# 客户端部分

1. \client\hypertable

CephFS与Hypertable适配实现

1. \client\barrier.h

Barrier的目的是为了对内存进行屏蔽处理，使得块数据真正提交到非易失性介质后，才允许后面的IO写入，目前client端已经去掉

Client的元数据缓存策略使用Midpoint Insertion Strategy来实现inode的热点数据进行迁移

1. 客户端标准接口

|  |  |  |
| --- | --- | --- |
| 函数 | 接口功能 | 备注 |
| opendir | 获取path子目录下的所有文件和目录列表信息。 | 如果path是一个文件，则返回NULL |
| closedir | 关闭打开的目录 |  |
| readdir | 返回指定目录下的目录流的下一个进入点，即循环读取Dentry的信息。 | 成功返回下一个目录进入点 |
| chdir | 改变当前工作目录，目的目录在参数提供 |  |
| readdirplus\_r | 除了具有readdir的特性外，还有获取单个Dentry的stats状态信息 | 会提示效率吗？NFS有类似的接口，主要是为了适配NFS的接口? |
| getdents | 通过目录名称获取目录下的所有Dentry信息 |  |
| getdnames | 通过目录名称获取目录下的所有name信息 |  |
| rewinddir | 将目录流的位置重新置为开始位置 | 无返回值 |
| telldir | 获取目录流的当前位置 | 返回当前位置地址 |
| seekdir | 通过参数offset，来跳转到offset指定的目录流位置 |  |
| mkdir | 创建目录 |  |
| mkdirs | 一次创建多级目录 | 比如mkdir /a/b/c/e |
| rmdir | 删除目录 |  |
| link | 为一个文件创建Dentry信息 | 注意：目录不能创建多余1个Dentry |
| symlink | 创建符号链接 | 即软链接 |
| readlink | 读取符号链接 |  |
| unlink | 移除link，包含符号链接 |  |
| rename | 修改目录或者文件的名称 |  |
| chmod | 修改目录或者文件的权限，即mode bits信息 |  |
| fchmod | 修改已经打开的文件的权限 |  |
| chown | 修改目录或者文件的所属用户或者用户组 |  |
| fchown | 修改已经打开的文件的权限 |  |
| lchown | 与chown的区别是，这个修改不会对软链接生效 |  |
| utime | 修改文件和目录最后访问和修改的时间 |  |
| flock | 文件锁，锁定指定的文件 | 支持共享锁(LOCK\_SH)、互斥锁(LOCK\_EX)、解除文件锁(LOCK\_UN)、无法建立锁状态(LOCK\_NB)  只能对整个文件进行锁定，无法对文件某段进行锁定 |
| truncate | 增大或者减小指定文件长度，通过参数size来确定。 | 增加的部分使用0来填充，减少，则会导致文件数据丢失 |
| mknod | 创建linux中字符设备文件和块设备文件，比如fifo文件 |  |
| open | 打开或者创建一个文件 | 支持创建和打开一个已经存在的文件   1. 参数flags：用于指定文件创建和打开模式   **必选的3个参数：（flag需要与O\_ACCMODE屏蔽字段来获取以下3个字段信息）**   * O\_RDONLY只读模式 * O\_WRONLY只写模式 * O\_RDWR读写模式   **可选参数：**   * O\_APPEND每次写操作都写入文件的末尾 * O\_CREAT如果指定文件不存在，则创建这个文 * 件 * O\_EXCL如果要创建的文件已存在，则返回-1，并且修改errno的值 * O\_TRUNC如果文件存在，并且以只写/读写方式打开，则清空文件全部内容(即将其长度截短为0) * O\_NOCTTY如果路径名指向终端设备，不要把这个设备用作控制终端。 * O\_NONBLOCK如果路径名指向FIFO/块文件/字符文件，则把文件的打开和后继I/O设置为非阻塞模式 * O\_DSYNC等待物理I/O结束后再write。在不影响读取新写入的数据的前提下，不等待文件属性更新。 * O\_RSYNCread等待所有写入同一区域的写操作完成后再进行 * O\_SYNC等待物理I/O结束后再write，包括更新文件属性的I/O * O\_DIRECT 该描述符提供对直接 I/O 的支持 * O\_NOFOLLOW 若设置该描述符，则不解析路径名尾部的符号链接 * O\_DIRECTORY 该描述符表明所打开的文件必须是目录，否则打开操作失败 * FASYNC 若设置该描述符，则 I/O 事件通知是通过信号发出的  1. Mode只在创建新文件(O\_CREAT)时设置，具体定义见《Inode st\_mode》章节介绍 |
| close | 关闭一个文件 |  |
| lseek | 将打开的目录指向的指针移动到给定的位置 |  |
| read | 对取给定文件中的内容 |  |
| preadv | 在一次执行过程中可以原子地作用于多个缓存区，一般这些缓存区是非连续的，且是基于文件的fd进行 | 其中iov的结构体为：  struct iovec {  void \*iov\_base;  size\_t iov\_len;  }  iov\_base为缓存区首地址，iov\_len为缓冲区长度，另外iovcnt指定了有多少个这样的缓存区 |
| write | 写入数据到给定文件 |  |
| pwritev | 和preadv类似 |  |
| fsync | 将指定的文件描述符进行sync操作，并等待修改过的数据写入到磁盘结束后，再返回 |  |
| fallocate | 为文件预分配或者释放指定大小范围的字节值的磁盘空间，即我们平时所说的文件打洞 | 主要应用于大空间信息分配 |
| getxattr | 获取文件的扩展属性，包含UID\GID\TIME等信息 |  |
| setxattr | 设置文件的扩展属性 |  |

1. 设置quota流程

Cephfs的Quota是通过操作系统自带的操作文件系统的扩张属性命令来进行设置，命令为setfattrgetfattr，示例如下：

*setfattr -n ceph.quota.max\_files -v 5 /share-fs/export/test3/*

*getfattr -n ceph.quota.max\_files /share-fs/export/test3/*

在Ceph中实现设置配额的流程如下：

handle\_client\_setxattr------>handle\_set\_vxattr------>parse\_quota\_vxattr

服务端的quota定义在mdstypes.h

配额的结构体为：struct quota\_info\_t

客户端的结构体在：struct Inode

客户端扩展属性定义：map<string,bufferptr> xattrst

服务端的结构体在：struct inode\_t

服务端扩展属性定义：compact\_map<snapid\_t, old\_inode\_t> old\_inodes

1. Inode st\_mode

Inode设置mask掩码来做权限管理，在open()函数创建文件时，通过mode参数来设置，具体含义如下：

|  |  |  |
| --- | --- | --- |
| **St\_mode** | **Mask** | **Function** |
| S\_IFMT | 0170000 | 文件类型 |
| S\_IFSOCK | 0140000 | socket |
| S\_IFLNK | 0120000 | 符号链接(symbolic link) |
| S\_IFREG | 0100000 | 一般文件 |
| S\_IFBLK | 0060000 | 区块装置(block device) |
| S\_IFDIR | 0040000 | 目录 |
| S\_IFCHR | 0020000 | 字符装置(character device) |
| S\_IFIFO | 0010000 | 先进先出(fifo) |
| S\_ISUID | 04000 | set-user-ID bit |
| S\_ISGID | 02000 | set-group-ID bit |
| S\_ISVTX | 01000 | sticky bit(是linux目录下除rwx权限意外的第4种权限：t，查看/tmp发现它的other组的最后一位即为t，表示即使使用用户对tmp都有rw权限，但不同用户创建的文件，只能由它自己本身或者root才能删除) |
| S\_IRWXU | 00700 | owner has read, write, and execute permission |
| S\_IRUSR | 00400 | owner has read permission |
| S\_IWUSR | 00200 | owner has write permission |
| S\_IXUSR | 00100 | owner has execute permission |
| S\_IRWXG | 00070 | group has read, write, and execute permission |
| S\_IRGRP | 00040 | group has read permission |
| S\_IWGRP | 00020 | group has write permission |
| S\_IXGRP | 00010 | group has execute permission |
| S\_IRWXO | 00007 | others (not in group) have read, write, andexecute permission |
| S\_IROTH | 00004 | others have read permission |
| S\_IWOTH | 00002 | others have write permission |
| S\_IXOTH | 00001 | others have execute permission |
|  |  |  |
| S\_ISLNK |  | 是否是一个连接 |
| S\_ISREG |  | 是否是一个常规文件 |
| S\_ISDIR |  | 是否是一个目录 |
| S\_ISCHR |  | 是否是一个字符设备 |
| S\_ISBLK |  | 是否是一个块设备 |
| S\_ISFIFO |  | 是否 是一个FIFO文件 |
| S\_ISSOCK |  | 是否是一个SOCKET文件 |

1. Lseek和O\_APPEND

文件在操作系统中的表示，就是通过数据流的方式存在于内存中，当对一个文件操作时，则需要通过文件指针来指定操作文件的位置，文件指针不能直接方法，需要通过lseek函数来实现访问。write和read函数是自带移动文件指针的功能，因此可以看到write可以连续写，而不需要显示的移动指针。但是如果需要显示的指定文件指针位置，则需要使用lseek函数来指定

Linux中并没有计算文件长度的函数，我们可以通过lseek函数来实现，另外也可以基于lseek来构建空洞文件，来实现大文件在多线程中并发写的实现；在一个进程中两次打开同一文件，如果想要这2次打开的操作文件是连续的，则需要在open函数中增加O\_APPEND标志，O\_APPEND标记可以让write和read多做一件时，即可以同时移动相关多个fd位置，实现同步原子操作。

与fseek区别在于lseek是系统函数，而lseek是库函数。

1. CAP(Capabilities)

Capabilities在CephFS中简称为CAP功能，客户端定义在inode.h中struct Cap，服务端定义在Capability中class Capability，其主要作用是限制客户端拥有不同的操作权限，比如读权限、写权限和读写权限等，相比NFS/CIFS控制粒度更小

#define CEPH\_CAP\_PIN 1

|  |  |  |
| --- | --- | --- |
| /\* generic cap bits \*/  #define CEPH\_CAP\_GSHARED 1 /\* client can reads (s) \*/  #define CEPH\_CAP\_GEXCL 2 /\* client can read and update (x) \*/  #define CEPH\_CAP\_GCACHE 4 /\* (file) client can cache reads (c) \*/  #define CEPH\_CAP\_GRD 8 /\* (file) client can read (r) \*/  #define CEPH\_CAP\_GWR 16 /\* (file) client can write (w) \*/  #define CEPH\_CAP\_GBUFFER 32 /\* (file) client can buffer writes (b) \*/  #define CEPH\_CAP\_GWREXTEND 64 /\* (file) client can extend EOF (a) \*/  #define CEPH\_CAP\_GLAZYIO 128 /\* (file) client can perform lazy io (l) \*/ | | |
| /\* per-lock shift \*/  #define CEPH\_CAP\_SAUTH 2 /\* A \*/  #define CEPH\_CAP\_SLINK 4 /\* L \*/  #define CEPH\_CAP\_SXATTR 6 /\* X \*/  #define CEPH\_CAP\_SFILE 8 /\* F \*/  在代码中实现：  inline string ccap\_string(int cap)  {  string s;  if (cap & CEPH\_CAP\_PIN) s += "p";  int a = (cap >> CEPH\_CAP\_SAUTH) & 3;  if (a) s += 'A' + gcap\_string(a);  a = (cap >> CEPH\_CAP\_SLINK) & 3;  if (a) s += 'L' + gcap\_string(a);  a = (cap >> CEPH\_CAP\_SXATTR) & 3;  if (a) s += 'X' + gcap\_string(a);  a = cap >> CEPH\_CAP\_SFILE;  if (a) s += 'F' + gcap\_string(a);  if (s.length() == 0)  s = "-";  return s;  }  inline string gcap\_string(int cap)  {  string s;  if (cap & CEPH\_CAP\_GSHARED) s += "s";  if (cap & CEPH\_CAP\_GEXCL) s += "x";  if (cap & CEPH\_CAP\_GCACHE) s += "c";  if (cap & CEPH\_CAP\_GRD) s += "r";  if (cap & CEPH\_CAP\_GWR) s += "w";  if (cap & CEPH\_CAP\_GBUFFER) s += "b";  if (cap & CEPH\_CAP\_GWREXTEND) s += "a";  if (cap & CEPH\_CAP\_GLAZYIO) s += "l";  return s;  }  其中  虽然这就是授予（和传播）功能的方式，但重要的是它们实际允许客户端执行的操作：  PIN：这只是将inode固定到内存中。这足以允许客户端获取inode编号，以及其他不可变的内容，例如设备inode中的主要或次要编号，或符号链接内容。  AUTH：这授予了获取与身份验证相关的元数据的能力。特别是所有者，组和模式。请注意，执行完全权限检查可能还需要获取ACL，这些ACL存储在xattrs中。  LINK：inode的链接数  XATTR：访问或操纵xattrs的能力。请注意，由于ACL存储在xattrs中，因此在检查权限时有时也需要访问它们。  FILE：这是最重要的一个。这些允许客户端访问和操作文件数据。它还包括与文件数据相关的某些元数据 - 特别是大小，mtime，atime和ctime。  pAsLsXsFrl  示例：   |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | p | A | s | L | s | X | s | F | r | l | | Pin | 权威能力 | client can reads (s) | 链接能力 | client can reads (s) | 文件扩展属性 | client can reads(s) | 文件 | (file) client can read (r) | (file) client can perform lazy io (l) |   下面解释这几种类型的区别   |  |  |  | | --- | --- | --- | | P | pin | the inode exists | | A | auth | authentication metadata: mode, uid, gid | | L | link | the inode’s link count (number of dentries linked to an inode) | | X | xattr | the inode’s xattrs; their presence and values | | F | file | file data, file size, mtime and other | | s | shared | The client has shared access to this state; one of many | | x | exclusive | The client is the only one with access to this state | | r | read | The client can read state | | w | write | The client can write the state | | c | cache | The client can cache the state locally | | b | buffer | The client can buffer changes to the state locally |   示例   |  |  | | --- | --- | | Fs | can cache and read mtime, size locally | | Fx | can write mtime, size locally | | Fr | can read the file data (...synchronously from OSD) | | Fc | can cache file data for local reads | | Fw | can write the file data (synchronously to OSD) | | Fb | can buffer data writes; flush in the background | |  |  |   那么cap是怎么通过MDS生成?  （1）元数据最终通过MDS来决定  （2）每一个元数据通过lock来控制，lock包括（SimpleLock，ScatterLock，  FileLock），比如那些功能可以发布给哪些客户端，复制的MDS获得客户  端没有的某些权限？  （3）客户端发送元数据更新和cap更新请求  （4）一大堆启发式算法会根据这些输入改变锁定状态以尝试和优化?  （5） mds可以通过grant和revoke个人客户的cap（通常基于其它客户活动状态?）  （6）在revoke时，客户端必须停止使用cap，并向MDS发送一条MclientMSG  确认更改的消息到MDS，对应停止使用操作，如果是cache或者共享读，  则丢弃数据和更改状态即可，但如果是buffer wirte操作，则必须先flush脏  操作，这需要一些时间。  （7） MDS会记录所有client固定的信息，因此mds的cache大小要大于所有client  缓存之和，由于cap是合作进行的，客户端如果需要正常工作和访问正确，  则需要能被其它块阻止掉 | | |
| Cap标志 | | 默认值 | 参数说明 | |
| CEPH\_CAP\_FLAG\_AUTH | | (1 << 0) | 消息由权威MDS发起cap要求 | |
| CEPH\_CAP\_FLAG\_RELEASE | | (1 << 1) | 要求客户释放cap | |
| CEPH\_CAP\_AUTH\_SHARED | | CEPH\_CAP\_GSHARED << CEPH\_CAP\_SAUTH  (1 << 2) | 授予客户端具有元数据读权限 | |
| CEPH\_CAP\_AUTH\_EXCL | | CEPH\_CAP\_GEXCL << CEPH\_CAP\_SAUTH  (2 << 2) | 授予客户端具有元数据读和更新权限 | |
| CEPH\_CAP\_LINK\_SHARED | | CEPH\_CAP\_GSHARED << CEPH\_CAP\_SLINK  (1 << 4) | 授予客户端对链接有读的权限 | |
| CEPH\_CAP\_LINK\_EXCL | | CEPH\_CAP\_GEXCL << CEPH\_CAP\_SLINK  (2 << 4) | 授予客户端对链接有读和更新的权限 | |
|  | |  |  | |
|  | |  |  | |
| CEPH\_CAP\_FILE\_EXCL | |  | 授予客户端对文件的读和更新权限 | |
| CEPH\_CAP\_FILE\_CACHE | |  | 授予客户端能在cache中读文件的读和更新权限 | |
| CEPH\_CAP\_FILE\_RD | | CEPH\_CAP\_GRD << CEPH\_CAP\_SFILE  (8 << 8) | 授予客户端对文件的读权限 | |
| CEPH\_CAP\_XATTR\_SHARED | | CEPH\_CAP\_GSHARED << CEPH\_CAP\_SXATTR  (1 << 6) | 授予客户端对文件属性的权限 | |
| CEPH\_CAP\_FILE\_SHARED | | CEPH\_CAP\_GSHARED << CEPH\_CAP\_SFILE  (1 << 8) | 授予客户端对文件内容有读的权限 | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |

8 快照 snap

Client在保持快照snapid时，分为2种情况：

如果是直接调用libcephfs的方式，则将snapid保存到stat::st\_dev中，见函数Client::fill\_stat

如果是使用fuse的方式，则将snapid encode到高16为的ino\_t中，在64位操作系统中低48位仅仅应用于inode id号，当然如果是32位操作系统，则ino\_t只有32位时，需要引入faked机制，具体见https://github.com/ceph/ceph/pull/5849

9 dir\_result\_t结构体信息

10 客户端一致性要求

在有多个客户端混合读写或者多写时，mds会向所有的client发起revoke操作来撤销之前授予客户端的read caching和write buffering功能，强制要求client的IO进行同步，这会导致所有的应用程序将被阻塞，直到数据同步完成

也就是说，每个应用程序读取或写入操作将被阻止，直到OSD确认有效地将更新序列化和与存储每个对象的OSD同步的负担。 当写入跨越对象边界时，实现原子性会更复杂当写入跨越对象边界时，实现原子性会更复杂。 原型目前使用简单的锁定机制来实现正确的序列化，尽管正在考虑实现更接近真实事务的替代方法

11 lazy

很显然客户端的一致性要求，导致在多写或者混合读写时的性能下降，对于HPC（high-performance computer）来说则无法接受，因此CephFS client专门为HPC扩展了POSIX特性：O\_LAZY标记，其放开共享写或读写时的要求，由应用程序自己保证数据的一致性要求（比如对同一个文件多写时，写入到不同的offset位置）。同时lazyio propagate可以指定刷新对象范围，并通过lazyio synchronize来确保刷新的数据能体现在后续读写中。

12 client初始化

（1）client初始化流程

client初始化从mount开始：

a> subscribe\_mdsmap 获取fsmap信息，并记录到monclient中，开启tick定时获

取mdsmap信息，同时判断是否有active的mds状态，如果没有则等待

tick会定时调用renew\_caps函数，实现client与mds端之间的session状态更新

b> 选取挂载点，可以指定，默认使用"/"

c> 调用make\_request向mds端发起挂载点目录GETATTR的操作，获取此inode的所

有信息，mask为：CEPH\_STAT\_CAP\_INODE\_ALL，make\_request会一直等待mds

的恢复后才结束?同步等待???

d> mds获取到信息后，会通过调用reply\_client\_request回复给client

e> client调用handle\_client\_reply来确认回复，并且调用insert\_trace记录信息到客户端

缓存中

13 client缓存机制

client的缓存使用的并非标准的LRU算法，而是使用的LRU改进算法Midpoint Insertion Strategy，即将LRU缓存分为2部分：warm sub-chain和hot sub-chain，并根据client\_cache\_mid来划分，比如：设置client\_cache\_mid为0.75，则warm整个LRU chain的比例为75%，而25%则为hot。Hot的数据会放置在LRU队列的最底部，而warm的数据则放置到LRU的最顶部（即随时可能被踢出LRU队列），hot到warm的切换计数方法一般为：设N为key cache中的inode个数，key\_cache\_age\_threshold为时间阀值，如果在最近的（N\*key\_cache\_age\_threshold/100）次访问中，keycache顶部的inode仍然没有被访问到，那么就会被移到warmsub-chain的顶部。默认设置为0.75

参考：https://dev.mysql.com/doc/refman/5.7/en/midpoint-insertion.html

14.Linux操作系统dirent结构体

struct dirent

{

　　long d\_ino; /\* inode number 索引节点号 \*/

off\_t d\_off; /\* offset to this dirent 在目录文件中的偏移 \*/

unsigned short d\_reclen; /\* length of this d\_name 文件名长 \*/

unsigned char d\_type; /\* the type of d\_name 文件类型 \*/

char d\_name [NAME\_MAX+1]; /\* file name (null-terminated) 文件名，最长255字符 \*/

}

15.Linux操作系统stat结构体

struct stat {

mode\_t st\_mode; //文件访问权限

ino\_t st\_ino; //索引节点号

dev\_t st\_dev; //文件使用的设备号

dev\_t st\_rdev; //设备文件的设备号

nlink\_t st\_nlink; //文件的硬连接数

uid\_t st\_uid; //所有者用户识别号

gid\_t st\_gid; //组识别号

off\_t st\_size; //以字节为单位的文件容量

time\_t st\_atime; //最后一次访问该文件的时间

time\_t st\_mtime; //最后一次修改该文件的时间

time\_t st\_ctime; //最后一次改变该文件状态的时间

blksize\_t st\_blksize; //包含该文件的磁盘块的大小

blkcnt\_t st\_blocks; //该文件所占的磁盘块

};

# 服务端部分

1. auth\_pins

大多数对元数据的操作都需要一定时间才能完成，为了在操作日志时，缓存中的元数据对象不被并行修改，需要锁定它直到操作完成，这就是auth\_pins的含义，另外如果要冻结子目录，则对应的父目录也需要做冻结操作。

1. dir目录的几种状态

|  |  |  |
| --- | --- | --- |
| STATE\_COMPLETE | complete | 在cache中具有完成的内容 |
| STATE\_FREEZINGTREE | freezingtree | 正在freezing过程中 |
| STATE\_FROZENTREE | frozentree | 树的根出口？ |
| STATE\_AUXSUBTREE | auxsubtree | 没有子树merge |
| STATE\_FROZENDIR | frozendir | 冻结的dir，整个dir被冻结 |
| STATE\_FREEZINGDIR | freezingdir | Dir正在被冻结 |
| STATE\_EXPORTBOUND | exportbound | 导出边缘的树？ |
| STATE\_IMPORTBOUND | importbound | 导入边缘的树？ |
| STATE\_BADFRAG | badfrag | 已经损坏的子树 |
| STATE\_FRAGMENTING | fragmenting | ? |

1. MDS中通过name到Dentry中查找它的位置信息

函数如下：CDir::lookup

第一步：在CDir中会存储如下name+snap, Dentry对应的Map表信息：

std::map<dentry\_key\_t, CDentry\*> map\_t;

第二步：dentry\_key\_t的值本身为一个key关键值，它是通过（name+snap）的方式来

生成的，因此在查找过程中，需要通过hash的方式来查找，具体如下：

输入值： name+snap

Hash算法：Linux Dcache Hash

结果： dentry\_key\_t关键可以

第三步：通过dentry\_key\_t的key在map表中查找到第一个Dentry位置信息

1. Dirfrag中的版本号

Dirfrag是单个目录的分片集合，目前是为了解决单个目录中过多的内容影响文件存储的性能，具体的介绍，可以查看之前写的dirfrag介绍，下面重点介绍基于单个dirfrag分片做的版本记录：

1. *Version*

*表示当前dirfrag分片的版本号*

1. *projected version*

*表示当前的dirfrag正在写入journal中，还未完成，甚至还未应用于mds cache中*

1. *committing version*

将dirfrag的值提交到disk中，正在提交中

1. *committed version*

*已经提交完成*

*committed* <= *committing* <= *version* <= *pro jected*

1. 使用bloom filter(布伦过滤器)来管理Dentry

在MDCache中将需要trim的Dentry信息保存到bloom\_filter中(add\_to\_bloom)，方便后续在Dir中的bloom对象中查找。

bloom filter是一种二进制向量数据结构，使用hash算法来保存和查找数据，且时间复杂度都为常数，因此它具有很好的空间和时间效率，被用来检测一个元素是不是在一个集合中，检测结果为是，则该元素不一定在此集合中，检测结果为否，则确定该元素一定不在此集合中，因此它的缺点是不能正确判断元素是否在集合内。

Bloom Filter的核心是一个m位的bitset和k个hash函数。空间复杂度为O(m)，时间复杂度为O(k)，另外Bloom Filter跟单哈希函数Bit-Map不同之处在于：Bloom Filter使用了k个哈希函数，每个字符串跟k个bit对应。从而降低了冲突的概率。另外原生的bloom不支持删除，变体bloom可支持删除。

参考：<https://www.jianshu.com/p/8193d7dc8348>

<http://iluoxuan.iteye.com/blog/1718254>

1. Beacon在MDS中的作用

Beacon在MDS中起到MDS之间代理通信的作用，其功能包含消息转发，心跳检测、主备倒换等等。

1. MDS cache可选用的两种Hash算法

1.1 linux dcache hash

1.2 robert jenkins hash

1. CDentry中get\_linkage和get\_projected\_linkage的区别

linkage\_t linkage;//记录本CDentry自身的linkage

list<linkage\_t> projected;// 记录本CDentry下记录的linkage信息

1. MDS模块中各子模块（基于文件整理）的作用

|  |  |  |
| --- | --- | --- |
| 模块文件名称 | 作用 | 说明 |
| Beacon.cc | 将MDS的状态通知Mon来统一管理，比如MDS的健康状态、MDS的Active/Down/Standby等状态 |  |
| Capability.cc | 这个不太确定具体功能，代码与本文档中的Capabilities 介绍的不太一致? | 从MDS到Client，有2种类型Cap事件（1）在MClientReply或者MClientCaps “IMPORT op”操作中使用（2）更新撤销或者授予操作，通过MClientCaps消息； 客户端有cap，则mds也应该有cap功能。  Cap中的issued是什么意思？ |
| CDentry.cc | MDS服务端Dentry结构  Dentry是一个存储目录树的内存结构，保存了文件的名称和文件指向的inode号；  它双向指针，即指向父节点位置，也指向子节点；  一般情况下会将Dentry放置到缓存中保存。 | 与 CDir.cc有紧密的联系 |
| CDir | 内存模型，用于管理目录结构，以及目录的dirfrag管理 |  |
| CInode |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

1. MDS锁机制

MDS中的锁机制，实际上是使用分布式状态机来实现的一种简单锁，而非调用linux系统锁来实现，通过这些锁来实现对部分字段的读或者写进行控制。其最终的目的是在对目录树中的某个path进行操作之前，先对这个操作中涉及到的dentry或者inode加锁，保证操作一致性，在MDS中将涉及到以下几种类型的锁：

**（1）基于单个OP操作的锁：**

|  |
| --- |
| // held locks  structLockOp {  enum {  RDLOCK    = 1,  WRLOCK    = 2,  XLOCK   = 4,  REMOTE\_WRLOCK = 8,  };  ......  }; |

* RDLOCK

rdlocks，读写锁中的读锁，即可以同时读（线程使用读模式加锁：

pthread\_rwlock\_rdlock()）

* WRLOCK

wrlocks，读写锁中的写锁，即单写（线程以写模式加锁：pthread\_rwlock\_wrlock()）

* XLOCK

xlocks，排它锁

* REMOTE\_WRLOCK

远程锁是应用于MDS目录分片时，部分目录被分类到其它MDS上时，需要使用远程

锁来锁定

（2）MutationImpl

MutationImpl是MDS操作单个OP的入口，包括设置inode、Dir和ScatterLock等功能

（2）分散锁

ScatterLock

mds使用分布式状态机来实现锁机制，通过这种方式来控制mds指定字段的读操作、修改操作，虽然对单个Dentry都被single lock保护（控制这个命名空间的条目是否可读），但每个inode实际上还设置了5种类型的锁：每种类型的锁控制不同的fileds，比如：

1. link count
2. filename 归属和文件模式
3. 文件大小和mtime
4. 目录分片
5. 还有一个是?

另外这些锁的获取被严格排序通过如下配置<locktype,object>来避免死锁

构造每个锁的状态机以最小化MDS交互以满足受保护字段的预期使用， 大多数fileds都受到简单锁的保护，这些锁的目的是保证副本一致性和可读性，但在权威条目更新时，会使用独占write锁（相比较single lock）；另外一方面，文件大小和mtime受到具有与客户端文件访问模式相对应的状态的锁的保护，客户端访问模式有：单客户端访问、共享read、混合read/write、共享write；还有一种为分散锁（scatter）,这种锁会在如下场景下生效（比如对单个inode标记的utime，但是这个inode被设置为fragment，因此可能存在单个inode的多个fragment在不同的mds上），设置了分散锁的业务在被允许更新之前，需要满足锁的状态以及包含了所有fragment的状态信息（从各mds上获取），并且同时需要保证mtime是可读状态。

我们知道所有原始的更新操作，最终都转发到元数据对象中进行序列化和日志化操作。一些操作（比如link、unlink、rename）可能会触发在多个mds进行操作，比如在本地创建一个硬链接，但对应指向的inode则可能在另外的mds上，对应这种情况，本端会触发一个slave更新请求来增加link的个数，需要注意slave的更新是分为2阶段提交：第一阶段为：一旦所有的mds上的slave已经持有 prepare日志事件，则协调所有的mds开始进行committing事务，第二阶段为：当slave收到committed的事件后，则关闭事务。

MDS中定义了4种类型的锁，可以通过LockType定义看到，在服务端CInode中定义了10个锁（属于这4类），CDentry中定义了2个锁（属于这4类）。

|  |
| --- |
| struct LockType {  int type;  const sm\_t \*sm;  explicit LockType(int t) : type(t) {  switch (type) {  case CEPH\_LOCK\_DN:  case CEPH\_LOCK\_IAUTH:  case CEPH\_LOCK\_ILINK:  case CEPH\_LOCK\_IXATTR:  case CEPH\_LOCK\_ISNAP:  case CEPH\_LOCK\_IFLOCK:  case CEPH\_LOCK\_IPOLICY:  sm = &sm\_simplelock;  break;  case CEPH\_LOCK\_IDFT:  case CEPH\_LOCK\_INEST:  sm = &sm\_scatterlock;  break;  case CEPH\_LOCK\_IFILE:  sm = &sm\_filelock;  break;  case CEPH\_LOCK\_DVERSION:  case CEPH\_LOCK\_IVERSION:  sm = &sm\_locallock;  break;  default:  sm = 0;  }  }  }; |

|  |
| --- |
| LockType CInode::versionlock\_type(CEPH\_LOCK\_IVERSION);  LockType CInode::authlock\_type(CEPH\_LOCK\_IAUTH);  LockType CInode::linklock\_type(CEPH\_LOCK\_ILINK);  LockType CInode::dirfragtreelock\_type(CEPH\_LOCK\_IDFT);  LockType CInode::filelock\_type(CEPH\_LOCK\_IFILE);  LockType CInode::xattrlock\_type(CEPH\_LOCK\_IXATTR);  LockType CInode::snaplock\_type(CEPH\_LOCK\_ISNAP);  LockType CInode::nestlock\_type(CEPH\_LOCK\_INEST);  LockType CInode::flocklock\_type(CEPH\_LOCK\_IFLOCK);  LockType CInode::policylock\_type(CEPH\_LOCK\_IPOLICY); |

CInode定义锁

|  |
| --- |
| SimpleLock authlock;  SimpleLock linklock;  ScatterLock dirfragtreelock;  ScatterLock filelock;//分散锁  SimpleLock xattrlock;  SimpleLock snaplock;  ScatterLock nestlock;  SimpleLock flocklock;  SimpleLock policylock; |

Locker本身的锁状态和种类：

|  |
| --- |
| // -- lock states --  // sync <-> lock  enum {  LOCK\_UNDEF = 0,  // auth rep  LOCK\_SYNC, // AR R . RD L . / C . R RD L . / C .  LOCK\_LOCK, // AR R . .. . X / . . . .. . . / . .  LOCK\_PREXLOCK, // A . . .. . . / . . (lock)  LOCK\_XLOCK, // A . . .. . . / . . (lock)  LOCK\_XLOCKDONE, // A r p rd l x / . . (lock) <-- by same client only!!  LOCK\_XLOCKSNAP, // also revoke Fb  LOCK\_LOCK\_XLOCK,  LOCK\_SYNC\_LOCK, // AR R . .. . . / . . R .. . . / . .  LOCK\_LOCK\_SYNC, // A R p rd l . / . . (lock) <-- lc by same client only  LOCK\_EXCL, // A . . .. . . / c x \* (lock)  LOCK\_EXCL\_SYNC, // A . . .. . . / c . \* (lock)  LOCK\_EXCL\_LOCK, // A . . .. . . / . . (lock)  LOCK\_SYNC\_EXCL, // Ar R . .. . . / c . \* (sync->lock)  LOCK\_LOCK\_EXCL, // A R . .. . . / . . (lock)  LOCK\_REMOTEXLOCK, // on NON-auth  // \* = loner mode  LOCK\_MIX,  LOCK\_SYNC\_MIX,  LOCK\_SYNC\_MIX2,  LOCK\_LOCK\_MIX,  LOCK\_EXCL\_MIX,  LOCK\_MIX\_SYNC,  LOCK\_MIX\_SYNC2,  LOCK\_MIX\_LOCK,  LOCK\_MIX\_LOCK2,  LOCK\_MIX\_EXCL,  LOCK\_TSYN,  LOCK\_TSYN\_LOCK,  LOCK\_TSYN\_MIX,  LOCK\_LOCK\_TSYN,  LOCK\_MIX\_TSYN,  LOCK\_PRE\_SCAN,  LOCK\_SCAN,  LOCK\_SNAP\_SYNC,  LOCK\_XSYN,  LOCK\_XSYN\_EXCL,  LOCK\_EXCL\_XSYN,  LOCK\_XSYN\_SYNC,  LOCK\_XSYN\_LOCK,  LOCK\_XSYN\_MIX,  LOCK\_MAX,  };  // -------------------------  // lock actions  // for replicas  #defineLOCK\_AC\_SYNC-1  #defineLOCK\_AC\_MIX-2  #defineLOCK\_AC\_LOCK-3  #defineLOCK\_AC\_LOCKFLUSHED-4  // for auth  #defineLOCK\_AC\_SYNCACK1  #defineLOCK\_AC\_MIXACK2  #defineLOCK\_AC\_LOCKACK3  #defineLOCK\_AC\_REQSCATTER7  #defineLOCK\_AC\_REQUNSCATTER8  #defineLOCK\_AC\_NUDGE9  #defineLOCK\_AC\_REQRDLOCK10  #defineLOCK\_AC\_FOR\_REPLICA(a) ((a) <0)  #defineLOCK\_AC\_FOR\_AUTH(a) ((a) >0)  staticinlineconstchar \*get\_lock\_action\_name(inta) {  switch (a) {  case LOCK\_AC\_SYNC: return"sync";  case LOCK\_AC\_MIX: return"mix";  case LOCK\_AC\_LOCK: return"lock";  case LOCK\_AC\_LOCKFLUSHED: return"lockflushed";  case LOCK\_AC\_SYNCACK: return"syncack";  case LOCK\_AC\_MIXACK: return"mixack";  case LOCK\_AC\_LOCKACK: return"lockack";  case LOCK\_AC\_REQSCATTER: return"reqscatter";  case LOCK\_AC\_REQUNSCATTER: return"requnscatter";  case LOCK\_AC\_NUDGE: return"nudge";  case LOCK\_AC\_REQRDLOCK: return"reqrdlock";  default: return"???";  }  } |

下面是创建一个新目录时，涉及到的打印信息



简单锁管理的元数据有文件的LINK数、mode、uid、gid等，这些信息在多个副本间要么是共享读，要么是互斥写，和传统的锁类似。

fcntl\_locks ceph\_lock\_state\_t

flock\_locks ceph\_lock\_state\_t

/\*

\* metadata lock types.

\* - these are bitmasks.. we can compose them

\* - they also define the lock ordering by the MDS

\* - a few of these are internal to the mds

\*/

// 元数据锁类型

// 这里使用bitmask来进行组合，同时也定义了锁定顺序，在这些锁中有一些是内

部锁

// CDentry相关的锁(CEPH\_LOCK\_DVERSION不确定)

#define CEPH\_LOCK\_DVERSION 1

#define CEPH\_LOCK\_DN 2

#define CEPH\_LOCK\_IVERSION 16 /\* mds internal \*/

// CInode相关的锁

#define CEPH\_LOCK\_ISNAP 32

#define CEPH\_LOCK\_IFILE 64

#define CEPH\_LOCK\_IAUTH 128

#define CEPH\_LOCK\_ILINK 256

#define CEPH\_LOCK\_IDFT 512 /\* dir frag tree \*/

#define CEPH\_LOCK\_INEST 1024 /\* mds internal \*/

#define CEPH\_LOCK\_IXATTR 2048

#define CEPH\_LOCK\_IFLOCK 4096 /\* advisory file locks \*/

#define CEPH\_LOCK\_INO 8192 /\* immutable inode bits; not a lock \*/

#define CEPH\_LOCK\_IPOLICY 16384 /\* policy lock on dirs. MDS internal \*/

3种cap的能力类型:

#defineCAP\_ANY0

#defineCAP\_LONER1

#defineCAP\_XLOCKER2

1. Load Balance

每一个MDS通过关联的计数器监视缓存元数据（inode和fragment）的流行程度使用情况。每一个inode会从2个维度，即read、write来统计流行度。而directotry fragment则会监视readdir操作和元数据从objects中获取和提交频率。除了以上的流行度统计外，每个目录还会维护3个额外的元数据嵌套负载统计信息：其中一个应用于所有的嵌套元数据统计；一个统计当前所有嵌套的权威元数据；最后一个是应用于当前子树权威元数据。

当MDS被client访问时，对于元数据和对应的祖先元数据都需要递增计算，已构造一个元数据流程度的分布图，该图将做为元数据复制和迁移的决策基础，并同时使用指数因子来提供近似平滑的流行度迁移策略(?)。

每个MDS会定期同步自己的负载信息，以及基于每个子树的直接祖先权威MDS分组来管理的元数据流程程度。这允许高负载的节点确定整体的工作负载的比例，以及减少到低负载的节点上，同时支持重新与父统一子树迁移(?)。然后基于分层流程度来觉得迁移合适的子树。

虽然流行度是测量元数据访问频率，但最终在每个directory中会测量流行度spread，即每一个目录有一个简短客户端列表，可以访问其记录信息。仅当处理新客户端的请求时，扩展计数器才会递增。

Balance的几种状态

|  |  |  |
| --- | --- | --- |
| 宏定义 | 作用 | 说明 |
| META\_POP\_IRD | 读split类型 |  |
| META\_POP\_IWR |  |  |
| META\_POP\_READDIR |  |  |
| META\_POP\_FETCH |  |  |
| META\_POP\_STORE |  |  |
| META\_NPOP |  |  |

1. 子树迁移

每一个子树迁移都是子树缓存元数据的迁移，此交换包括非脏元数据（严格说，正确性不需要），因为缓存可能与MDS管理它无关，迁移期间的一次性成本远远低于从元数据存储池中获取。迁移涉及到初始消息的交换，然后再进行2阶段提交：import-start阶段，将导入节点记录所有待导入元数据副本，Export阶段：导出节点将提交迁移日志通过Export event，并通过ImportFinish来关闭事务日志。对于复制了涉及迁移子树的元数据的其它mds节点，则会收到通知，保证这些mds会知晓新的子树位置。同时子树的迁移必须保证可靠性，因此必须要保证一放完全可用的情况下，才会删除迁出方（？）。inode和Dentry的嵌入方式方便了子树的迁移流程，将所有元数据放置在具有明确定义的分区单个层次命名空间中，类似地，使用共享对象存储库进行元数据存储有助于迁移文件系统的任意大的子树，因为每次迁移仅涉及缓存的元数据的传输。如果要加载对象则通过共享的方式加载即可（对于迁移来说，几乎没有成本）。

1. 服务端消息

|  |  |  |
| --- | --- | --- |
| 消息号 | 消息处理逻辑 |  |
| CEPH\_CAP\_OP\_GRANT | mds->client grant |  |
| CEPH\_CAP\_OP\_REVOKE | mds->client revoke |  |
| CEPH\_CAP\_OP\_TRUNC | mds->client trunc notify |  |
| CEPH\_CAP\_OP\_EXPORT | mds has exported the cap |  |
| CEPH\_CAP\_OP\_IMPORT | mds has imported the cap |  |
| CEPH\_CAP\_OP\_UPDATE | client->mds update |  |
| CEPH\_CAP\_OP\_DROP | client->mds drop cap bits |  |
| CEPH\_CAP\_OP\_FLUSH | client->mds cap writeback |  |
| CEPH\_CAP\_OP\_FLUSH\_ACK | mds->client flushed |  |
| CEPH\_CAP\_OP\_FLUSHSNAP | client->mds flush snapped metadata |  |
| CEPH\_CAP\_OP\_FLUSHSNAP\_ACK | mds->client flushed snapped metadata |  |
| CEPH\_CAP\_OP\_RELEASE | client->mds release (clean) cap |  |
| CEPH\_CAP\_OP\_RENEW | client->mds renewal request |  |
|  |  |  |

文件访问的6种模式

1. Flag标记

/\* file access modes \*/

#define CEPH\_FILE\_MODE\_PIN 0

#define CEPH\_FILE\_MODE\_RD 1

#define CEPH\_FILE\_MODE\_WR 2

#define CEPH\_FILE\_MODE\_RDWR 3 /\* RD | WR \*/

#define CEPH\_FILE\_MODE\_LAZY 4 /\* lazy io \*/

#define CEPH\_FILE\_MODE\_NUM 8 /\* bc these are bit fields.. mostly \*/ bit最后标志位

// 打开的标记位，这个是ceph内部的flag标记，与POXIS标准文件系统的flag对应

#define CEPH\_O\_RDONLY 00000000

#define CEPH\_O\_WRONLY 00000001

#define CEPH\_O\_RDWR 00000002

#define CEPH\_O\_CREAT 00000100

#define CEPH\_O\_EXCL 00000200

#define CEPH\_O\_TRUNC 00001000

#define CEPH\_O\_LAZY 00020000

#define CEPH\_O\_DIRECTORY 00200000

#define CEPH\_O\_NOFOLLOW 00400000

下面可以看到flags其实是分为两种类型，第一类3个是互斥的，其它的都可以是OR的方

式同时使用

（1）一类为：

O\_RDONLY    以只读方式打开文件

O\_WRONLY    以只写方式打开文件

O\_RDWR      以可读写方式打开文件

（2）二类为：

O\_CREAT   若欲打开的文件不存在则自动建立该文件。

O\_EXCL    如果O\_CREAT也被设置，此指令会去检查文件是否存在。文件若不存在则建立

该文件，否则将导致打开文件错误。此外，若O\_CREAT与O\_EXCL同时设置，并且欲打

开的文件为符号连接，则会打开文件失败。

O\_NOCTTY  如果欲打开的文件为终端机设备时，则不会将该终端机当成进程控制终端机。

O\_TRUNC   若文件存在并且以可写的方式打开时，此标志位会令文件长度清为0，而原来

存于该文件的资料也会消失。

O\_APPEND  当读写文件时会从文件尾开始移动，也就是所写入的数据会以附加的方式加入

到文件后面。

O\_NONBLOCK    以不可阻断的方式打开文件，也就是无论有无数据读取或等待，都会立即

返回进程之中。

O\_NDELAY  同O\_NONBLOCK。

O\_SYNC    以同步的方式打开文件。

O\_NOFOLLOW    如果参数pathname所指的文件为一符号连接，则会令打开文件失败。

O\_DIRECTORY   如果参数pathname所指的文件并非为一目录，则会令打开文件失败。

O\_PATH 将不会真正打开一个文件，而只是准备好该文件的文件描述符，并且会

忽视大部分其它的标志位

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **POSIX标志**  **(flags)** | **含义** | **Ceph定义字段(cflags)**  **ceph\_flags\_sys2wire** | **文件访问模式(cmode)**  **ceph\_flags\_to\_mode** | **文件模式到Cap(want)**  **ceph\_caps\_for\_mode** |
| O\_ACCMODE& O\_WRONLY | 只写模式 | CEPH\_O\_RDWR | CEPH\_FILE\_MODE\_WR |  |
| O\_ACCMODE& O\_RDWR | 读写模式 | CEPH\_O\_WRONLY | CEPH\_FILE\_MODE\_RDWR | CEPH\_CAP\_FILE\_EXCL  CEPH\_CAP\_FILE\_WR  CEPH\_CAP\_FILE\_BUFFER  CEPH\_CAP\_AUTH\_SHARED  CEPH\_CAP\_AUTH\_EXCL  CEPH\_CAP\_XATTR\_SHARED  CEPH\_CAP\_XATTR\_EXCL |
| O\_CREAT | 如果指定文件不存在，则创建这个文件 | CEPH\_O\_CREAT |  |  |
| O\_TRUNC | 如果文件存在，并且以只写/读写方式打开，则清空文件全部内容(即将其长度截短为0) | CEPH\_O\_TRUNC |  |  |
| O\_APPEND | 每次写操作都写入文件的末尾 |  |  |  |
| O\_ACCMODE& O\_RDONLY | 只读模块 | CEPH\_O\_RDONLY | CEPH\_FILE\_MODE\_RD | CEPH\_CAP\_FILE\_SHARED  CEPH\_CAP\_FILE\_RD  CEPH\_CAP\_FILE\_CACHE |
| O\_EXCL | 如果要创建的文件已存在，则返回-1，并且修改errno的值 | CEPH\_O\_EXCL |  |  |
| O\_NONBLOCK | 如果路径名指向FIFO/块文件/字符文件，则把文件的打开和后继I/O设置为非阻塞模式 | CEPH\_ |  |  |
| O\_SYNC | 等待物理I/O结束后再write，包括更新文件属性的I/O | CEPH\_ |  |  |
| O\_NOCTTY | 如果路径名指向终端设备，不要把这个设备用作控制终端 | CEPH\_ |  |  |
| O\_DSYNC | 等待物理I/O结束后再write。在不影响读取新写入的数据的前提下，不等待文件属性更新 | CEPH\_ |  |  |
| O\_RSYNCread | 等待所有写入同一区域的写操作完成后再进行 | CEPH\_ |  |  |
| Ceph特有标志 | 异步写入,应用于HPC对读写一致要求不高的场景 | CEPH\_O\_LAZY | CEPH\_FILE\_MODE\_LAZY | CEPH\_CAP\_FILE\_LAZYIO |
| O\_NOFOLLOW | 如果参数pathname所指的文件为一符号连接，则会令打开文件失败 | CEPH\_ O\_NOFOLLOW |  |  |
| O\_DIRECTORY | 如果参数pathname所指的文件部是目录时，则会令打开文件失败 | CEPH\_O\_DIRECTORY | CEPH\_FILE\_MODE\_PIN |  |
| O\_PATH | 只打开文件描述符 |  |  |  |

1. mode标记

在open操作时，需要第3个参数mode来设置创建文件的权限，其参数的标记和解释如下：

|  |  |
| --- | --- |
| **mode参数标记** | **含义** |
| S\_IRUSR | 文件属主，读权限 |
| S\_IWUSR | 文件属主，写权限 |
| S\_IXUSR | 文件属主，执行权限 |
| S\_IRGRP | 文件所属组，读权限 |
| S\_IWGRP | 文件所属组，写权限 |

1. Ceph客户端驱逐实现

由于网络或者其它原因，导致客户端长期无响应时，则MDS非常有必要将其提出，原因有以下2点：

1. 如果一个客户端连接mds，并在做buffer io操作，长期的无效的连接，可能存在丢失

un-flush data数据。

1. 如果某个连接长期连接时，获取的mds可能一直不能释放，导致其它client无法正常

获取。

驱逐的方式有以下2种形式：

1. 自动驱逐

有以下3种情况会自动驱逐

1. 如果client与mds连接超过mds\_session\_autoclose（默认300s）时，14版本为硬编码了：MDSMap.h session\_autoclose = 300 session\_timeout = 60
2. 在活动的MDS上，如果在mds\_cap\_revoke\_eviction\_timeout（默认0s，未开启）没有收到某个client的cap revoke message时
3. 在MDS启动或者切换过程中，client重新连接到新的mds的时间超过mds\_reconnect\_timeout（默认45s）时
4. 手工驱逐

第一步：先通过ceph tell mds.sky client ls 或者 ceph daemon mds.sky session ls先来获取客户端id号

第二步：使用以下3种方式来kill客户端：

1. ceph tell mds.sky client evict id=4305
2. ceph tell mds.sky client evict client\_metadata.=4305
3. ceph daemon mds.sky session evict 4305
4. Ceph 黑名单模式

驱逐客户端（手动或者自动）的前提的条件是需要（大部分）先将此客户端放置到blacklist中，这样才能保证数据的安全性，默认通过mds\_session\_blacklist\_on\_evict（默认为true）设置，来达到所有的驱逐client行为都被加入到黑名单。需要注意：加入黑名单的client是无法重新连接的，需要重新mount才能连接。在某些情况下，需要被驱逐的客户端能重新连接到mds上，则要达到这种要求，需要将此客户端从blacklist中拿掉，另外需要客户端有相应的重新连接机制，ceph-fuse的client\_reconnect\_stale（默认为false）测试貌似不行，只能是stale状态可以重新连接，驱逐后的不行???---需要主动去查询挂载点，这样才能client才能与MDS主动重新建立连接(https://github.com/ceph/ceph/pull/25343需要合入)

但有一种情况下，我们不要client加入到blacklist中，即当网络不稳定时，还是希望client能提供服务，在这种情况下，需要将**mds\_session\_blacklist\_on\_timeout（默认为true）**设置为flase。

需要注意：如果blacklist功能没有开启时驱逐client，那么只会对当前指定的mds有效，在多个active mds中，需要都进行手动驱逐；在blacklist开启时，则只需要发送到一个即可，因为blacklist会进行传播

**手工设置黑名单方式：**

Ceph client是通过osd的blacklist功能来实现黑名单功能，即通过先获取客户端的地址后：使用ceph osd blacklist add<address>来添加，使用ceph osd blacklist rm <address>删除名单。

Background: Blacklisting and OSD epoch barrier，没有看明白?

以上链接为：https://github.com/ceph/ceph/pull/24086/files

1. damageTable

DamageTable将会记录在meta存储池中，已经被scrub标记位不可用对象，或者在获取对象异常后被标记位不可用。当然它实际的记录的信息不是对象信息，而是基于对象信息来解析哪些dentry或者dirfrag(包含inode)不可用，并在此结构体中记录。

is\_dentry\_damaged，记录dentry损坏信息

is\_dirfrag\_damaged，记录dirfrag损坏信息

is\_remote\_damaged，记录remote损坏信息

1. CInode在MDS分类

CInode 类定义：

|  |
| --- |
| CInode结构体  CInode主要继承了InodeStoreBase  struct inode\_t {  inodeno\_t ino = 0;  uint32\_t rdev = 0; // if special file  ......  }  class InodeStoreBase {  ......  typedef inode\_t<mempool::mds\_co::pool\_allocator> mempool\_inode;  mempool\_inode inode;  ......  }  class CInode : public MDSCacheObject, public InodeStoreBase, public Counter<CInode> {  ......  } |

CInode 在MDS上有几种分类

|  |  |  |
| --- | --- | --- |
| **种类** | **定义值** | **大小** |
| is\_root（） | inode.ino == MDS\_INO\_ROOT | MDS\_INO\_ROOT=1 |
| is\_mdsdir() | MDS\_INO\_IS\_MDSDIR  (inode.ino) | [256,512] |
| is\_stray() | MDS\_INO\_IS\_STRAY(inode.ino) | [1536, 2560] |
| is\_base() | MDS\_INO\_IS\_BASE(inode.ino)  MDS\_INO\_GLOBAL\_SNAPREALM=3 | [1] || [256,512] || [3] |
| is\_system() | MDS\_INO\_SYSTEM\_BASE=4096 | 小于4096都属于system inode |
| is\_normal() | !(is\_base() || is\_system() || is\_stray()) | 4096以上，都属于normal状态 |

1. Inode操作保护类

Inode的保护类作用是应用于保存inode在修改时状态，直到对应的修改写入到日志后，才会将此记录从inode清理掉，在CInode中声明为projected\_nodes

|  |
| --- |
| class projected\_inode {  public:  static sr\_t\* const UNDEF\_SRNODE;  mempool\_inode inode;  std::unique\_ptr<mempool\_xattr\_map> xattrs;//修改xattrs时使用  sr\_t \*snapnode = UNDEF\_SRNODE;//快照结构修改是使用  projected\_inode() = delete;  explicit projected\_inode(const mempool\_inode &in) : inode(in) {}  }; |

1. ACL实现

ACL中单个dentry有以下类型的tag

#defineACL\_USER\_OBJ0x01

#defineACL\_USER0x02

#defineACL\_GROUP\_OBJ0x04

#defineACL\_GROUP0x08

#defineACL\_MASK0x10

#defineACL\_OTHER0x20

1. 操作系统定义读写宏定义

（1）linux文件系统vfs定义的宏

|  |
| --- |
| // unix-style capabilities  enum {  MAY\_READ  = (1<<0),  MAY\_WRITE   = (1<<1),  MAY\_EXECUTE = (1<<2),  MAY\_CHOWN = (1<<4),  MAY\_CHGRP = (1<<5),  MAY\_SET\_VXATTR = (1<<6),  MAY\_SNAPSHOT  = (1<<7),  }; |

MDS\_REF\_SET

1. MDS\_REF\_SET CInode和cache状态统计

MDS\_REF\_SET是做为调试开关加入到MDS中，大部分是用来在MDScache和CInode等之间进行统计使用。

1. 在MDSCacheObject使用

|  |
| --- |
| #ifdefMDS\_REF\_SET  mempool::mds\_co::flat\_map<int,int> ref\_map;  #endif  ref\_map用于记录  get(PIN\_DIRTYRSTAT); 设置ref\_map[PIN\_DIRTYRSTAT]++;  put(PIN\_DIRTYRSTAT);减去ref\_map[PIN\_DIRTYRSTAT]--; |

1. MDS sticky功能

sticky功能是一个权限bit位，在文件或目录中标识只有其归属的用户和root用户才能删除它或者修改名称，其它用户不能做此操作。

1. dirfrag的实现

实现的结构体为：fragtree\_t

1. CInode状态信息

static const int STATE\_EXPORTING = (1<<0); // on nonauth bystander.

static const int STATE\_OPENINGDIR = (1<<1);

static const int STATE\_FREEZING = (1<<2);

static const int STATE\_FROZEN = (1<<3);

static const int STATE\_AMBIGUOUSAUTH = (1<<4);

static const int STATE\_EXPORTINGCAPS = (1<<5);

static const int STATE\_NEEDSRECOVER = (1<<6);

static const int STATE\_RECOVERING = (1<<7);

static const int STATE\_PURGING = (1<<8);

static const int STATE\_DIRTYPARENT = (1<<9);

static const int STATE\_DIRTYRSTAT = (1<<10);

static const int STATE\_STRAYPINNED = (1<<11);

static const int STATE\_FROZENAUTHPIN = (1<<12);

static const int STATE\_DIRTYPOOL = (1<<13);

static const int STATE\_REPAIRSTATS = (1<<14);

static const int STATE\_MISSINGOBJS = (1<<15);

static const int STATE\_EVALSTALECAPS = (1<<16);

static const int STATE\_QUEUEDEXPORTPIN = (1<<17);

static const int STATE\_TRACKEDBYOFT = (1<<18); // tracked by open file table

// orphan inode needs notification of releasing reference

static const int STATE\_ORPHAN = STATE\_NOTIFYREF;

|  |  |  |
| --- | --- | --- |
| 宏定义 | 状态的意义 | 说明 |
| STATE\_EXPORTING | 正在在导出 |  |
| STATE\_OPENINGDIR | 打开的dir |  |
| STATE\_FREEZING | 正在进行freezing |  |
| STATE\_FROZEN | 已经是frozen状态 |  |
| STATE\_AMBIGUOUSAUTH | 模棱两可的auth |  |
| STATE\_EXPORTINGCAPS | 已经导出记录 | 表示当前的inode cap正在变化? |
| STATE\_NEEDSRECOVER | 需要避免覆盖 |  |

1. MDlog

MDlog是MDS元数据落盘的方式

（1）MDlog发起流程

start\_entry函数记录操作标记，比如是读、写、truncate等操作

实例：EUpdate \*le = new EUpdate(mdlog, "open\_truncate");

mdlog->start\_entry(le);

（2）submit\_entry

提交单条日志记录

predirty\_journal\_parents函数 待学习???

// flags for predirty\_journal\_parents()

static const int PREDIRTY\_PRIMARY = 1; // primary dn, adjust nested accounting

static const int PREDIRTY\_DIR = 2; // update parent dir mtime/size

static const int PREDIRTY\_SHALLOW = 4; // only go to immediate parent (for easier rollback)

# 公共部分

# 性能部分---原理

FFS有一个特性:cylinder groups,它能将局部的数据落在相同的圆柱形区域内,即一个旋转硬盘盘片内。关联的数据和元数据被存储在同样的cylinder groups内。通过文件系统获取数据主要分为2个部分，首先是需要搜索文件在硬盘上的位置，其次是读取具体的数据，对于大部分的性能瓶颈都在数据在搜索文件位置上，因此提升这部分功能也成为重点之一。LFS文件系统设计之初就考虑了定位数据的延时问题，它是通过日志的方式来来顺序的记录位置信息，并保存到磁盘新的位置上，来保证数据被重新定位并读取。LFS为了从新利用空闲的（已经被删除的空间），需要调用Cleaner功能来对空间进行从新分配，这样会显烈的降低LFS的读写性能。

DualFS是基于元数据和数据分离的系统，它是通过类似LFS的方式，将元数据通过log segment的方式进行序列化来达到提升性能，同时限制log segment的大小，保证Cleaner时的性能。虽然DualFS也像LFS一样需要inode map表将inode number映射为inode 在磁盘上的地址信息，但DualFS在log中，会将最新修改后的inodes和Dentries信息存储在一起，来提升性能。

FFS引入的机构虽然经过很大的演进，但随着文件系统中文件大小扩张速度快于磁盘架构，导致元数据这种体系架构不再适应，比如文件大小越来越大，但block size默认大小还是4K（为了保证小文件读写效率），但这种默认配置对大文件的存储，则导致极大的块分配列表，导致性能下降。因此一种新型的现代本地文件系统（代表：XFS）将使用extents（标记start~length对）代替固定大小分配的方式来紧凑的记录大文件。以XFS为代表的新型文件系统为了简化系统异常时的恢复流程，增加了基于存储在磁盘上的journal文件，来记录文件系统中元数据的变化。虽然这样会导致元数据出现两次写的问题，而导致性能下降，但是它也有好处，即可以在文件系统异常后，进行快速的进行一致性检测，特别是在文件系统容量越来越大的情况下。

Ceph MDS则是结合了LFS的log方式来提升元数据的写效率，同时也利用Journaling来解决MDS crash后的一致性检查。

WAFL 文件系统是Netapp公司创建的，它是使用COW（copy-on-write）技术，即只有有更新的数据才会进行copy，并写入到新的磁盘空间上，这种方式只需要更新新的地址信息，即可和原来的root tree node进行连接，WAFL也会使用类似的journal方式，但它是为了保持更新提交之前的数据不会被丢失，而不是为了维护文件系统的一致性。

对于集中式CS架构模型的存储系统，比如NFS、CIFS。这种集中式的方式导致存储系统的扩展性比较差，因为所有的数据都需要经过某几个代理节点来处理所有的访问数据。对于性能方面，这种CS模型的网络文件系统为了保证性能，会放宽一致性的语义。比如NFS客户端会通过异常的方式将数据写回到服务端，这样会出现另外的客户端访问时，不一定会获取到最新的数据，同样客户端会缓存元数据信息（比如stat信息，且一般都有时间时效），来减少客户端与服务端的交互。这种一致性的宽松，会导致某些业务从一开始就会放弃使用NFS文件系统。

分布式文件系统主要是解决负载均衡和扩展性的问题，早期的分布式文件系统有AFS、Coda、Sprite，这种分布式系统一般是通过中央服务器来协调访问，并发布leases明确数据或者元数据的有效性，以这种方式提供给client的cache，如果随后的访问出现了冲突（比如多个client同时写一个文件），则对应的回调函数会撤销之前承诺的有效性。其并不是所有的文件系统都是这样处理的，比如Sprite是在有冲突时，disable client的cache，AFS则使用受约束的一致性模型来代替一致性模型来打开和关闭文件，而非单独的读写事件？，NFS是无状态连接协议（即stateless，NFS是无状态的协议。无状态是指NFS服务器不必维护它的每一台客户机的状态信息，客户机必须记住它自己的状态信息，缺点是NFS服务端应答之前，必须全部落盘？），在共享文件下，牺牲了一致性。其它文件系统，比如xFS，是基于失效缓存机制来促进client的缓存，以换取昂贵的网络通讯；OceanStore的目标是使用EC和位置无关路由来构建全球统一的文件系统；Pangaea是针对广域网，积极复制副本数据，但不做强一致性要求，总之这些分布式文件系统的目标是要减少昂贵的网络通讯，甚至牺牲一致性和实现广泛的扩展性，对ceph来说，提升性能的方式则与上面不一样。

# Client基础---原理

1. 客户端换取数据流
2. Client携带Filename发送消息到MDS，
3. MDS将Filename的名称转换为具体某个Inode信息
4. Inode信息包含Inode number信息、file owner、mode、size和其它元数据信息(stime\mtime等)
5. 如果文件存在，且客户端访问被授权（granted），则MDS返回给客户端inode number和file size以及striping策略（即layout方式）映射到object，同时MDS会发出哪些功能允许读或写。Cap最开始设计时，包含4个bit，分别表示client是否具有如下功能：read、cache read、write、buffer write。现在已经增加key信息来确保client是否有权限read或者write到osd的数据。
6. client同步机制

Posix语义要求读操作必须反应先前任何写操作，且写是原子操作的。对应并发写操作，则在实际应用中还是会有顺序要求。当有多个Client同时写或者混合读写时，则MDS会撤销(revoke)之前发布的读取cache和写入buffer的功能，并强制所有客户端进行IO同步。也就是说所有的客户端的读写都会被阻塞，直到得到osd的回复，这种“粗暴”的方式简化了更新读写序列的复杂度。当写入原子性在涉及到objects对象边界时（即写操作涉及到多个object对象），则显示的相当复杂，目前解决方案是使用lock机制来保证序列化，社区有计划改进这块?

很明显，IO同步这种机制会导致性能急速下降，尤其对于小文件读写延时非常明显。读写共享在通用应用中，其实比较少用到，一般出现在科学计算中。在应用程序确定不会有读写共享的情况下，我们可以放松一致性来解决性能问题，但鱼和熊掌不兼得。为了高性能应用，HPC（high-performance computer）组提出了Posix扩展接口，其中一部分由Ceph实现，Ceph实现的HPC接口部分通过flag为O\_LAZY标志标识。使用HPC接口，就默认放松了存储端的数据一致性，因此需要应用程序自己保证数据一致性，比如多写入时，确保是写入不同对象或者对象中的不同位置（？）当应用程序需要显示的进行同步时，则可以调用如下2个接口：

1. *lazyio propagate*

*将指定范围内的bytes更新到object对象中*

1. *lazyio synchronize*

*指进行同步更新操作，保证之前有变化的内容在本身更新后都可见，实现方式是通过置cache无效的方式，将client的请求发送到osd端*

*以上2个接口在保证高性能的同时，也为前端应用提供了在读写共享模式下一致性的需求。*

1. 命名空间的操作

命名空间的操作，即指对元数据的操作，比如read操作：readdir、stat和updates操作：unlink、chmod等通过MDS来保证元数据的序列化和安全性。为了简单起见，没有元数据锁和租期发送到客户端，特别是对于HPC工作负载，因为callback起到作用有限，但增加系统的复杂程度。

相反，ceph优化了最常用的元数据访问，比如readdir操作后随每个文件的统一信息就是一种常用的访问方式，默认readdir是一个简单的MDS请求操作，来获取整某个目录下的信息，包含inode信息。默认使用readdir调用后立即跟随stats，则只会返回cache中简要信息，否则丢弃。虽然放松与inode的交互会丢弃一部分信息，但是Ceph却可以以此获取极大的性能提升。如果想要获取完全一致的stats数据，则可以通过readdirplus扩展属性来获取。Ceph 可以通过设置更长时间的缓存（如设置30s）来放宽一致性，类似早期的NFS 版本。但是这样对于某些应用却有着致命的影响，比如某些应用需要使用stat来获取文件更新状态。为此Ceph也提供了扩展接口statlite掩码指定哪些位置需要更新。

1. 消息模型

Ceph是基于异步消息模型来实现节点之间的通讯，与基于RPC通讯的方式形成对比。异步消息的方式在消息传送出去后不会阻塞，而是在消息队列中等待下次调用，并且数据交换不需要请求和相应对。

# MDS实现---原理

一般分布式文件系统会将元数据和数据进行分离，同时业界多项研究表明，元数据的访问量占到整个文件系统访问量的30%~70%，因此可以看出元数据的性能对文件系统的重要性，元数据独立保存，元数据的访问都是一些非常小的IO结构，且一般元数据需要进行数据同步，因此提升元数据的读写效率，显得非常重要。

随着分布式文件越来越大，无论是data数据，还是meta数据，都不能使用静态分区的方式放置数据，data数据则会使用crush来动态的将数据打散到不同节点上，但meta则不太适应这种方式（原因见ceph设计原理），需要一种新的方式来实现meta的动态元数据分配，即我们后面说到的动态子树分区方式加负载均衡迁移机制。

Ceph MDS设计相对传统的元数据服务器有2个非常特别的优点：（1）Inode会嵌入到关联的Dentry中，并通过辅助表来实现hard link（最新版本没有使用这种方式），这种方式带来了2个巨大大好处：预取元数据信息提升性能和间接的简化了动态子树迁移功能的实现难度；（2）MDS允许日志保留比大，可以达到几百兆甚至上G的日志文件，这中方式带了3个好处：优化了重复提交、防止短生命周期的文件落盘到共享存储池、优化随机IO读写和简化了MDS恢复流程和新的MDS恢复效率。

本章主要关注MDS故障恢复、元数据I/O效率和适应性结合对元数据存储、日志和工作负载的影响

。。。。。。4.1

4.3.3 Lock

QA

1. 为什么会出现Client <name> failing to respond to capability release

MDS会发送revoke消息给client，如果客户端没有响应，则会有此打印

原因可能是：网络原因；客户端繁忙处理不过来；软件bug；非兼容的client

mds\_cap\_revoke\_eviction\_timeout参数默认为0，可以设置相应的值来强制将client

剔除

2. 为什么会出现Client <name> failing to respond to capability release

MDS缓存中已满，要求client释放缓存信息，来减少服务端内存使用，但客户端并

没有释放掉，导致MDS端的缓存持续增加

原因为：client还实际占用了待释放的inode信息，即元数据是pin状态；网络异常；

软件bug；非兼容的客户端