAT,gfp_mask);    /*/mm/filemap.c*/
}
```

pagecache_get_page()函数是一个比较重要的函数，它是许多接口函数调用的函数，函数可完成缓存页的查找、分配/插入/设置标记位等操作。

pagecache_get_page()函数流程由参数 fgp_flags 控制，参数取值定义如下（/include/linux/pagemap.h\）：

```
#define   FGP_ACCESSED  0x00000001      /*设置分配缓存页调用 mark_page_accessed()函数*/
#define   FGP_LOCK      0x00000002      /*锁定分配的缓存页*/
#define   FGP_CREAT     0x00000004      /*未查找到缓存页则分配（创建）缓存页*/
#define   FGP_WRITE     0x00000008
#define   FGP_NOFS      0x00000010
#define   FGP_NOWAIT    0x00000020
```

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

```
struct page *pagecache_get_page(struct address_space *mapping, pgoff_t offset, \
                                                    int fgp_flags, gfp_t gfp_mask)
{
    struct page *page;
```

```
repeat:
    page = find_get_entry(mapping, offset);     /*查找缓存页*/
    if (radix_tree_exceptional_entry(page))
        page = NULL;
    if (!page)
        goto no_page;        /*没找到，跳至 no_page 处分配缓存页*/

    if (fgp_flags & FGP_LOCK) {          /*需要锁定页*/
        if (fgp_flags & FGP_NOWAIT) {
            if (!trylock_page(page)) {
                page_cache_release(page);
                return NULL;
            }
        } else {
            lock_page(page);
        }

        /* Has the page been truncated? */
        if (unlikely(page->mapping != mapping)) {        /*page 不关联到 mapping 地址空间了，释放*/
            unlock_page(page);
            page_cache_release(page);
            goto repeat;
        }
        VM_BUG_ON_PAGE(page->index != offset, page);
    }

    if (page && (fgp_flags & FGP_ACCESSED))
        mark_page_accessed(page);       /*设置缓存页访问标记位*/

no_page:
    if (!page && (fgp_flags & FGP_CREAT)) {     /*缓存页不存在，需要分配*/
        int err;
        if ((fgp_flags & FGP_WRITE) && mapping_cap_account_dirty(mapping))
            gfp_mask |= __GFP_WRITE;
        if (fgp_flags & FGP_NOFS)
            gfp_mask &= ~__GFP_FS;

        page = __page_cache_alloc(gfp_mask); /*从伙伴系统分配缓存页，/include/linux/pagemap.h*/
        if (!page)
            return NULL;

        if (WARN_ON_ONCE(!(fgp_flags & FGP_LOCK)))
            fgp_flags |= FGP_LOCK;

        /* Init accessed so avoid atomic mark_page_accessed later */
        if (fgp_flags & FGP_ACCESSED)
```

```
                __SetPageReferenced(page);

            err = add_to_page_cache_lru(page, mapping, offset,gfp_mask & GFP_RECLAIM_MASK);
                                        /*将缓存页插入地址空间和物理内存域页面 LRU 链表，见上文*/
            ...
        }
        return page;     /*返回缓存页 page 实例指针*/
    }
```

前面介绍的查找缓存页的 find_get_page()函数内部调用就是 pagecache_get_page()函数，标记参数设为 0，即查找不成功时不创建缓存页。

grab_cache_page_write_begin()函数内也是调用 pagecache_get_page()函数查找和创建缓存页，主要用于文件系统类型地址空间操作结构的 write_begin()函数，在进程写文件操作前在页缓存中查找或创建缓存页。

11.1.3 块缓存

块缓存是在内存中以缓存块为大小建立的块设备中数据的缓存，块缓存中缓存块的大小不是固定的，因具体的设备和文件系统而不同，后面将详细介绍。

块缓存组织结构有两种形式，一种是独立的块缓存，寄存在块设备文件地址空间缓存页中；另一种是寄生在普通文件缓存页的块缓存，即将文件缓存页划分成多个缓存块，以便对文件内容进行更细粒度的控制。独立的块缓存实际上也是与缓存页关联的，只不过它寄存的是块设备文件中的缓存页，而不是普通文件的缓存页。

以前版本的内核（2.5 版本前）以缓存块为单位向块设备发起读写操作，现在的内核是以 bio 实例为单位向块设备发起读写操作的，如今块缓存的重要性有所下降。缓存块中数据与块设备中数据的传输通过接口函数 **submit_bh(rw, bh)**完成，函数中由缓存块构建 bio 结构体实例并提交到块设备请求队列，完成数据传输。

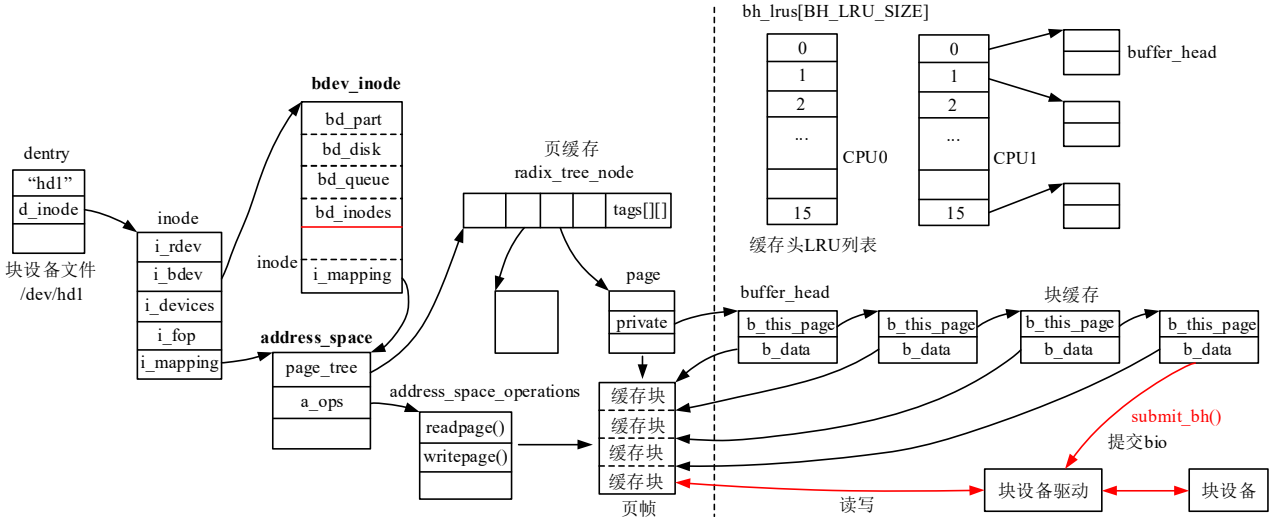
1 概述

独立的块缓存寄生在块设备文件的缓存页中，如下图所示。内核在打开块设备文件或挂载文件系统（分区）时，为块设备创建 bdev_inode 结构体实例，其中内嵌 inode 结构体成员，inode 成员中地址空间实例中缓存了块设备数据。

块设备文件中缓存页按一定的大小划分成若干个缓存块（图中为 4 个），每个缓存块由 buffer_head 块缓存头结构体表示。buffer_head 结构体中包含缓存块内存起始地址、长度以及映射的块设备数据块等信息。

缓存页 page 实例中 private 成员指其包含的缓存块的 buffer_head 实例链表。除此之外，内核为系统内每个 CPU 核维护了一个块缓存头 buffer_head 实例的 LRU 列表（指针数组），以便快速地查找缓存头实例。LRU 列表的长度（数组项数）是固定的，为 BH_LRU_SIZE（16），最近创建或使用的块缓存头将排在 LRU 列表的头部，其余顺序往下移，超出列表的对象将被挤出列表。

在对块设备中指定数据块进行操作时，内核首先在 LRU 列表中查找对应的块缓存头实例，找到则返回实例指针。若未找到，再到块设备文件页缓存中查找缓存块所在的缓存页是否存在，如果存在则根据 page 实例 private 成员查找对应的块缓存头实例，并返回实例指针。如果对应缓存页不存在，则需要在页缓存中创建对应的缓存页，并创建对应的块缓存头实例链表，返回所需缓存块缓存头实例指针。最后如果缓存块中数据无效，则调用 submit_bh(rw, bh)函数完成数据的传输。



普通文件缓存页中寄生的块缓存与块设备文件缓存页中寄生的块缓存结构，以及操作方式都是一样的。不同之处在于，块设备文件内容与块设备中数据块是线性映射的关系，将缓存页或缓存块转换成块设备中数据块比较简单，而将普通文件的缓存页或缓存块转换成块设备中数据块时，需要由文件系统类型代码进行转换，因为普通文件内容在块设备中是分散保存的，不是线性映射。

■buffer_head

块缓存头 buffer_head 结构体定义如下（/include/linux/buffer_head.h）：

```
struct buffer_head {
    unsigned long   b_state;            /*缓存块状态*/
    struct buffer_head *b_this_page;    /*指向在同一缓存页中的下一个缓存块*/
    struct page *b_page;                /*指向寄生的缓存页 page 实例*/

    sector_t    b_blocknr;          /*映射到块设备的起始数据块号（在整个块设备中的编号）*/
    size_t      b_size;     /*映射数据长度（字节数），可大于一个缓存块，如映射到连续的数据块*/
    char        *b_data;            /*缓存块内存起始地址*/

    struct block_device    *b_bdev;         /*指向 block_device 实例*/
    bh_end_io_t *b_end_io;              /*bio 实例数据传输完成的回调函数*/
    void *b_private;                    /*回调函数 b_end_io()私有数据指针（参数）*/
    struct list_head    b_assoc_buffers;    /*添加到地址空间 private_list 双链表*/
    struct address_space *b_assoc_map;  /*指向地址空间*/
    atomic_t b_count;                   /*块缓存头引用计数*/
};
```

buffer_head 结构体主要成员简介如下：

●**b_this_page**：块缓存头 buffer_head 结构体指针，指向同一缓存页中的下一个缓存头实例。

●**b_blocknr**：缓存块对应块设备上的数据块号，通常是按文件系统中的数据块大小划分所得的编号，这个编号是在整个块设备内部的编号，而不只是在文件系统（分区）内部的编号。

●**b_size**：映射块设备中连续数据块的长度，可大于一个缓存块，字节数。

●**b_data**：缓存块在内存中起始地址。

●**b_bdev**：块设备 block_device 结构体指针。

●**b_end_io**：bh_end_io_t 类型函数指针，bio 实例数据传输完成的回调函数，函数原型定义如下：

typedef void (bh_end_io_t)(struct buffer_head *bh, int uptodate);

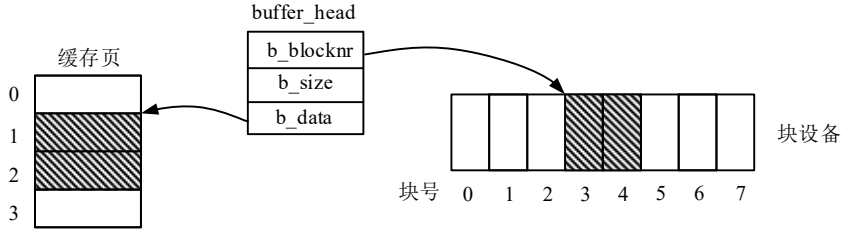
●**b_private**：提供给 b_end_io()函数使用的私有数据指针。
●**b_bdev**：指向 block_device 实例，其 bd_block_size 成员为缓存块（数据块）大小。

●**b_state**：缓存块状态位图，位图各位语义定义如下（/include/linux/buffer_head.h）：

```
enum bh_state_bits {
    BH_Uptodate,          /*缓存块数据有效标记位，bit0*/
    BH_Dirty,             /*缓存块数据脏标记位，未同步到块设备，bit1，后面的依此类推*/
    BH_Lock,              /*缓存块被锁住*/
    BH_Req,               /*已经向块设备提交 bio */
    BH_Uptodate_Lock,     /*是缓存页关联的第一个块缓存头*/

    BH_Mapped,          /*缓存块已建立到块设备数据块的映射*/
    BH_New,             /*刚由 get_block ()函数获取的新映射*/
    BH_Async_Read,      /* Is under end_buffer_async_read I/O（bio 完成的回调函数）*/
    BH_Async_Write,     /* Is under end_buffer_async_write I/O （bio 完成的回调函数）*/
    BH_Delay,           /*缓存块在块设备中尚未分配映射的数据块*/
    BH_Boundary,        /*本缓存块之后的映射不连续*/
    BH_Write_EIO,       /*写错误*/
    BH_Unwritten,       /*缓存块在块设备中分配了数据块，但尚未写（写操作时使用）*/
    BH_Quiet,           /* Buffer Error Prinks to be quiet */
    BH_Meta,            /* Buffer contains metadata，缓存块包含元数据*/
    BH_Prio,            /* Buffer should be submitted with REQ_PRIO，提交时需带 REQ_PRIO 标记*/
    BH_Defer_Completion,  /*由工作队列执行异步 IO 操作*/

    BH_PrivateStart ,       /*第一个私有标记位*/
};
```

内核在/include/linux/buffer_head.h 头文件定义了标记位操作函数，例如：
void set_buffer_***name***(struct buffer_head *bh)：置位标记位，name 为标记位名称（小写），如 mapped。
void clear_buffer_***name***(struct buffer_head *bh)：清零标记位。
int test_set_buffer_***name***(struct buffer_head *bh)：测试并设置标记位。
int test_clear_buffer_***name***(struct buffer_head *bh)：测试并清零标记位。
int buffer_***name***(const struct buffer_head *bh)：测试标记位值。

在以前的内核版本中，buffer_head 结构体表示一个单一的缓存块，现在可以表示缓存页中映射到块设备中连续数据块的多个缓存块，如下图所示。



上图中缓存页中第 1 和第 2 个缓存块映射到块设备中连续的数据块 3 和 4，buffer_head 实例可表示两个连续缓存块的映射关系，b_blocknr 保存了第 1 个缓存块映射的数据块号，b_size 表示映射大小（长度），此处为 2 个数据块的大小，b_data 指向缓存页中第 1 个缓存块的起始地址。

文件系统类型代码中定义的 get_block_t()类型函数用于提取从指定文件内容逻辑数据块开始映射到块设备中多个连续数据块的信息。

■初始化

内核为 buffer_head 结构体创建了 slab 缓存，初始化函数为 buffer_init()，定义如下（/fs/buffer.c）：

```
void __init buffer_init(void)        /*由启动内核函数 start_kernel()调用*/
{
    unsigned long nrpages;

    bh_cachep = kmem_cache_create("buffer_head",
            sizeof(struct buffer_head), 0,
                (SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|
                SLAB_MEM_SPREAD),
                NULL);          /*创建 buffer_head 结构 slab 缓存*/

    /*限制缓存头 slab 缓存占用的内存不超过内存数量的 10%*/
    nrpages = (nr_free_buffer_pages() * 10) / 100;
    max_buffer_heads = nrpages * (PAGE_SIZE / sizeof(struct buffer_head));
    hotcpu_notifier(buffer_cpu_notify, 0); /*注册 CPU 核通知链通知，通知用于清空 bh_lrus[]列表*/
}
```

struct buffer_head ***alloc_buffer_head**(gfp_t gfp_flags)函数用于从 slab 缓存中分配 buffer_head 实例。

内核为 CPU 核定义的块缓存头 LRU 列表如下（/fs/buffer.c）：

```
struct bh_lru {
    struct buffer_head *bhs[BH_LRU_SIZE];        /*指针数组，BH_LRU_SIZE 为 16*/
};

static DEFINE_PER_CPU(struct bh_lru, bh_lrus) = {{ NULL }};          /*percpu 变量*/
```

■设置缓存块大小

这里我们要搞清楚几个数据块大小的含义。

块设备是按块大小访问的设备，这个块大小称它为物理块大小，例如，磁盘通常是 1 个扇区（512 字节），NAND Flash 是 2KB、4KB 等。这个块大小是指最小的访问单位，可以一次访问多个连续的物理块。

在将分区格式化成文件系统时，会设置一个文件系统数据块大小，这是文件系统保存文件内容的最小单位。例如：格式化为 FAT32 文件系统时，数据块大不可设为 4KB、8KB 等。文件系统数据块应包含若干个连续的物理块（包含整数个连续的物理块）。

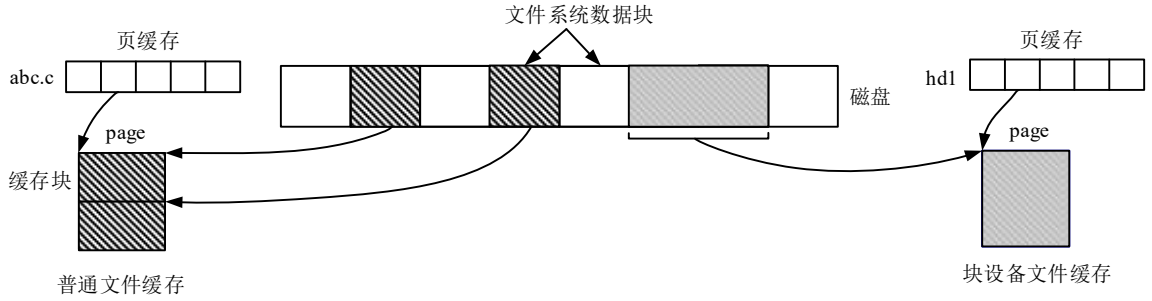
在内核中是按页大小缓存文件内容的，页大小由内核配置选项决定，通常是 4KB、8KB 等。

如果文件系统数据块大小不小于内核页大小，也就是文件系统数据块大小是页的整数倍，那么缓存页就不需要划分成缓存块了，因为缓存页已经是访问文件系统的最小单位了。

如果文件系统数据块大小小于页，也就是说一个缓存页映射多个文件系统数据块时，如果缓存页映射的文件系统数据块在块设备中是不连续的，就要将缓存页按文件系统数据块进行划分了，也就是划分成缓存块进行访问。如果映射的文件系统数据块是连续的，则可以不划分，可一次访问。

如下图所示，当访问裸块设备时（块设备文件），将磁盘（分区）数据块大小（文件系统数据块）设

为内存页（缓存页）大小，按页访问，以提高效率。



普通文件系统中的数据块大小在格式化文件系统时设置，数据块大小写入文件系统超级块中。格式化文件系统时设置的数据块大小可能大于页，但是文件系统反映给内核时，文件系统中数据块大小不能大于一页。也就是说，从内核角度看，文件系统中数据块大小最大为一页，内核最大是按页访问块设备的。一个缓存页可以包含若干个文件系统数据块（缓存块）。

如果实际文件系统数据块大小大于一页（为整数个页），内核仍将按页访问文件内容，缓存页与文件系统数据块的映射关系由文件系统类型代码确定。

内核在挂载文件系统时，在创建超级块 super_block 实例后，文件系统类型填充超级块实例的函数将数据块大小设置到超级块 super_block 实例 **s_blocksize_bits** 成员，通用函数 sb_set_blocksize(s, size)用于完成此项工作。

例如，ext2 文件系统类型挂载函数在创建超级块实例后，将调用 sb_set_blocksize(s, size)函数设置文件系统数据块大小至 s_blocksize_bits 成员。

sb_set_blocksize(s, size)函数定义在/fs/block_dev.c 文件内，代码如下：

```
int sb_set_blocksize(struct super_block *sb, int size)
/*size：大小为 512 至 PAGE_SIZE，且应当是 2 的整数次方*/
{
    if (set_blocksize(sb->s_bdev, size))     /*设置 block_device 实例数据块大小成员，/fs/block_dev.c*/
        return 0;                            /*成功返回 0*/

    sb->s_blocksize = size;       /*设置 super_block 中数据块大小成员*/
    sb->s_blocksize_bits = blksize_bits(size);
    return sb->s_blocksize;          /*返回数据块大小*/
}

set_blocksize(sb->s_bdev, size)函数用于将数据块大小设置到 block_device 实例中，函数定义如下：
int set_blocksize(struct block_device *bdev, int size)
{
    /*size 必须不小于 512，不大于 PAGE_SIZE，且是 2 的整数次方*/
    if (size > PAGE_SIZE || size < 512 || !is_power_of_2(size))
        return -EINVAL;

    /*文件系统数据块大小不能小于请求队列中逻辑数据块大小*/
    if (size < bdev_logical_block_size(bdev))
                                /*请求队列 q->limits.logical_block_size，/include/linux/blkdev.h*/
        return -EINVAL;

    /*size 值设置到 block_device 实例*/
    if (bdev->bd_block_size != size) {
```

```
            sync_blockdev(bdev);
            bdev->bd_block_size = size;                /*重设 bd_block_size 成员值*/
            bdev->bd_inode->i_blkbits = blksize_bits(size);
            kill_bdev(bdev);
        }
        return 0;       /*成功返回 0*/
    }
```

block_device 实例 **bd_block_size** 成员值初始值为内存页大小，即 PAGE_SIZE。内核通过 bdev 伪文件系统创建 bdev_inode 实例时，block_device 实例 bd_block_size 成员值来源于 bdev_inode.vfs_inode->i_blkbits 成员值。而 bdev_inode.vfs_inode->i_blkbits 成员值，在创建 bdev_inode 实例时来源于 bdev 伪文件系统超级块实例的 sb->s_blocksize_bits 成员值。

在挂载 bdev 伪文件系统时，其超级块 s_blocksize_bits 成员赋值为 PAGE_SHIFT，即 bdev->bd_block_size 成员值初始值为 PAGE_SIZE，即内存页大小。

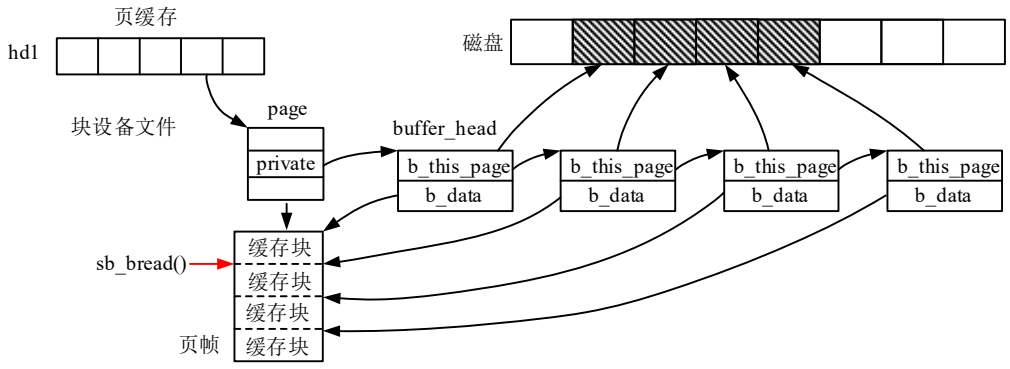
另外，传递给内核的文件系统数据块大小不能小于请求队列 q->limits.logical_block_size 值，后者可视为块设备驱动程序访问块设备的最小单位，通常设为物理块大小。

内核在打开文件为其创建 inode 实例时，在初始化 inode 实例的 inode_init_always()函数中将 **i_blkbits** 成员值设为 sb->s_blocksize_bits 成员值。如果需要将文件内容缓存页划分成缓存块时，缓存块大小值取决于 **inode.i_blkbits** 成员值，也就是说缓存块大小等于超级块中设置的文件系统数据块大小。

2 独立的块缓存

正常情况下，用户通过块设备文件访问裸块设备时，内核是以缓存页为单位访问块设备的。但是，块设备中包含一些分区、文件系统的元数据，如启动扇区、超级块、节点等。这些数据比较小，访问时并不需要按页来访问。

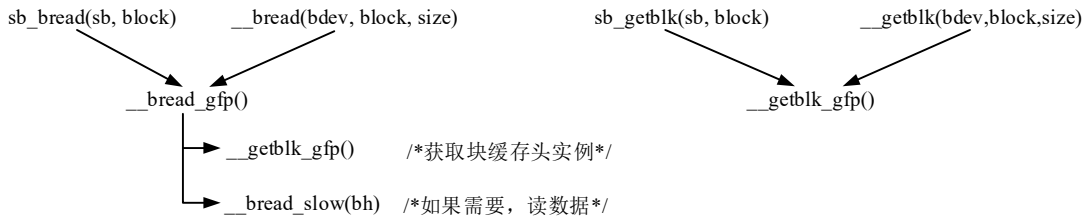
因此，内核为块设备提供了独立的块缓存，如下图所示：



独立的块缓存就是可以指定缓存块的大小，按此大小划分块设备，并访问指定的缓存块。通过独立块缓存访问块设备时，也是要先建立缓存页，然后对缓存页进行划分，最后只访问缓存页中的指定的缓存块。简单地说就是强行将缓存页划分成缓存块，按缓存块访问块设备，这一操作只有内核能进行。文件系统类型的代码会通过独立块缓存访问文件系统的元数据，如超级块、节点等。

下图示意了内核提供的按缓存块读取块设备中数据的主要接口函数，sb_bread()和 sb_getblk()函数用于读取文件系统超级块数据，__bread()和__getblk()函数用于读取指定数据块的数据，这些函数执行成功都将返回块缓存头 buffer_head 实例指针。sb_bread()和__bread()函数保证读取数据的有效性（与块设备中数据同步），sb_getblk()和__getblk()函数不保证数据的有效性（不一定与块设备中数据一致），这些函数定义

在/fs/buffer.c 文件内或/include/linux/buffer_head.h 头文件。



■按缓存块读

下面以 **sb_bread()**函数为例，说明如何读取块设备中指定数据块（缓存块）中数据，这是同步读函数。

sb_bread()函数定义如下（/include/linux/buffer_head.h）：

```
static inline struct buffer_head *sb_bread(struct super_block *sb, sector_t block)
/*sb：指向超级块，block：数据块编号，数据块大小为超级块中指定的数据块大小*/
{
    return __bread_gfp(sb->s_bdev, block, sb->s_blocksize, __GFP_MOVABLE);
}
```

__bread_gfp()函数是一个公共的函数，代码如下（/fs/buffer.c）：

```
struct buffer_head *__bread_gfp(struct block_device *bdev, sector_t block,unsigned size, gfp_t gfp)
/*
*bdev：block_device 指针，block：数据块编号，size：指定数据块大小，字节数，gfp：分配掩码*/
*读取块设备中按 size 大小划分的第 block 个数据块的数据，返回 buffer_head 实例指针。
*/
{
    struct buffer_head *bh = __getblk_gfp(bdev, block, size, gfp);   /*获取缓存块，/fs/buffer.c*/

    if (likely(bh) && !buffer_uptodate(bh))
        bh = __bread_slow(bh);          /*同步缓存块中数据，/fs/buffer.c*/
    return bh;
}
```

__bread_gfp()函数的参数 bdev 指定了块设备的 block_device 实例，block 表示读块设备中第几个缓存块，size 表示将块设备按 size 大小进行划分，划分缓存块，gfp 是分配掩码。

__bread_gfp()函数主要分两步，如下：

（1）调用__getblk_gfp()函数查找或创建缓存块 buffer_head 实例。

（2）如果缓存块数据无效，则调用__bread_slow(bh)函数从块设备中读数据至缓存块。

__bread_gfp()函数最后返回 buffer_head 实例指针 bh，bh->data 指向读取到的缓存块数据。

sb_getblk()和__getblk()接口函数只调用__getblk_gfp()函数，从现有缓存块中查找或创建缓存块，但并不从块设备中读取数据，也就是说只保证缓存块的存在，不保证数据的有效性。

__bread_gfp()函数调用关系如下图所示：

__getblk_gfp()函数首先调用__find_get_block()函数在本地 CPU 核块缓存头 LRU 列表中查找是否有匹配的块缓存头实例，如果有则直接返回缓存头实例指针。如果没有找到，则再到块设备文件的页缓存中查找缓存块所在的缓存页是否存在，存在则调用 bh_lru_install(bh)函数将对应的块缓存头插入到 LRU 列表头部，函数返回缓存头实例。

如果在 LRU 列表和页缓存中都没有查找到需要的块缓存头，则调用__getblk_slow()函数创建（或查找）缓存块所在的缓存页，并为其分配块缓存头实例链表，而后再查找块缓存头实例。

__bread_gfp()函数在获取块缓存头实例后，如果缓存块中数据无效，将调用__bread_slow(bh)函数，从块设备中读取数据至缓存块，读数据的函数为 **submit_bh(READ, bh)**，详见下文。

●创建缓存页

我们先看一下 **grow_dev_page()**函数的实现，它用于在块设备文件中创建欲读取缓存块所在的缓存页，以及为其创建块缓存头 buffer_head 实例链表并初始化。__getblk_gfp()函数调用的其它函数请读者自行阅读源代码。

grow_dev_page()函数定义如下（/fs/buffer.c）：

```
static int grow_dev_page(struct block_device *bdev, sector_t block,pgoff_t index, int size,   \
                                                                    int sizebits, gfp_t gfp)
/*
*bdev：block_device 实例指针，block：数据块编号，index：缓存块所在缓存页索引值，
*size：缓存块（数据块）大小，字节数，
*sizebits：缓存块大小字节数以 2 为底取对数，  gfp：分配缓存页的分配掩码。
*/
{
    struct inode *inode = bdev->bd_inode;   /*bdev_inode.vfs_inode*/
    struct page *page;
    struct buffer_head *bh;
    sector_t end_block;
    int ret = 0;            /* Will call free_more_memory() */
    gfp_t gfp_mask;

    gfp_mask = (mapping_gfp_mask(inode->i_mapping) & ~__GFP_FS) | gfp;
    gfp_mask |= __GFP_NOFAIL;               /*缓存页分配掩码*/
```

```
    page = find_or_create_page(inode->i_mapping, index, gfp_mask);
                                                    /*查找或创建缓存页，见上文*/
    if (!page)
        return ret;

    BUG_ON(!PageLocked(page));

    if (page_has_buffers(page)) {       /*如果缓存页具有块缓存头实例链表*/
        bh = page_buffers(page);
        if (bh->b_size == size) {        /*缓存块大小为 size*/
            end_block = init_page_buffers(page, bdev,(sector_t)index << sizebits,size);
                                                /*初始化块缓存头实例，/fs/buffer.c*/
            goto done;
        }
        if (!try_to_free_buffers(page))
                    /*如果缓存块大小不同，则释放块缓存头链表实例，/fs/buffer.c*/
            goto failed;
    }

    /*缓存页没有块缓存头实例链表，创建块缓存头实例链表*/
    bh = alloc_page_buffers(page, size, 0);
                /*为缓存页分配块缓存头 buffer_head 实例并初始化，缓存块大小为 size，/fs/buffer.c*/
    if (!bh)
        goto failed;

    spin_lock(&inode->i_mapping->private_lock);
    link_dev_buffers(page, bh);         /*关联块缓存头和缓存页，/fs/buffer.c*/
    end_block = init_page_buffers(page, bdev, (sector_t)index << sizebits,size);
                /*初始化块缓存头实例，回调函数为 NULL，成功返回块设备结束块号，/fs/buffer.c*/
    spin_unlock(&inode->i_mapping->private_lock);
done:
    ret = (block < end_block) ? 1 : -ENXIO;
failed:
    unlock_page(page);
    page_cache_release(page);
    return ret;         /*成功返回 1*/
}
```

grow_dev_page()函数首先在块设备文件的页缓存中查找或创建缓存块所在的缓存页，然后判断其块缓存头实例链表是否存在且有效，如果是则初始化块缓存头链表实例后，函数返回。如果块缓存头实例链表不存在，则创建块缓存头链表实例并初始化，函数返回。如果块缓存头实例链表存在但无效（缓存块大小不一致）则释放缓存头链表中实例，函数返回。

注意：grow_dev_page()函数只是为欲读取缓存块所在的缓存页分配并初始化 buffer_head 实例，并没有返回块缓存头 buffer_head 实例。__getblk_slow()函数随后会调用__find_get_block()函数查找块缓存头实例，并返回。

●读缓存块

__bread_gfp()函数在调用__getblk_gfp()函数获取块缓存头实例后，如果缓存块中数据无效，将继续调用__bread_slow(bh)函数从块设备中读取数据至缓存块。

__bread_slow()函数定义如下（/fs/buffer.c）：

```
static struct buffer_head *__bread_slow(struct buffer_head *bh)
{
    lock_buffer(bh);
    if (buffer_uptodate(bh)) {
        unlock_buffer(bh);
        return bh;
    } else {
        get_bh(bh);
        bh->b_end_io = end_buffer_read_sync;        /*bio 实例完成后调用，唤醒等待进程*/
        submit_bh(READ, bh);            /*创建关提交 bio 实例，/fs/buffer.c*/
        wait_on_buffer(bh);             /*当前进程在块缓存头上睡眠等待，等待读数据完成*/
        if (buffer_uptodate(bh))        /*唤醒后块缓存中数据有效，返回 buffer_head 指针*/
            return bh;
    }
    brelse(bh);
    return NULL;
}
```

__bread_slow()函数调用 submit_bh(rw, bh)函数创建并提交 bio 实例，然后当前进程在块缓存头中睡眠等待，bio 实例数据传输完成后，在 bio 实例的 bi_end_io()函数中将调用 end_buffer_read_sync()函数唤醒在块缓存头中睡眠等待的进程，也就是说__bread_gfp()函数是同步读函数。

依块缓存头创建并提交 bio 实例的 **submit_bh(rw, bh)**函数定义在/fs/buffer.c 文件内，代码如下：

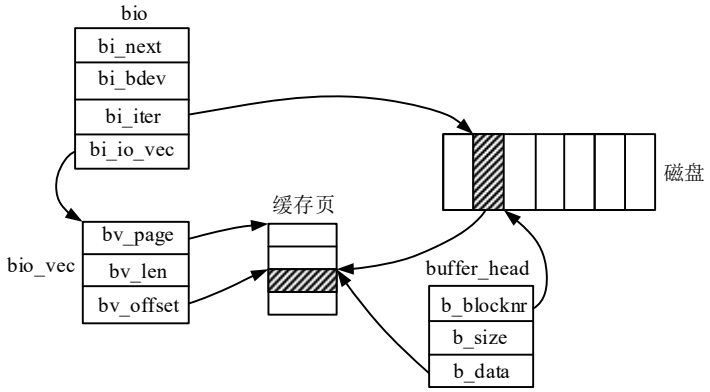
```
int submit_bh(int rw, struct buffer_head *bh)
 /*rw：读写操作，bh：buffer_head 实例指针*/
{
        return submit_bh_wbc(rw, bh, 0, NULL);     /*/fs/buffer.c*/
}
```

submit_bh_wbc()函数定义在/fs/buffer.c 文件内，代码如下：

```
static int submit_bh_wbc(int rw, struct buffer_head *bh,unsigned long bio_flags, \
                                                        struct writeback_control *wbc)
/*bio_flags：这里为 0，wbc：这里为 NULL*/
{
    struct bio *bio;

    BUG_ON(!buffer_locked(bh));
    BUG_ON(!buffer_mapped(bh));
    BUG_ON(!bh->b_end_io);          /*函数调用者必须设置 bh->b_end_io*/
    BUG_ON(buffer_delay(bh));
    BUG_ON(buffer_unwritten(bh));
```

```
    if (test_set_buffer_req(bh) && (rw & WRITE))
        clear_buffer_write_io_error(bh);

    bio = bio_alloc(GFP_NOIO, 1);     /*分配 bio 实例，只需要一个 bio_vec 实例，见第 10 章*/

    if (wbc) {              /*写缓存块时需要使用此参数，见本章下文*/
        wbc_init_bio(wbc, bio);
        wbc_account_io(wbc, bh->b_page, bh->b_size);
    }/*没有选择 CGROUP_WRITEBACK 配置选项以下两函数为空操作，/include/linux/writeback.h*/

    /*设置 bio 实例*/
    bio->bi_iter.bi_sector = bh->b_blocknr * (bh->b_size >> 9); /*数据块号转换扇区号（512 字节）*/
    bio->bi_bdev = bh->b_bdev;                  /*block_device 实例*/
    bio->bi_io_vec[0].bv_page = bh->b_page;       /*缓存页 page 实例*/
    bio->bi_io_vec[0].bv_len = bh->b_size;          /*缓存块大小*/
    bio->bi_io_vec[0].bv_offset = bh_offset(bh);   /*bh->b_data 在 bh->b_page 缓存页中的偏移量*/

    bio->bi_vcnt = 1;                       /*包含一个 bio_vec 实例*/
    bio->bi_iter.bi_size = bh->b_size;       /*块大小，字节数*/

    bio->bi_end_io = end_bio_bh_io_sync;   /*设置回调函数，调用 bh->b_end_io()函数，/fs/buffer.c*/
    bio->bi_private = bh;         /*bio 实例私有数据指针指向 buffer_head 实例*/
    bio->bi_flags |= bio_flags;

    /* Take care of bh's that straddle the end of the device */
    guard_bio_eod(rw, bio);

    if (buffer_meta(bh))
        rw |= REQ_META;
    if (buffer_prio(bh))
        rw |= REQ_PRIO;

    submit_bio(rw, bio);     /*提交 bio，完成读操作，见第 10 章*/
    return 0;
}
```

submit_bh_wbc()函数由块缓头 buffer_head 实例创建的 bio 实例结构如下图所示：



对单个缓存块的读/写操作，在 bio 实例中只需要一个 bio_vec 实例。__bread_slow()函数将 buffer_head 实例回调函数（b_end_io 指针）赋值为 end_buffer_read_sync()函数，bio 实例回调函数 end_bio_bh_io_sync()内将调用此函数，用于唤醒在块缓存头上睡眠等待的进程。

对于读缓存块操作，块缓存头中赋予的 end_buffer_read_sync()函数定义如下（/fs/buffer.c）：

```
void end_buffer_read_sync(struct buffer_head *bh, int uptodate)
{
    __end_buffer_read_notouch(bh, uptodate);      /*设置缓存块数据有效标记等，/fs/buffer.c*/
    put_bh(bh);    /*引用计数减 1，释放 buffer_head 实例，唤醒睡眠进程，/include/linux/buffer_head.h*/
}
```

11.1.4 读写缓存页

这里说的读写缓存页是指从块设备读数据写入缓存页和将缓存页中数据写入块设备，而不是说用户进程读写缓存页。下一节将介绍用户进程从页缓存中读写文件内容数据。

页缓存与块设备之间的数据传输由地址空间操作结构中的函数实现。在文件系统类型打开文件，为其创建 inode 实例的函数中将对 inode->i_mapping->**a_ops** 和 inode->**i_fop** 等成员赋值。

下面列出 ext2 文件系统类型中文件以及块设备文件的地址空间操作结构实例，看看其中对缓存页的读写操作是如何进行的。

ext2 文件系统中普通文件地址空间操作结构实例如下（/fs/ext2/inode.c）：

```
const struct address_space_operations ext2_aops = {
    .readpage           = ext2_readpage,      /*整页读函数，调用 mpage_readpage()函数*/
    .readpages          = ext2_readpages,     /*读多个缓存页，调用 mpage_readpages()函数*/
    .writepage          = ext2_writepage,     /*单页写函数，调用 block_write_full_page()函数*/
    .write_begin        = ext2_write_begin,
    .write_end          = ext2_write_end,
    .bmap               = ext2_bmap,
    .direct_IO          = ext2_direct_IO,     /*直接读写函数，调用 blockdev_direct_IO()函数*/
    .writepages         = ext2_writepages,    /*调用 mpage_writepages()函数，回写缓存页*/
    .migratepage        = buffer_migrate_page,
    .is_partially_uptodate    = block_is_partially_uptodate,
    .error_remove_page      = generic_error_remove_page,
};
```

块设备文件地址空间操作结构实例为 def_blk_aops，定义如下（/fs/block_dev.c）：

```
static const struct address_space_operations def_blk_aops = {
    .readpage       = blkdev_readpage,      /*单页读函数，调用 block_read_full_page()函数*/
    .readpages      = blkdev_readpages,     /*多页读函数，调用 mpage_readpages()函数*/
    .writepage      = blkdev_writepage,     /*单页写函数，调用 block_write_full_page()函数*/
    .write_begin    = blkdev_write_begin,
    .write_end      = blkdev_write_end,
    .writepages     = generic_writepages,   /*页缓存回写通用函数*/
    .releasepage    = blkdev_releasepage,   /*调用 try_to_free_buffers(page)函数释放缓存页*/
    .direct_IO      = blkdev_direct_IO,     /*直接读写函数，/fs/block_dev.c*/
    .is_dirty_writeback = buffer_check_dirty_writeback,
};
```

块设备文件地址空间使用的是 bdev_inode.**vfs_inode** 成员（inode 实例）中的地址空间，地址空间操作结构在创建 bdev_inode 实例时赋值，赋为 def_blk_aops 实例指针。

读缓存页函数：

（1）普通文件：调用通用的 **mpage_readpage()**函数，按页进行读操作，由缓存页构建 bio 实例，提交到请求队列。

（2）块设备文件：调用 **block_read_full_page()**函数，将缓存页划分成缓存块，逐块（数据无效缓存块）调用通用的缓存块读写函数，完成缓存页中缓存块的读操作，按缓存块进行读。

写缓存页函数：

普通文件和块设备文件：调用的 **block_write_full_page()**函数，将缓存页划分成缓存块进行写操作，按缓存块进行写操作。

读写多页的函数简单地说就是执行多次单页的读写操作，下面介绍单页的读写函数，多页读写函数请读者自行阅读。

在介绍读写缓存页的函数前，先介绍一下 get_block_t 类型函数的定义（/include/linux/fs.h）：

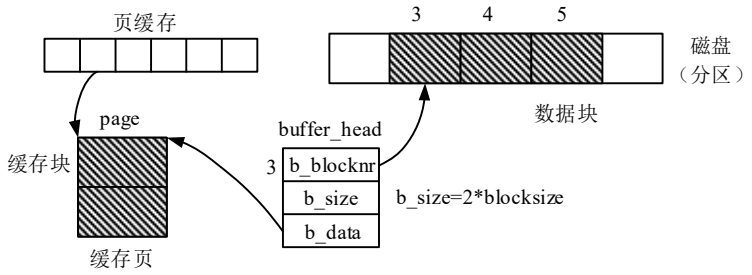
```
typedef int    (get_block_t)(struct inode *inode, sector_t iblock,struct buffer_head *bh_result, int create);
```

参数 inode 表示文件 inode 实例指针，iblock 表示读取文件内容的逻辑块号（超级块中给定的块大小），即将文件内容按数据块大小为单位进行划分所得的逻辑块号，bh_result 表示块缓存头 buffer_head 实例指针，create 指示如果块设备中不存在映射数据块时是否创建映射的数据块，读操作函数中为 0（不创建），写操作函数中为 1（表示扩展文件内容）。

get_block_t()函数需由文件系统类型代码定义，功用是获取从 iblock 逻辑块开始，文件内容逻辑块到块设备中数据块的映射关系，并写入 buffer_head 实例。

在调用 get_block_t()函数前 buffer_head 实例 b_size 成员需要设置检查映射区的最大长度，在函数中会将 iblock 逻辑块映射的数据块编号写入块缓存头 **b_blocknr** 成员，b_size 设为块设备中连续映射数据块的长度，但不会超过初始值。

下面用图示来说明 get_block_t()函数的功用：



如上图中所示，分区中从第 3 个缓存块开始，连续的 3 个数据块保存的是同一个文件的连续内容。假设一个缓存页中包含 2 个缓存块，buffer_head 实例 b_size 成员设为缓存页大小，逻辑块号 iblock 设为缓存页第 1 个缓存块对应的块号，调用 get_block_t()函数，执行完后 buffer_head 实例 b_blocknr 成员保存数据块号 3，b_size 为 2 个数据块大小值（保持不变），b_data 指向缓存页起始地址。

1 按块读函数

文件系统类型定义的地址空间操作结构中，缓存页的读写操作可按缓存块（数据块）进行，也可以按缓存页进行。

下面先看按缓存块进行读操作的函数，**block_read_full_page()**通用函数用于对缓存页按缓存块进行读操作，后面再介绍按页读和写缓存页函数。

■通用读函数

block_read_full_page()函数内首先将缓存页按缓存块大小划分成若干个块，并为每个缓存块创建块缓存头 buffer_head 实例，添加到缓存页 page 实例管理的链表中，最后逐个对缓存块发起读操作。

块设备文件地址空间操作结构实例中，读缓存页函数 blkdev_readpage()调用 block_read_full_page()函数完成读操作，这是通用的按块读缓存页函数，函数定义如下（/fs/block_dev.c）：

```
static int blkdev_readpage(struct file * file, struct page * page)
{
    return block_read_full_page(page, blkdev_get_block);     /*检查映射函数为 blkdev_get_block()*/
}
```

块设备文件的检查数据块映射函数为 blkdev_get_block()，这个函数比较简单，因为块设备中数据块与块设备文件内容是一一映射的，总是连续映射的，函数内只需要设置 bh->b_blocknr = iblock，不需要修改 b_size 成员值。

按缓存块读缓存页的 **block_read_full_page()**函数定义在/fs/buffer.c 文件内，代码如下：

```
int block_read_full_page(struct page *page, get_block_t *get_block)
/*page：缓存页 page 指针，get_block：检查数据块映射的函数*/
{
    struct inode *inode = page->mapping->host;     /*文件 inode，块设备文件为 bdev_inode.vfs_inode*/
    sector_t iblock, lblock;
    struct buffer_head *bh, *head, *arr[MAX_BUF_PER_PAGE];   /*arr[]保存需要执行读的块缓存头*/
    unsigned int blocksize, bbits;
    int nr, i;
    int fully_mapped = 1;

    head = create_page_buffers(page, inode, 0);       /*为缓存页创建块缓存头实例链表，/fs/buffer.c*/
                    /*由 inode->i_blkbits 确定缓存块大小，块设备文件为内存页大小 PAGE_SIZE*/
```

```
blocksize = head->b_size;       /*缓存块大小*/
bbits = block_size_bits(blocksize);

iblock = (sector_t)page->index << (PAGE_CACHE_SHIFT - bbits);
                                                    /*首个缓存块在文件内容中的逻辑块号*/
lblock = (i_size_read(inode)+blocksize-1) >> bbits;     /*文件内容最后对应的缓存块号*/
bh = head;
nr = 0;
i = 0;

do {            /*遍历缓存页中的块缓存头实例链表*/
      if (buffer_uptodate(bh))          /*缓存块中数据有效，跳过无需处理*/
            continue;

      if (!buffer_mapped(bh)) {       /*如果缓存块还没有建立到块设备的映射*/
          int err = 0;

          fully_mapped = 0;
          if (iblock < lblock) {
             WARN_ON(bh->b_size != blocksize);    /*b_size 必须是一个数据块大小*/
             err = get_block(inode, iblock, bh, 0);    /*建立缓存块到块设备的映射，成功返回 0*/
             if (err)
                  SetPageError(page);
          }
          if (!buffer_mapped(bh)) {     /*缓存块映射未建立，访问了文件空洞，缓存块用 0 填充*/
              zero_user(page, i * blocksize, blocksize);        /*清零缓存块*/
              if (!err)
                  set_buffer_uptodate(bh);          /*设置有效标记位*/
              continue;                         /*下一个缓存块*/
          }
          if (buffer_uptodate(bh))        /*缓存块数有效，无需执行读操作*/
               continue;
      }   /*未建立映射 if 语句结束*/

      arr[nr++] = bh;     /*数据无效块缓存头关联到指针数组，需要执行读操作*/
} while (i++, iblock++, (bh = bh->b_this_page) != head);       /*遍历块缓存头实例链表结束*/

if (fully_mapped)           /*如果缓存页中缓存块映射都已建立且数据有效*/
    SetPageMappedToDisk(page);

if (!nr) {            /*如果没有需要执行读操作的缓存块*/
    if (!PageError(page))
        SetPageUptodate(page);          /*设置缓存页数据有效标记位*/
    unlock_page(page);
    return 0;               /*返回*/
}
```

```
        for (i = 0; i < nr; i++) {       /*设置需要执行读操作块缓存头实例*/
            bh = arr[i];
            lock_buffer(bh);
            mark_buffer_async_read(bh);          /*bh->b_end_io = end_buffer_async_read，/fs/buffer.c*/
        }

        for (i = 0; i < nr; i++) {       /*对需要执行读操作的缓存块逐个执行读操作*/
            bh = arr[i];
            if (buffer_uptodate(bh))       /*如果缓存块数据有效*/
                end_buffer_async_read(bh, 1);     /*调用异步读操作完成函数*/
            else
                submit_bh(READ, bh);     /*缓存块数据无效，发起缓存块读操作，见本章上文*/
        }
        return 0;
    }
```

block_read_full_page()函数比较好理解，主要工作如下：

（1）调用 **create_page_buffers()**函数为缓存页创建块缓存头实例链表。

（2）遍历块缓存头实例，建立与块设备中数据块的映射关系，并找出数据无效的块缓存头。

（3）对数据无效的块缓存头调用 submit_bh(READ, bh)函数，执行读操作。

数据无效块缓存头 buffer_head 实例的 b_end_io()函数指针成员设为 **end_buffer_async_read()**函数指针，这个函数在 bio 实例中数据传输完成时调用，主要是设置块缓存头数据有效，检查缓存页中所有块缓存头是否都设置了数据有效标记位，如果是则设置缓存页的数据有效标记位。

执行缓存块读操作的 submit_bh(READ, bh)函数前面介绍过了，下面介绍一个 create_page_buffers()函数和 end_buffer_async_read()函数的实现。

●创建块缓存

create_page_buffers()函数用于为缓存页创建块缓存头实例链表，文件 inode 实例 i_blkbits 成员指示了缓存块（文件系统中数据块）的大小（1<<(inode->i_blkbits)），函数定义如下（/fs/buffer.c）：

```
static struct buffer_head *create_page_buffers(struct page *page, struct inode *inode, unsigned int b_state)
{
    BUG_ON(!PageLocked(page));

    if (!page_has_buffers(page))        /*page->private 成员为 NULL*/
        create_empty_buffers(page, 1 << ACCESS_ONCE(inode->i_blkbits), b_state);
                                        /*创建块缓存头实例链表，并关联 page->private，/fs/buffer.c*/
    return page_buffers(page);      /*返回 page->private，buffer_head 实例指针*/
}
```

create_page_buffers()函数内判断 page->private 成员是否为 NULL，如果是则说明尚未创建块缓存头实例链表，则调用执行创建操作，创建函数定义如下：

```
void create_empty_buffers(struct page *page,unsigned long blocksize, unsigned long b_state)
/*page：缓存页 page 实例指针，blocksize：缓存块大小，为 1<<(inode->i_blkbits)，b_state：状态，0*/
{
    struct buffer_head *bh, *head, *tail;
```

```
        head = alloc_page_buffers(page, blocksize, 1);     /*创建并初始化块缓存头实例链表，/fs/buffer.c*/
        bh = head;
        do {                    /*设置块缓存头*/
            bh->b_state |= b_state;
            tail = bh;
            bh = bh->b_this_page;
        } while (bh);
        tail->b_this_page = head;       /*块缓存头实例组成循环单链表*/

        spin_lock(&page->mapping->private_lock);
        if (PageUptodate(page) || PageDirty(page)) {       /*缓存页有效或脏，表示不用从块设备读数据*/
            bh = head;
            do {                /*根据缓存页标记设置各缓存块标记，有效和脏标记*/
                if (PageDirty(page))            /*页脏*/
                    set_buffer_dirty(bh);
                if (PageUptodate(page))         /*页数据有效*/
                    set_buffer_uptodate(bh);
                bh = bh->b_this_page;
            } while (bh != head);
        }
        attach_page_buffers(page, head); /*page->private=head，设置标记位，/include/linux/buffer_head.h*/
        spin_unlock(&page->mapping->private_lock);
    }
```

create_empty_buffers()函数根据文件系统数据块大小为缓存页创建相应数量的块缓存头实例，初始化，并组成链表，根据状态参数和 page 实例标记设置块缓存头实例相应成员（如标记位），最后建立 page 实例与 buffer_head 实例链表的关联。

●读完成回调函数

由前面介绍的 submit_bh()函数可知，由块缓存头构建的 bio 实例，其传输完成的回调函数（bi_end_io 成员）设为 end_bio_bh_io_sync()函数指针，此函数将调用 bh->b_end_io()函数。

在这里需要执行读操作的 buffer_head 实例的 b_end_io()函数设为 **end_buffer_async_read()**函数，定义如下（/fs/buffer.c）：

```
    static void end_buffer_async_read(struct buffer_head *bh, int uptodate)
    /*uptodate：缓存块中数据是否有效*/
    {
        unsigned long flags;
        struct buffer_head *first;
        struct buffer_head *tmp;
        struct page *page;
        int page_uptodate = 1;          /*缓存页中数据是否有效*/

        BUG_ON(!buffer_async_read(bh));

        page = bh->b_page;          /*缓存页*/
        if (uptodate) {
            set_buffer_uptodate(bh);        /*设置数据有效标记*/
```

```
        } else {                                /*数据无效，读失败*/
            clear_buffer_uptodate(bh);
            buffer_io_error(bh, ", async page read");
            SetPageError(page);
        }

        first = page_buffers(page);             /*缓存页中第一个块缓存头*/
        local_irq_save(flags);
        bit_spin_lock(BH_Uptodate_Lock, &first->b_state);
        clear_buffer_async_read(bh);
        unlock_buffer(bh);
        tmp = bh;
        do {            /*遍历缓存块*/
            if (!buffer_uptodate(tmp))          /*缓存块数据是否有效*/
                page_uptodate = 0;              /*缓存块数据无效，缓存页数据也无效*/
            if (buffer_async_read(tmp)) {
                BUG_ON(!buffer_locked(tmp));
                goto still_busy;
            }
            tmp = tmp->b_this_page;             /*下一个块缓存头*/
        } while (tmp != bh);
        bit_spin_unlock(BH_Uptodate_Lock, &first->b_state);
        local_irq_restore(flags);

        if (page_uptodate && !PageError(page))
            SetPageUptodate(page);              /*设置缓存页数据有效标记位*/
        unlock_page(page);
        return;
        ...
    }
```

end_buffer_async_read()函数比较好理解，就是根据参数 uptodate 设置 buffer_head 实例的数据有效标记位，然后检查同一缓存页中其它块缓存头数据是否有效，如果都有效则设置缓存页数据有效，表示读缓存页操作完成。

由上可知，block_read_full_page()函数是一个异步读函数，函数返回后，并不代表缓存页中的数据就有效，需要检查缓存页的数据有效性标记位，置 1 才表示缓存页读操作完成了。

2 按页读函数

前面介绍了按块读取缓存页的函数，下面介绍按页读缓存页的函数。

按页读缓存页的先决条件是缓存页映射到块设备中连续的数据块，如果缓存页映射到块设备不是连续的，那也不能按页读，只能拆分成缓存块读。

在 address_space_operations 结构体中包含读单页的 readpage()函数和读多页的 readpages()函数指针成员，例如：ext2 文件系统类型中读单个缓存页的函数定义如下（/fs/ext2/inode.c）：

```
static int ext2_readpage(struct file *file, struct page *page)
/*file：文件 file 实例指针，page：所读缓存页 page 实例指针*/
{
```

```
    return mpage_readpage(page, ext2_get_block);        /*/fs/mpage.c*/
};
```

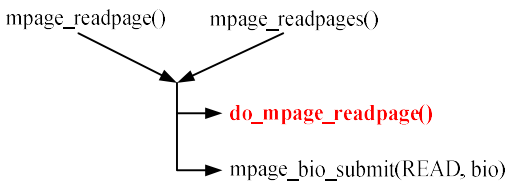
以上函数中调用通用的读单页缓存页的 mpage_readpage(struct page *page, get_block_t get_block)函数，参数 ext2_get_block()函数指针是由 ext2 文件系统类型实现的映射函数，这里不介绍其具体实现了。

在进程读文件的操作中，多数时候是调用读取多个缓存页函数执行文件预读，ext2 文件系统类型读多个缓存页的函数定义如下（/fs/ext2/inode.c）：

```
static int ext2_readpages(struct file *file, struct address_space *mapping,struct list_head *pages, \
                                                                    unsigned nr_pages)
/*pages：所读缓存页双链表头，nr_pages：缓存页数量*/
{
    return mpage_readpages(mapping, pages, nr_pages, ext2_get_block);
}
```

pages 参数是一个双链表头，链接了读缓存页的 page 实例，nr_pages 为链表中缓存页数量，函数内调用通用的读多个缓存页函数 mpage_readpages()实现缓存页的读操作。

通用的读单个缓存页和读多个缓存页的函数内部调用的都是相同的函数，如下图所示：



单页读操作函数 mpage_readpage()定义在/fs/mpage.c 文件内，代码如下（不经常调用）：

```
int mpage_readpage(struct page *page, get_block_t get_block)
{
    struct bio *bio = NULL;                 /*bio 指针*/
    sector_t    last_block_in_bio = 0;
    struct buffer_head map_bh;      /*块缓存头实例*/
    unsigned long first_logical_block = 0;

    map_bh.b_state = 0;
    map_bh.b_size = 0;          /*大小初设为 0*/
    bio = do_mpage_readpage(bio, page, 1, &last_block_in_bio,&map_bh, \
                                        &first_logical_block, get_block);
                                    /*处理缓存页（可能已经执行了读操作），见下文，/fs/mpage.c*/
    if (bio)        /*bio 关联的缓存页可可按页读才不为 NULL，否则为 NULL*/
        mpage_bio_submit(READ, bio);
            /*缓存页映射到块设备中连续数据块时，提交 bio 实例，/fs/mpage.c*/
    return 0;
}
```

多页读操作函数 mpage_readpages()定义如下（/fs/mpage.c）：

```
int mpage_readpages(struct address_space *mapping, struct list_head *pages,unsigned nr_pages, \
                                                                    get_block_t get_block)
{
```

```
    struct bio *bio = NULL;
    unsigned page_idx;
    sector_t last_block_in_bio = 0;
    struct buffer_head map_bh;     /*暂存 get_block()函数获取的缓存页映射信息*/
    unsigned long first_logical_block = 0;

    map_bh.b_state = 0;
    map_bh.b_size = 0;
    for (page_idx = 0; page_idx < nr_pages; page_idx++) {     /*遍历每个缓存页 page 实例*/
        struct page *page = list_entry(pages->prev, struct page, lru);

        prefetchw(&page->flags);
        list_del(&page->lru);
        if (!add_to_page_cache_lru(page, mapping,page->index, GFP_KERNEL)) {       /*添加缓存页*/
            bio = do_mpage_readpage(bio, page,nr_pages - page_idx, \
                    &last_block_in_bio, &map_bh,&first_logical_block,get_block);
        }     /*逐个处理缓存页，见下文，/fs/mpage.c*/
        page_cache_release(page);
    }
    BUG_ON(!list_empty(pages));
    if (bio)                                  /*bio 关联的缓存页可按页读*/
        mpage_bio_submit(READ, bio);     /*提交 bio 实例，/fs/mpage.c*/
    return 0;
}
```

以上两个函数内部都是调用 do_mpage_readpage()和 mpage_bio_submit()函数执行读操作，单页读函数中调用一次 do_mpage_readpage()函数，而多页读函数中对每个缓存页都调用一次 do_mpage_readpage()函数。如果缓存页不能按页读取，则在 do_mpage_readpage()函数中就执行按块读操作，返回 NULL。如果可以按页读取，返回值才不为 NULL，将由随后调用的 mpage_bio_submit()函数执行按页读操作，后面将介绍这个函数的定义。

这里先来看下 mpage_bio_submit()函数的实现，此函数用于提交 bio 实例，按页读缓存页：

```
static struct bio *mpage_bio_submit(int rw, struct bio *bio)
{
    bio->bi_end_io = mpage_end_io;   /*bio 执行完的回调函数，更新 page 标记位等，/fs/mpage.c*/
    guard_bio_eod(rw, bio);     /*/fs/buffer.c*/
    submit_bio(rw, bio);       /*提交 bio 实例*/
    return NULL;               /*始终返回 NULL*/
}
```

mpage_bio_submit()函数设置 bio 执行完的回调函数为 mpage_end_io()，其主要工作是调用 page_endio()函数（/mm/filemap.c）完成 page 实例数据有效标记位的更新等。

mpage_bio_submit()函数调用 submit_bio(rw, bio)函数向块设备请求队列提交 bio 实例，函数始终返回 NULL，这里的读操作也是异步读操作。

■读单页函数

下面重点来看一下 do_mpage_readpage()函数的实现，这个函数用于处理单个缓存页的读操作，在多页

读操作中，依次对每个缓存页调用此函数处理。

do_mpage_readpage()函数原型如下：

```
static struct bio *do_mpage_readpage(struct bio *bio, struct page *page, unsigned nr_pages,
        sector_t *last_block_in_bio, struct buffer_head *map_bh,
        unsigned long *first_logical_block, get_block_t get_block)
```

参数简介如下 ：

bio：bio 实例指针，用于收集可按页读的缓存页，且映射内容也是连续的。

page：当前处理缓存页的 page 实例指针。

nr_pages：含当前缓存页，一共有多少个缓存页还需要读，这些缓存页的内容逻辑上是连续的。

***last_block_in_bio**：bio 指向实例中，最后关联缓存页中最后缓存块映射的块设备数据块号。

map_bh：指向 buffer_head 实例，用于获取块设备中一段连续映射区的信息。

***first_logical_block**：map_bh 指向 buffer_head 实例中最近一次调用 get_block()函数获取映射信息时，起始的文件内容逻辑块号。

get_block()：文件系统类型代码定义的获取映射信息的函数指针。

do_mpage_readpage()函数调用 get_block()函数获取文件内容在块设备中一段连续映射区域的信息，映射信息暂存在 map_bh 指向实例中，bio 指向实例用于收集整页映射到块设备中连续区域的缓存页，也就是连续映射到块设备中连续区域的缓存页，以便一次访问块设备更多的区域。

下面以一个例子来说明 do_mpage_readpage()函数的执行流程。

如下图所示，假设要读取 3 个连续的缓存页 page1~3，将对每个缓存页调用 do_mpage_readpage()函数，最开始 bio 实例指向 NULL。每个缓存页包含 2 个缓存块，page1、page2 和 page3 中第 1 个缓存块映射到块设备中一段连续的区域，page3 第 2 个缓存块映射到离散的数据块。

缓存页中缓存块的逻辑块号（文件内容中块号）隐含在 page->index 成员中，这个成员值是缓存页位于文件内容中的逻辑页号，由缓存页中包含的缓存块数量，就可以计算得缓存页中缓存块的逻辑块号。

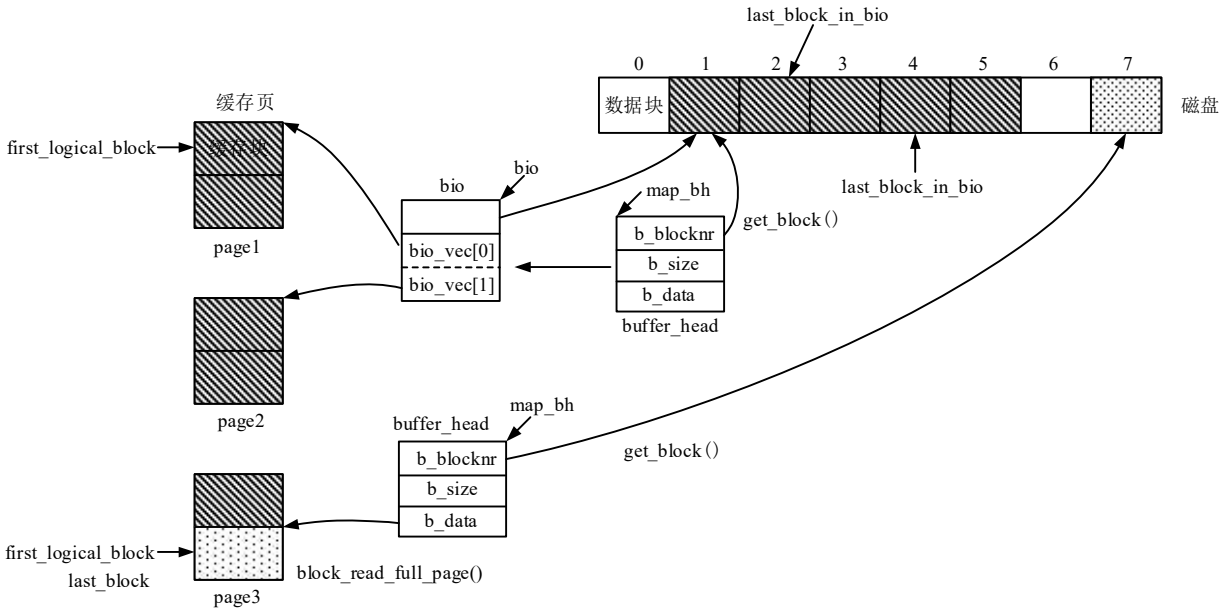
对 page1 调用 do_mpage_readpage()函数时，将调用 get_block()函数获取连续映射区信息，需要注意的是 do_mpage_readpage()函数每次调用 get_block()函数时，都将 **map_bh->b_size** 值设为当前缓存块至欲读取最后缓存块之间的大小，也就是说每次都试图获取最大的连续映射区信息。

第一次调用 get_block()函数时，map_bh 实例中包含 5 个连续映射数据块。do_mpage_readpage()函数从 map_bh 实例中获取 page1 中 2 个缓存块映射的数据块，发现它们是连续的，就创建 bio 实例，关联 page1，但是不提交，函数返回 bio 实例指针，因为可能与下一页也映射到连续的数据块，可以合并一交提交。

现在对 page2 调用 do_mpage_readpage()函数，此时 bio 指向为 page1 创建的 bio 实例，map_bh 实例中还包含上一次调用 get_block()函数获取的映射信息。当发现 page2 的映射信息也包含在 map_bh 实例中时，就不需要调用 get_block()函数了，直接从中获取 page2 缓存页映射的数据块编号。判断 page2 每一个缓存块映射的数据块号与 bio 实例映射的最后数据块连续，表示 page2 可以 page1 合并到一个 bio 实例，则执行合并操作。

现在对 page3 调用 do_mpage_readpage()函数，此时可从 map_bh 实例中获取第 1 个缓存块映射的数据块，但是不包含第 2 个缓存块的映射信息，此时就要再次调用 get_block()函数，获取第 2 个缓存块的映射信息。当发现第 1 个和第 2 个缓存块映射的数据块不连续时，提交之前创建的 bio 实例，此实例中包含 page1 和 page2 的映射信息，最后对 page3 调用 block_read_full_page()函数按块进行读操作，3 个缓存页的读操作完成，函数返回 NULL，以上操作中一共调用了 2 次 get_block()函数。

简单地说，就是通过 bio 实例来收集连续映射的缓存页，当发现与下一页的映射不连续，或下一页不能按页访问时，提交 bio 实例，对不能按页读的缓存页执行按块读操作。



下面看一下 do_mpage_readpage()函数的定义（/fs/mpage.c），代码如下，其中包含对文件空洞的处理：

```
static struct bio *do_mpage_readpage(struct bio *bio, struct page *page, unsigned nr_pages,
                                     sector_t *last_block_in_bio, struct buffer_head *map_bh,
                                     unsigned long *first_logical_block, get_block_t get_block)
{
    struct inode *inode = page->mapping->host;
    const unsigned blkbits = inode->i_blkbits;        /*数据块（缓存块）大小，以 2 为底取对数*/
    const unsigned blocks_per_page = PAGE_CACHE_SIZE >> blkbits;
                                                      /*缓存页中包含缓存块的数量*/
    const unsigned blocksize = 1 << blkbits;          /*数据块（缓存块）大小，字节数*/
    sector_t block_in_file;
    sector_t last_block;
    sector_t last_block_in_file;
    sector_t blocks[MAX_BUF_PER_PAGE];   /*缓存页中缓存块映射的数据块号*/
    unsigned page_block;                 /*缓存页中已解决映射关系的缓存块数量*/
    unsigned first_hole = blocks_per_page;
    struct block_device *bdev = NULL;
    int length;
    int fully_mapped = 1;    /*缓存页是否整页映射到连续的数据块中*/
    unsigned nblocks;
    unsigned relative_block;

    if (page_has_buffers(page))       /*page->private 包含缓存头 buffer_head 实例，跳至按块读操作*/
        goto confused;

    block_in_file = (sector_t)page->index << (PAGE_CACHE_SHIFT - blkbits);
                                      /*缓存页中首个缓存块的逻辑块号（文件内容中块号）*/
    last_block = block_in_file + nr_pages * blocks_per_page;     /*欲读的最后缓存块号加 1*/
    last_block_in_file = (i_size_read(inode) + blocksize - 1) >> blkbits;
                                      /*实际文件内容最后的数据块号*/
```

```
if (last_block > last_block_in_file)          /*读内容不能超出文件内容*/
    last_block = last_block_in_file;
page_block = 0;

/*查看上次 get_block()函数获得的映射结果*/
nblocks = map_bh->b_size >> blkbits;          /*上次 get_block()函数获取的映射区数据块数量*/
if (buffer_mapped(map_bh) && block_in_file > *first_logical_block &&
        block_in_file < (*first_logical_block + nblocks)) {
                                              /*本页进入了上次 get_block()函数获取的映射区*/
    unsigned map_offset = block_in_file - *first_logical_block;
                                               /*block_in_file 在映射区中的偏移量（块偏移量）*/
    unsigned last = nblocks - map_offset;    /*映射区 block_in_file 之后的缓存块数量*/

    for (relative_block = 0; ; relative_block++) {
        if (relative_block == last) {
            clear_buffer_mapped(map_bh);
            break;
        }
        if (page_block == blocks_per_page)
            break;
        blocks[page_block] = map_bh->b_blocknr + map_offset +relative_block;
        page_block++;                  /*设置缓存页中缓存块映射的数据块号*/
        block_in_file++;
    }
    bdev = map_bh->b_bdev;
}

/*判断是否要调用 get_block()函数获取更多映射信息*/
map_bh->b_page = page;
while (page_block < blocks_per_page) {     /*已解决映射缓存块数量比一页中包含块数量少*/
    map_bh->b_state = 0;
    map_bh->b_size = 0;

    if (block_in_file < last_block) {        /*从 block_in_file 逻辑块开始获取映射关系*/
        map_bh->b_size = (last_block-block_in_file) << blkbits;
                                              /*当前数据块至欲读最后缓存块之间大小*/
        if (get_block(inode, block_in_file, map_bh, 0))       /*获取映射信息*/
                                              /*映射成功或无映射返回 0，错误返回错误码*/
            goto confused;
        *first_logical_block = block_in_file;     /*保存映射关系中起始缓存块号*/
    }

    if (!buffer_mapped(map_bh)) {
                        /*块缓存头没有设置映射标记，表示遇到文件空洞区，扫描下一缓存块*/
        fully_mapped = 0;
        if (first_hole == blocks_per_page)
```

```
                first_hole = page_block;       /*空洞缓存块在页内的编号*/
            page_block++;
            block_in_file++;
            continue;
        }

        if (buffer_uptodate(map_bh)) {       /*文件系统类型代码中可能在 get_block()函数中读数据*/
            map_buffer_to_page(page, map_bh, page_block);
            goto confused;
        }

        if (first_hole != blocks_per_page)     /*表示缓存页中有空洞，跳至按块读函数*/
            goto confused;        /* hole -> non-hole */

        /*本次 get_block()获取的第一个缓存块映射块号与上一个缓存块映射块号不连续*/
        if (page_block && blocks[page_block-1] != map_bh->b_blocknr-1)
            goto confused;                    /*按块读取*/

        /*以下是本次 get_block()获取的映射与前面缓存块的映射连续*/
        nblocks = map_bh->b_size >> blkbits;     /*本次 get_block()获取的映射区长度（块数）*/
        for (relative_block = 0; ; relative_block++) {     /*填充 blocks[]数组，缓存块映射的数据块号*/
            if (relative_block == nblocks) {              /*本次映射区已使用完，清映射标记*/
                clear_buffer_mapped(map_bh);       /*清除块缓存头 BH_Mapped 标记位*/
                break;
            } else if (page_block == blocks_per_page)     /*到达缓存页最后缓存块*/
                break;
            blocks[page_block] = map_bh->b_blocknr+relative_block;     /*设置映射数据块号*/
            page_block++;
            block_in_file++;
        }
        bdev = map_bh->b_bdev;
    }
    /*while (page_block < blocks_per_page)循环结束，解决页内缓存块映射关系结束*/

    /*到这里表示缓存页映射到连续的数据块，或者整个缓存页为空洞*/
    if (first_hole != blocks_per_page) {
        zero_user_segment(page, first_hole << blkbits, PAGE_CACHE_SIZE); /*空洞缓存页填 0*/
        if (first_hole == 0) {
            SetPageUptodate(page);
            unlock_page(page);
            goto out;
        }
    } else if (fully_mapped) {
        SetPageMappedToDisk(page);          /*缓存页映射到连续数据块*/
    }
```

```
    /*运行到这里表示整个缓存页映射到块设备连续的数据块中*/
    if (fully_mapped && blocks_per_page == 1 && !PageUptodate(page) &&
        cleancache_get_page(page) == 0) {
        SetPageUptodate(page);
        goto confused;
    }

    if (bio && (*last_block_in_bio != blocks[0] - 1))      /*如果本页和上页的映射区不连续*/
        bio = mpage_bio_submit(READ, bio);            /*提交上页创建（或合并）的 bio 实例*/

alloc_new:              /*创建或合并至 bio 实例*/
    if (bio == NULL) {              /*bio 实例不存在，考虑创建*/
        if (first_hole == blocks_per_page) {      /*没有空洞*/
            if (!bdev_read_page(bdev, blocks[0] << (blkbits - 9),page))   /*/fs/block_dev.c*/
              /*调用块设备操作结构中定义的读整页函数，成功返回 0，函数未定义返回错误码*/
                goto out;
        }
        bio = mpage_alloc(bdev, blocks[0] << (blkbits - 9),
                    min_t(int, nr_pages, bio_get_nr_vecs(bdev)),GFP_KERNEL);
                                                  /*创建 bio 实例并初始化，/fs/mpage.c*/
        if (bio == NULL)
            goto confused;
    }

    length = first_hole << blkbits;
    if (bio_add_page(bio, page, length, 0) < length) {
                                  /*将 page 关联到 bio 实例中下一个 bio_vec 数组项，/block/bio.c*/
        bio = mpage_bio_submit(READ, bio);
        goto alloc_new;
    }

    relative_block = block_in_file - *first_logical_block;
    nblocks = map_bh->b_size >> blkbits;
    if ((buffer_boundary(map_bh) && relative_block == nblocks) || (first_hole != blocks_per_page))
        bio = mpage_bio_submit(READ, bio);
    else
        *last_block_in_bio = blocks[blocks_per_page - 1]; /*bio 中最后一个缓存块映射的数据块号*/
out:
    return bio;    /*返回创建（或合并）的 bio 实例或 NULL*/

confused:              /*按块读取当前缓存页*/
    if (bio)           /*先提交现有 bio 实例，再对本页进行按块读操作*/
        bio = mpage_bio_submit(READ, bio);
    if (!PageUptodate(page))        /            *缓存页数据无效，按块读缓存页*/
        block_read_full_page(page, get_block);      /*按块读函数*/
    else
```

```
            unlock_page(page);
        goto out;               /*跳至 out 处，返回*/
    }
```

do_mpage_readpage()函数请读者结合前面的介绍和注释自行阅读。

另外需要注意，如果本页是映射到连续的数据块，且参数没有传递 bio 实例，则对本页调用块设备操作结构 block_device_operations 实例（来自 gendisk 实例）中定义的 **rw_page()**函数对本页执行整页的读操作。如果 block_device_operations 实例没有定义 rw_page()函数，则创建 bio 实例并初始化，关联缓存页 page 实例，通过提交 bio 实例完成缓存页的读操作。

3 单页写函数

对块设备进行写操作是一件比较费时的事，因此内核并不是在进程写文件时就立即将其写入到块设备，而是先写入到文件页缓存，在适当的时机再写入到块设备。地址空间操作结构中写缓存页的函数就是用于将缓存页中的数据写入到底层块设备中，在回写和同步机制中将被调用。

大部分文件系统类型地址空间操作结构中写缓存页函数内调用都是通用的 **block_write_full_page()**函数，即对缓存页按缓存块进行写操作。因为按块写可对缓存页的写操作进行更细粒度的控制，没有修改数据的缓存块不需要回写，可提高回写的效率。

在介绍写缓存页函数前，先看一下控制缓存页回写的数据结构（/include/linux/writeback.h）：

```
struct writeback_control {
    long nr_to_write;               /*需要回写的页数，每回写一页数值减 1*/
    long pages_skipped;             /*本次操作中跳过没有执行回写的页数*/

    loff_t range_start;             /*需要回写起止字节在文件内容中的偏移量*/
    loff_t range_end;

    enum writeback_sync_modes   sync_mode;    /*回写同步模式，/include/linux/writeback.h*/

    unsigned for_kupdate:1;         /* A kupdate writeback，周期回写*/
    unsigned for_background:1;      /* A background writeback，脏页平衡回写*/
    unsigned tagged_writepages:1;   /* tag-and-write to avoid livelock */
    unsigned for_reclaim:1;         /* Invoked from the page allocator ，由页回收发起的回写*/
    unsigned range_cyclic:1;        /*回写整个页缓存，不受 range_start 和 range_end 控制*/
    unsigned for_sync:1;            /* sync(2) WB_SYNC_ALL writeback */

#ifdef CONFIG_CGROUP_WRITEBACK      /*回写控制组*/
    struct bdi_writeback *wb;     /* wb this writeback is issued under */
    struct inode *inode;        /* inode being written out */

    /* foreign inode detection, see wbc_detach_inode() */
    int wb_id;                  /* current wb id */
    int wb_lcand_id;            /* last foreign candidate wb id */
    int wb_tcand_id;            /* this foreign candidate wb id */
    size_t wb_bytes;            /* bytes written by current wb */
    size_t wb_lcand_bytes;          /* bytes written by last candidate */
    size_t wb_tcand_bytes;          /* bytes written by this candidate */
#endif
```

};
writeback_control 结构体主要用于对文件内容回写的的控制，其主要成员简介如下：
●**range_start，range_end**：回写文件内容的起止字节偏移量。
●**sync_mode**：同步模式，由枚举类型 writeback_sync_modes 表示（/include/linux/writeback.h）：

```
enum writeback_sync_modes {
    WB_SYNC_NONE,     /*不等待任何事，异步写，Don't wait on anything */
    WB_SYNC_ALL,      /*等待映射全部建立，  Wait on every mapping */
};
```

●**range_cyclic**：标记位，如果置位将回写整个页缓存，而不受 range_start 和 range_end 成员的控制。

按缓存块回写缓存页的函数 **block_write_full_page()**定义在/fs/buffer.c 文件内：

```
int block_write_full_page(struct page *page, get_block_t *get_block,struct writeback_control *wbc)
/*page：缓存页 page 实例指针，get_block：获取映射函数指针，wbc：回写控制结构实例指针*/
{
    struct inode * const inode = page->mapping->host;     /*文件 inode 实例指针*/
    loff_t i_size = i_size_read(inode);            /*现有文件大小*/
    const pgoff_t end_index = i_size >> PAGE_CACHE_SHIFT;   /*当前文件内容最后页编号*/
    unsigned offset;

    /*如果所写缓存页在现有文件内部，不需要扩展文件内容*/
    if (page->index < end_index)
        return __block_write_full_page(inode, page, get_block, wbc,end_buffer_async_write);
                                                                    /*回写缓存页*/

    /*所写缓存页在现有文件内容之外，需要扩展文件内容*/
    offset = i_size & (PAGE_CACHE_SIZE-1);
    if (page->index >= end_index+1 || !offset) {     /*只能紧接现有文件末尾写，不能有空洞？？？ */
        do_invalidatepage(page, 0, PAGE_CACHE_SIZE);
                                        /*调用 a_ops->invalidatepage()，无效缓存页*/
        unlock_page(page);
        return 0;       /* don't care */
    }

    zero_user_segment(page, offset, PAGE_CACHE_SIZE);   /*清零缓存页，/include/linux/highmem.h*/
    return    __block_write_full_page(inode, page, get_block, wbc,end_buffer_async_write);
                                                                    /*回写缓存页*/
}
```

缓存页 page 若位于文件内容之内，则调用__block_write_full_page()函数进行回写。如果位于文件内容之外（需紧接在当前文件内容之后？），则先对缓存页清零，再调用__block_write_full_page()函数进行回写。

■写单页函数

下面看一下回写单个缓存页的__block_write_full_page()函数的定义，如下（/fs/buffer.c）：

```
static int    __block_write_full_page(struct inode *inode, struct page *page,     \
                        get_block_t *get_block, struct writeback_control *wbc,bh_end_io_t *handler)
```

```
/*handler：块缓存头实例回调函数，这里为 end_buffer_async_write()*/
{
    int err;
    sector_t block;
    sector_t last_block;
    struct buffer_head *bh, *head;
    unsigned int blocksize, bbits;
    int nr_underway = 0;
    int write_op = (wbc->sync_mode == WB_SYNC_ALL ? WRITE_SYNC : WRITE);

    head = create_page_buffers(page, inode,(1 << BH_Dirty)|(1 << BH_Uptodate));
                    /*创建块缓存头实例链表，回写前数据已在缓存页中，所以设置脏和有效位*/
    bh = head;                  /*块缓存头实例链表头*/
    blocksize = bh->b_size;                 /*缓存块大小*/
    bbits = block_size_bits(blocksize);     /*缓存块大小值以 2 为底取对数*/

    block = (sector_t)page->index << (PAGE_CACHE_SHIFT - bbits);    /*缓存页内起始缓存块号*/
    last_block = (i_size_read(inode) - 1) >> bbits;       /*文件内容最末尾缓存块号*/

    do {                    /*遍历块缓存头实例链表，获取缓存块映射信息*/
        if (block > last_block) {           /*需要扩展文件内容*/
            clear_buffer_dirty(bh);         /*清块缓存头实例脏标记位*/
            set_buffer_uptodate(bh);        /*设置块缓存头实例数据有效标记位*/
        } else if ((!buffer_mapped(bh) || buffer_delay(bh)) &&buffer_dirty(bh)) {    /*未建立映射*/
            WARN_ON(bh->b_size != blocksize);       /*bh->b_size 必须为数据块大小值*/
            err = get_block(inode, block, bh, 1);       /*获取缓存块映射信息，成功返回 0*/
            if (err)
                goto recover;     /*如果建立映射失败跳至 recover*/
            clear_buffer_delay(bh);
            if (buffer_new(bh)) {
                clear_buffer_new(bh);
                unmap_underlying_metadata(bh->b_bdev,bh->b_blocknr);
            }
        }
        bh = bh->b_this_page;     /*下一块缓存头*/
        block++;
    } while (bh != head);           /*遍历块缓存头实例链表结束，缓存块映射建立*/

    do {        /*遍历块缓存头，设置标记位*/
        if (!buffer_mapped(bh))
            continue;
        if (wbc->sync_mode != WB_SYNC_NONE) {
            lock_buffer(bh);
        } else if (!trylock_buffer(bh)) {
            redirty_page_for_writepage(wbc, page);
                                /*重置页脏标记，本次跳过此页不回写，/mm/page-writeback.c*/
```

```
                continue;
            }
            if (test_clear_buffer_dirty(bh)) {     /*检测并清零脏标记位*/
                mark_buffer_async_write_endio(bh, handler);     /*/fs/buffer.c*/
                        /*设置异步写标记位，并设置块缓存头实例回调函数为 handler*/
            } else {
                unlock_buffer(bh);
            }
        } while ((bh = bh->b_this_page) != head);

        BUG_ON(PageWriteback(page));
        set_page_writeback(page);          /*设置缓存页回写标记位，/include/linux/page-flags.h*/

        do {        /*遍历块缓存头执行写操作*/
            struct buffer_head *next = bh->b_this_page;
            if (buffer_async_write(bh)) {              /*设置了异步写标记*/
                submit_bh_wbc(write_op, bh, 0, wbc);   /*写缓存块，见本章上文*/
                nr_underway++;
            }
            bh = next;
        } while (bh != head);
        unlock_page(page);

        err = 0;
done:
        if (nr_underway == 0) {
            end_page_writeback(page);
        }
        return err;            /*函数返回，成功返回 0*/

recover:        /*get_block()函数获取任一缓存块映射信息失败，跳至此处*/
        bh = head;
        do {
            if (buffer_mapped(bh) && buffer_dirty(bh) &&!buffer_delay(bh)) {
                lock_buffer(bh);
                mark_buffer_async_write_endio(bh, handler);
            } else {
                clear_buffer_dirty(bh);
            }
        } while ((bh = bh->b_this_page) != head);
        SetPageError(page);              /*设置缓存页错误标记位*/
        BUG_ON(PageWriteback(page));
        mapping_set_error(page->mapping, err);
        set_page_writeback(page);
        do {                             /*逐块进行写操作*/
            struct buffer_head *next = bh->b_this_page;
```

```
            if (buffer_async_write(bh)) {
                clear_buffer_dirty(bh);
                submit_bh_wbc(write_op, bh, 0, wbc);
                nr_underway++;
            }
            bh = next;
        } while (bh != head);
        unlock_page(page);
        goto done;
    }
```

__block_write_full_page()函数简单地说就是为缓存页创建块缓存头实例链表，并获取映射信息，随后对缓存块进行锁定操作，如果所有缓存块锁定成功则对缓存页按块进行异步写操作，锁定不成功则设置缓存页脏标记位，跳过此页。

写缓存页的函数 submit_bh_wbc()在前面介绍过了，这里就不讲解了。在执行回写的过程中，内核会锁定缓存块并设置其异步写标记位，缓存页也会设置回写标记位，因此在回写执行完之后需要解锁缓存块，清除异步写和回写标记，这些工作由 **end_buffer_async_write()**函数完成，此函数在在 bio 完成后的回调函数中被调用。

■写完回调函数

block_write_full_page()函数中将 buffer_head 实例中回调函数设为 end_buffer_async_write()，这在 bio 数据写入完成时调用。

end_buffer_async_write()函数代码简列如下（/fs/buffer.c）：

```
void end_buffer_async_write(struct buffer_head *bh, int uptodate)
{
    unsigned long flags;
    struct buffer_head *first;
    struct buffer_head *tmp;
    struct page *page;

    BUG_ON(!buffer_async_write(bh));

    page = bh->b_page;
    if (uptodate) {          /*缓存页数据有效，已经回写*/
        set_buffer_uptodate(bh);     /*设置缓存块数据有效标记位*/
    } else {             /*回写失败*/
        buffer_io_error(bh, ", lost async page write");
        set_bit(AS_EIO, &page->mapping->flags);
        set_buffer_write_io_error(bh);
        clear_buffer_uptodate(bh);
        SetPageError(page);
    }

    first = page_buffers(page);        /*第一个块缓存头实例*/
    local_irq_save(flags);
    bit_spin_lock(BH_Uptodate_Lock, &first->b_state);
```

```
        clear_buffer_async_write(bh);       /*清异步写标记位*/
        unlock_buffer(bh);                  /*解锁缓存块*/
        tmp = bh->b_this_page;
        while (tmp != bh) {       /*检测缓存页所有缓存块是否都回写完毕，是则清除缓存页回写标记*/
            if (buffer_async_write(tmp)) {
                BUG_ON(!buffer_locked(tmp));
                goto still_busy;
            }
            tmp = tmp->b_this_page;
        }
        bit_spin_unlock(BH_Uptodate_Lock, &first->b_state);
        local_irq_restore(flags);
        end_page_writeback(page);   /*清回写标记位，唤醒等待回写完成进程等，/mm/filemap.c*/
        return;
        ...
    }
```

end_buffer_async_write()函数清除本缓存块的异步写标记位，并解锁缓存块，然后判断缓存页所有的缓存块是否都执行完了回写操作，如果是则调用 end_page_writeback(page)函数清除缓存页的回写标记位，并唤醒等待缓存页回写的进程。如果缓存页中还有缓存块没有执行完回写，end_buffer_async_write()函数直接返回。

end_page_writeback(page)函数定义在/mm/filemap.c 文件内，代码如下：

```
    void end_page_writeback(struct page *page)
    {
        if (PageReclaim(page)) {       /*如果页正在被回收，回写完成后清该标记位*/
            ClearPageReclaim(page);
            rotate_reclaimable_page(page);
        }

        if (!test_clear_page_writeback(page))   /*清回写标记位*/
            BUG();

        smp_mb__after_atomic();
        wake_up_page(page, PG_writeback);   /*唤醒等待页回写完成的进程*/
    }
```

回写单个缓存页的函数，除 block_write_full_page()外，还有按页写缓存页的 mpage_writepage()函数，如同按页读缓存页的 mpage_readpage()函数，但似乎内核中没有文件系统类型采用此函数，源代码请读者自行阅读（/fs/mpage.c）。

4 页缓存回写函数

文件地址空间操作结构中包含回写整个页缓存中脏页的 writepages()函数指针成员，内核定义了通用函数 mpage_writepages()和 generic_writepages()，用于回写页缓存中所有的脏页至块设备。

在介绍函数实现前，先看一下 mpage_data 结构体的定义（/fs/mpage.c）：

```
    struct mpage_data {
```

```
    struct bio *bio;
    sector_t last_block_in_bio;
    get_block_t *get_block;          /*获取映射信息函数*/
    unsigned use_writepage;          /*是否使用 mapping->a_ops->writepage()函数写单页*/
};
```

mpage_writepages()函数定义在/fs/mpage.c 文件内，用于回写指定文件内容区域的脏页或回写所有脏页，代码如下：

```
int mpage_writepages(struct address_space *mapping,struct writeback_control *wbc, get_block_t get_block)
{
    struct blk_plug plug;          /*进程用于缓存请求 request 实例的链表，见第 10 章*/
    int ret;

    blk_start_plug(&plug);     /*tsk->plug = plug，blk_plug 实例赋予当前进程，/block/blk-core.c*/

    if (!get_block)
        ret = generic_writepages(mapping, wbc); /*没有传递 get_block()映射函数，调用通用函数*/
    else {
        struct mpage_data mpd = {
            .bio = NULL,
            .last_block_in_bio = 0,
            .get_block = get_block,
            .use_writepage = 1,
        };

        ret = write_cache_pages(mapping, wbc, __mpage_writepage, &mpd);
            /*回写页缓存中所有脏页，对每页调用__mpage_writepage()函数，/mm/page-writeback.c*/
        if (mpd.bio)
            mpage_bio_submit(WRITE, mpd.bio);   /*提交最后按页写的 bio 实例*/
    }
    blk_finish_plug(&plug);       /*current->plug = NULL，/block/blk-core.c*/
    return ret;
}
```

blk_plug 结构体用于进程缓存提交的请求，blk_plug 实例在回写前赋予当前进程，回写后清除，详见第 10 章。

mpage_writepages()函数中，如果参数传递了 get_block()函数指针参数，则调用 **write_cache_pages()**函数完成页缓存回写操作，否则调用 generic_writepages()函数执行回写。

下面先看 write_cache_pages()函数的实现，函数定义在/mm/page-writeback.c 文件内：

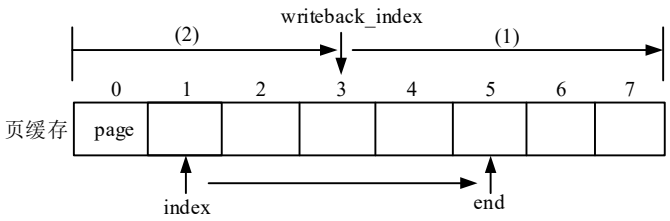
```
int write_cache_pages(struct address_space *mapping,      \
                      struct writeback_control *wbc, writepage_t writepage,void *data)
/*wbc：回写控制结构实例指针，writepage：缓存页回写函数，data：mpage_data 实例指针*/
{
    int ret = 0;
    int done = 0;
    struct pagevec pvec;       /*页向量，page 实例指针数组，用于缓存 page 实例*/
```

```
    int nr_pages;
    pgoff_t uninitialized_var(writeback_index);
    pgoff_t index;
    pgoff_t end;          /* Inclusive */
    pgoff_t done_index;
    int cycled;
    int range_whole = 0;
    int tag;

    pagevec_init(&pvec, 0);     /*初始化页向量*/
    if (wbc->range_cyclic) {     /*如果设置了 range_cyclic 标记位，需要分段扫描，回写整个页缓存*/
                                 /*扫描分两段进行分别是[writeback_index，end]和[0,writeback_index)*/
        writeback_index = mapping->writeback_index;   /*上次扫描结束页索引值，本次的起点*/
        index = writeback_index;
        if (index == 0)
            cycled = 1;     /*index 为 0 则不需要分段扫描*/
        else
            cycled = 0;
        end = -1;
    } else {          /*回写 writeback_control 实例中指定文件内容区域的脏页*/
        index = wbc->range_start >> PAGE_CACHE_SHIFT;      /*起始页编号*/
        end = wbc->range_end >> PAGE_CACHE_SHIFT;          /*结束页编号*/
        if (wbc->range_start == 0 && wbc->range_end == LLONG_MAX)
            range_whole = 1;          /*表示回写整个地址空间缓存页*/
        cycled = 1;                   /*只扫描指定区间*/
    }

    if (wbc->sync_mode == WB_SYNC_ALL || wbc->tagged_writepages)
        tag = PAGECACHE_TAG_TOWRITE;      /*页缓存基数树标记，写出标记*/
    else
        tag = PAGECACHE_TAG_DIRTY;          /*脏标记*/

retry:      /*如果是分两段扫描，retry 之后代码要运行两遍*/
    if (wbc->sync_mode == WB_SYNC_ALL || wbc->tagged_writepages)
        tag_pages_for_writeback(mapping, index, end);     /*/mm/page-writeback.c*/
            /*设置基树数[index,end]区间脏页的 PAGECACHE_TAG_TOWRITE 标记*/
    done_index = index;          /*起始缓存页编号*/

    while (!done && (index <= end)) {        /*遍历基数树中缓存页*/
        int i;
        nr_pages = pagevec_lookup_tag(&pvec, mapping, &index, tag,
                        min(end - index, (pgoff_t)PAGEVEC_SIZE-1) + 1);     /*/mm/swap.c*/
                        /*在基数树中查找 tag 标记的页，保存至页向量 pvec*/
        if (nr_pages == 0)
            break;
```

```
        for (i = 0; i < nr_pages; i++) {        /*遍历页向量*/
            struct page *page = pvec.pages[i];
            if (page->index > end) {
                done = 1;
                break;
            }
            done_index = page->index;     /*缓存页编号*/
            lock_page(page);     /*锁定页*/

            if (unlikely(page->mapping != mapping)) {
continue_unlock:
                unlock_page(page);
                continue;
            }

            if (!PageDirty(page)) {       /*脏标记被清除，不需要回写*/
                goto continue_unlock;
            }

            if (PageWriteback(page)) {       /*如果缓存页正在被其它进程回写*/
                if (wbc->sync_mode != WB_SYNC_NONE)
                    wait_on_page_writeback(page);       /*等待回写完成*/
                else
                    goto continue_unlock;
            }

            BUG_ON(PageWriteback(page));
            if (!clear_page_dirty_for_io(page))     /*清缓存页脏标记位，返回原脏标记位取值*/
                goto continue_unlock;

            trace_wbc_writepage(wbc, inode_to_bdi(mapping->host));
            ret = (*writepage)(page, wbc, data);
                        /*执行缓存页的回写操作，参数传递的函数指针，成功返回 0*/
            if (unlikely(ret)) {
                if (ret == AOP_WRITEPAGE_ACTIVATE) {
                    unlock_page(page);
                    ret = 0;
                } else {
                    done_index = page->index + 1;
                    done = 1;
                    break;
                }
            }
            if (--wbc->nr_to_write <= 0 &&wbc->sync_mode == WB_SYNC_NONE) {
                done = 1;
                break;
```

```
                }
            }    /*遍历页向量结束*/
            pagevec_release(&pvec);      /*释放页向量中页*/
            cond_resched();          /*条件调度*/
        }   /*遍历页缓存结束*/

        /*至此已经扫描了指定区域或 writeback_index 之后的页缓存基数树*/
        if (!cycled && !done) {     /*如果 cycled=0 且 done=0，扫描页缓存前段*/
            cycled = 1;
            index = 0;
            end = writeback_index - 1;          /*结束页编号*/
            goto retry;         /*再执行一遍*/
        }

        if (wbc->range_cyclic || (range_whole && wbc->nr_to_write > 0))
            mapping->writeback_index = done_index;      /*设置 writeback_index 成员值*/

        return ret;       /*执行成功返回 0*/
    }
```

write_cache_pages()函数的执行流程不难理解，如下图所示：



（1）如果是回写整个页缓存中脏页，则分两段进行，第一段从 mapping->writeback_index 指示页开始至地址空间页缓存最后页，收集脏缓存页，调用 writepage()参数传递的函数逐个回写页，第二段回写从 0 页至 mapping->writeback_index-1 页的脏页。

（2）如果是回写页缓存中[index,end]区域脏缓存页，则只扫描这个区域的脏页，调用 writepage()函数回写。

write_cache_pages()函数内定义了页向量 pagevec 结构体实例 pvec 用于暂存扫描到的需要回写的缓存页。页向量填满时，则对其中各缓存页调用 writepage()函数（参数传递的函数指针）执行回写操作，回写完成后释放页向量中缓存页，继续扫描页缓存中下一批需要回写的缓存页，如此循环，直至扫描完页缓存回写区间。

mpage_writepages()函数内调用 write_cache_pages()函数传递的 writepage 参数为**__mpage_writepage()**函数指针。__mpage_writepage()函数定义在/fs/mpage.c 文件内，它与读缓存页函数中的 do_mpage_readpage()函数类似，也是用于处理单个缓存页。

__mpage_writepage()函数中只有缓存页中所有缓存块都是脏的需要回写时，才执行按页写，否则调用 mapping->a_ops->writepage(page, wbc)函数执行单页写操作（须 mpd.use_writepage = **1**）。按页写时也会合并映射到连续数据块的相邻缓存页至同一个 bio 实例。__mpage_writepage()函数源代码请读者自行阅读。

另外，如果 mpage_writepages()函数没有传递 get_block()函数指针参数，将调用 **generic_writepages()**函数执行回写操作，而不调用 write_cache_pages()函数。

generic_writepages()函数定义在/mm/page-writeback.c，其内部也调用 write_cache_pages()函数，如下所

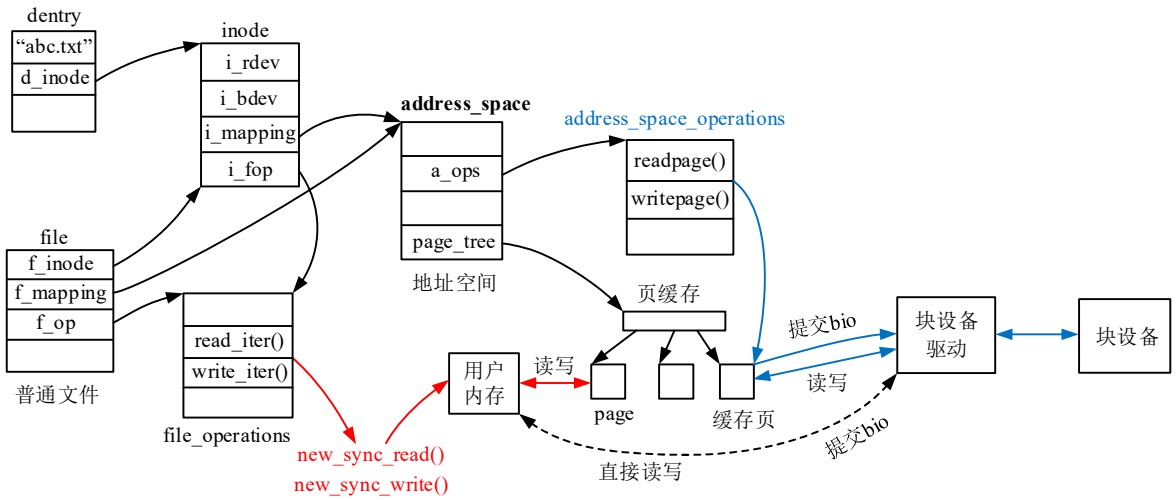
示：

```
int generic_writepages(struct address_space *mapping,struct writeback_control *wbc)
{
    struct blk_plug plug;
    int ret;

    /* deal with chardevs and other special file */
    if (!mapping->a_ops->writepage)
        return 0;

    blk_start_plug(&plug);
    ret = write_cache_pages(mapping, wbc, __writepage, mapping);
    blk_finish_plug(&plug);
    return ret;
}
```

在这里 write_cache_pages()函数内调用__writepage()函数回写单页（脏页），__writepage()函数调用地址空间操作结构中的 a_ops->writepage(page, wbc)函数回写单页，请读者自行阅读源代码。

11.2 通用读写文件函数

用户进程对块设备中文件的读写访问流程如下图所示：



通常情况下文件操作 file_operations 实例中的读写函数调用通用的 new_sync_read()/new_sync_write()函数（同步读写函数）从页缓存中读数据至用户内存，或将用户内存中数据写入页缓存，然后由内核调用地址空间操作结构中的函数实现页缓存与块设备的数据传输。通用函数是各文件系统类型通用的，各具体文件系统类型定义需要定义的是地址空间操作结构实例。

内核也提供了不经过页缓存，在用户内存与块设备之间直接实现数据传输的机制，称为直接读写操作，不过此种方式限制较多，效率也低，并不常用。

每次读写操作只能对文件内容中一段连续区域进行读写，但是在用户内存中可以用多段离散的内存来保存这段连续的文件内容。

本节介绍内核提供的通用读写文件函数的实现。

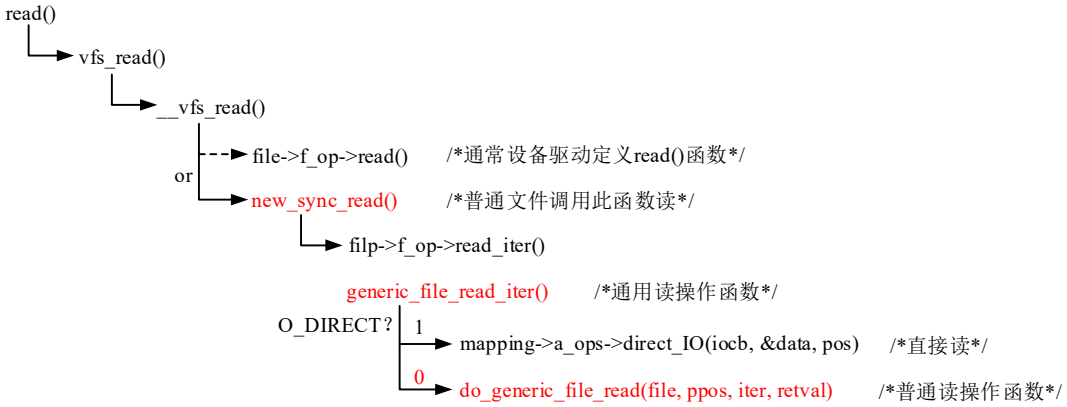
11.2.1 普通读文件函数

内核定义了通用的读文件函数，通常是从文件页缓存中读数据至用户内存，若缓存页不存在或数据无

效，则调用地址空间的读函数从块设备中读数据至缓存页，然后再复制到用户内存。如果是直接读操作，则直接调用地址空间操作结构中的直接读写函数实现。

1 通用读文件函数

读文件内容的 read()系统调用，函数调用关系如下图所示（详见第 7 章）：



如果 file_operations 实例定义了 read()函数，则调用它执行读操作（设备文件通常定义此函数），否则将调用 new_sync_read()函数执行读操作（同步读函数）。

内核大部分的文件系统类型 file_operations 实例中都没有定义 read()函数而只定义 **read_iter()**函数，也就是说读文件操作将由 new_sync_read()函数完成，这是一个通用的函数。

new_sync_read()函数内调用 file_operations 实例中的 **read_iter()**函数完成读操作。内核大多数文件系统类型定义的 file_operations 实例中 **read_iter()**函数为通用的 **generic_file_read_iter()**函数，此函数判断进程是以普通方式还是直接方式读取文件内容。

如果是以普通方式读取文件内容，则调用函数 **do_generic_file_read()**执行基于页缓存的读操作。如果是直接读方式，则调用地址空间操作结构定义的 direct_IO()函数执行直接读操作。

下面将介绍 generic_file_read_iter()函数的实现，直接读操作后面会专门介绍。

■数据结构

在介绍具体函数的实现之前，先介绍几个相关数据结构的定义。

iovec 和 iov_iter 结构体用于收集保存文件内容的用户内存信息，定义如下（/include/uapi/linux/uio.h）：

```
struct iovec                    /*表示一个内存段*/
{
    void __user   *iov_base;      /*指向起始内存地址，用户空间虚拟地址*/
    __kernel_size_t   iov_len;    /*内存段长度*/
};

struct iov_iter {               /*/include/linux/uio.h*/
    int   type;                 /*表示结构体中联合体成员的类型*/
    size_t iov_offset;          /*用户内存偏移量，初始值为 0*/
    size_t count;               /*用户内存中尚未处理的数据长度，字节数*/
        /*读操作表示尚未从页缓存读入的数据长度，写操作表示尚未写入页缓存的数据长度*/
    union {                     /*联合体，可指向以下数据结构中的一种*/
        const struct iovec *iov;        /*指向 iovec 实例（数组），type==ITER_IOVEC*/
        const struct kvec *kvec;        /*type==ITER_KVEC*/
        const struct bio_vec *bvec;     /*type==ITER_BVEC*/
```

```
    };
    unsigned long nr_segs;        /*联合体成员指向数据结构数组的项数*/
};
```

iov_iter 结构体用于由收集用户内存段信息，联合体成员可指向 iovec、kvec 或 bio_vec 结构体（数组）用于具体表示内存段信息。读写文件操作中通常使用 iovec 结构体表示内存段信息，如果有多个内存段，则 iov 指向 iovec 结构体数组。

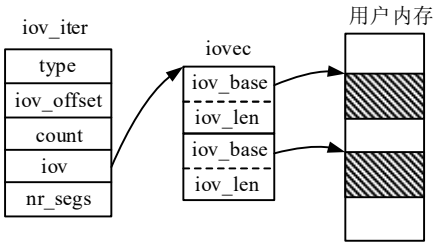
iov_iter 结构体中 type 成员表示联合体成员的类型，由枚举类型定义（/include/linux/uio.h）：

```
enum {
    ITER_IOVEC = 0,          /*联合体成员指向 iovec 实例（数组）*/
    ITER_KVEC = 2,
    ITER_BVEC = 4,
};
```

初始化 iov_iter 实例的函数定义在/lib/iov_iter.c 文件内，代码如下：

```
void iov_iter_init(struct iov_iter *i, int direction, \
                        const struct iovec *iov, unsigned long nr_segs,size_t count)
/*iov：指向 iovec 数组，nr_segs：iovec 数组项数，count：字节数量*/
{
    if (segment_eq(get_fs(), KERNEL_DS)) {
        direction |= ITER_KVEC;
        i->type = direction;
        i->kvec = (struct kvec *)iov;
    } else {
        i->type = direction;        /*联合体成员类型，枚举类型值*/
        i->iov = iov;               /*指向 iovec 实例（数组）*/
    }
    i->nr_segs = nr_segs;       /*数组项数*/
    i->iov_offset = 0;          /*用户内存中的偏移量，各用户内存段视为逻辑上连续的*/
    i->count = count;           /*尚未处理数据的长度，每次操作后会减小，为 0 表示操作完成*/
}
```

iov_iter 实例初始化的结果如下图所示，iov_iter 结构体与块设备驱动中的 bio 结构体比较类似，iov 成员指向 iovec 结构体数组，可包含多段用户内存信息。



kiocb 结构体用于表示进程发起 IO 操作的状态/控制信息，定义如下（/include/linux/fs.h）：

```
struct kiocb {
    struct file     *ki_filp;       /*文件 file 实例指针*/
    loff_t          ki_pos;         /*文件内容起始位置*/
    void (*ki_complete)(struct kiocb *iocb, long ret, long ret2);     /*同步操作为 NULL*/
    void            *private;
    int             ki_flags;       /*标记成员*/
```

```
};
```

kiocb 结构体主要成员语义如下：

●**ki_filp**：读写文件的 file 实例指针。

●**ki_pos**：文件操作在文件内容中的起始位置（字节偏移量）。

●**ki_flags**：标记成员，取值定义如下：

```
#define   IOCB_EVENTFD        (1 << 0)
#define   IOCB_APPEND         (1 << 1)    /*写文件操作将数据写在文件末尾*/
#define   IOCB_DIRECT         (1 << 2)    /*直接读/写文件操作，不经过页缓存*/
```

初始化 kiocb 实例（同步操作）的函数定义如下（/include/linux/fs.h）：

```
static inline void init_sync_kiocb(struct kiocb *kiocb, struct file *filp)
{
    *kiocb = (struct kiocb) {       /*设置 kiocb 指向的结构体实例*/
        .ki_filp = filp,            /*文件 file 指针*/
        .ki_flags = iocb_flags(filp),       /*设置标记成员值，/include/linux/fs.h*/
    };
}
```

iocb_flags()函数用于依 file 实例，确定 kiocb 结构体 ki_flags 标记成员值，代码如下（/include/linux/fs.h）：

```
static inline int iocb_flags(struct file *file)
{
    int res = 0;
    if (file->f_flags & O_APPEND)   /*数据写到文件末尾，file->f_flags 在打开文件时设置*/
        res |= IOCB_APPEND;
    if (io_is_direct(file))                 /*打开文件设置了 O_DIRECT 标记位，/include/linux/fs.h*/
        res |= IOCB_DIRECT;         /*直接读写操作，不经过页缓存*/
    return res;
}
```

iocb_flags()函数根据 file->f_flags 成员值确定 ki_flags 标记成员值，file->f_flags 成员在打开文件的 open()系统调用中设置，详见第 7 章。

■通用读函数

大部分文件系统类型 file_operations 实例中都没有定义 read()函数而只定义 **read_iter()**函数，因此 read()系统调用通过 **new_sync_read()**函数执行读文件内容操作，这是一个通用的函数。

new_sync_read()函数定义在/fs/read_write.c 文件内，代码如下：

```
static ssize_t new_sync_read(struct file *filp, char __user *buf, size_t len, loff_t *ppos)
/*filp：文件 file 实例指针，buf：用户内存地址，len：读取文件内容长度，*ppos：文件位置*/
{
    struct iovec iov = { .iov_base = buf, .iov_len = len };      /*初始化 iovec 实例，指向用户内存*/
    struct kiocb kiocb;         /*kiocb 实例*/
    struct iov_iter iter;       /*iov_iter 实例*/
    ssize_t ret;

    init_sync_kiocb(&kiocb, filp);      /*初始化 kiocb 实例，同步读*/
    kiocb.ki_pos = *ppos;               /*设置文件位置*/
```

```
    iov_iter_init(&iter, READ, &iov, 1, len);       /*初始化 iov_iter 实例，读操作，只有 1 个内存段*/

    ret = filp->f_op->read_iter(&kiocb, &iter);   /*调用 file_operations 实例中的 read_iter()函数*/
    BUG_ON(ret == -EIOCBQUEUED);
    *ppos = kiocb.ki_pos;     /*重置文件位置，读操作之后会修改此值*/
    return ret;
}
```

new_sync_read()函数内创建 iov_iter 和 kiocb 结构体实例并初始化，然后调用 file_operations 实例中的 **read_iter()**函数完成读文件内容操作。

大多数文件系统类型 file_operations 实例中的 read_iter()函数，都是直接调用 **generic_file_read_iter()** 通用函数，下文将介绍此函数的实现。

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

```
ssize_t generic_file_read_iter(struct kiocb *iocb, struct iov_iter *iter)
/*iocb：IO 操作控制块，iter：保存收集的用户内存信息*/
{
    struct file *file = iocb->ki_filp;
    ssize_t retval = 0;
    loff_t *ppos = &iocb->ki_pos;       /*ppos 指向 iocb->ki_pos 成员，表示读操作起始文件位置*/
    loff_t pos = *ppos;

    if (iocb->ki_flags & IOCB_DIRECT) {        /*直接读操作，打开文件设置了 O_DIRECT 标记位*/
        struct address_space *mapping = file->f_mapping;
        struct inode *inode = mapping->host;
        size_t count = iov_iter_count(iter);       /*count 成员值，即读取文件长度，字节数*/
        loff_t size;

        if (!count)
            goto out;               /* skip atime */
        size = i_size_read(inode);     /*当前文件大小*/
        retval = filemap_write_and_wait_range(mapping, pos,pos + count - 1);     /*页缓存回写*/
                                        /*等待欲读文件内容回写完成，成功返回 0，/mm/filemap.c*/
        if (!retval) {
            struct iov_iter data = *iter;     /*复制结构体实例*/
            retval = mapping->a_ops->direct_IO(iocb, &data, pos);
                                    /*调用地址空间操作结构中的直接读写操作函数*/
        }

        if (retval > 0) {
            *ppos = pos + retval;       /*读操作后重置文件位置*/
            iov_iter_advance(iter, retval);     /*count 值减 retval，表示未读取的数据减少*/
        }
        if (retval < 0 || !iov_iter_count(iter) || *ppos >= size || IS_DAX(inode)) {
            file_accessed(file);
            goto out;
        }
```

```
            /*直接读操作未处理完的数据由下面的普通读操作完成*/
        }
        retval = do_generic_file_read(file, ppos, iter, retval);    /*普通读操作函数，/mm/filemap.c*/
out:
        return retval;
}
```

generic_file_read_iter()函数判断进程是采取直接读的方式还是普通读的方式读取文件内容。

如果是直接读方式，则先等待页缓存中欲读文件内容回写完成，然后调用地址空间操作结构实例中的direct_IO()函数，将块设备中数据直接读入到用户内存，后面将详细介绍。

如果是普通读操作（通过页缓存读），则调用 do_generic_file_read()函数逐页从页缓存中复制数据到用户内存，如果缓存页不存在或数据无效则需要先从块设备中读取数据至缓存页，然后执行复制操作。

下面将介绍执行普通读操作的 do_generic_file_read()函数实现。

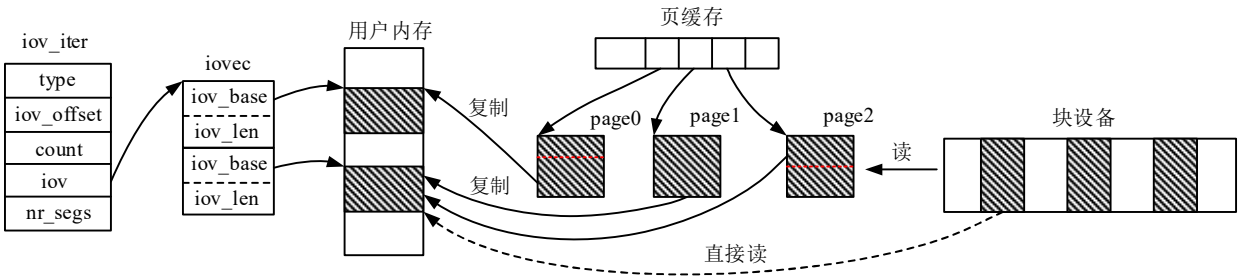
2 普通读函数

对于普通读操作，generic_file_read_iter()函数将调用 do_generic_file_read()函数执行读操作。普通读操作流程如下图所示，为不失一般性，图中假设读取文件内容的起止位置都没有页对齐，读取内容跨越了 3 个缓存页，用户进程用两个内存段来保存读取的内容。

读操作逐个对缓存页执行复制操作，图中首先复制缓存页 0 中的后半部分数据至用户内存，然后复制整个缓存页 1 的数据至用户内存，最后复制缓存页 2 的前半部分数据至用户内存。如果缓存页尚不存在（或数据无效），则创建缓存页并从块设备中读取数据至缓存页，再执行数据复制操作。

两段用户内存被视为逻辑上连续的，复制数据到用户内存时，是按顺序依次存放到用户内存的。

为提高读取效率，内核使用了预读机制，即如果所需缓存页不存在，则从块设备中读取当前页及其后若干页的数据至页缓存，因为进程对文件内容顺序访问的概率较大。



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

```
static ssize_t do_generic_file_read(struct file *filp, loff_t *ppos,struct iov_iter *iter, ssize_t written)
{
    struct address_space *mapping = filp->f_mapping;    /*地址空间实例*/
    struct inode *inode = mapping->host;            /*文件 inode 实例*/
    struct file_ra_state *ra = &filp->f_ra;         /*预读控制结构*/
    pgoff_t index;
    pgoff_t last_index;
    pgoff_t prev_index;
    unsigned long offset;           /*文件位置，页对齐*/
    unsigned int prev_offset;
    int error = 0;

    index = *ppos >> PAGE_CACHE_SHIFT;    /*欲读第一个缓存页编号*/
```

```
prev_index = ra->prev_pos >> PAGE_CACHE_SHIFT;     /*上次读操作最后字节位置*/
prev_offset = ra->prev_pos & (PAGE_CACHE_SIZE-1);
last_index = (*ppos + iter->count + PAGE_CACHE_SIZE-1) >> PAGE_CACHE_SHIFT;
                                                    /*欲读取内容最后一个缓存页编号*/
offset = *ppos & ~PAGE_CACHE_MASK;       /*文件当前位置，页对齐*/

for (;;) {     /*循环遍历欲读缓存页，复制数据至用户内存，需要时先从块设备读数据至页缓存*/
      struct page *page;
      pgoff_t end_index;
      loff_t isize;
      unsigned long nr, ret;

      cond_resched();
find_page:
      page = find_get_page(mapping, index);     /*查找缓存页，index 缓存页编号*/
      if (!page) {                  /*如果缓存页不存在，则执行同步预读操作后再查找*/
          page_cache_sync_readahead(mapping,ra, filp,index, last_index - index);
                                                /*同步预读所需页及其随后的若干页，见下文*/
          page = find_get_page(mapping, index);     /*再次查找*/
          if (unlikely(page == NULL))
              goto no_cached_page;     /*预读不成功，缓存页不存在，跳至 no_cached_page*/
      }
      /*缓存页存在且设置了 Readahead 标记位，则启动异步预读*/
      if (PageReadahead(page)) {
          page_cache_async_readahead(mapping,ra, filp, page,index, last_index - index);
                                                    /*异步文件预读，见下文*/
      }
      if (!PageUptodate(page)) {       /*如果缓存页存在但数据无效*/
          if (inode->i_blkbits == PAGE_CACHE_SHIFT ||!mapping->a_ops->is_partially_uptodate)
              goto page_not_up_to_date;
          if (!trylock_page(page))
              goto page_not_up_to_date;
          if (!page->mapping)
              goto page_not_up_to_date_locked;
          if (!mapping->a_ops->is_partially_uptodate(page,offset, iter->count))
              goto page_not_up_to_date_locked;          /*跳转至读缓存页处*/
        unlock_page(page);
      }

page_ok:        /*缓存页存在且数据有效，复制数据*/
      isize = i_size_read(inode);          /*文件大小*/
      end_index = (isize - 1) >> PAGE_CACHE_SHIFT;     /*文件内容映射末尾缓存页编号*/
      if (unlikely(!isize || index > end_index)) {
          page_cache_release(page);
          goto out;
      }
```

```
        /* nr is the maximum number of bytes to copy from this page */
        nr = PAGE_CACHE_SIZE;          /*能从缓存页读取的最大数据字节数*/
        if (index == end_index) {
            nr = ((isize - 1) & ~PAGE_CACHE_MASK) + 1;
            if (nr <= offset) {
                page_cache_release(page);
                goto out;
            }
        }
        nr = nr - offset;

        if (mapping_writably_mapped(mapping))
            flush_dcache_page(page);

        if (prev_index != index || offset != prev_offset)
            mark_page_accessed(page);          /*标记缓存页被访问*/
        prev_index = index;

        ret = copy_page_to_iter(page, offset, nr, iter);     /*复制缓存页数据至用户内存，count 减小*/
            /*缓存页映射到持久映射区，数据复制完成后解除映射，返回复制的字节数，/lib/iov_iter.c*/
        offset += ret;
        index += offset >> PAGE_CACHE_SHIFT;
        offset &= ~PAGE_CACHE_MASK;
        prev_offset = offset;

        page_cache_release(page);
        written += ret;          /*写入用户空间内存的字节数*/
        if (!iov_iter_count(iter))     /*count 值为 0 表示复制完成*/
            goto out;                  /*跳至 out 处返回*/
        if (ret < nr) {
            error = -EFAULT;
            goto out;
        }
        continue;       /*处理下一缓存页*/

page_not_up_to_date:     /*缓存页数据无效*/
        error = lock_page_killable(page);
        if (unlikely(error))
            goto readpage_error;

page_not_up_to_date_locked:
        if (!page->mapping) {
            unlock_page(page);
            page_cache_release(page);
            continue;
```

```
            }

            if (PageUptodate(page)) {
                unlock_page(page);
                goto page_ok;
            }

  readpage:     /*从块设备读数据至缓存页*/

            ClearPageError(page);
            error = mapping->a_ops->readpage(filp, page);     /*调用读单个缓存页函数*/
            ...
            if (!PageUptodate(page)) {            /*如果缓存页数据无效*/
                error = lock_page_killable(page);
                if (unlikely(error))
                    goto readpage_error;
                if (!PageUptodate(page)) {
                    if (page->mapping == NULL) {
                        unlock_page(page);
                        page_cache_release(page);
                        goto find_page;
                    }
                    unlock_page(page);
                    shrink_readahead_size_eio(filp, ra);
                    error = -EIO;
                    goto readpage_error;
                }
                unlock_page(page);
            }

            goto page_ok;     /*缓存页数据有效，返回 page_ok 处执行复制操作*/

readpage_error:
            page_cache_release(page);
            goto out;

no_cached_page:        /*缓存页不存在，创建缓存页，再跳至读缓存页处*/
            page = page_cache_alloc_cold(mapping);   /*缓存页不存在，创建缓存页*/
            ...
            error = add_to_page_cache_lru(page, mapping, index,   \
                                    GFP_KERNEL & mapping_gfp_mask(mapping));
                                                /*添加缓存页到基数树和页 LRU 链表*/
            ...
            goto readpage;     /*跳至读缓存页处，执行读操作*/
        }
        /*for 循环遍历欲读缓存页结束*/
```

```
out:    /*从这里开始返回，在预读数据结构实例中设置此次读操作结束时的文件位置，字节数*/
        ra->prev_pos = prev_index;
        ra->prev_pos <<= PAGE_CACHE_SHIFT;
        ra->prev_pos |= prev_offset;

        *ppos = ((loff_t)index << PAGE_CACHE_SHIFT) + offset;     /*重置文件位置*/
        file_accessed(filp);
        return written ? written : error;     /*返回实际写入用户内存的字节数*/
    }
```

do_generic_file_read()函数虽然代码比较长，但是去除错误处理的代码后结构比较简单，流程比较清晰，此函数遍历欲读的缓存页，将缓存页中数据复制到用户内存。

如果缓存页存在且数据有效，则直接执行复制操作。如果缓存页不在页缓存中，将执行同步预读操作后再执行复制操作。如果缓存页设置了预标记位，将启动异步预读操作。如果预读不成功，或缓存页中数据无效，将执行单页读操作，然后执行数据复制操作。

复制完数据后，函数返回实际写入到用户内存的数据字节数，并修改 iocb->ki_pos 成员（ppos 参数指向它）中保存的文件当前位置值。

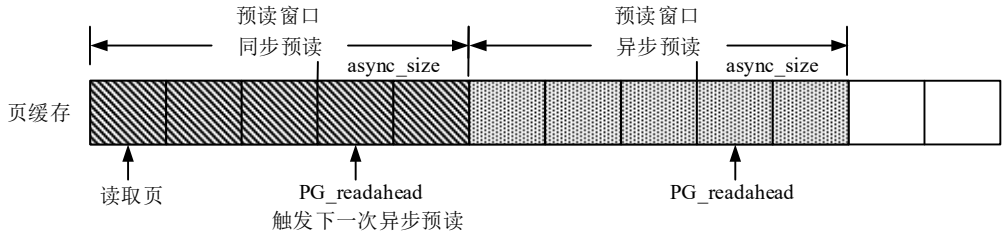
从缓存页复制数据至用户内存的 copy_page_to_iter(page, offset, nr, iter)函数内会减小 iter 参数指向实例中的 count 成员值（复制的字节数），数据按顺序保存到用户内存，当 count 值为 0 时表示读操作完成。

■页缓存预读

下面看一下页缓存预读机制是如何工作的。在读取文件内容的过程中，如果读取缓存页不在页缓存中，读取函数将会先发起同步预读操作，即从块设备中顺序读取当前页及其后若干页（预读窗口）的内容至页缓存，然后再复制缓存页数据至用户内存。如果当前缓存页设置了预读标记位，将启动异步预读。

预读操作分为同步预读和异步预读，执行过程如下图所示。假设需要读取的缓存页不在页缓存中，将触发同步预读，如图中所示预读包含读取页在内的连续若干个缓存页（由预读窗口确定，图中为 5 个），并在某一缓存页中设置 PG_readahead 预读标记位（同 PG_reclaim 标记位，图中为倒数第 2 页），然后再尝试从欲读取缓存页复制数据。

当读文件操作访问到设置了 PG_readahead 标记位的缓存页时，说明剩下的可用缓存页不多了，这时将会触发异步预读，提前读取随后的若干个缓存页。异步预读与同步预读类似，先设置预读窗口（需要读取的缓存页数量），执行读取缓存页操作，并在某一页中设置了 PG_readahead 标记位（如果需要），以便触发后续的预读操作。在预读操作中，每次预读的页数（预读窗口）需要根据上一次预读的结果进行计算。



进程文件 file 结构体中 file_ra_state 结构体成员，包含了文件预读窗口的信息，如下所示：

```
file{
    ...
    struct file_ra_state  f_ra;
    ...
}
```

file_ra_state 结构体定义在/include/linux/fs.h 头文件内：

```
struct file_ra_state {
    pgoff_t start;                  /*开始预读位置，缓存页编号*/
    unsigned int size;              /*预读页数量，又称预读窗口，size<=ra_pages*/
    unsigned int async_size;        /*预读窗口中还剩 async_size 页用户未读时，则启动异步预读*/

    unsigned int ra_pages;          /*预读窗口最大页数*/
    unsigned int mmap_miss;         /* Cache miss stat for mmap accesses */
    loff_t prev_pos;                /*在 do_generic_file_read()函数末尾修改此值*/
                                    /*上次普通读操作最后读取的文件位置，字节数*/
};
```

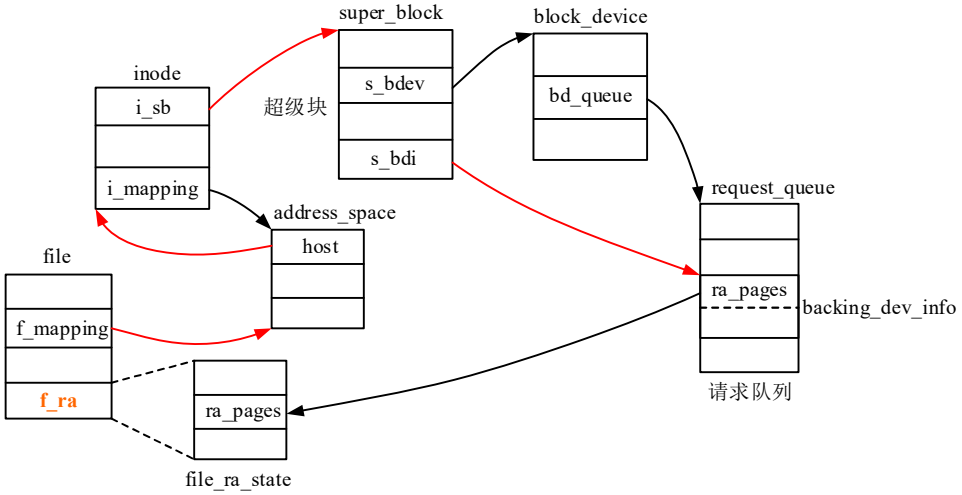
file_ra_state 结构体表示了预读窗口的起始缓存页、窗口长度（缓存页数量）、最大窗口长度等信息。

内核在打开文件的 open()系统调用最后阶段调用的 do_dentry_open()函数内调用 **file_ra_state_init()**函数初始化 file 实例中 file_ra_state 结构体成员。

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

```
void file_ra_state_init(struct file_ra_state *ra, struct address_space *mapping)
{
    ra->ra_pages = inode_to_bdi(mapping->host)->ra_pages;      /*/include/linux/backing-dev.h*/
    ra->prev_pos = -1;
}
```

file_ra_state 实例中 prev_pos 成员初始化为-1，而 ra_pages 成员来自于块设备请求队列 backing_dev_info 结构体成员中的 ra_pages 成员，关系图如下所示。



请求队列中 backing_dev_info 结构体成员在分配请求队列 request_queue 实例时被初始化（见第 10 章），函数代码简列如下：

```
struct request_queue *blk_alloc_queue_node(gfp_t gfp_mask, int node_id)
{
    ...

    /*初始化 backing_dev_info 结构体成员*/
    q->backing_dev_info.ra_pages =(VM_MAX_READAHEAD * 1024) / PAGE_CACHE_SIZE;
                                                    /*内存页大小为 4KB 时，ra_pages 为 32 页*/
    q->backing_dev_info.capabilities = BDI_CAP_CGROUP_WRITEBACK;          /*0x00000020*/
```

```
    q->backing_dev_info.name = "block";
    q->node = node_id;

    err = bdi_init(&q->backing_dev_info);   /*初始化 backing_dev_info 成员，/mm/backing-dev.c*/
    ...
}
```

如果内核配置页大小为 4KB，则最大预读窗口页数量设为 32。

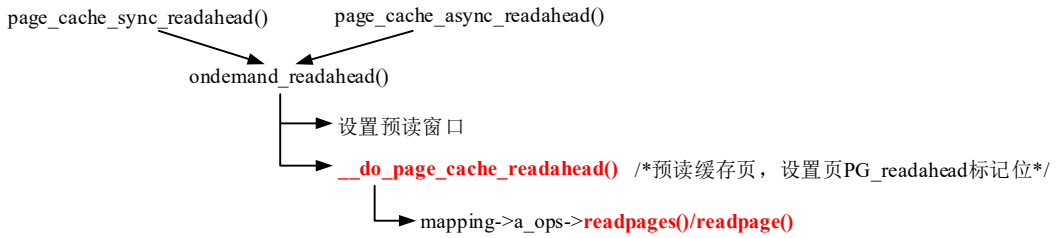
get_init_ra_size(size, max)函数用于确定初始预读窗口的大小，定义在/mm/readahead.c 文件内，size 表示本次读操作的页长度（数量），max 表示预读窗口的最大长度，通常为 file_ra_state.ra_pages。对于长度较小的读操作，预读窗口为读数据长度的 4 倍或 2 倍，如果读数据长度较长，则预读长度为 ra_pages。

get_next_ra_size(ra,max)函数根据上一次预读窗口的长度确定本次预读窗口的长度，函数内设置本次的预读窗口长度为 ra->size（上次预读窗口长度）的 4 倍或 2 倍，最大一般不超过 file_ra_state.ra_pages。

●执行预读

在 do_generic_file_read()函数中，如果读取的缓存页不在页缓存中，将调用 page_cache_sync_readahead()函数执行同步预读，即读取当前缓存页及其随后若干页的数据。如果当前缓存页设置了 PG_readahead 标记位，将调用 page_cache_async_readahead()函数启动异步预读操作。

预读操作函数调用关系如下图所示，同步和异步预读操作都是调用 ondemand_readahead()函数执行预读操作，函数内设置预读窗口信息（如果需要的话，由 file.f_ra 成员表示），调用__do_page_cache_readahead()函数执行缓存页（多页）的读取操作，并按需设置某一缓存页的 PG_readahead 标记位，最终调用地址空间操作结构中的 readpages()或 readpage()函数从块设备中读取数据至缓存页。



同步预读函数 page_cache_sync_readahead()定义如下（/mm/readahead.c）：

```
void page_cache_sync_readahead(struct address_space *mapping,   \
                               struct file_ra_state *ra, struct file *filp,pgoff_t offset, unsigned long req_size)
/*offset：当前读取页在文件中的编号，req_size：当前页至读操作最后页的长度（页数）*/
{
    if (!ra->ra_pages)      /*初始值为 32 页，如果为 0 则不预读*/
        return;

    /* be dumb */
    if (filp && (filp->f_mode & FMODE_RANDOM)) {      /*进程随机访问文件*/
        force_page_cache_readahead(mapping, filp, offset, req_size);    /*强制预读，/mm/readahead.c*/
        return;
    }

    /*执行预读*/
    ondemand_readahead(mapping, ra, filp, false, offset, req_size);      /*/mm/readahead.c*/
}
```

异步预读函数 page_cache_async_readahead()定义如下（/mm/readahead.c）：

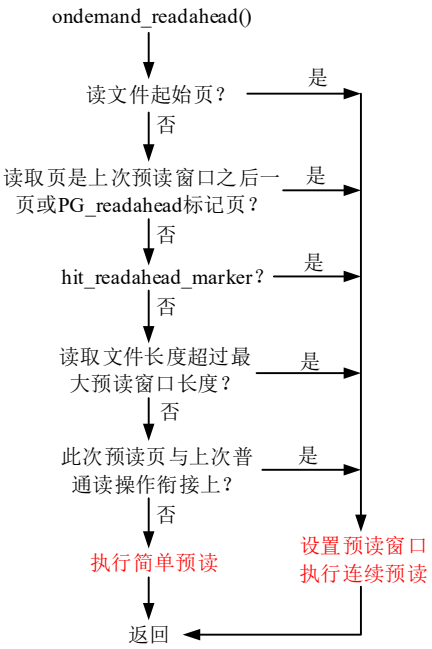
```
void page_cache_async_readahead(struct address_space *mapping,struct file_ra_state *ra, struct file *filp, \
                                struct page *page, pgoff_t offset,unsigned long req_size)
/*
*page：设置 PG_readahead 标记位的页，offset：设置 PG_readahead 标记位缓存页编号，
*req_size：当前页到本次读文件内容最后页的长度。
*/
{
    if (!ra->ra_pages)     /*ra_pages 为 0 则不预读*/
        return;

    /*读取页正在回写，返回*/
    if (PageWriteback(page))
        return;

    ClearPageReadahead(page);       /*清除页的 PG_readahead 标记位*/

    /*文件是否拥塞，见下节数据回写*/
    if (inode_read_congested(mapping->host))    /*/include/linux/backing-dev.h*/
        return;

    /*执行预读*/
    ondemand_readahead(mapping, ra, filp, true, offset, req_size);       /*/mm/readahead.c*/
}
```

同步预读和异步预读函数内都是调用 ondemand_readahead()函数执行预读操作，两者之间差别比较小，同步预读直接发起读取操作，而异步预读会考虑当前页是否是回写页和 inode 是否拥塞等情况。

ondemand_readahead()函数内需要判断预读操作是简单（一次性）预读，还是需要执行多次预读的连续预读。简单预读是指读文件操作中，读取的文件内容较短，只需要一次预读即可将本次读操作需要的数据读入页缓存，可以理解成读文件内容的长度没有超过一个预读窗口的长度。此时，不需要重置预读窗口和某一缓存页的 PG_readahead 标记位，因为本次读文件操作不需要再次触发预读操作。

连续预读是指读文件操作读取的文件内容较长，一次预读不能读入所有所需的数据，需要执行多次预读的情况。此时执行预读需要设置预读窗口，并设置窗口中某一缓存页的 PG_readahead 标记位，以便触发下一次的预读操作。

ondemand_readahead()函数执行流程如下图所示：



ondemand_readahead()函数内需要考虑多种需要执行连续预读的情况，函数定义如下（/mm/readahead.c）：

```
static unsigned long ondemand_readahead(struct address_space *mapping, struct file_ra_state *ra, \
                        struct file *filp,bool hit_readahead_marker, pgoff_t offset,unsigned long req_size)
/*hit_readahead_marker：同步预读为 false，异步预读为 true（强制指定执行连续预读）*/
{
    unsigned long max = max_sane_readahead(ra->ra_pages);     /*最大预读页数，ra_pages 或 32*/
    pgoff_t prev_offset;

    if (!offset)       /*offset 为 0，表示从文件开始处读取，跳至 initial_readahead 处，执行连续预读*/
        goto initial_readahead;

    /*当前页需启动异步预读，或者是上次预读窗口之后的一页，执行连续预读*/
    if ((offset == (ra->start + ra->size - ra->async_size) ||offset == (ra->start + ra->size))) {
                                            /*设置本次预读窗口信息*/
        ra->start += ra->size;                 /*预读窗口起始页*/
        ra->size = get_next_ra_size(ra, max);     /*预读窗口长度（缓存页数量）*/
        ra->async_size = ra->size;             /*读取窗口第一页时触发下一次异步预读*/
        goto readit;       /*跳至执行连续预读*/
    }

    if (hit_readahead_marker) {    /*参数指定了执行连续预读*/
        pgoff_t start;

        rcu_read_lock();
        start = page_cache_next_hole(mapping, offset + 1, max);    /*下一个尚未读入的缓存页*/
        rcu_read_unlock();

        if (!start || start - offset > max)
            return 0;
```

```
            ra->start = start;          /*设置预读窗口信息*/
            ra->size = start - offset;  /* old async_size */
            ra->size += req_size;
            ra->size = get_next_ra_size(ra, max);
            ra->async_size = ra->size;
            goto readit;      /*跳至执行连续预读*/
        }

        /*hit_readahead_marker 为 false，读取文件长度超过了预读窗口最大值，执行连续预读*/
        if (req_size > max)
            goto initial_readahead;

        /*当前缓存页与上次普通读文件操作衔接上（不是上次预读），执行连续预读*/
        prev_offset = (unsigned long long)ra->prev_pos >> PAGE_CACHE_SHIFT;
        if (offset - prev_offset <= 1UL)
            goto initial_readahead;

        if (try_context_readahead(mapping, ra, offset, req_size, max))
            goto readit;

        return    __do_page_cache_readahead(mapping, filp, offset, req_size, 0);     /*/mm/readahead.c*/
                        /*执行简单预读，即顺序读取 req_size 个缓存页，不设置 PG_readahead 标记位*/

    initial_readahead:                  /*设置初始预读窗口信息*/
        ra->start = offset;             /*本次预读的起始页*/
        ra->size = get_init_ra_size(req_size, max);       /*设置预读窗口初始长度（页数量）*/
        ra->async_size = ra->size > req_size ? ra->size - req_size : ra->size;
                                                          /*设置 PG_readahead 标记位页的位置*/
    readit:       /*执行连续预读*/
        /*如果 offset 是预读窗口起始页，则合并下一个预读窗口*/
        if (offset == ra->start && ra->size == ra->async_size) {
            ra->async_size = get_next_ra_size(ra, max);
            ra->size += ra->async_size;
        }
        return    ra_submit(ra, mapping, filp);   /*执行连续预读，/mm/internal.h*/
                  /*调用__do_page_cache_readahead(mapping, filp,ra->start, ra->size, ra->async_size)*/
    }
```

__do_page_cache_readahead()函数负责执行简单预读和连续预读操作，即在页缓存中创建若干缓存页并从块设备中读取数据填充缓存页，并设置某一缓存页的 PG_readahead 标记位，代码如下(/mm/readahead.c)：

```
    int __do_page_cache_readahead(struct address_space *mapping, struct file *filp,    \
                                  pgoff_t offset, unsigned long nr_to_read,unsigned long lookahead_size)
    /*
    *从 offset 缓存页开始，预读 nr_to_read 个缓存页，
    *设置预读窗口内倒数第 lookahead_size 个缓存页的 PG_readahead 标记位，
    *如果 lookahead_size 为 0，表示不设置缓存页 PG_readahead 标记位（简单预读），
```

```
*如果 ra->async_size=ra->size（lookahead_size=nr_to_read），则设置窗口第一页的
*PG_readahead 标记位，即读文件操作访问到窗口第一页时触发下一次的异步预读。
*/
{
    struct inode *inode = mapping->host;
    struct page *page;
    unsigned long end_index;        /*读取的最后一页*/
    LIST_HEAD(page_pool);       /*缓存页 page 实例的临时链表*/
    int page_idx;               /*设置 PG_readahead 标记位页的编号*/
    int ret = 0;
    loff_t isize = i_size_read(inode);     /*文件大小*/

    if (isize == 0)
        goto out;

    end_index = ((isize - 1) >> PAGE_CACHE_SHIFT);       /*文件内容最后页编号*/

    /*预先分配缓存页，插入临时链表*/
    for (page_idx = 0; page_idx < nr_to_read; page_idx++) {
        pgoff_t page_offset = offset + page_idx;

        if (page_offset > end_index)
            break;

        rcu_read_lock();
        page = radix_tree_lookup(&mapping->page_tree, page_offset);   /*在基数树中查找缓存页*/
        rcu_read_unlock();
        if (page && !radix_tree_exceptional_entry(page))     /*查找到则跳过，否则创建缓存页*/
            continue;

        page = page_cache_alloc_readahead(mapping);
                                    /*从伙伴系统分配页，/include/linux/pagemap.h*/
        if (!page)
            break;
        page->index = page_offset;
        list_add(&page->lru, &page_pool);       /*插入临时链表*/
        if (page_idx == nr_to_read - lookahead_size)
            SetPageReadahead(page);         /*设置触发异步预读页的 PG_readahead 标记位*/
        ret++;
    }

    if (ret)
        read_pages(mapping, filp, &page_pool, ret);     /*/mm/readahead.c*/
        /*调用 mapping->a_ops->readpages()或 readpage()读缓存页，并插入基数树和页 LRU 链表*/
    BUG_ON(!list_empty(&page_pool));
out:
```

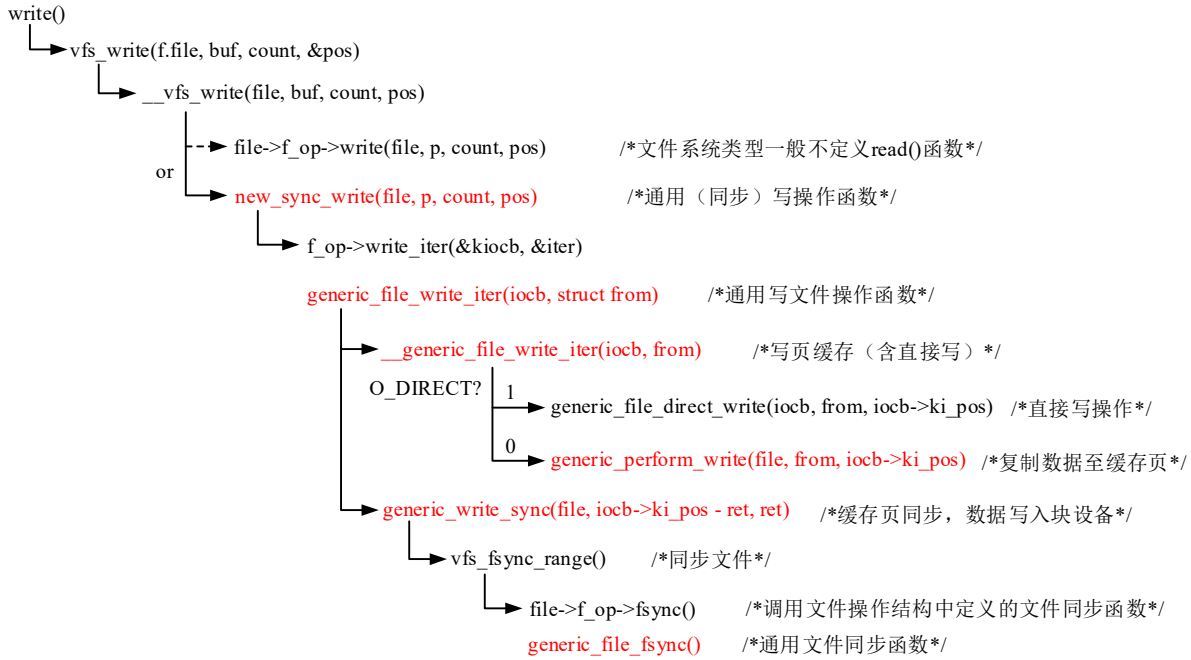
```
    return ret;
}
```

__do_page_cache_readahead()函数将需要执行预读缓存页的 page 实例插入临时双链表，最后调用函数 read_pages()从块设备中读数据至缓存页，并将缓存页插入到页缓存基数树和 LRU 双链表。

read_pages()函数将调用地址空间操作结构中的 a_ops->readpages()或 readpage()函数执行读缓存页操作。

11.2.2 普通写文件函数

写文件内容的 write()系统调用与 read()系统调用类似，函数调用关系如下图所示（详见第 7 章）：



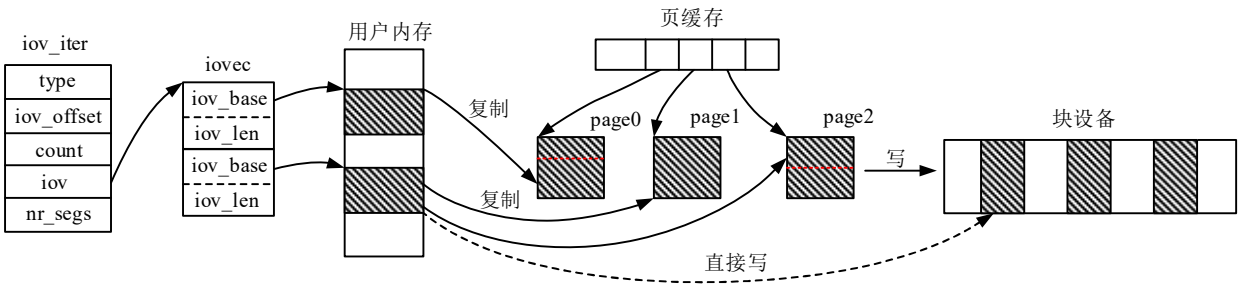
内核大部分文件系统类型都没有定义 file_operations 实例中 write()成员函数，而是定义 write_iter()成员函数。因此 write()系统调用内调用通用同步写操作函数 new_sync_write()完成写操作，此函数调用文件操作结构 file_operations 实例中定义的 write_iter()成员函数执行写操作。写操作与读操作类似，write_iter()函数通常调用（引用）通用的 **generic_file_write_iter()**函数。

generic_file_write_iter()函数主要分两步，一是将用户内存数据复制到页缓存（包括处理直接写），二是将页缓存中数据同步至块设备。

本小节主要介绍 generic_file_write_iter()函数的实现。

1 通用写文件函数

写文件内容操作流程如下图所示，写操作是读操作的逆操作，普通的写操作先将用户内存数据复制到页缓存（逐页复制），再由页缓存写出到块设备。写操作也可执行直接写，即直接将用户内存数据写入块设备，不过有条件限制，效率低。



通用写文件函数 **generic_file_write_iter()**定义在/mm/filemap.c 文件内，代码如下：

```
ssize_t generic_file_write_iter(struct kiocb *iocb, struct iov_iter *from)
{
    struct file *file = iocb->ki_filp;
    struct inode *inode = file->f_mapping->host;
    ssize_t ret;

    mutex_lock(&inode->i_mutex);
    ret = generic_write_checks(iocb, from);     /*写操作检查，返可写数据长度，/mm/filemap.c*/
    if (ret > 0)
        ret = __generic_file_write_iter(iocb, from);   /*将用户内存数据写入页缓存，/mm/filemap.c*/
    mutex_unlock(&inode->i_mutex);

    if (ret > 0) {     /*ret 为写入页缓存数据长度*/
        ssize_t err;
        err = generic_write_sync(file, iocb->ki_pos - ret, ret);
                                    /*同步页缓存，数据写出至块设备，/include/linux/fs.h*/
        if (err < 0)
            ret = err;
    }
    return ret;
}
```

generic_file_write_iter()函数在进行了写操作检查后，调用__generic_file_write_iter()函数将用户内存数据写入文件页缓存（或执行直接写操作），然后调用 generic_write_sync()函数同步页缓存数据，即将脏缓存页（数据修改过的页）写出到块设备。下文将分别介绍写页缓存和同步页缓存函数的实现。

2 写入页缓存

__generic_file_write_iter(iocb, from)函数用于将用户数据写入页缓存，函数定义在/mm/filemap.c 文件内：

```
ssize_t __generic_file_write_iter(struct kiocb *iocb, struct iov_iter *from)
/*iocb：IO 操作状态/控制信息（写入文件位置及长度等），form：用户内存信息（待写入数据）*/
{
    struct file *file = iocb->ki_filp;
    struct address_space * mapping = file->f_mapping;
    struct inode    *inode = mapping->host;
    ssize_t         written = 0;
    ssize_t         err;
    ssize_t         status;

    current->backing_dev_info = inode_to_bdi(inode);     /*后备存储设备信息，实例在请求队列中*/
    err = file_remove_privs(file);          /*移除文件特权，如 SUID，/fs/inode.c*/
    if (err)
        goto out;

    err = file_update_time(file);   /*修改 inode 实例中 mtime 和 ctime 时间信息，/fs/inode.c*/
    if (err)
```

```
            goto out;

        if (iocb->ki_flags & IOCB_DIRECT) {     /*直接写操作，打开文件设置了 O_DIRECT 标记*/
            loff_t pos, endbyte;

            written = generic_file_direct_write(iocb, from, iocb->ki_pos);   /*通用直接写操作函数*/

            if (written < 0 || !iov_iter_count(from) || IS_DAX(inode))       /*count 为 0，函数可返回*/
                    goto out;

            /*直接写操作未完成的数据，采用普通写操作完成*/
            status = generic_perform_write(file, from, pos = iocb->ki_pos); /*用户内存数据写入页缓存*/
            ...
            endbyte = pos + status - 1;
            err = filemap_write_and_wait_range(mapping, pos, endbyte);
                受   /*等待页缓存回写完成，因为这里是直接写，所有要等待回写，成功返回 0*/
            if (err == 0) {          /*文件内容回写成功*/
                iocb->ki_pos = endbyte + 1;
                written += status;
                invalidate_mapping_pages(mapping,pos >> PAGE_CACHE_SHIFT, \
                                                    endbyte >> PAGE_CACHE_SHIFT);
                        /*从页缓存中移除页（需是非锁定、非回写、非映射、非脏等页），/mm/truncate.c*/
            } else {

            }
        } else {   /*普通写操作，写入页缓存*/
             written = generic_perform_write(file, from, iocb->ki_pos);   /*/mm/filemap.c*/
             if (likely(written > 0))
                 iocb->ki_pos += written;
        }
    out:
        current->backing_dev_info = NULL;
        return written ? written : err;         /*返回写入页缓存数据长度*/
    }
```

__generic_file_write_iter()函数不仅会将用户数据写入文件页缓存，还会更新文件时间等元信息，这些信息保存在文件节点（inode 实例）中。

__generic_file_write_iter()函数通过判断 iocb->ki_flags 成员 IOCB_DIRECT 标记位是否置位确定是直接写操作还是普通写操作。这里我们只讨论普通写操作的情况，下一小节将讨论直接写操作的情况。

普通写操作调用 generic_perform_write(file, from, iocb->ki_pos)函数将用户内存数据复制到页缓存。与读操作类似，用户内存与页缓存之间的数据复制以缓存页为单位进行，由 iocb->ki_pos 成员值确定起始缓存页索引值和页内偏移量（定位文件），然后将用户数据逐页复制到页缓存。

generic_perform_write()函数定义如下（/mm/filemap.c）：

```
ssize_t generic_perform_write(struct file *file,struct iov_iter *i, loff_t pos)
{
    struct address_space *mapping = file->f_mapping;            /*地址空间*/
    const struct address_space_operations *a_ops = mapping->a_ops;       /*地址空间操作结构*/
```

```
long status = 0;
ssize_t written = 0;
unsigned int flags = 0;

if (!iter_is_iovec(i))       /*iov_iter 实例中是否由 iovec 实例表示内存信息，/include/linux/uio.h*/
        flags |= AOP_FLAG_UNINTERRUPTIBLE;

do {          /*遍历需要写入的缓存页，将用户内存数据逐页复制到缓存页*/
        struct page *page;
        unsigned long offset;      /*缓存页内偏移量，字节数*/
        unsigned long bytes;       /*写入缓存页数据字节数*/
        size_t copied;             /*复制操作是否完成*/
        void *fsdata;

        offset = (pos & (PAGE_CACHE_SIZE - 1));     /*当前写文件位置在缓存页内偏移量*/
        bytes = min_t(unsigned long, PAGE_CACHE_SIZE - offset,iov_iter_count(i));
                                                        /*需向当前缓存页复制数据的字节数*/
again:
        if (unlikely(iov_iter_fault_in_readable(i, bytes))) {
                status = -EFAULT;
                break;
        }
        status = a_ops->write_begin(file, mapping, pos, bytes, flags,&page, &fsdata);
                                        /*创建（获取）缓存页，置页脏标记、处理日志文件系统信息等*/
          /*通常调用 block_write_begin()函数，需保证缓存页数据有效，若无效需要先执行读操作*/
        if (unlikely(status < 0))
                break;

        if (mapping_writably_mapped(mapping))
                flush_dcache_page(page);       /*空操作*/
        copied = iov_iter_copy_from_user_atomic(page, i, offset, bytes);
                  /*从进程内存复制数据到缓存页（原子操作），返回复制的字节数，/lib/iov_iter.c*/
        flush_dcache_page(page);

        status = a_ops->write_end(file, mapping, pos, bytes, copied,page, fsdata);
                                        /*调用写缓存页结束函数，标记 inode 脏、处理日志等*/
        if (unlikely(status < 0))
                break;
        copied = status;
        cond_resched();       /*条件调度*/

        iov_iter_advance(i, copied);       /*count 减小 copied 值，即此次复制的字节数*/
        if (unlikely(copied == 0)) {        /*写入不成功，重试*/
                bytes = min_t(unsigned long, PAGE_CACHE_SIZE - offset,iov_iter_single_seg_count(i));
                goto again;
        }
```

```
            pos += copied;          /*修改文件位置*/
            written += copied;      /*统计写入页缓存数据字节数*/

            balance_dirty_pages_ratelimited(mapping); /*数据回写机制，见下文，/mm/page-writeback.c*/
            if (fatal_signal_pending(current)) {
                status = -EINTR;
                break;
            }
        } while (iov_iter_count(i));     /*count 为 0 则表示复制数据完成，循环结束*/
        return written ? written : status;     /*返回写入数据总的字节数*/
    }
```

generic_perform_write()函数依次将用户内存数据复制到缓存页，对每个缓存页的处理流程如下：

确定写入数据在缓存页内偏移量，调用 a_ops->write_begin()函数查找（创建）缓存页，并保证缓存页数据有效（无效需要执行读操作），然后调用 iov_iter_copy_from_user_atomic()函数将用户内存中数据复制到缓存页，调用地址空间操作结构中定义的 write_end()函数完成写缓存页后的工作等。

每次从用户内存复制数据至缓存页后会减小 iov_iter 实例中的 count 值（此次复制的字节数），即表示剩余尚未复制的数据减少了，当 count 值为 0 时，表示所有数据复制完成，generic_perform_write()函数退出。

3 同步页缓存

通用写文件函数 generic_file_write_iter()在将用户内存数据复制到页缓存后，将调用 generic_write_sync()函数同步数据，即将写入到页缓存的数据写出到块设备，也包括同步 inode 中元数据。

generic_write_sync()函数定义如下（/include/linux/fs.h）：

```
static inline int generic_write_sync(struct file *file, loff_t pos, loff_t count)
/*file：文件 file 实例指针，pos：写出数据起始文件位置，count：写出数据长度*/
{
    if (!(file->f_flags & O_DSYNC) && !IS_SYNC(file->f_mapping->host))   /*不立即同步的条件*/
        return 0;
    return vfs_fsync_range(file, pos, pos + count - 1,(file->f_flags & __O_SYNC) ? 0 : 1);
                                                        /*执行同步操作，写出到块设备*/
}
```

如果打开文件的 open()系统调用中标记 flags 参数没有设置 **O_DSYNC** 标记位且文件 inode 实例 i_flags 成员没有设置 **S_SYNC** 标记位，则此时不执行页缓存同步，而是交由数据回写机制在适当的时候执行数据同步（见下一节）。

如果以上两个标记位中任意一个设置（或两个都设置），此时将调用 **vfs_fsync_range**()函数立即执行页缓存数据同步。

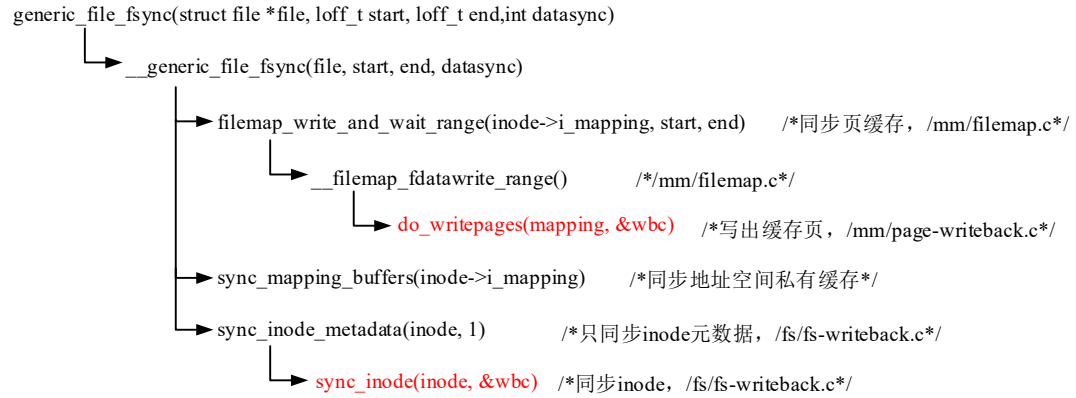
vfs_fsync_range()函数定义在/fs/sync.c 文件内（同步文件内容和元数据），代码如下：

```
int vfs_fsync_range(struct file *file, loff_t start, loff_t end, int datasync)
/*
*file：文件 file 实例指针，start：同步数据文件起始位置，end：结束文件位置，
*datasync：表示是否只同步文件内容数据而不同步 inode 元数据，
*打开文件设置了__O_SYNC 标记位 datasync 为 0（同步文件内容和 inode 元数据），否则为 1。
*/
{
    struct inode *inode = file->f_mapping->host;
```

```
    if (!file->f_op->fsync)
        return -EINVAL;
    if (!datasync && (inode->i_state & I_DIRTY_TIME)) {        /*datasync 为 0，同步元数据*/
        spin_lock(&inode->i_lock);
        inode->i_state &= ~I_DIRTY_TIME;
        spin_unlock(&inode->i_lock);
        mark_inode_dirty_sync(inode);          /*标记 inode 要同步*/
    }
    return file->f_op->fsync(file, start, end, datasync);   /*调用文件操作结构中的 fsync()函数*/
}
```

vfs_fsync_range()函数内调用 file_operations 实例中的 fsync()函数执行文件同步操作，虽然这是一个由具体文件系统类型定义的函数，但是内核依然提供了通用的实现函数，供具体文件系统类型调用，通用函数为 **generic_file_fsync()**。

generic_file_fsync()函数调用关系如下图所示：



generic_file_fsync()函数定义在/fs/libfs.c 文件内，代码如下：

```
int generic_file_fsync(struct file *file, loff_t start, loff_t end,int datasync)    /*/fs/libfs.c*/
{
    struct inode *inode = file->f_mapping->host;
    int err;

    err = __generic_file_fsync(file, start, end, datasync);    /*同步文件内容，成功返回 0，/fs/libfs.c*/
    if (err)
        return err;
    return blkdev_issue_flush(inode->i_sb->s_bdev, GFP_KERNEL, NULL);
                                        /*向块设备驱动发送 WRITE_FLUSH 请求，/block/blk-flush.c*/
}
```

__generic_file_fsync()函数定义在/fs/libfs.c 文件内，代码如下：

```
int __generic_file_fsync(struct file *file, loff_t start, loff_t end,int datasync)
{
    struct inode *inode = file->f_mapping->host;
    int err;
    int ret;

    err = filemap_write_and_wait_range(inode->i_mapping, start, end);
                                                /*同步页缓存中指定区域，/mm/filemap.c*/
```

```
    if (err)
        return err;

    mutex_lock(&inode->i_mutex);
    ret = sync_mapping_buffers(inode->i_mapping);
                                /*同步地址空间相关缓存（mapping->private_list），/fs/buffer.c*/
                                /*mapping->private_list 双链表中是 buffer_head 实例，执行它们*/
    if (!(inode->i_state & I_DIRTY_ALL))
        goto out;
    if (datasync && !(inode->i_state & I_DIRTY_DATASYNC))
        goto out;

    err = sync_inode_metadata(inode, 1);    /*同步文件 inode 中元数据，/fs/fs-writeback.c*/
    if (ret == 0)
        ret = err;

out:
    mutex_unlock(&inode->i_mutex);
    return ret;
}
```

__generic_file_fsync()函数内完成的主要工作有三项：

（1）调用 filemap_write_and_wait_range()函数将页缓存中指定区域写出到块设备：函数内构建回写控制结构 writeback_control 实例，调用 do_writepages(mapping, &wbc)函数执行回写，此函数最终地址空间操作结构实例中定义的 **a_ops->writepages(mapping, wbc)**函数或 **generic_writepages(mapping, wbc)**通用函数将页缓存中多页写出到块设备，见本章上一节。

（2）调用 sync_mapping_buffers()函数回写地址空间中的私有块缓存数据，例如日志文件系统中的日志信息等（个人理解，可能不正确），即执行 mapping->private_list 双链表中的 buffer_head 实例。

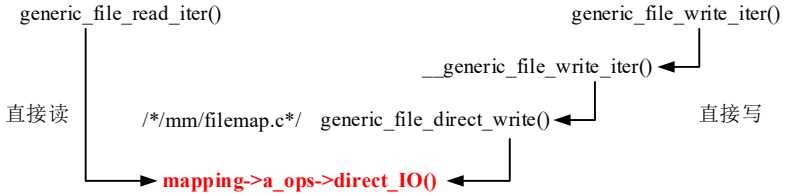
（3）如果参数 datasync 为 0，表示需要回写文件 inode 元数据，则调用 sync_inode_metadata(inode, 1)函数执行此操作，函数内构建回写控制结构 writeback_control 实例后，调用 **sync_inode(inode, &wbc)**函数进行 inode 回写操作。inode 的回写本章下一节再介绍。

至此，写文件操作完成，在打开文件的 open()系统调用中如果设置了 **O_DSYNC** 标记位，在写操作中将立即同步文件内容至块设备（不同步 inode 元信息），如果设置了 **O_SYNC** 标记位在写操作中将立即同步文件所有信息，包括文件内容和 inode 元信息。如果没有设置这些标记位，写操作将只把用户内存中数据复制到文件页缓存，由内核回写机制将页缓存中数据同步到块设备，数据回写机制将在下一节介绍。

11.2.3 直接读写文件函数

由前面介绍的通用文件读写文件函数可知，内核提供了不经过页缓存直接读写文件内容的机制，只需在打开文件的 open()系统调用参数中设置 **O_DIRECT** 标记位。

直接读写操作函数调用关系如下图所示：



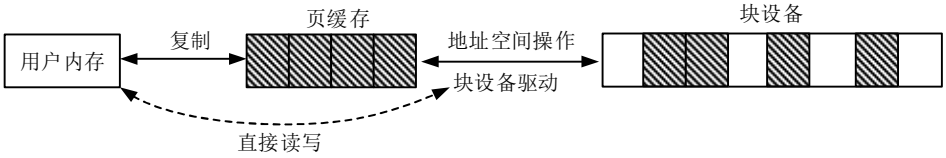
直接读操作在通用读文件函数中直接调用地址空间操作结构中定义的 direct_IO()函数执行直接读操作，直接写操作要稍微复杂一些。通用写文件函数中调用 generic_file_direct_write()函数（/mm/filemap.c）处理直接写操作，函数内首先需要将页缓存中的文件内容写回块设备，然后才能调用 direct_IO()函数执行直接写操作，最后要设置页缓存中所写区域无效（因为页缓存中数据与块设备中数据不一样了）。

内核定义了地址空间操作结构中 direct_IO()函数的通用实现函数，本节主要介绍此通用函数的实现，直接读写操作的数据结构和函数代码位于/fs/direct-io.c 文件内。

1 概述

直接读写操作就是直接将块设备中数据读入用户内存或将用户内存中数据写入块设备。读者可能会觉得直接读写不是效率更高吗，直接就是用户内存与块设备的交互。其实不然，就如同 CPU 核中有 cache 一样，它用于缓存内存中的数据和指令，可提高 CPU 访存的效率。同理，页缓存是内存中为缓慢的块设备建立的缓存，同样也能提高访问的效率。直接读写反而效率更低，只在一些特殊的场景下使用。

直接读写看似简单，其实也有它的复杂之处，先看下图所示的普通读写过程：



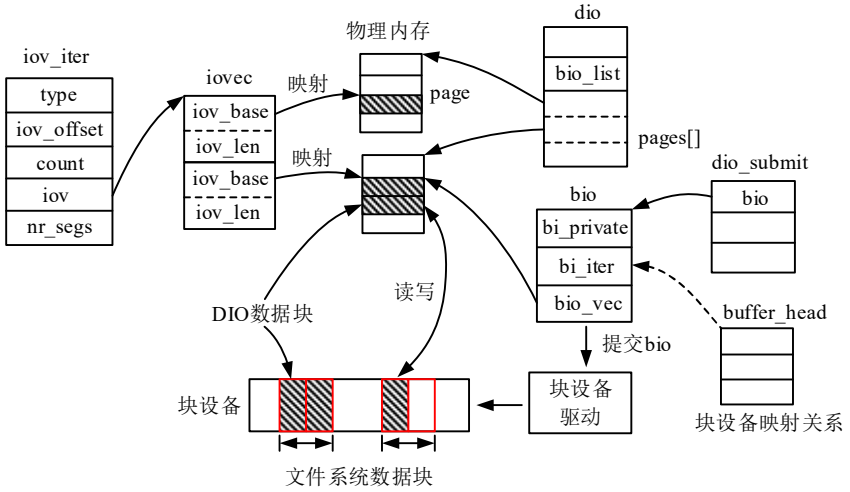
由于有页缓存建立的连续的文件内容缓存，用户可以随意地访问页缓存中的文件内容，因为它们只是在内存中来回复制数据。

当直接将用户内存数据写入块设备或从块设备读数据时，就需要考虑用户内存地址和长度的对齐问题，因为块设备（驱动程序）限定了其只能按块读写，因此相应的用户内存起始位置和长度也是要按块对齐，而不能是随意的。

直接读写时的数据块大小称它为 DIO 数据块大小，这个大小如何确定呢？

块设备请求队列中 q->limits.**logical_block_size** 限制了驱动程序访问块设备的最小块大小，缓存页中按块访问的大小为文件系统数据块大小（内核中表示的大小）。直接读写时，DIO 数据块大小必须取这两者之一，用户内存起始位置和长度都要按 DIO 数块大小对齐。

直接读写操作流程简列如下图所示：



上图中假设 DIO 数据块大小是文件系统数据块大小的一半，进程用 2 段内存来保存一段连续的文件内容，第 1 段大小为 1 个 DIO 数据块，第 2 段为 2 个 DIO 数据块，这 2 段内存位于不同的页，2 段用户内存的起始地址都要求 DIO 块大小对齐。这 3 个 DIO 数据块保存的是一段连续的文件内容。

dio 结构体中有一个 pages[]数组成员，用于收集用户内存的 page 实例，它类似于普通读写中的页缓存。

dio_submit 结构体用于遍历 dio 实例中管理的 page 实例，获取页内 DIO 数据块的映射信息，为各页构建 bio 实例，向块设备驱动程序提交 bio 实例，实现数据传输。dio_submit 结构体所取的作用类似于地址空间操作结构。

内核通过 buffer_head 实例来保存获取的 DIO 数据块映射信息，这里获取的映射信息是按文件系统数据块大小获取的映射信息，DIO 数据块的映射关系还需要转换一下。

如果是执行同步直接读写操作，提交所有 bio 后，当前进程会进入睡眠等待，数据传输完成后将唤醒当前进程。异步直接读写则不等待，提交所有 bio 后，函数返回。

■数据结构

dio 结构体用于收集用户内存信息，结构体定义在/fs/direct-io.c 文件内：

```
struct dio {
    int flags;              /*标记成员*/
    int rw;                 /*读 0，写 1*/
    struct inode *inode;        /*指向读写文件的 inode 实例*/
    loff_t i_size;              /* i_size when submitted */
    dio_iodone_t *end_io;       /*dio 操作完成的回调函数*/

    void *private;          /*与 map_bh.b_private 成员值相同*/

    /* BIO 完成状态*/
    spinlock_t bio_lock;        /* protects BIO fields below */
    int page_errors;            /*调用 get_user_pages()函数获取用户内存 page 的错误*/
    int is_async;               /*是异步操作吗*/
    bool defer_completion;      /*延时后释放 dio 实例，由工作队列完成*/
    int io_error;               /* IO error in completion path */
    unsigned long refcount;     /*提交但未处理的 bio 实例数量*/
    struct bio *bio_list;       /*管理处理完的 bio 实例（同步读写）*/
    struct task_struct *waiter; /*等待进程*/

    /*异步 IO 相关成员*/
    struct kiocb *iocb;         /*文件位置信息等*/
    ssize_t result;             /*实际读写的字节数*/
    union {
        struct page *pages[DIO_PAGES];      /*用户进程内存 page 实例指针数组，DIO_PAGES=64*/
        struct work_struct complete_work;   /* deferred AIO completion */
    };
} ____cacheline_aligned_in_smp;
```

dio 结构体主要成员简介如下：

●**flags**：标记成员，由枚举类型定义（/include/linux/fs.h）：

```
enum {
    DIO_LOCKING         = 0x01,     /*操作需锁定*/
    DIO_SKIP_HOLES      = 0x02,     /*跳过空洞*/
    DIO_ASYNC_EXTEND = 0x04,    /*写操作可扩展文件内容*/
    DIO_SKIP_DIO_COUNT = 0x08,      /* inode/fs/bdev does not need truncate protection */
```

};
●**end_io**：dio_iodone_t 类型函数指针，表示 DIO 完成的回调函数，函数原型如下（/include/linux/fs.h）：
typedef void (dio_iodone_t)(struct kiocb *iocb, loff_t offset,ssize_t bytes, void *private);

●**pages[]**：page 指针数组，数组项数 DIO_PAGES 为 64，用于收集用户内存页。

内核为 dio 结构体创建了 slab 缓存，函数为 **dio_init(void)**，在内核初始化子系统时调用。

dio_submit 结构体用于控制 bio 实例的创建提交等，结构体定义如下（/fs/direct-io.c）：
```
struct dio_submit {
    struct bio *bio;            /*当前 bio 实例指针*/
    unsigned    blkbits;        /*2^blkbits 表示 DIO 数据块大小*/
    unsigned    blkfactor;      /*i_blkbits - blkbits，文件系统数据块大小是 DIO 数据块大小的倍数*/
    unsigned    start_zero_done;  /**/
    int pages_in_io;                /*IO 操作大约占用的页数*/
    sector_t block_in_file;         /*以 DIO 数据块为单位，当前操作文件内容的数据块号*/
    unsigned blocks_available;
                    /*get_block()获取映射的 DIO 数据块数量，提交 bio 后减少，可用的映射数据块*/
    int reap_counter;               /* rate limit reaping */
    sector_t    final_block_in_request;     /*操作内容以 DIO 数据块为单位的结束数据块号，不改变*/
    int boundary;                       /*上一个 DIO 数据块是否在页边界上*/
    get_block_t *get_block;             /*获取块设备映射函数*/
    dio_submit_t *submit_io;            /*提交 IO 操作函数*/

    loff_t    logical_offset_in_bio;    /*bio 中当前第一个 DIO 数据块号*/
    sector_t    final_block_in_bio;     /*bio 中最后一个 DIO 数据块号+1 */
    sector_t    next_block_for_io;      /*下一个 DIO 数据块号*/

    struct page *cur_page;          /*当前处理页 page 指针*/
    unsigned cur_page_offset;       /*页内偏移量，字节数*/
    unsigned cur_page_len;          /*当前页读写数据长度*/
    sector_t cur_page_block;        /* Where it starts */
    loff_t cur_page_fs_offset;      /* Offset in file */

    struct iov_iter *iter;          /*用户内存信息*/

    /*指向 dio 结构体中 pages[]数组起始和结束数组项索引值*/
    unsigned    head;           /*下一个处理页，pages[]数组项，从 0 开始*/
    unsigned    tail;           /*最后有效 pages[]数组项数加 1*/
    size_t    from, to;       /*pages[]数组项所有页中读写数据起止偏移量（逻辑上视为连续的）*/
};
```
dio_submit 结构体中成员简介如下：
●**submit_io**：提交 dio_submit 实例的函数，函数原型如下（/include/linux/fs.h）：
typedef void (dio_submit_t)(int rw, struct bio *bio, struct inode *inode,loff_t file_offset);

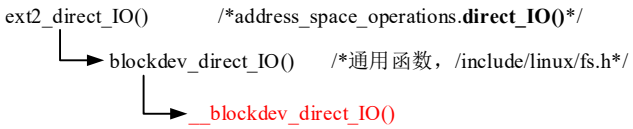
2 通用函数

地址空间操作结构中的 direct_IO()函数负责执行文件内容的直接读写操作，函数声明如下：

ssize_t (*direct_IO) (struct kiocb *, struct iov_iter *iter, loff_t offset);

内核提供了直接读写文件内容的通用实现函数，供具体文件系统类型地址空间操作结构中 direct_IO()函数调用，通用实现函数为**__blockdev_direct_IO()**。

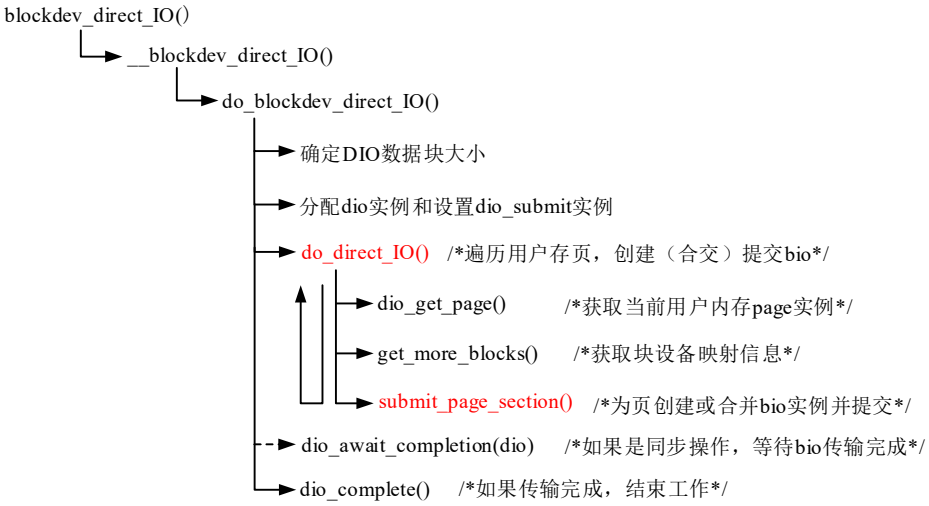
例如：ext2 文件系统类型地址空间操作结构中 direct_IO()函数调用关系如下：



blockdev_direct_IO()函数也是一个通用函数，定义在/include/linux/fs.h 头文件，定义如下：

```
static inline ssize_t blockdev_direct_IO(struct kiocb *iocb,
                        struct inode *inode,struct iov_iter *iter, loff_t offset,get_block_t get_block)
/*offset：读写文件起始位置，get_block：获取文件映射函数指针*/
{
    return __blockdev_direct_IO(iocb, inode, inode->i_sb->s_bdev, iter,
                        offset, get_block, NULL, NULL,DIO_LOCKING | DIO_SKIP_HOLES);
}
```

__blockdev_direct_IO()函数调用关系简列如下图所示：



do_blockdev_direct_IO()函数主要执行以下工作：

（1）确定 DIO 数据块大小，并检查用户内存起始位置和长度以及访问文件位置是否满足对齐要求，满足对齐要求才执行后面的操作，否则函数返回。

（2）分配 dio 实例，并设置 dio_submit 实例（局部变量）。

（3）调用 do_direct_IO()函数循环从 dio.**pages[]**数组获取用户内存 page 实例（当前页），如果数组项为空或页都处理完了，则从用户内存获以下一批 page 实例。

do_direct_IO()函数对当前页调用 get_more_blocks()函数获取页内 DIO 数据块的块设备映射信息，调用 submit_page_section()函数为 page 中的 DIO 数据块创建或合并 bio 实例，并提交，执行读写操作。

do_direct_IO()函数正确返回时，已经为所有用户内存页创建并提交了 bio 实例（最后 bio 实例是若干个整页映射到块设备连续区域时暂不提交，在 do_direct_IO()函数之后提交）。

（4）如果是同步读写，等待操作完成，异步则不等待。最后，同步读写调用 dio_complete()函数结束

操作，如释放 dio 实例等。

__blockdev_direct_IO()函数定义如下（/fs/direct-io.c）：

```
ssize_t __blockdev_direct_IO(struct kiocb *iocb, struct inode *inode,
                struct block_device *bdev, struct iov_iter *iter,loff_t offset, get_block_t get_block,
                dio_iodone_t end_io, dio_submit_t submit_io,int flags)
/*
*iocb：文件状态信息，iter：用户内存信息，offset：当前文件位置。
*get_block:获取文件内容块设备映射函数，
*end_io：这里为 NULL，submit：提交 dio_submit 实例函数，这里为 NULL，
*flags：dio 结构体 flags 成员值，这里为 DIO_LOCKING | DIO_SKIP_HOLES。
*/
{
        prefetch(&bdev->bd_disk->part_tbl);      /*读取指定内存数据至 CPU 缓存*/
        prefetch(bdev->bd_queue);
        prefetch((char *)bdev->bd_queue + SMP_CACHE_BYTES);

        return do_blockdev_direct_IO(iocb, inode, bdev, iter, offset, get_block,end_io, submit_io, flags);
}
```

do_blockdev_direct_IO()函数定义如下：

```
static inline ssize_t do_blockdev_direct_IO(struct kiocb *iocb, struct inode *inode,
            struct block_device *bdev, struct iov_iter *iter,
            loff_t offset, get_block_t get_block, dio_iodone_t end_io,dio_submit_t submit_io, int flags)
/*bdev：指向块设备 block_device 实例*/
{
    unsigned i_blkbits = ACCESS_ONCE(inode->i_blkbits);
                                                /*inode 中给出的文件系统数据块大小*/
    unsigned blkbits = i_blkbits;    /*DIO 数据块大小（2^blkbits），初始设为文件系统数据块大小*/
    unsigned blocksize_mask = (1 << blkbits) - 1;    /*DIO 数据块大小掩码，检查对齐要求*/
    ssize_t retval = -EINVAL;
    size_t count = iov_iter_count(iter);      /*读写操作数据长度，字节数*/
    loff_t end = offset + count;              /*读写文件内容结束位置加 1（文件偏移量，字节数）*/
    struct dio *dio;                          /*dio 实例指针*/
    struct dio_submit sdio = { 0, };          /*dio_submit 实例*/
    struct buffer_head map_bh = { 0, };       /*块缓存头实例*/
    struct blk_plug plug;                     /*blk_plug 实例*/
    unsigned long align = offset | iov_iter_alignment(iter);    /*/lib/iov_iter.c*/
    /*align 包含了各内存块起始位置和长度，以及文件位置的对齐要求（所有这些值的或）*/
    if (align & blocksize_mask) {      /*如果不是文件系统数据块对齐*/
        if (bdev)
            blkbits = blksize_bits(bdev_logical_block_size(bdev));    /*q->limits.logical_block_size*/

        blocksize_mask = (1 << blkbits) - 1;          /*对齐掩码*/
        if (align & blocksize_mask)          /*如果不满足对齐要求，不能执行直接读写操作，返回*/
            goto out;
```

```
}
/*以上代码要求访问文件起始位置，用户内存起始位置及长度都要 2^blkbits 对齐（各段都要）*/
/*2^blkbits 为文件系统数据块大小，或 q->limits.logical_block_size 大小（不一定理解准确）*/

/*长度为 0，不需要执行*/
if (iov_iter_rw(iter) == READ && !iov_iter_count(iter))
        return 0;

dio = kmem_cache_alloc(dio_cache, GFP_KERNEL);     /*从 slab 缓存中分配 dio 实例*/
retval = -ENOMEM;
...
memset(dio, 0, offsetof(struct dio, pages));        /*清零 dio 实例 page 指针数组*/

dio->flags = flags;        /*赋值标记成员*/
if (dio->flags & DIO_LOCKING) {        /*如果是带锁的操作*/
        if (iov_iter_rw(iter) == READ) {
                struct address_space *mapping =iocb->ki_filp->f_mapping;
                mutex_lock(&inode->i_mutex);

                retval = filemap_write_and_wait_range(mapping, offset,end - 1);
                                                                /*等待页缓存脏页回写完成*/
                ...
        }
}

if (is_sync_kiocb(iocb))       /*返回 kiocb->ki_complete == NULL?*/
      dio->is_async = false;              /*是不是异常操作，read()、write()系统调用为同步读写*/
else if (!(dio->flags & DIO_ASYNC_EXTEND) && \
                        iov_iter_rw(iter) == WRITE && end > i_size_read(inode))
      dio->is_async = false;
else
      dio->is_async = true;

dio->inode = inode;
dio->rw = iov_iter_rw(iter) == WRITE ? WRITE_ODIRECT : READ;    /*读还是写操作*/

if (dio->is_async && iov_iter_rw(iter) == WRITE &&
      ((iocb->ki_filp->f_flags & O_DSYNC) ||IS_SYNC(iocb->ki_filp->f_mapping->host))) {
            retval = dio_set_defer_completion(dio);        /*异常读，但打开文件要求同步写*/
                                /*为超级块实例创建工作队列赋予 s_dio_done_wq 成员*/
            ...
}

if (!(dio->flags & DIO_SKIP_DIO_COUNT))
      inode_dio_begin(inode);        /*增加 inode->i_dio_count 计数，/include/linux/fs.h*/
```

```
 retval = 0;
 sdio.blkbits = blkbits;                    /*DIO 数据块大小*/
 sdio.blkfactor = i_blkbits - blkbits;      /*系数，i_blkbits 为文件系统数据块大小，0 表示两者相等*/
 sdio.block_in_file = offset >> blkbits;    /*起始 DIO 数据块号（文件内容中逻辑块号）*/

 sdio.get_block = get_block;      /*获取文件内容映射函数*/
 dio->end_io = end_io;            /*这里为 NULL*/
 sdio.submit_io = submit_io;      /*这里为 NULL*/
 sdio.final_block_in_bio = -1;
 sdio.next_block_for_io = -1;

 dio->iocb = iocb;
 dio->i_size = i_size_read(inode);     /*文件大小，字节数*/

 spin_lock_init(&dio->bio_lock);
 dio->refcount = 1;               /*提交但未处理的 bio 实例，初值设为 1*/

 sdio.iter = iter;
 sdio.final_block_in_request =(offset + iov_iter_count(iter)) >> blkbits;
                                                    /*读写操作最后 DIO 数据块逻辑块号*/

 if (unlikely(sdio.blkfactor))   /*blkfactor 不为 0，DIO 数据块与文件系统数据块大小不一样*/
        sdio.pages_in_io = 2;       /*page 加 2 页*/

 sdio.pages_in_io += iov_iter_npages(iter, INT_MAX);     /*用户内存占用的页数，/lib/iov_iter.c*/

 blk_start_plug(&plug);       /*blk_plug 实例赋予进程*/

 retval = do_direct_IO(dio, &sdio, &map_bh);   /*收集用户页构建 bio 实例并提交，见下文*/
 if (retval)                        /*成功 retval 为 0*/
        dio_cleanup(dio, &sdio);     /*释放 dio 实例中页面*/


...
dio_zero_block(dio, &sdio, 1, &map_bh);
                /*用户内存地址和长度、文件位置不与文件系统数据块大小成倍数关系时的处理*/

if (sdio.cur_page) {            /*最后整页映射*/
    ssize_t ret2;
    ret2 = dio_send_cur_page(dio, &sdio, &map_bh);          /*为 page 创建或合并 bio 实例*/
    if (retval == 0)
            retval = ret2;
    page_cache_release(sdio.cur_page);
    sdio.cur_page = NULL;
}
if (sdio.bio)          /*处理最后剩余的 bio 实例*/
    dio_bio_submit(dio, &sdio);              /*提交 bio 实例*/
```

```
    blk_finish_plug(&plug);

    dio_cleanup(dio, &sdio);       /*释放页面*/

     /*读写操作完成*/
     if (iov_iter_rw(iter) == READ && (dio->flags & DIO_LOCKING))
          mutex_unlock(&dio->inode->i_mutex);

     BUG_ON(retval == -EIOCBQUEUED);
     if (dio->is_async && retval == 0 && dio->result &&    \
           (iov_iter_rw(iter) == READ || dio->result == count))          /*异步读写*/
          retval = -EIOCBQUEUED;
     else                                                  /*同步读写*/
          dio_await_completion(dio);           /*睡眠等待 bio 传输完成*/

     if (drop_refcount(dio) == 0) {          /*减 1*/
          retval = dio_complete(dio, offset, retval, false);     /*结束工作，释放 dio 实例等*/
     } else
          BUG_ON(retval != -EIOCBQUEUED);

out:
     return retval;
}
```

do_blockdev_direct_IO()函数执行的主要工作前面介绍过了，调用的 do_direct_IO()函数用于收集用户内存并逐页为其获取映射关系，创建或合并 bio 实例，并提交。do_direct_IO()函数返回后，可能还有剩余的页没有提交，在 do_direct_IO()函数之后提交。

如果是同步读写，则等待所有提交 bio 数据传输完成，异步读写不等待。最后，调用 drop_refcount(dio)函数对 dio->**refcount** 值减 1，若为 0 则调用 dio_complete()函数结束工作（释放 dio 实例等）。同步读写最后会调用 dio_complete()函数，异常读写一般不会（因为 dio->refcount 不为 0）。

■执行读写

do_direct_IO()函数用于遍历用户内存页，解析映射关系，创建提交 bio 实例，代码如下：

```
static int do_direct_IO(struct dio *dio, struct dio_submit *sdio,struct buffer_head *map_bh)
/*dio：指向 dio 实例，sdio：指向 dio_submit 实例，map_bh：指向 buffer_head 实例*/
{
     const unsigned blkbits = sdio->blkbits;       /*DIO 数据块大小*/
     int ret = 0;

     while (sdio->block_in_file < sdio->final_block_in_request) {       /*遍历用户内存区 DIO 数据块*/
          struct page *page;
          size_t from, to;
          /*以下是处理一个内存页，while()循环用于遍历所有内存页*/

          page = dio_get_page(dio, sdio);             /*获取 dio->pages[head]指向 pgae 实例*/
```

```
                                        /*若数组为空（或处理完了）先获取下一批 page 实例*/
    ...
    from = sdio->head ? 0 : sdio->from;          /*页内数据起始偏移量，相对于用户内存开始处*/
    to = (sdio->head == sdio->tail - 1) ? sdio->to : PAGE_SIZE;        /*页内结束字节偏移量*/
    sdio->head++;       /*dio->pages[]下一数组项*/

    while (from < to) {     /*遍历页内 DIO 数据块，创建（合并）并提交 bio 实例*/
        unsigned this_chunk_bytes;     /* # of bytes mapped */
        unsigned this_chunk_blocks;   /* # of blocks */
        unsigned u;

        if (sdio->blocks_available == 0) {     /*需要获取块设备映射信息*/
            unsigned long blkmask;
            unsigned long dio_remainder;

            ret = get_more_blocks(dio, sdio, map_bh); /*获取文件内容映射关系，/fs/direct-io.c*/
                                /*注意 map_bh 中是以文件系统数据块为单位表示的映射关系*/
            ...
            if (!buffer_mapped(map_bh))          /*访问了空洞*/
                goto do_holes;
                                    /*map_bh 中是以文件系统数据块号转换成 DIO 数据块号*/
            sdio->blocks_available =map_bh->b_size >> sdio->blkbits;        /*DIO 数据块数量*/
            sdio->next_block_for_io =map_bh->b_blocknr << sdio->blkfactor;
                                        /*本次获取映射起始 DIO 数据块号（文件系统中编号）*/
            if (buffer_new(map_bh))
                clean_blockdev_aliases(dio, map_bh);

            if (!sdio->blkfactor)    /*若 DIO 数据块与文件系统数据块相等，不执行后面的调整*/
                goto do_holes;

             /*调整 sdio->blocks_available*/
            blkmask = (1 << sdio->blkfactor) - 1;
            dio_remainder = (sdio->block_in_file & blkmask);
            if (!buffer_new(map_bh))
                sdio->next_block_for_io += dio_remainder;
            sdio->blocks_available -= dio_remainder;
        }

do_holes:
        if (!buffer_mapped(map_bh)) {          /*处理文件空洞情况*/
            loff_t i_size_aligned;
            if (dio->rw & WRITE) {
                page_cache_release(page);
                return -ENOTBLK;
            }
```

```
                i_size_aligned = ALIGN(i_size_read(dio->inode),1 << blkbits);
                if (sdio->block_in_file >=i_size_aligned >> blkbits) {
                    /* We hit eof */
                    page_cache_release(page);
                    goto out;
                }
                zero_user(page, from, 1 << blkbits);       /*用户内存写 0*/
                sdio->block_in_file++;
                from += 1 << blkbits;
                dio->result += 1 << blkbits;
                goto next_block;
            }    /*处理空洞结束*/

            if (unlikely(sdio->blkfactor && !sdio->start_zero_done))
                dio_zero_block(dio, sdio, 0, map_bh);
            this_chunk_blocks = sdio->blocks_available;   /*此次连续映射区 DIO 数据块数量*/
            u = (to - from) >> blkbits;         /*当前页包含的 DIO 数据块总数量*/
            if (this_chunk_blocks > u)
                this_chunk_blocks = u;
            u = sdio->final_block_in_request - sdio->block_in_file; /*实际还没有读写的 DIO 块数量*/
            if (this_chunk_blocks > u)
                this_chunk_blocks = u;
            this_chunk_bytes = this_chunk_blocks << blkbits;   /*此次连续映射区的字节数*/
            BUG_ON(this_chunk_bytes == 0);

            if (this_chunk_blocks == sdio->blocks_available)
                 sdio->boundary = buffer_boundary(map_bh);   /*正好处理到映射区的边界数据块*/

            ret = submit_page_section(dio, sdio, page,from,this_chunk_bytes,
                                                    sdio->next_block_for_io,map_bh);
                /*处理当前页，构建或合并 bio 实例，若不能与后页的页合并则提交 bio 实例*/
            ...
            sdio->next_block_for_io += this_chunk_blocks;     /*下一个处理的文件系统中 DIO 块*/

            sdio->block_in_file += this_chunk_blocks; /*下一个处理的文件内容 DIO 块（逻辑块）*/
            from += this_chunk_bytes;
            dio->result += this_chunk_bytes;
            sdio->blocks_available -= this_chunk_blocks;
next_block:
            BUG_ON(sdio->block_in_file > sdio->final_block_in_request);
            if (sdio->block_in_file == sdio->final_block_in_request)
                break;
        }       /*当前页处理结束*/

        /* Drop the ref which was taken in get_user_pages() */
        page_cache_release(page);
```

```
    }       /*在用户内存处理结束，page[]数组处理结束*/
out:
    return ret;
}
```

do_direct_IO()函数处理流程简述如下：遍历需要读写的文件内容 DIO 数据块，逐页获取保存数据的用户内存 page 实例，对每页先获取映射信息，然后创建或合并 bio 实例并提交，如果页内数据不是映射到块设备中连续数据块，则需要多个 bio 实例。

dio_get_page(dio, sdio)函数获取 dio->pages[sdio->head]指向内存页，即当前处理页，如果 dio->pages[]数组为空或关联页都处理完了，则从用户内存中获取下一批 page 实例。

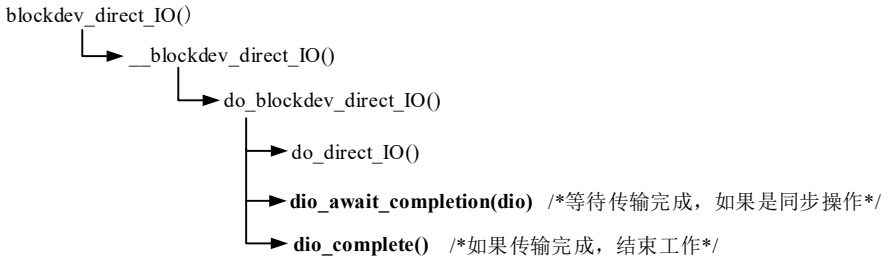
submit_page_section()函数用于为当前页创建或合并 bio 实例，并提交。如果是同步读写，bio 实例的完成回调函数设为 dio_bio_end_io()，异步读写设为 dio_bio_end_aio()，下文将介绍这两个函数实现。

■结束工作

下面介绍同步和异步读写时，直接读写是如何结束的，read()和 write()系统调用执行的是同步读写。

●同步结束

如果是同步读写，当前进程将会睡眠等待提交的 bio 实例都处理完成，然后执行清理工作，函数调用关系如下图所示：

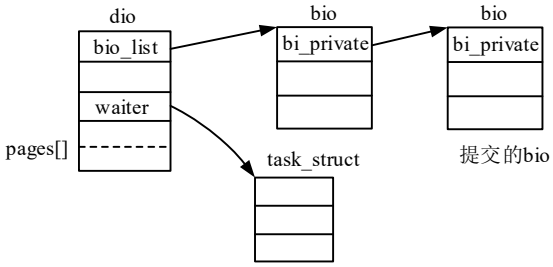


dio_await_completion(dio)函数使当前进程在 dio 上睡眠等待所有 bio 传输完成，完成后继续往下执行，即调用 dio_complete(dio, offset, retval, false)函数执行释放 dio 实例等工作。

下面先看一下同步读写时提交的 bio 实例中赋予的处理完成回调函数，如下所示：

```
static void dio_bio_end_io(struct bio *bio, int error)
{
    struct dio *dio = bio->bi_private;
    unsigned long flags;

    spin_lock_irqsave(&dio->bio_lock, flags);
    bio->bi_private = dio->bio_list;
    dio->bio_list = bio;
    if (--dio->refcount == 1 && dio->waiter)          /*dio->refcount 为 1*/
        wake_up_process(dio->waiter);                 /*唤醒进程*/
    spin_unlock_irqrestore(&dio->bio_lock, flags);
}
```

dio_bio_end_io()函数实际的功用是将 bio 实例添加到 dio->bio_list 链表头部，bio 实例通过 bi_private 成员插入链表，如下图所示。



dio->refcount 成员（初值为 1）记录了提交的但还没有处理完的 bio 实例的数量。bio 实例处理完后，会将 dio->refcount 成员值减 1，如果减 1 之后值为 1 就唤醒在 dio 上睡眠等待的进程。

当前进程在 dio 睡眠等待的 dio_await_completion()函数定义如下：

```
static void dio_await_completion(struct dio *dio)
{
    struct bio *bio;
    do {
        bio = dio_await_one(dio);            /*等待有 bio 处理完成，唤醒*/
        if (bio)
            dio_bio_complete(dio, bio);        /*处理完成的 bio 实例*/
    } while (bio);              /*dio->bio_list 链表为空时跳出循环*/
}
```

dio_await_one(dio)函数定义如下，用于等待 bio 实例处理完成：

```
static struct bio *dio_await_one(struct dio *dio)
{
    unsigned long flags;
    struct bio *bio = NULL;

    spin_lock_irqsave(&dio->bio_lock, flags);

    while (dio->refcount > 1 && dio->bio_list == NULL) {
                            /*提交尚未处理完的 bio 实例大于 1，且没有完成的 bio，睡眠*/
        __set_current_state(TASK_UNINTERRUPTIBLE);        /*不可中断睡眠*/
        dio->waiter = current;
        spin_unlock_irqrestore(&dio->bio_lock, flags);
        io_schedule();            /*进程调度*/
        spin_lock_irqsave(&dio->bio_lock, flags);
        dio->waiter = NULL;
    }
    if (dio->bio_list) {          /*如果 dio->bio_list 链表不为空，有完成的 bio*/
        bio = dio->bio_list;
        dio->bio_list = bio->bi_private;        /*取出链表中第一个完成的 bio，并返回*/
    }
    spin_unlock_irqrestore(&dio->bio_lock, flags);
    return bio;
}
```

dio_await_one(dio)函数如果返回非 NULL，表示有已经处理完的 bio 实例，则 dio_await_completion(dio)函数将调用 dio_bio_complete()函数对完成传输的 bio 实例执行清理工作，代码如下：

```
static int dio_bio_complete(struct dio *dio, struct bio *bio)
{
    const int uptodate = test_bit(BIO_UPTODATE, &bio->bi_flags);
    struct bio_vec *bvec;
    unsigned i;

    if (!uptodate)
        dio->io_error = -EIO;

    if (dio->is_async && dio->rw == READ) {          /*异步读*/
        bio_check_pages_dirty(bio);          /* transfers ownership */
    } else {                                 /*同步读，或写操作*/
        bio_for_each_segment_all(bvec, bio, i) {
            struct page *page = bvec->bv_page;

            if (dio->rw == READ && !PageCompound(page))
                set_page_dirty_lock(page);
            page_cache_release(page);            /*释放内存页*/
        }
        bio_put(bio);
    }
    return uptodate ? 0 : -EIO;
}
```

最后，当提交的 bio 实例都处理完，并执行了 dio 定义的 bio 实例清理工作后，当前进程被唤醒，继续调用函数 dio_complete()，执行 dio 实例的完成工作，函数定义如下：

```
static ssize_t dio_complete(struct dio *dio, loff_t offset, ssize_t ret,bool is_async)
{
    ssize_t transferred = 0;

    if (ret == -EIOCBQUEUED)
        ret = 0;

    if (dio->result) {
        transferred = dio->result;

        /* Check for short read case */
        if ((dio->rw == READ) && ((offset + transferred) > dio->i_size))
            transferred = dio->i_size - offset;
    }

    if (ret == 0)
        ret = dio->page_errors;
    if (ret == 0)
```

```
        ret = dio->io_error;
    if (ret == 0)
        ret = transferred;

    if (dio->end_io && dio->result)
        dio->end_io(dio->iocb, offset, transferred, dio->private);

    if (!(dio->flags & DIO_SKIP_DIO_COUNT))
        inode_dio_end(dio->inode);

    if (is_async) {                 /*异步操作*/
        if (dio->rw & WRITE) {      /*异步写*/
            int err;

            err = generic_write_sync(dio->iocb->ki_filp, offset,transferred);          /*同步页缓存*/
            if (err < 0 && ret > 0)
                ret = err;
        }

        dio->iocb->ki_complete(dio->iocb, ret, 0);
    }

    kmem_cache_free(dio_cache, dio);            /*释放 dio 实例*/
    return ret;
}
```

总之，结束工作包括 bio 实例和 dio 实例的处理，同步操作中会等到所有 bio 实例都执行完传输，并处理完成后，再处理 dio 实例。最后，执行读写的 do_blockdev_direct_IO()函数才返回。

●异步结束

异步读写时，bio 实例完成时的回调函数设为 **dio_bio_end_aio()**，定义如下：

```
static void dio_bio_end_aio(struct bio *bio, int error)
{
    struct dio *dio = bio->bi_private;
    unsigned long remaining;
    unsigned long flags;

    dio_bio_complete(dio, bio);         /*执行 bio 实例的完成工作，见上文*/

    spin_lock_irqsave(&dio->bio_lock, flags);
    remaining = --dio->refcount;            /*bio 实例数减 1*/
    if (remaining == 1 && dio->waiter)
        wake_up_process(dio->waiter);
    spin_unlock_irqrestore(&dio->bio_lock, flags);

    /*do_blockdev_direct_IO()函数最后会调用 drop_refcount(dio)函数将 dio->refcount 减 1*/
```

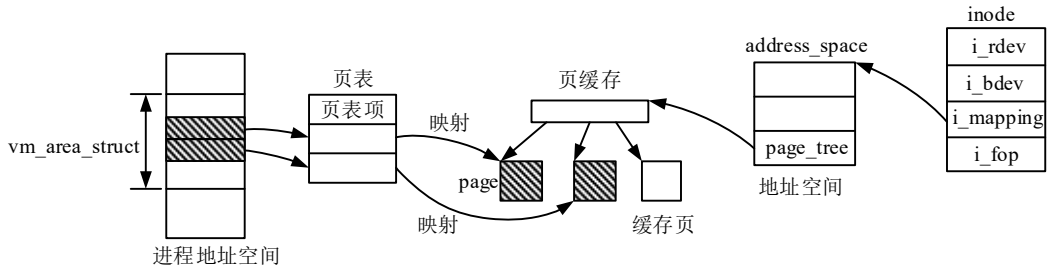
```
    if (remaining == 0) {                /*remaining 为 0，表示所有 bio 都处理完了*/
        if (dio->result && dio->defer_completion) {
            INIT_WORK(&dio->complete_work, dio_aio_complete_work);
            queue_work(dio->inode->i_sb->s_dio_done_wq,&dio->complete_work);
                /*工作添加到工作队列，dio_aio_complete_work()函数调用 dio_complete()函数*/
        } else {
            dio_complete(dio, dio->iocb->ki_pos, 0, true);
        }
    }
}
```

dio_bio_end_aio()函数首先执行 bio 实例的完成工作，将 dio->refcount 值减 1，然后判断 dio->refcount 值是否为 0，如果是则执行 dio 实例的完成工作。

如果 dio->defer_completion 非 0，则由 dio->complete_work 工作调用 dio_complete()函数执行 dio 实例完成工作（延后执行），否则立即调用 dio_complete()函数执行 dio 实例完成工作。

11.2.4 文件映射函数

所谓文件映射，就是进程通过 mmap()/mmap2()系统调用等，将一段连续的文件内容映射到进程连续的虚拟内存上（线性映射，也有非线性映射），如下图所示：



建立映射就是修改进程页表项，使进程虚拟页映射到文件内容页缓存中的页，从而使进程可以像操作内存一样操作缓存页。

mmap()/mmap2()系统调用创建文件映射时，将调用映射文件的文件操作结构中的 **f_op->mmap(file, vma)**函数建立文件映射。对于基于块设备的普通文件，mmap()函数主要是对虚拟内存域 vm_area_struct 实例的 vm_operations_struct 结构体指针成员 vm_ops 赋值，并没有建立映射，映射在缺页异常处理中建立。

vm_operations_struct 结构体中包含的函数在缺页异常处理函数中调用，用于查找或创建缓存页、建立映射等，详见第 4 章。

本节主要介绍内核定义的通用 vm_operations_struct 实例中函数的实现。

1 通用映射函数

内核多数文件系统类型定义的文件操作 file_operations 实例中，mmap()函数引用的是内核通用映射函数 **generic_file_mmap()**，定义在/mm/filemap.c 文件内，代码如下：

```
int generic_file_mmap(struct file * file, struct vm_area_struct * vma)
{
    struct address_space *mapping = file->f_mapping;

    if (!mapping->a_ops->readpage)
        return -ENOEXEC;
    file_accessed(file);
```

```
    vma->vm_ops = &generic_file_vm_ops;    /*/mm/filemap.c*/
    return 0;
}
```

通用映射函数的主要工作就是将通用的 vm_operations_struct 结构体实例 generic_file_vm_ops 赋予虚拟内存域 vma->vm_ops 成员。

generic_file_vm_ops 实例定义如下（/mm/filemap.c）：

```
const struct vm_operations_struct   generic_file_vm_ops = {
    .fault          = filemap_fault,          /*查找/创建缓存页，/mm/filemap.c*/
    .map_pages      = filemap_map_pages,   /*映射现有缓存页，读缺页异常调用，/mm/filemap.c*/
    .page_mkwrite= filemap_page_mkwrite,    /*置缓存页脏标记，/mm/filemap.c*/
};
```

在文件操作结构中的 mmap()函数中并没有建立映射，映射在缺页异常处理中建立。在缺页异常处理函数中需要调用 vm_operations_struct 实例中的函数。

2 查找/创建缓存页

当进程访问了尚未建立映射的文件映射区时，将会触发缺页异常，在缺页异常处理程序中将会调用内存域操作结构中的 **vm_ops->fault(vma, &vmf)**函数，在文件内容页缓存中查找或创建对应的缓存页，然后再修改进程页表项，最终建立虚拟内存与缓存页之间的映射关系，详见第 4 章。

generic_file_vm_ops 实例中 fault()函数为 **filemap_fault()**，用于查找或创建所需的缓存页。

在介绍函数实现之前，先回顾一下 vm_fault 结构体的定义，它用于表示引起缺页异常的内存域信息：

```
struct vm_fault {
    unsigned int flags;         /*缺页异常标记  FAULT_FLAG_xxx*/
    pgoff_t pgoff;              /*所缺页在虚拟内存域中的偏移量，页偏移*/
    void __user *virtual_address;     /*异常虚拟地址*/
    struct page *cow_page;              /*写时复制新分配内存页 page 指针*/
    struct page *page;                  /*所需映射缓存页 page 实例指针*/

    /* map_pages()函数专用成员*/
    pgoff_t max_pgoff;        /*映射页最大偏移量*/
    pte_t *pte;                 /*页表项指针*/
};
```

filemap_fault()函数定义在/mm/filemap.c 文件内，如果执行成功参数 vmf->page 成员将指向所需缓存页 page 实例，函数代码如下：

```
int filemap_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
{
    int error;
    struct file *file = vma->vm_file;              /*文件 file 实例*/
    struct address_space *mapping = file->f_mapping;
    struct file_ra_state *ra = &file->f_ra;       /*文件预读结构实例，见上文*/
    struct inode *inode = mapping->host;
    pgoff_t offset = vmf->pgoff;          /*缺页在内存域中偏移量*/
    struct page *page;
```

```
    loff_t size;
    int ret = 0;

    size = round_up(i_size_read(inode), PAGE_CACHE_SIZE);    /*文件大小页对齐*/
    if (offset >= size >> PAGE_CACHE_SHIFT)   /*页偏移量比文件内容大，返回错误码*/
        return VM_FAULT_SIGBUS;

    page = find_get_page(mapping, offset);    /*在页缓存中查找页，看是否已经存在所需缓存页*/
    if (likely(page) && !(vmf->flags & FAULT_FLAG_TRIED)) {
        do_async_mmap_readahead(vma, ra, file, page, offset);   /*/mm/filemap.c*/
                /*如果缓存页已存在且 page 设置了 PG_readahead 标记位，执行异步预读*/
    } else if (!page) {         /*未查找到所需缓存页*/
        do_sync_mmap_readahead(vma, ra, file, offset);    /*执行同步预读，/mm/filemap.c*/
        count_vm_event(PGMAJFAULT);
        mem_cgroup_count_vm_event(vma->vm_mm, PGMAJFAULT);
        ret = VM_FAULT_MAJOR;
retry_find:
        page = find_get_page(mapping, offset);   /*再次执行查找缓存页*/
        if (!page)                    /*如果查找仍然不成功，跳至 no_cached_page 处执行*/
            goto no_cached_page;
    }

    /*运行至此表示已经获得了所需的缓存页*/
    if (!lock_page_or_retry(page, vma->vm_mm, vmf->flags)) {    /*锁定缓存页，调用者负责解锁*/
         page_cache_release(page);
         return ret | VM_FAULT_RETRY;
    }

    if (unlikely(page->mapping != mapping)) {
        unlock_page(page);
        put_page(page);
        goto retry_find;
    }
    VM_BUG_ON_PAGE(page->index != offset, page);

    if (unlikely(!PageUptodate(page)))     /*检查缓存页数据有效性*/
        goto page_not_uptodate;            /*无效跳至 page_not_uptodate 处*/

    /*至此表示缓存页存在，且数据有效*/
    size = round_up(i_size_read(inode), PAGE_CACHE_SIZE);   /*检查文件大小*/
    if (unlikely(offset >= size >> PAGE_CACHE_SHIFT)) {       /*超过文件实际大小*/
        unlock_page(page);
        page_cache_release(page);    /*释放缓存页*/
        return VM_FAULT_SIGBUS;
    }
```

```
        vmf->page = page;        /*缓存页 page 实例*/
        return ret | VM_FAULT_LOCKED;     /*已锁定缓存页*/

    no_cached_page:     /*如果预读操作中不能读取所需的缓存页，跳转至此处*/
        error = page_cache_read(file, offset);     /*分配缓存页，插入页缓存，读数据，/mm/filemap.c*/
        if (error >= 0)
            goto retry_find;     /*重试，再查找一遍*/

        /*page_cache_read()函数发生错误*/
        if (error == -ENOMEM)
            return VM_FAULT_OOM;     /*需启动 OOM 机制，释放内存*/
        return VM_FAULT_SIGBUS;

  page_not_uptodate:     /*缓存页数据无效，跳转至此处*/
        ClearPageError(page);
        error = mapping->a_ops->readpage(file, page);    /*从块设备读取数据至缓存页*/
        ...        /*错误处理*/
        page_cache_release(page);

        if (!error || error == AOP_TRUNCATED_PAGE)
            goto retry_find;    /*重试，再次查找*/

        shrink_readahead_size_eio(file, ra);
        return VM_FAULT_SIGBUS;
    }
```

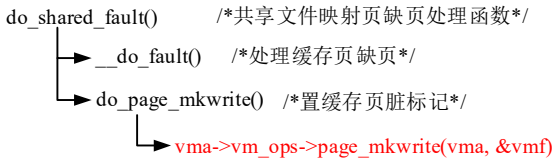
filemap_fault()函数执行流程比较清晰，函数在页缓存中查找所需的缓存页，如果不存在则启动预读操作后再进行查找（如果预不成功则执行单页读操作），获得缓存页后再判断缓存页数据是否有效，无效则从块设备中读取数据。

缓存页存在且数据有效，则将 page 实例将赋予 vmf->page 成员，返回值设置缓存页已被锁定标记，由函数调用者负责解锁。

generic_file_vm_ops 实例中 map_pages()成员函数 **filemap_map_pages()**在读文件映射页的缺页异常处理函数中调用，主要是在页缓存中查找缺页周边的页，如果存在且数据有效，则修改进程页表项，建立映射，函数源代码请读者自行阅读。

3 设置缓存页脏标记

用户空间缺页异常处理程序在遇到共享文件映射页缺页时，调用 do_shared_fault()函数进行处理（写操作异常），函数调用关系如下图所示：



上面的__do_fault()函数负责查找/创建缓存页，do_page_mkwrite()函数调用 vm_ops->page_mkwrite()函数用于通知地址空间设置缓存页的脏标记（使其可写），以便数据回写机制能回写此缓存页。

generic_file_vm_ops 实例中 page_mkwrite()函数为 **filemap_page_mkwrite()**，定义如下（/mm/filemap.c）：

```
int filemap_page_mkwrite(struct vm_area_struct *vma, struct vm_fault *vmf)
{
    struct page *page = vmf->page;
    struct inode *inode = file_inode(vma->vm_file);
    int ret = VM_FAULT_LOCKED;          /*正常返回值*/

    sb_start_pagefault(inode->i_sb);   /*获取写超级块权限，/include/linux/fs.h*/
    file_update_time(vma->vm_file);
    lock_page(page);
    ...
    set_page_dirty(page);          /*设置缓存页脏标记，需执行回写，/mm/page-writeback.c*/
    wait_for_stable_page(page);
out:
    sb_end_pagefault(inode->i_sb);
    return ret;
}
```

filemap_page_mkwrite()函数主要是调用 **set_page_dirty()**函数设置页脏标记，此函数是一个通用的接口函数，在内核多处被调用，函数定义如下（/mm/page-writeback.c）：

```
int set_page_dirty(struct page *page)
{
    struct address_space *mapping = page_mapping(page);

    if (likely(mapping)) {              /*页缓存中的页*/
        int (*spd)(struct page *) = mapping->a_ops->set_page_dirty;
                                                    /*地址空间操作结构中定义的函数*/
        if (PageReclaim(page))       /*清除 PG_readahead 标记位（预读标记位）*/
            ClearPageReclaim(page);
#ifdef CONFIG_BLOCK
        if (!spd)      /*如果地址空间操作结构中没有定义 set_page_dirty()函数则设为通用的函数*/
            spd = __set_page_dirty_buffers;     /*通用函数，设置缓存页脏标记，/fs/buffer.c*/
#endif
        return (*spd)(page);       /*调用地址空间操作结构中函数或通用函数*/
    }
    if (!PageDirty(page)) {
        if (!TestSetPageDirty(page))          /*测试并设置页脏标记*/
            return 1;
    }
    return 0;        /*返回 0，表示原来就是脏的，返回 1 表示新置脏标记*/
}
```

set_page_dirty()函数内将调用 a_ops->set_page_dirty()函数设置缓存页脏标记，如果没有定义此函数（通常没有定义）则调用通用的__set_page_dirty_buffers()函数，此函数定义在/fs/buffer.c 文件内。

```
int __set_page_dirty_buffers(struct page *page)
{
    int newly_dirty;
```

```
    struct mem_cgroup *memcg;
    struct address_space *mapping = page_mapping(page);

    if (unlikely(!mapping))
        return !TestSetPageDirty(page);

    spin_lock(&mapping->private_lock);
    if (page_has_buffers(page)) {       /*缓存页具有块缓存头实例链表*/
        struct buffer_head *head = page_buffers(page);
        struct buffer_head *bh = head;

        do {
            set_buffer_dirty(bh);     /*设置所有块缓存头脏标记位*/
            bh = bh->b_this_page;
        } while (bh != head);
    }
    memcg = mem_cgroup_begin_page_stat(page);
    newly_dirty = !TestSetPageDirty(page);  /*返回原脏标记位值（取反），并置位页脏标记位*/
    spin_unlock(&mapping->private_lock);

    if (newly_dirty)       /*缓存页原来不是脏页*/
        __set_page_dirty(page, mapping, memcg, 1);     /*设置基数树中脏标记，/fs/buffer.c*/

    mem_cgroup_end_page_stat(memcg);

    if (newly_dirty)       /*缓存页原来不是脏页*/
        __mark_inode_dirty(mapping->host, I_DIRTY_PAGES);
                                /*设置 inode 脏标记，见下一节，/fs/fs-writeback.c*/
    return newly_dirty;         /*本次是否是新置脏标记（原来不脏）*/
}
```

__set_page_dirty_buffers()函数需要设置缓存页中包含的所有块缓存头的脏标记，设置页 page 实例中的脏标记。如果缓存页原来没有设置脏标记，还需要设置基数树中的脏标记和文件 inode 实例中的脏标记。

__mark_inode_dirty()函数用于设置 inode 脏标记，后面再介绍函数实现。

__set_page_dirty()函数用于设置基数树中的脏标记标签，以及更新系统脏页统计量，函数代码如下：

```
static void __set_page_dirty(struct page *page, struct address_space *mapping,
                             struct mem_cgroup *memcg, int warn)
{
    unsigned long flags;

    spin_lock_irqsave(&mapping->tree_lock, flags);
    if (page->mapping) {     /* Race with truncate? */
        WARN_ON_ONCE(warn && !PageUptodate(page));
        account_page_dirtied(page, mapping, memcg);         /*更新统计量，/mm/page-writeback.c*/
        radix_tree_tag_set(&mapping->page_tree,
                page_index(page), PAGECACHE_TAG_DIRTY);         /*设置基数树中的标签值*/
    }
```

```
    spin_unlock_irqrestore(&mapping->tree_lock, flags);
}
```

account_page_dirtied()函数用于更新脏页统计量，这在脏页平衡、页回收等时机会用到，函数定义如下：

```
void account_page_dirtied(struct page *page, struct address_space *mapping,struct mem_cgroup *memcg)
{
    struct inode *inode = mapping->host;

    trace_writeback_dirty_page(page, mapping);

    if (mapping_cap_account_dirty(mapping)) {
        struct bdi_writeback *wb;

        inode_attach_wb(inode, page);          /*inode 绑定 bdi_writeback 实例*/
        wb = inode_to_wb(inode);

        mem_cgroup_inc_page_stat(memcg, MEM_CGROUP_STAT_DIRTY);
        __inc_zone_page_state(page, NR_FILE_DIRTY);        /*增加脏页统计量值*/
        __inc_zone_page_state(page, NR_DIRTIED);
        __inc_wb_stat(wb, WB_RECLAIMABLE);
        __inc_wb_stat(wb, WB_DIRTIED);
        task_io_account_write(PAGE_CACHE_SIZE);
        current->nr_dirtied++;                  /*当前进程置脏页数量加 1*/
        this_cpu_inc(bdp_ratelimits);          /*当前 CPU 核置脏页数量加 1*/
    }
}
```

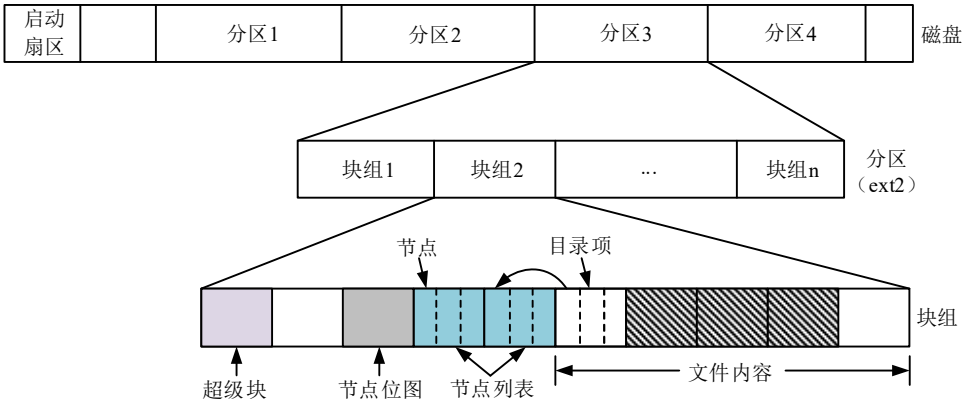
11.3 数据回写

前面介绍普通文件写操作函数时，如果不是同步写，则只是将用户数据写入页缓存，而由内核数据回写机制负责在适当的时候将页缓存中数据写出到块设备。

回写的数据不只有文件内容，还有保存在 inode 中的文件元数据、超级块数据等。本节介绍内核执行回写的策略，以及执行回写的方式。

11.3.1 概述

先回顾一下磁盘保存数据的格式，如下图所示：



磁盘开头是启动扇区等，保存的是磁盘的分区信息。分区被格式化成文件系统，下面以分区 3 的 ext2 文件系统为例，说明文件系统的格式。

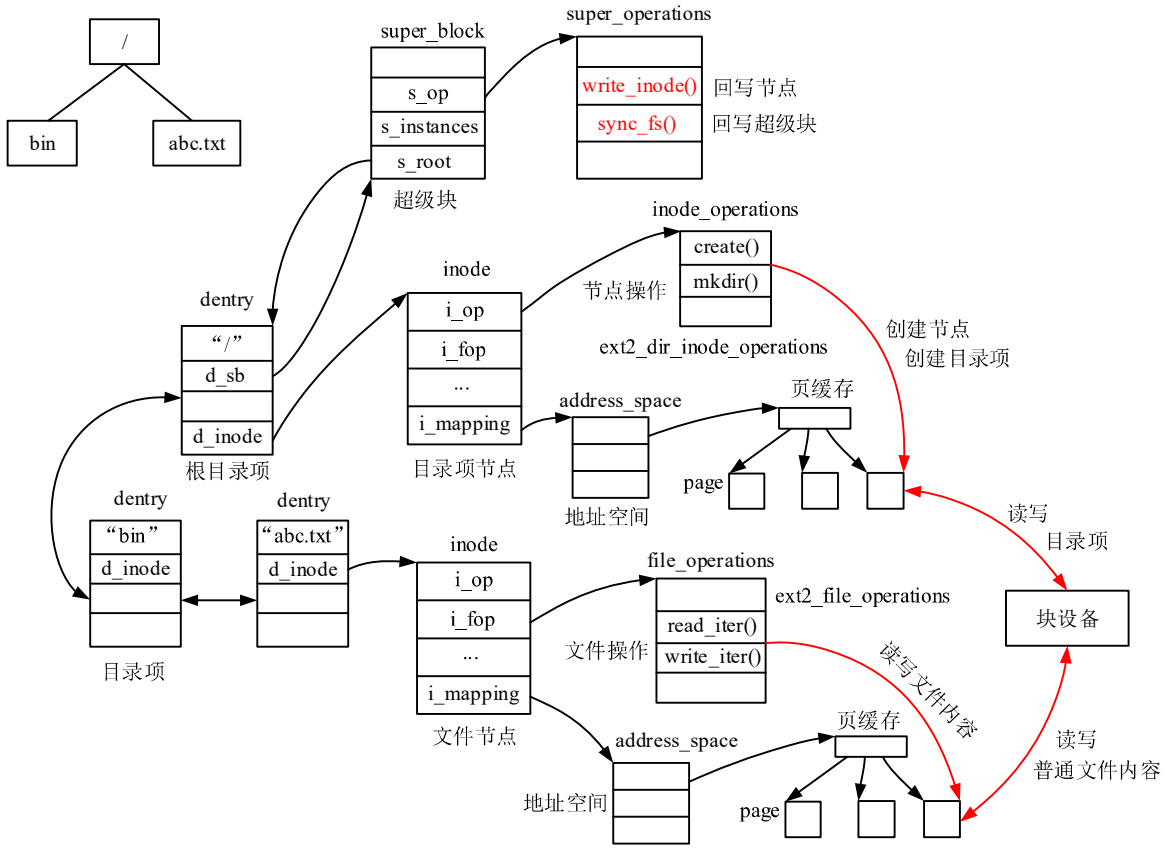
ext2 文件系统由块组组成，每个块组中包含超级块、节点位图、节点列表、保存文件内容的数据块，以及组描述符列表、数据块位图（位于超级块与节点位图之间，图中未画出）等。

超级块保存了文件系统整体的信息，节点列表中保存的是节点，节点中保存的是文件或目录项的元信息，目录和文件名称保存在目录项中。目录项作为目录文件的内容保存在数据块中，目录项中记录了关联的节点编号，节点中记录了文件的元信息，包括文件内容存放在哪些数据块中等。

目录项也是文件，只不过其文件内容保存的是目录项，节点列表中第一个节点对应根目录项文件，其文件内容从文件内容区域中的第一个数据块开始。

以上是块设备中数据的存放方式，导入内核后，由数据结构实例来表示以上信息。例如，超级块由 super_block 表示，目录项由 dentry 表示，节点由 inode 表示等。

下图示意了将块设备中某个分区挂载为根文件系统时，在内核中创建的数据结构实例：



用户对普通文件内容的写操作，写入页缓存后，内核会将写入数据同步到保存文件内容的数据块中，对文件元信息的修改将通过 inode 同步到文件系统节点中。

创建目录项、文件或重命名文件等操作，会修改目录项内容，目录项其实是上一级目录文件的内容，这些操作就是对上一级目录文件内容的修改，其实与普通文件内容的修改类似。

节点操作结构中的 create()函数用于创建普通文件，函数内将从文件系统中分配节点，修改文件系统中节点位图等信息，mkdir()函数用于创建目录项，与创建文件操作类似。还有其它的如重命名、创建设备节点、删除文件和目录项等，都是对上一级目录文件内容的操作。

目录文件内容也由地址空间页缓存管理，也由地址空间操作结构中的函数实现与块设备同步，这与普文件内容的操作相同。

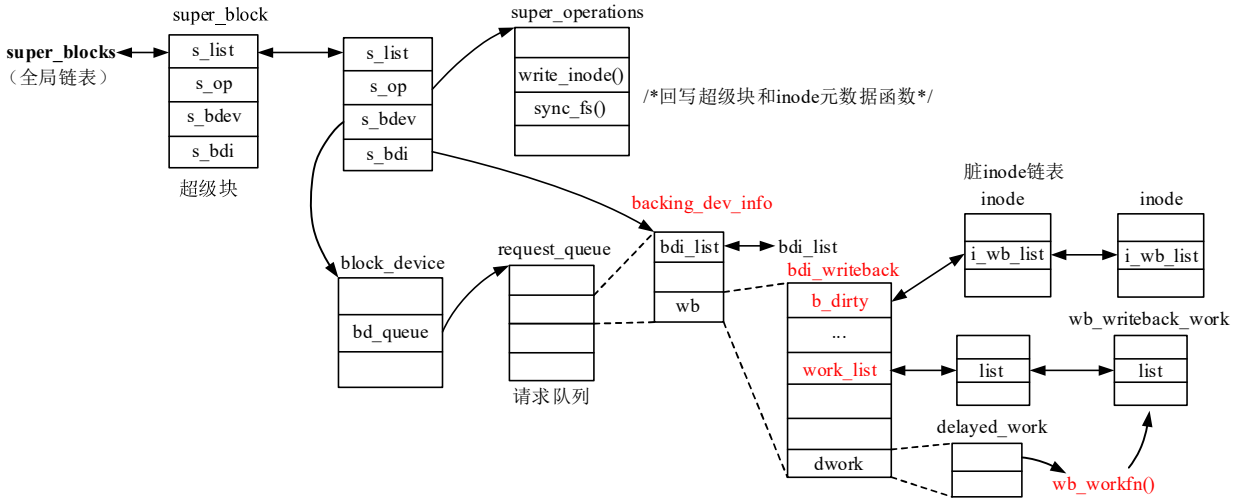
节点（inode）数据修改后，调用超级块操作结构中的 **write_inode()**函数写出到块设备，超级块数据修改后由超级块操作结构中的 **sync_fs()**函数写出到块设备。

如果是对裸块设备进行写操作，就是将用户数据写入块设备文件 inode 中管理的页缓存中，内核会专

门同步块设备文件 inode 中的页缓存数据。

总之，数据回写概括起来主要包含两个内容：一是超级块 super_block 实例的回写，二是节点 inode 实例的回写，包括文件内容（页缓存）的回写和节点元数据的回写（节点自身的回写）。

那么内核如何执行回写操作呢？请先看下图：



内核所有挂载文件系统的超级块 super_block 实例由全局双链表管理，其 s_bdi 成员指向块设备请求队列中的 backing_dev_info 结构体成员。backing_dev_info 结构体中的 bdi_writeback 结构体成员，用于实现数据回写。超级块操作结构中的 write_inode()函数用于回写 inode 元数据，sync_fs()函数用于回写超级块。

bdi_writeback 结构体包含脏 inode 双链表。当文件内容或节点元数据修改时（变脏），会将文件 inode 实例添加到 bdi_writeback 结构体中的脏 inode 实例双链表。

bdi_writeback 结构体中 work_list 双链表管理的是 wb_writeback_work 结构体实例，每个实例表示发起的一次回写操作，其中包含回写控制参数。

bdi_writeback 结构体中 dwork 成员是延时工作 delayed_work 结构体成员，其执行函数为 wb_workfn()，此函数负责执行回写操作。

wb_workfn()函数首先遍历 wb_writeback_work 实例双链表，对每个实例执行一次回写操作，回写操作将取出脏 inode 双链表中实例，对 inode 实例执行回写操作（回写缓存页和元数据），然后判断是否要执行周期回写和脏页平衡（后台）回写，如果需要则构建 wb_writeback_work 实例（不插入 work_list 双链表），执行回写操作。

如果要回写指定文件系统，则获取 super_block 实例关联的 backing_dev_info 实例，将其内嵌的 bdi_writeback 结构体中 dwork 成员（延时工作），添加到工作队列中，激活工作（延时调用 wb_workfn()函数）。

如果要进行系统回写，则遍历 backing_dev_info 实例双链表，对每个实例激活其中的 dwork 延时工作。

回写操作在哪些时机会被触发呢？内核回写的策略如下：

（1）周期性地触发数据回写。

（2）在系统脏页数量较多时（写缓存页时判断），触发数据回写，称为脏页平衡或后台回写。

（3）空闲页紧张需要更多空闲页时，触发数据回写，以释放内存。

（4）用户进程通过系统调用发出数据同步的指令时，触发数据回写。

11.3.2 准备工作

准备工作先介绍回写机制相关数据结构的定义及初始化，然后介绍设置 inode 脏标记接口函数的实现。

1 数据结构

块设备请求队列 request_queue 结构体中 backing_dev_info 结构体成员包含了后备存储设备的信息：

```
struct request_queue {
    ...
    struct backing_dev_info   backing_dev_info;     /*/include/linux/backing-dev-defs.h*/
    ...
}
```

■backing_dev_info

backing_dev_info 结构体定义在/include/linux/backing-dev-defs.h 头文件：

```
struct backing_dev_info {
    struct list_head    bdi_list;       /*双链表成员，将实例添加到 bdi_list 全局双链表*/
    unsigned long    ra_pages;      /*预读窗口中最大预读页数*/
    unsigned int    capabilities;
                    /*设备能力，初始为 BDI_CAP_CGROUP_WRITEBACK，/include/linux/backing-dev.h*/
    congested_fn    *congested_fn;     /*函数指针，Function pointer if device is md/dm */
    void    *congested_data;            /*congested_fn()函数参数*/

    char    *name;          /*名称*/
    unsigned int    min_ratio;
    unsigned int    max_ratio, max_prop_frac;
    atomic_long_t    tot_write_bandwidth;

    struct bdi_writeback wb;       /*数据回写的根信息，/include/linux/backing-dev-defs.h*/

#ifdef CONFIG_CGROUP_WRITEBACK
    struct radix_tree_root    cgwb_tree;          /* radix tree of active cgroup wbs */
    struct rb_root    cgwb_congested_tree;       /* their congested states */
    atomic_t    usage_cnt;          /* counts both cgwbs and cgwb_contested's */
#else
    struct bdi_writeback_congested    *wb_congested;
                                        /*回写拥塞控制，/include/linux/backing-dev-defs.h*/
#endif
    wait_queue_head_t wb_waitq;          /*进程等待队列头*/

    struct device *dev;                 /*指向表示实例的 device 实例，注册实例时创建*/
    struct timer_list    laptop_mode_wb_timer;       /*定时器*/

#ifdef CONFIG_DEBUG_FS
    struct dentry *debug_dir;
    struct dentry *debug_stats;
#endif
};
```

backing_dev_info 结构体主要成员简介如下：

●**bdi_list**：双链表成员，内核所有 backing_dev_info 实例添加到全局链表 bdi_list，系统回写时遍历此双链表。

●**ra_pages**：页缓存预读中预读的最大缓存页数。

●**capabilities**：设备能力，初始化时设为 BDI_CAP_CGROUP_WRITEBACK。

能力标记位如下所示（/include/linux/backing-dev.h）：

```
#define   BDI_CAP_NO_ACCT_DIRTY   0x00000001      /*不统计脏页*/
#define   BDI_CAP_NO_WRITEBACK    0x00000002      /*不执行回写*/
#define   BDI_CAP_NO_ACCT_WB      0x00000004      /*不自动统计回写页*/
#define   BDI_CAP_STABLE_WRITES   0x00000008
#define   BDI_CAP_STRICTLIMIT     0x00000010     /*脏页数量保持在阀值以下*/
#define   BDI_CAP_CGROUP_WRITEBACK   0x00000020     /*初始值*/

#define BDI_CAP_NO_ACCT_AND_WRITEBACK \
            (BDI_CAP_NO_WRITEBACK | BDI_CAP_NO_ACCT_DIRTY | BDI_CAP_NO_ACCT_WB)
```

●**congested_fn**：函数指针，函数原型如下：

```
typedef int (congested_fn)(void *, int);
```

●**congested_data**：指向的数据为 congested_fn()函数参数。

●**wb**：bdi_writeback 结构体成员，表示数据回写的根信息，后面将专门介绍。

●**wb_congested**：bdi_writeback_congested 结构体指针，结构体定义如下：

```
struct bdi_writeback_congested {
    unsigned long    state;          /* WB_[a]sync_congested flags */
    atomic_t    refcnt;              /* nr of attached wb's and blkg */

#ifdef CONFIG_CGROUP_WRITEBACK
    struct backing_dev_info *bdi;    /* the associated bdi */
    int blkcg_id;                    /* ID of the associated blkcg */
    struct rb_node rb_node;          /* on bdi->cgwb_congestion_tree */
#endif
};
```

bdi_writeback_congested 结构体中 **state** 状态成员，取值如下（/include/linux/backing-dev-defs.h）：

```
enum wb_congested_state {
    WB_async_congested,   /*异步操作请求拥塞*/
    WB_sync_congested,    /*同步操作请求拥塞*/
};
```

■bdi_writeback

backing_dev_info 结构体内嵌的 bdi_writeback 结构体，用于表示数据回写所需的信息，结构体定义在头文件/include/linux/backing-dev-defs.h：

```
struct bdi_writeback {
    struct backing_dev_info    *bdi;          /*指向 backing_dev_info 实例*/

    unsigned long    state;                 /*状态*/
    unsigned long    last_old_flush;        /*最近一次执行周期回写的时间*/
```

```
    /*脏 inode 链表*/
    struct list_head    b_dirty;        /*元数据或缓存页修改了的脏 inode 实例链表*/
    struct list_head    b_io;           /*正在执行回写的 inode 实例，来自 b_dirty 链表*/
    struct list_head    b_more_io;      /*正在回写，且要增回写的 inode 实例*/
    struct list_head    b_dirty_time;   /*只修改了时间戳的 inode 实例链表*/
    spinlock_t list_lock;               /*保护以上 inode 双链表的自旋锁*/

    struct percpu_counter stat[NR_WB_STAT_ITEMS];
                                        /*各 CPU 核的统计量，/include/linux/percpu_counter.h*/

    struct bdi_writeback_congested *congested;
                                        /*回写拥塞，与 backing_dev_info.wb_congested 相同*/
    /*时间戳及带宽成员*/
    unsigned long    bw_time_stamp;         /* last time write bw is updated */
    unsigned long    dirtied_stamp;
    unsigned long    written_stamp;         /* pages written at bw_time_stamp */
    unsigned long    write_bandwidth;       /* the estimated write bandwidth */
    unsigned long    avg_write_bandwidth;   /* further smoothed write bw, > 0 */

    unsigned long    dirty_ratelimit;
    unsigned long    balanced_dirty_ratelimit;

    struct fprop_local_percpu    completions;       /*percpu 变量，/include/linux/flex_proportions.h*/
    int dirty_exceeded;

    spinlock_t work_lock;               /*保护 work_list 双链表的自旋锁*/
    struct list_head work_list;         /*wb_writeback_work 实例双链表，每个实例表示一次回写工作*/
    struct delayed_work dwork;          /*延时工作，执行实质性的数据回写操作*/

  #ifdef CONFIG_CGROUP_WRITEBACK
    struct percpu_ref refcnt;       /* used only for !root wb's */
    struct fprop_local_percpu memcg_completions;
    struct cgroup_subsys_state *memcg_css;      /* the associated memcg */
    struct cgroup_subsys_state *blkcg_css;      /* and blkcg */
    struct list_head memcg_node;        /* anchored at memcg->cgwb_list */
    struct list_head blkcg_node;        /* anchored at blkcg->cgwb_list */

    union {
        struct work_struct release_work;
        struct rcu_head rcu;
    };
  #endif
};
```

bdi_writeback 结构体中主要成员简介如下：

●**b_dirty**：inode 链表，设置 inode 脏时，将 inode 实例添加到此链表，表示需要回写的 inode 实例。

●**b_dirty_time**：只修改了时间戳的 inode 实例双链表。

●**b_io**：在执行回写的 inode 实例链表，来自于 b_dirty 双链表。

●**b_more_io**：正在回写，且要增回写的 inode 实例。

●**work_list**：双链表成中，管理 wb_writeback_work 结构体实例，这是释放页等时机创建的，结构体定义见下文。

●**dwork**：延时工作 delayed_work 结构体成员，工作执行函数 **wb_workfn()**用于执行实际的回写工作，详见下文。

●**state**：状态成员，各比特位语义定义如下（/include/linux/backing-dev-defs.h）：

```
enum wb_state {
    WB_registered,          /*backing_dev_info 实例已经注册，bit0*/
    WB_writeback_running,   /*正在回写，bit1*/
    WB_has_dirty_io,        /*有脏 inode 需要回写，bit2*/
};
```

●**completions**：fprop_local_percpu 变量（percpu），定义在/include/linux/flex_proportions.h 头文件：

```
struct fprop_local_percpu {
    struct percpu_counter events;       /*本地事件计数*/

    unsigned int period;        /* Period in which we last updated events */
    raw_spinlock_t lock;        /* Protect period and numerator */
};
```

●**stat[NR_WB_STAT_ITEMS]**：percpu_counter 结构体数组，percpu 的统计量，统计项目定义如下：

```
enum wb_stat_item {
    WB_RECLAIMABLE,
    WB_WRITEBACK,
    WB_DIRTIED,
    WB_WRITTEN,
    NR_WB_STAT_ITEMS
};
```

■wb_writeback_work

bdi_writeback 结构体中 **work_list** 双链表管理的是 wb_writeback_work 结构体实例，表示一次数据回写工作（控制数据回写，可视为 writeback_control 的子集），结构体定义在/fs/fs-writeback.c 文件内：

```
struct wb_writeback_work {
    long    nr_pages;           /*剩余需要回写页数量*/
    struct super_block    *sb;  /*超级块实例指针*/
    unsigned long *older_than_this;
    enum writeback_sync_modes    sync_mode;
    unsigned int tagged_writepages:1;
    unsigned int for_kupdate:1;         /*周期回写*/
    unsigned int range_cyclic:1;
    unsigned int for_background:1;      /*由脏页平衡触发的数据回写（后台回写）*/
    unsigned int for_sync:1;            /* sync()系统调用或 WB_SYNC_ALL 回写*/
    unsigned int auto_free:1;           /* free on completion */
    unsigned int single_wait:1;
```

```
    unsigned int single_done:1;
    enum wb_reason reason;          /*回写原因，/include/linux/writeback.h*/

    struct list_head list;          /*双链表成员，将实例添加到 bdi_writeback 实例中双链表*/
    struct wb_completion *done;  /*只含一个计数值，如果调用者在等待回写，则设置此函数指针*/
};
```

wb_writeback_work 结构体主要成员简介如下：

●**nr_pages**：剩余需要回写的页数。

●**sb**：文件系统超级块 super_block 实例指针。

●**list**：双链表成员，将实例添加到 bdi_writeback 实例中 work_list 双链表。

●**done**：指向 wb_completion 结构体，定义如下（/fs/fs-writeback.c）：

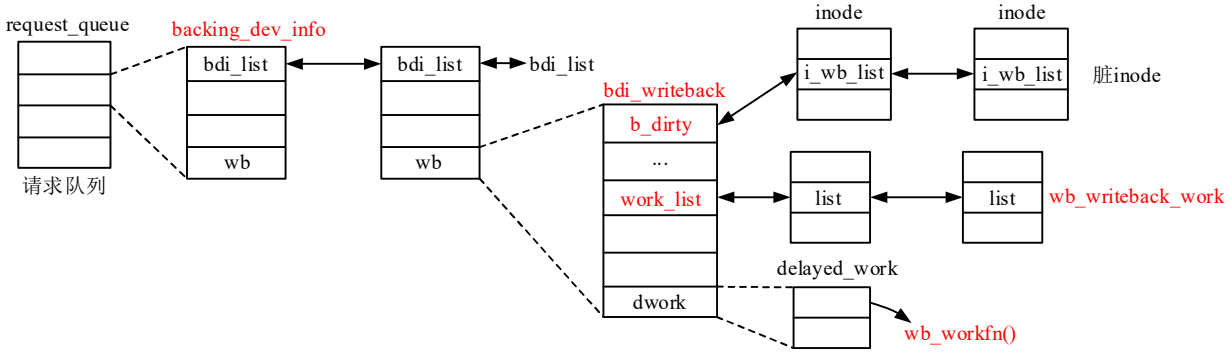
```
struct wb_completion {
    atomic_t        cnt;        /*计数值*/
};
```

●**reason**：回写原因，即由谁发起的数据回写，由枚举类型表示，如下（/include/linux/writeback.h）：

```
enum wb_reason {
    WB_REASON_BACKGROUND,           /*脏页平稳*/
    WB_REASON_TRY_TO_FREE_PAGES,    /*try_to_free_pages()函数发起*/
    WB_REASON_SYNC,                 /*由 sync()系统调用发起*/
    WB_REASON_PERIODIC,             /*周期回写*/
    WB_REASON_LAPTOP_TIMER,
    WB_REASON_FREE_MORE_MEM,        /*由 free_more_memory()函数发起*/
    WB_REASON_FS_FREE_SPACE,
    WB_REASON_FORKER_THREAD,        /*不使用的标记*/

    WB_REASON_MAX,
};
```

需要执行数据回写时，将构建 wb_writeback_work 结构体实例，并添加到 bdi_writeback 实例 work_list 双链表中，也可能不插入链表。执行回写的 wb_workfn()函数中将依 wb_writeback_work 实例的控制，执行回写操作。

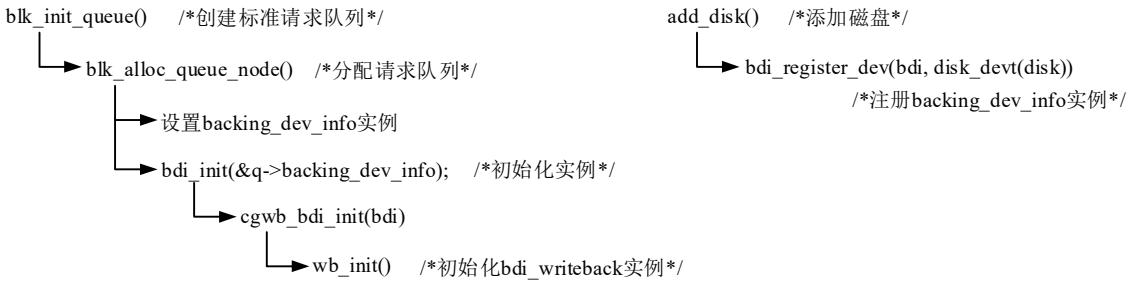
以上数据结构组织关系如下图所示：



2 初始化

在内核初始化中将为 backing_dev_info 实例创建 bdi_class 设备类，并创建工作队列。

请求队列中内嵌的 backing_dev_info 实例在分配请求队列时初始化，并在添加磁盘时注册，函数调用关系如下图所示：



■创建工作队列

backing_dev_info 实例关联的 device 实例归入 bdi_class 设备类，它在什么时候创建呢？此设备类的创建函数定义在/mm/backing-dev.c 文件内。

```
static __init int bdi_class_init(void)
{
    bdi_class = class_create(THIS_MODULE, "bdi");        /*创建设备类*/
    ...        /*错误处理*/

    bdi_class->dev_groups = bdi_dev_groups;          /*设备属性组，请读者自行查阅*/
    bdi_debug_init();
    return 0;
}
postcore_initcall(bdi_class_init);          /*内核初始化子系统时调用*/

static int __init default_bdi_init(void)
{
    int err;

    bdi_wq = alloc_workqueue("writeback", WQ_MEM_RECLAIM | WQ_FREEZABLE |
                                          WQ_UNBOUND | WQ_SYSFS, 0);          /*创建工作队列*/
    if (!bdi_wq)
        return -ENOMEM;

    err = bdi_init(&noop_backing_dev_info);          /*初始化内核定义的 backing_dev_info 实例*/

    return err;
}
subsys_initcall(default_bdi_init);        /*内核初始化子系统时调用*/
```

default_bdi_init()函数中将创建一个工作队列，赋予 **bdi_wq** 全局变量。bdi_writeback 结构体实例中的延时工作 dwork 在激活时将添加到此工作队列。

■初始化 backing_dev_info

内核在分配请求队列 request_queue 实例的 blk_alloc_queue_node()函数中将对其 backing_dev_info 结构体成员进行初始化，函数代码如下（/block/blk-core.c）：

```
struct request_queue *blk_alloc_queue_node(gfp_t gfp_mask, int node_id)
{
    ...
    /*初始化 backing_dev_info 成员*/
    q->backing_dev_info.ra_pages =(VM_MAX_READAHEAD * 1024) / PAGE_CACHE_SIZE;
                                        /*内存页大小为 4KB 时，ra_pages 为 32 页*/
    q->backing_dev_info.capabilities = BDI_CAP_CGROUP_WRITEBACK;      /*设备能力*/
    q->backing_dev_info.name = "block";              /*名称*/
    q->node = node_id;

    err = bdi_init(&q->backing_dev_info);
                        /*初始化 backing_dev_info 实例，/mm/backing-dev.c*/
    ...
}
```

bdi_init()函数用于初始化 request_queue 实例中 backing_dev_info 结构体成员，如下（/mm/backing-dev.c）：

```
int bdi_init(struct backing_dev_info *bdi)
{
    bdi->dev = NULL;

    bdi->min_ratio = 0;
    bdi->max_ratio = 100;
    bdi->max_prop_frac = FPROP_FRAC_BASE;
    INIT_LIST_HEAD(&bdi->bdi_list);          /*初始化双链表成员*/
    init_waitqueue_head(&bdi->wb_waitq);     /*初始化等待队列头*/

    return cgwb_bdi_init(bdi);      /*/mm/backing-dev.c*/
}
```

cgwb_bdi_init(bdi)函数用于为 backing_dev_info 实例分配 bdi_writeback_congested 结构体实例，以及初始化内嵌的 **bdi_writeback** 结构体成员等。

内核如果没有选择 CGROUP_WRITEBACK 配置选项，则 cgwb_bdi_init(bdi)函数定义如下：

```
static int cgwb_bdi_init(struct backing_dev_info *bdi)
{
    int err;

    bdi->wb_congested = kzalloc(sizeof(*bdi->wb_congested), GFP_KERNEL);
                                        /*分配 bdi_writeback_congested 结构体实例*/
    if (!bdi->wb_congested)
        return -ENOMEM;

    err = wb_init(&bdi->wb, bdi, 1, GFP_KERNEL);     /*初始化 bdi_writeback，/mm/backing-dev.c*/
    ...
    return 0;
}
```

cgwb_bdi_init()函数用于分配 bdi_writeback_congested 实例，调用 wb_init()函数对内嵌的 **bdi_writeback**

结构体成员进行初始化。

wb_init()函数定义如下（/mm/backing-dev.c）：

```
static int wb_init(struct bdi_writeback *wb, struct backing_dev_info *bdi,int blkcg_id, gfp_t gfp)
{
    int i, err;

    memset(wb, 0, sizeof(*wb));        /*清 0*/

    wb->bdi = bdi;                     /*指向 backing_dev_info 实例*/
    wb->last_old_flush = jiffies;      /*当前时间*/
    INIT_LIST_HEAD(&wb->b_dirty);     /*初始化 inode 双链表*/
    INIT_LIST_HEAD(&wb->b_io);
    INIT_LIST_HEAD(&wb->b_more_io);
    INIT_LIST_HEAD(&wb->b_dirty_time);
    spin_lock_init(&wb->list_lock);

    wb->bw_time_stamp = jiffies;
    wb->balanced_dirty_ratelimit = INIT_BW;          /*脏页平衡参数，初始带宽 100 MB/s*/
    wb->dirty_ratelimit = INIT_BW;
    wb->write_bandwidth = INIT_BW;
    wb->avg_write_bandwidth = INIT_BW;

    spin_lock_init(&wb->work_lock);
    INIT_LIST_HEAD(&wb->work_list);            /*初始化 work_list 双链表*/
    INIT_DELAYED_WORK(&wb->dwork, wb_workfn);      /*初始化延时工作，/fs/fs-writeback.c*/

    wb->congested = wb_congested_get_create(bdi, blkcg_id, gfp);    /*/include/linux/backing-dev.h*/
                /*如果没有选择 CGROUP_WRITEBACK 配置选项，指向 bdi->wb_congested*/
    if (!wb->congested)
        return -ENOMEM;

    err = fprop_local_init_percpu(&wb->completions, gfp);
                        /*为 completions 成员分配内存，fprop_local_percpu 实例（percpu 变量）*/
    ...

    for (i = 0; i < NR_WB_STAT_ITEMS; i++) {            /*为统计量分配内存，percpu 变量*/
        err = percpu_counter_init(&wb->stat[i], 0, gfp);
        ...
    }

    return 0;
    ...
}
```

wb_init()函数重点要关注的是延时工作 dwork 成员的初始化，其执行函数设为 **wb_workfn()**，后面将介绍此函数的实现。

■注册 backing_dev_info

在添加磁盘的 add_disk()函数中将注册 backing_dev_info 实例，注册函数定义如下：

```
int bdi_register_dev(struct backing_dev_info *bdi, dev_t dev)      /*/mm/backing-dev.c*/
/*dev：块设备号*/
{
    return bdi_register(bdi, NULL, "%u:%u", MAJOR(dev), MINOR(dev));
}
```

bdi_register()函数定义如下（/mm/backing-dev.c）：

```
int bdi_register(struct backing_dev_info *bdi, struct device *parent,const char *fmt, ...)
/*parent：NULL*/
{
    va_list args;
    struct device *dev;

    if (bdi->dev)   /* The driver needs to use separate queues per device */
        return 0;

    va_start(args, fmt);
    dev = device_create_vargs(bdi_class, parent, MKDEV(0, 0), bdi, fmt, args);
                                        /*创建并添加 device 实例，属 bdi_class 设备类*/
    va_end(args);
    ...

    bdi->dev = dev;             /*指向 device 实例*/

    bdi_debug_register(bdi, dev_name(dev));
    set_bit(WB_registered, &bdi->wb.state);          /*置位已注册标记位*/

    spin_lock_bh(&bdi_lock);
    list_add_tail_rcu(&bdi->bdi_list, &bdi_list);       /*实例添加到全局 bdi_list 双链表末尾*/
    spin_unlock_bh(&bdi_lock);

    trace_writeback_bdi_register(bdi);
    return 0;
}
```

注册函数 bdi_register()比较简单，主要是为 backing_dev_info 实例创建和添加 device 实例，将实例添加到全局的 **bdi_list** 双链表末尾。

3 设置 inode 脏标记

若要使内核回写缓存页和 inode，前提是要设置缓存页和 inode 的脏标记。

在前面的文件映射函数中，介绍了设置缓存页脏标记的 **set_page_dirty()**函数，在这个函数中将会设置缓存页 page 和 inode 实例的脏标记。

内核提供了单独设置 inode 脏标记的接口函数，例如（/include/linux/fs.h）：

```
static inline void mark_inode_dirty(struct inode *inode)
{
    __mark_inode_dirty(inode, I_DIRTY);     /*inode 元数据和缓存页都有修改/
}

static inline void mark_inode_dirty_sync(struct inode *inode)
{
    __mark_inode_dirty(inode, I_DIRTY_SYNC);   /*只有 inode 元数据被修改*/
}
```

以上两个接口函数内部都是调用__mark_inode_dirty()函数用于设置 inode 脏标记，并将 inode 实例添加到 bdi_writeback 实例中的脏 inode 双链表。set_page_dirty()函数中也是调用的__mark_inode_dirty()函数设置 inode 脏标记。

__mark_inode_dirty()函数定义在/fs/fs-writeback.c 文件内，代码如下：

```
void __mark_inode_dirty(struct inode *inode, int flags)
/*flag：inode 脏的类型，元数据脏或缓存页脏等，对应 inode.i_state 成员值*/
{
    struct super_block *sb = inode->i_sb;
    int dirtytime;

    trace_writeback_mark_inode_dirty(inode, flags);

    /*inode 元数据修改了*/
    if (flags & (I_DIRTY_SYNC | I_DIRTY_DATASYNC | I_DIRTY_TIME)) {
        trace_writeback_dirty_inode_start(inode, flags);
        if (sb->s_op->dirty_inode)
            sb->s_op->dirty_inode(inode, flags);      /*调用标记 inode 元数据脏函数*/
        trace_writeback_dirty_inode(inode, flags);
    }
    if (flags & I_DIRTY_INODE)   /* I_DIRTY_INODE= (I_DIRTY_SYNC | I_DIRTY_DATASYNC)*/
        flags &= ~I_DIRTY_TIME;
    dirtytime = flags & I_DIRTY_TIME;
    smp_mb();

    if (((inode->i_state & flags) == flags) ||(dirtytime && (inode->i_state & I_DIRTY_INODE)))
        return;

    if (unlikely(block_dump))       /*全局变量*/
        block_dump___mark_inode_dirty(inode);         /*输出信息*/

    spin_lock(&inode->i_lock);
    if (dirtytime && (inode->i_state & I_DIRTY_INODE))
        goto out_unlock_inode;
    if ((inode->i_state & flags) != flags) {       /*inode.i_state 成员需要修改*/
        const int was_dirty = inode->i_state & I_DIRTY;   /*是不是已经设置了脏标记*/
                            /*I_DIRTY=(I_DIRTY_SYNC | I_DIRTY_DATASYNC | I_DIRTY_PAGES)*/
        inode_attach_wb(inode, NULL);
```

```
                /*没有选择 CGROUP_WRITEBACK 选项为空操作，/include/linux/writeback.h*/

        if (flags & I_DIRTY_INODE)
            inode->i_state &= ~I_DIRTY_TIME;
        inode->i_state |= flags;
        if (inode->i_state & I_SYNC)        /*inode 正在回写*/
            goto out_unlock_inode;

        if (!S_ISBLK(inode->i_mode)) {     /*不是块设备文件*/
            if (inode_unhashed(inode))          /*inode 从散列表移出*/
                goto out_unlock_inode;
        }
        if (inode->i_state & I_FREEING)
            goto out_unlock_inode;

        /*如果 inode 已经在 b_dirty/b_io/b_more_io 双链表（was_dirty!=0）不需要修改其位置*/
        if (!was_dirty) {       /*inode 原来不脏，现在脏，需要将其插入 b_dirty/b_io/b_more_io 链表*/
            struct bdi_writeback *wb;
            struct list_head *dirty_list;
            bool wakeup_bdi = false;

            wb = locked_inode_to_wb_and_lock_list(inode);
                                                /*获取 bdi_writeback 实例，/fs/fs-writeback.c*/
              /*inode->i_sb->s_bdi->wb（/include/linux/backing-dev.h），!CGROUP_WRITEBACK*/

            ...
            inode->dirtied_when = jiffies;          /*变脏时间*/
            if (dirtytime)
                inode->dirtied_time_when = jiffies;

            if (inode->i_state & (I_DIRTY_INODE | I_DIRTY_PAGES))
                dirty_list = &wb->b_dirty;      /*元数据或页缓存内容修改了，添加到 b_dirty 链表*/
            else                                /*只是修改了时间*/
                dirty_list = &wb->b_dirty_time;     /*添加到 b_dirty_time 链表*/

            wakeup_bdi = inode_wb_list_move_locked(inode, wb,dirty_list);   /*/fs/fs-writeback.c*/
                                            /*将 inode 添加到 bdi_writeback 实例中 inode 双链表*/
            spin_unlock(&wb->list_lock);
            trace_writeback_dirty_inode_enqueue(inode);
            if (bdi_cap_writeback_dirty(wb->bdi) && wakeup_bdi)   /*wakeup_bdi 为 true 时*/
                wb_wakeup_delayed(wb);          /激活工作，/mm/backing-dev.c*/
            return;
        }
    }
out_unlock_inode:
    spin_unlock(&inode->i_lock);
```

}

__mark_inode_dirty()函数首先判明 inode 哪些数据修改了，是缓存页、元数据，还是只修改了时间戳，然后判断 inode 是不是新变脏的（原来不脏），是则需要获取 bdi_writeback 实例，将 inode 插入其中的双链表，最后如果需要则激活 wb->dwork 工作，触发数据回写。

locked_inode_to_wb_and_lock_list(inode)函数中调用 inode_to_wb(inode)函数，获取 bdi_writeback 实例，后者定义在/include/linux/backing-dev.h 头文件，返回 inode->i_sb->s_bdi->**wb**（!CGROUP_WRITEBACK）。

inode_wb_list_move_locked()函数用于将 inode 实例插入指定双链表，若此函数返回 true，随后将调用 wb_wakeup_delayed(wb)函数激活 wb->dwork 延时工作，详见下文。

■插入链表

inode_wb_list_move_locked()函数用于将 inode 实例插入指定双链表，定义如下（/fs/fs-writeback.c）：

```
static bool inode_wb_list_move_locked(struct inode *inode,struct bdi_writeback *wb,struct list_head *head)
{
    assert_spin_locked(&wb->list_lock);

    list_move(&inode->i_wb_list, head);     /*将 inode 移动到 head 双链表头部*/

    if (head != &wb->b_dirty_time)              /*head 不是 wb->b_dirty_time 双链表*/
        return wb_io_lists_populated(wb);     /*是否新置位了状态成员的 WB_has_dirty_io 标记位*/

    wb_io_lists_depopulated(wb);     /*清零 WB_has_dirty_io 标记位，如果需要*/
    return false;
}
```

inode_wb_list_move_locked()函数将 inode 实例插入指定双链表头部，如果 head 不是 wb->b_dirty_time 双链表，则调用 wb_io_lists_populated(wb)函数检测 wb 状态成员的 WB_has_dirty_io 标记位，若已置位函数返回 false，没有置位则置位，函数返回 true。如果 head 表示 wb->b_dirty_time 双链表，则根据需要清零 WB_has_dirty_io 标记位，函数返回 fale。

wb_io_lists_populated()和 wb_io_lists_depopulated()函数定义如下（/fs/fs-writeback.c）：

```
static bool wb_io_lists_populated(struct bdi_writeback *wb)
{
    if (wb_has_dirty_io(wb)) {
                        /*状态值中 WB_has_dirty_io 标记位是否置位，/include/linux/backing-dev.h*/
        return false;        /*已经置位了返回 false*/
    } else {              /*没有置位，置位，返回 true*/
        set_bit(WB_has_dirty_io, &wb->state);
        WARN_ON_ONCE(!wb->avg_write_bandwidth);
        atomic_long_add(wb->avg_write_bandwidth,&wb->bdi->tot_write_bandwidth);
        return true;
    }
}

static void wb_io_lists_depopulated(struct bdi_writeback *wb)
{
```

```
        if (wb_has_dirty_io(wb) && list_empty(&wb->b_dirty) &&
            list_empty(&wb->b_io) && list_empty(&wb->b_more_io)) {/*只有 wb->b_dirty_time 链表非空*/
            clear_bit(WB_has_dirty_io, &wb->state);          /*清标记位*/
            WARN_ON_ONCE(atomic_long_sub_return(wb->avg_write_bandwidth,
                        &wb->bdi->tot_write_bandwidth) < 0);
        }
    }
```

■激活延时工作

如果 inode_wb_list_move_locked()函数返回 true，且块设备允许回写，则随后调用 wb_wakeup_delayed()函数激活 wb->dwork 工作，函数定义如下（/mm/backing-dev.c）：

```
    void wb_wakeup_delayed(struct bdi_writeback *wb)
    {
        unsigned long timeout;

        timeout = msecs_to_jiffies(dirty_writeback_interval * 10);     /*500*10 毫秒*/
        spin_lock_bh(&wb->work_lock);
        if (test_bit(WB_registered, &wb->state))
            queue_delayed_work(bdi_wq, &wb->dwork, timeout); /*wb->dwork 添加到 bdi_wq 工作队列*/
        spin_unlock_bh(&wb->work_lock);
    }
```

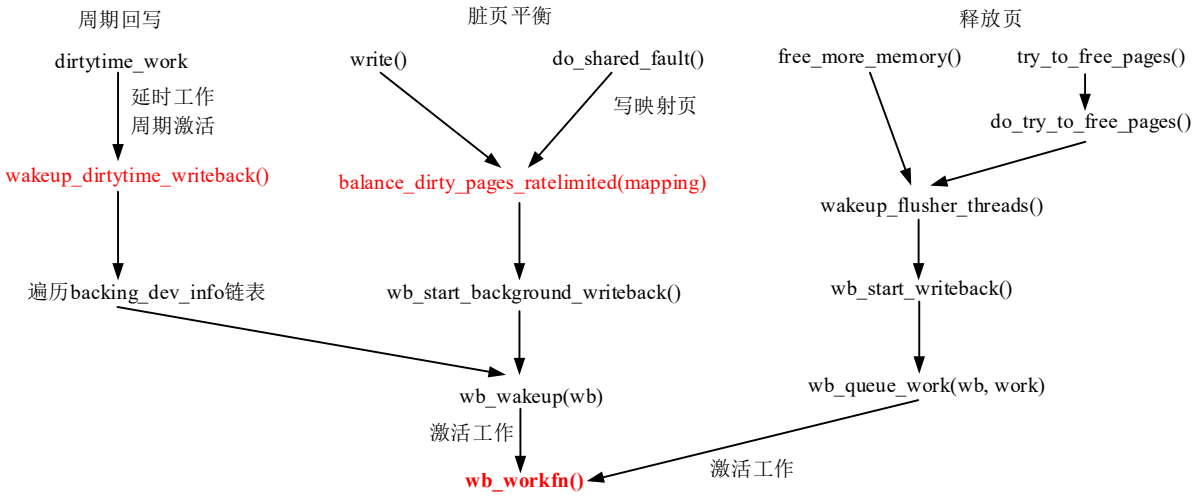
由以上分析可知，如果本次设置 inode 脏标记时，将 bdi_writeback 状态成员 WB_has_dirty_io 标记位从 0 置为 1（向空链表中插入实例），则激活 wb->dwork 工作。

激活延时工作就是对延时工作设置一个延期时间，到期时自动将其添加到 bdi_wq 指向的工作队列。

11.3.3 回写策略

回写策略就是指内核在什么场合、什么时机触发数据回写，可以由内核发起对脏数据的回写，也可由用户进程通过系统调用发起数据回写，其实前面介绍的标记 inode 脏的函数中也可能触发数据回写。本小节先介绍由内核发起的数据回写，后面将介绍用户进程发起的数据回写。

内核发起数据回写的时机有：周期性地触发回写，内核中脏页较多时触发回写（脏页平衡），空闲内存较少（页分配/页回收）时触发回写等。各时机触发数据回写的函数调用关系简列如下图所示：



周期回写：内核在启动初始化阶段定义了延时工作 dirtytime_work，此延时工作会以一定的时间间隔周期性地运行，执行内核脏数据的回写。

脏页平衡：在写文件的 write()等系统调用和写共享文件映射页缺页处理函数等处，将会对文件地址空间页缓存中脏页执行平衡操作，并有可能触发数据回写。

释放页：在伙伴系统分配内存页或分配块缓存头结构实例时，如空闲内存紧张也将触发数据回写，回写一定数量的脏页后，再执行页回收（释放页）。

释放页触发数据回写时，将创建 wb_writeback_work 实例，添加到 bdi_writeback 实例中的 work_list 双链表，然后激活延时工作。

周期回写和脏页平衡时只激活延时工作，而没有创建 wb_writeback_work 实例。

延时工作的执行函数 wb_workfn()将先处理 work_list 双链表中的 wb_writeback_work 实例，依实例执行数据回写，然后再判断是否要执行周期回写和脏页平衡（后台）回写，如果需要则创建 wb_writeback_work 实例，并直接依此实例执行数据回写。

1 周期回写

内核定义了延时工作 **dirtytime_work** 用于周期性地触发数据回写，在内核启动阶段后期将初次调度激活此延时工作（/fs/fs-writeback.c）：

```
static DECLARE_DELAYED_WORK(dirtytime_work, wakeup_dirtytime_writeback);

static int __init start_dirtytime_writeback(void)
{
        schedule_delayed_work(&dirtytime_work, dirtytime_expire_interval * HZ); /*注意时间间隔参数*/
        return 0;
}
__initcall(start_dirtytime_writeback);          /*内核初始时调用*/
```

dirtytime_expire_interval 参数表示周期回写的周期，单位秒，内核将此值初始设为 12 个小时。

dirtytime_work 延时工作执行函数为 wakeup_dirtytime_writeback()，定义如下（/fs/fs-writeback.c）：

```
static void wakeup_dirtytime_writeback(struct work_struct *w)
{
    struct backing_dev_info *bdi;

    rcu_read_lock();
    list_for_each_entry_rcu(bdi, &bdi_list, bdi_list) {      /*遍历内核 backing_dev_info 实例链表*/
        struct bdi_writeback *wb;
        struct wb_iter iter;

        bdi_for_each_wb(wb, bdi, &iter, 0)
          if (!list_empty(&bdi->wb.b_dirty_time))     /*如果 wb.b_dirty_time 双链表不为空*/
              wb_wakeup(&bdi->wb);
                        /*激活 wb->dwork 延时工作，负责执行数据回写*/
    }
    rcu_read_unlock();
    schedule_delayed_work(&dirtytime_work, dirtytime_expire_interval * HZ);
```

```
                                        /*再次调度延时工作，以便下次运行*/
}
```

wakeup_dirtytime_writeback()函数主要是遍历 backing_dev_info 实例双链表，对每个 wb.b_dirty_time 双链表不为空的实例调用 wb_wakeup()函数，激活 bdi_writeback 实例中的延时工作，函数最后还会调度执行 dirtytime_work 延时工作（延时到期后插入工作队列），以便其能周期地被执行。

wb_wakeup()函数用于激活 wb->dwork 延时工作，函数定义如下（/fs/fs-writeback.c）：

```
static void wb_wakeup(struct bdi_writeback *wb)
{
    spin_lock_bh(&wb->work_lock);
    if (test_bit(WB_registered, &wb->state))
        mod_delayed_work(bdi_wq, &wb->dwork, 0);
                                        /*立即激活 wb->dwork 延时工作，/include/linux/workqueue.h*/
    spin_unlock_bh(&wb->work_lock);
}
```

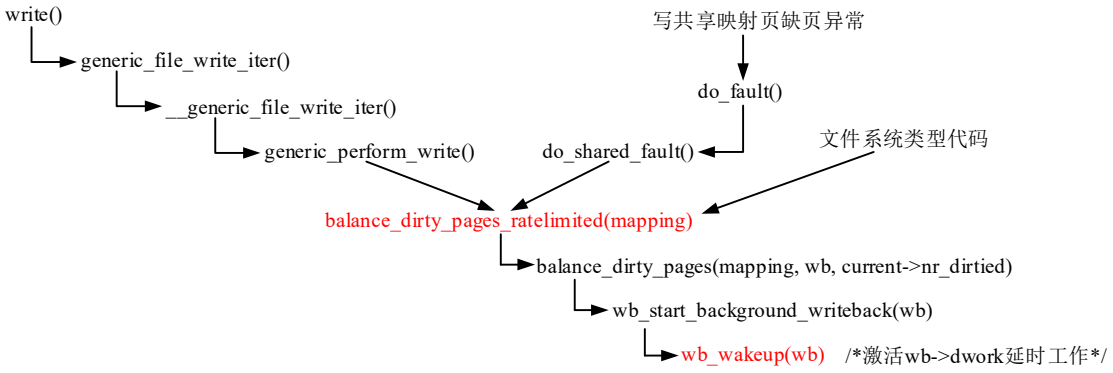
wb_wakeup()函数修改 wb->dwork 工作延期时间为 0，即立即将延时工作插入 bdi_wq 工作队列。

2 脏页平衡

脏页平衡，又称后台回写，是指在写缓存页（设置脏标记）等时机，评估系统脏页的数量，如果脏页数量超过限制值，则触发数据回写。 脏页平衡相关代码位于/mm/page-writeback.c 文件内。

■概述

脏页平衡函数调用关系如图所示：



balance_dirty_pages_ratelimited()函数检测当前进程置脏页数量是否超过限制值，如果是则调用执行脏页平衡函数 balance_dirty_pages()。balance_dirty_pages()函数再检测系统脏页数量是否超过限制值（阀值），如果是则最终调用 wb_wakeup(wb)函数激活 wb->dwork 延时工作，执行数据回写。

通常，如果当前进程置脏页数量没有超过限制值，或系统脏页数量没有超过阀值，都不会执行数据回写。

●数据结构

回写域 wb_domain 结构体定义如下（/include/linux/writeback.h）：

```
struct wb_domain {
    spinlock_t lock;
```

```
        struct fprop_global completions;
        struct timer_list period_timer;     /* timer for aging of completions */
        unsigned long period_time;

        unsigned long dirty_limit_tstamp;           /*时间戳*/
        unsigned long dirty_limit;          /*脏页阀值*/
    };
```

bdi_writeback 实例属于某一回写域，通常内核只有一个全局的回写域，所有 bdi_writeback 实例都属于这个域。

内核在/mm/page-writeback.c 文件内定义了 wb_domain 结构体实例 global_wb_domain：

```
    struct wb_domain global_wb_domain;          /*全局回写域*/
```

dirty_throttle_control 结构体表示脏页平衡的控制参数，由 balance_dirty_pages()及其子函数使用，结构体定义如下（/mm/page-writeback.c）：

```
    struct dirty_throttle_control {
    #ifdef CONFIG_CGROUP_WRITEBACK
        struct wb_domain    *dom;
        struct dirty_throttle_control *gdtc;      /* only set in memcg dtc's */
    #endif
        struct bdi_writeback      *wb;          /*指向 bdi_writeback 实例*/
        struct fprop_local_percpu *wb_completions;
                                    /*percpu 变量，统计计数，指向 bdi_writeback->completions*/

        unsigned long       avail;          /*系统中可以置脏的页总数*/
        unsigned long       dirty;      /* file_dirty + write + nfs，脏缓存页+正在回写页+nfs（已脏页）*/
        unsigned long       thresh;         /* dirty threshold，脏页阀值*/
        unsigned long       bg_thresh;      /* dirty background threshold，后台阀值，用于控制回写的触发*/

        unsigned long       wb_dirty;           /*bdi_writeback 实例的参数值*/
        unsigned long       wb_thresh;
        unsigned long       wb_bg_thresh;

        unsigned long       pos_ratio;
    };
```

●初始化

内核在/mm/page-writeback.c 文件内定义了脏页平衡的相关控制参数，例如：

```
    static DEFINE_PER_CPU(int, bdp_ratelimits);         /*一段时间内 CPU 核置脏页数量*/
    DEFINE_PER_CPU(int, dirty_throttle_leaks) = 0;    /*进程退出时却没有触发脏页平衡，
                                *将其脏页数量累加到此 percpu 变量，累加到任一进程置脏页数量上*/
    static long ratelimit_pages = 32;
                                /*CPU 核置脏页达到此数量，检测是否要回写或制止（不让其置脏页）*/
    int dirty_background_ratio = 10;                /*用于计算 bg_thresh 阀值，10%*/
    int vm_dirty_ratio = 20;        /*用于计算 thresh 阀值，20%*/
```

脏页平衡初始化函数 page_writeback_init()定义如下（/mm/page-writeback.c）：

```
void __init page_writeback_init(void)
{
    BUG_ON(wb_domain_init(&global_wb_domain, GFP_KERNEL));

    writeback_set_ratelimit();          /*设置 global_wb_domain 实例等*/
    register_cpu_notifier(&ratelimit_nb);       /*通知执行函数中调用 writeback_set_ratelimit()函数*/
}
```

writeback_set_ratelimit()函数用于设置全局回写域 global_wb_domain 实例的脏页阀值，以及全局变量 ratelimit_pages，函数定义如下：

```
void writeback_set_ratelimit(void)
{
    struct wb_domain *dom = &global_wb_domain;        /*全局 wb_domain 实例*/
    unsigned long background_thresh;            /*后台阀值*/
    unsigned long dirty_thresh;                 /*脏页阀值*/

    global_dirty_limits(&background_thresh, &dirty_thresh);
                                /*设置 background_thresh（bg_thresh）和 dirty_thresh（thresh）局部变量*/
    dom->dirty_limit = dirty_thresh;            /*脏页阀值赋予全局回写域*/
    ratelimit_pages = dirty_thresh / (num_online_cpus() * 32);        /*CPU 核触发检测的脏页数量*/
    if (ratelimit_pages < 16)
        ratelimit_pages = 16;       /*最小设为 16*/
}
```

global_dirty_limits()函数用于根据系统可置脏的页数量等信息，计算脏页阀值和后台阀值，函数定义如下：

```
void global_dirty_limits(unsigned long *pbackground, unsigned long *pdirty)
{
    struct dirty_throttle_control gdtc = { GDTC_INIT_NO_WB };       /*dirty_throttle_control 实例*/

    gdtc.avail = global_dirtyable_memory();         /*全局可置脏的页数量，可用于页缓存的页数量*/
    domain_dirty_limits(&gdtc);
        /*为 gdtc 计算 thresh（gdtc.avail 的 20%）和 bg_thresh 值（gdtc.avail 的 10%）*/

    *pbackground = gdtc.bg_thresh;      /*向参数返回 bg_thresh 值（后台阀值）*/
    *pdirty = gdtc.thresh;              /*向参数返回 thresh 值（脏页阀值）*/
}
```

■检测脏页限制

balance_dirty_pages_ratelimited()函数用于检测脏页数量是否超过限制值，如果是则执行脏页平衡，否则不执行，函数定义在/mm/page-writeback.c 文件内，代码如下：

```
void balance_dirty_pages_ratelimited(struct address_space *mapping)
/*mapping：指向文件地址空间*/
{
```

```
    struct inode *inode = mapping->host;
    struct backing_dev_info *bdi = inode_to_bdi(inode);
                                    /*由 inode 指向 super_block 实例获取 backing_dev_info 实例*/
    struct bdi_writeback *wb = NULL;
    int ratelimit;
    int *p;

    if (!bdi_cap_account_dirty(bdi))
        return;

    if (inode_cgwb_enabled(inode))          /*没有选择 CGROUP_WRITEBACK，返回 false*/
        wb = wb_get_create_current(bdi, GFP_KERNEL);
    if (!wb)
        wb = &bdi->wb;       /*backing_dev_info 实例中 bdi_writeback 结构体成员*/

    ratelimit = current->nr_dirtied_pause; /*限制进程置脏页的阀值，copy_process()函数中赋初值*/
    if (wb->dirty_exceeded)
        ratelimit = min(ratelimit, 32 >> (PAGE_SHIFT - 10));

    preempt_disable();      /*禁止内核抢占*/
    p =  this_cpu_ptr(&bdp_ratelimits);          /*当前 CPU 核置脏页数量*/
    if (unlikely(current->nr_dirtied >= ratelimit))      /*当前进程置脏页数量超过限制值*/
        *p = 0;                                  /*当前 CPU 核置脏页数量清零*/
    else if (unlikely(*p >= ratelimit_pages)) {   /*当前 CPU 核置脏页数量超过限制值*/
        *p = 0;                                  /*当前 CPU 核置脏页数量清零*/
        ratelimit = 0;
    }
    p = this_cpu_ptr(&dirty_throttle_leaks);            /*已退出进程遗留的脏页数量*/
    if (*p > 0 && current->nr_dirtied < ratelimit) {      /*累加到当前进程脏页数量中*/
        unsigned long nr_pages_dirtied;
        nr_pages_dirtied = min(*p, ratelimit - current->nr_dirtied);
        *p -= nr_pages_dirtied;
        current->nr_dirtied += nr_pages_dirtied;        /*当前进程脏页数量不能超过限制值*/
    }
    preempt_enable();

    if (unlikely(current->nr_dirtied >= ratelimit))      /*当前进程脏页数量超过限制值*/
        balance_dirty_pages(mapping, wb, current->nr_dirtied);   /*/mm/page-writeback.c*/
                                                                 /*执行脏页平衡*/
    wb_put(wb);
}
```

由以上函数可知，触发脏页平衡的条件是：当前进程置脏页数量大于或等于限制值 **ratelimit**。

进程置脏页数量限制值 ratelimit 来自于 **current->nr_dirtied_pause** 成员值，在 copy_process()函数创建进程时赋初值（128 >> (PAGE_SHIFT - 10)），进程置脏页数量保存在 **current->nr_dirtied** 成员中，初始值为 0。在设置缓存页脏标记的 set_page_dirty()函数调用的 account_page_dirtied()函数中，将更新当前进程 current->nr_dirtied 成员值，以及其它脏页统计值，如 CPU 核置脏页数量。

dirty_throttle_leaks 变量中保存了已经退出，但没有触发脏页平衡的进程置脏页的数量，它将累加到当前进程的置脏页数量中。

balance_dirty_pages()函数用于执行脏页平衡，详见下文。

●执行脏页平衡

balance_dirty_pages()函数还将检测系统脏页数量是否超过阀值，如果是则触发数据回写，否则不触发，函数定义如下（/mm/page-writeback.c）：

```
static void balance_dirty_pages(struct address_space *mapping,struct bdi_writeback *wb,  \
                                                        unsigned long pages_dirtied)
/*pages_dirtied：进程置脏页数量*/
{
    struct dirty_throttle_control gdtc_stor = { GDTC_INIT(wb) };     /*dirty_throttle_control 实例*/
    struct dirty_throttle_control mdtc_stor = { MDTC_INIT(wb, &gdtc_stor) };
    struct dirty_throttle_control * const gdtc = &gdtc_stor;
    struct dirty_throttle_control * const mdtc = mdtc_valid(&mdtc_stor) ?&mdtc_stor : NULL;
                                    /*没有选择 CGROUP_WRITEBACK 选择项，mdtc 为 NULL*/
    struct dirty_throttle_control *sdtc;
    unsigned long nr_reclaimable;       /* = file_dirty + unstable_nfs */
    long period;
    long pause;
    long max_pause;
    long min_pause;
    int nr_dirtied_pause;
    bool dirty_exceeded = false;
    unsigned long task_ratelimit;
    unsigned long dirty_ratelimit;
    struct backing_dev_info *bdi = wb->bdi;          /*backing_dev_info 实例*/
    bool strictlimit = bdi->capabilities & BDI_CAP_STRICTLIMIT;
    unsigned long start_time = jiffies;         /*当前时间*/

    for (;;) {            /*for 循环开始*/
        unsigned long now = jiffies;
        unsigned long dirty, thresh, bg_thresh;         /*全局的参数*/
        unsigned long m_dirty, m_thresh, m_bg_thresh;

        nr_reclaimable = global_page_state(NR_FILE_DIRTY) +
                                global_page_state(NR_UNSTABLE_NFS);   /*总的可回收页数量*/
        gdtc->avail = global_dirtyable_memory();              /*系统可置脏页数量*/
        gdtc->dirty = nr_reclaimable + global_page_state(NR_WRITEBACK); /*当前系统脏页数量*/
        domain_dirty_limits(gdtc);              /*为 gdtc 计算 thresh 和 bg_thresh 值*/

        if (unlikely(strictlimit)) {            /*是否限制脏页数量，默认不限制*/
            wb_dirty_limits(gdtc);

            dirty = gdtc->wb_dirty;
            thresh = gdtc->wb_thresh;
```

```
        bg_thresh = gdtc->wb_bg_thresh;
} else {
        dirty = gdtc->dirty;                    /*当前系统脏页数量*/
        thresh = gdtc->thresh;                  /*阀值*/
        bg_thresh = gdtc->bg_thresh;            /*后台阀值*/
}

if (mdtc) {
        ...
}

/*如果 dirty <=(thresh + bg_thresh) / 2，不触发回写，跳出 for 循环*/
if (dirty <= dirty_freerun_ceiling(thresh, bg_thresh) &&
        (!mdtc ||m_dirty <= dirty_freerun_ceiling(m_thresh, m_bg_thresh))) {
        unsigned long intv = dirty_poll_interval(dirty, thresh);
                                /*进程置脏 intv 数量页后，要检测是否执行脏页平衡*/
        unsigned long m_intv = ULONG_MAX;

        current->dirty_paused_when = now;
        current->nr_dirtied = 0;
        if (mdtc)
                m_intv = dirty_poll_interval(m_dirty, m_thresh);
        current->nr_dirtied_pause = min(intv, m_intv);          /*修改进程 nr_dirtied_pause 值*/
        break;          /*跳出 for 循环*/
}

/*dirty>(thresh + bg_thresh) / 2，触发数据回写*/
if (unlikely(!writeback_in_progress(wb)))       /*当前 wb 不在执行回写*/
        wb_start_background_writeback(wb);      /*激活延时工作*/
                                        /*调用 wb_wakeup(wb)函数，/fs/fs-writeback.c*/

if (!strictlimit)
        wb_dirty_limits(gdtc);

dirty_exceeded = (gdtc->wb_dirty > gdtc->wb_thresh) &&
                                ((gdtc->dirty > gdtc->thresh) || strictlimit);  /*脏页数量是否超限*/

wb_position_ratio(gdtc);        /*计算 gdtc->pos_ratio 值，/mm/page-writeback.c*/
sdtc = gdtc;            /*全局 dirty_throttle_control 实例*/

if (mdtc) {
        ...
}

if (dirty_exceeded && !wb->dirty_exceeded)
        wb->dirty_exceeded = 1;
```

```
        if (time_is_before_jiffies(wb->bw_time_stamp +BANDWIDTH_INTERVAL)) {
            spin_lock(&wb->list_lock);
            __wb_update_bandwidth(gdtc, mdtc, start_time, true);
                                                /*更新 bdi_writeback 实例中带宽值、时间戳*/
            spin_unlock(&wb->list_lock);
        }

        /*计算参数*/
        dirty_ratelimit = wb->dirty_ratelimit;
        task_ratelimit = ((u64)dirty_ratelimit * sdtc->pos_ratio) >>RATELIMIT_CALC_SHIFT;
        max_pause = wb_max_pause(wb, sdtc->wb_dirty);
        min_pause = wb_min_pause(wb, max_pause,task_ratelimit, dirty_ratelimit,&nr_dirtied_pause);
                                                        /*最大、最小睡眠时长*/
        if (unlikely(task_ratelimit == 0)) {
            period = max_pause;
            pause = max_pause;
            goto pause;
        }
        period = HZ * pages_dirtied / task_ratelimit;          /*计算得进程睡眠时长*/
        pause = period;
        if (current->dirty_paused_when)
            pause -= now - current->dirty_paused_when;
        if (pause < min_pause) {          /*小于最小睡眠时长，不睡眠*/
            ...
            if (pause < -HZ) {
                current->dirty_paused_when = now;
                current->nr_dirtied = 0;
            } else if (period) {
                current->dirty_paused_when += period;
                current->nr_dirtied = 0;
            } else if (current->nr_dirtied_pause <= pages_dirtied)
                current->nr_dirtied_pause += pages_dirtied;
            break;          /*跳出循环，不睡眠*/
        }
        /*当前进程要睡眠一段时间*/
        if (unlikely(pause > max_pause)) {
            /* for occasional dropped task_ratelimit */
            now += min(pause - max_pause, max_pause);
            pause = max_pause;
        }

pause:      /*当前进程需要睡眠（暂停）一段时间*/
        ...
        __set_current_state(TASK_KILLABLE);        /*修改进程状态*/
        io_schedule_timeout(pause);        /*进程调度，睡眠 pause 时间*/
```

```
            /*唤醒后执行以下代码*/
            current->dirty_paused_when = now + pause;
            current->nr_dirtied = 0;
            current->nr_dirtied_pause = nr_dirtied_pause;

            if (task_ratelimit)
                break;

            if (sdtc->wb_dirty <= wb_stat_error(wb))
                break;

            if (fatal_signal_pending(current))
                break;
        }  /*for()循环结束*/

        if (!dirty_exceeded && wb->dirty_exceeded)
            wb->dirty_exceeded = 0;

        if (writeback_in_progress(wb))
            return;

        if (laptop_mode)          /*laptop_mode 全局变量非 0（系统控制参数，初始为 0），函数返回*/
            return;

        if (nr_reclaimable > gdtc->bg_thresh)   /*laptop_mode 为 0，可能要触发数据回写，以保持脏页少*/
            wb_start_background_writeback(wb);
    }
```

简单地说，balance_dirty_pages()函数就是判断当前系统脏页数量是否超过阀值，如果超过则调用函数wb_start_background_writeback()激活 wb->dwork 延时工作，然后当前进程睡眠一段时间，再判断系统脏页情况，确定是否要继续执行数据回写，直至脏页数量低于阀值，函数返回（**函数中还有算法没研究！！**）。

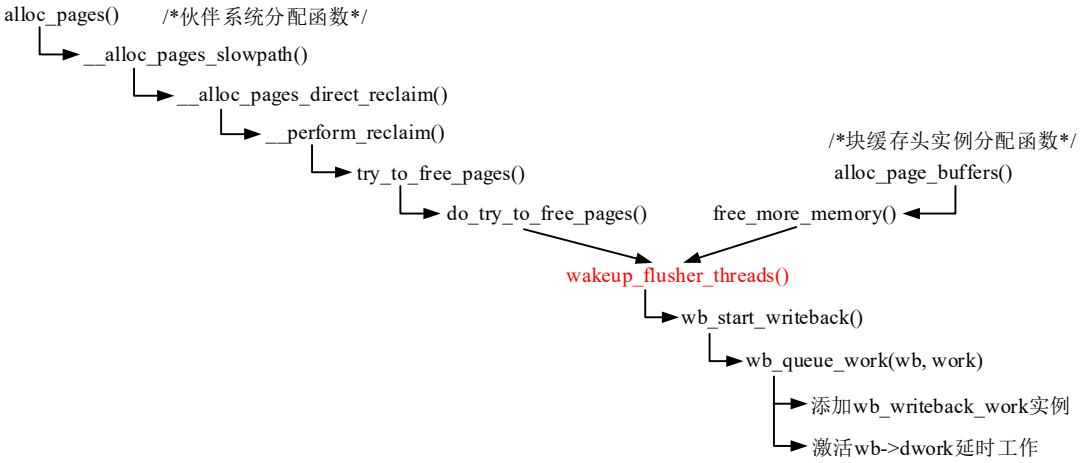
wb_start_background_writeback()函数用于激活 wb->dwork 延时工作，定义如下（/fs/fs-writeback.c）：

```
void wb_start_background_writeback(struct bdi_writeback *wb)
{
    trace_writeback_wake_background(wb->bdi);
    wb_wakeup(wb);            /*激活 wb->dwork 延时工作，见上文*/
}
```

3 释放页

在伙伴系统分配函数中，如果直接从当前空闲页链表中分配不成功，分配函数将会进入低速分配路径，低速分配路径中将尝试对脏页进行回写以释放出文件映射缓存页。

在分配块缓存头实例等函数中，如果内存不足也将激活对脏文件缓存页的回写操作，函数调用关系如下图所示：



以上函数中最终调用 wakeup_flusher_threads(long nr_pages, enum wb_reason reason)函数触发数据回写，参数 nr_pages 表示需回写页数量，为 0 表示回写所有脏数据，reason 参数表示回写原因，由 wb_reason 枚举类型表示。

wakeup_flusher_threads()函数定义如下（/fs/fs-writeback.c）：

```
void wakeup_flusher_threads(long nr_pages, enum wb_reason reason)
{
    struct backing_dev_info *bdi;

    if (!nr_pages)        /*参数 nr_pages 为 0*/
        nr_pages = get_nr_dirty_pages();        /*内核中所有脏页数量，/fs/fs-writeback.c*/

    rcu_read_lock();
    list_for_each_entry_rcu(bdi, &bdi_list, bdi_list) {        /*遍历 backing_dev_info 实例*/
        struct bdi_writeback *wb;
        struct wb_iter iter;

        if (!bdi_has_dirty_io(bdi))
            continue;

        bdi_for_each_wb(wb, bdi, &iter, 0)
            wb_start_writeback(wb, wb_split_bdi_pages(wb, nr_pages),false, reason);
                                                                    /*/fs/fs-writeback.c*/
    }
    rcu_read_unlock();
}
```

wakeup_flusher_threads()函数遍历所有 backing_dev_info 实例，调用 **wb_start_writeback()**函数，构建 wb_writeback_work 实例并添加到实例内嵌 bdi_writeback 实例的 work_list 双链表中。

wb_start_writeback()函数定义在/fs/fs-writeback.c 文件内，代码如下：

```
void wb_start_writeback(struct bdi_writeback *wb, long nr_pages,bool range_cyclic, \
                                                        enum wb_reason reason)
{
    struct wb_writeback_work *work;
```

```
    if (!wb_has_dirty_io(wb))
        return;

    work = kzalloc(sizeof(*work), GFP_ATOMIC);        /*分配 wb_writeback_work 实例*/
    if (!work) {            /*分配失败*/
        trace_writeback_nowork(wb->bdi);
        wb_wakeup(wb);              /*激活 wb->dwork 延时工作*/
        return;
    }

    /*初始化 wb_writeback_work 实例*/
    work->sync_mode  = WB_SYNC_NONE;
    work->nr_pages     = nr_pages;
    work->range_cyclic = range_cyclic;
    work->reason  = reason;
    work->auto_free     = 1;

    wb_queue_work(wb, work);     /*wb_writeback_work 实例添加至 bdi_writeback 实例中双链表*/
}
```

wb_queue_work(wb, work)主要将 wb_writeback_work 实例添加到 bdi_writeback 实例中 work_list 双链表末尾，函数代码如下：

```
static void wb_queue_work(struct bdi_writeback *wb,struct wb_writeback_work *work)
{
    trace_writeback_queue(wb->bdi, work);

    spin_lock_bh(&wb->work_lock);
    if (!test_bit(WB_registered, &wb->state)) {        /*确定 bdi_writeback 已经注册*/
        if (work->single_wait)
            work->single_done = 1;
        goto out_unlock;
    }

    if (work->done)
        atomic_inc(&work->done->cnt);
    list_add_tail(&work->list, &wb->work_list);      /*插入 bdi_writeback 实例 work_list 链表末尾*/
    mod_delayed_work(bdi_wq, &wb->dwork, 0); /*激活 wb->dwork 延迟工作，立即插入工作队列*/
out_unlock:
    spin_unlock_bh(&wb->work_lock);
}
```

11.3.4 执行回写

前面介绍的内核触发的回写操作，最终都是激活 backing_dev_info 实例中 bdi_writeback 结构体成员中的 dwork 延时工作中来执行数据回写。所谓激活延时工作，就是给延时工作设置一个到期时间，到期后自

动将延时工作添加到 **bdi_wq** 工作队列，在内核线程中调用延时工作的执行函数实施数据回写。

在 backing_dev_info 实例的初始化函数中，将最终调用 wb_init()函数初始化内嵌的 bdi_writeback 实例，其中 dwork 延时工作执行函数设为 **wb_workfn()**。

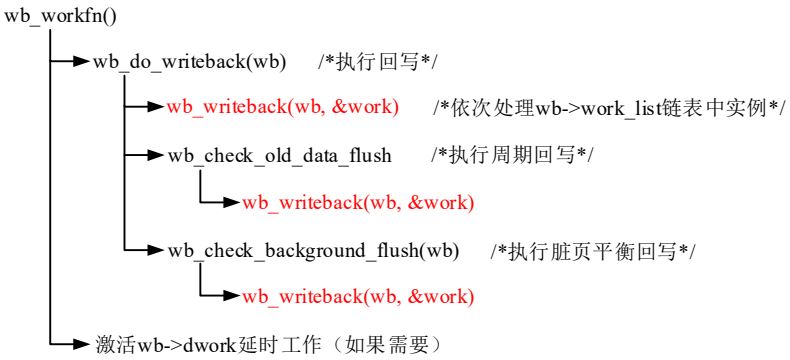
wb_workfn()函数是真正用于执行数据回写的函数，本小节介绍此函数的实现。

1 延时工作执行函数

延时工作执行函数 wb_workfn()定义在/fs/fs-writeback.c 文件内，代码如下：

```
void wb_workfn(struct work_struct *work)
 /*work：wb->dwork.work 成员*/
{
    struct bdi_writeback *wb = container_of(to_delayed_work(work),struct bdi_writeback, dwork);
    long pages_written;

    set_worker_desc("flush-%s", dev_name(wb->bdi->dev));
    current->flags |= PF_SWAPWRITE;        /*当前线程正在回写数据*/

    if (likely(!current_is_workqueue_rescuer() ||!test_bit(WB_registered, &wb->state))) {
        /*当前线程不是 workqueue_rescuer，/kernel/workqueue.c，以下是正常执行路径*/
        do {
            pages_written = wb_do_writeback(wb);
                                                /*执行回写操作，/fs/fs-writeback.c*/
            trace_writeback_pages_written(pages_written);
        } while (!list_empty(&wb->work_list));        /*若 wb->work_list 不为空，继续循环*/
    } else {
        /*当前线程是 workqueue_rescuer，不是正常的执行路径*/
        pages_written = writeback_inodes_wb(wb, 1024,WB_REASON_FORKER_THREAD);
                                                /*回写缓存页数量为 1024，/fs/fs-writeback.c*/
        trace_writeback_pages_written(pages_written);
    }

    if (!list_empty(&wb->work_list))          /*wb->work_list 双链表不为空*/
        mod_delayed_work(bdi_wq, &wb->dwork, 0);          /*立即激活延时工作*/
    else if (wb_has_dirty_io(wb) && dirty_writeback_interval)     /*设置了 WB_has_dirty_io 状态标记*/
        wb_wakeup_delayed(wb);          /*激活延时 wb->dwork 工作，/mm/backing-dev.c*/

    current->flags &= ~PF_SWAPWRITE;
}
```

wb_workfn()函数正常执行流程如下图所示：



wb_workfn()函数调用 wb_do_writeback()函数执行回写操作，最后根据情况再次激活延时工作。

wb_do_writeback()函数执行流程如下：扫描 wb->work_list 双链表中的 wb_writeback_work 实例，对每个实例调用一次 wb_writeback(wb, work)函数执行一次回写操作，处理完双链表中 wb_writeback_work 实例后，再调用 wb_check_old_data_flush(wb)函数执行周期回写，调用 wb_check_background_flush(wb)函数执行脏页平衡回写，这两个函数内根据需要创建 wb_writeback_work 实例，并调用 wb_writeback(wb, work)函数执行一次回写操作。

释放页触发数据回写时，将会创建 wb_writeback_work 添加到 wb->work_list 双链表。wb_do_writeback()函数先处理 wb->work_list 双链表中实例，再执行周期回写和脏页平衡回写，因此释放页触发的数据回写具有更高的优先级。

wb_do_writeback()函数定义如下（/fs/fs-writeback.c）：

```
static long wb_do_writeback(struct bdi_writeback *wb)
{
    struct wb_writeback_work *work;
    long wrote = 0;

    set_bit(WB_writeback_running, &wb->state);        /*设置状态运行标记位*/
    while ((work = get_next_work_item(wb)) != NULL) {        /*遍历 wb->work_list 双链表中实例*/
        struct wb_completion *done = work->done;
        bool need_wake_up = false;

        trace_writeback_exec(wb->bdi, work);

        wrote += wb_writeback(wb, work);        /*执行回写，函数定义见下文*/

        if (work->single_wait) {
            WARN_ON_ONCE(work->auto_free);
            /* paired w/ rmb in wb_wait_for_single_work() */
            smp_wmb();
            work->single_done = 1;
            need_wake_up = true;
        } else if (work->auto_free) {
            kfree(work);
        }

        if (done && atomic_dec_and_test(&done->cnt))
            need_wake_up = true;
```

```
        if (need_wake_up)
            wake_up_all(&wb->bdi->wb_waitq);      /*唤醒在 backing_dev_info 上睡眠等待的进程*/
    }

    /*wb->work_list 双链表中实例处理完了*/
    wrote += wb_check_old_data_flush(wb);              /*执行周期回写*/
    wrote += wb_check_background_flush(wb);            /*执行脏页平衡回写*/
    clear_bit(WB_writeback_running, &wb->state);       /*清除状态运行标记位*/

    return wrote;          /*返回回写的页数据和 inode 数量之和*/
}
```

wb_do_writeback()函数中调用的 wb_writeback()函数用于根据 wb_writeback_work 实例执行一次数据回写操作，函数定义后面再专门介绍。

前面介绍的周期回写和脏页平衡回写，只是激活了 wb->dwork 延迟工作，没有创建 wb_writeback_work 实例，在这里的 **wb_check_old_data_flush(wb)**和 **wb_check_background_flush(wb)**函数将创建实例，并调用 wb_writeback()函数执行回写，下面介绍这两个函数的实现。

■执行周期回写

wb_check_old_data_flush()函数用于执行周期回写，函数定义如下（/fs/fs-writeback.c）：

```
static long wb_check_old_data_flush(struct bdi_writeback *wb)
{
    unsigned long expired;
    long nr_pages;

    if (!dirty_writeback_interval)        /*dirty_writeback_interval 为 0，则关闭周期回写*/
        return 0;

    expired = wb->last_old_flush +msecs_to_jiffies(dirty_writeback_interval * 10);
    if (time_before(jiffies, expired))          /*没到时间间隔，不回写*/
        return 0;

    /*需要执行回写*/
    wb->last_old_flush = jiffies;             /*更新时间值*/
    nr_pages = get_nr_dirty_pages();      /*系统脏页数量*/

    if (nr_pages) {        /*脏页数量非 0*/
        struct wb_writeback_work work = {           /*wb_writeback_work 实例*/
            .nr_pages = nr_pages,                   /*脏页数量*/
            .sync_mode    = WB_SYNC_NONE,       /*不等待，异步执行回写*/
            .for_kupdate   = 1,
            .range_cyclic  = 1,
            .reason          = WB_REASON_PERIODIC,        /*周期回写*/
        };
```

```
        return wb_writeback(wb, &work);     /*执行回写，见下文*/
    }

    return 0;
}
```

■执行脏页平衡回写

wb_check_background_flush()函数用于执行脏页平衡回写，函数定义如下（/fs/fs-writeback.c）：

```
static long wb_check_background_flush(struct bdi_writeback *wb)
{
    if (wb_over_bg_thresh(wb)) {            /*脏页是否超过阀值，/mm/page-writeback.c*/

        struct wb_writeback_work work = {
            .nr_pages = LONG_MAX,
            .sync_mode    = WB_SYNC_NONE,          /*不等待，异步回写*/
            .for_background     = 1,
            .range_cyclic  = 1,
            .reason          = WB_REASON_BACKGROUND,          /*后台回写*/
        };

        return wb_writeback(wb, &work);          /*执行回写，见下文*/
    }

    return 0;
}
```

2 回写函数

下面看一下依 wb_writeback_work 实例执行数据回写的 **wb_writeback()**函数定义（/fs/fs-writeback.c）：

```
static long wb_writeback(struct bdi_writeback *wb,struct wb_writeback_work *work)
{
    unsigned long wb_start = jiffies;        /*当前时间*/
    long nr_pages = work->nr_pages;       /*需要回写的脏页数量*/
    unsigned long oldest_jif;
    struct inode *inode;
    long progress;

    oldest_jif = jiffies;
    work->older_than_this = &oldest_jif;

    spin_lock(&wb->list_lock);
    for (;;) {          /*for 循环*/
        if (work->nr_pages <= 0)
            break;
```

```
        /*如果 wb->work_list 双链表非空，则停止本次周期回写或脏页平衡回写*/
        if ((work->for_background || work->for_kupdate) &&!list_empty(&wb->work_list))
            break;

        /*系统脏页数量没有超过阀值，停止脏页平衡（后台）回写*/
        if (work->for_background && !wb_over_bg_thresh(wb))    /*/mm/page-writeback.c*/
            break;

        if (work->for_kupdate) {        /*周期回写*/
            oldest_jif = jiffies -msecs_to_jiffies(dirty_expire_interval * 10);
        } else if (work->for_background)        /*脏页平衡回写*/
            oldest_jif = jiffies;

        trace_writeback_start(wb->bdi, work);
        if (list_empty(&wb->b_io))          /*wb->b_io 双链表为空*/
            queue_io(wb, work);             /*脏 inode 移入 wb->b_io 双链表，/fs/fs-writeback.c*/
        if (work->sb)           /*关联了超级块，只回写此超级块下的脏 inode*/
            progress = writeback_sb_inodes(work->sb, wb, work);        /*/fs/fs-writeback.c*/
        else
            progress = __writeback_inodes_wb(wb, work);            /*回写脏 inode*/
        trace_writeback_written(wb->bdi, work);

        wb_update_bandwidth(wb, wb_start);        /*更新带宽*/

        if (progress)        /*progress 非 0（执行了实际的回写），继续循环*/
            continue;

        /*没有脏 inode 了*/
        if (list_empty(&wb->b_more_io))         /*wb->b_more_io 双链表为空，跳出循环*/
            break;

        /*wb->b_more_io 双链表非空，等待链表中 inode 回写完成*/
        if (!list_empty(&wb->b_more_io))    {
            trace_writeback_wait(wb->bdi, work);
            inode = wb_inode(wb->b_more_io.prev);   /*b_more_io 双链表中最后一个 inode 实例*/
            spin_lock(&inode->i_lock);
            spin_unlock(&wb->list_lock);
            /* This function drops i_lock... */
            inode_sleep_on_writeback(inode);
                                    /*当前进程睡眠等待 inode 实例 I_SYNC 状态标记位清零*/
            spin_lock(&wb->list_lock);
        }
    }       /*for 循环结束*/
    spin_unlock(&wb->list_lock);

    return nr_pages - work->nr_pages;       /*返回回写的脏页数量*/
```

}

wb_writeback()函数内是一个 for 循环，循环体中首先调用 queue_io()函数将脏 inode 移入 b_io 双链表；然后调用 writeback_sb_inodes()或**__writeback_inodes_wb()**函数回写 b_io 双链表中的脏 inode，前者只回写指定超级块（文件系统）下的脏 inode，后者回写所有脏 inode；最后如果 b_more_io 双链表为空，则跳出循环，函数返回，否则当前进程睡眠等待 b_more_io 双链表中所有 inode 实例 I_SYNC 状态标记位清零，清零后再返回。

下面介绍 queue_io()函数和__writeback_inodes_wb()函数的定义，writeback_sb_inodes()函数请读者自行阅读源代码。

■移入 b_io 双链表

queue_io()函数用于将脏 inode 移入 wb->b_io 双链表，函数定义如下（/fs/fs-writeback.c）：

```
static void queue_io(struct bdi_writeback *wb, struct wb_writeback_work *work)
{
    int moved;

    assert_spin_locked(&wb->list_lock);
    list_splice_init(&wb->b_more_io, &wb->b_io);
                                        /*wb->b_more_io 双链表成员移入 wb->b_io 双链表*/
    moved = move_expired_inodes(&wb->b_dirty, &wb->b_io, 0, work);
                                        /*wb->b_dirty 双链表成员移入 wb->b_io 双链表*/
    moved += move_expired_inodes(&wb->b_dirty_time, &wb->b_io,EXPIRE_DIRTY_ATIME, work);
                                /*wb->b_dirty_time 双链表成员移入 wb->b_io 双链表（超期 inode）*/
    if (moved)
        wb_io_lists_populated(wb);      /*设置状态成员 WB_has_dirty_io 标记位等，见下文*/
    trace_writeback_queue_io(wb, work, moved);
}
```

queue_io()函数将 wb->b_more_io、wb->b_dirty 和 wb->b_dirty_time 双链表中的 inode 都移入 wb->b_io 双链表，表示要执行回写操作的 inode，然后设置 wb 实例状态成员 WB_has_dirty_io 标记位等。

wb_io_lists_populated(wb)函数定义如下（/fs/fs-writeback.c）：

```
static bool wb_io_lists_populated(struct bdi_writeback *wb)
{
    if (wb_has_dirty_io(wb)) {      /*是否设置状态成员 WB_has_dirty_io 标记位*/
        return false;
    } else {
        set_bit(WB_has_dirty_io, &wb->state);       /*没有设置，置位标记位*/
        WARN_ON_ONCE(!wb->avg_write_bandwidth);
        atomic_long_add(wb->avg_write_bandwidth,&wb->bdi->tot_write_bandwidth);
                                                                /*增加带宽计数*/
        return true;
    }
}
```

■回写脏 inode

__writeback_inodes_wb()函数用于回写 wb->b_io 双链表中所有脏 inode，定义如下（/fs/fs-writeback.c）：

```
static long __writeback_inodes_wb(struct bdi_writeback *wb,struct wb_writeback_work *work)
/*work：指向 wb_writeback_work 实例，用于控制回写操作*/
{
    unsigned long start_time = jiffies;
    long wrote = 0;

    while (!list_empty(&wb->b_io)) {          /*遍历 wb->b_io 双链表中脏 inode*/
        struct inode *inode = wb_inode(wb->b_io.prev);    /*从后往前取 inode*/
        struct super_block *sb = inode->i_sb;             /*超级块*/

        if (!trylock_super(sb)) {          /*锁定超级块失败，/fs/super.c*/
            redirty_tail(inode, wb);
                                /*inode 重新插入 wb->b_dirty 链表，修改时间戳，/fs/fs-writeback.c*/
            continue;
        }
        wrote += writeback_sb_inodes(sb, wb, work);
                                /*回写指定超级块下的脏 inode，/fs/fs-writeback.c*/
        up_read(&sb->s_umount);

        if (wrote) {          /*实际回写的脏页和 inode 数量*/
            if (time_is_before_jiffies(start_time + HZ / 10UL))
                break;
            if (work->nr_pages <= 0)          /*回写够数量的缓存页，跳出循环*/
                break;
        }
    }        /*while 循环结束*/
    return wrote;        /*返回实际回写的脏页和 inode 数量*/
}
```

__writeback_inodes_wb()函数从 wb->b_io 双链表从后往前扫描 inode 实例，调用 writeback_sb_inodes()函数回写与 inode 同超级块（文件系统）下的脏 inode，函数定义如下：

```
static long writeback_sb_inodes(struct super_block *sb,
                                struct bdi_writeback *wb,struct wb_writeback_work *work)
{
    struct writeback_control wbc = {          /*由 wb_writeback_work 设置 writeback_control*/
        .sync_mode      = work->sync_mode,
        .tagged_writepages = work->tagged_writepages,
        .for_kupdate    = work->for_kupdate,
        .for_background = work->for_background,
        .for_sync       = work->for_sync,
        .range_cyclic   = work->range_cyclic,
        .range_start    = 0,
        .range_end      = LLONG_MAX,
    };
    unsigned long start_time = jiffies;
    long write_chunk;
```

```
long wrote = 0;    /*累计回写页数和 inode 数量*/

while (!list_empty(&wb->b_io)) {            /*wb->b_io 双链表不为空*/
    struct inode *inode = wb_inode(wb->b_io.prev);          /*双链表最后 inode 实例*/

    if (inode->i_sb != sb) {          /*非指定超级块下的 inode*/
        if (work->sb) {               /*如果 work 关联的超级块，重新插入 wb->b_io 双链表*/
            redirty_tail(inode, wb);
            continue;
        }
        break;        /*如果 work 没有关联超级块，直接跳出循环*/
    }

    spin_lock(&inode->i_lock);
    if (inode->i_state & (I_NEW | I_FREEING | I_WILL_FREE)) {
        spin_unlock(&inode->i_lock);
        redirty_tail(inode, wb);          /*inode 设置了以上标记位，重新插入双链表*/
        continue;
    }
    /*inode 正在同步，有本次回写不等待（异步）*/
    if ((inode->i_state & I_SYNC) && wbc.sync_mode != WB_SYNC_ALL) {
        spin_unlock(&inode->i_lock);
        requeue_io(inode, wb);          /*inode 实例移入 wb->b_more_io 双链表*/
        trace_writeback_sb_inodes_requeue(inode);
        continue;
    }
    spin_unlock(&wb->list_lock);

    if (inode->i_state & I_SYNC) {        /*正在同步，且 wbc.sync_mode== WB_SYNC_ALL*/
        /* Wait for I_SYNC. This function drops i_lock... */
        inode_sleep_on_writeback(inode);
                                /*当前进程睡眠等待 inode 实例 I_SYNC 状态标记位清零*/
        /* Inode may be gone, start again */
        spin_lock(&wb->list_lock);
        continue;
    }

    inode->i_state |= I_SYNC;          /*设置 I_SYNC 状态标记位*/
    wbc_attach_and_unlock_inode(&wbc, inode);
                                    /*writeback_control 关联 inode*/
    write_chunk = writeback_chunk_size(wb, work);        /*计算可回写缓存页数*/
    wbc.nr_to_write = write_chunk;
    wbc.pages_skipped = 0;

    __writeback_single_inode(inode, &wbc);          /*回写单个 inode，/fs/fs-writeback.c*/
```

```
            wbc_detach_inode(&wbc);         /*writeback_control 解绑 inode*/
            work->nr_pages -= write_chunk - wbc.nr_to_write;        /*还剩多少页需要回写*/
            wrote += write_chunk - wbc.nr_to_write;          /*写了多少页*/
            spin_lock(&wb->list_lock);
            spin_lock(&inode->i_lock);
            if (!(inode->i_state & I_DIRTY_ALL))
                wrote++;              /*inode 数量也加在其中*/
            requeue_inode(inode, wb, &wbc);       /*如果还脏则插入链表，不脏则移出链表*/
            inode_sync_complete(inode);       /*清 I_SYNC 状态标记位，唤醒等待进程*/
            spin_unlock(&inode->i_lock);
            cond_resched_lock(&wb->list_lock);          /*重调用*/

            if (wrote) {
                if (time_is_before_jiffies(start_time + HZ / 10UL))
                    break;
                if (work->nr_pages <= 0)
                    break;
            }
        }     /*while 循环结束*/
        return wrote;        /*回写的脏页和 inode 数量*/
    }
```

writeback_sb_inodes()函数从 wb->b_io 双链表末尾扫描指定超级块下的脏 inode，最终调用回写单个 inode 的**__writeback_single_inode()**函数回写 inode（包括页缓存和元数据），下面将介绍此函数的定义。

●回写单个脏 inode

回写单个 inode 的__writeback_single_inode()函数定义如下（/fs/fs-writeback.c）：

```
static int    __writeback_single_inode(struct inode *inode, struct writeback_control *wbc)
{
    struct address_space *mapping = inode->i_mapping;
    long nr_to_write = wbc->nr_to_write;
    unsigned dirty;
    int ret;

    WARN_ON(!(inode->i_state & I_SYNC));          /*此时应当设置了 I_SYNC 标记位*/

    trace_writeback_single_inode_start(inode, wbc, nr_to_write);

    ret = do_writepages(mapping, wbc);   /*调用写缓存页函数，回写脏页，/mm/page-writeback.c*/

    if (wbc->sync_mode == WB_SYNC_ALL && !wbc->for_sync) {
        int err = filemap_fdatawait(mapping);
                /*等待缓存页回写操作完成（真正写入块设备，而不是只提交请求），/mm/filemap.c*/
        if (ret == 0)
            ret = err;
    }
```

```
        /*文件系统类型代码可能使 inode 变脏*/
        spin_lock(&inode->i_lock);

        dirty = inode->i_state & I_DIRTY;
        if (inode->i_state & I_DIRTY_TIME) {              /*修改了时间戳*/
            if ((dirty & (I_DIRTY_SYNC | I_DIRTY_DATASYNC)) ||
                unlikely(inode->i_state & I_DIRTY_TIME_EXPIRED) ||
                unlikely(time_after(jiffies,(inode->dirtied_time_when +dirtytime_expire_interval * HZ)))) {
                dirty |= I_DIRTY_TIME | I_DIRTY_TIME_EXPIRED;
                trace_writeback_lazytime(inode);
            }
        } else
            inode->i_state &= ~I_DIRTY_TIME_EXPIRED;
        inode->i_state &= ~dirty;        /*清脏标记*/

        smp_mb();

        /*检测是否有缓存页设置了 PAGECACHE_TAG_DIRTY 标记*/
        if (mapping_tagged(mapping, PAGECACHE_TAG_DIRTY))
            inode->i_state |= I_DIRTY_PAGES;

        spin_unlock(&inode->i_lock);

        if (dirty & I_DIRTY_TIME)       /*inode 修改了时间戳*/
            mark_inode_dirty_sync(inode);          /*标记 inode 脏，I_DIRTY_SYNC，/include/linux/fs.h*/

        /*如果只设置了 I_DIRTY_PAGES 标记位则不回写 inode 元数据*/
        if (dirty & ~I_DIRTY_PAGES) {
            int err = write_inode(inode, wbc);          /*/fs/fs-writeback.c*/
                        /*回写 inode 元数据，调用 sb->s_op->write_inode(inode, wbc)*/
            if (ret == 0)
                ret = err;
        }
        trace_writeback_single_inode(inode, wbc, nr_to_write);
        return ret;
    }
```

__writeback_single_inode()函数简单地说就是调用 do_writepages()函数回写页缓存，调用 write_inode()函数回写 inode 元数据。

do_writepages()函数调用 mapping->a_ops->writepages()或 generic_writepages()函数写出缓存页至块设备。

write_inode()函数调用超级块操作结构中的 sb->s_op->write_inode()函数回写 inode 元数据。

11.3.5 用户同步

内核会自动发起数据回写，用户进程也可能通过系统调用触发数据回写。用户发起的回写操作和内核发起的回写操作本质上是一样的，只是触发的时机或回写的内容不同而已。

用户数据回写，暂且称它为用户同步，相关的系统调用主要有：

●**fsync(fd)/fdatasync(fd)**：回写进程文件，前者包含回写文件内容和元数据，后者只回写文件内容。

●**sync_file_range()**：同步文件指定区域内容。

●**msync()**：回写进程映射文件指定区域的文件内容。

●**syncfs(fd)**：回写 fd 文件所在文件系统的所有数据。

●**sync()**：系统同步，回写有的文件系统和块设备文件。

本小节简要介绍以上系统调用的实现。

1 同步文件

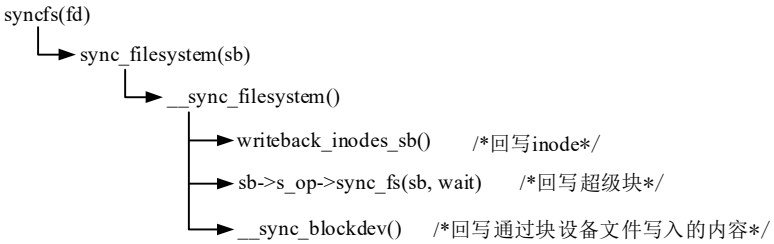
fsync(fd)和 fdatasync(fd)系统调用用于同步 fd 表示的进程文件，前者包含回写文件内容和元数据，后者只回写文件内容。两个系统调用内部都是调用 do_fsync(unsigned int fd, int datasync)函数执行回写操作，回写操作与前面介绍的同步写文件时执行的回写操作相同，即调用 file->f_op->fsync()函数执行回写，源代码请读者自行阅读（/fs/sync.c）。

sync_file_range()系统调用用于同步指定文件指定区域内容（/fs/sync.c）。

msync()系统调用回写指定映射文件范围内的数据，依然是调用文件操作结构中的 file->f_op->fsync()函数完成缓存页数据的同步（回写），源代码请读者自行阅读（/mm/msync.c）。

2 同步文件系统

syncfs(fd)系统调用用于回写 fd 表示的文件所在文件系统的数据，执行函数如下图所示：



同步文件系统主要有三项内容：一是回写文件系统下 inode（文件数据），二是回写文件系统超级块据，三是回写表示块设备文件的 inode（裸块设备）。以上函数源代码请读者自行阅读（/fs/sync.c）。

3 系统同步

sync()系统调用用于系统回写，系统调用执行函数如下（/fs/sync.c）：

```
SYSCALL_DEFINE0(sync)
{
    int nowait = 0, wait = 1;

    wakeup_flusher_threads(0, WB_REASON_SYNC);
                                        /*激活所有 backing_dev_info 实例的延时工作，见前文*/
    iterate_supers(sync_inodes_one_sb, NULL);
                        /*迭代超级块实例，调用 sync_inodes_one_sb()函数，回写脏 inode*/
    iterate_supers(sync_fs_one_sb, &nowait);        /*同步超块块*/
    iterate_supers(sync_fs_one_sb, &wait);          /*等待同步完成*/
    iterate_bdevs(fdatawrite_one_bdev, NULL);
                                /*遍历 bdev 伪文件系统，回写块设备文件（裸块设备）*/
    iterate_bdevs(fdatawait_one_bdev, NULL);        /*等待裸块设备文件回写完成*/
    if (unlikely(laptop_mode))
```

```
        laptop_sync_completion();       /*删除定时器 bdi->laptop_mode_wb_timer*/
    return 0;
}
```

sync()系统调用执行的操作其实与 syncfs()系统调用类似，就是相当于对系统所有挂载的文件系统执行 syncfs()系统调用。sync()系统调用是同步的，会等待所有回写完成才返回（确实写入块设备）。

wakeup_flusher_threads()函数在前面介绍过了，就是激活延时工作。iterate_supers()函数用于迭代内核中的超级块实例，对实例执行指定的函数。

sync_inodes_one_sb()函数回写超级块下 inode 实例，sync_fs_one_sb()函数调用 sb->s_op->sync_fs()函数回写超级块。iterate_bdevs()函数遍历 bdev 伪文件系统中表示块设备的 inode 实例，回写通过块设备文件写入的内容。

11.4 页回收与页交换

系统中的物理内存总是不够用的，因为运行的程序（软件）越来越多，消耗内存越来越大，况且打开的文件还会在内存中建立缓存。如果同时打开几个超大的文件，是不是物理内存一下就耗光了，这个作者没有仔细研究过，只是想说明内存总是不够用的现实。

物理内存不够就得对有限的内存进行管理调度。内核对分配给进程使用的内存，在空闲内存紧张时，会按照最近最少使用原则，对进程不怎么使用的内存页进行回收（释放），使其成为空闲内存。

分配给内核使用的内存一般不回收，除非内核自己把它释放，不过页回收机制可能会收缩 slab 缓存。

对用户内存的回收主要包括匿名映射页和缓存页（可能映射到进程地址空间）的回收。

对缓存页的回收比较简单，就是将缓存页中数据（脏页）写回块设备即可释放页，因为它本来就来自块设备中的文件。如果缓存页被映射到进程地址空间，则还要断开所有映射关系。下次再需要缓存页时，再分配页，从块设备读回数据即可。

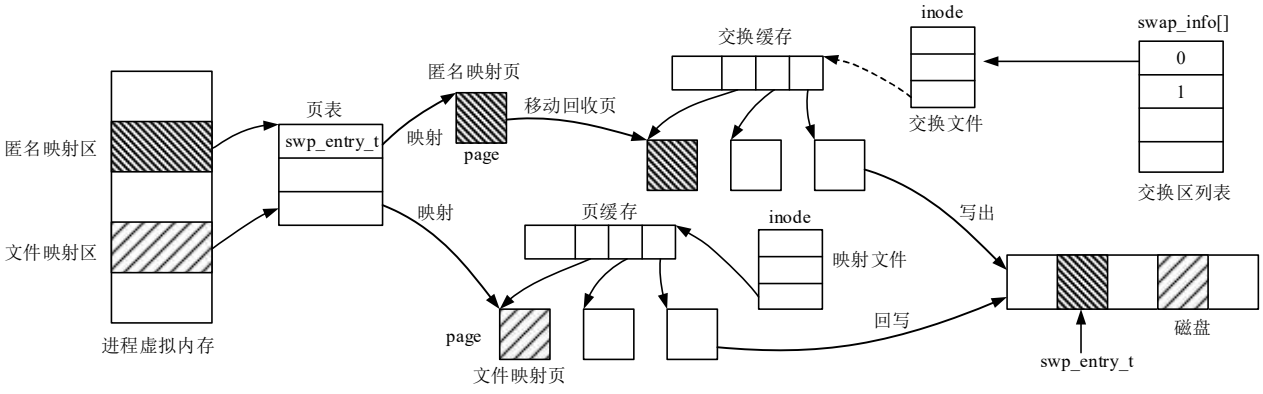
对匿名映射页（保存进程运行数据）的回收稍微复杂一些，因为页中数据不是来自块设备，而是进程运行时产生的，如果要回收匿名映射页，得为其在块设备中找个地方暂存数据，下次需要时再读回来，这称为页交换。页交换其实是页回收的一个副产品，只用于匿名映射页的回收。

用户可设置一个或数个文件或分区，用于暂存进程匿名映射页数据，这称为交换区，交换区可用一个文件来表示（分区由块设备文件表示），称它为交换文件。要回收的匿名映射页作为交换文件的内容添加到其页缓存中（交换缓存），然后写出到块设备，并把其在交换文件中的位置（简称槽位）记录在匿名映射页原来映射的页表项中，以便下次需要恢复页时能找回此页数据。

本节介绍页回收与页交换的实现。

11.4.1 概述

页回收机制简列如下图所示：



文件内容缓存页中的数据来自块设备，被选择回收的缓存页，如果为脏，则将其回写出块设备，如果缓存页被映射到进程地址空间，则还需要清除映射页表项，最后就可以释放缓存页使其成为空闲页了。

匿名映射页用于保存进程运行时产生的数据，如堆、栈区域就是匿名映射区。匿名映射页没有后备存储器，要回收它而不至于数据丢失，就要在块设备中为其找个位置暂存数据，下次需要时再恢复。

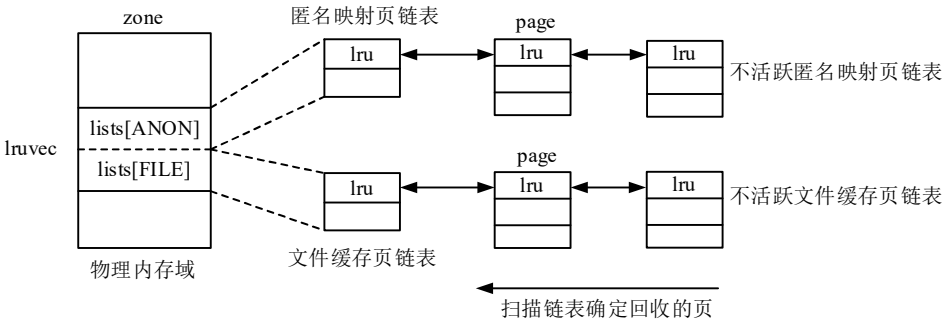
用户可将文件或分区设为交换区，用于暂存交换出去的匿名映射页的数据。交换区其实也是一个文件，称它为交换文件，如果是分区，则由块设备文件表示。被选择回收的匿名映射页添加到交换文件的页缓存中，作为文件内容，然后写出到块设备。

交换区（文件内容）按页进行划分，每个页位置称为一个槽位，槽位信息要记录至匿名映射页映射的页表项中。

当进程再次访问被交换出去的匿名映射页时，将触发缺页异常，在缺页异常处理程序中重新分配页，依页表项中保存的槽位信息，将页添加到交换缓存，并从交换区中读回数据，修改页表项，重新建立页与进程内存的映射关系，从而恢复匿名映射页。

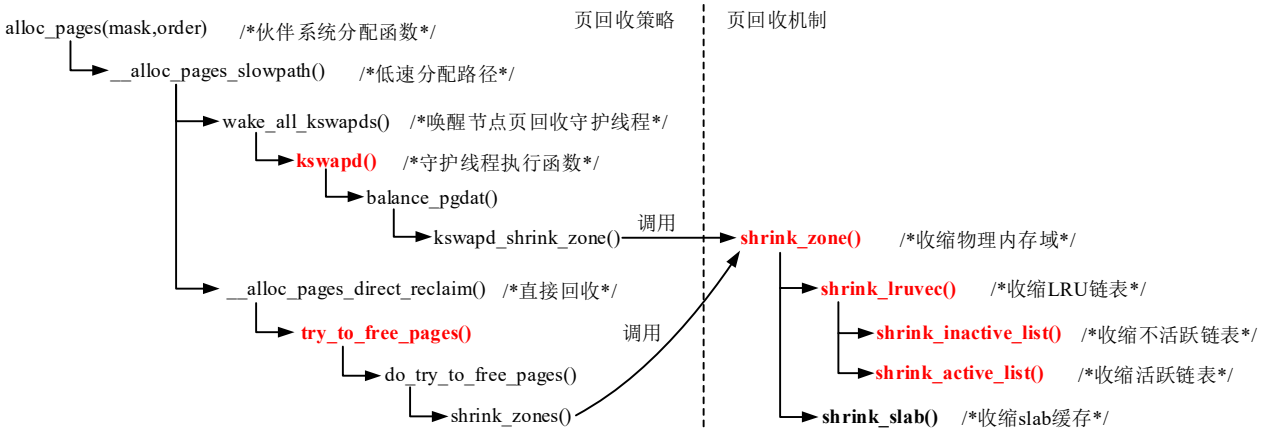
页回收机制中如何选择回收的匿名映射页和缓存页呢？

内核在物理内存域 zone 实例中分别建立了匿名映射页和文件缓存页的 LRU（最近最少使用）双链表，如下图所示。内核为进程分配的匿名映射页和文件缓存页将会插入到相应的 LRU 链表，包含活跃页链表和不活跃页链表。在系统运行过程中，内核通过一定的算法，使最近最少使用的页移动到 LRU 链表的末尾。页回收机制将从 LRU 链表末尾开始扫描，回收最近最不常使用的页。



页回收机制在什么时候触发呢？

在伙伴系统分配函数中，当空闲页帧不足时将会先激活各结点的 kswapd 守护线程，对结点进行页回收，如果回收后还是分配不成功，则调用__alloc_pages_direct_reclaim()函数执行直接页回收（更激进地回收页），函数调用关系如下图所示：



shrink_zone()函数用于执行真正的页回收工作，函数内调用 shrink_lruvec()函数扫描物理内存域页 LRU 双链表，回收匿名映射页或文件缓存页，调用 shrink_slab()函数收缩 slab 缓存。

物理内存域 LRU 双链表其实是双链表数组，包括匿名映射页的活跃、不活跃链表，文件缓存页的活跃、不活跃链表，以及不可回收页链表。活跃链表中是常被访问的页，不活跃链表中是不常被访问的页，页可能在活跃和不活跃链表之间迁移。

回收页主要是回收不活跃链表中页，回收的匿名映射页数据将写出到交换区，文件缓存页数据写出至

块设备（文件），断开进程与回收页之间的映射关系（需要将匿名映射页数据在交换区中位置写入页表项），最后释放回收页至伙伴系统。

shrink_active_list()函数用于扫描活跃页 LRU 链表，负责将扫描的页放回到活跃链表头部或移动到不活跃链表。shrink_inactive_list()函数用于扫描不活跃页 LRU 链表，对可回收的页执行回收操作，不可回收的页放回至适当的 LRU 链表。

shrink_slab()函数用于收缩注册了收缩器的 slab 缓存，以释放物理页帧。

本节首先介绍物理内存域中页 LRU 双链表的定义以及向 LRU 双链表插入页的操作，然后介绍页回收机制，收缩 LRU 链表、收缩内存域函数的实现，页回收守护线程及直接页回收函数的实现，最后介绍页交换机制的实现。

11.4.2 LRU 链表

物理内存域 zone 结构体中 lruvec 结构体成员 lruvec 表示页 LRU 链表，如下所示：

```
struct zone {
    ...
    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 双链表数组，用于页回收机制*/
    ...
}
```

lruvec 结构体定义如下（/include/linux/mmzone.h）：

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

lruvec 结构体中包含的双链表数量为 NR_LRU_LISTS：

```
#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*/
};
```

lruvec 结构体中包含 5 个 LRU 双链表，分别是：

（1）不活跃匿名映射页链表（回收页来自此链表）。

（2）活匿名映射页链表。

（3）不活跃文件缓存页链表（回收页来自此链表）。

（4）活跃文件缓存页链表。

（5）不可回收页链表。

其中不活跃和活跃 LRU 链表中的页是可以相互迁移的，页回收机制中将扫描不活跃链表，从中回收页。

这里的文件缓存页，是指外部块设备中文件的缓存页。匿名映射页包括映射到进程匿名映射区的页，以及基于内存的文件系统中文件缓存页（如 ramfs、tmpfs）等，也就是除了前面的文件缓存页，其它数据只存在于内存中的页。

lruvec 结构体中 reclaim_stat 成员是 zone_reclaim_stat 结构体，表示页回收中的统计量，结构体定义如下（/include/linux/mmzone.h）：

```
struct zone_reclaim_stat {
    unsigned long    recent_rotated[2];    /*整数数组，[0]：匿名映射页，[1]：缓存页*/
                                           /*活跃链表中分离出访问计数大于 0 的页数量，不活跃链表迁入的页*/
    unsigned long    recent_scanned[2];    /*整数数组，[0]：匿名映射页，[1]：缓存页*/
                                           /*活跃链表分离页数量*/
};
```

1 插入链表

在介绍将页插入 LRU 链表的函数前，先回顾一下页 page 结构体中相关的标记位，标记位位于 **flags** 成员中，标记位语义如下所示：

```
enum pageflags {
    PG_locked,          /*bit0，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,           /*bit1，置位表示基于页的 IO 操作产生错误*/
    PG_referenced,      /*bit2，用于页回收机制，置位表示页被引用*/
    PG_uptodate,        /*bit3，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,           /*bit4，脏标记，置位表示页中数据与块设备不同步，需要回写*/
    PG_lru,             /*bit5，页位于物理内存域可回收页 LRU 链表中*/
    PG_active,          /*bit6，活跃页*/
    PG_slab,            /*bit7，置位表示页帧被 slab（slub 或 slob）缓存使用*/
    ...
    PG_swapcache,       /*bit15，页位于交换缓存中，swp_entry_t 实例保存在 private 成员内*/
    PG_mappedtodisk,    /* bit16，Has blocks allocated on-disk */
    PG_reclaim,         /*bit17，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,      /*bit18，页正在从交换区读回至交换缓存*/
    PG_unevictable,     /*bit19，页位于不可回收页 LRU 链表*/
    ...
};
```

■插入页

lru_cache_add()函数用于将页插入适当的 LRU 链表（延迟进行），对于设置了 **PG_active** 标记位的页将插入到相应类型的活跃链表，没有设置 **PG_active** 标记位的页将插入到不活跃链表。内核分配给进程的匿名映射页通常插入到活跃匿名映射页链表，如果虚拟内存域是锁定的，则插入到不可回收页链表。分配给文件的缓存页（含文件映射页）通常插入到不活跃文件映射页链表。

内核定义了页向量 pagevec 结构体，用于成批收集需要插入到 LRU 链表中的页，定义如下：

```
#define PAGEVEC_SIZE     14            /*/include/linux/pagevec.h*/
struct pagevec {
    unsigned long nr;       /*pages[]数组中关联 page 实例的数量（已用数组项）*/
    unsigned long cold;
    struct page *pages[PAGEVEC_SIZE];       /*page 指针数组*/
};
```

内核在/mm/swap.c 文件内定义了全局 pagevec 结构体实例（percpu 变量）：

```
static DEFINE_PER_CPU(struct pagevec, lru_add_pvec);       /*需要添加到 LRU 链表的页*/
static DEFINE_PER_CPU(struct pagevec, lru_rotate_pvecs);
static DEFINE_PER_CPU(struct pagevec, lru_deactivate_file_pvecs);
                                                    /*需要添加到不活跃文件缓存页 LRU 链表的页*/
```

需要插入到 LRU 链表中的页将由以上全局变量收集，当 pages[PAGEVEC_SIZE]指针数组满时，将批量将指针数组中关联的页插入到 LRU 链表。

lru_cache_add(page)函数是将 page 指向实例添加到 LRU 链表的接口函数，对设置了 PG_active 标记位的实例将添加到活跃 LRU 链表，没有设置 PG_active 标记位的实例将添加到不活跃 LRU 链表。

lru_cache_add()函数定义如下（/mm/swap.c）：

```
void lru_cache_add(struct page *page)
{
    VM_BUG_ON_PAGE(PageActive(page) && PageUnevictable(page), page);
    VM_BUG_ON_PAGE(PageLRU(page), page);     /*page 需不在 LRU 链表中*/
    __lru_cache_add(page);     /*/mm/swap.c*/
}
```

lru_cache_add(page)函数调用__lru_cache_add(page)函数添加页，函数定义如下：

```
static void __lru_cache_add(struct page *page)
{
    struct pagevec *pvec = &get_cpu_var(lru_add_pvec);     /*获取当前 CPU 核的 pagevec 实例*/

    page_cache_get(page);          /*page 实例_count 计数加 1*/
    if (!pagevec_space(pvec))       /*如果 pages[]数组填满，则将数组中 page 实例添加到 LRU 链表*/
        __pagevec_lru_add(pvec);          /*将 pages[]数组中 page 批量插入 LRU 链表，/mm/swap.c*/
    pagevec_add(pvec, page);       /*将 page 实例关联到 pages[]数组项*/
    put_cpu_var(lru_add_pvec);     /*释放当前 CPU 核的 pagevec 实例*/
}
```

__lru_cache_add(page)函数内首先检查当前 CPU 核的 pagevec 实例中页向量是否已满，如果已满则将页向量中 page 实例添加到 LRU 链表，这由__pagevec_lru_add(pvec)函数完成，然后将 page 实例添加到页向量。如果页向量未满，则直接将 page 实例添加到页向量，暂不添加到 LRU 链表。

下面看一下__pagevec_lru_add(pvec)函数如何将页向量中 page 实例批量插入到 LRU 链表中，函数定义如下（/mm/swap.c）：

```
void __pagevec_lru_add(struct pagevec *pvec)
{
```

```
    pagevec_lru_move_fn(pvec, __pagevec_lru_add_fn, NULL);       /*/mm/swap.c*/
}

pagevec_lru_move_fn()函数定义如下：
static void pagevec_lru_move_fn(struct pagevec *pvec,   \
                void (*move_fn)(struct page *page, struct lruvec *lruvec, void *arg),void *arg)
{
    int i;
    struct zone *zone = NULL;        /*物理内存域*/
    struct lruvec *lruvec;
    unsigned long flags = 0;

    for (i = 0; i < pagevec_count(pvec); i++) {     /*遍历页向量中的 page 实例*/
        struct page *page = pvec->pages[i];
        struct zone *pagezone = page_zone(page);       /*page 所在物理内存域*/
                                    /*page 实例 flags 成员高位，保存了页所在物理内存域编号*/
        if (pagezone != zone) {
            if (zone)
                spin_unlock_irqrestore(&zone->lru_lock, flags);
            zone = pagezone;        /*指向 page 所在物理内存域*/
            spin_lock_irqsave(&zone->lru_lock, flags);
        }

        lruvec = mem_cgroup_page_lruvec(page, zone);
        (*move_fn)(page, lruvec, arg);     /*调用 move_fn()函数，此处为__pagevec_lru_add_fn()*/
    }      /*遍历页向量结束*/

    if (zone)
        spin_unlock_irqrestore(&zone->lru_lock, flags);

    release_pages(pvec->pages, pvec->nr, pvec->cold);   /*添加到页向量时，引用计数加 1，/mm/swap.c*/
            /*页向量中 page 实例引用计数减 1，如果为 0，则从 LRU 链表移除并释放至伙伴系统*/
    pagevec_reinit(pvec);        /*重新初始化 pvec 实例*/
}
```

pagevec_lru_move_fn()函数遍历 pvec 页向量中的 page 实例，对每个 page 实例调用 move_fn()函数，此处为__pagevec_lru_add_fn()函数，函数将 page 实例添加到相应的 LRU 链表。

```
__pagevec_lru_add_fn()函数定义如下：
static void __pagevec_lru_add_fn(struct page *page, struct lruvec *lruvec,void *arg)
{
    int file = page_is_file_cache(page); /*是否是外部块设备文件中缓存页，/include/linux/mm_inline.h*/
    int active = PageActive(page);                  /*是否是活跃页*/
    enum lru_list lru = page_lru(page);       /*确定 LRU 链表类型，/include/linux/mm_inline.h*/

    VM_BUG_ON_PAGE(PageLRU(page), page);       /*page 不能已在 LRU 链表*/
```

```
        SetPageLRU(page);          /*设置 page 的 PG_lru 标记*/
        add_page_to_lru_list(page, lruvec, lru);     /*/include/linux/mm_inline.h*/
                                /*将 page 实例插入到相应 LRU 链表头部，并更新物理内存域统计值*/
        update_page_reclaim_stat(lruvec, file, active);       /*/mm/swap.c*/
                                                /*更新 lruvec.reclaim_stat 成员统计值*/
        trace_mm_lru_insertion(page, lru);
    }
```

__pagevec_lru_add_fn()函数首先调用 page_lru(page)函数确定 page 实例插入的 LRU 链表类型，然后调用 add_page_to_lru_list()函数将 page 插入到相应链表头部，并更新统计值。

下面看一下 page_lru(page)函数的定义（/include/linux/mm_inline.h）：

```
    static __always_inline enum lru_list page_lru(struct page *page)
    {
        enum lru_list lru;

        if (PageUnevictable(page))          /*设置了 PG_unevictable 标记位，不可回收页*/
            lru = LRU_UNEVICTABLE;       /*不可回收页 LRU 链表*/
        else {
            lru = page_lru_base_type(page);
                                /*页类型（匿名映射页或文件缓存页），/include/linux/mm_inline.h*/
            if (PageActive(page))        /*设置了 PG_active 标记位，活跃页*/
                lru += LRU_ACTIVE;       /*添加到活跃 LRU 链表，否则为不活跃链表*/
        }
        return lru;
    }
```

总之，lru_cache_add(page)函数就是将 page 实例添加到对应类型的活跃或不活跃 LRU 链表头部，不过这里的添加是有延迟的，要等到页向量满了才会批量添加。

另外，lru_add_drain()/lru_add_drain_cpu(int cpu)/lru_add_drain_all()函数用于将当前 CPU 核/指定 CPU 核/所有 CPU 核的所有页向量（不只是 lru_add_pvec 页向量）中 page 实例插入到 LRU 链表（清空页向量），源代码请读者自行阅读（/mm/swap.c）。

■插入匿名映射页

用户进程缺页异常处理程序中，分配的匿名映射页将调用 **lru_cache_add_active_or_unevictable()**函数将添加到匿名映射页 LRU 链表。匿名映射页通常插入到活跃 LRU 链表，如果虚拟内存域是锁定的，匿名页将被添加到不可回收页 LRU 链表。

lru_cache_add_active_or_unevictable()函数代码如下（/mm/swap.c）：

```
    void lru_cache_add_active_or_unevictable(struct page *page,struct vm_area_struct *vma)
    {
        VM_BUG_ON_PAGE(PageLRU(page), page);

        if (likely((vma->vm_flags & (VM_LOCKED | VM_SPECIAL)) != VM_LOCKED)) {
                        /*（没有设置 VM_LOCKED）或（同时设置了 VM_LOCKED 和 VM_SPECIAL）*/
            SetPageActive(page);       /*设置 PG_active 标记位，活跃页*/
            lru_cache_add(page);       /*page 添加到活跃 LRU 链表，见上文，/mm/swap.c*/
```

```
        return;
    }
    if (!TestSetPageMlocked(page)) {
        __mod_zone_page_state(page_zone(page), NR_MLOCK,hpage_nr_pages(page));
        count_vm_event(UNEVICTABLE_PGMLOCKED);
    }
    add_page_to_unevictable_list(page);
                        /*锁定虚拟内存域的 page 实例添加到不可回收 LRU 链表，/mm/swap.c*/
}
```

缺页异常处理函数中分配的匿名映射页将设置 PG_uptodate 和 PG_swapbacked 标记位，如果不是锁定虚拟内存域的页将设置 PG_active 标记位，并添加到活跃 LRU 链表。锁定虚拟内存域的页添加到不可回收 LRU 链表。

■插入缓存页

内核分配文件缓存页时，通常调用 add_to_page_cache_lru()函数将其插入 LRU 链表，文件缓存页通常是被插入到不活跃 LRU 链表。

add_to_page_cache_lru()函数代码如下（前面也介绍过，/mm/filemap.c）：

```
int add_to_page_cache_lru(struct page *page, struct address_space *mapping,   \
                                                    pgoff_t offset, gfp_t gfp_mask)
{
    void *shadow = NULL;
    int ret;

    __set_page_locked(page);        /*置位页锁定标记位*/
    ret = __add_to_page_cache_locked(page, mapping, offset,gfp_mask, &shadow);
                    /*将 page 添加到基数树，成功返回 0，否则返回错误码，/mm/filemap.c*/

    if (unlikely(ret))      /*添加失败，清零锁定标记位*/
        __clear_page_locked(page);
    else {                  /*缓存页添加成功*/
        if (shadow && workingset_refault(shadow)) {
            SetPageActive(page);        /*缓存页添加到活跃 LRU 链表*/
            workingset_activation(page);
        } else
            ClearPageActive(page);        /*清除 PG_active 标记，插入不活跃 LRU 链表*/
        lru_cache_add(page);   /*将缓存页 page 添加到不活跃 LRU 链表，/mm/swap.c*/
    }
    return ret;
}
```

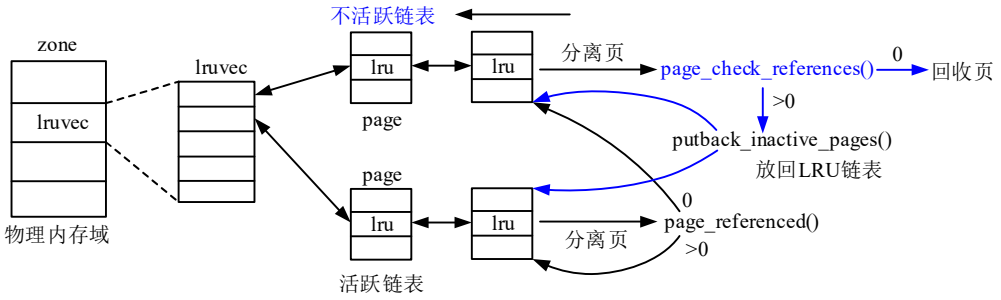
add_to_page_cache_lru()函数内调用__add_to_page_cache_locked()函数将缓存页 page 实例添加到地址空间基数树中，并设置 **page->mapping = mapping** 和 **page->index = offset**（缓存页编号），随后调用函数 lru_cache_add(page)将 page 实例添加到物理内存域中不活跃 LRU 链表（通常应该是这样），便于对缓存页进行回收。

2 回收机制

页回收主要包括回收物理内存域 LRU 链表中页，以及收缩 slab 缓存，这里先看 LRU 链表中的页的回收，称之为收缩 LRU 链表。

前面介绍的插入页至 LRU 链表的函数可知，通常匿名映射页插入活跃 LRU 链表头部，文件缓存页插入不活跃 LRU 链表头部。

回收页时，先扫描不活跃 LRU 链表，从后往前回收满足条件页，然后扫描活跃链表，从后往前扫描页，将满足条件的页移入不活跃链表，或保留在活跃链表中，如下图所示。



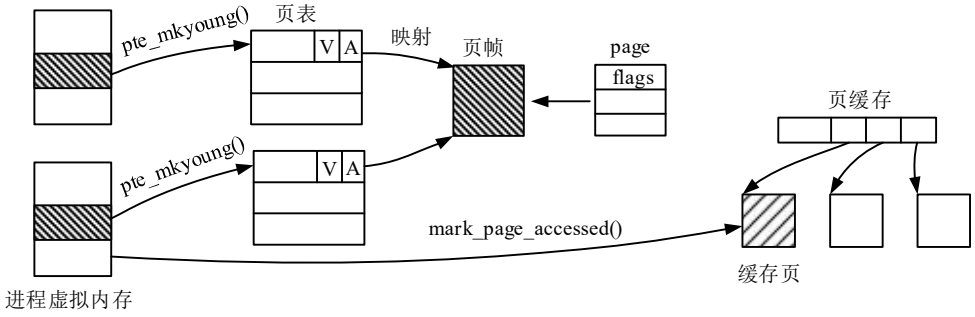
如果只是这样回收的话，那么先分配出来的页就先被回收，后分配出来的页将后被回收。这是不公平的，因为先分配出来的页可能一直在被使用，而后分配出来的页可能不怎么使用，如此容易引起页颠簸（页频繁换入换出）。

为此，内核增加了检查页的访问计数，扫描不活跃 LRU 链表时，只有访问计数为 0 的页才考虑回收，其它放回 LRU 链表。

扫描活跃 LRU 链表时，只有访问计数大于 0 且是具有映射的可执行文件缓存页，才放回活跃链表，其它页都移入不活跃链表。

那么，页的访问计数如何计算呢？

如下图所示，映射到进程虚拟地址空间的页，包括匿名映射页和文件映射页（映射缓存页），在每个进程的页表中有一个对应的页表项。页表项中 V 标记位表示页中数据是否有效，A 标记表示进程刚访问了此页。



在前面的图中，从活跃 LRU 链表中取出的页，由 page_references()函数统计页所有映射页表项中，A 标记位置 1 的页表项数，并将置 1 的页表项的 A、V 标记位都清 0，而且还会刷新此页表项在 TLB 中的项。

当进程再次访问该页时，由于 V 标记位为 0，将会触发 TLB 无效异常，在缺页异常处理函数中将会调用 **pte_mkyoung()**函数，重置 A、V 标记位为 1。

前面图中检测不活跃 LRU 链表中页的 page_check_references()函数也将调用 page_references()函数，统计页访问计数。

也就是说，只要是页回收扫描到的页都会调用 page_references()函数。由此可知，page_references()函数实际统计的是页在两次被扫描到之间被多少个进程访问了，只统计访问进程数，而不是统计被进程访问了多少次。在两次被扫描到之间，如果页没有被任何进程访问，将会被考虑回收或移入不活跃 LRU 链表。

对于没有映射到进程地址空间的缓存页，由于没有映射页表项，访问计数始终为 0，那该怎么标识页被访问呢？

为此，内核使用了 page 实例 flags 成员中的另一个标记位，即 PG_referenced 标记位，用于标识页被访问（引用）。mark_page_accessed()函数用于标记页被访问了，在读文件内容的 read()系统调用等时机会调用此函数标识缓存页被访问了。除缓存页外，mark_page_accessed()函数也会用来标识其它页被访问了。

mark_page_accessed()函数执行结果如下：

●如果 PG_active=0，PG_referenced=0，则转为 PG_active=0，PG_referenced=1（00+1=01）。

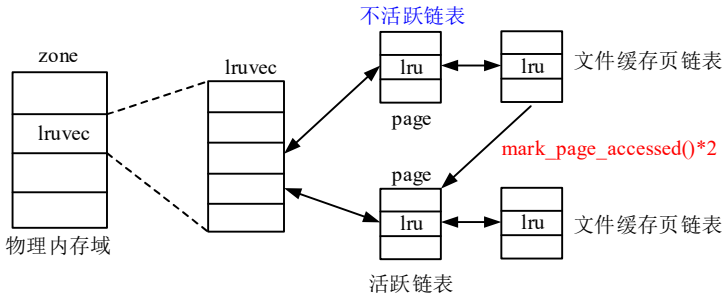
●如果 PG_active=0，PG_referenced=1，则转为 PG_active=1，PG_referenced=0（01+1=10）。

●如果 PG_active=1，PG_referenced=0，则转为 PG_active=1，PG_referenced=1（10+1=11）。

●如果 PG_active=1，PG_referenced=1，保持不变（11+1=11）。

PG_active 和 PG_referenced 标记位可视为组成一个 2bit 的二进制数，统计页被访问的次数（最小为 0，最大为 3），PG_active 表示 bit1，PG_referenced 表示 bit0。

不活跃 LRU 链表中的页，当 PG_referenced 标记位为 0 时，两次被 mark_page_accessed()函数标记被访问后将移入活跃 LRU 链表（暂时不被回收），如下图所示（这被称为第二次机会法）：



活跃 LRU 链表中的页被 mark_page_accessed()函数标记被访问后，位置不变，仍在活跃链表。活跃 LRU 链表中的页，在被页回收扫描到后，由于访问计数为 0，又会被移入不活跃 LRU 链表。不活跃 LRU 链表中的页被页回收扫描到后，由于访问计数为 0，将会被考虑回收。

下面介绍一下以上 page_references()、page_check_references()和 mark_page_accessed()函数的定义。

■统计页访问计数

page_referenced()函数用于统计映射页被多少个进程在页表项中标记为访问过（页表项中 A 标记位置位）。在介绍函数实现前先看一下 page_referenced_arg 结构体的定义（/mm/rmap.c）：

```
struct page_referenced_arg {
    int mapcount;          /*页映射计数*/
    int referenced;        /*A 标记位置位的映射页表项数量*/
    unsigned long vm_flags;      /*映射虚拟内存域标记*/
    struct mem_cgroup *memcg;
};
```

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

```
int page_referenced(struct page *page,int is_locked,struct mem_cgroup *memcg,unsigned long *vm_flags)
/*is_locked：页是否被锁定，vm_flags：指向虚拟内存域标记*/
{
    int ret;
    int we_locked = 0;
```

```
    struct page_referenced_arg pra = {              /*page_referenced_arg 实例*/
        .mapcount = page_mapcount(page),      /*(page->_mapcount) + 1，页映射计数*/
        .memcg = memcg,
    };
    struct rmap_walk_control rwc = {            /*遍历反向映射控制结构*/
        .rmap_one = page_referenced_one,      /*回调函数，/mm/rmap.c*/
        .arg = (void *)&pra,                        /*rmap_one()函数参数，page_referenced_arg 实例*/
        .anon_lock = page_lock_anon_vma_read,
    };

    *vm_flags = 0;
    if (!page_mapped(page))       /*页没有被映射（映射计数为 0），返回 0，/include/linux/mm.h*/
        return 0;

    if (!page_rmapping(page))       /*非匿名映射页或文件映射页，返回 0，/mm/util.c*/
        return 0;

    if (!is_locked && (!PageAnon(page) || PageKsm(page))) {        /*未锁定的文件映射页*/
        we_locked = trylock_page(page);        /*设置 PG_locked 标记位，返回原标记位值取反*/
        if (!we_locked)          /*PG_locked 标记位原置位，返回 1*/
            return 1;
    }

    if (memcg) {
        rwc.invalid_vma = invalid_page_referenced_vma;
    }

    ret = rmap_walk(page, &rwc);
                /*遍历反向映射结构，对各映射虚拟内存域调用 page_referenced_one()函数*/
    *vm_flags = pra.vm_flags;

    if (we_locked)
        unlock_page(page);

    return pra.referenced;   /*返回页被访问计数值*/
}
```

page_referenced()函数调用 rmap_walk(page, &rwc)函数遍历页反向映射结构（或文件地址间空间）中各虚拟内存域，对各虚拟内存域调用 rmap_walk_control.rmap_one()函数执行相应的操作。

在这里调用的是 page_referenced_one()函数，arg 参数指向 page_referenced_arg 实例（/mm/rmap.c）。

```
static int page_referenced_one(struct page *page, struct vm_area_struct *vma,   \
                                                    unsigned long address, void *arg)
{
    struct mm_struct *mm = vma->vm_mm;
    spinlock_t *ptl;
    int referenced = 0;        /*页被标记访问过的次数*/
```

```
        struct page_referenced_arg *pra = arg;

        if (unlikely(PageTransHuge(page))) {        /*巨型页*/
            ...
        } else {                /*普通页*/
            pte_t *pte;
            pte = page_check_address(page, mm, address, &ptl, 0);
                                    /*确认 page 映射到 address 地址，返回页表项指针（带锁）*/
            if (!pte)
                return SWAP_AGAIN;      /*rmap_walk()函数继续遍历反向映射结构中虚拟内存域*/

            if (vma->vm_flags & VM_LOCKED) {        /*虚拟内存域是锁定的，返回 SWAP_FAIL*/
                pte_unmap_unlock(pte, ptl);
                pra->vm_flags |= VM_LOCKED;
                return SWAP_FAIL;       /*rmap_walk()函数中止遍历反向映射结构*/
            }

            /*页表项没有设置 A 标记位，返回 0；设置了 A 标记位，返回 1，并清零 A，V 标记位*/
            /*注意，还会刷出内存页表项对应的 TLB 表项*/
            if (ptep_clear_flush_young_notify(vma, address, pte)) {
                if (likely(!(vma->vm_flags & VM_SEQ_READ)))     /*虚拟内存域未设置顺序读标记*/
                    referenced++;
            }
            pte_unmap_unlock(pte, ptl);     /*解锁页表项*/
        }

        if (referenced) {       /*页刚被标记访问过*/
            pra->referenced++;      /*增加访问计数*/
            pra->vm_flags |= vma->vm_flags;
        }

        pra->mapcount--;        /*映射计数减 1*/
        if (!pra->mapcount)
            return SWAP_SUCCESS;    /*映射计数归 0，表示遍历完了所有映射虚拟内存域*/

        return SWAP_AGAIN;      /*rmap_walk()函数继续遍历反向映射结构*/
    }
```

总之，page_referenced()函数就是统计页所有映射页表项中置 A 标记位的页表项数量，并将 A、V 标记位清 0，表示本次统计结束，从现在开始进入下次统计周期。

page_referenced()函数实际统计的是从上次对页调用 page_referenced()函数之后开始，到本次调用为止，在这个期间有多少个进程访问该页，统计的是进程数，而不是访问次数。

在扫描活跃链表中页时，page_referenced()函数统计访问计数为 0 的页，将移入不活跃 LRU 链表。

■检查不活跃页状态

扫描不活跃 LRU 链表时，对分离出的页将调用 page_check_references()函数检查页状态，以确定是回

收页，还是将页放回不活跃链表，或者移入活跃链表。

page_check_references()函数返回值为枚举类型 page_references，定义如下（/mm/vmscan.c）：

```
enum page_references {
    PAGEREF_RECLAIM,              /*可回收页*/
    PAGEREF_RECLAIM_CLEAN,    /*可回收干净页，不需要回写*/
    PAGEREF_KEEP,             /*页保持在不活跃链表*/
    PAGEREF_ACTIVATE,     /*页移动到活跃链表*/
};
```

page_check_references()函数定义如下（/mm/vmscan.c）：

```
static enum page_references page_check_references(struct page *page,struct scan_control *sc)
{
    int referenced_ptes, referenced_page;
    unsigned long vm_flags;

    referenced_ptes = page_referenced(page, 1, sc->target_mem_cgroup,&vm_flags);   /*访问计数值*/
    referenced_page = TestClearPageReferenced(page);   /*返回原 PG_referenced 标记值，并清零*/

    /*锁定的虚拟内存域，返回可回收状态，在处理分离页的操作中将其移动到不可回收链表*/
    if (vm_flags & VM_LOCKED)
        return PAGEREF_RECLAIM;

    /*页访问计数大于 0*/
    if (referenced_ptes) {
        if (PageSwapBacked(page))    /*非块设备中文件缓存页（PG_swapbacked），设为活跃页*/
            return   PAGEREF_ACTIVATE;

        SetPageReferenced(page);                  /*设置 PG_referenced 标记位*/

        if (referenced_page || referenced_ptes > 1)   /*原 PG_referenced 标记位置位或访问计数大于 1*/
            return   PAGEREF_ACTIVATE;        /*加入活跃链表*/

        if (vm_flags & VM_EXEC)          /*可执行文件映射缓存页，移动到活跃链表*/
            return   PAGEREF_ACTIVATE;         /*加入活跃链表*/

        return PAGEREF_KEEP;     /*保持在不活跃链表*/
    }

    /*页访问计数等于 0 且原 PG_referenced 置 1 的块设备中文件缓存页，可回收的干净页*/
    if (referenced_page && !PageSwapBacked(page))
        return PAGEREF_RECLAIM_CLEAN;      /*可回收干净缓存页*/

    /*访问计数等于 0 且原 PG_referenced 为 0 的文件缓存页，
    *或访问计数等于 0 的匿名映射页，可回页*/
    return PAGEREF_RECLAIM;
}
```

page_check_references()函数检查不活跃页的可回收状态，检查结果如下：

（1）访问计数大于 0 的匿名映射页，移入活跃链表。

（2）访问计数大于 0 的文件缓存页（块设备中文件），置位 PG_referenced 标记位，并再做如下判断：

①原 PG_referenced 标记位置 1 或访问计数大于 1，移入活跃链表。

②原 PG_referenced 标记位为 0，且引用计数小于等于 1 的可执行文件映射页，移入活跃链表，非可执行文件映射页，保持在不活跃链表。

（3）访问计数等于 0 且原 PG_referenced 标记位置 1 的文件缓存页，可回收的干净页。

（4）访问计数等于 0 且原 PG_referenced 标记位为 0 的文件缓存页，或访问计数为 0 的匿名映射页，可回收。

总之，访问计数大于 0 的不活跃页将移入活跃链表或保留在不活跃链表，访问计数为 0 的不活跃页可回收。

■标记页被访问

mark_page_accessed(struct page *page)函数用于标记页被访问过，函数定义在/mm/swap.c 文件内：

```
void mark_page_accessed(struct page *page)
{
    if (!PageActive(page) && !PageUnevictable(page) &&PageReferenced(page)) {
                                        /*不活跃、可回收且 PG_referenced=1*/
        if (PageLRU(page))              /*页在 LRU 链表*/
            activate_page(page);        /*设置 PG_active 标记位，并移入活跃链表，/mm/swap.c*/
        else
            __lru_cache_activate_page(page);      /*只设置 PG_active 标记位，/mm/swap.c*/

        ClearPageReferenced(page);      /*清零 PG_referenced 标记位*/
        if (page_is_file_cache(page))       /*文件缓存页*/
            workingset_activation(page);
                                    /*增加 page_zone(page)->inactive_age 计数，/mm/workingset.c*/
    } else if (!PageReferenced(page)) {
        SetPageReferenced(page);        /*设置 PG_referenced 标记位*/
    }
}
```

mark_page_accessed(page)函数执行后 page 实例 flags 成员的 PG_active 和 PG_referenced 标记位变化前面介绍过了，这里就不重复了。需要注意的是，如果页从不活跃变为活跃，将迁移到活跃 LRU 链表。

另外，SetPageActive(page)/ClearPageActive(page)和 SetPageReferenced(page)/ClearPageReferenced(page)函数可单独设置或清除 PG_active 或 PG_referenced 标记位（不改变页在链表中位置）。

11.4.3 收缩 LRU 链表

shrink_lruvec()函数用于扫描指定物理内存域的 LRU 链表，回收页。本小节介绍此函数的实现。

1 shrink_lruvec()

在介绍 shrink_lruvec()函数前，先看一下控制 LRU 链表扫描的 scan_control 结构体的定义：

```
struct scan_control {                   /*/mm/vmscan.c*/
    unsigned long nr_to_reclaim;    /*期望回收多少页*/
    gfp_t gfp_mask;         /*引发页回收的内存分配掩码*/
```

```
    int order;                  /*分配阶*/
    nodemask_t    *nodemask;      /*扫描结点掩码，NULL 表示可扫描所有结点*/
    struct mem_cgroup *target_mem_cgroup;
    int priority;       /*优先级，扫描页数量为 (链表中页数>> priority) ，值越小扫描页数量越多*/

    /*以下标记表示页回收过程中是否允许对页执行某种操作*/
    unsigned int may_writepage:1;     /*是否允许回写页（是否可回收脏页）*/
    unsigned int may_unmap:1;        /*是否允许解除页映射关系（是否回收映射页）*/
    unsigned int may_swap:1;          /*是否允许页交换*/
    unsigned int may_thrash:1;          /*cgroups*/
    unsigned int hibernation_mode:1;
    unsigned int compaction_ready:1;
    unsigned long nr_scanned;     /*不活跃链表扫描分离页数量*/
    unsigned long nr_reclaimed;   /*实际回收的页数量*/
};
```

收缩物理内存域 LRU 链表的 **shrink_lruvec()**函数定义如下（/mm/vmscan.c）：

```
static void shrink_lruvec(struct lruvec *lruvec, int swappiness,struct scan_control *sc,   \
                                                                    unsigned long *lru_pages)
/*
*lruvec：指向 zone.lruvec，sc：指向 scan_control 实例，lru_pages：指向整数用于统计页数，
*swappiness：换出匿名映射页的积极程度[0,100]，值越大越极，0 表示不换出，通常为 60。
*/
{
    unsigned long nr[NR_LRU_LISTS];           /*各 LRU 链表扫描的页数量，扫描过程中会减小*/
    unsigned long targets[NR_LRU_LISTS];     /*保存初始计算的需要扫描的 LRU 链表页数量*/
    unsigned long nr_to_scan;
    enum lru_list lru;     /*LRU 链表类型*/
    unsigned long nr_reclaimed = 0;
    unsigned long nr_to_reclaim = sc->nr_to_reclaim;     /*需要回收的页数量*/
    struct blk_plug plug;
    bool scan_adjusted;

    /*确定各 LRU 链表需扫描的页数量，函数定义见下文，/mm/vmscan.c*/
    get_scan_count(lruvec, swappiness, sc, nr, lru_pages);
    /*
     * nr[0]：不活跃匿名映射页链表扫描页数；  nr[1]：活跃匿名映射页链表扫描页数；
     * nr[2]：不活跃文件映射页链表扫描页数；nr[3]：活跃文件映射页链表扫描页数。
     */

    /*复制 nr[]数组至 targets[]数组*/
    memcpy(targets, nr, sizeof(nr));

    scan_adjusted = (global_reclaim(sc) && !current_is_kswapd() && sc->priority == DEF_PRIORITY);
                                                        /*直接页回收，且优先级值最小*/
    blk_start_plug(&plug);
```

```
while (nr[LRU_INACTIVE_ANON] || nr[LRU_ACTIVE_FILE] ||nr[LRU_INACTIVE_FILE]) {
                    /*除活跃匿名映射页链表外，其余链表有的扫描页数量尚未达到要求*/
    unsigned long nr_anon, nr_file, percentage;
    unsigned long nr_scanned;

    for_each_evictable_lru(lru) {   /*遍历可回收页 LRU 链表，先收缩不活跃链表，后活跃链表*/
        if (nr[lru]) {              /*需要扫描链表页*/
            nr_to_scan = min(nr[lru], SWAP_CLUSTER_MAX);    /*本次扫描的页数*/
            nr[lru] -= nr_to_scan;      /*减去本次扫描的页数*/
            nr_reclaimed += shrink_list(lru, nr_to_scan,lruvec, sc);
                        /*扫描指定 LRU 链表，返回回收的页数量，/mm/vmscan.c*/
       }
    }    /*遍历 LRU 链表结束，完成了一轮扫描，但可能还要进行下一轮扫描*/

    if (nr_reclaimed < nr_to_reclaim || scan_adjusted)
          continue;          /*没有回收足够的页或 scan_adjusted==1，继续扫描*/


    /*回收了足够数量的页且 scan_adjusted==0*/
    nr_file = nr[LRU_INACTIVE_FILE] + nr[LRU_ACTIVE_FILE];
                                                /*剩余未扫描的缓存页数量*/
    nr_anon = nr[LRU_INACTIVE_ANON] + nr[LRU_ACTIVE_ANON];
                                                /*剩余未扫描的匿名映射页数量*/

    if (!nr_file || !nr_anon)    /*扫描完指定数量的缓存页或匿名映射页，跳出循环*/
        break;

    if (nr_file > nr_anon) {    /*剩余缓存页数量多于匿名映射页*/
        unsigned long scan_target = targets[LRU_INACTIVE_ANON] +
                    targets[LRU_ACTIVE_ANON] + 1;    /*原定的扫描匿名映射页总数+1*/
        lru = LRU_BASE;
        percentage = nr_anon * 100 / scan_target;      /*未扫描匿名映射页所占百分比*/
    } else {                    /*剩余缓存页数量少于匿名映射页*/
        unsigned long scan_target = targets[LRU_INACTIVE_FILE] +
                targets[LRU_ACTIVE_FILE] + 1;    /*原定的扫描缓存页总数*/
        lru = LRU_FILE;
        percentage = nr_file * 100 / scan_target;       /*未扫描缓存页所占百分比*/
    }

/*未扫描页数量少的 LRU 链表停止扫描*/
nr[lru] = 0;
nr[lru + LRU_ACTIVE] = 0;

lru = (lru == LRU_FILE) ? LRU_BASE : LRU_FILE;   /*继续扫描 LRU 链表类型*/
nr_scanned = targets[lru] - nr[lru];        /*已经扫描的页数*/
```

```
            nr[lru] = targets[lru] * (100 - percentage) / 100;    /*重新计算不活跃链表扫描页数量*/
            nr[lru] -= min(nr[lru], nr_scanned);

            lru += LRU_ACTIVE;                    /*重新计算活跃链表扫描页数量*/
            nr_scanned = targets[lru] - nr[lru];
            nr[lru] = targets[lru] * (100 - percentage) / 100;
            nr[lru] -= min(nr[lru], nr_scanned);

            scan_adjusted = true;          /*已调整扫描页数量，继续收缩 LRU 链表*/
        }    /*while 循环结束*/

        blk_finish_plug(&plug);
        sc->nr_reclaimed += nr_reclaimed;
        /*不活跃匿名映射页比例是否太低（系统需设有交换区）*/
        if (inactive_anon_is_low(lruvec))    /*/mm/vmscan.c*/
            shrink_active_list(SWAP_CLUSTER_MAX, lruvec,sc, LRU_ACTIVE_ANON);
                                    /*收缩活跃匿名映射页 LRU 链表，向不活跃链表迁移页*/
        throttle_vm_writeout(sc->gfp_mask);
    }
```

shrink_lruvec()函数还是比较好理解的，简要执行流程如下：

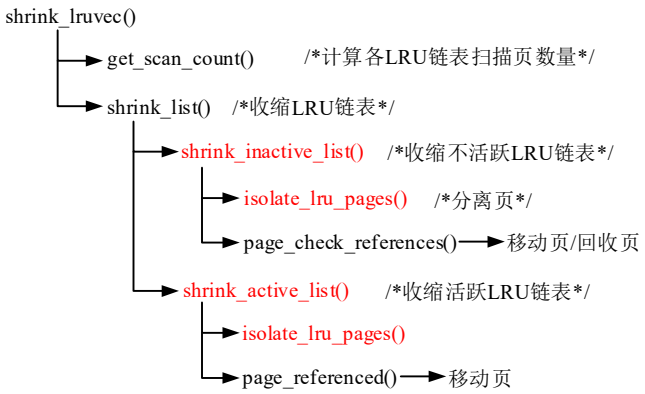
（1）调用 **get_scan_count**()函数计算需要扫描各 LRU 链表的页数量，保存至 nr[]数组（targets[]数组留有备份）。

（2）遍历物理内存域 **lruvec** 中各 LRU 链表，调用 **shrink_list()**函数收缩 LRU 链表，重复此步骤，直到回收到了足够的页，或需要扫描的页数量都扫描完了。

（3）如果回收了足够的页，但是扫描页数量还没有完成，则只对剩余未扫描页数少的类型的 LRU 链表继续扫描回收页。

（4）如果系统设有交换区，则检查不活跃匿名映射页比例是否过低，如果是则收缩活跃匿名映射页链表，迁移页到不活跃链表，否则不收缩。

shrink_lruvec()函数调用关系简列如下图所示：



get_scan_count()函数用于计算需要扫描各 LRU 链表的页数量，后面将介绍此函数的实现。

shrink_list()函数用于收缩指定的 LRU 链表，如果是不活跃链表则调用 shrink_inactive_list()函数处理，活跃链表调用 shrink_active_list()函数处理。这两个函数都调用 isolate_lru_pages()函数从链表中分离出部分页，然后对分离了来的页进行处理，后面将详细介绍这两个函数的实现。

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

```
static unsigned long shrink_list(enum lru_list lru, unsigned long nr_to_scan,    \
                                                    struct lruvec *lruvec, struct scan_control *sc)
{
    if (is_active_lru(lru)) {            /*活跃 LRU 链表*/
        if (inactive_list_is_low(lruvec, lru))       /*同类型的不活跃链表页数量比例是否太低*/
            shrink_active_list(nr_to_scan, lruvec, sc, lru);     /*收缩活跃链表，/mm/vmscan.c*/
        return 0;
    }
    return shrink_inactive_list(nr_to_scan, lruvec, sc, lru);     /*收缩不活跃链表，/mm/vmscan.c*/
}
```

如果收缩的是活跃链表且同类型不活跃链表页数量比例低，则调用 shrink_active_list()函数收缩活跃链表，从活跃链表迁移页到不活跃链表，否则不收缩活跃链表。

如果收缩的是不活跃链表，则直接调用 shrink_inactive_list()函数收缩不活跃链表。

inactive_list_is_low()函数（/mm/vmscan.c）用于计算不活跃链表中页数量比例是否过低，如果是缓存页链表，若活跃链表中页总数大于不活跃链表中页总数，则函数返回 true，表示要收缩活跃链表，否则不收缩。

如果是匿名映射页链表，系统需设有交换区，才会考虑收缩活跃链表，否则不收缩。如果系统设有交换区，且 active>inactive * **zone->inactive_ratio**（active 为活跃链表页数量，inactive 为不活跃链表页数量，inactive_ratio 为活跃链表页数与不活跃链表页数最大比值，值大于 1），则收缩活跃匿名映射页链表，否则不收缩。

2 准备工作

在介绍收缩不活跃和活跃 LRU 链表的函数前，先介绍 get_scan_count()和 isolate_lru_pages()函数的实现，前者用于计算要扫描链表中页的数量，后者用于从链表中分离出部分页。

■计算扫描页数量

get_scan_count()函数用于计算各 LRU 链表需要扫描的页数量，函数定义如下（/mm/vmscan.c）：

```
static void get_scan_count(struct lruvec *lruvec, int swappiness,    \
                        struct scan_control *sc, unsigned long *nr,unsigned long *lru_pages)
/*
*swappiness：换出匿名映射页的积极程度[0,100]，值越大越积极，0 表示不换出，通常为 60；
*nr：指向整数数组，保存各链表需要扫描的页数。
*nr[0] =不活跃匿名映射页链表扫描页数; nr[1] = 活跃匿名映射页链表扫描页数；
*nr[2] = 不活跃文件映射页链表扫描页数; nr[3] = 活跃文件映射页链表扫描页数;

*sc：指向扫描控制结构，*lru_pages：保存各 LRU 链表中页数量之和。
*/
{
    struct zone_reclaim_stat *reclaim_stat = &lruvec->reclaim_stat;
    u64 fraction[2];
    u64 denominator = 0;    /* gcc */
    struct zone *zone = lruvec_zone(lruvec);
    unsigned long anon_prio, file_prio;
```

```
enum scan_balance scan_balance;     /*表示扫描哪些 LRU 链表，/mm/vmscan.c*/
unsigned long anon, file;
bool force_scan = false;
unsigned long ap, fp;
enum lru_list lru;
bool some_scanned;
int pass;

if (current_is_kswapd()) {          /*如果当前进程是页回收守护线程*/
    if (!zone_reclaimable(zone))
        force_scan = true;          /*强制扫描链表*/
    if (!mem_cgroup_lruvec_online(lruvec))
        force_scan = true;
}
if (!global_reclaim(sc))            /*没有选择 MEMCG，global_reclaim(sc)返回 true*/
    force_scan = true;

/*如果页回收控制不允许页交换或交换区内页数量小于等于 0*/
if (!sc->may_swap || (get_nr_swap_pages() <= 0)) {        /*没有交换区不扫描匿名映射页链表*/
     scan_balance = SCAN_FILE;      /*只扫描文件映射页链表*/
     goto out;
}

if (!global_reclaim(sc) && !swappiness) {        /*swappiness 为 0，也不扫描匿名映射页链表*/
     scan_balance = SCAN_FILE;
     goto out;
}

if (!sc->priority && swappiness) {
     scan_balance = SCAN_EQUAL;       /*扫描匿名映射和缓存页链表*/
     goto out;
}

if (global_reclaim(sc)) {       /*没有选择 MEMCG，直接返回 true*/
     unsigned long zonefile;
     unsigned long zonefree;

    zonefree = zone_page_state(zone, NR_FREE_PAGES);       /*空闲页数量*/
    zonefile = zone_page_state(zone, NR_ACTIVE_FILE) +
                                zone_page_state(zone, NR_INACTIVE_FILE);
                                        /*活跃和不活跃缓存页数量之和*/

    if (unlikely(zonefile + zonefree <= high_wmark_pages(zone))) {
         scan_balance = SCAN_ANON;         /*只扫描匿名映射页链表*/
         goto out;
    }
```

```
    }

    /*有足够的不活跃文件缓存页，只扫描缓存页链表*/
    if (!inactive_file_is_low(lruvec)) {
            scan_balance = SCAN_FILE;
            goto out;
    }

    scan_balance = SCAN_FRACT;

    /*按比例计算匿名映射页和缓存页链表扫描页数*/
    anon_prio = swappiness;
    file_prio = 200 - anon_prio;

    anon    = get_lru_size(lruvec, LRU_ACTIVE_ANON) +
                                    get_lru_size(lruvec, LRU_INACTIVE_ANON);
    file    = get_lru_size(lruvec, LRU_ACTIVE_FILE) +
                                    get_lru_size(lruvec, LRU_INACTIVE_FILE);

    spin_lock_irq(&zone->lru_lock);
    if (unlikely(reclaim_stat->recent_scanned[0] > anon / 4)) {
            reclaim_stat->recent_scanned[0] /= 2;
            reclaim_stat->recent_rotated[0] /= 2;
    }

    if (unlikely(reclaim_stat->recent_scanned[1] > file / 4)) {
            reclaim_stat->recent_scanned[1] /= 2;
            reclaim_stat->recent_rotated[1] /= 2;
    }

    ap = anon_prio * (reclaim_stat->recent_scanned[0] + 1);
    ap /= reclaim_stat->recent_rotated[0] + 1;

    fp = file_prio * (reclaim_stat->recent_scanned[1] + 1);
    fp /= reclaim_stat->recent_rotated[1] + 1;
    spin_unlock_irq(&zone->lru_lock);

    fraction[0] = ap;
    fraction[1] = fp;
    denominator = ap + fp + 1;
out:                                    /*计算各 LRU 链表扫描的页数*/
    some_scanned = false;
    for (pass = 0; !some_scanned && pass < 2; pass++) {     /*第一个 for 循环*/
            *lru_pages = 0;
            for_each_evictable_lru(lru) {     /*第二个 for 循环，遍历各可回收 LRU 链表*/
                    int file = is_file_lru(lru);
```

```
                unsigned long size;
                unsigned long scan;

                size = get_lru_size(lruvec, lru);     /*lru 指定链表中页数量*/
                scan = size >> sc->priority;       /*按优先级计算的扫描页数*/

                if (!scan && pass && force_scan)           /*scan 为 0 的情形*/
                        scan = min(size, SWAP_CLUSTER_MAX);

                switch (scan_balance) {        /*扫描哪些链表*/
                case SCAN_EQUAL:
                        break;
                case SCAN_FRACT:
                        scan = div64_u64(scan * fraction[file],denominator);     /*按比例计算的扫描页数*/
                        break;
                case SCAN_FILE:
                case SCAN_ANON:
                        if ((scan_balance == SCAN_FILE) != file) {
                            size = 0;       /*匿名映射页扫描页数设为 0*/
                            scan = 0;
                        }
                        break;
                default:
                        BUG();
                }

                *lru_pages += size;     /*各 LRU 链表页数之和*/
                nr[lru] = scan;           /*lru 链表扫描页数*/
                some_scanned |= !!scan;      /*计算出了扫描页数，（第一个 for 循环只运行一次）*/
            }  /*第二个 for 循环结束，遍历 LRU 链表结束*/
        }    /*第一个 for 循环结束*/
    }
```

get_scan_count()函数首先确定扫描哪些 LRU 链表，例如：只扫描文件缓存页链表（未使用交换区）、平等或按比例扫描缓存页和匿名映射页链表等；然后遍历各 LRU 链表，根据链表中页数量和扫描优先级，计算扫描链表的页数量（size >> sc->priority，可能还要修正），赋予参数 nr[]数组项。

■分离页

isolate_lru_pages()函数用于从活跃或不活跃 LRU 链表分离出部分页。分离函数需设置分离模式参数，即分离页需满足的条件，定义如下（/include/linux/mmzone.h）：

```
#define    ISOLATE_CLEAN       ((__force isolate_mode_t)0x1)       /*分离干净的文件缓存页*/
#define    ISOLATE_UNMAPPED      ((__force isolate_mode_t)0x2)   /*分离未映射的文件缓存页*/
#define    ISOLATE_ASYNC_MIGRATE     ((__force isolate_mode_t)0x4)    /*分离异步合并的页*/
#define    ISOLATE_UNEVICTABLE       ((__force isolate_mode_t)0x8)    /*分离不可回收页*/

typedef unsigned __bitwise__    isolate_mode_t;      /*分离模式数据结构*/
```

isolate_lru_pages()函数定义如下（/mm/vmscan.c）：

```
static unsigned long isolate_lru_pages(unsigned long nr_to_scan,struct lruvec *lruvec, struct list_head *dst,
                unsigned long *nr_scanned, struct scan_control *sc,isolate_mode_t mode, enum lru_list lru)
/*
*nr_to_scan：扫描链表中的页数量，*nr_scanned：实际扫描的页数量，lru：LRU 链表类型，
*dst：暂存分离页的双链表，mode：分离模式，sc：指向扫描控制结构。函数返回实际分离页数量。
*/
{
    struct list_head *src = &lruvec->lists[lru];       /*扫描的 LRU 链表*/
    unsigned long nr_taken = 0;
    unsigned long scan;

    for (scan = 0; scan < nr_to_scan && !list_empty(src); scan++) {    /*扫描链表中 nr_to_scan 数量页*/
        struct page *page;
        int nr_pages;

        page = lru_to_page(src);     /*获取链表末尾页，从后向前扫描*/
        prefetchw_prev_lru_page(page, src, flags);

        VM_BUG_ON_PAGE(!PageLRU(page), page);

        switch (__isolate_lru_page(page, mode)) {     /*判断页是否可分离*/
        case 0:          /*可分离页*/
            nr_pages = hpage_nr_pages(page);
            mem_cgroup_update_lru_size(lruvec, lru, -nr_pages);
            list_move(&page->lru, dst);       /*将页移动到 dst 链表头部*/
            nr_taken += nr_pages;             /*分离页数量增加*/
            break;

        case -EBUSY:
            list_move(&page->lru, src);       /*将忙页移动到原 LRU 链表头部*/
            continue;

        default:
            BUG();
        }
    }     /*for 循环结束，扫描链表结束*/

    *nr_scanned = scan;     /*实际扫描的页数量*/
    trace_mm_vmscan_lru_isolate(sc->order, nr_to_scan, scan, nr_taken, mode, is_file_lru(lru));
    return nr_taken;     /*返回实际分离页数量*/
}
```

isolate_lru_pages()函数从后往前遍历 LRU 链表中页，调用__isolate_lru_page()函数判断页是否可分离，

可分离则移入 dst 临时双链表，否则移入原 LRU 链表头部，函数返回实际分离的页数量。

__isolate_lru_page()函数用于判断 LRU 链表中页是否可分离，函数定义如下（/mm/vmscan.c）：

```
int __isolate_lru_page(struct page *page, isolate_mode_t mode)
{
        int ret = -EINVAL;          /*返回值*/

        if (!PageLRU(page))         /*page 不在 LRU 链表中，返回错误码*/
            return ret;

        if (PageUnevictable(page) && !(mode & ISOLATE_UNEVICTABLE))    /*不可分离不可回收页*/
            return ret;

        ret = -EBUSY;           /*对象忙错误码*/
        if (mode & (ISOLATE_CLEAN|ISOLATE_ASYNC_MIGRATE)) {
            if (PageWriteback(page))     /*分离页正在回写*/
                return ret;

            if (PageDirty(page)) {          /*脏页*/
                struct address_space *mapping;
                if (mode & ISOLATE_CLEAN)     /*设置了 ISOLATE_CLEAN 模式，但页脏*/
                    return ret;
                mapping = page_mapping(page);
                if (mapping && !mapping->a_ops->migratepage)
                    return ret;
            }
        }

        if ((mode & ISOLATE_UNMAPPED) && page_mapped(page))     /*只分离非映射缓存页*/
            return ret;         /*分离非映射页，但页已映射*/

        if (likely(get_page_unless_zero(page))) {
                                /*引用计数加 1 后，计数值是否不为 0，/include/linux/mm.h*/
            ClearPageLRU(page);           /*清 PG_lru 标记位*/
            ret = 0;                    /*返回 0 表示页可分离*/

        }
        return ret;
}
```

__isolate_lru_page()函数返回-EINVAL 表示分离页的操作无效，-EBUSY 表示页正忙（如正在回写等）暂时不能执行分离操作，0 表示页可以从 LRU 链表中分离。

3 收缩活跃链表

收缩活跃 LRU 链表的 shrink_active_list()函数要简单一些，因此先看此函数的实现。

shrink_active_list()函数主要工作是从链表中分离出一定数量的页，然后逐一判断分离页的访问计数，

对访问计数大于 0 且是具有映射的可执行文件缓存页，放回活跃链表，其它页都移入不活跃链表。

shrink_active_list()函数定义如下（/mm/vmscan.c）：

```
static void shrink_active_list(unsigned long nr_to_scan,struct lruvec *lruvec,struct scan_control *sc, \
                                                                    enum lru_list lru)
{
    unsigned long nr_taken;
    unsigned long nr_scanned;
    unsigned long vm_flags;
    LIST_HEAD(l_hold);      /*暂存从 LRU 链表中分离出的页*/
    LIST_HEAD(l_active);    /*暂存放回活跃链表的页*/
    LIST_HEAD(l_inactive);  /暂存插入不活跃链表的页*/
    struct page *page;
    struct zone_reclaim_stat *reclaim_stat = &lruvec->reclaim_stat;
    unsigned long nr_rotated = 0;
    isolate_mode_t isolate_mode = 0;
    int file = is_file_lru(lru);        /*是否是缓存页 LRU 链表*/
    struct zone *zone = lruvec_zone(lruvec);

    lru_add_drain();    /*将 CPU 核所有页向量中缓存的页添加到 LRU 链表*/

    if (!sc->may_unmap)     /*没有设置 sc->may_unmap*/
        isolate_mode |= ISOLATE_UNMAPPED;   /*分离未映射页，/include/linux/mmzone.h*/
    if (!sc->may_writepage)     /*没有设置 sc->may_writepage*/
        isolate_mode |= ISOLATE_CLEAN;      /*分离干净页（非脏页）*/

    spin_lock_irq(&zone->lru_lock);         /*持有链表自旋锁*/

    nr_taken = isolate_lru_pages(nr_to_scan, lruvec, &l_hold,&nr_scanned, sc, isolate_mode, lru);
                            /*分离页，添加到 l_hold 链表，返回分离页数量，/mm/vmscan.c*/
    if (global_reclaim(sc))
        __mod_zone_page_state(zone, NR_PAGES_SCANNED, nr_scanned);   /*增加扫描页统计量*/

    reclaim_stat->recent_scanned[file] += nr_taken;   /*增加分离页统计量*/

    __count_zone_vm_events(PGREFILL, zone, nr_scanned);
    __mod_zone_page_state(zone, NR_LRU_BASE + lru, -nr_taken);   /*减小链表页数统计量*/
    __mod_zone_page_state(zone, NR_ISOLATED_ANON + file, nr_taken);
    spin_unlock_irq(&zone->lru_lock);       /*释放链表自旋锁*/

    while (!list_empty(&l_hold)) {          /*遍历 l_hold 链表中分离页*/
        cond_resched();
        page = lru_to_page(&l_hold);     /*从链表末尾取 page 实例*/
        list_del(&page->lru);               /*page 从 l_hold 链表移除*/

        if (unlikely(!page_evictable(page))) {      /*不可回收页，放回 LRU 链表*/
```

```
                putback_lru_page(page);     /*/mm/vmscan.c*/
                continue;
            }

            if (unlikely(buffer_heads_over_limit)) {
                if (page_has_private(page) && trylock_page(page)) {
                    if (page_has_private(page))
                        try_to_release_page(page, 0);   /*释放块缓存头实例，/mm/filemap.c*/
                    unlock_page(page);
                }
            }

            if (page_referenced(page, 0, sc->target_mem_cgroup,&vm_flags)) {     /*页访问计数非 0*/
                nr_rotated += hpage_nr_pages(page);          /*访问计数大于 0 的页数量*/
                if ((vm_flags & VM_EXEC) && page_is_file_cache(page)) {
                                                    /*可执行文件的映射页，留在活跃链表*/
                    list_add(&page->lru, &l_active);       /*将页添加到 l_active 链表*/
                    continue;
                }
            }

            /*访问计数非 0 的非映射可执行文件缓存页，以及访问计数为 0 的页，移入不活跃链表*/
            ClearPageActive(page);                  /*清 PG_active 标记位*/
            list_add(&page->lru, &l_inactive);     /*将页添加到 l_inactive 链表，将移到不活跃链表*/
        }    /*遍历分离页结束*/

        spin_lock_irq(&zone->lru_lock);          /*持有自旋锁*/
        reclaim_stat->recent_rotated[file] += nr_rotated;       /*访问计数大于 0 的页数量*/
        move_active_pages_to_lru(lruvec, &l_active, &l_hold, lru);
                                                    /*分离页放回活跃链表，/mm/vmscan.c*/
        move_active_pages_to_lru(lruvec, &l_inactive, &l_hold, lru - LRU_ACTIVE);
                                                    /*分离页移到不活跃链表*/
        __mod_zone_page_state(zone, NR_ISOLATED_ANON + file, -nr_taken);   /*修改分离页统计值*/
        spin_unlock_irq(&zone->lru_lock);     /*释放自旋锁*/

        mem_cgroup_uncharge_list(&l_hold);
        free_hot_cold_page_list(&l_hold, true);   /*释放引用计数为 0 的页至伙伴系统*/
    }
```

shrink_active_list()函数遍历分离页，对页进行以下处理：

（1）访问计数为 0 的页，放入 l_inactive 暂存链表。

（2）访问计数大于 0 的页，且是可执行文件的映射页（缓存页映射到用户空间），放入 l_active 暂存链表。访问计数大于 0 的其它页，放入 l_inactive 暂存链表。

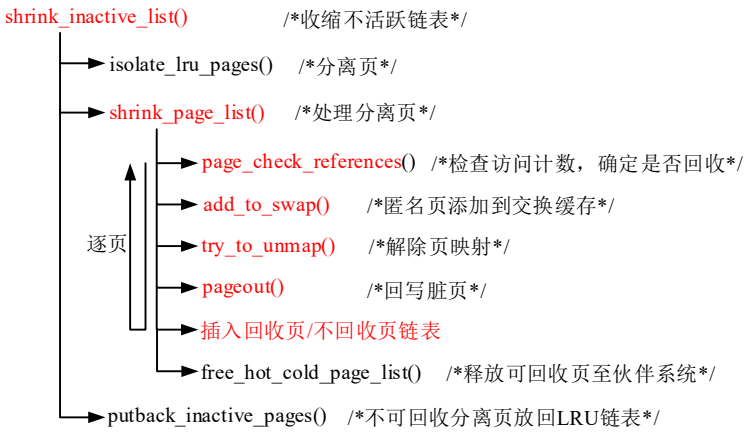
shrink_active_list()函数随后将 l_active 链表中页放回活跃链表，l_inactive 链表中的页移入不活跃链表，在这个过程中引用计数（page->_count）为 0 的页将放回 l_hold 链表，最后释放回伙伴系统（回收页）。

4 收缩不活跃链表

收缩不活跃链表比收缩活跃链表的操作要复杂一些，因为它需要执行实质的页回收操作。对于回收的匿名映射页，需要将其移入交换缓存，对回收的映射页要解除映射，回收的脏页要执行回写，这其中包括将交换缓存中的匿名映射页写入交换区，最后才能将回收页释放回伙伴系统。

■收缩链表函数

收缩不活跃 LRU 链表的 shrink_inactive_list()函数执行流程简列如下图所示：



shrink_inactive_list()函数执行步骤如下：

（1）调用 isolate_lru_pages()函数分离出部分页。

（2）调用 shrink_page_list()函数逐一处理分离出来的页，回收的页最后由 free_hot_cold_page_list()函数统一释放，不回收的页放回原分离页链表，最后由 putback_inactive_pages()函数放回 LRU 链表。

对每个分离页执行以下操作：

①调用 page_check_references()函数检查页访问计数，只有访问计数为 0 的页才继续执行后面的操作，考虑回收，访问计数大于 0 的页，放入不回收页链表。

②对回收的匿名映射页调用 add_to_swap()函数添加到交换缓存，即交换文件的页缓存。

③对映射页调用 try_to_unmap()函数解除映射。

④对脏页调用 pageout()函数进行回写，其中包括将放入交换缓存的匿名映射页写入交换区。

⑤如果回收此页则放入回收页链表，否则放入不回收页链表。

（3）调用 putback_inactive_pages()函数将不回收页放回 LRU 链表。

shrink_inactive_list()函数定义如下（/mm/vmscan.c）：

```
static noinline_for_stack unsigned long shrink_inactive_list(unsigned long nr_to_scan, \
                                struct lruvec *lruvec,struct scan_control *sc, enum lru_list lru)
{
    LIST_HEAD(page_list);          /*分离页临时双链表*/
    unsigned long nr_scanned;
    unsigned long nr_reclaimed = 0;
    unsigned long nr_taken;
    unsigned long nr_dirty = 0;
    unsigned long nr_congested = 0;
```

```
unsigned long nr_unqueued_dirty = 0;
unsigned long nr_writeback = 0;
unsigned long nr_immediate = 0;
isolate_mode_t isolate_mode = 0;
int file = is_file_lru(lru);
struct zone *zone = lruvec_zone(lruvec);
struct zone_reclaim_stat *reclaim_stat = &lruvec->reclaim_stat;

while (unlikely(too_many_isolated(zone, file, sc))) { /*当前正在处理的分离页过多，/mm/vmscan.c*/
    congestion_wait(BLK_RW_ASYNC, HZ/10);       /*等待块设备不拥塞，/mm/backing-dev.c*/
    if (fatal_signal_pending(current))
        return SWAP_CLUSTER_MAX;
}       /*等待已分离页处理完成，只有直接回收才等待，页回收线程不等待*/

lru_add_drain();            /*刷出当前 CPU 核所有页向量*/

if (!sc->may_unmap)         /*只分离未映射页*/
    isolate_mode |= ISOLATE_UNMAPPED;
if (!sc->may_writepage)     /*只分离干净的文件缓存页，不发起回写*/
    isolate_mode |= ISOLATE_CLEAN;

spin_lock_irq(&zone->lru_lock);         /*持有自旋锁*/

nr_taken = isolate_lru_pages(nr_to_scan, lruvec, &page_list,&nr_scanned, sc, isolate_mode, lru);
                                        /*分离页添加到 page_list 链表，返回分离页数量*/
__mod_zone_page_state(zone, NR_LRU_BASE + lru, -nr_taken);      /*修改统计量*/
__mod_zone_page_state(zone, NR_ISOLATED_ANON + file, nr_taken);

if (global_reclaim(sc)) {       /*返回 true*/
    __mod_zone_page_state(zone, NR_PAGES_SCANNED, nr_scanned);      /*修改统计值*/
    if (current_is_kswapd())
        __count_zone_vm_events(PGSCAN_KSWAPD, zone, nr_scanned);
    else
        __count_zone_vm_events(PGSCAN_DIRECT, zone, nr_scanned);
}
spin_unlock_irq(&zone->lru_lock);       /*释放自旋锁*/

if (nr_taken == 0)      /*如果分离页数量为 0，函数返回*/
    return 0;

nr_reclaimed = shrink_page_list(&page_list, zone, sc, TTU_UNMAP,
        &nr_dirty, &nr_unqueued_dirty, &nr_congested,
        &nr_writeback, &nr_immediate,false);
                        /*处理分离页，返回回收页数量，见下文，/mm/vmscan.c*/
spin_lock_irq(&zone->lru_lock);         /*持有自旋锁*/
```

```
        reclaim_stat->recent_scanned[file] += nr_taken;     /*增加扫描页数量统计计数*/

        if (global_reclaim(sc)) {
            if (current_is_kswapd())
                __count_zone_vm_events(PGSTEAL_KSWAPD, zone,nr_reclaimed);
            else
                __count_zone_vm_events(PGSTEAL_DIRECT, zone,nr_reclaimed);
        }

        putback_inactive_pages(lruvec, &page_list);
                                        /*page_list 中不回收页放回 LRU 链表，见下文*/
        __mod_zone_page_state(zone, NR_ISOLATED_ANON + file, -nr_taken);
        spin_unlock_irq(&zone->lru_lock);        /*释放自旋锁*/

        mem_cgroup_uncharge_list(&page_list);
        free_hot_cold_page_list(&page_list, true);      /*释放引用计数值为 0 的页至伙伴系统*/

        if (nr_writeback && nr_writeback == nr_taken)          /*回收的全是脏页*/
            set_bit(ZONE_WRITEBACK, &zone->flags);     /*设置物理内存域标记*/

        if (sane_reclaim(sc)) {        /*没有选择 MEMCG 配置项，返回 true*/
            if (nr_dirty && nr_dirty == nr_congested)
                set_bit(ZONE_CONGESTED, &zone->flags);

            if (nr_unqueued_dirty == nr_taken)
                set_bit(ZONE_DIRTY, &zone->flags);

            if (nr_immediate && current_may_throttle())
                congestion_wait(BLK_RW_ASYNC, HZ/10);
        }

        if (!sc->hibernation_mode && !current_is_kswapd() &&current_may_throttle())
                wait_iff_congested(zone, BLK_RW_ASYNC, HZ/10);

        trace_mm_vmscan_lru_shrink_inactive(zone->zone_pgdat->node_id,zone_idx(zone),   \
                                nr_scanned, nr_reclaimed,sc->priority,trace_shrink_flags(file));
        return nr_reclaimed;      /*返回回收页数量*/
    }
```

shrink_inactive_list()函数调用 isolate_lru_pages()函数分离出部分页，添加到 page_list 临时双链表，然后调用 **shrink_page_list()**函数对分离的页逐页进行处理，尝试回收，回收页在 shrink_page_list()函数最后统一释放，不回收的页留在 page_list 双链表，最后由 **putback_inactive_pages()**函数放回 LRU 链表，使用计数为 0 的页将放回伙伴系统（回收）。

下面看一下 shrink_page_list()和 putback_inactive_pages()函数的定义。

●处理分离页

shrink_page_list()函数逐页处理分离页，执行回收操作，函数定义如下（/mm/vmscan.c）：

```
static unsigned long shrink_page_list(struct list_head *page_list,struct zone *zone,struct scan_control *sc,
                        enum ttu_flags ttu_flags,unsigned long *ret_nr_dirty,
                        unsigned long *ret_nr_unqueued_dirty,
                        unsigned long *ret_nr_congested,unsigned long *ret_nr_writeback,
                        unsigned long *ret_nr_immediate,bool force_reclaim)
/*
*page_list：分离页链表，
*force_reclaim：是否对分离页执行强制回收，这里为 false。
*/
{
    LIST_HEAD(ret_pages);       /*暂存不回收的页，由 shrink_inactive_list()函数处理*/
    LIST_HEAD(free_pages);      /*暂存可释放的分离页*/
    int pgactivate = 0;
    unsigned long nr_unqueued_dirty = 0;
    unsigned long nr_dirty = 0;
    unsigned long nr_congested = 0;
    unsigned long nr_reclaimed = 0;
    unsigned long nr_writeback = 0;
    unsigned long nr_immediate = 0;

    cond_resched();

    while (!list_empty(page_list)) {        /*遍历分离页链表中的页*/
        struct address_space *mapping;
        struct page *page;
        int may_enter_fs;
        enum page_references references = PAGEREF_RECLAIM_CLEAN;     /*/mm/vmscan.c*/
        bool dirty, writeback;

        cond_resched();

        page = lru_to_page(page_list);     /*从分离页链表末尾取页*/
        list_del(&page->lru);              /*分离页从链表移除*/

        if (!trylock_page(page))
            goto keep;

        VM_BUG_ON_PAGE(PageActive(page), page);      /*必须已清零 PG_active 标记位*/
        VM_BUG_ON_PAGE(page_zone(page) != zone, page);

        sc->nr_scanned++;       /*已扫描分离页数量加 1*/

        if (unlikely(!page_evictable(page)))       /*如果是不可回收页，不回收*/
            goto cull_mlocked;
```

```
    if (!sc->may_unmap && page_mapped(page))     /*如果不回收映射页*/
        goto keep_locked;

    if (page_mapped(page) || PageSwapCache(page))   /*映射页和在交换缓存中页算两页*/
        sc->nr_scanned++;

    may_enter_fs = (sc->gfp_mask & __GFP_FS) ||   \
                    (PageSwapCache(page) && (sc->gfp_mask & __GFP_IO));

    page_check_dirty_writeback(page, &dirty, &writeback);   /*/mm/vmscan.c*/
                                            /*判断页是否是脏的或正在回写*/
    if (dirty || writeback)     /*脏页或正在回写页*/
        nr_dirty++;          /*脏页计数值加 1*/

    if (dirty && !writeback)
        nr_unqueued_dirty++;     /*脏页但非回写页计数加 1*/

    mapping = page_mapping(page);
    if (((dirty || writeback) && mapping &&inode_write_congested(mapping->host)) ||   \
                                        (writeback && PageReclaim(page)))
         nr_congested++;     /*拥塞页计数加 1*/

    if (PageWriteback(page)) {     /*页正在回写*/
        if (current_is_kswapd() &&PageReclaim(page) &&
            test_bit(ZONE_WRITEBACK, &zone->flags)) {
              nr_immediate++;
              goto keep_locked;     /*不回收*/

    } else if (sane_reclaim(sc) ||!PageReclaim(page) || !may_enter_fs) {
        SetPageReclaim(page);
        nr_writeback++;
        goto keep_locked;       /*不回收*/

     } else {
        wait_on_page_writeback(page);     /*等待页回写完成*/
     }
}       /*处理正在回写页完成*/

if (!force_reclaim)     /*非强制回收*/
      references = page_check_references(page, sc);   /*检查页可回收状态，见上一小节*/

switch (references) {       /*根据 page_check_references()返回值，确定页回收状态*/
case PAGEREF_ACTIVATE:
      goto activate_locked;     /*放回活跃链表，跳转*/
case PAGEREF_KEEP:
      goto keep_locked;        /*保持在不活跃链表，跳转*/
```

```
    case PAGEREF_RECLAIM:     /*可回收页*/
    case PAGEREF_RECLAIM_CLEAN:
        ;       /*可回收页，继续往下执行*/
    }

    if (PageAnon(page) && !PageSwapCache(page)) {   /*匿名映射页且不在交换缓存中*/
        if (!(sc->gfp_mask & __GFP_IO))
            goto keep_locked;
            if (!add_to_swap(page, page_list))       /*成功函数返回 1，失败返回 0*/
                                /*匿名页添加到交换缓存，见本节下文，/mm/swap_state.c*/
                goto activate_locked;          /*添加失败，放回活跃链表*/
            may_enter_fs = 1;

        /*修改 mapping，指向交换文件的地址空间*/
        mapping = page_mapping(page);
                    /*返回交换文件地址空间（需设置 PG_swapcache 标记位），/mm/util.c*/
    }

    if (page_mapped(page) && mapping) {              /*映射页，解除映射关系*/
        switch (try_to_unmap(page, ttu_flags)) {     /*解除页映射，详见第 4 章*/
        case SWAP_FAIL:          /*解除映射失败，移动到活跃链表*/
            goto activate_locked;
        case SWAP_AGAIN:       /*只解除部分映射，保持在不活跃链表*/
            goto keep_locked;
        case SWAP_MLOCK:     /*页映射到锁定虚拟内存域，放回到合适的 LRU 链表*/
            goto cull_mlocked;
        case SWAP_SUCCESS:       /*解除映射成功*/
            ; /*继续往下运行，回收页*/
        }
    }

    if (PageDirty(page)) {       /*脏页*/
        if (page_is_file_cache(page) &&(!current_is_kswapd() ||!test_bit(ZONE_DIRTY, &zone->flags)))
        {
            /*文件缓存页且（当前不是回收线程或内存域 ZONE_DIRTY 标记为 0），暂时不回写*/
            inc_zone_page_state(page, NR_VMSCAN_IMMEDIATE);
            SetPageReclaim(page);        /*设置页 PG_reclaim 标记位*/
            goto keep_locked;     /*缓存页仍保留在不活跃链表*/
        }

        if (references == PAGEREF_RECLAIM_CLEAN)   /*原来是可回收干净页，现在弄脏了*/
            goto keep_locked;        /*留在不活跃链表*/
        if (!may_enter_fs)
            goto keep_locked;
        if (!sc->may_writepage)       /*如果本次操作不允许回写操作*/
            goto keep_locked;        /*留在不活跃链表*/
```

```
            /*尝试回写脏页，包括将匿名映射页写入交换区（交换文件）*/
            switch (pageout(page, mapping, sc)) {    /*回写脏页，见本节下文，/mm/vmscan.c*/
            case PAGE_KEEP:
                goto keep_locked;              /*保持在不活跃链表*/
            case PAGE_ACTIVATE:
                goto activate_locked;          /*移动到活跃链表*/
            case PAGE_SUCCESS:                 /*回写成功*/
                if (PageWriteback(page))
                    goto keep;
                if (PageDirty(page))
                    goto keep;
                if (!trylock_page(page))
                    goto keep;
                if (PageDirty(page) || PageWriteback(page))
                    goto keep_locked;
                mapping = page_mapping(page);
            case PAGE_CLEAN:     /*干净页，回写操作完成*/
                ; /*尝试回收页*/
            }
        }      /*脏页处理完成*/

        /*以下处理的是非脏页（干净页）*/
        if (page_has_private(page)) {
            if (!try_to_release_page(page, sc->gfp_mask))
                goto activate_locked;
            if (!mapping && page_count(page) == 1) {
                unlock_page(page);
                if (put_page_testzero(page))     /*页引用计数减 1*/
                    goto free_it;     /*如果引用计数为 0，添加到释放页链表*/
                else {
                    nr_reclaimed++;
                    continue;
                }
            }
        }
        if (!mapping || !__remove_mapping(mapping, page, true))
            goto keep_locked;

        __clear_page_locked(page);

free_it:        /*将分离页添加到释入页链表*/
        nr_reclaimed++;     /*回收页数量加 1*/
        list_add(&page->lru, &free_pages);   /*将页添加到释放页临时链表*/
        continue;
```

```
cull_mlocked:    /*锁定页，不回收，添加到 ret_pages 双链表*/
        if (PageSwapCache(page))
            try_to_free_swap(page);
        unlock_page(page);
        list_add(&page->lru, &ret_pages);
        continue;

activate_locked:        /*不回收页，放回活跃 LRU 链表*/
        if (PageSwapCache(page) && vm_swap_full())
            try_to_free_swap(page);
        VM_BUG_ON_PAGE(PageActive(page), page);
        SetPageActive(page);   /*设置活跃标记*/
        pgactivate++;
keep_locked:            /*不回收页，保持在不活跃链表*/
        unlock_page(page);
keep:
        list_add(&page->lru, &ret_pages);      /*不回页（回收失败）添加到 ret_pages 临时链表*/
        VM_BUG_ON_PAGE(PageLRU(page) || PageUnevictable(page), page);
    }    /*遍历分离页结束*/

    mem_cgroup_uncharge_list(&free_pages);
    free_hot_cold_page_list(&free_pages, true);    /*释放回收页至伙伴系统*/

    list_splice(&ret_pages, page_list);
        /*将 ret_pages 链表中页移到 page_list 双链表，由后面的 putback_inactive_pages()函数处理*/
    count_vm_events(PGACTIVATE, pgactivate);

     /*返回统计量*/
    *ret_nr_dirty += nr_dirty;
    *ret_nr_congested += nr_congested;
    *ret_nr_unqueued_dirty += nr_unqueued_dirty;
    *ret_nr_writeback += nr_writeback;
    *ret_nr_immediate += nr_immediate;
    return nr_reclaimed;      /*返回实际回收页数量*/
}
```

shrink_page_list()函数执行流程前面介绍过了，这里就不再介绍了，请读者参考上面的注释自行阅读。

总之，可回收的页在 shrink_page_list()函数最后统一释放回伙伴系统，不回收的页放回 page_list 双链表，由 putback_inactive_pages()函数放回 LRU 链表。

回收的匿名映射页由 add_to_swap()函数添加到交换缓存，try_to_unmap()函数负责解除页映射关系，pageout()函数负责写出脏页至块设备，包括将位于交换缓存的匿名映射页写出交换区，页交换的内容本节最后将会介绍。

●放回不回收页

不活跃 LRU 链表中分离出来的不回收页，最后由 putback_inactive_pages()函数放回 LRU 链表，函数定义如下（/mm/vmscan.c）：

```
static noinline_for_stack void putback_inactive_pages(struct lruvec *lruvec, struct list_head *page_list)
/*page_list：不回收页链表*/
{
    struct zone_reclaim_stat *reclaim_stat = &lruvec->reclaim_stat;
    struct zone *zone = lruvec_zone(lruvec);        /*物理内存域*/
    LIST_HEAD(pages_to_free);           /*暂存使用计数为 0 的页*/

    while (!list_empty(page_list)) {       /*扫描 page_list 链表中页*/
        struct page *page = lru_to_page(page_list);       /*从 page_list 链表末尾取页*/
        int lru;

        VM_BUG_ON_PAGE(PageLRU(page), page);
        list_del(&page->lru);                          /*页从链表移除*/
        if (unlikely(!page_evictable(page))) {       /*不可回收页*/
            spin_unlock_irq(&zone->lru_lock);
            putback_lru_page(page);            /*将页放回 LRU 链表，/mm/vmscan.c*/
            spin_lock_irq(&zone->lru_lock);
            continue;
        }

        lruvec = mem_cgroup_page_lruvec(page, zone);

        /*根据 PG_active 标记值，将页放回到活跃或不活跃链表*/
        SetPageLRU(page);
        lru = page_lru(page);
        add_page_to_lru_list(page, lruvec, lru);          /*放回 LRU 链表*/

        if (is_active_lru(lru)) {     /*如果是放回到活跃 LRU 链表*/
            int file = is_file_lru(lru);
            int numpages = hpage_nr_pages(page);
            reclaim_stat->recent_rotated[file] += numpages;     /*增加放回到活跃链表页数量*/
        }
        if (put_page_testzero(page)) {    /*如果 page 使用计数减 1 后值为 0，要释放的页*/
            __ClearPageLRU(page);     /*清标记位*/
            __ClearPageActive(page);
            del_page_from_lru_list(page, lruvec, lru);       /*从 LRU 链表移除*/

            if (unlikely(PageCompound(page))) {
                spin_unlock_irq(&zone->lru_lock);
                mem_cgroup_uncharge(page);
                (*get_compound_page_dtor(page))(page);
                spin_lock_irq(&zone->lru_lock);
            } else
                list_add(&page->lru, &pages_to_free);     /*添加到释放页临时链表*/
        }
    }    /*遍历 page_list 链表结束*/
```

```
    list_splice(&pages_to_free, page_list);
                /*将使用计数为 0 的页放回 page_list 双链表，在 shrink_inactive_list()函数最后释放*/
}
```

putback_inactive_pages()函数遍历不回收页链表，对各页根据其 PG_active 标记值，将页放回到活跃或不活跃链表，然后对页使用计数减 1，若减 1 后为 0，则放入 page_list 双链表，在 shrink_inactive_list()函数最后释放回伙伴系统。

11.4.4 收缩物理内存域

在页回收策略中，不管是由守护线程发起的页回收，还是直接页回收，都是对各物理内存域进行收缩。收缩操作包括收缩 LRU 链表和 slab 缓存。

本小节介绍收缩物理内存域 shrink_zone()函数的实现。

1 shrink_zone()

内核页回收是以物理内存域为单位的，**shrink_zone()**函数用于对单个物理内存域执行页回收操作，称为收缩内存域，函数定义如下（/mm/vmscan.c）：

```
static bool shrink_zone(struct zone *zone, struct scan_control *sc,bool is_classzone)
/*sc：指向扫描控制结构，is_classzone：参数 zone 是否是最高期望收缩内存域*/
{
    struct reclaim_state *reclaim_state = current->reclaim_state;
                                        /*收缩 slab 缓存的页数，/include/linux/swap.h*/
    unsigned long nr_reclaimed, nr_scanned;
    bool reclaimable = false;

    do {
        struct mem_cgroup *root = sc->target_mem_cgroup;
        struct mem_cgroup_reclaim_cookie   reclaim = {
            .zone = zone,                  /*物理内存域*/
            .priority = sc->priority,      /*扫描优先级*/
        };
        unsigned long zone_lru_pages = 0;
        struct mem_cgroup *memcg;

        nr_reclaimed = sc->nr_reclaimed;    /*上次操作已经回收的页数量*/
        nr_scanned = sc->nr_scanned;        /*上次操作不活跃链表扫描分离页数量*/

        memcg = mem_cgroup_iter(root, NULL, &reclaim);      /*没有选择 MEMCG，返回 NULL*/
        do {            /*没有选择 MEMCG，以下循环只运行一次*/
            unsigned long lru_pages;   /*物理内存域各 LRU 链表中页数之和*/
            unsigned long scanned;
            struct lruvec *lruvec;
            int swappiness;

            if (mem_cgroup_low(root, memcg)) {
                if (!sc->may_thrash)
```

```
                continue;
            mem_cgroup_events(memcg, MEMCG_LOW, 1);
        }

        lruvec = mem_cgroup_zone_lruvec(zone, memcg);      /*&zone->lruvec*/
        swappiness = mem_cgroup_swappiness(memcg);
                                /*返回 vm_swappiness = 60，表示将匿名映射页换出的积极程度*/
        scanned = sc->nr_scanned;

        shrink_lruvec(lruvec, swappiness, sc, &lru_pages);   /*收缩 LRU 链表，/mm/vmscan.c*/
        zone_lru_pages += lru_pages;          /*LRU 链表中页总数*/

        if (memcg && is_classzone)
            shrink_slab(sc->gfp_mask, zone_to_nid(zone),   \
                        memcg, sc->nr_scanned - scanned,lru_pages);

        if (!global_reclaim(sc) &&sc->nr_reclaimed >= sc->nr_to_reclaim) {
            mem_cgroup_iter_break(root, memcg);
            break;
        }
    } while ((memcg = mem_cgroup_iter(root, memcg, &reclaim)));

    /*收缩 slab 缓存*/
    if (global_reclaim(sc) && is_classzone)          /*只有最合适分配的内存域才收缩 slab 缓存*/
        shrink_slab(sc->gfp_mask, zone_to_nid(zone), NULL,
                sc->nr_scanned - nr_scanned,zone_lru_pages);     /*收缩 slab 缓存，/mm/vmscan.c*/

    if (reclaim_state) {
        sc->nr_reclaimed += reclaim_state->reclaimed_slab;          /*增加已回收页数量*/
        reclaim_state->reclaimed_slab = 0;
    }

    vmpressure(sc->gfp_mask, sc->target_mem_cgroup,
            sc->nr_scanned - nr_scanned,sc->nr_reclaimed - nr_reclaimed);

    if (sc->nr_reclaimed - nr_reclaimed)    /*本次操作回收了页*/
            reclaimable = true;

} while (should_continue_reclaim(zone, sc->nr_reclaimed - nr_reclaimed, \
                                 sc->nr_scanned - nr_scanned, sc));   /*是否继续回收*/

return reclaimable;    /*内存域回收了页*/
}
```

shrink_zone()函数内是一个大循环，循环体内主要是调用 **shrink_lruvec()**函数收缩 LRU 链表，调用函数 **shrink_slab()**（is_classzone 为 true 时）收缩注册了收缩器的 slab 缓存，最后调用 **should_continue_reclaim()** 函数判断是否要继续执行循环。

shrink_lruvec()函数前面介绍过了，下面介绍 shrink_slab()和 should_continue_reclaim()函数的实现。

■收缩 slab 缓存

shrink_slab()函数用于收缩注册了收缩器的 slab 缓存，以期望释放更多的物理内存。

slab 收缩器在由 shrinker 结构体表示，定义如下（/include/linux/shrinker.h）：

```
struct shrinker {
    unsigned long (*count_objects)(struct shrinker *,struct shrink_control *sc);
                                /*函数指针，函数返回 slab 缓存中可回收对象数量*/
    unsigned long (*scan_objects)(struct shrinker *,struct shrink_control *sc);
                                /*函数指针，扫描并释放对象，返回释放对象的数量*/
    int seeks;
    long batch;
    unsigned long flags;
    struct list_head list;   /*双链表成员，将 shrinker 实例添加到全局链表*/
    atomic_long_t *nr_deferred;
};
```

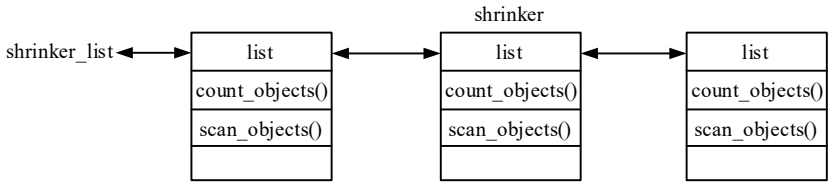
shrinker 结构体主要成员简介如下：

●**count_objects**：函数指针，函数返回 slab 缓存中可释放对象数量。

●**scan_objects**：函数指针，当 count_objects()函数返回值为非零时，调用此函数释放对象，函数返回释放对象的数量。

●**list**：双链表成员，将 shrinker 实例添加到全局链表 shrinker_list。

内核中所有 shrinker 实例添加到全局双链表 shrinker_list，如下图所示：



向内核注册 slab 收缩器的函数为 **register_shrinker**(struct shrinker *shrinker)（/mm/vmscan.c），其主要工作就是将 shrinker 实例插入 shrinker_list 双链表末尾。

内核中注册的收缩器并不多（大约 20 多处）且绝大部分是由驱动程序和文件系统类型代码注册的。内核自身注册的收缩器很少，因此这里并不打算详细介绍收缩器的细节。内核自身注册收缩器的典型例子是超级块 super_block 结构体中嵌入的收缩器（s_shrink 成员），在查找或创建超级块实例的 sget()函数中初始化（alloc_super()）并注册此收缩器。

收缩 slab 缓存的 **shrink_slab()**函数（/mm/vmscan.c）扫描 shrinker_list 双链表中的 shrinker 实例，对每个实例先调用 count_objects()函数计算可释放对象的数量，如果数量大于 0 则调用 scan_objects()函数释放对象，如果数量为 0，则跳过此收缩器进入下一收缩器，函数源代码请读者自行阅读。

■是否继续收缩

shrink_zone()函数中调用 should_continue_reclaim()函数判断是否要继续执行收缩物理内存域操作，函数定义如下（/mm/vmscan.c）：

```
static inline bool should_continue_reclaim(struct zone *zone,
                    unsigned long nr_reclaimed,unsigned long nr_scanned,struct scan_control *sc)
/*nr_reclaimed：上次操作回收的页数，nr_scanned：上次操作不活跃链表分离页数*/
```

```
{
    unsigned long pages_for_compaction;
    unsigned long inactive_lru_pages;

    if (!in_reclaim_compaction(sc))      /*没有选择 COMPACTION，返回 false，/mm/vmscan.c*/
        return false;              /*只有选择了 COMPACTION，才会进行下面的判断，否则返回 false*/

    if (sc->gfp_mask & __GFP_REPEAT) {
        if (!nr_reclaimed && !nr_scanned)
            return false;
    } else {
        if (!nr_reclaimed)
            return false;
    }

    /*
     * If we have not reclaimed enough pages for compaction and the
     * inactive lists are large enough, continue reclaiming
     */
    pages_for_compaction = (2UL << sc->order);      /*内存规整页数*/
    inactive_lru_pages = zone_page_state(zone, NR_INACTIVE_FILE);
    if (get_nr_swap_pages() > 0)
        inactive_lru_pages += zone_page_state(zone, NR_INACTIVE_ANON);
    if (sc->nr_reclaimed < pages_for_compaction &&inactive_lru_pages > pages_for_compaction)
        return true;

    /* If compaction would go ahead or the allocation would succeed, stop */
    switch (compaction_suitable(zone, sc->order, 0, 0)) {        /*内存域是否适合执行内存规整*/
    case COMPACT_PARTIAL:
    case COMPACT_CONTINUE:
        return false;      /*可以进行内存规整，返回 false，停止收缩内存域*/
    default:
        return true;       /*不进行内存规整，返回 true，继续收缩内存域*/
    }
}
```

should_continue_reclaim()函数判断如果适合执行内存规整，是则返回 true，即表示停止收缩内存域。在伙伴系统分配函数中，随后将会执行内存规模，然后再尝试分配内存，如果还分配不成功则执行直接页回收后再分配。

2 shrink_zones()

在直接页回收中将调用 shrink_zones()函数遍历借用内存域，对目标内存域及可借用内存的内存域调用函数 shrink_zone()收缩内存域。

shrink_zones()函数定义如下（/mm/vmscan.c）：

```
static bool shrink_zones(struct zonelist *zonelist, struct scan_control *sc)
{
```

```
struct zoneref *z;
struct zone *zone;
unsigned long nr_soft_reclaimed;
unsigned long nr_soft_scanned;
gfp_t orig_mask;
enum zone_type requested_highidx = gfp_zone(sc->gfp_mask);        /*最合适的分配内存域*/
bool reclaimable = false;

orig_mask = sc->gfp_mask;          /*引发页回收的分配掩码*/
if (buffer_heads_over_limit)
    sc->gfp_mask |= __GFP_HIGHMEM;

/*遍历借用内存域列表*/
for_each_zone_zonelist_nodemask(zone, z, zonelist,requested_highidx, sc->nodemask) {
    enum zone_type classzone_idx;        /*物理内存域类型*/

    if (!populated_zone(zone))          /*内存域不存在，遍历下一内存域*/
        continue;

    classzone_idx = requested_highidx;
    while (!populated_zone(zone->zone_pgdat->node_zones +classzone_idx))
        classzone_idx--;

    if (global_reclaim(sc)) {
        if (!cpuset_zone_allowed(zone,GFP_KERNEL | __GFP_HARDWALL))
            continue;

        if (sc->priority != DEF_PRIORITY &&!zone_reclaimable(zone))
            continue;       /* */
        if (IS_ENABLED(CONFIG_COMPACTION) &&
            sc->order > PAGE_ALLOC_COSTLY_ORDER &&
            zonelist_zone_idx(z) <= requested_highidx &&compaction_ready(zone, sc->order)) {
                sc->compaction_ready = true;
                continue;
        }

        nr_soft_scanned = 0;
        nr_soft_reclaimed = mem_cgroup_soft_limit_reclaim(zone,
                                    sc->order, sc->gfp_mask,&nr_soft_scanned); /*!MEMCG，返回 0*/
        sc->nr_reclaimed += nr_soft_reclaimed;
        sc->nr_scanned += nr_soft_scanned;
        if (nr_soft_reclaimed)
            reclaimable = true;
    }

    if (shrink_zone(zone, sc, zone_idx(zone) == classzone_idx))   /*收缩内存域，/mm/vmscan.c*/
```

```
                reclaimable = true;             /*回收了页*/

            if (global_reclaim(sc) &&!reclaimable && zone_reclaimable(zone))
                reclaimable = true;
        }       /*遍历借用内存域列表结束*/

        sc->gfp_mask = orig_mask;           /*原始分配掩码*/
        return    reclaimable;     /*返回是否回收了页*/
    }
```

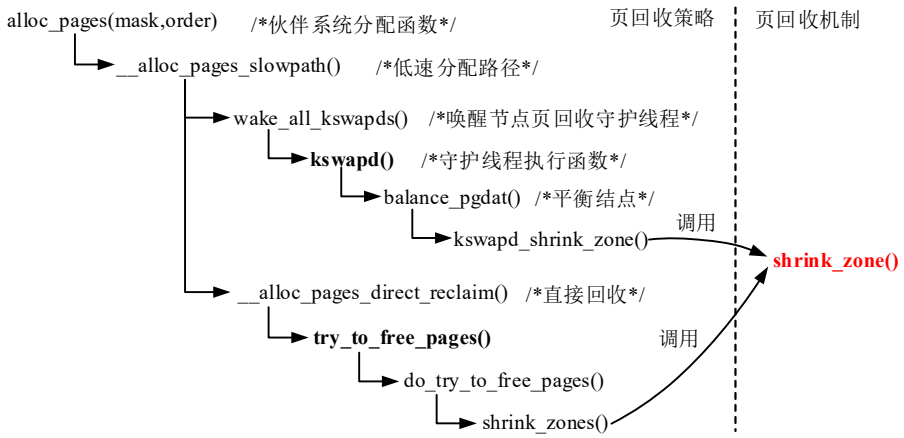
伙伴系统分配函数中，由分配掩码通过 gfp_zone(sc->gfp_mask)函数确定最适合分配的内存域。

shrink_zones()函数从最适合分配的内存域开始，对包括后面可借用内存的内存域都调用 **shrink_zone()** 函数对各内存域执行收缩操作（收缩 LRU 链表和 slab 缓存），函数返回是否回收了页。

11.4.5 页回收策略

前面介绍了页回收的机制问题，现在讨论页回收的策略问题，即什么时机和场合发起页回收，页回收的程度如何。

物理内存页都是通过伙伴系统分配的，因此页回收机制也是在页分配函数中触发的，如下图所示：



分配函数不能直接在伙伴系统获取空闲页帧时，将进入慢速路径，慢速路径中将先唤醒页回收守护线程，尝试回收页，然后再次尝试分配页，如果不成功，则执行内存规整，而后再尝试分配页，如果还不成功，将执行直接页回收。直接页回收之后若还是不能分配页，分配函数将可能启动 OOM 机制。

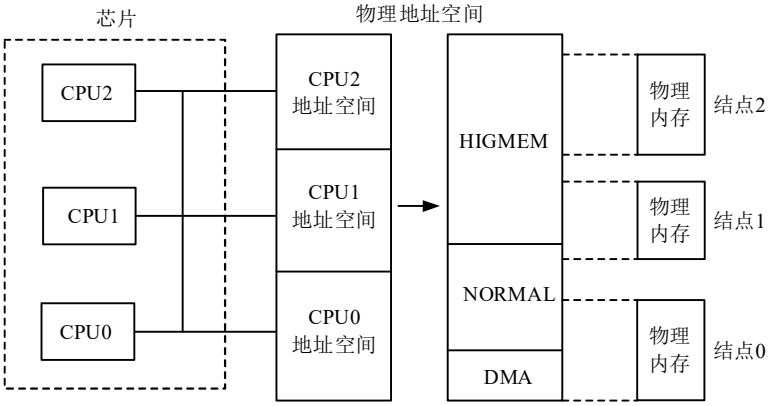
每个内存结点都对应一个页回收守护线程，分配函数会唤醒目标内存域及其下内存域所跨结点的守护线程回收页。

页回收守护线程中执行的是温和点的页回收，而直接页回收就是执行比较激进的页回收了。每个页回收守护线程只对本结点执行页回收，目标是使结点达到平衡状态，即空闲页数量达到一定要求。

直接页回收会对目标内存域及可借用内存域进行页回收，并且会触发数据回写，不断地提高优先级尝试回收页，直到回收了足够数量的页或优先级无法再提高了。

1 页回收线程

在介绍页回收守护线程前，先回顾一下物理内存域和内存结点的关系，如下图所示：



NUMA系统

在 NUMA 系统中，每个 CPU 核有段自己私有的物理地址空间，CPU 核访问这段地址空间速度较快，通过芯片内的互连网络，CPU 核也可以访问其它 CPU 核私有的地址空间，但是速度会更慢。

物理地址空间又划分成 DMA、NORMAL、HIGMEM（编号从低到高）等内存域，真正的物理内存其地址位于某一 CPU 核私有地址空间内部，CPU 核访问此内存速度较快。每个物理内存称为一个结点，结点对应 CPU 核，CPU 核访问本地结点速度较快，而访回其它结点速度较慢。分配内存时优先从本地结点分配。

结点所占的物理地址可能跨越了一个或多个物理内存域，如上图所示，结点 0 跨越了 DMA 和 NORMAL 内存域，因此在结点 pglist_data 结构体中包含一个内存域 zone 结构体数组，表示位于各内存域的物理内存信息。例如，结点 0 的 pglist_data 结构体中 zone 结构体数组的有效项数为 2，分别表示 DMA 和 NORMAL 内存域。

上图只是一个示意，并不代表某个实际的系统。UMA 系统中各 CPU 核访问所有内存的速度都是一样的，通常只有一个内存结点，因此更加简单，这里不再复述了，请读者参考第 3 章。

内核为每个内存结点创建一个守护线程，用于实现本结点的内存回收。

■创建线程

内核在启动阶段为每个物理内存结点创建守护线程用于对该结点进行页回收。

初始化函数 kswapd_init()用于创建守护线程，定义如下（/mm/vmscan.c）：

```
static int __init kswapd_init(void)
{
    int nid;

    swap_setup();        /*页交换初始化，见下一小节，/mm/swap.c*/
    for_each_node_state(nid, N_MEMORY)       /*遍历内存结点*/
        kswapd_run(nid);      /*为结点创建守护线程，/mm/vmscan.c*/
    hotcpu_notifier(cpu_callback, 0);
    return 0;
}
module_init(kswapd_init)       /*内核初始化阶段调用此函数*/
```

为指定内存结点创建守护线程的 kswapd_run(nid)函数定义在/mm/vmscan.c 文件内，代码如下：

```
int kswapd_run(int nid)
{
    pg_data_t *pgdat = NODE_DATA(nid);
    int ret = 0;
```

```
    if (pgdat->kswapd)
        return 0;

    pgdat->kswapd = kthread_run(kswapd, pgdat, "kswapd%d", nid);     /*创建内核线程，并运行*/
    ...        /*错误处理*/
    return ret;
}
```

kswapd_run(nid)函数为编号为 nid 的结点创建名称为 kswapd*nid* 的页回收守护线程，并立即唤醒运行，线程 task_struct 实例指针赋予结点 **pgdat->kswapd** 成员，线程执行函数为 **kswapd()**，下文中将介绍。

■唤醒线程

伙伴系统分配函数的低速分配路径中将调用 **wake_all_kswapds(order, ac)**函数唤醒物理内存结点的页回收守护线程，函数定义如下（/mm/page_alloc.c）：

```
static void wake_all_kswapds(unsigned int order, const struct alloc_context *ac)
/*order：分配函数传递的分配险，ac：目标内存域信息的 alloc_context 结构体指针（见第 3 章）*/
{
    struct zoneref *z;
    struct zone *zone;
    /*唤醒 ac->high_zoneidx 及以下物理内存域所对应内存结点的页回收守护线程*/
    for_each_zone_zonelist_nodemask(zone, z, ac->zonelist,ac->high_zoneidx, ac->nodemask)
        wakeup_kswapd(zone, order, zone_idx(ac->preferred_zone));     /*/mm/vmscan.c*/
}
```

wakeup_kswapd()函数用于唤醒 ac->preferred_zone 及其下内存域所对应内存结点的页回收守护线程，代码如下：

```
void wakeup_kswapd(struct zone *zone, int order, enum zone_type classzone_idx)
/*classzone_idx：最适合分配内存域编号*/
{
    pg_data_t *pgdat;

    if (!populated_zone(zone))
        return;

    if (!cpuset_zone_allowed(zone, GFP_KERNEL | __GFP_HARDWALL))
        return;
    pgdat = zone->zone_pgdat;     /*结点结构*/
    if (pgdat->kswapd_max_order < order) {     /*结点最大分配阶小于 order*/
        pgdat->kswapd_max_order = order;     /*重设 kswapd_max_order*/
        pgdat->classzone_idx = min(pgdat->classzone_idx, classzone_idx);
                                                    /*分配内存域编号最小值*/
    }
    if (!waitqueue_active(&pgdat->kswapd_wait))
        return;                /*守护线程会在 pgdat->kswapd_wait 等待队列睡眠，在睡眠则唤醒它*/
    if (zone_balanced(zone, order, 0, 0))     /*内存域已经平衡，直接返回，/mm/vmscan.c*/
```

```
        return;

    trace_mm_vmscan_wakeup_kswapd(pgdat->node_id, zone_idx(zone), order);
    wake_up_interruptible(&pgdat->kswapd_wait);     /*唤醒 pgdat->kswapd_wait 等待队列线程*/
}
```

创建页回收守护线程时，调用的是 kthread_run()函数，线程创建后将会立即唤醒运行，线程睡眠时将会添加到 pgdat->kswapd_wait 等待队列，因此唤醒线程就是调用 wake_up_interruptible(&pgdat->kswapd_wait)函数唤醒在 pgdat->kswapd_wait 队列中的睡眠等待线程（页回收线程）。

在唤醒守护线程前，会将分配函数当前申请的最大分配阶赋予 pgdat->kswapd_max_order 成员，最适合目标内存域的最小编号值赋予 pgdat->classzone_idx 成员，然后还会检查物理内存域是否平衡，如果平衡将不唤醒守护线程。内存域平衡表示指定及以上阶空闲页数量足够，zone_balanced()函数用于判断内存域是否平衡，后面将介绍此函数实现。

2 页回收线程执行函数

在介绍页回收守护线程的执行函数 kswapd()前，先简要介绍一下结点平衡和平衡阶。

分配函数由分配掩码可计算出最合适的分配内存域，设为 classzone_idx，如果此内存域没有空闲内存，则可以在其下的内存域中分配。例如，假设需要从 HIGMEM 内存域分配内存，此内存域不能分配时，可考虑从 NORMAL、DMA 内存域分配，也就是可以从比 classzone_idx 低的内存域中分配。

内存域平衡是指内存域中 order 及以上阶的空闲内存数量充足，也就是分配 order 阶内存不成问题。结点平衡是指对于 0 阶，结点中所有内存域都是平衡的，大于 0 阶时，则要求有部分内存域是平衡的即可。简单地说，判断结点是否平衡需要指定一个分配阶，这里称它为平衡阶，结点平衡表示从结点中分配平衡阶的内存没有压力。

下面来看一下页回收守护线程的执行函数 kswapd()，函数定义在/mm/vmscan.c 文件内，代码如下：

```
static int kswapd(void *p)
/*p：指向结点 pg_data_t 实例*/
{
    unsigned long order, new_order;
    unsigned balanced_order;
    int classzone_idx, new_classzone_idx;
    int balanced_classzone_idx;
    pg_data_t *pgdat = (pg_data_t*)p;
    struct task_struct *tsk = current;      /*守护线程 task_struct 指针*/

    struct reclaim_state reclaim_state = {      /*结构体中只有一个成员，/include/linux/swap.h*/
        .reclaimed_slab = 0,            /*不回收 slab 缓存*/
    };
    const struct cpumask *cpumask = cpumask_of_node(pgdat->node_id);
                                /*能访问本结点的 CPU 核掩码（默认为所有在线 CPU 核）*/

    lockdep_set_current_reclaim_state(GFP_KERNEL);

    if (!cpumask_empty(cpumask))
        set_cpus_allowed_ptr(tsk, cpumask);
    current->reclaim_state = &reclaim_state;
```

```
/*设置线程标记，用于通知伙伴系统分配器*/
tsk->flags |= PF_MEMALLOC | PF_SWAPWRITE | PF_KSWAPD;
set_freezable();

order = new_order = 0;
balanced_order = 0;
classzone_idx = new_classzone_idx = pgdat->nr_zones - 1;     /*结点所跨最高内存域编号*/
balanced_classzone_idx = classzone_idx;
/*以上代码只在线程第一次运行时才会执行*/

for ( ; ; ) {            /*for 循环，线程停止（退出）时才会跳出循环*/
    bool ret;

    /*本次新最高不平衡内存域编号不比上次高，且平衡阶相等*/
    if (balanced_classzone_idx >= new_classzone_idx &&balanced_order == new_order) {
        new_order = pgdat->kswapd_max_order;          /*唤醒线程的分配函数的分配阶*/
        new_classzone_idx = pgdat->classzone_idx;      /*平衡内存域的编号*/
        pgdat->kswapd_max_order =    0;
        pgdat->classzone_idx = pgdat->nr_zones - 1;
    }

    if (order < new_order || classzone_idx > new_classzone_idx) {
        order = new_order;                /*平衡阶取最大值*/
        classzone_idx = new_classzone_idx;          /*最高不平衡内存域取最小值*/
    } else {
        kswapd_try_to_sleep(pgdat, balanced_order,balanced_classzone_idx);
                                    /*尝试进入睡眠，唤醒后从此处开始运行，/mm/vmscan.c*/
        order = pgdat->kswapd_max_order;      /*唤醒线程时传递的参数*/
        classzone_idx = pgdat->classzone_idx;
        new_order = order;
        new_classzone_idx = classzone_idx;
        pgdat->kswapd_max_order = 0;
        pgdat->classzone_idx = pgdat->nr_zones - 1;
    }

    ret = try_to_freeze();
    if (kthread_should_stop())     /*内核线程是否要停止，停止的跳出循环*/
        break;

    if (!ret) {
        trace_mm_vmscan_kswapd_wake(pgdat->node_id, order);
        balanced_classzone_idx = classzone_idx;        /*目标内存域编号*/
        balanced_order = balance_pgdat(pgdat, order,&balanced_classzone_idx);
                                    /*平衡结点，返回平衡阶，/mm/vmscan.c*/
    }
```

```
        /*平衡执行完后（页回收完成后）继续跳转至循环开始处*/
    }   /*for 循环结束，退出循环即表示线程退出*/

    /*线程停止（退出）时执行下面代码*/
    tsk->flags &= ~(PF_MEMALLOC | PF_SWAPWRITE | PF_KSWAPD);
    current->reclaim_state = NULL;
    lockdep_clear_current_reclaim_state();

    return 0;
}
```

kswapd()函数内是一个无限循环，线程停止（退出）时跳出循环。

kswapd_try_to_sleep()函数用于判断当前线程是否要进入睡眠，若结点平衡则可以进入睡眠，否则不睡眠。守护线程睡眠时添加到结点 pgdat->kswapd_wait 等待队列，在 wake_all_kswapds()函数中会将其唤醒。

balance_pgdat()函数用于平衡结点，即通过收缩内存域释放页，以使结点空闲页数量达到平衡状态，函数返回结点平衡阶数值，返回后 balanced_classzone_idx 参数保存此次平衡前结点中最高不平衡内存域编号。

下面先看平衡结点的 balance_pgdat()函数实现，然后再介绍使线程睡眠的 kswapd_try_to_sleep()函数的实现。

■平衡结点

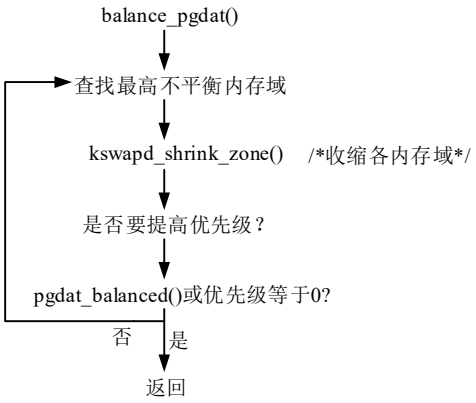
平衡结点的 balance_pgdat()函数定义如下（/mm/vmscan.c）：

```
static unsigned long balance_pgdat(pg_data_t *pgdat, int order,int *classzone_idx)
/*order：通常为 pgdat->kswapd_max_order，*classzone_idx：内存域编号*/
{
    int i;
    int end_zone = 0;          /* Inclusive.   0 = ZONE_DMA，保存结点最高不平衡内存域编号*/
    unsigned long nr_soft_reclaimed;
    unsigned long nr_soft_scanned;
    struct scan_control sc = {
        .gfp_mask = GFP_KERNEL,
        .order = order,
        .priority = DEF_PRIORITY,      /*DEF_PRIORITY=12，/include/inux/mmzone.h*/
        .may_writepage = !laptop_mode,      /*全局变量，初始值为 0，允许回写*/
        .may_unmap = 1,         /*可回收映射页*/
        .may_swap = 1,          /*允许页换出*/
    };
    count_vm_event(PAGEOUTRUN);

    do {
        unsigned long nr_attempted = 0;         /*期望回收页数量*/
        bool raise_priority = true;       /*是否需要提高优先级*/
        bool pgdat_needs_compaction = (order > 0);   /*order 大于 0 则需要内存规整*/

        sc.nr_reclaimed = 0;       /*实际回收的页数量清零*/
        for (i = pgdat->nr_zones - 1; i >= 0; i--) { /*从高至低遍历内存域，查找第一个不平衡内存域*/
```

```
        struct zone *zone = pgdat->node_zones + i;

        if (!populated_zone(zone))
            continue;

        if (sc.priority != DEF_PRIORITY &&!zone_reclaimable(zone))
            continue;

        age_active_anon(zone, &sc);   /*收缩活跃匿名映射页链表（如果需要），/mm/vmscan.c*/

        if (buffer_heads_over_limit && is_highmem_idx(i)) {
            end_zone = i;
            break;
        }

        if (!zone_balanced(zone, order, 0, 0)) {      /*找到第一个（最高）不平衡的内存域*/
            end_zone = i;       /*最高不平衡内存域编号*/
            break;              /*跳出循环*/
        } else {
            /*如果内存域平衡，清除 ZONE_CONGESTED 和 ZONE_DIRTY 标记位*/
            clear_bit(ZONE_CONGESTED, &zone->flags);
            clear_bit(ZONE_DIRTY, &zone->flags);
        }
    }   /*遍历结点内存域循环结束*/

    if (i < 0)    /*结点内所有内存域都平衡，不需要执行页回收*/
        goto out;

    /*结点内有内存域不平衡*/
    for (i = 0; i <= end_zone; i++) {      /*从低至 end_zone 遍历内存域，判断是否需要内存规整*/
        struct zone *zone = pgdat->node_zones + i;

        if (!populated_zone(zone))
            continue;
        if (pgdat_needs_compaction &&
                zone_watermark_ok(zone, order,low_wmark_pages(zone),*classzone_idx, 0))
            pgdat_needs_compaction = false;
    }

    if (sc.priority < DEF_PRIORITY - 2)
        sc.may_writepage = 1;          /*允许回写*/

    for (i = 0; i <= end_zone; i++) {      /*遍历 0 至 end_zone 内存域，收缩各内存域*/
        struct zone *zone = pgdat->node_zones + i;
        if (!populated_zone(zone))
            continue;
```

```
            if (sc.priority != DEF_PRIORITY &&!zone_reclaimable(zone))
                continue;

            sc.nr_scanned = 0;
            nr_soft_scanned = 0;
            /*是否设置了软限制？ */
            nr_soft_reclaimed = mem_cgroup_soft_limit_reclaim(zone,
                                    order, sc.gfp_mask,&nr_soft_scanned);   /*!MEMCG，返回 0*/
            sc.nr_reclaimed += nr_soft_reclaimed;

            if (kswapd_shrink_zone(zone, end_zone,&sc, &nr_attempted))     /*收缩内存域*/
                raise_priority = false;
                    /*不活跃链表扫描分离页数量不小于期望回收的页数量，不需要提高优先级*/
        }      /*遍历内存域结束*/

        /*唤醒 pgdat->pfmemalloc_wait 睡眠等待进程，如果空闲页达标*/
        if (waitqueue_active(&pgdat->pfmemalloc_wait) &&pfmemalloc_watermark_ok(pgdat))
            wake_up_all(&pgdat->pfmemalloc_wait);

        /*回收了足够的页，order 和 sc.order 置 0*/
        if (order && sc.nr_reclaimed >= 2UL << order)
            order = sc.order = 0;

        /* kswapd 是否需要冻结或停止*/
        if (try_to_freeze() || kthread_should_stop())
            break;

        /*内存规整，如果需要*/
        if (pgdat_needs_compaction && sc.nr_reclaimed > nr_attempted)
            compact_pgdat(pgdat, order);

        /*如果需要（没有回收到页），提高优先级再收缩一遍各内存域*/
        if (raise_priority || !sc.nr_reclaimed)
            sc.priority--;
    } while (sc.priority >= 1 && !pgdat_balanced(pgdat, order, *classzone_idx));
                        /*判断结点是否平衡量，是否需要提高优先级收缩内存域*/
out:
    *classzone_idx = end_zone;        /*返回最高不平衡内存域编号*/
    return order;          /*返回平衡阶*/
}
```

balance_pgdat()函数用于平衡结点，执行流程简列如下图所示：



balance_pgdat()函数内是一个大循环，循环内先查找结点中最高不平衡内存域编号赋予 end_zone 变量，然后对 0 至 end_zone 内存域分别调用 kswapd_shrink_zone()函数对每内存域进行收缩，随后根据是否回收了页等信息，判断是否要提高优先级，是则提高，否则不提高。循环退出的判断条件是结点已经平衡或优先级无法再提高了。

balance_pgdat()函数最后将最高不平衡内存域编号 end_zone 赋予 classzone_idx 参数指向变量，返回结点平衡的分配阶 order，即结点中 order 及以下阶空闲页是充足的。

下面介绍守护线程中收缩内存域的 kswapd_shrink_zone()函数，以及判断结点平衡的 pgdat_balanced()函数的实现。

●收缩内存域

kswapd_shrink_zone()函数用于在守护线程中收缩内存域，函数定义如下（/mm/vmscan.c）：

```
static bool kswapd_shrink_zone(struct zone *zone,int classzone_idx,   \
                                        struct scan_control *sc,unsigned long *nr_attempted)
/*
*zone：当前收缩内存域，classzone_idx：最高不平衡内存域编号，
* *nr_attempted：累加期望回收的页数量。
*/
{
    int testorder = sc->order;
    unsigned long balance_gap;
    bool lowmem_pressure;

    /*期望回收 zone->watermark[WMARK_HIGH]数量的页*/
    sc->nr_to_reclaim = max(SWAP_CLUSTER_MAX, high_wmark_pages(zone));

    if (IS_ENABLED(CONFIG_COMPACTION) && sc->order &&
            compaction_suitable(zone, sc->order, 0, classzone_idx)!= COMPACT_SKIPPED)
        testorder = 0;

    balance_gap = min(low_wmark_pages(zone), DIV_ROUND_UP(
            zone->managed_pages, KSWAPD_ZONE_BALANCE_GAP_RATIO));
                                        /*返回 zone->watermark[WMARK_LOW]*/

    lowmem_pressure = (buffer_heads_over_limit && is_highmem(zone));      /*高端内存*/
    if (!lowmem_pressure && zone_balanced(zone, testorder,balance_gap, classzone_idx))
        return true;                                    /*内存域平衡，返回，不收缩内存域*/
```

```
        shrink_zone(zone, sc, zone_idx(zone) == classzone_idx);   /*收缩内存域，/mm/vmscan.c*/

        /*累加期望回收的页数量*/
        *nr_attempted += sc->nr_to_reclaim;

        clear_bit(ZONE_WRITEBACK, &zone->flags);

        if (zone_reclaimable(zone) &&
            zone_balanced(zone, testorder, 0, classzone_idx)) {      /*内存域已平衡，清标记位*/
            clear_bit(ZONE_CONGESTED, &zone->flags);
            clear_bit(ZONE_DIRTY, &zone->flags);
        }

        return sc->nr_scanned >= sc->nr_to_reclaim;
                                    /*不活跃链表扫描分离页数量是否不小于期望回收的页数量*/
    }
```

kswapd_shrink_zone()函数首先确定期望回收的页数量，然后判断是否要收缩内存域，如果需要则调用函数 shrink_zone()收缩内存域，执行页回收，最后函数返回不活跃链表扫描分离页数量是否不小于期望回收的页数量，用于后面确定是否要提高扫描优先级，参数*nr_attempted 累加了期望回收的页数量。

●结点是否平衡

balance_pgdat()函数对 end_zone 及以下内存域执行收缩后，调用 pgdat_balanced()函数判断结点是否平衡，如果平衡则不需要再执行内存域收缩了，否则可能还需要执行收缩内存域。

pgdat_balanced()函数定义如下（/mm/vmscan.c）：

```
static bool pgdat_balanced(pg_data_t *pgdat, int order, int classzone_idx)
/*classzone_idx：最高内存域编号，判断 classzone_idx 及以下内存域是否平衡*/
{
    unsigned long managed_pages = 0;
    unsigned long balanced_pages = 0;
    int i;

    for (i = 0; i <= classzone_idx; i++) {     /*遍历 0 至 classzone_idx 的内存域*/
        struct zone *zone = pgdat->node_zones + i;          /*zone 实例*/

        if (!populated_zone(zone))       /*跳过结点中不存在的内存域*/
            continue;

        managed_pages += zone->managed_pages;
        if (!zone_reclaimable(zone)) {         /*/mm/vmscan.c*/
                /*目前为止页回收扫描的页数不小于可回收页数量（可回收 LRU 链表中页）的 6 倍*/
            balanced_pages += zone->managed_pages;
            continue;
        }

        if (zone_balanced(zone, order, 0, i))       /*内存域是否平衡，空闲页数量是否足够，见下文*/
```

```
                balanced_pages += zone->managed_pages;
            else if (!order)          /*所有内存域 0 阶必须平衡，否则结点不平衡*/
                return false;
        }       /*遍历 0 至 classzone_idx 的内存域结束*/

        if (order)        /*大于 0 阶*/
            return balanced_pages >= (managed_pages >> 2);
                                        /*平衡内存域页数量之和不小于所有内存域页总数的 1/4*/
        else              /*0 阶*/
            return true;
    }
```

由函数可知，对于 order=0 的情况，只有内存结点中 0 至 classzone_idx 的内存域都是平衡的，才认为结点是平衡的，否则认为是不平衡的。

对于 order>0 的情况，只需要 0 至 classzone_idx 中平衡内存域中页数量之和不小于 0 至 classzone_idx 内存域中页数量之和的 1/4（伙伴系统可管理的页），即认为结点是平衡的，而并不要求 0 至 classzone_idx 内存域都是平衡的。

zone_balanced()函数用于判断指定物理内存域是否平衡，函数定义如下（/mm/vmscan.c）：

```
static bool zone_balanced(struct zone *zone, int order,unsigned long balance_gap, int classzone_idx)
{
    if (!zone_watermark_ok_safe(zone, order, high_wmark_pages(zone) +balance_gap, classzone_idx, 0))
      /*order 及以上阶空闲页数量是否大于 high_wmark_pages(zone) +balance_gap，/mm/page_alloc.c*/
        return false;

    if (IS_ENABLED(CONFIG_COMPACTION) && order && compaction_suitable(zone,
                order, 0, classzone_idx) == COMPACT_SKIPPED)
        return false;          /*内存域不平衡*/

    return true;       /*内存域不平衡*/
}
```

zone_balanced()函数调用 zone_watermark_ok_safe()函数判断物理内存域中伙伴系统链表中 **order** 及以上阶空闲页数量之和是否大于 high_wmark_pages(zone) +balance_gap 值，high_wmark_pages(zone)函数返回 z->watermark[WMARK_HIGH]水印值，balance_gap 参数由调用者指定（如 z->watermark[WMARK_LOW]）。

zone_balanced()函数返回 true 表示内存域平衡，返回 false 表示内存域不平衡。

■尝试进入睡眠

页回收守护线程执行函数 kswapd()中调用 kswapd_try_to_sleep()函数用于尝试使守护线程进入睡眠，函数定义如下（/mm/vmscan.c）：

```
static void kswapd_try_to_sleep(pg_data_t *pgdat, int order, int classzone_idx)
/*order：结点平衡阶，classzone_idx：上次结点平衡时的最高不平衡内存域编号*/
{
    long remaining = 0;
    DEFINE_WAIT(wait);     /*添加到 pgdat->kswapd_wait 等待队列*/

    if (freezing(current) || kthread_should_stop())          /*冻结或停止线程*/
```

```
            return;

        prepare_to_wait(&pgdat->kswapd_wait, &wait, TASK_INTERRUPTIBLE);   /*/kernel/sched/wait.c*/
                    /*将守护线程以 TASK_INTERRUPTIBLE 状态添加到 pgdat->kswapd_wait 等待队列*/

        /*判断是否可进入短暂睡眠*/
        if (prepare_kswapd_sleep(pgdat, order, remaining, classzone_idx)) {
                                                /*若结点平衡，准备进入睡眠，/mm/vmscan.c*/
            remaining = schedule_timeout(HZ/10);            /*短暂睡眠*/
            finish_wait(&pgdat->kswapd_wait, &wait);            /*设为运行状态，从等待队列移除*/
            prepare_to_wait(&pgdat->kswapd_wait, &wait, TASK_INTERRUPTIBLE);   /*加入队列*/
        }

        /*判断是否要长期睡眠，是则进入睡眠，如果前面短暂睡眠被提前唤醒，则不进入长期睡眠*/
        if (prepare_kswapd_sleep(pgdat, order, remaining, classzone_idx)) {
            trace_mm_vmscan_kswapd_sleep(pgdat->node_id);
            set_pgdat_percpu_threshold(pgdat, calculate_normal_threshold);
            reset_isolation_suitable(pgdat);

            if (!kthread_should_stop())         /*不停止，进入睡眠*/
                schedule();             /*进程调度，本线程睡眠，唤醒后从此处开始往下运行*/

            set_pgdat_percpu_threshold(pgdat, calculate_pressure_threshold);       /*/mm/vmstat.c*/
        } else {            /*不睡眠*/
            if (remaining)
                count_vm_event(KSWAPD_LOW_WMARK_HIT_QUICKLY);
            else
                count_vm_event(KSWAPD_HIGH_WMARK_HIT_QUICKLY);
        }
        finish_wait(&pgdat->kswapd_wait, &wait);     /*设为运行状态，从睡眠等待队列移除*/
    }
```

kswapd_try_to_sleep()函数首先判断线程是否可进入短暂睡眠，若结点平衡则可进入短暂睡眠，线程在schedule_timeout()函数中睡眠，唤醒后再判断是否可长期睡眠（若短暂睡眠被提前唤醒，不进入长期睡眠），若结点平衡则考虑进入长期睡眠。如果结点不平衡，则任何睡眠都不会进入。

守护线程睡眠时添加到结点 pgdat->kswapd_wait 等待队列，在 wake_all_kswapds()函数中会将其唤醒。

3 直接页回收

伙伴系统分配函数在唤醒页回收守护线程后，若分配仍不成功，将执行内存规整，然后再尝试分配，分配不成功将启动直接页回收。

直接页回收函数调用关系如下图所示：

try_to_free_pages()函数是直接页回收的执行函数，函数首先判断直接页回收是否应中止，如果是则函数返回，不进行页回收，否则调用 do_try_to_free_pages()函数执行页回收。

do_try_to_free_pages()函数调用 shrink_zones()函数对目标内存域及其下内存域执行页回收，如果回收了足够数量的页，则函数返回，否则激活数据回写，降低优先级数值（提高优先级）后再次执行页回收，直到回收了足够数量的页或优先级无法再提高为止。最后，如果还是没有回收到页，则 do_try_to_free_pages()函数将判断直接页回收后是否要启动 OOM 机制。

直接页回收函数返回 0 的话，分配函数最后将启动 OOM 机制杀死一个进程，返回值大于 0 则不启动 OOM 机制。

■执行函数

try_to_free_pages()函数定义如下（/mm/vmscan.c）：

```
unsigned long try_to_free_pages(struct zonelist *zonelist, int order,gfp_t gfp_mask, nodemask_t *nodemask)
/*zonelist：借用内存域列表，nodemask：结点位图*/
{
    unsigned long nr_reclaimed;
    struct scan_control sc = {          /*扫描控制*/
        .nr_to_reclaim = SWAP_CLUSTER_MAX,    /*期望回收页数量，32，/include/linux/swap.h*/
        .gfp_mask = (gfp_mask = memalloc_noio_flags(gfp_mask)),
        .order = order,
        .nodemask = nodemask,
        .priority = DEF_PRIORITY,
        .may_writepage = !laptop_mode,    /*全局变量，默认为 0，/mm/page-writeback.c*/
        .may_unmap = 1,       /*可回收映射页*/
        .may_swap = 1,        /*允许换出匿名映射页*/
    };

    /*是否中止直接页回收*/
    if (throttle_direct_reclaim(gfp_mask, zonelist, nodemask))   /*/mm/vmscan.c*/
        return 1;    /*返回 1，分配函数将不执行 OOM 机制*/

    trace_mm_vmscan_direct_reclaim_begin(order,sc.may_writepage,gfp_mask);
    nr_reclaimed = do_try_to_free_pages(zonelist, &sc);    /*尝试释放页，/mm/vmscan.c*/
    trace_mm_vmscan_direct_reclaim_end(nr_reclaimed);
    return nr_reclaimed;        /*返回回收页数*/
}
```

try_to_free_pages()函数调用 throttle_direct_reclaim()函数判断是否要中止直接页回收，如果是则函数返回 1，否则继续调用 do_try_to_free_pages()函数尝试执行页回收，释放页。

●是否中止直接页回收

throttle_direct_reclaim()函数义在/mm/vmscan.c 文件内，返回 true 表示中止直接页回收，返回 flase 表示不中止，函数代码如下：

```
static bool throttle_direct_reclaim(gfp_t gfp_mask, struct zonelist *zonelist,nodemask_t *nodemask)
{
    struct zoneref *z;
    struct zone *zone;
    pg_data_t *pgdat = NULL;

    /*如果是内核线程调用的分配函数，返回 false，不中止直接页面回收*/
    if (current->flags & PF_KTHREAD)
        goto out;

    /*如果有挂起的致命信号，函数返回 false，不中止*/
    if (fatal_signal_pending(current))
        goto out;

    /*遍历所有借用列表中内存域*/
    for_each_zone_zonelist_nodemask(zone, z, zonelist,gfp_zone(gfp_mask), nodemask) {
        if (zone_idx(zone) > ZONE_NORMAL)     /*跳过 ZONE_NORMAL 之上的内存域*/
            continue;

        /* Throttle based on the first usable node */
        pgdat = zone->zone_pgdat;
        if (pfmemalloc_watermark_ok(pgdat))     /*ZONE_NORMAL 及以下内存域空闲页数*/
            /*空闲页超过 watermark[WMARK_MIN]值一半，不中止页回收，/mm/vmscan.c*/
            goto out;
        break;      /*空闲页小于 watermark[WMARK_MIN]值一半，当前进程在后面进入睡眠*/
    }

    /*如果没有可供分配的内存结点，返回 false，继续页回收*/
    if (!pgdat)
        goto out;

    count_vm_event(PGSCAN_DIRECT_THROTTLE);

    if (!(gfp_mask & __GFP_FS)) {          /*如果调用者不能进入块设备文件系统*/
        wait_event_interruptible_timeout(pgdat->pfmemalloc_wait,
                pfmemalloc_watermark_ok(pgdat), HZ);      /*等待空闲页足够*/

        goto check_pending;
    }
```

```
        /*将当前进程添加到 pgdat->pfmemalloc_wait 等待队列，由 kswapd 守护线程唤醒*/
        wait_event_killable(zone->zone_pgdat->pfmemalloc_wait,pfmemalloc_watermark_ok(pgdat));

    check_pending:
        if (fatal_signal_pending(current))
            return true;     /*中止直接页回收*/
    out:
        return false;          /*不中止直接页回收*/
    }
```

以上函数中，将调用 pfmemalloc_watermark_ok()函数检查第一个结点中 ZONE_NORMAL 及以下内存域中空闲页数量之和是否大于最小水印值 watermark[WMARK_MIN]之和，如果是则继续直接页回收；否则在 pfmemalloc_watermark_ok()函数唤醒页回收守护线程，当前进程（调用分配函数的进程）进入睡眠等待，随后由页回收守护线程唤醒。

●尝试释放页

如果要执行页回收，try_to_free_pages()函数调用 do_try_to_free_pages()函数执行页回收，尝试释放页，函数定义如下（/mm/vmscan.c）：

```
static unsigned long do_try_to_free_pages(struct zonelist *zonelist,struct scan_control *sc)
{
    int initial_priority = sc->priority;
    unsigned long total_scanned = 0;     /*累加扫描的页数量*/
    unsigned long writeback_threshold;
    bool zones_reclaimable;
retry:
    delayacct_freepages_start();

    if (global_reclaim(sc))
        count_vm_event(ALLOCSTALL);

    do {        /*不断减小 sc->priority 值，执行循环*/
        vmpressure_prio(sc->gfp_mask, sc->target_mem_cgroup,sc->priority);   /*!MEMCG，为空*/
        sc->nr_scanned = 0;
        zones_reclaimable = shrink_zones(zonelist, sc);     /*收缩内存域，见上文，/mm/vmscan.c*/

        total_scanned += sc->nr_scanned;     /*累加扫描的页数量*/
        if (sc->nr_reclaimed >= sc->nr_to_reclaim)     /*回收了足够数量的页，跳出循环*/
            break;

        if (sc->compaction_ready)
            break;

        /*没有回收足够的页，触发数据回写*/
        if (sc->priority < DEF_PRIORITY - 2)
            sc->may_writepage = 1;
```

```
            writeback_threshold = sc->nr_to_reclaim + sc->nr_to_reclaim / 2; /*sc->nr_to_reclaim 的 1.5 倍*/
            if (total_scanned > writeback_threshold) {
                                        /*扫描页数大于欲回收页的 1.5 倍，触发数据回写*/
                wakeup_flusher_threads(laptop_mode ? 0 : total_scanned,    \
                                WB_REASON_TRY_TO_FREE_PAGES);   /*数据回写，见本章上文*/
                sc->may_writepage = 1;
            }
        } while (--sc->priority >= 0);   /*提高优先级后（数值减小），再收缩内存域*/

        delayacct_freepages_end();

        if (sc->nr_reclaimed)     /*回收了页，返回回收页数量*/
            return sc->nr_reclaimed;

        /*以下是没有回收到页，判断直接页回收返回后，是否要启动 OOM 机制*/
        /* Aborted reclaim to try compaction? don't OOM, then */
        if (sc->compaction_ready)
            return 1;            /*返回 1，不启动 OOM 机制*/

        /* Untapped cgroup reserves?  Don't OOM, retry. */
        if (!sc->may_thrash) {
            sc->priority = initial_priority;
            sc->may_thrash = 1;
            goto retry;
        }

        /* Any of the zones still reclaimable?  Don't OOM. */
        if (zones_reclaimable)
            return 1;

        return 0;          /*要启动 OOM 要机制*/
    }
```

do_try_to_free_pages()函数内的主要工作就是调用 shrink_zones()函数收缩各内存域，如果回收了足够的页，函数返回。否则先判断是否发起数据回写，并提高优先级后再调用 shrink_zones()函数收缩内存域，如此循环，直至回收了足够的页或优先级不能再提高为止。

do_try_to_free_pages()函数最后如果还是没有回收到页，则判断是否要启动 OOM 机制，要则返回 0，否则返回 1。

11.4.6 页交换

前面介绍了页回收的机制和策略，以上还有一些问题没有介绍，就是如何将回收的匿名映射页加入交换缓存，如何将脏页（包括交换缓存中的匿名映射页和缓存页）写出至块设备，以及匿名映射页的换入。

被回收匿名映射页的换入换出称为页交换，它只是页回收过程的中一个步骤，本小节介绍页交换的实现。

1 概述

内核配置需选择 SWAP 选项，才支持交换区，方可实现页交换。

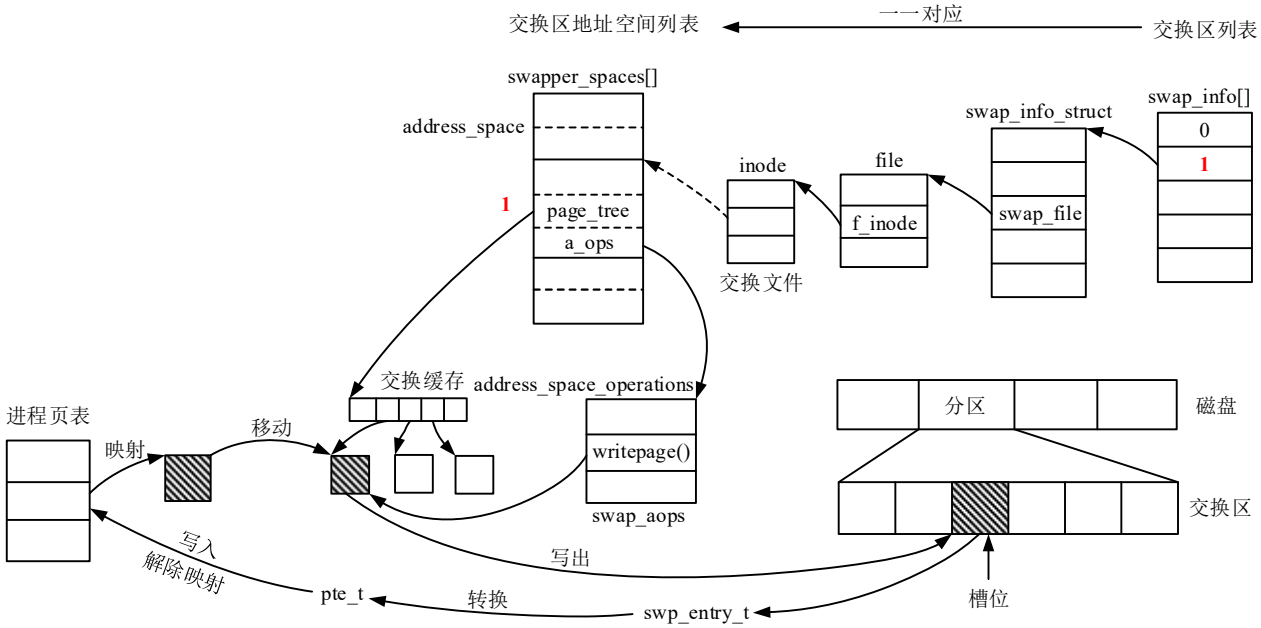
交换区是磁盘中的一个分区或文件，在内核中都有文件来表示，分区由块设备文件表示。用户需先创建交换区（指定文件名称），然后启用，页回收才能使用页交换。mkswap 命令用于创建交换区，swapon 命令用于启用交换区，swapoff 命令用于关闭交换区。

页交换机制如下图所示，内核定义了表示交换区信息的指针列表 swap_info[]，由 swap_info_struct 结构体表示交换区信息。

用户在启用交换区时，将创建 swap_info_struct 实例并关联到 swap_info[]数组项，内核设置了交换区的最大数量，即 swap_info[]数组项数。交换区是有优先级的，匿名映射页将会优先写出到优先级高的交换区。内核定义了交换区地址空间实例列表，与交换区一一对应，也就是说内核专门定义了交换区的地址空间实例，用于管理交换缓存。

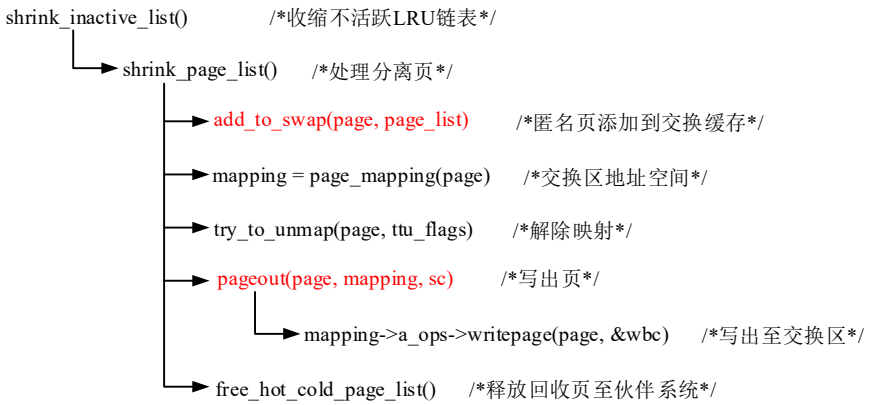
回收的匿名映射页将由内核确定添加到优先级高的交换缓存中，并执行写出操作。匿名映射页数据保存在交换区中的位置称为槽位，显然槽位的大小就是页大小。槽位就是文件内容的页偏移量。

槽位信息将写入到所有映射匿名映射页的进程页表项中。进程下次访问换出的匿名页时，将分配新页，根据页表项中的槽位信息从交换区中读回数据，重新建立映射，恢复原匿名映射页数据。



用户需要通过 mkswap 命令（程序）创建交换区，交换区信息将会写入交换区中第一页内，然后通过 swapon 命令启用交换区，只有启用了交换区，内核才会回收匿名映射页。

页回收操作中扫描不活跃链表分离出的匿名映射页，若可回收将首先调用 add_to_swap()函数添加到交换缓存（并设置脏标记），并将槽位信息写入 page 实例中 private 成员，随后调用 try_to_unmap()函数解除所有映射，将槽位信息合成 pte_t 页表项写入进程页表，调用 pageout()函数将此页从交换缓存中移除并写出至交换区，最后在 free_hot_cold_page_list()函数中释放页至伙伴系统，函数调用关系如下图所示。



当进程访问换出的匿名映射页时，在缺页异常处理函数中将调用 do_swap_page()函数，重新分配页，根据内存页表项中保存的交换区中槽位信息，从交换区中读回匿名映射页数据，重新建立映射。

2 交换区

用户需先创建交换区，然后启用，才能实现页交换。mkswap 命令用于创建交换区，swapon 命令用于启用交换区，swapoff 命令用于关闭交换区。

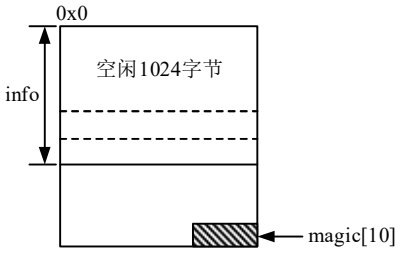
■创建交换区

交换区是用户指定的磁盘分区或文件，由用户负责创建和启用。交换区是磁盘中的分区或文件，分区用块设备文件表示，因此用户创建交换区就只需要指定文件名称。

在创建交换区时，需要获取交换区的物理信息并按指定的格式对交换区第一页写入内容。交换区中第一页的内容由内核中的 swap_header 联合体表示，定义如下（/include/linux/swap.h）：

```
union swap_header {
    struct {
        char reserved[PAGE_SIZE - 10];      /*前（PAGE_SIZE - 10）字节*/
        char magic[10];                     /*最后 10 个字节，保存 SWAPSPACE2 字符*/
    } magic;
    struct {
        char        bootbits[1024];     /*页开头空闲 1024 字节，用于保存启动装载程序等*/
        __u32           version;        /*第 1024 字节，版本号*/
        __u32           last_page;      /*交换区最后一页编号*/
        __u32           nr_badpages;    /*不可用页数量*/
        unsigned char sws_uuid[16];         /*全局唯一 ID*/
        unsigned char sws_volume[16];   /*卷名称？ */
        __u32           padding[117];
        __u32           badpages[1];    /*坏块列表，不可用页列表，可包含多项*/
    } info;
};
```

swap_header 联合体中包含 2 个结构体，但是这 2 个结构体的有效数据是错开的，如下图所示。magic 结构体表示的魔数位于页最后 10 个字节，页开头是 info 结构体，且前 1024 个字节是空闲的，后面接着的是 info 结构体中其它成员。



交换区第一页

创建交换区由用户空间命令 mkswap 完成，它只需要一个参数，即用于交换区的分区设备文件名称或普通文件名称。实用工具并不需要新系统调用的支持，现有系统调用即可完成其工作。

mkswap 命令主要完成下列工作：

●将交换区的长度除以内核页长度（PAGE_SIZE），以确定交换区能容纳多少个页。

●逐一检查交换区的各个磁盘块是否有读写错误，以确定有缺陷的区域（坏页）。

●将包含所有坏块地址（所在页编号）的列表写入到交换区的第一页（info.badpages[]数组）。

●将“SWAPSPACE2”字符写入第一页最后 10 个字节处。

●将可用槽位的数目保存在第一页中。可用槽位数量由交换区长度（info.last_page）减去交换区坏页数量（info.nr_badpages），再减 1（除去第一页）得到。

总之，mkswap 工具就是获取交换区的物理信息并按 swap_header 联合体格式写入到交换区第一页中，在启用交换区时将从第一页读取交换区信息。

●swap_info_struct

交换区在内核中由 swap_info_struct 结构体表示，内核定义了此结构体指针数组（/mm/swapfile.c）：

```
struct swap_info_struct *swap_info[MAX_SWAPFILES];        /*管理 swap_info_struct 实例*/
```

数组项数为 MAX_SWAPFILES 定义在/include/linux/swap.h 头文件，通常为 32，这表示用户能设置的交换区最大数量。

在启用交换区时，将创建 swap_info_struct 结构体实例，并从交换区第一页读取信息，初始化实例。此外，内核还定义了几个全局的交换区统计量（/mm/swapfile.c）：

```
static unsigned int nr_swapfiles;        /*启用交换区数量*/
atomic_long_t nr_swap_pages;             /*当前可用槽位总数*/
long total_swap_pages;                   /*总的槽位总数*/
static int least_priority;               /*交换区最低优先级*/
```

swap_info_struct 结构体定义在/include/linux/swap.h 头文件：

```
struct swap_info_struct {
    unsigned long    flags;          /*标记成员*/
    signed short     prio;           /*交换区优先级，数值大，优先级高*/
    struct plist_node list;          /*将实例添加到 swap_active_head 链表，/include/linux/plist.h*/
    struct plist_node avail_list;    /*将实例添加到 swap_avail_head 链表*/
    signed char      type;           /*实例关联到 swap_info[]数组项索引值*/
    unsigned int     max;            /*交换区中可用页最大值，含首页*/
    unsigned char *swap_map;         /*可用槽位引用计数，指向字符数组项对应各槽位*/
    struct swap_cluster_info *cluster_info;        /*槽位聚集信息（只用于 SSD）*/
    struct swap_cluster_info free_cluster_head;    /*空闲聚集链表头*/
    struct swap_cluster_info free_cluster_tail;    /*保存空闲聚集链表末尾节点信息*/
    unsigned int lowest_bit;         /*swap_map 中第一个空闲页索引值*/
    unsigned int highest_bit;        /*swap_map 中最后一个空闲页索引值*/
```

```
    unsigned int pages;             /*交换区实际可用槽位的数量*/
    unsigned int inuse_pages;       /*当前已使用槽位数量*/
    unsigned int cluster_next;      /*当前聚集中下一个搜索的槽位编号*/
    unsigned int cluster_nr;        /*当前聚集中可用槽位数量*/
    struct percpu_cluster __percpu *percpu_cluster;   /*CPU 核当前槽位聚集*/
    struct swap_extent *curr_swap_extent;
                        /*当前使用的 swap_extent 实例，初始值指向 first_swap_extent 成员*/
    struct swap_extent first_swap_extent;       /*swap_extent 实例链表中第一个实例*/
    struct block_device *bdev;      /*交换区所在块设备的 block_device 实例*/
    struct file *swap_file;         /*表示交换区文件*/
    unsigned int old_block_size;        /* seldom referenced */
#ifdef CONFIG_FRONTSWAP
    unsigned long *frontswap_map;    /* frontswap in-use, one bit per page */
    atomic_t frontswap_pages;       /* frontswap pages in-use counter */
#endif
    spinlock_t lock;
    struct work_struct discard_work;    /* discard worker */
    struct swap_cluster_info discard_cluster_head;    /* list head of discard clusters */
    struct swap_cluster_info discard_cluster_tail;    /* list tail of discard clusters */
};
```

swap_info_struct 结构体主要成员语义如下：

●**prio**：交换区的优先级。

●**type**：表示 swap_info_struct 实例关联到 swap_info[]指针数组的哪一项，可认为是交换区的编号。

●**max**：交换区可用槽位的最大数量，含首页。

●**swap_file**：表示交换文件或分区设备文件的 file 实例指针。

●**flags**：交换区标记成员，定义如下（/include/linux/swap.h）：

```
enum {
    SWP_USED    = (1 << 0),     /*交换区可用，分配 swap_info_struct 实例时设置，bit0*/
    SWP_WRITEOK    = (1 << 1),     /*可向交换区写出数据，启用交换区时设置*/
    SWP_DISCARDABLE = (1 << 2),     /* blkdev support discard */
    SWP_DISCARDING     = (1 << 3),     /* now discarding a free cluster */
    SWP_SOLIDSTATE     = (1 << 4),     /* blkdev seeks are cheap */
    SWP_CONTINUED      = (1 << 5),     /* swap_map has count continuation */
    SWP_BLKDEV     = (1 << 6),     /*交换区是分区*/
    SWP_FILE           = (1 << 7),     /*交换区是普通文件，且文件内容在块设备中是连续的*/
    SWP_AREA_DISCARD = (1 << 8),    /* single-time swap area discards */
    SWP_PAGE_DISCARD = (1 << 9),    /* freed swap page-cluster discards */
                /* add others here before... */
    SWP_SCANNING = (1 << 10),   /*交换区正在被扫描*/
};
```

●**list、avail_list**：plist_node 结构体成员（优先级链表节点），结构体定义如下（/include/linux/plist.h）：

```
struct plist_node {
    int             prio;           /*优先级*/
    struct list_head    prio_list;    /*链表各优先级第一个成员*/
    struct list_head    node_list;   /*添加到 plist_head 链表*/
```

};

plist_node 结构体是带优先级的双链表节点成员，链表头是 plist_head 结构体，其实就是一个普通的双链表头。

plist_node 结构体中 node_list 成员用于节点添加到 plist_head 链表，plist_node 实例按优先级顺序插入双链表，相同优先级节点按插入时间先后排序。

plist_node 结构体中 prio_list 用于链接各优先级中第一个加入的成员，也就是说相同优先级的节点只有第一个会加入 prio_list 成员组成的链表。

plist_add()函数用于将 plist_node 实例添加到 plist_head 链表。

内核在/mm/swapfile.c 内定义了全局的 plist_head 链表头：

```
PLIST_HEAD(swap_active_head);     /*管理活跃的交换区实例*/
static PLIST_HEAD(swap_avail_head);   /*管理具有可用槽位的交换区实例*/
```

swap_info_struct 结构体中 list 和 avail_list 成员分别插入到 swap_active_head 和 swap_avail_head 链表。

●**curr_swap_extent、first_swap_extent**：swap_extent 结构体指针和实例成员，curr_swap_extent 指向当前使用 swap_extent 结构体实例，first_swap_extent 表示交换区 swap_extent 实例链表中的第一个成员。

交换区不管是分区（块设备）还是普通文件，在内核中都用文件表示。内核将文件内容按页进行划分，为提高效率，写入换出页时，只考虑映射到块设备中连续数据块的页，也就是说文件内容中映射到不连续数据块的页将弃之不用。这样可以用来写入换出页，在块设备中连续的数据块（大小为一页）称为一个槽位。

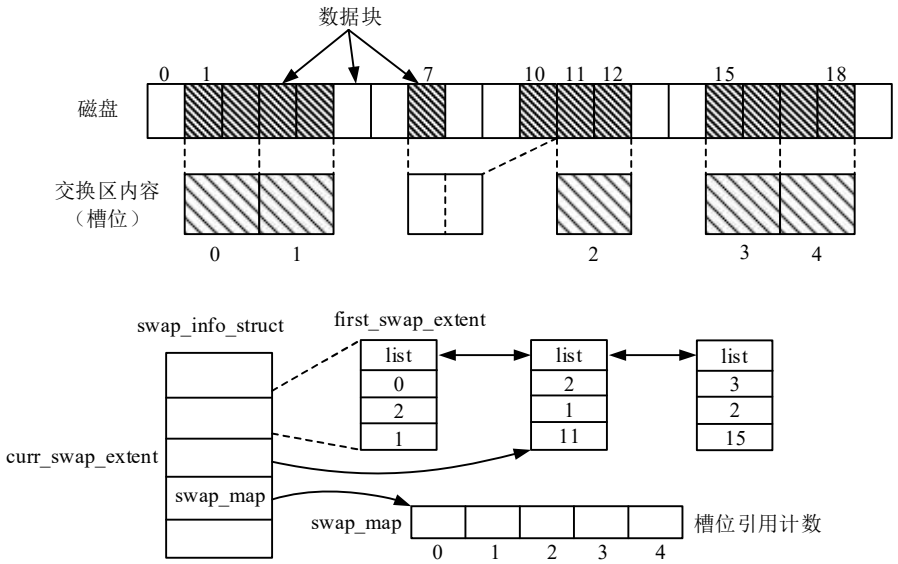
swap_extent 结构体表示交换区中一段连续可用的槽位信息，结构体定义在/include/linux/swap.h 头文件：

```
struct swap_extent {
    struct list_head list;        /*双链表成员，swap_extent 实例组成双链表*/
    pgoff_t start_page;           /*连续数据块表示的起始槽位编号*/
    pgoff_t nr_pages;             /*本连续映射区槽位数量*/
    sector_t start_block;         /*连续数据块的起始数据块号（文件系统数据块号）*/
};
```

对于分区，其内容必然是连续的，因此只需要一个 swap_extent 实例。如果是普通文件，且文件内容在块设备中也是连续的，则也只需要一个 swap_extent 实例。

下面来看一下交换区是普通文件，且文件内容不是映射到块设备中连续数据块的情形。

如下图所示，假设交换区是文件，包含 12 个数据块，两个数据块映射一页。在第一个连续的数据块中包含 4 个数据块，映射交换文件前 2 页（0、1 页），第 7、10 个数据块是不连续的，将被丢弃，不计入统计的范围内（不对其进行槽位编号），不能用于写出换出页。11-12 数据块映射第 2 页可用，数据块 15-18 映射第 3、4 页可用。在启用交换区时，会根据数据块映射关系，创建 swap_extent 实例链表。



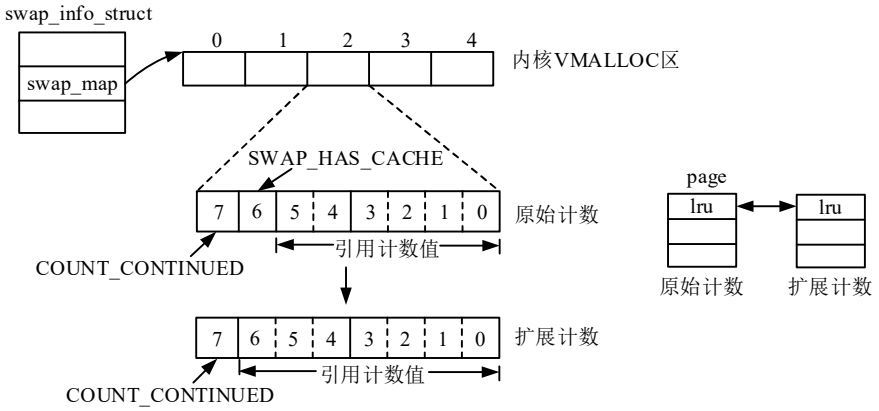
swap_info_struct 结构体中 first_swap_extent 成员表示交换区 swap_extent 实例链表中的第一个成员，指针成员 curr_swap_extent 指向当前使用的 swap_extent 结构体实例。

在启用交换区时需要获取磁盘映射信息，创建 swap_extent 结构体实例链表。如果交换区是块设备分区，由于数据块是连续的，因此只需要一个 swap_extent 结构体实例即可。如是交换区是普通文件则稍复杂些，因为文件内容映射的数据块可能是不连续的。

●**swap_map**：指向字节数组，每个数组项对应交换区中槽位的引用计数值，空闲槽位值为 0，数组项取值定义如下：

```
#define   SWAP_MAP_MAX      0x3e     /*原始计数最大值*/
#define   SWAP_MAP_BAD      0x3f     /*坏页*/
#define   SWAP_HAS_CACHE    0x40     /*槽位已被使用*/
#define   SWAP_CONT_MAX     0x7f     /*扩展计数最大值*/
#define   COUNT_CONTINUED   0x80     /*是否有扩展计数*/
#define   SWAP_MAP_SHMEM    0xbf     /* Owned by shmem/tmpfs, in first swap_map */
```

槽位引用计数值如下图所示：



保存 swap_map 指向数组的内存位于内核 VMALLOC 区（间接映射区），这个数组被称为原始计数。原始计数中 SWAP_HAS_CACHE 标记位（bit6）表示槽位已使用，暂存了匿名映射页数据，低 6 位表示计数值，最大为 SWAP_MAP_MAX（0x3e）加 1。

当计数值大于 SWAP_MAP_MAX 加 1 时，就需要使用扩展计数，COUNT_CONTINUED 标记位（bit7）表示本计数值后面有没有扩展计数，扩展计数中低 7 位用于计数值，最大值为 SWAP_CONT_MAX（0x7f），扩展计数之后还可以有扩展计数，并需要设置扩展计数的 COUNT_CONTINUED 标记位，此标记位其实是进位标记。

所有计数中的引用计数值位域组合在一起生成的数，就是真实的引用计数值。

保存扩展计数的内存也映射到内核 VMALLOC 区，原始计数值及其扩展计数值所在页 page 实例通过 lru 成员组成双链表，以便找到所有的计数值。

匿名映射页在解除映射时，每解除一个映射就会将对应槽位引用计数值加 1，每换入内存一次计数值减 1。槽位引用计数表示还有多少个进程没有恢复本匿名映射页数据，计数值为 0 时，就可以释放槽位了。

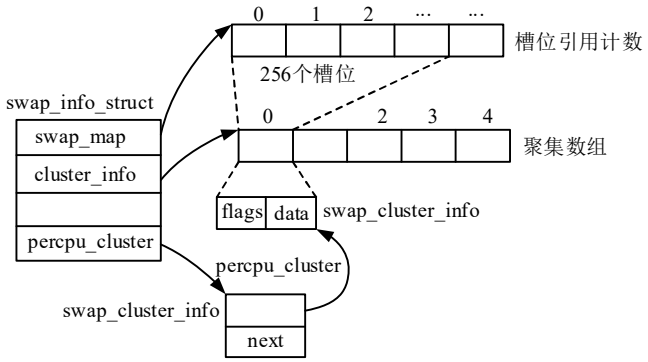
对于 SSD，还将槽位按 256（SWAPFILE_CLUSTER）个为单位进行划分，分成聚集。写出页时，先在聚集中查找槽位，以使同一 CPU 核在同一时段写出的页保存在交换区中相邻的位置，在换入页时以便执行预读，提高效率。

swap_info_struct 结构体中相关成员简介如下：

●**cluster_info**：指向 swap_cluster_info 结构体数组，表示聚集，结构体定义如下（/include/linux/swap.h）：

```
struct swap_cluster_info {
    unsigned int data:24;     /*下一个空闲聚集的编号或聚集中已使用槽位数*/
    unsigned int flags:8;      /*标记*/
};
```

swap_cluster_info 结构体数组与槽位引用计数数组对应关系如下图所示，每 256 个槽位对应一个聚集：

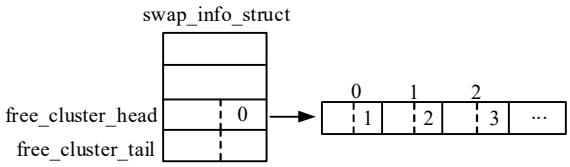


swap_cluster_info 结构体数组中的索引值就是聚集的编号。swap_cluster_info 结构体中 flags 成员取值如下：

```
#define CLUSTER_FLAG_FREE          1     /*本聚集是空闲的，也就是所有槽位都是空闲的*/
#define CLUSTER_FLAG_NEXT_NULL     2     /*本聚集之后没有聚集了*/
```

flags 成员取值为 CLUSTER_FLAG_FREE 时，data 保存下一个空闲聚集的编号，否则表示聚集中已使用槽位数量。

所有空闲聚集添加到 swap_info_struct 结构体中以 free_cluster_head（swap_cluster_info 结构体）为头的链表中，也就是前一个实例的 data 成员中保存了本聚集的编号，free_cluster_tail 成员记录了末尾聚集的编号，如下图所示（初始状态）：



当聚集不再空闲时，将从空闲链表中移出，data 成员保存已使用槽位数量。

每个 CPU 核有一个当前使用的聚集，信息保存在 swap_info_struct 结构体 percpu_cluster 成员中：

●**percpu_cluster**：指向 percpu_cluster 结构体，percpu 变量，结构体定义如下（/include/linux/swap.h）：

```
struct percpu_cluster {
    struct swap_cluster_info index;      /*当前聚集*/
    unsigned int next;                   /*聚集中下一个期望分配的槽位*/
```

};

在查找空闲槽位时，将在 CPU 核当前聚集中查找，若没有槽位则获取空闲聚集，再从中分配槽位。聚集的目的就是使 CPU 核同一时段内换出的页尽量保存在交换区中相邻的位置。

■启用交换区

用户通过 mkswap 命令创建交换区后，交换区对内核还是不可见的，因为在内核中并没有创建表示交换区的数结构实例。用户要通过 swapon 命令启用交换区，具体工作由 swapon()系统调用实现。

swapon()系统调用执行函数需要两个参数，第一个是表示交换区的文件名称，第二个是交换区标记。标记参数取值定义如下（/include/linux/swap.h）：

```
#define   SWAP_FLAG_PREFER          0x8000   /*标记中设置了交换区优先级*/
#define   SWAP_FLAG_PRIO_MASK       0x7fff   /*优先级掩码，标记低 15 位表示优先级*/
#define   SWAP_FLAG_PRIO_SHIFT      0
#define   SWAP_FLAG_DISCARD         0x10000   /* enable discard for swap */
#define   SWAP_FLAG_DISCARD_ONCE    0x20000   /* discard swap area at swapon-time */
#define   SWAP_FLAG_DISCARD_PAGES   0x40000   /* discard page-clusters after use */
```

swapon()系统调用执行函数定义如下（/mm/swapfile.c）：

```
SYSCALL_DEFINE2(swapon, const char __user *, specialfile, int, swap_flags)
/*specialfile：交换区文件名称，swap_flags：标记，低 15 位为优先级*/
{
    struct swap_info_struct *p;
    struct filename *name;
    struct file *swap_file = NULL;
    struct address_space *mapping;
    int i;
    int prio;
    int error;
    union swap_header *swap_header;
    int nr_extents;
    sector_t span;
    unsigned long maxpages;
    unsigned char *swap_map = NULL;
    struct swap_cluster_info *cluster_info = NULL;
    unsigned long *frontswap_map = NULL;
    struct page *page = NULL;
    struct inode *inode = NULL;

    if (swap_flags & ~SWAP_FLAGS_VALID)      /*没有设置标记参数，返回错误码*/
        return -EINVAL;

    if (!capable(CAP_SYS_ADMIN))        /*当前进程需要管理员权限*/
        return -EPERM;

    p = alloc_swap_info();      /*分配 swap_info_struct 实例，设置 SWP_USED 标记，/mm/swapfile.c*/
     /*分配并初始化实例，关联到第一个未使用 swap_info[]数组项，全局变量 nr_swapfiles++*/
```

```
...        /*错误处理*/
INIT_WORK(&p->discard_work, swap_discard_work);

name = getname(specialfile);     /*交换区文件名称*/
...
swap_file = file_open_name(name, O_RDWR|O_LARGEFILE, 0);     /*打开交换区文件*/
...

p->swap_file = swap_file;       /*指向交换区文件 file 实例*/
mapping = swap_file->f_mapping;     /*交换区文件 inode 实例中内嵌的地址空间*/

for (i = 0; i < nr_swapfiles; i++) {       /*避免交换区重复启用*/
      struct swap_info_struct *q = swap_info[i];
      if (q == p || !q->swap_file)
            continue;
      if (mapping == q->swap_file->f_mapping) {       /*交换区被重复启用*/
            error = -EBUSY;
            goto bad_swap;
      }
}

inode = mapping->host;     /*交换区文件 inode 实例*/
error = claim_swapfile(p, inode);    /*设置 p->bdev 成员，/mm/swapfile.c*/
...

/*读取交换区第一页信息（交换区头）*/
...
page = read_mapping_page(mapping, 0, swap_file);     /*读取交换区文件第 0 个缓存页内容*/
...
swap_header = kmap(page);     /*缓存页临时映射到内核空间*/

maxpages = read_swap_header(p, swap_header, inode);    /*/mm/swapfile.c*/
                /*读取交换区头信息，设置 swap_info_struct 实例，返回交换区最大页数*/
...
swap_map = vzalloc(maxpages);     /*为 swap_map 数组分配空间并清零（内核映射区）*/
...
if (p->bdev && blk_queue_nonrot(bdev_get_queue(p->bdev))) {
                        /*返回请求队列 QUEUE_FLAG_NONROT 标记，设置表示 SSD 设备*/
      p->flags |= SWP_SOLIDSTATE;
      p->cluster_next = 1 + (prandom_u32() % p->highest_bit);

      cluster_info = vzalloc(DIV_ROUND_UP(maxpages, \
                                    SWAPFILE_CLUSTER) * sizeof(*cluster_info));
                        /*分配 cluster_info 数组，每个聚集对应一个数组项，映射到内核空间*/
      ...
```

```
        p->percpu_cluster = alloc_percpu(struct percpu_cluster);        /*分配 percpu 变量*/
        ...
        for_each_possible_cpu(i) {
            struct percpu_cluster *cluster;
            cluster = per_cpu_ptr(p->percpu_cluster, i);
            cluster_set_null(&cluster->index);     /*初始化 p->percpu_cluster 中 cluster_info 实例*/
        }
}

error = swap_cgroup_swapon(p->type, maxpages);   /*没有选择 MEMCG_SWAP 配置项，返回 0*/
...

nr_extents = setup_swap_map_and_extents(p, swap_header, swap_map, \
                                        cluster_info, maxpages, &span);
                                        /*创建 swap_extent 实例链表，见下文*/
...

if (frontswap_enabled)       /*初值由 FRONTSWAP 配置选项确定*/
        frontswap_map = vzalloc(BITS_TO_LONGS(maxpages) * sizeof(long));
                                                        /*分配位图，一位对应一页*/

if (p->bdev &&(swap_flags & SWAP_FLAG_DISCARD) && swap_discardable(p)) {
        p->flags |= (SWP_DISCARDABLE | SWP_AREA_DISCARD |
                     SWP_PAGE_DISCARD);
        if (swap_flags & SWAP_FLAG_DISCARD_ONCE)
            p->flags &= ~SWP_PAGE_DISCARD;
        else if (swap_flags & SWAP_FLAG_DISCARD_PAGES)
            p->flags &= ~SWP_AREA_DISCARD;

        if (p->flags & SWP_AREA_DISCARD) {
            int err = discard_swap(p);
            if (unlikely(err))
                ...
        }
}       /*设置交换区标记位*/

mutex_lock(&swapon_mutex);
prio = -1;
if (swap_flags & SWAP_FLAG_PREFER)
        prio =(swap_flags & SWAP_FLAG_PRIO_MASK) >> SWAP_FLAG_PRIO_SHIFT;
                                                /*交换区优先级，swap_flags 低 15 位表示优先级*/
enable_swap_info(p, prio, swap_map, cluster_info, frontswap_map);   /*/mm/swapfile.c*/
/*设置优先级、p->swap_map = swap_map、p->cluster_info = cluster_info 等成员*/

...     /*输出信息*/
```

```
        mutex_unlock(&swapon_mutex);
        atomic_inc(&proc_poll_event);
        wake_up_interruptible(&proc_poll_wait);   /*唤醒在 proc_poll_wait 等待队列睡眠等待进程*/

        if (S_ISREG(inode->i_mode))
            inode->i_flags |= S_SWAPFILE;          /*普通文件标记为交换文件*/
        error = 0;
        goto out;       /*启用交换区成功*/

    bad_swap:
        ...         /*启用交换区失败*/
    out:            /*启用交换区成功*/
        if (page && !IS_ERR(page)) {
            kunmap(page);       /*解除第 0 个缓存页到内核空间的映射*/
            page_cache_release(page);         /*释放页*/
        }
        if (name)
            putname(name);
        if (inode && S_ISREG(inode->i_mode))
            mutex_unlock(&inode->i_mutex);
        return error;
    }
```

swapon()系统调用执行流程还是比较清晰，简列如下：

（1）为交换区分配 swap_info_struct 实例并初始化，关联到第一个未使用的 swap_info[]数组项，全局计数 nr_swapfiles 加 1。

（2）打开交换文件，获取其地址空间结构和文件 file 实例。

（3）读取交换文件第 0 个缓存页数据（交换区头数据），并映射到内核空间。

（4）由第 0 个缓存页获取交换区数据，设置 swap_info_struct 实例。

（5）计算并分配槽位引用计数数组空间，为 cluster_info 实例分配空间。由于 cluster_info 实例只用于 SSD 设备，请读者自行研究。

（6）创建交换区 swap_extent 实例链表。

（7）设置 swap_info_struct 实例标记。

（8）设置并使能交换区，解除交换区首页映射等。

下面介绍一下创建交换区 swap_extent 实例链表的 setup_swap_map_and_extents()函数以及设置并使能交换区的 enable_swap_info()函数的实现，这两个函数都定义在/mm/swapfile.c 文件内。

●创建 swap_extent 实例链表

setup_swap_map_and_extents()函数用于为交换区创建 swap_extent 实例链表，代码如下：

```
static int setup_swap_map_and_extents(struct swap_info_struct *p,union swap_header *swap_header,
                        unsigned char *swap_map,struct swap_cluster_info *cluster_info,
                        unsigned long maxpages,sector_t *span)
/*
*swap_header：交换区头联合体指针，swap_map：交换区页引用计数数组指针，
*span：跨越的数据块编号。
*/
```

```
{
    int i;
    unsigned int nr_good_pages;     /*交换区实际可用的页数*/
    int nr_extents;
    unsigned long nr_clusters = DIV_ROUND_UP(maxpages, SWAPFILE_CLUSTER);
    unsigned long idx = p->cluster_next / SWAPFILE_CLUSTER;

    nr_good_pages = maxpages - 1;    /*交换区最大页数去掉首页*/

    cluster_set_null(&p->free_cluster_head);
    cluster_set_null(&p->free_cluster_tail);
    cluster_set_null(&p->discard_cluster_head);
    cluster_set_null(&p->discard_cluster_tail);

    for (i = 0; i < swap_header->info.nr_badpages; i++) {       /*扫描坏页数组，标记坏页*/
        unsigned int page_nr = swap_header->info.badpages[i];
        if (page_nr == 0 || page_nr > swap_header->info.last_page)
            return -EINVAL;
        if (page_nr < maxpages) {
            swap_map[page_nr] = SWAP_MAP_BAD;       /*设置坏页对应的 swap_map[]数组项*/
            nr_good_pages--;              /*可用页数量减 1*/
            inc_cluster_info_page(p, cluster_info, page_nr);
        }
    }

    for (i = maxpages; i < round_up(maxpages, SWAPFILE_CLUSTER); i++)
        inc_cluster_info_page(p, cluster_info, i);       /*各聚集 data 值加 1*/

    if (nr_good_pages) {
        swap_map[0] = SWAP_MAP_BAD;     /*首页已使用，标记坏页*/
        inc_cluster_info_page(p, cluster_info, 0);
        p->max = maxpages;
        p->pages = nr_good_pages;
        nr_extents = setup_swap_extents(p, span);     /*创建 swap_extent 实例链表，/mm/swapfile.c*/
        if (nr_extents < 0)
            return nr_extents;
        nr_good_pages = p->pages;
    }
    ...

    if (!cluster_info)
        return nr_extents;

    for (i = 0; i < nr_clusters; i++) {          /*初始化聚集数组，组成空闲链表*/
        if (!cluster_count(&cluster_info[idx])) {
            cluster_set_flag(&cluster_info[idx], CLUSTER_FLAG_FREE);
```

```
                if (cluster_is_null(&p->free_cluster_head)) {
                    cluster_set_next_flag(&p->free_cluster_head,idx, 0);
                    cluster_set_next_flag(&p->free_cluster_tail,idx, 0);
                } else {
                    unsigned int tail;

                    tail = cluster_next(&p->free_cluster_tail);
                    cluster_set_next(&cluster_info[tail], idx);
                    cluster_set_next_flag(&p->free_cluster_tail,idx, 0);
                }
            }
            idx++;
            if (idx == nr_clusters)
                idx = 0;
        }
        return nr_extents;        /*返回 swap_extent 实例数量*/
    }
```

setup_swap_map_and_extents()函数首先在 swap_map[]数组中将坏页和第一页标记为坏页，因为不可用，然后调用 setup_swap_extents()函数创建交换区 swap_extent 实例链表，数定义在/mm/swapfile.c 文件内：

```
    static int setup_swap_extents(struct swap_info_struct *sis, sector_t *span)
    {
        struct file *swap_file = sis->swap_file;
        struct address_space *mapping = swap_file->f_mapping;     /*交换区文件地址空间*/
        struct inode *inode = mapping->host;
        int ret;

        if (S_ISBLK(inode->i_mode)) {            /*交换区是块设备分区*/
            ret = add_swap_extent(sis, 0, sis->max, 0);       /*/mm/swapfile.c*/
                /*添加 swap_extent 实例，通常分区只需要一个 swap_extent 实例*/
            *span = sis->pages;       /*实际可用的页数*/
            return ret;
        }

        /*如果交换区是普通文件，且定义了 a_ops->swap_activate()函数*/
        if (mapping->a_ops->swap_activate) {
            ret = mapping->a_ops->swap_activate(sis, swap_file, span);   /*地址空间操作结构定义的函数*/
            if (!ret) {            /*如果返回 0，则只需一个 swap_extent 实例*/
                sis->flags |= SWP_FILE; /*注：只有文件内容在块设备中是连续的，才设置 SWP_FILE*/
                ret = add_swap_extent(sis, 0, sis->max, 0);       /*添加一个 swap_extent 实例*/
                *span = sis->pages;
            }
            return ret;
        }
        /*如果没有定义 mapping->a_ops->swap_activate()函数，则调用通用的函数*/
        return generic_swapfile_activate(sis, swap_file, span);       /*/mm/page_io.c*/
```

}

如果交换区是分区则处理比较简单，因为分区数据块总是连续的，因此只需要一个 swap_extent 实例即可。

如果交换区是文件则稍微复杂一些，如果文件地址空间操作结构中定义了 swap_activate()函数，则由此函数构建 swap_extent 实例链表。如果 swap_activate()函数返回 0，则认为文件内容在块设备中是连续的，则只需要一个 swap_extent 实例，并设置交换区的 **SWP_FILE** 标记位。

如果普通文件其地址空间操作结构中没有定义 swap_activate()函数，则调用 generic_swapfile_activate()函数为交换区创建 swap_extent 实例链表。

generic_swapfile_activate()函数定义在/mm/page_io.c 文件内，函数内逐页扫描文件内容页，判断是否映射到连续的数据块，是则调用 add_swap_extent()函数向链表添加 swap_extent 实例，如果映射数据块不连续，则跳过此页，扫描下一页。此函数中对页（槽位）的编号跳过了映射数据块不连续的页，不是其作为文件内容的顺序编号。

add_swap_extent()函数内将考虑当前页是否可与现有的 swap_extent 实例合并（映射数据块连续），可以则合并。

generic_swapfile_activate()函数最后将可用槽位数量赋予 sis->pages 成员，sis->max 成员赋值为可用槽位数加 1（含首页），*span 返回可用页跨越的数据块编号区间。

●使能交换区

enable_swap_info()函数用于设置/使能 swap_info_struct 实例，函数定义如下（/mm/swapfile.c）：

```
static void enable_swap_info(struct swap_info_struct *p, int prio,unsigned char *swap_map,  \
                                struct swap_cluster_info *cluster_info,unsigned long *frontswap_map)
{
    frontswap_init(p->type, frontswap_map);
    spin_lock(&swap_lock);
    spin_lock(&p->lock);
    _enable_swap_info(p, prio, swap_map, cluster_info);      /*使能交换区，/mm/swapfile.c*/
    spin_unlock(&p->lock);
    spin_unlock(&swap_lock);
}
```

_enable_swap_info()函数定义如下：

```
static void _enable_swap_info(struct swap_info_struct *p, int prio, \
                    unsigned char *swap_map,struct swap_cluster_info *cluster_info)
{
    if (prio >= 0)      /*设置优先级*/
        p->prio = prio;
    else                /*使用默认值*/
        p->prio = --least_priority;

    p->list.prio = -p->prio;          /*优先级取反，负数*/
    p->avail_list.prio = -p->prio;
    p->swap_map = swap_map;           /*指向槽位引用计数数组*/
    p->cluster_info = cluster_info;       /*指向聚集数组*/
    p->flags |= SWP_WRITEOK;          /*可写标记*/
    atomic_long_add(p->pages, &nr_swap_pages);   /*增加系统实际可用交换页数量*/
    total_swap_pages += p->pages;      /*增加总的交换区槽位计数*/
```

```
    assert_spin_locked(&swap_lock);
    plist_add(&p->list, &swap_active_head);    /*将实例添加到全局活跃交换区链表*/
    spin_lock(&swap_avail_lock);
    plist_add(&p->avail_list, &swap_avail_head);    /*将实例添加到全局可用交换区链表*/
    spin_unlock(&swap_avail_lock);
}
```

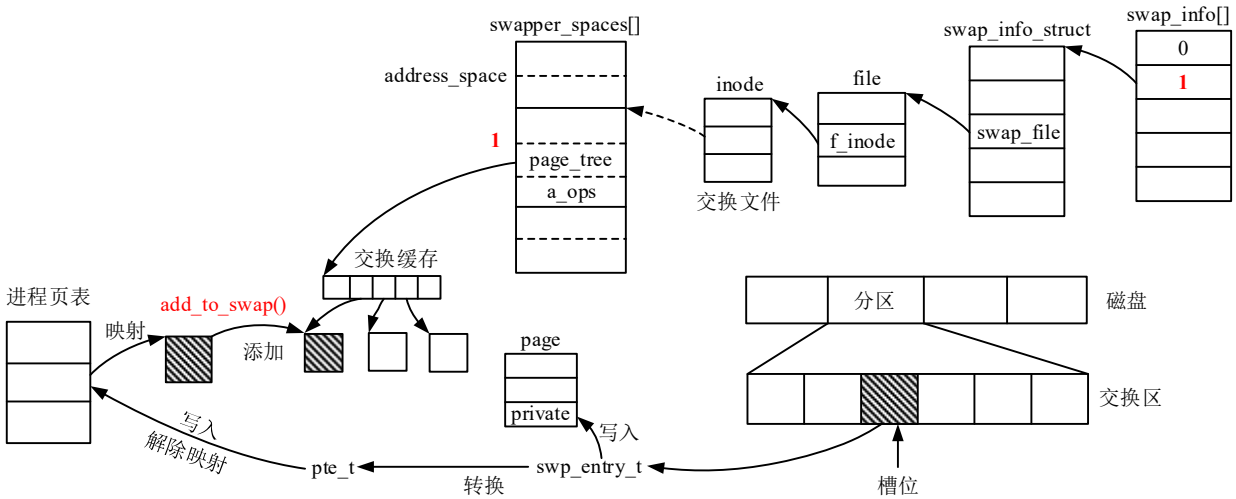
至此，启用交换区介绍完了。swapoff 命令用于关闭交换区，命令由 swapoff()系统调用实现，源代码请读者自行阅读（/mm/swapfile.c）。

3 交换缓存

启用交换区后，页回收操作就可以将回收的匿名映射页写入交换区，回收匿名映射页了。回收匿名映射页需要先选择交换区并记录槽位信息，将匿名映射页移入交换缓存，然后解除映射，将匿名映射页数据写入交换区，最后就可以释放页了。

■概述

内核为每个交换文件定义了专门的地址空间实例，由 swapper_spaces[]数组表示，与交换区一一对应。此地址空间中的页缓存，称它为交换缓存，回收的匿名映射页先移入交换缓存，然后被写入交换区，如下图所示。



内核为换出的匿名映射页选择槽位时，按优先级从高到低遍历交换区，选择第一个可用的槽位。在选择槽位后，需要将槽位信息记录下来，否则下次访问时就找不回数据了。swp_entry_t 结构体用于表示槽位信息，其中包含交换区的编号和交换区内槽位的编号。swp_entry_t 实例写入换出页 page 实例 private 成员。

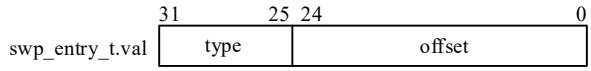
在收缩不活跃 LRU 链表函数中，对回收的匿名映射页将调用 **add_to_swap()**函数，为其选择槽位并记录槽位信息，并将页移入交换缓存。

■分配槽位

swp_entry_t 结构体用于表示槽位信息，定义如下（/include/linux/mm_types.h）：

```
typedef struct {
    unsigned long val;
} swp_entry_t;
```

swp_entry_t 结构体中只包含一个无符号整数成员 val，其布局分如下图所示（/include/linux/swapops.h）：



val 成员高 7 位 type 值用于表示交换区的索引值（swap_info[]数组中索引值），低位 offset（右对齐）表示交换区中槽位的编号，也是交换缓存基数树中匿名映射页的索引值。

swp_entry_t 结构体 val 成员不能直接进行操作，只能通过内核提供的标准函数进行操作，例如：

●swp_entry_t swp_entry(unsigned long type, pgoff_t offset)：由交换区索引值和槽位编号合成 swp_entry_t 实例。

●unsigned swp_type(swp_entry_t entry)：由 swp_entry_t 实例获取交换区索引值。

●pgoff_t swp_offset(swp_entry_t entry)：由 swp_entry_t 实例获取交换区槽位值。

get_swap_page()函数用于在交换区中查找可用槽位，返回 swp_entry_t 实例，定义如下（/mm/swapfile.c）：

```
swp_entry_t get_swap_page(void)
{
    struct swap_info_struct *si, *next;
    pgoff_t offset;

    if (atomic_long_read(&nr_swap_pages) <= 0)
        goto noswap;
    atomic_long_dec(&nr_swap_pages);      /*可用槽位计数减 1*/

    spin_lock(&swap_avail_lock);

start_over:
    plist_for_each_entry_safe(si, next, &swap_avail_head, avail_list) {   /*遍历可用交换区链表*/
        plist_requeue(&si->avail_list, &swap_avail_head);       /*重新插入链表*/
        spin_unlock(&swap_avail_lock);
        spin_lock(&si->lock);
        if (!si->highest_bit || !(si->flags & SWP_WRITEOK)) {       /*没有可用槽位*/
            spin_lock(&swap_avail_lock);
            if (plist_node_empty(&si->avail_list)) {
                spin_unlock(&si->lock);
                goto nextsi;
            }
            ...
            plist_del(&si->avail_list, &swap_avail_head); /*交换区从可用链表中移除*/
            spin_unlock(&si->lock);
            goto nextsi;
        }

        /*在交换区中分配槽位，由于交换区在链表中是按优先级排列的，因此优先级高的先分配*/
        offset = scan_swap_map(si, SWAP_HAS_CACHE);      /*/mm/swapfile.c*/
        /*查找交换区中第一个可用的槽位，并设置对应 swap_map[]数组项为 SWAP_HAS_CACHE*/
        spin_unlock(&si->lock);
```

```
            if (offset)
                return swp_entry(si->type, offset);      /*返回生成的 swp_entry_t 实例*/
            pr_debug("scan_swap_map of si %d failed to find offset\n",si->type);
            spin_lock(&swap_avail_lock);
nextsi:
            if (plist_node_empty(&next->avail_list))
                goto start_over;
        }

        spin_unlock(&swap_avail_lock);
        atomic_long_inc(&nr_swap_pages);
noswap:
        return (swp_entry_t) {0};          /*没有可用槽位，返回 0*/
}
```

get_swap_page()函数扫描可用交换区链表，在第一个具有可用槽位交换区中分配槽位，由于交换区在链表中按优先级从高到低排列，因此优先从高优先级的交换区中分配。

选中交换区后，scan_swap_map()函数用于在交换区中查找一个未使用的槽位（swap_map[]数组项值为0），并设置 swap_map[]数组项值为 SWAP_HAS_CACHE，修改交换区结构中相应成员值，函数最后返回由交换区编号和槽位编号生成的 swp_entry_t 实例。

如果交换区中有聚集数组，scan_swap_map()函数中将在聚集中查找槽位，源代码请读者自行阅读。

■添加页至交换缓存

内核定义了地址空间 address_space 实例数组 swapper_spaces[**MAX_SWAPFILES**]，它与交换区一一对应，表示交换文件的地址空间。回收匿名映射页时，页将移动到交换区对应的 swapper_spaces[]数组项的地址空间交换缓存中。

swapper_spaces[**MAX_SWAPFILES**]数组定义如下（/mm/swap_state.c）：

```
struct address_space swapper_spaces[MAX_SWAPFILES] = {
    [0 ... MAX_SWAPFILES - 1] = {
        .page_tree      = RADIX_TREE_INIT(GFP_ATOMIC|__GFP_NOWARN),
        .i_mmap_writable = ATOMIC_INIT(0),
        .a_ops          = &swap_aops,     /*地址空间操作结构实例*/
    }
};
```

内核为交换缓存地址空间定义了专用的地址空间操作结构实例 **swap_aops**，如下所示：

```
static const struct address_space_operations swap_aops = {
    .writepage      = swap_writepage,              /*写出页函数，见下文，/mm/page_io.c*/
    .set_page_dirty     = swap_set_page_dirty,   /*设置页脏标记，/mm/page_io.c*/
#ifdef CONFIG_MIGRATION
    .migratepage    = migrate_page,
#endif
};
```

在初始化函数 kswapd_init()中将调用 swap_setup()函数（/mm/swap.c）初始化 swapper_spaces[].tree_lock 各自旋锁，以及对全局变量 page_cluster 赋值，源代码请读者自行阅读。

将匿名映射页移动到交换缓存的 **add_to_swap()**函数定义在/mm/swap_state.c 文件内，代码如下：

```
int add_to_swap(struct page *page, struct list_head *list)
{
    swp_entry_t entry;
    int err;

    VM_BUG_ON_PAGE(!PageLocked(page), page);
    VM_BUG_ON_PAGE(!PageUptodate(page), page);

    entry = get_swap_page();     /*获取交换区槽位，返回 swp_entry_t 实例，/mm/swapfile.c*/
    if (!entry.val)
        return 0;

    if (unlikely(PageTransHuge(page)))
        if (unlikely(split_huge_page_to_list(page, list))) {
            swapcache_free(entry);
            return 0;
        }

    /*将匿名映射页添加到交换缓存*/
    err = add_to_swap_cache(page, entry,__GFP_HIGH|__GFP_NOMEMALLOC|__GFP_NOWARN);
                                                    /*注意分配标记，/mm/swap_state.c*/

    if (!err) {         /*添加匿名页成功*/
        SetPageDirty(page);      /*设置页 page 脏标记，以便在 pageout()函数中写出*/
        return 1;       /*成功返回 1*/
    } else {            /*添加失败*/
        swapcache_free(entry);
        return 0;       /*失败返回 0 */
    }
}
```

add_to_swap()函数主要工作如下：

（1）调用 get_swap_page()函数在系统交换区中寻找一个空闲的槽位，并返回表示槽位信息的 swp_entry_t 实例。

（2）调用 add_to_swap_cache()函数将匿名映射页移动到交换区对应的交换缓存中，并将 swp_entry_t 实例值写入到 page 实例 private 成员，并设置 page 实例 PG_swapcache 标记位等。

（3）设置 page 实例脏标记，以便写出。

●移动到交换缓存

add_to_swap_cache()函数用于将指定页移动到交换缓存，函数定义如下（/mm/swap_state.c）：

```
int add_to_swap_cache(struct page *page, swp_entry_t entry, gfp_t gfp_mask)
{
    int error;

    error = radix_tree_maybe_preload(gfp_mask);         /*基数树中可能要分配节点*/
```

```
    if (!error) {
        error = __add_to_swap_cache(page, entry);    /*/mm/swap_state.c*/
        radix_tree_preload_end();
    }
    return error;
}
```

__add_to_swap_cache()函数完成将匿名映射页添加到交换缓存的工作，定义如下：

```
int __add_to_swap_cache(struct page *page, swp_entry_t entry)
{
    int error;
    struct address_space *address_space;

    VM_BUG_ON_PAGE(!PageLocked(page), page);
    VM_BUG_ON_PAGE(PageSwapCache(page), page);
    VM_BUG_ON_PAGE(!PageSwapBacked(page), page);

    page_cache_get(page);       /*增加页引用计数*/
    SetPageSwapCache(page);
                /*设置 PG_swapcache 标记位，以使 page_mapping(page)函数返回交换缓存地址空间*/
    set_page_private(page, entry.val);     /*swp_entry_t 实例值写入 private 成员*/

    address_space = swap_address_space(entry);          /*返回交换缓存地址空间*/
                        /*&swapper_spaces[swp_type(entry)]，/include/linux/swap.h*/
    spin_lock_irq(&address_space->tree_lock);
    error = radix_tree_insert(&address_space->page_tree,entry.val, page);
                                                        /*将页添加到交换缓存基数树*/
    if (likely(!error)) {           /*添加成功*/
        address_space->nrpages++;
        __inc_zone_page_state(page, NR_FILE_PAGES);
        INC_CACHE_INFO(add_total);
    }
    spin_unlock_irq(&address_space->tree_lock);

    if (unlikely(error)) {        /*如果添加失败*/
        ...
    }
    return error;        /*在功返回 0*/
}
```

添加匿名映射页至交换缓存的工作比较简单，添加前将设置 page 实例的 PG_swapcache 标记位，并将 swp_entry_t 实例值写入 private 成员，随后将匿名映射页添加到交换缓存地址空间的基树数，swp_entry_t 实例中包含了匿名映射页在基数树中的索引值。

4 写出页

在收缩不活跃 LRU 链表函数中，对回收的匿名映射页调用 **add_to_swap()**函数添加到交换缓存后，将解除页映射，然后调用 pageout()将页数据写出交换区，最后就可以释放页了。

回收匿名映射页的处理代码简列如下：

```
static unsigned long shrink_page_list()          /*处理从不活跃 LRU 链表分离出的页*/
{
    ...
    if (!add_to_swap(page, page_list))          /*匿名映射页添加到交换缓存*/
        ...
    mapping = page_mapping(page);      /*指向交换缓存地址空间*/

     if (page_mapped(page) && mapping) {
        switch (try_to_unmap(page, ttu_flags)) {     /*解除页所有映射关系*/
          ...
        }
     }
     ...
     if (PageDirty(page)) {       /*脏页*/
        ...
        switch (pageout(page, mapping, sc)) {   /*写出页，mapping 指向交换缓存地址空间*/
        ...
        }
       ...
     }
     ...
}
```

解除页映射的 try_to_unmap()函数在第 4 章介绍过了，这里需要注意的是，在解除映射时，需要将 page 实例 private 成员中保存的 swp_entry_t 实例，转换成内存页表项，写入页所有映射页表项，这到后面介绍换入页时再介绍。

这里先介绍写出回收页的 **pageout()**函数的实现，此函数不仅用于写出匿名映射页至交换区，也用于将回收的脏文件缓存页写出块设备。

■写出函数

pageout()函数返回值为枚举类型，定义如下（/mm/vmscan.c）：

```
typedef enum {
    PAGE_KEEP,          /*页被锁定，回写页失败*/
    PAGE_ACTIVATE,    /*页被锁定，页需移动到活跃链表*/
    PAGE_SUCCESS,     /*回写成功，未锁定*/
    PAGE_CLEAN,         /*页是干净的，但被锁定*/
} pageout_t;
```

pageout()函数定义如下（/mm/vmscan.c）：

```
static pageout_t pageout(struct page *page, struct address_space *mapping,struct scan_control *sc)
```

```
{
    if (!is_page_cache_freeable(page))     /*页是否可释放（没有被使用），/mm/vmscan.c*/
        return PAGE_KEEP;
    if (!mapping) {     /*地址空间指针为 NULL*/
        if (page_has_private(page)) {
            if (try_to_free_buffers(page)) {        /*释放块缓存头*/
                ClearPageDirty(page);
                pr_info("%s: orphaned page\n", __func__);
                return PAGE_CLEAN;
            }
        }
        return PAGE_KEEP;
    }

    /*地址空间指针不为 NULL，通常是这种情况，解除映射时并不会清 page->mapping*/
    if (mapping->a_ops->writepage == NULL)      /*地址空间操作结构中没有定义写缓存页操作*/
        return PAGE_ACTIVATE;                   /*页移动到活跃链表*/
    if (!may_write_to_inode(mapping->host, sc))      /*如果 inode 不可写，/mm/vmscan.c*/
        return PAGE_KEEP;       /*保留在不活跃链表*/

    if (clear_page_dirty_for_io(page)) {     /*清脏标记，准备回写，/mm/page-writeback.c*/
        int res;
        struct writeback_control wbc = {        /*回写控制*/
            .sync_mode = WB_SYNC_NONE,          /*异常写*/
            .nr_to_write = SWAP_CLUSTER_MAX,
            .range_start = 0,
            .range_end = LLONG_MAX,
            .for_reclaim = 1,
        };

        SetPageReclaim(page);     /*设置页回收标记 PG_reclaim*/
        res = mapping->a_ops->writepage(page, &wbc);     /*写缓存页（写单页）*/
        if (res < 0)
            handle_write_error(mapping, page, res);
        if (res == AOP_WRITEPAGE_ACTIVATE) {
            ClearPageReclaim(page);
            return PAGE_ACTIVATE;
        }

        if (!PageWriteback(page)) {      /*回写标记位没有置位，表示回写完成*/
            ClearPageReclaim(page);      /*清回收标记*/
        }
        trace_mm_vmscan_writepage(page, trace_reclaim_flags(page));
        inc_zone_page_state(page, NR_VMSCAN_WRITE);     /*页回收回写页数统计*/
        return PAGE_SUCCESS;     /*回写成功*/
    }
```

```
        return PAGE_CLEAN;   /*页干净，但被锁定*/
    }
```

pageout()函数主要工作就是调用 mapping->a_ops->writepage(page, &wbc)函数回写单个缓存页，如果是文件缓存页，调用的是普通文件地址空间操作结构中的函数 writepage()。

如果是匿名映射页，page_mapping(page)函数返回的交换缓存地址空间，即 swapper_spaces[]数组中的地址空间，地址空间关联的 address_space_operations 实例为 swap_aops，其 writepage()函数如下：

```
    static const struct address_space_operations swap_aops = {
        .writepage      = swap_writepage,        /*写出交换缓存页函数，见下文，/mm/page_io.c*/
        ...
    };
```

●写出匿名映射页

交换缓存地址空间操作结构中写缓存页的函数为 swap_writepage()，用于将匿名映射页写出到交换区，函数定义在/mm/page_io.c 文件内，代码如下：

```
    int swap_writepage(struct page *page, struct writeback_control *wbc)
    {
        int ret = 0;

        if (try_to_free_swap(page)) {     /*页是否可从交换缓存移除，0 不可以，1 可以，/mm/swapfile.c*/
            unlock_page(page);
            goto out;            /*直接从交换缓存移除，不需要回写，可以释放页*/
        }
        if (frontswap_store(page) == 0) {
            set_page_writeback(page);
            unlock_page(page);
            end_page_writeback(page);
            goto out;
        }
        ret = __swap_writepage(page, wbc, end_swap_bio_write);     /*需要写出页，/mm/page_io.c*/
    out:
        return ret;
    }
```

swap_writepage()函数先调用 try_to_free_swap(page)函数判断是否可将页从交换缓存中移除，可以则移除，页可以直接释放了，不可移除则要进行后面的写出操作。

swap_writepage()函数随后调用__swap_writepage()函数执行页写出操作，__swap_writepage()函数定义如下（/mm/page_io.c）：

```
    int __swap_writepage(struct page *page, struct writeback_control *wbc,   \
                                                void (*end_write_func)(struct bio *, int))
    /*end_write_func：回写操作完成的回调函数，用于交换区是分区的情况*/
    {
        struct bio *bio;
        int ret, rw = WRITE;
        struct swap_info_struct *sis = page_swap_info(page);   /*交换区结构实例*/

        if (sis->flags & SWP_FILE) {      /*交换区是普通文件，且文件内容在块设备中是连续的*/
```

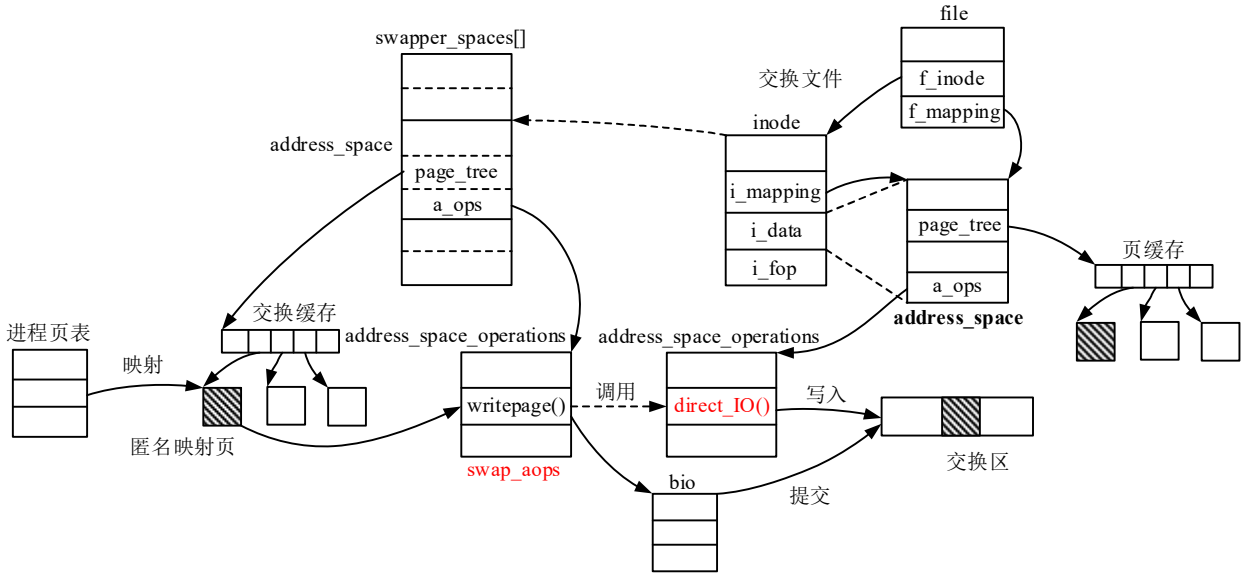
```
        struct kiocb kiocb;
        struct file *swap_file = sis->swap_file;        /*交换文件*/
        struct address_space *mapping = swap_file->f_mapping;     /*交换文件地址空间*/
        struct bio_vec bv = {     /*交换缓存页对应连续的磁盘数据块，因此只需要一个 bio_vec 实例*/
            .bv_page = page,
            .bv_len    = PAGE_SIZE,        /*大小为一页*/
            .bv_offset = 0
        };
        struct iov_iter from;

        iov_iter_bvec(&from, ITER_BVEC | WRITE, &bv, 1, PAGE_SIZE);
        init_sync_kiocb(&kiocb, swap_file);
        kiocb.ki_pos = page_file_offset(page); /*文件内容连续，所有文件内容可用（有槽位）*/
                                               /*由 private 成员中保存的槽位编号计算文件内容位置*/
        set_page_writeback(page);     /*设置 PG_writeback 标记位，回写结束后清除*/
        unlock_page(page);
        ret = mapping->a_ops->direct_IO(&kiocb, &from, kiocb.ki_pos);
                                               /*调用交换区文件地址空间操作结构中直接写函数*/
        if (ret == PAGE_SIZE) {        /*写出成功*/
            count_vm_event(PSWPOUT);
            ret = 0;
        } else {            /*写出失败*/
            set_page_dirty(page);
            ClearPageReclaim(page);
            pr_err_ratelimited("Write error on dio swapfile (%Lu)\n",page_file_offset(page));
        }
        end_page_writeback(page);     /*清页面回收、回写标记位，唤醒等待回写完成的进程*/
        return ret;
}

/*交换区是分区，或是映射不连续的普通文件*/
ret = bdev_write_page(sis->bdev, swap_page_sector(page), page, wbc);    /*/fs/block_dev.c*/
            /*调用 block_device_operations 实例中 rw_page()函数执行回写，成功返回 0*/
            /*没有定义 rw_page()函数，或写出失败，返回错误码*/

if (!ret) {       /*定义了 rw_page()函数，且写出成功*/
    count_vm_event(PSWPOUT);
    return 0;
}

/*没有定义 rw_page()函数，或写出失败，构建 bio 实例，执行写出操作*/
ret = 0;
bio = get_swap_bio(GFP_NOIO, page, end_write_func);
            /*构建 bio 实例，end_write_func 赋予 bi_end_io 指针成员（回调函数），/fs/block_dev.c*/
...
if (wbc->sync_mode == WB_SYNC_ALL)
```

```
            rw |= REQ_SYNC;
        count_vm_event(PSWPOUT);
        set_page_writeback(page);     /*bio 回调函数中清回写标记*/
        unlock_page(page);
        submit_bio(rw, bio);          /*提交 bio 实例*/
    out:
        return ret;
    }
```

__swap_writepage()函数将分情况执行写出操作，简述如下：

（1）交换区是普通文件且内容映射连续

此时，所有的文件内容都是用于槽位，如下图所示，交换缓存与交换文件页缓存其实就是一样的。由槽位编号就可以计算出所处文件内容的位置（文件内容映射不连续则不行，因为槽位编号跳过了映射不连续的页）。__swap_writepage()函数调用地址空间操作结构中的 direct_IO()函数，直接写出页至块设备。



（2）交换区是分区或普通文件内容映射不连续

首先调用 bdev_write_page()函数尝试通过块设备操作结构中定义的 rw_page()读写页函数执行写出操作，如果没有定义此函数或函数执行失败，则调用 get_swap_bio()函数查找 swap_extent 实例链表获取槽位映射的起始数据块号，据此构建 bio 实例，向块设备请求队列提交，执行写出操作。bio 完成回调函数设为 end_write_func()。上面的 swap_page_sector(page)函数用于查找 swap_extent 实例链表获取槽位映射的起始数据块号，函数源代码请读者自行阅读。

下面看一下 bdev_write_page()函数和 end_write_func()的定义。

bdev_write_page()函数用于写出整页，定义如下（/fs/block_dev.c）：

```
int bdev_write_page(struct block_device *bdev, sector_t sector,
                                        struct page *page, struct writeback_control *wbc)
{
    int result;
    int rw = (wbc->sync_mode == WB_SYNC_ALL) ? WRITE_SYNC : WRITE;
    const struct block_device_operations *ops = bdev->bd_disk->fops;
    if (!ops->rw_page || bdev_get_integrity(bdev))
        return -EOPNOTSUPP;          /*没有定义 rw_page()函数返回错误码*/
    set_page_writeback(page);        /*设置回写标记位*/
    result = ops->rw_page(bdev, sector + get_start_sect(bdev), page, rw);    /*写出页，成功返回 0*/
```

```
    if (result)
        end_page_writeback(page);        /*结束回写，/mm/filemap.c*/
    else
        unlock_page(page);
    return result;      /*成功返回 0*/
}
```

如果 block_device_operations 实例中没有定义 rw_page()函数，或写出失败，函数返回错误码，后面将构建 bio 实例执行回写。

bio 实例结束的回调函数设为 end_swap_bio_write()，函数定义如下（/mm/page_io.c）：

```
void end_swap_bio_write(struct bio *bio, int err)
{
    const int uptodate = test_bit(BIO_UPTODATE, &bio->bi_flags);
    struct page *page = bio->bi_io_vec[0].bv_page;

    if (!uptodate) {        /*如果数据无效，写出失败*/
        SetPageError(page);
        set_page_dirty(page);
        ...  /*输出信息*/
        ClearPageReclaim(page);
    }
    end_page_writeback(page);
                    /*清页回收、回写标记位，唤醒等待回写完成的进程等，/mm/filemap.c*/
    bio_put(bio);
}
```

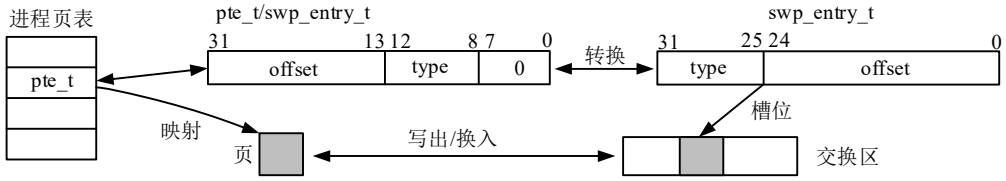
5 换入页

在交换出匿名映射页时，将由槽位信息生成 swp_entry_t 实例，并写入 page 实例的 private 成员。在解除匿名映射页映射关系时，将由 swp_entry_t 实例生成内存页表项 pte_t 实例，并写入映射页表项中，以便下次访问该页时，能找到保存匿名映射页数据的槽位，从中恢复数据。

进程访问换出的匿名映射页时，将触发缺页异常，在缺页异常处理函数中将页表项 pte_t 实例转换成 swp_entry_t 实例，从中获取保存匿名映射页数据的槽位信息，并从交换区槽位中读回数据至内存页。

■设置映射页表项

内存页表项 pte_t 结构体是体系结构相关的，对于 MIPS32 系统，swp_entry_t 实例与 pte_t 实例的转换关系如下图所示：



pte_t 实例中，bit[8...12]保存交换区 type 值，bit[13...31]保存槽位偏移是 offset 值。

swp_entry_to_pte()函数用于将 swp_entry_t 实例转换成页表项 pte_t 实例。

swp_entry_to_pte()函数定义如下（/include/linux/swapops.h）：

```
static inline pte_t swp_entry_to_pte(swp_entry_t entry)
{
    swp_entry_t arch_entry;      /*体系结构相关的 swp_entry_t 实例*/

    arch_entry = __swp_entry(swp_type(entry), swp_offset(entry));
                    /*转换成体系结构相关的 swp_entry_t 实例，/arch/mips/include/asm/pgtable-32.h*/
    return __swp_entry_to_pte(arch_entry);      /*/arch/mips/include/asm/pgtable-32.h*/
                                    /*体系结构相关的 swp_entry_t 实例转成 pte_t 实例*/
}
```

__swp_entry()函数用于将体系结构无关的 swp_entry_t 实例转换成体系结构相关的 swp_entry_t 实例。__swp_entry_to_pte()函数用于将体系结构相关的 swp_entry_t 实例转换成 pte_t 实例。

MIPS32 体系结构中体系结构相关的 swp_entry_t 实例与 pte_t 实例的布局是一样的，转换函数如下：

```
/*/arch/mips/include/asm/pgtable-32.h*/

#define   __swp_type(x)                 (((x).val >> 8) & 0x1f)      /*由 pte_t 获取交换区 type 值*/
#define   __swp_offset(x)               ((x).val >> 13)     /*由 pte_t 获取槽位 offset 值*/
#define   __swp_entry(type,offset)      ((swp_entry_t)      { ((type) << 8) | ((offset) << 13) })
                                                    /*swp_entry_t 实例转换成体系结构相关实例*/
#define   __pte_to_swp_entry(pte)       ((swp_entry_t) { pte_val(pte) })
#define   __swp_entry_to_pte(x)         ((pte_t) { (x).val })
                                                /*体系结构相关 swp_entry_t 实例与 pte_t 实例相同*/
```

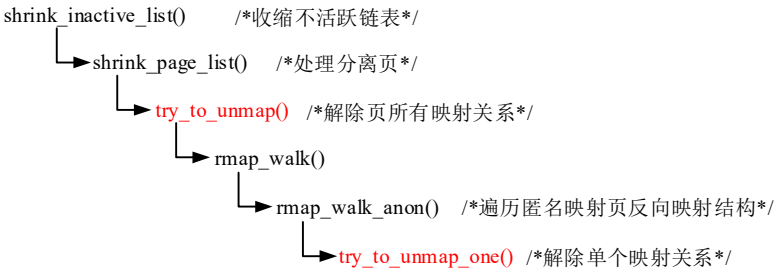
pte_to_swp_entry()函数用于将 pte_t 实例转换成体系结构无关的 swp_entry_t 实例，函数定义如下：

```
static inline swp_entry_t pte_to_swp_entry(pte_t pte)     /*/include/linux/swapops.h*/
{
    swp_entry_t arch_entry;

    if (pte_swp_soft_dirty(pte))      /*函数返回 0*/
        pte = pte_swp_clear_soft_dirty(pte);
    arch_entry = __pte_to_swp_entry(pte);      /*pte_t 转体系结构相关的 swp_entry_t 实例*/
    return swp_entry(__swp_type(arch_entry), __swp_offset(arch_entry));
                    /*体系结构相关的 swp_entry_t 实例转体系结构无关的 swp_entry_t 实例*/
}
```

在访问换出匿名映射页的缺页异常处理函数中，就需要将内存页表项中的 pte_t 实例转换成 swp_entry_t 实例，以便查找交换区中的槽位，读回换出页数据。

■解除映射

在收缩不活跃 LRU 链表时，对回收的映射页需要解除页所有映射关系，try_to_unmap()函数用于解除页所有映射关系，函数调用关系如下图所示：



try_to_unmap()函数在第 4 章介绍过了，函数内将遍历所有映射了该页的页表项，try_to_unmap_one()函数用于解除单个页表项的映射关系。

下面简要看一下 try_to_unmap_one()函数中与解除匿名映射页映射关系相关的代码（/mm/rmap.c）：

```
static int try_to_unmap_one(struct page *page, struct vm_area_struct *vma,   \
                                                    unsigned long address, void *arg)
{
    struct mm_struct *mm = vma->vm_mm;        /*进程地址空间*/
    pte_t *pte;
    pte_t pteval;
    spinlock_t *ptl;
    int ret = SWAP_AGAIN;
    enum ttu_flags flags = (enum ttu_flags)arg;

    pte = page_check_address(page, mm, address, &ptl, 0);
                                        /*映射页表项指针，/include/linux/rmap.h*/
    ...
    flush_cache_page(vma, address, page_to_pfn(page));    /*刷新缓存*/
    pteval = ptep_clear_flush(vma, address, pte);      /*清零内存页表项，返回原页表项值*/

    if (pte_dirty(pteval))        /*脏页*/
        set_page_dirty(page);    /*设置 page 脏标记*/

    update_hiwater_rss(mm);    /*更新水印值*/

    if (PageHWPoison(page) && !(flags & TTU_IGNORE_HWPOISON)) {
        ...
    } else if (pte_unused(pteval)) {
        ...
    } else if (PageAnon(page)) {          /*匿名映射页*/
        swp_entry_t entry = { .val = page_private(page) }; /*page->private 保存的 swp_entry_t 实例*/
        pte_t swp_pte;

        if (PageSwapCache(page)) {          /*页在交换缓存中*/
            if (swap_duplicate(entry) < 0) {    /*swap_map[]中引用计数值加 1，/mm/swapfile.c*/
                set_pte_at(mm, address, pte, pteval);
                                        /*swap_map[]引用计数值加 1 失败，重新写回原页表项*/
                ret = SWAP_FAIL;
                goto out_unmap;
            }
```

```
            ...
        } else if (IS_ENABLED(CONFIG_MIGRATION)) {
            ...
        }
        swp_pte = swp_entry_to_pte(entry);      /*swp_entry_t 实例转 pte_t 实例*/
        if (pte_soft_dirty(pteval))
            swp_pte = pte_swp_mksoft_dirty(swp_pte);
        set_pte_at(mm, address, pte, swp_pte);      /*pte_t 实例写入内存页表项*/
    } else if (IS_ENABLED(CONFIG_MIGRATION) &&(flags & TTU_MIGRATION)) {
        ...
    } else
        ...

    page_remove_rmap(page);      /*将页从反向映射结构中移除，映射计数减 1 等，/mm/rmap.c*/
    page_cache_release(page);      /*释放页，引用计数值_count 减 1*/
    ...
out:
    return ret;        /*函数返回*/
    ...
}
```

try_to_unmap_one()函数对于匿名映射页，主要是从 page->private 中获取 swp_entry_t 实例转换成 pte_t 实例，并写入映射内存页表项。

这里只处理了内存页表项，没有处理 TLB 表项。因为分离出来回收的页都是访问计数为 0 的页。在分离页统计其访问计数时，会刷出 TLB 表项，访问计数为 0，表示没有进程在两次统计之间访问了此页，因此在 TLB 中也就不会有对应表项，不需要处理。

try_to_unmap_one()函数还需要调用 swap_duplicate(entry)函数对槽位对应的交换区 swap_map[]数组项引用计数值加 1，下面看一下此函数的实现。

●增加槽位引用计数

swap_duplicate(entry)函数用于在解除映射时对槽位引用计数增 1，定义如下（/mm/swapfile.c）：

```
int swap_duplicate(swp_entry_t entry)
{
    int err = 0;

    while (!err && __swap_duplicate(entry, 1) == -ENOMEM)
        err = add_swap_count_continuation(entry, GFP_ATOMIC);      /*增加扩展计数*/
    return err;          /*成功返回 0*/
}
```

在前面已经简要介绍过了槽位引用计数 swap_map[]数组的结构，__swap_duplicate()函数用于向引用计数值加 1，成功返回 true，返回-ENOMEM 错误码表示要增加扩展计数。

如果增加扩展计数，while ()循环中随后调用 add_swap_count_continuation()函数判断是否要增加扩展计数，需要则扩展，不需要则不扩展，函数返回 0。然后，再调用__swap_duplicate()函数增加计数值。

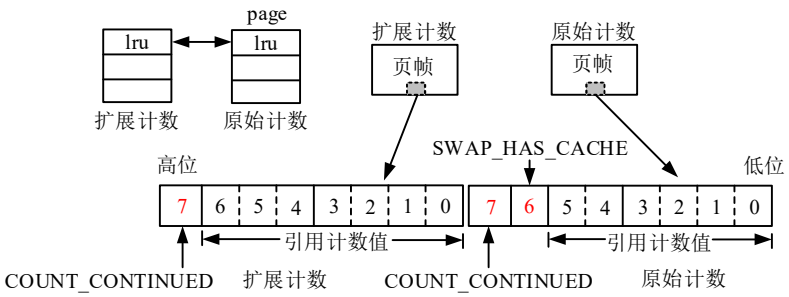
__swap_duplicate()函数定义如下（/mm/swapfile.c）：

```
static int __swap_duplicate(swp_entry_t entry, unsigned char usage)
/*usage：增加的引用计数，这里为 1*/
```

```
{
    struct swap_info_struct *p;
    unsigned long offset, type;
    unsigned char count;
    unsigned char has_cache;
    int err = -EINVAL;

    if (non_swap_entry(entry))
        goto out;

    type = swp_type(entry);         /*交换区编号*/
    if (type >= nr_swapfiles)
        goto bad_file;
    p = swap_info[type];            /*交换区 swap_info_struct 实例*/
    offset = swp_offset(entry);     /*交换区内槽位编号*/

    spin_lock(&p->lock);
    if (unlikely(offset >= p->max))
        goto unlock_out;

    count = p->swap_map[offset];        /*原始引用计数值*/
    if (unlikely(swap_count(count) == SWAP_MAP_BAD)) {
        err = -ENOENT;
        goto unlock_out;
    }

    has_cache = count & SWAP_HAS_CACHE;     /*分配槽位时设置此标记位*/
    count &= ~SWAP_HAS_CACHE;           /*count 清 SWAP_HAS_CACHE 标记位*/
    err = 0;
    if (usage == SWAP_HAS_CACHE) {
        /*SWAP_HAS_CACHE 表示分配了槽位，但没有对应的缓存页*/
        if (!has_cache && count)
            has_cache = SWAP_HAS_CACHE;
        else if (has_cache)         /* someone else added cache */
            err = -EEXIST;
        else                        /* no users remaining */
            err = -ENOENT;
    } else if (count || has_cache) {        /*增加引用计数*/
        if ((count & ~COUNT_CONTINUED) < SWAP_MAP_MAX)      /*累加到原始计数*/
            count += usage;             /*这里表示加 1*/
        else if ((count & ~COUNT_CONTINUED) > SWAP_MAP_MAX)
            err = -EINVAL;
        else if (swap_count_continued(p, offset, count))        /*count==(SWAP_MAP_MAX+1)*/
                                    /*在扩展计数中加 1，成功返回 true，失败返回 false*/
            count = COUNT_CONTINUED;            /*原始计数进位了*/
        else
```

```
                err = -ENOMEM;          /*需要增加扩展计数*/
        } else
                err = -ENOENT;                  /* unused swap entry */

        p->swap_map[offset] = count | has_cache;        /*赋值原始引用计数*/
unlock_out:
        spin_unlock(&p->lock);
out:
        return err;
        ...
}
```

槽位引用计数的结构前面介绍过了，下面简要说明一下增加槽位计数的过程，如下图所示：



计数值就是将原始计数和所有扩展计数拼接起来表示的一个数，但是原始计数中的 bit6 要除去，所有计数中的 bit7 也要除去，它用于表示有没有进位，即下一个计数是否使用。

各计数值保存在不同的页中，各页由 lru 成员组成链表，各计数在各页中的相对位置是相同的。

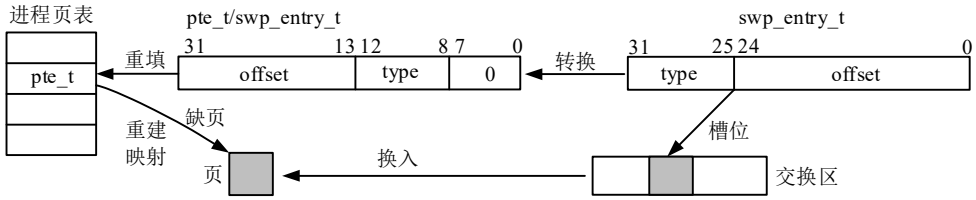
前面介绍的 add_swap_count_continuation()函数用于为扩展计数分配页，并添加到原始计数所在页的链表中，swap_count_continued()函数表示原始计数值达到了最大值，需要将增加值加到扩展计数中，因此原始计数值清零，并设置 COUNT_CONTINUED 标记位，表示进位了（如果 99 加 1 为 100，个位十位清零）。

swap_free(swp_entry_t entry)函数用于对槽位引用计数值减 1，若为 0 则释放槽位，函数源代码请读者自行阅读（/mm/swapfile.c）。

■载入页

匿名映射页被换出后，在原来的映射页表项中将写入由 swp_entry_t 实例生成的 pte_t 实例，如下图所示。当进程再次访问换出页时，将首先触发 TLB 重填异常，将内存页表项中的 pte_t 实例写入 TLB。TLB 重填异常返回后，再次访问时，又将触发 TLB 缺页异常，因为这时的 pte_t 表项是由 swp_entry_t 实例生成的，是无效的。

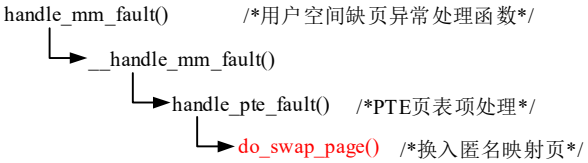
在 TLB 缺页异常处理函数中，会将 pte_t 实例还原成 swp_entry_t 实例，获取保存匿名映射页数据的槽位信息，分配页，从槽位读回原数据，重新建立映射关系。



对于 MIPS32 体系结构，TLB 重填异常会将 pte_t 实例值写入到 CPU 核 TLB 页表项中（右移 6 位），如下图所示。由于 TLB 页表项中有效性标记位 V 为 0，随后将再次触发 TLB 无效异常。TLB 无效异常由

第 4 章介绍的缺页异常处理程序处理，处理程序检测到对应的内存页表项不为空（保存了 swp_entry_t/pte_t 实例），将调用 **do_swap_page()**函数处理换出到交换区的匿名映射页。

缺页异常处理程序中相关的函数调用关系如下图所示：



do_swap_page()函数根据内存页表项保存的 swp_entry_t/pte_t 实例获取换出页所在的交换区槽位，然后到对应的交换缓存中查找该页是否已经在交换缓存中（其它映射该页的进程可能已将其读入），如果不在交换缓存中，则分配页并插入交换缓存，从交换区读取数据，最后设置进程页表项建立到新分配页的映射关系。如果匿名映射页已经在交换缓存中，则直接建立映射即可。

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

```
static int do_swap_page(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pte_t *page_table, pmd_t *pmd,unsigned int flags, pte_t orig_pte)
{
    spinlock_t *ptl;
    struct page *page, *swapcache;
    struct mem_cgroup *memcg;
    swp_entry_t entry;
    pte_t pte;
    int locked;
    int exclusive = 0;
    int ret = 0;

    if (!pte_unmap_same(mm, pmd, page_table, orig_pte))
        goto out;

    entry = pte_to_swp_entry(orig_pte);      /*pte_t 实例转体系结构无关的 swp_entry_t 实例*/
    if (unlikely(non_swap_entry(entry))) {
        if (is_migration_entry(entry)) {
            migration_entry_wait(mm, pmd, address);
        } else if (is_hwpoison_entry(entry)) {
            ret = VM_FAULT_HWPOISON;
        } else {
            print_bad_pte(vma, address, orig_pte, NULL);
            ret = VM_FAULT_SIGBUS;
        }
        goto out;
    }
```

```
delayacct_set_flag(DELAYACCT_PF_SWAPIN);
page = lookup_swap_cache(entry);
                    /*在交换区对应交换缓存中查找匿名页，没找到返回 NULL，/mm/swap_state.c*/
if (!page) {
      page = swapin_readahead(entry,GFP_HIGHUSER_MOVABLE, vma, address);
                     /*分配页，插入交换缓存，从交换区读数据（异步）等，/mm/swap_state.c*/
      if (!page) {
            ...
      }
      ret = VM_FAULT_MAJOR;
      count_vm_event(PGMAJFAULT);
      mem_cgroup_count_vm_event(mm, PGMAJFAULT);
} else if (PageHWPoison(page)) {
      ret = VM_FAULT_HWPOISON;
      delayacct_clear_flag(DELAYACCT_PF_SWAPIN);
      swapcache = page;
      goto out_release;
}

swapcache = page;          /*匿名映射页*/
locked = lock_page_or_retry(page, mm, flags);       /*锁定页*/

delayacct_clear_flag(DELAYACCT_PF_SWAPIN);
...

if (unlikely(!PageSwapCache(page) || page_private(page) != entry.val))
      goto out_page;

page = ksm_might_need_to_copy(page, vma, address);
...

if (mem_cgroup_try_charge(page, mm, GFP_KERNEL, &memcg)) {
      ret = VM_FAULT_OOM;
      goto out_page;
}

page_table = pte_offset_map_lock(mm, pmd, address, &ptl);       /*内存页表项指针*/
if (unlikely(!pte_same(*page_table, orig_pte)))
      goto out_nomap;

if (unlikely(!PageUptodate(page))) {
      ret = VM_FAULT_SIGBUS;
      goto out_nomap;
}

inc_mm_counter_fast(mm, MM_ANONPAGES);
```

```
        dec_mm_counter_fast(mm, MM_SWAPENTS);
        pte = mk_pte(page, vma->vm_page_prot);     /*生成内存页表项 pte_t 实例*/
        if ((flags & FAULT_FLAG_WRITE) && reuse_swap_page(page)) {
                pte = maybe_mkwrite(pte_mkdirty(pte), vma);
                flags &= ~FAULT_FLAG_WRITE;
                ret |= VM_FAULT_WRITE;
                exclusive = 1;
        }
        flush_icache_page(vma, page);
        if (pte_swp_soft_dirty(orig_pte))     /*返回 0*/
                pte = pte_mksoft_dirty(pte);
        set_pte_at(mm, address, page_table, pte);   /*写入内存页表项*/
        if (page == swapcache) {
                do_page_add_anon_rmap(page, vma, address, exclusive);        /*加入反向映射结构*/
                mem_cgroup_commit_charge(page, memcg, true);
        } else {        /* ksm created a completely new copy */
                page_add_new_anon_rmap(page, vma, address);
                mem_cgroup_commit_charge(page, memcg, false);
                lru_cache_add_active_or_unevictable(page, vma);
        }

        swap_free(entry);       /*槽位引用计数值减 1，/mm/swapfile.c*/
        if (vm_swap_full() || (vma->vm_flags & VM_LOCKED) || PageMlocked(page))
                try_to_free_swap(page);       /*试图释放槽位，/mm/swapfile.c*/
        unlock_page(page);
        if (page != swapcache) {
                unlock_page(swapcache);
                page_cache_release(swapcache);
        }

        if (flags & FAULT_FLAG_WRITE) {   /*如果匿名映射页具有写保护，还需要处理此种情况/
                ret |= do_wp_page(mm, vma, address, page_table, pmd, ptl, pte);
                                                /*分配页，复制数据，与新分配页建立映射关系，见第 4 章*/
                if (ret & VM_FAULT_ERROR)
                        ret &= VM_FAULT_ERROR;
                goto out;
        }

        update_mmu_cache(vma, address, page_table);
unlock:
        pte_unmap_unlock(page_table, ptl);
out:
        return ret;
        ...
}
```

do_swap_page()函数比较好理解，pte_to_swp_entry(orig_pte)函数将内存页表项 pte_t 实例转换成体系结

构无关的 swp_entry_t 实例；然后据此在对应的交换缓存中查找所需的页，如果没有找到则分配页，加入交换缓存，读数据，如果存在则跳过这些步骤。在读取数据的过程中设置页面的 PG_swapbacked 标记位，读取结束后清除。

do_swap_page()函数随后由分配/查找的页生成内存页表项，写入触发缺页异常的内存页表项中，并将页添加到反向映射结构中，调用 swap_free(entry)函数对槽位引用计数值减 1，调用 try_to_free_swap(page)尝试释放槽位。最后，如果匿名映射页是写保护页，且本次是写操作引发的异常，则需要再分配新页，复制数据，再与新页建立映射关系。

至此，回收匿名映射页的换入换出（页交换）操作就介绍完了。

11.5 小结

本章主要介绍了内存与块设备之间的数据交互。

内核通过文件的形式来访问块设备，访问裸块设备使用块设备文件，通过普通文件访问分区文件系统。打开文件时，内核会在内存中建立文件内容的页缓存，由地址空间管理，页缓存中以页为单位缓存文件内容，也就是说在内存中是以页为单位保存文件内容的。

进程读写文件时，通常是对页缓存中文件内容的读写，页缓存与块设备之间的数据传输由地址空间操作结构中的函数完成。

在读文件操作中会预读文件内容至页缓存，写操作时，可能只是写到页缓存，页缓存与块设备之间的同步会延后进行，这由数据回写机制完成。用户也可以通过系统调用发起对文件和文件系统的同步。

系统中的物理内存总是紧张的，内核会对分配给用户进程使用的页进行回收，包括匿名映射页和文件缓存页。匿名映射页和缓存页分配后，被添加到物理内存域的 LRU 链表中，回收操作将扫描链表，对最近最少使用的页进行回收。

回收文件缓存页比较简单，如果页是脏的则执行回写，然后如果有进程映射了该页，则解除映射，而后就可以释放页至伙伴系统，使其成为空闲页了。

回收匿名映射页需要用户创建和启用交换区，交换区是块设备（分区）或普通文件。回收匿名映射页时，需要将数据写出到交换区，解除映射时将写出交换区的位置信息写入原映射页表项。进程下次访问该页时，在缺页异常处理函数中分配页，从原映射页表项中获取交换区信息，从交换区中恢复数据，重新建立映射。