

# CurveFS 支持多挂载

- [背景](#)
- [本地文件系统的文件并发读写](#)
  - [write 和 read 行为](#)
  - [内核数据结构](#)
  - [多进程共享同一个文件](#)
    - [O\\_APPEND](#)
    - [文件锁 flock/fcntl](#)
    - [总结](#)
- [多挂载文件系统调研](#)
  - [GPFS](#)
    - [简介](#)
    - [数据和元数据资源并发保护](#)
      - [数据保护：字节区间锁](#)
      - [元数据保护：共享写锁](#)
      - [Allocation Map 保护](#)
      - [其他元数据信息保护](#)
    - [总结](#)
  - [Lustre](#)
    - [简介](#)
    - [Lustre 分布式锁管理器模型](#)
      - [基本锁模型](#)
      - [意图锁](#)
      - [范围锁](#)
      - [总结](#)
  - [JuiceFS](#)

- [简介](#)
- [JuiceFS 使用 DBServer 管理锁](#)
  - [flock 锁](#)
  - [fcntl 区间锁](#)
- [总结](#)
- [ChubaoFS](#)
- [CephFS](#)
  - [cephfs 使用 cap 实现分布式锁](#)
    - [caps permission](#)
    - [caps combination](#)
    - [caps 管理](#)
    - [总结:](#)
    - [fuse write 实例](#)
  - [总结](#)
- [NFS](#)
  - [简介](#)
  - [数据和元数据缓存一致性](#)
  - [客户端缓存](#)
  - [结论](#)
- [业务场景](#)
  - [AI 场景](#)
- [CurveFS 如何支持多挂载](#)

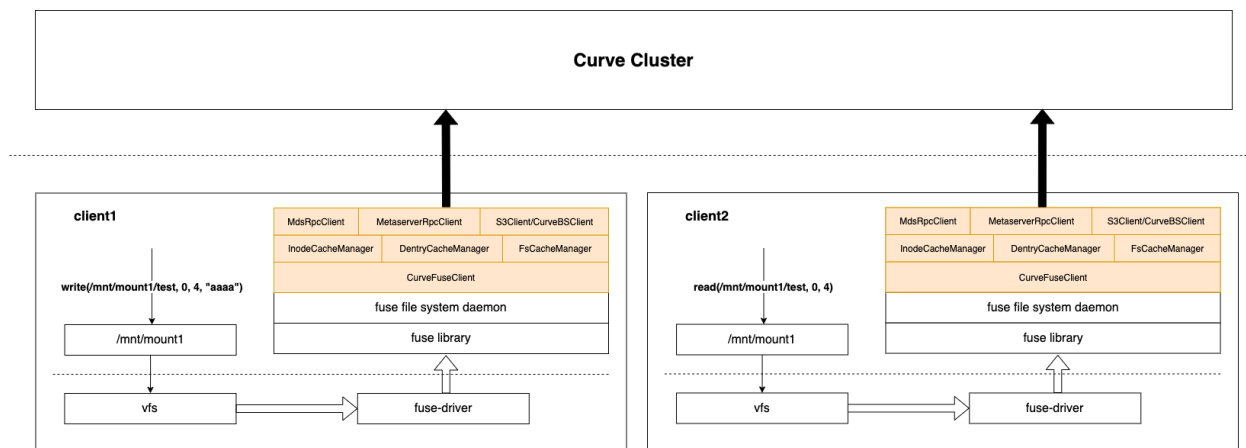
## 背景

当前有较多应用场景对于文件系统有一写多读、多写的需求，即同一个文件系统可以在多台机器上同时挂载并进行读写。例如：

- 云原生数据库：多台计算节点共同使用一个文件系统，其中主计算节点支持读写，从计算节点支持读。

- AI 训练：多台计算节点共享同一个文件系统，基本没有写同一个文件的场景，一个节点的写入数据是需要对其他节点立即可见。

下图描述的是 curvefs 的多挂载场景，client1 和 client2 分别在不同的节点上，挂载到相同的文件系统 fs1 上。



一写多读场景：client1 写入文件 `write(/mnt/mount1/test, 0, 4, "aaaa")`，client2 读取文件 `read(/mnt/mount1/test, 0, 4)`，期望可以读到“aaaa”。**对于一个文件系统，同一时刻只有一个节点在写入。**

多写多读场景 1：client1 写入文件 `write(/mnt/mount1/test, 0, 4, "aaaa")`，同时 client2 写入文件 `write(/mnt/mount1/test2, 0, 4, "bbbb")`，在两个节点写入完成之后，client2 读取到 `read(/mnt/mount1/test, 0, 4)` 为“aaaa”，client1 读取到 `read(/mnt/mount1/test2, 0, 4)` 为“bbbb”。**对于同一个文件系统，同一时刻有多个节点写入，但写入的是不同文件。**

多写多读场景 2：client1 写入文件 `write(/mnt/mount1/test, 0, 4, "aaaa")`，同时 client2 写入文件 `write(/mnt/mount1/test, 4, 4, "bbbb")`，在两个节点写入完成之后，client1 和 client2 读取到 `read(/mnt/mount1/test, 0, 8)` 为“aaaabbbb”。**对于同一个文件系统，同一时刻有多个节点写入同一个文件的不同位置。**

多读多写场景 3：client1 写入文件 `write(/mnt/mount1/test, 0, 4, "aaaa")`，同时 client2 写入文件 `write(/mnt/mount1/test, 0, 4, "bbbb")`，在两个节点写入完成之后，client1 和 client2 读取到 `read(/mnt/mount1/test, 0, 4)` 要不是“aaaa”，要不是“bbbb”。**对于同一个文件系统，同一时刻有多个节点写入同一个文件的相同位置。**

**对于以上四中场景，curvefs 在并发读写场景下需要支持何种一致性？如何支持？是本文需要得出的结论。**

## 本地文件系统的文件并发读写

在调研多挂载文件系统之前，我们梳理一下本地文件系统对于文件读写的并发操作行为。

## write 和 read 行为

posix 接口说明：

1. posix 并没有规定并发写行为。

This volume of POSIX.1 - 2008 does not specify behavior of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control.

2. posix 规定 read 在 write 返回后可以获取新的数据。无论读写在同一个进程还是在不同进程，读写都要保证这样的语义。
3. POSIX requires that a read(2) that can be proved to occur after a write() has returned will return the new data. Note that not all filesystems are POSIX conforming.  
If a read() of file data can be proven (by any means) to occur after a write() of the data, it must reflect that write(), even if the calls are made by different processes. A similar requirement applies to multiple write operations to the same file position. This is needed to guarantee the propagation of data from write() calls to subsequent read() calls. This requirement is particularly significant for networked file systems, where some caching schemes violate these semantics.

但是 linux 提供了一些系统调用，使得用户可以在使用这些接口实现期望的并发行为。

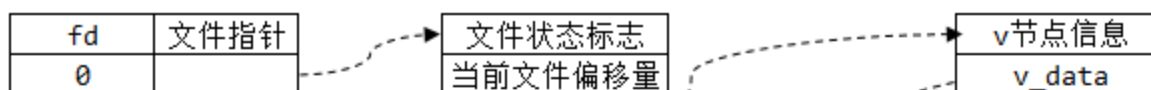
## 内核数据结构

linux 支持多进程间共享打开文件，同一时刻允许多个进程同时打开文件，每个进程之间的读写操作互不影响。为了实现这一个机制，linux 内核使用了三种数据结构表示打开的文件。

- 1、每个进程的进程表中有一个记录项，包含了当前进程所有打开的文件描述符，它包含了一个指向文件表项的指针和文件描述符标志。
- 2、内核中，为所有打开的文件维持一张表，它包含了以下内容：
  - 当前文件打开状态：以何种方式打开该文件，只读、只写或者可读可写
  - 当前文件的偏移量：当前文件指针所处的位置
  - 指向该文件节点表的指针：节点包含当前文件的属性信息

3、每个文件的信息被封装在 v 节点表项中，包含了当前文件名、所有者以及 inode 等信息。

三者之间的关系为：



当对文件进行写操作时，在文件表项中的文件偏移量将增加写入的字节数。如果此时文件偏移量超过了文件长度，则更新文件长度为当前的文件偏移量

## 多进程共享同一个文件

因为每个文件描述符都有一个属于自己的文件表项，所以每个进程之间的文件指针偏移互相独立，互相读写不干扰：每次 write 完成之后，文件表项的当前文件指针偏移量会立马加上写入的字节数。

那如何协调多进程之间的数据写入呢？linux 提供了以下几种方式：

### O\_APPEND

如果打开文件的时候加了 **O\_APPEND** 参数，每次写入数据前先把偏移量设置到文件末尾

## 文件锁 flock/fcntl

### flock

flock - apply or remove an advisory lock on an open file

- **flock** 提供的文件锁是建议性质的。即一个进程可以忽略其他进程加的锁，直接对目标文件进行读写操作。只有当前进程主动调用 flock 去检查，文件锁才会在多进程同步中起到作用。
- 文件锁必须作用在一个打开的文件上。

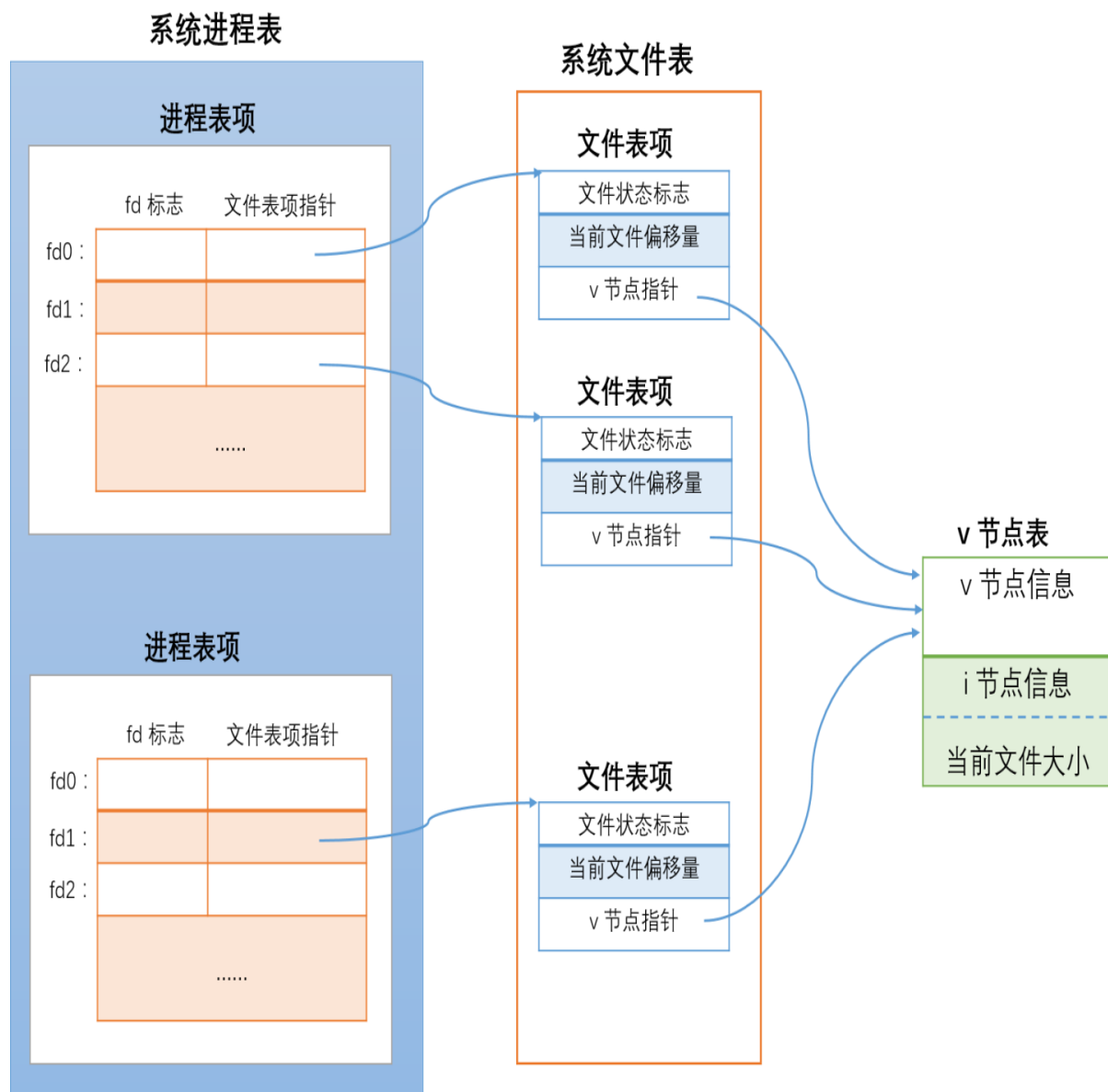
Locks created by flock() are associated with an open file table entry. flock() creates locks on systems's "Open file descriptions". "Open file descriptions" are generated by open() calls. a filedescriptor (FD) is a reference to a "Open file description". FDs generated by dup() or fork() refer to the same "Open file description".

**flock 对已打开的文件加锁时，是加在文件表项上。**考虑这种场景：进程 1 打开 file1 拿到 fd0，进程 2 打开 file1 拿到 fd1：

- `flock` 提供的文件锁是建议性质的。即一个进程可以忽略其他进程加的锁，直接对目标文件进行读写操作。只有当前进程主动调用 `flock` 去检查，文件锁才会在多进程同步中起到作用。
- 文件锁必须作用在一个打开的文件上。

Locks created by `flock()` are associated with an open file table entry. `flock()` creates locks on systems's "Open file descriptions". "Open file descriptions" are generated by `open()` calls. a filedescriptor (FD) is a reference to a "Open file description". FDs generated by `dup()` or `fork()` refer to the same "Open file description".

**`flock` 对已打开的文件加锁时，是加在文件表项上。**考虑这种场景：进程 1 打开 `file1` 拿到 `fd0`，进程 2 打开 `file1` 拿到 `fd1`：



如果进程 2 对 fd1 加上了排他锁，实际就是加在了 fd1 指向的文件表项上。此时，进程 1 对 fd0 加锁会失败。

因为操作系统会检测到进程 1 中 fd0 对应的文件表项 和 进程 2 中 fd1 对应的文件表项指向相同的 v 节点，且进程 2 中 fd2 对应的文件表项已经加了排它锁。

## fcntl

fcntl locks work as a Process <--> File relationship, ignoring filedescriptors

```
#include <unistd.h>
#include <fcntl.h>
```

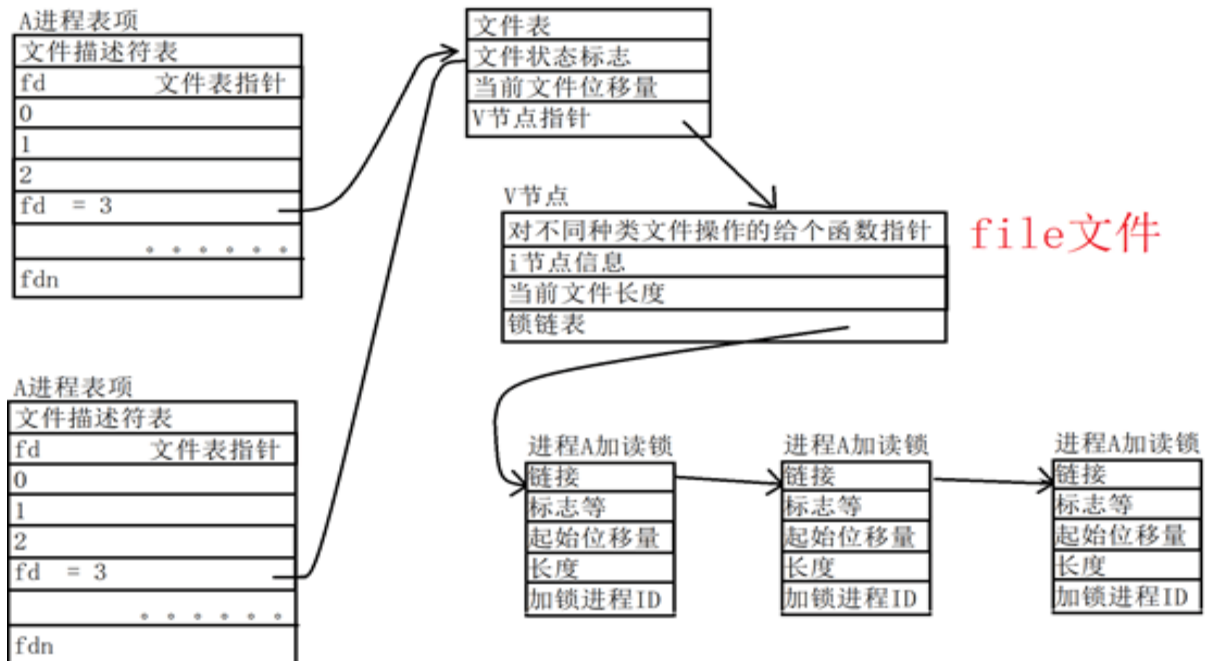
```
int fcntl(int fd, int cmd, ... /* arg */ );
```

fcntl 有多种功能，这里主要介绍文件锁相关的，当 cmd 被设置的是与文件锁相关的宏时，fcntl 就是用于实现文件锁。

cmd 关于锁的三个命令：F\_GETLK, F\_SETLK, F\_SETLKW

**fcntl 对已打开文件加锁时，是加在进程上，和文件描述符无关。**fcntl 可以对文件区间上锁。所有进程共享操作同一个文件的时候，只有一个 v 节点。共享同一个文件相当于共享同一个链表。





- 对于一个文件的任意字节, 最多只能存在一种类型的锁(读锁或者写锁);
- 一个给定字节可以有多个读锁, 但是只能有一个写锁;
- 当一个文件描述符不是打开来用于读时, 如果我们对它请求一个读锁, 会出错, 同理, 如果一个描述符不是打开用于写时, 如果我们对它请求一个写锁, 也会出错;
- 正如前面所讲的, 锁不能通过 fork 由子进程继承.

## 总结

- linux 通过文件锁 flock 和 区间锁 fcntl, 支持多进程访问时的写写互斥、读写互斥、读读共享;
- linux 通过 open 时设置 O\_APPEND 标志, 以及 pread/pwrite 实现原子写, 支持多进程追加写的原子性;

GPFS 的 DLM，使用一个集中的 global lock manager 运行在集群中的某个 node，与运行在每个 node 的 local lock manager 协作，global lock manager 通过分发 lock token 在 local lock manager 之间协调锁资源，lock token 传递授予分布式锁权利，而不需要每次获取或释放锁时产生一个单独的消息交换，同一个 inode 重复访问相同的 disk object，只需要一次消息交互，当另一个 inode 需要该锁时，需要额外的消息从第一个 inode 撤回 lock token，以授予第二个 node。local token 也扮演这维护 cache 一致性的作用。

GPFS 使用字节区间锁(byte-range locking)同步 file data 读写。GPFS 的区间锁并没有简单的实现为：在读写调用期间获取一个字节范围的令牌并在读写完成之后释放，因为这种方式锁的开销是不可接受的。GPFS 使用更复杂的字节区间锁协议以减少常见访问模式下的令牌竞争。

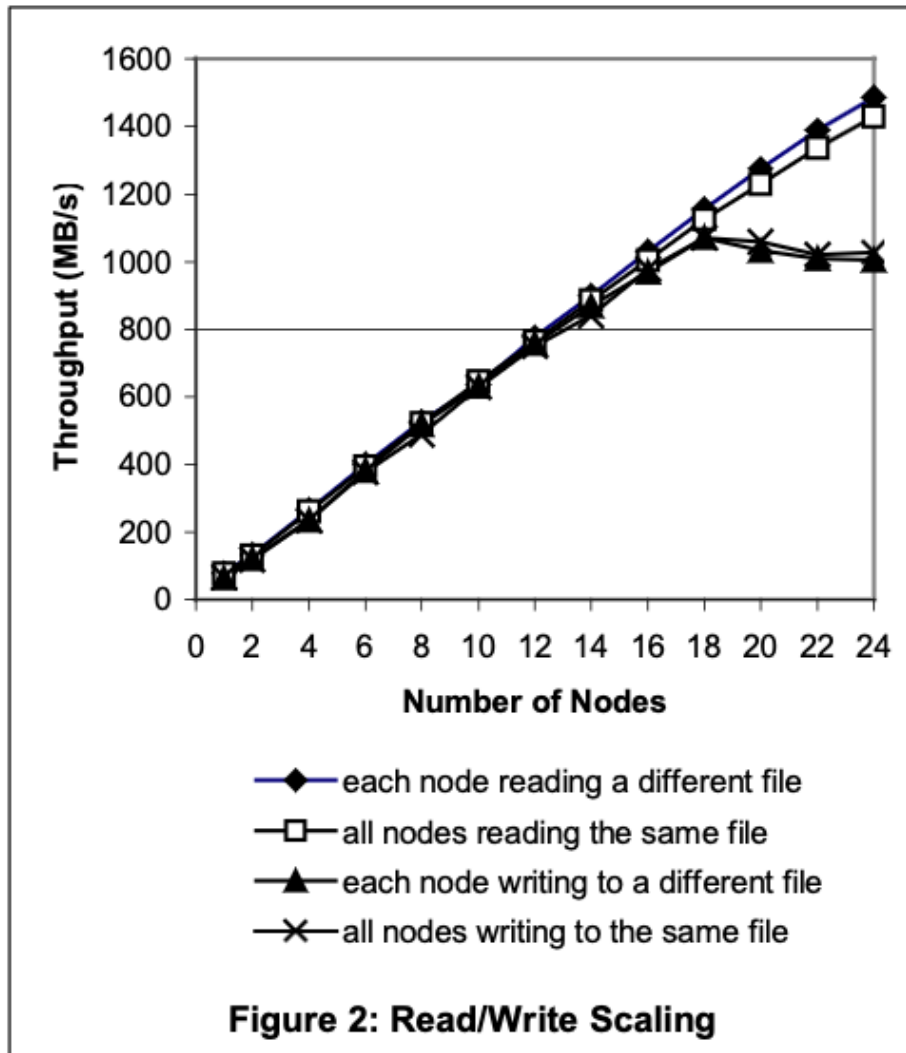
区间令牌的实现如下：

1. 第一个节点写文件时会获取整个文件范围的区间锁  $[0, \infty]$ 。在没有其他节点竞争的情况下，所有的读写操作都不需要再和其他节点交互。
2. 第二个节点写同样的文件时，发起撤销第一个节点持有的部分区间锁的请求
3. 第一个节点接受到请求后，检查当前文件是否还在写入。
  1. 文件已经关闭，第一个节点会解锁所有区间，第二个节点会获取整个文件范围的区间锁。**因此在没有并发写的共享场景下，GPFS 中的字节区间锁就是整个文件锁，单个锁就足以访问整个文件**
  2. 文件还在写入，第一个节点会解锁部分字节区间锁。如果当前是顺序写，第一个节点从 offset1 开始，第二个节点需要从 offset2 开始，当  $\text{offset1} < \text{offset2}$ ，第一个节点会解锁  $[\text{offset2}, \infty]$ ，当  $\text{offset1} > \text{offset2}$ ，第一个节点会解锁  $[0, \text{offset1}]$ 。这种方式允许两个节点写一定的范围。

通常多个节点顺序写入同一文件的非重叠部分时，每个节点只需要获取一次字节区间锁。

4. 发起区间锁的请求包含两个信息：一个是当前正在处理的 write 请求中的参数 offset 和 length，另一个是未来可能访问的范围，对于顺序写，这个范围是  $[\text{当前写入偏移量}, \infty]$ 。当访问模式允许预测被访问文件在特定 node 即将访问的区域，token 协商协议能够通过分割 byte-range token 最小化冲突，不仅仅是简单的顺序访问模式，也包括方向顺序、向前或者向后跨越式访问等访问模式。

下图对比了多节点读写同一个文件和读写不同文件的性能。在单文件测试中，文件被分为 n 个连续的区间，每个节点读或写其中一个区间。读、写性能可以线性扩展。在 GPFS 中多节点写入单个文件的速度和每个节点写入不同文件的速度相同，这说明了区间令牌的有效性。



这种方式适用于每个节点在文件的不同且相对较大的区域中写入。如果每个节点写入很多更小的区域，令牌的状态和相应的消息流将会增长。

字节区间标记保证 posix 语义，IO 的最小区间是一个扇区，字节区间令牌的粒度不能小于一个扇区。GPFS 也使用字节区间令牌来同步数据分配，因此即使单个写入操作不重叠（“虚假共享”），多个节点写入同一数据块也会导致令牌冲突。为了优化不需要 posix 语义的应用程序的细粒度共享，GPFS 允许禁用字节区间锁。

#### 元数据保护：共享写锁

和其他文件系统一样，GPFS 使用 inode 存储文件属性和数据块。

- inode 的更新。多个节点写入同一个文件会导致并发更新文件的 inode (修改文件大小、时间、分配空间)。如果 inode 独占写锁来同步，那会导致每次写操作都有锁冲突。

GPFS 在 inode 上使用共享写锁(shared write lock), 允许多个 node 的并发写入, 仅需要精确的文件大小或 mtime 的操作会冲突(如: stat 系统调用)。某个访问该文件的 node 会被指定为 metanode, 只有 metanode 可以读写 inode。其他 node 仅更新本地缓存的 inode 拷贝, 定期或者当 shared write token 被另一个 node 上的 stat() 或 read() 操作撤回时, 将其更新传给 metanode, metanode 通过保留最大 size 和最新的 mtime 合并来自多个 node 的 inode 更新。非单调更新文件大小或 mtime 的操作, 如(trunc() 或者 utimes())需要独占 exclusive inode lock 锁。

- 数据块的更新。使用类似方式的同步。

GPFS 使用分布式锁保证 posix 语义(例如 stat() 系统调用), inode 及间接快使用集中方式。这允许多个 node 写入同一个文件, 更新 metanode 没有锁冲突, 每次写操作不需从 metanode 获取信息。指定文件的 metanode 是借助 token server 动态选举的, 当一个 node 第一次访问文件, 它尝试获取该文件的 metanode token, 其他 node 学习, 当 metanode 不再访问该文件, 并且相关修改已移出 cache, 该 node 会放弃它的 metanode token, 并停止相应行为, 当它随后收到一个来自其他 node 的 metadata 请求, 它会发送一个拒绝的答复, 其他 node 会尝试通过获取 metanode token 接管 metanode 角色。

#### Allocation Map 保护

allocation map 用来记录文件系统中 disk block 的分配状态。每一个 disk block 可以划分为 32 个 subblock 存储小文件, 用 32 位的 allocation map 标记。用于查找指定大小空闲 disk block 或者 subblock 的链表。

分配磁盘空间需要更新 allocation map, 这个资源也是要在个节点之间同步的。

将 map 分成若干个固定数量的可独立锁定的 region, 不同的 node 可以从不同的 region 分配空间, 仅一个 node 负责维护所有 region 的空闲空间统计信息。

该 allocation manager node 在 mount 阶段读取 allocation map 初始化空闲空间统计信息, 其他 node 定期汇报分配回收情况。每当 node 正在使用的 region 空间耗尽时, 它会向 allocation map 请求一个 region, 而不是所有的 node 单独查找包含空闲空间的 region。

一个文件可能包含不同 region 的空间, 删除一个文件需要更新 allocation map, 其他 node 正在使用的 region 会发送到相应的 node 执行, allocation manager 会定期发布哪些 node 正在使用哪些 region, 促进构造发送释放请求。

#### 其他元数据信息保护

除了上述资源外，GPFS 还包含的全局 metadata 有：文件系统配置信息，空间分配 quota，访问控制权限，以及扩展属性。GPFS 使用中心管理节点来协调或者收集来自不同节点的元数据更新。例如：quota

manager 向写入文件的各节点分发较大的磁盘空间，配额检查在本地完成，仅偶尔与 quota manager 交互。

总结

GPFS 用户手册：  
[https://www.ira.inaf.it/centrocal/tecnica/GPFS/GPFS\\_31\\_Admin\\_Guide.pdf](https://www.ira.inaf.it/centrocal/tecnica/GPFS/GPFS_31_Admin_Guide.pdf)

GPFS 提供用户态文件接口：

Chapter 9. GPFS programming interfaces

Table 9 summarizes the GPFS programming interfaces.

Table 9. GPFS programming interfaces

Interface	Purpose
“gpfs_acl_t Structure” on page 279	Contains buffer mapping for the <b>gpfs_getacl()</b> and <b>gpfs_putacl()</b> subroutines.
“gpfs_close_inodescan() Subroutine” on page 280	Closes an inode scan.
“gpfs_cmp_fssnapid() Subroutine” on page 281	Compares two file system snapshot IDs.
“gpfs_direntx_t Structure” on page 283	Contains attributes of a GPFS directory entry.
“gpfs_fcntl() Subroutine” on page 284	Performs operations on an open file.
“gpfs_fgetattrs() Subroutine” on page 287	Retrieves all extended file attributes in opaque format.
“gpfs_fputattrs() Subroutine” on page 288	Sets all the extended file attributes for a file.
“gpfs_free_fssnaphandle() Subroutine” on page 290	Frees a file system snapshot handle.
“gpfs_fssnap_handle_t Structure” on page 291	Contains a handle for a GPFS file system or snapshot.
“gpfs_fssnap_id_t Structure” on page 292	Contains a permanent identifier for a GPFS file system or snapshot.
“gpfs_fstat() Subroutine” on page 293	Returns exact file status for a GPFS file.
“gpfs_get_fsname_from_fssnaphandle() Subroutine” on page 294	Obtains a file system name from its snapshot handle.

GPFS 对 posix 部分接口的支持：

posix	是否支持
open with O_APPEND	支持

flock/fcntl	支持多节点的 flock 支持 maxFcntlRangePerFile 个数的 fcntl()
-------------	---

## Lustre

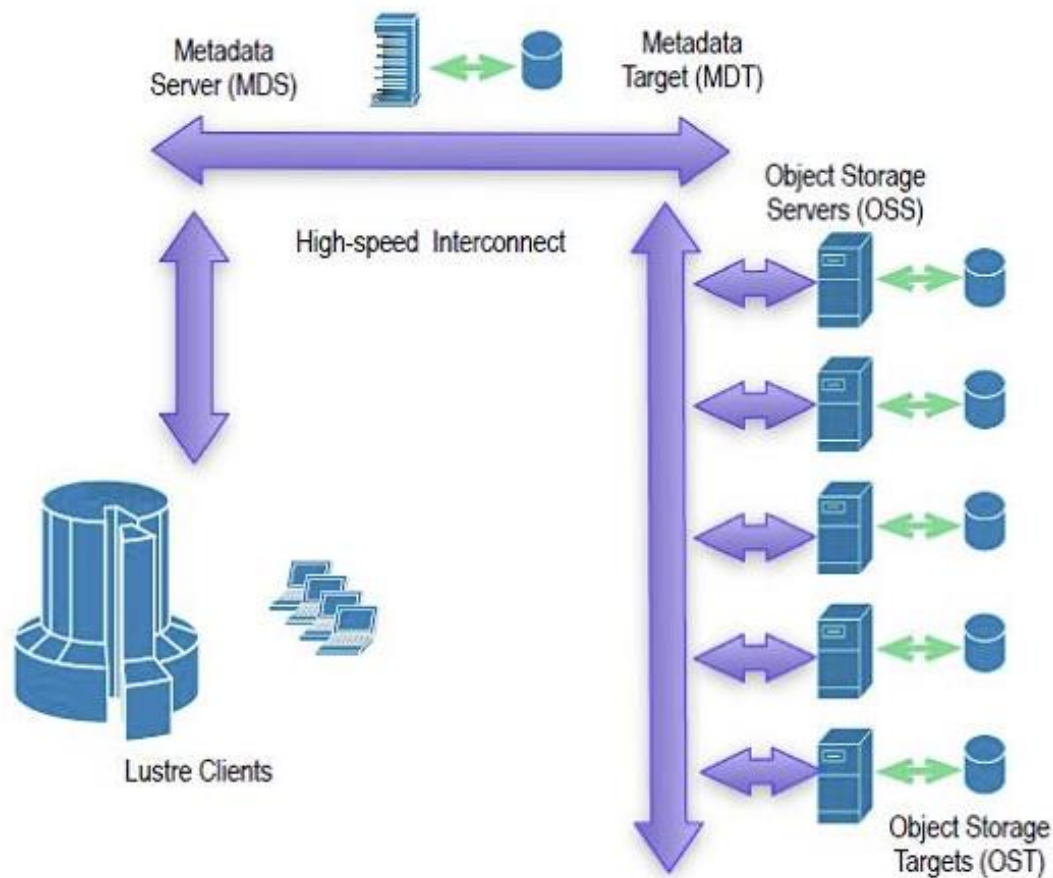
### 简介

Lustre 文件系统是一个具有高性能、高可扩展和高可靠性的基于对象的分布式文件系统。它通过 Lustre 锁管理器(Lustre Distributed Lock Manager, LDLM)技术为整个系统的共享资源提供同步访问和一致性视图。

Lustre 提供了 Posix 兼容的 UNIX 文件系统接口。Lustre 采用了基于对象存储的技术，元数据和存储数据分离的存储结构，带宽和存储容量可以随存储服务器增加而动态扩展。

Lustre 由三个部件组成：

- 元数据服务器 (Metadata Server, MDS) 维持整个系统全局一致性命名空间，负责处理 lookup、create、unlink、stat 等与文件元数据有关的命名空间操作的交互以及元数据相关的锁服务。
- 对象存储服务器 (Object Storage Server, OSS) 负责与文件数据相关的锁服务及实际的文件 I/O，并将 I/O 数据保存到后端基于对象存储的设备中。
- 客户端 (Client File System, CFS)



Lustre 文件系统中，每个节点都有 DLM 模块，为并发文件访问提供了各种锁服务。基于 DLM，Lustre 使用一种范围锁来维护细粒度的文件数据并发访问的一致性。

在访问文件数据前，客户端首先要从 OSS 获得对文件数据访问的范围锁，获得对文件访问授权；结合范围锁，Lustre 还实现了客户端数据的写回缓冲功能。

在打开文件时，客户端先和 MDS 交互，同时获得文件数据对象布局属性；在进行文件读写时，根据已获得的数据对象布局属性，在 DLM 范围锁的保护下可以直接并行地与多个 OSS 进行并行数据 I/O 交互。

### Lustre 分布式锁管理器模型

Lustre 在很大程度上借鉴了传统的分布式锁管理器的设计理念，其基本模型在 Lustre 文件系统中被称为普通锁(Plain Lock)。再此基础上，Lustre 对普通锁模型进行了扩张，引入了两种新类型锁：意图锁(Intent Lock)和范围锁(Extent Lock)。

#### 基本锁模型

Lustre DLM 模型使用锁命名空间 (Lock Namespace) 来组织和管理共享资源以及资源上的锁。当要获得某个资源的锁时，先要将该资源在锁命名空间中命名。任何资源都属于一个锁资源命名空间，而且都有一个相应的父资源。每个锁资源都有一个资源名，该资源名对其父节点是唯一的。所有的锁资源组成一个锁资源树型结构。当要获得某个资源的锁时，系统必须首先从该资源的所有祖先获得锁。Lustre 支持六种锁模式，如下：

模式	名称	访问授权	含义
EX	执行	RW	允许资源的读写访问，且其他任何进程不能获得读写权限
PW	保护写	W	允许资源的写访问，且其他任何进程不能获得写权限
PR	保护读	R	允许资源的读访问，且其他任何进程不能获得写权限但可共享读
CW	并发写	W	对其他进程没有限制，可以对资源并发写访问，无保护方式写
CR	并发读	R	对其他进程没有限制，可以对资源并发读访问，无保护方式读
NL	空	无	仅仅表示对该资源有兴趣，对资源没有访问权限

每种锁都有一定的兼容性，如果一个锁可以与已经被授权的锁共享访问资源，则称为锁模式兼容。下图是锁兼容性转换表，“1”表示锁模式可以并存。执行锁严格性最高，兼容性最差；空锁严格性最低，兼容性最好。

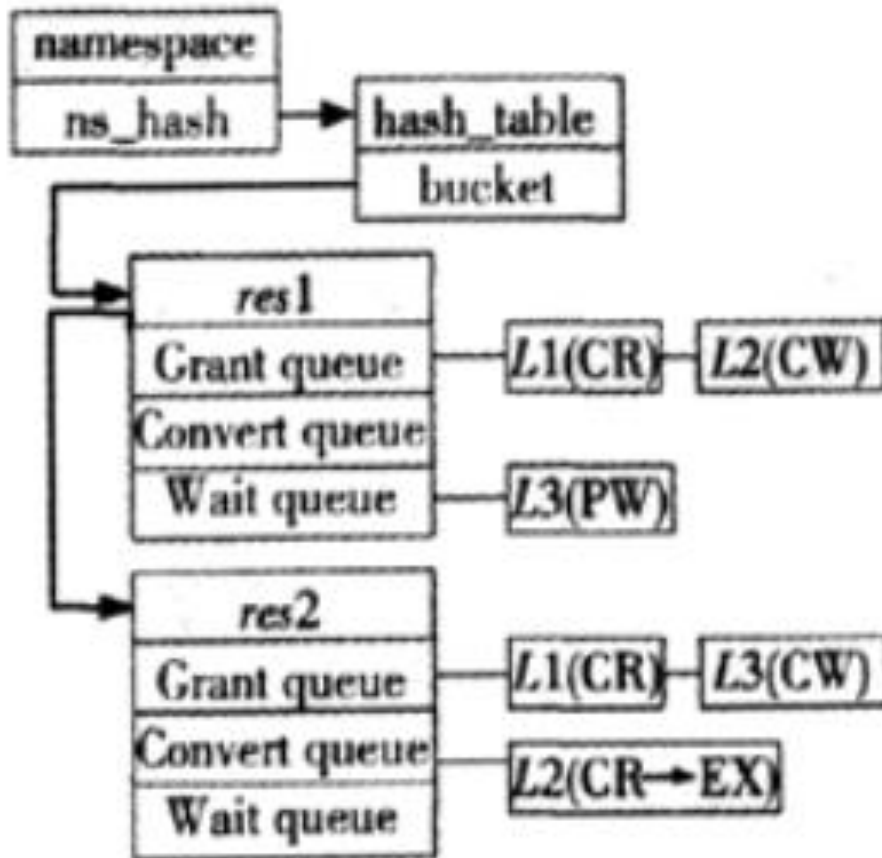
	EX	PW	PR	CW	CR	NL
EX	0	0	0	0	0	1
PW	0	0	0	0	1	1
PR	0	0	1	0	1	1
CW	0	0	0	1	1	1
CR	0	1	1	1	1	1
NL	1	1	1	1	1	1

在 Lustre 锁管理器中，锁命名空间中每个资源都维持三个锁队列：



1. 授权锁队列：该队列包含所有已被锁管理器授权的锁。但正在转换为与已授权锁模式不兼容的锁除外，该队列的锁的持有者可以对资源进行访问。
2. 转换锁队列：该队列锁已经授权，但试图转换的锁模式与当前授权队列中的锁模式不兼容，正在等待其他锁的释放或降低访问权限。锁管理器按照 FIFO 的顺序处理转化队列中的锁。
3. 等待锁队列：该队列包含所有正在等待授权的新锁请求。该队列中锁模式与已授权的锁模式不兼容。锁管理器按 FIFO 的顺序处理等待队列中的锁。

新锁请求只有在三个队列为空或者请求的锁模式与该资源所有的锁模式兼容时才会被授权，否则就会被加入等待锁队列中。如下图所示：



对于资源 res1

- CR 锁 L1 和 CW 锁 L2 是相互兼容的锁，都已经获得了访问授权；
- 新请求 PW 锁 L3 因为与锁 L1 和锁 L2 模式不兼容，所以被加入到等待队列中。

对于资源 res2

- CR 锁 L1、CR 锁 L2 和 CW 锁 L3 都获得访问授权；

- 假设某个时刻持有锁 L2 的客户请求将锁转换为 EX 模式，因与授权的 L1、L3 不兼容，所以转换请求必须等待 L1、L3 的释放才能获得锁而被加入到转换队列中。

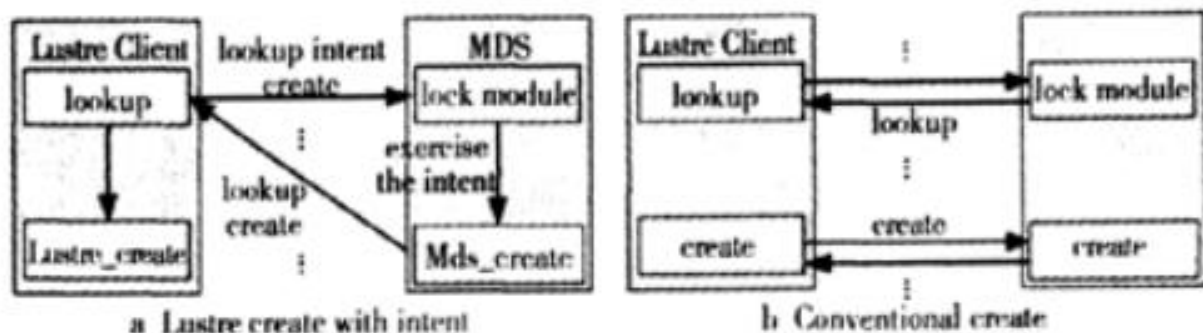
## 意图锁

Lustre 中，意图锁主要用于文件元数据的访问，它通过执行锁的意图来减少元数据访问所需要的消息传递次数，从而减少每次操作的延迟。

当客户端向 MDS 发起元数据操作请求时，在请求中指明操作意图，在交付给锁管理器处理之前，先由 MDS 执行意图锁的意图，然后返回不同锁资源。

例如，在客户端请求创建一个新文件时：

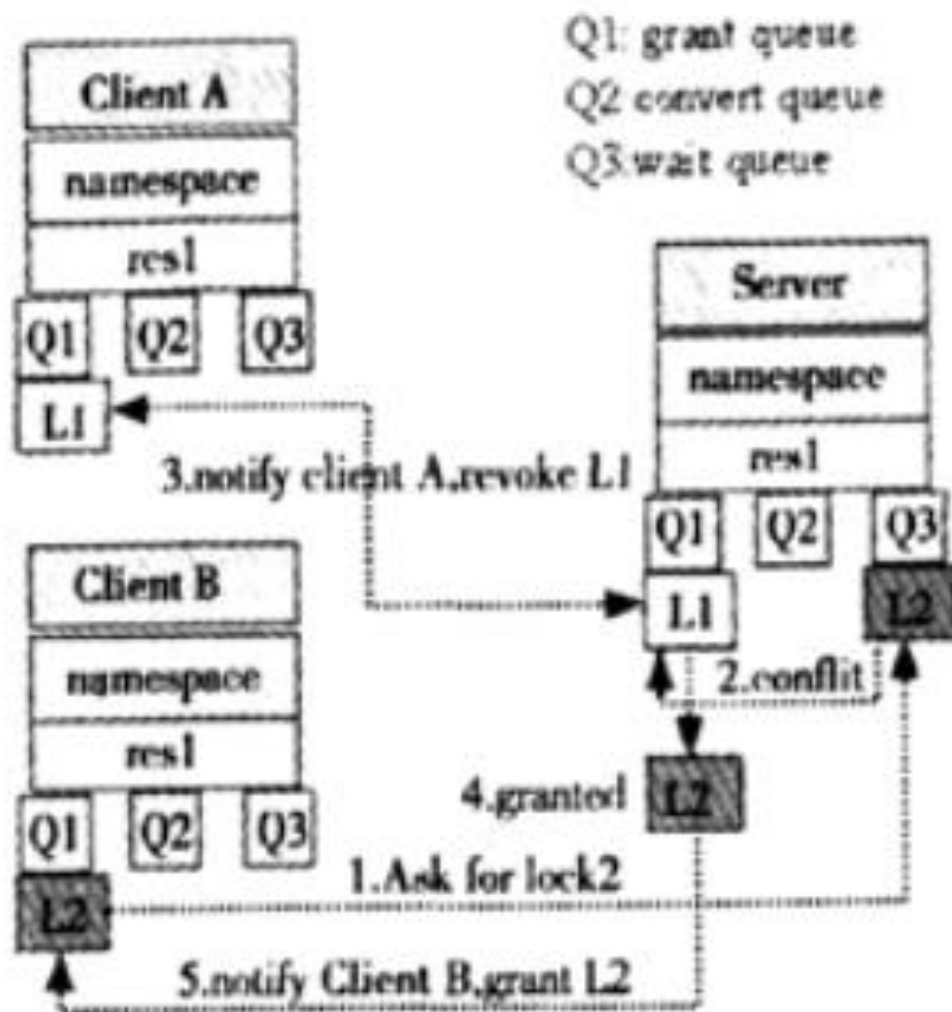
- 首先将该请求标志为文件创建的意图 (IT\_CREATE)
- 然后必须请求从 MDS 获得它的父目录的锁来执行 lookup 操作
  - 如果锁请求被授权，那么 MDS 就用请求指定的意图来修改目录，创建请求的文件，成功之后并不返回父目录锁，而是返回创建文件的锁给客户端



## 范围锁

范围锁用于维护细粒度的文件数据并发访问。其实现过程与 GPFS 文件系统采用的范围锁类似。与普通锁不同的是，范围锁增加了一个表示获得授权的锁定文件范围的域。由于同一资源的不同锁的锁定范围之间是有关联的，所以范围锁的语义也发生了一些变化。

Lustre 结合范围锁和本地锁管理器实现了文件数据的写回缓冲策略。下图展示了使用范围锁的示例：



1. 客户端 A 要对某文件的存储对象 obj 的[a, b]进行读写时, 先从 OST 上的锁服务器获得的存储对象。obj 对应锁资源的范围锁  $L \langle PW, [a, b] \rangle$ , 并将该锁资源的副本加入到本地锁管理器中
2. 客户端 A 利用本地文件系统 VFS 层的缓存机制进行读写操作
3. 客户端 A 执行完操作后并不立即将锁释放给锁服务器, 而是采用一种 lazy 的思想, 只有当其他的客户端 B 要获得的范围锁与该锁有冲突并且锁服务器通过阻塞回调函数通知 A 客户端时, 才将脏的缓存数据批处理地刷新到 OST, 并将锁释放返还给锁服务器, 然后再将锁资源授权给客户端 B

总结

Lustre 对 posix 部分接口的支持:

posix	是否支持
-------	------

open with O_APPEND	<p>支持。有如下问题：</p> <ol style="list-style-type: none"> <li>1. 每个客户端都需要对所有 OST 进行 EOF 锁定，有很大的锁开销</li> <li>2. 第二个客户端在第一个客户端完成之前不能获取所有锁，客户端只能顺序写入</li> <li>3. 为了避免死锁，它们以已知的一致性顺序获取锁</li> </ol>
flock/fcntl	<p>支持多节点的 flock/fcntl，使用“-o flock”的方式挂载，可以获取全局一致性；使用“-o localflock”可以保证本地节点上的一致性。</p>

关于 lustre, Lustre wiki 上有几个值得参考的问题：

1. Lustre 数据缓存和缓存一致性的方法是什么？  
为元数据(names, readdir, lists, inode attribute) 和 文件数据提供了完整的缓存一致性。客户端和服务端都使用分布式锁管理服务获取锁；在释放锁之前刷新缓存。
2. Lustre 是否符合 posix? 有例外吗？  
严格来说，posix 并没有说明文件系统将如何在多个客户端上运行。Lustre 合理解释了单节点 posix 要求对应到集群环境中意味着什么。  
比如：通过 Lustre 分布式锁管理器强制执行读写操作的一致性；如果在多个节点上运行的应用程序同时读取和写入文件的同一部分，它们将看到一致的结果。有两个例外：
  1. atime 的更新。在高性能集群文件系统中保持完全一致的 atime 更新是不实际的。Lustre 延迟更新文件的时间：需要更新 inode 时搭载更新 atime；文件关闭时更新 atime。客户端读取或者写入数据时，只刷新本地缓存中的时间更新。
  2. flock 和 lockf 通过分布式锁管理器实现全局一致性。
3. 为什么不使用 IBM 的 DLM?  
Lustre DLM 设计大量借鉴了 VAX Clusters DLM。尽管我们因不使用现有包(例如：IBM 的 DLM)而受到一些合理的批评，但迄今为止表明我们做出了正确的选择：它更小、更简单，并且至少对于我们的需求而言，更具可扩展性。  
Lustre DLM 大约有 6000 行代码，已被证明是一项可监督的维护任务，尽管其复杂性令人生畏。相比之下 IBM DLM 几乎是所有 Lustre 总和的大小。然而，这不一定是对 IBM DLM 的批评。值得称赞的是，它是一个完整的 DLM，它实现了我们在 Lustre 中不需要的许多功能。  
Lustre 的 DLM 并不是真正的分布式，至少在与其他此类系统相比时不是。Lustre DLM 中的锁始终由管理特定 MDT 或 OST 的服务节点管理，并且不会像其他系统允许的那样更改主服务器。省略这种类似的功能使我们能够快速开发和稳定文件系统所需的核心 DML 功能，并添加扩展（扩展锁、意图锁、策略函数等）。

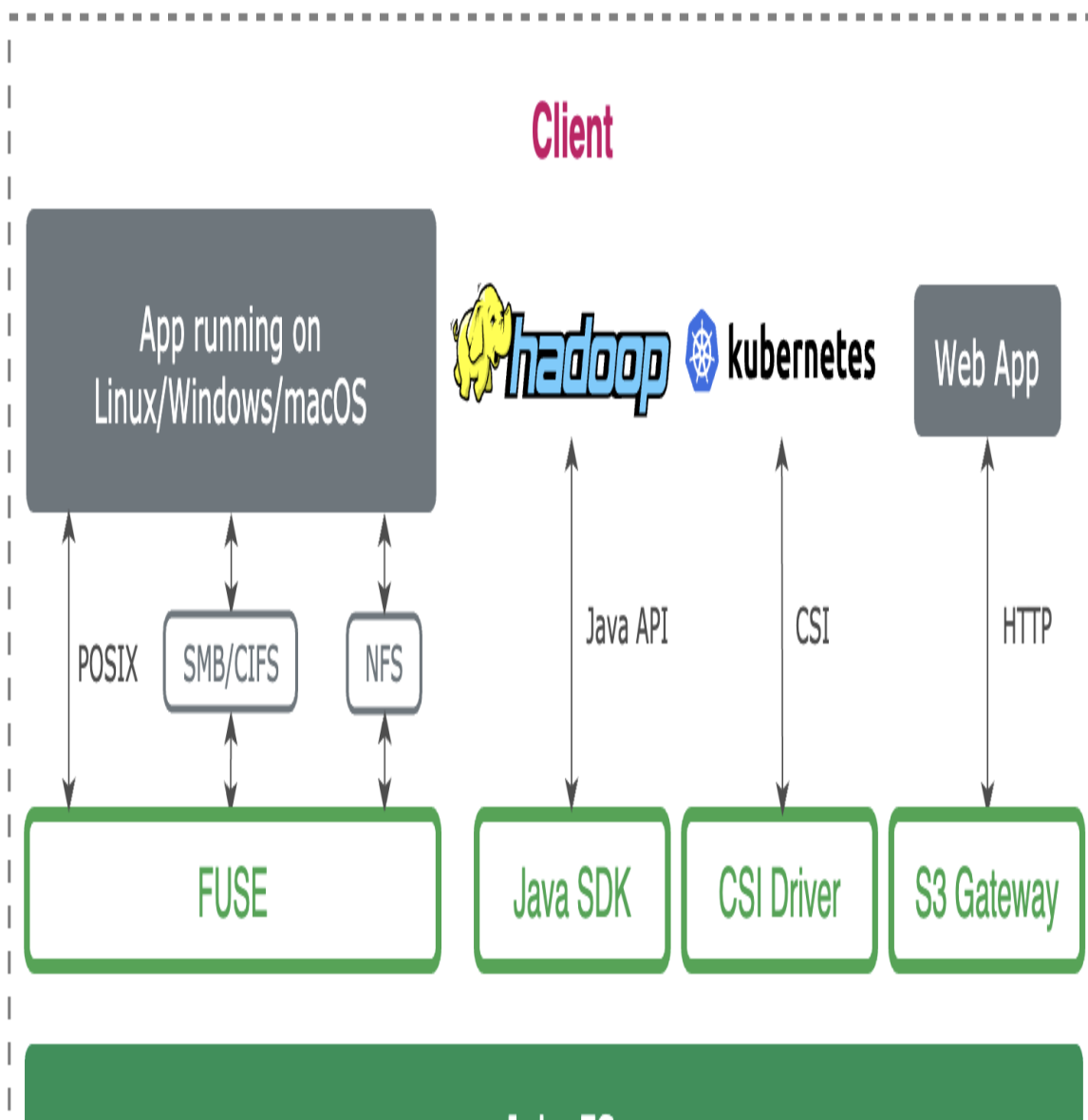
# JuiceFS

## 简介

JuiceFS 是高性能 posix 文件系统，数据本身被持久化在对象存储，元数据根据场景需求被持久化在 Redis、MySQL、SQLite 等多种数据库引擎中。

JuiceFS 由三部分组成：

1. JuiceFS 客户端：协调对象存储和元数据存储引擎，以及 POSIX、Hadoop、Kubernetes、S3 Gateway 等文件系统接口的实现；
2. 数据存储：存储数据本身，支持本地磁盘、对象存储；
3. 元数据引擎：存储数据对应的元数据，支持 Redis、MySQL、SQLite 等多种引擎；



## JuiceFS 使用 DBServer 管理锁

### flock 锁

flock 锁提供三种类型：F\_RDLCK, F\_WRLCK, F\_UNLCK

DBServer 中对于每个锁的记录如下：一条记录由 inode、sid、owner、ltype 组成。其中 sid 是 sessionid，全局递增，mount 的时候去 DBServer 申请。

因此 flock 可以对应 fuse 进程打开的 inode，可以作用到不同的节点。JuiceFS 用 flock 来同时保护数据和元数据资源。

## JuiceFS 使用 DBServer 管理锁

### flock 锁

flock 锁提供三种类型：F\_RDLCK, F\_WRLCK, F\_UNLCK

DBServer 中对于每个锁的记录如下：一条记录由 inode、sid、owner、ltype 组成。其中 sid 是 sessionid，全局递增，mount 的时候去 DBServer 申请。

因此 flock 可以对应 fuse 进程打开的 inode，可以作用到不同的节点。JuiceFS 用 flock 来同时保护数据和元数据资源。

```
type flock struct {  
    Inode Ino    `xorm:"notnull unique(flock)"`  
    Sid   uint64 `xorm:"notnull unique(flock)"`  
    Owner int64  `xorm:"notnull unique(flock)"`  
    Ltype byte   `xorm:"notnull"`  
}
```

- 用户请求 flock 锁时，客户端从 DBServer 中获取关于该 inode 的所有锁请求，并判断请求锁是否与已有锁冲突
- 如果没有冲突，将改锁插入数据库，加锁成功；如果有冲突，并且申请锁是阻塞类型，则等待指定时间后重试；如果有冲突，并且申请锁是非阻塞类型，则加锁失败。

### fcntl 区间锁

区间锁用 plock 表示。DBServer 中对于每个锁的记录如下：一条记录由 inode、sid、owner、records 组成。

其中 records 的数据结构是 plockRecord，每个 Record 由 ltype、pid、start、end 组成。

因此 plock 可以记录 fuse 进程打开的 inode 上各区间的上锁情况。

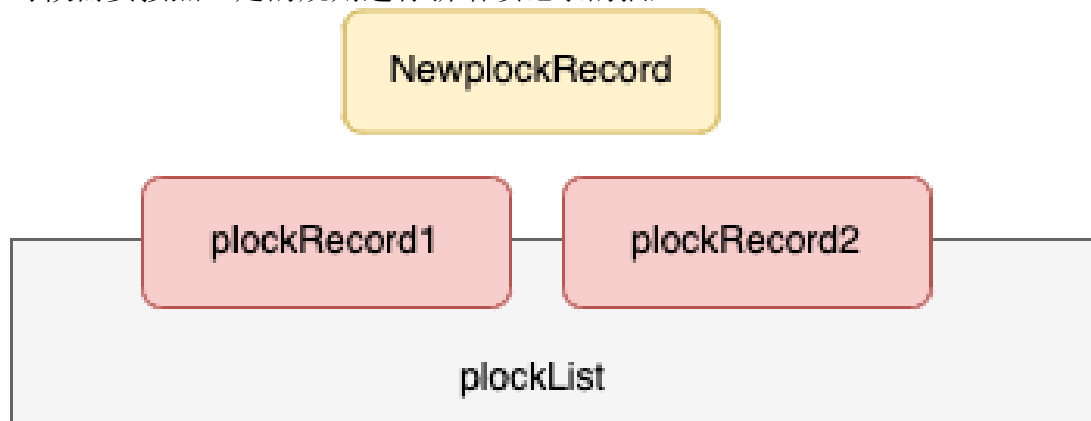
```
type plock struct {  
    Inode Ino    `xorm:"notnull unique(plock)"`  
    Sid   uint64 `xorm:"notnull unique(plock)"`  
    Owner int64  `xorm:"notnull unique(plock)"`  
    Records []byte `xorm:"blob notnull"`  
}  
  
type plockRecord struct {  
    ltype uint32
```

```

    pid    uint32
    start  uint64
    end    uint64
}

```

1. 用户请求 plock 锁的流程处理和 flock 一样，但是在检测锁冲突的时候，要和所有客户端持有的区间锁一一比较。
2. 更新锁列表有所不同，flock 的更新是一条记录的删除或者插入，plock 的更新的时候需要按照一定的规则进行新增锁记录的插入。



## 总结

JuiceFS 提供「关闭再打开（close-to-open）」一致性保证，即当两个及以上客户端同时读写相同的文件时，客户端 A 的修改在客户端 B 不一定能立即看到。但是，一旦这个文件在客户端 A 写入完成并关闭，之后在任何一个客户端重新打开该文件都可以保证能访问到最新写入的数据，不论是否在同一个节点。

「关闭再打开」是 JuiceFS 提供的最低限度一致性保证，在某些情况下可能也不需要重新打开文件才能访问到最新写入的数据。例如多个应用程序使用同一个 JuiceFS 客户端访问相同的文件（文件变更立即可见），或者在不同节点上通过 `tail -f` 命令查看最新数据。

juicefs 对 posix 部分接口的支持：

posix	是否支持
open with O_APPEND	不支持。多节点情况下会相互覆盖
flock/fcntl	支持多节点的 flock/fcntl，允许多节点写入，保证最终一致性

## ChubaoFS



CurveFS 的元数据设计架构类似 ChubaoFS，这里就不再说明了。

chubaofs 支持：

1. 一个客户端的写不能立马被其他客户端看到，fsync 才会明确去刷这个 buffer，不然只是定期刷；只有 buffer 数据刷出去了，另外的客户端才能看到。

chubaofs 不支持：

1. 多个客户端同时写同一个文件的同一个位置
2. 多个客户端的同一个文件 O\_APPEND flag

chubaofs 对 posix 部分接口的支持：

posix	是否支持
open with O_APPEND	不支持。多节点情况下会相互覆盖
flock/fcntl	不支持。chubao 使用的 bazil/fuse 不支持锁的设置  case opGetlk: panic("opGetlk") case opSetlk: panic("opSetlk") case opSetlkw: panic("opSetlkw")

## CephFS

[cephfs 的官方文档](#) 提到了 cephfs 和 posix 标准在多客户端并发写入情况下的差别：

1. CephFS 比本地 Linux 内核文件系统的 posix 语义更宽松（例如，跨越对象边界的写入可能会不一致）。在多客户端一致性方面，它严格小于 NFS，在写入原子性方面通常小于 NFS。

换句话说，当涉及到 POSIX 时：

HDFS < NFS < CephFS < {XFS, ext4}

2. 如果客户端尝试编写文件失败，写入操作不一定是原子的。也就是说，客户端可能会在使用 8MB 缓冲的 O\_SYNC 标记打开的文件中调用 write() 系统调用，然后意外终止，且只能部分应用写入操作。几乎所有文件系统（甚至本地文件系统）都有此行为。

3. 在同时发生写操作的情况下，超过对象边界的写入操作不一定是原子的。例如，写入器 *A* 写入 “aa|aa” 和 writer *B* 同时写入 “bb |bb”，其中 “|” 是对象边界，编写 “aa|bb” 而不是正确的 “aa|aa” 或 “bb|bb”。

**cephfs 使用 cap 实现分布式锁**

类似的实现有 samba 的 oplocks 以及 NFS 的 delegation。

caps 是 mds 授予 client 对文件进行操作的许可证，当一个 client 想要对文件元数据进行变更时，比如读、写、修改权限等操作，它都必须先获取到相应的 caps 才可以进行这些操作。

ceph 对 caps 的划分粒度很细，且允许多个 client 在同一个 inode 上持有不同的 caps。

根据元数据内容的不同，cephfs 将 caps 分为了多个类别，每种类别只负责作用于某些特定的元数据：

类别	功能
Pin — p	mds 是否将 inode pin 在 cache 中
Auth — A	权限属性相关的元数据，主要是 owner、group、mode；但如果是完成的鉴权是需要查看 ACL 的，这部分信息保存在 xattr 中，这就需要 xattr 相关的 cap
Link — L	inode 的 link count
Xattr — X	xattr
File — F	最重要也是最复杂的一个，用于文件数据，以及和文件数据相关的 size、atime、ctime、mtim 等

**caps permission**

权限类型	权限说明
CEPH_CAP_GSHARED	client can reads (s)
CEPH_CAP_GEXCL	client can read and update (x)
CEPH_CAP_GCACHE	(file) client can cache reads (c)

CEPH_CAP_GRD	(file) client can read (r)
CEPH_CAP_GWR	(file) client can write (w)
CEPH_CAP_GBUFFER	(file) client can buffer writes (b)
CEPH_CAP_GWREXTEND	(file) client can extend EOF (a)
CEPH_CAP_GLAZYIO	(file) client can perform lazy io (l)

caps combination

一个完整的 cap 通过[类别+permission]组成，client 可以同时申请多个类别的 caps。但并不是每种 caps 都可以使用每种 permission，有些 caps 只能搭配部分 permission。有关 caps 种类和 permission 的结合使用，有一些几个规则：

类型	说明
Pin	二值型，有 pin 就代表 client 知道这个 inode 存在，这样 mds 就一定会在其 cache 中保存这个 inode。
Auth、Link、Xattr	<p>只能为 shared 或者 exclusive</p> <ul style="list-style-type: none"><li>shared: client 可以将对应元数据保存在本地并缓存和使用</li><li>exclusive: client 不仅可以在本地缓存使用，还可以修改</li></ul> <p>下面是两个例子：</p> <ul style="list-style-type: none"><li>[A]s: 某 client 对 inode 0x11 有 As 的 cap，此时收到一个查看 0x11 状态的系统调用，那么 client 不需要再向 mds 请求，直接通过查询自身缓存并进行处理和回复</li><li>[A]x: 某 client 对 inode 0x11 有 Ax 的 cap，此时收到一个修改 mode 的系统调用，client 可以直接在本地进行修改并回复，并且在之后才将修改变更通知 mds</li></ul>

File	<p>最复杂的一种，下面是 File cap 的各个类别：</p> <p>file cap 种类 client 权限</p> <p>Fs: client 可以将 mtime 和 size 在本地 cache 并读取使用</p> <p>Fx: client 可以将 mtime 和 size 在本地 cache 并进行修改和读取</p> <p>Fr: client 可以同步地从 osd 读取数据，但不能 cache</p> <p>Fc: client 可以将文件数据 cache 在本地内存，并直接从 cache 中读</p> <p>Fw: client 可以同步地写数据到 osd 中，但是不能 buffer</p> <p>Fb: client 可以 buffer write，先将写的数据维护在自己的内存中，在统一 flush 到后端落盘</p>
------	---

## caps 管理

### lock

caps 由 mds 进行管理，其将元数据划分为多个部分，每个部分都有专门的锁（SompleLock, ScatterLock、FileLock）来保护，mds 通过这些锁的状态来确定 caps 可以怎么样分配。

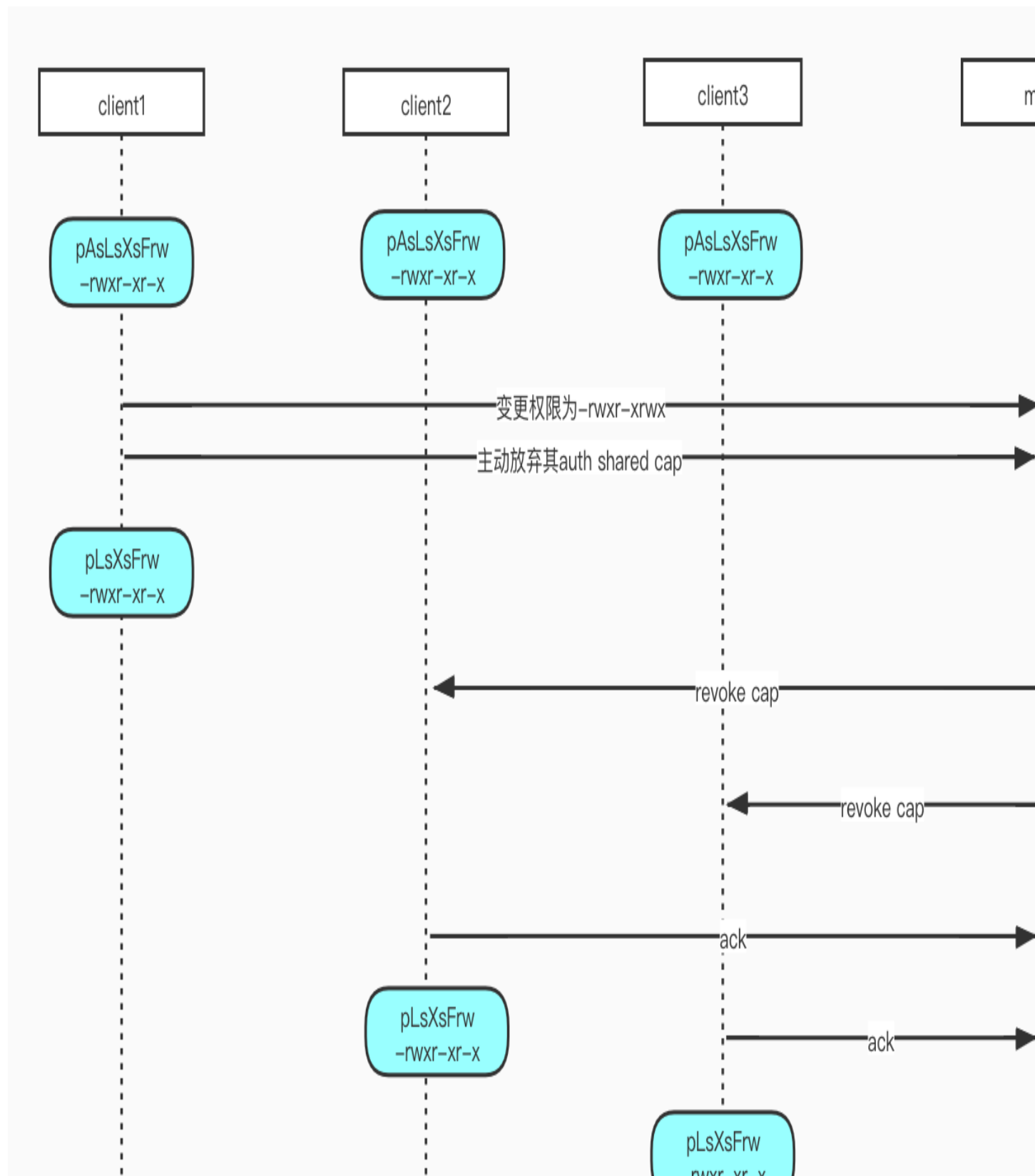
mds 内部维护了每个锁的状态机，其内容非常负责，也是 mds 保证 caps 分配准确性和数据一致性的关键。

### caps 如何变更

- mds 可以针对每个 client 进行授予和移出 caps，通常是由其他 client 的行为触发
  - 例：比如 client1 已经拥有了 inode 0x11 的 cache read 的 cap，此时 client2 要对这个文件进行写，那显然除了授予 client2 响应的写 caps 的同时，还要剥夺 client1 的 cache read cap
- 当 client 被移出 caps 时，其必须停止使用该 cap，并给 mds 回应确认消息。mds 需要等待收到 client 的确认消息后才会 revoke。（如果 client 挂掉或者处于某种原因没有回复 ack 怎么办？）
- client 停止使用并不简单，在不同场景下需要完全不同的处理：
  - 例 1: client 被移出 cache read cap，直接把该 file 的 cache 删掉，并变更状态就行，这样下次的 read 请求过来时，还是到 osd 去读

- 例 2: client 被移出 buffer write cap, 已经缓存了大量的数据还没有 flush, 那就需要先 flush 到 osd, 再变更状态和确认, 这可能就需要较长的时间

下面是一个修改权限的例子:



总结:

总结:

1. mds 需要记住所有 client pin 的 inode
2. mds 的 cache 需要比 client 的 cache 更多
3. caps 是由 mds 和 client 共同协作维护的, 所以 client 需要正常运行, 否则可能会 block 其他 client

#### fuse write 实例

以 fuse client write 为例, 简要分析下 fuse write(主要流程)时 caps 的代码逻辑:

```
int64_t Client::_write(Fh *f, int64_t offset, uint64_t size, const char *buf,
                      const struct iovec *iov, int iovcnt)
{
    want = CEPH_CAP_FILE_BUFFER;
    // 需要拥有 file write 和 auth shared caps 才(即 FwAs)能够写, get_caps 中如果检
    // 查没有 caps 则会
    // 去向 mds 申请并等待返回
    int r = get_caps(in, CEPH_CAP_FILE_WR|CEPH_CAP_AUTH_SHARED, want, &have,
endoff);
    if (r < 0)
        return r;

    // 增加该 inode 对该 caps 的引用计数并检查该 caps 是否正在使用中
    put_cap_ref(in, CEPH_CAP_AUTH_SHARED);

    // 如果有 buffer 或者 lazy io cap 则直接在 objectcacher cache 中写
    if (cct->_conf->client_oc &&
        (have & (CEPH_CAP_FILE_BUFFER | CEPH_CAP_FILE_LAZYIO))) {
        // 缓存写的调用, 异步、cache、非阻塞
        r = objectcacher->file_write(&in->oset, &in->layout,
                                     in->snaprealm->get_snap_context(),
                                     offset, size, bl, ceph::real_clock::now(),
                                     0);
    } else {
        // 如果没有 buffer cap, 则直接通过 osd 写
        if (f->flags & O_DIRECT)
            _flush_range(in, offset, size);

        // 同步写的调用
        filer->write_trunc(in->ino, &in->layout,
in->snaprealm->get_snap_context(),
```

```

        offset, size, bl, ceph::real_clock::now(), 0,
        in->truncate_size, in->truncate_seq,
        &onfinish);
    }
}

```

## 总结

cephfs 不需要用户做指定，对文件访问时都需要获取相应的权限。

posix	是否支持
open with O_APPEND	不支持
flock/fcntl	内核版本支持，fuse 客户端不确定

## NFS

### 简介

在存储系统中，NFS(Network File System，即网络文件系统)是一个重要的概念，已成为兼容 Posix 语义分布式文件系统的基础。它允许在多个主机之间共享公共文件系统，并提供数据共享的优势，从而最小化所需的存储空间。

NFS 作为类 UNIX 系统的标准网络文件系统，在发展过程中逐步原生地支持了文件锁(从 NFSv4 开始)。NFS 从上个世界 80 年代诞生至今，共发布了 3 个版本：NFSv2、NFSv3、NFSv4。NFSv4 最大的变化是有“状态”了。某些操作需要服务端维持相关状态，如文件锁，例如客户端申请了文件锁，服务端就需要维护该文件锁的状态，否则其他客户端冲突的访问就无法检测。

### 数据和元数据缓存一致性

NFS 客户端是弱缓存一致性，用于满足绝大多数文件共享场景。

- **Close-to-open 缓存一致性 (weak cache consistency, CT0)**。通常文件共享是完全顺序的，比如：clientA 进行了 open, write, close; clientB 进行 open, 然后可以读到 clientA 的写入数据。NFS client 在文件 open 的时候向 server 发送请求查询 open 权限；NFS client 在文件 close 的时候将本地文件的变更写入到 server 以便再次打开可以读取。在 mount 的时候可以通过 **nocto** 选项关闭。
- **弱缓存一致性 (weak cache consistency, WCC)**。client 的 data cache 中的数据不总是最新的。当客户端有很多并发操作同时更新文件时（多客户端异步



写），文件上的数据最终是什么是不确定的。但 NFS 为 client 端提供了接口来检查当前的数据是否被其他客户端修改过。

- **属性缓存 (Attribute caching)**。在挂载的时候加上 noac 参数保证多 client 情况下的缓存一致性，此时 client 端不能缓存文件元数据，元数据每次需要从 server 端获取。这种方式使得客户端能及时获取到文件的更新信息，但会增加很多网络开销。需要注意的是，noac 参数情况下，data cache 是允许的，因此，data cache 的一致性是不能保证的。**如果需要强缓存一致性，应用应该使用文件锁。**或者应用程序可以使用 O\_DIRECT 方式打开文件以禁用数据缓存。

客户端缓存

当应用程序共享文件的时，无论是否在同一个 client 上，应用程序都需要考虑冲突。NFSv4 支持 Share reservations 和 byte-range locks 两种实现互斥的方式。NFSv4 需要有 data cache 以支持一些应用。

**Share reservations** 是一种控制文件访问权限的机制，独立于 byte-range locking。当 client 端 OPEN 文件时，他需要指定访问类型 (READ, WRITE, or BOTH) 以及访问权限 (OPEN4\_SHARE\_DENY\_NONE, OPEN4\_SHARE\_DENY\_READ, OPEN4\_SHARE\_DENY\_WRITE, or OPEN4\_SHARE\_DENY\_BOTH)。伪代码如下：

```
if (request.access == 0)
    return (NFS4ERR_INVAL)
else if ((request.access & file_state.deny) ||
         (request.deny & file_state.access))
    return (NFS4ERR_DENIED)
```

为了提供正确的共享语义，client 必须使用 OPEN 操作来获取初始文件句柄，并指出想要的访问以及拒绝哪些访问。OPEN/CLOSE 还需要遵循一下规则：

- client OPEN 文件时，client 需要从服务器重新获取数据并判断缓存中的数据是否已经失效。
- client CLOSE 文件时，需要将所有的缓存数据持久化。

对于选择文件锁的应用，有一组类似的规则。

- 当 client 获得指定区域的文件锁后，这段区域的数据缓存失效。
- 在 client 释放指定区域文件锁之前，客户端所有的修改必须持久化。

结论

posix	是否支持
-------	------

open with O_APPEND	不支持（ <a href="https://man7.org/linux/man-pages/man2/open.2.html">https://man7.org/linux/man-pages/man2/open.2.html</a> ）
flock/fcntl	支持。并且在文件有强一致性要求时，推荐应用使用。

## 业务场景

1. 对多挂载的要求是什么？支持挂载到同一个文件系统？支持同时读写同一个文件？
2. 对一致性的要求是什么？close-to-open？还是更严格的要求？

业务	对文件系统的需求
goblinceph	<ol style="list-style-type: none"> <li>1. 大部分场景是多个节点写一个文件系统的不同文件，很少对相同文件进行读写。在对相同文件进行读写时，不要求文件系统做正确性和一致性的保证，如果业务有需求，可以调用文件锁保证；</li> <li>2. 一致性要求其实不是很强烈，正常一致性应该通过其他更专业的方案。</li> </ol>
eassyai	使用 tensorflow 平台，所以和 AI 场景的通用痛点类似的。

## AI 场景

人工智能是数据的消耗大户，对存储有针对性的需求。

AI 访问存储的几个特点：

- **海量文件**，训练模型的精准程度依赖于数据集的大小，样本数据集越大，就为模型更精确提供了基础。通常，训练任务需要的文件数量都在几亿，十几亿的量级，对存储的要求是能够承载几十亿甚至上百亿的文件数量。

- **小文件**，很多的训练模型都是依赖于图片、音频片段、视频片段文件，这些文件基本上都是在几 KB 到几 MB 之间，对于一些特征文件，甚至只有几十到几百个字节。
- **读多写少**，在大部分场景中，训练任务只读取文件，把文件读取上来之后就开始计算，中间很少产生中间数据，即使产生了少量的中间数据，也是会选择写在本机，很少选择写回存储集群，**因此是读多写少，并且是一次写，多次读。**
- **目录热点**，由于训练时，业务部门的数据组织方式不可控，系统管理员不知道用户会怎样存储数据。很有可能用户会将大量文件存放在同一个目录，这样会导致多个计算节点在训练过程中，会同时读取这一批数据，这个目录所在的元数据节点就会成为热点。跟一些 AI 公司的同事交流中，大家经常提到的一个问题就是，用户在某一个目录下存放了海量文件，导致训练的时候出现性能问题，其实就是碰到了存储的热点问题。

综上，对于 AI 场景来说，**对多挂载没有一致性需求**，主要的挑战是：

1. 海量文件的存储
2. 小文件的访问性能
3. 目录热点（这里有一个解决方案虚拟目录，后续我们解决这个问题的时候可以参考。目前 Curve 使用的是静态子树+哈希的方案）

## CurveFS 如何支持多挂载

本文主要调研了传统分布式文件系统：GPFS、Lustre、CephFS、NFS 和 面向云原生环境设计的分布式文件系统：JuiceFS 和 ChubaoFS。

上述分析可以得出：

- GPFS、Lustre、NFS、CephFS 都支持多节点的读写同步，提供数据和元数据多个级别的一致性。
- JuiceFS 使用中心节点管理锁，支持多节点读写同步，保证最终一致。
- ChubaoFS 支持多节点写不同区域，保证最终一致。不保证多节点写同一区域的一致性。

**对于以上四中场景，curvefs 在并发读写场景下需要支持何种一致性？如何支持？是本文需要得出的结论。**

1. CurveFS 需要支持写写互斥，读写互斥？
  1. posix 文件接口的语义是针对多进程情况下的读写并发行为，并没有规定多节点下的读写并发行为。

2. NFS 给出了读写并发行为下一致性的保证：close-to-open 缓存一致性、弱缓存一致性(提供接口检查是否被其他客户端修改过，其他不做保证)、属性缓存(挂载的时候指定 noac)
  3. **所以 CurveFS 无需默认支持写写互斥、读写互斥。CurveFS 对读写并发行为的规定如下：**
    1. close-to-open 一致性并提供开关(cto/nocto)。close 完再打开文件，一定可以读到全部的最新数据。
    2. 支持文件锁 fcntl、flock。让用户可以通过该接口实现并发读写情况下的强一致语义。（O\_APPEND 可以不做支持，因为性能必然很差，实际估计不会应用。）
    3. 更多挂载选项支持：cto/ncto（是否开启 cto，ncto 可以提高只读挂载的性能）、ac/noac（是否开启元数据缓存）、local\_lock（non、posix、flock、all 是否仅支持本地锁）
2. 如何实现？
1. cto 一致性，cto/ncto 开关。ncto 时打开文件允许缓存不更新。
  2. flock、fcntl  
实现可以参照：LDML（参考资料：[Understanding Lustre Filesystem Internals](#)）、NFS（[NFS: Opens and Byte-Range Locks](#)）、JuiceFS（源码阅读）。
  3. 更多挂载选项支持：可以参考 NFS。

## 参考文档

1. [GPFS: A Shared-Disk File System for Large Computing Clusters](#)
2. [GPFS—三大关键组件](#)
3. [Lustre 分布式锁管理器的分析与改进](#)
4. [文件系统那些事-第 4 篇 并行文件系统之开源解决方案 Lustre](#)
5. [Lustre wiki](#)
6. [juicefs/docs/zh cn/cache management.md](#)
7. [被遗忘的桃源——flock 文件锁](#)
8. [多进程之间的文件锁](#)
9. [juicefs readme](#)
10. 多写文件系统与 DLM 设想



多写DLM. pdf

11. [CEPH 文件系统限制和 POSIX 标准](#)
12. [CephFS Caps](#)
13. [CephFS caps 简介](#)
14. [What are “caps” ? \(And Why Won’ t my Client Drop Them?\) - Gregory Farnum, Red Hat](#)
15. [CAPABILITIES IN CEPHFS](#)
16. [cephfs: 用户态客户端 write](#)
17. [NFS 文件锁一致性设计原理解析](#)
18. [Network File System \(NFS\) Version 4 Protocol](#)
19. [AI 场景的存储优化之路](#)
20. [man5nfs](#)
21. [Lustre 文件系统 I/O 锁的应用与优化](#)
22. [Lustre & Application I/O](#)