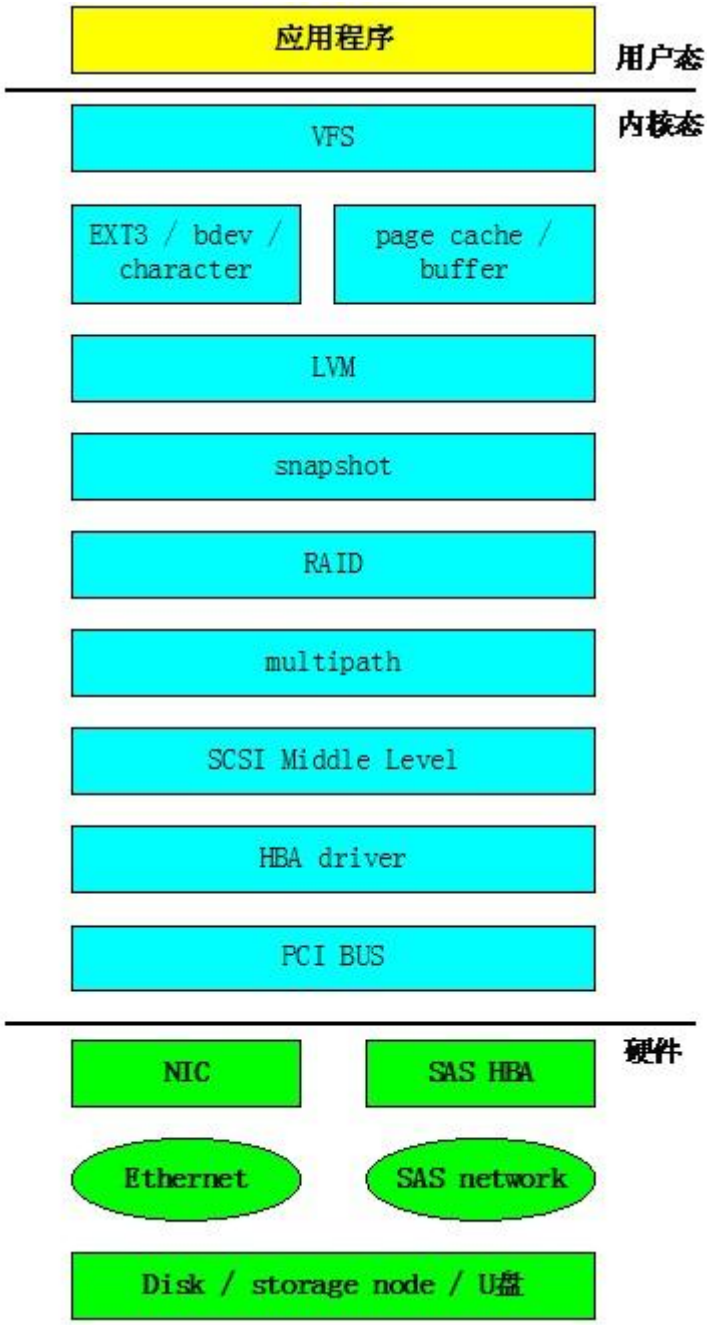
一个 IO 的传奇一生

前言

前几天同事提议写一篇文章来仔细分析一下一个 IO 从创建到消亡的整个过程，我觉得这个想法很好，一个 IO 从创建到消亡经历了千山万水，从软件到硬件涉及到很多很多的技术。一个看似简单的 IO 读写操作，其实汇集了从计算机软件技术、硬件技术、电子技术、信号处理等各个方面的内容。所以，我想把 IO 的一生通过自己的认识把他描述一下，让世人看清在执行一个简单的 IO 操作究竟汇集了多少的智慧！汇集了工程师、科学家多少的心血！在此，将此系列文章定名为《一个 IO 的传奇一生》，与大家一起分享。

针对不同的操作系统，IO 历程是有所差别的，但是很多基本思想是相同的。在此，我想以 Linux 操作系统为样本，对整个 IO 历程进行深入分析，最主要的是设计思想方面的考虑。



上图描述了 IO 操作中所涉及到的软硬件模块，从这张图中我们可以一窥整个系统还是很庞大的，主要涉及了文件系统、块设备层、SCSI 层、PCI 层、SAS/Ethernet 网络以及磁盘/U 盘。本文会根据作者的理解对 IO 在上述各个层次的游历过程进行详细阐述。

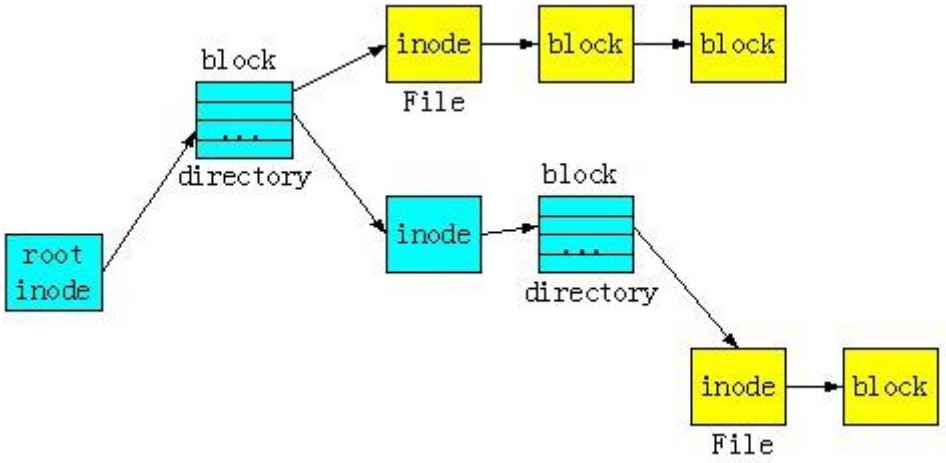
文件基本操作

如果你想保存你的 Word 资料至本地硬盘，你就会触发一个文件系统写操作。如果你想将一个文件从本地电脑拷贝到 U 盘时，你会触发一次文件系统的读写过程。大家知道，为了简化用户对文件的管理，操作系统提供了文件系统对数据资料进行了管理，文件系统是操作系统最为重要的组成部分。一旦你想往文件系统写入数据时，一个新的 IO 请求就会在用户态诞生，但是，其绝大部分的人生旅程都会在内核空间。对于不同的应用类型，IO 请求的属性会

大相径庭。除了文件本身应该具备的基本属性（读写权限等）之外，我们还需要考虑文件的访问模式：异步 IO 还是同步 IO？对文件系统的 Cache 是如何控制的？应用程序和内核程序之间是如何交互的？所以，在创建一个 IO 时，我们需要考虑很多这样的因素。
我们知道，当我们需要进行文件操作的时候，5 个 API 函数是必不可少的。Create，Open，Close，Write 和 Read 函数实现了对文件的所有操作。Create 函数用来打开一个文件，如果该文件不存在，那么需要在磁盘上创建该文件。Open 函数用于打开一个指定的文件。如果在 Open 函数中指定 O_CREATE 标记，那么 Open 函数同样可以实现 Create 函数的功能。Close 函数用于释放文件句柄。Write 和 Read 函数用于实现文件的读写过程。举个例子，如果用户需要对一个文件进行写操作，那么首先调用 Open 函数打开想要操作的文件，函数完成之后获取所要操作文件的句柄；然后调用 Write 函数将数据写入文件；最后采用 Close 函数释放文件句柄，结束文件写入过程。上述过程大家应该都非常的熟悉，在上述过程中，整个系统到底发生了哪些操作呢？

打开文件

众所周知，用户态的 API 函数通过系统调用陷入内核。对于 Open 函数对应了 sys_open 函数例程。该函数的主要职责是查找指定文件的 inode，然后在内核中生成对应的文件对象。在 Linux 中，Sys_open 函数调用 do_sys_open 完成具体功能。在 do_sys_open 中通过 do_filp_open 函数完成文件名解析、inode 对象查找，然后创建 file 对象，最后执行特定文件对应的 file->open 函数。Do_filp_open 过程中的核心处理函数是 link_path_walk。该函数完成了基本的文件路径的解析功能，是名字字符串解析处理实现的核心。该函数的实现基于分级解析处理的思想。例如，当需要解析“/dev/mapper/map0”字符串时，其首先需要判断从何处开始解析，根目录还是当前目录？这个例子是从根目录开始解析的，那么首先获取根目录的 dentry 对象并开始分析后继字符串。处理过程是以‘/’字符为界按序提取字符串。根据规则，首先我们可以提取“dev”字符串，并且计算该字符串的 Hash 值，通过该 Hash 值查找 dentry 下的 inode Hash 表，就可以很快的找到/dev/目录下的 inode 对象。Hash 值的计算是比较简单的，把所有字符对应的值累加起来就可以得到一个 Hash 值。根据规则，依此类推，最后解析得到”/dev/mapper/”目录的 inode 对象以及文件名字符串“map0”。到这一步为止，link_path_walk 函数的使命完成，最后可以通过 do_last 函数获取或者创建文件 inode。如果用户态程序设置了 O_CREATE 标记，那么系统如果找不到用户指定的 inode，do_last 会创建一个新的文件 inode，并且把这些信息以元数据的形式写入磁盘。当指定文件的 inode 找到之后，另一件很重要的事情就是初始化 file 文件对象。初始化文件对象通过__dentry_open 函数来实现。文件对象通过 inode 参数进行初始化，并且把 inode 的操作方法函数集告诉给 file 对象。一旦 file 对象初始化成功之后，调用文件对象的 open 函数执行进一步的初始化工作。
通过上述分析，整个过程看似比较复杂，涉及到 dentry，inode 以及 file 对象。其实这个模型还是很简单的。Dentry 用来描述文件目录，在磁盘上会采用元数据的方式存储在一个 block 中，文件目录本身在 Linux 中也是一个文件。Inode 描述一个具体的文件，也通过元数据的方式在磁盘上保存。如果对一个文件系统从根目录开始往下看，整个文件系统是一颗庞大的 inode 树：



在打开一个文件的过程中，文件系统所要做的事情就是找到指定文件的 inode，所以在这个过程中会有磁盘元数据读操作。一旦文件所属的 inode 被找到，那么需要在内存中初始化一个描述被打开文件的对象，这个对象就是 file。所以，dentry，inode 之类的信息在磁盘上是永久存储的，file 对象是在内存中是临时存在的，它会随着文件的创建而生成，随着文件的关闭而消亡。

在 Linux 系统中文件类型是多种多样的，一个 USB 设备也是一个文件，一个普通的 Word 文档也是一个文件，一个 RAID 设备也是一个文件。虽然他们在系统中都是文件，但是，他们的操作方式是截然不同的。USB 设备可能需要采用字符设备的方式和设备驱动交互；RAID 设备可能需要采用块设备的方式和设备驱动进行交互；普通 Word 文件需要通过 cache 机制进行性能优化。所以，虽然都是文件，但是，文件表面下的这些设备是不相同的，需要采用的操作方法显然是截然不同的。作为一个通用的文件系统，如何封装不同的底层设备是需要考虑的问题。在 Linux 中，为了达到这个目的，推出了 VFS 概念。在 VFS 层次对用户接口进行了统一封装，并且实现了通用的文件操作功能。例如打开一个文件和关闭一个文件的操作都是相同的。在 VFS 下面会有针对不同需求的具体文件系统，例如针对 Word 文档可以采用 EXT3 文件系统进行操作，对于磁盘设备可以采用 bdev 块设备文件系统进行操作。在打开一个文件，对文件对象 file 进行初始化的时候，会将具体的文件系统操作方法关联到 file->f_op 和 file->f_mapping 对象。在后面的读写过程中，我们将会看到针对不同的文件类型，会采用不同的 f_op 和 f_mapping 方法。

读写文件

当一个文件被打开之后，用户态程序就可以得到一个文件对象，即文件句柄。一旦获取文件句柄之后就可以对其进行读写了。用户态的读写函数 write 对应内核空间的 sys_write 例程。通过系统调用可以陷入 sys_write。Sys_write 函数在 VFS 层做的工作及其有限，其会调用文件对象中指定的操作函数 file->f_op->write。对于不同的文件系统，file->f_op->write 指向的操作函数是不同的。对于 EXT3 文件系统而言，在文件 inode 初始化的时候会指定 ext3_file_operations 操作方法集。该方法集说明了 EXT3 文件系统的读写操作方法，说明如下：

```
const struct file_operations ext3_file_operations = {
.llseek= generic_file_llseek,
.read= do_sync_read,
.write= do_sync_write,
.aio_read= generic_file_aio_read,
.aio_write= ext3_file_write,
.ioctl= ext3_ioctl,
#ifdef CONFIG_COMPAT
.compat_ioctl= ext3_compat_ioctl,
#endif
.mmap= generic_file_mmap,
.open= generic_file_open,
.release= ext3_release_file,
.fsync= ext3_sync_file,
.splice_read= generic_file_splice_read,
.splice_write= generic_file_splice_write,
};
```

如果文件设备是一个 USB 设备，并且采用的是字符设备的接口，那么在初始化文件 inode 的时候会调用 init_special_inode 初始化这些特殊的设备文件。对于字符设备会采用默认的 def_chr_fops 方法集，对于块设备会采用 def_blk_fops 方法集。不同的文件类型会调用各自的方法集。下面章节会对 EXT3 文件写和块设备文件写进行详细阐述。由于字符设备类型比较简单，在此进行简单说明。

Def_chr_fops 方法集其实就定义了 open 方法，其它的方法都没有定义。其实字符设备的操作方法都需要字符设备驱动程序自己定义，每个设备驱动程序都需要定义自己的 write、read、open 和 close 方法，这些方法保存在字符设备对象中。当用户调用文件系统接口 open 函数打开指定字符设备文件时，VFS 会通过上述讲述的 sys_open 函数找到设备文件 inode 中保存的 def_chr_fops 方法，并且执行该方法中的 open 函数（chrdev_open），chrdev_open 函数完成的一个重要功能就是将文件对象 file 中采用的方法替换成驱动程序设定的设备操作方法。完成这个偷梁换柱的代码是：

```
filp->f_op = fops_get(p->ops)
```

一旦这个过程完成，后继用户程序通过文件系统的 write 方法都将会调用字符设备驱动程序设定的 write 方法。即对于字符设备文件而言，在 VFS 的 sys_write 函数将直接调用字符设备驱动程序的 write 方法。所以，对于字符设备驱动程序而言，整个过程很简单，用户态程序可以直接通过系统调用执行字符设备驱动程序的代码。而对于块设备和普通文件，这个过程将会复杂的多。

在用户程序发起写请求的时候，通常会考虑如下三个问题：第一个问题是用户态数据如何高效传递给内核？第二个问题是采用同步或者异步的方式执行 IO 请求。第三个问题是如果执行普通文件操作，需不需要文件 Cache？

第一个问题是数据拷贝的问题。对于普通文件，如果采用了 page cache 机制，那么这种拷贝合并在很大程度上是避免不了的。但是对于网卡之类的设备，我们在读写数据的时候，需要避免这样的数据拷贝，否则数据传输效率将会变的很低。我第一次关注这个问题是在做本科毕业设计的时候，那时候我设计了一块 PCI 数据采集卡。在 PCI 采集卡上集成了 4KB 的 FIFO，数据采集电路会将数据不断的压入 FIFO，当 FIFO 半满的时候会对 PCI 主控芯片产生一个中断信号，通知 PCI 主控制器将 FIFO 中的 2KB 数据 DMA 至主机内存。CPU 接收到这个中断信

号之后，分配 DMA 内存，初始化 DMA 控制器，并且启动 DMA 操作，将 2KB 数据传输至内存。并且当 DMA 完成操作之后，会对 CPU 产生一个中断。板卡的设备驱动程序在接收到这个中断请求之后，面临一个重要的问题：如何将内核空间 DMA 过来的数据传输给用户空间？通常有两种方法：一种是直接将内核内存映射给用户程序；另一种是进行数据拷贝的方式。对于 PCI 数据采集卡而言，一个很重要的特性是实时数据采集，在板卡硬件 FIFO 很小的情况下，如果主机端的数据传输、处理耗费太多的时间，那么整条 IO 流水线将无法运转，导致 FIFO 溢出，数据采集出现漏点的情况。所以，为了避免这样的情况，在这些很严格应用的场合只能采用内存映射的方法，从而实现数据在操作系统层面的零拷贝。在 Linux 中，我们可以采用 memory map 的方法将内核空间内存映射给用户程序，从而实现用户程序对内核内存的直接访问。在 Windows 操作系统中，这种内核空间和用户空间的数据交互方式定义成两种：Map IO 和 Direct IO。Map IO 就是采用内存拷贝的方式，Direct IO 就是采用 MDL 内存映射的方式。在编写 WDM Windows 设备驱动程序的时候经常会用到这两种数据传输模式。值得注意的是，Windows 中的 Direct IO 和 Linux 中的 Direct IO 是完全不同的两个概念。在 Linux 中 Direct IO 是指写穿 page cache 的一种 IO 方法。

第二个问题是异步 IO 和同步 IO 的问题。对于普通文件而言，为了提高效率，通常会采用 page cache 对文件数据在内存进行缓存。Cache 虽然提高了效率，但是有些应用一旦发出写请求，并且执行完毕之后，其期望是将数据写入磁盘，而不是内存。例如，有些应用会有一些元数据操作，在元数据操作的过程中，通常期望将数据直接刷新至磁盘，而不是 Cache 在内存。这就提出了同步 IO 的需求。为了达到这个效果，可以在打开文件的时候设置 O_SYNC 标记。当数据在 page cache 中聚合之后，如果发现 O_SYNC 标记被设置，那么就会将 page cache 中的数据强制的刷新到磁盘。对于 EXT3 文件系统，该过程在 ext3_file_write 函数中实现。

第三个问题是普通文件的 cache 问题。对于普通文件，由于磁盘性能比较低，为了提高读写性能，通常会采用内存作为磁盘的 cache。文件系统会采用预读等机制对文件读写性能进行优化，避免磁盘随机 IO 性能过低对文件读写性能造成影响。但是，page cache 虽然提高了性能，但是也会对文件系统的可靠性造成一定影响。例如，当数据已经被写入内存之后，系统 Crash，内存中的磁盘数据将会遭到破坏。为了避免这种情况，Linux 文件系统提供了 Direct IO 的 IO 方式。该方式就是让一次 IO 过程绕过 page cache 机制，直接将文件内容刷新到磁盘。与上面的同步 IO 相比，Direct IO 达到的效果似乎有点类似。其实，同步 IO 是一种 write through 的 Cache 机制，而 Direct IO 是完全把 Cache 抛弃了。同步 IO 的数据在内存还是有镜像的，而 Direct IO 是没有的，这是两者的区别。在 Linux 中的 __generic_file_aio_write_nolock 函数中，会判断 O_DIRECT 标记是否被设置，如果该标记被设置，那么调用 generic_file_direct_write 函数完成数据磁盘写入过程。如果该标记不存在，那么调用 generic_file_buffered_write 函数将数据写入 page cache。

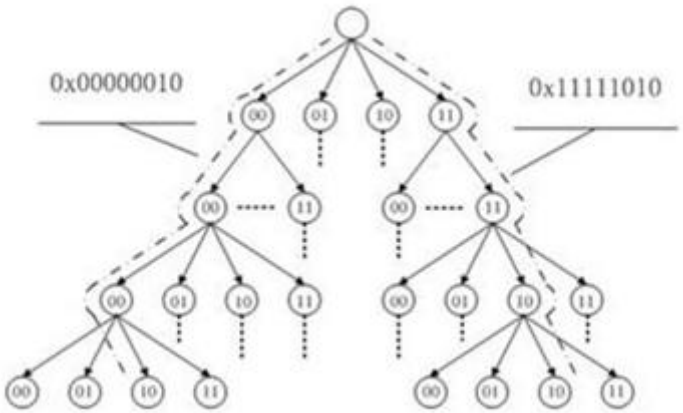
EXT3 文件写操作

如果应用层发起的是一个 Word 文档的写操作请求，那么通过上述分析，IO 会走到 sys_write 的地方，然后执行 file->f_op->write 方法。对于 EXT3，该方法注册的是 do_sync_write。Do_sync_write 的实现如下：

```
ssize_t do_sync_write(struct file *filp, const char __user *buf, size_t len, loff_t *ppos)
{
struct iovec iov = { .iov_base = (void __user *)buf, .iov_len = len };
struct kiocb kiocb;
ssize_t ret;
init_sync_kiocb(&kiocb, filp);
kiocb.ki_pos = *ppos;
kiocb.ki_left = len;
for (;;) {
ret = filp->f_op->aio_write(&kiocb, &iov, 1, kiocb.ki_pos);
if (ret != -EIOCBRETRY)
break;
wait_on_retry_sync_kiocb(&kiocb);
}
if (-EIOCBQUEUED == ret)
ret = wait_on_sync_kiocb(&kiocb);
*ppos = kiocb.ki_pos;
return ret;
}
```

该方法会直接调用非阻塞写处理函数，然后等待 IO 完成。对于具体 EXT3 文件读写过程函数调用关系可以参考《Ext3 文件系统读写过程分析》。对 EXT3 文件写操作主要考虑两种情况，一种情况是 DIRECT IO 方式；另一种情况是 page cache 的写方式。Direct IO 方式会直接绕过缓存处理机制，page cache 缓存方式是应用中经常采用的方式，性能会比 Direct IO 高出不少。对于每一个 EXT3 文件，都会在内存中维护一棵 Radix tree，这棵 radix tree 就是用来管理来 page cache 页的。当 IO 想往磁盘上写入的时候，EXT3 会查找其对应的 radix tree，看是否已经存在与写入地址相匹配的 page 页，如果存在那么直接将数据合并到这个 page 页中，并且结束一次 IO 过程，直接结束应用层请求。如果被访问的地址还没有对应的 page 页，那么需要为访问的地址空间分配 page 页，并且从磁盘上加载数据到 page 页内存，最后将这个 page 页加入到 radix tree 中。

对于一些大文件，如果不采用 radix tree 去管理 page 页，那么需要耗费大量的时间去查找一个文件内对应地址的 page 页。为了提高查找效率，Linux 采用了 radix tree 的这种管理方式。Radix tree 是通用的字典类型数据结构，radix tree 又被称之为 PAT 位树（Patricia Trie or crit bit tree）。Radix tree 是一种多叉搜索树，树的叶子节点是实际的数据条目。下图是一个 radix tree 的例子，该 radix tree 的分叉为 4，树高为 4，树中的每个叶子节点用来快速定位 8 位文件内偏移地址，可以定位 256 个 page 页。例如，图中虚线对应的两个叶子节点的地址值为 0x00000010 和 0x11111010，通过这两个地址值，可以很容易的定位到相应的 page 缓存页。



通过 radix tree 可以解决大文件 page 页查询问题。在 Linux 的实现过程中，EXT3 写操作处理函数会调用 generic_file_buffered_write 完成 page 页缓存写过程。在该函数中，其实现逻辑说明如下：

```
ssize_t
generic_file_buffered_write(struct kiocb *iocb, const struct iovec *iov,
unsigned long nr_segs, loff_t pos, loff_t *ppos,
size_t count, ssize_t written)
{
do {
/* 找到缓存的page页 */
page = __grab_cache_page(mapping,index,&cached_page,&lru_pvec);
/* 处理ext3日志等事情 */
status = a_ops->prepare_write(file, page, offset, offset+bytes);
/* 将用户数据拷贝至page页 */
copied = filemap_copy_from_user(page, offset,  buf, bytes);
/* 设置page页为dirty */
status = a_ops->commit_write(file, page, offset, offset+bytes);
}
}
```

第一步通过 radix tree 找到内存中缓存的 page 页，如果 page 页不存在，重新分配一个。第二步通过 ext3_prepare_write 处理 EXT3 日志，并且如果 page 是一个新页的话，那么需要从磁盘读入数据，装载进 page 页。第三步是将用户数据拷贝至指定的 page 页。最后一步将操作的 Page 页设置成 dirty。便于 writeback 机制将 dirty 页同步到磁盘。

Page cache 会占用 Linux 的大量内存，当内存紧张的时候，需要从 page cache 中回收一些内存页，另外，dirty page 在内存中聚合一段时间之后，需要被同步到磁盘。应该在 3.0 内核之前，Linux 采用 pdflush 机制将 dirty page 同步到磁盘，在之后的版本中，Linux 采用了 writeback 机制进行 dirty page 的刷新工作。有关于 writeback 机制的一些源码分析可以参考《writeback 机制源码分析》。总的来说，如果用户需要写 EXT3 文件时，默认采用的是 writeback 的 cache 算法，数据会首先被缓存到 page 页，这些 page 页会采用 radix tree 的方式管理起来，便于查找。一旦数据被写入 page 之后，会将该页标识成 dirty。Writeback 内核线程或者 pdflush 线程会周期性的将 dirty page 刷新到磁盘。如果，系统出现内存不足的情况，那么 page 回收线程会采用 cache 算法考虑回收文件系统的这些 page 缓存页。

EXT3 文件系统设计要点

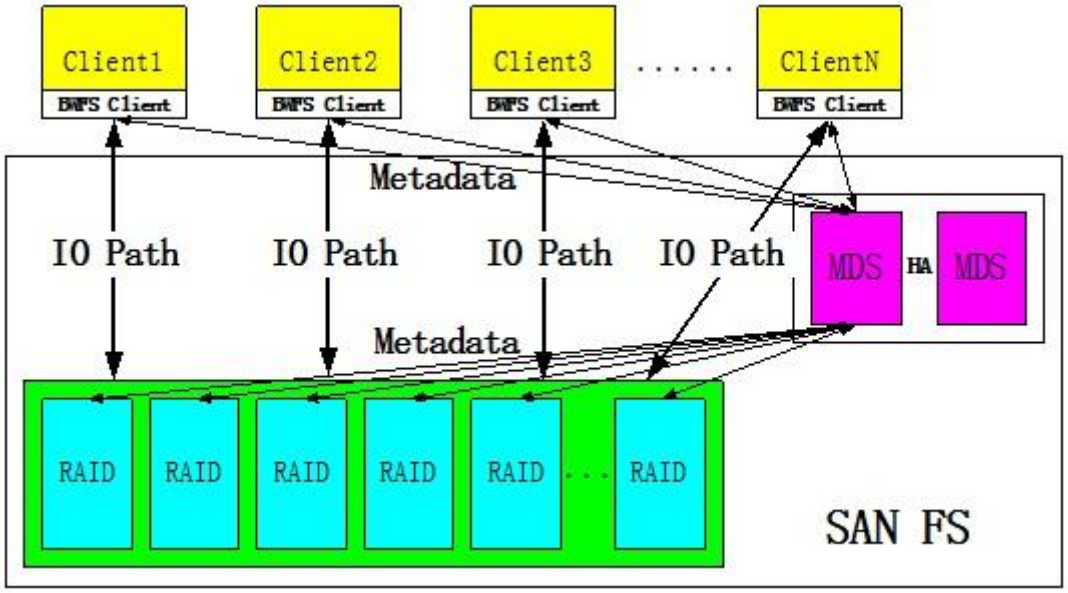
EXT3 文件系统是 Linux 中使用最为广泛的一个文件系统，其在 EXT2 的基础上发展起来，在 EXT2 的基础上加入了日志技术，从而使得文件系统更加健壮。考虑一下，设计一个文件系统需要考虑哪些因素呢？根据我的想法，我认为设计一个文件系统主要需要考虑如下几个方面的因素：

1) 文件系统使用者的特征是什么？大文件居多还是小文件居多？如果基本都是大文件应用，那么数据块可以做的大一点，使得元数据信息少点，减少这方面的开销。
2) 文件系统是读应用为主还是写应用为主？这点也是很重要的，如果是写为主的应用，那么可以采用 log structured 的方式优化 IO pattern。例如在备份系统中，基本都是以写请求，那么对于这样的系统，可以采用 log structured 的方式使得底层 IO 更加顺序化。
3) 文件系统的可扩展性，其中包括随着磁盘容量的增长，文件系统是否可以无缝扩展？例如以前的 FAT 文件系统由于元数据的限制，对支持的容量有着很强的限制。
4) 数据在磁盘上如何布局？数据在磁盘上的不同布局会对文件系统的性能产生很大的影响。如果元数据信息离数据很远，那么一次写操作将会导致剧烈的磁盘抖动。
5) 数据安全性如何保证？如果文件系统的元数据遭到了破坏，如何恢复文件数据？如果用户误删了文件，如何恢复用户的数据？这些都需要文件系统设计者进行仔细设计。
6) 如何保证操作事务的一致性？对于一次写操作设计到元数据更新和文件数据的更新，这

两者之间的操作次序如何设计？既能保证很好的性能，又能在系统 crash 的时候保证文件系统的一致性？在很多设计中采用数据先于元数据的方式，并且通过日志机制保证事务一致性。

7) 文件系统元数据占用多少系统内存？如果一个文件系统占用太多的系统内存，那么会影响整个系统的性能。

在设计一个文件系统的时候，需要考虑很多方面的问题，上述仅仅是列出了一些基本点，针对不同的文件系统，会有很多的具体问题。例如对于淘宝的 TFS 集群文件系统，由于系统前端采用了大量的缓存，从而使得 TFS 的输入全是大文件，因此可以采用大文件设计思想对其进行优化。比如减少目录层次，这样在检索一个文件的时候，不需要花费太多的时间，这种处理可以提升系统性能。另一个例子是中科蓝鲸的 BWFS 在非线编领域的应用。BWFS 是一个带外的集群存储系统，是一种 SAN 集群文件系统。这种文件系统架构的最大好处是可扩展性极高，由于元数据和数据 IO 流分离，所以，在很多应用中可以得到很高的集群性能。但是，这种架构也有一个问题，当应用文件以小文件为主的时候，元数据服务器会成为整个系统的性能瓶颈。因为，Client 在访问一个文件的时候首先需要访问元数据服务器获取具体的块地址信息。

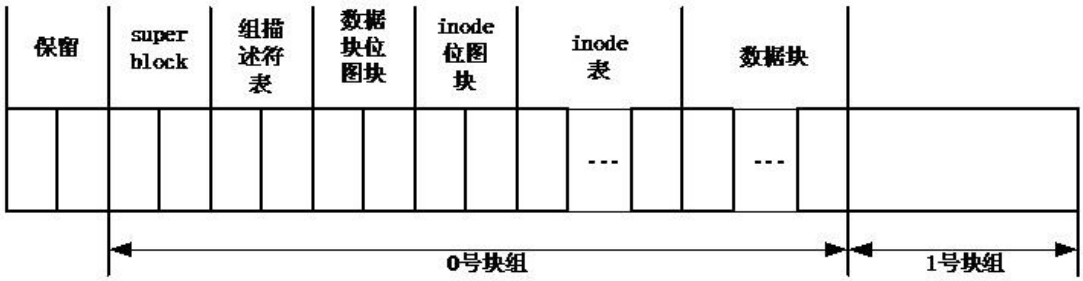


为了解决这个问题，文件系统设计人员需要对小文件这一个问题进行优化，其采用的策略是在 Client 进行元数据缓存。通过元数据缓存可以减少元数据访问的频率，自然就解决了这个系统性能瓶颈点问题。

从上述分析可以看出，文件系统设计是一个非常复杂的过程，其需要考虑很多应用的特征，特别是 IO 的 Pattern 和应用模式。很多文件系统的优化都是在深入分析了应用模式之后才得出的解决方案。

对于 EXT3 这样一个通用文件系统而言，其主要应用领域是个人桌面应用，个人以为可扩展性、性能并不是第一位的，而可靠性是比较重要的。要理解 EXT3 文件系统，我以为最好的方式是需要理解文件系统在磁盘上的布局，一旦理解了数据布局之后，就可以比较容易的去理解 Linux 中的很多 ext3 算法、策略了。

EXT3 将整个磁盘空间划分成多个块组，每个块组都采用元数据信息对其进行描述。块组内的具体格式可以描述如下：



从上图我们可以看出一个块组内的基本信息包括：

1) Superblock（超级块）。Superblock 描述了整个文件系统的信息。为了保证可靠性，可以在每个块组中对 superblock 进行备份。为了避免 superblock 冗余过多，可以采用稀疏存储的方式，即在若干个块组中对 superblock 进行保存，而不需要在所有的块组中都进行备份。
2) 组描述符表。组描述符表对整个组内的数据布局进行了描述。例如，数据块位图的起始地址是多少？inode 位图的起始地址是多少？inode 表的起始地址是多少？块组中还有多少空闲块资源等。组描述符表在 superblock 的后面。
3) 数据块位图。数据块位图描述了块组内数据块的使用情况。如果该数据块已经被某个文件使用，那么位图中的对应位会被置 1，否则该位为 0。
4) Inode 位图。Inode 位图描述了块组内 inode 资源使用情况。如果一个 inode 资源已经使用，那么对应位会被置 1。
5) Inode 表（即 inode 资源）和数据块。这两块占据了块组内的绝大部分空间，特别是数据块资源。

通过上述分析，我们了解了 EXT3 在磁盘上的数据布局。而一个文件是由 inode 进行描述的。一个文件占用的数据块是通过 inode 管理起来的。在 inode 结构中保存了直接块指针、一级间接块指针、二级间接块指针和三级间接块指针。对于一个小文件，直接可以采用直接块指针实现对文件块的访问。对于一个大文件，需要采用间接块指针实现对文件块的访问。

在理解数据布局之后，想要实现 EXT3 文件系统并非难事了。最主要的问题是结合 Linux 提供的页缓存机制实现 page cache，另外通过设备的 writeback 机制同步 dirty page。

数据回写

对于 EXT3 文件系统而言，在绝大多数情况下，IO 请求走到 page cache 之后就认为这个 IO 处理已经完成了。用户的 IO 请求被放入 Cache 之后，用户操作结束。实际上，此时的 IO 处境非常的危险，如果系统在此时 Crash，那么内存中缓存的数据将会彻底丢失。为了保证数据不丢失，需要有一种机制及时的将缓存中的数据同步（回写）到磁盘。对于 2.6.23 版本的 Linux，采用了 Pdflush 的数据同步机制；在 3.2 版本的 Linux 中，采用了 writeback 的数据同步机制。这种机制的变革主要考虑了系统性能提升方面的因素。关于这两种机制的对比可以参考文章《Linux 3.2 中回写机制的变革》。

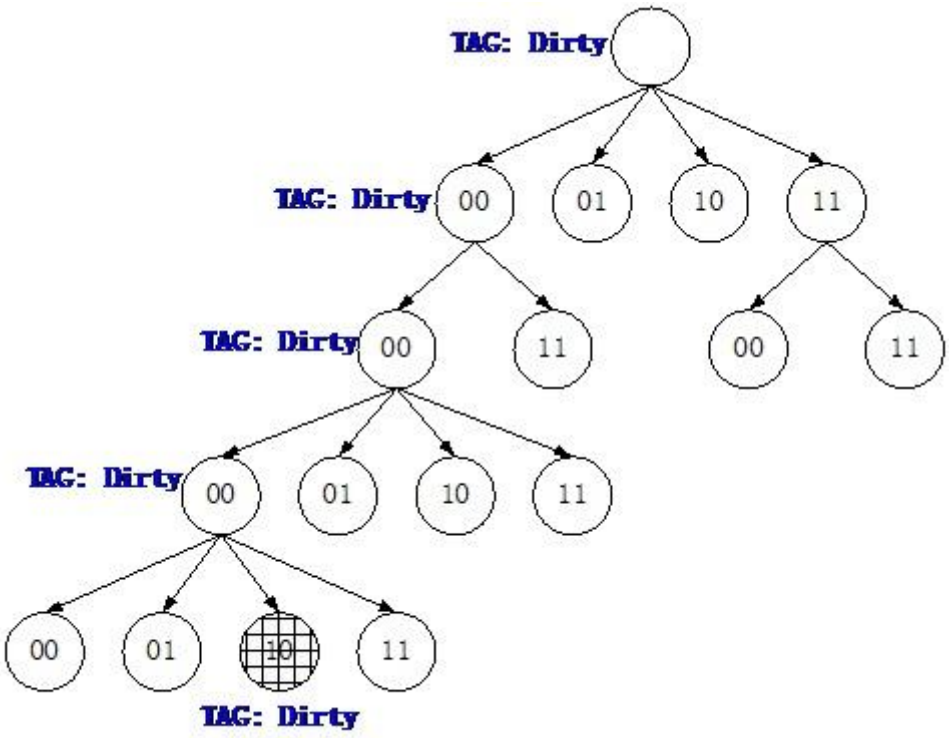
在 Linux3.2 中，当一个 IO 数据被写入 Page 缓存之后，EXT3 文件系统会调用 ext3_ordered_write_end 函数（注意，在 Linux2.6.23 中调用 aops->commit_write，会有所差别）结束整个过程。在该函数中主要完成 page 页 dirty 标志位的设置、日志信息的清除工作。其中，设置 page 页为 dirty 的操作就是用来唤醒（或者创建）一个 writeback 线程去处理缓存中的脏数据。设置 page 页 dirty 标志的核心函数是__set_page_dirty（调用关系为：

ext3_ordered_write_end→block_write_end→__block_commit_write→mark_buffer_dirty→__set_page_dirty）。__set_page_dirty 函数说明如下：

```
1  static void __set_page_dirty(struct page *page,
2  struct address_space *mapping, int warn)
3  {
4  spin_lock_irq(&mapping->tree_lock);
5  if (page->mapping) { /* Race with truncate? */
6  WARN_ON_ONCE(warn && !PageUptodate(page));
7  account_page_dirtied(page, mapping);
8  /* 设置radix tree中的Tag标志 */
9  radix_tree_tag_set(&mapping->page_tree,
10 page_index(page), PAGECACHE_TAG_DIRTY);
11 }
12 spin_unlock_irq(&mapping->tree_lock);
13 /* 将inode加入到writeback线程处理的事务队列中 */
14 __mark_inode_dirty(mapping->host, I_DIRTY_PAGES);
15 }
```

在__set_page_dirty 函数中主要完成了两件事情：

1) 在 radix tree 中设置需要回写的 page 页为 dirty。我们知道 Linux 为了加速 dirty 页的检索，在 radix tree 中采用了 tag 机制。Tag 本质上就是一种 flag，radix tree 的每层都有 tag，上层的 tag 信息是下层信息的汇总。所以，如果 radix tree 中没有脏页，那么最顶层的 tag 就不会被标识成 dirty。如果 radix tree 中有脏页，脏页所在的那条分支 tag 才会出现 dirty flag。显然，采用这种方法可以加速脏页的检索。如下图所示，如果 0x00000010 地址所在的 page 页为脏，那么其所在的访问路径会被标识成 Dirty，而其他的路径不受影响。因此，文件系统在查询脏页的时候，会节省很多的检索时间。



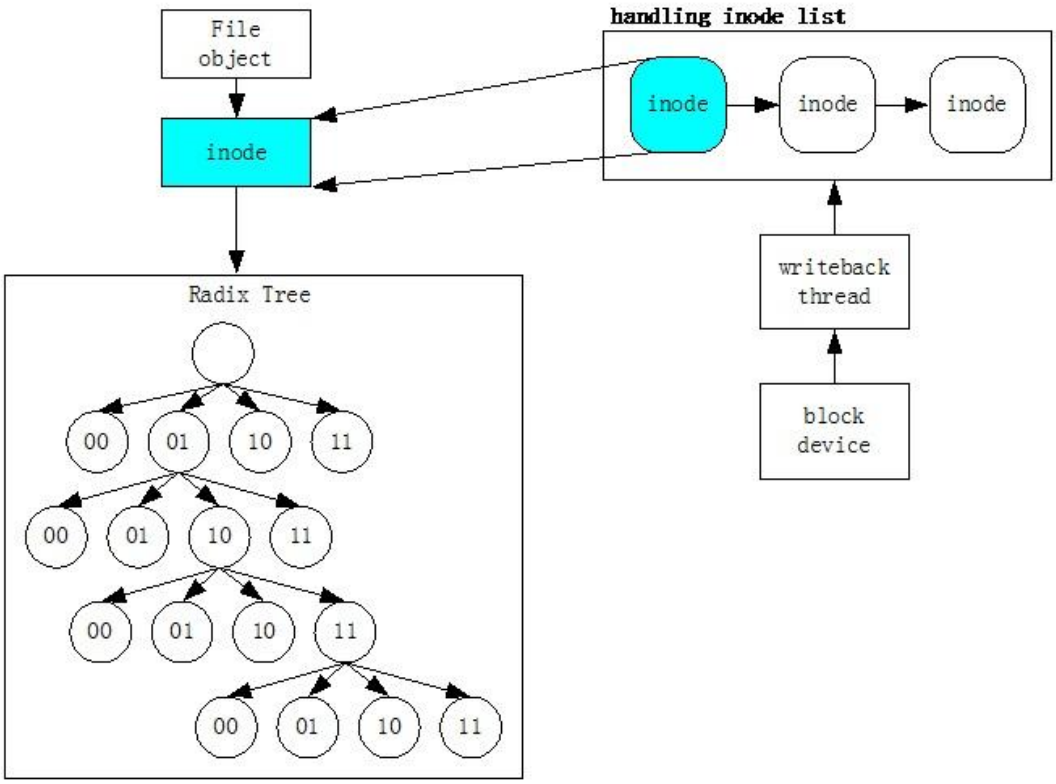
2) 将文件 inode 标识成 dirty。通过__mark_inode_dirty 函数将文件 inode 设置成脏，并且

将该 inode 交给回写线程进行处理。通知回写线程的处理方式比较简单，直接将该 inode 挂载到 writeback 线程调度处理的 inode 链表中。我们知道，每个设备都会有一个 writeback 线程处理脏页回写过程，每个 writeback 都会维护一条需要处理的 inode 链表。当 writeback 线程被唤醒之后，其会从 inode 链表中获取需要处理的 inode，并且从该 inode 所在的 radix 树中获取脏页，然后生成 IO 将数据写入磁盘。对于 EXT3 文件系统而言，其一定会架构在一个块设备之上，因此，在 mount 文件系统的时候，会将底层块设备的 writeback 对象（bdi）告诉给 EXT3 文件系统的 root_inode。这样文件系统内部的 inode 需要进行回写数据时，直接将该 inode 设置成 dirty，然后通过 root_inode 获取 writeback，并且将需要处理的 inode 挂载到任务链表中，最后唤醒 writeback 线程进行数据回写操作。

EXT3 所在设备的 writeback 线程被唤醒之后会将文件中的 dirty page 页刷新到磁盘。Linux 中处理完成该功能的函数是 do_writepages（writeback_single_inode→ do_writepages）。由于 EXT3 文件系统没有注册自己的 writepages 方法，因此，直接调用系统提供的 generic_writepages 处理脏页。刷新处理一个 inode 中 dirty page 的核心函数是 write_cache_pages。该函数的主要功能是搜索 inode 对应 radix tree 中的脏页，然后调用 EXT3 的 writepage 方法将脏页中的数据同步到磁盘。在刷新数据的过程中与块设备层相接口的的方法是 submit_bh。每个 page 页都会对应一个 buffer head，EXT3 想要刷新数据的时候直接调用 submit_bh 将 page 页中生成若干个 bio，然后转发至底层物理设备。至此，IO 旅程从文件系统开始转向块设备层。

通过上述分析，我们可以了解到一个 EXT3 写操作基本可以分成如下两个步骤：

1） 将数据写入 page cache，每个文件都会采用一个 radix tree 对 page 页进行高效检索管理。当数据被写入 page 页之后，需要将该页标识成 dirty，说明该页中有新的数据需要刷新到磁盘。为了提高 radix tree 检索 dirty page 的性能，Linux 采用了 Tag 机制。当 page 页被标识成 dirty 之后，需要将该 inode 加入到对应的 writeback 线程任务处理队列中，等待 writeback 线程调度处理。Radix tree 和 writeback 之间的关系如下图所示。



2） 每个块设备都会有一个 writeback 内核线程处理 page cache/buffer 的回写任务。当该线程被调度后，会检索对应 inode 的 radix tree，获取所有的脏页，然后调用块设备接口将脏数据回写到磁盘。

通过上述两大步骤，最常见的 EXT3 文件写操作完成。一个 IO 从用户态通过系统调用进入内核，然后在 radix tree 上小住了一段时间，即将踏上新的征程——块设备层。

块设备 buffer cache 机制

在 EXT3 文件 IO 踏上新的征程之前，需要介绍一位 EXT3 文件 IO 的同伴，他们即将踏上相同的旅程。只不过这位同伴没有经历过 EXT3 文件系统的精彩，却领略了另外一番略有差别的风情。这位同伴是在块设备写操作时创建诞生的，我们可以称它为块设备 IO。

在很多应用中都会直接进行块设备操作，我们常用的命令 dd 就可以进行块设备的读写。例如：dd if=/dev/sda of=./abc bs=512 count=1 命令可以实现将/dev/sda 设备的第一个扇区读入到当前目录的 abc 文件中。读到这里我们想一下：访问一个块设备文件和一个 EXT3 文件到底有何本质上的区别呢？说到区别还是可以列举一二的：

1） EXT3 是一个通用的文件系统，数据在磁盘上的分布是采用元数据的形式进行描述的。所以，对于一个 EXT3 文件的读写操作会涉及到元数据和文件数据两种类型。由于这两种数据类型存在相关性，所以为了保证两者之间操作的原子性，EXT3 通常会采用日志的方式保证操作的原子性。

2） 块设备没有 EXT3 文件系统那么复杂，写入的数据直接在磁盘存储，不存在任何的元数据信息。

所以，比较块设备和 EXT3 文件系统，数据的读写方式存在差别。两者之间一个很大的共同

点是都存在磁盘访问的性能问题，都可以采用内存对磁盘进行性能优化。EXT3 采用 page cache 对 IO 性能进行优化，块设备同样可以采用 page cache 对其进行性能优化。前面我们已经了解到，每个 EXT3 文件都有一棵 radix tree 用于维护这个文件内容的 page cache，而裸设备可以等价成一个 EXT3 文件，同样可以采用一棵 radix tree 对块设备上的数据进行 cache 维护。所以，在这两者之间是有很大共同点的。

正因为如此，Linux 在实现块设备 IO 操作的时候和 EXT3 是类似的，对于块设备访问的这个子系统可以称之为 bdev 文件系统。VFS 实现了所有类型文件访问的功能，应用程序所调用的 API 是完全相同的。对于块设备的访问，穿过 VFS 层会调用 bdev 文件系统提供的相关函数。

在初始化块设备的时候，会调用 init_special_inode 函数初始化这个块设备的 inode：

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
inode->i_mode = mode;
if (S_ISCHR(mode)) {
/* 初始化字符设备操作方法 */
inode->i_fop = &def_chr_fops;
inode->i_rdev = rdev;
} else if (S_ISBLK(mode)) {
/* 初始化块设备操作方法 */
inode->i_fop = &def_blk_fops;
inode->i_rdev = rdev;
} else if (S_ISFIFO(mode))
inode->i_fop = &def_fifo_fops;
else if (S_ISSOCK(mode))
inode->i_fop = &bad_sock_fops;
else
printk(KERN_DEBUG "init_special_inode: bogus i_mode (%o) for"
" inode %s:%lu\n", mode, inode->i_sb->s_id,
inode->i_ino);
}
```

在用户程序调用 open 函数打开指定块设备的时候，会初始化一个 file 对象，并且采用上述 inode 对 file 对象进行初始化。因此，通过 file 对象的文件操作方法就可以调用通用的块设备操作函数。Linux 中定义的通用块设备操作函数说明如下：

```
const struct file_operations def_blk_fops = {
.open        = blkdev_open,
.release     = blkdev_close,
.llseek      = block_llseek,
.read        = do_sync_read,
.write       = do_sync_write,
.aio_read    = generic_file_aio_read,
.aio_write   = blkdev_aio_write,
.mmap        = generic_file_mmap,
.fsync       = blkdev_fsync,
.unlocked_ioctl = block_ioctl,
#ifdef CONFIG_COMPAT
.compat_ioctl   = compat_blkdev_ioctl,
#endif
.splice_read    = generic_file_splice_read,
.splice_write   = generic_file_splice_write,
};
```

因此，对于一个块设备的写操作，通过 write 函数陷入内核执行 do_sync_write。在

do_sync_write 函数中会调用 blkdev_aio_write 函数实现块设备的异步写入过程。blkdev_aio_write 函数的核心是__generic_file_aio_write。到此为止，我们发现块设备的所有调用函数和 EXT3 文件系统基本一样。前面已经分析过，__generic_file_aio_write 分成两种情况：Direct_IO 和 Buffer_IO。对于 buffer_IO 分成关键的几个步骤：

1） write_begin

2） copy buffer

3） write_end

write_begin 和 write_end 函数是每类文件系统具体相关的方法，对于 bdev 文件系统，其定义为：

```
static const struct address_space_operations def_blk_aops = {
.readpage   = blkdev_readpage,
.writepage  = blkdev_writepage,
.write_begin    = blkdev_write_begin,
.write_end  = blkdev_write_end,
.writepages = generic_writepages,
.releasepage    = blkdev_releasepage,
.direct_IO  = blkdev_direct_IO,
};
```

对于 EXT3 文件而言，write_begin 会进行日志操作，块设备文件系统没有这种操作，只会进行一些 page 页初始化方面的工作。对于 write_end 函数，EXT3 文件系统会清除日志，并且需要通知 writeback 守护线程去回写数据。对于块设备文件系统而言，write_end 函数的主要工作就是将 page 页标识成脏，然后通知回写线程去处理这个块设备中的脏页。实现设置脏页的函数说明如下：

```
static int __block_commit_write(struct inode *inode, struct page *page,
unsigned from, unsigned to)
{
unsigned block_start, block_end;
int partial = 0;
unsigned blocksize;
struct buffer_head *bh, *head;
blocksize = 1 << inode->i_blkbits;
for(bh = head = page_buffers(page), block_start = 0;
bh != head || !block_start;
block_start=block_end, bh = bh->b_this_page) {
block_end = block_start + blocksize;
if (block_end <= from || block_start >= to) {
if (!buffer_uptodate(bh))
partial = 1;
} else {
set_buffer_uptodate(bh);
/*将page页和inode设置成脏，等待回写调度处理*/
mark_buffer_dirty(bh);
}
clear_buffer_new(bh);
}
/*
* If this is a partial write which happened to make all buffers
* uptodate then we can optimize away a bogus readpage() for
* the next read(). Here we 'discover' whether the page went
* uptodate as a result of this (potentially partial) write.
*/
if (!partial)
SetPageUptodate(page);
return 0;
}
```

从整个分析来看，裸块设备的写操作中 Cache 的机制、原理和 EXT3 并没有什么本质上的区别。大家都采用了 radix tree 管理的 page cache。如果不采用 Direct_IO 的方式，那么都会首先将数据写入 page cache，然后再通过 writeback 机制将数据回写到磁盘。整个机制是完全相同的。所不同的是，EXT3 和块设备的缓存块大小是不相同的。对于 EXT3 而言，缓存块大小就是 page size，对于块设备而言，缓存块大小会采用一定的策略得到。具体关于 buffer cache 的缓存块大小参考《Linux 中 Buffer cache 性能问题一探究竟》。
虽然块设备和 EXT3 文件看上去差别很大，但是，由于系统所要解决的问题基本类似，因此，在 IO 处理的机制上是类似的。好！言归正传，到目前为止，EXT3 文件 IO 以及块设备 IO 都已经准备完毕，writeback 回写机制已经已经将这些 IO 统统回写到底层设备中。这些 IO 都将离开短暂的 page cache，一同踏上块设备层，即将面临块设备层即公平又难以公平的调度处理过程。

块设备层分析

无论是经过 EXT3 文件系统还是块设备文件,最终都要通过 writeback 机制将数据刷新到磁盘，除非用户在对文件进行读写的时候采用了 DirectIO 的方式。为了提高性能，文件系统或者是裸设备都会采用 Linux 的 cache 机制对数据读写性能进行优化，因此都会采用 writeback 回写机制将数据写入磁盘。
通过上述分析我们已经知道，writeback 机制会调用不同底层设备的 address_space_operations 函数将数据刷新到设备。例如，EXT3 文件系统会调用 blkdev_writepage 函数将 radix tree 中的 page 页写入设备。在确认需要将一个 page 页写入设备时，最终都需要调用 submit_bh 函数，该函数描述如下：

```c
int submit_bh(int rw, struct buffer_head * bh)
{
struct bio *bio;
int ret = 0;
BUG_ON(!buffer_locked(bh));
BUG_ON(!buffer_mapped(bh));
BUG_ON(!bh->b_end_io);
BUG_ON(buffer_delay(bh));
BUG_ON(buffer_unwritten(bh));
/*
* Only clear out a write error when rewriting
*/
if (test_set_buffer_req(bh) && (rw & WRITE))
clear_buffer_write_io_error(bh);
/*
* from here on down, it's all bio -- do the initial mapping,
* submit_bio -> generic_make_request may further map this bio around
*/
bio = bio_alloc(GFP_NOIO, 1);
/* 封装BIO */
bio->bi_sector = bh->b_blocknr * (bh->b_size >> 9);  /* 起始地址 */
bio->bi_bdev = bh->b_bdev;                /* 访问设备 */
bio->bi_io_vec[0].bv_page = bh->b_page;       /* 数据buffer地址 */
bio->bi_io_vec[0].bv_len = bh->b_size;            /* 数据段大小 */
bio->bi_io_vec[0].bv_offset = bh_offset(bh);     /* 数据在buffer中的offset */
bio->bi_vcnt = 1;
bio->bi_idx = 0;
bio->bi_size = bh->b_size;
/* 设定回调函数 */
bio->bi_end_io = end_bio_bh_io_sync;
bio->bi_private = bh;
bio_get(bio);
/* 提交BIO至对应设备，让该设备对应的驱动程序进行进一步处理 */
submit_bio(rw, bio);
if (bio_flagged(bio, BIO_EOPNOTSUPP))
ret = -EOPNOTSUPP;
bio_put(bio);
return ret;
}
```

Submit_bh 函数的主要任务是为 page 页分配一个 bio 对象，并且对其进行初始化，然后将 bio 提交给对应的块设备对象。提交给块设备的行为其实就是让对应的块设备驱动程序对其进行处理。在 Linux 中，每个块设备在内核中都会采用 bdev（block_device）对象进行描述。通过 bdev 对象可以获取块设备的所有所需资源，包括如何处理发送到该设备的 IO 方法。因此，在初始化 bio 的时候，需要设备目标 bdev，在 Linux 的请求转发层需要用到 bdev 对象对 bio 进行转发处理。

在通用块设备层，提供了一个非常重要的 bio 处理函数 generic_make_request，通过这个函数实现 bio 的转发处理，该函数的实现如下：

```
void generic_make_request(struct bio *bio)
{
struct bio_list bio_list_on_stack;
if (!generic_make_request_checks(bio))
return;
/*
* We only want one ->make_request_fn to be active at a time, else
* stack usage with stacked devices could be a problem.  So use
* current->bio_list to keep a list of requests submited by a
* make_request_fn function.  current->bio_list is also used as a
* flag to say if generic_make_request is currently active in this
* task or not.  If it is NULL, then no make_request is active.  If
* it is non-NULL, then a make_request is active, and new requests
* should be added at the tail
*/
if (current->bio_list) {
bio_list_add(current->bio_list, bio);
return;
}
/* following loop may be a bit non-obvious, and so deserves some
* explanation.
* Before entering the loop, bio->bi_next is NULL (as all callers
* ensure that) so we have a list with a single bio.
* We pretend that we have just taken it off a longer list, so
* we assign bio_list to a pointer to the bio_list_on_stack,
* thus initialising the bio_list of new bios to be
* added.  ->make_request() may indeed add some more bios
* through a recursive call to generic_make_request.  If it
* did, we find a non-NULL value in bio_list and re-enter the loop
* from the top.  In this case we really did just take the bio
* of the top of the list (no pretending) and so remove it from
* bio_list, and call into ->make_request() again.
*/
BUG_ON(bio->bi_next);
bio_list_init(&bio_list_on_stack);
current->bio_list = &bio_list_on_stack;
do {
/* 获取块设备的请求队列 */
struct request_queue *q = bdev_get_queue(bio->bi_bdev);
/* 调用对应驱动程序的处理函数 */
q->make_request_fn(q, bio);
bio = bio_list_pop(current->bio_list);
} while (bio);
current->bio_list = NULL; /* deactivate */
}
```

在 generic_make_request 函数中，最主要的操作是获取请求队列，然后调用 make_request_fn 方法处理 bio。在 Linux 中一个块设备驱动通常可以分成两大类：有 queue 和无 queue。有 queue 的块设备就是驱动程序提供了一个请求队列，make_request_fn 方法会将 bio 放入请求队列中进行调度处理，调度处理的方法有 CFQ、Deadline 和 Noop 之分。设置请求队列的目的是考虑了磁盘介质的特性，普通磁盘介质一个最大的问题是随机读写性能很差。为了提高性能，通常的做法是聚合 IO，因此在块设备层设置请求队列，对 IO 进行聚合操作，从而提高读写性能。关于 IO scheduler 的具体算法分析请见后续文章。

在 Linux 的通用块层，提供了一个通用的请求队列压栈方法：blk_queue_bio，在老版本的 Linux 中为 __make_request。在初始化一个有 queue 块设备驱动的时候，最终都会调用 blk_init_allocated_queue 函数对请求队列进行初始化，初始化的时候会将 blk_queue_bio 方法注册到 q->make_request_fn。在 generic_make_request 转发 bio 请求的时候会调用

q->make_request_fn，从而可以将 bio 压入请求队列进行 IO 调度。一旦 bio 进入请求队列之后，可以好好的休息一番，直到 unplug 机制对 bio 进行进一步处理。
另一类块设备是无 queue 的。无 queue 的块设备我们通常可以认为是一种块设备过滤驱动，这类驱动程序可以自己实现请求队列，绝大多数是没有请求队列的，直接对 bio 进行转发处理。这类驱动程序一个很重要的特征是需要自己实现 q->make_request_fn 方法。这类驱动的 make_request_fn 方法通常可以分成如下几个步骤：

1） 根据一定规则切分 bio，不同的块设备可能存在不同的块边界，因此，需要对请求 bio 进行边界对齐操作。
2） 找到需要转发的底层块设备对象。
3） 直接调用 generic_make_request 函数转发 bio 至目标设备。

因此，无 queue 的块设备处理过程很直观。其最重要的作用是转发 bio。在 Linux 中，device_mapper 机制就是用来转发 bio 的一种框架，如果需要开发 bio 转发处理的驱动程序，可以在 device_mapper 框架下开发一个 target，从而快速实现一个块设备驱动。
通过上述描述，我们知道，IO 通过 writeback 或者 DirectIO 的方式可以抵达块设备层。到了块设备层之后遇到了两类块设备处理方法。如果遇到无 queue 块设备类型，bio 马上被转发到其他底层设备；如果遇到了有 queue 块设备类型，bio 会被压入请求队列，进行合并处理，等待 unplug 机制的调度处理。IO 曾经在 page cache 游玩了很长时间，大家都很高兴。所有得请求在 page cache 受到得待遇是相同的，大家都会比较公平得被调度走，继续下面的旅程。但是，在块设备层情况就变的复杂了，不同 IO 受到的待遇会有所不同，这就需要看请求队列中的 io scheduler 具体算法了。因此，IO 旅程在块设备这一站，最为重要的核心就是 io scheduler。

IO 调度器

当 IO 旅行到调度器的时候，发现自己受到的待遇竟然很不一样，有些 IO 倚仗着特权很快就放行了；有些 IO 迟迟得不到处理，甚至在有些系统中居然饿死！面对这样的现状，IO 显然是很不高兴的，凭什么别人就能被很快送到下一个旅程，自己需要在调度器上耗费青春年华？这里是拼爹的时代，人家出身好，人家是读请求，人家就可以很快得到资源。咱们是写请求，出生贫寒，只能等着，但也不会让你饿死。这就是我们常见的 deadline 策略。在更加糟糕的地方，deadline 都没有，拼的是家族血脉关系，相邻的 IO 可以很快处理掉，其他的等着吧，那就会出现饿死的情况。这就是我们常说的 noop 策略，其实就是 Linus 电梯。在文明一点的社会，大家会比较公平，从应用的整体来看，大家会享有相同的 IO 带宽，但是，从 IO 的个体来看，公平性还是没有的。这个社会没有绝对的公平，只要保证所有家庭的公平性，那么社会就会比较和谐。当然，我们发现有些家庭（应用）不是特别合群，我们也可以对其进行惩罚，IO 带宽的分配就会对其进行缩减。这就是我们常见的 CFQ 策略。在 IO 调度器层，可以有很多的策略，不同的系统可以定义不同的策略，目的都是在与更好的聚合 IO，并且对不同的应用进行 QOS 控制。
在 Linux 系统中，可以注册自己的调度算法，如果不注册自己的调度器，那么可以采用上述提到的三种调度器之一。其中，deadline 是在 Linus 电梯的基础上发展起来的，其对读写请求进行了有区别的调度，还会考虑到 IO 饥饿的情况。最为传统的调度器不能规避 IO 饥饿问题。CFQ 调度器考虑了应用的公平性，在很多情况下可以得到最佳性能，有关于这三种调度器的设计比较会在下面篇章中详细阐述。
当 IO 请求通过 generic_make_request 进行转发时，如果被访问的设备是一个有 queue 的块

设备，那么系统会调用 blk_queue_bio 函数进行 bio 的调度合并。blk_queue_bio 函数说明如下：

```
void blk_queue_bio(struct request_queue *q, struct bio *bio)
{
const bool sync = !!(bio->bi_rw & REQ_SYNC);
struct blk_plug *plug;
int el_ret, rw_flags, where = ELEVATOR_INSERT_SORT;
struct request *req;
unsigned int request_count = 0;
/*
* low level driver can indicate that it wants pages above a
* certain limit bounced to low memory (ie for highmem, or even
* ISA dma in theory)
*/
blk_queue_bounce(q, &bio);
if (bio->bi_rw & (REQ_FLUSH | REQ_FUA)) {
spin_lock_irq(q->queue_lock);
where = ELEVATOR_INSERT_FLUSH;
goto get_rq;
}
/*
* Check if we can merge with the plugged list before grabbing
* any locks.
*/
/* 尝试将bio合并到当前plugged的请求队列中 */
if (attempt_plug_merge(q, bio, &request_count))
return;
spin_lock_irq(q->queue_lock);
/* elv_merge是核心函数，找到bio前向或者后向合并的请求 */
el_ret = elv_merge(q, &req, bio);
if (el_ret == ELEVATOR_BACK_MERGE) {
/* 进行后向合并操作 */
if (bio_attempt_back_merge(q, req, bio)) {
if (!attempt_back_merge(q, req))
elv_merged_request(q, req, el_ret);
goto out_unlock;
}
} else if (el_ret == ELEVATOR_FRONT_MERGE) {
/* 进行前向合并操作 */
if (bio_attempt_front_merge(q, req, bio)) {
if (!attempt_front_merge(q, req))
elv_merged_request(q, req, el_ret);
goto out_unlock;
}
}
/* 无法找到对应的请求实现合并 */
get_rq:
/*
* This sync check and mask will be re-done in init_request_from_bio(),
* but we need to set it earlier to expose the sync flag to the
* rq allocator and io schedulers.
*/
rw_flags = bio_data_dir(bio);
if (sync)
rw_flags |= REQ_SYNC;
/*
* Grab a free request. This is might sleep but can not fail.
* Returns with the queue unlocked.
*/
/* 获取一个empty request请求 */
req = get_request_wait(q, rw_flags, bio);
if (unlikely(!req)) {
bio_endio(bio, -ENODEV);    /* @q is dead */
goto out_unlock;
}
/*
* After dropping the lock and possibly sleeping here, our request
* may now be mergeable after it had proven unmergeable (above).
* We don't worry about that case for efficiency. It won't happen
* often, and the elevators are able to handle it.
*/
/* 采用bio对request请求进行初始化 */
init_request_from_bio(req, bio);
if (test_bit(QUEUE_FLAG_SAME_COMP, &q->queue_flags))
req->cpu = raw_smp_processor_id();
plug = current->plug;
if (plug) {
/*
* If this is the first request added after a plug, fire
* of a plug trace. If others have been added before, check
* if we have multiple devices in this plug. If so, make a
* note to sort the list before dispatch.
*/
if (list_empty(&plug->list))
trace_block_plug(q);
else {
if (!plug->should_sort) {
struct request *__rq;
__rq = list_entry_rq(plug->list.prev);
if (__rq->q != q)
plug->should_sort = 1;
}
if (request_count >= BLK_MAX_REQUEST_COUNT) {
/* 请求数量达到队列上限值，进行unplug操作 */
blk_flush_plug_list(plug, false);
trace_block_plug(q);
}
}
/* 将请求加入到队列 */
list_add_tail(&req->queuelist, &plug->list);
drive_stat_acct(req, 1);
} else {
/* 在新的内核中，如果用户没有调用start_unplug，那么，在IO scheduler中是没有合并的，一旦加入到request queue中
spin_lock_irq(q->queue_lock);
/* 将request加入到调度器中 */
add_acct_request(q, req, where);
/* 调用底层函数执行unplug操作 */
__blk_run_queue(q);
out_unlock:
spin_unlock_irq(q->queue_lock);
}
}
```

对于 blk_queue_bio 函数主要做了三件事情：

1） 进行请求的后向合并操作

2） 进行请求的前向合并操作

3） 如果无法合并请求，那么为 bio 创建一个 request，然后进行调度

在 bio 合并过程中，最为关键的函数是 elv_merge。该函数主要工作是判断 bio 是否可以进行后向合并或者前向合并。对于所有的调度器，后向合并的逻辑都是相同的。在系统中维护了一个 request hash 表，然后通过 bio 请求的起始地址进行 hash 寻址。Hash 表的生成原理比较简单，就是将所有 request 的尾部地址进行分类，分成几大区间，然后通过 hash 函数可以寻址这几大区间。Hash 函数是：

hash_long(ELV_HASH_BLOCK((sec)), elv_hash_shift)

一旦通过 hash 函数找到所有位于这个区间的 request 之后，通过遍历的方式匹配到所需要的 request。具体该过程的实现函数如下：

```
static struct request *elv_rqhash_find(struct request_queue *q, sector_t offset)
{
struct elevator_queue *e = q->elevator;
/* 通过hash函数找到区间内的所有request */
struct hlist_head *hash_list = &e->hash[ELV_HASH_FN(offset)];
struct hlist_node *entry, *next;
struct request *rq;
/* 遍历地址区间内的所有request */
hlist_for_each_entry_safe(rq, entry, next, hash_list, hash) {
BUG_ON(!ELV_ON_HASH(rq));
if (unlikely(!rq_mergeable(rq))) {
__elv_rqhash_del(rq);
continue;
}
/* 如果地址匹配，那么找到所需的request */
if (rq_hash_key(rq) == offset)
return rq;
}
return NULL;
}
```

采用 hash 方式维护 request，有一点需要注意：当一个 request 进行合并处理之后，需要对该 request 在 hash 表中进行重新定位。这主要是因为 request 的尾地址发生了变化，有可能会超过一个 hash 区间的范围。

如果后向合并失败，那么调度器会尝试前向合并。不是所有的调度器支持前向合并，如果调度器支持这种方式，那么需要注册 elevator_merge_fn 函数实现前向调度功能。例如 deadline 算法采用了红黑树的方式实现前向调度。如果前向调度无法完成合并。那么调度器认为该合并失败，需要产生一个新的 request，并且采用现有 bio 对其进行初始化，然后加入到 request queue 中进行调度处理。

当 IO 利用 generic_make_request 来到块设备层之后，对其进行处理的重要函数 blk_queue_bio 主要任务是合并 IO。由于不同的调度器有不同的合并方法、IO 分类方法，所以，具体调度器的算法会采用函数注册的方式实现。blk_queue_bio 仅仅是一个上层函数，最主要完成后向合并、调用调度器方法进行前向合并以及初始化 request 准备调度。

IO 调度器设计考虑

通过前面的分析已经知道 IO 调度器主要是为了解决临近 IO 合并的问题。磁介质存储盘最大的性能瓶颈在于寻道。当用户访问一个指定地址时，磁盘首先需要进行寻道操作，找到访问地址所属的区域。这种操作往往是毫秒级别的，相对于性能不断提升的 CPU 而言，这种性能显然是不可接受的。所以，磁盘最大的问题在于机械操作引入的寻道时间过长，对外表现

就是随机读写性能太差了。
为了弥补这种性能弱点，Linux 操作系统系统在设计的时候引入了 IO 调度器，尽最大可能将随机读写转换成大块顺序读写，减少磁盘抖动，降低若干 IO 操作之间的寻道时间。IO 调度器的目的就在于此，它的初衷是面向存储介质设计的。因此，当一个系统的存储介质发生变化之后，IO 调度器就需要改变。例如，针对 SSD 存储介质，传统意义上的 IO 调度算法就不再适用了。SSD 不存在机械操作，不存在漫长的寻道时间问题，因此，不存在传统磁介质的随机读写问题。但是，SSD 存在写放大的问题，一个小写会引入大量的数据读写操作，从而使得 IO 性能下降。所以，传统磁盘的 IO 调度器在 SSD 面前就没有价值了，可以采用最简单的 Noop 调度器对 IO 进行后向合并就可以了，而写放大等问题往往都在 SSD 内部的 Firmware 解决了。
考虑一下，如果设计的 IO 调度器完全面向存储介质，那么设计的调度算法只需要考虑 IO 请求的前向/后向合并就可以了，这样的 IO 合并减少了磁头的抖动，就像一部电梯一样，一直往一个方向移动。这种算法看上去很完美，但是，在实践中我们会发现有些请求会长时间得不到服务，就像一辆电梯厂时间停在一个楼层，其他楼层的人长时间得不到服务。特别对于一些读操作，往往是同步请求，如果长时间得不到服务，那么会大大影响应用的性能。所以，仅仅简单的考虑存储介质的特性，进行 IO 的前向/后向合并是不够的，还需要考虑 IO 的本身属性。
从 IO 的属性来看，最需要区分的是读写请求，读写请求的优先级是不一样的，大多数写请求可以异步完成，读请求需要同步完成，所以，对于读写请求可以分开处理。虽然分开之后可能会引入更多的磁盘抖动，但是，应用的整体性能还是会提高。
考虑了 IO 的基本属性之外，是不是就够了呢？其实还是不够的。例如在一个服务器中存在多个应用，这些应用会访问相同的存储，有些应用读请求多，有些应用写请求多，如果仅仅考虑 IO 的基本属性，那么对于这些不同的应用就会表现出不同的 IO 性能。有些应用得到较多的 IO 带宽，性能较好；有些应用得到较少的带宽，性能较差。这显然是不合理的，由于调度算法的策略问题，导致一个系统中，不同应用具有不同的 IO 性能。为了解决这个问题，需要考虑应用属性。
在现有的 Linux 系统中，提供了多种 IO 调度器，这些调度器有些仅仅考虑了存储介质属性（Noop），有些考虑了 IO 属性（Deadline），还有的考虑了应用属性（cfq）。不同调度器的设计考虑范畴不同，所以复杂度也有很大差别。下面会对 Linux 中的这几种调度器进行阐述。

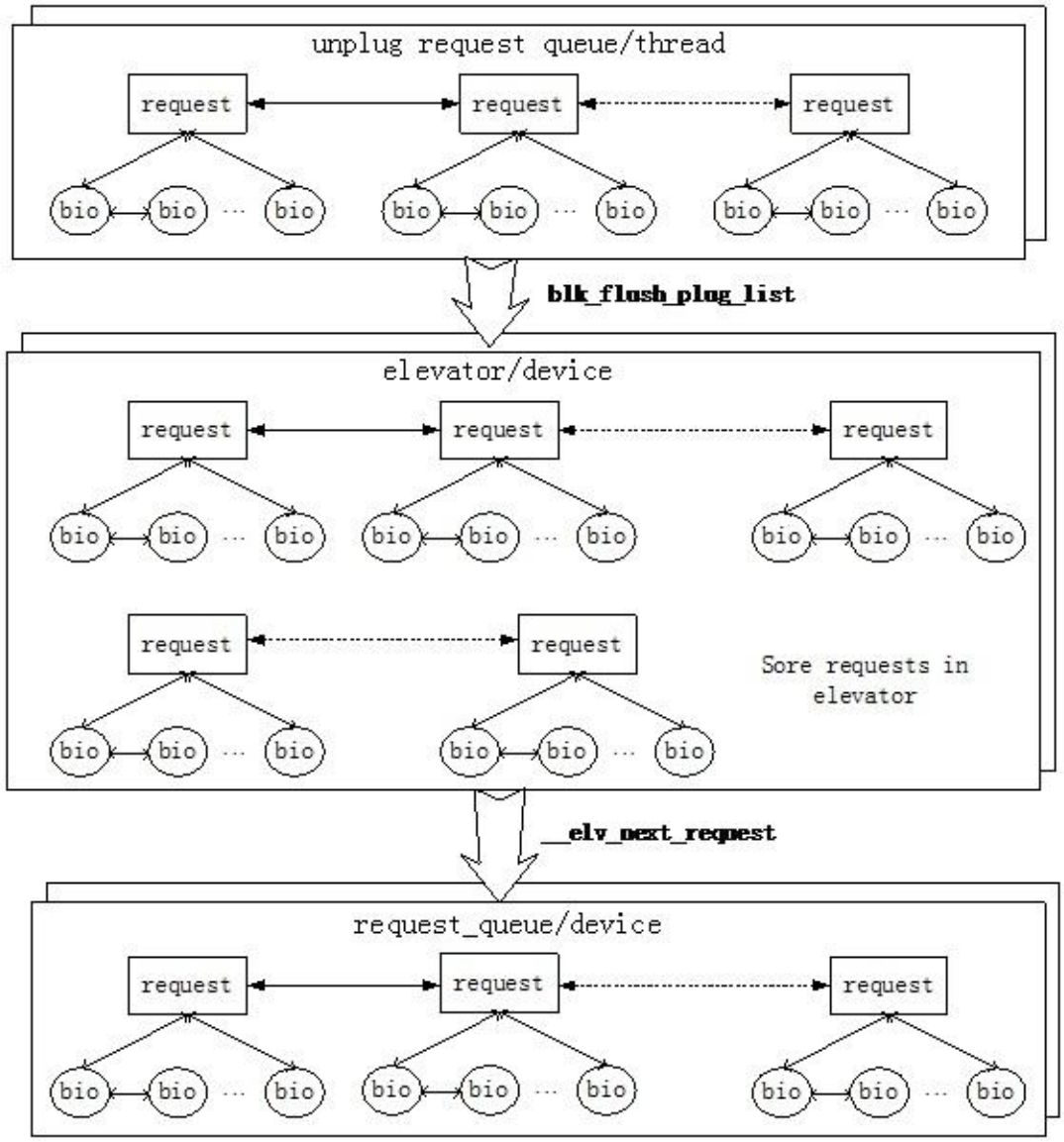
Elevator 子系统介绍

Elevator 子系统是 IO 路径上非常重要的组成部分，前面已经分析过，elevator 中实现了多种类型的调度器，用于满足不同应用的需求。那么，从整个 IO 路径的角度来看，elevator 这层主要解决 IO 的 QoS 问题，通常需要解决如下两大问题：

1） Bio 的合并问题。主要考虑 bio 是否可以和 scheduler 中的某个 request 进行合并。因为从磁盘的角度来看，临近的请求需要合并，所有的 IO 需要顺序化处理，这样磁头才能往一个方向运行，避免无规则的乱序运行。
2） Request 的调度问题。request 在何时可以从 scheduler 中取出，并且送入底层驱动程序继续进行处理？不同的应用可能需要不同的带宽资源，读写请求的带宽、延迟控制也可以不一样，因此，需要解决 request 的调度处理，从而可以更好的控制 IO 的 QoS。

通过上面分析，一个 IO 在经过块设备层处理之后，终于来到了 elevator 层。我们熟知，一个 request 在送往设备之前会被放入到每个设备所对应的 request queue。其实，通过分析一个 IO 在 elevator 层其实会经过很多 request queue，不同的 request queue 会有不同的作用。

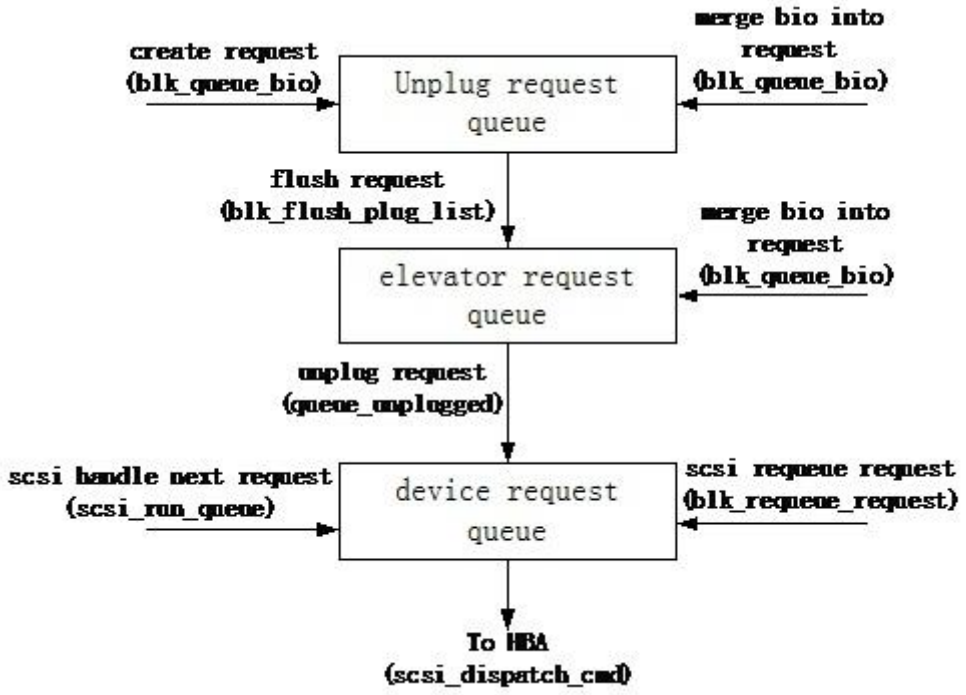
如下图所示，一个 IO 在经历很多层 queue 的调度处理之后，最后才能达到每个设备的 request queue。Linux 中各个 request queue 之间的关系如下图所示：



在 Linux-3.2 中，已经采用新的 unplug 机制对请求进行批量 unplug 处理，相对于 2.6.23 kernel 这是新的一层。在老 Kernel 中，没有这层 unplug 机制，request 请求可以直接进入 elevator，然后通过内核中的 unplug 定时器对 elevator 中的 request 进行 unplug 调度处理。在新 kernel 中，每个线程可以对自己的 request 进行 unplug 调度处理。例如，ext3 文件系统的 writeback 线程可以主动 unplug 自己的 request，这种 application awareness 的方法可以最大限度的减少请求处理的延迟时间。

从上图可以看出，一个 IO 请求首先进入每个线程域所在的 unplug 请求队列。如果这个线程没有 unplug 请求队列，那么 IO request 直接被送入 elevator。在 unplug 请求队列中等待的 request 会在请求 unplug 的过程中被送入 elevator 的请求队列。每个设备可以采用不同类型的 IO 调度方法，因此，在 elevator 中的 IO 分类方法会有所不同。这里 Elevator 的类型也就是我们通常所说的 Noop、deadline 以及 CFQ 方法。最后，Elevator 中的 request 会在一定的策略控制下被送入每个设备的 request queue。从这个结构中，我们可以看出，只要控制住了 elevator 的调度器，那么我们就可以控制每个设备 IO 的优先级，从而达到 IO QoS 的目的。

通过分析，我们已经知道 Request 在三类 request queue 中被调度处理，其主要处理时机点可以描述如下：



在一般的请求处理过程中，request 被创建并且会被挂载到 unplug request queue 中，然后通过 flush request 方法将 request 从 unplug request queue 中移至 elevator request queue 中。当一个新的 BIO 需要被处理时，其可以在 unplug request queue 或者 elevator request queue 中进行合并。当需要将请求发送到底层设备时，可以通过调用 run_queue 的方法将 elevator 分类处理过的 request 转移至 device request queue 中，最后调用 scsi_dispatch_cmd 方法将请求发送到 HBA。在这个过程有一些问题需要处理：底层设备可能存在故障；HBA 的处理队列是有长度限制的。因此，如何连续调度 device request queue 及重新调度 request 成了一个需要考虑的问题。在 Linux 中，如果 scsi 层需要重新调度一个 request，可以通过 blk_requeue_request 接口来完成。通过该接口，可以把 request 重新放回到 device request queue 中。另外，在一个 request 结束之后的回调函数中，需要通过 scsi_run_queue 函数来再次调度处理 device request queue 中的剩余请求，从而可以保证批量处理 device request queue 中的请求，HBA 也一直运行在最大的 queue depth 深度。

Elevator 层关键函数分析

Elv_merge

当一个 IO 离开块设备层，需要发送到底层设备时，首先需要判断该 IO 是否可以和正在等待处理的 request 进行合并。这一步主要是通过 elv_merge()函数来实现的，需要注意的是，在调用 elv_merge 进行合并操作之前，首先需要判断 unplug request queue 是否可以进行合并，如果不能合并，那么才调用 elv_merge 进行 elevator request queue 的合并操作。一旦 bio 找到了可以合并的 request，那么，这个 IO 就会合并放入对应的 request 中，否则需要创建一个新的 request，并且放入到 unplug request queue 中。

Elevator 层提供的 bio 合并函数分析如下：

```
int elv_merge(struct request_queue *q, struct request **req, struct bio *bio)
{
struct elevator_queue *e = q->elevator;
struct request *__rq;
int ret;
/*
* Levels of merges:
*  nomerges:  No merges at all attempted
*  noxmerges: Only simple one-hit cache try
*  merges:    All merge tries attempted
*/
if (blk_queue_nomerges(q))
return ELEVATOR_NO_MERGE;
/*
* First try one-hit cache.
* 尝试和最近的request进行合并
*/
if (q->last_merge) {
ret = elv_try_merge(q->last_merge, bio);
if (ret != ELEVATOR_NO_MERGE) {
/* 可以和last_merge进行合并 */
*req = q->last_merge;
return ret;
}
}
if (blk_queue_noxmerges(q))
return ELEVATOR_NO_MERGE;
/*
* See if our hash lookup can find a potential backmerge.
* 查找elevator中的后向合并的hash table，获取可以合并的request
*/
__rq = elv_rqhash_find(q, bio->bi_sector);
if (__rq && elv_rq_merge_ok(__rq, bio)) {
*req = __rq;
return ELEVATOR_BACK_MERGE;
}
/* 查找scheduler检查是否可以进行前向合并，如果可以，那么进行前向合并 */
if (e->ops->elevator_merge_fn)
return e->ops->elevator_merge_fn(q, req, bio);
return ELEVATOR_NO_MERGE;
}
```

__elv_add_request

需要将一个 request 加入到 request queue 中时，可以调用__elv_add_request 函数。通过该函数可以将 request 加入到 elevator request queue 或者 device request queue 中。该函数的实现如下：

```
void __elv_add_request(struct request_queue *q, struct request *rq, int where)
{
trace_block_rq_insert(q, rq);
rq->q = q;
if (rq->cmd_flags & REQ_SOFTBARRIER) {
/* barriers are scheduling boundary, update end_sector */
if (rq->cmd_type == REQ_TYPE_FS ||
(rq->cmd_flags & REQ_DISCARD)) {
q->end_sector = rq_end_sector(rq);
q->boundary_rq = rq;
}
} else if (!(rq->cmd_flags & REQ_ELVPRIV) &&
(where == ELEVATOR_INSERT_SORT ||
where == ELEVATOR_INSERT_SORT_MERGE))
where = ELEVATOR_INSERT_BACK;
switch (where) {
case ELEVATOR_INSERT_REQUEUE:
case ELEVATOR_INSERT_FRONT:
/* 将request加入到device request queue的队列前 */
rq->cmd_flags |= REQ_SOFTBARRIER;
list_add(&rq->queuelist, &q->queue_head);
break;
case ELEVATOR_INSERT_BACK:
/* 将request 加入到device request queue的队列尾 */
rq->cmd_flags |= REQ_SOFTBARRIER;
elv_drain_elevator(q);
list_add_tail(&rq->queuelist, &q->queue_head);
/*
* We kick the queue here for the following reasons.
* - The elevator might have returned NULL previously
*   to delay requests and returned them now.  As the
*   queue wasn't empty before this request, ll_rw_blk
*   won't run the queue on return, resulting in hang.
* - Usually, back inserted requests won't be merged
*   with anything.  There's no point in delaying queue
*   processing.
*/
__blk_run_queue(q);
break;
case ELEVATOR_INSERT_SORT_MERGE:
/* 尝试对request进行合并操作，如果无法合并将request加入到elevator request queue中 */
/*
* If we succeed in merging this request with one in the
* queue already, we are done - rq has now been freed,
* so no need to do anything further.
*/
if (elv_attempt_insert_merge(q, rq))
break;
case ELEVATOR_INSERT_SORT:
/* 将request加入到elevator request queue中 */
BUG_ON(rq->cmd_type != REQ_TYPE_FS &&
!(rq->cmd_flags & REQ_DISCARD));
rq->cmd_flags |= REQ_SORTED;
q->nr_sorted++;
if (rq_mergeable(rq)) {
elv_rqhash_add(q, rq);
if (!q->last_merge)
q->last_merge = rq;
}
/*
* Some ioscheds (cfq) run q->request_fn directly, so
* rq cannot be accessed after calling
* elevator_add_req_fn.
*/
q->elevator->ops->elevator_add_req_fn(q, rq);
break;
case ELEVATOR_INSERT_FLUSH:
rq->cmd_flags |= REQ_SOFTBARRIER;
blk_insert_flush(rq);
break;
default:
printk(KERN_ERR "%s: bad insertion point %d\n",
__func__, where);
BUG();
}
}
```

Elv_dispatch_sort

当 elevator request queue 中的 request 需要发送到 device request queue 中时，可以调用 elv_dispatch_sort 函数，通过该函数可以对 request 进行排序，插入到合适的位置。Elv_dispatch_sort 函数的实现如下：

```
void elv_dispatch_sort(struct request_queue *q, struct request *rq)
{
sector_t boundary;
struct list_head *entry;
int stop_flags;
if (q->last_merge == rq)
q->last_merge = NULL;
elv_rqhash_del(q, rq);
q->nr_sorted--;
boundary = q->end_sector;
stop_flags = REQ_SOFTBARRIER | REQ_STARTED;
list_for_each_prev(entry, &q->queue_head) {
struct request *pos = list_entry_rq(entry);
if ((rq->cmd_flags & REQ_DISCARD) !=
(pos->cmd_flags & REQ_DISCARD))
break;
if (rq_data_dir(rq) != rq_data_dir(pos))
break;
if (pos->cmd_flags & stop_flags)
break;
if (blk_rq_pos(rq) >= boundary) {
if (blk_rq_pos(pos) < boundary)
continue;
} else {
if (blk_rq_pos(pos) >= boundary)
break;
}
if (blk_rq_pos(rq) >= blk_rq_pos(pos))
break;
}
list_add(&rq->queuelist, entry);
}
```

Elevator 子系统小结

Elevator 子系统是实现 IO 调度处理的框架，功能不同的 scheduler 可以做为一种 elevator type 加入到这个框架中来。所以，如果需要设计实现一个自定义的 scheduler，那么首先必须需要了解 elevator 子系统。

Linux 中常见 IO 调度器

Noop 调度器算法

Noop 是 Linux 中最简单的调度器，这个调度器基本上没做什么特殊的事情，就是把邻近 bio

进行了合并处理。从 IO 的 QoS 角度来看，这个 Noop 调度器就是太简单了，但是从不同存储介质的特性来看，这个 Noop 还是有一定用武之地的。例如，对于磁盘介质而言，为了避免磁头抖动，可以通过调度器对写请求进行合并。对于 SSD 存储介质而言，这个问题不存在了，或者说不是那么简单的存在了。如果 SSD 内部能够很好的处理了写放大等问题，那么调度器这一块就不需要做什么特殊处理了，此时 Noop 就可以发挥作用了。

通过阅读 Noop 调度器的代码，我们可以了解到一个调度器是如何实现的，对外的接口是什么？所以，了解一个调度器的框架，从 Noop 开始是非常好的一个选择。

Noop 调度器的实现非常简单，其主要完成了一个 elevator request queue，这个 request queue 没有进行任何的分类处理，只是对输入的 request 进行简单的队列操作。但是，需要注意的是，虽然 Noop 没有做什么事情，但是 elevator 还是对 bio 进行了后向合并，从而最大限度的保证相邻的 bio 得到合并处理。Noop 调度器实现了 elevator 的基本接口函数，并将这些函数注册到 linux 系统的 elevator 子系统中。

需要注册到 elevator 子系统中的基本接口函数声明如下：

```
1  static struct elevator_type elevator_noop = {
2  .ops = {
3  /* 合并两个request */
4  .elevator_merge_req_fn      = noop_merged_requests,
5  /* 调度一个合适的request进行发送处理 */
6  .elevator_dispatch_fn       = noop_dispatch,
7  /* 将request放入调度器的queue中 */
8  .elevator_add_req_fn        = noop_add_request,
9  /* 获取前一个request */
10 .elevator_former_req_fn     = noop_former_request,
11 /* 获取后一个request */
12 .elevator_latter_req_fn     = noop_latter_request,
13 .elevator_init_fn       = noop_init_queue,
14 .elevator_exit_fn       = noop_exit_queue,
15 },
16 .elevator_name = "noop",
17 .elevator_owner = THIS_MODULE,
18 };
```

由于 Noop 调度器没有对 request 进行任何的分类处理、调度，因此上述这些函数的实现都很简单。例如，当调度器需要发送 request 时，会调用 noop_dispatch。该函数会直接从调度器所管理的 request queue 中获取一个 request，然后调用 elv_dispatch_sort 函数将请求加入到设备所在的 request queue 中。Noop dispatch 函数实现如下：

```
static int noop_dispatch(struct request_queue *q, int force)
{
struct noop_data *nd = q->elevator->elevator_data;
if (!list_empty(&nd->queue)) {
struct request *rq;
/* 从调度器的队列头中获取一个request */
rq = list_entry(nd->queue.next, struct request, queuelist);
list_del_init(&rq->queuelist);
/* 将获取的request放入到设备所属的request queue中 */
elv_dispatch_sort(q, rq);
return 1;
}
return 0;
}
```

当需要往 noop 调度器中放入 request 时，可以调用 noop_add_request，该函数的实现及其简单，就是将 request 挂入调度器所维护的 request queue 中。Noop_add_request 函数实现如下：

```
static void noop_add_request(struct request_queue *q, struct request *rq)
{
struct noop_data *nd = q->elevator->elevator_data;
/* 将request挂入noop调度器的request queue */
list_add_tail(&rq->queuelist, &nd->queue);
}
```

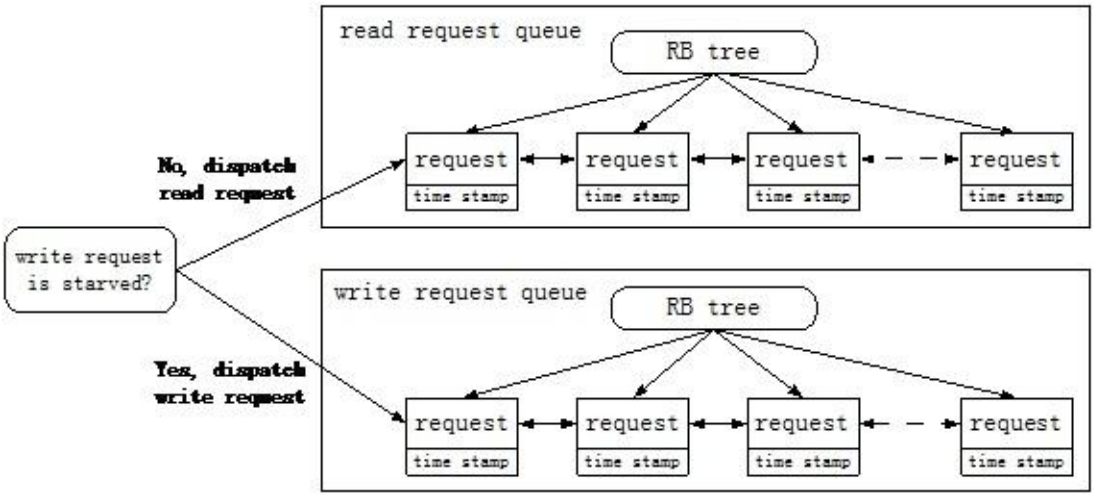
由此可见，noop 调度器的实现是很简单的，仅仅实现了一个调度器的框架，用一条链表把所有输入的 request 管理起来。通过 noop 调度器的例子，我们可以了解到实现一个调度器所需要的基本结构：

```c
/* 包含基本的头文件 */
#include <linux/blkdev.h>
#include <linux/elevator.h>
#include <linux/bio.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/init.h>
/* 定义调度器所需要的数据结构，一条管理request的队列是必须的 */
struct noop_data {
struct list_head queue;
};
/* 实现调度器的接口函数 */
static struct elevator_type elevator_noop = {
.ops = {
/* 调度器的功能函数 */
.elevator_merge_req_fn      = noop_merged_requests,
……
/* 初始化/注销调度器，通常在下面这些函数初始化调度器内部的一些数据结构，例如noop_data */
.elevator_init_fn       = noop_init_queue,
.elevator_exit_fn       = noop_exit_queue,
},
.elevator_name = "noop",
.elevator_owner = THIS_MODULE,
};
/* 注册调度器 */
static int __init noop_init(void)
{
elv_register(&elevator_noop);
return 0;
}
/* 销毁调度器 */
static void __exit noop_exit(void)
{
elv_unregister(&elevator_noop);
}
/* 模块加载时调用noop_init */
module_init(noop_init);
/* 模块退出时调用noop_exit */
module_exit(noop_exit);
```

Deadline 调度器算法

Deadline 这种调度器对读写 request 进行了分类管理，并且在调度处理的过程中读请求具有较高优先级。这主要是因为读请求往往是同步操作，对延迟时间比较敏感，而写操作往往是异步操作，可以尽可能的将相邻访问地址的请求进行合并，但是，合并的效率越高，延迟时间会越长。因此，为了区别对待读写请求类型，deadline 采用两条链表对读写请求进行分类管理。但是，引入分类管理之后，在读优先的情况下，写请求如果长时间得到不到调度，会出现饿死的情况，因此，deadline 算法考虑了写饿死的情况，从而保证在读优先调度的情况下，写请求不会被饿死。

Deadline 这种调度算法的基本思想可以采用下图进行描述：



读写请求被分成了两个队列，并且采用两种方式将这些 request 管理起来。一种是采用红黑树（RB tree）的方式将所有 request 组织起来，通过 request 的访问地址作为索引；另一种方式是采用队列的方式将 request 管理起来，所有的 request 采用先来后到的方式进行排序，即 FIFO 队列。每个 request 会被分配一个 time stamp，这样就可以知道这个 request 是否已经长时间没有得到调度，需要优先处理。在请求调度的过程中，读队列是优先得到处理的，除非写队列长时间没有得到调度，存在饿死的状况。

在请求处理的过程中，deadline 算法会优先处理那些访问地址临近的请求，这样可以最大程度的减少磁盘抖动的可能性。只有在有些 request 即将被饿死的时候，或者没有办法进行磁盘顺序化操作的时候，deadline 才会放弃地址优先策略，转而处理那些即将被饿死的 request。

总体来讲，deadline 算法对 request 进行了优先权控制调度，主要表现在如下几个方面：

1） 读写请求分离，读请求具有高优先调度权，除非写请求即将被饿死的时候，才会去调度处理写请求。这种处理可以保证读请求的延迟时间最小化。

2） 对请求的顺序批量处理。对那些地址临近的顺序化请求，deadline 给予了高优先级处理权。例如一个写请求得到调度后，其临近的 request 会在紧接着的调度过程中被处理掉。这种顺序批量处理的方法可以最大程度的减少磁盘抖动。

3） 保证每个请求的延迟时间。每个请求都赋予了一个最大延迟时间，如果达到延迟时间的上限，那么这个请求就会被提前处理掉，此时，会破坏磁盘访问的顺序化特征，回影响性能，但是，保证了每个请求的最大延迟时间。

与 deadline 相关的请求调度发送函数是 deadline_dispatch_requests，该函数的实现、分析如下：

```
static int deadline_dispatch_requests(struct request_queue *q, int force)
{
struct deadline_data *dd = q->elevator->elevator_data;
const int reads = !list_empty(&dd->fifo_list[READ]);
const int writes = !list_empty(&dd->fifo_list[WRITE]);
struct request *rq;
int data_dir;
/*
* batches are currently reads XOR writes
请求批量处理入口
*/
if (dd->next_rq[WRITE])
rq = dd->next_rq[WRITE];
else
rq = dd->next_rq[READ];
/* 如果批量请求处理存在，并且还没有达到批量请求处理的上限值，那么继续请求的批量处理 */
if (rq && dd->batching < dd->fifo_batch)
/* we have a next request are still entitled to batch */
goto dispatch_request;
/*
* at this point we are not running a batch. select the appropriate
* data direction (read / write)
*/
/* 优先处理读请求队列 */
if (reads) {
BUG_ON(RB_EMPTY_ROOT(&dd->sort_list[READ]));
/* 如果写请求队列存在饿死的现象，那么优先处理写请求队列 */
if (writes && (dd->starved++ >= dd->writes_starved))
goto dispatch_writes;
data_dir = READ;
goto dispatch_find_request;
}
/*
* there are either no reads or writes have been starved
*/
/* 没有读请求需要处理，或者写请求队列存在饿死现象 */
if (writes) {
dispatch_writes:
BUG_ON(RB_EMPTY_ROOT(&dd->sort_list[WRITE]));
dd->starved = 0;
data_dir = WRITE;
goto dispatch_find_request;
}
return 0;
dispatch_find_request:
/*
* we are not running a batch, find best request for selected data_dir
*/
if (deadline_check_fifo(dd, data_dir) || !dd->next_rq[data_dir]) {
/* 如果请求队列中存在即将饿死的request，或者不存在需要批量处理的请求，那么从FIFO队列头获取一个request */
/*
* A deadline has expired, the last request was in the other
* direction, or we have run out of higher-sectored requests.
* Start again from the request with the earliest expiry time.
*/
rq = rq_entry_fifo(dd->fifo_list[data_dir].next);
} else {
/* 继续批量处理，获取需要批量处理的下一个request */
/*
* The last req was the same dir and we have a next request in
* sort order. No expired requests so continue on from here.
*/
rq = dd->next_rq[data_dir];
}
dd->batching = 0;
dispatch_request:
/* 将request从调度器中移出，发送至设备 */
/*
* rq is the selected appropriate request.
*/
dd->batching++;
deadline_move_request(dd, rq);
return 1;
}
Deadline调度器需要处理的核心数据结构是deadline_data，该结构描述如下：
struct deadline_data {
/*
* run time data
*/
/*
* requests (deadline_rq s) are present on both sort_list and fifo_list
*/
/* 采用红黑树管理所有的request，请求地址作为索引值 */
struct rb_root sort_list[2];
/* 采用FIFO队列管理所有的request，所有请求按照时间先后次序排列 */
struct list_head fifo_list[2];
/*
* next in sort order. read, write or both are NULL
*/
/* 批量处理请求过程中，需要处理的下一个request */
struct request *next_rq[2];
/* 计数器：统计当前已经批量处理完成的request */
unsigned int batching;		/* number of sequential requests made */
sector_t last_sector;		/* head position */
/* 计数器：统计写队列是否即将饿死 */
unsigned int starved;		/* times reads have starved writes */
/*
* settings that change how the i/o scheduler behaves
*/
/* 配置信息：读写请求的超时时间值 */
int fifo_expire[2];
/* 配置信息：批量处理的request数量 */
int fifo_batch;
/* 配置信息：写饥饿值 */
int writes_starved;
int front_merges;
};
```

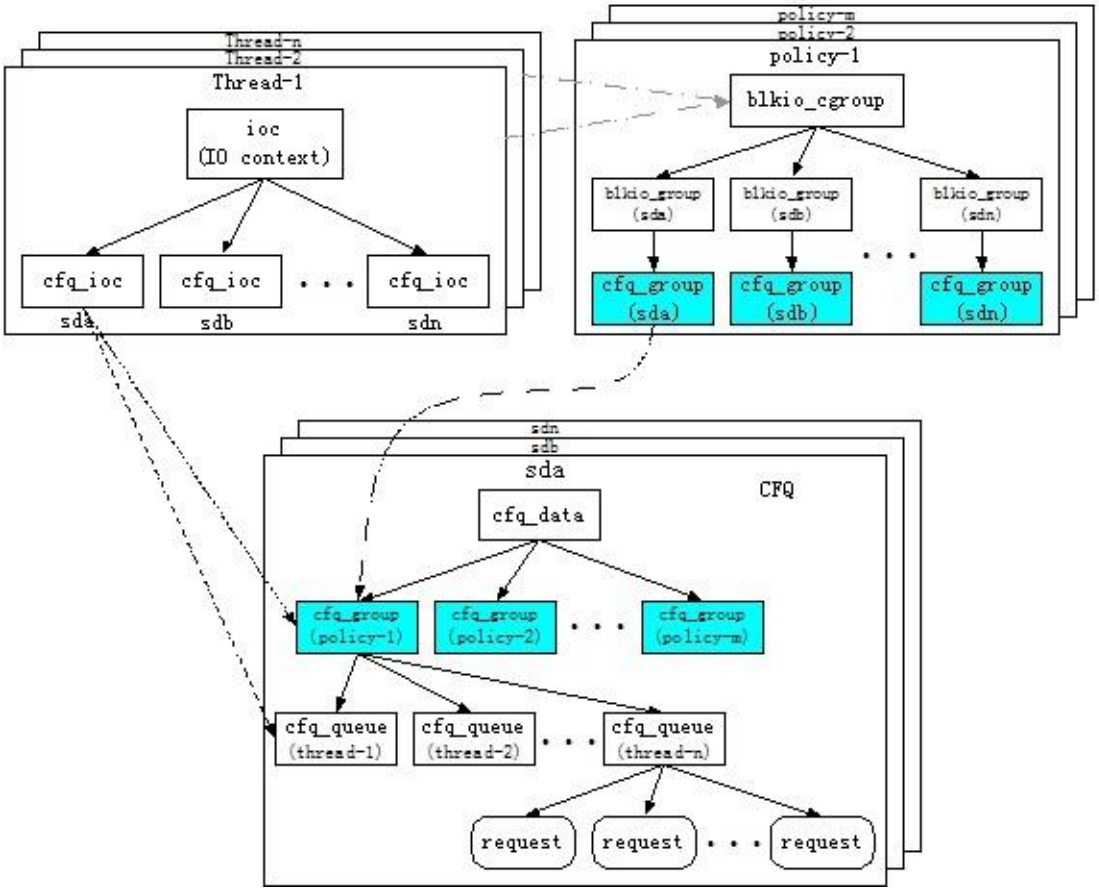
总体而言,Noop 和 Deadline 算法实现是比较简单的.

CFQ 调度器算法

在相对比较老的 Linux 中，CFQ 机制的实现还比较简单，仅仅是针对不同的 thread 进行磁盘带宽的公平调度。但是，自从新 Kernel 引入 Cgroup 机制之后，CFQ 的机制就显得比较复杂了，因为，此时的 CFQ 不仅可以对 IO 带宽进行公平调度，更重要的是对磁盘 IO 进行 QoS 控制。

IO 的 QoS 控制是非常必要的，在同一系统中，不同进程/线程所享有的 IO 带宽可以是不同的，为了达到此目的，Linux 在 Cgroup 框架下引入了 blkio_cgroup 对不同线程的带宽资源进行调度控制。由于 cgroup 的引入，CFQ 算法中引入了 cfq_group 的概念，所以使得整个 IO 资源调度的算法看起来复杂多了。

为了更好的理解 CFQ 算法的整个框架，需要理清几个关键数据结构之间的关系，下图是 cfq_group，cfq_data，cfq_queue，ioc，cfq_ioc，blkio_cgroup 以及 blkio_group 数据结构之间的关系图。



通过上图，我们可以知道 ioc 是每个线程所拥有的 IO 上下文（context），由于一个线程可以对多个磁盘设备进行操作，因此，每个线程会对每个正在操作的磁盘对象创建一个 cfq_ioc。在 cfq_ioc 中会保存与这个磁盘设备相关的策略对象 cfq_group 以及 IO 调度对象 cfq_queue。因此，当一个线程往一个磁盘设备进行写操作时，可以通过 IO 上下文获取这个 request 所应该附属的策略对象 cfq_group 和调度对象 cfq_queue。

IO 上下文是线程相关的，多个线程可以定义相同的策略行为，因此，在 Linux Cgroup 机制中为每个 IO 策略集定义了一个 blkio_cgroup 对象，该对象负责管理 IO 的控制策略。例如，控

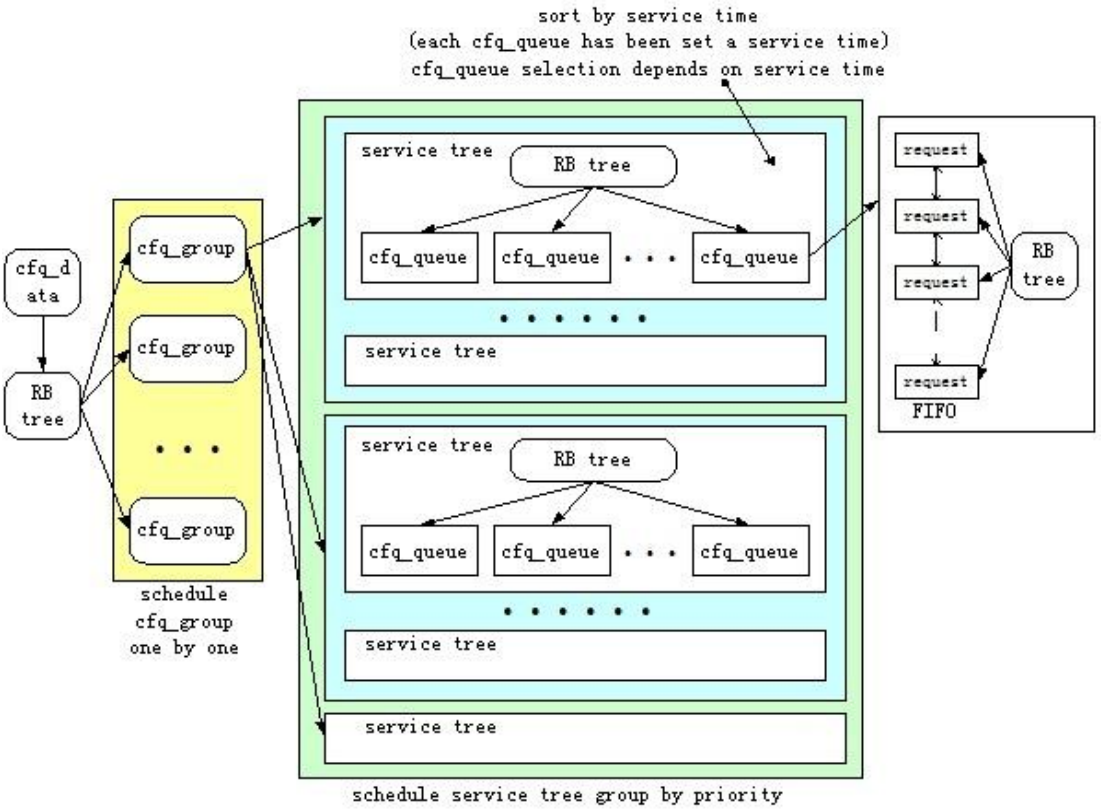
制 IO 的策略主要有带宽、延迟。由于一个线程需要对多个磁盘设备进行 IO 操作，因此，每个 blkio_cgroup 对象管理了针对每个磁盘设备的策略集，该对象为 blkio_group。假设系统中有两个线程 ThreadA 和 ThreadB，他们拥有相同的策略并且都在对同一个设备 Sda 进行操作，那么在 blkio_cgroup 中只存在一个和 sda 相关的 blkio_group。如果 ThreadA 和 ThreadB 分别对 sda 和 sdb 进行操作，那么在 blkio_cgroup 中将会存在两个 blkio_group 对象。在 CFQ 算法中，对 blkio_group 对象进行了封装，形成了 cfq_group 对象。因此，可以认为 cfq_group 和 blkio_group 对象是一一对应的。也就是说 cfq_group 对象是针对一个磁盘设备的一种策略，当然同一个磁盘设备会存在多种 IO 策略，他们分别归属不同的线程。因此，一个磁盘的 cfq 对象中会管理多个 cfq_group 对象。

CFQ 算法是针对一个磁盘设备的，即每个磁盘设备都会管理一个 cfq 对象。在磁盘设备的 request queue 对象中会保存这个 cfq 对象，这点和其他的调度算法是相同的。Cfq_data 就是那个 cfq 对象。前面提到，由于一个磁盘设备会被多个线程进行访问，因此可以存在多种 IO 策略，所以，一个 cfq 对象需要管理多个 cfq_group，每个 cfq_group 管理了一种 IO 策略。在 cfq 算法中，每个按时间片调度的基本单元是线程，由于多个线程可以共享一种策略，即共享 cfq_group，因此，在 cfq_group 对象中为每个线程独立管理了一个 request 队列，这个队列对象是 cfq_queue。Cfq_queue 是与线程相关的调度对象，cfq 算法在调度的过程中会给被调度的 cfq_queue 分配时间片，然后在这个时间片内处理 cfq_queue 中的 request。

调度 cfq_queue 中的 request 方法和 deadline 算法是基本一致的。因此，在 cfq_queue 中会维护一棵红黑树以及一条 FIFO 队列，FIFO 队列中的 request 按照时间先后次序排列，红黑树上的 request 采用 LBA 访问地址进行索引。并且在处理的过程中，也会考虑临近 IO 优先处理的策略，使得磁盘临近 IO 批量处理，提高了 IO 性能。

从上面的数据结构关系图，我们可以发现，和老版本的 cfq 算法相比，新版本的 CFQ 引入了 blkio_cgroup 以及 blkio_group 对象，在 cfq 中的体现为 cfq_group。引入 cfq_group 之后，IO 的调度处理将变的更加灵活。例如，如果 ThreadA 被定义成 PolicyA，其访问 sda 将有 80%的带宽，ThreadB 被定义成 PolicyB，其访问 sda 将有 20%的带宽。对于上述情况，将会创建两个 blkio_cgroup，并且拥有两个 cfq_group 对象，这两个 cfq_group 对象都会被加入到 sda 设备的 cfq_data 对象中进行管理。Cfq_groupA 对应着 policyA，管理 threadA 的 IO 请求，cfq_groupB 对应着 policyB，管理 threadB 的 IO 请求。由于 cfq_groupA 策略占有 80%带宽，那么在调度过程中，cfq 算法会把 80%的时间片分给 cfq_groupA，而 cfq_groupB 只能分到 20%的时间片，因此达到了对 ThreadA 和 B 的 IO QoS 控制目的。

理清楚 cfq 算法中关键数据结构之间的关系之后，最重要的问题就是如何实现 request 的调度处理。下图展示了 cfq 中各层对象的调度处理方法。



在调度一个 request 时，首先需要选择一个一个合适的 cfq_group。Cfq 调度器会为每个 cfq_group 分配一个时间片，当这个时间片耗尽之后，会选择下一个 cfq_group。每个 cfq_group 都会分配一个 vdisktime，并且通过该值采用红黑树对 cfq_group 进行排序。在调度的过程中，每次都会选择一个 vdisktime 最小的 cfq_group 进行处理。

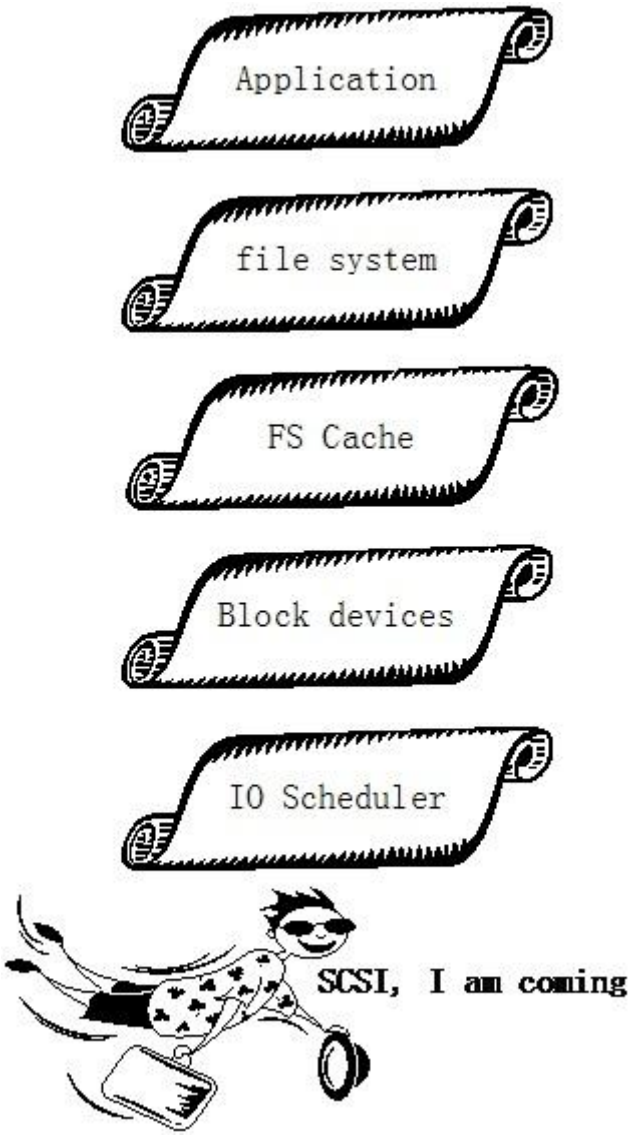
一个 cfq_group 管理了 7 棵 service tree，每棵 service tree 管理了需要调度处理的对象 cfq_queue。因此，一旦 cfq_group 被选定之后，需要选择一棵 service tree 进行处理。这 7 棵 service tree 被分成了三大类，分别为 RT、BE 和 IDLE。这三大类 service tree 的调度是按照优先级展开的，即 RT 的优先级高于 BE，BE 的高于 IDLE。通过优先级可以很容易的选定一类 Service tree。当一类 service tree 被选定之后，采用 service time 的方式选定一个合适的 cfq_queue。每个 Service tree 是一棵红黑树，这些红黑树是按照 service time 进行检索的，每个 cfq_queue 都会维护自己的 service time。分析到这里，我们知道，cfq 算法通过每个 cfq_group 的 vdisktime 值来选定一个 cfq_group 进行服务，在处理 cfq_group 的过程通过优先级选择一个最需要服务的 service tree。通过该 Service tree 得到最需要服务的 cfq_queue。该过程在 cfq_select_queue 函数中实现。

一个 cfq_queue 被选定之后，后面的过程和 deadline 算法有点类似。在选择 request 的时候需要考虑每个 request 的延迟等待时间，选择那种等待时间最长的 request 进行处理。但是，考虑到磁盘抖动的问题，cfq 在处理的时候也会进行顺序批量处理，即将那些在磁盘上连续的 request 批量处理掉。

在 cfq 调度的过程中，选择 cfq_queue 的重要函数实现如下：

```
static struct cfq_queue *cfq_select_queue(struct cfq_data *cfqd)
{
struct cfq_queue *cfqq, *new_cfqq = NULL;
/* 如果cfq调度器当前没有服务的对象，那么直接跳转去选择一个新的cfq_queue */
cfqq = cfqd->active_queue;
if (!cfqq)
goto new_queue;
/* 如果在cfq中没有request，直接返回 */
if (!cfqd->rq_queued)
return NULL;
/*
* We were waiting for group to get backlogged. Expire the queue
*/
if (cfq_cfqq_wait_busy(cfqq) && !RB_EMPTY_ROOT(&cfqq->sort_list))
goto expire;
/*
* The active queue has run out of time, expire it and select new.
当前的cfq_queue已经服务到限，重新选择一个新的cfq_queue
*/
if (cfq_slice_used(cfqq) && !cfq_cfqq_must_dispatch(cfqq)) {
/*
* If slice had not expired at the completion of last request
* we might not have turned on wait_busy flag. Don't expire
* the queue yet. Allow the group to get backlogged.
*
* The very fact that we have used the slice, that means we
* have been idling all along on this queue and it should be
* ok to wait for this request to complete.
*/
if (cfqq->cfqg->nr_cfqq == 1 && RB_EMPTY_ROOT(&cfqq->sort_list)
&& cfqq->dispatched && cfq_should_idle(cfqd, cfqq)) {
cfqq = NULL;
goto keep_queue;
} else
goto check_group_idle;
}
/*
* The active queue has requests and isn't expired, allow it to
* dispatch.
Cfq_queue还存在很多需要处理的request，继续处理这个cfq_queue
*/
if (!RB_EMPTY_ROOT(&cfqq->sort_list))
goto keep_queue;
/*
* If another queue has a request waiting within our mean seek
* distance, let it run.  The expire code will check for close
* cooperators and put the close queue at the front of the service
* tree.  If possible, merge the expiring queue with the new cfqq.
当前处理的cfq_queue已经超时，需要选择一个在磁盘上与当前处理的request临近的cfq_queue
*/
new_cfqq = cfq_close_cooperator(cfqd, cfqq);
if (new_cfqq) {
if (!cfqq->new_cfqq)
cfq_setup_merge(cfqq, new_cfqq);
goto expire;
}
/*
* No requests pending. If the active queue still has requests in
* flight or is idling for a new request, allow either of these
* conditions to happen (or time out) before selecting a new queue.
*/
if (timer_pending(&cfqd->idle_slice_timer)) {
cfqq = NULL;
goto keep_queue;
}
/*
* This is a deep seek queue, but the device is much faster than
* the queue can deliver, don't idle
**/
if (CFQQ_SEEKY(cfqq) && cfq_cfqq_idle_window(cfqq) &&
(cfq_cfqq_slice_new(cfqq) ||
(cfqq->slice_end - jiffies > jiffies - cfqq->slice_start))) {
cfq_clear_cfqq_deep(cfqq);
cfq_clear_cfqq_idle_window(cfqq);
}
if (cfqq->dispatched && cfq_should_idle(cfqd, cfqq)) {
cfqq = NULL;
goto keep_queue;
}
/*
* If group idle is enabled and there are requests dispatched from
* this group, wait for requests to complete.
*/
check_group_idle:
if (cfqd->cfq_group_idle && cfqq->cfqg->nr_cfqq == 1 &&
cfqq->cfqg->dispatched &&
!cfq_io_thinktime_big(cfqd, &cfqq->cfqg->ttime, true)) {
cfqq = NULL;
goto keep_queue;
}
expire:
/* 将当前cfq_queue设置成超时 */
cfq_slice_expired(cfqd, 0);
new_queue:
/*
* Current queue expired. Check if we have to switch to a new
* service tree
如果没有找到一个临近的cfq_queue，那么选择一个新的service tree进行处理
*/
if (!new_cfqq)
cfq_choose_cfqg(cfqd);
/* 初始化一个调度处理的cfq_queue */
cfqq = cfq_set_active_queue(cfqd, new_cfqq);
keep_queue:
return cfqq;
}
```

辗转反侧，一个 IO 终于从应用层达到了 IO Scheduler 层，并且经过 scheduler 调度处理之后准备前往 SCSI 层，向目的地 Disk 方向前进。一个 IO 历尽千辛万苦从应用层来到 IO 调度器，正准备调度走，回眸往事，不禁潸然泪下，路漫漫，一路走来真是不易！从应用层走到文件系统；在文件系统的 Cache 中停留许久之后，又被 Writeback 线程写入到块设备层；经过块设备层的调度处理之后，终于来到 IO 调度层；经过 IO 层严格调度之后，最终得到机会往 SCSI 协议层奔去。不就是一个 IO 操作吗？居然要经过这么多层的调度处理，想到这里，IO 难免会觉得委屈，但不管怎样，从 IO 调度器出来之后，IO 还是可以春风得意地奔向 SCSI 协议层的怀抱，在那里准备打包，并梦想着乘坐高速列车驶向心仪已久的目的地——Disk Drive

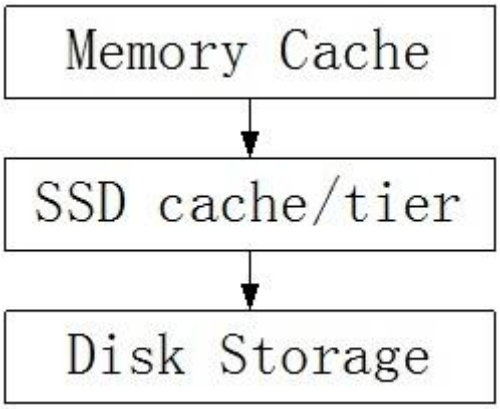


春风得意的 IO 那里知道，他其实是一个幸运儿，他才走了很短的路程，未来的路还很漫长。更为幸运的是，他前面走过的路非常的平顺，其实没有太多的波折，是每个 IO 都会经历的过程。通常来讲，在很多高端的存储系统中，一个 IO 在块设备层会经历更多的软件层，这些软件层可以被称之为块设备层的 IO 堆栈。

块设备的 IO 堆栈包括 SSD 缓存系统、卷管理器、Snapshot 快照系统、CDP 持续数据保护系统、RAID 磁盘阵列系统。这些软件层可以堆叠在一起，也可以单独存在。在一个后端存储系统中，SSD 缓存系统、Snapshot、卷管理、RAID 磁盘阵列往往会被堆叠在一起，从而构成一套高性能、高可用、易管理的存储系统。一个具有完整 IO 堆栈的系统层次结构如下图所示：

近几年 SSD 的发展的确飞速迅猛，其最大表现在于大容量的 Nand Flash 存储芯片的研发和 SSD Firmware 的推进发展。Nand Flash 技术在电子技术领域其实不是一个新东西，很早就采用了。但是，那时候的 Nand Flash 性能和容量远远不如现在，那时 Nand Flash 仅仅作为电子系统设计中的一个持久化数据保存地，我在 10 多年前就西门子合作采用 Nand Flash 来设计存储高速公路收费信息的系统，将 Nand Flash 作为一个裸磁盘使用。另外，在航天航空领域也会采用 Nand Flash 来替代机械硬盘，但是性能很一般，诸如 DOM 卡、USB 存储之类的东西。随着 Nand Flash 制造工艺的不断突破，存储容量不断增大。基于 Nand Flash 的 SSD 终于诞生了，并且曾经一度想替代存储领域的主角——机械硬盘。SSD 的最大优点在于随机读写性能强，不存在机械寻道时间，因此具有很高的 IOPS 和带宽。当然，节能、可靠性也是 SSD 一大优点。SSD 的最大缺点在于使用寿命，每个 Flash 块都存在擦写次数的限制，会影响数据存储的安全性。另外，Nand Flash 在写操作时会存在写放大等问题，这个问题会影响到 SSD 的小写性能以及使用寿命。因此，如何解决 SSD 本身存在的这些问题是 SSD 研发的重点，各个 SSD 公司都会设计自己的 Firmware 来应对这些问题。正是因为这个原因，SSD Firmware 是一个 SSD 公司的核心技术机密。从外部使用者的角度来看，SSD 比机械磁盘具有更好的随机/顺序读写性能，但是容量远远小于磁盘，所以在短时间内无法采用 SSD 整体替换磁盘系统，除非是一些规模不是很大的存储应用系统。为了更好的结合 SSD 和磁盘，可以将 SSD 作为一种缓存系统应用于磁盘存储系统之上。大家知道，在现有的存储架构下，内存作为磁盘存储系统的 Cache，那么引入 SSD 之后，可以将 SSD 作为磁盘存储的二级 Cache。正是这个

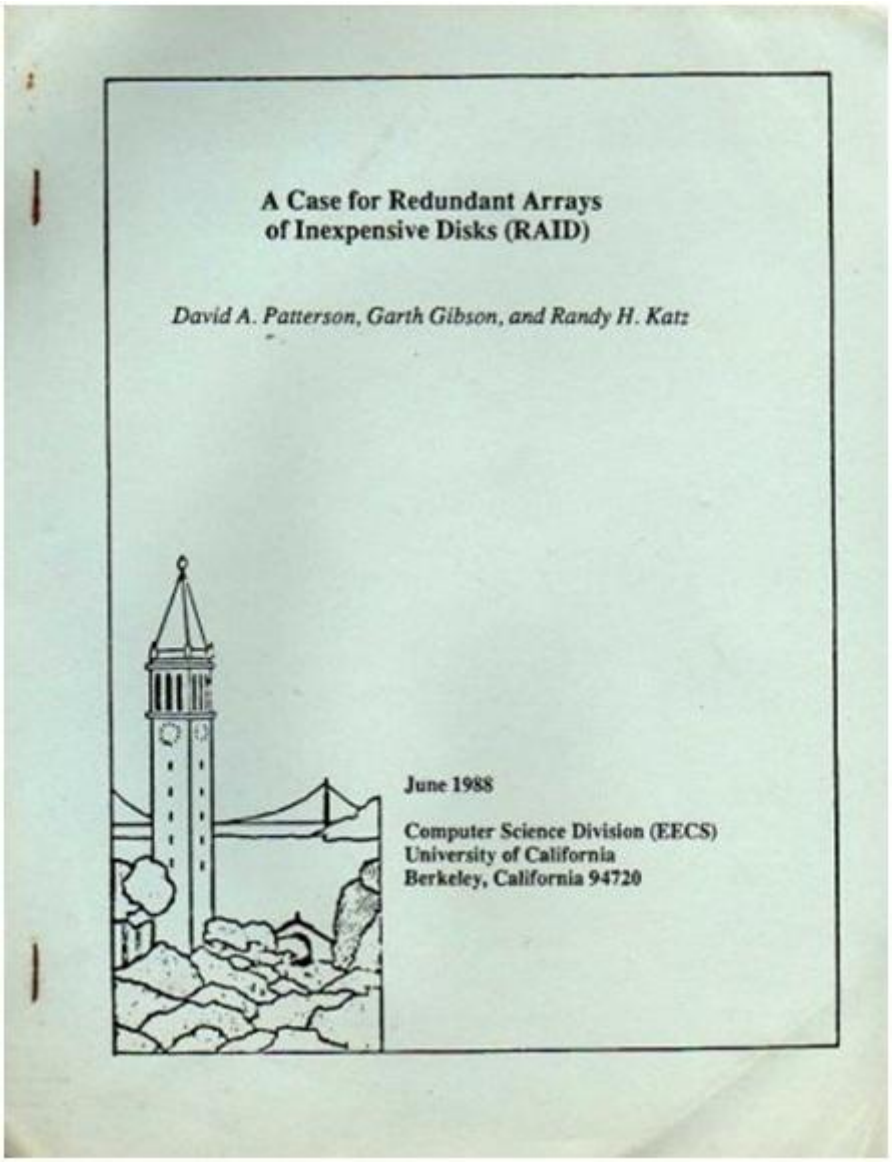
原因，在目前的很多存储系统中都会加入 SSD 缓存系统作为读 Cache，或者作为数据自动分层的高效存储资源。SSD 缓存层加入之后，对于很多应用可以极大提高系统的 IOPS 和 IO 带宽。



大家知道 snapshot 是一种快照系统，为了达到数据在时间轴上可回滚的目的，可以采用 snapshot 的技术对数据在时间轴上做切片。如果用户想要回滚到历史上某个时间点的数据集，那么可以采用这个时间点的快照。快照可以在文件系统层实现，也可以在块层达到相同的目的。由于文件系统本身采用 inode tree 之类的树维护了数据块的映射信息，因此，实现起来相对容易。在块层如果需要实现快照，那么需要数据块映射机制作为支撑。块层快照往往在卷管理器之上实现快照语义，卷管理器可以作为资源块分配机制为快照进行资源动态分配。快照算法通常有写时拷贝 COW 和写时映射 ROW 两种，对于不同的应用需求，这两种算法有各自的优缺点。从结构上分，快照有草型快照和链式快照之分，不同的结构有不同的效率和性能。

卷管理器是一个老生常谈的话题，Linux 中的逻辑卷管理 LVM 名声在外，很多系统都会采用 LVM 作为卷管理器。LVM 可以整合不同厂商的存储资源，然后根据用户的需求形成逻辑卷供应用层使用，在 LVM 中物理卷池和逻辑卷都可以动态扩容，从而简化了存储资源的管理。在现代存储系统中，为了提高 IO 性能、存储资源的使用率，在卷管理层引入了 Thin provisioning 的功能。Thin Provisioning 的最大特点在于可以对存储资源进行按需动态分配，即对存储进行了虚拟化管理。从用户的角度来看，一个逻辑卷有 100TB 的空间，但是，实际的资源可能只有 10TB，这就是 Thin Provisioning 的核心思想。存储虚拟化的另外一大好处在于可以优化 IO 性能。在多用户的一个系统中，从单个用户的角度来看，可能都是顺序写操作，但是，到了存储后端，由于多用户并发的效果，导致这些顺序 IO 会混合在一起，从而对后端存储系统而言形成了随机 IO。多用户并发操作使得存储系统的性能极为降低，从而导致诸如 VMware 虚拟化环境下出现的多用户并发操作导致每个用户的性能体验都很差。为了解决这个问题，可以在卷管理层引入虚拟化机制，将这种随机 IO 顺序化，从而提高整体存储性能。

磁盘阵列 RAID 技术是大家非常熟悉的技术，其主要目的是用来保护用户数据和提高读写性能。由于 RAID 技术可以在多个磁盘之间并发 IO，因此使得应用 IO 性能大为提高。另外，在 RAID 技术中可以采用 XOR、RS 编码以及擦除码等编解码技术进行数据信息冗余，因此，可以提高数据可靠性。RAID 技术从诞生之初到现在已经发展的极为成熟，几乎所有的存储系统都会采用 RAID 作为系统的数据保护层和性能优化层。下图是 1988 年 Patterson, Gibson 和 Katz 联合发表的 RAID 学术论文：



通常来讲，一个系统可以采用硬件 RAID 卡或者纯软件的技术手段来实现 RAID 功能，在中低端系统中通常都会采用硬件 RAID 方案来实现 RAID 功能，但是在高端系统中，都会考虑采用软件的方式来实现 RAID 功能。软件方式实现 RAID 功能的优点在于可以根据应用特征实现复杂的数据保护方式。例如，在备份系统中对数据保护的要求比较高，实现的 RAID 需要比其它 RAID 具有更强的数据保护能力，因此，软件 RAID 的方案是一个比较好的选择。伴随着磁盘容量的不断增大，RAID 本身存在的问题也越来越突出：一个问题是 RAID 的数据重构时间变的越来越长；另一个问题是 RAID 数据重构 IO 对应用 IO 性能造成严重影响。为了解决上述 RAID 问题，传统 RAID 正面临着架构转型。

由此可见在存储系统中，块设备层具有很多的软件层，这些软件层用于解决性能、灵活性、数据可靠性等问题。前面讲到春风得意的 IO 准备离开 IO Scheduler 层，通过磁盘驱动程序前往 SCSI 协议层。回过头来，我们还是有必要细细探讨 SSD 缓存、快照系统、卷管理器以及 RAID 系统，从中体验 IO 旅程的不易。后面我们将对块设备层的这些主要功能层的设计实现进行详细剖析。

从技术研发者角度看 RAID

在一个 IO 的旅程中基本上都会经历一个非常重要站点，他就是 RAID。提起 RAID，基本上是无人不晓，每个人都能说上一点。例如 RAID5、RAID6 之类的概念，此外，RAID 可以提高数据可靠性，但是考虑到 IO 性能等问题，很多人都会采用硬件 RAID 卡对 IO 进行加速。诸如此类的东西，或许大家都知道。从表面上看，RAID 似乎很简单，多块磁盘聚合在一起，采用一定的数据分布算法将数据分布到多块磁盘上，这就构成了 RAID。是的，从外表来看，RAID 的确很简单，经过这么多年的发展，很多人也将 RAID 放入了古老技术的行列。但是，从技术研发者的角度来看，RAID 其实很复杂，为什么这么说呢？这主要是因为在设计 RAID 的时候，有很多棘手的问题需要解决。这些棘手的问题主要包括如下几个方面：

1） 数据小写问题。RAID 把多块磁盘绑定在一起，并且在水平方向进行条带切分，每个条带都会横跨多块磁盘。重要的是每个条带会生成一个或者多个校验冗余数据。在写入操作的时候，需要更新校验冗余数据。在写入数据量比较小的情况下，为了生成校验冗余数据，那么需要读取条带中没有被更新过的数据，然后计算得到校验数据，最后将条带数据写入磁盘。显然，在整个过程中，存在一个写放大的问题，一个简单的写操作变成了“读-更新-写”的操作。所以，小写问题对 RAID 的性能带来极大的影响。

2） 校验数据计算问题。对于常用的 RAID5、RAID6，是需要计算校验数据的。这些校验数据的计算看起来不是很复杂，其实是很耗 CPU 时间的。常用的 CPU 不擅长数值计算，如果直接将乘法等运算交给 CPU 来做，那么 RAID6 的复杂度就会变得很高。大量的计算会使得 IO 延迟大为增加，从而使得 RAID 很难在对实时性和吞吐量要求比较高的场合应用。

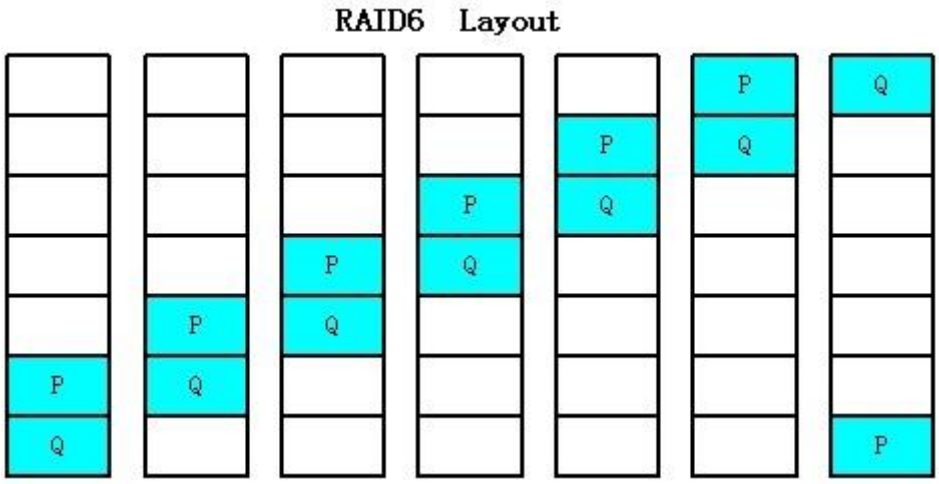
3） 数据重构问题。随着磁盘容量的不断增加，数据重构的时间会变得越来越长。较长的数据重构时间会使得数据可靠性变得越来越差。数据重构时间是一个 RAID 非常棘手的问题，该问题导致现有 RAID 技术很难在大数据环境下应用。要想解决该问题，需要从架构设计上颠覆现有 RAID 技术，并且打破现有 RAID 在数据重构过程中的读、写性能瓶颈。只有这样，才有可能解决 RAID 数据重构的性能问题。

4） 一致性性能问题。当 RAID 处于降级模式时，系统中不仅会存在上层应用的 IO，而且还会存在数据重构的 IO，并且这两类 IO 会交织融合在一起，使得双方的性能都变得很差。这主要是因为，应用 IO 和数据重构 IO 的交织，使得磁盘读写出现抖动，从而应用 IO 和数据重构 IO 的性能都急剧下降。所以，如何保证应用 IO 性能在正常模式下和降级模式下都达到一个比较均衡的水平，是 RAID 设计者需要考虑的问题。

上述的四个问题是 RAID 研发过程中面临的最大问题，有些问题至今都很难完满解决。所以，RAID 技术不是一种古老技术，而是一个欣欣向荣的存储底层技术。抛开 RAID 这个传统的概念，剖析一下 RAID 技术所要解决的问题究竟是什么？其实，大家也知道，RAID 主要是为了解决存储性能问题和数据可靠性问题。存储性能问题的解决其实是一个附加值的体现，数据可靠性问题的解决才是 RAID 的精华所在。所以，简单来看，RAID 技术的本质是想解决数据可靠性问题。那么为了解决这个问题，技术研发者是可以抛开现有 RAID 架构，采用新的架构、方法实现数据冗余，从而达到数据可靠性的效果。所以，在大数据环境下，广义 RAID 还会有其重要作用的。

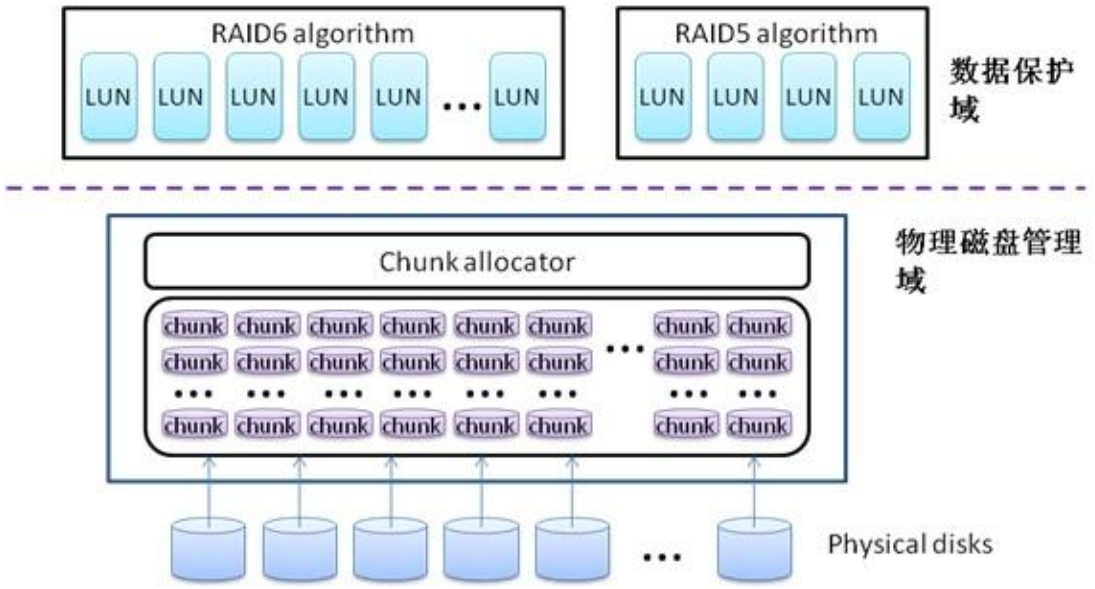
RAID 数据分布

RAID 的数据分布是很讲究的。在 RAID5 之前，所有的校验数据集中在一块磁盘上，此时，这个磁盘就成了校验数据瓶颈。为了解除这种瓶颈，RAID5、RAID 将校验数据采用分布是存储的方式。下图是 RAID 的校验数据分布。



这种数据分布被称之为左循算法，PQ 分布是可以通过条带号计算出来的。从上图我们也可以知道 RAID 的数据分布和物理磁盘是一一对应的，通过逻辑条带号就可以知道数据在每个物理磁盘上的分布情况。仔细考虑一下，RAID6 之类的算法其实是一种数据保护算法，是一种数据编解码算法。传统的 RAID 将这种数据保护算法和物理磁盘进行了紧耦合绑定，这是传统 RAID 的一大特征。

其实，数据保护算法完全可以在一种逻辑域中完成，物理磁盘的管理可以在一种物理域中实现。逻辑域和物理域可以通过某种映射关系联系在一起。逻辑域和磁盘物理域拆分的最主要目的是加速数据重构过程，并且可以优化应用层性能一致性的问题。拆分之后的两个域之间关系可以表示如下：



通过这种架构上的拆分，RAID 中的数据分布将会发生本质上的变化。例如一个盘阵系统中

有 20 块磁盘，其中创建了一个 RAID5 和一个 RAID6。那么，通过物理磁盘管理域中 allocator 的资源分配，RAID5 和 RAID6 中的数据将会均匀分布在所有的 20 块磁盘中，而不是 RAID5 的数据分布在固定 10 块磁盘上，而 RAID6 的数据分布在另外 10 块磁盘上。这种数据布局打破了传统 RAID 的规则数据分布，带来的最大好处就是提升数据重构性能。通过上述描述，我们可以知道一块物理磁盘上存储的数据涉及到所有数据保护域的 RAID。所以，当一块物理磁盘损坏之后，所有的 RAID 都会参与到数据重构的过程中，而不仅仅是一个 RAID 的事情了。这就好比传统 RAID 在处理数据重构时，都是“自家各扫门前雪”，但是，引入新型架构之后，数据重构过程就变成了“众人拾材火焰高”。
随着磁盘容量的进一步增大，数据重构和一致性性能将会变成 RAID 的两大最重要问题。在现有 RAID 基础上，如果不改变 RAID 架构，那么这个问题将会变得很棘手，难于解决。除非采用影响性能的擦除码来增强冗余度。

RAID 算法实现

RAID6 算法是 RAID 实现过程中需要考虑的重要问题，从算法本身来看，RAID6 算法是比较简单的：

$$P = D_0 + D_1 + D_2 + D_3 + \cdots + D_{n-1}$$

$$Q = g_0 * D_0 + g_1 * D_1 + g_2 * D_2 + g_3 * D_3 + \cdots + g_{n-1} * D_{n-1}$$

P 码运算很简单，只需要把所有的数据累加起来就可以了。Q 略有复杂，需要涉及到乘法运算。将上面的算式表示成矩阵的形式，表示如下：

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ & & & \cdots & & \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ g_0 & g_1 & g_2 & \cdots & g_{n-2} & g_{n-1} \end{bmatrix} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ \cdot \\ \cdot \\ \cdot \\ D_{n-2} \\ D_{n-1} \end{bmatrix} = \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ \cdot \\ \cdot \\ \cdot \\ D_{n-2} \\ D_{n-1} \\ P \\ Q \end{bmatrix}$$

输入数据和编码矩阵的乘积就是我们需要存储的 RAID 数据。
我们知道 CPU 的强项不在于算术运算，一次乘法运算需要耗费很多个指令周期，因此，如何避免编解过程中的乘法运算成了 RAID 技术实现的重点。要做到高效编码，需要引入一个有限域——加逻华域。在加逻华域中，加法运算变成了异或 XOR 运算，显然能够提高效率。但是，在加逻华域中，乘法运算还是二进制乘法，还具有一定的复杂度，为了将这种乘法运算转换成简单的异或操作，又引入了对数/反对数操作。我们知道，通过对数操作可以将乘法转换成加法操作，这个特性被 RAID 编解码采用了。例如，A*B 的操作就可以通过如下方式进行转换：

$$A * B = C \leftrightarrow \log_g^A + \log_g^B = \log_g^C$$

$$C = g^{(\log_g^A + \log_g^B)}$$

所以，A*B 的计算过程可以分解成如下四步：
1） 通过对数表查到 logA 的值 A1
2） 通过对数表查到 logB 的值 B1
3） 将 A1 和 B1 进行异或运算，得到 C1
4） 通过反对数表查到 C1 对应的值 C
通过上述方法，可以将 RAID 的所有编解码操作转换成了异或 XOR 操作，从而可以提高 IO 效率。

Linux 中的 MD 开源 RAID（1）

1、前言

RAID 是 IO 路径中的关键模块，甚至在整个存储系统中，RAID 是最为核心和复杂的模块。在 Linux 操作系统中，提供了一个开源的 RAID，那就是 MD RAID。该 RAID 可以实现 RAID0、RAID1、RAID5 和 RAID6 的功能，在产业界得到了广泛的应用。MD RAID 对外采用块设备的形式导出，因此，通过块设备层的调度，BIO 请求会被发送到 RAID 模块进行进一步的 IO 处理。在 MD RAID 一层，请求会被按照条带的方式进行切分，然后再按照 RAID 中磁盘的个数进行磁盘请求的映射。在请求的处理过程中，会存在条带小写问题，因此会引入“读-修改-写”的过程。在一个写请求发送往多个磁盘之前，校验信息（Parity）需要被计算处理，并且按照 left-asymmetric 或者 right-asymmetric 的数据布局方式将校验信息放置到多个磁盘中。多年前，本人对 MD-RAID 进行了深入的代码分析，从中可以一窥 RAID 的实现机制。本代码分析基于 Linux-2.6.18 版本。

2、标记定义

MD-RAID 中定义了很多标记，这些标记其实是一个 Stripe 在处理过程中所处的状态信息。具体的标记定义如下所示：

```
#define     R5_UPTODATE        0       /* 缓存页中有当前数据 */
#define     R5_LOCKED          1       /* IO请求已经被分发下去 */
#define     R5_OVERWRITE       2       /* 整页写操作 */
/* and some that are internal to handle_stripe */
#define     R5_Insync          3       /* rdev && rdev->in_sync at start */
#define     R5_Wantread        4       /*调度一个读IO的过程 */
#define     R5_Wantwrite       5       /*调度一个写IO的过程 */
#define     R5_Syncio          6       /* this io need to be accounted as resync io */
#define     R5_Overlap         7       /* There is a pending overlapping request on this block */
```

标记说明：
R5_UPTODATE 和 R5_LOCKED 组合代表的操作状态

Empty：
！R5_UPTODATE，！R5_LOCKED：没有任何数据，也没有活动的请求
Want：
！R5_UPTODATE， R5_LOCKED：一个 read 请求已经被分发到相应的逻辑块
Dirty：
R5_UPTODATE， R5_LOCKED：缓存区中有新的数据，并且需要往磁盘上写
Clean：
R5_UPTODATE，！R5_LOCKED：缓存区中有数据，用户可以读取，和磁盘上的数据一致

写方法定义：
更新写：
#define RECONSTRUCT_WRITE 1
读-修改-写
#define READ_MODIFY_WRITE 2

条带状态定义：

```
1  #define STRIPE_ERROR          1 : 条带出现错误
2  #define STRIPE_HANDLE         2 : 条带正在处理
3  #define STRIPE_SYNCING        3 : 条带数据正在同步之中
4  #define STRIPE_INSYNC         4 : 条带已经同步
5  #define STRIPE_PREREAD_ACTIVE 5 : 条带准备活动/有效
6  #define STRIPE_DELAYED        6 : 条带删除
```

3、主要数据结构

3.1 Stripe_head 数据结构

条带头部信息定义如下：

```
struct stripe_head {
        /* hash表的指针，连接所有stripe结构 */
        struct stripe_head  *hash_next, **hash_pprev;
        struct list_head        lru;
        /* RAID5的私有配置信息 */
        struct raid5_private_data       *raid_conf;
        sector_t            sector;                         /* sector of this row */
        /* 校验盘的索引号 */
        int                         pd_idx;
        /* 操作状态标记信息 */
        unsigned long           state;
        atomic_t            count;                          /* nr of active thread/requests */
        spinlock_t                  lock;               /* 自旋锁 */
        struct r5dev {
                struct bioreq;
                struct bio_vecvec;
                struct page       *page;        /* page缓存 */
                /* toread: MD设备的读请求bio
                 * towrite: MD设备的写请求bio，第一阶段写的bio指针
                 * written: MD设备的血请求bio，已经开始被调度写，第二阶段写的bio指针
                 */
                struct bio*toread, *towrite, *written;/* three kinds of queue */
                /* 这一页中的块 */
                sector_t   sector;
                unsigned long  flags;       /* 操作标记 */
         } dev[1]; /* allocated with extra space depending of RAID geometry */
};
```

RAID5 的写过程分成两个阶段：首先从 stripe 上读取数据，然后通过校验和计算之后，再往相应的 stripe 上写入数据。towrite 这个 bio 对应于前一阶段的写，即从 stripe 上读取数据，written 对应于后一阶段的写，即将计算完毕的数据写到相应的 stripe（磁盘）上去。towrite 和 written 结构体的交换发生在 compute_parity（）函数中，以便调度函数处理。

这个数据结构很重要，RAID5 的操作采用条带化的方式，即每个磁盘上面分出一个 chunk，多个磁盘的 chunk 组成一个 stripe，在 n 个 chunk 中，有一个 chunk 用作校验盘。从 RAID5 开始，stripe 的思想被应用到磁盘阵列中。

在 stripe_head 结构体中，定义了*toread，*towrite，*written 三个 bio 结构体，用于数据读写缓存。将读写缓存分开简化了读写操作，

条带是 RAID5 操作的最基本单元，其采用 hash 表的方式将多个条带信息组织在一起。

3.2 Private_data 数据结构

在 MD 驱动中，每类 RAID 都定义了一个 private_data 的数据结构体，在该结构体中包含了操作磁盘阵列的链表、属性参数等内容。RAID5 的结构体定义如下：

```c
struct raid5_private_data {
        /* 条带操作的hash链表 */
        struct stripe_head  **stripe_hashtbl;
        /* MD设备结构体指针 */
        mddev_t                 *mddev;
        /* MD的扩展设备 */
        struct disk_info        *spare;

/* chunk_size：RAID5设备的chunk长度，可能包括一个或者多个扇区，
* chunk_size >> 9将返回每个chunk包含的扇区数（sectors per chunk）。RAID5
* 设备对磁盘的操作是以扇区为单位的，但是，单位条带的长度为chunk，因此，
* 必须完成扇区和chunk之间的转换
*/

/* algorithm：该域描述了RAID5所采用的校验分布算法，通过该算法RAID5可以
* 得到parity disk的分布。RAID5支持四种不同的校验分布算法，即：
* 1、向左非对称算法  ALGORITHM_LEFT_ASYMMETRIC
* 2、向右非对称算法  ALGORITHM_RIGHT_ASYMMETRIC
* 3、向左对称算法    ALGORITHM_LEFT_SYMMETRIC
* 4、向右对称算法    ALGORITHM_RIGHT_SYMMETRIC
*/
        int             chunk_size, level, algorithm;

        /* raid_disks：磁盘总数
         * working_disks：工作盘的数量
         * failed_disks：失效盘的数量
         */
        int             raid_disks, working_disks, failed_disks;
        int             max_nr_stripes;

        /*下面这些list是条带列表*/
        struct list_head        handle_list; /* 需要信息处理的条带列表 */
        struct list_head        delayed_list; /* 已经发送请求，延迟处理的条带列表 */

        atomic_t            preread_active_stripes; /* */

        char                cache_name[20];
        kmem_cache_t        *slab_cache; /* for allocating stripes */
        /*
         * Free stripes pool
         */
        atomic_t            active_stripes;

        /* inactive的stripe列表，当stripe hash中没有需要的stripe时，
         * 可以在inactive_list中申请一个stripe。
         */
        struct list_head        inactive_list;
        wait_queue_head_t       wait_for_stripe;
        wait_queue_head_t       wait_for_overlap;
        /* release of inactive stripes blocked,
         * waiting for 25% to be free
         */
        int                     inactive_blocked;

        spinlock_t              device_lock;
        struct disk_info        disks[0];
};
```

4、主要函数说明

4.1 RAID5 中主要函数总结

1. error( )，该函数用来出错处理，出错处理的过程是设置标志位，然后守护进程调用 recovery 的时候检测到标志位，就对出错信息进行处理。
2. raid5_computer_sector( )，compute_blocknr( )函数用来将上层的逻辑块映射成物理设备

的逻辑块号，另外还用来计算逻辑块号对应的 stripe 索引以及校验盘的索引号，这个函数应该实现了 parity-algorithm。

3. handle_stripe( )，该函数是处理 stripe 中 IO 请求的关键函数
4. make_request( )，进行设备的 I/O 操作
5. sync_request( )，进行设备的数据同步操作
6. raid5d( )，内核的 RAID 管理守护进程
7. run( )，RAID 运行函数，这个函数在 personality 中

从上面的总结可以看出，系统中最重要的函数是 handle_stripe( )。与 RAID 磁盘阵列打交道的函数都要通过该函数。

4.2 raid5_compute_sector

该函数实现了 RAID5 条带（stripe）的管理，通过这个函数可以实现如下三个方面的功能：

1. 计算逻辑块号在实际物理设备中的块号
2. 设备块号在 raid 磁盘阵列中的索引号
3. 计算得到校验盘在磁盘中的索引号

对 RAID5 设备的块 I/O 请求最终映射到一个或多个成员磁盘上，这个映射是在函数 raid5_compute_sector（）中完成的，函数原型如下：

```
static unsigned long raid5_compute_sector(unsigned long r_sector, unsigned int raid_disks, unsigned int
```

参数说明如下：

r_sector：请求 I/O 的逻辑扇区号

raid_disks：RAID5 设备的磁盘总数

data_disks：数据磁盘数，data_disks=raid_disks-1

conf：RAID5 设备的私有数据结构体

raid5_compute_sector（）函数的目的是计算出响应该 IO 请求的数据磁盘索引和校验磁盘索引，以及在这些磁盘上的物理扇区索引。前两者的计算结果通过指针 dd_idx 和 pd_idx 返回，物理扇区索引值直接返回。

函数执行过程：

1. 根据逻辑扇区编号 r_sector 计算出扇区所在的 chunk 编号和在 chunk 内的扇区偏移

```
chunk_number = r_sector / sectors_per_chunk
chunk_offset = r_sector % sectors_per_chunk
```

2. 如果不考虑校验数据的因素，编号为 chunk_number 的数据块所在的磁盘编号为：

```
dd_idx = chunk_number % data_disks
```

chunk 在对应磁盘上的物理 stripe 号为：stripe = chunk_number / data_disks。由此，可以计算出逻辑扇区号在目标磁盘上的编号：

```
new_sector = (sector_t)stripe * sectors_per_chunk + chunk_offset
```

在数据操作的时候，可以直接往 dd_idx 盘的 new_sector 位置处写数据。

3. 以 raid_disk 为一组，则对应 chunk 在组中的编号为：stripe % raid_disks。

根据不同的算法得到不同的 parity disk 和 data disk 的索引。

向左算法的 parity disk 的编号为：data_disks – stripe % raid_disks

向右算法的 parity disk 的编号为：stripe % raid_disks

4. 数据盘编号计算如下：

不对称算法：chunk_number % data_disks，如果该值大于/等于校验磁盘编号，则数据盘需要加 1

对称算法：(校验磁盘号 + 1 + chunk_number % data_disks) % raid_disks

4.3 compute_block

函数原型：static void compute_block(struct stripe_head *sh, int dd_idx);

输入参数：

*sh：需要操作的条带

dd_idx：数据盘索引号，需要恢复数据的盘

无论是从故障盘上恢复数据，还是在同步过程中将故障盘的信息恢复到备用盘，都需要从磁盘条带上将故障信息恢复出来，这一过程可以在 compute_block 函数中完成。

其中 sh 指向要计算的条带单元所在的条带头结构体，dd_idx 为故障磁盘的编号。需要注意的是，dd_idx 所处的条带单元在条带头结构体中可能代表一个数据条带，或者一个校验条带。计算方法采用异或算法，所有操作都是在成员磁盘缓冲头结构体（page）上进行。为保证计算正确，在调用前，已经将所有其它磁盘上条带单元的数据读取到对应这个磁盘的缓冲头结构体所关联的缓冲区中。在计算过程中，进行检查，如果缺少任何一个磁盘的数据，则打印内核错误消息。在计算完成后，修改故障磁盘对应缓冲头结构体的状态标志位，以表明其中的条带数据是有效的。

4.4 compute_parity

compute_parity（）函数实现如下两个方面的功能：

1. RAID5 校验和的计算
2. 将 towrite 结构体挂接到 written 中，使得写操作进入第二个阶段

在 RAID5 程序中，完成数据校验有两种方法：

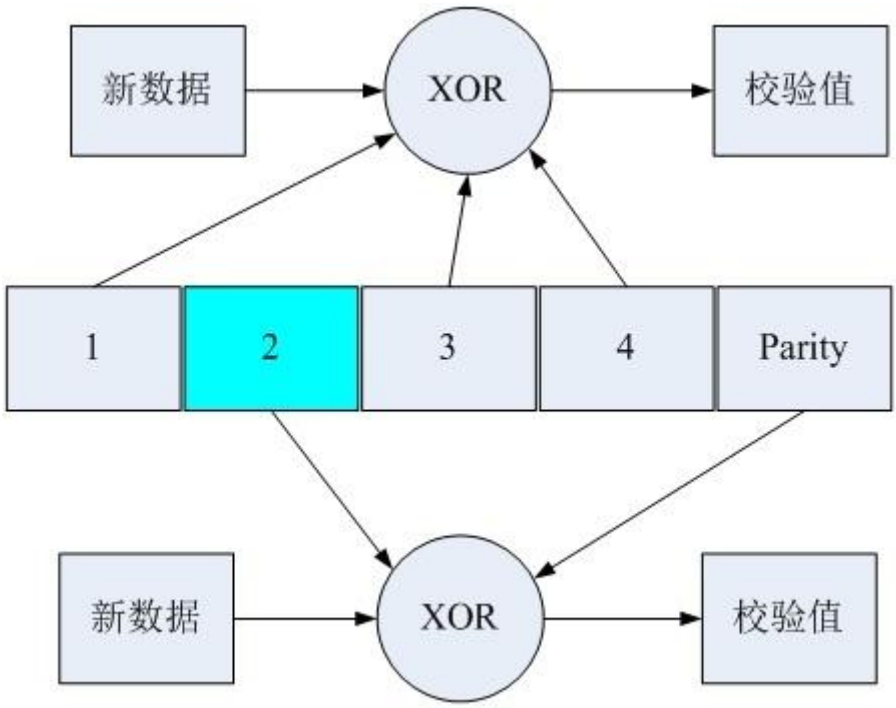
1. read-modefy-write：读-修改-写

采用这种方法的基本运算过程如下：[（旧校验数据）XOR（旧数据）]XOR（新数据）=新校验值。

2. 2reconstruct-write：更新写

采用这种方法的基本运算过程如下：（好的旧数据）XOR（新数据）=新的校验值

这两种方法实现的示意图如下：



（图中假设第 2 块数据需要更新）

RAID5 校验算法的实现：

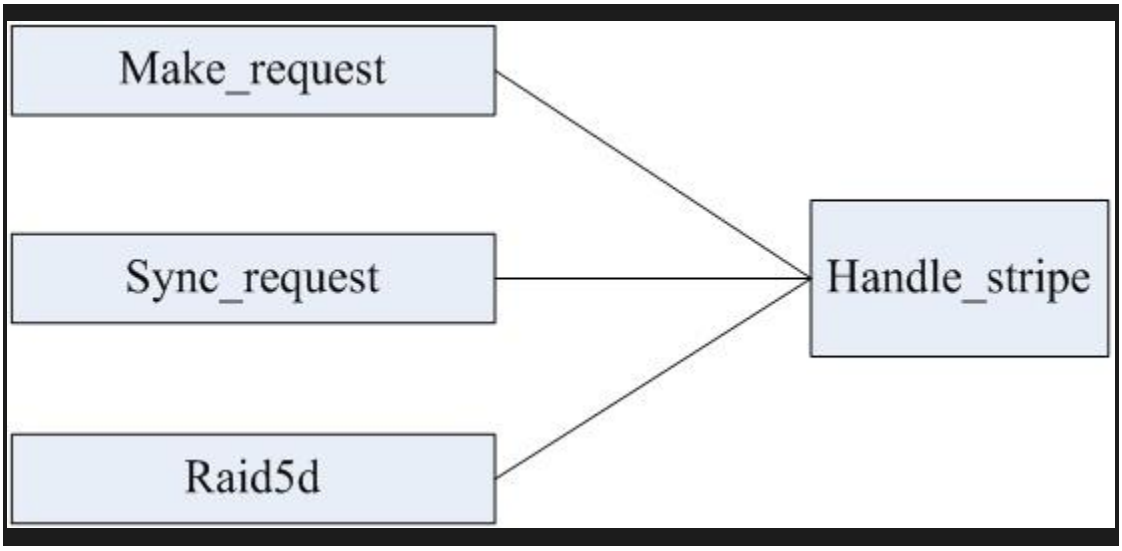
RAID5 在写的时候有两种方法，一种是读-修改-写操作，另一种是更新写操作。

在读-修改-写操作过程中，需要将旧的校验值、旧的更新数据读出来，然后需要做两次 XOR 操作，其算法的基本过程如下：[（旧校验数据）XOR（旧数据）]XOR（新数据）=新校验值。

更新写过程需要将无须更新的旧数据读出来，做一次 XOR 操作（这里所说的一次 XOR 操作是指调用 check_xor()这个宏一次），其算法的基本过程如下：（好的旧数据）XOR（新数据）=新的校验值。

4.5 handle_stripe

handle_stripe 函数是 RAID5 驱动中最重要的函数。在 RAID5 驱动中有三个函数会调用 handle_stripe（），如下图所示：



访问该函数的话可能返回的结果：

1. 如果有数据，那么返回读请求的数据
2. 如果数据被成功的写入磁盘，那么返回写请求
3. 调度读进程
4. 调度写进程
5. 返回奇偶校验的确认信息

handle_stripe 函数实现了 IO 读写等功能，主要功能说明如下：

1. 当 Page 页（缓存）中存在数据的时候，可以将缓存中的数据拷贝到 bio 中，实现 IO 的正常读写，并且返回
2. 当 RAID 出现故障的时候，不需要进行任何操作了，但是需要将系列 IO 读写请求取消掉。
3. 实现 IO 写操作，RAID5 的 IO 写操作比较复杂，其分为满块写和非满块写，当为非满块写的时候需要首先读取 stripe 中的数据，然后再计算校验和写入磁盘（置相应的标志位）。调度一个写过程。
4. 实现 IO 读操作，完成一个读过程的调度，其操作过程是置相应的标志位 want_read。
5. 实现 IO request 的请求分发，构造 bio，调用 generic_make_request ( )函数。

下面对 handle_stripe 函数中实现的功能单元进行分析。

1、IO 读完成操作

当 handle_stripe（）调用 generic_make_request（）函数向底层驱动程序分发 bio 请求以后，接下来的读数据操作就由底层的驱动和硬件设备完成。当驱动程序完成读操作之后回调读完成函数：raid5_end_read_request（）。在这个回调函数中，清除 IO 读标记 R5_LOCKED，并且有可能置读数据有效标记 R5_UPTODATE（如果此次操作没有错误的话）。关键代码如下：

```
1  set_bit(R5_UPTODATE, &sh->dev[i].flags);
2  clear_bit(R5_LOCKED, &sh->dev[i].flags);
```

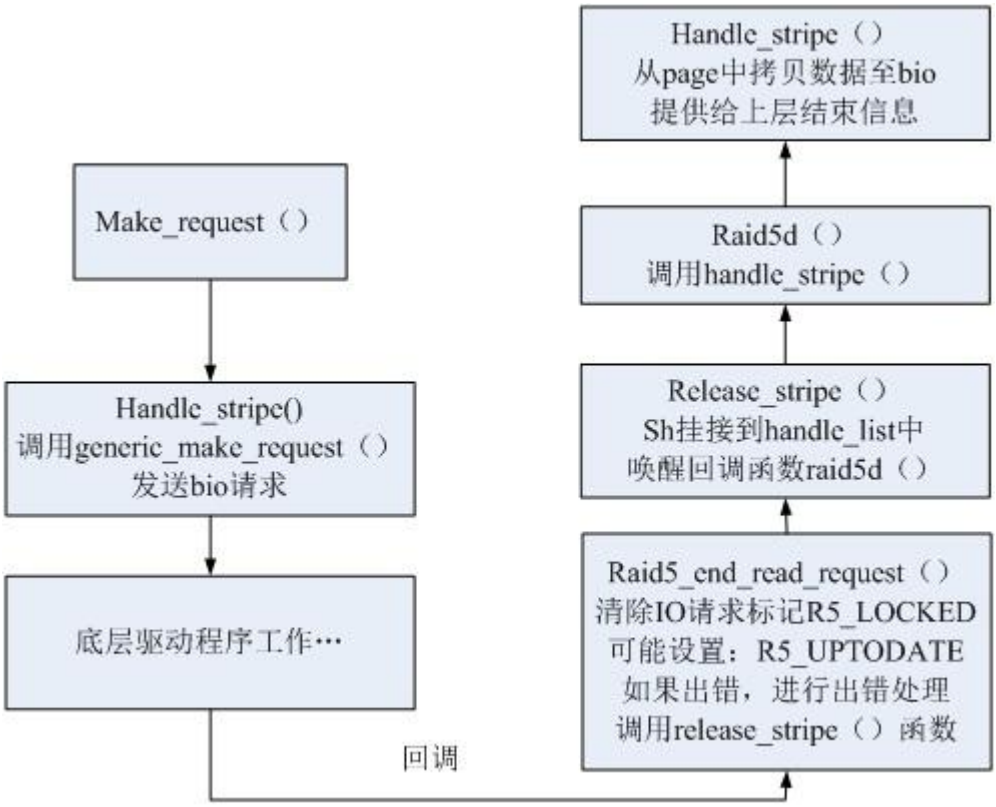
并且设定 list 挂接标记 STRIPE_HANDLE，代码如下：

```
1  set_bit(STRIPE_HANDLE, &sh->state);
```

最后调用 list 分配函数 release_stripe（）。

在 release_stripe（）函数中，将 sh 挂接到 handle_list 中，并且唤醒守护线程 Raid5d（）。通过 Raid5d（）函数调用 handle_stripe（）。

此时，程序满足条件 toread R5_UPTODATE，因此，进入 IO 读完成代码区，该代码的实现比较简单，通过 copy_data（）函数调用将 page 缓存中的数据拷贝到 bio 中，并且返回给上层。具体的操作过程可以描述如下：



2、故障发生时清除 IO 请求

RAID5 只能处理最多坏一个盘的情况，如果坏盘数目大于 1 的话，那么 IO 请求等操作将变得没有任何意义。

在 handle_stripe（）中，会统计所有的标记位数量和坏盘的数目，但出现 failed > 1 的情况时，停止一切的 IO 读写操作，调用结束函数直接结束 IO 读写操作。

当驱动程序正在进行 syncing 的时候，如果发现坏盘数据大于 1，那么立即停止 sync 操作，调用 md_done_sync(conf->mddev, STRIPE_SECTORS,0)函数来停止同步操作。

3、写操作完成功能

写操作是一个比较复杂的过程，守护进程 raid5d（）第二次调用 handle_stripe（）函数，即第三次进入 handle_stripe（）函数时（第一次由 make_request（）调用，第二次由 raid5d（）调用），会执行写操作完成功能。

写操作结束时的条件如下：

1. written 有效
2. RAID 盘阵同步，R5_Insync
3. 没有 IO 请求操作，!R5_LOCKED
4. 数据是有效的，R5_UPTODATE

在（1）满足的条件下，或者满足如下两个条件：

存在一个故障盘，failed == 1

故障盘的索引号正确，failed_num == sh->pd_idx

4、读调度功能

在以下四种情况下，内核需要调度读操作：

- ✧ 正常数据读。标记的状态为：!R5_LOCKED !R5_UPTODATE toread
- ✧ 非整页写的时候，需要读取数据，标记状态为：towrite !R5_OVERWRITE
- ✧ 同步的时候需要读取数据，标记的状态为：syncing

✧ 出错的时候需要读取数据，标记的状态为：failed toread || (towrite && !R5_OVERWRITE)
如果在降级模式下，即存在一块坏盘的情况下，可以通过 compute_block（）来计算得到所需要的数据。如果在同步的情况下，首先设定相应的标记，来调度一个读数据过程，关键代码如下：

```
set_bit(R5_LOCKED, &dev->flags);
set_bit(R5_Wantread, &dev->flags);
```

在 handle_stripe（）函数的最后，通过 R5_Wantread 标记构造一个读 bio，并且调用 generic_make_request（）函数将这个读 bio 请求分发到底层驱动程序，那么具体的读操作就由低层驱动程序和硬件来实现了。实现完毕之后再调用回调函数 r5_end_read_request（）。接下来的过程就回到 IO 读完成操作这个地方了。

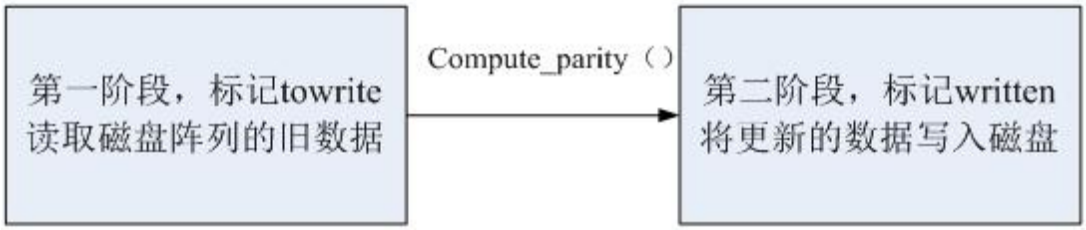
5、写调度功能

RAID5 写操作是驱动程序中比较复杂的一块。他通过多次调用 handle_stripe（）函数来实现一个完整的写过程。

对于 RAID5 的写操作有多种方式，包括满块写和非满块写，又有 read_modefy-write 和 reconstruct_write 两种写方法，这两种写方法和盘阵的具体数据校验方法相关。

整个 RAID5 的写操作可以分为两个阶段，第一个阶段实际上是数据读阶段，标记为 towrite，等到将磁盘里的旧数据读上来之后，调用 compute_parity（）函数进行相应的数据校验，得到新的校验值。然后进入第二阶段，将 towrite 挂接到 written 上，开始将实际缓存中有效的数据写到磁盘上去，完成整个 IO 写过程。

这两个 IO 写过程可以用图表示如下：



写过程的具体描述：

上层发起ＩＯ请求，通过 make_request_fn（）->make_request（）来调用 handle_stripe（）函数，这是写过程第一次进入这个函数。由于没有此时的状态处于 Empty（！R5_LOCKED，！R5_UPTODATE），因此，程序需要判断其需要进行的读写方法，即读-修改-写还是更新写，用两个变量来表示，即 rmw 和 rcw。

选择好读写方法之后，程序会调度一个写进程，因为他需要读取磁盘上的旧数据，关键代码为：

```
set_bit(R5_LOCKED, &dev->flags);
set_bit(R5_Wantread, &dev->flags);
```

在 handle_stripe（）函数的最后会根据这些设定的标志来构造一个读 bio，最后会将这个读请求 bio 通过 gengeric_make_request（）函数发送到底层驱动程序。

接下来的事情由低层驱动程序和硬件完成…

读操作完成之后，底层驱动程序回调 raid5_end_read_request（）函数，该函数是读请求结束处理函数。在该函数中将 R5_LOCKED 置位无效，并且如果没有读错误的情况下，设定 R5_UPTODATE 标记有效（实际上，该标记有没有效，并不是说还要去将 page 缓存中的数据读到 bio 中，后面所涉及到的数据校验操作都是在 page 缓存中完成的）。另外，需要设置 STRIPE_HANDLE 标记，如下：

```
set_bit(STRIPE_HANDLE, &sh->state);
```

该标记的作用是将 sh 挂接到 handle_list 中，然后由 raid5d（）守护进程对 sh 进行操作。
因此接下来的操作为：

raid5_end_read_request（）-> release_stripe（）-> 唤醒 raid5d（）

raid5d（）守护进程调用 handle_stripe（），这是第二次进入 handle_stripe（）。

此次进入 handle_stripe（）函数，满足如下条件：

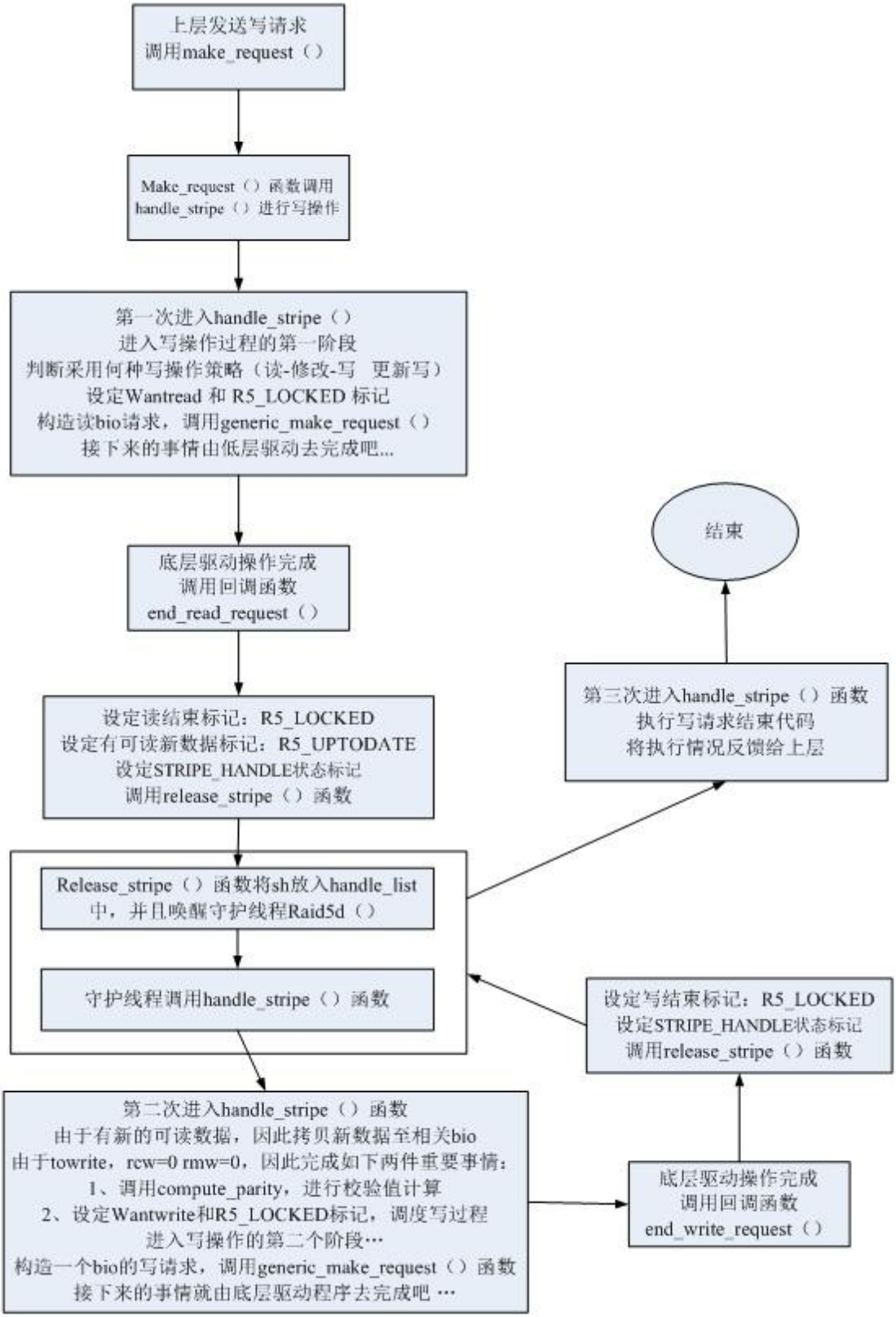
towirte　rmw == 0　rcw == 0

此时可以开始一个正常的写过程调度…

首先需要进行缓存区的数据校验，调用 compute_parity（），通过该函数一方面得到了正确的校验数据，另一方面使得写过程进入第二个阶段，towrite 已近挂接到 written。
标记 R5_Wantwrite 调度一个写过程，即：

```
set_bit(R5_Wantwrite, &sh->dev[i].flags);
```

在 handle_stripe（）函数的最后通过上面的标记构造一个写 IO 的 bio 数据块，并且调用 generic_make_request（）函数将该请求发送到底层驱动程序。
接下来的工作就让底层驱动程序和硬件来完成吧…
请求完成之后，底层驱动程序会调用 raid5_end_write_request（）回调函数，该回调函数清除请求标记 R5_LOCKED，设置 STRIPE_HANDLE 标记，然后调用 release_stripe（）函数，该函数又将 sh 挂接到 handle_list 上，然后唤醒守护进程 raid5d（），raid5d（）会第二次调用 handle_stripe（）。
第三次进入 handle_stripe（）。
此次进入 handle_stripe（）完成写操作结束处理。
写过程的详细描述见下图：



6、IO 请求分发功能

在 handle_stripe（）函数中实现 IO 请求分发功能。构造读写 IO 的 bio。

调用 generic_make_request（）函数将请求分发给底层驱动程序。

7、同步操作的支持

Raid5d() -> md_check_recovery() -> md_do_sync() -> sync_request() -> handle_stripe()

上述过程是一个同步操作过程的函数调用关系链。从这个关系链中可以知道，同步操作最后是由 handle_stripe（）函数来实现的。
程序会判断如下三个标记来决定是否执行同步数据处理程序段：

syncing：该标记说明驱动程序此刻需要完成同步的操作

STRIPE_INSYNC：该标记说明磁盘阵列是否处于同步的状态，如果!(test_bit(STRIPE_INSYNC))，那么说明磁盘处于非同步状态。

failed：标记了故障磁盘实效的数目

满足数据同步的条件之后，驱动进入数据同步的代码段。
该代码段有三种功能：

1. 校验和的确认，if(failed == 0)。当 md_check_recovery（）程序结束同步操作的时候，会仍然置 RECOVERY_NEEDED 标记，这看起来很奇怪，实际上做一次校验确认罢了。这时候，需要用到 if(failed == 0){}下面的代码。

   在这段代码中，调用 compute_parity(sh, CHECK_PARITY)来计算校验和，然后通过 memcmp（）函数进行比较确认。
2. 调用 compute_block（）函数计算恢复数据，然后设置 R5_LOCKED 和 R5_Wantwrite 标记来调度一个写过程。
3. 结束 stripe 同步操作。当处于同步操作时，数据已经处于同步状态后调用 md_done_sync（）函数结束整个同步过程，并且清除 STRIPE_SYNCING 标记。

4.6 make_request 函数说明

函数原型：static int make_request (request_queue_t *q, struct bio * bi)
参数：*q，请求队列

*bi，IO 请求数据结构

各个 RAID Level 的 IO 请求函数相同，但是他们的实现是不一样的。RAID1 中 make_request（）函数的主要功能是将上层的 bio 分发到底层驱动中去，但是，RAID5 中的函数并没有实现这样的功能，其主要实现的功能如下：

1. 通过 raid5_compute_sector（）函数得到逻辑块号所对应的实际物理块号，另外还得到了 RAID 磁盘阵列中所对应的数据盘索引和校验盘索引。
2. 通过 get_active_stripe（）函数得到一个 stripe，如果在 stripe 的 hash 表中无法找到 sector 对应的条带，那么就从 inactive_list 中分配一个 stripe，如果没有多余的条带，那么整个操作无法进行。如果能够得到一个 active 的 stripe，那么将输入的 bio 直接挂接到 active_stripe 上。
3. 调用 handle_stripe（）函数实现真正的 IO 读写请求操作。

从上面的分析可以看出，RAID5 的 make_request（）函数实际上实现了条带（stripe）的查找/申请和 bio 请求数据结构的挂接事情。真正的读写操作由 handle_stripe（）实现。

4.7 sync_request 函数说明

Sync_request（）这个函数是 RAID5 的同步处理函数。
该函数注册到 md 的 mdk_personality_s 结构体下的 sync_request 中。
因此，在 md_do_sync（）函数中可以采用如下方法来调用 RAID5 的同步处理函数：

```
mddev->pers->sync_request(mddev, j, currspeed < sysctl_speed_limit_min);
```

在分析 md_check_recovery（）这个函数的时候，我们可以看到，当需要做数据同步或者数据恢复的时候，md_check_recovery（）是需要调用 md_do_sync（）过程的。
函数原型：static int sync_request (mddev_t *mddev, sector_t sector_nr, int go_faster)
输入参数：　*mddev，md 设备
　　Sector_nr，起始扇区
　　go_faster，需不需要延迟操作
返回值：这一次完成的扇区数目

4.8 raid5d 函数说明

这是 RAID5 的守护线程，该函数在 RAID5 初始化的时候被注册：
mddev->thread = md_register_thread(raid5d, mddev, "%s_raid5");

在守护线程运行的一开始会调用 md_check_recovery()，通过该函数来检查存储是否有故障，如果有故障，那么调用 md_do_sync（）函数。md_do_sync（）函数实际上是不会完成具体同步工作的，它会调用相应级别的 RAID 同步处理函数 sync_request（）去实现具体功能。
Raid5d()守护线程是一个 while(1)的死循环，他的退出条件是：list_empty(&conf->handle_list)。即当 handle_list 为空的情况下，raid5d 退出睡眠。
Raid5d（）守护线程从 handle_list 中得到 active stripe，然后调用 handle_stripe（）函数对该 stripe 进行处理。
Handle_stripe（）处理完之后，该 stripe 又被挂接到不活动的 list（inactive list）上。

4.9 两个 IO 读写回调函数说明

RAID5 中有两个请求结束回调函数，他们为：
1、raid5_end_read_request（）
2、raid5_end_write_request（）
这两个回调函数在 generic_make_request（）的时候被注册到 bio 中。
Raid5_end_read_request（）函数实现如下功能：

- ✧ 清除 IO 请求标志：R5_LOCKED
- ✧ 如果数据有效（update），那么设置数据有效标记：R5_UPTODATE
- ✧ 如果数据读写错误，那么调用 md_error（），需要 recovery。
- ✧ 设置 stripe 的 handle_list 标记，STRIPE_HANDLE，说明要让 Raid5d()调用 handle_stripe 进行处理

Raid5_end_write_request（）函数实现如下功能：

- ✧ u 清除 IO 请求标记：R5_LOCKED
- ✧ u 如果写发生错误（uptodate == 0），那么调用 md_error（），需要 recovery。
- ✧ u 设置 stripe 的标记 STRIPE_HANDLE，说明要让 raid5d（）调用 handle_stripe（）进行处理。

4.10 出错函数 error 说明

error（）函数是 RAID5 出错处理函数，其被注册到 mdk_personality_t 的 error_handler 函数上，所以在 MD 驱动程序中当出现 IO 读写错误的时候，直接调用 md_error（）函数即可。在 raid5.c 文件中，有两个地方调度 md_error（），他们是 raid_end_read_request（）和 raid_end_write_request（）这两个回调函数。当读写 I/O 错误的时候，回调函数就会调用 md_error（），然后设置相应的标志位，进行 recovery 操作。

函数原型：static void error(mddev_t *mddev, mdk_rdev_t *rdev)

参数： *mddev，md 设备的数据结构

*rdev，具体出错的设备

4.11 list 链表处理函数 release_stripe 说明

release_stripe（）函数封装了_release_stripe（）。因此，讨论_release_stripe（）。

函数原型：static inline void __release_stripe(raid5_conf_t *conf, struct stripe_head *sh)

参数：*conf，RAID5 私有数据结构体

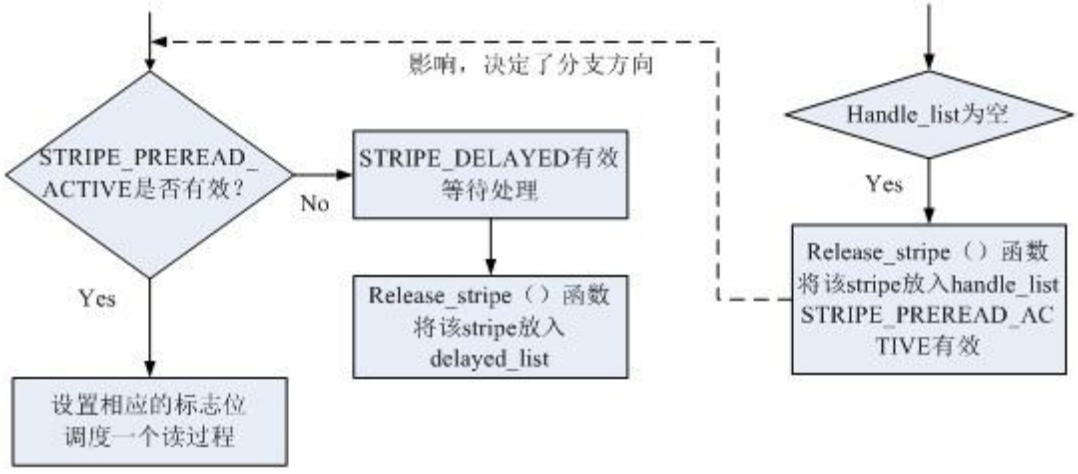
*sh，需要处理的 stripe

该函数实现了 handle_list，delayed_list 和 inactive_list 链表之间的转换关系处理。

首先需要讲一下几个重要的状态标记：

1. STRIPE_PREREAD_ACTIVE：该标记为预读标记，当 RAID5 进行 IO 写的时候需要进行该标记的判断，即在读的第一阶段需要判断该标记。实际上根据字面意思也知道，在写操作的时候需要一个 pre_read 过程，即读操作的第一个步骤。
2. STRIPE_DELAYED：该标记为延迟处理标记，该标记有效时，release_stripe（）函数会将 list 挂接到 delayed_list 中。

从上面的分析中，我们可以看到写操作的第一个步骤是一个 pre_read 的过程，并且是一个延迟操作的过程。延迟操作往往需要等到 handle_list 中的 stripe 处理完成之后，再从 delayed_list 挂接到 handle_stripe 中。因为在写操作的第一阶段需要置 Wantread 标记，调度一个读操作，那么 STRIPE_PREREAD_ACTIVE 标记必须有效。而该标记的设置在 raid5_activate_delayed（）函数中实现。该函数的调用又需要等到 handle_list 为空（raid5d（）中实现）。这个过程可以描述成如下流程：



Release_stripe（）函数执行过程：

1. 当 STRIPE_HANDLE 标记有效的时候，可以将 stripe 挂接到 handle_list 或者 delayed_list 上，否则这个 stripe 将会被挂接到 inactive_list 上。
2. 当 STRIPE_DELAYED 标记有效的时候，stripe 将会被挂接到 delayed_list 上，实现一个延迟处理。否则，stripe 将会被挂接到 handle_list 上。
3. 挂接到 handle_list 或者 delayed_list 上之后，调用 md_wakeup_thread（conf->mddev->thread）函数唤醒守护进程。

5、RAID5 中 I/O 读写方法

RAID5 中 I/O 的读写操作由 make_request 发起，该函数被注册到 mdk_personality_s 结构的 make_request 函数中，当操作系统调用 make_request_fn 函数进行块设备读写操作的时候，直接调用 make_request()函数实现相应功能。
在 RAID1 的 make_request 函数中直接将上层的 bio 分发下去，实现 IO 读写操作，但是在 RAID5 中的实现方法有所不同，其调用了 handle_stripe 函数实现读写操作。
RAID5 中实现 IO 读写操作的函数主要有：

1. make_request（）。该函数传递上层发送的 IO 请求
2. handle_stripe（）。实现 IO 读写请求的主干函数
3. generic_make_request（）。发送 IO 请求至底层驱动程序
4. raid5_end_read_request（）。读操作结束回调函数
5. raid5_end_write_request（）。写操作结束回调函数
6. release_stripe（）。Sh 挂接至 handle_list 处理函数
7. raid5d（）。守护线程

I/O 写操作过程：
写操作过程历经如下函数调用过程：
读取数据过程：
Make_request（）-> handle_stripe（）->generic_make_request（）底层驱动工作…
计算/写数据过程：
Raid5_end_read_request（）-> release_stripe（）-> raid5d ()-> handle_stripe（）底层驱动工作…
结束写过程：
Raid5_end_write_request（）->release_stripe（）->raid5d（）->handle_stripe（）
具体的 IO 写过程参考 handle_stripe（）写过程分析。
I/O 读操作过程：
IO 读过程比较简单，其经历的函数调用过程如下：
调度一个读过程：
Make_request（）-> handle_stripe（）-> generic_make_request（）底层驱动工作…
从 page 缓存中拷贝数据：
Raid5_end_read_request（）-> release_stripe（）-> raid5d（）-> handle_stripe（）
具体的分析可以参考 handle_stripe（）读过程分析。

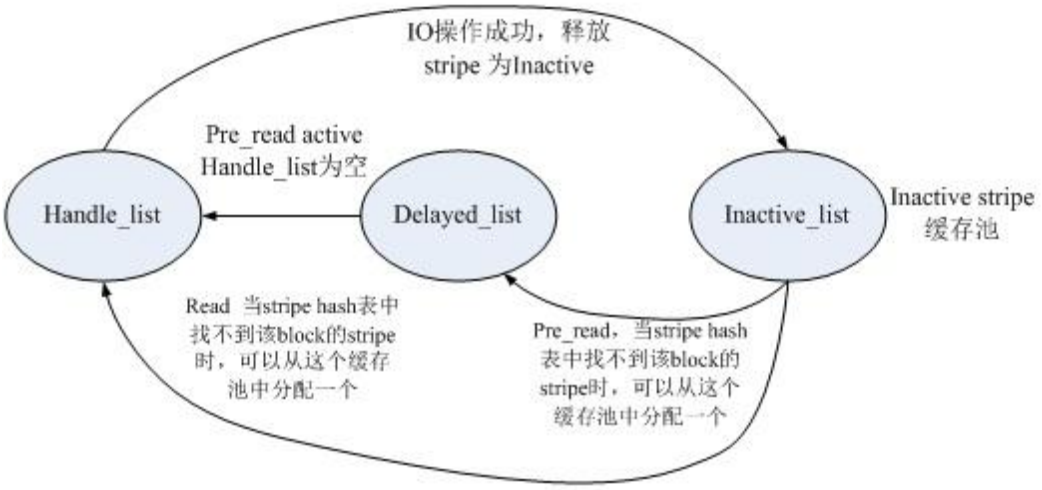
6、几个 list 的关系及数据挂接关系

RAID5 中涉及的几个 list：

1. handle_list：这个 list 中的 stripe 需要分发执行
2. delayed_list：这个 list 中的 stripe 延迟分发执行
3. inactive_list：这个 list 中的 stripe 为不活动的条带

当需要进行一个 IO 操作的时候，首先要获取一个 active stripe（get_active_stripe（）函数实现），这个 stripe 可以从 hash 表中找到，当找不到的时候，可以从 inactive_list 中请求一个（get_free_stripe（）函数实现）。当 handle_stripe（）函数将 stripe 处理完毕之后，release_stripe（）函数又将 stripe 放入 inactive_list。

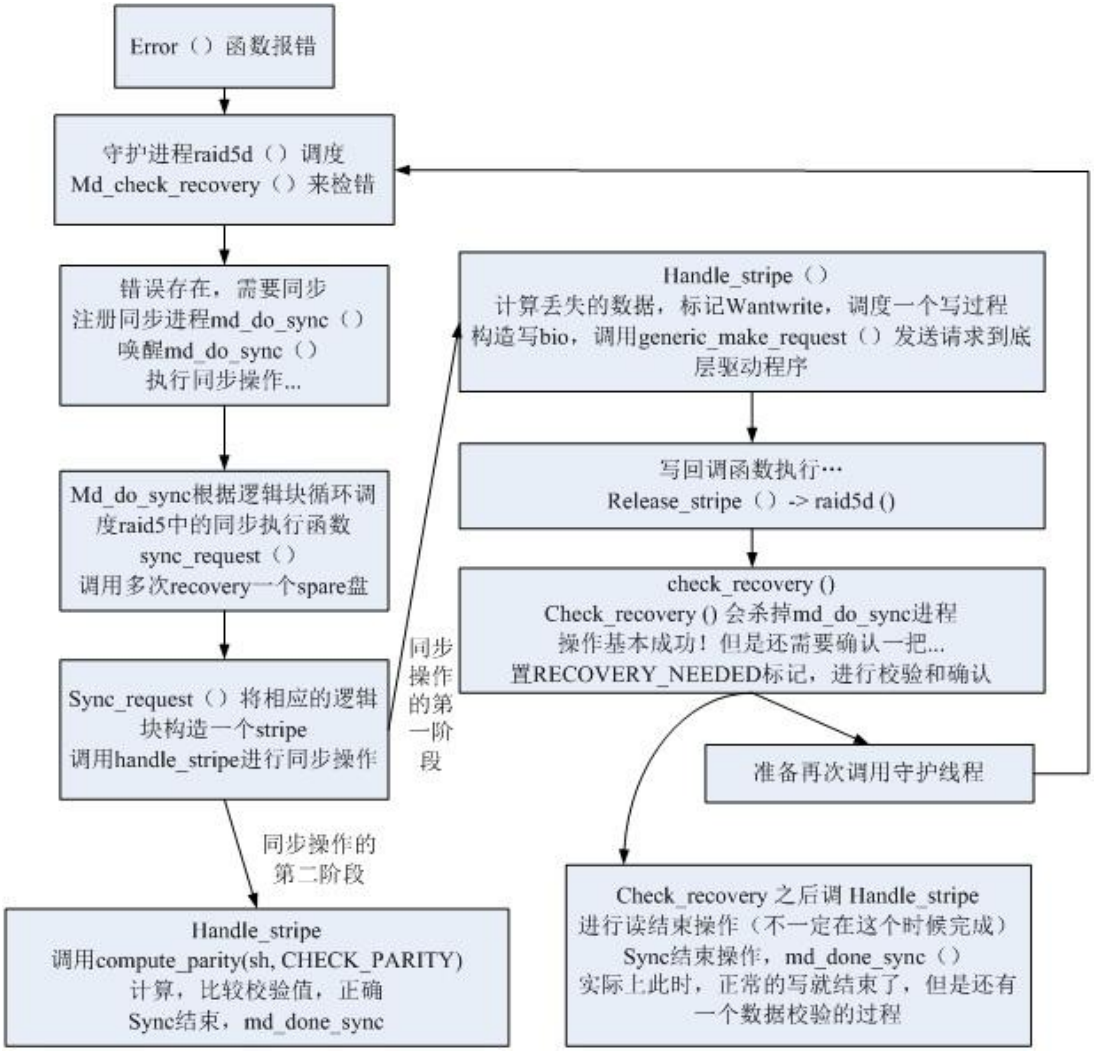
几个 list 的挂接关系可以基本描述如下：



7、错误处理数据恢复方法

RAID 需要进行 IO 的出错处理。在 RAID5 这个级别可以纠正由于一个磁盘故障导致的错误，在 raid 驱动中通过 error（）函数来报错，然后通过 md_check_recovery（）函数来检错，通过 md_do_sync（）、sync_request（）函数来纠错。他的运行机制和相互之间的逻辑关系又是怎样的呢？

基本的数据恢复过程如下图所示：



在 RAID5 系统中，其数据同步/恢复操作分为两个阶段：

1. 数据写阶段，将有效数据写到 spare 盘上去。
2. 数据校验阶段，确认校验和是否正确，如果正确，那么整个 recovery 操作才算真正的结束。