CTO 元数据性能优化方案

前言

目前 CTO 的实现

我认为目前的 CTO 属于以下 tradeoff 的 2 个端点：

- 在关闭 CTO 的情况下，一个客户端的修改在另外一个客户端是不可见的。性能较好，但是一致性没法得到保证
- 在开启 CTO 的的情况下，抛弃了所有缓存。一致性得到保证，但是导致元数据性能比较差

优化的目标和方向

目标

- 缓存一致性和性能类似于天平的两端，一致性越强则性能越弱，反之亦然。从目前调研来看，CubeFS/JuiceFS/NFS 都无法保证全部场景下的元数据一致并保证性能
  对于元数据来说，我们需要实现大部分场景下的缓存一致性，并且提供足够好的性能，而对于数据我们始终保持 close-to-open 的策略。对于一致性，我们将从以下 3 个维度衡量：
  - 文件可见性：即在一个客户端执行文件的创建、删除、重命名等操作，另外一个客户端要立马能看到。对于这一维度，我们要完全保证
  - 属性一致性：指一个文件的属性（如 atime, ctime, mtime, mode, size...）在一个客户端修改后，另外一个能立即看到。针对这一维度，我们允许某些属性在某些场景下存在一定的延时（相当于弱一致性）
  - 内容一致性：文件内容要完全符合 close-to-open 语义，即关闭后，重新打开后即能看到之前全部的更改
- 对于以上的 3 个维度中出现的某些场景不一致情况，需要分析其产生的原因和对业务的影响

方向

- 对于优化方向，能用内核缓存就用内核缓存（包括 dentry 缓存、atttibute 缓存、page cache），因为相对用户态内存缓存来说，内核缓存可以有效较少 IO 路径，性能提升更为明显
- 提供足够灵活的配置，根据用户的业务需求，调整配置以满足缓存一致性需求和性能要求（当然，调整配置也可以实现与现在 cto/nocto 几乎一样的表现）

关于目前已接入的业务

我们分析了以下我们目前接入的几个业务的 IO 模型，应该是可以满足需求的：

运维 ES（关闭 CTO）

- ES环境curvefs使用接口情况
- es问题分析与定位

云音乐广告算法 K8S AI（开启 CTO）

- 云音乐广告算法k8s存储对接沟通

杭研多媒体 AI（开启 CTO）

- 多媒体ai的存储特性
- 其他参考
  - 2022-2023 杭研多媒体业务
  - 多媒体ai环境curvefs挂载

方案设计

方案来源：

方案主要来源与 JuiceFS 与 NFS 的实现：

JuiceFS:

- JuiceFS 文档中心 - 缓存
- Github: juicedata/juicefs

NFS：

- nfs(5) - Linux man page
- [RFC 7530] Network File System (NFS) Version 4 Protocol
- Kernel documents
- Linux NFS Version 4: Implementation and Administration
- How the NFS Service Works
- An Enhanced Disk-Caching NFS Implementation for Linux
- Managing NFS and NIS, 2nd Edition

NFS vs JuiceFS:

> NFS: 缓存 → 校验不一致 (getattr/mtime)→ 刷新(ESTALE/drop cache) → 保持一致
>
> NFS: only getattr/lookup: BBQ，others: 处理不一致（抓住一切机会保证一致性）
>
> JuiceFS: 内核缓存、内核缓存、内核缓存.....（不一致性 → 缩短缓存时间/换使用方式 → 一致 ）
>
> 性能：NFS > JuiceFS
>
> 一致性：NFS >> JuiceFS

优化点 1：ls

问题

一个 ls 操作通常会执行以下这些系统调用，以及其对应的 fuse 请求如下（各系统调用对应的 fuse 请求可参见：<系统调用下发到 Fuse 的请求>）：

- opendir(): lookup * n + getattr * n + opendir
- readdir(): readdir * 2+
- stat()/fstat(): lookup * n + getattr * n
- closedir(): releasedir

从上面看到，会有大量的 lookup 和 getattr 请求下发到 curve-fuse，因为目前的 CTO 实现抛弃了所有内核和内存的 attribute/dentry 缓存，这些操作每次都需要回 MetaServer，导致文件越多，性能下降越明显。

方案 1：NFS

> NFS 在开启 ac 后会缓存 inode attribute，但是这个有超时时间（actimeo）过期后会从服务端重新校验，如果没有更改，仍然使用本地缓存，并将其在内核中重新设置 timeout（actimeo）
>
> NFS 在开启 lookupcache=positive 后，会保存 lookup 结果（即 dentry）

opendir

- 对于 opendir 来说，NFS 无论如何都会向后服务端发送 getattr 请求，来获取该目录的属性，并通过该目录 mtime（或者 change_info）和本地缓存的目录 mtime 作对比，来判断该目录内容是否有更改（如在该目录下创建文件、删除文件、rename 文件）。如果更改了则扔掉该目录下的全部缓存，重新从服务端获取，否则该目录下的所有文件将重新在内核中获取 actimeo 的超时时间
- 具体的相关使用及基本原理可参考：nfs(5) - Linux man page

readdir:

- 如果上一步验证目录没有更改，那么直接使用缓存，不会有任何请求回服务端
- 如果对比后目录有更改，那么丢弃所有内存缓存，该目录下的每个文件的 lookup/getattr 都将回服务端拿

方案 2：JuiceFS

opendir():

- 如果内核缓存了，lookup/getattr 不会下发到 fuse，只会下发 opendir
- 对于 opendir 来说，正常回服务端拿 inode

readdir:

- 通过一个 HSCAN 获取该目录下的所有 dentry
- 再通过一个 batch 请求（MGET）将该目录下所有的 inode 获取过来，并将 attribute/entry 都缓存在内核中。所以之后这些 stat() 不会再有 lookup/getattr 请求下发到 JuiceFS 中

从上面可以看到 NFS 主要在 opendir 这里做动作，通过 getattr 来判断目录是否有修改，一旦无修改将使用缓存，不会有任何回服务端的请求。而 JuiceFS 主要在 readdir 做动作，每次通过 2 个请求获取所有的 dentry/inode 并缓存起来。

对于首次或缓存失效（目录已变更）需要回服务端的请求来说：JuiceFS 会有更好的性能，因为它是一个 batch 请求获取所有的 inode，而 NFS 则是一个个去拿

对于缓存未失效（目录未变更）来说：NFS 具有更好的性能，因为它只需要发送一个 getattr 请求到服务端校验即可，而 JuiceFS 则每次需要发送 2 个 RPC（一个获取 dentry，一个 batch 请求获取所有 inode）。但是对于属性一致性来说，JuiceFS 更有优势，因为它每次获取的都是最新的 inode，而 NFS 只在文件创建、删除等场景才会去服务端拿。

所以，对于该点，我们可以采用 JuiceFS 的做法，batch 拿回后根据用户配置的缓存时间，直接缓存在内核中即可。对于 ls 请求来说，正常情况下只需要 2 个回 MetaServer 的请求（ListDentry、GetBatch Inode）

另外，ES 之前存在的问题〈es问题分析与定位〉主要是缓存导致的，跟该方案是没关系的。目前该问题是规避掉了，我们提供了各类缓存超时时间，同样也可以规避。

优化点 2：缓存配置

关于缓存及其配置

- 使用内核缓存（dentry 缓存、atttibute 缓存），因为相对用户态内存缓存来说，内核缓存可以有效较少 IO 路径，性能提升更为明显
- 我们需要足够灵活的配置，以适应用户各类的使用场景。以下各个配置都是相互独立，不相互影响（缓存的配置参考自 JuiceFS 和 NFS）
- 在默认请求下就需要满足共享文件的需求，并保证性能

配置项	默认值	说明
cto	开启	我认为文件系统共享是它的一个基本特性，默认应该开启。这个只针对文件内容一致性，不影响 attribute/dentry 的缓存（目前是关联的）
attr-cache-timeout	1 秒	属性的缓存
entry-cache-timeout	1 秒	文件 dentry 的缓存
dir-entry-cache-timeout	1 秒	目录 dentry 的缓存
meta-memory-cache	关闭	默认关闭，只有在用户单挂载或者确定内存缓存不影响一致性的情况下开启（这个理解能 metaserver 上的内存缓存可能更好理解，只不过架构在客户端，在目前的方案下需要回 metaserver 拿的，有内存后就会先过一次内存）

优化点 3：close-to-open

问题

目前的方案，由于在内核中缓存了 inode 的属性，所以当另外一个客户端追加写后，缓存在内核中 inode 的 size 可能还是老的，就会导致 write 出错。（对于 read 来说，即使 size 是错的也没关系，下面会说明）

目前 JuiceFS 的实现，也存在该问题，size 不正确会导致数据被覆盖丢失，详见测试 <测试场景 1：close-to-open 一致性测试（读写轮询>。关于 JuiceFS，我们猜测可能是通过以下方法来避开这个问题，但是这个不一致问题是存在的，而且是不符合 close-to-open 语义的：

- 配置比较短的缓存时间，这样一个客户端写会，切到另外一个客户端时，另外一个客户端的 attriubte 已经失效了，所以会从服务端拿最新的 attribute
- 业务某些就可能存在只在一个客户端写，见<测试场景 2：close-to-open 一致性测试（全写）>（但是其实这样，这个问题还是会发生....）

open

对于目前 open 接口来说，只要修改为直接从 MetaServer 获取 inode 即可

read

> 总的来说，系统为了 page 对齐(4k~128k)，相当于提供了一个自适应的过程。所以即使我们将错误的 inode 缓存在内核中，也不会导致 read() 操作出现任何问题。
>
> 这个对齐特性，也在本地验证测试过

因为我们将 attribute 和 entry 都缓存在内核中了，所以系统调用 open() 中的 lookup/getattr 不会下发到 fuse，只会下发 open 请求到 fuse。VFS 会以目前缓存的错误 inode 的 size 下发 read 请求（4k 对齐）到 fuse，

- 如果发现 read 返回是 0，那么
  - 会重新绕过内核缓存下发 getattr 到 fuse，之后重新下发 read 请求到 fuse
  - 并且在下次打开该文件之前，会绕过内核缓存下发一个 getattr 请求到 fuse，所以下次 fuse 层会收到 getattr 和 open 2 个请求
- 如果 read 的值小于下发的 size，那么
  - 会重新绕过内核缓存下发 getattr 到 fuse，之后重新下发 read 请求到 fuse

所以内核存在一个自适应的过程，所以对于目前 open 接口来说，只要修改为直接从 MetaServer 获取 inode 即可。

write

> 如果 2 个客户端轮流写这个文件就会存在数据不一致/丢失的请求，我们以追加写为列：

时序	客户端 A	说明	客户端 B	说明
	open(f1)	这时候 size 是 0		
	write("1")	这时候下发的 off 是 0		
	close(f1)			
	cat(f1)	这时候 inode attibute 就缓存起来了，size 是 1		
			open(f1)	因为没有缓存，那么 open() 下发的 getattr 是从服务端拿的，所以 size 是 1
			write("2")	这时候下发的 off 是 1
			close(f1)	
	open(f1)	因为有缓存，那么 open() 下发的 getattr 直接从 VFS 返回了，拿到的 size 是 1		
	write("3")	这时候下发的 off 是 1，就出错了，写入 "3" 后，文件内容就变成 "13"，而不是 "123"		
	close(f1)			

也可以参见 <测试场景 1：close-to-open 一致性测试（读写轮询>，里面有详细的分析。从上面的分析来看，我们严格遵循了 close-to-open 的操作顺序，但是最终的结果却是不一致的（数据丢失了）

close

> 对于开启 cto 来说，要刷到 s3，而对于 nocto 来说，只要刷到本地缓存盘即可

NFS v4 的实现：

nfs4_file_open

```
static int
nfs4_file_open(struct inode *inode, struct file *filp)
{
 ...

 struct dentry *dentry = file_dentry(filp);

    ...

 /*
  * If no cached dentry exists or if it's negative, NFSv4 handled the
  * opens in ->lookup() or ->create().
  *
  * We only get this far for a cached positive dentry.  We skipped
  * revalidation, so handle it here by dropping the dentry and returning
  * -EOPENSTALE.  The VFS will retry the lookup/create/open.
  */

 dprintk("NFS: open file(%pd2)\n", dentry);
 inode = NFS_PROTO(dir)->open_context(dir, ctx, openflags, &attr, NULL);  //  open_context

    ...

 if (inode != d_inode(dentry))  //  inode  inode  EOPENSTALE
  goto out_drop;
    ...

out_drop:
 d_drop(dentry);
 err = -EOPENSTALE;
 goto out_put_ctx;
}
```

VFS 处理逻辑

对于 VFS 层来说，如果下面的文件系统返回 ESTALE 的话，它会带着 LOOKUP_REVAL 重新重新发起一次 open 请求，而 LOOKUP_REVAL 标记的作用就是忽略缓存（包括 attribute cache, entry cache）

#define LOOKUP_REVAL 0x0020 /* tell ->d_revalidate() to trust no cache

```
static struct file *path_openat(struct nameidata *nd,
   const struct open_flags *op, unsigned flags)
{
     ...
   error = do_open(nd, file, op);
     ...

 if (error == -EOPENSTALE) {  //  EOPENSTALE ESTALE
  if (flags & LOOKUP_RCU)
   error = -ECHILD;
  else
   error = -ESTALE;
 }
 return ERR_PTR(error);
}


struct file *do_filp_open(int dfd, struct filename *pathname,
  const struct open_flags *op)
{
     ...

 filp = path_openat(&nd, op, flags | LOOKUP_RCU);
     ...
 if (unlikely(filp == ERR_PTR(-ESTALE)))  //  ESTALE LOOKUP_REVAL  open
  filp = path_openat(&nd, op, flags | LOOKUP_REVAL);

     ...
}
```

关于 ESTALE 错误码：

这个错误码当初设计就是为了解决 inode 不一致的问题，可参见以下这些说明：

- errno(3) — Linux manual page
- https://www.kernel.org/doc/html/latest/filesystems/path-lookup.html
- enhanced ESTALE error handling (v3)

```
ESTALE Stale file handle (POSIX.1-2001).

              This error can occur for NFS and for other filesystems.
```

```
vfs_open() can fail with -EOPENSTALE if the cached information wasn't quite current enough.
Rather than restarting the lookup from the top with LOOKUP_REVAL set,
lookup_open() is called instead, giving the filesystem a chance to resolve small inconsistencies.
If that doesn't work, only then is the lookup restarted from the top
```

方案

所以我们只要在 open() 的时候去服务端拿 inode，对比是否 mtime 更改，如果 mtime 更改了则返回 ESTALE。那么此时 VFS 会无视该文件路径上的缓存重新发起 open，而这些 open 所下发的 lookup/getattr 等 fuse 操作都会回到 MetaServer。 此时我们把正确的 inode/dentry 缓存起来即可。

另外，我们可以利用 ESTALE 这个机制来实现一些更灵活的策略，比如对比其他 inode 属性，来返回这个错误码

POC 验证

返回 ESTALE 对应下发的 fuse 请求，看到一个 open 系统调用，会有 2 个 open 请求下发到 fuse 层，而第 2 个 open 请求会绕过所有缓存：

> cat /mnt/jfs/dir1/dir2/f1
>
> /mnt/jfs 为挂载点

- 关闭 attribute/entry 缓存

```
2023.02.19 21:58:30.392226 [uid:0,gid:0,pid:4056850] getattr (1): OK
(1,[drwxrwxrwx:0040777,3,0,0,1676512087,1676620382,1676620382,4096]) <0.000490>
2023.02.19 21:58:30.392869 [uid:0,gid:0,pid:4056850] lookup (1,dir1): OK
(2,[drwxr-xr-x:0040755,3,0,0,1676620382,1676620387,1676620387,4096]) <0.000440>
2023.02.19 21:58:30.393349 [uid:0,gid:0,pid:4056850] getattr (2): OK
(2,[drwxr-xr-x:0040755,3,0,0,1676620382,1676620387,1676620387,4096]) <0.000288>
2023.02.19 21:58:30.393839 [uid:0,gid:0,pid:4056850] lookup (2,dir2): OK
(3,[drwxr-xr-x:0040755,2,0,0,1676620387,1676624652,1676624652,4096]) <0.000324>
2023.02.19 21:58:30.394229 [uid:0,gid:0,pid:4056850] getattr (3): OK
(3,[drwxr-xr-x:0040755,2,0,0,1676620387,1676624652,1676624652,4096]) <0.000167>
2023.02.19 21:58:30.394669 [uid:0,gid:0,pid:4056850] lookup (3,f1): OK
(102,[-rw-r--r--:0100644,1,0,0,1676624652,1676725091,1676725091,12345]) <0.000272>
2023.02.19 21:58:30.394963 [uid:0,gid:0,pid:4056850] getattr (102): OK
(102,[-rw-r--r--:0100644,1,0,0,1676624652,1676725091,1676725091,3897]) <0.000141>
2023.02.19 21:58:30.395286 [uid:0,gid:0,pid:4056850] open (102): OK [fh:2] <0.000206>  #  open
2023.02.19 21:58:30.395540 [uid:0,gid:0,pid:4056850] getattr (1): OK
(1,[drwxrwxrwx:0040777,3,0,0,1676512087,1676620382,1676620382,4096]) <0.000197>
2023.02.19 21:58:30.395754 [uid:0,gid:0,pid:4056850] lookup (1,dir1): OK
(2,[drwxr-xr-x:0040755,3,0,0,1676620382,1676620387,1676620387,4096]) <0.000163>
2023.02.19 21:58:30.395979 [uid:0,gid:0,pid:4056850] getattr (2): OK
(2,[drwxr-xr-x:0040755,3,0,0,1676620382,1676620387,1676620387,4096]) <0.000181>
2023.02.19 21:58:30.396313 [uid:0,gid:0,pid:4056850] lookup (2,dir2): OK
(3,[drwxr-xr-x:0040755,2,0,0,1676620387,1676624652,1676624652,4096]) <0.000197>
2023.02.19 21:58:30.396554 [uid:0,gid:0,pid:4056850] getattr (3): OK
(3,[drwxr-xr-x:0040755,2,0,0,1676620387,1676624652,1676624652,4096]) <0.000108>
2023.02.19 21:58:30.396885 [uid:0,gid:0,pid:4056850] lookup (3,f1): OK
(102,[-rw-r--r--:0100644,1,0,0,1676624652,1676725091,1676725091,12345]) <0.000175>
2023.02.19 21:58:30.397089 [uid:0,gid:0,pid:4056850] getattr (102): OK
(102,[-rw-r--r--:0100644,1,0,0,1676624652,1676725091,1676725091,3897]) <0.000105>
2023.02.19 21:58:30.397297 [uid:0,gid:0,pid:4056850] open (102): OK [fh:3] <0.000104> #  open
```

- 打开 attribute/entry 缓存

```
2023.02.19 21:57:47.289938 [uid:0,gid:0,pid:4054851] open (102): OK [fh:2] <0.000394>
2023.02.19 21:57:47.290716 [uid:0,gid:0,pid:4054851] lookup (1,dir1): OK
(2,[drwxr-xr-x:0040755,3,0,0,1676620382,1676620387,1676620387,4096]) <0.000434>  #  lookup getattr lookup
attribute
2023.02.19 21:57:47.291372 [uid:0,gid:0,pid:4054851] lookup (2,dir2): OK
(3,[drwxr-xr-x:0040755,2,0,0,1676620387,1676624652,1676624652,4096]) <0.000365>
2023.02.19 21:57:47.292163 [uid:0,gid:0,pid:4054851] lookup (3,f1): OK
(102,[-rw-r--r--:0100644,1,0,0,1676624652,1676725091,1676725091,12345]) <0.000512>
2023.02.19 21:57:47.292744 [uid:0,gid:0,pid:4054851] open (102): OK [fh:3] <0.000311>
```

优化点 4：page cache

问题

用 curvefs-fuse-bt 脚本追踪读取文件可以发现，对于同一个文件，即使这个文件没有改变，curve-fuse 在每次读取该文件时都会接收到完整的 read 请求，并没有充分利用 page cache。

```
sudo ./curvefs-fuse-bt -p 1919638
```

```
...
FuseOpRead: pid(1919638) tid(1920688) ino(11534336) size(262144) off(0)
FuseOpRead: pid(1919638) tid(1928691) ino(11534336) size(262144) off(262144)
FuseOpRead: pid(1919638) tid(1920689) ino(11534336) size(262144) off(524288)
FuseOpRead: pid(1919638) tid(1921062) ino(11534336) size(262144) off(786432)
FuseOpRead: pid(1919638) tid(1928691) ino(11534336) size(102400) off(1048576)
...
```

JuiceFS 的做法

内核页缓存

在 JuiceFS 0.15.2 及以上，对于已经读过的文件，内核会为其建立页缓存（Page Cache），下次再打开的时候，如果文件没有被更新，就可以直接从内核页缓存读取，获得最好的性能。

JuiceFS 客户端会跟踪所有最近被打开的文件，要重复打开相同文件时，它会根据该文件是否被修改决定是否可以使用内核页数据，如果文件被修改过，则对应的页缓存也将在再次打开时失效，这样保证了客户端能够读到最新的数据。

当重复读 JuiceFS 中的同一个文件时，速度会非常快，延时可低至微秒，吞吐量可以到每秒几 GiB。

方案

CurveFS 可以在 open 的时候判断当前 inode 内容有没有改变（可以通过 mtime 判断），若没有改变，可以将 keep_cache 设为 true

- 在 CTO 开启下，open() 是直接从 MetaServer 拿 inode

```
struct fuse_file_info {
    ...

 /** Can be filled in by open. It signals the kernel that any
     currently cached file data (ie., data that the filesystem
     provided the last time the file was open) need not be
     invalidated. Has no effect when set in other contexts (in
     particular it does nothing when set by opendir()). */
 unsigned int keep_cache : 1;

    ...
};
```

POC 验证

> 读取一个 4MB 的文件

- 当前版本

```
root@debian10-002:~# time cat /mnt/cfs/dir1/dir2/f > /dev/null


real    0m0.028s

user    0m0.000s

sys     0m0.007s
```

- 开启 keep_cache

```
root@debian10-002:~# time cat /mnt/cfs/dir1/dir2/f > /dev/null

real    0m0.005s
user    0m0.000s
sys     0m0.004s
```

优化点 5：特殊文件的缓存

问题

目前的方案可以从以下 3 个维度来判断一致性：

- 文件可见性：即在一个客户端执行文件的创建、删除、重命名等操作，另外一个客户端要立马能看到。对于这一维度，我们要完全保证
- 属性一致性：指一个文件的属性（如 atime, ctime, mtime, mode, size...）在一个客户端修改后，另外一个能立即看到。针对这一维度，我们允许某些属性在某些场景下存在一定的延时（相当于弱一致性）
- 内容一致性：文件内容要完全符合 close-to-open 语义，即关闭后，重新打开后即能看到之前全部的更改

虽然能满足大多数场景，但是有些用户可能存在一些以下需求：

- 对某些特定文件需要实时能看到属性的变化
- 不希望牺牲性能

方案

对于这些需求可以可以提供更灵活的配置，针对特定前缀/后缀的文件提供指定的缓存策略，以上这些通用的缓存配置相当于是全局配置：

```
attr_cache_timeout: 1
entry_cache_timeout: 1
dir_entry_cache_timeout: 1

specify:
   *.index: 0  # no_cache
   log*: 0     # no_cache
```

实现的话，我们只要将这些前缀/后缀读入后在内存中构建一个 AC 自动机，就可以处理好这些匹配。对于 AC 自动机算法来说，其空间和时间的复杂度都是 O(n)，所以内存和 CPU 消耗都可以忽略不计。

加上这个策略后，在首次 ls 的时候，会将目录下其余文件的 attribute/entry 都缓存起来，而针对这些特定的文件则不缓存即可。那么在用户再次 ls 的时候，缓存起来的文件都可以从内核直接返回，而这些特定的文件只需要再回 MetaServer 拿一次即可。

其他

对于部分元数据缓存不一致性的情况，我们分析了其触发的场景，以及实际对业务的影响：

> 这部分不一致主要是缓存导致，一旦缓存超时，最终会看到一致的试图（也就是说有延时），而这部分超时时间长短都是可由用户控制的

具体测试及说明详见 <NFS 测试报告>

优化点 6：file delegation

作用

关于 NFS 实现的 file delegation，我认为主要有以下 2 个作用：

- 提供更高的一致性要求。因为我们目前是 close-to-open
  的语义，就得要求用户读写文件的时候按照这个特定的顺序，否则无法保证其一致性（文件内容的正确性）。而在一些场景中，用户可能使用其他顺序，比如一个客户端在写，但是写到一半（内容更改还在本地），另外一个客户端就开始读了，它希望另外一个客户端能读到你更改的内容。
- 更好的性能。因为握有 read 授权（delegation）或 write
  授权的客户端是可以独享缓存的，在没有读写冲突发生被收回授权之前，他们的修改可以不用刷到服务端，而在本地使用。因为授权某个客户端就意味着只有你一个写，或者多个读，没有其他客户端在写，即没有读写冲突，你是可以独享缓存的。

实现

授权（delegation）：

- 服务端在接收到客户端 open() 时授予，授权在客户端/服务端 inode 都记录（客户端需要 clientid 来唯一标记）
- 握有某个 inode 授权的客户端，就代表当前整个文件系统中该 inode 不存在独享冲突，本客户端可以独享缓存

收回授权：

- 客户端需要起一个 RPC Service，作为服务端的 callback，当出现冲突时，服务端会通过该 callback 向该客户端收回授权（客户端需要在启动时向服务端注册 clientid）
- 当某个 inode 出现重写冲突时，如已经向某个客户端给与 write delegation，这时候就需要通过 callback 收回该授权，并将该客户端的修改全部同步到服务端
- 如果召回失败，在服务端标记，该客户端针对该 inode 的所有修改，都将返回 IO Error（只有在网络分区的情况下会出现这种情况）
- 召回成功，对于失去 write 授权的客户端来说，它需要将对该文件的修改试试刷到服务端，而对于 read 来说，需要直接从服务端读
- 对于已经出现冲突的 inode，服务端不会再向其他客户端下发授权（delegation）
- 文件 close 的时候，客户端会主动归还授权

delegation 的实现策略，主要的性能消耗在 open() 操作，因为重现冲突时，需要像客户端发送 callback 来收回授权，所以可能 open 操作比之前更慢些

关于 direcoty delegation

在 NFS v4.1 中，增加了目录 delegation，详见<NFSv4.1: Directory Delegations and Notifications> 或 [RFC 5661] Network File System (NFS) Version 4 Minor Version 1 Protocol

关于 CurveFS

我觉得目前 CurveFS 只要 close-to-open 的一致性就足够了，如果之后需要更高的要求，可以考虑将 delegation 实现

关于测试及说明

为了更好了解最终的实现效果，我们对 NFS 做了一个测试，包括性能和一致性。对于部分元数据缓存不一致性的情况，我们分析了其触发的场景，以及实际对业务的影响：

> 这部分不一致主要是缓存导致，一旦缓存超时，最终会看到一致的试图（也就是说有延时），而这部分超时时间长短都是可由用户控制的

具体测试及说明详见〈NFS 测试报告〉

另外，我认为我们需要一个集成测试框架来保证这些代码、代码质量（QA 保证最终的验收），可以利用 Go 快速开发个 FS 测试框架，集成到 CurveAdm 也可以。测试框架需要比较强的表达能力：

```
m1 := jtest.mount(cfg)
m2 := jtets.mount(cfg)

m1.write("f1", "123")
buf := m2.read("f1")
assert.Equal(buf, "123")
```