Ceph分布式文件锁

在基于客户端服务器模型的分布式文件系统中，需要解决多客户端并发访问控制和缓存一致性的问题。

> CIFS实现了一种oplock的文件锁机制，锁的粒度以文件为单位，它支持对文件单写者多读者的并发访问。
>
> 很多分布式文件系统使用的锁协议都与CIFS类似，它们只提供对文件的粗粒度的并发访问，任何时候只能允许一个客户端向文件写数据，即使其他客户端写入同一个文件的不同数据部分。
>
> 在它们的实现中，服务器一般授予单写者互斥锁来维护客户端缓冲的一致性，此时客户端可以读写并缓冲脏数据。当服务器撤销授权的互斥锁时，客户端会刷新并废除缓冲；当有多个客户端同时访问某个文件时，如果都为读访问，则服务器可以授予客户端共享锁，各个客户可以缓冲读取的数据，但不能脏写数据；如果存在写者，则服务器需要撤销所有被访问文件上的锁，清除客户端缓冲，文件I/O以Direct I/O的模式进行，所有的数据都要写透到服务器。

Ceph分布式文件锁保证了**多个客户端并发且细粒度的访问同一文件/目录，同时保证一致性、可靠性。**

ceph的分布式文件锁分为两部分：**客户端可见部分是caps**，**MDS可见部分是caps和各种lock。**每个类型的lock又有多种状态机。根据客户端的请求，lock转换状态，并和客户端同步caps信息，最终实现分布式锁。

客户端会发出caps消息请求锁，拿不到想要的caps会睡眠等待，这是典型的客户端行为设计。

MDS进行中心化管理锁，要想切换lock状态，必须在auth的mds进行（其他mds只是保留副本，方便客户端读取），这样通过消息传递把lock的中心化控制做在MDS。

注意：分布式锁与本地锁（mutex、spinlock）的区别

分布式锁只保证分布式缓存的一致性，并不保证MDS本地缓存的临界区的原子性（原子性由MDS的单线程和mutex锁保证）

```
Mutex        mds_lock;
```

Capabilities

caps，是一些二进制bits的集合。caps可以保证客户端中元数据缓存和数据缓存一致性，以及并发读写控制

下面是几个generic的capability bits，这些bits表示capability允许的能力，如下

```
/* generic cap bits */
#define CEPH_CAP_GSHARED     1  /* 2^0, client can reads (s) */
#define CEPH_CAP_GEXCL       2  /* 2^1, client can read and update (x) */   // 可以去掉？？
#define CEPH_CAP_GCACHE      4  /* 2^2, (file) client can cache reads (c) */
#define CEPH_CAP_GRD         8  /* 2^3, (file) client can read (r) */
#define CEPH_CAP_GWR        16  /* 2^4, (file) client can write (w) */
#define CEPH_CAP_GBUFFER    32  /* 2^5, (file) client can buffer writes (b) */
// 下面两个比较少见
#define CEPH_CAP_GWREXTEND  64  /* 2^6, (file) client can extend EOF (a) */
#define CEPH_CAP_GLAZYIO   128  /* 2^7, (file) client can perform lazy io (l) */
```

基本含义如下

cap bit	含义	作用
CEPH_CAP_GSHARED	客户端可以读元数据缓存	加速读元数据;
CEPH_CAP_GEXCL	独占;客户端可以读写元数据缓存	加速写元数据; 独占的客户端修改Inode时，采用异步的方式（类似数据的异步写），即只修改本地缓存就返回成功，由异步线程发送rpc到MDS。目前只有setattr用到，可以去掉？？？
CEPH_CAP_GCACHE	客户端可以cache	加速读数据; 与CEPH_CAP_GRD组合才有用;
CEPH_CAP_GRD	客户端可以读文件数据	读文件能力; 如果没有CEPH_CAP_GCACHE，则需要通过rpc，找osd拿数据; 如果有CEPH_CAP_GCACHE，则先命中缓存；没命中，再找osd拿
CEPH_CAP_GWR	客户端可以写文件数据	写文件能力; 如果没有CEPH_CAP_GBUFFER，则需要通过rpc，写数据到osd 如果有CEPH_CAP_GBUFFER，则直接写buffer，由writebehind线程去下刷脏数据
CEPH_CAP_GBUFFER	客户端可以buffer	加速写数据; 与CEPH_CAP_GWR组合才有用

MDS授予的capabilities非常细粒，将Inode的拆分成多段，多个客户端可以在同一个inode上拥有不同的capabilities。

将这些capability bits移动特定数量的位，就表示授予了inode's data或元数据的相应部分的能力。如下

```
/* 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 */
```

上面两者结合，可以得到一些常量，他们是通过将generic bits移动到相应的位上来生成的。例如

```
#define CEPH_CAP_AUTH_SHARED  (CEPH_CAP_GSHARED  << CEPH_CAP_SAUTH)
```

可以将这些bits组合在一起，形成一个表示一组capabilities的bitmask。所有的Capabilities的构成如下：

```
    +-0-+-1-+-2-+-3-+-4-+-5-+-6-+-7-+-8-+-9-+-10-+-11-+-12-+-13-+-14-+-15-+
    | p | _ |As   x |Ls   x |Xs   x |Fs   x   c    r    w    b    a    l  |
The second bit is currently unused.
```

有一个特殊的位:

```
#define CEPH_CAP_PIN  1  /* no specific capabilities beyond the pin */
```

pin只是将inode固定到内存中，而不授予任何其他caps。这足以让客户端获得inode number，以及设备inode中其他不可变的内容（major/minor numbers），或者symlink contents。

客户端和MDS的日志中可以提供capabilities的紧凑表达形式，例如：

```
pAsLsXsFs
'p'表示pin。每个大小字母对应于shift值，shift后的小写字母表示每个shift授予的实际
capabilities。
```

Abilities granted by each cap

授予的capabilities方式已经知道了，重要的是它们实际上允许客户端做什么：

Cap	含义	Ability
PIN	this just pins the inode into memory.	
AUTH	grants访问与身份验证相关的元数据的能力。特别是，owner、group和mode。Note：执行一个完全权限检查可能还需要获取xattr中的ACL。	As（CEPH_CAP_GSHARED << CEPH_CAP_SAUTH）表示该客户端有能力读取与认证相关的元数据
		Ax（CEPH_CAP_GEXCL << CEPH_CAP_SAUTH）: 表示该客户端有能力读取和更新与认证相关的元数据。
LINK	inode中的link	Ls（CEPH_CAP_GSHARED << CEPH_CAP_SLINK）: 表示客户端有能力读取inode中的link
		Lx（CEPH_CAP_GEXCL << CEPH_CAP_SLINK）: 表示该客户端有能力读取和更新inode中的link。
XATTR	能够访问或操作xattrs	Xs（CEPH_CAP_GSHARED << CEPH_CAP_SXATTR）
		Xx（CEPH_CAP_GEXCL << CEPH_CAP_SXATTR）
FILE	允许客户端访问和操作文件数据。它还涵盖了与文件数据相关的某些元数据 - 特别是size, mtime, atime, ctime。	Fs（CEPH_CAP_GSHARED << CEPH_CAP_SFILE）： 表示客户端有能力读取inode中的size,mtime等元数据。一旦客户端拥有它，所有其他客户端都被拒绝Fw。
		Fx（CEPH_CAP_GEXCL << CEPH_CAP_SFILE）： 只有loner client才允许此capability。一旦锁的状态转换成LOCK_EXCL，loner client将获得所有file capabilities除了Fl。
		Fr（CEPH_CAP_GRD << CEPH_CAP_SFILE）： 表示客户端有能力读取文件数据。一旦客户端拥有它，Fb capability将已经从所有其他客户端中revoked。如果客户端只请求读取文件，则锁状态将直接转换成LOCK_SYNC的稳定状态。所有客户端可以从auth MDS中获得Fscrl capabilities，从replica MDS获得Fscr capabilities。
		Fw（CEPH_CAP_GWR << CEPH_CAP_SFILE）： 表示客户端有能力写文件数据。如果多个客户端读取和写入同一文件，则锁状态最终将转换位LOCK_MIX稳定态，并且所有客户端都可以从auth MDS获取Frwl capabilities，从replica MDSes获取Fr。Fcb capabilities不会granted给客户端，客户端将执行sync read/write。

Cap	含义	Ability
		Fc（CEPH_CAP_GCACHE << CEPH_CAP_SFILE）：此capability意味着客户端可以缓存文件读取，并且应该与Fr capability 一起issued，只有在此用例中才有意义。虽然实际上在一些稳定或临时过渡状态下，他们倾向于允许保持Fc，即使Fr capability不被授予，因为这可以避免强制客户端删除完整缓存，例如在简单的文件大小扩展或截断用例上。
		Fb（CEPH_CAP_GBUFFER << CEPH_CAP_SFILE）：此capability 意味着客户端可以缓冲文件写入，并且应与 Fw capability 一起issued，只有在此用例中才有意义。虽然实际上在一些稳定或临时过渡状态下，他们倾向于允许保持Fb，即使没有授予Fw capability，因为这可以避免强制客户端drop脏缓冲区，例如在简单的文件大小扩展或截断用例上。
		Fl（CEPH_CAP_GLAZYIO << CEPH_CAP_SFILE）：此功能意味着客户端可以执行lazy io。lazy IO放松了POSIX的语义。即使文件由多个客户端上的多个应用程序打开，也允许缓冲读取/写入。应用程序负责管理缓存一致性本身。在内核客户端和fuse客户端已实现，面向的是HPC场景。

Cap结构

Client Cap 结构

每个Inode中都嵌入了一个Cap

```
struct Inode {
    ......
    // per-mds caps
    map<mds_rank_t, Cap*> caps;           // mds -> Cap
    Cap *auth_cap;
    ......
};
```

Cap的定义如下

```
struct Cap {
  MetaSession *session;
  Inode *inode;
  xlist<Cap*>::item cap_item;
  uint64_t cap_id;
  unsigned issued;              // MDS已给的
  unsigned implemented;         // ？？？
  unsigned wanted;              // as known to mds.自己想要的
  uint64_t seq, issue_seq;
  __u32 mseq;  // migration seq
  __u32 gen;
```

```
  UserPerm latest_perms;
  ......
};
```

MDS Cap 结构

每个CInode中记录了客户端已分配的cap

```
class CInode
{
    ...
    using cap_map = mempool::mds_co::map<client_t, Capability*>;
    cap_map client_caps;          // client -> caps
    ...
};
```

Capability的定义如下

```
class Capability {
    ...
    CInode *inode;
    client_t client;
    __u32 _wanted;           // what the client wants (ideally)
    __u32 _pending;          // 将要给客户端的caps
    __u32 _issued;           // 给过的caps集合
    ...
};
```

cap grant和cap revoke

cap grant

两种方式，第一种是通过client request的reply，回复给客户端

比如，MDS回复给客户端的getattr请求中带有cap信息，如下

```
/* capability issue, for bundling with mds reply */
struct ceph_mds_reply_cap {
    __le32 caps, wanted;           /* caps issued, wanted */
    __le64 cap_id;
    __le32 seq, mseq;
    __le64 realm;                  /* snap realm */
    __u8 flags;                    /* CEPH_CAP_FLAG_* */
} __attribute__ ((packed));
```

第二种是直接通过一个单独的caps消息传递，比如MDS更新客户端写的max_size，就通过MClientCaps消息传递

```
class MClientCaps : public Message {
public:
  struct ceph_mds_caps_head head;

  uint64_t size = 0;
  uint64_t max_size = 0;
  ......
}
```

cap revoke

大多数情况下，客户端收到revoke后，只需要更新caps就可，除了"Fc"和"Fb"（drop "Fc"需要释放缓存，drop "Fb"需要将脏的缓存下刷）。

```
// update caps
int revoked = old_caps & ~new_caps;             // 得到revoke的caps
if (revoked) {
    cap->issued = new_caps;
    cap->implemented |= new_caps;
    ...
    // revoke "Fb", 需要下刷掉脏数据
    if ((used & revoked & CEPH_CAP_FILE_BUFFER) && !_flush(in, new
C_Client_FlushComplete(this, in))) {
      // waitin' for flush
    } else if (revoked & CEPH_CAP_FILE_CACHE) {     // revoke "Fc", 需要等数据用
完，再丢掉缓存
      if (_release(in))
          check = true;
    } else {
      cap->wanted = 0;                      // don't let check_caps skip sending
a response to MDS
      check = true;
    }

    ......
}
```

caps valid机制

客户端通过session的心跳机制确保判断Inode中记录的caps是valid

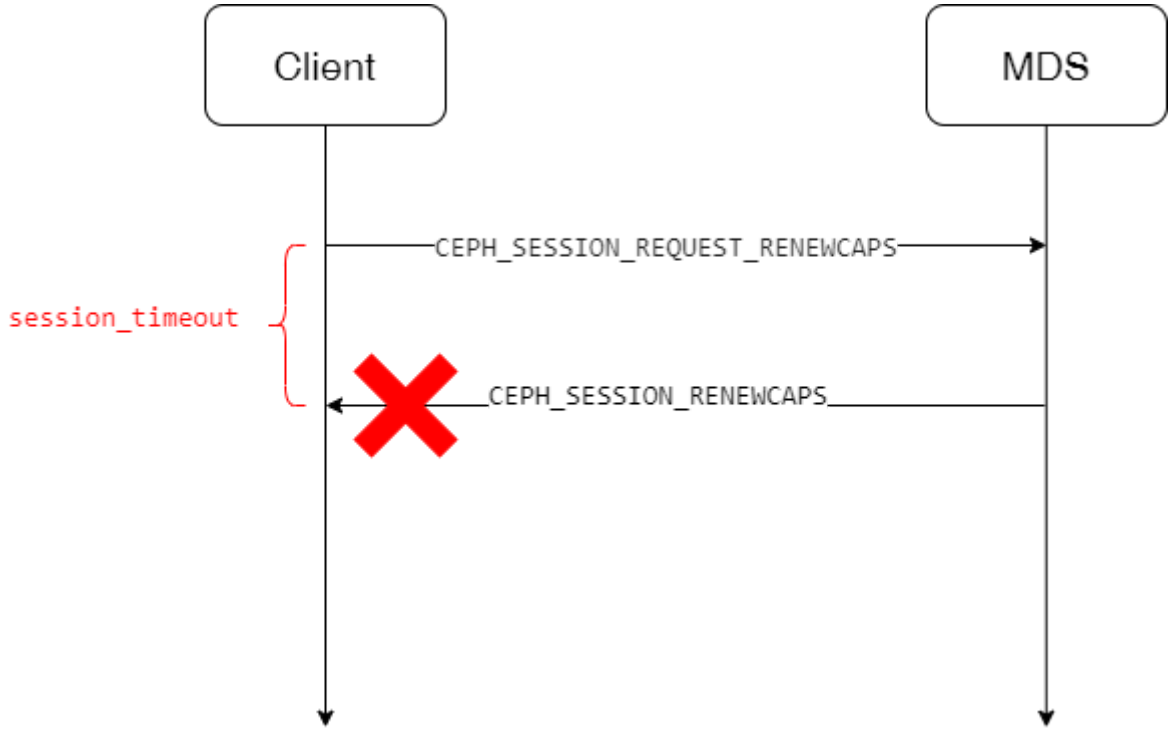
```
bool Inode::cap_is_valid(Cap* cap) const
{
  if ((cap->session->cap_gen <= cap->gen) &&
      (ceph_clock_now() < cap->session->cap_ttl)) {
    return true;
  }
  return false;
}
```

- **消息异常情况下，caps invalidate，心跳超时**

MDS回复心跳后，更新session->cap_ttl

```
void Client::handle_client_session(MClientSession *m)
{
    switch (m->get_op()) {
        ......
        case CEPH_SESSION_RENEWCAPS:
            if (session->cap_renew_seq == m->get_seq()) {
                session->cap_ttl = session->last_cap_renew_request + mdsmap-
>get_session_timeout();
                wake_inode_waiters(session);
            }
            break;
        ......
}
```

简单流程图如下



- **MDS主动invalidate client的caps**

正常情况下cap->gen == cap->session->cap_gen，当MDS主动发送session stale之后

```
void Client::handle_client_session(MClientSession *m)
{
    switch (m->get_op()) {
        ......
        case CEPH_SESSION_STALE:
            // invalidate session caps/leases
            session->cap_gen++;
            session->cap_ttl = ceph_clock_now();
            session->cap_ttl -= 1;
            renew_caps(session);
            break;
        ......
}
```

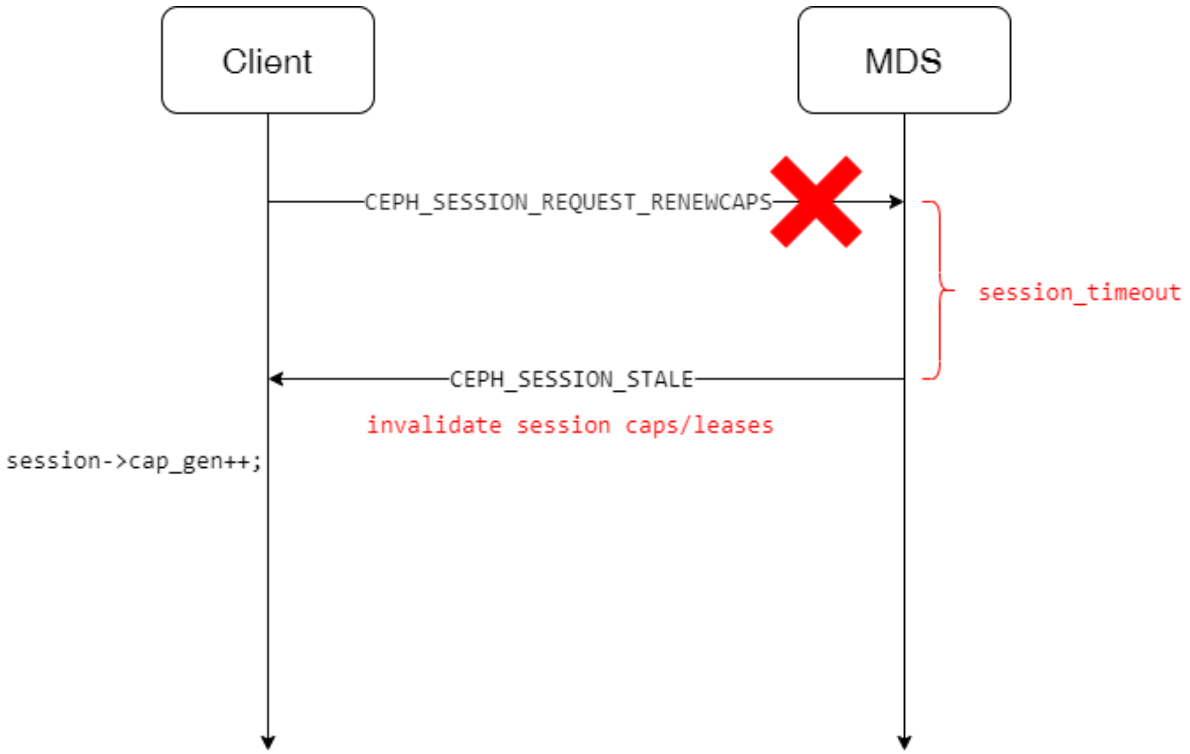
Note：出现这种情况后，客户端的所有的 Inode缓存不再有效，后续所有的请求都需要发往MDS，重新更新缓存。

> Client::handle_client_reply(MClientReply *reply)
>
> Client::insert_trace(MetaRequest *request, MetaSession *session)
>
> Client::add_update_inode
>
> Client::add_update_cap
>
> cap->gen = mds_session->cap_gen;

MDS发送session stale的条件

> MDSRankDispatcher::tick() // 定时器
>
> Server::find_idle_sessions()
>
> double queue_max_age = mds->get_dispatch_queue_max_age(ceph_clock_now());
>
> double cutoff = queue_max_age + mds->mdsmap->get_session_timeout();
>
> auto last_cap_renew_span = std::chrono::duration(now - session->last_cap_renew).count();
>
> 比较last_cap_renew_span 和cutoff

简单流程图如下



lock

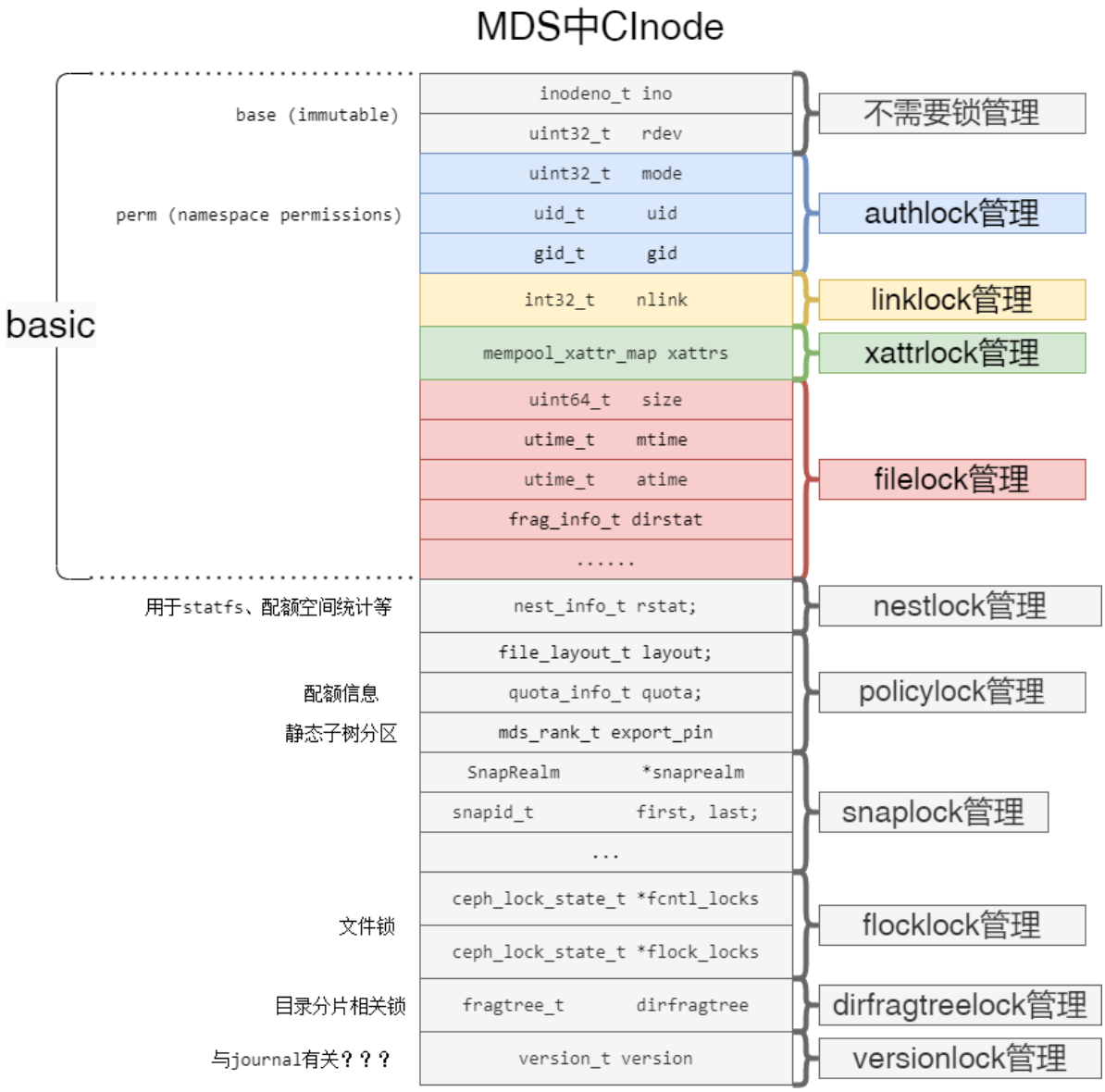
CInode lock

在MDS中，每个inode有4种不同类型的锁：sm_simplelock，sm_scatterlock，sm_filelock，sm_locallock，CInode中共有10把锁，如下

类型	锁	控制的粒度
sm_simplelock	SimpleLock authlock;	mode，uid，gid，ctime，
	SimpleLock linklock;	nlink
	SimpleLock xattrlock;	xattr
	SimpleLock snaplock;	
	SimpleLock flocklock;	
	SimpleLock policylock	layout等
sm_scatterlock	ScatterLock dirfragtreelock;	dirfragtree，目录分片
	ScatterLock nestlock;	rstat
sm_filelock	ScatterLock filelock;	mtime，atime，size，dirstat，dirfrags
sm_locallock	LocalLock versionlock;	version，？？？

锁保护的内容

元数据



数据

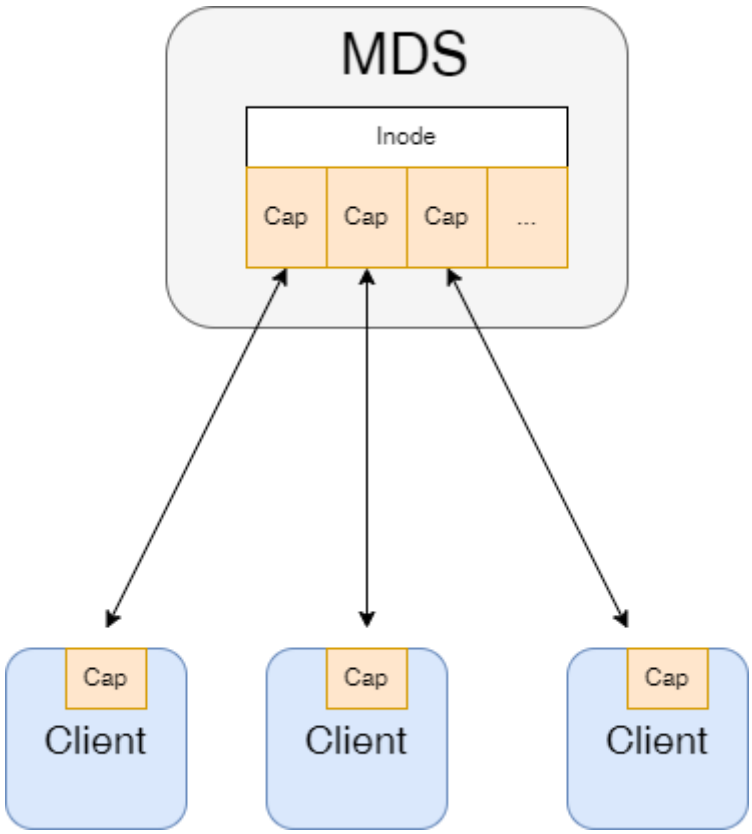
数据读写：由filelock来控制客户端的读写。

锁如何控制一致性

MDS进行中心化控制：以cap的方式来控制客户端的行为

原则：读共享；元数据读写互斥；数据缓存读写互斥；数据direct io共享

权限通过cap字段实现，每一种lock状态都对应着对应的cap



dirstat和rstat的定义如下

```
//记录该分片下的文件/目录数和mtime
struct frag_info_t {
    // this frag
    utime_t mtime;
    int64_t nfiles = 0;
    int64_t nsubdirs = 0;
}

//记录该分片以及嵌套子树的统计信息。
struct nest_info_t {
    // this frag + children
    utime_t rctime;
    int64_t rbytes = 0;
    int64_t rfiles = 0;
    int64_t rsubdirs = 0;
}
```

每种锁都有几个不同的锁状态机，MDS根据锁状态向客户端issue capabilities。

在每个锁状态下，MDS Locker将尝试向客户端issue能给的所有的capabilities，甚至客户端not needed or wanted某些capabilities。因为pre-issuing capabilities在某些情况下可以减少latency。

如果只有一个客户端，通常它将是所有inodes的loner client。在多个客户端的情况下，MDS将尝试对每个inode计算一个loner client，根据客户端的capabilities（needed | wanted），但是经常会失败。loner client将始终获得所有capabilities。

CDentry lock（待完善）

CDentry中有两把锁

```
SimpleLock lock; // FIXME referenced containers not in mempool
LocalLock versionlock; // FIXME referenced containers not in mempool
```

略

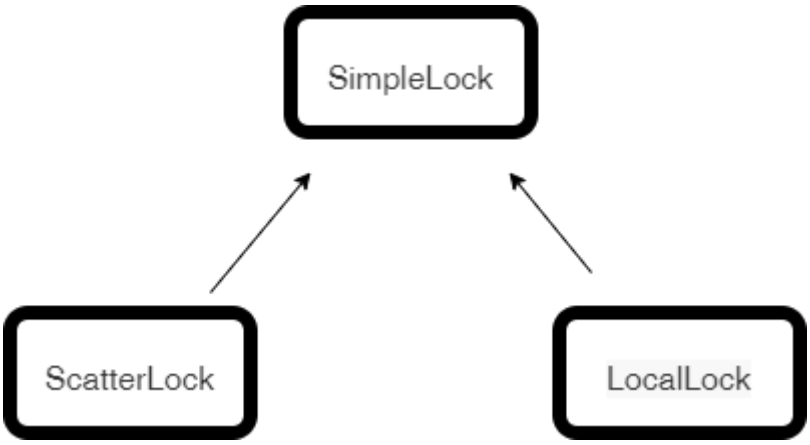
lease（待完善）

```
mempool::mds_co::map<client_t,ClientLease*> client_lease_map;

struct ClientLease {
  client_t client;
  MDSCacheObject *parent;
  ceph_seq_t seq = 0;
  utime_t ttl;
  xlist<ClientLease*>::item item_session_lease; // per-session list
  xlist<ClientLease*>::item item_lease;         // global list
};
```

锁结构

三种结构



ScatterLock

继承自SimpleLock，对比SimpleLock多了一个成员_more，其实就是链表节点。

```
  struct more_bits_t {
    xlist<ScatterLock*>::item item_updated;
    utime_t update_stamp;

    explicit more_bits_t(ScatterLock *lock) :
      item_updated(lock)
    {}
  };

  mutable std::unique_ptr<more_bits_t> _more;
```

锁的状态

常用状态：stable状态

锁状态	含义	caps	注释
LOCK_SYNC	同步状态	Inode：CEPH_CAP_GSHARED 数据：CEPH_CAP_GCACHE | CEPH_CAP_GRD	所有的分布式缓存都是一致的，可以随便读缓存。 所有请求都可以去加rdlock
LOCK_LOCK	写状态（多客户端操作）	无	MDS的缓存是最新的，所以不能给所有客户端缓存权限。 基本上这种状态只有wrlock才会用，sm_scatterlock和sm_filelock才会用这种状态。 sm_simplelock中LOCK_LOCK是在加xlock时，会转换成这种状态，但只是转换成xlock的中间一种状态，LOCK_XXXX --> LOCK_LOCK -->LOCK_XLOCK，疑问就是为啥要先切换成LOCK_LOCK,然后再往XLOCK转换？？？
LOCK_XLOCK（LOCK_XLOCKDONE）	写状态（单客户端操作）	无	与LOCK_LOCK类似，不能给其他客户端缓存权限，只能给xlocker CEPH_CAP_GSHARED权限。 目前只有在setattr时，会主动用到XLOCK；其他在涉及到nlink的修改时也会去对linklock去加XLOCK。
LOCK_EXCL	独占状态	所有权限	该客户端占用所有权限；其他客户端不能有任何权限
LOCK_MIX	混合读写状态	WR | RD	只适用于多客户端操作数据和副本元数据，

loner的定义

```
// 如果仅有一个client wanted中有wr，则它就是loner
client_t CInode::calc_ideal_loner()
{
  ...
  int n = 0;
  client_t loner = -1;
  for (map<client_t,Capability*>::iterator it = client_caps.begin(); it !=
client_caps.end(); ++it)
    if (!it->second->is_stale() &&
```

```
    ((it->second->wanted() &
(CEPH_CAP_ANY_WR|CEPH_CAP_FILE_WR|CEPH_CAP_FILE_RD)) || (inode.is_dir() &&
!has_subtree_root_dirfrag()))) {
      if (n)
          return -1;
      n++;
      loner = it->first;
    }
  return loner;
}
```

加锁

rdlock：共享读锁

- **满足加rdlock的条件**

是否可以加锁，需要根据当前锁的状态来判断的

```
bool can_rdlock(client_t client) const {
         // state == LOCK_SYNC，可以直接加rdlock
  return get_sm()->states[state].can_rdlock == ANY ||
      // state == LOCK_TSYN or LOCK_XSYN or LOCK_XSYN_SYNC，可以直接加rdlock，这几种
状态不常见。
      (get_sm()->states[state].can_rdlock == AUTH && parent->is_auth()) ||
      // state == LOCK_XLOCKDONE or LOCK_EXCL等，可以加rdlock，即独占的客户端可以加
rdlock
      (get_sm()->states[state].can_rdlock == XCL && client >= 0 &&
get_xlock_by_client() == client);
  }
```

- **条件满足**

加rdlock，其实就是增加num_rdlock引用计数

```
bool Locker::rdlock_start(SimpleLock *lock, MDRequestRef& mut, bool as_anon)
{
  ...
  while (1) {
    // can read?  grab ref.
    if (lock->can_rdlock(client)) {
      lock->get_rdlock();                              // 引用计数num_rdlock加1
      mut->rdlocks.insert(lock);
      mut->locks.insert(lock);
      return true;
    }
    ...
  }
```

- **条件不满足，进行锁切换**

需要去转换锁状态：**将锁状态LOCK_XXXX 转换成LOCK_SYNC**

> rdlock_start
>
> _rdlock_kick
>
> simple_sync

一般转换过程：LOCK_XXXX --> LOCK_XXXX_SYNC --> LOCK_SYNC。

切换成LOCK_XXXX_SYNC时，根据该状态对应的caps去回收不允许的caps（导致不一致的caps，如Fb，Fw，Fx等）。

锁切换逻辑

1，先将锁切换成LOCK_XXXX_SYNC，unstable状态

2，这里有一个概念叫gather，即收集。

如果锁已经加了wrlock、则需要gather wrlock，也称回收wrlock

如果锁需要回收caps，则需要gather caps。

如果不能切换成LOCK_SYNC状态，加入等待队列

```
wait_on = SimpleLock::WAIT_STABLE;
lock->add_waiter(wait_on, new C_MDS_RetryRequest(mdcache, mut));
```

锁唤醒

1，wrlock解锁

```
void Locker::wrlock_finish(SimpleLock *lock, MutationImpl *mut, bool
*pneed_issue)
{
```

```
    ...
    lock->put_wrlock();
    ...
    if (!lock->is_wrlocked()) {                                  // 如果锁已经没有被
wrlock
        if (!lock->is_stable())                                  // 如果不是stable状态，
说明需要评估下是否gather完成。
            eval_gather(lock, false, pneed_issue);
        else if (lock->get_parent()->is_auth())
            try_eval(lock, pneed_issue);
  }
}

Locker::wrlock_finish
    eval_gather(lock, false, pneed_issue)
        如果还有caps没有回收完，则啥都不做
        如果caps已经回收完，即没有revoke的caps
            lock->set_state(next);              // 设置为LOCK_SYNC
            lock-
>finish_waiters(SimpleLock::WAIT_STABLE|SimpleLock::WAIT_WR|SimpleLock::WAIT_RD|
SimpleLock::WAIT_XLOCK); // 把等待的请求重新拿出来，再执行一遍
            try_eval(lock, &need_issue);        // 尝试评估下，看是否需要issue caps，
即给与之前未给的caps（caps grant），或回收caps
                eval(lock, pneed_issue);        // 评估
                    simple_eval(lock, need_issue);  // 要么切换成excl，要么切换成
sync；只要进行锁的切换，都需要issue caps
```

2，caps回收后，需要评估所有caps对应的锁

```
Locker::handle_client_caps
    eval(in, CEPH_CAP_LOCKS);               // 评估cap的lock
        eval_any(&in->filelock
            // 如果此刻锁不是stable的
            eval_gather(lock, first, need_issue, pfinishers)
                 // 如果还有wrlock，则啥都不做
                 // 如果没有wrlock
                    lock->set_state(next);              // 设置为LOCK_SYNC
                    lock-
>finish_waiters(SimpleLock::WAIT_STABLE|SimpleLock::WAIT_WR|SimpleLock::WAIT_RD|
SimpleLock::WAIT_XLOCK); // 把等待的请求重新拿出来，再执行一遍
        eval_any(&in->authlock
        eval_any(&in->linklock
        eval_any(&in->xattrlock
```

- **rdlock成功**

锁状态通常为LOCK_SYNC，允许的操作权限：pAsLsXsFscr（文件），pAsLsXsFs（目录）

- **rdlock失败**

---

client的request会放入等待队列，等待唤醒。

如果有副本

存在多个MDS都有元数据缓存时，需要保证所有节点的缓存都需要加锁。由auth的MDS，通过RPC来控制锁的状态切换。

当前MDS是auth

```
send_lock_message(lock, LOCK_AC_SYNC, data);    // 本地切换成LOCK_SYNC后，发给所有副本的MDS去切换锁
```

当前MDS是副本：

```
mds->send_message_mds(new MLock(lock, LOCK_AC_REQRDLOCK, mds->get_nodeid()), auth);
```

wrlock : 共享写锁

只有filelock和nestlock才可以加wrlock

- **满足加wrlock的条件**

是否可以加锁，需要根据当前锁的状态来判断的

```
bool can_wrlock(client_t client) const {
   // state == LOCK_MIX，可以直接加wrlock
   return get_sm()->states[state].can_wrlock == ANY ||
    // state == LOCK_LOCK，可以直接加wrlock
      (get_sm()->states[state].can_wrlock == AUTH && parent->is_auth()) ||
    // state == LOCK_EXCL or LOCK_EXCL_MIX等，可以直接加wrlock，即独占的客户端可以加wrlock
      (get_sm()->states[state].can_wrlock == XCL && client >= 0 && (get_xlock_by_client() == client ||
                                    get_excl_client() == client));
  }
```

- **条件满足**

加wrlock，其实就是增加num_wrlock引用计数

```
bool Locker::wrlock_start(SimpleLock *lock, MDRequestRef& mut, bool nowait)
{
  ...
  while (1) {
    // wrlock?
    if (lock->can_wrlock(client) && (!want_scatter || lock->get_state() ==
LOCK_MIX)) {
      lock->get_wrlock();                       // num_wrlock引用计数加1
      mut->wrlocks.insert(lock);
      mut->locks.insert(lock);
      return true;
    }
   ...
 }
```

- **条件不满足，进行锁切换**

需要去转换锁状态：**将锁状态LOCK_XXXX 转换成LOCK_LOCK/LOCK_MIX**

```
if (want_scatter)
    scatter_mix(static_cast<ScatterLock*>(lock));       // lock-
>set_state(LOCK_MIX);
else
    simple_lock(lock);                                  // lock-
>set_state(LOCK_LOCK)
```

want scatter的意义、逻辑？？ 简单来说，有副本就需要切换成LOCK_MIX

如果不能切换成功，加入等待队列

```
lock->add_waiter(SimpleLock::WAIT_STABLE, new C_MDS_RetryRequest(mdcache, mut));
```

- **simple_lock: 锁切换成LOCK_LOCK**

锁切换逻辑

1，先将锁切换成LOCK_XXXX_LOCK

2，然后gather

如果锁已经加了rdlock，则需要gather rdlock

如果锁需要回收caps，则需要gather caps。

锁唤醒

1，rdlock解锁

```
void Locker::rdlock_finish(SimpleLock *lock, MutationImpl *mut, bool
*pneed_issue)
{
  // drop ref
  lock->put_rdlock();
  ...
  // last one?
  if (!lock->is_rdlocked()) {                    // 如果锁已经没有被rdlock
    if (!lock->is_stable())                      // 如果不是stable状态，说明需要评估下
是否gather完成。
      eval_gather(lock, false, pneed_issue);
    else if (lock->get_parent()->is_auth())
      try_eval(lock, pneed_issue);
  }
}

//后面的流程跟共享流程一样
```

2，caps回收后，需要评估所有caps对应的锁

- **scatter_mix: 锁切换成LOCK_MIX**

1, 如果锁已经是LOCK_LOCK，则直接转换成LOCK_MIX，并通知副本去切换成LOCK_MIX

2, 将锁切换成LOCK_XXXX_MIX

3, 进行gather

如果锁已经加了rdlock，则需要gather rdlock

如果锁需要回收caps，则需要gather caps

锁唤醒同上

- **wrlock成功**

锁状态为LOCK_LOCK， 允许的操作权限：Fcb

锁状态为LOCK_MIX， 允许的操作权限：Frw

- **wrlock失败**

client的request会放入等待队列，等待唤醒。

如果有副本

如果auth，

```
send_lock_message(lock, LOCK_AC_LOCK);                    // 发给所有有副本的MDS
```

如果是副本：

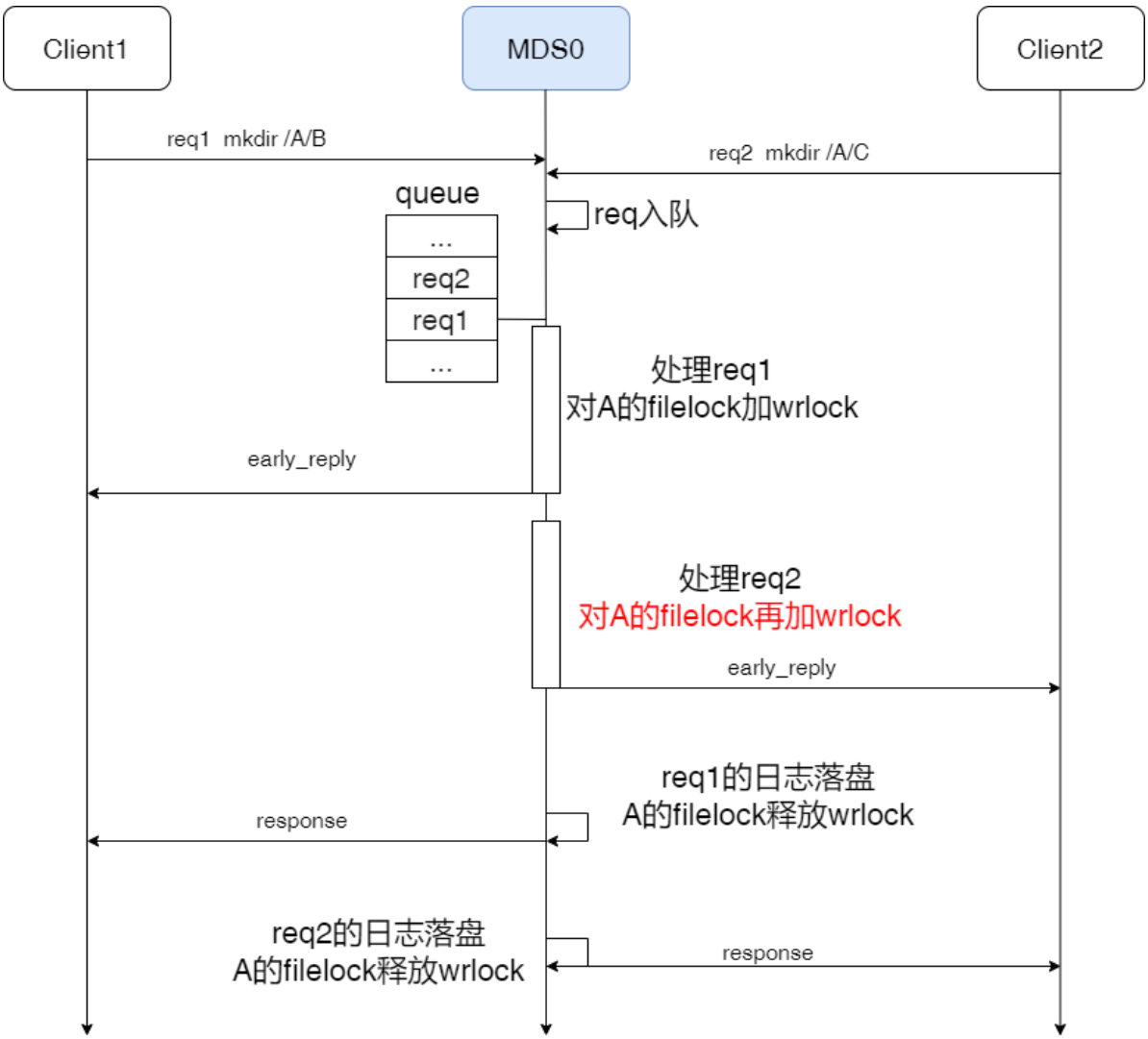
```
mds->send_message_mds(new MLock(lock, LOCK_AC_REQSCATTER, mds->get_nodeid()),
auth)
```

- **共享写锁的含义**

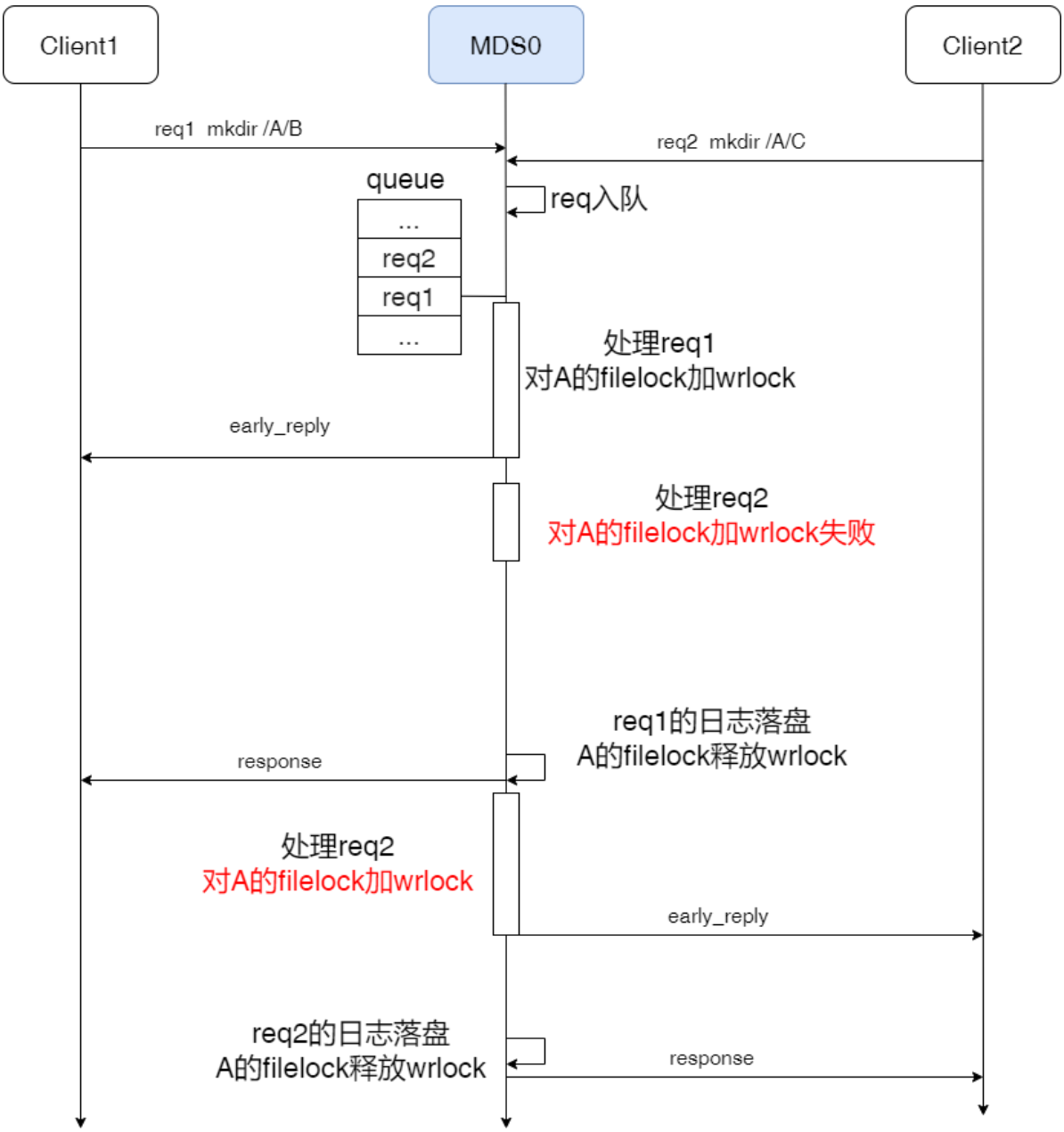
通常意义上的写锁是独占的，即加了写锁后，再加写锁会阻塞，直到先前加的写锁被释放。

ceph里面的写锁，是共享的。这并不意味着多线程可以同时修改共享数据，这是由ceph的线程模型和业务模型决定的。

ceph是单线程处理请求，并且为了加速元数据的写操作，采用了二次回复机制（early_reply和response），举个例子。



作为对比，如果将wrlock设置为独占的，那么例子如下



很明显，将wrlock设置为独占的，会导致多客户端并发性能下降。

xlock：独占锁

- **满足加xlock的条件**

```
bool can_xlock(client_t client) const {
  // state == LOCK_PREXLOCK or LOCK_LOCK，可以直接加xlock
  return get_sm()->states[state].can_xlock == ANY ||
    // 特殊情况，只有locallock可以加锁
    (get_sm()->states[state].can_xlock == AUTH && parent->is_auth()) ||
    // state == LOCK_LOCK_XLOCK，可以直接加xlock
    (get_sm()->states[state].can_xlock == XCL && client >= 0 &&
get_xlock_by_client() == client);
  }
```

- **条件满足**

设置状态为LOCK_XLOCK，并记录请求加锁的client

```
bool Locker::xlock_start(SimpleLock *lock, MDRequestRef& mut)
{
  ...
  // auth?
  if (lock->get_parent()->is_auth()) {
    // auth
    while (1) {
      if (lock->can_xlock(client)) {
          lock->set_state(LOCK_XLOCK);          // 设置为LOCK_XLOCK
          lock->get_xlock(mut, client);         // 记录获得xlock的客户端
          mut->xlocks.insert(lock);
          mut->locks.insert(lock);
          mut->finish_locking(lock);
          return true;
      }
     ...
 }
```

- **条件不满足，进行锁切换**

需要去转换锁状态：**将锁状态LOCK_XXXX 转换成LOCK_LOCK/LOCK_PREXLOCK**

```
if (lock->get_state() == LOCK_LOCK || lock->get_state() == LOCK_XLOCKDONE) {
    mut->start_locking(lock);
    simple_xlock(lock);             // 将锁转换成LOCK_PREXLOCK
} else {
    simple_lock(lock);              // 将锁转换成LOCK_LOCK
}
```

例如：LOCK_SYNC --> LOCK_SYNC_LOCK --> LOCK_LOCK --> LOCK_LOCK_XLOCK --> LOCK_PREXLOCK --> LOCK_XLOCK

加锁失败，放入等待队列

```
lock->add_waiter(SimpleLock::WAIT_WR|SimpleLock::WAIT_STABLE, new
C_MDS_RetryRequest(mdcache, mut));
```

simple_lock已分析

simple_xlock（比较特殊）

1，先将锁切换成LOCK_LOCK_XLOCK

2，gather

如果锁已经加了rdlock，则需要gather rdlock

如果锁已经加了wrlock，则需要gather wrlock

如果锁需要回收caps，则需要gather caps。

锁唤醒同上述流程

- **xlock成功**

锁的状态为LOCK_XLOCKDONE，能给的权限为CEPH_CAP_GSHARED

- **xlock失败**

client的request会放入等待队列，等待唤醒。

- **如果有副本**

如果是auth，且有副本时

```
send_lock_message(lock, LOCK_AC_LOCK);                    // 发给所有有副本的MDS
```

如果是副本：

```
MMDSSlaveRequest *r = new MMDSSlaveRequest(mut->reqid, mut->attempt,
MMDSSlaveRequest::OP_XLOCK);
mds->send_message_mds(r, auth);
```

posix操作涉及的MDS锁操作

CInode锁操作

posix 操作	**加锁类型**	**锁**	**注释**	**锁目标状态**	**cap revoke（可能）**	**cap grant**	**疑问**
getattr	rdlock	authlock、linklock、xattrlock、filelock	需要读取inode的所有域	LOCK_SYNC	Ax、Lx、Xx、Fxwb	As、Ls、Xs、Fscr	--
lookup	同上	同上	同上	同上	同上	同上	--
open rw（单客户端）	无	--	open操作并没有修改元数据，所以不需要加锁	authlock: LOCK_EXCL xattrlock: LOCK_EXCL filelock: LOCK_EXCL nestlock: LOCK_LOCK	As、Xs、Fscr	Asx、Xsx、Fsxcrwb	为啥需要Ax、Xx权限？？？
open rw（多客户端）	无	--	open操作并没有修改元数据，所以不需要加锁	authlock: LOCK_SYNC xattrlock:LOCK_SYNC filelock:LOCK_MIX nestlock: LOCK_LOCK	Ax、Xx、Fscb	As、Xs、Frw	--
read	无	--	如果没有Fr权限，则阻塞等待； 如果没有Fc权限，则直接通过rpc读数据，如果有Fc，则读objectcache	--	--	--	--
write	无	--	如果没有Fw权限，则阻塞等待； 如果没有Fb权限，则直接通过rpc写数据，如果有Fb，则写objectcache。	--	--	--	--
readdir	rdlock	filelock	需要读取目录下的数据	LOCK_SYNC	Fx	Fs	--
		dirfragtreelock	需要读取dirfragtree	LOCK_SYNC	不涉及	不涉及	--
mkdir (mkdir /A/B)	rdlock	authlock（"A"）	需要读取父目录的perm，构造子目录的perm信息	LOCK_SYNC	Ax	As	--
	wrlock	filelock（"A"）	需要修改父目录("A")的mtime，dirstat等	LOCK_LOCK	Fs	--	--
		nestlock（"A"）	需要修改父目录("A")的rstat等	LOCK_LOCK	--	--	--
create	同上	同上	同上	同上	同上	同上	--
mknod	同上	同上	同上	同上	同上	同上	--
setattr	**xlock**	**authlock**	**修改mode、gid、uid、btime**	**LOCK_XLOCKDONE**	**As（其他客户端）**	**As（xlocker客户端）**	
		filelock	修改mtime、atime、size	LOCK_XLOCKDONE	Fs	Fcb	
	wrlock	versionlock	？？？				
setxattr	xlock	xattrlock	修改xattr	LOCK_XLOCKDONE	**Xs（其他客户端）**	**Xs（xlocker客户端）**	
unlink (unlink /A/B)	rdlock	filelock（"B"）	to verify it's empty。 需要根据dirstat，来判断目录是否为空，如果目录不为空，返ENOTEMPTY	LOCK_SYNC	Fx	Fs	--
	wrlock	filelock（"A"）	修改父目录("A")的mtime、dirstat等	LOCK_SYNC	Fs	--	--
		nestlock（"A"）	需要修改父目录("A")的rstat等	LOCK_LOCK	--	--	--

posix 操作	加锁类型	锁	注释	锁目标状态	cap revoke（可能）	cap grant	疑问
		stray目录的filelock	修改stray目录的mtime、dirstat	**暂不研究**	--	--	--
		stray目录的nestlock	修改stray目录的rstat	**暂不研究**	--	--	--
	xlock	**linklock（"B"）**	**nlink--**	**LOCK_XLOCKDONE**	**Ls（其他客户端）**	**Ls（xlocker客户端）**	**--**
link 硬链接（ln /test/testfile /test/testfile-ln）	wrlock	filelock（父目录"test"）	修改父目录"test"的mtime、dirstat	LOCK_LOCK	Fs	--	--
		nestlock（父目录"test"）	修改父目录"test"的rstat	LOCK_LOCK	--	--	--
	xlock	**linklock（"testfile"）**	**nlink++**	**LOCK_XLOCKDONE**	**Ls（其他客户端）**	**Ls（xlocker客户端）**	**--**
symlink 软链接（ln -s /test/testfile /test/testfile-ln）	rdlock	authlock（父目录"test"）	需要读取父目录的perm，构造子目录的perm信息	LOCK_SYNC	Ax	As	--
	wrlock	filelock（父目录"test"）	修改"test"的mtime、dirstat	LOCK_LOCK	Fs	--	--
		nestlock（父目录"test"）	修改"test"的rstat	LOCK_LOCK	--	--	--
rename （rename /testA/A /testB/B ）	wrlock	filelock（父目录"testB"）	修改"testB"的mtime、dirstat	LOCK_LOCK	Fs	--	--
		nestlock（父目录"testB"）	修改"testB"的rstat	LOCK_LOCK	--	--	--
	remote_wrlock	filelock（父目录"testA"）	修改"testA"的mtime、dirstat	LOCK_LOCK	Fs	--	--
		nestlock（父目录"testA"）	修改"testA"的rstat	LOCK_LOCK	--	--	--
	xlock	**linklock**	**nlink--**	**LOCK_XLOCKDONE**	**Ls（其他客户端）**	**Ls（xlocker客户端）**	**--**

解锁

第一次回复后，释放rdlock，对之前的请求进行cap revoke | cap grant（可能）

第二次回复后，释放所有锁，cap revoke | cap grant

> Server::reply_client_request
>
> MDCache::request_finish
>
> MDCache::request_cleanup
>
> MDCache::request_drop_locks
>
> Locker::drop_locks
>
> Locker::_drop_rdlocks

```
Locker::rdlock_finish

  lock->put_rdlock();
```