CurveFS方案设计（总体设计，只实现了部分）

时间	修订人	修订内容
2021-03-23	李小翠	初稿(背景，调研，架构设计)
2021-03-30	李小翠	增加快照部分
2021-04-13	李小翠、陈威	补充元数据数据结构
2021-04-19	李小翠、吴汉卿、许超杰等	补充文件空间分配，讨论与确认

背景

为更好的支持云原生的场景，Curve需要支持高性能通用文件系统，其中高性能主要是适配云原生数据库的场景。当前Curve是实现了块存储，向上提供块设备服务，CurveFS会基于此实现。第一阶段的目标是实现满足数据库场景的文件接口。

调研

开源fs

当前对已有的开源分布式文件系统进行了调研，主要包括系统架构，元数据内存结构，元数据持久化，调研文档如下：

chubaofs： ChubaoFS

moosefs: https://kms.netease.com/team/km_curve/article/27786

fastcfs: https://kms.netease.com/team/km_curve/article/29140

cephfs: https://kms.netease.com/team/km_curve/article/27909

性能对比

并对以上文件系统在相同环境进行了元数据节点性能测试：调研测试。测试结果c开发的moosefs和fastcfs元数据性能远优于go开发的chubaofs和c开发的cephfs，理论上分析这个结果是合理的，分布式的元数据设计会涉及到多次rpc的交互。这里需要确认的一点是：我们需要怎样的元数据节点的性能？

可行性分析

方案对比

根据上述调研和测试结果，我们考虑了三种curvefs的元数据设计方案：

1. CurveFS kv方案设计
   curve实现块设备时，元数据不是扁平化的设计，而是采用来有目录层级的 namespace 方式，namespace 已经实现了 fs 元数据管理的雏形，具备了基本的元数据管理功能。（当时为什么要设计为 namespace 的管理形式？留有租户这个概念），直接基于 namespace 开发：
   a. 功能
      软/硬链接：目前是都不支持的。软链接可以通过标识文件类型解决；由于 prefix + parentid + filename 作为 key , filename 直接和 fileInfo 关联，硬链接无法支持
   b. 性能
      list：list在通用文件系统中是很常见的操作，目前 curve 的元数据缓存使用的 lru cache，因此 list 只能依赖 etcd 的 range 获取方式。如果需要对 list 加速，需要新的缓存结构
   c. 扩展性/可用性/可靠性
      依赖于第三方kv存储，目前是etcd

2. CurveFS 单机内存元数据设计
   类似 fastcfs 和 moosefs 的元数据设计方式，采用通用的 dentry，inode 两层映射关系，所有的元数据都缓存在内存中，持久化在 binlog 文件中，binlog采用定期dump的方式删除。基于这种方式的开发：
   a. 性能
      加载：数据量较大的情况下，元数据节点启动较慢；但是元数据使用 master-slave 可以降低 failover 情况下的加载时间
   b. 扩展性/可用性/可靠性
      扩展性不够，受限于单机的内存和磁盘，只能纵向扩展
      可用性足够，由于是 master-slave 的方式，master 以同步方式调用 slave，slave 在内存中也缓存了全部元数据信息
      master-slave 多副本数据

3. CurveFS 分布式元数据设计
   类似 chubaofs 的元数据设计方式，同样是采用 dentry，inode 两层映射关系，所有的元数据都缓存在内存中。元数据是分片的，使用 multi-raft 持久化元数据以及保证多副本数据一致性。基于这种方式开发：
   a. 性能
      由于元数据分片，获取元数据需要跟多个节点进行rpc的交互，因此性能相比单机要弱一些
   b. 扩展性/可用性/可靠性
      使用 multi-raft，扩展性、可用性和可靠性与元数据节点一致

对比结论

CurveFS 近期要能支持mysql所要接口，长期需要支持通用文件接口。

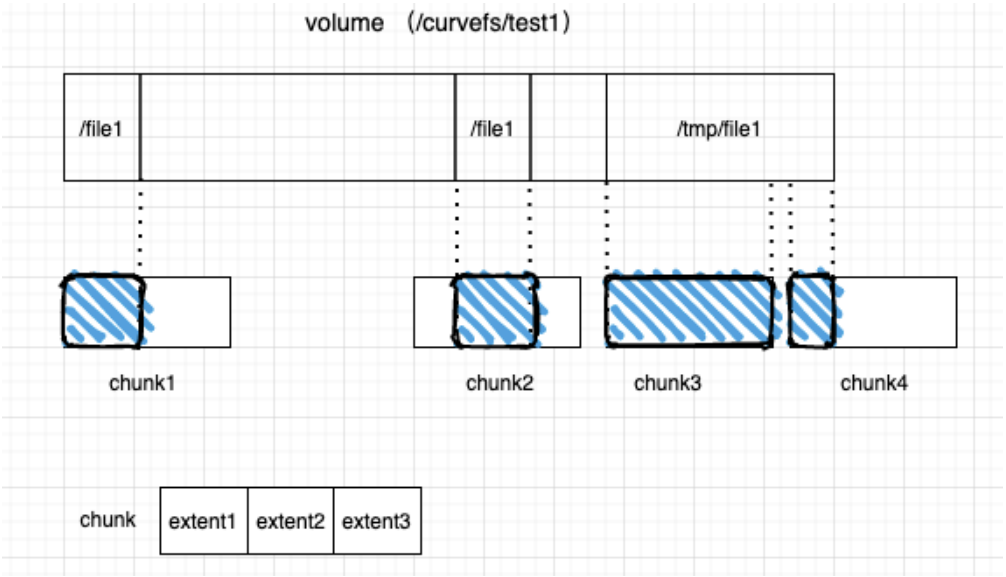
kv 虽然改造简单，短期内对基本功能的支持没有问题，但这个架构不利于 Curve 长期的规划和演进，因此选择通用的 dentry，inode 两层映射的元数据结构。对于 fs

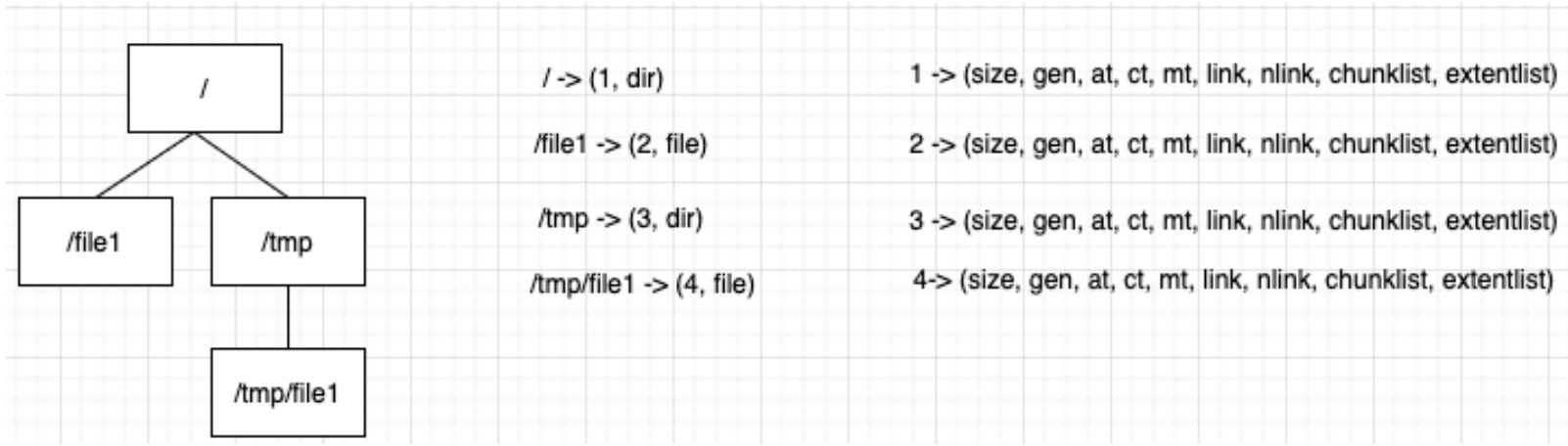
的场景，元数据的量比块存储场景会多很多，长期看元数据节点的设计也是需要满足高可用、高可扩、高可靠的。

因此对元数据节点的要求总结为：高可用、高可扩、高可靠、高性能。

架构设计

卷和文件系统





一个卷对应一个文件系统

1. 文件系统中文件数据和chunk是一对多的关系。
    1. 底层 chunk 固定大小，一个 chunk 可以分为多个固定大小的 extent
    2. 大文件可以包含多个 chunk， 小文件可以共用 chunk
2. 文件的目录数结构有单独的元数据节点存储
    1. 元数据包含两层映射，dentry，inode
    2. inode 在每个文件系统中是全局唯一的，inode 中包含文件的信息，包括用户，时间，软/硬链，数据分布等

元数据架构

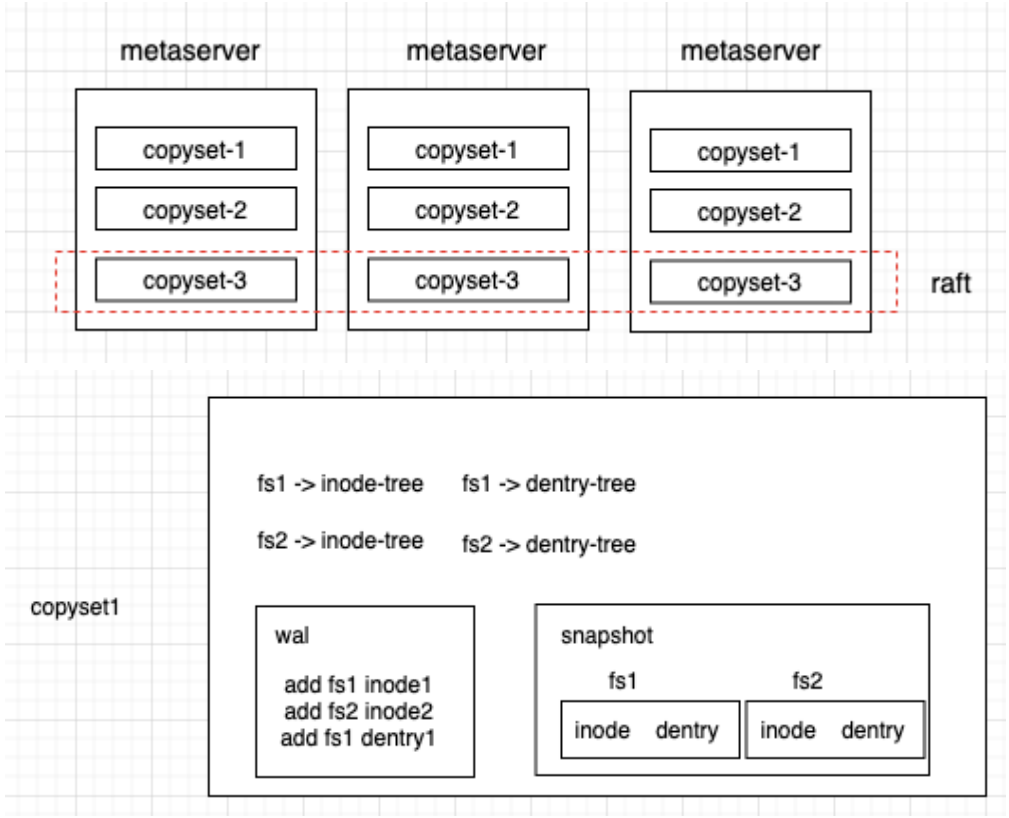
元数据包含两个部分

1. 卷的元数据管理
   这部分 mds 已经实现。在上面架了一层文件系统后，卷信息中还需要包含文件系统元数据的路由信息

2. 文件系统的元数据管理
   需要记录 dentry，inode 这两层元数据。包括内存结构和持久化结构

下面先介绍文件系统的元数据管理，再介绍卷的元数据管理的变化

元数据节点的架构如下



metaserver: 元数据服务进程。一个进程管理多个复制组

copyset: 复制组，使用 raft 保证数据一致性。复制组中保存文件系统的部分元数据信息

文件系统元数据和复制组是多对多的关系

1. 一个复制组可以包含多个文件的元数据信息
    1. 复制组 wal 记录元数据操作
    2. 定期 snapshot 对 wal 进行清理。snapshot 中存储的是键值对，其中 inode 文件中存储 inode-inodeInfo; dentry 中存储 filename-dentryInfo 信息
    3. copyset 启动的时候根据 inode 和 dentry 分别建立对应的内存结构，再回放 wal 日志完成构建

卷的元数据管理

卷的元数据中需要包含建立在该卷之上的文件系统元数据分片的位置，以便进行元数据的索引

常见的元数据操作

1. Create
    1. 与 mds 交互获取 inode 和 dentry 的 copyset 位置
    2. 创建 inode

    3. 创建 dentry
2. Mkdir
    1. 与 mds 交互获取 inode 和 dentry 的 copyset 位置
    2. 创建 inode
    3. 创建 dentry
3. Lookup （/A/B）
    1. 与 mds 交互获取 /(inodeid=1) 所在的 copyset
    2. 根据 parent-inode=1 和 name=A 获取对应的 dentry，从而获取到 /A 的 inode
    3. 根据 /A 的 inodeId 查询 /A/B 所在的 copyset
    4. 根据 parent-inode=* (/A的的inodeid) 和 name=B 获取对应的 dentry，从而获取到 /A/B 的 inode
4. ReadDirAll （/A/B）
    1. 先获取 /A/B 的 inode 所在的位置，即可在同一个 copyset 中获取所有的子项信息
5. Rename（/A → /B）
    1. 获取 /A 所在的 copyset
    2. /A 对应的dentry新增计数
    3. 创建 /B 节点
    4. 删除 /A 节点
6. Symlink
    1. 创建新的inode节点，dentry中标明符号链接
    2. 实际数据保存链接到的路径
7. Link
    1. 创建新的dentry，指向同一个inode

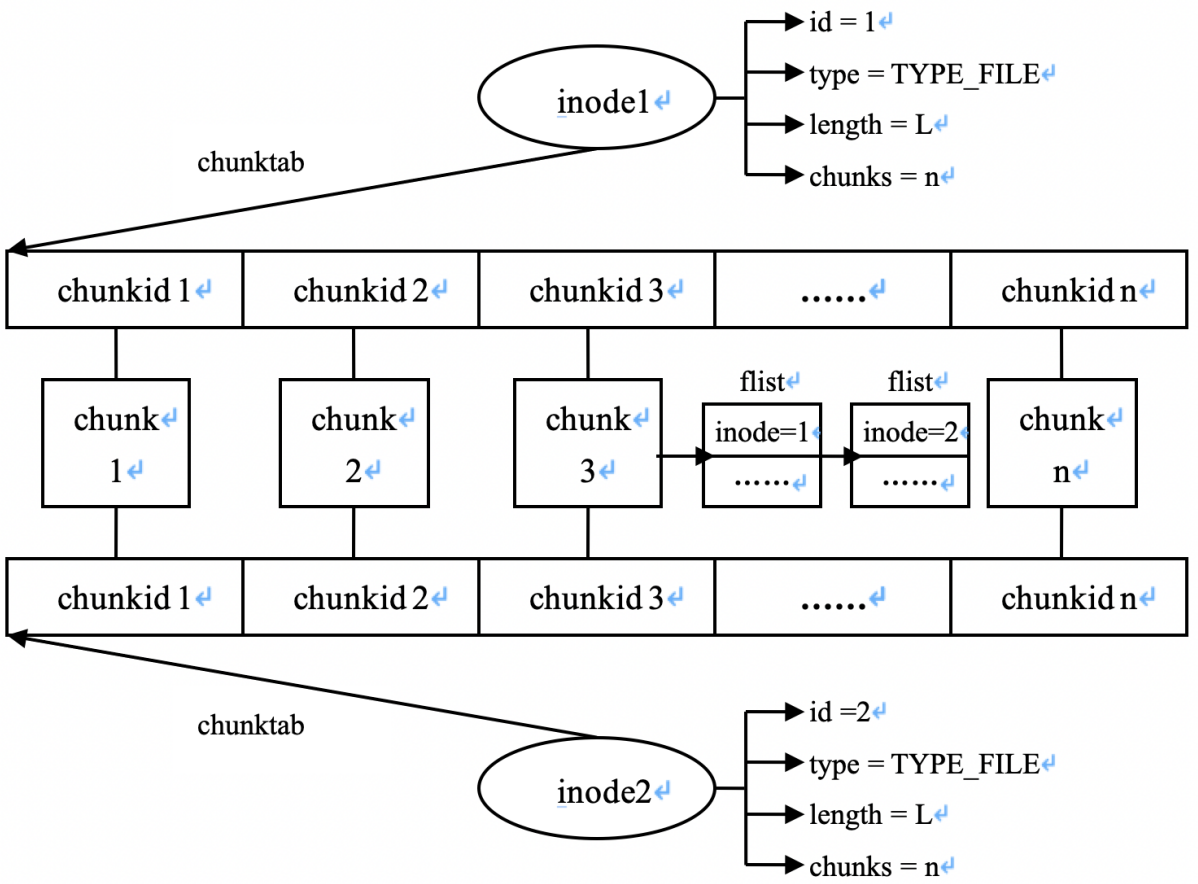
文件系统快照

方案一：文件/目录级别快照

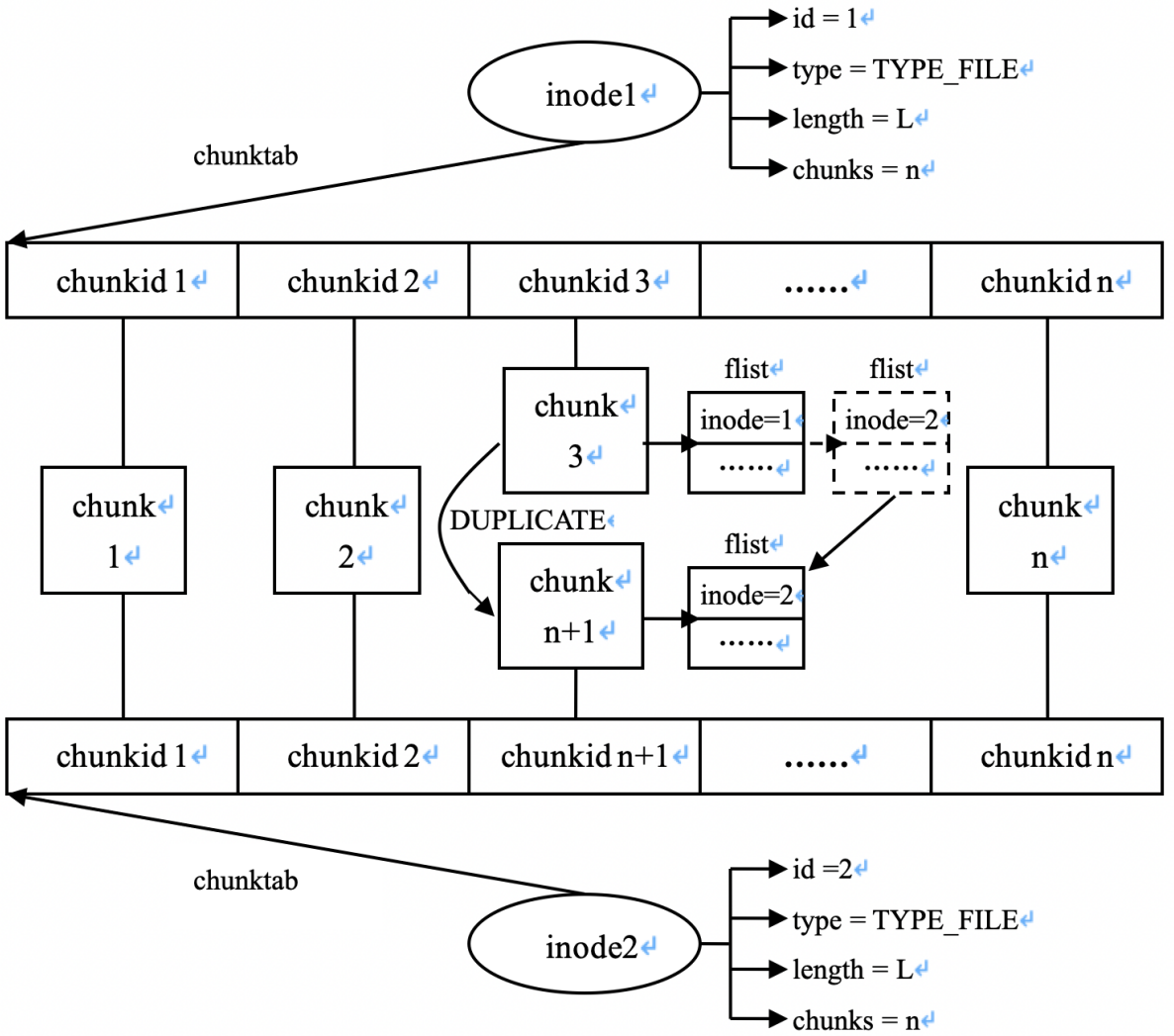
快照是文件系统或卷的只读副本，快照要求可以即时创建。类似 moosefs，curvefs 可以计划支持目录及文件级别的快照，目录级别和文件级别的快照可以认为就是cp的实现。

对于文件/目录级别的快照：

- 检查目的节点的父节点中是否有同名文件存在：
    - 存在
        - 若源节点类型为TYPE_DIRECTORY则对源节点目录下的所有子文件进行快照
        - 若源节点类型为TYPE_FILE则开始比较源节点与目的节点的 inode 是否相同，若完全一样则说明目的节点已经是源节点的快照了不需要做任何处理，否则删除目的节点，再创建新的 dentry 指向源节点的 inode
        - 若源节点类型为TYPE_SYMLINK，重新设置目的节点与源节点保持一致
    - 不存在
        - 创建新的dentry
        - 若源节点type= TYPE_DIRECTORY，递归创建源节点目录下的所有子文件进行快照
        - 若源节点type= TYPE_FILE，则设置length、chunks使其与源节点的对应属性一致
        - 若源节点类型为TYPE_SYMLINK，设置目的节点的path与源节点保持一致
        - 为 dentry 中对应chunk添加一个快照结构 flist 表示该chunk新关联了一个fsnode


inode1
id = 1
type = TYPE_FILE
length = L
chunks = n
chunktab
chunkid 1
chunkid 2
chunkid 3
.....
chunkid n
chunk 1
chunk 2
chunk 3
flist
inode=1
.....
flist
inode=2
.....
chunk n
chunkid 1
chunkid 2
chunkid 3
.....
chunkid n
chunktab
inode2
id =2
type = TYPE_FILE
length = L
chunks = n

快照的cow



优势在于可以做lazy-copy，速度很快。

劣势在于当前的快照逻辑复用需要做较大变动

方案二：文件系统快照

复用当前的逻辑，文件系统快照就是当前卷的快照，因此数据的快照就已经有了，需要的就是元数据的快照。

由于元数据使用raft, apply的时候是以kv的方式写入到文件，因此可以复用这个逻辑。

- 客户端感知文件版本号。如果版本号变更，就触发raft的sanpshot，并且只apply小于版本号的部分
- 这种方式相当于每次都是全量缓存当前元数据，不做增量快照，考虑到转储逻辑，这也是可以接受的

对比这两种方案，第一种方案对于copy场景是友好的，但需要重新实现一套快照逻辑；第二种方案的改动和实现相对简单，并且对于需要备份的场景也是够用的。从可解决程度和解决的必要性考虑，选择第二种方案。

关键点

1. mds
    1. volume
    2. 文件空间管理
    3. 文件系统的元数据所在的copyset分配策略（前期可以考虑都分配到同一个copyset上）
2. metaserver
    1. inode/dentry的内存组织形式
    2. 数据持久化
3. client
    1. curvefs 的 client 开发
4. 快照逻辑
5. 各接口实现元数据交互流程

元数据设计

元数据设计分以下几个部分

1. inode 和 dentry 的数据结构（ inode 和 dentry 两个结构描述 还是 由一个dentry描述所有信息）
2. inode 和 dentry 的索引设计（btree / skiplist / hashmap ?）
3. 元数据的持久化（以 kv 的方式存入文件？存储 rocksdb ?）
4. 元数据节点的高可用
5. 元数据分片策略（哪些范围的元数据存储在哪些复制组上）

数据结构

在元数据设计上，扁平化元数据（用 parentID+Filename → FileInfo 表示一个文件）和分级元数据（ParentID+Filename → Inode; Inode → FileInfo）最大的区别在硬链接的实现上。扁平化元数据无法做到共用同一个数据区域，对于硬链接的实现很不友好。

根据之前的调研，分级元数据可以分为两种实现方式。一是类似 fastcfs 把 inode 和 dentry 合并为一个 dentry 的结构，dentry 中包含一些指针，通过指针实现数据的共享；第二类类似 moosefs 和 chubaofs，使用 inode 和 dentry 两种结构，dentry 中记录 parentID、InodeID、filename 等信息，inode 中记录文件空间占用、文件属性等信息，通过共享 inodeID 实现数据共享。curve的文件元数据管理设计为分布式的，因此第一种通过内存实现数据共享的方式并不适用，我们选择第二种方式。具体的元数据结构设计：Curve文件系统元数据管理（已实现）

索引设计

文件空间管理

文件空间管要解决的问题是：一个文件的数据如何存储？物理空间如何分配给不同的文件，如何从不同的文件回收？从这两个角度出发，分别调研了以下系统的空间管理策略：

1. bluestore: CurveFS空间分配调研#bluestore
2. chubaofs: CurveFS空间分配调研#chubaofs
3. moosefs: moosefs空间分配调研
4. polarfs: CurveFS文件存储设计参考方案（元数据存储在卷上方案）#PolarFS

上述fs可以分为两类

- chubaofs/moosefs 属于利用本地文件系统去构建分布式fs。一个文件的数据对应本地文件系统上的一个文件，通过本地文件系统的打洞功能实现部分空间的回收。
- blustore/polarfs 直接在块设备上构建分布式fs。一个文件的数据对应块设备上某个空间，因此需要知道块设备的哪些空间是空闲的，哪些是已经分配出去的，需要一个空间分配管理器。blustore有两个空间分配器bitmap和stupid。polarfs开源部分有空间映射关系，但空间分配器没有公布。

当前curve已经实现了块设备。curve的数据节点采用了chunkfilepool实现性能优化，同时也绕过了文件系统的空间管理，通过mds的segment/chunk实现了简单的空间管理。

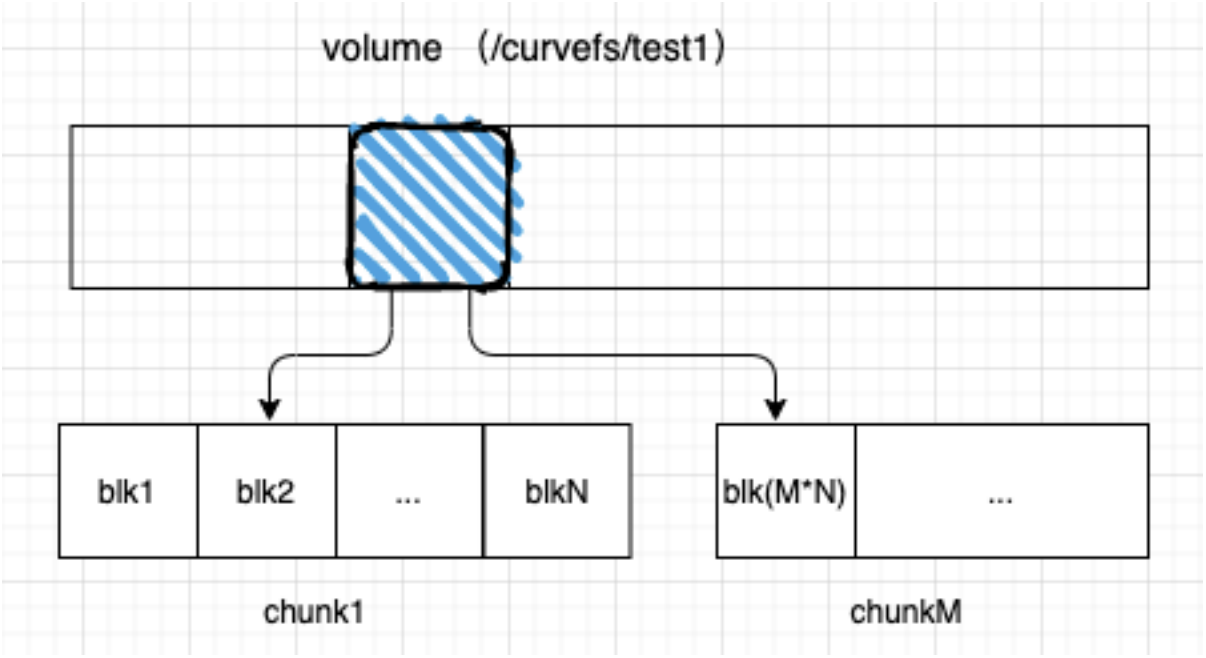
- 利用本地文件系统去构建curvefs，采用分布式文件和本地文件系统相对应的方式，curve当前数据节点需要变更。因为chunk16MB的定长不适用于文件系统。另外快照逻辑无法复用。
- 利用已有的块设备构建curvefs，需要实现空间分配器，可以复用快照逻辑做文件系统级别的快照。

对比两种方案：首先curve设计的初衷是提供一个存储底座，在这个底座上构建文件、块、对象等，第一种方式相当于重新开发了一套文件系统，并没有用到块设备的能力。另外第一种方式虽然更加灵活，在空间管理方面更加简单（本地文件系统已经进行了空间管理），但数据层面需要重新设计，工作量是比较大的。因此我们选择基于块设备构建curvefs。

空间管理设计如下：

inode → blk_list {blk1, blk(M*N)}

在文件系统mount的时候，读取所有inode的信息就可以重建出当前哪些block是已经分配的，哪些未分配，因此空间分配信息的表无需另外做持久化。这一信息可以缓存在 client 或者 metaserver。



1. blk的粒度为多少？
   从调研的系统来看，如果chunk是固定的分配力度，会选择64k，以一个20TB的盘为例：
   blk=4k，需要bitmap的大小为640MB
   blk=64k，需要bitmap的大小为40MB
   在设计过程中，对于每个文件系统，blk应该是可以根据业务形态来配置的
2. bitmap重建时间？
   如果通过获取所有inode，重建出当前的空间分配情况，我们常见的业务形态有以下两种：
   ① 在AI训练等场景，文件的目录层级较少，文件数量较多，文件较小。这种情况inode比较聚集，一般分布在几个复制组上。inode数量多。
   ② 在数据库等场景，文件的目录层级较少，文件数量较少，文件很大。这种情况inode比较聚集，一般分布在几个复制组上。inode数量少。
   以上这两种情况，以20TB为例，数据量在MB级别，client获取数据可以使用stream类型的rpc？
   ③ 正常使用场景，有一定的目录层级，文件分配数量较多，文件较小。这种情况inode比较分散，一般分布在多个复制组上，inode数量较多。
   client获取数据可以并发从多个复制组中获取。
3. 数据结构的选取？
   考虑类似bluestore建立多层索引？CurveFS空间分配调研#bluestore

开发计划及安排

2021M1			
事项	备注	时间	人员

元数据节点 1. dentry/inode 2. 数据结构	内存结构确认	2021-05-13	@陈威
	代码框架完成，主要涉及接口对接	2021-05-20	@陈威
	开发完成	2021-05-28	@陈威等
空间分配	空间分配方案确认	2021-05-14	@吴汉卿
	代码框架完成，主要涉及接口对接	2021-05-20	@吴汉卿等
	模块开发完成	2021-05-28	@吴汉卿
curvefs client端	主要接口及流程梳理和确认	2021-05-19	@许超杰
	代码框架开发	2021-05-27	@许超杰
	主要接口代码开发完成	2021-06-09	@许超杰等
联调		2021-06-10起	@所有人