**可活化的半中心NFS分布式存储系统（DNFS）**

# 方案选型

智能平台依赖分布式存储系统搭建完整的存储管理支撑，主要需求如下：

1. **离散存储资源聚合：**将所有任务服务器中的可用磁盘聚合形成完整的庞大的存储池，以获得更加庞大的存储支撑；
2. **文件系统存储热扩展：**能够简单的选择一个目录作为存储节点，在分布式存储系统运行状态下实现热扩展，不影响业务的常态化运行；
3. **兼容Docker动态NFS挂载：**经过测试Docker对NFS的挂载支持并不依赖于主机的NFS客户端支撑，而来自于本身的代码封装，同时仅支持V3版本的NFS协议；
4. **物理与虚拟存储卷：**存储管理的核心在于分发虚拟磁盘给不同的用户，实现存储份额的管理，物理卷上搭建虚拟存储卷的默认支持非常关键；
5. **数据传输直连：**在NFSV3协议基础上实现类pNFS协议的直连操作可以大幅降低客户端与数据端的网络距离，提升数据访问效率；
6. **遥感数据智能解译针对性优化：**在底层数据层面对数据读写形式进行优化，实现对数据密集型的遥感智能解译与训练任务加速。

针对上述需求，目前有两种合适的方案选型：第一种方案使用GFS+nfs-ganesha的方式实现分布式存储，可以满足大部分需求，但是不能支持数据传输直连，属于中心化结构，对主服务器带宽要求较高，并且不支持快速的虚拟存储卷的建立与管理；第二种方案是搭建基于pNFS协议的客户端与服务端程序，抛弃了Docker动态NFS挂载的支持，如果需要使用必须在主机挂载目录后以host的形式挂载到Docker内部，对平台的复杂度有不良影响。本次则选择搭建一个半中心化的服务端与客户端程序，能够兼顾上述主要的需求，在吸收主流分布式存储系统优点的同时，为遥感智能解译训推任务提供针对性优化，配合平台实现智能化的分布式存储系统方案构建。

# 半中心化架构

## 基础架构

中心化架构涉及对中心节点的带宽与计算压力很大，但是系统同步、协同效率更高，处理逻辑更清晰；而去中心化架构设计完全抛弃了中心节点，每一个节点都可以作为独立的节点进行连接和管理，这样操作可以对带宽计算压力进行均衡，但是随着分布式存储节点数目的上升，数据同步的操作复杂度将会呈现指数级增长。

本次提出的半中心化架构是一种介于中心化与去中心化架构之间的架构设计，能够尽可能地兼顾中心化架构和去中心化架构的优点，大幅提升数据传输效率并降低中心节点的计算和带宽压力。

以下面的网络拓扑结构为例，所有服务器均与一个万兆网络交换机连接，其中每一台服务器与交换机之间的网络传输带宽（上行以及下行）为10Gpbs。同时每一个服务器配备raid阵列硬盘或普通的企业级机械硬盘，连续写速度不高于500MB/s，低于网络传输带宽，连续读速度一般为5-10GB/s，远高于网络传输带宽。

 

在半中心化架构中，只能有一台DS同时为客户端提供V3协议的连接，但是所有的DS均可以作为NFS-V3的服务端提供相关的服务功能。需要注意的是只作为MDS功能的服务器不支持挂载，但是一个服务器可以同时提供DS和MDS功能，这是该服务器可以对外提供挂载功能。在图中示例的场景中，当DS3提供NFS服务并且正在读取一个存在备份或者拆分过的文件数据，那么它将首先调用MDS1或MDS2进行元数据查询（MDS1和MDS2之间属于双活备份，也可以进行计算的负载均衡），查询到数据分布信息后，与目标DS2进行数据同步，然后将读写的相关操作与客户端进行互通。同时MDS1和MDS2之间也会进行文件存储信息的同步处理。

由于该结构每一个服务节点都可以作为NFS的服务端，因此如何针对挂载数据选择最佳的服务端至关重要，对所有服务器之间建立网络带宽以及网络延迟图，结合数据分布进行最优化选择将是均衡网络带宽压力并提升客户端读写性能的关键。

## 时间同步

在实际进行分布式存储部署的时候，一个很重要的数据便是操作时间，这个时间决定了系统是否可以按照实际的时间顺序恢复操作现场，并保证数据结果的时间轴结果一致性。然而为所有的节点部署时间同步服务只会增加系统的安装难度，降低用户的使用热情。因此系统会在不同服务器节点之间配置基础的时间同步服务，这个服务并不会对服务器本地的时间进行同步，以确保服务器所属时间同步集群差异导致的时间误差。系统提供的时间同步服务会在更长间隔的HeartBeat触发操作时，由MDS向所有的DS节点发送同步请求，将主MDS的时间戳发送给各个服务器，由各个服务器配合请求延迟数据预估两个服务器之间的时间差，并在下一个同步节点到来之前，使用这个固定的时间差以及本地时间戳计算出MDS的标准时间戳用于时间一致性相关联的处理操作。

## 节点位置

针对典型的云存储服务器机房，常见的服务器灾害类型包括了单点失效和多点失效两类，前者具有很大的随机性，没有任何规律可言。而多点失效常常与服务器的物理链接方式挂钩，最常见的就是机架整体失效、机房整体失效，这两种失效都是物理连接位置相关的，可以通过手动设置备份策略，来避免大规模失效后的服务中断。因此节点在实际运行中的物理连接位置，对容灾能力的影响十分重大，如果使用完全随机的节点备份策略，将会有较低的容灾能力。

系统允许通过命令或配置参数对节点的物理位置进行标记，系统会基于标记的位置对节点搭建网络拓补结构，在进行数据备份的时候考虑物理位置的容错。如果用户没有为节点配置默认的位置标记信息或者只为部分机器配置了位置标记信息，那么系统将会将没有信息的节点单独组织成一个子集，按照IP以及网络延迟\带宽测试结果搭建预估的网络拓扑，其余的节点按照标记信息建立网络拓扑结构，基于这个拓扑结构实现备份选择策略。当然所有的网络拓扑构建是以物理卷为单位设置的，系统不会做物理卷之间的备份，因此也无需建立物理卷之间的网络拓扑。

/Beijing/CSC-CY/Room212/Rack12-33/Server\_127\_122

系统配置的节点位置采用上面这种类似文件系统的路径方式进行设置，因此依托这个路径整体搭建目录树形态的网络拓扑结构。这种结构自由度更高，系统也会尝试从最低等级的目录到更顶级的目录逐级寻找备份存放位置。对于一种特殊的情况比如在机房一级没有同级替代来存放备份（只有一个机房，没有其他机房做机房），那么系统将会向更上级寻找备份的可能性。如果最极端的情况到最顶级都无法找到备份替代，那么系统将会从底层循环重复上述过程在同级别不同服务器增加备份数量，直到备份数量达到用户的冗余备份数量要求。

# 卷管理

## 卷结构

为了能够实现存储虚拟化以及存储隔离，系统设置了挂在卷、虚拟卷和物理卷三种概念，虚拟卷为用户提供定额虚拟存储空间，物理卷基于底层物理存储实现分布式存储的聚合。



**物理存储不允许同时服务于多个物理卷，同时同一个物理卷只能使用一种条带协议进行几个物理存储的分布式管理。**为了保证系统存储的效率，以及支持更高效的文件管理，系统会在配置文件中设置物理存储的容量下限（如256GB以上），避免物理存储的碎片化对处理效率产生影响。

每一个物理卷的空间是由加入的物理存储节点决定的，物理卷向上则可以建立虚拟卷并为虚拟卷分配空间，单个虚拟卷的大小不能超过物理卷的总大小。虚拟卷的剩余空间等于当前虚拟卷分配的空间减去已使用的虚拟卷存储空间，当物理卷的剩余空间小于虚拟卷剩余空间的时候，虚拟卷的剩余空间会被物理空间限制侵蚀以确保虚拟卷剩余空间的合理性。

虚拟卷本身不能进行挂载，虚拟卷之上可以继续创建虚拟卷或者挂载卷，只有挂载卷（或者也可以称为根虚拟卷）才可以向外进行直接基于NFSV3协议进行挂载，默认挂载源路径为虚拟化后的根路径。如果一个虚拟卷之上创建了挂载卷（红色虚拟卷），那么这个虚拟卷之上将无法继续创建虚拟卷。

## 虚拟路径

### 管理路径

由于分布式存储的文件系统是虚拟化出来的，为了有效的对整个分布式存储系统的文件进行管理，系统构建了一套系统级别的目录树。每一个挂载卷都拥有一套完整独立的目录树，而所有挂载卷的目录树则会挂靠在虚拟卷上，虚拟卷继续向上挂载高层级的虚拟卷，最终挂载到物理卷中，物理卷会最终挂载到根目录上，一个典型的管理层面的虚拟路径如下：

/PV-8a73dhe/VV-722jdu9/VV-jkd93ss/MV-i82uhyjj92dh9k/temp/test/test\_new.txt

上面的路径中*PFS-8a73dhe*代表的是一级物理卷，所有的虚拟路径最多只有一级物理卷。物理卷后面紧接的目录是虚拟卷编号*VV-722jdu9*，虚拟卷可以进行多级嵌套，但是最后一个级别的卷一定是挂载卷*MV-i82uhyjj92dh9k*。挂载卷之后就是用户可以看到的文件系统目录树了，因此在上面这个完整虚拟路径中，用户看到的实际是“*/temp/test/test\_new.txt*“。

### 挂载路径

但是在实际用户进行挂载的时候并不希望看到如此复杂的路径结构，因此系统还会维护一个快捷挂载路径，将一个用户自定义的一级短路径映射到完整的挂载路径上，比如：

/my-volume-001

系统要求短路径只有一级，因此也要求全局的短路径不能重复，一个挂载卷只能映射到一个快速挂载路径上，两者具有一对一的关系。每次用户创建一个挂载卷的时候，系统会给用户使用挂载卷编号*MFS-i82uhyjj92dh9k*为用户生成默认的快速挂载路径。用户如果希望自定义这个路径，可以通过客户端进程进行修改，并使用快速挂载路径进行挂载，完整的虚拟路径仅用于管理，无法进行挂载：

sudo mount.dnfs 192.168.125.12:/my-volume-001 /mydata

## 卷操作

本章节会对卷运行时的一些操作逻辑进行简单的描述，说明具体执行某些命令的时候存储管理系统的响应情况。所有的卷操作均可以通过客户端执行程序进行处理，并且卷处理可以在任意一个节点上执行，所有操作会在节点之间进行自动化的分发与同步。

### 卷挂载

系统支持在任意一个节点上挂载任意一个挂载卷，因此在挂载行为上来说，整个分布式存储系统是去中心化的。



然而对于上图中的存储情况，如果希望挂载挂载卷1-2，那么从DS3进行挂载是最差的选择，因为DS3本机所提供的物理存储与挂载卷1-2之间没有任何关联，所有的数据读写操作都需要进行跨机的处理，处理效率大打折扣。相对的从DS1和DS2进行挂载卷1-2的挂载都是更为合理的选择，因为有一定概率快速可以获取到本机的数据，从而避免跨机器的数据传输。当然为了最大化挂载卷的挂载效率和网络负载均衡的问题，对于挂载卷1-2来说，DS1和DS2之间的选择是会变化的，然而为了兼容Docker传统的V3版本的NFS协议，这种选择必须是预先选择好的，无法进行动态调整。

因此对于用户来说，卷挂载主要有两种方法：

1. 强制挂载

强制挂载指的是用户直接使用原生的NFS协议进行挂载，在这样的挂载模式中，NFS服务端地址是预先指定好的，如果用户选择的服务端服务器是一个远数据服务器，那么传输效率会大打折扣，这种情况是系统对NFSV3协议兼容的一种妥协。好处就是该协议可以避免客户端的任何修改，原生兼容linux系统内的nfs客户端和Docker的nfs客户端支持。

1. 动态挂载

为了解决强制挂载的弊端，系统开发了专门的客户端并开放相应的API接口，帮助操作者自动选择最佳的挂载服务端服务器。对于平台来说可以调用相应的API接口获取指定挂载卷实时的最佳挂载服务器，对于系统操作则无需提供IP地址直接进行虚拟路径挂载即可。同时客户端程序也被用来帮助用户进行卷操作以及其他特殊的功能。

### 存储失效

物理存储失效是一种比较常见的服务器错误，一种典型的场景就是某一个DS节点失去响应。如果系统配置了冗余备份机制，那么所有的数据依然可以正常访问，只有以失效DS节点为服务端的NFSV3客户端会出现故障；如果系统没有设置冗余备份机制，那么一个DS节点失去响应就会导致所有存储在当前DS节点的数据失效，以及所有以该DS节点为服务端的NFSV3客户端无法正常工作。

当一个DS节点出现失效的情况后，MDS节点的HeartBeat机制会察觉到DS节点的丢失，这个时候

# 元数据管理

元数据指的是系统内部所有存储文件的结构信息一级其他系统界别的管理数据，这些数据由MDS节点进行统一管理。如何在MDS节点上组织管理和同步这些信息，确保主备节点之间的元数据同步和一致性，决定了系统处理的稳定性和效率。下面表格中给出了MDS节点负责维护的元数据信息，

|  |  |  |
| --- | --- | --- |
| 元数据 | 持久化方法 | 处理方法 |
| 文件信息 | 定长数据列表 | * 文件信息中存放的是以FH为索引的所有文件元信息，其中包括了文件大小、文件名、文件存储位置等信息； * 这种索引关系会在MDS的缓存中建立起来，对应的持久化文件中并不存储这个映射关系； * 系统会使用类似软件包的形式，按照FH的某些位将信息拆分成多份，分成多个文件进行存储； * 在存储的文件内部对FH的后几位进行拆分，标记不同位在实际数据中的索引范围。 |
| 目录信息 | 变长数据列表 | * 目录信息存放的时以FH为索引的一个FH信息列表，标记了指定目录下面所有的子文件和子目录； * 这个FH索引列表的映射关系会在系统加载相应的数据到缓存中的时候会建立起来； * 系统会使用类似软件包的形式，按照FH的某些位将信息拆分成多份，分成多个文件进行存储； * 在存储的文件内部对FH的后几位进行拆分，标记不同位在实际数据中的索引范围。 |
| 卷信息 | 多个定长数据列表 | * 卷信息存放了物理存储、物理卷、虚拟卷、挂载卷的相关信息，几个不同类型的卷拥有不同的数据结构信息； * 由于卷信息相对来说内容并不会占用较大的存储，因此MDS节点会将所有卷信息加载到缓存中，并定时执行持久化同步。 |
| 节点信息 | 定长数据列表 | * 节点信息存放了不同节点最近的一些状态信息，这些节点包括了MDS和DS节点； * 存放的信息多数是短期内变化较小的节点信息，比如节点主机名、IP、权限用户等等； |

## 元数据持久化

所有的元数据都会进行持久化落盘，以保证系统能够在重启后正确完成恢复操作。持久化到硬盘的元数据读取速度，要远远低于内存中的数据结构读取速度，因此系统也会设置元数据缓存来加速常用元数据的存取操作。同时系统为了进一步提升持久化数据的查询速度，也会在持久化文件中增加一定的元数据位置分布信息，对数据进行分区存储增加数据索引读取效率。其中目录信息和文件信息的持久化**是以挂载卷为单位进行隔离的**，因此在一定程度上避免的不同挂载卷内部的元数据干扰，可以一定程度上提升挂在卷内的索引速度。

### 文件信息持久化

文件信息持久化的索引核心时完整的FH，这个数值使用固定的规则生成，每一个文件用于独特的FH信息，系统永远可以使用FH定位到一个指定的文件。下图展示了持久化文件信息的存储方式：



FH信息的持久化除了可以将文件信息存储防止丢失，还需要采用多级拆分的方法进行索引加速，所有的索引都是基于FH字母序排序后的结果进行拆解的。

上图给出的结构中，系统首先使用FH的首位进行拆分，将文件信息进行初步拆分到不同文件夹。然后在文件夹中使用更多位进行索引，按照范围定义具体的二进制文件存放具体的数据。单个文件索引使用的位个数以及范围，系统会按照FH的数量进行拆分，确保单个文件夹下面的内容不会过多，在过多的时候对文件进行拆分，确保单个文件内部数据的规模不会对读取产生较大的影响。

最右侧展示了单个具体文件内存放持久化文件信息数据的格式组织。其中文件的前64位是固定魔数，存放的是8个字符，用来区分系统持久化的文件类别。然后存放了文件的分区个数，这里的分区在文件内部对数据进行二次拆分，可以进一步避免数据遍历范围加快处理速度。第三部分是文件信息的单个分区容量，这个数值一般是系统默认数值。第四部分存放了分区信息，里面存放了每一个分区第一个FH。系统在获取位置前会完整的读入分区位置信息，以找到FH所在分区数据索引，结合单个分区的大小可以计算出相应的数据偏移量。然后系统会从对应的偏移量开始读取这个分区的所有数据，并比对FH，直到找到FH的信息。需要注意的是即便是在一个分区内部，文件信息也是按照FH的字母序排列的，因此如果查找过程发现字母序出现问题，即没有找到FH，可以提前结束搜索过程。所有的顺序查找都可以使用二分查找实现，可以加速查找速度。

### 目录信息持久化



目录信息持久化针对的是目录内文件信息查询，这个信息一方面会被用于加速目录内的处理操作，另一方面也会在一定程度上指导预取。在持久化文件的目录结构上，索引方式和文件信息完全一致，具体类型参考上图。下图则给出了目录信息单个文件内二进制数据的组织形式，具体形式与文件信息有相似但不完全相同。



在目录信息的元数据文件中第一个部分依然是64位的魔数，实际存放的是一个字符串“DIRFH”。然后存放了一个32位的目录FH分区个数，目录分区可以加速目录FH的查找效率。第三部分存放了一个16位的目录FH分区大小，默认设置为256。下面的两个部分就和FH文件信息十分相似了，用于对目录FH进行分区查找。当系统在目录信息分区找到对应的目录FH时，便可以得到对应目录FH列表数据所在的数据位置，结合下一个目录的数据位置，便可以计算出当前目录下文件的总数。最终直接找到实际FH列表数据区域，便可以获得指定目录下面的FH列表了。如果需要查询每一个FH的信息，系统还需要从文件信息中读取并找到他们，为了加速这个处理效率，系统会采用多线程的方式完成这个任务。

### 其他信息持久化

其他需要持久化的信息主要是卷结构信息以及节点信息，这些信息的持久化相对来说比较专用，同时由于本身数据量较少，所有的数据被存放在单独的两个文件中：



上图给出了两类信息的持久化文件的结构划分情况。左侧文件存放了节点信息，文件将MDS和DS信息列表进行了隔离，以便进行单独更新维护。右侧文件存放的是卷结构信息，也是通过将不同类型卷信息进行隔离，提升响应数据的维护效率。所有的列表信息均由专门的反序列化和序列化函数进行数据的读写。

## 元数据缓存

为了避免MDS节点对持久化文件的频繁读写，系统会设置元数据缓存将常用的元数据加载到内存中，提升处理效率。MDS节点的元数据缓存规模要大于DS节点的数值，这个倍率是在MDS的配置文件中设置的。在具体的缓存结构上，MDS的元数据缓存结构与DS结构缓存一致，具体结构在下一章进行详细说明。

## MDS均衡与同步

在传统的大规模网络系统架构中负载均衡需要一个专门进行均衡工作的服务器和集群，这个服务单位会作为对外的统一入口接受请求，然后再分发到不同的均衡节点上处理。因此实际上负载均衡操作依然会在最初的入口服务器上存在瓶颈，网络层面的解决办法便是dns协议会针对一个域名设置多个等效服务IP，在这些IP中由客户端选择一个最近的IP（一般会按照华北地区、华南区域这样的物理位置设置不同的IP入口服务器）处理服务，然后才由均衡服务器分发后续的请求处理任务。



这种传统的负载均衡策略存在一个典型的问题，主入口服务器性能带宽要求高，即使按照地理位置区域进行拆分，每一个服务器依然承受超大量的请求数据。同时多级均衡策略也会带来网络链路上延迟的增加，这种延迟对于网络浏览这种延迟敏感度要求较低的业务影响较小，这一类业务更多需要应对的是业务的横向峰值压力，即同时支持的并发服务个数，而非延迟，因此使用少量延迟换取更大的服务带宽更加重要。而对于在线网络游戏这种延迟敏感的业务，服务带宽的重要性就要远远低于延迟重要性了，这个时候服务方常常会通过服务器拆分来避免均衡所带来的长网络链路延迟，这就是为什么网络游戏需要用户首先选区的原因。这种方式相当于把均衡的任务交由用户选择而非自动化，均衡后客户端直接连接到服务服务器，从而优化了网络链路长度。

### MDS负载均衡

在分布式实际上使用唯一一个MDS作为元数据管理节点是无法避免中心化管理的弊端的，随着元数据规模的上升，MDS服务器的性能势必会成为整个系统效率的主要瓶颈。为了充分利用备份MDS的性能，系统允许读取操作在备份MDS端执行，修改变更更操作则只能在主MDS端执行，这种方法在一定程度上大幅降低了主MDS服务端的负载，同时也有效避免可多个MDS节点写操作同步的复杂逻辑。具体操作过程如下图所示，这种均衡方法是优点类似于网络游戏业务的均衡策略，同时依靠读写分离简化业务逻辑，整体上是实现可能性与均衡性最好的负载均衡处理方案。由于主MDS只有修改操作，因此主MDS的缓存使用量并不大，每到一个持久化周期，就会将缓存中的内容更新到持久化数据中。



在该策略中，DS节点执行读操作如何选择MDS节点也会对性能产生影响，由于每一个MDS节点的缓存空间有限，如果一个DS节点在多个MDS之间摇摆不定，导致每一个DS常使用的数据在多个MDS缓存中均有备份，这会导致缓存利用率低下。甚至在DS节点增多以后，元数据缓存压力更大的情况下，导致缓存命中率很低，出现严重的缓存震荡。一种较为合理的方法便是DS与固定的MDS进行数据交换，可以在一定程度上避免缓存震荡，但是这种固定的关联与负载均衡并不匹配，无法做到动态。因此为了进一步提升处理效率，系统设置了MDS定期切换策略和MDS均衡选取策略两种算法对MDS选取进行优化。

MDS定期切换策略是固定MDS选取和动态MDS选取的这种，每一个MDS交互都由主MDS选择均衡后的MDS节点并不高效。因此系统会在一定周期后更新连接的MDS节点，来避免长期与固定的MDS沟通导致单个节点负载过高。此外系统的卷结构注定了，针对同一个挂载卷的连接更倾向于在一个MDS上统一处理元数据，这样会具有更好的元数据局部性。因此每一个客户端和DS服务端的连接，会分配一个指定的MDS读节点来处理这个连接相关的元数据信息。

在DS申请一个新的MDS均衡节点时，主MDS会收集不同读MDS节点平均响应时间统计信息，其中包括了整体平均响应时间，指定挂载卷的平均响应时间，以及MDS到某一个DS的操作平均响应时间。针对这三个响应时间，系统会设置一个均衡压力阈值，最终系统会在三个响应时间都没有超出阈值的MDS中选择响应时间最短的MDS节点作为最终的均衡节点，排序时挂载卷的平均响应时间优于MDS-DS操作平均响应时间，然后优于MDS的整体平均响应时间。

### MDS数据同步

MDS节点并不仅仅只有一个，因此主MDS在收到任何元数据信息变更的时候，不仅通知其他所有的MDS节点进行数据同步，还要更新本地的持久化文件。对元数据的修改操作是实时同步的，MDS在收到一个元数据修改处理操作的时候，会优先在不同MDS之间进行数据同步，并记录变更信息。所有的MDS和DS节点只有在达到全局持久化更新节点的时候，才会根据周期内的变更日志，将更新同步到本地的持久化文件中，并验证持久化文件的正确性，正常情况下不同MDS之间的持久化文件应该是完全一致的。整个处理流程如下：

1. 客户端删除文件，向DS节点发送了文件删除请求，DS接收到了请求并尝试更新本地的元数据缓存数据，然后向主MDS发送删除请求；
2. 主MDS收到删除请求，同步处理本地的元数据缓存，并记录这个删除操作到持久化操作日志中，用于后续持久化文件更新；
3. 主MDS查询本地的MDS节点与挂载卷的关联关系，同步的向具有关联关系的节点发送修改操作，并且要确保得到回复，而其他没有关系的节点（只更新持久化数据，不涉及缓存内容）则会采用异步处理，从而可以减少同步的延迟开销；
4. 主MDS返回处理结果，DS节点完成对应元数据的处理操作；
5. 进入新的持久化周期节点，主MDS节点触发全局的同步持久化处理，首先MDS本地将前一个持久化周期的内容持久化到文件中，然后同步将持久化操作信号和验证信息发送给所有的MDS，所有MDS完成持久化处理；
6. 主MDS释放上一个周期的持久化记录数据结构，进入新的一个周期；如果持久化验证发生错误，系统将会开始执行持久化文件的修复操作，并对相应的信息缓存进行清理和锁定，在完成持久化文件同步修复后解除锁定。

# 节点缓存

为了有效的提高节点处理数据的效率，系统为每一个节点配置了不同的缓存用于加速数据处理效率。目前主要配置的缓存包括以下几类：卷结构信息缓存、FH信息缓存、目录信息缓存和文件数据缓存。

## 卷结构信息缓存

卷结构信息存放的是卷逻辑中卷之间的关系信息，以及关联关系信息。首先卷逻辑关系本身是一个关系树，每一个几点对应一个卷类型。

### 卷结构信息缓存分布



下图展示了一种典型的卷结构信息缓存分布模式，在MDS服务器中存放的是完整的卷结构信息，里面包含了整个分布式存储的卷信息和物理存储信息。而DS服务器中存放的则是当前节点用到的卷结构信息，无关卷结构信息并不会存放到相应的服务器中，特别是卷结构信息非常复杂的情况下，存储有限量的卷结构信息缓存可以大幅减少缓存同步所产生的网络开销。

### 卷结构信息缓存更新

卷结构信息被用来确定当前节点NFS操作所属挂载卷对应的物理卷、存储条带协议等相关信息，该信息缓存从MDS节点拉取并长期存在。卷结构可以在任意一个节点进行修改，原始信息只会存放在MDS节点上，一旦卷结构信息发生了变化会对所有节点进行广播，更新所有节点的卷结构信息缓存。但是广播更新卷结构信息缓存存在一个延迟，因此卷结构信息变更的操作会在操作端进行阻塞，直到信息更新完成并且所有可用节点的卷结构信息缓存完成更新，才会返回结果在操作端提示操作完成。

卷结构信息缓存的全量更新耗时随着节点数量的增加而增加，是否执行异步处理决定了操作端的回执速度；不同节点卷信息的更新延迟也可能会导致异常操作的出现，比如虚拟卷的空间修改，而某一个挂载卷正在使用该虚拟卷进行写入操作，那么这个时候是需要先广播进行卷锁定的，然后才可以完成卷信息修改和操作。因此实际卷结构更改会根据操作类型触发不同的操作，并选择异步还是实时进行卷结构信息缓存的更新。

卷结构信息会存放一个更新时间戳，这个时间戳来自于MDS节点，借助该时间戳，节点可以验证卷结构信息是否进行过修改，从而可以避免传输相同的数据信息。

|  |  |  |
| --- | --- | --- |
| 操作类型 | 处理模式 | 处理方法 |
| 创建虚拟卷 | 异步执行 | 本地修改结构信息并同步到MDS后结束操作；MDS将会在下一次与DS节点的HeartBeat过程中强制同步相关信息；相关信息在其他节点的加载属于懒加载模式，只有对应的节点使用到了与新创建虚拟卷关联的挂载卷的时候才会触发卷结构信息的查询操作并获取相关信息 |
| 创建挂载卷 | 异步执行 | 操作流程与创建虚拟卷类似 |
| 修改虚拟卷 | 同步执行 | 在执行虚拟卷修改操作前会对涉及当前虚拟卷的所有DS节点（包含虚拟卷相关物理存储的节点）进行通信，执行虚拟卷锁定操作，然后才会执行虚拟卷信息的修改；当且仅当所有虚拟卷关联的DS节点都返回了修改成功的操作结果时，操作节点才会返回操作结果 |
| 修改挂载卷 | 同步执行 | 操作流程与修改挂载卷类似 |
| 删除虚拟卷 | 同步执行 | 删除操作等效于修改操作 |
| 删除挂载卷 | 同步执行 | 删除操作等效于修改操作 |
| 创建物理卷 | 异步执行 | 创建物理卷与创建虚拟卷操作类似 |
| 修改物理卷 | 同步执行 | 操作流程与修改虚拟卷操作类似 |
| 删除物理卷 | 同步执行 | 操作流程与修改虚拟卷操作类似 |
| 增加物理存储 | 异步/同步执行 | 操作流程与创建虚拟卷操作类似，但为了加快热更新效率，只有所属物理卷剩余空间小于一个阈值的时候，才会进行同步更新，以避免物理卷存储满的情况；否则将会执行异步更新，在下一次HeartBeat的时候进行物理存储信息更新，将存储卷和物理卷更新信息同步到相关联的节点 |
| 删除物理存储 | 同步执行 | 操作流程与修改虚拟卷操作类似，删除物理存储可能会导致存储的失效，因此删除物理存储的时候会对存储进行均衡化处理，比如二次分配拷贝和迁移，在此期间会对整个物理卷锁定，这会对业务产生影响 |

## FH信息缓存

FH信息用于快速获取指定文件的FInfo，其中包含了文件所属的物理存储DS信息，文件本身的属性信息等等。在MDS服务器中，所有的FH信息会以二进制的形式存储到文件中进行持久化存储，这里的FH是分布式系统本身生成的FH，并不会与底层的FH挂钩，具备持久化特征。NFS协议本身在对文件进行读写操作的时候，每一次都会进行复杂的ACCESS、GETATTR等于FInfo相关的操作，因此DS端都会对最近常用的FH数据进行缓存以提高相关信息的处理效率。同时考虑未来条带层协议的扩展，底层文件的信息与实际挂载后展示的处理信息并不一致，文件信息主要存在于FInfo中。

### 信息缓存结构与替换

为了避免空间的浪费，同时考虑使用复杂Map的可能性，FH信息缓存并不是统一处理。首先一个FH信息多达64字节，使用FH信息直接进行索引Map的开销是很高的，因此将当前服务器相关的所有FH构建一个Map是不现实的。为了减轻Map的索引压力，加快索引速度，会对FH缓存信息按照挂载卷进行划分，对每一个挂载卷下面的FH信息缓存单独管理。同时可以基于FH的生成算法对FH进行初步分组，然后再在每一个分组中使用短哈希的Map进行映射搜索，进一步减少Map的压力。

此外为了解决FH信息过期排序的问题，系统会为每一个挂载卷设置一个FH指针链表，里面按照最近访问顺序排列了所有的FH，当FH信息被访问的时候，会从链表任意位置提升到链表顶部。当FH信息被剔除的时候会同步将FH信息映射从相应的Map中清除。为了减少FH多份拷贝对空间的占用，系统维护了一个FH池子，里面存放了FH所有相关信息的实体，可以大幅减少内存压力，并采用类似智能指针的方式避免一个FH的任意释放。



### 信息缓存处理流程



上图给出了一个典型的GETATTR操作处理流程，这里不涉及FH内容的修改处理，仅涉及FH的访问操作，其中包括了FH创建的相关操作。

### 信息缓存变更

FH信息缓存的读操作并不会触发任何操作，但是写操作将会触发全局信息同步处理，因为一个FH信息可以同时被缓存在多个节点中。如果指定的FH信息在某一个DS节点被修改，那么修改将会可以进行同步执行策略，修改本地缓存的同时将信息同步到所有包含该缓存的节点以及MDS节点。虽然修改内容较小，但是随着通信节点数量的增加，整体修改数据的同步成本依然很高。因此基于MDS节点的统一更新时间戳对一致性进行统一处理是比较合理的处理方法。

#### FH权限用户信息更新

FH信息中主要包含的是访问权限、所属用户组、操作时间、文件大小几类信息，其中访问权限以及所属用户组一般是通过NFS-V3中的SETATTR进行单独操作，具备一定的操作独立性，而操作时间和文件大小并一般不SETATTR直接进行修改，而是依赖于服务器端的操作进行修改和更新。下面几个更新例子主要针对的是文件的权限和所属用户组信息，不涉及文件的更新时间和大小信息。

1. **网络延迟无差异化处理（独立）**



典型的无差异化网络传输一般不会在具体处理过程中出现穿插传输，不同DS的请求和返回属于是完全分离的状态，这也就使得最新的同步请求才能够获得最新的缓存信息，但是为了确保缓存信息保持为最新的状态，DS会依靠HeartBeat机制进行缓存的刷新和同步。

1. DS1出现了FH缓存修改，从1.0变更为1.1，并向MDS发起了同步请求；
2. MDS接受了变更处理并将中心FH信息更新到1.1；
3. DS2出现了FH缓存修改，从1.0变更为1.2，并向MDS发起了同步请求；
4. MDS接受了变更处理并将中心FH信息更新到1.2；
5. MDS到达一个HeartBeat时间点，开始向所有DS发送HeatBeat更新信息以及其他处理请求操作，这里会告诉目标DS机器当前HeartBeat时间段内它需要更新的信息；
6. DS1获取到需要更新的信息并更新本地FH缓存信息并返回MDS需要的状态信息；
7. **网络延迟无差异化处理（穿插）**



上图给出了一种比较典型的无差异化网络延迟处理策略模型，在这种模型中，每一个DS到MDS之间的延迟都是足够小的：

1. DS1出现了FH缓存修改，从1.0变更为1.1，并向MDS发起了同步请求，MDS接受了变更处理并将中心FH信息更新到1.1；
2. DS2出现了FH缓存修改，从1.0变更为1.2，并向MDS发起了同步请求，MDS接受了变更处理并将中心FH信息更新到1.2；
3. DS1接收到了MDS的缓存信息返回，将本地FH信息缓存从1.1更新到了1.2；
4. DS2接收到了MDS的缓存信息返回，不做任何处理。
5. **网络延迟差异化处理（修改相同信息）**



上图给出了一种典型的DS到MDS端网络延迟差异大的场景。

1. DS1出现了FH缓存修改，从1.0变更为1.1，并向MDS发起了同步请求；
2. DS2出现了FH缓存修改，从1.0变更为1.2，并向MDS发起了同步请求；
3. DS2到MDS的网络延迟很低，导致DS2的变更请求优先到达MDS，MDS接受了这个请求将FH状态更新至1.2；
4. DS1到MDS的网络延迟远高于DS2到MDS的延迟，导致DS1的请求后到达MDS，为了降低更新缓存成本开销，MDS将不会进行时间顺序一致性校验，将FH更新到1.1；
5. MDS将处理后的FH状态1.1返回给DS2，DS2本地进行校验后将信息更新至1.1，并同步到本地FH信息缓存中，至此DS2的FH信息修改操作完成；
6. MDS将处理后的FH状态1.1返回给DS1，DS1本地进行校验后直接将信息更新到1.1版本，并同步到本地FH信息缓存中，至此DS1的FH信息修改操作完成。

#### FH时间大小信息更新

FH中的文件的大小信息即文件大小，单位为字节，时间信息则包括了创建时间、修改时间和访问时间。如果底层存储条带层协议采用的是独立文件存储，那么这些信息会由文件所在的DS服务器进行维护和处理，然后在MDS服务器进行同步。如果底层存储条带层协议对文件进行了拆解，那么所有信息的更新操作与权限用户信息保持一致。下面只对独立文件存储模式下的更新处理场景进行说明：

1. **网络延迟无差异化处理**



针对独立文件存储，多个DS服务器都会持有响应FH的缓存数据，上图中给出了一种典型场景：

1. DS1出现了文件写操作，但是DS1本身没有持有原始数据，DS1基于FH信息中的DS存储信息向原始数据所在的服务器DS2发起Remote Write操作；
2. DS2收到了Remote Write请求，然后基于请求数据执行Local Write，并在本地对FH的信息进行动态更新到1.1并将最新的FH信息同步返回到DS1，DS1获得更新后的写信息；
3. DS2因为出现了FH信息修改，从1.0变更为1.1，向MDS发起了同步请求，将最新的FH缓存信息更新到MDS中；
4. MDS接收到了同步请求，将本地的记录进行同步，FH缓存信息从1.0变成了1.1；
5. MDS到达了执行HeartBeat时间点，向所有与FH有关的，包含相关缓存的DS发送同步请求，将FH修改情况发送出去并由DS3接受，进行了同步，DS3上面的FH缓存升级到1.1。
6. **网络延迟差异化处理**



针对FH时间大小信息的更新，在存在DS之间网络延迟差异化较大的情况下，也会出现实际更新达到时间与实际更新时间差异大的情况。比如上图中DS3首先触发的写操作遭遇DS2的写操作，但是两者的写操作到达DS1的时间却是相反的，这时候DS1将会按照实际操作达到的时间执行操作。这可能会导致DS2的操作被抛弃，也有可能DS2和DS3的写操作均生效（写入位置不冲突的情况下），具体结果与DS1处理冲突写操作的逻辑有关。总的来说整个分布式存储系统并不会对多客户端的写操作一致性进行处理，所有的结果具有一定的随机性，但不会导致出现客户端写出现致命错误。

#### FH脏信息与HeartBeat同步

根据前面搭建的几种场景来看FH信息缓存会被多个DS存放，同时也可以被多个DS修改，在执行FH信息修改的时候会明确返回一次最新的FH信息用于刷新本地的FH信息缓存。但是如果某一个DS长时间未进行写操作，那么他的FH信息缓存大概率会有过期的问题，这将导致该DS的下一次处理操作有概率存在异常。这个时候就需要MDS向其他所有DS服务器发送HeartBeat请求进行缓存信息同步，并同时进行响应缓存信息的同步。

系统向哪些DS发送哪些FH信息更新，可以通过脏位标记的方式来实现。即当MDS完成本轮HeartBeat后会清空所有FH缓存信息的脏信息，在下一次HeartBeat到来之前对所有期间进行过修改的FH信息进行标记。MDS在下一次HeartBeat的时候会向相关联的DS发送相应的FH脏信息更新。FH可能与哪些DS相关在MDS是有记录的，记录是根据每次挂载操作的结果得到的。比如某一个FH相关的物理卷只从某一个DS向外提供了活跃的挂载，那么HeartBeat不需要向这个DS提供这个FH的信息缓存更新，因此所有FH的修改都来自于唯一的DS。如果有两个DS提供指定FH的挂载，那么HeartBeat会向这两个DS发送FH信息更新。这样便可以尽可能地减少HeartBeat的更新信息量，降低HeartBeat对所有服务节点的性能干扰。

## 目录信息缓存

客户端最常用的操作除了对文件本身的读写和信息操作外，就是目录信息查询和遍历操作，对于复杂的目录树结构或者单个目录下存放大量文件的情况，ls操作将会非常缓慢。因此系统针对目录关联关系也设置了缓存，来辅助加速ls操作，在协议层面主要用于Lookup操作加速和ReadDirPlus操作加速。

### 信息缓存结构与替换

每一个DS服务器中的目录信息缓存使用的是Map存放目录FH到文件FH列表的映射，为了限制单个目录缓存子FH的数量，设置了FH缓存列表的最大长度，从而可以避免部分目录单个目录下文件数量过多导致缓存溢出和内存占用的问题。FH缓存列表使用了一个LRU-FH指针链表和一个快速索引Map组成，其中前者用于缓存数据中的更替，后者用于Lookup查找操作的快速响应。为此目录信息缓存结构的基本结构设计效果图如下：



上图场景中第一次成功命中使得File3的FH提高到了队列最前端，让File3的Lookup更不容易出现延迟。第二次Lookup操作查找的是一个没有存在于当前缓存中文件信息，这个时候缓存没有命中，因此会从MDS拉取相关的FH信息记录，并加入到本地FH信息缓存中然后返回到客户端。

ReadDirPlus操作与LookUp操作有所不同，它需要获取整个目录下面所有FH的信息，但是对于部分存放文件数量超出最大缓存数量的目录，上述结构无法进行ReadDirPlus操作的加速。因此系统也会为部分ReadDirPlus访问操作频繁，并且上述结构无法完整存储目录信息的目录设置一个相对叫少量的缓存，可以存放完整的目录FH列表信息，以便对该操作进行加速。同时为了对目录信息缓存进行更替，针对目录的FH也会设置一个LRU链表，当新的目录访问到达的时候会按照链表的顺序剔除一个最近没有使用过的目录信息。

下图给出了典型的目录信息缓存结构，类似FH信息缓存，目录信息缓存也是与挂载卷挂钩的，也会使用类似FH信息缓存中的Group操作进行分组。而每一个目录内部设置了一个名字到FH-LRU指针的Map，以及一个LRU链表，链表中存放的是FH指针，最终指向一个统一个FH池子，可以减少FH多份拷贝占用大量的内存空间。借助Name-FH映射可以实现快速的LOOKUP操作，并通过FH指针找到相应的FInfo。同时系统会维护一个针对目录的LRU列表，来解决目录整体信息缓存的清退操作。



需要注意的是ReadDirPlus本身只存放了FH映射关系，并不存放FH的信息。然而实际ReadDirPlus操作返回的不仅是FH信息，还会返回GETATTR相应的所有信息，因此目录信息缓存需要和FH信息缓存进行联动。为了节省缓存信息需要的空间，全局的FH信息只会存储一份，并放在全局FH Pool中，无论是FH信息缓存还是目录信息缓存都指向相应的FH的指针，而非多个复制的实例对象。同时为了确保目录信息缓存中关联的FH信息不会在FH信息缓存中被剔除，FH信息缓存会给在尝试剔除一个FH信息时检查引用情况，当且仅当FH信息缓存中是唯一引用的情况下，才可以剔除这个FH信息。

### 信息缓存处理流程

#### LOOKUP



上图给出了系统接收到Lookup请求时的整体响应处理流程，其中实现部分是理想情况下缓存命中的路线，可见理想情况下命中可以节省很多操作流程。配合客户端的缓存，可以有效降低文件查找的延迟。

#### READIRPLUS



上图给出了READDIRPLUS的一种典型处理流程，由于对于文件量超大的READDIRPLUS操作是分步骤执行的，遇到这一类处理，系统也会从MDS分步骤请求相关的数据信息，并逐步更新到本地的无限制目录缓存中。同时有限制目录缓存的单个目录容量大小，也按照READDIRPLUS单次请求的上限作为限制数量，这样可以加速小目录的READDIRPLUS操作。

### 信息缓存变更

目录信息在运行过程中也会出现变更，首先FH是与文件绑定的，即便文件的目录发生了移动相应的FH不应该发生变化，这样对FH信息缓存的更新可以产生较小的压力。而目录信息缓存信息的变更主要来自于目录本身的删除、目录内文件的移动、目录内文件删除、目录内文件新增这几种操作，具体的处理流程如下：

|  |  |  |
| --- | --- | --- |
| 操作类型 | 处理模式 | 处理方法 |
| 目录内文件新增 | 异步操作 | 文件新增操作即使晚到达某一个DS，也不会对相应的业务产生致命影响，因此新增处理在当前DS处理完成后返回；并异步的向MDS发送处理操作，等待MDS将新增FH信息同步到所有DS的无限制目录信息缓存中后，整个操作完成。 |
| 目录内文件删除 | 异步操作 | 删除目录内文件类似新增操作，会异步执行FH强制失效处理，如果文件删除导致一个目录下不再拥有文件，那么相应的目录信息缓存也会被完全清理。 |
| 目录删除 | 异步操作 | 删除目录会首先对当前DS处理，然后将处理异步发送给MDS，由MDS通知所有可能存放当前目录的DS节点清除对应目录的缓存；递归删除目录可能会同时导致多个目录失效，但是递归处理是由客户端处理的。 |
| 目录内文件移动 | 异步操作 | 本质上是目录内文件新增和目录内文件删除的操作组合：异步处理的时候执行目录内FH的缓存清除处理，并对新目录下的文件FH执行新增，两个操作在一次到各个DS的操作中完成，其中新增操作只会对无限制的目录信息缓存中生效。 |
| 目录内文件  名称修改 | 异步操作 | 修改当前DS内目录内文件映射关系，会异步执行FH映射失效处理，由MDS发送处理到所有可能关联的DS执行缓存失效/更新操作。 |

## 文件数据缓存

除了上述几个用于基本数据结构的缓存意外，每一个DS也会对文件的数据块本身进行缓存，这个缓存并不会用于写操作，只会用于频繁的读取操作。下面会对文件数据缓存的结构设计以及READ、WRITE操作下的缓存处理流程进行介绍。

### 信息缓存结构与替换



上图给出了一个DS节点中的文件数据缓存结构，基于NFS-V3中READ操作的粒度，基础Data Block的默认大小被设置一个固定的大小，这个数值与NFS协议的读取分块策略有关。由于每一个节点的内存并不相同，因此系统也会自适应的选择最佳的缓存总大小，以确保当前机器的整体处理性能不会因为数据交换而严重下降。一般来说，对于一个内存大小为32GB的DS节点，系统会默认分配1GB大小的空间作为文件数据缓存的上限，这个比例可以在当前节点的配置文件中设置，如果没有设置，将会从MDS节点的配置文件中拉取作为默认配置。

当一个读取请求到来的时候，系统会使用文件的FH和Offset组合起来生成一个短Hash，并使用当前Hash索引到具体的数据块。鉴于短Hash存在重复的风险，每一个数据块中也会存放完整的FH信息和Offset，以便用于二次验证。通过短Hash便可以在完整的缓存块集合中查找到相应的数据块。同时为了方便对缓存数据块进行更换，系统也会使用短Hash做一个LRU列表，用于对长期未执行读取操作的数据块进行剔除处理。

关于Block LRU下面的数据结构会在READ操作处理流程中进行详细说明，该模块用于进行文件界别的读取历史追踪，来辅助进行文件数据块预取。

### 信息缓存处理流程

#### READ

对于读取操作如果命中缓存，那么相应的处理是十分迅速的。但是如果没有命中本地的读缓存数据，则需要首先依据FH获取相应文件的详细信息，其中最重要的是对应文件的大小信息以及所在服务器的信息。系统会首先从本地的FH信息缓存中尝试查询FH具体信息，如果没有成功，那么将触发FH信息拉取操作。当得到FH完整的信息以后，则会向当前文件所在的DS节点直接发起请求获取相应文件块的数据，并将相应的数据缓存到本地以便后续重新读取。

在实际的NFS客户端处理操作中，READ请求的分块粒度并没有规律性，但是针对超大文件的连续读写，一般NFS协议会在进行较大范围数据的读取操作时使用512KB的区域进行单次请求，因此文件数据缓存的块大小也更倾向于设置为512KB。



在实际的操作过程中，读取主要以随机读写、连续读写两类为主，并且在应用中读文件习惯存在明显的差异：有的文件会进行反复读取，文件数据缓存作用较大；有的文件则只读取一次，缓存信息并不能起到作用，反而会占用本地的数据缓存空间。但是这种文件读取习惯很难从数据块的层面察觉，因此为了优化文件读取的效果，实际的缓存除了对数据块本身设置LRU。还会为所有存在数据缓存的FH进行排序，排序的依据便是最近一段时间内的访问次数，这个次数会在指定间隔自动消减1，当某一个文件计数消减到0的时候，相应的缓存数据便会被剔除。同时对于次数超过一定阈值的文件，当出现相应文件的访问时，会猜测该文件存在连续读的可能性，在读取某一个块数据的同时，异步拉取当前数据位置后面的其他数据块进行缓存，以加速预期的读取操作。

#### WRITE



相比于Read，Write操作具有更加明显的不确定性，多数属于是追加写操作。同时写操作也会分为Unstable以及FileSync两种，前者写操作偏向于大批量数据的写，而后者属于少量数据写操作。区别就在于Unstable写需要在全部数据写完以后调用Commit完成写数据提交，相当于客户端的单次写入操作因为RPC请求限制被拆分成多个操作，在提交的时候被还原成一次操作处理，避免单次写入原子性被破坏。后者因为写入数据量较小，会直接在一次Write操作后完成处理，无需commit处理进行提交。



在实际的Write操作时，会同步对本地的缓存进行操作，同时同步调用文件所在的DS执行远程的处理，从而可以有效避免数据不同步的问题。上图给出了一种典型的多DS端写操作的处理方式：

1. DS1出现了文件写操作，但是DS1本身没有持有原始数据（此图不考虑FH同步的问题），DS1基于FH信息中的DS存储信息向原始数据所在的服务器DS2发起Remote Write操作；
2. DS2收到Remote Write操作，执行Local Write，这个操作如果是Unstable写，那么写数据会优先进入到缓存，不会立刻同步到本地磁盘，否则会强制触发磁盘写操作，整体处理流程延迟更长；
3. DS3同时持有同一个文件的句柄，并且收到了写请求，但是DS3本地持有相应的缓存信息，因此相应的写操作立即在本地生效，并同时发送了一个Remote Write请求给数据持有服务节点DS2进行同步写；
4. DS2收到Remote Write操作，执行Local Write，这个操作如果是Unstable写，那么写数据会优先进入到缓存，不会立刻同步到本地磁盘，否则会强制触发磁盘写操作，整体处理流程延迟更长；同时新的写操作会在DS1的写操作基础上执行处理，这种处理很可能存在冲突（比如两者都试图从同一个Offset开始写数据），系统会自动按照没有冲突的形式进行操作，因为系统不做写操作的严格一致性处理；
5. DS2处理了DS1的写操作之后，FH发生了变化，最新的FH信息会被同步更新到MDS，同时MDS对这个FH进行脏数据预标记，说明当前FH在HeartBeat周期内执行了数据修改和信息修改，但是这个修改如果没有被多个不同的节点处理，那么结果数据无需进行一致性处理；
6. DS2处理了DS3的写操作之后，FH发生了变化，最新的FH信息会被同步更新到MDS，但是由于当前数据块在当前HeartBeat中被多个节点执行了操作，同时MDS对这个FH进行正式的脏数据标记；
7. DS1完成了同步写操作，并同时获取到了对应文件块的缓存数据，将数据保存到本地，这个数据可能是DS3处理后的数据，也可能是DS1处理过的数据，最终结果与DS2的处理顺序有关；
8. DS3完成了同步写操作，并同时获取到了写操作的结果，这个结果如果和DS3写操作预期一致可能不会返回数据，否则会返回最新的修改数据，也可能不是最新的；
9. MDS进入到下一次HeartBeat周期，由于FH被标记了脏写操作，因此会在HeartBeat的时候通知可能相关的DS服务器，当前FH指定数据块出现了变更，这个变更会由对应的DS进行验证，如果这个服务器持有对应的数据块，那么会在接下来向DS2进行数据同步，数据同步期间，所有请求会被阻塞。

### 文件热操作缓存

文件热操作缓存是一个特殊的数据结构，用于解决文件本身处于特殊操作时期的正常访问和读写问题。其中读操作相对比较简单，而写操作较为复杂，当出现写操作的时候热操作缓存机会起效，这个缓存被用来记录文件底层处理期间所有的写入数据。



上图给出了文件热操作缓存的数据结构，存储模式与普通数据缓存类似，但是当索引方式有所改变。系统首先使用FH生成短hash用于进行相关缓存块的索引，索引得到的是一个Offset列表，通过这个Offset列表系统可以快速获取到制定文件的所有写操作日志。同时系统也可以使用FH和Offset联合生成短哈希直接索引目标块的数据。热操作缓存数据结构只会存在于物理存储所在的DS节点，用于对文件数据的读写操作进行重定向。

当一个文件数据被锁定的时候在没有任何写操作之前，所有的读操作不会触发生成热操作缓存，一旦出现写操作，那么写操作相关的块数据将会被读取到热操作缓存中，然后在缓存中执行影响的写操作处理，同时FH信息会进行同步的标记更新。如果后续出现读操作会优先从当前热操作缓存中查询数据，如果没有命中才会继续从原始数据中查找数据。也就是说，热操作缓存记录了所有因为写操作导致变更的数据块。一旦文件数据锁定被解开，则系统就会使用文件FH获取到所有的块数据进行写操作覆盖，确保信息更新的准确性。写更新覆盖会持续到所有缓存处理完毕后，然后对文件进行锁定，修改相关的元信息完成数据迁移的最终步骤后就可以使用正常的方法继续执行数据读写了。

在实际的应用中文件热操作缓存本身也会占用内存空间，但是内存空间与DS节点服务器的物理配置有关，并不是所有的节点服务器能够接受所有的写操作缓存。如果遇到DS节点内存不足，当前正在迁移一个较大的文件，系统又遇到了对该文件的大规模写操作，导致热操作缓存的溢出内存，那么系统将会受到严重的性能损失。针对这种情况系统会设置一个最大的热操作缓存限制，一旦当前缓存数量超出限制，系统将会暂停文件迁移，并启动工作进程根据迁移进度把写缓存数据同步传递给源文件迁移目标服务器，并同步到本地文件来释放本地的写缓存数据直到写缓存释放到一定水平才会继续执行迁移操作。

# 文件数据管理

文件数据的管理时通过物理卷与物理存储之间的条带层协议实现的，这一层设置了统一的处理接口规范，对上层提供统一的接口，对下提供不同的文件数据管理策略。目前主要设计了几种协议：全拆分条带、半拆分条带和无拆分条带：

|  |  |  |
| --- | --- | --- |
| 协议类型 | 协议说明 | 优缺点 |
| 全拆分条带 | 全拆分条带的文件管理更加接近于块设备管理，所有的文件都会被拆解成单独的文件块，并分布在不同的设备上，但是考虑到处理效率，一般文件块并不会完全均匀的分散在各个存储节点，而是有一定规律的分布，以避免高延迟的连续读操作。 | **优点：**存储负载均衡性更好，可以有效的保证不同节点存储的利用效率  **缺点：**复杂的管理操作和读写流程，小文件分块后处理效率较低 |
| 半拆分条带 | 相比于全拆分条带，半拆分条带更加关注拆分与独立存储之间的平衡，这种条带协议会将不适合进行拆分的小文件进行独立存储，而适合拆分的大文件以合适的粒度拆分后分配到不同的物理存储中。 | **优点：**存储负载均衡性较好，能够针对存储中不宜均衡的大文件进行分块和均衡化，兼顾小文件处理速度  **缺点：**相比于全拆分条带更加复杂的管理操作和读写流程 |
| 无拆分条带 | 无拆分条带更加易于理解，类似GFS中的分布式卷，将文件完整的存放于不同的DS节点中不进行任何拆分。 | **优点：**管理操作十分简单，整体的处理流程更加清晰，对于单个文件的读写速度较快  **缺点：**由于没有进行大文件拆分，在碎片化存储场景下存储均衡性较差 |

## 条带协议难点

条带在均衡策略设计时需要考虑问题有很多，比如文件读写效率、文件分布均衡性等，由于系统的NFSV3协议兼容需求，在不适用pNFS协议的情况下，这些问题存在一定的互斥性，因此均衡策略也存在很多折中的在几个核心问题上进行权衡：

### 存储均衡问题

均衡策略本质上解决的核心问题是文件分布均衡性问题，对于分布式存储，聚合的存储资源本身就有一定的差异性，特别是本系统目标群体包括了昂贵的企业级存储服务器以及廉价的单机工作站。均衡策略以物理卷为单位进行存储负载均衡，如果一个物理卷搭建在成百上千的物理存储之上，特别是对于物理存储差异性很大的情况，均衡难度会指数级上升。比如某一个物理卷包含了一个100GB的物理存储和一个100TB的物理存储，两者存储差距悬殊，负载均衡的是使用香槟塔模式还是均压模式会产生不同的效果。

### 访问均衡问题

由于系统默认使用NFSV3协议向客户端提供远程数据服务，所有的访问所带来的网络带宽压力都会集中在与客户端直连的服务器上。同时系统可以在单个物理卷上生成很多个挂载卷，这些挂载卷可以使用系统集群内的任意节点作为服务器端，如果挂载卷的服务端集中到某一个节点上，势必会造成这个节点的网络和计算的双重压力，造成客户端效率的降低。因此如何合理安排挂载卷客户端位置，在保证读写效率的同时均衡每一个节点可能的网络带宽压力，将会成为策略的关键一环。同时系统还需要考虑跨节点数据访问，所带来的系统内节点间通信网络开销，如果所有的挂载卷都使用距离物理存储较远的节点进行挂载，那么系统内通信网络开销对系统的影响将是灾难性的。

### 读写效率问题

第三个需要优化的问题便是文件的读写效率，在单个挂载卷存储没有分配到多个物理存储的情况下，就近挂载原则一定可以获得最佳的读写效率。然而存储均衡问题的处理一般不允许一个挂载卷下的所有文件存储在单一物理存储中，理想状态下经过存储均衡指定挂载卷的数据会分布在少量物理存储中，并且这些物理存储所在节点与常用来作为该挂载卷服务端的节点具有更近的网络距离。一旦一个挂载卷的内容被分配到不同物理存储，读写效率就一定会有所下降，如果系统可以学习挂载卷的文件访问习惯，将经常访问的文件集中在服务端节点可以大幅提升挂载卷的访问效率。

### 存储压缩问题

存储压缩问题其实并不是分布式存储需要解决的主要问题，这个问题主要包含了两个层面的内容，分别是物理压缩和存储共享。

物理压缩是对存储数据进行直接的压缩存储，这个操作对性能的消耗非常大，会严重拖累整个系统的效率。当然对于很少访问的数据进行简单的低级压缩，可以在牺牲少量性能的情况下带来较大的存储效率提升。如何定义少访问的数据，并定时对数据进行压缩并不是一个简单的问题。但是如果用户为一个物理卷设置了冗余备份，那么系统可以对基本不进行写操作的文件备份进行压缩，可以很大的提升存储利用效率。

存储共享则更多的出现在类似百度云盘这样的共享存储中。在共享存储中，存储压缩本质上类似文件系统中的硬链接和软链接，可以对单个文件进行几乎不占用实际存储空间的复制拷贝，由多个人同时使用。共享存储中则可以通过标记原始数据路径的方法，在多个用户的个人网盘中呈现一个相同的文件，却只占用一份存储空间。这个方法主要解决的是文件由大量重复的情况，适用场景比较有限，而且如果希望得到最佳的共享存储效率，需要对所有文件的哈希进行比较，计算量会严重拖累系统运行。

## 无拆分条带

### 新文件均衡

文件均衡的策略本质上可以转换为基于物理存储、网络拓扑，对某一个挂载卷的某一个文件进行排序的算法问题，类似Ceph的CRUSH算法，本系统提出了一个更加精确的排名策略与建模方法，兼顾访问均衡、存储均衡和读写效率问题。

#### 访问效率评分

首先系统会给每一个挂载卷维护一个计分表格，这个表格用于从多个维度评价挂载卷与DS和物理存储之间的关联度。关联度越高，代表文件存放在当前物理存储中可以获得更高的综合读写效率。具体的表格评分结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 含义 | 更新策略 |
| Mount | 有上限的数值 | 当前DS节点的Mount关联度 | 每一个活跃的客户端连接都会为指定DS增加Mount字段的分数；这个分数会在指定周期进行统计更新；如果该数值为0，则对应的DS的计分信息会被删除 |
| Access | 有上限的饱和计数器 | 当前物理存储的访问频次 | 所有基础文件的处理操作包括查询、读写等，都会触发对该计数器的递增，但是该递增存在上限；系统会在指定周期对该字段减去固定数值，作为定时损耗 |
| NetLatency | 有上限的数值 | 当前DS节点的平均网络延迟 | 系统会在指定的周期统计所有连接当前DS节点客户端的网络延迟均值，转换为固定分值后更新到当前字段内 |
| RPCLatency | 有上限的数值 | 当前DS节点的平均RPC请求响应延迟 | 系统会对使用追加平均数计算RPC的访问延迟，每一次访问都会更新该数值，该数值越大代表当前节点越忙 |
| ReadCnt | 有上限的饱和计数器 | 当前DS节点的读次数 | 系统会为每一次读操作进行一次记录，任意的读操作（一次RPC调用），无论读取数据大小，都会被记作一次读取操作；系统会在指定周期对该字段减去固定数值，作为定时损耗 |
| WriteCnt | 有上限的饱和计数器 | 当前DS节点的写次数 | 系统会为每一次写操作进行一次记录，任意的写操作（一次RPC调用），无论写数据大小，都会被记作一次写操作；系统会在指定周期对该字段减去固定数值，作为定时损耗 |
| CacheMiss | 有上限的饱和计数器 | 当前DS节点所处理的缓存未命中次数 | 系统在每一个节点都会设置缓存，这里的缓存指的是尝试在当前DS节点处理源挂在卷相关请求时发生的Miss次数，每一次Miss都会被递增的记录下来; 系统会在指定周期对该字段减去固定数值，作为定时损耗 |

上面表格给出的评分信息后续会依据实际的使用场景进行不同的调整，借助上述统计信息可以在一定程度上了解到挂载卷在不同的DS节点的响应情况，以及历史的访问规律。系统会按照下面的公式计算出一个整体分值，这个分值越高的DS节点，选取所带来的读写效率就越好：

在上述公式中，Mount作为前置乘数可以避免未参与挂载的DS节点参与到排名计算当中。Mount的客户端数量直接决定了文件存储到对应DS节点的访问效率收益，后面部分出现的ConfigParam参数来自于配置文件，用于配置不同访问内容对访问效率的影响权重。一般来说访问延迟和CacheMiss导致的延迟是最为直接的统计数据，这两个部分的权重要更大一些；相对的带宽和读写频次在有明显劣化倾向的时候，延迟也会随之增加，因此这个部分的权重参数可以小一些。同时考虑到创建目标文件的客户端会更加倾向于随后进行读写，因此对直接连接客户端的DS节点会给予更多的分数加成。

#### 存储均衡评分

存储均衡的处理一个难点就是存储均衡化与访问效率之间的不可兼得，特别是在使用NFS-V3协议的情况下，存储的均衡必然会导致访问效率的下降。系统为了确定一个挂载卷是否存在当前问题，会记录该挂在卷当前正在挂载的协议数量，目前统计的主要是NFS-V3和NFS-V4两种协议的比例。当NFS-V3协议占主要（超过配置文件中设置的阈值）的时候系统会优先考虑访问效率，对访问效率的评价给予更高的权重；当NFS-V4（PNFS）协议占主要的时候，系统会优先对存储进行均衡，以便可以同步获取到更高的访问效率。

* NFS-V3协议占据主导

在NFSV3协议占据主导的情况下，存储均衡会在排名访问效率评分排名靠前的几个DS节点中进行筛选。存储均衡粒度在物理存储级别，而单个DS可能同时提供多个物理存储，因此存储均衡实在物理存储层面上进行评估的。系统会选择剩余空间最多的磁盘作为存储均衡选择的物理存储，这种策略所存在的问题就是在物理存储本身空间差异化严重的情况下，整体存储会严重集中在最大的物理存储中，可能导致当前物理存储所在DS节点的较大压力。但是由于系统已经优先基于访问效率进行了评分和排行，所以最终所选择的物理存储应该是访问效率更高的DS节点中剩余空间最多的一个。

在这里系统设置了几个可配置参数获取访问效率均衡要求较高的DS节点，一个是最多的候选节点个数，一个是第一名的下调差异限制（百分比）。其中第一个数值用于确保不会产生较多的候选节点，一般这个数值可以设置的相对大一点，可以确保系统候选范围足够宽泛。后者则用于过滤掉排名靠前但是分值差异较大的DS节点，这个配置参数可以是10%，那么候选DS节点的访问效率评分不能低于第一名的90%，否则将不能作为候选参与下一步的存储均衡处理。

* NFS-V4协议占据主导

在NFSV4协议占主导的情况下，使用pNFS协议，客户端与服务端进行直接沟通，这导致存储均衡更希望通过将文件分不到不同的DS节点，以增加文件的并发读写效率。具体策略如下：

暂未确定

### 动态文件均衡

动态文件均衡所对应的问题是存储的热更新，比如某一个物理卷下面在运行状态下增加或减少物理存储。新物理存储的加入会使得当前物理卷下的物理存储严重倾斜，而旧物理存储的删除则会导致物理存储需要执行转移操作，以确保删除的物理存储内容不会丢失。同时文件的不断写入，导致单个文件超出预期甚至超出所在物理存储的承受极限，那么系统将不得不重新调整存储结构以保证大文件的继续写入。动态均衡只会在物理卷内出现，不同物理卷之间的存储是完全隔离的，不会出现任何存储共享交集。

#### 增加物理存储

增加新的物理存储是否触发系统界别的均衡迁移操作，主要取决于新增物理存储的位置和容量大小。当新增物理存储的存储容量超过现有物理存储的最大容量，并且在扩容前总容量的30%（该数值可以在配置文件中修改）以上或达到一定固定数值的时候，会触发动态文件迁移。迁移主要是为了缓解现有物理存储的存储压力，因此迁移文件的选择也会基于现有物理存储的情况进行选择。系统基于存储空余百分比、剩余存储空间、近期读写数据规模对现有的物理存储进行排序，选择较为拥挤的物理存储进行文件迁移。系统没有采用类似访问效率评分的计分制，而是使用排名权重加和的方式获取优先均衡的物理存储。使用下面的评分对所有当前物理卷中的物理存储进行排序，对排名靠前的物理存储进行逐个处理，直到满足动态文件均衡的停止条件（比如第一名的分数和最后一名的分数差值小于某一个阈值）。

需要均衡处理的物理存储选择完成后，就要执行具体的动态文件均衡操作了。文件动态均衡在最简单的场景下就是数据的直接迁移，从旧存储迁移到新存储，存储迁移势必会对带宽产生很大的占用，这将会导致系统网络的不稳定。同时迁移的文件如果很大，迁移过程中的读写操作如何处理也是一个很复杂的问题。因此新增物理存储后的动态文件均衡操作是在第一次新增以后的夜间执行具体操作的，这个具体执行的时间可以在配置文件中进行修改。在具体执行操作前可能系统已经开始对新增的物理存储进行使用，因此动态平衡会在评估之后根据实际的评估结果选择是否执行操作，具体的处理流程如下：

1. 依据存储均衡评分排行情况选定操作的目标物理存储和源物理存储，最终目标是通过从源物理存储迁移文件到目标物理存储实现存储分布均衡；
2. 系统对该物理卷分配在源物理存储中倾向于选择最近访问不是很频繁的文件进行迁移，避免迁移过程中文件的读写操作。为了避免排序的消耗，系统会对近一个月内访问过的文件进行标记，迁移会通过顺序访问从近一个月没有访问过的文件进行选择；
3. 系统会优先选择文件大小小于一定阈值的文件，避免长时间文件迁移对正常系统访问的影响，除非没有其他选项。
4. 对于迁移需要很长时间的超大文件，迁移会配合修改日志进行，从而可以避免迁移影响原始文件的正常读写。在迁移完成后，系统通过数据修改日志回放，实现两边数据的同步，经过验证之后完成文件迁移。

#### 删除物理存储

将一个物理存储从物理卷中删除，所产生的影响是很大的，相比于物理存储失效可能导致文件的直接丢失，物理存储的删除会触发一个漫长的物理存储迁移清理处理。物理存储的迁移清理处理，主要是将该物理存储中的文件均匀的迁移到其他存储中，并将当前物理存储中的文件逐个删除。这个迁移过程也很简单：

1. 遍历当前物理存储中的所有文件，逐个执行下面几个步骤；
2. 如果该文件是原始数据文件并且不存在备份，执行新文件创建流程，在新文件均衡阶段屏蔽删除的物理存储卷；如果该文件是原始数据文件但是存在备份，对文件进行标记修改，从备份文件中选择一个作为原始文件，然后将该文件当作备份文件处理；
3. 如果该文件是备份文件，删除文件并将备份更新信息入队列，在整个删除操作完成后，采用异步的形式对所有的备份文件进行更新优化。

从上面的步骤可以看出对于存在至少一份冗余备份的条带，删除物理存储的处理效率很高，并不涉及到文件的真实迁移，更多属于是文件信息的变更，如果文件数量较多，这个操作依旧会花费比较长的时间。后续所有实际的文件迁移操作会被转化为一系列文件的备份更新操作，当然如果系统没有配置冗余备份，物理存储删除将会触发实际的存储迁移，在数据迁移完毕前，删除操作将不会结束。在执行删除以后，对应的物理存储将会被屏蔽，但并不影响相关文件的读写操作。

#### 文件大小超限

文件大小超限是一个常态化服务经常遇到的场景，对于固定的块设备，文件写操作导致设备空间不足是一种较为常见的场景。如果使用无拆分条带，在对某一个文件执行连续写并且超出所在物理存储极限的情况下，写操作将会出现问题，这个时候系统会强制对该文件进行拆分，并将写操作分发到存储拆分后文件的物理存储中。拆分后的文件本身没有信息数据，相应的拆分文件信息数据会被追加到原始文件的信息数据中。这个拆分操作是可以无限制延伸的，每一次拆分都会触发一次内部的新文件创建，按照存储负载均衡的基本逻辑选择最佳的物理存储进行存储。无限制的拆分会造成文件信息的膨胀，因此系统会对这样的拆分文件设置单独的监控线程，当对应的文件处于未访问状态的时候对不同的文件块进行闲时合并，将小块文件合并成更大块的文件，以便减少文件信息的膨胀。合并操作会跟随增加物理存储后的存储均衡操作一起执行，在闲时执行数据的迁移，并在迁移完毕后执行原始文件的变更处理。

### 容灾备份

容灾备份的功能主要用于数据的抗灾抗融毁情况，不同的冗余备份策略往往针对不同的数据灾害场景设计。容灾备份的核心就是对所有的原始数据设置1份或者更多的备份数据，或者使用奇偶校验码等其他编码技术确保数据的完整性，不管使用何种方法都会需要牺牲一定的存储空间来存放备份或额外的校验信息。鉴于具体实现的便捷性，无拆分条带只能支持完整数据备份模式。完整数据备份是对存储空间消耗最大的一种备份形式，随着备份数量的增加，安全性也会随之增加，相对应的也会带来巨大的存储开销，整体的存储有效利用率降低非常迅速。因此综合使用成本和处理效率，一般会优先选择单一备份的版本进行处理。冗余备份的处理主要包含几个重要的问题：备份存储位置、备份数量、备份同步写，每一个问题都需要系统精准的控制策略才能保证处理的一致性和效果。

#### 备份存储位置

备份数据的作用主要在于在原始数据出现无法回复的错误时，及时进行补充并进行数据恢复。这个时候备份存储的位置十分关键，如果备份数据和原始数据存放在同一个物理存储，一旦该物理存储崩溃备份数据和原始数据同时损坏，将会遭遇不可恢复的问题。因此备份数据需要按照系统中的网络拓扑结构进行备份数据划分，将备份数据按照不同距离存放在灾害不容易波及的物理位置。

以系统设置4份备份的情况为例，第一份备份可以设置在当前DS节点的另一个物理存储中，以获取最快的备份恢复速度以及恢复期的访问效率；第二份备份可以设置在同一个机架的另一个DS节点的物理存储中；第三个备份可以设置在另外一个机架的DS节点的物理存储中；最后一个备份则存放在另一个相对较远的机房的物理存储，用于最严重的机房灾备。上述例子属于是理想的情况，实际情况如果用户没有对服务器位置进行标记，那么系统是无法按照最理想的方式进行备份管理的。

如果系统没有对纳管的服务器DS节点位置进行标记，则会选择优先选择不同的DS节点物理存储节点进行备份，然后在系统中按照操作延迟选择延迟较低的最近的几个DS节点进行备份。备份创建时的均衡方法可以参考新文件创建时的均衡位置选择，并将备份分布到不同的DS节点中。下面给出了一个简单的单备份数据创建构成的时序设计图：



1. 客户端发起新文件的创建请求，服务端会首先向MDS发送创建请求；
2. MDS接收到请求后基于本地整理的评分信息，筛选出最佳的原始数据存放位置和备份数据存放位置；
3. MDS向存放备份的DS节点发出请求要求创建对应数据的备份；
4. MDS备份数据创建完毕后将创建后的文件信息返回给文件所在DS节点（如果文件原始数据直接创建在DS2的话，否则返回数据前会同步创建原始数据信息），原始DS节点更新文件数据信息并完成文件创建操作。

#### 备份同步写

为文件配置备份文件可以有效的解决数据容灾问题，但是备份也会带来新的问题，诸如：存储空间利用率低、备份同步资源消耗等等。如果希望备份文件数据能够和原始数据能够保持一致，在出现数据灾难时获取到尽可能新版本的文件数据，近乎实时的备份同步十分关键。下面给出了一个单备份同步读写操作的时序设计图：



1. DS1执行对文件的写操作，本地存在文件的数据缓存，进行本地数据更新，同时基于本地的缓存信息找到原始数据存放节点DS2并发送远程写操作；
2. DS2接收到了远程写处理，执行本地写操作，并进行了修改数据记录，修改数据记录只记录一个两个HeartBeat周期的更新信息；
3. DS2处理了本地写操作以后，将FH信息同步更新到MDS中，MDS对这个文件进行了预标记，这个标记在开启文件备份的情况下是有用的；
4. DS2处理完成后将结果信息返回给DS1，DS1会对返回的FH信息进行二次确认；
5. 进入到新的HeartBeat周期，MDS会将这个周期内被标记过修改的数据分发给不同的DS节点，并进行信息确认；同时DS1清除上个HeartBeat周期存储的修改信息数据；
6. DS3作为备份存储节点，发现了数据有变化，因此从DS1的原始数据节点拉去最新的修改数据将数据进行回放并更新到本地的备份数据中完成写同步。

### 文件热操作

文件热操作针对的是文件处于迁移、恢复等特殊状态处理过程中，如何实现同时的读写操作，其实具体不同的场景处理方法并不复杂。由于无拆分条带本身并不能对文件数据进行拆分，出现的操作均属于文件级别的处理，具体的处理流程并不完全一致：

|  |  |  |
| --- | --- | --- |
| 文件状态 | 触发时机 | 热更新策略 |
| 均衡迁移 | 因为系统存储出现不均衡的现象，导致部分文件需要在闲时执行迁移操作 | **有备份：**存储均衡导致的迁移，在系统存在备份的情况下会优先将一个校验合格的备份文件转换成原始数据文件，然后会执行备份数据恢复，当备份数据恢复完毕后，根据均衡后的情况确定是否选择新的备份数据重新作为原始数据进行服务；  **无备份：**如果系统没有设置数据备份，那么文件迁移时原始数据会保持位置不变持续提供服务，但是数据的吸入和读取则会被强制以日志缓存的形式存储起来，数据迁移期间原始文件数据将不可变更；在数据完成迁移之后，系统则会保持日志缓存，并开始在迁移的数据中执行操作回放，直到没有新的日志出现时，完成FH信息更新，将迁移前的数据删除，实现数据迁移的热处理。 |
| 超限切分 | 因为连续写导致文件所在物理存储已满 | 当文件写操作导致溢出，DS节点会立即向MDS节点申请存储均衡后的分块信息，并更新到本地，然后远程调用新分块所在DS节点继续处理写操作，后续的读写操作都会通过offset计算得到实际的处理流程，调用对应的节点或使用本地缓存完成处理 |
| 文件恢复 | 当文件出现数据损坏，使用其他备份数据恢复原始数据 | **有备份：**有备份的情况下，文件恢复的处理策略与均衡迁移的处理策略一致；  **无备份：**如果文件恢复的时候没有备份数据，那么对于无拆分条带文件数据是无法恢复的，这个时候就系统就会按照文件错误的情况处理，清楚相关所有缓存信息 |

## 半拆分条带

暂不支持

## 全拆分条带

暂不支持

# 配置文件

配置文件是调整多个系统运行参数的最主要方法，部分系统运行时参数也可以通过客户端程序进行修改，相应的修改如果涉及配置文件，那么配置文件中的数值也会被系统进行同步调整。目前系统的配置文件主要分为两个部分，一个采用类NFS的“/etc/exports”配置文件，一个则是系统全局配置参数对应的配置文件“/etc/dnfs/main.conf”。下面的内容将会对这两个配置文件的配置信息进行详细描述说明。

## DS端配置文件

DS端配置文件主要用于配置当前系统的物理存储信息以及DS相关的配置信息，默认配置文件路径采用NFS服务相同的“/etc/exports”，由于本系统本身与NFS服务存在冲突无法同时部署，因此共用配置文件路径并不会带来使用上的冲突。下面给出了一个典型的DS端配置文件示例，配置文件采用yaml文件格式进行数据组织：

MDS: # DS端的MDS配置，每一个DS只能连接唯一一个MDS搭建分布式存储

Location:192.168.12.118 # MDS服务器的地址，IP或域名均可

Token:uy7dy63yhe8ytat98ijju7627dgd7 # 如果MDS开启了ds\_authentication，那么这一项需要填写正确的密钥才可以正常使DS和MDS完成通讯连接

DS:

PhysicalLocation:/Beijing/CSC-CY/Room212/Rack12-33/Server\_127\_122 # DS端的物理位置信息，用于显式标明位置，辅助系统进行高效的备份数据位置选取

log: # DS本地的日志信息配置，如果没有设置则会采用MDS端的配置

limit\_type:time # 日志轮转的处理方法，可以是time、size，如果不填写则不会进行日志轮转，所有日志都会写入到一个文件中

limit\_info:MIDNIGHT # 日志轮转处理信息：如果轮转类型是time，这里说明的是日志轮转时间点；如果轮转类型是size，这里说明的是日志轮转时原始文件的大小限制

backup\_count:30 # 日志轮转后历史日志文件的最大个数，超出这个个数的历史日志文件会被删除

formatter:”%(levelname)<%(asctime)><PID-%(process)>: %(message)” # 日志文件的输出日志格式化字符串

path: /var/log/dnfsd # 日志文件所在目录，系统会自动在对应目录下创建不同的文件记录不同模块的日志信息

exports: # 当前DS节点对外开放的存储路径或存储设备

- /mnt/sdb1/data # 开放一个目录，系统基于VFS对该物理存储进行管理

- 192.168.2.11:/mnt/sdb2/temp/data # 一个NFS协议对应的目录地址，系统会基于NFS协议对该存储进行管理

- /mnt # 如果存在多个开放目录，系统会对其所在块设备进行冲突检查

- /dev/sdc2 # 开放一个块设备，系统使用自己的块设备策略实现块设备格式化和数据高效存储

其中Config中存放了两项配置信息，分别是MDS和DS。MDS中存放的是MDS服务器的地址Location以及验证密钥Token，如果系统在MDS一侧设置了免密钥登录那么Token选项可以不设置。DS中存放的是本DS服务端相关的私有配置信息，比如本地的物理位置信息PhysicalLocation，被用来显式标记服务器的网络拓扑位置。

Exports中存放的是当前DS节点希望对外开放的物理存储，这个物理存储可以是一个目录也可以是一个块设备，系统会自动进行识别并使用匹配的文件管理后端实现兼容性处理。同时系统也会进行冲突检查，确保当前DS节点暴露的物理存储之间没有冲突。

使用配置文件管理配置信息而不是使用类似Ceph的命令行进行配置管理，存在配置文件变更后服务重启导致变更不可察觉的风险。因此本系统支持同时通过配置文件和命令行进行配置修改：前者需要通过重启完成配置更新，系统会在重启时自动对存储卷结构进行比对完成物理存储的修改处理；后者则会在完成修改处理的同时完成对应DS节点配置文件的内容修改。

## MDS端配置文件

系统支持多个MDS端服务器，而MDS端配置文件在各个MDS端是完全同步的，在主MDS创建并生成MDS配置文件进行部署启动就可以完成配置同步。MDS配置文件只会存在于主MDS服务器中，一旦出现了MDS失效主备切换，备份MDS服务器会将缓存中的配置文件信息写入到本地物理地址中，并切换为主MDS；如果出现主MDS恢复，原本的主MDS切换为备份MDS，则会在主MDS完成配置文件生成后删除本地的配置文件。MDS配置文件默认被存放在“/etc/dnfs/main.conf”中，其中“/etc/dnfs”下面存放着不同的系统服务级别的元数据信息。下面给出了一个完整的配置文件示例，其中包含了所有可以进行修改的配置信息，系统会在初始化部署的时候自动生成一个默认的配置文件，内部的参数均为系统的默认数值，用户也可以自行选择创建这个文件并预先设置好配置。

MDS: # MDS相关的配置信息

servers:# MDS服务器配置，需要用户手动设置或通过命令修改，增加至少一个主MDS服务器，否则系统无法运行

- 192.168.1.25(main,/etc/dnfs/meta\_data) # 主MDS服务器。括号中的第一个代表当前服务器的类型，可以是main或backup，backup则代表备份MDS服务器；括号中的第二个代表系统希望在哪一个目录下存放MDS中元数据的物理数据，这个会占用较大的存储空间

- 192.168.45.25(backup,/mnt/sdb1/dnfs) # 备份MDS服务器

- 192.168.4.21(backup,/mnt/data/dnfs) # 备份MDS服务器

frame: # 系统框架架构层面的配置

max\_mds\_count:3 # 最大的MDS数量，包含主MDS服务器，3代表可以有2个备份MDS服务器

heartbeat: # HeartBeat相关配置

timer:5s # HeartBeat间隔，5s触发一次MDS到DS的HeartBeat

timeout:1s # HeartBeat的超时时间设置，默认1s

server\_lost\_times:10 # 服务器失效尝试次数，10次HeartBeat都没有响应的情况下

metadata: # 元数据相关配置

persistence:

fh\_partition\_size: 256 # 文件信息持久化数据中fh的分区大小

dirfh\_partition\_size: 256 # 目录信息持久化数据中dirfh的分区大小

DS:

limit:

max\_ds\_count:4096 # 最大的DS服务器数量

max\_ps\_count:65536 # 最大的物理存储数量

min\_ps\_size: 256GB # 最小的物理存储容量要求，小于该数值将被禁止作为存储节点加入

security:

ds\_authentication:False # 是否开启DS的认证，如果开启认证，则DS端需要设置MDS部署形成的密钥才能正确连接到MDS

cache: # 缓存相关配置

adaptive: # 缓存自适应设置，用于调整不同配置DS节点下的缓存规模

basic\_memory\_size:16GB # 默认配置对应的系统内存大小

min\_memory\_size:4GB # 系统要求的最小内存大小，小于该数值的DS节点不会被接纳

cache\_size\_adaptive:True # 是否开启大小自适应，如果没有开启则所有DS节点无论内存多大都会使用相同的缓存大小配置

mds\_cache\_size\_times:16 # MDS缓存大小倍率，在MDS中的所有缓存大小限制是基于DS乘上当前倍率后得到的，因此无需单独设置MDS缓存单项大小

fh\_cache: # FH缓存相关配置

max\_group\_count:1024 # FH缓存组的最大个数，系统会基于这个数值进行分组

max\_group\_size:1024 # 单个FH缓存组中能够存放的FH最大个数

replace\_policy:LRU # FH缓存组内的替换策略，其他策略另作讲解说明

dir\_cache: # 目录信息缓存配置

dir\_replace\_policy:LRU # 目录信息的替换策略

limit\_cache: # 有限目录缓存配置

max\_dir\_count:256 # 可以存放的最大目录个数

max\_fh\_count:1024 # 单个目录下可以存放的最多的映射关系数量

fh\_replace\_policy:LRU # 单个目录下FH超出限制的替换策略

unlimit\_cache: # 无限制目录缓存配置

max\_fh\_total:16777216 # 无限制目录缓存中最大的FH总数

max\_dir\_count:32 # 无限制目录缓存的目录限制个数，这个参数和FH总数限制都会作为限制确保对应信息存储量不会过大

data\_cache: # 文件数据缓存设置

block\_size:256KB # 文件数据块大小

file\_data\_cache: # 文件访问缓存

max\_block\_count:1024 # 缓存可以存放的最大块个数

replace\_policy:LRU # 缓存块超出限额后的替换规则

fh\_access\_count\_max:64 # 文件访问计数器上限

fh\_access\_count\_decay\_time:15s # 文件访问计数器消退计时间隔

fh\_access\_count\_decay:16 # 文件访问计数器消退大小

write\_log\_cache: # 文件异常状态写操作缓存

max\_block\_count:512 # 最大缓存的文件块个数

block\_overflow\_ratio:110% # 写缓存的溢出比例，当写缓存超出基准最大值的时候就会触发写回，在写缓存到达溢出比例的时候，将会强制停止写操作

block\_writeback\_stop\_ratio:80% # 写缓存触发写回操作的时候，写回到最大值多少比例的时候才停止写回

log: # 日志相关配置，DS自己的日志配置会覆盖总的MDS日志配置

limit\_type:time # 日志轮转的处理方法，可以是time、size，如果不填写则不会进行日志轮转，所有日志都会写入到一个文件中

limit\_info:MIDNIGHT # 日志轮转处理信息：如果轮转类型是time，这里说明的是日志轮转时间点；如果轮转类型是size，这里说明的是日志轮转时原始文件的大小限制

backup\_count:30 # 日志轮转后历史日志文件的最大个数，超出这个个数的历史日志文件会被删除

formatter:”%(levelname)<%(asctime)><PID-%(process)>: %(message)” # 日志文件的输出日志格式化字符串

path: /var/log/dnfsd # 日志文件所在目录，系统会自动在对应目录下创建不同的文件记录不同模块的日志信息

# 关键优化技术

## 数据预取

### 元数据预取

为了避免对持久化文件的频繁读写，DS节点会采用一定的预取策略来提前将相近的文件信息以及目录信息提前加载到缓存中。主要的元数据预取策略均在DS端实现，MDS因为缓存容量较高，则不执行预取策略，具体的预取策略和触发时机如下。

#### 目录访问

当客户端执行cd路径操作并触发ACCESS请求时，意味着用户接下来的操作可能是ls，或者进入下一级目录，或者直接操作当前目录下面的文件。在遇到前述场景的时候，DS服务端会开始预取目标目录的目录信息、目标目录上一级目录的目录信息，同时引导MDS节点预取目标目录中所有文件信息，如果这个数量很大，则会按照访问频次取前几位的文件进行预取；目标目录下一级目录中访问频次最高的几个目录的目录信息。

为了能够统计一个文件夹下面数据的访问频率，元数据方面追加一个针对访问频次的统计数据，记录在目录信息中。其中目录下的每一个FH记录一个访问次数，同时目录本身记录一个目录下文件访问的总次数，某一个FH的访问会在两者都加1。当目录下文件访问总次数溢出时，总次数以及每一个FH的访问次数都会右移若干位，进行递减。总次数相当于一个计数装置，在特定访问量时触发访问次数递减，以确保访问次数记录的永远是最近的访问分布情况。系统通过对一个目录下FH的访问频率排序，便可以找到最常用的文件或目录并进行精准的预取操作。

#### 文件访问

针对某一个文件的访问大概率与同一个目录下其他文件的访问没有关联，这就使得文件访问操作无法为文件元信息预取提供辅助，因此系统暂未设计针对文件访问的预取策略。

#### 缓存污染

在缓存中，预取的数据本身不应该污染之前通过实际访问拉取到缓存中的数据，否则较低的预取准确率会导致缓存的严重污染。因此预取数据达到缓存的时候，在LRU结构中会放置在中间位置，或者中间靠后的位置，以保证最近常用的信息不会受到预取的干扰。

系统还会为每一个客户端维护一个统计，记录该客户端在服务端节点的预取命中概率，系统会在执行预取时会更加倾向于预取命中率更高的客户端，提供更多的预取内容或者更长的缓存生命周期。

### 文件数据预取

相比于文件元信息，文件读写习惯更加难以察觉，执行预取并不像CPU中的缓存一样可以获得代码级别的访问规律，这种访问规律在文件块上几乎无法体现。

#### next-block预取

文件数据在客户端也会建立缓存，整个处理过程更倾向于连续读多个文件块的处理，因此文件读写的预取更加倾向于next-block的预取方式。然而文件数据的连续读取往往只读取一定区域内的数据，超出范围后不会执行读取操作，next-block策略在区域边界会产生预取失效，但对于连续读取的处理效果较好。这个策略在执行文件写操作和读操作的时候均会触发预取。我们假设DS服务端直接拥有访问文件的原始数据，当DS服务端在收到一个READ操作的时候，系统会自动将当前读取内容的下一个Block或2个Block预取到缓存中。预取操作读取Block的个数与触发预取的READ读取的数据大小有关。

#### 预取限制

系统会维护一个近期预取的准确率，在预取准确率下降的时候降低预取的概率。为了保证预取概率拉升响应足够快速，即使预取操作被取消，预取的信息也会被保留下来，因此系统依然可以计算得到一个预期准确率。每一个客户端与会维护一个预期预取准确率统计，同时也会计算得到一个准确率对应的概率，准确率越高，预取决策被采用的概率越高，在这个概率基础上预取是否执行是完全随机的。同时这个概率有一个最小值，以确保预取策略变准确的时候能够快速拉升预取概率。

## 自适应端端连接

当客户端尝试挂载一个挂载卷的时候，可以与任意一个节点创建连接关系，但是由于不同节点的服务压力并不相同，因此存在一个最佳的连接节点可以获得最好的处理效率。针对NFS-V3协议，服务端是固定的，但是系统会提供一个最优服务端的选择算法，DS端可以根据选择的结果创建连接。最优服务端的选择算法主要考虑的是服务端的实时性能状态，确保负载均衡，不会在单个DS节点上集中过多的客户端导致节点访问效率下降。同时最优服务端还会也会考虑文件局部性，优先选择挂载卷文件最集中的DS节点。为此系统设置了一个评分公式，使用到的统计参数如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 含义 | 更新策略 |
| Mount | 有上限的数值 | 当前DS节点的Mount关联度（任意客户端） | 每一个活跃的客户端连接都会为指定DS增加Mount字段的分数；这个分数会在指定周期进行统计更新；如果该数值为0，则对应的DS的计分信息会被删除 |
| ClientMount | 有上限的数值 | 当前客户端在DS节点上的挂载连接数 | 当前客户端与DS节点之间的挂载关系连接个数；这个分数会在指定周期进行统计更新 |
| MVMountRatio | 有上限的数值 | 当前挂载卷在DS节点上的历史挂载比率 | 当前客户端与各个节点之间的历史挂载次数中当前DS节点的所占比率，这个数值在MDS节点进行统一收集处理 |
| NetLatency | 有上限的数值 | 当前DS节点到客户端的延迟 | 系统会在创建连接之前，会对客户端到DS之间的延迟进行测试，这个测试会由MDS发起，由各个DS完成操作 |
| RPCLatency | 有上限的数值 | 当前DS节点的平均RPC请求响应延迟 | 系统会使用“追加平均数”计算RPC的访问延迟，每一次访问都会更新该数值，该数值越大代表当前节点越忙 |
| MVRPCLatency | 有上限的数值 | 指定挂载卷在指定DS节点上的响应延迟 | 系统会使用“追加平均数”计算RPC的访问延迟，但是该延迟来自于指定的挂载卷，每一次访问都会更新该数值，该数值越大代表当前指定挂载卷在该节点越忙；对于没有关联的挂载卷，这个数值被替换为RPCLatency |
| MVFileHoldRatio | 0-1的浮点数 | 指定挂载卷元数据在当前DS节点的占比 | 系统会在每次创建文件或者目录的时候，记录挂载卷在不同DS中的元数据分布统计信息，记录这个比例到统计数据中 |

上面表格给出了自适应服务端选取所需要的统计信息，依靠上述统计信息，系统提供了一个评分计算公式。借助这个公式系统会对所有可以提供服务端功能的节点进行排序，选择分数更高的节点作为最优的服务端提供给客户端，完成NFS-V3的连接。具体公式如下：

依托上面的公式主MDS会对所有可用的DS节点进行打分，然后排序后获取排名靠前的节点作为最终的服务端返回给客户端建立连接。

## 大文件多备份高并发访问

高并发访问实在采用NFSV4中的pNFS协议时才能采用的方法。在这种策略下，如果系统物理卷使用的是拆分条带，可以直接通过并发访问不同物理存储的文件块来加速访问；如果使用的是无拆分条带，那么文件数据不会分布在不同的服务器上，这个时候只有系统开启了冗余备份策略，才可以使用备份文件实现并发数据读取，一定程度上加速大文件的读取速度。

# RPC层优化

本章节对NFSV3的RPC层接口函数进行逐个的分析和说明，并对其中部分较为常用的接口函数如何加速进行一定的介绍。下表给出了所有RPC层的函数处理情况：

|  |  |  |
| --- | --- | --- |
| 函数名 | 专用加速 | 处理方法 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# 客户端