[**理解 QEMU/KVM 和 Ceph（1）：QEMU-KVM 和 Ceph RBD 的 缓存机制总结**](http://www.cnblogs.com/sammyliu/p/5066895.html)

本系列文章会总结 QEMU/KVM 和 Ceph 之间的整合：

[（1）QEMU-KVM 和 Ceph RBD 的 缓存机制总结](http://www.cnblogs.com/sammyliu/p/5066895.html%20)

[（2）QEMU 的 RBD 块驱动（block driver）](http://www.cnblogs.com/sammyliu/p/5095976.html%20)

[（3）存储卷挂接和设备名称](http://www.cnblogs.com/sammyliu/p/5325492.html%20)

  QEMU-KVM 的缓存机制的概念很多，Linux/KVM I/O 软件栈的层次也很多，网上介绍其缓存机制的文章很多。边学习边总结。本文结合 Ceph 在 QEMU/KVM 虚机中的使用，总结一下两者结合时缓存的各种选项和原理。

**1. QEMU/KVM 缓存机制**

   先以客户机（Guest OS） 中的应用写本地磁盘为例进行介绍。客户机的本地磁盘，其实是 KVM 主机上的一个镜像文件虚拟出来的，因此，客户机中的应用写其本地磁盘，其实就是写到KVM主机的本地文件内，这些文件是保存在 KVM 主机本地磁盘上。

  先来看看 I/O 协议栈的层次和各层次上的缓存情况。

**1.1 Linux 内核中的缓存**

  熟悉 Linux Kernel 的人都知道在内核的存储体系中主要有两种缓存，一是 Page Cache，二是 Buffer Cache。Page Cache 是在 Linux IO 栈中为文件系统服务的缓存，而 Buffer Cache 是处于更下层的 Block Device 层，由于应用大部分使用的存储数据都是基于文件系统，因此 Buffer Cache 实际上只是引用了 Page Cache 的数据，而只有在直接使用块设备跳过文件系统时，Buffer Cache 才真正掌握缓存。关于 Page Cache 和 Buffer Cache 更多的讨论参加 [What is the major difference between the buffer cache and the page cache?](http://www.quora.com/Linux-Kernel/What-is-the-major-difference-between-the-buffer-cache-and-the-page-cache)。

  这些 Cache 都由内核中专门的数据回写线程负责来刷新到块设备中，应用可以使用如 fsync, fdatasync（见下面第三部分说明）之类的系统调用来完成强制执行对某个文件数据的回写。像数据一致性要求高的应用如 MySQL 这类数据库服务通常有自己的日志用来保证事务的原子性，日志的数据被要求每次事务完成前通过 fsync 这类系统调用强制写到块设备上，否则可能在系统崩溃后造成数据的不一致。而 fsync  的实现取决于文件系统，文件系统会将要求数据从缓存中强制写到持久设备中。

 仔细地看一下 fsync 函数（[来源](http://man7.org/linux/man-pages/man2/fsync.2.html)）：

fsync() transfers ("flushes") all modified in-core data of (i.e., modified buffer cache pages for) the file referred to by the file

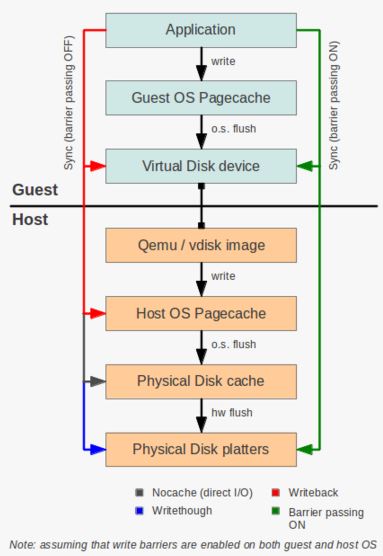
descriptor fd to the disk device (or other permanent storage device) so that all changed information can be retrieved even after the system crashed or was rebooted. This includes writing through or flushing a disk cache if present. The call blocks until the device

reports that the transfer has completed. It also flushes metadata information associated with the file (see stat(2)).

 它会 flush 系统中所有的 cache，包括 Page cache 和 Disk write cache 以及 RBDCache，将数据放入持久存储。也就说说，操作系统在 flush Page cache 的时候， RBDCache 也会被 flush。

**1.2 I/O 协议栈层次及缓存**

**1.2.1 组成**



（[来源](http://www.ilsistemista.net/index.php/virtualization/23-kvm-storage-performance-and-cache-settings-on-red-hat-enterprise-linux-62.html?start=2)）

主要组成部分：

* Guest OS 和 HOST OS Page Cache

Page Cache 是客户机和主机操作系统维护的用来提高存储 I/O 性能的缓存，它是 Linux 虚拟文件系统缓存的一部分，位于操作系统内存中，它是易失性的，因此，在操作系统奔溃或者系统掉电时，这些数据会消失。数据是否写入 Page cache 可以被控制。当会写入 page cache 时，当数据被写入 page cache 后，应用就认为写入完成了，随后的读操作也会从 page cache 中读取数据，这样性能会提高。可以使用 fsync 来将数据从 page cache 中拷贝到持久存储。

在 KVM 环境中，host os 和 guest os 都有 page cache，因此，最好是能绕过一个来提高性能。

* + 如果 guest os 中的应用使用 direct I/O 方式，guest os 中 page cache 会被绕过。
  + 如果 guest os 使用 no cache 方式，host os 的 page cache 会被绕过。

该缓存的特点是读的时候，操作系统先检查页缓存里面是否有需要的数据，如果没有就从设备读取，返回给用户的同时，加到缓存一份;写的时候，直接写到缓存去，再由后台的进程定期涮到磁盘去。这样的机制看起来非常的好，在实践中也效果很好。

考虑到其易失性，需要考虑它的大小，特别是在 KVM 主机上。现在 KVM 主机的内存可以很大。其内存越大， 那么在 Page cache 中还没有 flush 到磁盘（虚拟或者物理的）的脏数据就越多，其丢失的后果就越严重。默认的话，Linux 2.6.32 在脏数据达到内存的 10% 的时候会自动开始 flush。

* Guest Disk （virtual disk device）：客户机虚机磁盘设备
* QEMU image：QEMU 镜像文件
* Physical disk cache

这是磁盘的 write cache，它会提高数据到存储的写性能。写到 disk write cache 后，写操作会被认为完成了，即使数据还没真正被写入物理磁盘。这样，如果 disk write cache 没有备份电池的话，断电将导致尚未写入物理磁盘的数据丢失。要强制数据被写入磁盘，应用可以通过操作系统可以发出 fsync 命令。因此，disk write cache 会提到写I/O 性能，但是，需要确保应用和存储栈会将数据写入磁盘中。如果 disk write cache 被关闭，那么写性能将下降，但是断电时数据丢失将会避免。

* Physical disk platter：物理磁盘

**1.2.2 GUEST 应用读 I/O 过程**

1. GUEST OS 中的一个应用发出 read request。
2. OS 在 guest page cache 中检查。如果有（hit），则直接将 data 从 guest page cache 拷贝到 application space。
3. 如果没有（miss），请求被转到 guest virtual disk。该 request 会被 QEMU 转化为对 host 上镜像文件的 read request。
4. Host OS 在 HOST Page cache 中检查。如果 hit，则通过 QEMU 将 data 从 host page cache 传到 guest page cache，再拷贝到 application space。
5. 如果没有（miss），则启动 disk （或者 network）I/O 请求去从实际文件系统中读取数据，读到后再写入 host page cache，在写入 guest page cache，再到 GUEST OS application space。

从该过程可以看出：

* 两重 page cache 会对数据重复保存，这会带来内存浪费
* 两重 page cache 也会提高 hit ratio，因为往往 guest page cache 比 host page cache 会小很多

QEMU-KVM Linux 支持关闭和开启任一一个 Page cache，也就是说有四种组合模式，分别会带来不同的效果。在各种I/O的过程中，最好是绕过一个或者两个 Page cache。

**1.2.3 Guest 应用 写 I/O 过程**

写 I/O 过程比较复杂，本文其余部分会详细阐述。从 1.3 表格总结，基本上

* writeback/unsafe：app ----qemu write----> host page cache  --- os flush ---> disk cache --- hw flush ---> disk
* none:                   app  --- qemu write----> disk write cache  ---- hw flush ---> disk
* writethrough:        app --- qemu write----> host page cache, disk
* directsync:            app --- qemu write ---> disk

关于 guest os page cache，看起来它主要是作为读缓存，而对于写，没有一种模式是以写入它作为写入结束标志的。

**1.3 客户机磁盘（drive）的缓存模式**

在 libvirt xml 中使用 'cache' 参数来指定driver的缓存模式，比如：

<disk type='file' device='disk'> #对于 type，'file' 表示是 host 上的文件，'network' 表示通过网络访问，比如Ceph

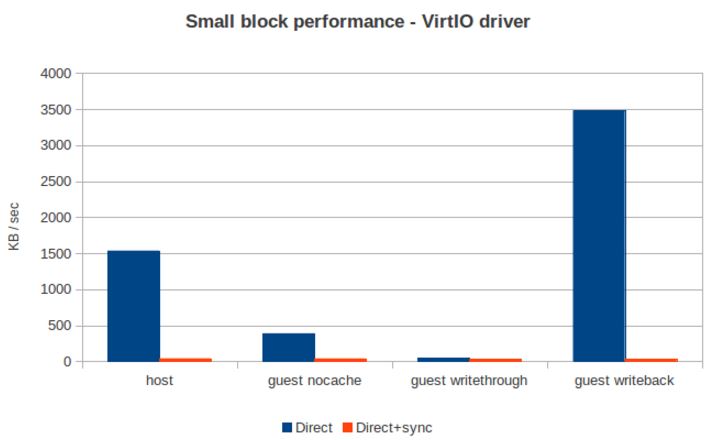
<driver name='qemu' type='raw' cache='writeback'/>

QEMU/KVM 支持如下这些缓存模式作为 ‘cache’ 的可选值：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 缓存模式 | 说明 | GUEST OS Page cache | Host OS Page cache | Disk write cache | 被认为数据写入成功 | 数据安全性 |
| cache = unsafe | 跟 writeback 类似，只是会忽略 GUEST OS 的 flush 操作，完全由 HOST OS 控制 flush | |  | | --- | | Bypass（？不确定） | | E | E | Host page cache | 最不安全，只有在特定的场合才会使用 |
| Cache=writeback | I/O 写到 HOST OS Page cache 就算成功，支持 GUEST OS flush 操作。 效率最快，但是也最不安全 | Bypass（？不确定） | E | E | Host Page Cache | 不安全. （only for temporary data where potential data loss is not a concern  ） |
| Cache=none | 客户机的I/O 不会被缓存到 page cache，而是会放在 disk write cache。  这种模式对写效率比较好，因为是写到 disk cache，但是读效率不高，因为没有放到 page cache。因此，可以在大 I/O 写需求时使用这种模式。  综合考虑下，基本上这是最优模式，而且是支持实时迁移的唯一模式  也就是常说的 O\_DIRECT I/O 模式 | Bypass | Bypass | E | Disk write cache | 不安全. 如果要保证安全的话，需要disk cache备份电池或者电容，或者使用 fync |
| Cache=writethrough | I/O 数据会被同步写入 Host Page cache 和 disk。相当于每写一次，就会 flush 一次，将 page cache 中的数据写入持久存储。  这种模式，会将数据放入 Page cache，因此便于将来的读；而绕过 disk write cache，会导致写效率较低。因此，这是较慢的模式，适合于写I/O不大，但是读I/O相对较大的情况，最好是用在小规模的有低 I/O 需求客户机的场景中。 当不需要支持实时迁移时，如果不支持writeback 则可用。  也就是常说的 O\_SYNC I/O 模式 | E | E | Bypass | disk | 安全 |
| Cache=DirectSync | 跟 writethrough 类似，只是不写入 HOST OS Page cache  也就是常说的 O\_DIRECT & O\_SYNC 模式 | Bypass | Bypass | Bypass | disk | 同 ”O\_SYNC“，对一些数据库应用来说，往往会直接使用这种模式，直接将数据写到数据盘 |
| cache=default | 使用各种driver 类型的默认cache 模式  qcow2：默认 writeback |  |  |  |  |  |

* 性能：  writeback = unsafe > none > writethrough = directsync
* 安全性: writethrough = directsync > none > writeback > unsafe

看看性能比较：



基本结论：

* 各种模式的性能差别非常大
* 对于数据库这样的应用，使用 directsync 模式，数据直接写入物理磁盘才算成功
* 对于重要的数据或者小 I/O 的场景，使用 writethrough
* 对于一般的应用，或者大 I/O 场景，使用 none。这个可以说是大部分情况下的最优选项。
* 对于丢失了也无所谓的数据，可以使用 writeback

**1.4 KVM Write barrier**

**1.4.1 什么是 KVM write barrier**

  上面的基本结论中，writethrough 是最安全的，但是效率也是最低的。它将数据放在 HOST Page Cache 中，一方面来支持读缓存，另一方面，在每一个 write 操作后，都执行 fsync，确保数据被写入物理存储。只有在数据被写入磁盘后，写操作才会标记为成功。这种模式下，客户机的 virtual storage adapter 会被通知不会使用 writeback 模式，因此，它不会主动发送 fsync 命令，因为它是重复的，不需要的。

  那还有没有什么办法使它在保持数据可靠性的同时，使它的效率提高一些呢？答案是 KVM Write barrier 功能。新的 KVM 版本中，启用了 “barrier-passing” 功能，它能保证在不管是用什么缓存模式下，将客户机上应用写入的数据 100% 写入持久存储。

  好吧，这真是个神器。。那它是如何实现的呢？以 fio 工具为例，在支持 write barrier 的客户机操作系统上，在使用 direct 和 sync 参数的情况下，会使用这种模式。它在写入部分数据以后，会使得操作系统发出一个 fdatasync 命令，这样 QEMU-KVM 就会将缓存中的数据 flush 到物理磁盘上。

基本过程：

1. 在一个会话中写入数据
2. 发出 barrier request
3. 会话中的所有数据被 flush 到物理磁盘
4. 继续下一个会话

 看起来和 writethrough 差不多是吧。但是它的效率比 writethrough 高。两者的区别在于，writethrough 是每次 write 都会发 fsync，而 barrier-passing 是在若干个写操作或者一个会话之后发 fdatasync 命令，因此其效率更高。

  也可以看到，使用它是有条件的：

* KVM 版本较新
* 客户机操作系统支持：在较新的 Linux 发行版中都会支持
* 客户机中的文件系统支持 barrier （ext4 支持并默认开启；ext3 支持但默认不开启），而且整个 I/O 协议栈中的各个层次都支持 flush 操作
* 应用需要在需要的时候发出 flush 指令。

  也可以看到，应用在需要的时候发出 flush 指令是关键。一方面，Cache 都由内核中专门的数据回写线程负责来刷新到块设备中；另一方面，应用可以使用如 fsync(2), fdatasync(2) 之类的系统调用来完成强制执行对某个文件数据的回写。像数据一致性要求高的应用如 MySQL 这类数据库服务通常有自己的日志用来保证事务的原子性，日志的数据被要求每次事务完成前通过 fsync(2) 这类系统调用强制写到块设备上，否则可能在系统崩溃后造成数据的不一致。而 fsync(2) 的实现取决于文件系统，文件系统会将要求数据从缓存中强制写到持久设备中。类似地，支持 librbd 的QEMU 在适当的时候也会发出 flush 指令。

  以 fio 为例，设置有两个参数时，会有 flush 指令发出：

fsync=int

How many I/Os to perform before issuing an fsync(2) of dirty data. If 0, don't sync. Default: 0.

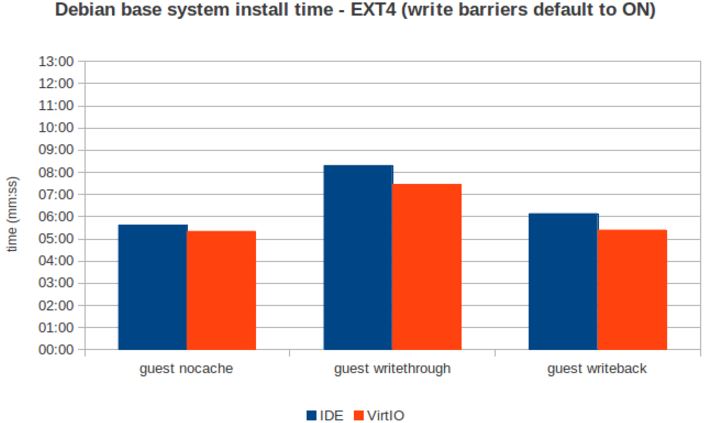
fdatasync=int

Like fsync, but uses fdatasync(2) instead to only sync the data parts of the file. Default: 0.

需要注意的是，频繁的发送（int 值设置的比较小），会影响 IOPS 的值。为了测得最大的IOPS，可以在测试准备阶段发一个sync，然后再收集阶段就不发sync，完全由 RBDCache 自己的机制去 flush；或者需要的话，把 int 值设得比较大，来模拟一些应用场景。

**1.4.2 KVM write barrier 和 KVM 缓存模式的结合**

考虑到 KVM write barrier 的原理和 KVM 各种缓存模式的原理，显而易见，writeback + barrier 的方式下，可以实现 效率最高+数据安全 这种最优效果。



**1.5 小结**

* 在客户机可以启用 write barrier 时，使用 write-back or nocache + barrier，然后应用会在合适的时候发出 flush 指令。
* 在客户机不支持 write barrier 时，如果对读敏感应用，使用 write-back （可以使用 pagecache）；对需要同步数据的应用，使用 noncache；最安全的情况下，使用 writethrough。
* 对于一些能过备用电池或者别的技术（比如设备上有电容等）保证了在掉电情况下数据也不会丢失的情况下，barrier 最好被禁止。比如企业存储的Adatper，或者 SSD。
  + ”If the device does not need cache flushes it should not report requiring flushes, in which case nobarrier will be a noop.“
  + ”With a RAID controller with battery backed controller cache and cache in write back mode, you should turn off barriers - they are unnecessary in this case, and if the controller honors the cache flushes, it will be harmful to performance. But then you \*must\* disable the individual hard disk write cache in order to ensure to keep the filesystem intact after a power failure.“。
  + 一个例子是，Ceph OSD 节点上的 SSD 分区，一般都使用 ”nobarrier“参数 来禁用 barrier。

**1.6 Linux page cache 补充**

Linux 系统中被用于 Page cache 的主内存可以通过 free -m 命令来查看：

root@controller:~# free -m

total used free shared buffers cached

Mem: 4867 3586 1280 0 193 557

-/+ buffers/cache: 2834 2032

Swap: 2022 0 2022

写 Page cache：

（1）当数据被写时，通常情况下，它首先会被写入 page cache，被当作一个  dirty page 来管理。’dirty‘ 的意思是，数据还保存在 page cache 中，还需要被写入底下的持久存储。dirty pages 中的内容会被系统周期性地、以及使用诸如 sync 或者 fsync 的系统调用来写入持久存储。

root@controller:~# cat /proc/meminfo | grep Dirty

Dirty: 148 kB

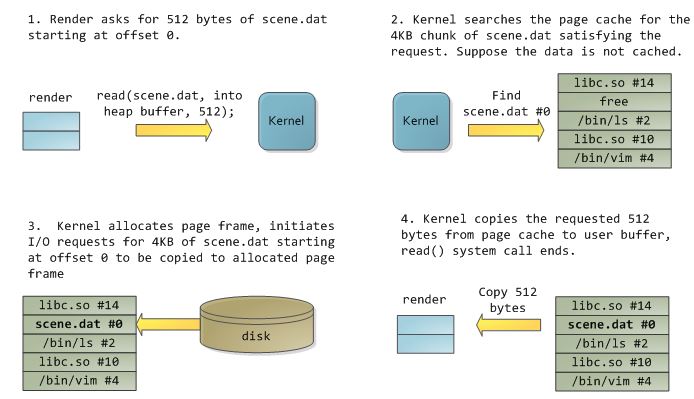
root@controller:~# sync

root@controller:~# cat /proc/meminfo | grep Dirty

Dirty: 0 kB

sync writes any data buffered in memory out to disk. This can include (but is not limited to) modified superblocks, modified inodes, and delayed reads and writes.

（2）当数据被从持久性存储读出时，它也会被写入 page cache。因此，当连续两次读时，第二次会比第一次快，因为第一次读后把数据写入了 page cache，第二次就直接从这里读了。完整过程如下：



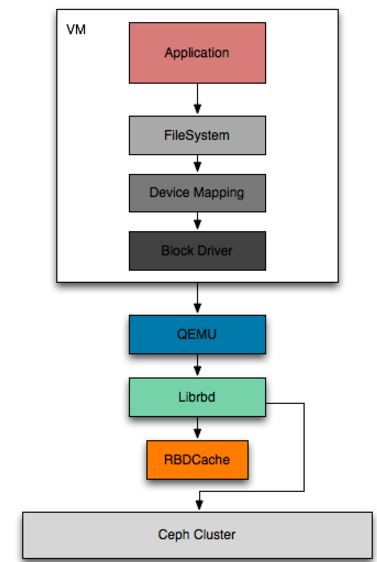
需要注意的是，需要结合应用的需要，来决定是否在写数据时一并写入 page cache。

**2. QEMU+RBDCache**

  QEMU 能够支持将主机的一个块设备映射给客户机，但是从 0.15 版本开始，就不再需要先将一个 Ceph volume 映射到主机再给客户机了。现在，QEMU 可以直接通过 librbd 来象 virtual block device 一样访问 Ceph image。这既提高了性能，也使得可以使用 RBDCache 了。

  RBDCache 是 Ceph 的块存储接口实现库 Librbd 用来在客户端侧缓存数据的目的，它主要提供了读数据缓存，写数据汇聚写回的目的，用来提高顺序读写的性能。需要说明的是，Ceph 既支持以内核模块的方式来实现对 Linux 动态增加块设备，也支持以 QEMU Block Driver 的形式给使用 QEMU 虚拟机增加虚拟块设备，而且两者使用不同的库，前者是内核模块的形式，后者是普通的用户态库，本文讨论的 RBDCache 针对后者，前者使用内核的 Page Cache 达到目的。

**2.1 librbd I/O 协议栈**



  从这个栈可以看出来，RBDCache 类似于磁盘的 write cache。它应该有三个功能：

* 写缓存：开启时，librdb 将数据写入 RBDCache，然后在被 flush 到 Ceph 集群，其效果就是多个写操作被合并，但是有一定的时间延迟。
* 读缓存：数据会在缓存中被保留一段时间，这期间的 librbd 读数据的话，会直接从缓存中读取，提高读效率。
* 合并写操作：对同一个 OSD 上的多个写操作，应该会合并为一个大的写操作，提高写入效率。 ”Due to several objects map to the same physical disks, the original logical sequential IO streams mix together (green, orange, blue and read blocks).  [来源](https://software.intel.com/en-us/blogs/2013/11/20/measure-ceph-rbd-performance-in-a-quantitative-way-part-ii)“

  因此，需要注意的是，理论上，RBDCache 对顺序写的效率提升应该非常有帮助，而对随机写的效率提升应该没那么大，其原因应该是后者合并写操作的效率没前者高（也就是能够合并的写操作的百分比比较少）。具体效果待测试。

  在使用 QEMU 实现的 VM 来使用 RBD 块设备，那么 Linux Kernel 中的块设备驱动是 virtio\_blk，它会对块设备各种请求封装成一个消息通过 virtio 框架提供的队列发送到 QEMU 的 IO 线程，QEMU 收到请求后会转给相应的 QEMU Block Driver 来完成请求。当 QEMU Block Driver 是 RBD 时，缓存就会交给 Librbd 自身去维护，也就是一直所说的 RBDCache；用户在使用本地文件或者 Host 提供的 LVM 分区时，跟 RBDCache 同样性质的缓存包括了 Guest Cache 和 Host Page Cache，见本文第一部分的描述。

**2.2 RBDCache 的原理**

**2.2.1 RBDCache 的配置**

  在 ceph.conf 中，设置 rbd cache = true 即可以启用 RBDCache。它有以下几个主要的配置参数：

|  |  |  |
| --- | --- | --- |
| 配置项 | 含义 | 默认值 |
| rbd cache | 是否启用 RBDCache | 0.87 版本开始：true，启用  0.87 版本之前：false，禁用 |
| rbd\_cache\_size | Librbd 能使用的最大缓存大小 | 32 MiB |
| rbd\_cache\_max\_dirty | 缓存中允许脏数据的最大值，用来控制回写大小，不能超过 rbd\_cache\_size。超过的话，应用的写入应该会被阻塞，  这时候IOPS就会下降 | 24 MiB |
| rbd\_cache\_target\_dirty | 开始执行回写过程的脏数据大小，不能超过 rbd\_cache\_max\_dirty | 16MiB |
| rbd\_cache\_max\_dirty\_age | 缓存中单个脏数据最大的存在时间，避免可能的脏数据因为迟迟未达到开始回写的要求而长时间存在 | 1 秒 |

  可见，默认情况下：

* 在主机操作系统内内存内会分配 32MiB 的空间用于 RBD 做缓存使用
* 允许最大的脏数据大小为 24MiB，超过的话，可能会阻止继续写入（需要确认）
* 在脏数据总共有 16MiB 时，开始回写过程，将数据写入Ceph集群
* 在单个脏数据（目前在 Librbd 用户态库中主要以 Object Buffer Extent 为基本单位进行缓存，这里的粒度应该是 Object Buffer Extent）存在超过 1 秒时，对它启用回写

  也能看出，RBDCache 从空间和时间来方面，在效率和数据有效性之间做平衡。

几个重要的注意事项：

（1）QEMU 和 ceph 配置项的相互覆盖问题

<http://ceph.com/docs/master/rbd/qemu-rbd/#qemu-cache-options>

* 在没有在 Ceph 配置文件中显式配置 RBD Cache 的参数（尽管Ceph 支持配置项的默认值，但是，看起来，是否在Ceph配置文件中写还是不写，会有不同的效果。。真绕啊。。）时，QEMU 的 cache 配置会覆盖 Ceph 的默认配置。
  + qemu driver 'writeback' 相当于 rbd\_cache = true
  + qemu driver ‘writethrough’ 相当于 ‘rbd\_cache = true,rbd\_cache\_max\_dirty = 0’
  + qemu driver ‘none’ 相当于 rbd\_cache = false
  + 一个典型场景是，在 nova.conf 中配置了 ”cache=writeback”，而没有在客户端节点上配置 Ceph 配置文件，这时候将直接打开 RBDCache 并使用 writeback 模式，而不是先 writethrough 后 writeback。
* 在在 Ceph 配置文件中显式配置了缓存模式的时候，Ceph 的 cache 配置会覆盖 QEMU 的 cache 配置。
* 如果在 QEMU 的命令行中使用了 cache 配置，则它会覆盖 Ceph 配置文件中的配置。

优先级：QEMU 命令行中的配置 > Ceph 文件中的显式配置 > QEMU 配置 > Ceph 默认配置

（2）在启用 RBDCache 时，必须在 QEMU 中配置 ”cache=writeback”，否则可能会导致数据丢失。在使用文件系统的情况下，这可能会导致文件系统损坏。

Important

If you set rbd\_cache=true, you must set cache=writeback or risk data loss. Without cache=writeback, QEMU will not send flush requests to librbd. If QEMU exits uncleanly in this configuration, filesystems on top of rbd can be corrupted.

<http://ceph.com/docs/master/rbd/qemu-rbd/#running-qemu-with-rbd>

（3）使用 raw 格式的 Ceph 卷设备 “ <driver name='qemu' type='raw' cache='writeback'/>“

<http://ceph.com/docs/master/rbd/qemu-rbd/#creating-images-with-qemu>

理论上，你可以使用其他 QEMU 支持的格式比如 qcow2 或者 vmdk，但是它们会带来 overhead  
The raw data format is really the only sensible format option to use with RBD. Technically, you could use other QEMU-supported formats (such as qcow2 or vmdk), but doing so would add additional overhead, and would also render the volume unsafe for virtual machine live migration when caching (see below) is enabled.

（4）在新版本的 Ceph 中（将来的版本，尚不知版本号），Ceph 配置项 rbd cache 将会被删除，RBDCache 是否开启将由 QEMU 配置项决定。

也就是说，如果 QEMU 中设置 cache 为 ‘none’ 的话， RBDCache 将不会被使用；设置为 ‘writeback’ 的话，RBDCache 将会被启用。参考链接：[ceph : [client] rbd cache = true override qemu cache=none|writeback](https://forum.proxmox.com/threads/ceph-client-rbd-cache-true-override-qemu-cache-none-writeback.22459/)。

（5）对 Nova 来说，不设置 disk\_cachemode 值的话，默认的 driver 的 cache 模式是 ‘none’。但是，在不支持 ‘none’ 模式的存储系统上，会改为使用 ‘writethrough’ 模式。（[来源](https://review.openstack.org/#/c/5606/5/nova/virt/libvirt/connection.py)）

[复制代码](javascript:void(0);)

def disk\_cachemode(self):

if self.\_disk\_cachemode is None:

# We prefer 'none' for consistent performance, host crash

# safety & migration correctness by avoiding host page cache.

# Some filesystems (eg GlusterFS via FUSE) don't support

# O\_DIRECT though. For those we fallback to 'writethrough'

# which gives host crash safety, and is safe for migration

# provided the filesystem is cache coherant (cluster filesystems

# typically are, but things like NFS are not).

self.\_disk\_cachemode = "none"

if not self.\_supports\_direct\_io(FLAGS.instances\_path):

self.\_disk\_cachemode = "writethrough"

return self.\_disk\_cachemode

[复制代码](javascript:void(0);)

**2.2.2 缓存中的数据被 flush 到 Ceph cluster**

  有两种类型的 flush：

* RBD 主动的，在 RBDCache 规定的空间或者数据保存时间达到阈值之后，会触发回写
* RBD 被动的，librbd 的 flush 接口被调用，全部缓存中的数据也会被回写。又可以细分为两种类型：
  + QEMU 在合适的时候会自动发出 flush：QEMU 作为最终使用 Librbd 中 RBDCache 的用户，它在 VM 关闭、QEMU 支持的热迁移操作或者 RBD 块设备卸载时都会调用 QEMU Block Driver 的 Flush 接口，确保数据不会被丢失。因此，此时，需要用户在使用了开启 RBDCache 的 RBD 块设备 VM 时需要给 QEMU 传入 “cache=writeback” 确保 QEMU 知晓有缓存的存在，不然 QEMU 会认为后端并没有缓存而选择将 Flush Request 忽略。
  + 应用发出 flush，比如 fio，可以设置 fdatasync 为一个大于零的整数，从而在若干次写操作后执行fdatasync。（ fdatasync=int Like fsync, but uses fdatasync(2) instead to only sync the data parts of the file. Default: 0“

  关于第二种 flush，这里的一个问题是，什么时候会有这种主动 flush 指定发出。有文章说，”QEMU 作为最终使用 Librbd 中 RBDCache 的用户，它在 VM 关闭、QEMU 支持的热迁移操作或者 RBD 块设备卸载时都会调用 QEMU Block Driver 的 Flush 接口“。同时，一些对数据的安全性敏感的应用也可以通过操作系统在需要的时候发出 flush 指定，比如一些数据库系统。你可以使用 fio 工具的 fdatasync 参数在指定的写入操作后发出 fdatasync 指令。具体效果还待测试。

  librados 的 flush API：

[复制代码](javascript:void(0);)

CEPH\_RADOS\_API int rados\_aio\_flush(rados\_ioctx\_t io)

Block until all pending writes in an io context are safe

This is not equivalent to calling rados\_aio\_wait\_for\_safe() on all write completions, since this waits for the associated callbacks to complete as well.

Parameters

io -

the context to flush

[复制代码](javascript:void(0);)

  下面是一个 Linux 系统上文件操作的伪代码（[来源](http://www.orczhou.com/index.php/2009/08/innodb_flush_method-file-io/)）。可见，该程序知道只有在 fdatasync 执行成功后，数据才算写入成功。

[复制代码](javascript:void(0);)

#include "stdlib.h" /\* for exit \*/

#include "unistd.h" /\* for write fdatasync\*/

#include "fcntl.h" /\* for open \*/

int main(void){

int fd;

if((fd=open("/home/zzx/test.file",O\_WRONLY|O\_APPEND|O\_DSYNC))<0){

exit(1);

}

char buff[]="abcdef";

if(write(fd,buff,6)!= 6){

exit(2);

}

if(fdatasync(fd)==-1){

exit(3);

}

exit(0);

}

[复制代码](javascript:void(0);)

**2.2.3 RBDCache 中数据的易失性和 librbd rbd\_cache\_writethrough\_until\_flush 配置项**

  因为 RBDCache 是利用内存来缓存数据，因此数据也是易失性的。那么，最安全的是，设置 rbd\_cache\_max\_dirty = 0，就是不缓存数据，相当于 writethrough 的效果。很明显，这没有实现 RBDCache 的目的。

  另外，Ceph 还提供 rbd\_cache\_writethrough\_until\_flush 选项，它使得 RBDCache 在收到第一个 flush 指令之前，使用 writethrough 模式，透传数据，避免数据丢失；在收到第一个 flush 指令后，开始 writeback 模式，通过 KVM barrier 功能来保证数据的可靠性。

  该选项的含义：

This option enables the cache for reads but does writethrough until we observe a FLUSH command come through, which implies that the guest OS is issuing barriers. This doesn't guarantee they are doing it properly, of course, but it means they are at least trying. Once we see a flush, we infer that writeback is safe.

  该选项的默认值到底是 true 还是 false 比较坑爹：

* ceph/librbd 在 0.80 版本中添加该选项，默认值是 false （[代码](https://github.com/ceph/ceph/blob/v0.80.10/src/common/config_opts.h)）
* ceph/librbd 在 0.87 版本中将默认值修改为 true （[代码](https://github.com/ceph/ceph/blob/v0.87/src/common/config_opts.h)）

  因此，你在使用不同版本的 librbd 情况下使用默认配置时，其 IOPS 性能是有很大的区别的：

* 0.80 版本中，一直是 writeback，IOPS 会从头就很好；
* 0.87 版本中，开始是 writethrough，在收到第一个操作系统发来的 flush 后，转为 writeback，因此，IOPS 是先差后好。

  在实现上，

（1）收到第一个flush 之前，相当于 rbd\_cache\_max\_dirty 被设置为0 了：

uint64\_t init\_max\_dirty = cct->\_conf->rbd\_cache\_max\_dirty;

if (cct->\_conf->rbd\_cache\_writethrough\_until\_flush)

init\_max\_dirty = 0;

（2）收到第一个 flush 之后，就转为 writeback 了

if (object\_cacher && cct->\_conf->rbd\_cache\_writethrough\_until\_flush) {

md\_lock.get\_read();

bool flushed\_before = flush\_encountered;

md\_lock.put\_read();

uint64\_t max\_dirty = cct->\_conf->rbd\_cache\_max\_dirty;

**2.3 小结**

  各种配置下的Ceph RBD 缓存效果：

|  |  |  |
| --- | --- | --- |
| 配置 | rbd\_cache\_writethrough\_until\_flush 的值 | 缓存效果 |
| rbd cache = false | N/A | 没有读写缓存，等同于 directsync |
| rbd cache = true  rbd\_cache\_max\_dirty = 0 | N/A | 只有读缓存，没有写缓存，等同于 writethrough |
| rbd cache = true  rbd\_cache\_max\_dirty > 0  “cache=writeback” | True | 在收到 QEMU 发出的第一个 flush 前，  使用 writethrough 模式；收到后，使用 writeback 模式 |
| rbd cache = true  rbd\_cache\_max\_dirty > 0  “cache=writeback” | False | 一直使用 writeback 模式，QEMU 会在特定时候发出 flush，可能会导致数据丢失 |
| rbd cache = true  rbd\_cache\_max\_dirty > 0  “cache=none” | True | 一直使用 writethrough 模式，没有写缓存，只有读缓存 |
| rbd cache = true  rbd\_cache\_max\_dirty > 0  “cache=writeback” | False | 一直使用 writeback 模式，QEMU 会发出 flush 使缓存数据写入Ceph 集群 |

**3. Linux 系统 I/O <补记于 2016/05/28>**

    备注：主要内容引用自 2016/05/27 发表于 Linux内核之旅 微信公众号的 [系统和进程信息与文件IO缓冲](http://mp.weixin.qq.com/s?__biz=MzI3NzA5MzUxNA==&mid=2664602044&idx=1&sn=f6afa80a6a601eda53fc70cd9f8fc9ab&scene=4#wechat_redirect) 一文。

    忽略文件打开的过程，通常我们会说“写文件”有两个阶段，一个是调用 write 我们称为写数据阶段（其实是受open的参数影响），调用 fsync（或者fdatasync）我们称为flush阶段。Linux上的块设备的操作可以分为两类：出于速度和效率的考虑。系统 I/O 调用(即内核)和标准 C语言库 I/O 函数，在操作磁盘文件时会对数据进行缓冲。

**3.1 stdio 缓冲的类型和设置**

使用 C 标准库中的 fopen/fread/fwrite 系列的函数，我们可以称其为 buffered I/O。具体的I/O path如下：

Application <-> Library Buffer <-> Operation System Cache <-> File System/Volume Manager <-> Device

其中，library buffer 是标准库提供的用户空间的 buffer，可以通过 setvbuf 改变其大小。

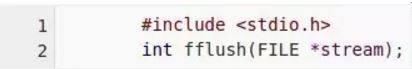
设置方法如下：

（1）通过 setvbuf 函数，结合不同的mode，可以设置缓冲的类型，和大小：



* 不缓冲：不对 I/O 进行缓冲，等同于write或read忽略其buf和size参数，分别指定为NULL和0。stderr默认属于这个类型。
* 行缓冲：对于输出流，在输出一个换行符前(除非缓冲区满了)将缓冲数据，对于输入流，每次读取一行数据。
* 全缓冲：对于输出流，会一直缓冲数据知道缓冲区满为止。

（2）我们也可以通过fflush来刷新缓冲区。



**3.2 内核缓冲**

使用 Linux 的系统调用的 open/read/write 系列的函数，我们可以称其为 non-buffered I/O。其I/O 路径为：

Application<-> Operation System Cache <-> File System/Volume Manager <->Device

OS Cache 即内核缓冲的缓冲区，它是在内核态的，没办法通过像 stdio 库的 setvbuf 那样给它指定一个用户态的缓冲区，只能控制缓冲区的刷新策略等。

* fsync 用于将描述符 fd 相关的所有元数据都刷新到磁盘上，相当于强制使文件处于同步I/O文件完整性。
* fdatasync 的作用类似于 fsync，只是强制文件处于数据完整性，确保数据已经传递到磁盘上。
* sync 仅在所有数据(数据块，元数据)已传递到磁盘上时才返回。

这些函数都只能刷新一次缓冲，此后每次发生的I/O操作都需要再次调用上面的三个系统调用来刷新缓冲，为此可以通过设置描述符的属性来保证每次IO的刷新缓冲。

* 打开文件的时候设置 O\_SYNC，相当于每次发生 IO 操作后都调用 fsync 和 fdatasync 系统调用。[写操作只有在数据和元数据比如修改时间等被写入磁盘后才返回]
* 使用 O\_DSYNC 标志则要求写操作是同步 I/O 数据完整性的也就是相当于调用 fdatasync。[写操作必须等到数据被写入磁盘后才返回，此时元数据可以没有被写入磁盘，比如文件的 inode 信息尚未被写入时]
* O\_RSYNC 标志需要结合 O\_DSYNC 和 O\_SYNC 标志一起使用。

将这些标志的写操作作用结合到读操作中，也就说在读操作之前，会先按照O\_DSYNC或O\_SYNC对写操作的要求，完成所有待处理的写操作后才开始读。

更详细信息，可以参考 <http://man7.org/linux/man-pages/man2/open.2.html>。

**3.3 Direct I/O（O\_DIRECT 标志）**

   Linux 允许应用程序在执行磁盘 I/O 时绕过缓冲区高速缓存（Host OS Page Cache），从用户空间直接将数据传递到文件或磁盘设备上（其实是 Disk cache 中），这我们称之为直接 I/O，或者裸 I/O。直接 I/O 只适用于特定 I/O 需求的应用，例如:数据库系统，其高速缓存和I/O优化机制自成一体，无需内核消耗CPU时间和内存去完成相同任务。通过在打开文件的时候指定 O\_DIRECT 标志设置文件读写为直接IO的方式。使用直接 IO 存在一个问题，就是I/O得对齐限制，因为直接 I/O 涉及对磁盘的直接访问，所以在执行I/O时，必须遵守一些限制：

* 用于传递数据的缓冲区，其内存边界必须对齐为块大小的整数倍。
* 数据传输的开始点，亦即文件和设备的偏移量，必须是块大小的整数倍。
* 待传递数据的 buffer 必须是块大小的整数倍。

**3.4 总结**

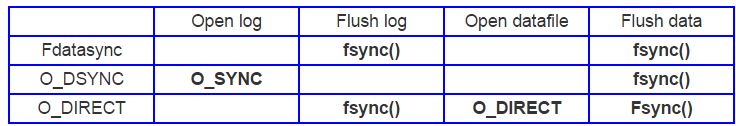
**3.4.1 I/O 栈**

**3.4.2 MySQL 中的 I/O 模式**

对于 MySQL 来说，该数据库系统是基于文件系统的，其性能和设备读写的机制有密切的关系。和数据库性能密切相关的文件I/O操作的三个操作：

* open 打开文件：比如，open("test.file",O\_WRONLY|O\_APPDENT|O\_SYNC))
* write 写文件：比如，write(fd,buf,6)
* fdatasync flush操作（将文件缓存刷到磁盘上）：比如，fdatasync(fd) == -1，表示 write操作后，我们调用了 fdatasync 来确保文件数据 flush 到了 disk上。fdatasync 返回成功后，那么可以认为数据已经写到了磁盘上。像这样的flush的函数还有fsync、sync。

MySQL 的 innodb\_flush\_log\_at\_trx\_commit 参数确定日志文件何时 write 和 flush。innodb\_flush\_method 则确定日志及数据文件如何 write、flush。在Linux下，innodb\_flush\_method可以取如下值：fdatasync, O\_DSYNC, O\_DIRECT，那这三个值分别是如何影响文件写入的？



* fdatasync 被认为是安全的，因为在 MySQL 总会调用 fsync 来 flush 数据。
* 使用 O\_DSYNC 是有些风险的，有些 OS 会忽略该参数 O\_SYNC。
* 我们看到 O\_DIRECT 和 fdatasync和 很类似，但是它会使用 O\_DIRECT 来打开数据文件。有数据表明，如果是大量随机写入操作，O\_DIRECT 会提升效率。但是顺序写入和读取效率都会降低。所以使用O\_DIRECT需要谨慎。

简单的，InnoDB 在每次提交事务时，为了保证数据已经持久化到磁盘（Durable），需要调用一次 fsync（或者是 fdatasync、或者使用 O\_DIRECT 选项等）来告知文件系统将可能在缓存中的数据刷新到磁盘（更多关于[fsync](http://www.orczhou.com/index.php/2009/08/innodb_flush_method-file-io/)）。而 fsync 操作本身是非常“昂贵”的（关于“昂贵”：消耗较多的IO资源，响应较慢）：传统硬盘（10K转/分钟）大约每秒支撑150个fsync操作，SSD（Intel X25-M）大约每秒支撑1200个fsync操作。所以，如果每次事务提交都单独做fsync操作，那么这里将是系统TPS的一个瓶颈。所以就有了Group Commit的概念。

当多个事务并发时，我们让多个都在等待 fsync 的事务一起合并为仅调用一次 fsync 操作。这样的一个简单的优化将大大提高系统的吞吐量，Yoshinori Matsunobu的[实验](http://yoshinorimatsunobu.blogspot.com/2009/08/great-performance-effect-of-fixing.html)表明，这将带来五到六倍的性能提升。

**4. I/O Cache 与 Linux write barrier**

**4.1 I/O Cache**

一个典型的存储 I/O 路径：

一些组件有自己的cache：

其中：

* page cache 是 VFS cache 的一部分，buffer cache 是内存中，两者都是易失性的。但是，buffer cache 的容量往往很大。
* storage controller write cache 存在于绝大多数中高端存储控制器中。
* Write cache on physical media 存在于磁盘中，绝大多数也是易失性的。一些SSD含有备份电容。现代磁盘往往带有 16 - 64 MB cache。

**4.2 Linux write barrier**

<作者注：作者对本文中的一些概念还有一些疑惑或者不解，因此，本文内容会持续更新>

参考资料：

* [解析Ceph: RBDCache 背后的世界](http://www.wzxue.com/rbdcache/)
* <http://docs.ceph.com/docs/hammer/rbd/rbd-config-ref/>
* [KVM storage performance and cache settings on Red Hat Enterprise Linux 6.2](http://www.ilsistemista.net/index.php/virtualization/23-kvm-storage-performance-and-cache-settings-on-red-hat-enterprise-linux-62.html?start=2)
* <http://linux.die.net/man/1/fio>
* <http://xfs.org/index.php/XFS_FAQ#Write_barrier_support.>
* <https://libvirt.org/formatdomain.html>
* <https://www.quora.com/What-is-the-major-difference-between-the-buffer-cache-and-the-page-cache>
* <https://bugs.launchpad.net/charms/+source/nova-compute/+bug/1412856>
* <http://duartes.org/gustavo/blog/post/page-cache-the-affair-between-memory-and-files/>
* <http://www.orczhou.com/index.php/2010/08/time-to-group-commit-1/>
* <http://www.orczhou.com/index.php/2009/08/innodb_flush_method-file-io/>
* <http://www.thesubodh.com/2013/07/what-are-exactly-odirect-osync-flags.html>
* <https://monolight.cc/2011/06/barriers-caches-filesystems/>

分类: [Ceph](http://www.cnblogs.com/sammyliu/category/738054.html),[原理](http://www.cnblogs.com/sammyliu/category/750167.html)