分布式文件系统元数据综述

叶泽坤[1)]

[1)]华中科技大学 计算机科学与技术学院 湖北省武汉市 435400

摘　要　一个大型文件系统需要支持数十亿个文件并为大量客户服务。这给分布式文件系统的元数据服务带来了严峻的挑战。元数据服务是大规模分布式文件系统的可扩展性瓶颈。分布式文件系统是数据中心的重要基础设施组件。随着数据中心内文件数量的快速增长，元数据服务成为分布式文件系统的可扩展性瓶颈。本文介绍了相关分布式文件系统得元数据存储结构与相关架构。

关键词　分布式 文件系统 文件元数据 存储

Overview of Distributed File System Metadata

YeZekun[1)]

[1)] School of computer science and technology of Huazhong University of Science and Technology, Wuhan City, Hubei Province 435400

Abstract　A large file system needs to support billions of files and serve a large number of customers. This poses a serious challenge to the metadata services of distributed file systems. The metadata service is a scalability bottleneck for large-scale distributed file systems. Distributed file systems are important infrastructure components of data centers. With the rapid growth of the number of files in data centers, metadata services have become a scalability bottleneck for distributed file systems. This article introduces the metadata storage structure and related architecture of distributed file systems.

Key words　distributed file system，file metadat，storage

1　介绍一

1.1　背景

现代数据中心为了快速扩展业务，通常包含大量文件，这很容易超过当前分布式文件系统单个实例的容量。目前的做法是将数据中心划分为相对较小的集群，每个集群运行一个分布式文件系统实例。然而，更好的方式是使用单个文件系统实例来管理整个数据中心，以实现全局数据共享、高资源利用率和低操作复杂性。

在大规模分布式文件系统中，元数据服务是可扩展性的瓶颈。分布式文件系统是数据中心的重要基础设施组件。随着数据中心内文件数量的快速增长，元数据服务成为分布式文件系统的可扩展性瓶颈。目前，数据中心通常由一组文件系统集群组成。例如，阿里云维护着数千个盘古分布式文件系统，共同支持数据中心中数十亿个文件。而 Facebook 则需要多个 HDFS 集群来存储数据集，因为由于元数据限制，每个 HDFS 集群最多支持 1 亿个文件。

早期的分布式文件系统（如 GFS、HDFS、Farsite 和 QFS）将文件数据分发到多个数据服务器，同时在单个专用的元数据服务器中管理所有元数据。然而，在拥有数十亿文件的超大规模场景中，它们的性能表现较差，因为元数据的数量超过了单个服务器的容量，并且由于资源有限，元数据操作的吞吐量受到限制。

一些分布式文件系统使用子树分区将目录树划分为子树，如 AFS Volume、Sprite Domai 和 HDFS Federation。基于子树的元数据分区可以实现较高的元数据局部性，但由于负载不平衡和数据迁移，可扩展性较低。一些分布式文件系统将目录树划分为用户可见的分区，并且不允许跨分区重命名。随着目录树的扩展，维护静态分区方案变得不切实际。CephFS 也将元数据划分为子

树，但当检测到负载不平衡时，它会跨元数据服务器迁移热子树。Mantle 提供了一个可编程接口，用于调整 CephFS 针对各种元数据工作负载的平衡策略。然而，当工作负载多样且频繁变化时，它们会受到频繁元数据迁移的高开销的影响。

一些分布式文件系统按照每个目录的粒度对目录树进行分区，如 IndexFS、HopsFS 和 Tectonic。由于细粒度分区，它们可以实现负载平衡和良好的可扩展性。然而它们牺牲了元数据的局部性，导致频繁的分布式锁或分布式事务。这些分布式协议带来了非常大的协调开销，导致高延迟和低吞吐量。

1.2 主要挑战

一个大型文件系统需要支持数十亿个文件并为大量客户服务。这给分布式文件系统的元数据服务带来了严峻的挑战。

一是负载均衡与元数据局部性。随着目录树的扩展和工作负载的多样化，目录树分区在实现高元数据局部性和良好的负载平衡方面具有挑战性。元数据局部性对于高效的元数据处理非常重要。文件系统操作通常处理多个元数据对象。例如，文件创建首先锁定父目录，以顺序进行目录列表操作，然后以原子方式更新三个元数据对象，包括文件元数据、条目列表和目录时间戳。通过元数据局部性，我们可以避免分布式锁和分布式事务，从而实现低延迟和高吞吐量的元数据操作。负载平衡对于实现高可扩展性非常重要。元数据操作通常会导致目录树中的负载不平衡。这对于真实数据中心工作负载尤其如此，其中相关文件被分组到子树中。来自连续子树的文件和目录可能在短时间内被大量访问，从而在存储子树的元数据服务器上造成性能瓶颈。在超大规模的场景中，现有的分区策略无法实现高局部性和良好的负载平衡。管理数据中心中的所有文件会使目录树在深度和广度上快速扩展。由于文件系统支持所有数据中心服务，因此它面临着具有不同特征的各种工作负载。这对现有的目录树分区策略是一个挑战。细粒度分区，例如直接将元数据对象散列到服务器如 HopsFS 和 Griaffa，可以实现负载平衡。然而，它牺牲了局部性并经常导致分布式锁，引入了昂贵的协调开销，导致高延迟和低吞吐量。粗粒度分区，例如将连续子树分组到同一服务器上，保留了局部性并避免了跨服务器操作。但是，它很容易受到工作负载倾斜的影响，从而导致负载不平衡。

二是路径解析时延。在超大规模的文件系统中，文件深度变得越来越深。当将所有服务整合到一个文件系统中时，文件的深度会迅速增加。深度目录层次结构对文件系统性能有很大影响。

三是客户端元数据缓存与近根目录热点问题。路径解析需要从根遍历目录树，并依次检查路径中所有中间目录的权限。这会导致近根目录被大量读取，即使对于均衡的元数据操作也是如此。然后，文件系统吞吐量将受到存储近根目录的服务器的限制。许多分布式文件系统依赖客户端元数据缓存来缓解近根热点，如 LocoFS 和 IndexFS。

2 相关研究

2.1 SingularFS

2.1.1 背景

数十亿规模的文件系统是云服务供应商和小型数据中心的构建要素。无须分布式事务的单点 metadata server 足够应付这个规模的元数据。它具有以下优点。一是 TCO 成本减少，即单个元数据服务器的安装、维护和日常成本比多个元数据服务器更便宜。二是简化实现，性能不错（无需分布式事务）。

同时当今的系统存在有一些问题。崩溃一致性的开销限制了多索引节点元数据更新操作的吞吐量。文件系统使用日志记录或日志结构化设计来提供崩溃的一致性。在日志方法中，数据写入两次，并按顺序检查。在日志结构的方法中，数据被写在新分配的地方，而把旧的地方作为垃圾。不幸的是，垃圾收集会导致很高的开销。共享目录中的操作在分布式文件系统中很常见，inode 上锁的争用限制了同目录下元数据操作的吞吐量。共享目录中的元数据操作的并发性会显著影响这些应用程序的整体性能。但是，并发 inode 在共享目录中创建和删除操作需要更新它们共同的公共父目录项（脏项）和 inode，这将导致高锁争用。这种争用限制了并行性和性能。文 NUMA 体系结构对于文件系统没有得到充分利用，特别是对于单个元数据服务器中的元数据操作。件系统在维护元数据操作的 NUMA 位置的同时，无法扩展到多个 NUMA 节点。将节点随机分散到多个 NUMA 节点中并不能避免这个问题，因为一些诸如 inode

创建和删除等操作需要更新多个节点，这会导致NUMA 间元数据访问。

2.1.2 SingularFS 设计

本篇论文有以下的关键技术：

提出了无日志数据操作，消除大多数元数据操作的额外崩溃一致性开销。关键思想是通过利用键值存储后端的单对象更新原子性以及父inode 和子 inode 的元数据语义依赖性来识别可能的元数据不一致性。SingularFS 向 inode 添加了两个字段，出生时间 btime 和死亡时间 dtime。

设计了分层并发控制，以最大化元数据操作的并行性。关键思想是以一种更细粒度的方式同步 inode 操作。具体来说，inode 写入器使用每个inode 的读写锁来与其他操作进行同步，inode 时间戳更新器和读取器使用无锁协议在它们之间进行额外的同步。

引入了混合内部分区，以减少 numa 间访问和numa 内锁争用。关键思想是将时间戳从目录 inode 中分离出来，并将其与目录的子节点分组到同一个 NUMA 节点，从而确保文件操作的 NUMA 本地性。SingularFS 进一步对 numa 内部的数据结构进行了分区，以减少其锁的争用。

目标是稳定的高吞吐量。远程内存直接访问技术（RDMA）和持久内存（PM）为它提供了机会。

2.1.1.1 整体架构

SingularFS 的体系结构，它包含两个组件，客户端和服务器。服务器在 PM 中维护一个全局文件系统目录树。客户端通过用户空间库提供的类似于 posix 的接口来执行文件系统操作。服务器和客户端都配备了用于网络通信的 RDMA 网卡。SingularFS 使用一个通用的键值存储（KV Store）作为其存储后端。KV 存储器应该能够执行点查询和前缀匹配。此外，它应该保证以低运行成本的单对象操作的原子性。

2.1.1.2 无日志元数据操作

元素据写入操作，有以下几类。一是单节点操作，只更新目标 inode 本身， 包括 open/ close/ read/ write。单节点操作的崩溃一致性可以直接保证。二是双节点操作，更新目标 inode 及其父目录的时间戳，如 create/delete。双节点操作将父目录的 ctime 设置为目标节点的 btime 或 dtime，所以父目录的 ctime 必然大于所有子目录的 btime 和 dtime。并且 d.time >max(c.time,b.time)。若创建 inode，inode 的创建包括两个原子步骤。首先，我们用 btime = t0 插入 inode，其中 t0 是当前的时间戳。其次，我们将其父目录的 ctime 和 mtime 设置为 t0。当创建两个步骤之间发生崩溃，或删除步骤 1 和步骤 2 之间发生崩溃时，可以通过检查父目录的最大 btime 和 dtime 是否通过将父目录的 ctime 和 mtime 设置为最大值来确定不一致性。若删除 inode，inode 的删除包括三个原子步骤。首先，我们使目标 inode 无效，并将其数据时间设置为当前时间戳 t0。其次，我们将其父目录的 ctime 和 mtime 设置为 t0。最后，我们从 KV 存储中物理去除目标内源。当在删除 inode 中的步骤 2 和步骤 3 之间发生崩溃时，可以通过检查目标 inode 是否无效，并通过物理删除无效的 inode 来确定不一致性。三是 rename 操作，更新原始 inode、新 inode、以及新老父目录的 inode。相对前两者，rename 占比较小（不到 10%）

2.1.1.3 无日志元数据操作

在传统文件系统中，共享目录的双节点操作会因为父节点 dentry 以及时间戳的竞争而被限制并发。dentry 由于 SingularFS 将 dentry 与 inode 用同一种 kv 融合了<ino+basename, ino+inode>, 所以这部分直接加一条 kv 就可以了。事实上时间戳的修改只涉及父目录的 ctime 和 mtime。SingularFS 因此把所有元数据操作分为三类：

1. updater：更新目标 inode 的 ctime 和 mtime 的操作；
2. writer：更新目标 inode 其它部分的操作；
3. reader：读目标 inode 的操作。

SingularFS 把修改父目录的 ctime 和 mtime 操作加读锁（而非写锁），然后用 16 字节的 CAS 指令来完成这部分的并发修改。最大化共享目录中的双节点操作的并行性具有挑战性，因为它们会导致关于父节点差异和时间戳的争用。在 SingularFS 中，父节点的更新与子 inode 的更新同位于一起，子节点的并发性由 KV 存储后端保证。因此，剩余的挑战在于并发时间戳更新。可以观察到，双节点操作只修改了父目录的 ctime 和 mtime，其大小总共为 16B。因此，通过利用 16B 原子的比较和交换指令，可以以无锁的方式执行时间戳更新的并发控制。

将目标 inode 上的操作分为三类：更新器、写入器和阅读器。更新程序包含 inode 的更新操作，它只涉及目标 inode 的 ctime 和 mtime，而写入器包含所有其他的更新操作。读取器包含所有的 inode 读取操作。例如，像文件创建/删除这样的双节点操作都是目标 inode 的写入器和父目录的更新器，并且文件 stat 是目标 inode 的读取器。
write 操作：更新目标 inode 其它部分的操作
read 操作：读目标 inode 的操作，ctime 单调增加，它与版本号具有相同的语义。ctime 由读者在获得整个 inode 之前和之后获取。阅读器通过比较两个时间来验证 inode。

update 操作：更新目标 inode 的 ctime 和 mtime 的操作。如果当前的 ctime 大于等于时间戳，则无需更新时间戳，因为它们已被另一个更新程序更新。如果当前的 ctime 小于时间戳参数，则原子地更新时间戳。

2.1.1.4 NUMA 优化

跨 NUMA Inode 间分区。对于单节点文件操作，将元数据请求委托给相应的 NUMA 节点，以确保它们的 NUMA 本地性。但是，这并不适用于双节点文件操作（创建/删除），因为这两个相关的内部节点可能驻留在不同的 NUMA 节点中。然而在 SingularFS 中，这些操作只修改了父目录的 ctime 和 mtime，因此通过将父目录的 ctime 和 m 时间与子文件内节点分组到相同的 NUMA 节点，可以保证它们的 NUMA 本地化性。因此，
SingularFS 将目录 inode 划分为时间戳元数据（atime、注册时间点和 mtime）和访问元数据 (inode ID、权限、btime、dtime 等)。对于每个目录，SingularFS 将目录时间戳元数据、它的子文件内节点以及它的子目录的访问元数据聚合到一个元数据组中。元数据组中的对象被放置在同一个 NUMA 节点中。SingularFS 使用一致散列将组分发到元数据服务器的 NUMA 节点，从而实现文件操作的 NUMA 本地性。

三节点（三个元数据对象）操作（mkdir）假设在 A 文件夹下创建了一个 B 文件夹。那么此时需要做更新的有 A 的时间戳元数据，B 的访问元数据，B 的时间戳元数据。因为 mkdir 时，所有时间戳会与 btime 一致。因此只需要先更新 A 的时间戳元数据和 B 的访问元数据，再更新 B 的时间戳元数据即可保证崩溃一致性

目录的权限修改操作（chmod）修改目录权限需要更新访问元数据和时间戳元数据的 ctime，这两部分不在一个元数据组里。这里 SingularFS 规定，chmod 操作会更新 btime。这样就可以利用访问元数据中的 btime 纠正时间戳元数据中的 ctime，从而保证崩溃一致性。

在一个 numa 内。kv store 依赖范围查询操作，来支持 dentry 的生成。

基于 B+树的有序索引在更新频繁的工作负载中会被遍历和节点分裂的锁争用拖后腿。为了缓解这个，Singular 在每个节点内部把元数据通过哈希分散到 n 个有序索引中。当发生范围查询时，需要到这 n 个索引中都查询一遍并合并结果。单点查询时，可通过哈希得到其结果在哪个有序索引中
另外可以在 inode 中添加一个 num_dent 字段来表示 dentry 的数量来缓解 rmdir 操作的一个前提(判断目录是否为空)。这个变量的崩溃一致性可以通过对目录执行完整的前缀匹配来恢复。

2.2 HadaFS

2.2.1 HadaFS 背景

近几年 AI，ML，HPC 大火， 针对这些场景的存储技术及方案也逐步衍生出两个分支，第一支：以 Lustre，BeeGFS 等为代表的分布式并行文件系统， 这些文件系统对 POSIX 提供了很好的支持，各种业务可以不经改造无缝运行，提供了很好的兼容性；第二支：以 Gekkofs，HadaFS 为代表的缓存系统 Burst Buffer（BB），这些系统都提供弱（宽松）语义的 POSIX 支持，通过自定义的客户端对上层业务提供文件访问能力，需要对上层的业务进行针对性的改造及适配，因为专门针对这些场景进行了优化，相对的也提供了更优的性能。HadaFS 是国家超算中心（无锡）联合多家高校设计实现的一款专为超算打造的宽松 POSIX 缓存系统（BB）。

Burst Buffer（BB）根据部署位置的不同，分为本地 BB 和共享 BB，本地 BB， 部署在计算节点的 SSD 上，专职服务于本节点，扩展性和性能要更好些，但是不适合用于 N-1 的共享数据场景，另外因为共享部署，不同 I/O 模型/负载的业务相互干扰，可能导致巨大的资源浪费，最后随着计算节点的升级，部署成本也会快速增加。代表产品有 Luster 的 LPCC，BeeGFS 的 BeeOND。共享 BB，部署在专用的节点上，它的优势是可以实现

数据共享以及具有更优的部署成本，但是在支持拥有数以万计客户端的超大规模计算集群上面临挑战。

两种 BB 各有优劣，而相比传统的文件系统，BB 的性能都较高，但是容量较小，所以 BB 通常和文件系统配合使用。但随着 E 级超算时代的到来，并发 I/O 的需求急剧增加，同时超算应用的 I/O 也更多样，这给当前的 BB 系统带来了巨大的挑战，并暴露出如下的不足

BB 的扩展性与应用行为的不匹配， E 级超算数以万计的并发 I/O 给 BB 系统的扩展性带来挑战，AI 及 ML 等共享数据的应用及工作流对 I/O 提出新的要求，大规模数据的高速共享变得愈发的重要。如：神威太湖之光使用的 Luster LPCC 本地 BB，在数据共享以及元数据访问密集的情况下就比较低效（为保证强语义的一致性，BB 中的数据需要回写到 Lustre 后，才能共享）。

复杂的元数据管理与应用行为的不匹配，传统的文件系统为通用场景而设计，基于标准的目录树管理及 POSIX 协议实现。而 HPC 场景，通常只进行文件的读写，很少进行（复杂的）目录树管理。所以采用宽松的 POSIX 协议是一个更好的选择，但由于 HPC 应用的多样，如何实现“宽松”也存在很大的挑战。下表展示了典型 HPC 应用的 POSIX 语义要求，越上面的，一致性越高，性能损耗也越大。

低效的数据管理，由于 BB 的容量要远远小于后端的 FS，其通常作为计算和 FS 间的非持久性加速缓存发挥作用，因此 BB 和 FS 间高效便捷的数据迁移就变得非常的重要，它们间的数据迁移通分为两种：透明迁移和非透明迁移，透明迁移是在软件层面实现自动的数据迁移，很方便但可能带来大量的无效数据迁移；非透明迁移，通常需要计算节点参与，过程中可能因为计算的空闲而导致资源浪费。两种方式都支持 FS 到 BB 的预加载，实现预读功能。但是两种方式都不支持计算任务运行时的数据迁移管理。

2.2.2 HadaFS 设计

HadaFS 包括客户端，服务端，数据管理工具三个主要子系统。HadaFS 属于共享 BB，服务端运行在部署有 NVMe SSD 的专有节点上，提供全局的数据及元数据服务，为每个客户端提供全局试图。HadaFS 实现了宽松 POSIX 语义，不支持 mv，rename 和 link 操作。客户端以静态或动态库的方式部署在计算节点上，负责将对应用的 POSIX I/O 进行解析及转发给服务端处理。数据管理工具 hadash，运行在管理节点上，用于数据迁移。

HadaFS 中的每个文件与两类服务发生联系，其中文件数据存储在数据服务器上 NVMe 介质的本地文件系统上，文件元数据存储在元数据服务器的 RocksDB 上。

2.2.2.1 本地优先架构（LTA）

文件系统的挂载通常有两种方式，第一种是内核态挂载，这种方式要求实现完整的 POSIX 语义，同时会带来 I/O 在内核态和用户态上下文切换的开销，一个挂载点可以给服务器节点上的所有应用共享；另一种是，应用直接在用户态挂载文件系统，这种方式可以规避内核态文件系统的限制，也能减少内核态和用户态切换的开销，但是太多的链接于系统的扩展性不利，也会带来稳定性风险。

HadaFS 采用应用直接在用户态挂载文件系统方式为应用提供文件访问服务，为避免传统应用态挂载带来的弊端，HadaFS 将一个客户端的所有链接到绑定到一个固定的服务节点（称为桥节点），该桥节点负责处理该客户端的所有 I/O 请求，每个用户文件对应桥节点本地 NVMe SSD 上的一个 Ext4 文件。如果客户端需要访问其他服务节点上的数据，需要通过桥节点转发（这意味着服务节点间是全互联结构）。

2.2.2.2 命名空间及元数据管理

HadaFS 舍弃了传统的目录树结构，采用全路径来索引文件。一个 HadaFS 文件，其数据内容存储在创建该文件的桥节点上，元数据存储在根据文件路径 hash 指向的 K-V 数据库中。HadaFS 的客户端通过全路径/路径前缀来查找文件，不需要执行逐级的目录搜索。

HadaFS 的文件元数据与 Linux 文件系统兼容，为了取得更好的扩展性和性能，HadaFS 将元数据分层 4 个类别：

第一类：创建文件时生成，包括：name，owner，mode 等；

第二类：访问文件时生成，包括：大小，修改时间，范围时间等；

第三类：这类信息，HadaFS 实际上不需要，

包括：ino， stdev 等；

第四类：按照 offset 排序的文件分片位置的有序列表，包括：服务器地址，offset，size，写入时间等。

每个 HadaFS 服务节点上包含两套元数据数据库，均基于 rocksdb 实现， 如下图。本地元数据库（LMDB）存储本地文件的第一类和第四类元数据（文件的本地标识 LID 相当于文件的本地路径），全局元数据库（GMDB）存储文件的第一类，第二类和第四类元数据，每个 HadaFS 文件的元数据，根据文件路径的 hash 只会存储到一个全局数据库。两个元数据库的键均是<UID, GID, PATH>组合的 hash。UID，GID 可以用来可控前缀查找的访问，提升性能。在 N-N 的 I/O 场景下，每个客户端写独立的文件，存储在 LMDB 上的元数据与存储在 GMDB 上的第一类和第四类元数据一致，在 N-1 的 I/O 场景下，多个客户端共享访问相同文件，可能会使用不同的桥节点，GMDB 负责合并多个 LMDB 上的元数据。

在文件读写过程中，LMDB 负责记录/维护一个根据 offset 排序的文件分片位置列表，同时会将该元数据发送给文件所属的 GMDB。GMDB 负责维护文件的分片位置列表，以支持各服务节点间的全局共享。

2.2.2.3 一致性及元数据优化

HadaFS 采用宽松 POSIX 语义，不提供客户端和服务端的缓存，一致性依赖于元数据的同步。HadaFS 提供了三种元数据一致性语义最终一致性，文件元数据先在本地 LMDB 更新，然后异步复制到 GMDS 中，可以提供最高的性能。适用于没有数据共享的场景会话一致性，第一类元数据采用同步方式复制，第二类元数据采用异步方式复制（弱）强一致性，所有元数据同步复制到 GMDS，因为不支持覆盖写一致性比强一致要弱些

由于 HadaFS 并不支持分布式事务，而 HadaFS 要求数据需要先写到本地桥节点，在 N-1 的共享场景中，覆盖写也就无法支持，原子写也只在上述第三种语义才被支持。宽松 POSIX 语义，要求用户对应用的共享模式非常的清楚

2.2.2.4 文件共享优化

本地优先架构（LTA）对于 N-N 的场景非常的友好，对于 N-I 的共享场景，HadaFS 采用后备文件的机制，每个客户端将数据写到独立的文件中，然后通过 GMDB 来管理各文件的元数据，并完成最终的元数据合并。这样可以将对共享文件的读写，转化为并发的读写，提升整体的性能。在没有 I/O 重叠的情况下，能够取得很好的效果。另外，HadaFS 通过有序链表管理文件分片，读写过程中的分片管理时间复杂度为 O(logN)

2.3 InfiniFS

2.3.1 InfiniFS 背景

InfiniFS 针对的是如何实现超大规模的单一分布式文件系统，目标上有些类似于 Facebook 的 Tectonic。但 InfiniFS 仍然是比较正统的、遵守 POSIX 语义的 fs，而 Tectonic 则是 HDFS 的升级版，目的是解决 Facebook 自身业务遇到的实际问题。InfiniFS 看上去是 LocoFS 的后继，延续了 LocoFS 将 metadata 分成两部分的设计（但针对所有 inode 而不只是 f-inode）。另外 InfiniFS 还从 HopsFS 借鉴了并发 load inode。除此之外 InfiniFS 还有如下独特设计：client 端可以通过 hash 预测 inode id。结合并发 load inode，可以有效降低 network trip。Client 与 metadata server 共同维护的一致性 cache。单独的 rename coordinator。

超大文件系统有如下挑战：data partitioning 难以兼顾良好的局部性和负载均衡性。相近的文件/目录经常被集中访问，造成热点。如果 hash partition，则分散了热点，但降低了局部性，同样的操作（如 path resolution/list dir）涉及更多节点。如果以子树为单位 partition，则与之相反。路径深度变大，path resolution 延时增加。Trivial 的按 dir id partition（Tectonic 再次出镜） 会导致长度为 N 的路径查找需要经历 N-1 个 trip。client 端 cache 的一致性维护负担加大。lease 机制（如 LocoFS）会导致越接近 root 的节点被 renew 的频率越高，无形中制造了热点。在 metadata server 本身分布式之后，lease 维护代价也会变高。但超大文件系统有如下的好处，全局 namespace 允许全局数据共享。资源利用率高，避免了跨系统操作，降低了复杂度。

2.3.2 InfiniFS 设计

Clients 支持 InfiniFS Library 和 Fuse 方式，利用 Speculative Path Resolution（可预测路径解析）和 Optimistic Metadata Cache（乐观型元数据缓存）

特性，解决元数据访问时延和 near-root 热点问题；

Metadata Servers 采用 Access-Content Decoupled Partitioning 机制（访问-内容元数据解耦分区机制），每个 Metadata Servers 负责一个目录分区，解决元数据访问负载均衡和局部性问题。Metadata Server 采用 KV-Store 保存元数据，内部维护一个 invalidation list(Inv.List)，用于 lazily validate 元数据缓存；

Rename Coordinator 处理目录 rename 和 set_permission 操作，并广播给 Metadata Servers 的 Inv.List。采用 Renaming Graph 处理并发目录 rename，避免出现孤儿目录；

2.3.2.1 目录分区

细粒度 or 粗粒度分区策略失效，根本原因在于，把目录元数据当成一个整体；事实上，目录元数据包含相互独立的两部分：因此需要解耦元数据

对目录 access 和 content 元数据单独分组分区，可以实现目录元数据访问负载均衡和局部性。

分析各类元数据操作，并归为三大类：单元数据对象操作，只涉及 target 目录/文件本身，如：open/close/stat；双元数据对象操作，涉及 target 目录/文件及其 parent，如：create/delete/readdir；多元数据对象操作，如：目录 rename；

因此，将目录的 Content 元数据和 Subdir 的 Access 元数据、files 元数据划分为一组，按照目录将整个目录树分割成相互独立的 locality-aware 元数据组，实现元数据访问的局部性（访问关联）；

细粒度分区策略，基于 locality-aware 元数据组的目录 ID，采用一致性哈希算法，实现元数据操作的负载均衡，同时保证元数据服务扩展时最小化数据迁移；

采用 KV store 作为元数据服务后端存储，分为三类 KV pairs：目录 Access 元数据和 Content 元数据、文件元数据；

2.3.2.2 可预测的目录解析

目录 create，通过对 birth triple <birth parent ID，name，name version>哈希计算，生成目录 ID；目录 rename，只需修改目录的 Access 元数据的 key <pid，name>，目录的 Content 元数据和目录 ID 保持不变，这样该目录的后裔目录/文件元数据不受影响；

Birth parent 目录维护 rename-list(RL，<name，version>)，记录该目录下的 born 子目录，但已经 rename 到其他地方，同时用于决定 name version。RL 与目录 Content 元数据保存在同个 MetaServer；Renamed 目录维护 back-pointer(BP，<birth parent ID，name version>)，指向其 birth parent 目录。BP 作为目录 Access 元数据保存；目录二次 rename，也只需要变更目录 Access 元数据的 key，目录 Content 元数据、BP 及 birth patent 的 RL 都不需要改变；目录 delete，删除 birth parent 时，只删除其 Access 和 Content 元数据，保留 RL。删除 renamed 目录时，利用 BP 清理 RL 中记录；

通过对 birth triple<pid，name，version>哈希计算目录 ID，其唯一性保证如下：文件系统语义，保证同个目录下 name 的唯一性；rename 场景，通过 birth parent 目录的 RL 保证 name version 唯一性；采用 cryptographic hash 低冲突算法(e.g.，SHA-256)，若发生冲突，在插入目录 Content 元数据时检测，并通过增加 version、RL、BP 办法解决；

基于可预测目录 ID，client 按照以下步骤执行并行目录解析：预测目录 ID，从 root 开始，以 version 为 0 生成所有中间目录项的 birth triple，并哈希计算其目录 ID，最终得到每个中间目录项的元数据的 Key；并行 lookup，client 并行发送所有中间目录项的 lookup 请求，server 检查目录的 permission 和预测 ID。若预测 ID 不正确，则终止 lookup，并返回正确的目录 ID；

2.3.2.3 乐观元数据缓存

Client 执行文件操作请求时，始终认为自身缓存有效，不会主动请求缓存更新同步，由 metadata servers 处理文件操作请求时验证缓存是否有效。Client-side 只缓存目录 Access Metadata，i.e.，目录名称、ID、permission，消除 near-root 目录解析请求；按照目录树层级结构组织元数据缓存，并通过 LRU 策略管理所有的 leaf entries（目录，非文件），尽量保证 near-root 目录元数据常驻缓存；

Client-side 缓存 lazy invalidation 策略。目录 rename 和 set_permission 操作会导致缓存失效，由于 clients 数量巨大，且 clients 间关系无法维护，不适合在每个 rename 和 set_permission 操作之后，同步进行所有 clients 的缓存更新操作；

lazy invalidation 策略，目录元数据修改后，

在所有 metadata servers 间广播，延迟到 client 下次执行元数据访问请求时，server 再验证目录元数据的有效性；

2.4 CFS

2.4.1 CFS 背景

文件系统的定义是一种采用树形结构存储和组织计算机数据的方法，这个树形结构通常被称为层级命名空间（Hierarchical Namespace）。如下图所示，这种命名空间的特点是整个结构像一棵倒挂的树，非叶子结点只能是目录。如果不考虑软链接跟随（follow symlinks）、硬链接（hardlink）的情况，从根节点（/）出发，每一个目录项（目录、文件、 软链接等合法类型）都可以由一条唯一的路径到达。

文件系统可以分为元数据（Metadata）和数据（Data）两个部分。数据部分指的是一个文件具体存储了哪些内容，元数据部分是指层级命名空间树形结构本身。例如，如果我们要读取 /a/b/file 这个文件的数据，元数据部分负责找到 /a/b/file 这个文件存储在哪儿，数据部分则负责把文件的内容读出来。文件系统主要有两种实现风格。

POSIX：全称 Portable Operating System Interface，是 IEEE 制定的 UNIX 可移植操作系统兼容标准。该标准定义了一个文件系统相关的接口子集，是文件系统领域最基础和权威的标准。POSIX 兼容文件系统就是指兼容该标准的文件系统；

HDFS：源自 Hadoop 大数据生态，对 POSIX 标准做了一些比较实用的简化和修改，主要是放弃了对 hardlink、随机写的支持，并增加了一些实用的递归操作。通常也将其归类为 POSIX-like 文件系统，意思是和 POSIX 相似。

在探索元数据问题的过程中，我们逐步建立了文件系统元数据问题的抽象，在此处先进行介绍，以方便展开后续的内容。 一个文件系统的元数据系统只要正确实现了这个抽象，就解决了 POSIX 兼容性问题的核心部分。

一方面，关系到元数据的一致性。例如，删除目录的时候需要判断目录是否非空，这在很多系统里会维护一个字段来记录，如果这个字段的信息滞后了，就有误判的可能性。当一个非空目录被误判为空目录，目录本身会被删除，目录下的子项残留了下来。这些残留的子项是所谓的孤儿节点，无法通过路径查找访问到；

另外一方面，关系到缓存的正确性和效率。为了优化文件系统元数据的性能，客户端通常会缓存 lookup 和 readdir 操作的结果，然后再次访问的时候可以通过获取父目录的属性信息，快速判断父目录是否被修改，从而确认缓存是否有效。如果没有这个机制，操作要么无条件重新执行保证数据正确，要么容忍可能存在的信息滞后。这个缓存机制对大目录的 readdir 效果尤其重要，可以节省重复执行大目录的 readdir 带来的大量 I/O 开销。

对于数据服务，由于不同文件的数据相互独立，同一文件的不同部分也很容易条带化和分块处理，天然具备并行处理的条件，因此较容易扩展到很大的规模。但对于元数据服务，层级命名空间的目录结构带来了很强的父子依赖关系，并行处理并不是件容易的事情。

2.4.2 CFS 设计

2.4.2.1 整体架构

整个系统架构上分成四个部分:

Namespace 存储层（TafDB）：TafDB 负责存储层级命名空间里除文件属性外的其它部分；

文件存储层（FileStore）：这一层是一个以平坦方式组织的块存储层，块是文件数据的基本单位。文件属性和文件数据一起存储在该系统里。对于文件属性而言，这是一个不支持范围读、只支持点读的 KV 系统；

Rename 服务（Renamer）：Multi-Raft 架构，每个文件系统由一个 Raft 复制组提供对复杂 rename，即所谓 Normal Path rename 的支持；

客户端库（ClientLib）：客户端负责接收具体的请求，拆解成上述模块的内部请求。ClientLib 提供必要的接口，以方便接入 Linux VFS 层，目前支持 FUSE、Samba、NFS-Ganesha。

2.4.2.2 元数据组织和分片

和 InfiniFS 类似，CFS 将每一个目录项的记录拆解成两部分，分别是 inode id record 和 attributes record。这个分解是进一步缩减冲突域的基础。TafDB 使用一张 inode_table 表来存储所有的数据，表的主键为 <kID, kStr>。这张表里实际上存储了混合存储的两类数据，包括所有的 inode id record 和目录的 attributes record。FileStore 负责存储文件的 attributes record。

对于 inode id record，<kID, kStr> 中的 kID 部分代表父目录的 inode，kStr 代表子项的名字。这种记录是为了满足 lookup 和 readdir 的需求，除了 inode 和 type，其它字段都是无效的。

对于目录的 attributes record，kID 就是目录自己的 inode，kStr 是保留字符串 /_ATTR。其它字段存放的是各个属性字段，inode 字段在这里留空没有实际作用。

inode_table 整体上按 <kID, kStr> 有序存储所有数据，分片规则是按照 kID 来的。TafDB 做了一个特殊的保证，即分片无论如何分裂和合并，同一个 kID 的数据始终存储在同一个分片上。这意味着同一个目录的关联变更涉及到的数据都在一个分片进行处理，CFS 实现了目录级别的 range partition。所有分片的分裂和合并操作都是 TafDB 自动进行的，不需要 CFS 关注和人工干预。

2.4.2.3 缩减分布式锁开销

将元数据分散到 TafDB 和 FileStore 两个组件存储之后，首先要保证的是两个系统的对外一致性。我们通过精心排列的执行顺序来解决这个问题：

对于所有的创建操作，先创建 FileStore file attribuets record，后创建 TafDB inode id record；

对于所有的删除操作，先删除 TafDB inode id record，再删除 FileStore file attribuets record；

文件系统在对文件操作之前都会经历从特定目录开始 lookup 的过程，只有 inode id record 存在对应的文件才会被看到，因此，只需要保证 file attributes record 比 inode id record 更早产生更晚消亡，就可以保证无效数据不会被用户看到，从外部视角观察，一致性没有被打破。图示给出的操作顺序可以达到这个效果，唯一的副作用是操作失败可能残留垃圾，这可以通过垃圾回收来解决。

2.4.2.4 缩减单分片锁

关联变更涉及到多条记录，在执行过程中还存在 read-and-modify 模式，特别的，对父目录属性的修改就需要先读出旧值再更新。一个朴素的实现方式会包含多轮条件检查、读和写操作，效率肯定是不高的。为了解决这个问题，我们提出了单分片原语（single-shard atomic primitive）的概念。

每一种原语实现了一个定制的单分片事务，这个事务原子地完成所需要的所有读、写和条件检查，保证执行过程是 all-or-nothing 的，只有全部条件检查为真，才会执行成功，否则不会对分片数据做任何修改。原语作为一种特殊的单分片事务，高度优化后的执行代价和写一条 WAL 日志相当。

如果沿用标准的实现，上述的原语仍然在父目录属性更新的时候导致排队。前文我们分析过，这里冲突的本质其实是对属性的更新操作，本质上是一些原子变量操作。根据这一点，我们进一步实现了两个增强机制用于弱化及合并处理事务冲突。

第一个机制是 delta apply，针对 links、children、size 这些数字属性。它们在变更的时候会加减增量，和原子变量加减一样，顺序并不重要，只要保证最终的叠加结果是对的。delta apply 就实现了这种合并加减效果。

第二个机制是 last-writer-win，针对权限、mtime、ctime 这些单纯的覆盖操作，我们简单的保留最后一个赋值操作的结果。

原语通过检查 SET 语句涉及的是加减运算（如 children+=1）还是赋值运算（如 mtime=@now）就可以自动决定运用 delta apply 还是 last-writer-win。这个检测完全不需要感知文件系统的语义，具有通用性，实际上可以推广到任何只是部分字段原子变量操作的场景，实现比写写冲突更小范围的冲突。

2.4.2.5 移除元数据代理层

将 Normal Path rename 之外的所有操作均拆解成了原语的组合之后，这些操作之间只有落到单个分片上时才可能会产生冲突，分离式架构里的元数据代理层存在的唯一价值变成了接口翻译，串联起各个环节来实现 POSIX 接口，但这一点完全可以由客户端来承担，因此我们做了一个大胆的决定，将这一层功能完全集成到客户端来实现。

2.5 TectonicFS

2.5.1 TectonicFS 背景

不同业务类型需求不一致，有的是 IOPS 敏感型，有的是吞吐敏感型，当前 facebook 不同的业务类型使用独立的集群导致不能同时充分利用集

群的吞吐和 IOPS 资源，造成资源浪费。

对于一些数据仓库业务，通常其容量需求巨大，单个集群无法满足，所以一个数据仓库的数据需要存储在多个小集群中，这种方式使得数据处理复杂度提升了。

因此，facebook 这次推出的 tectonic 存储系统有两个特点，一个是同一集群可以提供不同业务需求的服务，另一个是单个集群能够达到 EB 级别的扩展能力。

那么同一集群及提供 IOPS 敏感类型的服务也提供吞吐敏感类型的服务，那么就需要有能力提供性能隔离，减少一种业务类型对另一种业务类型的影响。另外，对于一个集群提供 EB 级别的存储能力，在小文件场景下，元数据量会很庞大，需要有能力保证随着存储数据量增加带来的元数据量膨胀，需要元数据存储具备同样的强悍的扩展能力。

2.5.2 TectonicFS 设计

与大多数分布式存储系统的架构一样，tectonic 也是 client、metaserver、chunksever 这样的组成结构，同时由于 tectonic 对外提供的是 appendonly 的读写接口，所以在底层不可避免的需要引入 GC 等后台任务处理角色。在这些角色中只有 metaserver 和 chunkserver 是有状态的，其他都是无状态服务。

2.5.2.1 元数据存储

tectonic 的元数据存储在名为 ZippyDB 的 KV 存储引擎中，这个 ZippyDB 内部是封装了 RocksDB。

为了提供简单的且更容易扩展的元数据存储方式，tectonic 采用细粒度的元数据切分方式，通过对元数据进行 hash partition 的方式在多个元数据存储节点之间打散元数据，这种方式同时也会避免目录热点带来的热点访问问题。

元数据根据 hash partition 策略切分成一个一个的 shard，然后这些 shard 会有一个高可用复制组，通过 Paxos 算法在复制组内复制元数据，保障数据可靠性。元数据 hash partition 切分成 shard；上层高可用，shard 之间高可用组，通过 paxos 算法复制元数据，保持一致性；

2.5.2.2 元数据优化

海量小文件的存储一直是一个比较有挑战的问题，海量小文件意味着庞大的元数据，在对读写延时有要求的场景中，元数据操作效率很低影响 IO 质量。

tectonic 其实做法也是常规做法，将多个小文件组合成大文件，记录小文件的 offset 映射关系。

seal 的 file/block/directory 由于不会再被改变，所以可以 cache 在存储节点，提高访问效率的同时也不违背一致性。

与 HDFS 类似，tectonic 也只提供 single writer 的写入模式，这样极大的简化了元数据一致性访问的设计。

存储优化。果目录 d1 包含了两个文件 foo 和 bar，那么 tectonic 在存储的时候存储了两个带有 d1 前缀的 key 在 Name 元数据 shard 中，这种方式好处在于如果需要修改其中的一个文件元数据信息，不需要读取整个目录，这种在海量元数据场景中还是很有帮助的。

2.5.2.3 资源平衡

Tectonic 将资源划分为： non-ephemeral and ephemeral。可以理解为可重复使用的和不可重复使用的，对于存储资源，一旦分配给一个租户就不能再分配给其他租户了，那么他就是不可重复使用的，是一个长期占用的资源。但是对于 IOPS 这样的资源，租户只会在发请求的时候使用到，并且这个请求结束后 IOPS 资源就被释放出来了，他是一个暂时的占用。

这个也是为什么大多数云厂商通常都是 IOPS 超售，存储资源不超售的原因。

因此，可以看出我们能够做文章的也就是这些 non-ephemeral 的资源。这里的挑战在于，首先我们既要满足每个用户的最基本的存储需求，另外还需要给不同类型的用户提供符合其业务类型需求的优化。

3 总结

在本文中，以元数据为分布式文件系统的切入点，介绍了现有的分布式文件系统的工作流程和储存结构。通过元数据的合理分配与存储来大幅提升文件系统的工作效率，提升了资源利用率。

参 考 文 献

[1] 如何将千亿文件放进一个文件系统，EuroSys23 CFS 论文背后的故事，https://blog.csdn.net/lihui49/article/details/130942836?ops_request_misc=&request_id=&biz_id=102&utm_term=CFS%20Scaling%20Metadata&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-0-130942836.142^v99^pc_search_result_base6&spm=1018.2226.3001.4187

[2] 近年几篇文件存储相关顶会论文，https://zhuanlan.zhihu.com/p/661249410

[3] 论文阅读 - InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems，https://zhuanlan.zhihu.com/p/652300760

[4] ebook’s Tectonic Filesystem: Efficiency from Exascale论文笔记，https://blog.csdn.net/weixin_43778179/article/details/134310110?ops_request_misc=%257B%2522request%255Fid%2522%253A%252217036903321680021506444 4%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request_id=170369033216800215064444&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-1-134310110-null-null.142^v99^pc_search_result_base6&utm_term=Facebook%E2%80%99s%20Tectonic%20Filesystem%3A%20Efficiency%20from%20Exascale&spm=1018.2226.3001.4187

[5] Wenhao Lv and Youyou Lu,Yiming Zhang,Peile Duan,Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems.In the Proceedings of the 20th USENIX Conference on File and Storage Technologies February 22–24, 2022 • Santa Clara, CA, USA

[6] Yiduo Wang†§ Yufei Wu† Cheng Li†ζ Pengfei Zheng§ Biao Cao§Yan Sun§ Fei Zhou§ Yinlong Xu†ζ Yao Wang§ Guangjun Xie§. CFS: Scaling Metadata Service for Distributed FileSystem via Pruned Scope of Critical Sections

[7] Xiaobin He, Bin Yang, Jie Gao ,Wei Xiao, Qi Chen, Shupeng Shi ,Dexun Chen, Weiguo Liu, Wei Xue. HadaFS: A File System Bridging the Local and Shared Burst Buffer for Exascale Supercomputers.In the Proceedings of the 21st USENIX Conference on File and Storage Technologies.February 21–23, 2023 • Santa Clara, CA, USA

附录 论文汇报记录

1. 如何保持文件一致性？
2. Posix语义需要确保一致性，那么在该文件系统中是如何实现的？

这两个问题均与文件系统的一致性相关，所以一起作答描述：

① 无日志数据操作，在节点崩溃时用于恢复，消除了节点崩溃时的文件一致性问题。关键思想是通过利用键值存储后端的单对象更新原子性以及父 inode 和子 inode 的元数据语义依赖性来识别可能的元数据不一致性。

② 设计了分层并发的思想，以一种更细粒度的方式同步inode操作。消除了多个客户端对同一个inode进行操作时的一致性问题。具体来说，inode写入器使用每个inode的读写锁来与其他操作进行同步，inode时间戳更新器和读取器使用无锁协议在它们之间进行额外的同步。