rename 接口实现方案（已实现，选用方案二）

背景

当前 curvefs 并没有实现 rename 接口，本文档是对 rename 接口实现的调研及方案设计。

rename 操作，主要操作的是 dentry，如 rename /dir1/file1 /dir2/file2，主要有 2 个步骤：(1) 删除 file1 的 dentry，(2) 增加 file2 的 dentry（该 dentry 的 inodeid 等同 file1 的 inode id）。

关于 rename 接口的实现，主要调研了 chubaofs 和 juicefs，而 rename 的实现难点主要在于其原子性的保证。

方案调研

Chubaofs

chubaofs 中的 rename 实现不是原子性的，它是通用创建源文件的硬连接，然后删除源文件的方式来实现的，主要有以下 4 步：

1. 将源文件的 nlink 加一
2. 创建目标文件的 dentry
3. 删除源文件的 dentry
4. 将源文件的 nlink 减一

而每一步骤都有可能出错，chubaofs 针对以上的 4 步骤中出现的错误处理如下：

1. 步骤 1 出错，啥事都没发生
2. 步骤 2 出错，等同于创建硬连接出错，恢复机制如下：
    1. 将源文件的 nlink 减一
3. 步骤 3 出错，相当于创建了硬链接，但是没有删除源文件，此时源文件和目标文件同时存在，恢复机制如下：
    1. 删除目标文件 dentry
    2. 将源文件的 nlink 减一

    备注：如果这一步骤出错，并且恢复机制没有执行成功，那么会导致一些问题：

(1) 用户不能在源目录或目标目录下创建相应文件了，因为文件名被占用了（如 rename /dir1/file1 /dir2/file2，既不能在 /dir1 目录下创建 file1，也不能在 /dir2 目录下创建 file2 ）

(2) 并且因为存在硬链接，不能通过再次 rename 来获取成功（一般用户 rename 返回失败后，有可能希望再次执行 rename 以获得成功）

4. 步骤 4 出错，会导致 inode 有可能没办法被正常回收（nlink 始终大于 0），恢复机制如下：
    1. 对于这一步出错，没有恢复机制，与 unlink 操作失败一样的处理（因为 dentry 删除了，而 inode 却没被回收，会被当成孤儿节点去处理）

如果采用 chubaofs 的方案，需要考虑以下问题：

1. 以上的恢复进制如果没执行成功怎么办？
    1. 客户端存活的情况下，应该多尝试几次，直至成功
2. 但是如果恢复机制尝试多次没成功，或者客户端挂掉、宕机该如何处理？
    1. 步骤 1：忽略
    2. 步骤 2：只是给 nlink + 1 了，这个等同于 unlink 操作时删除了 dentry 而 nlink 没减一的情况，同步骤 4 恢复机制一样，当做孤儿节点来处理
    3. 步骤 3：这一步出错，就会同时存在 src、dst 的 dentry，相当于多了一个硬链接，Linux 和 POSIX 接口中表明这允许一段时间内存在，但是最终还是要原子性，所以这一步出错会导致和本地文件系统不一致的行为：
        1. Linux 接口定义允许 rename 过程中某一段时间存在这样的硬链接（或者 rename 执行到一半断电也会存在）
        2. 而 POSIX 接口中提到了该函数得是原子（不断电的情况下，rename 操作不能被其他操作打断，不存在中间状态）
        3. 参考：
            1. Is rename() atomic?
            2. rename(2) - Linux man page

                > However, when overwriting there will probably be a window
                > in which both oldpath and newpath refer to the file being renamed

            3. rename(3) - Linux man page

                > This rename() function is equivalent for regular files to that defined by the ISO C standard.
                >
                > Its inclusion here expands that definition to include actions on directories and specifies behavior
                > when the new parameter names a file that already exists.
                > That specification requires that the action of the function be atomic.

            4. vfs_rename

```
 VFS  VFS  rename  3
     rename



rename()
 do_renameat2()
  ...

  lock_rename(struct dentry *p1, struct dentry *p2)
   mutex_lock(&p1->d_sb->s_vfs_rename_mutex);  //
   inode_lock_nested(p1->d_inode, I_MUTEX_PARENT);
   inode_lock_nested(p2->d_inode, I_MUTEX_PARENT2);

  ...

  vfs_rename()
```

4. 步骤 4：当做孤儿节点来处理

chubaofs 和本地文件系统的差异：

	chubaofs 方案	本地文件系统	差异
rename 全部步骤成功	/	/	无
rename 过程中某一步骤失败	有可能会出现中间状态 （如 nlink 多加一了，同时存在 src、dst 的 dentry） 对于用户来说，一旦创建了硬链接也无法通过再次 rename 恢复	整个过程原子性，要么成功，要么失败则恢复原始状态，不存在中间状态 失败了可以再次尝试 rename	有
执行到某一步骤掉电	有可能存在中间状态	有可能存在中间状态（待验证：这种情况 fsck 会不会修复）	无

源码实现伪代码如下：

```
func Rename(srcParentId, srcName, dstParentId, dstName)
 // nodeid nlink
 local srcNodeId = GetDentry(srcParentId, srcName).nodeId  //
 ilink(srcNodeId)  // srcNode.nlink++

 //  dentry
 local destDentry = Dentry{ dstParentId, dstName, srcNodeId }
    local err = CreateDentry(destDentry)
    if err == EEXIST and IS_FILE(srcNodeId) then  //  dentry  dentry  inode  srcNodeId
     oldNodeId, err = UpdateDentry(destDentry)
    end

 if err != nil then  //  dentry
     iunlink(srcNodeId)  //
    end

 //  dentry
 local srcDentry = Dentry{ srcParentId, srcName }
 local err = DeleteDentry(srcDentry)
 if err != nil then
  if oldNodeId == 0 then
   DeleteDentry(Dentry{ dstParentId, dstName })  //  dentry
  else
      UpdateDentry(Dentry{ dstParentId, dstName, oldNodeId })  //  dentry
  end
  iunlink(srcNodeId)
   end

 // nodeid nlink
 iunlink(secNodeId)
 if oldNodeId != 0 then  //
      iunlink(oldNodeId)
      ievict(oldNodeId)
   end
end
```

Juicefs

Juicefs 中 rename 的实现都是原子性的，主要得益于它元数据是存储在各类 KV/DB 中（如 redis、tikv...），而这些 KV 本身就支持事务，所以它只要把这些操作打包成事务扔给 KV 就可以了

如果采用 Juicefs 的方案，我们需要在 metaserver 层实现分布式事务

方案实现

方案一：chubaofs

从以上的分析来看，chubaofs 的方案是可行的，参照其实现就行

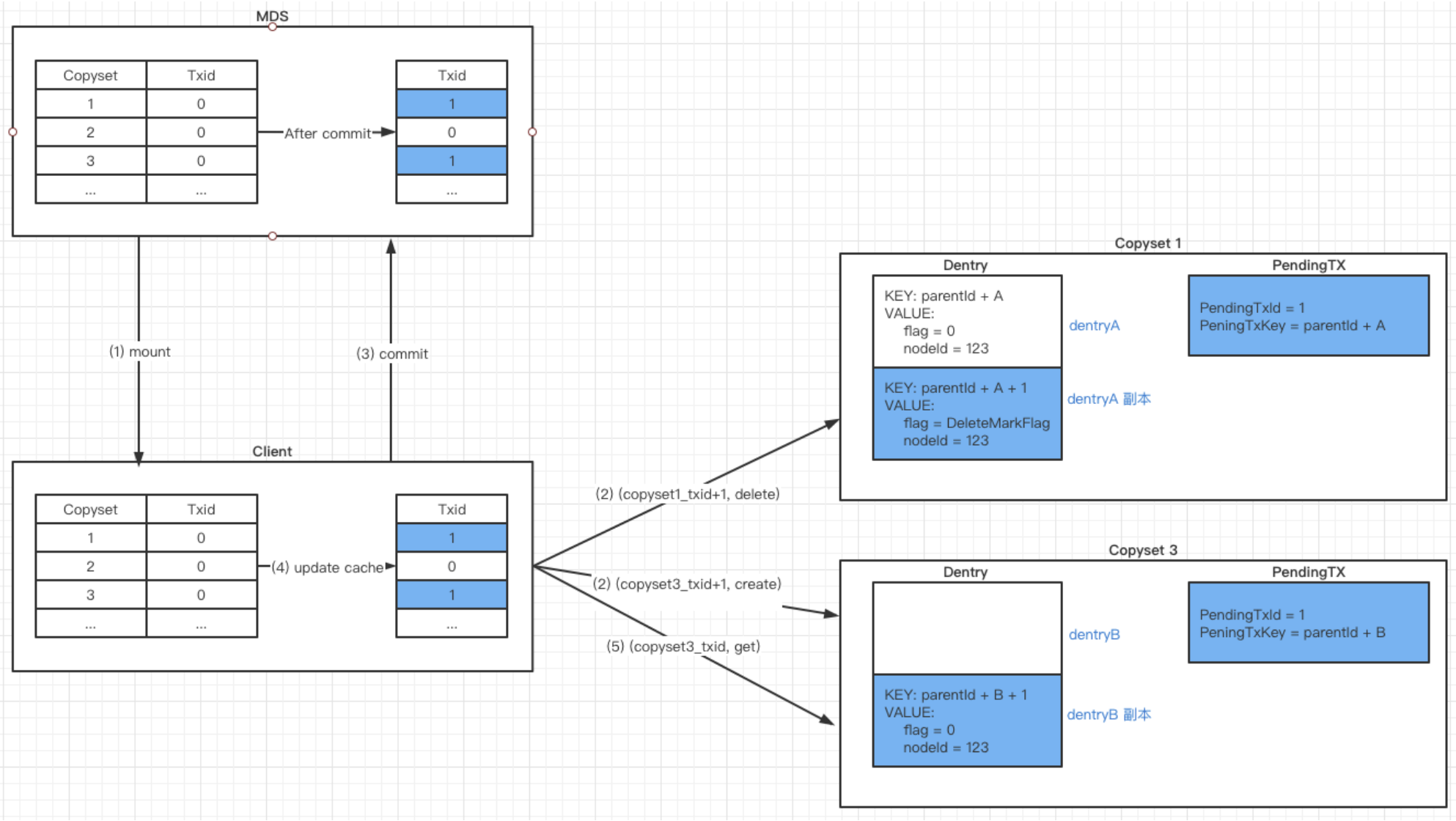
方案二：事务方案

前言（关于 MVCC）：

MVCC (Multi-version Concurrency Controller) ，即多版本并发控制，主要解决的是并发读写时的冲突问题，利用该机制在读写时候可以去除锁机制

(备注：我认为利用 MVCC 可以保证事务 ACID 中的 C(一致性) 和 I(隔离性))

方案主要借鉴 leveldb 与 etcd(boltdb) 中事务的实现（主要利用 mvcc），方案设计如下：



整体思路如下：

- 在 MDS 所有 copyset 中增加一个 txid 字段，保存当前 copyset 已成功的事务 id（该事务 id 顺序递增，事务每成功一次则加一）
- 每次 rename 开始时，将 srcDentry, dstDentry 所在 copyset 对应的 txid 分别加 1（copyset_txid+1）去删除/创建/修改 dentry（其实就是创建副本，不管是删除/创建/更改都是创建相应 copyset_txid+1 为 key 的副本，原始 dentry 不动），并设置 PendingTx 为本次事务
- 如果上一步骤成功了，就提交事务，将 srcDentry, dstDentry 所在 copyset 的 txid 都加 1（这一步是通过 etcd 的事务保证的），如果上一步或这一步失败，因为 txid 不变，原始数据版本也在，还是保证原子性（其实就是一个 txid 对应一个版本的数据）

- 下次访问的时候，带上对应 copyset 的最新 txid (copyset_txid)，判断 PendingTx，如果 (copyset_txid >= PendingTxId && rpc_request.key == PendingTxKey)，则表明 PendingTx 对应的事务是已经成功了的，并且 PendingTx 对应事务刚好操作的是请求的 dentry，则返回 PendingTxKey + PendingTxId 对应的副本 dentry，否则返回原始 dentry
- PendingTx 与 dentry 副本是一一对应的，下面有机制确保，每个 copyset 只需要一个 PendingTx（即整个 copyset 中最多只会存留一个副本 dentry）

下面是图中流程说明：

- (1) mount 的时候将 MDS 中所有 copyset 对应的 txid 缓存在本地
- (2) 以 (copyset_txid+1) 去对应的 copyset 删除/创建/修改 dentry
  - 对于 dentry 需要增加一个字段：
    - flag：用于标记是否删除，当该 flag 为 DeleteMarkFlag 时，表示该 dentry 已删除
  - 操作：
    - 删除 dentry：创建副本 dentry (KEY: parentId + name + copyset_txid+1, VALUE: flag = DeleteMarkFlag, nodeId=...)
    - 创建 dentry：创建副本 dentry (KEY: parentId + name + copyset_txid+1, VALUE: flag = 0, nodeId=...)
    - 修改 dentry: 创建副本 dentry (KEY: parentId + name + copyset_txid+1, VALUE: flag = 0, nodeId=...)
- (3) 提交事务，将 2 个 copyset 对应的 txid 都加一 （这一步是通过 etcd 的事务实现，不存在一个 copyset_txid 加一，一个没加一）
- (4) 如果事务提交成功了，更新 Client 的 txid 缓存
- (5) 下次访问的时候，带上对应 copyset 的最新 txid (copyset_txid)，判断 PendingTX，如果 (copyset_txid >= PendingTxId && rpc_request.key == PendingTxKey)，则表明 PendingTx 对应的事务是已经成功了的，并且 PendingTx 对应事务刚好操作的是请求的 dentry，则返回 PendingTxKey + PendingTxId 对应的副本 dentry，否则返回原始 dentry

关于其他说明点：

- 每个 copyset 保存一个 PendingTx，(1) 保存上一次事务的 txid (PendingTxId)，(2) 以及操作的 dentry 对应的 key (PendingTxKey)
  - 每次访问，不管什么操作，只需要与 PendingTx 做比较即可
    - copyset_txid >= PendingTxId && dentry.key == PendingTxKey，返回副本 dentry
    - 否则返回原始 dentry
  - 但是如果是 rename 事务的话，则需要先处理这个 PendingTx：
    - 如果当前事务带上来的 rpc_request.txid == PendingTxId 的话，则表示上一次事务失败了，则将该 PendingTxKey 对应的 dentry 的副本删除即可
    - 如果当前事务带上来的 rpc_request.txid > PendingTxId 的话，则表示上一次事务成功了，我们则更新 dentry 对应的 value 为副本 dentry 的 value，并删除副本 dentry（如果更新完发现 dentry 的 flag 为 DeleteMarkFlag，则直接删除这个 dentry）
    - 如果上面 2 个动作，有一个失败，则本次事务失败
- VFS 这层保证了每个挂载点只会有一个 rename 事务，所以这就变成了一写的事务场景
- 初略来看，这个方案只要 3 个 RPC 请求就够了，2 次 dentry 操作，一次提交事务 (txid) 操作

实现：

- MDS 中的 copyset 需要增加一个 txid 字段（需持久化）
  - txid 为 uint64（8 个字节），增加的存储量为 8 * copyset 的数量
- MeteServer 中的每个 Copyset 需要增加一个字段 PendingTx (持久化在 MetaSever 中)，因为每次 Client 来访问的时候都需要与这个 PendingTx 作比较
  - PeningTxId 为 int64 (8 个字节)，PendingTxKey (parentId + name) = (8 + 256) (256 为文件名最大长度)
  - 所以增加的存储字节数最多为 272 * copyset 的数量

方案三：利用 KV 自带的分布式事务

后期我们有其他需求可以替换 KV 的话，可以考虑替换成 TiKV 这些天生支持分布式事务的的 KV，参照 juicefs，将这些事务直接扔给 KV 就行

结论：方案一和方案二应该都是可以实现的，方案三目前短期应该实现不了，下面是方案一和方案二的对比：

	方案一：chubaofs	方案二：事务方案
优点	逻辑简单，实现方便	可以保证原子性，不会出现中间状态

缺点	没办法保证原子性（与本地文件系统行为不一致）	逻辑比 chubaofs 的稍微复杂，实现需要考虑全面
工作量	二者应该差不多，事务方案稍微多一点	二者应该差不多，事务方案稍微多一点

Q&A

1. 是否需要实现跨文件系统的 rename 操作？

不需要，因为在 VFS 这层如果发现 rename 操作的 2 个文件不属于同一挂载点的话就会返回 EXDEV (Invalid cross-device link) 错误：

```
rename()
 do_renameat2()
  ...

  error = -EXDEV;
  if (old_path.mnt != new_path.mnt)
   goto exit2;

  ...

  vfs_rename()
```

而 linux 下的 mv 操作之所以能跨文件系统，是因为它是通过 read 源文件，然后 write 目标文件的方式实现的：

```
 mv
$ strace -o log sh -c 'mv srcfile /tmp/dstfile'

...
rename("srcfile", "/tmp/dstfile")       = -1 EXDEV (Invalid cross-device link)
unlink("/tmp/dstfile")                  = 0
...
open("srcfile", O_RDONLY|O_NOFOLLOW)    = 3
open("/tmp/dstfile", O_WRONLY|O_CREAT|O_EXCL, 0600) = 4
read(3, "hello world\n", 131072)        = 12
write(4, "hello world\n", 12)           = 12
...
```

2. 在多客户端情况下，是否需要加锁来保证其原子性？

我认为是需要的，根据 POSIX 对接口原子的要求，rename() 操作期间不允许被其他操作打断，对于单客户端来说，VFS 层已经给我们保障了，不需其他开发，
而如果我们挂载多客户端的话，需要一个分布式锁，可以利用块设备中 open() 的 session 机制来实现文件锁，加锁的粒度可以参考 VFS

3. rename 流程举例说明？

例 1：rename A→B (A 存在，而 B 不存在)

一些假设：

- 将文件 A 对应的 dentry 称为 dentryA，并假设 dentryA 属于 copyset1
- 将文件 B 将要创建的 dentry 称为 dentryB，并假设 dentryB 属于 copyset3
- 将 copyset1 对应的 txid 称为 copyset1_txid，copyset3 对应的 txid 称为 copyset3_txid，并假设 copyset1_txid 与 copyset3_txid 都为 0

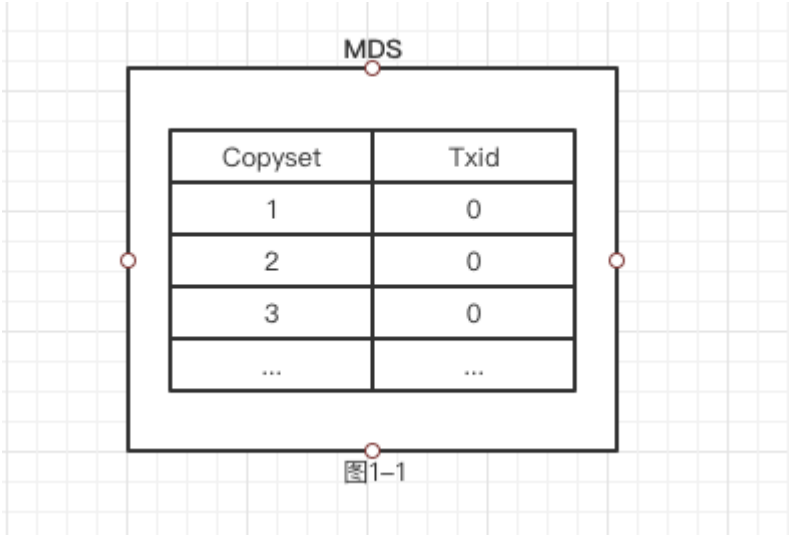


图1-1

步骤 1：获取 dentryA, dentryB 所属 copyset 的 txid

- copyset1_txid = 0
- copyset3_txid = 0

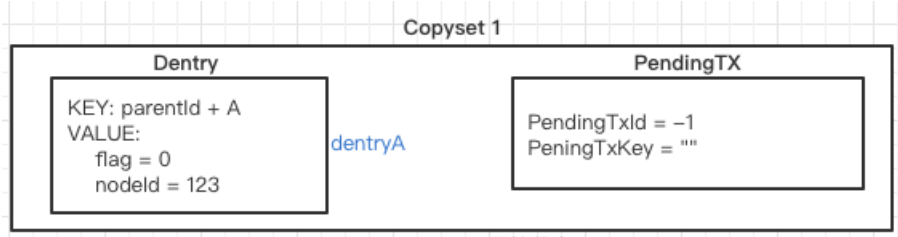


图2-1: copyset1 原始状态

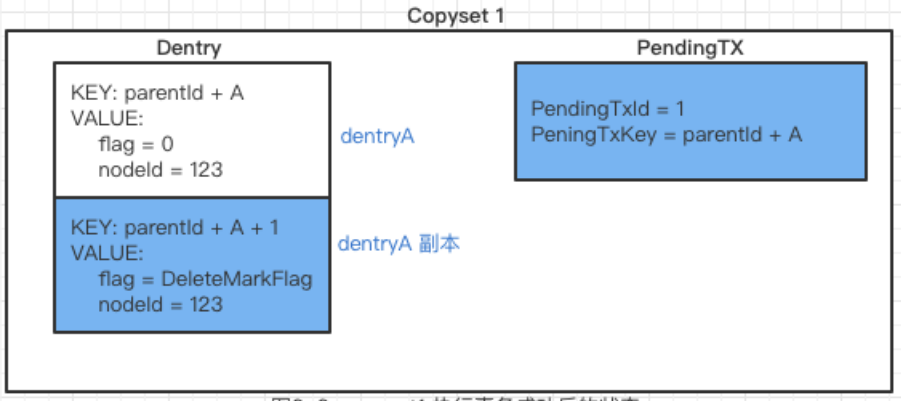


图2-2: copyset1 执行事务成功后的状态

步骤 2：Client 去 copyset1 删除对应 dentryA

- 2.1: Client 将 copyset1_txid + 1 去请求 copyset1，发送 RPC 事务请求 rpc_request(txid:1, key:parentId+A, type: tx, ...) 至 copyset1
- 2.2: copyset1 接收到 RPC 请求后做如下处理:
  - 2.2.1: 将 PendingTxId 设为 rpc_request.txid（即为 1)，PendingTxKey 设为 rpc_request.key (即为 parentId + A)
  - 2.2.2: 创建 dentryA 的副本 （如 <图2-2> 中 "dentryA 副本" 所示 ）
- 2.3: copyset1 返回成功响应给客户端

错误说明：这里分 2 类讨论步骤 2 中每一小步出错的处理机制：

类别1：该 rename 事务结束后的请求不是 rename 事务请求，即其他任何请求

2.1~2.3 中的任一步骤出错，都不会影响非 rename 事务请求，因为此时 copyset1_txid 仍为 0，Client 去请求 copyset1 都会带上这个 copyset1_txid，rpc_request(txid: 0, key: parentId + A, type: normal, ...)

- 2.1: 这步出错，相当于 copyset1 没收到 RPC 请求，copyset1 状态还是如 <图2-1> 所示，copyset1 根据 rpc_request.key 获取对应 value，并且 (PendingTxId = -1 && PeningTxKey == "") (代表没有副本)，直接将 "dentryA" 返回给客户端
- 2.2
    - 2.2.1: 只是修改了 PeningTX，这一步出错，相当于 PendingTx 没修改成功，而非事务请求并不去处理 PeningTX，所以逻辑同 2.1 处理
    - 2.2.2: 假如这一步出错，即 "dentryA 副本" 未创建成功，copyset1 判断 rpc_request.txid(0) < PeningTxId(1)，则表明 PendingTx 对应的事务没有成功，直接将 "dentryA" 返回给客户端
- 2.3: 这一步出错，copyset1 的状态已经如 <图2-2> 所示了，copyset1 判断 rpc_request.txid(0) < PeningTxId(1)，则表明 PendingTx 对应的事务没有成功（事务没有提交），直接将 "dentryA" 返回给客户端
- 其他：假如步骤 2 (指原始请求步骤) 中的步骤都成功了，但是 copyset1_txid 还是为 0（因为没提交），所以去请求时，copyset1 的处理逻辑还是同 2.3

类别2：该 rename 事务结束后的请求为 rename 事务请求

我们假设本地 rename 过程中，还是要去请求这个 copyset1，因为每个 copyset 的 txid 是独立的，如果去请求其他 coyyset 的话，copyset1 完全不受影响，这里就不不讨论了

因为事务没提交，copyset1_txid 仍然为 0，而且是事务请求 rpc_request(txid: 1, key: parentId + C, type: tx, ...)，copyset 判断 rpc_request.txid(1) == PendingTxId(1)，则表明上次事务没成功，copyset 需要先处理上次事务留下的 PendingTx，处理成功后，再处理本次的事务：

- 2.1：这一步出错，copyset1 仍为原始状态（如 <图1-1> 所示），啥都没改变，本次事务就是重头开始执行一样，无需处理
- 2.2：
    - 2.2.1: 这一步出错，相当于 PendingTx 没修改成功，无需处理
    - 2.2.2: 这一步出错，相当于 PendingTx 修改成功，但是 "dentryA 副本" 创建失败 ，无需处理
- 2.3: 这一步出错，copyset1 的状态已经如 <图2-2> 所示了，copyset 需要删除 "dentryA 副本"（首先根据 PendingTxId + PeningTxKey 判断 dentry 副本是否存在）
- 其他:
    - 以上的 PendingTx 处理，有一个处理失败，则本次事务失败，下次事务依旧是根据以上的流程处理 PendingTx（PendingTx 的处理流程都是幂等的）
    - 如果以上 PendingTx 的处理流程成功，上次事务失败的影响就全部没有了，都清理干净了（相当于 rollback），正式开始本地事务的时候将 PendindTx 设为本次事务相关的值即可
    - PendingTx 记录了上次事务的所有必要信息，根据 PendingTx 就可以将 copyset 回滚到事务前
    - 一般来说 PendingTx 回滚不容易失败，因为请求已经到达 copyset 了，它只是做删除 dentry 副本的操作，之所以 rollback 放到下一次事务处理，而不是上次事务出错就 rollback 主要是基于这个考虑：
        - 少几次 RPC 请求
        - 出错就 rollback 还是有可能出错

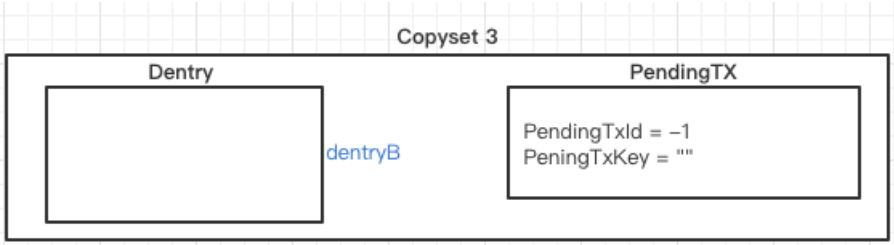


图3-1: copyset3 原始状态

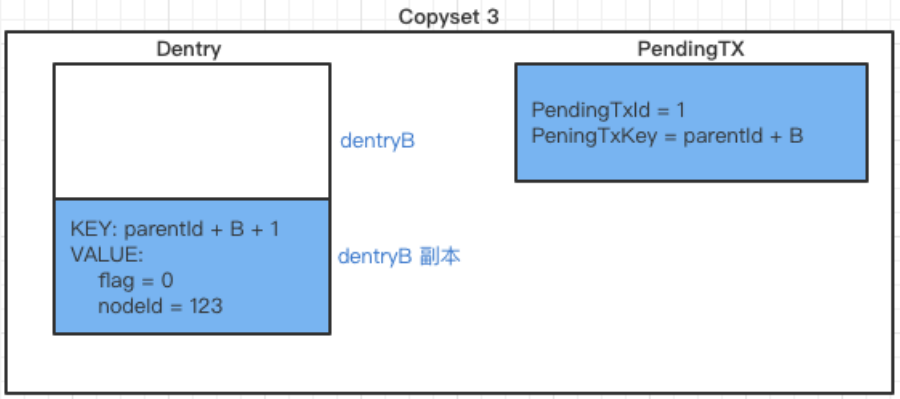


图3-2: copyset3 执行事务成功后的状态

步骤 3：Client 去 copyset3　创建 dentryB

步骤及错误处理都如同步骤 2

图4-1

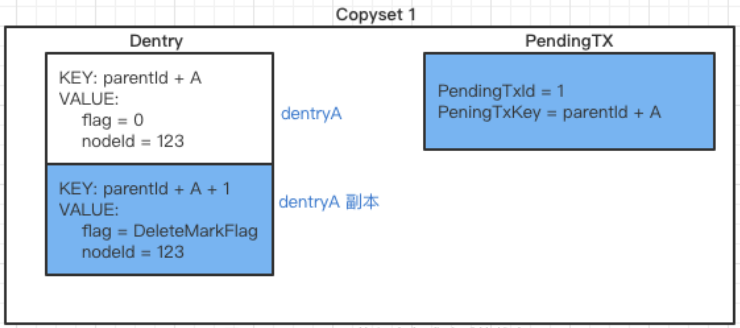


图4-2: copyset1 执行事务成功后的状态

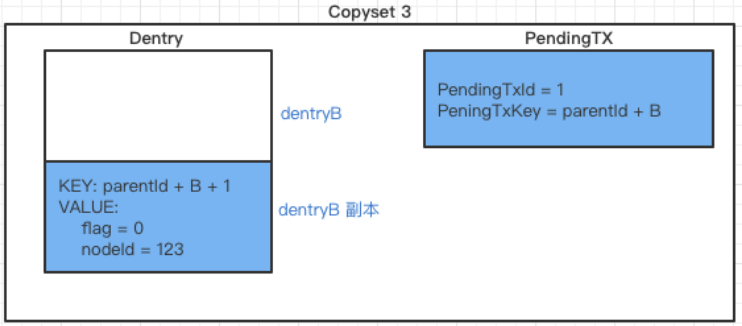


图4-3: copyset3 执行事务成功后的状态

步骤 4：Client 向 MDS 提交 txid，提交事务

- 4.1: 将 (copyset1_txid=1, copyset_txid=1) 提交给 MDS

假如 4.1 失败：

- 假如失败，copyset1_txid 和 copyset3_txid 仍为 0，相应的处理上面都有列出来

假如 4.1 成功：

- 那么各组件的状态就如 <图4-1>、<图4-2>、<图4-3> 所示，copyset1_txid 与 copyset3_txid 都为 1
- 如果不是 rename 事务，那么访问 dentryA rpc_request (txid:1, key: parentId + A, type: normal, ...), copyset1 判断 (rpc_request.txid == PendingTxId && rpc_request.key == PendingTxKey)，则根据 KEY(PendingTxKey + PendingTxId) 返回 "dentryA 副本"，并且发现 "dentryA 副本" 的 flag 为 DeleteMarkFlag，表示该 dentry 已被删除，则直接返回空（访问 dentryB 处理机制相同）
- 如果是 rename 事务，那么请求 dentryA 的 rpc 请求则为 rpc_request(txid:2, key: parentId + A, type: tx, ...), copyset 发现 rpc_request.txid(2) > PendingTxId(1)，则表示上次事务已经成功了，这里需要做的就是先清理副本，然后再执行本次事务：
    - (1) 将 ”dentryA“ 的值设为 “dentryA 副本” 对应的值，即处理完后，dentryA = (KEY: parentId + A, VALUE: { flag = DeleteMarkFlag, nodeId = 123 })
    - (2) 删除 "dentryA 副本"
    - (3) 删除 "dentryA 副本"后，判断 dentryA 的 flag 为 DeleteMarkFlag，则删除 dentryA
    - 以上 3 步骤中有一步失败则本次事务则失败，下次事务仍按以上流程处理（处理为幂等）
    - 以上 3 步骤中有一失败，都不会影响正常访问
    - 处理流程图如以下 <图4-4> 所示

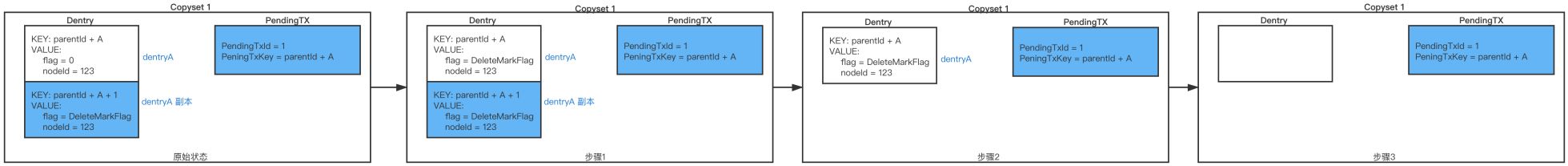


图4-4: copyset1 rename 事务处理上一次成功的 PendingTx 流程

例 2：rename A→C (A 存在，而 C 存在)

- 处理流程大部分同 <例1>
- dentryC 处理后的状态图如 <图5-1>、<图5-2> 所示
- 唯一不同的是，当事务处理成功后，下次事务开始时，处理 PendingTx 时需要删除原始 inode，处理 PendingTx 流程如下：
    - (1) 首先判断 "dentryC" 与 "dentryC 副本" 的 nodeId 是否相同，不同则删除原始 inode
    - (2) 将 "dentryC" 的值设为 "dentryC 副本" 对应的值
    - (3) 删除 "dentryC 副本"

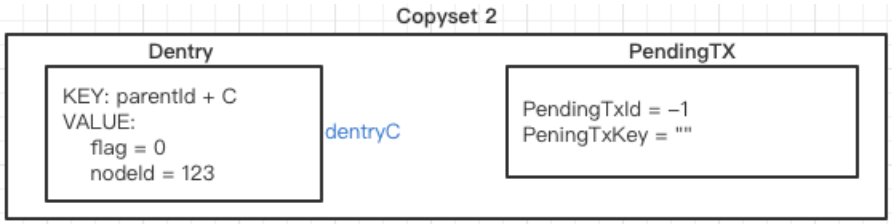


图5-1: copyset2 原始状态

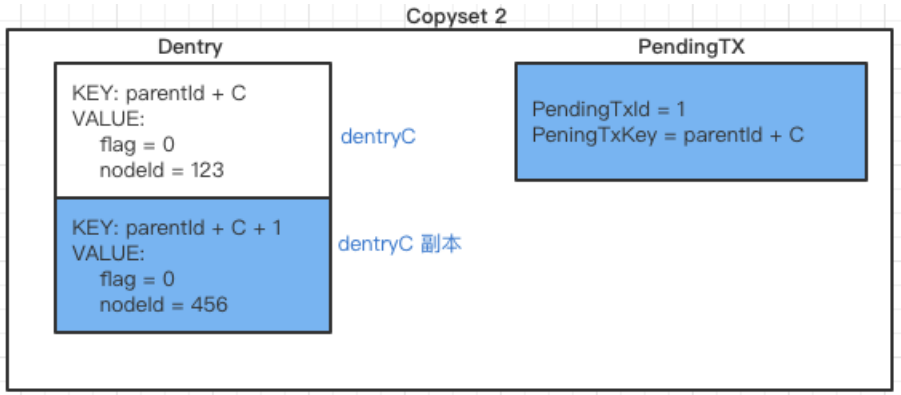


图5-2: copyset2 执行事务成功后的状态

4. 当 2 个操作的 dentry 属于同一个 copyset 有什么不一样？

- 基本流程和以上的〈例1〉都是一样的，主要不同的如下：
  - PendingTx 需要记录 2 个 PendingTxKey，如下图所示
  - 原本发送给 2 个 copyset 的 dentry 操作，合并成一个 rpc 请求发送给指定 copyset（这个 rpc 请求里包含 2 个 dentry 操作）