# 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实列。

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\*/

structblk\_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：



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。



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\_structdiscard\_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。