curvefs client 删除文件和目录功能设计

背景

目前curvefs client版本对删除unlink和rmdir的设计只有简单的删除inode和dentry结构，遗留了nlink和lookup count相关的内容还未实现，是不完备的。本文首先调研moosefs，chubaofs等分布式系统，参考并设计解决上述遗留问题。

当前删除接口代码如下：

```
CURVEFS_ERROR FuseClient::RemoveNode(fuse_req_t req, fuse_ino_t parent,
    const char *name) {
    Dentry dentry;
    CURVEFS_ERROR ret = dentryManager_->GetDentry(parent, name, &dentry);
    if (ret != CURVEFS_ERROR::OK) {
        LOG(ERROR) << "dentryManager_ GetDentry fail, ret = " << ret
                   << ", parent = " << parent
                   << ", name = " << name;
        return ret;
    }
    ret = dentryManager_->DeleteDentry(parent, name);
    if (ret != CURVEFS_ERROR::OK) {
        LOG(ERROR) << "dentryManager_ DeleteDentry fail, ret = " << ret
                   << ", parent = " << parent
                   << ", name = " << name;
        return ret;
    }
    // TODO(xuchaojie) : judge can inode be deleted
    ret = inodeManager_->DeleteInode(dentry.inodeid());
    if (ret != CURVEFS_ERROR::OK) {
        LOG(ERROR) << "inodeManager_ DeleteInode fail, ret = " << ret
                   << ", parent = " << parent
                   << ", name = " << name
                   << ", inode = " << dentry.inodeid();
        return ret;
    }
    return ret;
}
```

存在两个问题:

一是删除时nlink字段未考虑:

文件的nlink用于实现hard link。 hard link使用nlink字段表示文件的link的引用计数，第一次创建文件是nlink字段为1。每创建一个新的指向该文件的hard link时,nlink字段+1， 每删除一个hard link或指向的原文件时，nlink字段-1。

当nlink字段减到0时，才真正删除inode。所以在实现unlink接口或rmdir接口时，需要判断unlink字段的当前值，当nlink字段大于1时，只减nlink字段就可以了，当nlink字段减到0时，才真正的执行删除inode。

目录的nlink字段与文件的nlink字段不同，目录的nlink字段初始值为2， 并且在目录下，每创建一个新目录，nlink字段也会+1， 删除目录nlink相应的减1。

目录不支持硬链接。

二是删除时lookup count未考虑：

lookup count 指的是文件的访问计数。当文件/目录被打开时，即使文件/目录已经被另一个进程删除了（nlink==0），该文件/目录仍然可以被打开的进程访问，不会造成崩溃或报错，我们的curvefs也需要实现这样的语义。

这部分内容在fuse的相关接口中也有描述如下：

```
/**
* Forget about an inode
*
* This function is called when the kernel removes an inode
* from its internal caches.
*
* The inode's lookup count increases by one for every call to
* fuse_reply_entry and fuse_reply_create. The nlookup parameter
* indicates by how much the lookup count should be decreased.
*
* Inodes with a non-zero lookup count may receive request from
* the kernel even after calls to unlink, rmdir or (when
* overwriting an existing file) rename. Filesystems must handle
* such requests properly and it is recommended to defer removal
* of the inode until the lookup count reaches zero. Calls to
* unlink, rmdir or rename will be followed closely by forget
* unless the file or directory is open, in which case the
* kernel issues forget only after the release or releasedir
* calls.
*
* Note that if a file system will be exported over NFS the
* inodes lifetime must extend even beyond forget. See the
* generation field in struct fuse_entry_param above.
*
* On unmount the lookup count for all inodes implicitly drops
* to zero. It is not guaranteed that the file system will
* receive corresponding forget messages for the affected
```

```
 * inodes.
 *
 * Valid replies:
 *   fuse_reply_none
 *
 * @param req request handle
 * @param ino the inode number
 * @param nlookup the number of lookups to forget
 */
void (*forget) (fuse_req_t req, fuse_ino_t ino, uint64_t nlookup);


/**
 * Remove a file
 *
 * If the file's inode's lookup count is non-zero, the file
 * system is expected to postpone any removal of the inode
 * until the lookup count reaches zero (see description of the
 * forget function).
 *
 * Valid replies:
 *   fuse_reply_err
 *
 * @param req request handle
 * @param parent inode number of the parent directory
 * @param name to remove
 */
void (*unlink) (fuse_req_t req, fuse_ino_t parent, const char *name);


/**
 * Remove a directory
 *
 * If the directory's inode's lookup count is non-zero, the
 * file system is expected to postpone any removal of the
 * inode until the lookup count reaches zero (see description
 * of the forget function).
 *
```

```
 * Valid replies:
 *   fuse_reply_err
 *
 * @param req request handle
 * @param parent inode number of the parent directory
 * @param name to remove
```

```
 */
void (*rmdir) (fuse_req_t req, fuse_ino_t parent, const char *name);
```

其中的注释内容总结如下：

- 当lookup count在fuse_reply_entry和fuse_reply_create时增加1
- 当内核移除其inode cache时，会调用forget，此时lookup count需要减nlookup（forget的参数）
- 当umount时，所有lookup count减至0
- 不应该完全依赖forget接口去实现inode的移除，因为forget接口可能不会被内核调用（例如client崩溃）

相关调研

moosefs

1. moosefs 未对接forget
2. moosefs 实现了在mds上open，因此删除时可以判断文件是否被打开
3. moosefs使用了两种机制，来实现上述功能，分别是trash机制和reserve机制（最新版本叫sustained），两种机制如下：

trash机制：

- 对于所有TYPE_FILE类型的文件在删除时，若其trashtime大于0，则不会立即将该文件彻底删除，而是将其类型修改为TYPE_TRASH并且将该节点从文件树移除然后放到trash链表中表示该文件已经进入回收站。
- 通过META文件系统来访问trash
- 通过trash机制，可实现文件的恢复UNDEL
- 回收站实现了一个timer，定期判断trashtime，执行定期清理回收站
- 清理时，当文件仍处于打开状态，则还需要进入下sustained/reserve中。

sustained机制/reserve机制

- 当一个trashtime等于0的TYPE_FILE类型的文件被一个客户端正在打开，而同时有另一个客户端要删除它时，此时master对该文件节点的处理是并不立即删除该文件而是设置为TYPE_RESERVED类型并将该fsnode连接到reserved链表中，使该文件虽然已经从文件树中删除掉，但因为另一个正在打开该文件的客户端因为持有该节点inodeid,所以不影响它对该文件的读写操作，当所有客户端都关闭该文件后，该文件节点才会从reserve被清除。
- 使用了session机制，记录client端的open状态
- 通过META文件系统访问reserve
- 使用CUTOMA_FUSE_RESERVED_INODES消息保持和释放inode
- 实现了Timer，定期判断是否还有session，如果没有client打开，则进行清理。

优点：

1. 通过meta文件系统来管理trash，更为优雅。

缺点：

1. moosefs是单mds，所以不存在接口原子性的问题，这块要重新考虑，我们实现上会比moosefs复杂，需要引入一些额外的复杂性。
2. 由于是按目录管理trash，那么必须是两个trash（其中一个是reserve）以区分两种不同的情况。

chubaofs

chubaofs的方案如下：

- chubaofs实现了类似trash的机制，称为freelist，
  当inode被unlink时，client会发送UnlinkInodeRequest， 对应的metasever接收到请求后，如果是文件，使得nlink计数减1，如果减到0，则将inodeid放到freelist中。
- inode在freelist中存放7天，以应对有文件被打开的情况。
- 如果nlink减到0的是目录，则直接移除，不需要放到freelist中，目录由于是nlink从2开始，当目录的nlink=2时，连续减两次到0。
- freelist会被定期清理，清理时筛选出超过7天的inodeid，将其从inode tree和free list中移除。
- chubaofs中实现了forget接口：首先client端，在删除inode时，如果判断到nlink减到0，则加入client端维护的OrphanInodeList。forget接口执行时，判断inode是否在OrphanInodeList中，如果在OrphanInodeList中，则向metaserver发送EvictInodeRequest， metaserver在收到该请求后，则设置inode的DeleteMarkFlag，并将其放入freelist。
  freelist在定期清理时，当发现设置了DeleteMarkFlag，则直接从inode tree和free list中移除该inode，不再等待7天。
- chubaofs实现了强制从freelist中移除inode的机制，同样是使用设置DeleteMarkFlag的方式。
- chubaofs也实现了查询机制，来查询处于freelist当中的inode的情况，以便与运维，这一部分没有细看。

优点：

1. 实现简单，开发代价小，且后续可以增加metaserver端打开(session)等机制，向着moosefs的演进也是可以的。
2. 我们的整个架构设计本身就类似chubao方式，这个方案本身是chubaofs的成熟方案，说明是已经被验证过是可行的方案。

缺点：

1. 由于link、unlink等接口涉及跨服务器的两个请求的处理，可能会存在孤儿inode的问题，这一情况，chubaofs是通过运维手段去修复，见遗留问题。moosefs由于单mds，不存在这个问题。

方案设计思考

首先我们可以确定以下几个设计点：

1. 删除的大致过程如下，首先移除dentry，然后移除inode，可以容忍只存在inode，也就是孤儿inode情况，这部分内容见下面遗留问题。
2. 必须要实现（至少)一个trash机制，以作为回收站，不论是后续做UNDEL，还是应对打开的文件被其他进程删除的情况。
3. 必须实现某种机制，可以查看清理trash中的inode。
4. 孤儿节点只能在metaserver去定期清理，不会在client端，因为client会崩溃，也可能下线了，永远不再起来。所以实际的内存和外存中的inode的删除机制，必须是在metaserver中实现的。client端只是进行nlink-1的操作。
5. 不能完全依赖forget接口的调用来移除inode，因为client可能会崩溃，也可能下线。所以实际移除inode只能依赖于metaserver，两种方式：chubaofs的简单粗暴放7天就删，或者moosefs使用session机制来维护inode是否被打开。所以从这一点来看，forget接口可以先不实现。

除此之外，还有以下几个问题需要解决：

1.Trash机制是实现1个(类似chubaofs)，还是2个（类似moosefs）？

- moosefs中reseved中的inode数量一般来说不会很多，因为打开文件被另一个进程删除的场景应该不会太多，所以，考虑只实现一个trash就可以了，但是trash中应当有机制可以区分两种情况，比如增加一些flag，以便于使用查看。
- moosefs使用2个trash的原因可能是使用不同目录的方式来区分这两种inode，如果放在一个目录下，那么就难以区分两者。

2. Trash放在哪里？

Trash放在哪里的问题可能有以下几种方案：

第一种方案：

- Trash中只存放inode id， inode结构仍然在原地。由于inode放在原地，那么需要实现类似freelist一样的东西来保存当前已经"被删"的inode id，以便于扫描进程清理到期的inode。
- 由于inode放在原地，那么由于dentry已经被删除，那么查询工具就较为复杂，不能复用原有的client逻辑，需要组织成moosefs那样的meta文件系统可能需要引入额外的复杂性，但是依然可以实现简单的工具查询。
- 由于该方案，删除的inode是分散于每个partition中，那么查询工具可能需要遍历所有partion去查询所有的删除inode。

第二种方案：

- 将inode移动到隐藏的trash目录，这个trash目录可以是实际的目录结构，有dentry和inode，并遵循当前inode和dentry的放置方式（inode按照inodeid分布，dentry按照parentid分布）
- 这种方案的优点是便于工具对trash进行查询，毕竟是实际的目录结构，完全遵循文件系统，可能可以复用client的当前设计，甚至可以参考moosefs实现一个meta文件系统来管理，更为优雅。
- 但是缺点是DEL和UNDEL需要在trash下创建和删除dentry，这部分处理会引入额外的复杂性。（这个过程其实类似于rename）
- 由于moose是单文件系统，对于我们实现多文件系统，这里还有两种方案： 一是使用全局唯一的trash，二是每个fs一个trash，并且trash不能放在fs的根目录下，因为存在跟用户的目录重名的问题。

倾向于使用方案1，各方面实现上较为简单，异常处理不会很复杂，查询工具可以先实现一个简单的。

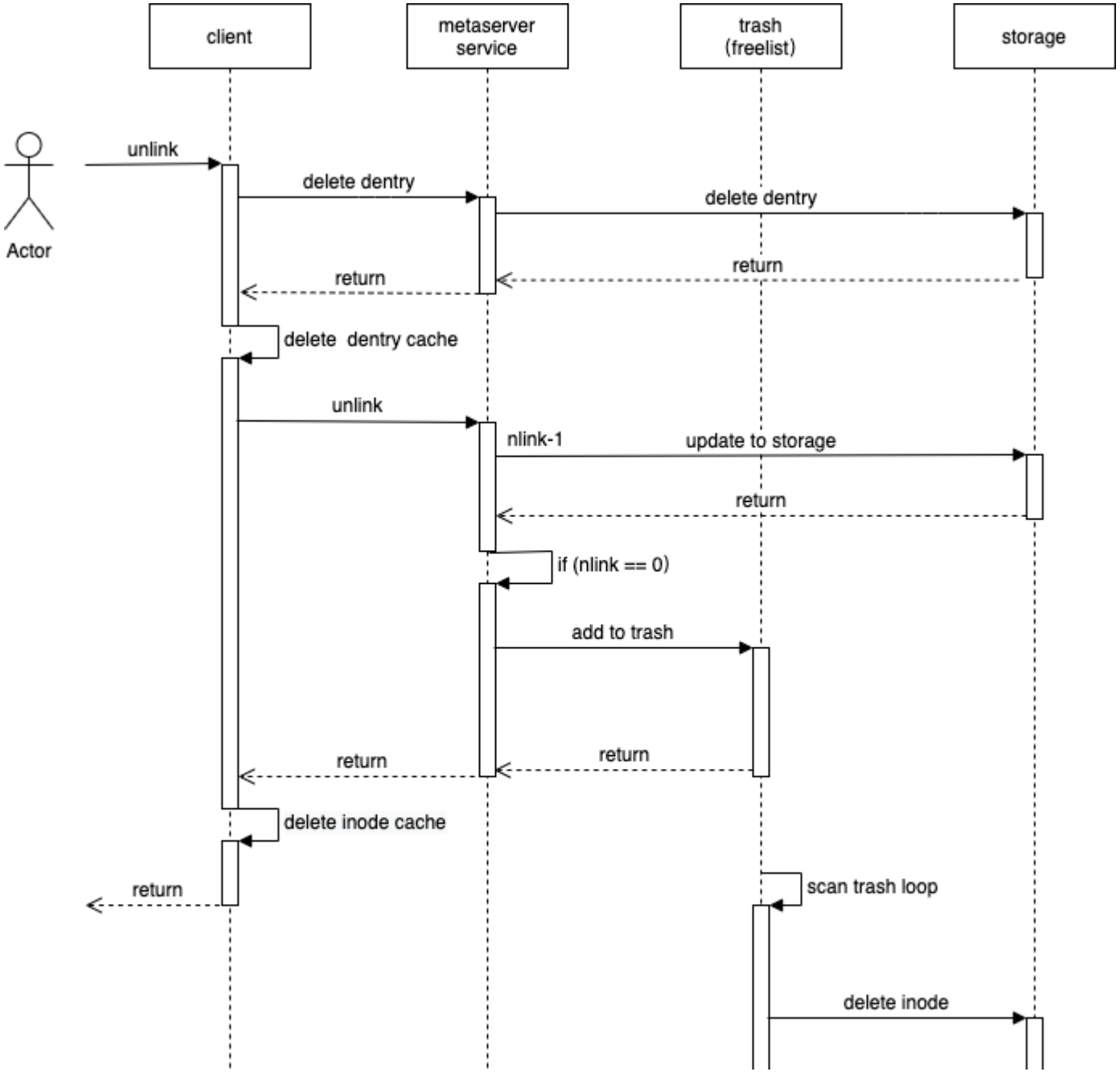
3. 是否需要做session机制（在metaserver打开），来维护inode的打开情况？

- 经讨论，需要实现session机制，以应对打开文件被另一个进程删除的场景的场景。

方案设计

经小组会议讨论，决定使用trash + session机制去实现上述功能。

ulink流程如下：


client
metaserver service
trash (freelist)
storage
Actor
unlink
delete dentry
delete dentry
return
return
delete dentry cache
unlink
nlink-1
update to storage
return
if (nlink == 0)
add to trash
return
return
delete inode cache
return
scan trash loop
delete inode

Trash机制：

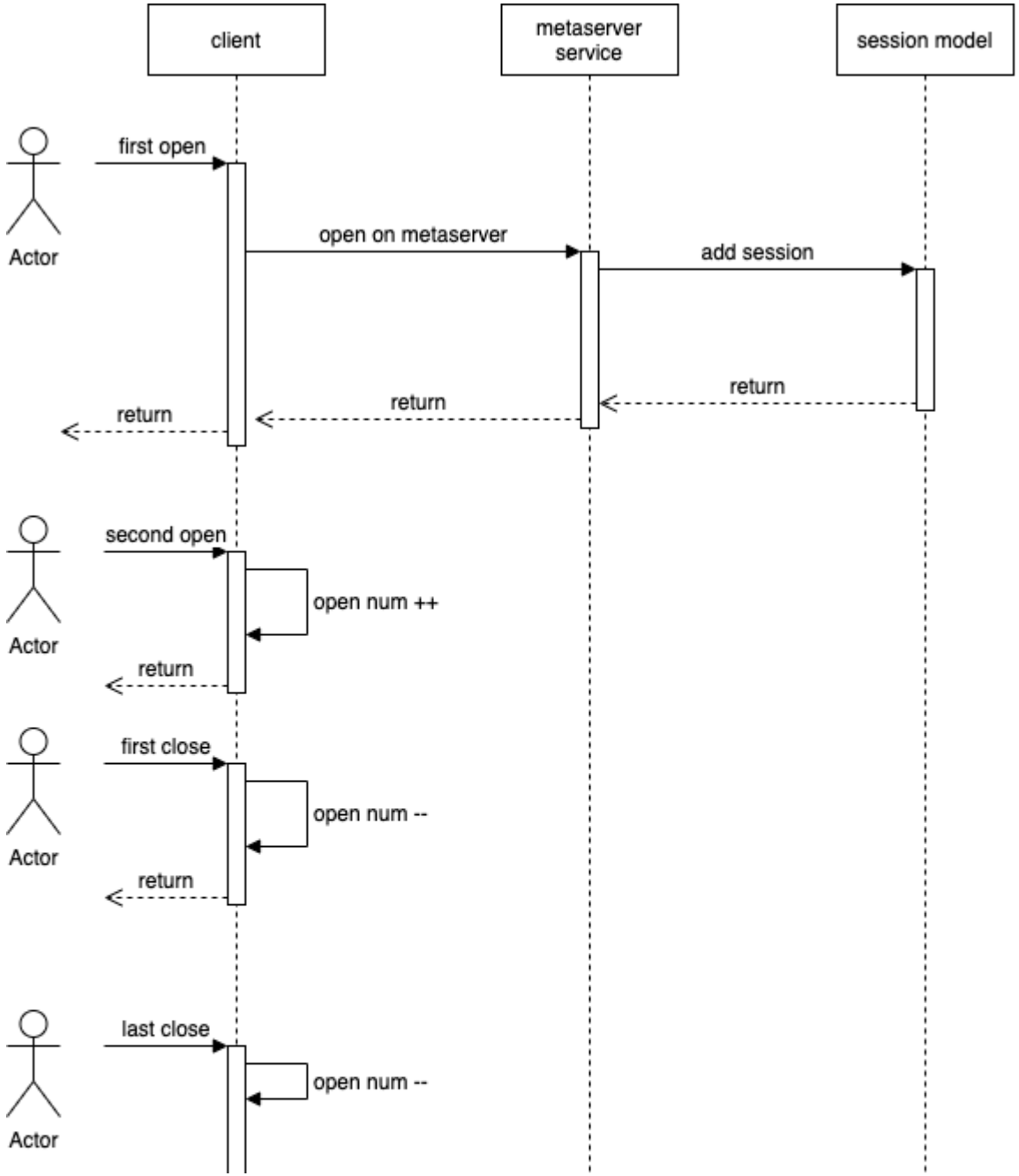
metaserver端实现一个trash机制，需要做的事情如下：

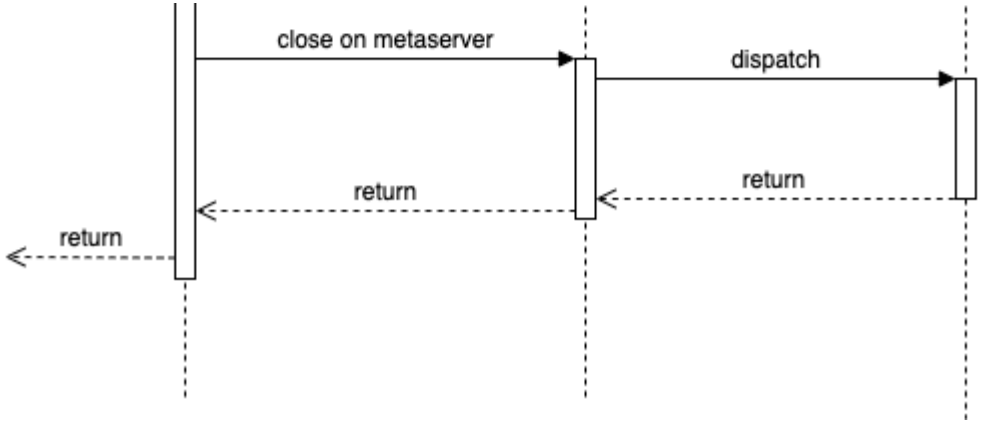
1. 以类似chubaofs的freelist的方式记录nlink==0的inode id， inode结构保存在原地，进入trash时记录进入trash的时间。
2. trash需要定期扫描freelist中的inode id， 当发现inode没有被打开已经进入trash时间大于7天（可配置）时，将inode清理，同时删除相关数据（s3上和卷上的）。
3. trash中需要区分inode是否被打开，以帮助工具在查询时，展示inode进入trash的情况，这部分可以实现一个Flag，这个Flag可以记在inode中。

Session机制：

session机制，如下图所示：

1. client端在open时，首先会去判断是否已经在metaserver上open，如果没有，则先去metaserver端执行open
2. metaserver端的open过程，会记录一条session到内存中，表示当前inode已经被client打开
3. client端后续的open只在本地将open num++
4. client端在close过程中，首先会去open num－， 当发现open num==0时，也就是所有的open都已经close了，此时调用close on metaserver
5. close on metaserver的过程，将移除内存中的session。


client
metaserver service
session model
Actor
first open
open on metaserver
add session
return
return
return
Actor
second open
open num ++
return
Actor
first close
open num --
return
Actor
last close
open num --



遗留问题

上述方案还存在一个遗留问题，就是孤儿inode的问题：

client无论在进行unlink过程时，需要两步，第一步是删除dentry，第二步是nlink-1，那么在执行完第一步之后，client如果崩溃或者掉电，或者是发送nlink-1的rpc失败（可以重试，但重试仍有可能失败），这种情况下，就会存在nlink未被减1的情况，当所有硬链接都被删除后，就会出现孤儿inode。

moosefs由于只有一个mds节点看，所以不存在这个问题。

chubaofs的解决方案是：

在Delete_ll（api.go） 函数中，在delete dentry后有一段注释如下：

// dentry is deleted successfully but inode is not, still returns success.
此时，nlink是没有-1的，删除接口直接忽略了第二步的错误。

根据其论文描述：

could significantly affect the system performance.

Our tradeoff is to relax this atomicity requirement *as long as a dentry is always associated with at least one inode*. All the metadata operations in CFS are based on this design principle. The downside is that there is a chance to create *orphan inodes*[8], which may be difficult to be released from the memory. To mitigate this issue, each metadata-operation workflow in CFS has been carefully designed to *minimize the chance of an orphan inode to appear*. In practice, a meta node rarely has too many orphan inodes in the memory. But if this happens, tools like *fsck* can be used to repair the files by the administrator.

chubaofs使用的是类似fsck的工具去修复这个问题，也就是运维手段。

工作量评估

需要修改的模块，如下：

- Client端：
  实现symlink、link接口；
  修改unlink、rmdir接口，删除dentry，调用metaserver unlink，而不直接删除inode
  修改open，增加release接口，调用metaserver open 和close接口，增加open计数，记录client端open的数量
  增量client与metaserver session模块，定期refresh session 到metaserver，这个要做客户端级别的，不是文件级别的，防止rpc请求数量过多

- MetaServer端功能一 Trash机制：
  需要实现unlink接口， 进行nlink－，当nlink==0时，将inodeid 放入trash
  需要实现trash逻辑，每个parition 实现一个trash将nlink==0 的 inode记录下来，并实现后台定期扫描清理inode的逻辑，定期清理需要对接上s3实际删除和卷的删除（卷的部分可先不做，预留接口）
  需要实现强制清理的接口；
  为工具实现查询trash接口；

- Metaserver端功能二 session机制：
  需要实现在metaserver open file的接口，在接口中保存session。（需不需要持久化？单节点metaserver可以不持久化，但是高可用之后，怎么通知另外两个metaserver，需要再考虑）
  需要实现在metaserver close file的接口，移除session。
  实现metaserver端session模块，如果长时间收不到client refresh session，即session超时，此时清理该client的所有文件打开的session记录。

- 工具实现：
  工具需要实现查询各个parition，组织展示trash中数据；
  工具实现强制清理trash的接口；

- S3实际删除部分：
  S3中对象的删除需要在metaserver中调用，而不是client调用，实现上删除接口应该不需要处理inode，这部分与原先有些区别。truncate接口可能仍由client端调用，这部分与原先区别不大。@程义

- 卷的实际删除部分：
  先不做