

摘 要

对象存储系统采用了一种新的接口——对象接口，有效综合了块接口的快速直接访问、存储设备可扩展的交换结构与文件接口的安全性、跨平台数据共享等优点，同时对象接口能够提供比其他任何一种接口更为丰富的语义，其基本单位是对象，对象除了包含用户数据外，还包含能描述对象特征的访问属性。

在大规模对象存储系统中，元数据访问非常频繁，是系统性能潜在的瓶颈，需要研究高性能、可扩展的元数据管理方法。在对象存储系统中，数据放置策略负责将文件映射为对象、为对象选择合适的对象存储设备存放，在需要访问对象时要快速定位到对象所在的对象存储设备，它对系统访问性能有关键性的影响，需要根据系统的规模选择合适的数据放置策略。另一方面，元数据服务器中的元数据记录了文件和目录信息与对象之间的关系，元数据的丢失将导致数据无法访问，因此元数据的可靠性维护至关重要。

提出一种分布式元数据管理方案，以提供高性能和可扩展的元数据访问。它采用仿层次目录结构，针对元数据的不同访问特性将元数据灵活分布在元数据服务器集群中。在深入分析传统文件系统中与用户组件部分相关的元数据组成结构的基础上，结合数据库能提供高事务吞吐量的特点，提出一种改进的元数据存储和管理方法，提高访问速度；在系统中不再用持久存储(如磁盘)来存储记录文件名到索引节点号映射关系的目录数据，而是采用一种间接的方案来模拟层次目录结构，避免层次目录结构自身成为热点，从而提供高性能、可扩展的元数据访问；引入目录转换元数据以避免子树分割方案中的目录遍历和哈希方案中的重命名目录导致的大量元数据迁移，提高元数据总体访问性能；针对每种元数据自身的访问特性，采用不同的分割方法将其分布在元数据服务器集群中，方便系统规模扩展。实验结果表明该方案在提高元数据访问性能和系统可扩展性方面有明显优势。

由于元数据访问负载随时间动态变化，元数据在元数据服务器集群中的静态分配可能会导致某一时刻某个元数据服务器成为访问瓶颈，为了在元数据服务器集群中提供高性能、可扩展的元数据服务，需要在元数据服务器之间均衡负载。提出一种以文件元数据请求的响应时间为衡量标准、应用于元数据服务器集群的负载均衡算法，使集群中所有元数据服务器的响应时间差别较小，达到元数据服务器集群的负载均衡，从而提高元数据服务器整体性能。

提出一种利用遗传算法根据文件的不同特性求解数据放置的策略，它用于在系统规模较小、对象存储设备总数固定的应用环境中寻求系统性能的近似最优解。提

出基于组的区分定位策略，它用于在系统规模较大且对象存储设备总数可能发生变化的应用环境中解决对象放置问题。它首先根据对象存储设备加入系统的不同时期将每个对象存储设备划分到不同的存储子集群，先采用分布式算法将对象映射到系统的某个子集群中，再在子集群内部根据不同类型的对象采用不同的映射方法，对新创建的大对象采用启发式方法来选择负载较轻的对象存储设备存放，对小对象采用改进哈希算法来决定其所在的对象存储设备，兼顾了对象分布的灵活性和系统可扩展性。实验结果表明该策略具有很好的性能和可扩展性。其中改进哈希算法是基于子集群内对象存储设备规模的变化规律提出的一种新的分布式算法，它既能继承简单哈希算法的计算开销小和均匀分配对象的优点，又能以近似最优的对象迁移开销有效支持子集群内的对象存储设备规模的变化。

提出一种采用扩展属性页来提高元数据可靠性的方法，它利用对象存储系统富有表达力的对象接口来提高系统元数据可靠性，并采用 Markov 模型对其可靠性进行分析。该方法不需要额外的硬件配置，且不排除其他的提高存储系统元数据可靠性的方法，为提供更高的元数据可靠性提供了一种补充方案。

关键词：对象存储系统，元数据，元数据服务器集群，负载均衡，放置策略，可靠性

Abstract

Object-Based Storage Systems (OBSS) adopt a new interface—object interface, which is an effective integration of fast direct access and scalability in block-based interfaces and high security and cross-platform data sharing in file-based interfaces. Object-based interfaces can provide more rich semantics than other interfaces. The basic access unit in an OBSS is an object. An object contains not only user data but also attributes that describes access characteristics of user data.

In a large-scale OBSS, metadata accesses are very frequent, and MetaData Server (MDS) is the potential performance bottleneck. Thus, we need to study the issue of high-performance and scalable metadata management. In an OBSS, the data placement strategy not only determines how a file is mapped into one or multiple objects but also assigns an appropriate Object-based Storage Device (OSD) for each newly created object. Meanwhile, it is also used to locate the right OSD storing accessed objects. Additionally, it has a critical impact on the system performance, and a reasonable data placement strategy is desired in terms of the system scale. Metadata in MDS record the corresponding relationship of files (or directories) and objects, and the loss of metadata will cause data not to be able to be accessed. Therefore, the reliability of metadata is very critical.

A novel distributed metadata management strategy is proposed to provide high performance and scalable metadata service through four techniques, including directory conversion metadata, mimic hierarchical directory structure, flexible partition methods targeting metadata of diverse characteristics, and the application of database to metadata backend. Firstly, on the basis of in-depth analysis of the metadata composition structure of the user component in conventional file systems, an improved method for metadata storage and management is proposed, which improves the metadata access speed due to high transaction throughput of database. Secondly, each metadata is stored as a record in the database, and directory data storing the mapping relationship of file names and inode numbers are not stored in any persistent storage (such as disk), and an indirect scheme is employed to mimic the hierarchical directory structure, which can avoid the structure itself becoming a hot-spot and thus it can provide high performance and scalable metadata service. Thirdly, directory conversion metadata are adopted to avoid the directories traversal in subtree partition shemes and file metadata migrations incurred by

renaming a directory in HASH schemes. Consequently, it can improve overall metadata access performance. Lastly, based on different characteristics of each kind of metadata, different partition methods are adopted to facilitate the expansion of the system scale. The experimental results show that the strategy can significantly improve the metadata access performance and the scalability of the system.

As the metadata workload changes over time, a static distribution of metadata workloads in MDS cluster may cause a MDS to become the system performance bottleneck at a certain time. Load balancing is implemented in the MDS cluster to provide high performance and scalable metadata service. A load balancing algorithm is introduced to solve the load imbalance in the MDS cluster. It adopts metadata request response time as the metric, and attempts reducing the difference of metadata request response times among all MDSs to achieve load balancing in MDS cluster.

A data placement strategy is proposed for small, determinate scale storage systems, which utilizes the genetic algorithm to solve the placement problem based on different file characteristics, and it endeavors to seek the approximate optimal solution. A group-based differentiated location strategy is proposed for large, volatile scale storage systems. It firstly arranges each OSD into a storage sub-cluster according to the period of the OSD joining the system, and a distributed algorithm is adopted to map objects into different sub-clusters. Different objects are placed with different methods in a sub-cluster. A heuristic method is employed to select a lighter load OSD to place a large object, and the advanced hash algorithm is adopted to place a small object. The strategy takes into account not only the flexibility of the object distribution but also the system scalability, and the experimental results show that the strategy has an outstanding performance and scalability. The advanced hash algorithm is a new distributed algorithm, and it is proposed based on the variation trend of the OSD scale in the sub-cluster. It not only inherits the merits of a simple hash algorithm that the computation overhead is small and objects can be uniformly distributed among OSDs, but also ensures that a small amount of objects migration overhead is induced to maintain the correctness of the algorithm when the number of OSDs changes.

A novel scheme is presented to enhance the metadata reliability of the system by adding an attributes page for each user object, which makes full use of the expressive object interface. Leveraging a Markov chain, the metadata reliability of the method is analyzed. The scheme requires no additional hardware equipments, and it also does not

exclude other schemes of improving the metadata reliability of storage systems. Therefore, it is a good supplement for achieving higher metadata reliability.

Key words: Object-based storage system, Metadata, Metadata server cluster, Load balancing, Placement strategy, Reliability

目 录

摘要.....	I
ABSTRACT.....	III
1 绪论	
1.1 元数据的重要作用	(1)
1.2 存储系统与元数据	(3)
1.3 相关的研究工作	(8)
1.4 本文研究的主要内容	(17)
1.5 论文组织.....	(19)
2 对象存储系统中元数据管理框架	
2.1 对象存储系统体系结构	(21)
2.2 对象存储系统中元数据描述	(25)
2.3 对象存储系统中元数据管理框架	(29)
2.4 本章小结.....	(32)
3 分布式元数据管理	
3.1 元数据存储.....	(34)
3.2 元数据分布.....	(39)
3.3 元数据负载均衡	(42)
3.4 访问流程.....	(48)
3.5 元数据一致性.....	(49)
3.6 性能评估.....	(53)
3.7 本章小结.....	(59)
4 数据放置策略研究	

4.1 采用遗传算法求解混合分布的静态放置策略	(61)
4.2 动态放置策略的选择	(67)
4.3 改进哈希算法.....	(70)
4.4 基于组的区分定位策略	(77)
4.5 基于组的区分定位策略性能评估	(86)
4.6 本章小结.....	(90)
5 元数据可靠性研究	
5.1 可靠性设计.....	(92)
5.2 马尔可夫可靠性模型分析	(95)
5.3 可靠性性能仿真	(98)
5.4 本章小结.....	(101)
6 全文总结	
致谢.....	(104)
参考文献.....	(105)
附录 1 攻读学位期间发表的学术论文目录	(114)
附录 2 攻读学位期间申请的专利与软件著作权	(115)

1 绪论

随着数字电视、数字照相机、全球定位系统、监控系统、数字油田、医学影像、科学计算等应用的不断发展，数据信息总量呈现出爆炸性增长的趋势，存储系统的存储容量需求越来越大。IDC（Internet Data Center）研究表明数字世界的存储容量需求在 2007 年已达到 281EB，2011 年时其一年内的数字信息总量就会达到 1800EB [1]。

在存储容量需求增长的同时，对存储系统的性能也提出了新的要求。高性能科学计算、生物医学等，比如核强子对撞等高能物理研究、地震数据分析、基因工程、医学影像等应用，都需要高聚合 IO 吞吐量、大存储容量的网络存储系统作为存储后台。随着数字化时代的到来，大部分传统业务开始数字化、网络化，大规模应用系统的广泛部署，对存储系统的性能和服务质量等指标提出了更高的要求，主要表现在高性能、高可靠/可用、高安全、可扩展性、可共享性、可管理性、实时性以及智能性等方面，这些需求也体现出未来存储的发展方向^[2]。

1.1 元数据的重要作用

在存储系统中，元数据是用于描述文件系统组织结构的数据，在访问文件数据之前必须先访问其元数据以获取要访问数据的描述信息和空间组织信息，然后才能根据这些信息访问到相关的数据。离开元数据的系统将是一盘散沙，无法对数据提供有效的检索和处理，导致数据无法向用户提供明确的含义。

“元数据”最本质的定义为：用于描述数据的数据(data about data)，它广泛应用于许多应用与研究领域。在图书馆与信息界，元数据是提供关于信息资源或数据的一种结构化的数据，是对信息资源的结构化的描述，它用于描述信息资源或数据本身的特征和属性，规定数字化信息的组织，具有定位、发现、证明、评估、选择等功能。在数据仓库领域中，元数据是描述数据及其环境的数据，它用于帮助用户使用数据，支持系统对数据的管理和维护。在软件构造领域，元数据在运行过程中起着以解释方式控制程序行为的作用。在存储领域，元数据是描述以及索引数据的数据，它用于帮助用户使用和管理数据，它能说明数据存储的物理位置、数据之间的依赖关系、数据的访问情况等。

本文关注的是存储系统的元数据。存储系统允许用户存储、检索和操作数据，为了实现这一目的，存储系统的文件系统需要保持一个内在的数据结构使得用户数据有组织并且便于访问，这一内在的数据结构被称为文件系统的元数据。元数据是

用来描述文件系统组织结构的数据，通常包括文件系统的超级块信息、目录数据(它与文件系统的名字空间相关，列出该目录下包含的目录项，即它包含的子文件或子目录的文件名与相应索引节点号的对应关系)、索引节点信息(包括该文件或目录的描述信息，如文件大小、访问权限等，以及数据块聚集信息，用于将逻辑块地址转换为真实的物理块地址，如直接块指针、间接块指针)和状态信息。上层应用通常并不与文件系统的元数据直接打交道，而是由存储系统的文件系统驱动程序执行相应的操作。

元数据在决定文件行为时起到了重要的作用，基于元数据的重要性，目前有很多研究者开展了相关的研究。

随着计算机和互联网技术的迅速发展，数据总量呈爆炸性增长，为了给用户提供高质量的服务，要提高用户的访问速度、减少访问延迟，可能需要将用户需要的数据预先存储到缓存中(即预取技术)。预取效率的提高取决于文件或数据块之间的相关度。有很多研究^[3]基于文件系统中提供的元数据信息和语义属性来挖掘文件或数据块之间的相关性以提高预取的准确性，进而提高系统访问性能。

随着数字技术的迅速发展，越来越多的信息被数字化，存储系统需要包括的文件总量迅速增加，文件的检索和管理变得异常困难，仅仅通过层次化目录结构来查找文件的效率不高，特别是当用户不知道文件保存位置的时候通常需要遍历整个目录或依赖于文件系统搜索功能来检索文件。在最近的一段时间里，新数据类型(如多媒体、电子邮件等)不断涌现，这些数据中包含了很多的元数据信息，学术界和工业界做了大量工作来研究如何利用丰富的元数据信息来提高文件的管理和搜索效率^{[4][5][6]}。如语义文件系统基于文件元数据对文件进行分类，提高用户对文件的搜索效率。

在当前的存储系统中，IO 性能依然慢于 CPU、内存和网络性能，成为系统访问性能瓶颈^[2]。为了提高存储系统的性能，要尽量减小同步磁盘请求数，如采用缓存等技术。文件元数据(如文件名、所属用户、所属组、模式等)与文件特性(文件大小、生命周期、访问模式、读写比率等)有很强的相关性^{[7][8]}，可以采用文件元数据(如文件名等)来预测文件的访问特性，预测的结果可以用来指导文件系统设计者改善文件在磁盘中的布局、提高缓存和预取效率^{[9][10][11]}。也有人根据对网络文件系统中的上述文件特性的观察，对网络文件系统的优化设计给出建议。例如，发现网络文件系统中文件读写混合的访问模式相比以前有增加趋势，则应该提高文件系统的随机访问性能，比如通过采用 flash 等中介介质的手段；发现文件读写比率相比以前有减少趋势，则要对写操作做优化，如在网络文件系统中采用 LFS^[12](log-structured file system)、WAFL^[13](Write Anywhere File Layout)或采用

NVRAM(Non-Volatile Random Access Memory)写缓存技术；重新打开文件操作不是很频繁，通常每个文件同时只被单个客户端访问，基于此有助于探索缓存策略；大多数被创建的文件不会被删除，少量的文件才会被多次访问，说明很多文件能迁移到更底层的存储备份设备中以在不影响性能的同时提高空间利用率和减少能量消耗；对于以写为主的文件采用 LFS 系统来减少写延迟，对于以读为主的文件可以采用复制技术来提高访问性能和可用性^[10]。

信息就是财富，信息以文件等形式存储在文件系统或数据库中，一旦丢失造成的损失可能是无法估计的，文件系统对文件信息的保护和管理起重要作用。在文件系统中，元数据指出了文件是由哪些数据如何组织在一起的，并且描述了它的特性。离开了元数据的数据是一盘散沙，当元数据不一致或不存在时，文件系统驱动程序则不能理解和操作元数据，散布的数据将不能再被用户访问。当系统非正常退出时可能会导致系统瘫痪，需要检查文件系统中元数据的数据结构的完整性和一致性，采取相应的技术^{[14][15][16]}来探测或修复大部分文件系统的常规错误以提高系统可靠性。

1.2 存储系统与元数据

文件系统主要由两大部分组成：文件系统用户组件部分和文件系统存储管理组件部分。文件系统用户组件部分包括如下的功能：目录层次管理、命名和用户访问控制等，它为用户提供全局唯一的树型逻辑结构视图(即树型目录结构和文件名列表)。文件系统存储管理组件部分负责存储和管理文件的物理视图，即将文件的逻辑块映射到底层物理设备。文件系统中用户组件部分包含的元数据主要有目录项和文件描述信息；文件系统存储管理组件部分包含的元数据主要有块聚集信息。

对文件系统来说，“文件”是指按一定的组织形式保存在存储介质上的信息，实际上包含两部分的信息：存储的数据本身和该文件的组织与管理信息，前者是文件的真实数据，后者就是该文件相应的元数据。当要访问一个文件时，需要至上而下逐层遍历该文件路径上涉及到的祖先目录的目录数据，在每层的目录数据中查找相应的目录项以得到期望祖先目录的索引节点号，最终得到要访问文件的索引节点号，根据索引节点号访问该文件的索引节点(inode)信息以得到该文件的元数据，从索引节点信息中获取该文件的数据块聚集信息，访问相应的数据块即可获取该文件包含的数据。在该流程中，涉及到的数据结构主要有四种：目录数据中的目录项(列出文件或目录的文件名对应的索引节点号)、块聚集信息(索引节点中包含该文件的数据块映射信息，即将逻辑块地址转换为真实的物理块地址，如直接块指针、间接块指针)、文件描述信息(除数据块聚集信息之外的索引节点信息、即用户可见的一

些信息，如修改时间，文件长度等)、用户数据；除用户数据外的数据结构均为元数据，在访问到用户数据之前必须要先访问到相应的元数据。其中目录项和文件描述信息属于文件系统中用户组件部分包含的元数据；块聚集信息属于文件系统存储管理组件部分包含的元数据。另外，对整个文件系统来说，还有一些重要的元数据，如超级块用来标识文件系统信息，并记录文件系统的其他信息，如系统的空闲块大小、已使用的空间大小等。

当前使用最多的存储结构主要有四种：直接依附存储（Direct Attached Storage, DAS）、网络附加存储（Network Attached Storage, NAS）、存储区域网（Storage Area Network, SAN）和基于对象的存储（Object-Based Storage, OBS），不同存储结构的文件系统中用户组件部分和存储管理组件部分可能位于不同的部件上，如它们可能位于主机端（即客户端）、存储网络组件、存储设备中，在以下给出相应的说明。

1.2.1 直接依附存储

在 DAS 中，主机通过外设总线（SCSI^{[17][18]}（Small Computer System Interface）、IDE（Integrated Device Electronics）、ATA（Advanced Technology Attachment）等）直接连接到一些私有的非共享块存储设备，它直接负责存储的管理与调度^[2]。在这种存储结构中，由主机集中管理和控制数据，具有一定程度的安全性，它主要用于小规模存储应用。由于 IO 总线限制了该存储结构的可扩展性，因此该存储结构不适用于大规模存储系统。DAS 存储构架如图 1.1 所示。

DAS 中采用块接口协议访问数据，DAS 中文件系统的用户组件部分在主机端实现，存储管理组件部分可能在主机或存储设备端实现。当主机带有逻辑卷管理器或软 RAID（Redundant Array of Independent Disk）功能时，块聚集在主机端实现，存储管理组件部分包含的块集聚元数据由主机提供；当主机不带有逻辑卷管理器时，块聚集在存储设备上（如磁盘阵列控制器）上实现，存储管理组件部分包含的块集聚元数据由存储设备提供。不论块聚集在哪个部件中实现，由于 DAS 中包含的主机和块存储设备个数受限，因此该存储结构的扩展性很差。

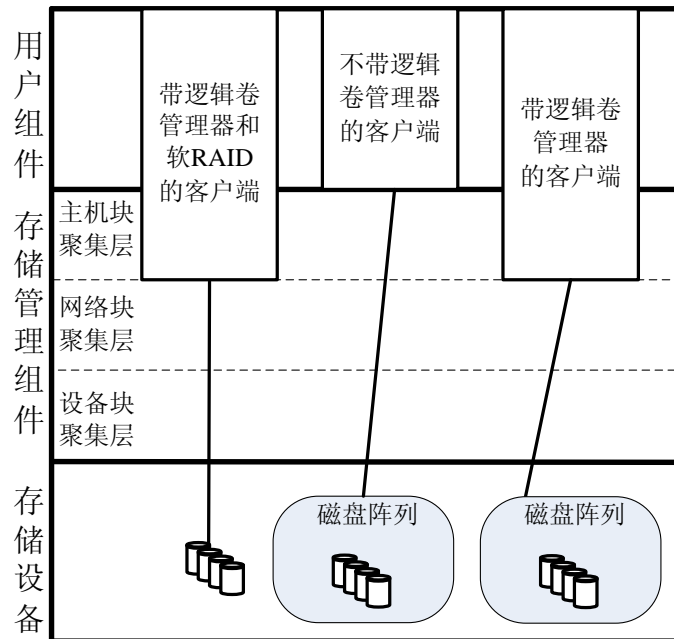


图1.1 DAS存储构架

1.2.2 网络附加存储

NAS 是基于文件接口的，NAS 服务器^{[19][20][21]}实现了网络文件系统的核心功能，负责维护所有的元数据。基于文件接口的 NAS 为用户提供高级别的存储抽象，以

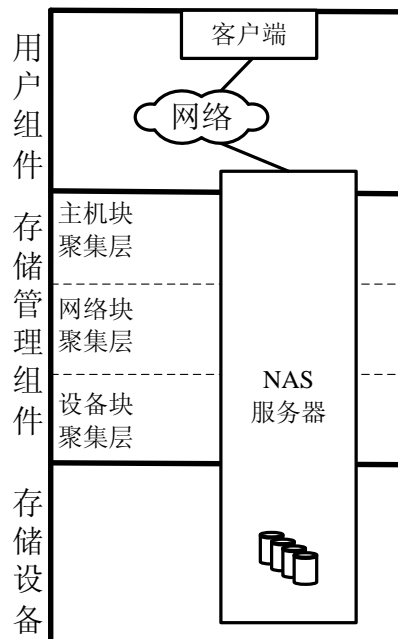


图1.2 NAS存储构架

此能实现数据跨平台（不同操作系统）的共享。客户端通过实现一个简单的网络重定向功能，访问请求经客户端本地文件系统的重定向器通过通用数据传输协议 CIFS/NFS(Common Internet File System/Network File System)转发到网络上远程的 NAS 服务器（Window 平台下 Samba 服务器或 Linux/Unix 平台下的 NFS 服务器）。NAS 存储构架如图 1.2 所示。

NAS 既可以在 DAS 上实现又可以在 SAN 上实现，它包含私有的存储设备。文件系统中用户组件部分和存储管理组件部分全部集成在 NAS 服务器中。由于所有元数据和数据的访问必须经过 NAS 服务器，当请求很多时，它会成为系统访问瓶颈。

1.2.3 存储区域网

在 SAN^{[19][22][23]}中，多个主机(客户端)、块存储设备、元数据服务器通过网络互联起来，通过块接口协议通信。客户端通过访问元数据服务器来获得要访问数据在存储设备上的块数据的布局信息(块元数据)，然后根据该信息直接访问存储设备获取数据。SAN 避开了传统服务器因“存储转发”带来的延迟，具有高吞吐量。SAN 所基于的块接口为用户提供了快速和可扩展的数据共享方式，但是由于块存储设备没有对 IO 访问进行认证的安全机制(在块粒度上建立安全机制的开销非常大)，因此其安全性很低。由于块粒度相对文件粒度要小很多，块元数据数量很多，当系统的规模扩大时，元数据服务器必将成为系统访问的瓶颈。而且虽然客户端可以共享块存储设备中的数据，但这是建立在客户端理解元数据的格式和结构的基础上的，而不同平台的元数据格式是不一样的，给 SAN 的跨平台数据共享带来困难。在 SAN 中，所有客户端共享系统中的元数据和数据块，因此需要采用某种方式保证各客户端之间的元数据一致性。该处理的复杂性导致共享数据块仅仅发生在紧耦合并要求高性能的存储应用中。SAN 存储构架如图 1.3 所示。

在 SAN 中，文件系统中用户组件部分在客户端中实现，存储管理组件部分在元数据服务器中实现(如果客户端带有逻辑卷管理功能，存储管理组件部分的部分块聚集会在客户端上实现；如果存储设备带有磁盘阵列控制器，存储管理组件部分的部分块聚集会在存储设备中实现)。元数据服务器用于将文件的逻辑块映射到块存储设备中的物理块，提供块聚集的空间管理功能。SAN 中元数据访问路径和数据访问路径相分离，其可扩展性优于 NAS。但是由于元数据服务器中元数据的存储和管理以块元数据为粒度，导致元数据数量很多，当系统的规模扩大时，元数据服务器会成为系统访问的瓶颈，其可扩展性没有下面要介绍的基于对象的存储的可扩展性好。在 SAN 中，由于文件包含的数据块元数据由元数据服务器集中管理，存储设

备不知晓数据块之间的联系，基于存储设备内文件级的预取难以实现。

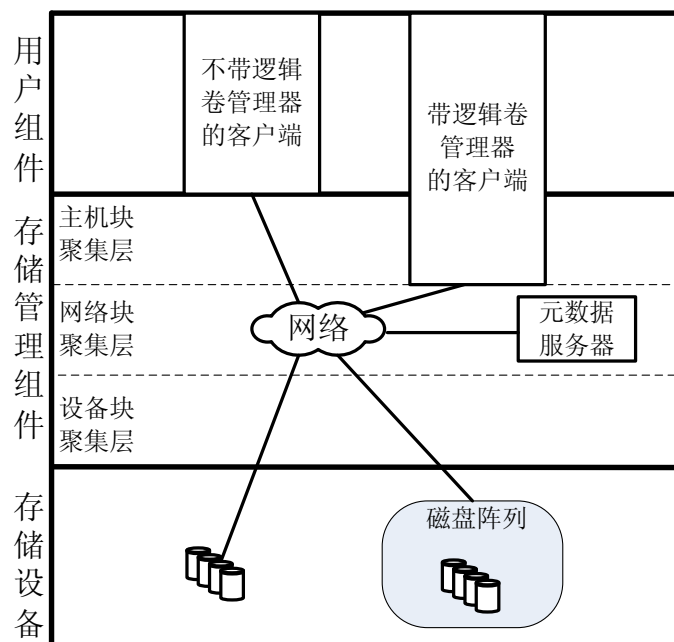


图1.3 SAN存储构架

1.2.4 基于对象的存储

基于文件接口的 NAS 具有安全性和跨平台共享的优点，但其可扩展性差；基于块接口的 SAN 具有快速访问和可扩展的优点，但其安全性和跨平台共享得不到保证。OBS^{[24][25][26]}有效综合了 NAS 和 SAN 的优点，是一种具有高性能、高扩展性、跨平台以及安全数据共享的存储体系结构。在 OBS 中，对象是数据存储的基本单位，与块是固定大小不一样，对象是可变长的，可包含任何类型的数据，如文件、数据库记录、图像以及多媒体视频音频等。对象接口有效地综合了文件接口和块接口二者的优点。同块类似，对象是简单的存储单元，用户能够直接访问对象存储设备来得到对象数据；对对象的直接访问如同对块的直接访问一样能够得到性能上的优势。同文件类似，对象数据的数据块映射信息由对象存储设备管理和维护而对客户端不可见，数据块元数据不需要在不同客户端的文件系统之间共享，这样就很容易对对象进行跨平台的访问。OBS 存储构架如图 1.4 所示。

在 OBS 中，元数据服务器中实现文件系统中用户组件部分的功能，文件系统中存储管理组件部分下移到对象存储设备，同时设备接口也由块接口转换为对象接口。将文件系统中存储管理组件部分下移到对象存储设备后，对象接口消除了

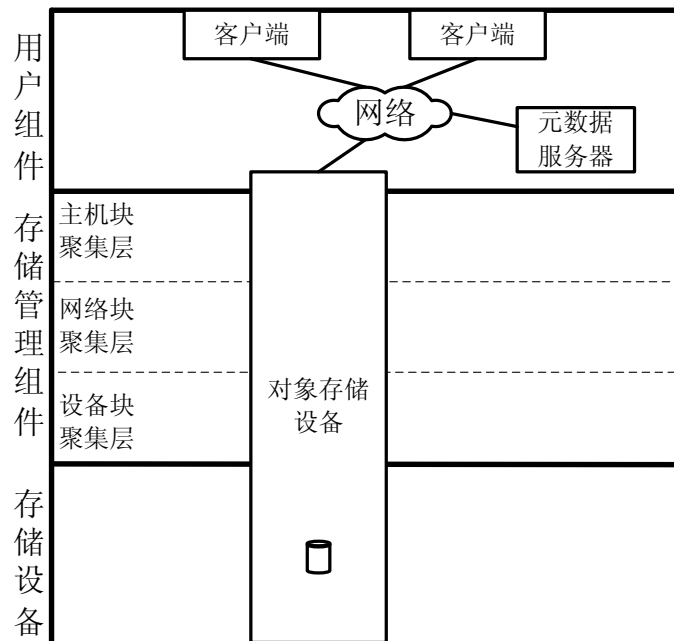


图 1.4 OBS 存储构架

户端的对象数据到对象数据块的块聚集信息，使得不同平台存储应用之间的数据共享切实可行。由于对象数据的空间分配由对象存储设备来完成，对象数据包含的数据块聚集信息由每个对象存储设备来管理和维护，元数据服务器的工作量大大减少，使得系统的可扩展性大大提高(优于 SAN 的可扩展性)。由于每个对象由独立的对象存储设备来处理，以对象为基础实现安全策略简单易行。同时在每个对象中，有关每个对象的数据块聚集信息的管理完全由对象存储设备完成，对象存储设备能够轻易实现对象的预取。

相对于前面的几种存储结构，OBS 的一个重要特色是为对象引入了属性，使得对象接口能提供比其他任何一种接口更为丰富的语义^{[27][28][29]}。对象的属性也属于元数据，用来描述一个 OSD 对象的具体特性、对象的动态和静态信息、可能的访问模式、数据布局方式以及服务质量等，并对外提供接口以便于客户端能够检索或存储其需要的属性。访问对象的属性可使存储系统更好地组织数据和提供服务，对象属性的引入使得系统的灵活性和可管理性得以提高。

1.3 相关的研究工作

基于对象的存储结构具有的高性能、高可扩展性、高安全性、跨平台数据共享的优点使得它成为网络存储研究中的热点。在对象存储系统中，由于元数据请求访问频繁，元数据的存储与管理方式以及元数据在系统的不同存储部件中的分布会对

元数据的访问效率以及系统的整体性能产生重要影响，对象存储系统中元数据管理研究异常重要。本节最先给出当前流行的一些对象存储系统中元数据管理的相关研究，然后给出本文元数据管理研究中涉及到的关键技术的相关研究情况。

1.3.1 对象存储系统的元数据管理

Cluster File Systems 公司研究开发的 Lustre^[30]是一个基于对象存储的高性能 Linux 集群，它已经在美国能源部和许多国家实验室等地方得到应用。在 Lustre 中，元数据集中存放在单个元数据服务器中，为了保证元数据的可靠性，采用备份元数据服务器提供失效接管元数据服务来避免单点失效。当存储系统规模增大时，单元数据服务器会成为系统访问瓶颈。在 Lustre 中，元数据服务器利用本地 Linux 底层 ext3 文件系统的索引节点与扩展属性来存放系统中的元数据信息。每个文件和目录对应元数据服务器中的一个索引节点(inode)，该索引节点用来记录系统中文件或目录的基本元数据信息(如文件大小、修改时间等)。由于文件或目录数据(集中存放该目录包含的文件或目录的文件名和对应的索引节点号的目录项信息)分割为一个或多个对象存放在对象存储设备上，元数据服务器中需要记录该文件或目录数据对应对象所在的对象存储设备信息，这些信息存放在与该文件或目录对应的索引节点相关的扩展属性中。由于 Lustre 的元数据服务器中的元数据存储与管理依赖于底层 ext3 文件系统，ext3 文件系统的容量受限导致其包含的索引节点总数受限，最终导致 lustre 系统中文件总数受限；ext3 文件系统中单个文件的最大长度受限导致每个目录下最多包含的子目录和文件数受限。这些限制最终导致采用单元数据服务器的 lustre 的系统可扩展性受限。由于对象分布信息存储在扩展属性中，而 ext3 文件系统中扩展属性被存放在索引节点之外的磁盘块中，导致对象分布信息的查找效率低。

Panasas^[31]是 Panasas 公司研究开发的基于对象的并行集群存储系统，它采用元数据服务器集群来管理系统中的元数据，而元数据的永久存储位于对象存储设备。系统的名字空间被分割为多个子树，每个子树作为一个卷由单个元数据服务器进行管理，即其分布式元数据管理方案采用了静态子树分割的方式。静态子树分割方案实现简单，但是由于每个子树包含的文件总数和访问热度差别很大，会导致元数据服务器之间的负载不均衡，从而形成系统访问瓶颈。在 Panansas 中，不同大小的文件采用不同的 RAID 方案映射为多个对象存放到不同的对象存储设备上。基本元数据信息(如文件大小、修改时间等)和文件的存储映射元数据信息(包括文件被如何分割为哪些对象等)被作为文件元数据永久存储在对象存储设备中该文件对应的前两个对象的属性中，该文件包含的其他对象中仅存储该对象的基本属性。和传统的

UNIX 文件系统一样，目录数据中包含了该目录下包含文件或目录的目录项信息，对目录项信息进行扩充，使其记录了该文件对应的对象标识符以及存储该文件元数据的对象位于的对象存储设备信息。由于目录数据和文件元数据被分散地永久存储在对象存储设备上，当初次访问某个文件元数据时，需要多次磁盘操作才能访问到该文件元数据并将其缓存到元数据服务器中，元数据访问效率不高。

Ceph^[32]是由 University of California, Santa Cruz (UCSC) 的存储系统研究中心 (SSRC) 研究开发的对象存储原型系统，元数据服务器集群采用动态子树分割对文件的基本元数据信息进行管理。动态子树分割是粗粒度的分割方案，它需要一定的时间来在元数据服务器之间均衡负载。在 Ceph 中，文件元数据嵌入到目录数据的目录项中，相比传统文件系统的组织方式减少了元数据访问的磁盘操作。目录数据被分割为对象永久存放到对象存储设备上，其锁粒度导致在该目录下创建或删除文件的元数据操作需要读取该目录数据所在对象的所有字节，降低了元数据访问性能。而且将文件元数据与数据混合存放在一起，并不能针对他们的不同访问特点做优化。Ceph 相比其他系统的一个最大区别是其不需要维护文件的存储映射元数据信息。在 Ceph 中，由于对象的长度是固定的，对象标识符基于该对象数据在文件的偏移地址顺序生成，不需记录该文件包含的对象列表信息。由于采用分布式算法直接计算出对象所在的对象存储设备，不需要记录对象到对象存储设备的映射关系。采用这种方式简化了设计，减少了元数据服务器需要维护的元数据种类，减轻了元数据服务器的负担，但是由于分布式算法的确定性，导致对象在对象存储设备之间的分布不灵活。

1.3.2 分布式元数据管理

当前越来越多的大规模分布式文件系统采用数据传输路径与控制路径相分离的体系结构，存储系统中的元数据由元数据服务器集中管理以提高数据路径上的可扩展性。通常会根据系统的规模和可扩展性的需求来选择是采用单元数据服务器还是元数据服务器集群来管理元数据。

在一些文件数量较少、访问模式比较单一的存储系统中，单元数据服务器足以能够满足用户需求，不会成为存储系统访问瓶颈，并且能简化系统硬件维护开销。如在 GFS(Google file system)中包含数量不多的大文件，并且大部分文件除了最初创建文件时的写入操作外几乎都是读操作，因此采用单元数据服务器就能满足访问的需求^[33]。

由于单元数据服务器能够提供的元数据访问能力有限，当存储系统规模较大或元数据访问请求频繁时，它会成为系统访问性能瓶颈。元数据服务器集群能提供高

性能的元数据访问，分布式元数据管理用来在元数据服务器集群内分布和管理元数据。当前存储系统中常见的分布式元数据管理方案的分类以及各种方案的优缺点描述如下。

- 子树分割

静态子树分割^{[34][35][36][37]}是一种简单而传统的在元数据服务器集群间分布元数据负载的方法。它将存储系统全局唯一的名字空间按照目录层次分割为独立的子树，元数据服务器集群中的每个元数据服务器负责管理其中的一个或多个。当要访问元数据时，通常仅仅需要访问某个元数据服务器即可直接获得期望的元数据，通常不需要在元数据服务器之间转发。当每个元数据服务器上分布的访问负载均匀时该方案很有效。由于静态子树分割方案的分割粒度很粗略(分割粒度是子树)，当元数据负载动态改变时，元数据负载不能在元数据服务器集群间均匀分布，可能会导致某个或某些重负载的元数据服务器成为整个存储系统的性能瓶颈。

动态子树分割^[38]是对静态子树分割的改进方案。单个目录层次子树能被继续细分到不同的元数据服务器。随着负载的动态变化，元数据负载能在元数据服务器之间动态重分布。相比静态子树分割，动态子树分割的粒度更小更灵活，它能将负载较重的元数据服务器上的部分子树重新委派给其他的不忙的元数据服务器；实现元数据服务器之间的均衡负载。动态子树分割将元数据存储在一个共享存储中，以避免将元数据存储在每个元数据服务器后端磁盘会出现的当委派改变时导致的不同磁盘间元数据迁移开销。当访问某个文件的元数据时，由于文件所在路径上涉及到的目录可能委派给了不同的元数据服务器，因此可能需要访问到多个元数据服务器。动态子树分割仍然是粗粒度的分割方案，它需要一定的时间来均衡负载。

- 哈希

静态哈希^{[39][40][41][42]}通常通过对文件标识符(如文件路径名)进行哈希来定位存储该文件元数据的元数据服务器。通过哈希计算，客户端能够直接定位到其所要访问元数据所在的元数据服务器。哈希能在元数据服务器之间均匀分布负载，然而哈希削弱了元数据访问的局部性。虽然文件元数据在元数据服务器集群中均匀分布，但为了支持 POSIX(Portable Operating System Interface of Unix)兼容的语义，仍然需要层次目录结构来支持目录操作，该层次目录结构可能会成为访问热点。由于静态哈希通常采用文件的全路径名来定位，当元数据服务器集群规模改变或者重命名一个目录时，可能会导致大量的文件元数据在元数据服务器之间迁移。该方案适合在元数据服务器规模不会发生改变的存储系统中运行重命名操作非常少的负载。

在以上的几种分布式元数据管理方案中，当需要检查或修改文件访问权限时需要沿着文件路径进行目录遍历，由于该路径上的目录可能位于不同的元数据服务

器，会导致高访问开销。为了避免目录遍历，Lazy Hybrid (LH)^[43]采用双项访问控制列表(Access Control List, ACL)来直接决定文件的访问权限。每个目录(或文件)都包含两个 ACL，一个是路径权限，一个是文件权限，其中路径权限在其被创建时通过将该目录(或文件)的父目录的路径权限和自身的文件权限做相与操作生成。LH 是对静态哈希方案的改进方案。在 LH 中，首次采用元数据查找表(Metadata lookup table, MLT)来减少由于元数据服务器集群规模改变导致的大量的元数据的迁移。同时，LH 推出懒惰元数据更新策略来推迟文件元数据的更新与迁移。相比静态哈希，它有效减少了元数据服务器之间突发的网络开销，但是它并没有从根本上减少或消除由于重命名一个目录带来的大量的元数据迁移。它适用于重命名操作很少的负载情况。

在基于目录路径的元数据管理^[44]中，引入目录路径索引项将目录路径名映射为全局唯一的目录路径标识符，采用目录路径标识符来定位一个目录下的文件元数据的位置。它避免了由于重命名一个目录导致的元数据的迁移。所有的目录路径索引项被集中存放在目录路径索引服务器中，可能会导致访问瓶颈。单个目录下的所有文件元数据被绑定成一个或多个桶，作为目录路径对象由单个元数据服务器来管理。当在某个目录下创建或删除文件或目录时，桶内的线性查找和为了维持文件系统正确性需要的以桶为粒度的锁(锁的粒度较大)会影响文件元数据的访问性能。当某个目录需要经常被访问到时，通常采用一个负载较轻的元数据服务器作为该目录路径的从元数据服务器(将主元数据服务器上该目录路径对象复制到从元数据服务器)来为元数据的读操作提供服务以避免主元数据服务器过载，但这样会招致严重的一致性开销。它适用于目录数量比较少的负载。

● 其他方案

Hierarchical Bloom Filter Arrays (HBA)^[45]是一种新颖的用于分布式元数据管理的技术。文件元数据能存储在任意元数据服务器上，由于文件元数据存储位置的不确定性，需要提供查询服务来获取所要访问的文件元数据驻留的元数据服务器。它采用两级 Bloom Filter^{[46][47][48][49]}矩阵来定位元数据服务器。第一级 Bloom Filter(BF)叫 LRU(Least Recently Used) BF，显示出该元数据服务器上 LRU 列表中包括的访问最频繁的文件元数据。第二级 BF 显示出该元数据服务器上包含的所有文件元数据。该元数据服务器也保存所有其他元数据服务器上的文件元数据的两级 BF 副本。在 BF 中进行查找时，将文件全路径名或者文件名转换为 BF 阵列的数字索引，然后随机连到一台元数据服务器上，在其上面查找第一级 BF 阵列，如果找到，则直接联系 BF 中所指出的存放该文件的元数据服务器去取得文件的元数据，第一级阵列的缺失导致对第二级阵列的查询，第二级阵列的缺失导致该元数据请求在元数据服务

器之间广播。

Group-based Hierarchical Bloom Filter Array (G-HBA)^[50]是对 HBA 的改进方案，它利用基于组的 BF 来在元数据服务器之间有效转发请求，提高了 MDS 集群的可扩展性，与 HBA 相比减少了空间开销。

这两种方案适合于大多数请求是读操作的负载，它们能提供细粒度的负载均衡。对于元数据查询服务，它们仅仅给出了如何查找文件元数据所在元数据服务器的方法，并没有给出在元数据服务器集群中如何分配文件元数据的方法，即如何决定一个文件的元数据应该放在哪个元数据服务器中。在实际的存储系统中，能采用简单、确定性的查找方案即可决定元数据是如何在元数据服务器集群中分布的，不需要将查找操作如此复杂化。另外，它们不能对目录相关的操作做出有效处理。

上述分布式元数据管理方案概括如表 1.1 所示。

表 1.1 各种分布式元数据管理方案概括

方案	分割粒度	负载均衡级别	是否支持目录语义	是否对热点目录进行处理	重命名开销	元数据服务器规模改变时迁移开销
静态子树分割	子树	低	是	否	无	小
动态子树分割	子树，能随负载变化继续细分	中	是	是	无	小
静态哈希	文件	高	是	是	大	大
LH	文件	高	是	是	大	小
基于目录路径的元数据管理	目录	高	是	否	小	小
HBA	文件	高	否	---	大	小
G-HBA	文件	高	否	---	大	小

1.3.3 数据放置策略

数据放置策略主要解决数据如何在存储系统中分布的问题，它对存储系统的性能有很大的影响。以往对放置策略的研究主要在三个不同的层次上展开：(1)磁盘内

部数据块的放置, (2)一个存储节点或磁盘阵列内多个磁盘之间的放置, (3)网络存储中多个存储设备之间的放置。本文中关注数据在网络存储中多个存储设备之间的放置。

磁盘内部的数据块放置策略研究实际上就是对存储管理组件的研究, 它用来完成数据逻辑视图到物理视图的映射。其研究目的是通过优化数据块在磁盘内部的布局来提高磁盘的 IO 性能, 减少访问时间。由于磁盘中间磁道访问速度最高, Organ pipe heuristic^[51]提出把访问最频繁的数据放置在磁盘中间的磁道、次经常访问的数据放置在中间磁道的两侧、访问最少的数据放置在磁盘的边缘, 通过这种方式可以将寻道时间缩短 40~45%。FFS^[52](Fast File System)提出同属于一个文件的数据块应尽可能的在磁盘物理地址上邻接。C-FFS^[53](Co-located FFS)为同一个目录下的小文件分配连续的空间, 便于同一个目录下相关文件局部性访问的预取。

针对磁盘在容量及存储速度上都无法跟上 CPU 及内存的发展, RAID^{[54][55][56]}通过多个磁盘之间数据的并行访问来提供高性能同时它可以提供容错功能。RAID5 中每个分条跨越了磁盘阵列的所有磁盘, 在磁盘失效后的每条数据重构中都需要访问所有磁盘, 导致它的在线恢复性能不好。针对 RAID5 的这一缺点, 校验分离(Parity declustering)^[57]技术对此进行改善。它减少了 RAID5 校验组的大小, 使得每个分条不需跨越整个磁盘阵列, 通过磁盘阵列中磁盘并行重构提高了重构性能。RAID 校验组大小的选择很关键, 校验组越大, 其用于存储校验数据的容量开销越小但其重构性能越差; 校验组越小, 其重构性能越好但其用于存储校验数据的容量开销越大。由于 RAID 的可扩展性很差, 当它包括的磁盘数目发生改变时, 需要重新创建 RAID 并重新放置数据。为了解决这些问题, RAID-II^{[58][59]}和 TickerTAIP^[60]等项目对此做了进一步探索。

针对应用场景和系统配置的不同, 网络存储中多个存储设备之间的放置策略通常可以分为两类: 静态放置策略和动态放置策略。

● 静态放置策略

静态放置策略常用于存储规模较小且设备总数不会发生变化的存储系统。静态放置问题描述为: 已知存储系统中有 N 个存储设备, 要存储 M 个文件, 如何分配才能使得系统性能最优, 这是个 NP 问题^[61]。在静态放置策略的研究中, 通常只采用一种分布方式对文件进行放置, 如绝大多数研究以文件或文件组为粒度来对文件进行放置, 使得每个文件仅被放置到单个存储设备上, 不能充分利用存储设备之间的并行性; 部分研究采用在存储设备之间条带化分布文件来对其进行放置, 当文件较小时引入了较大的系统开销。

Greedy^[62]、SP^[63]和 HP^[63]策略中都假定文件访问频率服从 zipf^[64]分布, 文件访

问频率与文件大小呈反比，它们的目标是减少系统的平均响应时间，在这三种方案中均是将每个文件存放到单个存储设备上。**Greedy** 既能用于在线执行又能用于离线执行。当其离线执行时，它的基本思想是：最先计算出所有文件的平均负载，然后将负载之和等于平均负载的连续的文件集放在每个存储设备上；当其在线执行时，它的基本思想是：将当前文件分配到有最小负载的存储设备上。**SP(sort partiton)**的基本思想是：按所有文件的服务时间的降序来将文件排序为列表。分配给每个存储设备一片连续的列表分段，直至该存储设备的负载到达平均磁盘负载。所有磁盘分配完后，如果还有剩余的文件，则统一安排到一个特定磁盘。**HP(Hybrid partiton)**的基本思想是：假定文件按批到达，首先对它们进行排序，尝试在最小化存储设备之间负载差值的同时最小化每个存储设备内部的服务时间的变化。

Tao Xie^[65]发现文件的大小与其访问频率并没有太大的相关性(即不服从 zipf 分布),因此需要重新衡量已存在的静态放置策略是否有效,提出 **SOR(static round-robin)** 用于处理服从泊松到达率和固定服务时间的文件的分布。**SOR** 将文件集按文件大小排序,使得大小相似的连续文件能被分配到同一个磁盘上,将最热的文件分到了不同的磁盘上,在不论相关性是否存在的情况下都提供好的平均响应时间,它只能离线执行。

一种流媒体文件的分块放置方法^[66]在分析了文件内部访问倾向性的基础上(文件数据的首部分块的访问频率远远大于其它分块),提出将多个流媒体文件对称放置来克服访问倾向性带来的负载不均衡。由于文件总数和每个文件的访问倾向性随时间变化,它只能为系统负载均衡提供概率保证。

中南大学^[67]提出了包括大文件分割策略 **KKFDA^[68](Knowledge Known File Declustering and Allocation, 已知知识的文件拆分与分配策略)**、小文件分组策略 **APD(Available Percent Decision-making, 可用百分比决策)**和 **CSSAPD(Combination of Subsection Select and Available Percent Decision-making, 分段选择与可用百分比决策相结合)**等一系列放置策略。其中 **KKFDA** 假定在进行文件放置前就已经得知了文件的磁盘数据访问模式,从而能根据数据访问模式的需要将文件进行分割存储。它需要将每个文件复制 N 次,然后将这 N 个副本在磁盘间进行有效分配。**APD** 和 **CSSAPD^[69]**是针对将大量小文件组成文件组的分配存储策略的讨论,它根据磁盘剩余空间的多少来决定如何分配文件组以提高系统可扩展性。

● 动态放置策略

在大规模存储系统中,存储规模可能会经常变化(如为了满足前端应用的存储容量或存储性能的需求,可能会周期性的添加一批新的存储设备到存储系统中,或将老化的存储设备从系统中移除,而且在大规模系统中存储设备的失效是常规事件

[33]), 而且其支持的应用多样化, 文件访问特性未知(无法预先知道文件的访问频率等特性), 存储系统包含的文件总数可能是数以亿计的, 此时无法利用上述的静态放置策略去求解近似最优解, 在这种环境下采用的一些放置策略称之为动态放置策略。

对于动态放置策略, 通常从以下几个方面来评价策略的好坏: (1)计算或查找对象到存储设备映射关系需要的计算或查找开销; (2)描述映射需要的存储开销; (3)映射的均匀性; (4)当存储设备个数发生改变时, 为了维护算法的一致性或存储设备之间的负载均衡性导致的对象在存储设备之间迁移开销。为对象选择存储设备的方法通常分为两种: 启发式方法和分布式算法。在启发式方法中, 对象能存放在任意的存储设备上, 采用映射表记录它们之间的映射关系。在这种方法中, 对象在存储系统中的分布灵活, 但映射表存储开销大。在分布式算法中, 能够直接计算出对象所在的存储设备, 不需要映射表来记录对象与对象存储设备的关系, 存储开销可以忽略不计, 计算开销随算法的不同而不同。由于分布式算法中对象到存储设备的映射是固定的, 对象分布不灵活。常见分布式算法的优缺点描述如下。

线性哈希^{[70][71][72][73]}和可扩展哈希^[74]都是常用的对简单哈希的改进方法, 它们解决了在简单哈希中由于存储系统中存储设备总数变化导致的大量不必要对象迁移的问题, 但是它们在新加入存储设备时, 通过计算要将某个存储设备的近一半对象迁移到新存储设备中, 迁移仅仅发生在这两个存储设备之间, 导致访问热点, 并且重新放置后的对象分布不均匀。

Brinkmann^{[75][76]}提出通过引入额外的抽象层 $[0,1)$ 区间来提供映射的灵活性, 将存储设备指派到该 $[0,1)$ 区间的某个子区间, 将映射到同一子区间的对象放置到该存储设备上。当存储设备发生改变时, 通过改变其对应的子区间来改变对象的映射关系, 但其迁移的对象不是最优的。ANU^[77]和 R-SIEVE^[78]均是建立在该策略的基础上。

DIM(Dynamic Interval Mapping)^{[79][80]}也将存储系统中的存储设备映射到 $[0,1)$ 中某个区间, 将被映射到该区间的对象存储在存储设备上。当存储设备数发生改变时, 将当前的小区间分割为多个更小的区间, 更新区间到存储设备的映射关系, 使得对象在存储设备间既能均匀分布又能使得对象迁移数最小。它与 IRR(Interval Round Robin)^[81]的基本原理一样, 但 IRR 只能支持存储设备的增加, 而它也同时能支持存储设备的减少。DIM 要记录区间到存储设备的映射关系, 其区间数在只考虑存储设备增加的情况下与(增加次数 \times (增加次数+初始存储设备数))的值呈线性关系, 每次对象的映射都需要查找其对应区间所属的存储设备。

Bit Window 中将对象通过 SHA-1 函数^[82]转换为 m 位标识符^[83]。设存储系统中存储设备总数为 N 个, 这 m 位被分割为窗口大小为 $\lceil \log_2(N) \rceil$ 的若干个窗口, 按从

右至左的优先级根据窗口中的值来判断对象所在存储设备。当存储设备个数发生改变时，该方法在常规情况下能迁移比较少的对象，但是当 N 的变化引起窗口的变化时，需要迁移系统中大约 50% 的对象，对象迁移开销大。

Honicky^{[84][85]}等提出一系列的策略来放置对象，其中每种方法都有一定的缺陷，如对于 $RUSH_P$ 来说，其计算时间开销是线性级的，且当删除存储设备时，会导致不必要的对象迁移； $RUSH_R$ 的最坏情况下的计算时间开销也是线性级的； $RUSH_T$ 也会在存储设备变化时引入额外的数据迁移。

以上的算法中都企图用单个原子性的对象放置策略来解决网络存储中的对象放置问题，但是由于每个策略不可能在各个方面的性能都是最优的，下面的策略通过采用多种原子性策略混合的方法来解决对象放置问题。

CRUSH^[86]中包括的四种原子性策略中，Uniform bucket 实际上就是简单哈希，它能快速地在存储设备之间均匀分配对象，但当存储系统中存储设备发生变化时会导致大量对象迁移；List bucket 的计算时间开销是线性级的，当存储设备增加时能保持最优对象数的迁移，但当存储设备减少时会导致大量不必要的对象迁移；Tree bucket 的计算时间开销是 \log 级，但也引入了额外的对象迁移；Straw bucket 虽然能在存储设备发生变化时保持最优对象数的迁移，但其计算时间开销是线性级的。CURSH 提出将存储设备组织成不同的集群，根据存储系统预期的集群增长模式，在对象迁移和计算开销之间对 bucket 类型做折中选择。

Sorrento^[87]中综合考虑了启发式方法和分布式算法的优缺点，根据对象的大小将对象分为两种：大对象和小对象，小对象采用一致性哈希^[88]策略放置，大对象通过以与剩余空间成正比的概率选择存储设备来放置，并用 Bloom filter^[46]来跟踪大对象所在的存储设备，它兼顾了对象分布的灵活性和系统的可扩展性，但其小对象的计算开销和大对象的存储开销均随系统中存储设备总数线性增长。

1.4 本文研究的主要内容

在对象存储系统中，元数据服务器集群管理的元数据是重要的系统数据，客户端在访问对象数据之前，需要将其要访问文件的文件名发送到元数据服务器中，由元数据服务器返回客户端访问文件包含的文件描述信息以及该文件到对象的空间组织信息，用户根据这些信息才能到相应的对象存储设备中访问对象数据。有研究表明^{[89][90]}，元数据的访问是非常频繁的，它的访问速度影响了系统的整体性能，是系统访问潜在的瓶颈，依据 Amdahl 定律^[91]，提供高效的元数据访问可以大大提高系统的整体访问性能。在对象存储系统中，数据放置策略负责将文件映射为对象，并为新创建的对象选择合适的对象存储设备存放，在需要访问对象时能够定位到对

象所在的对象存储设备，通过该策略，每个文件被分布到一个或多个对象存储设备上，它对系统的性能有关键性的影响。基于元数据在系统中的重要地位，对元数据的可靠性的维护异常重要。

根据前面的讨论，研究对象存储系统中元数据管理是本文的主要目的，主要内容包括：研究对象存储系统中元数据的描述以及元数据管理框架；研究元数据服务器集群中的元数据存储与分布方式、元数据负载均衡等；研究在不同系统规模时的数据放置策略、分布式算法；研究元数据可靠性。涉及到的关键技术主要包括：

1. 元数据服务器集群中的分布式元数据管理方案

在对象存储系统中，元数据服务器集群用于管理与上层文件系统相关的元数据(用户组件部分)，给客户端提供统一的逻辑视图，呈现出传统的 UNIX 目录树型结构。在深入分析传统文件系统中用户组件相关元数据的组成结构基础上，基于数据库能提供高事务吞吐量的特点，提出了一种改进的元数据存储和管理方法，它能提供高性能和可扩展的元数据访问。它将每条文件元数据作为一条记录(record)存储在数据库中，同时不再在持久存储(如磁盘)中存储层次目录结构(即，维护每个目录中包含的文件和子目录的文件名与其对应索引节点号映射关系的目录数据)，避免层次目录结构自身成为热点，采用一种间接的方案来模仿层次目录结构；引入目录转换元数据来避免子树分割方案中的目录遍历和哈希方案中的重命名目录导致的大量元数据迁移；针对每种元数据自身的访问特性，采用不同的分割方法将其在元数据服务器集群中分布以提供高性能的元数据访问和提高元数据服务器集群的可扩展性，对目录转换元数据按照记录关键字的字典序分割，对文件元数据采用元数据查找表来分割。

2. 元数据服务器集群中的负载均衡算法

由于元数据访问负载随时间动态变化，元数据在元数据服务器集群中的静态分配可能会导致某一时刻某个元数据服务器成为系统元数据访问性能的瓶颈，为了在元数据服务器集群中提供高性能、可扩展的元数据服务，需要在元数据服务器之间均衡负载。提出一种以文件元数据请求的响应时间为衡量标准、应用于元数据服务器集群的负载均衡算法。

3. 数据放置策略

在对象存储系统中，数据放置策略负责将文件映射为对象，并为新创建的对象选择合适的对象存储设备存放，在需要访问对象时能够定位到对象所在的对象存储设备。

针对对象存储系统存储规模较小、对象存储设备总数固定的应用环境，提出一种利用遗传算法根据文件的不同特性求解文件放置问题的策略，寻求系统开销与性

能的近似最优解。

在大规模对象存储系统中，包含的对象存储设备总数成百上千，对象存储设备的失效属于常规事件，系统规模会经常发生变化，此时静态放置策略已经不再适用。基于对象存储系统规模的变化趋势，提出基于组的区分定位策略，首先根据对象存储设备加入对象存储系统的不同时期将每个对象存储设备划分到不同的存储子集群，先采用分布式算法将对象映射到对象存储系统的某个子集群中，再在子集群内部根据不同类型的对象采用不同的映射方法，对新创建的大对象采用启发式方法来选择负载较轻的对象存储设备存放，对小对象采用分布式算法来确定其所在的对象存储设备，兼顾了对象分布的灵活性和系统可扩展性。

针对对象存储系统中子集群内对象存储设备规模的变化趋势，提出一种分布式算法——改进哈希算法，它既能在子集群内的对象存储设备之间均匀分布对象，又能以非常小的计算开销直接计算出对象所在的对象存储设备，并且它能以近似最优的对象迁移开销有效支持子集群内的对象存储设备规模的变化。

4. 元数据可靠性研究

在对象存储系统中，元数据服务器负责维护和管理对象存储系统中的元数据。对象存储设备上的对象数据仅仅采用 128 位的对象标识符来标识，它不了解用户对象相关的文件和目录信息，如果元数据服务器失效，对象就不能再被客户端访问到，因此元数据的可靠性的维护至关重要。利用对象存储系统的富有表达力的对象接口设计采用扩展属性页来提高元数据可靠性的方法能够充分利用对象存储的优势保障元数据的高可靠性。

1.5 论文组织

论文共分为五章，组织结构如图 1.5 所示，围绕对象存储系统中的元数据，对元数据的管理研究贯穿全文。

第一章首先阐明了元数据的重要作用以及它在当前使用最多的四种存储结构中的分布；介绍了相关的对象存储系统中元数据管理研究现状以及本文中元数据管理研究涉及到的关键技术的相关研究情况；最后介绍了本文研究的主要内容及论文组织。

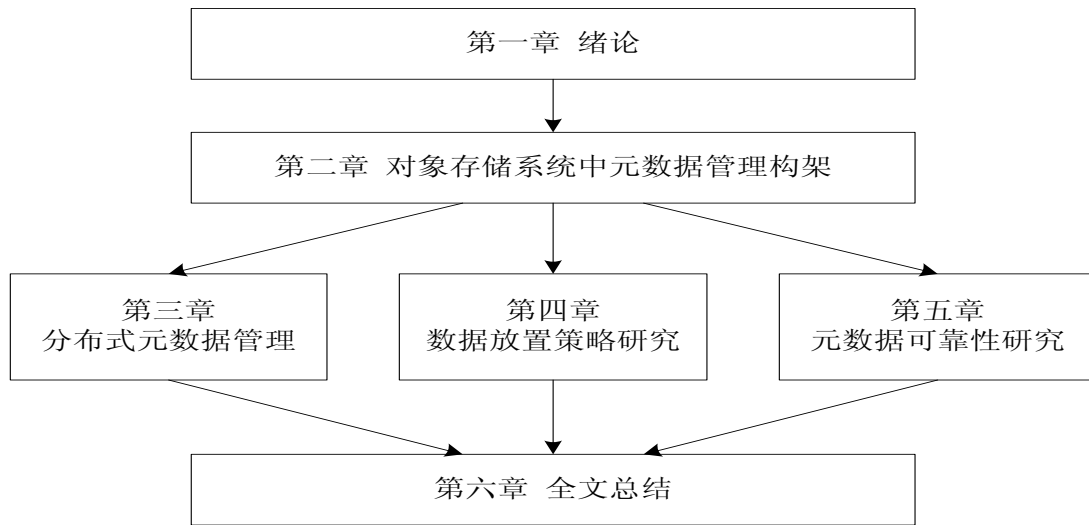


图 1.5 论文组织结构

第二章首先介绍了自行设计的对象存储系统，描述了该对象存储系统的体系结构以及它所包含的各个组成部件的主要软件框架；给出元数据在对象存储系统中的分布，并对元数据进行描述；提出对象存储系统中元数据管理框架。

第三章介绍了元数据服务器集群中分布式元数据管理方案的研究。主要介绍了元数据存储与分布方式、元数据负载均衡方法，给出了该分布式元数据管理方案的性能测试与分析，论证了该方案的有效性。

第四章介绍了数据放置策略的研究。针对存储规模较小、对象存储设备总数固定的应用环境，提出一种利用遗传算法根据文件的不同特性求解放置问题的策略，寻求系统开销与性能的近似最优解；针对大规模对象存储系统中子集群内对象存储设备规模的变化趋势，提出一种分布式算法——改进哈希算法，能以非常小的计算开销均匀分布对象，且以近似最优的对象迁移开销有效支持子集群内的对象存储设备规模的变化；针对大规模存储系统提出基于组的区分定位策略，兼顾了对象放置的灵活性和系统的可扩展性。

第五章介绍了元数据可靠性的研究。提出利用对象存储系统的富有表达力的对象接口设计采用扩展属性页来提高元数据可靠性的方法，它能充分利用对象存储的优势来保障元数据的高可靠性。

最后，第六章对全文总结。

2 对象存储系统中元数据管理框架

对象存储系统引入了一种新的存储接口——对象接口，使得网络存储发生了革命性的变化，从而具有高性能、高可扩展性、高安全性、跨平台数据共享的特点。对象存储系统相比于传统网络存储在体系结构上发生了一定的变化，其对象接口的引入使得元数据在系统中的分布和管理也发生了一定的改变。针对对象存储系统需求，本章主要设计了对象存储系统的元数据管理框架。

2.1 对象存储系统体系结构

对象存储系统中包含对象存储设备(Object-based Storage Device, OSD)、安全管理器(Security Manager, SM)、策略/存储管理器(Policy/Storage Manager, PSM)、客户端和元数据服务器(MetaData Server, MDS)集群，它们通过网络互联，数据传输路径和访问控制(元数据)路径相分离，系统具有很好的可扩展性。图 2.1 描述了对象存储系统体系结构。

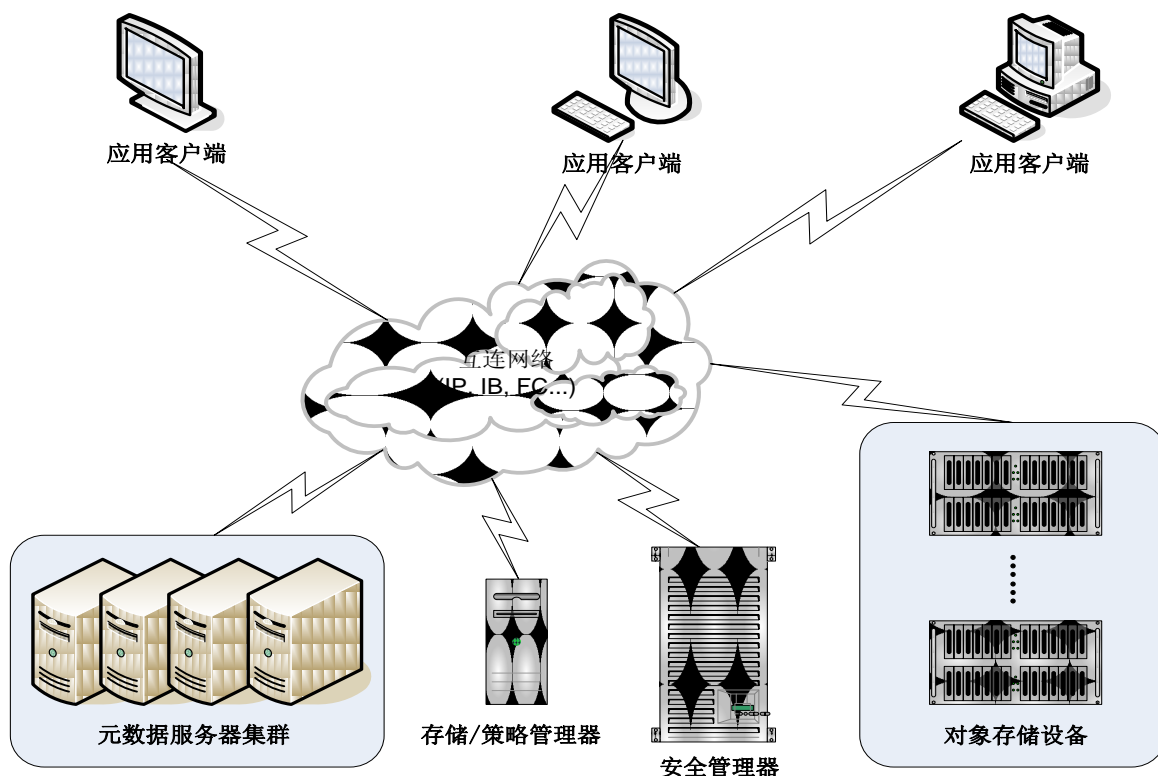


图 2.1 对象存储系统体系结构

OSD 是存储系统中共享的存储部件(例如， RAID 子系统，磁带驱动器，磁带

库，或者其它的存储设备)。通过网络，客户端使用多个共享的 SCSI 启动端口来直接访问 OSD。PSM 的作用是协调 OSD 和客户端之间的访问限制，为客户端准备放置在命令描述块(CDB)中的能力段(Capability)，该能力段用来控制对 OSD 中对象和命令单元的访问。SM 用于保护凭证中的能力段。MDS 集群管理与上层文件系统相关的元数据(用户组件部分)，提供给客户端节点统一的逻辑视图，呈现出传统的 UNIX 目录树型结构。PSM、SM 可以实现在 OSD 上，也可以实现在客户端上，也可以实现在 MDS 上，或者是作为一个独立的组件实现。

2.1.1 客户端

客户端文件系统 CFS(Client File System, CFS)作为一个可安装的内核模块运行在 Linux 内核下。它实现了标准的 VFS(Virtual File system Switch)接口，客户端挂载该文件系统，对外呈现一个 POSIX^[92]接口。CFS 能运行在 2.4.或 2.6 Linux 内核下，不需要对内核有任何修改，确保了系统的兼容性和可移植性。客户端软件模块如图 2.2 所示。上层文件请求通过 VFS 层传递到客户端文件系统 CFS，CFS 通过 Socket 访问 MDS 以获取要访问文件的描述信息和空间组织信息，然后根据数据放置策略得知要访问文件数据所属对象以及相应对象所在的 OSD。客户端然后向 SM 请求其要访问对象的凭证，凭证包括 PSM 产生的能力段。凭证授予客户端访问指定 OSD 的权力，能力段键码使得客户端和 OSD 可以用完整性校验值对它们之间传递的对象命令和数据进行验证。客户端将封装的对象命令通过 iSCSI 通道发送到相应的 OSD 执行。访问过的文件元数据和对象数据以一定的策略存储在客户端的缓存中，提高访问效率。基于访问效率的考虑，一般将 SM、PSM 的功能实现在 MDS 上^[93]。

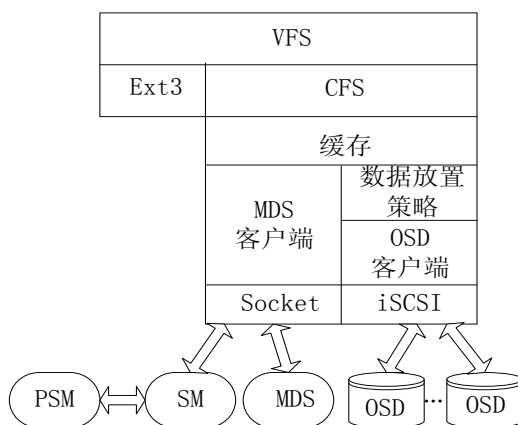


图2.2 客户端软件模块

2.1.2 策略/存储管理器与安全管理器

对象存储系统区别于 SAN 的一个重要特征是其安全性。在 OSD 中，各分区对象之间所采用的安全方法可能是不同的。除了 NOSEC(非安全)安全方法外，还包括基于凭证的访问模型的三种安全方法 CAPKEY(能力段的完整性)、CMDRSP(CDB、状态和侦听数据的完整性)和 ALLDATA(所有传输数据的完整性)。客户端能够根据他们的需求来选择使用不同的安全策略。在 NOSEC 安全方法中，OSD 不使用任何安全特性或算法。在 CAPKEY 安全方法中，OSD 将验证每个 CDB 中能力段信息的完整性，这种方法适用于存储系统所在网络是安全的情况下。在 CMDRSP 安全方法中，将对每个命令的 CDB、状态和检测数据的完整性进行验证，这种方法可以适用于存储系统所在网络不安全的情况下，能防止未授权的客户端对能力段的伪造、篡改或重放。ALLDATA 安全方法将验证 OSD 和客户端之间传输的所有数据的完整性，它以计算和验证大量的完整性校验值为代价来避免网络攻击，与那些不提供机密性安全的网络协议安全体系结构提供的安全保护相似。

对象存储系统的安全模型是一个基于凭证(Credential)的访问模型，其交互流程如图 2.3 所示。其中 SM 用来安全地存储长期密钥，与 PSM 协作、依据安全需求对客户端进行正确的访问控制。SM 的主要功能是为授权的应用客户产生凭证。凭证是一个包含相应能力段的数据结构，该能力段由 PSM 产生并受完整性校验值保护。SM 为每个凭证返回一个能力段键码。凭证给了应用客户访问指定 OSD 的权力。能力段键码使得应用客户和 OSD 可以用完整性校验值对它们之间传递的命令和数据进行验证。PSM 主要通过准备基于策略的能力段为客户端提供访问策略控制。每个 CDB 中均包含一个能力段，它与要处理的对象以及对象的操作命令单元相关联(比如，读、写、设置属性、检索属性)。

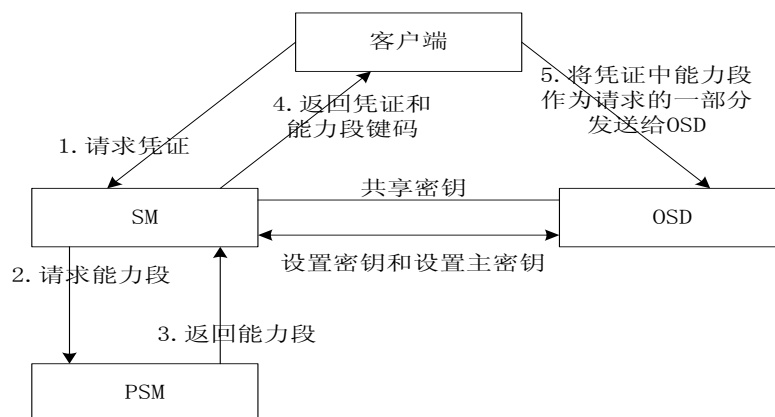


图2.3 安全模型的交互流程

2.1.3 对象存储设备

在对象存储系统中，OSD 负责对象数据的空间分配和空闲空间的管理。对象的属性也被存储在 OSD 上，它用来描述对象的特征信息。对象的属性使得 OSD 能够了解数据之间的关系，更好的组织数据，使其具有智能性。客户端可以直接与 OSD 进行对象数据传输；OSD 定期向 MDS 发送心跳信息，由 MDS 对存储系统中的 OSD 进行管理；SM 向 OSD 发送设置密钥和设置主密钥的命令，与 OSD 共享密钥。

对象存储设备软件模块如图 2.4 所示。iSCSI/OSD 解析负责 OSD 命令和数据的封装与解析，提供对象访问接口；安全认证验证请求是否合法；数据放置策略能够获取每个对象所在的 OSD 信息，便于对象的复制和迁移；对象文件系统管理对象和其属性的存储和访问，完成对象数据逻辑块到物理块的映射，当前能运行在 2.4 或 2.6 Linux 内核下；缓存用于提高对象数据和属性的访问速度；块设备驱动负责对磁盘块进行读写。

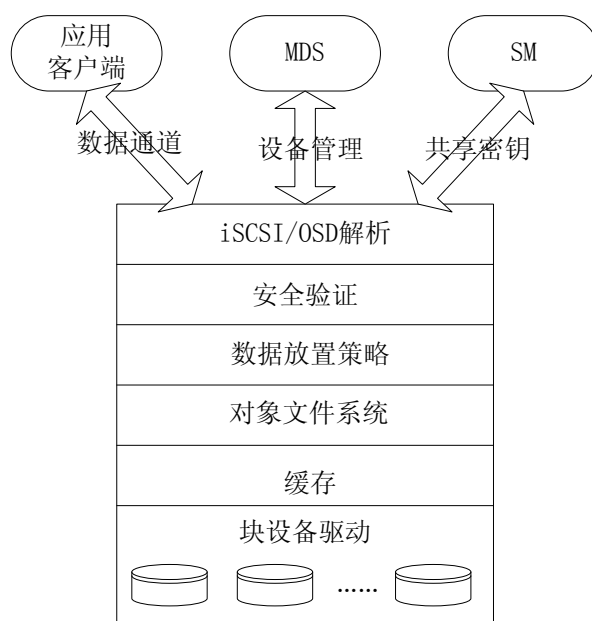


图2.4 对象存储设备软件模块

2.1.4 元数据服务器

对象存储系统中的 MDS 主要负责管理系统中用户组件部分的元数据，实现文件系统语义。它实现在用户态，支持多个客户端的并发访问，其软件构架如图 2.5 所示。

MDS 通过 Socket 接收来自客户端和其他 MDS 的元数据请求并对其进行处理，

同时也能将元数据请求通过 Socket 转发给其他 MDS。MDS 通过 iSCSI 实现对 OSD 的管理。由于单个 MDS 会成为系统性能瓶颈，采用 MDS 集群来提供元数据服务，负载均衡用来在 MDS 集群中的 MDS 之间均衡负载，以防止某个 MDS 成为系统中文件元数据访问的瓶颈，通过采用 MDS 集群提供高性能、可扩展的元数据服务。数据放置策略记录了文件到对象的映射策略，它也采用一定的策略在新对象被创建时为其选择合适的 OSD 存放，并且能够在对象被访问时定位到其所在 OSD 的位置。存储系统中元数据数量众多，且访问频繁，但每条元数据占用存储空间小，考虑到元数据访问性能需要将最近访问过的元数据缓存起来，元数据缓存结构在第三章中介绍。系统中的元数据和对象被多个客户端并发访问，锁服务为它们的一致性提供保证。后台元数据的存储采用 Berkeley DB，能为元数据处理提供很高的吞吐量。元数据描述了文件系统的组织结构，元数据的损坏将导致数据无法被访问到，利用对象存储系统中具有丰富语义的对象接口来保证元数据的高可靠性。

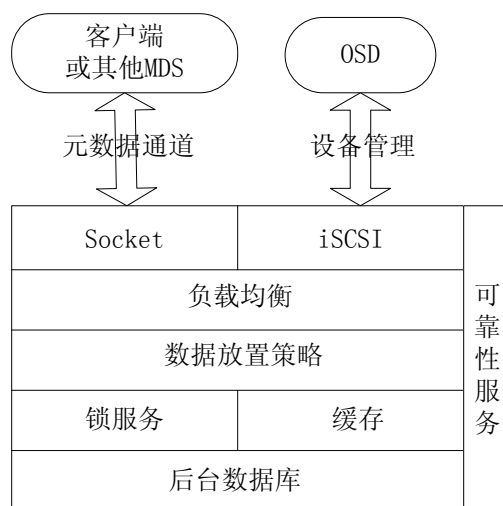


图2.5 元数据服务器软件构架

2.2 对象存储系统中元数据描述

在对象存储系统中，主要包含以下几种元数据：文件描述信息(也叫文件元数据)、目录转换元数据、文件到对象的映射元数据、对象到 OSD 的映射元数据、对象属性和对象元数据，这些元数据的分布如图 2.6 所示。由于本文中研究的重点不包括对象存储系统的安全性，因此并没有给出与对象安全有关的元数据描述。

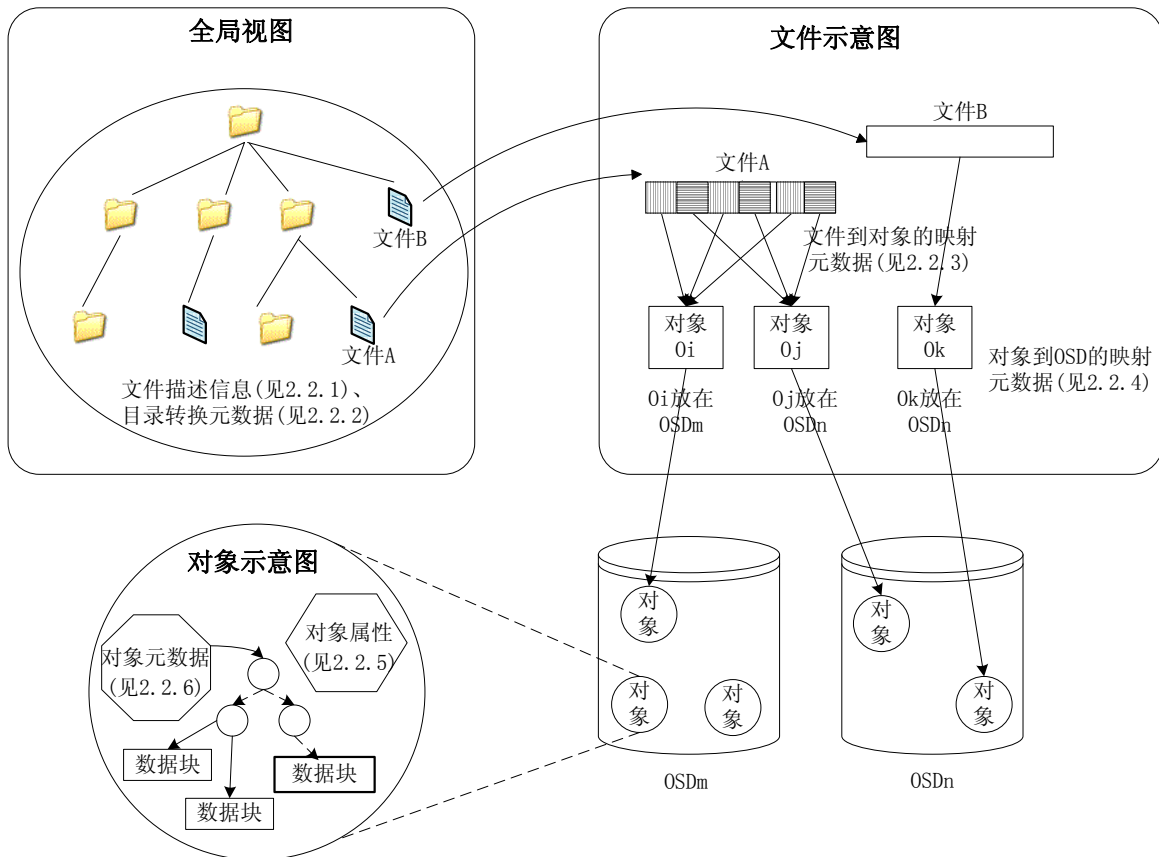


图 2.6 对象存储系统中元数据分布

2.2.1 文件描述信息

文件描述信息描述了文件的属性，它的主要内容包括标识该文件的全局标识符、文件的类型和访问权限、拥有者标识符、文件长度、最近修改时间等字段，即包含的内容与传统文件系统中除了数据块聚集信息之外的索引节点信息相似。在本文后面介绍中我们将其简称为文件元数据，它由 MDS 集中管理，是客户端要访问的主要元数据。

2.2.2 目录转换元数据

目录转换元数据主要包含目录的目录路径名，创建该目录统一分配的全局唯一的全局标识符。目录转换元数据的引入避免了传统方式中访问文件元数据时所需要的前缀目录的遍历，提高了对象存储系统中元数据的访问性能。它由 MDS 集中管理。

2.2.3 文件到对象的映射元数据

在对象存储系统中，以对象为单位将用户数据存储在 OSD 上。客户端要访问文件时，它需要知道该文件被如何映射成哪些用户对象，即文件到对象的映射元数据。该元数据由数据放置策略引入，它由 MDS 集中管理，与文件元数据存储在一起，用同一个文件元数据关键字标识。

数据放置策略根据不同的策略将文件映射为一个或多个用户对象。这一抽象层的引入使得文件到对象的映射具有灵活性。客户端能够根据自己的需求决定对文件采用何种映射方式。当客户端认为某个文件非常重要时，能够采用引入冗余数据的映射策略来保证对象数据的高可靠性，如采用 RAID1 或 RAID5 映射策略；当客户端认为文件访问中高聚合带宽比较重要时，它能够采用 RAID0 的映射策略将文件映射为多个用户对象，将每个对象放置到不同的 OSD 上，充分利用 OSD 之间的并行性。

由于文件的访问模式存在很大差别，客户端能在每个文件被创建时为其指定文件到对象的映射策略，若没有指定，则系统为新创建的文件采用默认的映射策略。由于文件到对象的映射信息与该文件元数据一起存放在 MDS 上，当应用客户端需要访问某个文件时，它首先访问 MDS 获取该文件的文件元数据以及该文件到对象的映射元数据(如采用何种映射策略以及其分片大小等信息)。

2.2.4 对象到 OSD 的映射元数据

在对象存储系统中，当客户端需要访问文件时，除了需要知道该文件被如何映射为哪些用户对象外，它还需要知道要访问的每个用户对象被存储在哪个 OSD 上，即对象到 OSD 的映射元数据。

根据数据放置策略的不同，用户对象到 OSD 的映射元数据既能被集中存储在 MDS 上，也能被分散存放在 OSD 上，也可以通过确定性的算法直接计算出(此时不需存储该元数据)。在对象存储系统中，若数据放置策略允许每个用户对象能被存放到任意的 OSD 上，则此时需要有映射元数据来记录对象与 OSD 之间的映射关系，采用这种方式使得对象的存储具有灵活性。当存储系统规模较小时，该映射元数据可集中存储在 MDS 上。当存储系统规模较大时，映射元数据的访问可能会成为 MDS 的性能瓶颈。为了维护对象分布的灵活性，同时为了避免集中式存放的瓶颈，可采用分散存放的方式在每个 OSD 上记录 OSD 上存放的对象信息。若为了节省掉该映射元数据占用的存储空间以及避免访问该映射元数据形成的性能瓶颈，可以将每个对象存放到确定的 OSD 上，采用确定性算法直接计算出对象所在的 OSD，确定性

算法为存储系统中任意节点已知，增加了系统访问的并行性。在对象存储系统中，根据系统的规模以及对象的不同特性采用不同的对象放置策略来决定对象到 OSD 的映射方式。

2.2.5 对象属性

相比传统的网络存储系统，对象存储系统的一个重要特色是为对象引入了属性，使得对象接口能提供比其他任何一种接口更为丰富的语义。对象属性用来记录与对象相关的一些元数据信息(例如对象大小，使用配额，与之相关的用户名信息等)，它与对象一起存放在 OSD 中。对象属性能够通过 OSD 命令为外界访问或设置，为对象提供了丰富的语义特性，通过对相应对象属性的设置或访问能够对存储系统的管理和性能作出优化。

对象属性以页的形式组织，同一属性页中的属性有相似的来源或用途，在每一个属性页中，不同属性由属性号来识别。属性条目由（属性页，属性号）来索引。对象的属性分为两种：一种是 OSD-2 标准中已经规定的；另一种是 OSD-2 标准为用户和其他标准的定义预留的。其中属性页号 80h~7FFFh 之间的属性页为保留的，8000h~EFFFh 之间的属性页可以给其他标准定义。用户可以通过扩展 OSD SCSI^[94] 协议中的属性页、利用这些自定义的属性定义一些属性来满足他们的需求。

2.2.6 对象元数据

在 OSD 中，逻辑结构被称为用户对象，它是一个虚拟实体，用于将客户端确定的逻辑相关的(部分)数据组织起来存储在 OSD 上；根对象、分区对象、汇集对象都为用户对象提供附加的定位帮助。在对象存储系统中，存储管理组件的下放增强了 OSD 的自我管理功能，使得主动存储能够在对象存储系统中得到更好的支持，增加了系统的高可扩展性、智能化等优势。

存储管理组件下移到 OSD 后，对象的空间管理功能由 OSD 负责，记录对象数据到 OSD 本地磁盘的数据块聚集信息的对象元数据由本地 OSD 的对象文件系统负责存储和管理，对外界不可见。

在当前的系统中，每个对象数据作为一个文件存储在 OSD 所在的 ext3 本地文件系统中，对象元数据即为该对象数据所对应文件的磁盘索引节点信息。数据块映射信息存放在对应文件的磁盘索引节点信息的 i_block 字段中^[95]。

2.3 对象存储系统中元数据管理框架

这里以研究对象存储系统元数据管理为出发点，涉及到的关键技术主要包括：分布式元数据管理、数据放置策略研究、元数据的可靠性研究。这几方面的相关技术综合起来形成了一个对象存储系统元数据管理框架，成为提供快速、高可靠性的元数据访问和高效管理对象的解决方案。

在对象存储系统中，文件元数据、目录转换元数据和文件到对象的映射元数据由 MDS 集中存储和管理。通过采用不同的对象放置策略，对象到 OSD 的映射元数据可能集中存放在 MDS 中或分散存放在 OSD 中或者通过确定性算法避免掉该元数据的维护。对象属性和对象元数据存储于 OSD 上。由于对象元数据对外不可见，后续不再对此进行关注。

2.3.1 分布式元数据管理

在文件系统中，虽然元数据占用的空间很小，但有统计表明 50%到 80%的访问都是元数据访问^[90]。在对象存储系统中，存储管理组件部分下移到 OSD，此时有关对象数据块的元数据的管理由 OSD 来负责，MDS 只需维护给用户提供一个全局唯一的树型逻辑结构视图(即树型目录结构和文件名列表)的名字空间，以及文件到对象的映射信息，相比 SAN 存储结构来说大大减少了 MDS 中需要管理的元数据的数量。由于对象存储系统是一个新型的系统，目前并没有研究给出在对象存储系统中元数据的访问特性以表明在对象存储系统中元数据访问是否仍然是整个系统的访问瓶颈，因此在我们研发的对象存储系统原型(当系统包含单个应用客户端，单个 MDS 和单个 OSD 时)中，采用 postmark^[96]工具(采用默认参数)测量了访问 MDS 中元数据的请求占总请求的百分比以更好的了解对象存储系统的 MDS 中元数据的访问特性，其结果如图 2.7 所示。其中当访问 1KB 大小的文件时，发现访问 MDS 中元数据请求占总访问请求的 65%^[97]。在大规模对象存储系统中，频繁的元数据访问仍会形成系统访问瓶颈，需要采用元数据服务器集群提供高性能的元数据访问，研究高效的分布式元数据管理方案对整个存储系统提供高性能和高可扩展性都至关重要。

提出一种分布式元数据管理方案来有效处理不同的元数据负载。它采用四种技术来提供高性能和可扩展的元数据服务，具体包括仿层次目录结构、目录转换元数据、针对不同类型元数据的访问特性提供不同的在 MDS 集群中灵活分布元数据的方法以及高效的元数据存储方式。

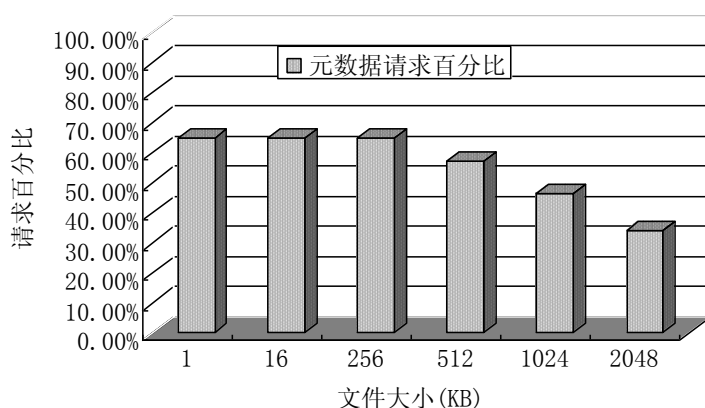


图2.7 对象存储系统中元数据请求分布

2.3.2 数据放置策略

放置策略负责将文件映射为用户对象，并为新创建的用户对象选择 OSD 存放，在需要访问用户对象时能够定位到对象所在的 OSD，通过该策略，每个文件被分布到一个或多个 OSD 上。这一抽象层的引入为文件到对象的映射以及对象到 OSD 的映射提供了映射的灵活性。数据放置策略引入了两种元数据：文件到对象的映射元数据和对象到 OSD 的映射元数据。

为获得存储系统的高聚合带宽和高可用性，为每个文件选择合适的放置策略非常重要。根据放置策略，文件数据和为了提高数据可用性引入的冗余数据被分配到多个 OSD，通过 OSD 的并行工作来提供高吞吐量。文件到对象的映射通常有两种方式，一是每个文件被映射为一个用户对象存储在单个 OSD 上，由于每个文件的大小和访问频率差别很大，可能会导致 OSD 之间不能有效负载均衡，也不能充分利用 OSD 之间的并行性。二是每个文件被映射为多个用户对象，可能是条分或线性划分等方式。条分是指将每个文件按照分片大小、分配数据布局等选择参数划分为不同的用户对象(类似 RAID0 或 RAID5 等)，它适用于需要通过多个 OSD 聚合 IO 带宽的文件应用。线性划分是指将文件顺序切割为多个用户对象，用索引段来指明用户对象之间的顺序，用户对象以该顺序串联起来即能形成该文件，它适用于顺序访问模式的文件。另外还有一种在实际应用中不太常见的文件到对象的映射模式：将多个相关的小文件映射为单个用户对象存放在一个 OSD 上。这种方式的采用需要明确多个文件之间的访问关系，否则会浪费不必要的带宽，影响系统的性能。由于每个文件的访问模式都可能不同，客户端能在每个文件被创建时指定其文件到用户对象的映射策略，若没有指定，则系统为新创建的文件采用默认的映射策略。

文件到对象的映射元数据与文件元数据一起存放在 MDS 中，当客户端需要访

问某个文件时，它首先访问 MDS 获取该文件的文件元数据和文件到对象的映射元数据。根据数据放置策略的不同，对象到 OSD 的映射元数据既能被集中存储在 MDS 上，也能被分散存放在 OSD 上，也可以通过确定性的算法直接计算出。

当存储系统规模不同时，可能需要的数据放置策略也不同。在存储系统规模较小、OSD 总数固定的情况下，可能需要尽力寻求存储系统性能的近似最优解。此时每个对象可能存储在任意的 OSD 上，需要维护对象到 OSD 的映射元数据。由于对象存储系统规模较小，该映射元数据能集中存放在 MDS 中。

在大规模对象存储系统中，OSD 数量成百上千，存储规模可能会经常变化(如为了提供更好的性能，可能会周期性的添加一批新的 OSD 到存储系统中，而且设备失效属于常规事件)，系统服务的文件总数可能是数亿计的，此时若仍然将对象到 OSD 的映射元数据集中存放在 MDS 中会形成系统访问瓶颈。由于系统中的大部分的存储空间被数量较少的大对象占据，少量的存储空间被数量众多的小对象占据，而且小对象面临着更多的访问和更频繁的创建和删除操作。这些特性决定了能够采用分布式算法来决定小对象所在的 OSD，由此消除了大量的对象到 OSD 的映射元数据，提高了系统的可扩展性。大对象仍然能被存放在任意的 OSD 上，选择负载较轻的 OSD 来存放大对象有利于系统的负载均衡，避免某个 OSD 成为系统访问瓶颈，同时由于大对象的数量较少，对象到 OSD 的映射元数据数量也较少，它们分散存放在 OSD 中。

2.3.3 元数据可靠性

在对象存储系统中，MDS 是联系客户端与 OSD 的桥梁，只有 MDS 记录了用户对象的文件和目录信息，一旦 MDS 由于软硬件或网络故障失效，存储系统就不能够提供存储服务，即使此时所有的 OSD 和存储在其上面的对象数据都完整无缺。MDS 对于整个存储系统来说至关重要。元数据仅仅保存在单个 MDS 上容易形成单点失效，因此需要寻求一种方法来提供高可靠的元数据服务。

对象存储系统的一个重要特色是为对象引入了属性，使得对象接口能提供比其他任何一种接口更为丰富的语义。对象属性用来记录与对象相关的一些元数据信息，为对象提供了丰富的语义特性，通过对相应对象属性的设置或访问能够对存储系统的管理和性能作出优化。

提出利用对象存储系统的富有表达力的对象接口设计采用扩展属性页来提高元数据可靠性的方法能够充分利用对象存储的优势保障元数据的高可靠性，而且它不排除其他的提高存储系统元数据可靠性的方法，为提供更高的元数据可靠性提高了很好的补充。

2.4 本章小结

本章首先概括性地介绍了自行设计的对象存储系统，描述了该对象存储系统的体系结构，以及它所包含的各个组成部件——客户端、策略/存储管理器(PSM)与安全管理器(SM)、对象存储设备(OSD)、元数据服务器(MDS)——的主要软件框架。

由于对象存储系统相比于传统网络存储在体系结构上发生了一定的变化，使得元数据在系统中的分布和管理也发生了一定的改变，给出了元数据在对象存储系统中的分布，并对元数据进行描述。

当对象存储系统包含单个 MDS，单个 OSD 和单个客户端时，测量了访问 MDS 中元数据的请求与访问 OSD 中对象请求分别占总请求的百分比以更好的了解对象存储系统的 MDS 中元数据的访问特性，结果发现当文件小于 1MB 时，元数据请求占总请求的 40% 以上。由于系统中包含数量众多的小文件，在大规模对象存储系统中，频繁的元数据访问会形成系统访问瓶颈，需要采用 MDS 集群提供高性能的元数据访问，提出一种分布式元数据管理方案在 MDS 集群中提供高性能、可扩展的元数据服务。针对对象存储系统的不同规模和对象的不同特性提出不同的数据放置策略，提供高性能的对象访问，兼顾系统的可扩展性和对象分布的灵活性。提出利用对象存储系统的富有表达力的对象接口设计采用扩展属性页来提高元数据可靠性。与现有的对象存储系统相比，本文提出的元数据管理框架具有分布式元数据管理、高效数据放置、元数据具有高可靠性的特点。

3 分布式元数据管理

在大规模对象存储系统中，频繁的元数据访问会形成系统访问瓶颈，MDS 集群用来提供高性能的元数据访问。针对元数据负载的访问特性，提出一种分布式元数据管理方案在 MDS 集群中提供高性能、可扩展的元数据服务。

3.1 仿层次目录结构方案总体设计

最近几年里，已经提出了大量的分布式元数据管理方案，它们大致可以归为两类：子树分割和哈希。子树分割方案将全局名字空间分割为不相交的层次子树，每个层次子树由一个 MDS 来负责管理；哈希方案计算文件标识符(如文件路径名)的哈希值来定位存储该文件的 MDS。这两种方案均不能有效满足实际系统的需求。子树分割方案在每次访问文件元数据时导致了对该文件的所有前缀目录的遍历，引入了额外的元数据访问开销；而且由于每个子树下包含的文件数以及这些文件的访问频率可能差别很大，导致元数据负载并不能有效的在 MDS 集群之间均衡分布。在哈希方案中当 MDS 集群规模发生改变或者请求元数据重命名操作时涉及到大量的文件元数据在 MDS 之间的迁移。

在基于目录路径的元数据管理^[44]中，引入目录路径索引项来避免子树分割方案中的目录遍历和哈希方案中的重命名目录操作引起的大量元数据的迁移，目录路径索引项被集中存放在一个目录路径索引服务器中，它可能会形成访问瓶颈。每个目录中的文件元数据被绑定为一个或多个桶来由单个 MDS 来管理，当需要对同一个目录下的文件进行创建或删除或访问操作时，为了维护目录下不允许存在同名文件的语义而引入的锁机制和在桶内的线性查找会大大减少元数据访问性能。

不同应用对元数据服务有不同需求，如科学计算应用要求高吞吐量，而一些特定应用可能对容错和安全的元数据服务更感兴趣，旨在提出一种有效的元数据服务方案来满足大多数应用的目标。这些目标包括：要访问的元数据能被快速取回同时吞吐量能随 MDS 数的增加而近似线性增加，由 MDS 规模改变或重命名操作导致的元数据在 MDS 之间的迁移要尽可能小。设计一个能满足这些目标的分布式元数据管理方案具有很大的挑战，构建这样的 MDS 集群需要考虑很多事情，比如说如何在 MDS 集群间均匀分布负载同时避免某个 MDS 成为系统性能瓶颈。

针对现有方案的优缺点，提出了 MHS(Mimic the Hierarchical directory Structure, 仿层次目录结构)分布式元数据管理方案^[98]。它能快速访问文件元数据，在能够有效处理重命名操作和 MDS 集群规模改变操作的同时也能在 MDS 集群间均匀分布负

载，它也能够同时支持数以万计的元数据访问。

在 MHS 中，除了采用与目录路径索引项类似的目录转换元数据来避免子树分割方案中的目录遍历和哈希方案中的重命名目录导致的大量元数据迁移之外，MHS 还采用以下的几个技术来提供高性能和可扩展的元数据服务。

每条文件元数据被作为一条记录(record)存储在 Bekeley DB^[9]中(数据库能提供高处理事务吞吐量)，层次目录结构(即，维护每个目录中包含的文件和子目录的文件名与其对应索引节点号映射关系的目录数据)不再存储在持久存储(如磁盘)中，这样避免了层次目录结构自己成为热点，采用一种间接的方案来模仿层次目录结构。由于每种元数据有不同的访问特性，因此对他们采用不同的分割方法将其分布到不同的 MDS 中来提供高性能的元数据访问和提高 MDS 集群的扩展性。对目录转换元数据按照记录关键字的字典序分割，对文件元数据采用元数据查找表(metadata lookup table ,MLT)来分割。

MHS 的总体设计如图 3.1 所示，其包含组件的描述在后文中详细说明。

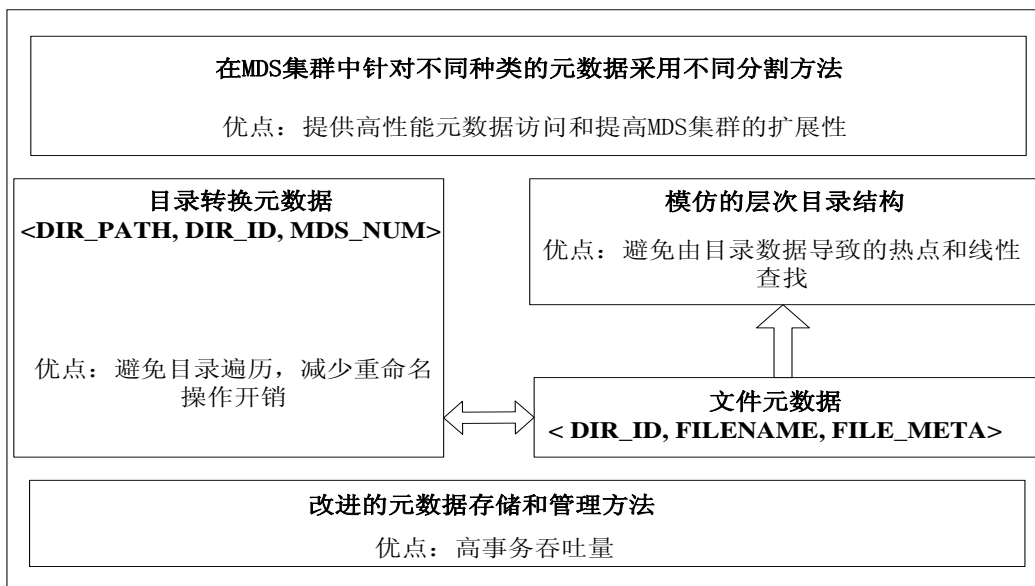


图3.1 MHS总体设计

3.2 元数据存储

系统的用户组件中主要维持两种元数据：文件描述信息(一些用户可见的信息，如修改时间、文件长度等，它包含在文件的索引节点信息中)；目录数据中的目录项(目录数据维护名字空间的目录层次结构，它包含的每项目录项列出该目录下包含的文件或子目录的文件名与其索引节点号的对应关系)。

由于同一个目录中可能会同时存在大量的创建或删除文件或子目录的活动，这时目录数据会形成热点，并且目录数据内的线性查找也会形成系统性能瓶颈。在常规文件系统访问中，又需要提供一个该目录包含的文件和子目录的视图给客户端，因此采用一种间接的方案来模仿目录层次结构，该方法的有效性在后面的性能评估中得到验证。

在 MHS 中，提供全局名字空间的用户组件中除了包含文件元数据(指文件描述信息)以外，还包含目录转换元数据。目录转换元数据用于将文件的父目录直接转换成全局唯一的目录标识符(DIR_ID)，文件元数据用其父目录的 DIR_ID 和它自己的文件名来标识和定位。目录标识符与在创建该目录对应的文件元数据时统一分配的全局标识符相等。它能减少子树分割方案中沿着文件路径的遍历，也能避免哈希方案中重命名导致的文件元数据在 MDS 集群中的重新分布。

由于数据库能够有效处理不同格式的元数据访问，并能提供高处理事务吞吐量，因此除了被存储在 MDS 内存之外，文件元数据和目录转换元数据都按照记录关键字的字典序永久存储在 Berkeley DB 中。为了避免层次目录结构成为访问热点，它仅仅通过内存中数据结构来模仿，并不被永久存储在磁盘中。

3.2.1 元数据种类

MHS 中提供全局名字空间的用户组件主要包含两种元数据：目录转换元数据和文件元数据(文件到对象的映射元数据与文件元数据有同样的关键字，与文件元数据存放在一起，本章不再对文件到对象的映射元数据做说明)。它们中均包括有访问控制列表(access control list, ACL)字段。MHS 中访问权限的生成与处理采用与 LH^[43] 相同的方式以避免检查文件访问权限引起的目录遍历，即每个目录(或文件)均有两个 ACL，一个是路径权限，另一个是文件权限，其中路径权限由其文件权限与其父目录的路径权限相与操作计算得到。论文后续不再考虑访问权限的问题。

● 目录转换元数据：

目录转换元数据用来将目录路径名转换为全局唯一的目录标识符，目录转换元数据格式为<DIR_PATH, DIR_ID, MDS_NUM>，其中 DIR_PATH 为目录转换元数据的关键字，其他内容为目录转换元数据的值项。DIR_PATH 表示目录路径名，DIR_ID 是当该目录被创建时统一分配的全局唯一的全局标识符，它也被存储在目录的文件元数据的值项 FILE_META 中。全局标识符可以用来唯一标识一个文件或目录，文件包含的用户对象的标识符(128 位，由分区标识符和用户对象标识符组成)就是根据文件的全局标识符来生成的，目录转换元数据中的目录标识符实际上是该目录的文件元数据的全局标识符的副本。MDS_NUM 用来标明单个目录下的文件元数据由

一个或多个 MDS 管理。当 MDS_NUM 为 1 时，该目录中的文件元数据由单个 MDS 来管理，说明该目录比较小并且该目录中的元数据访问也不太频繁；当目录中的文件数目增加或它包括的文件元数据被频繁访问时，它们能分布到多个 MDS 上由多个 MDS 来管理。在这种情况下，单个目录中的文件元数据被存储在 MDS_NUM 个连续的 MDS 上，第一个 MDS 通过 DIR_ID 来定位。该方法能避免其他的采用主从 MDS 的方法来解决热点问题而引起的元数据一致性开销。

- 文件元数据：

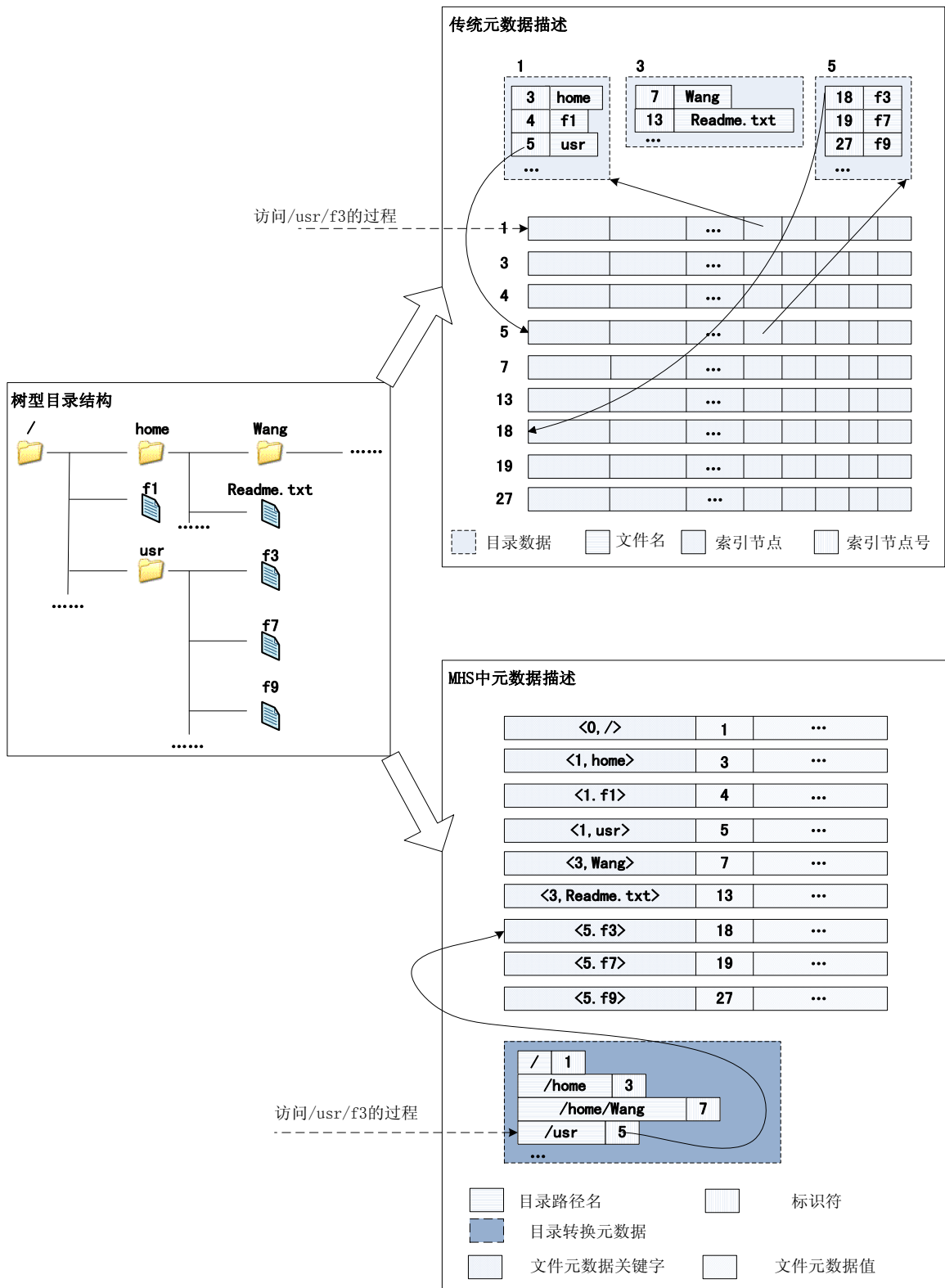
文件元数据格式为 < DIR_ID, FILENAME, FILE_META >，其中 < DIR_ID, FILENAME > 为文件元数据的关键字，FILE_META 为文件元数据的值项。DIR_ID 是要访问目录或文件的父目录的全局唯一的目录标识符；FILENAME 是该目录或文件的文件名；FILE_META 是该目录或文件包含的文件元数据的值，用于记录用户可见的目录或文件信息，如文件长度、修改时间等，它内部包含的一个重要字段是在创建该目录或文件时分配给它的全局唯一的 128 位的全局标识符，全局标识符已经在前面给出过介绍。

采用这种方式将文件元数据与其对应的目录项(dentry)紧耦合的存储在一起，当在一个目录中查找某个文件时，该文件元数据也一同返回。避免了将目录项与文件元数据分开存放导致的多次磁盘 IO，提高了元数据的访问性能。

- 实例说明：

图 3.2 给出传统文件系统元数据的描述方式与 MHS 中元数据的描述方式的对比说明。

假设用户要访问文件 “/usr/f3” 的元数据，在传统文件系统元数据访问中，它需要先访问根目录的索引节点信息得知根目录数据的数据块映射信息，然后根据该信息访问根目录数据中相应的目录项从中得知 user 文件名对应的索引节点号为 5，根据该信息再访问对应的索引信息得知 user 的目录数据的数据块映射信息，然后再根据该信息访问 user 的目录数据中相应的目录项从中得知 f3 文件名对应的索引节点号为 18，最后访问索引节点号为 18 的索引节点信息获得 /usr/f3 的元数据信息。为了最终访问到 /usr/f3 的元数据，中途涉及到 2 次访问前缀目录的索引节点信息和 2 次访问前缀目录的目录数据的操作，即总共需要 5 次读操作。



目录标识符为“5”，然后访问关键字为“<5,f3>”的文件元数据。在 MHS 中，为了最终访问到/user/f3 的文件元数据，中途仅仅涉及到 1 次查找目录转换元数据操作，即总共需要 2 次读操作。

由上述分析可知，当访问目录深度为 n 的文件或目录的文件元数据时，采用传统文件系统元数据的描述方式中途需要访问 n 次前缀目录的索引节点信息和 n 次前缀目录的目录数据最终才能获取要访问文件或目录的索引节点号，然后根据该文件或目录的索引节点号最终访问到该文件或目录的文件元数据，即需要 $2n+1$ 次读操作；当采用 MHS 中元数据的描述方式时，仅需先访问一次目录转换元数据获取要访问文件的父目录的目录标识符，再根据父目录的目录标识符与该文件的文件名最终访问到文件元数据，即仅仅需要两次读操作。

3.2.2 仿层次目录结构

在大多数存储系统中，通常采用层次目录结构来支持 `readdir` 等目录操作。在这种情况下，当一个新文件被创建（或删除）时，不仅要创建（或删除）该文件的文件元数据，也需要修改该文件的父目录的目录数据来表明该目录现在包括该新创建的文件或不包括已经删除的文件，而这两个操作可能需要在不同的 MDS 上完成(如采用哈希方案时，文件和其父目录很大可能位于不同的 MDS 上；采用子树分割时，由于分割粒度的影响，文件和其父目录也可能位于不同的 MDS 上)，因此当单个目录下的创建活动较多时，会引入较大的一致性开销和网络开销。因此在 MHS 中，不再存储目录数据，而采用一种间接的方法来模仿目录层次结构。

在访问文件元数据时，采用文件元数据关键字<DIR_ID, FILENAME>来标识要访问的文件元数据，其中 DIR_ID 是其父目录的目录标识符，同一目录下的文件元数据有着相同的 DIR_ID。Readdir 操作可通过访问与要访问的目录有相同目录标识符(DIR_ID)的所有文件元数据来实现(例如，假定目录“/usr”的目录标识符为 5，当要 readdir 该目录(`ls -l /usr`)时，将返回 DIR_ID 为 5 的所有文件元数据)，同时它们形成 B 树结构缓存在元数据缓存中并且这个 B 树结构由该 DIR_ID 来索引。文件元数据缓存结构如图 3.3 所示。在 MHS 中该结构仅仅被存储为内存数据结构，通过当某个目录下的文件被访问时或者当第一次通过 readdir 访问一个目录时取回该目录内所有文件元数据来形成。

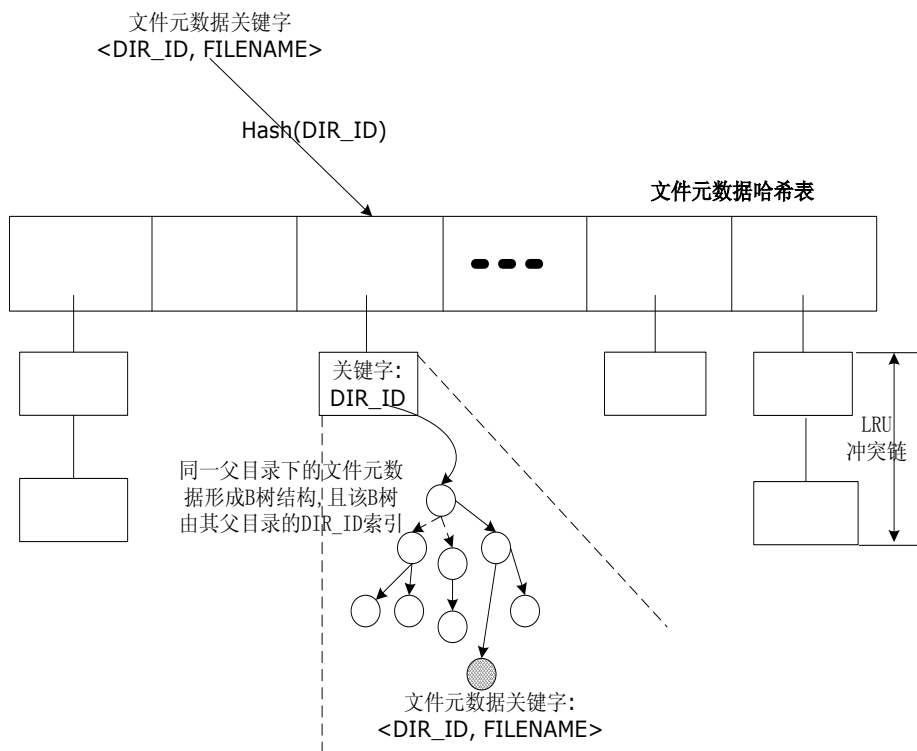


图 3.3 文件元数据缓存结构

3.3 元数据分布

不同种类的元数据有不同的访问模式，如目录转换元数据涉及到的绝大多数操作都是读操作，只有在新创建一个目录或删除一个目录或重命名一个目录时，才会修改对应的目录转换元数据。通过 trace 研究^[100]可以发现，在文件系统中目录的创建或修改或重命名操作相对于文件元数据操作来说少之又少。而文件元数据的访问模式则相对较为复杂，它随时间变化差别很大。因此在 MDS 集群中对不同元数据采用不同的分割方式将其分布到不同的 MDS 上来均衡负载和提高可扩展性以提供高性能的元数据访问。

3.3.1 目录转换元数据的分布

目录转换元数据按照记录关键字的字典序分割以保证有相似前缀路径的目录被存放在相邻的位置，这样当重命名目录操作发生时能减少网络开销。

将一组临近(按照记录关键字字典序)的目录转换元数据形成一个桶(bucket，桶是 MDS 管理目录转换元数据的一个单元)。

采用桶索引表来记录每个桶的开始条目和存储该桶的 MDS 的标识符 (MDS_ID), 该索引表被存储在每个 MDS 上, 当有新的客户端加入到存储系统中时, 该桶索引表将会载入到该客户端上。

只有在桶的个数发生变化时, 才会涉及到桶索引表的修改操作。当一个桶中包含的目录转换元数据的条目数超过 MAX(桶的最大条目数)或者小于 MIN(桶的最小条目数)时, 桶才会分裂或者合并。目录转换元数据的负载均衡守护进程负责在当一个大桶的总条目数超过 MAX 时将该大桶分割为两个小的子桶并将其中的一个子桶迁移到一个较轻负载的 MDS 上, 或者在当两个相邻的桶中其中一个桶的条目数小于 MIN 并且这两个桶的条目数的总和小于 MAX 时将它们合并到一个更大的桶中。可采用延迟更新策略来处理目录转换元数据的迁移。当一个大桶被分割为两个小的子桶时, 需要在桶索引表中增加一个条目用来标识新桶的加入, 并且采用一条日志记录来说明新桶的目录转换元数据原来存储在 MDS_i 上, 它们应该被迁移到 MDS_j 上。属于该新桶的新创建的目录转换元数据应该在 MDS_j 上创建。当改变发生后, 新桶内的目录转换元数据第一次被访问时, 可能会在 MDS_i 上找到它, 然后将需要将其迁移到 MDS_j 中同时从 MDS_i 中删除; 若不能在 MDS_i 中找到它, 则将在 MDS_j 中查找。桶的改变造成访问延迟增加的情况很少, 因为经常访问的目录转换元数据能在客户端缓存。当新桶的迁移被完成时, 该日志记录将被删除。两个小桶合并为一个大的桶的情况与上述情况类似。

目录转换元数据的条目大小与目录路径名的长度有关, 对文件元数据快照的研究^[11]表明, 名字空间树中目录层次深的文件毕竟是少数, 也间接说明具有长路径名的目录是少数, 即使按照每个目录的路径名长度为 256 个字节计算, 一个目录转换元数据条目的大小也小于 0.3KB, 当一个桶包含有 10^5 个目录转换元数据条目时, 其占用空间也小于 30MB。因此在实现中将 MIN 设置为 10^5 , MAX 设置为 2×10^5 。由于每个桶包含的条目数很多, 而且目录的修改操作(如创建、删除或重命名目录)很少, 因此桶索引表的修改操作非常非常少, 大部分都是读操作, 因此维护桶索引表一致性的开销非常少。

在 MDS 集群内分割目录转换元数据的例子如图 3.4 所示。

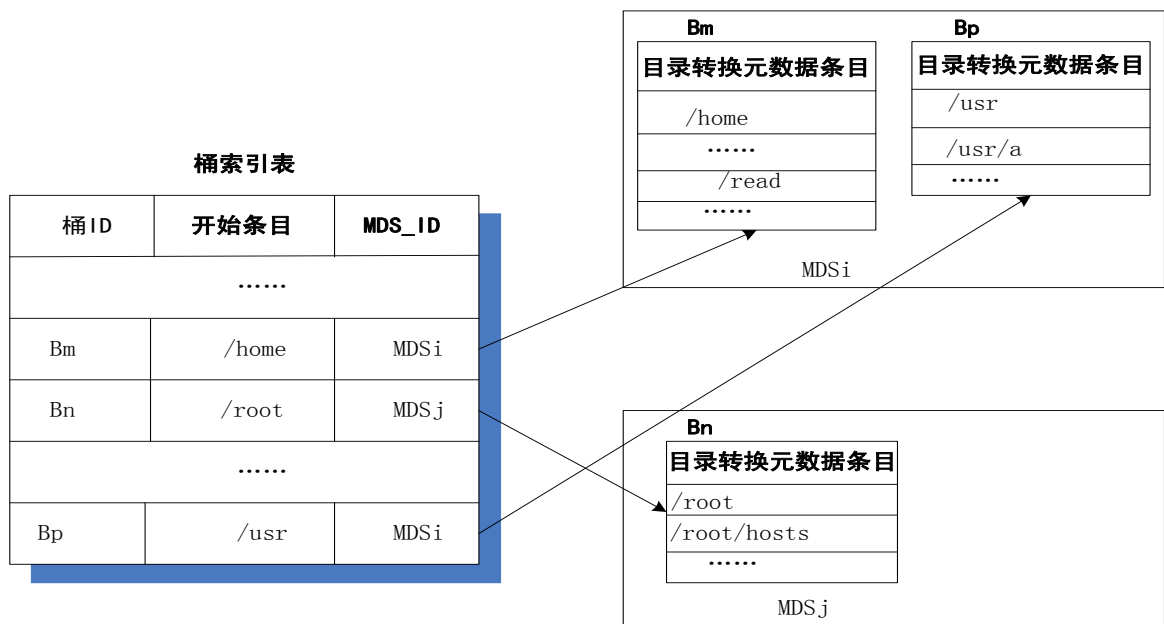


图 3.4 在 MDS 集群内分割目录转换元数据的例子

3.3.2 文件元数据的分布

类似于 LH，MHS 也采用元数据查找表(Metadata Lookup Table, MLT)在 MDS 集群中将文件元数据映射到不同的 MDS。文件元数据关键字中的 DIR_ID (即该文件父目录的目录标识符)的哈希值用来作为 MLT 的索引。重命名某个目录时，由于重命名目录下文件和子目录的 DIR_ID 并没有发生改变，因此并没有文件元数据发生迁移。MDS 集群个数发生改变时，需要修改 MLT 中一个或多个条目来将当前 MDS 集群中某些 MDS 的部分元数据迁移到新加入的 MDS 上以达到 MDS 集群内的负载的重新均衡，这时会招致一些文件元数据迁移。元数据查找表的更新被广播到所有的 MDS 上。由于 MDS 集群规模改变的情况很少，元数据查找表中由于集群规模改变导致的文件元数据的迁移也很少。元数据查找表的一个例子如表 3.1 所示。

表 3.1: 元数据查找表

哈希值索引	条目 ID	目标 MDS
0 ~ 99	0	MDS1
100 ~ 199	1	MDS0
200 ~ 299	2	MDS1
300 ~ 399	3	MDS2
.....

也考虑过采用一些确定性的算法来在 MDS 集群之间分布文件元数据，比如一致性哈希^[88]或 SIEVE^[76]等等。这些确定性的算法具有直接计算定位的特性，它们也继承了简单哈希算法的均匀分布的优点，同时它们也避免了简单哈希算法的缺陷：不能有效处理 MDS 规模的改变。实际上，确定性算法缺乏元数据分布的可调性和灵活性，虽然它们不需要在内存中存储文件元数据和 MDS 之间的映射关系，但是它们可能会导致负载的不均衡。由于 MDS 集群中 MDS 数量并不多，并且 MDS 集群中规模变化并不频繁，因此采用 MLT 会有比较小的存储开销和更新开销，但是它却可以提供很好的负载均衡和灵活的元数据分布，以及当 MDS 集群规模发生改变时能保证近似最小数量的文件元数据迁移。因此在 MHS 中最终采用了 MLT 来在 MDS 集群内不同 MDS 中分布文件元数据，它也能被缓存到客户端来提高文件元数据的访问性能。MLT 和确定性算法的优缺点的对比如表 3.2 所示。

表 3.2: MLT 与确定性算法优缺点对比

	元数据查找表(MLT)	确定性算法
分布灵活性	灵活	确定，不灵活
是否能处理热点	能	不能
存储映射的内存开销	与映射条目数成正比	无
集群规模变化导致的映射更新开销	与更新映射条目数成正比	无

3.4 元数据负载均衡

由于文件元数据的读写访问非常频繁，并且不同目录下或不同文件的访问热度差别很大，为了在 MDS 集群中提供高性能、可扩展的元数据服务，需要在 MDS 之间负载均衡，以防止某个 MDS 成为系统中文件元数据访问的瓶颈^[101]。

由于文件元数据的大小很小，请求频率很高，在此采用文件元数据请求的响应时间来衡量一个 MDS 的负载情况。

3.4.1 静态负载均衡

在系统运行初期，通过采用将文件元数据关键字中的 DIR_ID 的哈希值对当前 MDS 集群中 MDS 总数(MDS_NUM)求模的方式将文件元数据均匀分布到不同的 MDS 中。

设 MDS 集群的集合为 $M(n)$, MDS 的个数为 $\|M(n)\| = N$;

文件元数据到 MDS 的映射算法如下:

(1) $Val = Hash(DIR_ID)$;

(2) $Local(Val) = Val \% N = m$; 并且 $m \in M(n)$;

其中, DIR_ID 代表该文件的父目录的目录标识符, $Hash$ 代表一个哈希函数, 将 DIR_ID 映射为固定区间内的一个值 Val ; $Local$ 代表一个映射函数, 将哈希值为 Val 的文件元数据存放到 MDS 集群 $M(n)$ 中序号为 m 的 MDS 中。由于选择的哈希函数 $Hash$ 是伪随机均匀分布的, 映射函数 $Local$ 是均匀分布的, 所以文件元数据在 MDS 集群中是均匀分布的。

由表 3.1 可知, MLT 的每个条目中包含有如下两项: 文件元数据的哈希值索引、标识存储文件元数据的目标 MDS。在静态负载均衡中, 哈希值索引项的内容为 Val , 目标 MDS 项的内容为 m 。

3.4.2 动态负载均衡

由于系统为不同的应用提供服务, 应用负载随时都可能发生较大的改变致使每个目录下包含的文件数以及每个文件元数据被访问的频率(热度)差别很大, 使得 MDS 之间负载不均衡。通过判断 MDS 集群中每个 MDS 的请求响应时间与整个 MDS 集群的平均响应时间的差异来判断 MDS 负载是否均衡。当 MDS 集群中 MDS 间负载不均衡时, 应该采用动态负载均衡, 将负载重的 MDS 上的部分文件元数据复制或转移到负载轻的 MDS 上, 并根据动态负载均衡的结果在 MLT 中将迁移的文件元数据对应的条目中的目标 MDS 项的内容由迁移前所在 MDS 对应的 MDS_ID 更新为迁移后所在 MDS 的 MDS_ID 。

● MDS 的请求响应时间

假定序号为 x 的 MDS 对文件元数据请求的响应时间为 $R(x)$ ($x \in M(n)$);

一般认为, 每个 MDS 可以采用 M/G/1 排队模型来建模^[102]。根据排队论可得 MDS_x 对文件元数据请求的响应时间为:

$$R(x) = \frac{\lambda_x \times E(T_x^2)}{2 \times (1 - \rho_x)} + E(T_x) \quad (3.1)$$

其中:

$$\lambda_x = \sum_{k \in A(x)} \lambda_{xk} ; \quad \rho_x = \sum_{k \in A(x)} \lambda_{xk} \times T_{xk} ;$$

$$E(T_x) = \sum_{k \in A(x)} p_{xk} \times T_{xk} = \sum_{k \in A(x)} \frac{\lambda_{xk}}{\lambda_x} \times \frac{L_{xk}}{B_x};$$

$$E(T_x^2) = \sum_{k \in A(x)} p_{xk} \times T_{xk}^2 = \sum_{k \in A(x)} \frac{\lambda_{xk}}{\lambda_x} \times \left(\frac{L_{xk}}{B_x}\right)^2.$$

公式中的符号定义如下所示。

T_x : MDS_x 上的文件元数据请求服务时间;

λ_x : MDS_x 上的文件元数据请求到达率;

ρ_x : MDS_x 上的访问强度;

B_x : MDS_x 的处理能力;

$A(x)$: MDS_x 上收到的文件元数据请求的集合;

λ_{xk} : MDS_x 上的文件 k 的元数据请求到达率;

T_{xk} : MDS_x 上的文件 k 的元数据请求服务时间;

p_{xk} : MDS_x 上的文件 k 被访问的概率;

L_{xk} : MDS_x 上的文件 k 的文件元数据长度。

由于 MDS 中文件元数据是结构化数据, 大小相等, 同时假设每个 MDS 的处理能力都相同, 则每个 MDS 对每个文件元数据的请求服务时间都相同, 即对

$\forall x \in M(n), k \in A(x)$ 有 $T_{xk} = \frac{L_{xk}}{B_x} = C$ (C 为常数); 则公式(3.1)最终可化简为:

$$R(x) = \frac{\lambda_x \times C^2}{2 \times (1 - \lambda_x C)} + C \quad (3.2)$$

其中, 该 MDS 集群的平均访问强度为:

$$\rho(R) = \frac{1}{N} \sum_{i=1}^N \rho_i \quad (3.3)$$

该 MDS 集群的平均响应时间为:

$$E(R) = \frac{1}{N} \sum_{i=1}^N R(i) \quad (3.4)$$

该 MDS 集群中的响应时间方差为:

$$D(R) = \frac{1}{N} \sum_{i=1}^N (E(R) - R(i))^2 \quad (3.5)$$

● MDS 动态负载均衡的算法

在系统中，MDS 之间的负载均衡属于完全分布式的。MDS 节点之间需要周期性的交换它们相互的负载信息。

算法 3.1

动态负载均衡算法

1. $L_MDS \leftarrow \Phi$; /* 负载较轻的 MDS 的集合 */
 2. $W_MDS \leftarrow \Phi$; /* 负载过载的 MDS 的集合 */
 3. **for** $i \leftarrow 1$ **to** N **do**
 4. **if** $DM(i) \geq DM_{ALARM}$ **then**
 5. $W_MDS \leftarrow W_WDS \cup \{i\}$
 6. **else**
 7. **if** $DM(i) < 0$ **then**
 8. $L_MDS \leftarrow L_WDS \cup \{i\}$
 9. **end if**
 10. **end if**
 11. **end for**
 12. **while** $W_MDS \neq \Phi$ **and** $L_MDS \neq \Phi$ **do**
 13. $MDS_i \leftarrow \max(\{DM(x) | x \in W_MDS\})$ ， 找一个子集 $M_MDS \subseteq L_MDS$ ， 满足如下条件：

$$\left| DM(i) + \sum_{j \in M_MDS} DM(j) \right| \leq \left| DM(i) + \sum_{k \in SL_MDS} DM(k) \right|, \forall SL_MDS \subseteq L_MDS$$
 14. **for** $MDS_j \in M_MDS$ **do**
 15. 将 MDS_i 中的部分文件元数据传输到 MDS_j 上； MDS_i 和 MDS_j 之间传输的文件元数据集合的总的请求到达率大约为：
$$\frac{2 \times (E(R) - C)}{(2 \times E(R) - C) \times C} - \lambda_j$$
 16. **end for**
 17. $W_MDS \leftarrow W_WDS - \{i\}$
 18. $L_MDS \leftarrow L_WDS - M_WDS$
 19. **end while**
-

MDS 动态负载均衡的目标是使每个 MDS 对文件元数据请求的响应时间相差不大， 本文用下面的函数来衡量 MDS_x 的负载不均衡度：

$$DM(x) = R(x) - E(R) \quad (x \in M(n)) \quad (3.6)$$

函数 $DM(x)$ 是指 MDS_x 请求响应时间与 MDS 集群的平均响应时间的差异， 它的大小反映了 MDS_x 负载的均衡性。

首先定义一个负载差阈值 DM_{ALARM} ，在 MDS 集群中，如果 $\exists \forall x \in M(n)$ 使得 $DM(x) \geq DM_{ALARM}$ 时，则将在 MDS 上启动动态负载均衡算法，其描述如算法 3.1 所示。

在算法 3.1 中，并不是对超过平均响应时间的每个 MDS 都启动动态负载均衡，而是当其值超过一定的阈值 DM_{ALARM} 时才启动，这样可以减少文件元数据在 MDS 之间频繁的迁移，而只是在必要的时候才迁移。启动动态负载均衡后，从负载过载最多的 MDS_i 开始，将 MDS_i 中过载部分的文件元数据按照一定的比例分配到合适的负载较轻的 MDS 中，最终使得系统中的 MDS 负载均衡。

● 算法仿真结果

编写仿真程序验证上述负载均衡算法。其中假设 MDS 集群中 MDS 的个数 $N=4$ ，系统中文件元数据请求服从泊松到达。为了模拟文件元数据请求的重负载，仿真程序在 MDS 集群中不断创建新的文件元数据，文件元数据一旦创建后就留在 MDS 中并被客户端访问。随着文件元数据请求的增多， $MDS_x(x=1, 2, 3, 4)$ 上的访问强度 ρ_x 也不断增加，仿真程序一直运行直至 MDS 集群的平均访问强度 $\rho(R)$ 至 0.9 (因为每个 MDS 负载可能不均衡，当系统平均访问强度为该值时，实际上重负载的 MDS 已经超过了该值并几乎接近 1)。设 MDS 集群中被访问的文件元数据个数为

$\sum_{x=1}^N A(x) = X$ ，被访问的文件元数据 $f(i)$ 的请求到达率为 $\lambda_{f(i)}$ ，为了保证 MDS 集群的平均访问强度不大于 0.9，则 $\rho(R) = \frac{1}{N} \sum_{i=1}^N \rho_i = \frac{1}{N} \sum_{i=1}^N (C \times \sum_{k \in A_x} \lambda_{ik}) \leq 0.9$ 即

$\sum_{i=1}^X \lambda_{f(i)} \leq 0.9 \times \frac{N}{C}$ 。假设每个 MDS 中的每个文件元数据的服务时间为： $C =$

0.02ms，则由 4 个 MDS 组成的 MDS 集群中要保证被访问的总的文件元数据到达率

$\sum_{i=1}^X \lambda_{f(i)} \leq 0.9 \times (2 \times 10^5)$ (请求/秒)。

首先来看一下，在采用静态负载均衡后，MDS 集群中 MDS 响应时间情况。从图 3.5(a)和(b)中可以看出，在每个文件元数据请求到达率相同的情况下，采用静态负载均衡的系统的每个 MDS 的平均响应时间差别微小，MDS 响应时间方差的值很小。

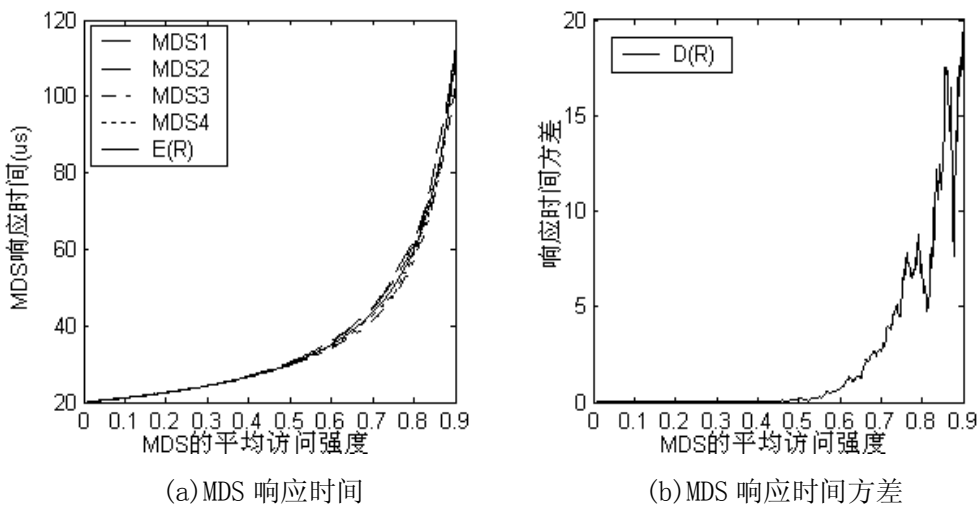


图3.5 元数据请求到达率相同情况下，静态负载均衡时的MDS的响应时间与响应时间方差

为了更好地说明问题，在下面的仿真中，假定按照静态负载均衡，文件元数据是完全均匀分布在 MDS 集群中的。下面给出在采用静态负载均衡时，当应用负载发生变化导致文件访问热度不同时，MDS 集群中 MDS 的响应时间情况。假定文件元数据请求到达率服从 Zipf 分布^[64]（一般认为，媒体文件的访问概率服从该分布），则在该负载下的结果如图 3.6 所示。

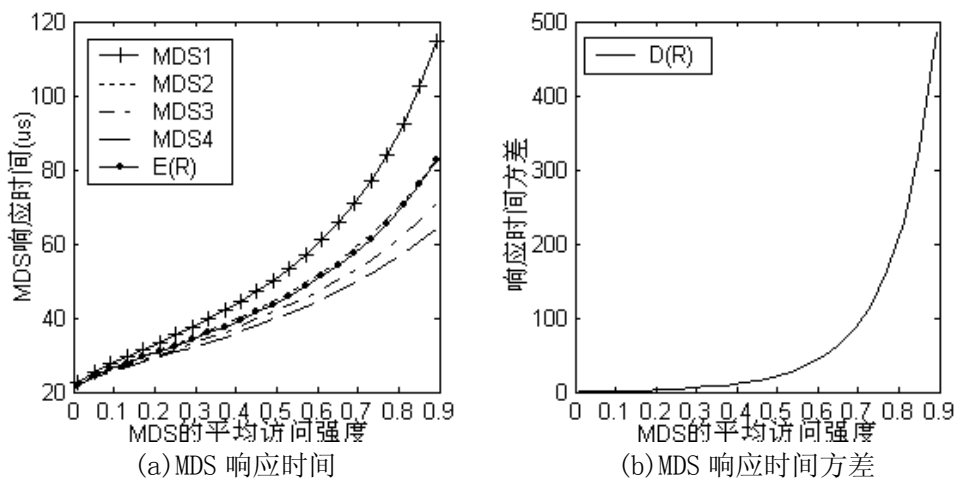


图3.6 元数据请求服从zipf分布时，MDS的响应时间与响应时间方差

由此可见，在采用了静态负载均衡后，虽然文件元数据被均匀分布在不同的 MDS 上，然而在文件元数据访问热度相差较大的情况下，每个 MDS 上的负载差别比较大，导致 MDS 之间的响应时间差别很大。这个时候需要引入动态负载均衡，将负载重的 MDS 上的部分文件元数据转移到负载轻的 MDS 上。

在仿真程序中分别设置负载差阈值 DM_{ALARM} 为 3us、5us、7us、10us 时，采用

动态负载均衡时的 MDS 集群的响应时间分布如图 3.7 所示。

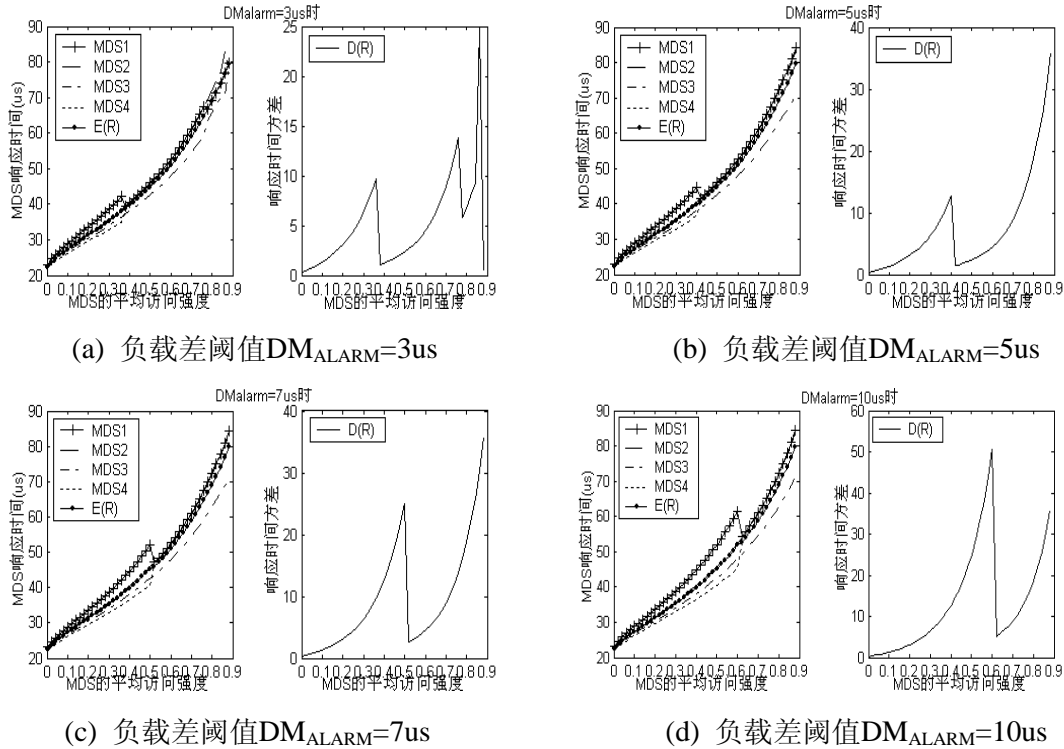


图3.7 元数据请求服从zipf分布时，动态负载均衡的结果

由图 3.7 可以看出负载差阈值 DM_{ALARM} 的设置决定了 MDS 之间的均衡度，当 DM_{ALARM} 较小时，MDS 集群中的响应时间方差较小，MDS 之间负载比较均衡，MDS 之间的迁移次数就比较多。当 DM_{ALARM} 较大时，MDS 集群中的响应时间方差较大，MDS 之间的迁移次数较少。因此要合适的选择一个 DM_{ALARM} 的值，作者最终选取了 $DM_{ALARM}=10\mu s$ 。综上所述，对比图 3.6 可以看出，在文件元数据访问频率差别很大的情况下，启动动态负载均衡算法能很大程度地减少 MDS 集群中的 MDS 的响应时间的差别，大大减少系统的响应时间方差。

3.5 访问流程

MHS 能提供有效的元数据服务。现以图 3.8 中的访问文件元数据 $/home/Wang/paper.doc$ 为例说明其访问流程。若客户端没有缓存 $/home/Wang$ 的目录转换元数据，它将最先查找本机中保存的桶索引表来获取存储该目录转换元数据 $/home/Wang$ 的 MDS 的 MDS_ID(假设是 MDS_i)和桶号 B_m ；然后客户端将文件 $/home/Wang/paper.doc$ 元数据请求发送给 MDS_i ， MDS_i 在其本机上 B_m 桶中查到该目录转换元数据 $/home/Wang$ 的 DIR_ID 是 7 并且 MDS_NUM 是 1， MDS_i 通过查找

本机上的 MLT 得知保存文件 <7, paper.doc> (即/home/Wang/paper.doc) 元数据的目标 MDS 为 MDS_j ; 最后, MDS_i 将元数据请求转发给 MDS_j , MDS_j 对客户端作出响应。文件的 DIR_ID 用来定位存储该文件元数据的 MDS, 在客户端不知道要访问文件的父目录的目录标识符的情况下取回文件元数据需要 1.5 个网络来回。经常被访问的目录转换元数据和 MLT 能被缓存在客户端, 在这种方式下, 大多数请求能被直接定位到正确的 MDS, 取回文件元数据仅仅需要单个网络来回。定位文件元数据不需要沿着目录路径进行目录遍历。

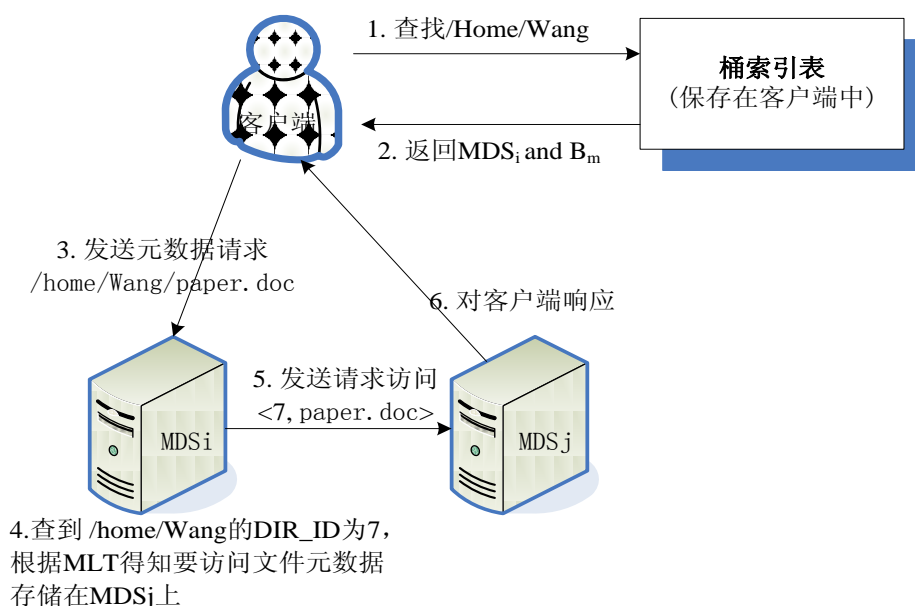


图3.8 MHS中文件元数据访问过程

3.6 元数据一致性

客户端发送的元数据请求的非原子性引入了 MDS 中的元数据一致性的问题, 元数据的一致性决定着存储系统的可靠性和可用性, 因此本节关注在 MHS 方案中如何保证元数据的一致性。

由于客户端发送到 MDS 的元数据请求通常是由一系列原子性的元数据操作组成, 每个元数据操作只修改一条元数据, 因此构成该元数据请求的一系列原子性的元数据操作是一个不可分割的整体, 即一个元数据请求类似于数据库中的一个事务, 需要尽力保证它的 ACID(Atomicity、Consistency、Isolation、Durability)特性^[103]。即: 一个元数据请求包括的所有原子性元数据操作要么全部完成要么全部不完成; 每个元数据请求开始前文件系统处于一致性状态, 每个元数据请求被完成后都能保证文件系统又处于一致性状态; 当有多个元数据请求被 MDS 同时服务时, 任意两

个元数据请求的执行是互不干扰的，MDS 上的结果是按照某种顺序串行服务这些元数据请求的结果；一旦元数据请求被 MDS 服务完成，则该元数据请求造成的修改被永久的存储在存储系统中。

在传统的采用目录数据来维护名字空间的目录层次结构的方案中，当创建文件时，其涉及到的原子性的元数据操作主要包括创建新文件的元数据，修改该新文件的父目录的目录数据的内容使其包含新创建文件的目录项来表明该目录现在包括该新创建的文件。常用的顺序有两种：(1)先创建该文件的文件元数据再在其父目录中增加相应的目录项；(2)先在其父目录中增加相应的目录项再创建该文件的文件元数据。当采用(1)的顺序时，若在该文件元数据创建之后、父目录中添加相应目录项之前机器宕机，则会导致系统中存在很多孤儿文件元数据，可能需要采用类 fsck 工具来将这些孤儿文件元数据清理掉；若当采用(2)的顺序时，若在父目录中添加相应目录项之后、该文件元数据创建之前机器宕机，则会导致要访问不存在的文件元数据，这个错误更严重些，所以在大部分系统中通常都是采用(1)的顺序来创建文件。

在 MHS 中，由于不再保存目录数据，文件元数据的关键字由其父目录的 DIR_ID 和该文件的文件名组成，其关键字已经能明确反映出其父目录包含的孩子中是否存在某个名字的文件，因此在创建文件或删除文件时，仅仅只需要创建或删除一条文件元数据。该类元数据请求只涉及到单条元数据，避免了由于创建文件或删除文件元数据请求导致出现元数据不一致的可能。相比传统的采用目录数据来维护名字空间的目录层次结构的方案，大大减少了为了维护元数据一致性带来的开销。在 MHS 中涉及到单条元数据的请求还有：获得文件属性 getattr，设置文件属性 setattr，读目录数据 readdir 等。

在 MHS 中，为了避免目录遍历导致的低性能，引入了目录转换元数据来将目录路径名直接转换为全局唯一的目录标识符，虽然元数据访问性能得到了提升，但是由于每个目录都有相应的目录转换元数据和文件元数据(目录转换元数据为目录独有，文件没有相应的目录转换元数据)，目录转换元数据与文件元数据可能位于不同的 MDS 中，当创建目录或删除目录或重命名目录时，元数据请求涉及到需要对目录的文件元数据和目录转换元数据同时操作(多条原子性的元数据操作)，下面给出保证文件元数据与目录转换元数据一致性的方法。

在 MHS 中，当创建目录/d0/d1/d2 时，假定其父目录/d0/d1 的 DIR_ID 是 7，要为目录/d0/d1/d2 分配的 DIR_ID 是 9，其涉及到的原子性的元数据操作主要包括创建新目录/d0/d1/d2 的文件元数据(其关键字为<7, d2>)和创建新目录/d0/d1/d2 的目录转换元数据(/d0/d1/d2->9)。常用的顺序有两种：(1)先创建该目录的文件元数据再创建该目录的目录转换元数据；(2)先创建该目录的目录转换元数据再创建该目录的

文件元数据。若采用顺序(2)时,在该目录的目录转换元数据创建之后、该目录的文件元数据创建之前机器宕机,该目录的创建不成功但是却遗留下多余的目录转换元数据。此时若用户想继续在该目录下创建子目录或子文件,虽然该目录的元数据不存在(返回给客户端的响应是该目录创建不成功),但由于其目录转换元数据存在,导致 MDS 以为其存在,因而导致其子目录或子文件的创建是成功的。若采用顺序(1)时,若在该目录的文件元数据创建之后、该目录的目录转换元数据创建之前机器宕机,则认为该目录的创建是成功的。此时若用户想继续在该目录下创建子目录或子文件,则会发现该目录的目录转换元数据不存在,则可以通过目录遍历的方法沿着该目录的路径名来检查该目录的文件元数据是否存在,若存在则将该目录转换元数据添加到相应的 MDS 中即可。其中(1)(2)两种处理顺序描述如图 3.9 所示,对应的例子导致的两种元数据的改变如图 3.10 所示。从本质上说采用顺序(1)是由文件元数据比目录转换元数据更重要、目录转换元数据可以通过文件元数据逐层遍历生成的原因来决定的。与创建目录类似,删除目录最终采用的顺序是:先删除目录转换元数据再删除目录的文件元数据。

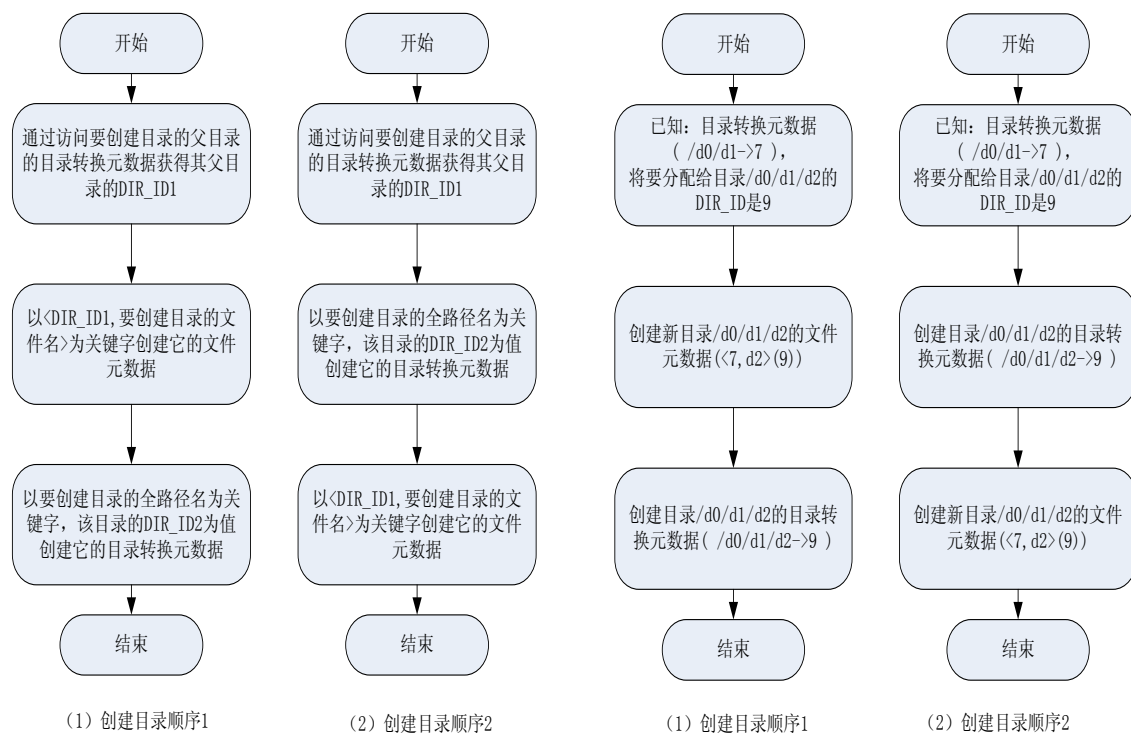


图3.9 创建目录的两种顺序

图3.10 创建目录的两种顺序的例子

若当由于创建或删除目录导致某个目录的文件元数据存在而其目录转换元数据不存在时,可以通过遍历文件元数据的方法来生成期望的目录转换元数据。当系

统长期运行后，也可以通过该方法来检查系统中目录转换元数据与文件元数据的一致性，并通过此方法进行恢复。其中当创建目录失效后的恢复处理方法如图 3.11 所示。

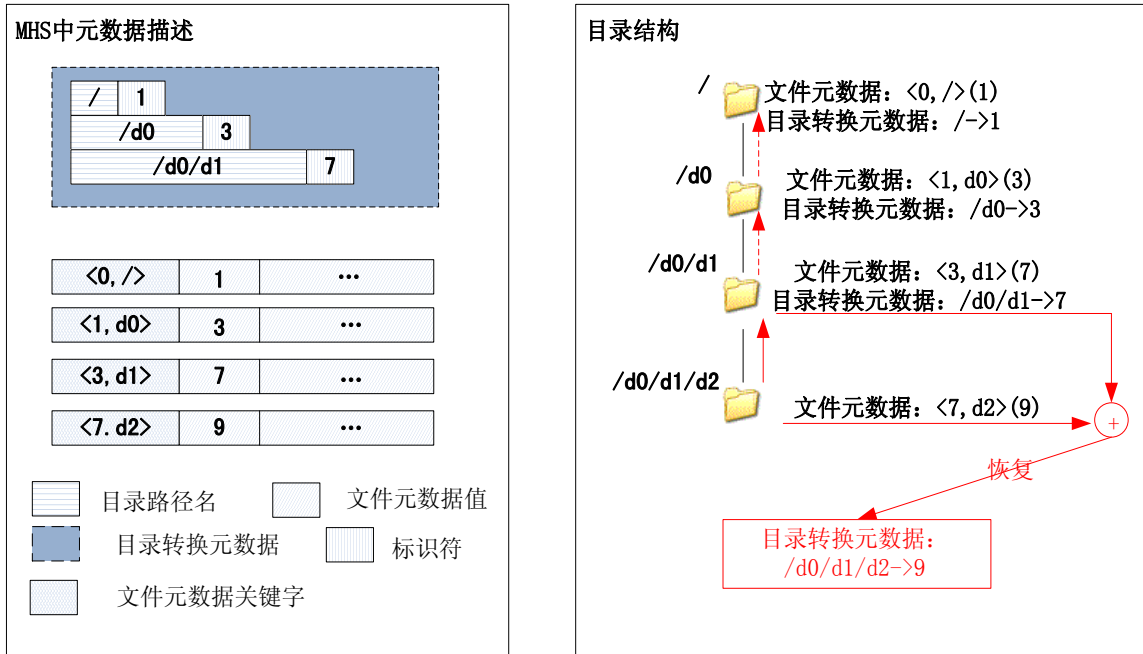


图 3.11 两种元数据不一致时目录转换元数据恢复实例

假定机器在创建目录/d0/d1/d2 的中途出现故障，导致其文件元数据已创建而目录转换元数据没有创建，则系统中元数据描述如图 3.11 左图所示，此时在目录转换元数据中缺少条目/d0/d1/d2->9。当用户想继续在目录/d0/d1/d2 下创建子目录或子文件时，如在创建目录/d0/d1/d2/d3 时，其首先会查找其父目录/d0/d1/d2 的目录转换元数据以确定父目录的 DIR_ID(为构成/d0/d1/d2/d3 的文件元数据的关键字作准备)，这时发现目录转换元数据/d0/d1/d2 并不存在，此时确定该目录/d0/d1/d2 是否在系统中存在的方式如下：

- (1). 查找/d0/d1/d2 的父目录/d0/d1 的目录转换元数据是否存在，若存在则读取该目录转换元数据以获得/d0/d1 的 DIR_ID(在该例子中，其存在且值为 7)，并继续执行下一步(2)；若不存在，则继续沿路径名向上一层父目录遍历，查找其目录转换元数据是否存在(即向上递归进行)；
- (2). 由于/d0/d1 的 DIR_ID 为 7，则/d0/d1/d2 的文件元数据的关键字为<7, d2>，以其关键字在系统中查找该关键字是否存在，若存在则继续执行步骤(3)；若不存在，则说明在系统中不存在/d0/d1/d2；
- (3). 访问关键字为<7, d2>的文件元数据的 FILEMETA 得知其确实为一个目录，且其全局标识符为 9，则由此得知该目录/d0/d1/d2 的目录转换元数据应该为

/d0/d1/d2->9; 则添加目录转换元数据条目/d0/d1/d2->9。

由于文件元数据比目录转换元数据更重要，目录转换元数据可以通过该目录所在路径上的文件元数据逐层遍历生成，上文给出了当文件元数据与目录转换元数据不一致时如何对目录转换元数据进行修复以恢复它们之间一致性的例子。

上述操作中仅仅涉及到单条文件元数据的修改，对于重命名操作则会涉及到多条文件元数据的修改。在传统的采用目录数据来维护名字空间的目录层次结构的方案中，重命名操作主要涉及到四条元数据的修改(删除旧文件名元数据、更新新文件名元数据、更新旧父目录的目录数据、更新新父目录的目录数据)。在 DCFS2^{[104][105]}中，出于简化元数据处理协议和提高元数据处理性能考虑，考虑到文件系统操作中重命名操作所占比例非常小，因此其提出的一致性协议并不对此类操作进行处理。

在 MHS 中，重命名操作涉及到的文件元数据有两条：旧文件名的文件元数据和新文件名的文件元数据。涉及到文件元数据的操作顺序可能有两种：(1)先删除旧文件名的文件元数据，再使用旧文件名的文件元数据的值更新新文件名的文件元数据；(2)先使用旧文件名的文件元数据的值更新新文件名的文件元数据，再删除旧文件名的文件元数据。当采用顺序(2)时，如果执行中途机器突然宕机，此时新文件名的文件元数据已经被更新而旧文件名的文件元数据没有被删掉，会导致旧文件名和新文件名它们拥有相同的全局标识符，假定旧文件名类型是目录，由于其没有被正常删除，则会导致仍然能在旧目录下面创建子文件或子目录，而由于新目录下也能创建子文件名和子目录，则会导致两个不同目录下的孩子有相同的父目录 DIR_ID，在用户通过 READIR 想读取某个目录下的子文件和子目录列表时，也会同时显示出另一个目录下的子文件和子目录列表。当采用顺序(1)时，如果执行中途机器突然宕机，此时旧文件名的文件元数据被删除，而新文件名的文件元数据的全局标识符还没有被更新为旧文件名的标识符，若旧文件名类型是目录，则会导致树型目录层次的断裂，即不能通过逐层遍历访问文件元数据的方式访问到旧目录下面的子树。顺序(1)(2)都导致了严重的一致性问题，因此对于重命名操作必须采用日志来解决文件元数据的一致性问题。

3.7 性能评估

为了评估 MHS 方案的性能，将其在对象存储系统中实现，同时也实现了 LH 方案，以用来与 MHS 方案做性能对比。选择 LH 方案来与 MHS 方案做性能对比的原因是：相对子树分割方案来说，哈希方案能够更好的在 MDS 集群之间均衡负载，而在哈希方案中 LH 方案是公认的优秀方案。采用详细的 micro-benchmark 来评估在不同操作中的元数据访问性能。

在以下的实验中，每个层次目录子树下嵌套目录的深度和子目录数分布是基于文章^{[11][106]}提出的概率生成模型来生成的，该概率生成模型与在 2001 年到 2004 年之间 4 年中从上万个文件系统中获取的元数据快照中的目录深度和子目录数分布几乎完全匹配。该概率生成模型的基本思想是：当新创建一个子目录时，选择已经存在的目录 d 作为该新创建目录的父目录的概率与 $C(d)+2$ 成正比，其中 $C(d)$ 是目录 d 当前包含的子目录数。其中当每个客户端创建一个包含有 8900 个目录(数字 8900 的选择也是来源于该论文，它是 2004 年每个文件系统包括目录数的算术平均值)的层次目录子树时其目录深度的统计情况如下表 3.3 所示，其中该子树总共包含 18 级目录，目录主要集中在第 3~9 级。

表 3.3 模型生成的包含 8900 个目录的层次子树的目录深度统计情况

目录深度	目录数量	占目录总数百分比(%)
0	1	0.011%
1	44	0.494%
2	271	3.045%
3	668	7.506%
4	1086	12.202%
5	1463	16.438%
6	1555	17.472%
7	1311	14.730%
8	1049	11.787%
9	695	7.809%
10	397	4.461%
11	202	2.270%
12	84	0.944%
13	47	0.528%
14	21	0.236%
15	4	0.045%
16	1	0.011%
17	1	0.011%

实验中每个 MDS 和客户端均分别运行在如表 3.4 所述的硬件配置的设备上。

表 3.4 硬件配置平台

主板	Super-Micro SUPER X6DH8-XB
处理器	Intel 604-pin EM64T Xeon 3.0GHz
内存	512MB PC2700 DDR-SDRAM
磁盘	Maxtor 6L200MO 7200RPM SATA 150 8MB buffer 200GB
操作系统	Linux 2.6.11-1.1369_FC4
网络	Gigabit Ethernet
交换机	Cisco Catalyst 3750 series Gigabit Ethernet Switch

3.7.1 MHS 方案与 LH 方案性能比较

首先对 MHS 方案和 LH 方案在创建目录、创建文件和读文件元数据操作中的元数据性能进行了比较,每次元数据的更新同步刷新到数据库中,测试结果如图 3.12 所示。在该测试中,最先创建大量的目录(mkdir),然后在每个存在的目录下创建文件(create),最后对每个文件执行读文件元数据操作(Read)。为了避免最初缓存对性能的影响,测试出真实性能,总是在每种操作测试前将服务重启以保证测试前缓存为空。

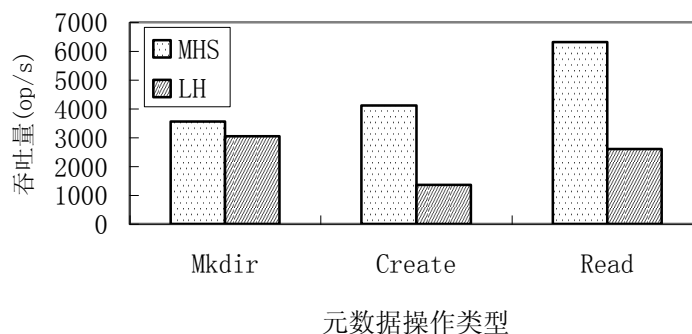


图 3.12 MHS 方案和 LH 方案的吞吐量比较

从图 3.12 中可以发现, MHS 方案中创建目录操作的吞吐量比 LH 方案中的略高一些。这是因为在 MHS 中创建一个新目录时,既要创建该目录的文件元数据,也要创建目录转换元数据;在 LH 中,当创建一个新目录时,既要创建该目录的文件元数据,也要修改其父目录的目录数据以添加新创建目录对应的目录项。因此 MHS 和 LH 对创建目录操作有相似的性能。由于在 MHS 中目录转换元数据的锁粒度是单条记录,而 LH 中目录数据的锁粒度是整个目录数据, LH 中锁粒度相对较大,更容易形成性能瓶颈,因此在创建目录操作中 MHS 的性能优于 LH。

MHS 在创建文件时，仅需要创建文件元数据，因此其创建文件性能要优于创建目录性能。在 LH 中，创建文件操作除了创建文件元数据外还需修改其父目录的目录数据以添加新创建文件对应的目录项，因此 LH 中创建文件性能与创建目录性能相似，但由于在实验中，创建文件操作发生在创建目录操作之后，目录数据中的查找开销随该目录下的文件和子目录数的增加而增加，因此在图中创建文件操作的性能低于创建目录操作的性能。

当执行文件元数据读操作时，MHS 和 LH 都只需要访问文件元数据。由于在 MHS 中，当访问一个目录下的某个文件元数据时，该目录下的所有文件元数据都被预取到缓存中，它充分利用了访问局部性原理；而在 LH 中，同一目录下的文件元数据被哈希后分散存放，丧失了局部性，因此 MHS 中读操作的性能要远高于 LH。实际上，在 MHS 方案中，在到达 MDS 服务能力的饱和值之前，每个元数据读操作的平均用户响应延迟随每个目录中包含的文件数的增加而减少，由于空间的限制这里没有给出它的详细性能。

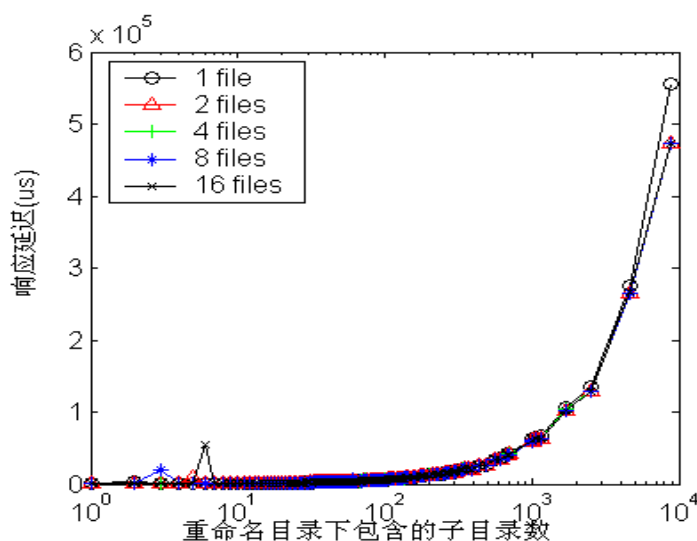


图 3.13 MHS 方案中重命名目录的用户响应延迟

下面给出 MHS 方案在重命名操作中的性能。图 3.13 给出了在每个目录中分别包括 1、2、4、8、16 个文件的不同情况下，重命名层次目录子树中一个目录的用户总响应延迟。在该实验中，接近根节点的最大目录中包括了 8900 个子目录，该层次目录子树中所有目录转换元数据属于同一个桶的概率为 91.1% 到 95.6% (由于每个桶的条目数在 $[10^5, 2 \times 10^5]$ 区间变化)。这里测量了这 8900 个子目录的目录转换元数据属于同一个桶的情况下的延迟，从测试结果中发现，重命名目录操作的响应延迟仅仅与该重命名目录下的子目录数相关，而与该重命名目录下的文件数无关。重命名包含 8900 个子目录的近根大目录的最大响应延迟小于 0.6s。

MHS 中重命名目录操作的性能依赖于相关的目录转换元数据所在桶的分布而与该目录下包含的文件元数据的分布无关，因此当重命名目录下相关目录转换元数据所在桶的分布确定时，其重命名操作的性能与 MDS 集群中 MDS 总数无关。在该实验中，元数据被立即更新来统计重命名目录的真实开销。当然，这里也能采用延迟元数据更新技术来将更新开销隐藏在后续的元数据操作中，甚至当目录转换元数据过期时，仍然能够通过目录遍历的方法来获取正确的文件元数据信息。

这里并没有以图的方式给出 LH 方案中的重命名操作性能，这是因为 LH 方案采用了元数据无效和延迟元数据更新技术来延缓由于重命名一个目录而导致的操作开销，它将操作开销分摊到后续的操作中，因此该开销不易评估，但是它却不能减少元数据的修改和迁移开销。当目录被重命名时，MHS 的开销与重命名目录下的子目录数相关，LH 的开销与重命名目录下所有子目录和文件数相关，由于目录数远小于文件数，因此 MHS 的开销远小于 LH 的开销。

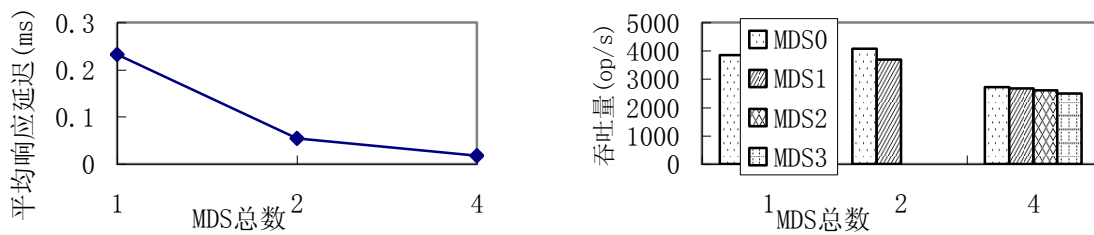
从上述比较中可以发现，MHS 方案优于 LH 方案。前面仅仅在单 MDS 的情况下比较和分析了 MHS 和 LH 的性能，这是因为 MHS 和 LH 的主要差别在于元数据的种类的不同，单 MDS 的情况足以说明它们的差别。MHS 和 LH 均采用 MLT 技术将文件元数据映射到 MDS 集群内不同的 MDS 中，MDS 集群中 MHS 和 LH 的性能对比与单 MDS 中类似。

3.7.2 MHS 方案的可扩展性

在 3.7.1 节中，给出了 MHS 方案和 LH 方案的性能比较，论证了 MHS 方案的有效性。在本节中，将测试和分析 MHS 方案在不同 MDS 集群规模下的元数据性能，验证 MHS 方案的可扩展性。

● MDS 吞吐量分布

这里首先给出在不同 MDS 集群规模情况下每个 MDS 的吞吐量分布。图 3.14 给出了创建目录负载下的测试结果。图 3.14(a)和(b)中分别给出了每个元数据请求的用户平均响应延迟和相应 MDS 的吞吐量。从图中可以看出，当元数据请求总数固定时，随着 MDS 数的增加用户平均响应延迟快速减少，并且在 MDS 集群中的每个 MDS 的吞吐量都大致相同。

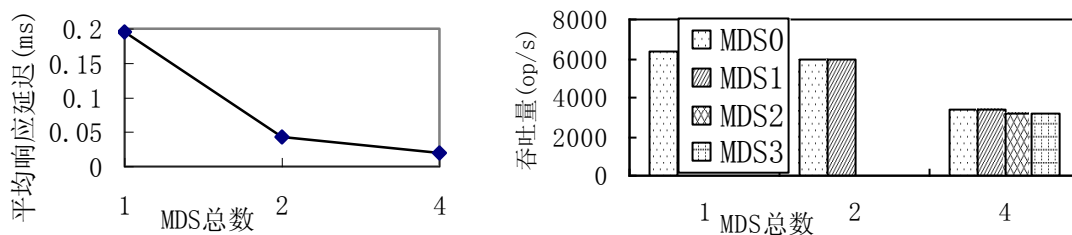


(a)每个请求的用户平均响应延迟(ms)

(b)每个 MDS 的吞吐量(op/s)

图 3.14 创建目录负载

在创建文件负载中，在存在的层次目录子树中的每个目录下分别创建 10 个文件。图 3.15 给出了创建文件负载下的测试结果。从图 3.15 中能够得到与图 3.14 相似的结论。当然，在创建文件负载下的每个 MDS 吞吐量比相同环境中创建目录负载下的每个 MDS 吞吐量近似增加了一倍，这是因为当创建目录时，不仅需要创建该目录的文件元数据，也需要创建该目录的目录转换元数据，创建目录的开销比创建文件的开销要大近似一倍。



(a)每个请求的用户平均响应延迟(ms)

(b)每个 MDS 的吞吐量(op/s)

图 3.15 创建文件负载

从上述测试中可以看出，在不同 MDS 集群规模中，每个 MDS 的吞吐量分布都大致相同，没有出现单个 MDS 成为系统访问瓶颈的情况。吞吐量的均匀分布间接论证了 MHS 方案的可扩展性。

● MDS 集群可扩展性

图 3.16 给出了在创建文件负载情况下，在 MDS 集群包括 1、2、4 个 MDS 情况下元数据请求的用户平均响应延迟与 MDS 集群中平均每个 MDS 吞吐量之间的对应关系。当元数据总请求数固定时，随着 MDS 规模的增大，MDS 集群能够提供更高的总吞吐量。从图中可以看出，当在 MDS 集群规模不同时，当平均每个 MDS 的吞吐量的取值相同时，MDS 集群规模越大时用户平均响应延迟就越高（即当图中 x 轴的取值相同时，MDS 集群的规模越大其 y 轴的值就越大），这是因为随着

MDS 集群规模的增加，引入了较多的通信开销。从图中可以看出其差值比较小，由此说明了 MHS 的可扩展性比较好。从图 3.16 中可以看出，当 MDS 集群中 MDS 总数为 4 时，MDS 集群每秒能够完成多于 22,000 个元数据操作，同时其用户平均响应延迟低于 0.04ms，它提供了高性能的元数据服务。当客户端每秒发出的元数据请求总数远大于 MDS 集群的服务能力时，平均响应延迟将快速增加，此时说明 MDS 集群需要添加更多的 MDS 以满足客户端的需求。

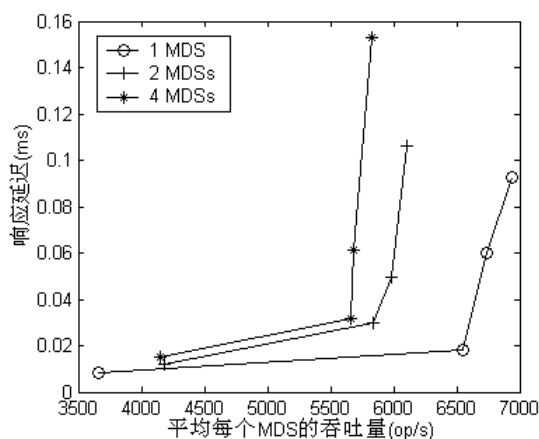


图 3.16 MDS 集群可扩展性

3.8 本章小结

本章提出一种分布式元数据管理方案 MHS 来有效处理不同的元数据负载。它采用四种技术来提供高性能和可扩展的元数据服务，具体包括仿层次目录结构、目录转换元数据、针对不同类型元数据的不同访问特性提供不同的在 MDS 集群中灵活分布元数据的方法以及高效的元数据存储方式。

MHS 消除了当前分布式元数据管理方案存在的以下的问题。第一，在子树分割方案中当访问元数据时总是需要遍历目录层次，并且元数据在 MDS 集群中分割粒度是子树，分割粒度很粗。MHS 通过采用目录转换元数据避免访问时的目录遍历，并且 MHS 在 MDS 集群中分割粒度是目录，其分割粒度小于子树分割方案中的分割粒度，因此更易在 MDS 之间均衡负载。第二，在哈希方案中，虽然采用文件全路径名哈希能快速定位到该文件元数据所在的目的 MDS，但是它不能有效处理一些情况，如目录重命名操作或 MDS 集群规模的变化。MHS 采用目录转换元数据来避免重命名目录下包含的文件元数据的重命名开销，并且采用元数据查找表来有效处理 MDS 集群规模的改变，并且它保留了哈希方案中快速定位到目的 MDS 的特性。第三，当维护目录数据时，其锁粒度和线性查找减少了元数据访问性能。在

MHS 中不再将目录数据存储于磁盘中，避免目录数据成为访问热点而导致访问瓶颈的出现。最后，当前大部分系统将同一个目录下的所有文件元数据绑定为用户对象数据存储在 OSD 中，如基于目录路径的元数据管理^[44]，Panasas^[31]，Ceph^[32]。在这种方式中，创建或删除文件的元数据操作需要读取该目录数据所在用户对象的所有字节，对于大目录来说，这个过程很慢。实际上，元数据与数据有不同的访问模式，元数据请求需要被快速响应。如果将元数据和数据混合存放在一起，并不能针对他们的不同的访问特点做优化。将元数据存储于 MDS 后端的数据库中，它能够提供很高的事务吞吐量、高可恢复性和很好的锁机制。

在文件元数据分布方法中采用了 MLT 来记录不同目录下元数据在 MDS 集群中的分布。由于文件元数据的读写访问非常频繁，并且不同目录下或不同文件的访问热度差别很大，为了在 MDS 集群中提供高性能、可扩展的元数据服务，需要在 MDS 之间负载均衡，以防止某个 MDS 成为系统中文件元数据访问的瓶颈。因此提出了一种应用于 MDS 集群的负载均衡算法。首先从映射算法上实现 MDS 集群的静态负载均衡。而针对由于文件元数据热度差别大而引起的负载不均衡，则引入动态负载均衡，从而使得整个 MDS 集群中的 MDS 的响应时间差别较小，达到 MDS 集群的负载均衡。

客户端发送的元数据请求的非原子性引入了 MDS 中元数据一致性的问题，由于 MHS 中不再保持目录数据，相比传统方案，在创建或删除文件时避免了出现元数据不一致的可能。MHS 中引入了目录转换元数据，需要保证目录转换元数据与文件元数据之间的一致性。文件元数据比目录转换元数据更重要，目录转换元数据可以通过该目录所在路径上的文件元数据逐层遍历生成，文中给出了恢复它们之间一致性的方法。对于重命名请求，其涉及到多条文件元数据的修改，需要采用日志来解决多条文件元数据之间的一致性问题。

最后对 MHS 方案的性能进行评估，首先比较了 MHS 方案和 LH 方案的性能，论证了 MHS 方案的有效性。然后给出了 MHS 方案在不同 MDS 集群规模下的元数据性能，论证了 MHS 方案的可扩展性。

4 数据放置策略研究

在对象存储系统中，以对象为单位将用户数据存储存储在 OSD 上。当客户端要访问文件时，它需要知道文件以何种方式被映射为哪些用户对象并存储在哪些 OSD 上，该工作由数据放置策略来完成。

针对存储系统规模的不同，本文提出了两种不同的放置策略。在存储系统规模较小、OSD 总数固定的情况下，提出了一种采用遗传算法来求解的静态放置策略 GA-Hybrid。在存储系统规模较大、OSD 总数可能变化的情况下，基于存储系统规模的变化趋势，提出一种动态放置策略——基于组的区分定位策略。

4.1 采用遗传算法求解混合分布的静态放置策略

当前的静态放置策略主要有两种。第一种称为随机分布，它将每个文件映射为一个对象，并根据当前存储系统中 OSD 的负载情况，将对象存放到负载较轻的一个 OSD 上^{[63][65]}。第二种称为条带化分布，它将每个文件分割为 Q(Q 为存储系统中 OSD 总数)个对象，并将这 Q 个对象分配到存储系统中的各个 OSD 上，通过所有 OSD 并行工作来得到更好的存储系统聚合吞吐量和更小的平均用户响应时间。近几年的 trace 研究^{[10][11]}表明，大部分小文件占据了存储系统的少量的存储空间，大部分存储空间被少量的大文件所占据。当采用随机分布时，单个文件由单个 OSD 存储，当某个文件过大或者访问频繁时，会导致存储该文件的 OSD 负载过重的同时而其他的 OSD 相对空闲，不能充分利用 OSD 的并行性。当采用条带化分布时，如果存储系统中的 OSD 过多或者存储的文件长度太小时，一个文件被分成了很多个对象，在各个 OSD 上查找对象的系统开销超过了 OSD 并行性带来的好处。因此应该根据文件的不同特性(如文件的大小、热点等)将文件分割为一个或多个对象存放在不同的 OSD 上，将该方法称之为混合分布。它一方面利用了 OSD 并行性带来的好处，另一方面也减少了不必要的文件条带化所引入的系统开销。

在条带化分布(Striping)中，当要服务的文件集确定后，每个 OSD 上的负载比较固定，存储系统有相对固定的聚合吞吐量。而在随机分布中，文件到 OSD 的不同映射方法使得存储系统有不同的聚合吞吐量，通常采用贪婪算法来求解^[62]，称之为 Greedy-Random。贪婪算法以当前情况为基础作出最优的选择，而不从整体最优上加以考虑，因此它是一种不追求最优解而只是在某种意义上得到局部最优解的方法。

结合以上两种静态放置策略的优缺点，提出了一种采用遗传算法来求解混合分

布的方法，称之为 GA-Hybrid。

4.1.1 分布模型

● 模型描述

- (1). 存储系统中 OSD 集合定义为 $D=\{D_0, D_1, \dots, D_{(Q-1)}\}$ ，其中 $Q=\|D\|$ 为存储系统中 OSD 总数， D_q ($0\leq q\leq Q-1$)表示存储系统中第 q 个 OSD；
- (2). 存储的文件集合定义为 $F=\{F_0, F_1, \dots, F_{(P-1)}\}$ ，其中 $P=\|F\|$ 为文件集内包含的文件总数， F_p ($0\leq p\leq P-1$)表示文件集内的第 p 个文件；
- (3). 定义文件到 OSD 的映射矩阵是 $M=[m_{pq}]_{P\times Q}$ ($m_{pq}\in\{0, 1\}$, $0\leq p\leq P-1, 0\leq q\leq Q-1$)；若

映射矩阵中的元素 $m_{pq}=1$ 则表示文件 p 的第 $\sum_{i=0}^q m_{pi}$ 个对象被分配到存储系统的第 q 个 OSD 上；若 $m_{pq}=0$ 则表示文件 p 中没有对象被映射到存储系统的第 q 个 OSD 上。 $n_p = \sum_{q=0}^{Q-1} m_{pq}$ 代表文件 p 被映射到 n_p 个 OSD 上,也即文件 p 被分割成 n_p 个对象。

- (4). 若 $m_{pq}=1$,也即文件 p 的第 $\sum_{i=0}^q m_{pi}$ 个对象被存储在存储系统的第 q 个 OSD 上,

则第 q 个 OSD 对该对象的一次存取时间 T_{pq} 包括三部分：寻道时间 T_{sq} ，等待扇区到达的时间 T_{rq} ，数据传输时间 T_{dq} ，即 $T_{pq}=T_{sq}+T_{rq}+T_{dq}$ 。若假定存储系统中 OSD 硬件配置相同，它们的数传率均为 B ，该文件 p 的大小为 S_p ，其被均匀分配到 n_p 个存储节点上，则 T_{pq} 为：

$$T_{pq} = \begin{cases} T_{sq} + T_{rq} + \frac{S_p}{n_p \times B} & \text{if } m_{pq} = 1 \\ 0 & \text{if } m_{pq} = 0 \end{cases}$$

若文件 p 的请求到达率为 λ_p ，则第 q 个 OSD 对该对象的总的存取时间为：

$$TT_{pq}=\lambda_p \times T_{pq}$$

假定文件 p 包含的 n_p 个对象完全并行操作，则存储系统对该文件的总服务时间为：

$$FT_p=\max\{TT_{p0}, TT_{p1}, \dots, TT_{p(Q-1)}\}$$

- (5). 存储系统的第 q 个 OSD 对文件集的总服务时间为：

$$DT_q = \sum_{p=0}^{P-1} TT_{pq}$$

则存储系统对文件集的总服务时间为: $T = \max\{DT_0, \dots, DT_q, \dots, DT_{(Q-1)}\}$

(6). 存储系统为文件集服务时的系统聚合吞吐量为:

$$B_{total} = \frac{\sum_{p=0}^{P-1} (\lambda_p \times S_p)}{T}$$

在存储系统中, 通常采用系统聚合吞吐量来衡量整个存储系统的性能。系统聚合吞吐量越高, 存储系统性能越好。由于对于一个确定的文件集, $\sum_{p=0}^{P-1} (\lambda_p \times S_p)$ 大小固定, 因此这里的成本函数通过 T 来体现。本文的目标是寻找映射矩阵的一种解 M' , 使得 T 近似最小。对于映射矩阵的近似最优解, 通过遗传算法进行搜索。

● 模型说明

● 模型说明

本文采用映射矩阵 $M=[m_{pq}]_{P \times Q}$ 来说明文件如何放置到存储系统的 OSD 上。在随机分布中, 一个文件仅被分配到一个 OSD 中, 即对所有的文件 $p(\forall p \in \{0, 1, 2, \dots, P-1\})$ 都有 $n_p=1$ 。在条带化分布中, 每个文件均分割为 Q 个对象分别存储在存储系统中的每个 OSD 上, 即对所有的文件 p 都有 $n_p=Q$, 也即对于文件集 F , 其映射矩阵中所有的元素均为 1。而在混合分布中, 根据不同的情况将不同的文件分割为一个或多个对象存储到一个或多个存储节点中, 即对文件 p 有 $1 \leq n_p \leq Q$ 。

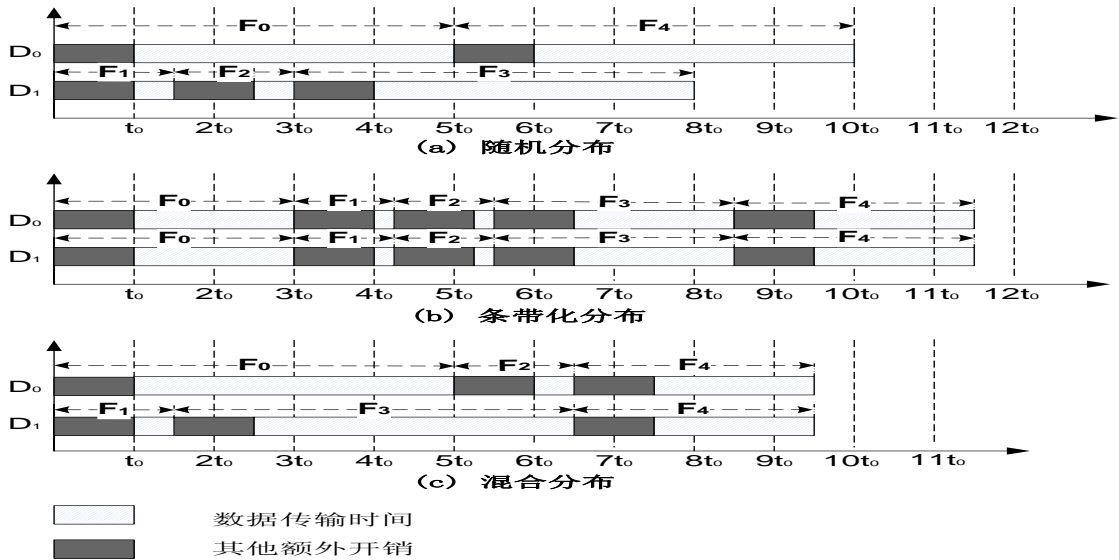


图 4.1 三种不同策略中的总服务时间分配图例子

假定 $T_{sq}+T_{rq}=t_0$ ，文件集 $F=\{F_0, F_1, F_2, F_3, F_4\}$ 的文件大小分别为 $\{4Bt_0, 0.5Bt_0, 0.5Bt_0, 4Bt_0, 4Bt_0\}$ ，每个文件的请求到达率 $\lambda_p (\forall p \in \{0, 1, 2, 3, 4\})$ 均为 1，存储节点集为 $D=\{D_0, D_1\}$ ，图 4.1 给出了三种不同放置策略的总服务时间分配图的一个例子，其相应的映射矩阵如表 4.1 所示。

由图 4.1(a) 可得，在随机分布中，存储系统对文件集的总服务时间 $T=\max\{10t_0, 8t_0\}=10t_0$ ；由图 4.1(b) 可得，在条带化分布中，存储系统对文件集的总服务时间 $T=\max\{11.5t_0, 11.5t_0\}=11.5t_0$ ；由图 4.1(c) 可得，在混合分布中，存储系统对文件集的总服务时间 $T=\max\{9.5t_0, 9.5t_0\}=9.5t_0$ 。比较可知，三种不同的放置策略求解中，采用混合分布策略时存储系统对文件集的总服务时间最小，从而系统聚合吞吐量最高。

表 4.1 不同放置策略对应的映射矩阵

Greedy-Random	Stripping	GA-Hybrid
$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$

4.1.2 遗传算法实现

遗传算法是模拟生物在自然环境中的遗传和进化工程而形成的一种自适应全局优化概率搜索算法^[107]。它提供了一种求解复杂系统优化问题的通用框架，不依赖于问题的具体领域，有很强的鲁棒性，广泛应用于很多学科。遗传算法具有的高效搜索性、收敛性以及对于 NP 完全问题求解的有效性表明其是求解文件放置问题的有效途径。

● 编码方法

将映射矩阵 $M=[m_{pq}]_{P \times Q}$ ($m_{pq} \in \{0,1\}$, $0 \leq p \leq P-1$, $0 \leq q \leq Q-1$) 表示成 $M=[M_0, M_1, \dots, M_{(P-1)}]^T$ ，其中 M_i ($0 \leq i \leq P-1$) 为映射矩阵的第 i 行，则该映射矩阵形成的个体编码是：

$$X: M_0 M_1 \cdots M_{(P-1)}$$

它反映了文件集内的文件与存储系统中 OSD 之间的一种映射关系。

它也等价于：

$$X: b_0 b_1 \cdots b_{P \times Q - 1}$$

其中， $b_i = m_{xy}$, $0 \leq i \leq P \times Q - 1$, $x = \left\lfloor \frac{i}{Q} \right\rfloor$, $y = i \% Q$

如对于文件集 $F=\{F_0, F_1, F_2\}$ 和存储节点集 $D=\{D_0, D_1\}$, 当映射矩阵 $M=\begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$ 时,

$M_0=[1, 1]$, $M_1=[1, 0]$, $M_2=[0, 1]$, 则此时个体编码是: X: 111001, 其代表的含义是: 文件 F_0 被分割为两个对象分别存储在 D_0 和 D_1 中, 文件 F_1 仅仅被看作一个对象存储在 D_0 中, 文件 F_2 仅仅被看作一个对象存储在 D_1 中。

● 适应度函数

适应度函数用来评价各个个体在优化计算中可能达到或接近于或有助于找到最优解的优良程度, 被用来度量解的好坏。适应度越高, 解的质量越好, 被遗传到下一代的概率就越大。适应度函数的选取至关重要。

本文中适应度函数定义为:

$$F(M) = \frac{1}{f(M)}$$

其中 $f(M) = \max\{ft(M, q) | q = 0, 1, \dots, Q-1\}$ 。 $ft(M, q)$ 表示在映射矩阵为 M 时存储系统的第 q 个 OSD 对文件集的总服务时间, 它表示为:

$ft(M, q) = \sum_{p=0}^{P-1} (\lambda_p \times (T_{sq} + T_{rq} + \frac{S_p}{n_p \times B}) \times m_{pq})$, $f(M)$ 是求取存储系统中 OSD 集合 $D=\{D_0, D_1, \dots, D_{(Q-1)}\}$ 中为文件集服务的总服务时间最长的 OSD 的服务时间。对于同一个文件集, 当 $f(M)$ 越小时, 映射矩阵 M 的适应度越高。

● 选择算子

遗传算法使用选择算子来对个体进行优胜劣汰操作, 适应度较高的个体被遗传到下一代的概率较大, 适应度较低的个体被遗传到下一代的概率较小, 选择算子用来确定如何从父代按某种方法选择哪些个体遗传到下一代, 其建立在对个体适应度进行评价的基础上。

本文采用将最优保持策略和比例选择相结合的方法来从父代选择个体遗传到下一代。最优保存策略是指用当前适应度最高的个体来替换掉本代群体中经过遗传操作后产生的适应度最低的个体。它保证迄今为止所得到的最优个体不会被破坏, 是遗传算法收敛性的一个重要保证条件。比例选择(也叫赌盘选择)是指个体被选中并遗传到下一代中的概率与该个体的适应度成正比。

● 交叉算子

交叉是指将通过选择算子依次选出的两个个体通过交配而重组, 产生新的个体。它是遗传过程中产生新个体的主要方法, 决定了遗传算法的全局搜索能力, 是进化过程中的一个主要环节。本文中采用最常用的交叉算子: 单点交叉算子, 将两

个相互配对的个体按照一定的交换概率 P_c 从某一位开始逐位互换。

- 变异算子

变异是指将某个个体编码串中的某些位(变异点)求反(0 变 1, 1 变 0)。它是遗传过程中产生新个体的辅助方法, 决定了遗传算法的局部搜索能力。本文中采用最简单的变异算子: 基本位变异算子, 先以一定的变异率 P_m 随机选择要进行变异的个体, 然后产生一个随机数, 将该随机数指定的位求反。

- 约束算子

编码 $X: b_0b_1 \cdots b_{P \times Q - 1}$ 反映了文件集内的所有文件与存储系统中 OSD 之间的一种映射关系, 其中如果 $b_i b_{i+1} \cdots b_{i+(Q-1)} = 00 \cdots 0$ ($\forall i \% Q = 0$ 并且 $0 < i \leq P \times Q - 1$), 即表示文件 $F_{(i/Q)}$ 不被存储到任意 OSD 上, 这种属于无效解。当产生有这种解时将随机选择 $b_i b_{i+1} \cdots b_{i+(Q-1)}$ 其中一位将其求反(即置 1), 从而保证得出解的有效性。

- 终止条件

运行 T 代后, 停止计算, 输出当前最好的解。

- 运算过程

1. 随机初始化种群 $era(t)$, 其中 $t=0$ 。
2. 计算种群 $era(t)$ 中各个个体的适应度。
3. 种群 $era(t)$ 依次经过选择算子、交叉算子、变异算子、约束算子的作用后得到下一代种群 $era(t+1)$ 。
4. 进行终止条件判断, 若还不符合终止条件则转到步骤 2, 否则终止运算并输出最优解。

4.1.3 实验结果

为了评估和验证 GA-Hybrid 算法的有效性, 这里测试了不同负载状况和存储系统包括不同 OSD 总数下 Greedy-Random、Stripping 和 GA-Hybrid 对存储系统聚合吞吐量的影响, 为了便于比较, 将其值正规化到 $[0,1]$ 区间。

实验中, 假定 OSD 的平均寻道时间为 3.4ms, 平均旋转延迟为 2ms, 数据传输率为 55MB/s^[108]。文件集内的文件总数为 250, 文件请求到达率相同。当文件大小服从 Zipf 分布^[64]且其平均文件大小为 3MB 时, 测试结果如图 4.2(a)所示; 当文件集内文件大小依次递增, 且服从 LLNL trace 分布^[109]时, 测试结果如图 4.2(b)所示。

从图 4.2 中可以看出, 随着存储系统 OSD 总数的增加, 每种方案的系统聚合吞吐量均增加, 并且不论在何种负载下, GA-Hybrid 的聚合吞吐量和其增长率都最高, 充分显示了该方案的优越性, 这是因为它能从全局考虑, 并在 OSD 并行性与系统开销之间折衷, 因此其性能最好。在图 4.2(a)中, Greedy-Random 的聚合吞吐量高

于 **stripping** 的系统聚合吞吐量，并且随着存储节点数目的增多，**Greedy-Random** 的聚合吞吐量近似线性增长，**Striping** 的聚合吞吐量增长率低于线性增长。这是因为在该负载中，文件集内的文件之间长度差异较小，**Greedy-Random** 中采用贪婪算法将每个文件分布到当前有最小响应时间的 **OSD** 上，各个 **OSD** 之间的响应时间差别较小，使得系统性能随 **OSD** 总数近似线性增长；在 **Striping** 中，随着 **OSD** 数目的增多，每个 **OSD** 上分布的文件分条的大小减少而额外开销相对固定，从而导致其聚合吞吐量的增长率低于线性增长（该聚合吞吐量的增长率与平均文件大小有关，若平均文件大小较小，则额外开销比率较大，导致其增长率较小）。在图 4.2(b)中，由于文件集内存在少量的大文件，当采用 **Greedy-Random** 时，文件不能在 **OSD** 之间均匀分配，导致 **OSD** 之间的响应时间差别较大，因此系统聚合吞吐量小于 **stripping** 中的系统聚合吞吐量，并且随着 **OSD** 数目的增多，**Greedy-Random** 的聚合吞吐量增长率低于线性增长，**GA-Hybrid** 此时相对于 **Greedy-Random** 和 **Stripe** 的优势更加突出。

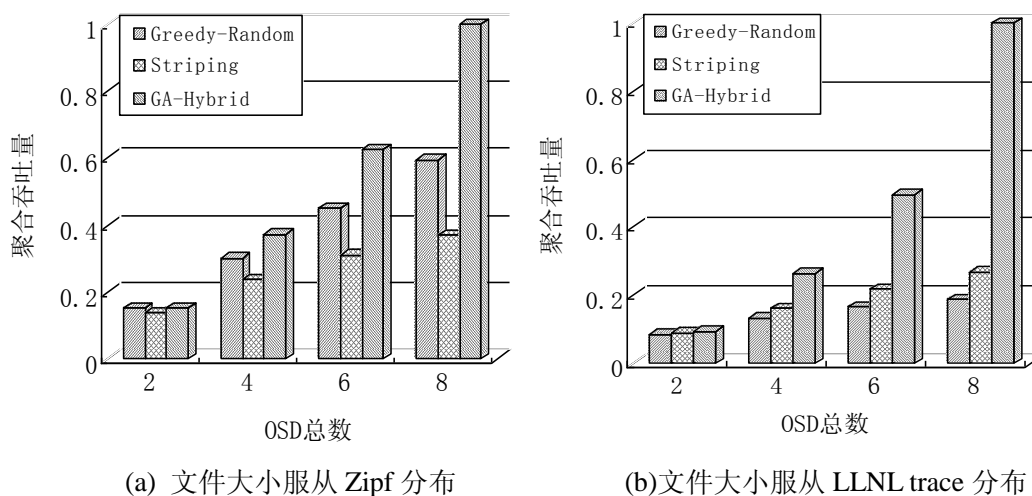


图 4.2 OSD 总数对三种方案性能的影响

4.2 动态放置策略的选择

由于大规模存储系统支持的应用灵活多变，文件的访问模式差别很大，客户端能在创建每个文件时指定其文件到对象的映射策略，如将文件映射为单个对象或采用条分的方法将文件映射为多个对象等。由于文件到对象的映射策略灵活多变，在此就不一一列出，这里主要关注的是对象与 **OSD** 之间的映射策略，即对象放置策略。

对象放置策略主要解决两种问题：如何在存储系统包括的 OSD 之间分布对象和定位对象。分布对象是指当新创建一个对象时，如何选择一个合适的 OSD 来存放该对象；定位对象是指当要访问某个对象时，如何知道该对象所在的 OSD。对象放置策略的选择通常分为两种，一种是采用启发式方法，一种是采用分布式算法。如 4.1 节的静态放置策略中为对象选择合适的 OSD 采用的是启发式方法，启发式方法中需要采用映射表记录对象到 OSD 之间的映射关系。

在启发式方法中，通常需要统计 OSD 的负载信息，选择负载较轻的 OSD 来存储新创建的对象，对象可存放在任意 OSD 上，由于对象存储位置的不确定性，需要采用映射表来记录对象到 OSD 的映射关系，以便在访问该对象时通过查找该映射表得知该对象所在的 OSD。在这种方法中，能实时根据系统中 OSD 负载来选择合适的 OSD 存储对象，均衡了 OSD 的负载；当 OSD 个数发生变化时，需根据 OSD 之间的负载情况在它们之间迁移对象，实现系统中 OSD 之间的负载均衡。当存储对象的 OSD 发生改变时，需要对映射表中相应的信息进行更新。随着系统规模的扩大，该映射表也会变大，导致大量内存开销以及访问映射表的慢速查找，形成系统性能瓶颈和潜在的单点失效。同时随着系统规模的增大，集中统计大量的 OSD 的实时运行开销也很大，也易形成系统瓶颈，即该方法的扩展性不是很好。

在分布式算法中，通过直接计算来确定对象与 OSD 之间的映射关系。在该类算法中，在存储系统中的存储设备层次描述信息已知的情况下，对象被映射到确定的 OSD 上。能在存储系统的任意设备上并行计算对象所在的 OSD 的位置，它不再需要记录对象到 OSD 之间的映射关系，节省了存储映射表的内存开销和查找开销，且避免了映射表的热点访问瓶颈。但是由于分布式算法计算的不确定性，导致对象不能在 OSD 间随意移动，为 OSD 之间的负载均衡带来了挑战(如实际系统中，存储系统的性能可能被一些慢的超负载的 OSD 拖累，采用启发式方法能够避免将对象分配到这类 OSD 中，而分布式算法却不能考虑到该情况)；当存储系统中 OSD 数量发生变化时，需要迁移对象来维护算法的一致性，迁移开销不小。

启发式方法和分布式算法的优缺点概括如表 4.2 所示。启发式方法具有对象分布的灵活性，更易实现 OSD 之间的负载均衡，但是其可扩展性不好，内存开销和查找开销与存储系统中存储的对象总数相关；分布式算法相比启发式方法几乎没有内存开销，其映射关系的计算能在存储系统中任意设备上快速完成，因此可扩展性好，但是由于其算法的不确定性，对象在 OSD 之间的分布不灵活，负载的均衡程度没有启发式方法高。基于前面的分析，需要一种对象放置策略能在对象分布的灵活性和系统可扩展性之间做权衡。

表 4.2 启发式方法与分布式算法比较

	启发式方法	分布式算法
对象所在 OSD 的选择	任意	确定
对象映射的灵活性	灵活	不灵活
是否需记录对象与 OSD 的映射关系	需要	不需要
存储映射的内存开销	与对象总数成正比	无
查找开销	大	无
执行映射的地方	一般集中存储在 MDS 上	分布式, 存储系统内任意设备
映射关系改变时是否有更新记录开销	有	无
负载均衡性	好	依赖于算法
可扩展性	差	好

在系统中, 大部分的存储空间被数量较少的大对象占据, 少量的存储空间被数量众多的小对象占据。大对象和小对象有着不同的访问特性和生命周期。有研究^{[110][111]}表明绝大操作是针对小对象的, 小对象面临更频繁的创建和删除操作。若对小对象采用启发式方法放置, 则会浪费大量的内存空间来记录与小对象对应的 OSD 位置, 并且由于其频繁的创建和删除操作导致经常需要对映射表进行修改。小对象数量众多, 采用映射均匀的分布式算法能将小对象均匀分布到 OSD 中, OSD 负载能够相对均衡。综合考虑可知, 分布式算法更适合于小对象。系统中大对象数量不多, 采用启发式方法需要的映射表并不会浪费太多存储空间和查找开销。大对象占用的存储空间差别很大, 大对象数量也不多, 若采用分布式算法则在 OSD 之间负载不会均衡, 而且当系统规模发生改变时, 为了维护算法的一致性导致大对象在 OSD 之间迁移占用了很大的网络开销(因为大对象占用的存储空间大)。由于大对象的创建删除频率没有小对象那么频繁, 因此对映射表的修改也不会那么频繁。综合考虑可知, 大对象更适合采用启发式方法。

Diffloc^[87]对不同类型的对象采用不同的放置策略。对小对象采用分布式算法(文章中采用一致性哈希)来放置, 既能将大量小对象均匀分配到 OSD 上又能避免采用启发式方法导致的大量的内存开销和查找开销。对大对象采用启发式方法(在剔除一些剩余空间小于均值的 OSD 后, 在剩下的 OSD 中以与剩余空间成正比的概率选择 OSD 来存放大对象)来放置, 并采用 Bloom Filter 来记录对象所在的 OSD, 既有利

于实现系统负载均衡又避免了全部采用启发式方法导致的大量的内存和查找开销。由于一致性哈希的计算开销和系统中 Bloom Filter Array 的存储开销均随系统中 OSD 总数线性增长,随着系统规模的增大,OSD 数量成百上千,这个时候 Diffloc 的存储开销和查找开销都很大。

针对 Diffloc,提出基于组的区分定位策略(Group-based differentiated location, G-Diffloc),它是对 Diffloc 策略的改进。在大规模存储系统中,OSD 数量成百上千,OSD 很少被一个一个的加入,通常是按照存储系统的需求,适时的加入一批新的 OSD 或淘汰一批旧的 OSD,同批加入或淘汰的 OSD 通常硬件配置相同,不同批次加入或淘汰的 OSD 的硬件配置通常不同(如近期加入的 OSD 通常比以前加入的 OSD 的硬件配置要好,反应为 OSD 权重不同),对象放置策略应该能够根据 OSD 的能力分配对象。首先根据 OSD 的不同批次将其组织成子集群,先采用分布式算法(straw bucket^[86])将对象映射到存储系统的某个子集群中,再在子集群内部根据不同类型的对象采用不同的放置方法,对大对象采用启发式方法从子集群中选取负载较轻的 OSD 放置,并采用 Bloom Filter 来记录该大对象在子集群内 OSD 的位置,对小对象采用分布式算法(改进哈希算法)来放置。相比 Diffloc, G-Diffloc 提高了其可扩展性和查询效率,并减少了内存开销。

在本文中,默认采用 512KB 作为区分大小对象的界限,该选择来源于对以往文件系统元数据快照^[11]的分析(文件小于 512KB 的文件占了文件总数的 90%,这些文件占用的总存储空间却只有所有文件所占存储空间的 20%),实际应用中可以根据不同的应用负载情况调整该参数。

4.3 改进哈希算法

由于简单哈希算法是通过 $x\%N$ 来定位对象 x 所在的 OSD,其中 N 为当前存储集合(存储集合既可以指存储系统包含的所有 OSD,也可以指存储系统包含的每个存储子集群)中 OSD 总数。由于存储集合中可能会因为增加新 OSD 或者移除老 OSD 而导致 OSD 总数发生改变,此时简单哈希算法会导致大量的对象在 OSD 之间迁移。这里提出了一种新的哈希算法,改进哈希算法,来处理这种情况。改进哈希算法既能继承简单哈希算法的计算开销小和均匀分配对象的优点,又能保证在存储集合发生变化时迁移近似最优的对象。改进哈希算法需要记录存储集合描述信息,它包括该存储集合中最初 OSD 总数 I 和当前 OSD 总数 C ,存储集合最初 OSD 地址信息列表 $Init_list\{ADDR_0, ADDR_1, \dots, ADDR_{(I-1)}\}$ 和存储集合中当前 OSD 地址信息列表 $Current_list\{ADDR_0, ADDR_1, \dots, ADDR_{(C-1)}\}$,存储集合中最初 OSD 状态信息 $Status$ 。其中 $Status$ 的元素个数等于 I ,它的元素与 $Init_list$ 中的元素一一对应,用于说明

Init_list 中每个 OSD 的状态, 若 OSD 为正常, 则 Status 中相应元素置为 VAILD, 若该 OSD 无效, 则 Status 中相应元素置为 INVALID。Status 状态的引入, 避免了大量的不必要的对象迁移。

在改进哈希算法中, 算法的执行依赖于对存储集合描述信息的判断, 当存储集合发生改变时, 存储集合描述信息的更新至关重要。存储集合描述信息更新方法的详细说明如下所示。

当该存储集合内有 x 个 OSD 移出时, 需要修改存储集合描述信息的相应位置。当要移出一个 OSD 且其地址为 A_i 时:

(a) 若此时 $C \leq I$:

- 1) 若 A_i 等于 Init_list[m], 则将 Status[m]置为 INVALID;
- 2) 若 A_i 等于 Current_list[n]:
 - i. 若此时 $n == C-1$, 则将 Current_list 中最后一个元素删除, 并将 $C=C-1$;
 - ii. 否则将 Current_list[n]的值用 Current_list[C-1]的值更新, 并将 Current_list 中最后一个元素删除, 并将 $C=C-1$;

(b) 若此时 $C > I$:

- 1) 若 A_i 不在 Init_list 中, 则不需修改 Init_list 和 Status 中的值; 若 A_i 等于 Init_list[m], 则将 Init_list[m]的值用 Current_list[C-1]的值更新。
- 2) 若 A_i 等于 Current_list[n]:
 - i. 若此时 $n == C-1$, 则将 Current_list 中最后一个元素删除, 并将 $C=C-1$;
 - ii. 否则将 Current_list[n]的值用 Current_list[C-1]的值更新, 并将 Current_list 中最后一个元素删除, 并将 $C=C-1$;

以此类推处理完剩余的 $x-1$ 个要移出的 OSD。

当该存储集合内要新加入 y 个 OSD 时, 需要修改存储集合描述信息相应位置。当要添加一个 OSD 且其地址为 A_j 时:

(a) 若 Status 中不存在值为 INVALID 的元素:

- 1) 为 Current_list 添加一个尾部元素, 其值为 A_j , 即 $Current_list[C] = A_j$;
- 2) 更新 $C=C+1$;

(b) 若 Status 中第一个状态为 INVAILD 的元素为 Status[n]:

- 1) 将 Init_list[n]的值更新为 A_j , 然后将 Status[n]的值更新为 VALID;
- 2) 为 Current_list 添加一个尾部元素, 其值为 $Current_list[n]$, 即 $Current_list[C] = Current_list[n]$;
- 3) 将 A_j 赋值给 $Current_list[n]$, 即 $Current_list[n] = A_j$;

4) 更新 $C=C+1$;

以此类推处理完剩余的 $y-1$ 个要移出的 OSD。

实际上，当 Status 中存在值为 INVALID 的元素时，必然有 $C < I$ ；当 Status 中不存在值为 INVALID 的元素时，必然有 $C \geq I$ 。引入 Status 后，当有 OSD 从该存储集合中移出时，Init_list 中的内容并不会发生改变，而仅仅是将 Status 中与该 OSD 对应的元素的值置为 INVALID，然后根据该算法将该无效 OSD 中的对象迁移到其他 OSD 中，避免了大量的不必要的对象迁移。

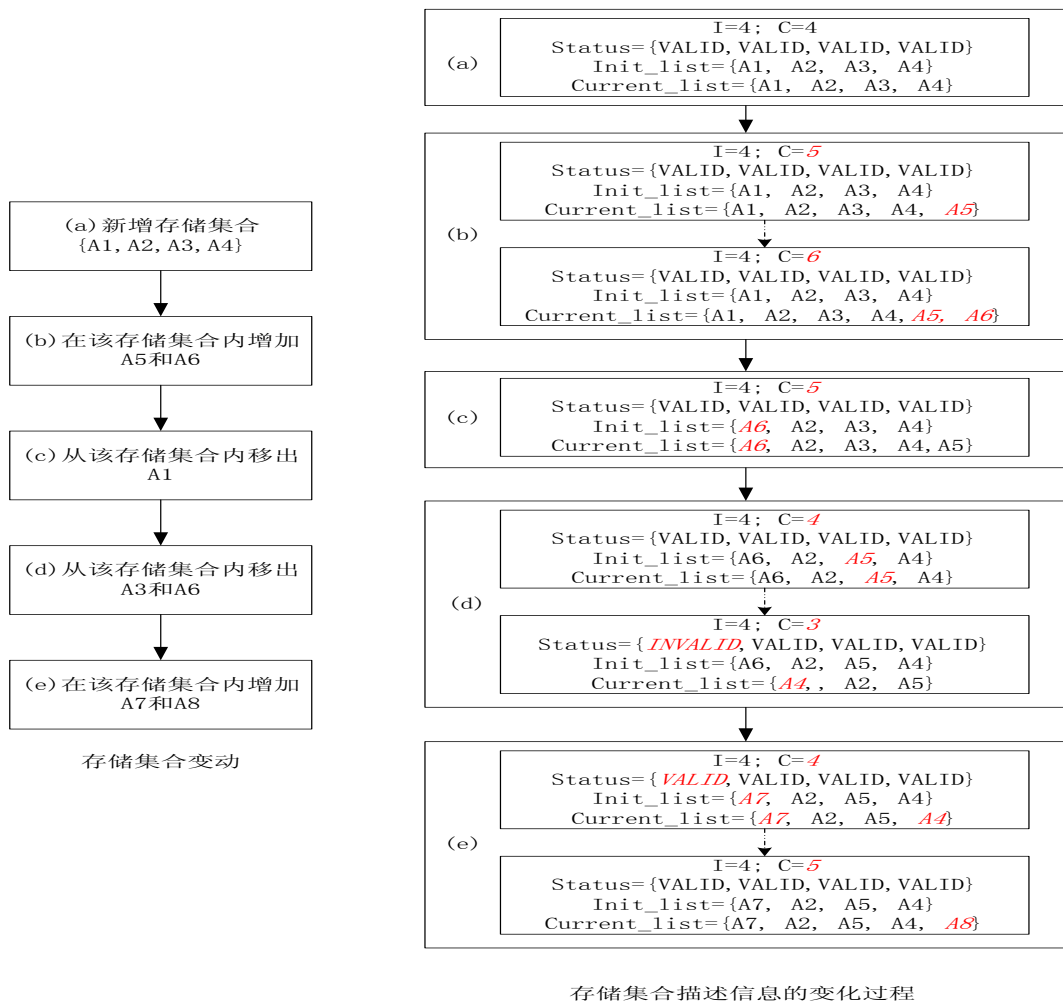


图4.3 存储集合描述信息变化实例

图 4.3 中给出一个存储集合不断发生改变时存储集合描述信息变化过程的实例。其中斜体部分信息是相对于上次的变化信息。

改进哈希算法

1. 输入: o : 对象标识符; Info: 存储集合描述信息($I, C, \text{Init_list}, \text{Current_list}, \text{status}$)
 2. 输出: 对象 o 所在的 OSD 的地址
 3. 功能: 返回存储集合中存放对象 o 的 OSD 的地址
 4. **if** $C \geq I$ **then** //如果 $C > I$ 则说明在该存储集合中添加了一些新的 OSD
 5. $T \leftarrow C$
 6. **while** $T > I$ **do**
 7. $k \leftarrow o \bmod T$
 8. **if** $k == T-1$ **then**
 9. $\text{host} \leftarrow \text{Current_list.ADDR}[k]$
 10. **return** host
 11. **else** $T \leftarrow T-1$
 12. **end if**
 13. **end while**
 14. **if** $T == I$ **then**
 15. $k \leftarrow o \bmod T$
 16. $\text{host} \leftarrow \text{Current_list.ADDR}[k]$
 17. **return** host
 18. **end if**
 19. **end if**
 20. **if** $C < I$ **then** //一些 OSD 已经从该存储集合中移除
 21. $k \leftarrow o \bmod I$
 22. **if** $\text{status}[k] \neq \text{INVALID}$ **then**
 23. $\text{host} \leftarrow \text{Init_list.ADDR}[k]$
 24. **return** host
 25. **else**
 26. $k \leftarrow o \bmod C$
 27. $\text{host} \leftarrow \text{Current_list.ADDR}[k]$
 28. **return** host
 29. **end if**
 30. **end if**
-

改进哈希算法描述如算法 4.1 所示。从算法 4.1 中可以看出, 当当前 OSD 总数小于最初 OSD 总数时, 计算开销是常数级, 当当前 OSD 总数大于最初 OSD 总数时, 计算开销与当前 OSD 总数与最初 OSD 总数的差值呈线性关系, 这一点在后面的实验中得到了论证。

图 4.4 给出了改进哈希算法运行的一个例子。假定该存储集合最初 OSD 总数为 4, 一些对象存储在该存储集合中。在运行期间, 有一个 OSD 从该存储集合中移出,

后来有一个 OSD 加入到该存储集合中，最后又有一个 OSD 加入到该存储集合中。该存储集合的存储集合描述信息和对象分布如图 4.4 所示，它们随该存储集合中 OSD 的变化在图中用斜体标识出。从例子中可以看出，当该存储集合中 OSD 总数发生变化时，该算法导致迁移的对象数与最优迁移量近似相同。

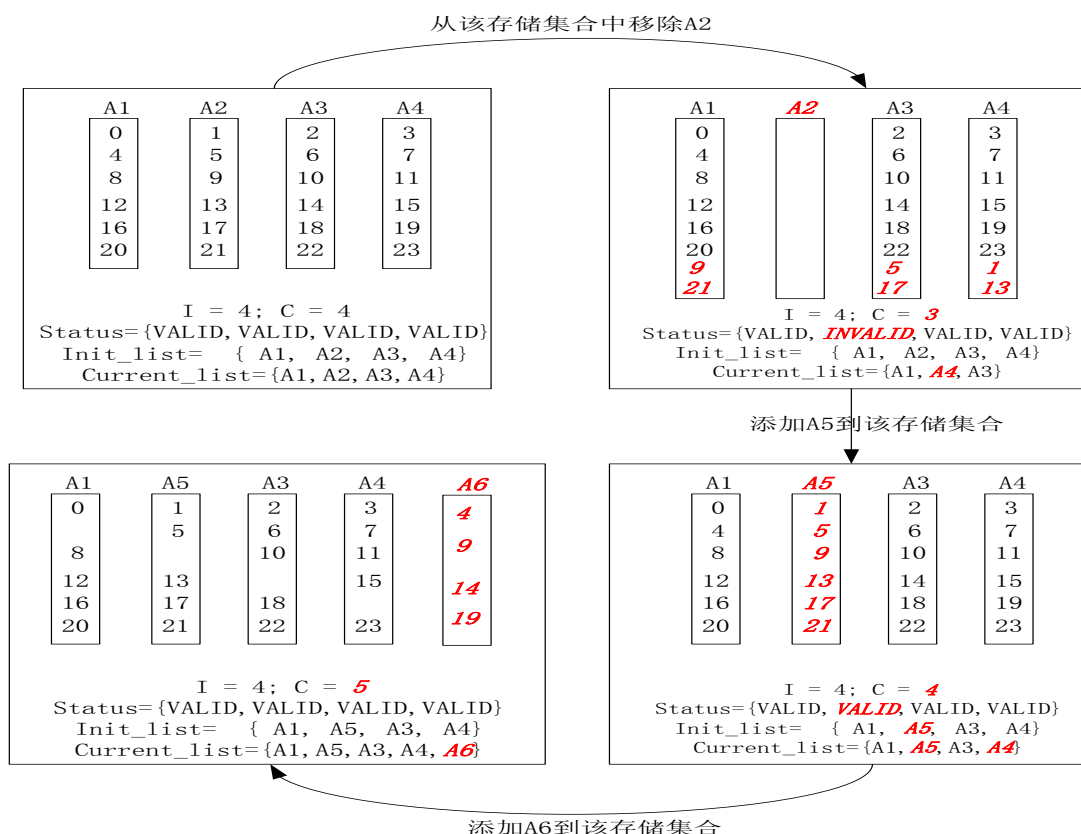


图4.4 采用改进哈希算法放置对象的一个例子

这里对改进哈希算法对象分布的均匀性以及当存储集合规模发生改变时该算法迁移的对象数进行评估。设映射到该存储集合中的对象数总共有 $NUM=10^6$ 个，设该存储集合最初包含有 $I=32$ 个 OSD。假定从该存储集合中移出后面 6、5、4、3、2、1 个 OSD 或添加 1、2、3、4、5、6 个 OSD 到该存储集合中，则该存储集合当前包含的 OSD 总数 $C=26、27、28、29、30、31、33、34、35、36、37、38$ ，评估在这 12 种情况下每个 OSD 上分配的对象数与平均理论值的比，以及在这几种情况中由于存储集合规模的变化为了维护算法一致性导致的对象总迁移数与迁移理论值的比，其结果分别如图 4.5 和图 4.6 所示。

当评估对象分布均匀性时，平均每个 OSD 上分布的对象数的平均理论值是 $Average_{optimal} = NUM/C$;

当评估由于存储集合规模改变导致的对象总迁移数 M 时，迁移理论值 $M_{optimal} =$

$((C-I)/C) \times \text{NUM}$ (当 $C > I$ 时)或者 $M_{\text{optimal}} = ((I-C)/I) \times \text{NUM}$ (当 $C < I$ 时)。

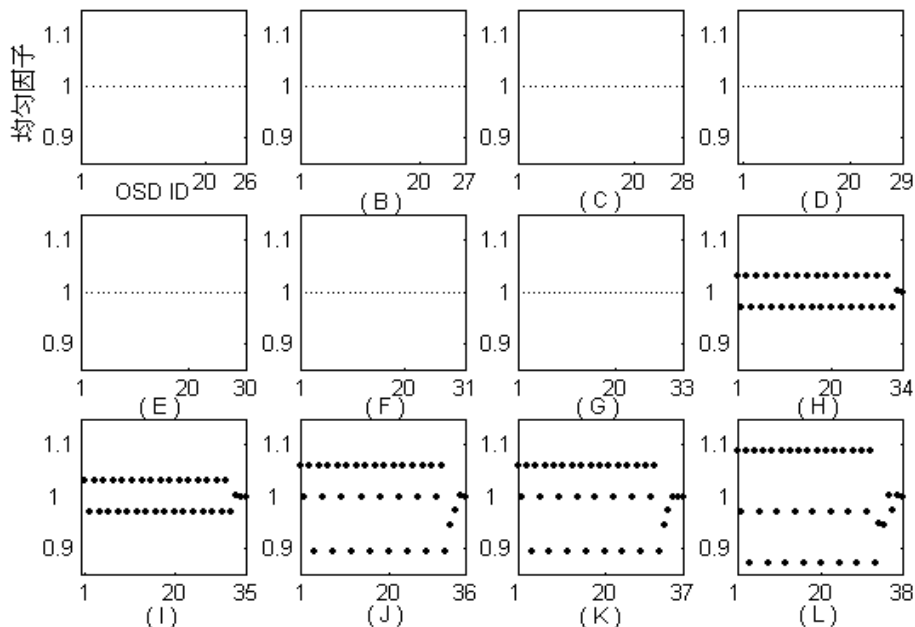


图4.5 改进哈希算法在存储集合规模变化时的分布有效性(1为最优)

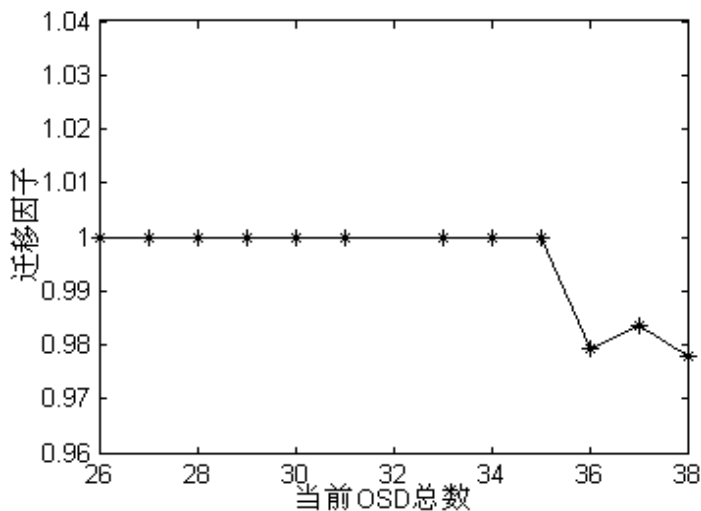


图4.6 改进哈希算法在存储集合规模变化时的迁移有效性

从图 4.5 和图 4.6 中可以看出，当存储集合中当前 OSD 总数小于最初 OSD 总数时能保证每个 OSD 上分布的对象数与平均理论值相等，并且实际迁移对象数等于迁移理论值；当存储集合中当前 OSD 总数大于最初 OSD 总数且差别不大时，每个 OSD 上分布的对象数与平均理论值相差无几，对象总迁移数与迁移理论值相等，随着当前 OSD 总数的逐渐增大，每个 OSD 上分布的对象数与平均理论值的差距以

及对象总迁移数与迁移理论值的差距逐渐增大。在测试过程中，对迁移的对象进行跟踪，迁移的对象是在要添加或删除的 OSD 与存储集合内其他剩余的 OSD 之间移动，而不会有对象在不会发生改变的 OSD 之间移动，因此迁移的对象总数与均匀分配时的迁移理论值相当。从上述实验中可以看出，在存储集合内当前 OSD 总数小于最初 OSD 总数或与最初 OSD 总数差别不大时，改进哈希算法能在 OSD 之间均匀分布对象同时涉及到的迁移对象数与迁移理论值相当，但当存储集合中当前 OSD 总数远大于最初 OSD 总数时，改进哈希算法使得 OSD 之间分配的对象数的差距逐渐变大，需要迁移的对象总数与迁移理论值的差距逐渐变大。

下面给出改进哈希算法的计算时间开销。实验环境与上述实验一样，设该存储集合最初包含有 $I=32$ 个 OSD，存储集合当前包含的 OSD 总数 $C=26、27、28、29、30、31、32、33、34、35、36、37、38$ ，评估在这 13 种情况下改进哈希算法的计算时间开销，其结果如图 4.7 所示。图中的耦合线是根据最小当前 OSD 总数、最初 OSD 总数、最大当前 OSD 总数以及它们的定位时间以及算法的理论分析画出的算法的理论变化情况。从结果中可以发现，该算法的计算时间开销与耦合线近似重合，即当存储集合内的当前 OSD 总数小于或等于最初 OSD 总数时，其计算时间与存储集合内 OSD 配置不发生改变时近似一样；当当前 OSD 总数大于最初 OSD 总数时，其计算时间开销随当前 OSD 总数与最初 OSD 总数的差值线性变化，即 $O(C-I)$ ；实验结果与理论情况一致。从结果中可以看出，改进哈希算法的计算时间开销非常小，是纳秒级，在后面的实验中可以看出，改进哈希算法的计算时间开销与 straw bucket、一致性哈希或者 Bloom Filter 定位时间开销相比可忽略不计。

从改进哈希算法的实验结果中可以看出，改进哈希算法在当存储集合内 OSD 数目添加较少或 OSD 数目减少时有非常好的性能，当存储集合内 OSD 数目添加较多时其性能在逐渐变差。当该存储集合是指存储系统的子集群时，由于子集群内的 OSD 总数不会增量较大(成批增加的 OSD 被作为新的子集群加入到存储系统中)，子集群充分利用了改进哈希算法的性能优势。

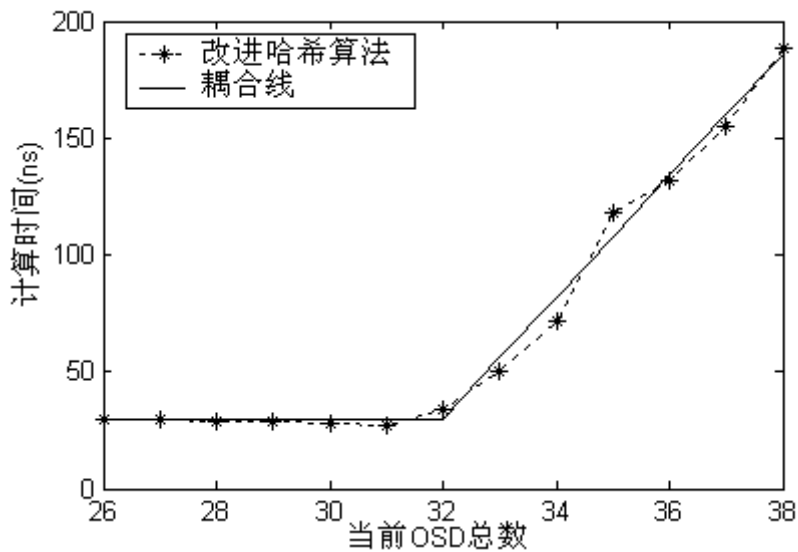


图4.7 改进哈希算法对每个对象的计算时间(最初OSD总数为32)

4.4 基于组的区分定位策略

提出一种灵活可扩展的区分对象放置策略——基于组的区分定位策略 (Group-based differentiated location, G-Diffloc)。本节首先需要概括性介绍 G-Diffloc 和 Diffloc 中需要使用到的一些分布式算法或方案,然后给出基于组的区分定位策略的详细描述。

4.4.1 Bloom Filter

Bloom Filter 是一种有损数据表示方案,在 1970 年由 Bloom 提出^[46],用于快速判断某个元素是否属于特定的集合 S 。Bloom Filter 最初为集合 S 初始化一个对应的

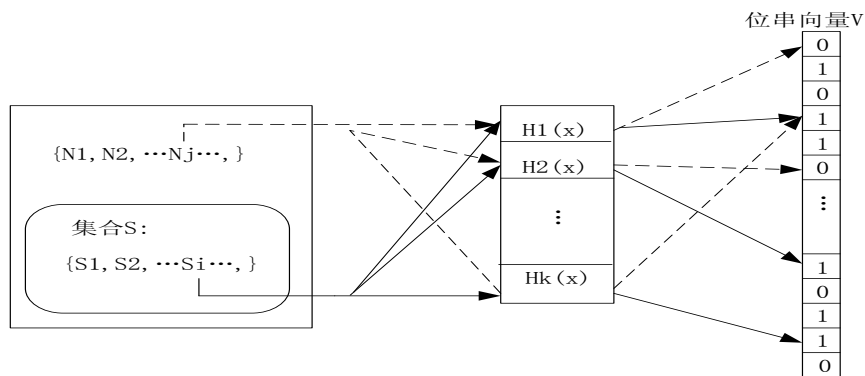


图4.8 Bloom Filter工作过程

位串向量 V ，当某个元素属于该集合时，则将该元素的关键字通过 k 个哈希函数 $\{H_1(x), H_2(x), \dots, H_k(x)\}$ 映射到属于该位串向量 V 的 k 个位的值置为 1。当判断一个元素是否属于该集合 S 时，检查该元素的关键字通过上述的 k 个哈希函数映射到的属于位串向量 V 的 k 个位的值是否均为 1，若不是，则该元素不属于集合 S ，若是则认为该元素属于集合 S 。Bloom Filter 工作过程如图 4.8 所示。

Bloom Filter 方案不存在错误否定的误判(对于属于集合的元素 S_x ，不会出现认为其不属于该集合的误判)，存在错误肯定的误判 (对于不属于集合的某个元素 N_x ，可能会出现认为其属于该集合的误判)。错误肯定的误判概率与位串向量的长度、哈希函数的个数以及该集合中元素个数相关。

设与该集合对应的位串向量长度为 m ，该集合包含元素个数为 n ，哈希函数个数为 k ，则位串向量中任一一位为 1 的概率为：

$$p = 1 - \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-kn/m}$$

则元素的错误肯定误判概率为：

$$F = p^k$$

通过求导可得出当 $k=(\ln 2) \times (m/n)$ 时错误肯定误判概率 F 的值最小为 $(0.6185)^{m/n}$ 。

使用 Bloom Filter，当增加一个元素到集合或者查找一个元素是否属于该集合，其计算开销是 $O(k)$ ，即常数级时间复杂度。存储空间开销为 $O(m)$ ，其中 m 为该位串向量长度，并且其中平均为每个元素保存 m/n 个位。由于该方案具有常数级查找时间并且存储空间开销小，因此被广泛应用到各种计算机领域。

4.4.2 一致性哈希算法

一致性哈希算法^[88]在存储系统的 OSD 之间均匀分配对象，当有 OSD 加入或从存储集合中移除时，仅需迁移少量的对象来维护该算法的一致性。它的基本思想是：将对象和存储集合包含的 OSD 都映射到一维区间 $[0,1)$ 内，然后计算对象和 OSD 之间的距离，将对象存放到与其具有最小距离的 OSD 上。当存储集合中有 OSD 加入或移除时，与该新加入或要移出 OSD 是最近邻居的(在 $[0,1)$ 区间内用 OSD 之间的间距度量)OSD 上的部分对象将发生迁移，对象迁移仅仅发生在两个 OSD 之间。为了确保每个 OSD 有相似长度的管辖范围，需要将每个 OSD 映射到 $[0,1)$ 上的 $K \log P$ 个点， P 为存储集合中 OSD 总数，这时对象分布均匀性的偏差由可调因子 K 控制。该算法的两个缺点是：当存储集合中 OSD 配置差别很大时，不能有效利用配置好

的 OSD；对象的迁移发生在两个 OSD 之间，迁移期间这两个 OSD 可能会成为系统访问瓶颈。

上述的一致性哈希算法是单维的，Hong Tang 提出采用高维一致性哈希算法 (High-Dimension Consistent Hashing, HDCH)^[87]来存放存储系统中的小对象。它将距离关系建立在高维空间上。基本思想是：将对象和存储集合包含的 OSD 都映射为 n 位整数，然后将对象存储在与具有最近汉明(hamming)距离的 OSD 上。如果一个对象与多个 OSD 有相同的汉明距离，则将该对象存储在与具有最小异或结果的 OSD 上。HDCH 相比传统一致性哈希算法的优点是其能更均匀的在存储集合的 OSD 之间分布对象。HDCH 和传统一致性哈希算法的计算开销随存储集合中 OSD 总数线性增长。图 4.9 给出当存储集合中 OSD 总数为 3，n 为 4 时决定一个对象所存储的 OSD 的例子。

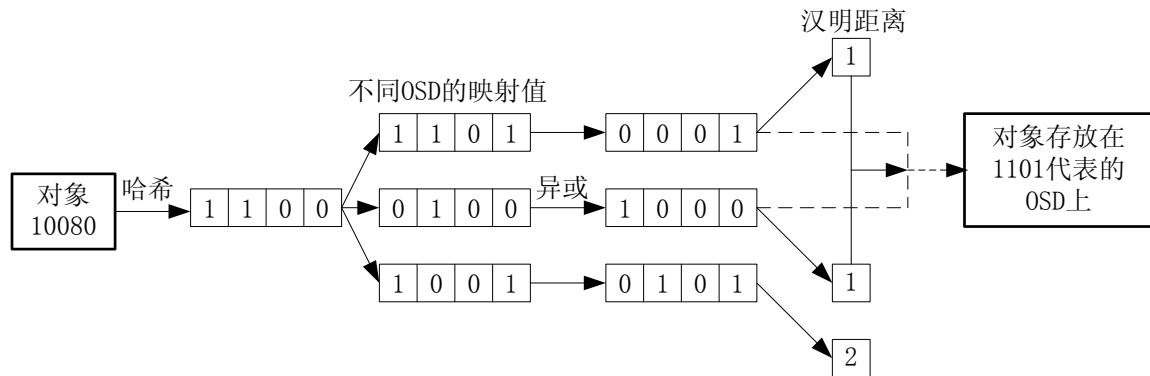


图 4.9 HDCH 决定对象所在的 OSD，其中 n=4

4.4.3 Straw Bucket 算法

CRUSH^[86]中提出了四种不同的原子性对象放置策略，其中包括了 straw bucket 算法。Straw bucket 算法允许存储集合内所有 OSD 根据它们的存储能力来决定对象被存储到哪个 OSD 中，类似于比较绳子的长短，长的绳子获胜。其算法描述如下：

$$C(r, x) = \max_i(f(w_i)\text{hash}(x, r, i))$$

其中 x 表示对象标识符，r 表示对象副本号，i 表示存储集合内 OSD 的标识符 (ID)， w_i 表示 ID 为 i 的 OSD 的存储能力。

采用该算法，有更强存储能力的 OSD 有可能存储更多的对象。它的计算开销随存储集合内 OSD 总数线性增长。该算法能将对象按照 OSD 的存储能力来分布，并且当存储集合内 OSD 总数发生变化时能迁移近似最优的对象数。当存储规模较小，且要求在存储规模变化时迁移尽可能少的对象时，采用该算法比较合适。在后

面提出的基于组的区分定位策略中，由于子集群总数较少，且子集群数变化并不频繁，而且要在子集群数变化时在子集群之间迁移尽可能少的对象，采用该算法完成对象到子集群的映射。

4.4.4 基于组的区分定位策略详细描述

在大规模存储系统中，存储系统的存储配置并不是固定不变的，通常会根据存储系统的需求，适时的加入若干新的 OSD 或淘汰若干旧的 OSD，同批加入或淘汰的 OSD 通常硬件配置相同，不同批次加入或淘汰的 OSD 的硬件配置通常不同(如近期加入的 OSD 通常比以前加入的 OSD 的硬件配置要好，反映为 OSD 权重不同)。根据 OSD 的不同批次及配置将其组织成子集群，并将这种信息反映为存储系统中的设备层次描述信息，其描述如图 4.10 所述。

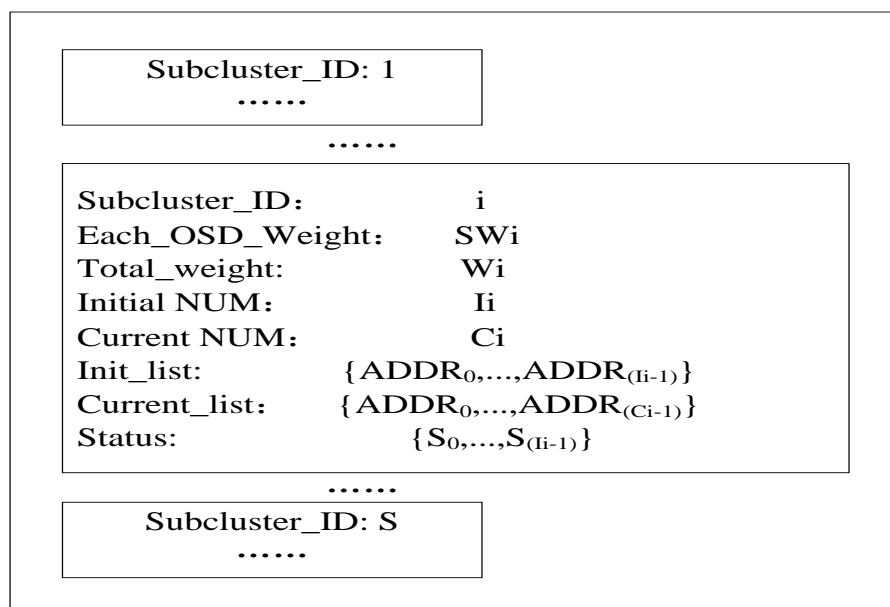


图4.10 存储系统中的设备层次描述信息

其中“Subcluster_ID”指出该子集群在存储系统中的标识符；“Each_OSD_Weight”指出该子集群内每个 OSD 的权重(用来说明其存储能力)；“Total_weight”指出该子集群所有 OSD 的总权重；“Initial num”指出当该子集群加入到存储系统中时该子集群的最初 OSD 总数，“Init_list”对应该子集群的最初 OSD 地址信息列表；“Current num”指出该子集群包括的当前 OSD 总数(相比才加入时，可能有 OSD 已经从该子集群中移出或后期加入该子集群)，“Current_list”对应该子集群当前 OSD 地址信息列表；Status用来说明该子集群内 Init_list 中每个 OSD 的状态。从描述中可以发现，存储系统中的设备层次描述信息中的大部分信息来源

于改进哈希算法中的存储集合描述信息。

在将对象映射到子集群的过程中，采用分布式算法直接定位，避免采用启发式方法导致的大量的设备监控开销和记录对象映射的内存开销和每次定位对象的查找开销。

当将对象映射到特定的子集群后，在子集群内部，根据该对象是大对象或者小对象再分别采用启发式方法和分布式算法定位。因为分配到每个子集群内的对象总数相比系统总对象数已经少了很多，并且大对象比小对象少很多，而且大对象占用了大量的存储空间，大对象的迁移会消耗大量的网络带宽，因此采用启发式方法来为大对象选择负载较轻的 OSD 来存放，减少由于采用分布式算法在该子集群内 OSD 总数发生变化时引起的大对象的迁移；子集群内的小对象采用分布式算法来定位，避免了采用启发式方法记录该对象的映射导致的大量的内存开销和定位查找开销。

算法 4.2

选择 OSD 存储对象算法

1. 输入：o：对象标识符；Info：存储系统中的设备层次描述信息
 2. 输出：对象 o 所在的 OSD 地址
 3. 功能：返回存放对象 o 的 OSD 地址
 4. Subcluster $j \leftarrow A1(o, Info)$ //定位对象o所在的子集群j
 5. **if** o.size < threshold **then** //小对象
 6. host $\leftarrow A2(o, Info_j)$ //小对象采用分布式算法A2定位
 7. **return** host
 8. **else** //大对象
 9. host $\leftarrow light_load(Info_j)$ //从子集群j内选择一个轻负载OSD存储对象o
 10. //更新Bloom Filter Array表明对象o存储在子集群j中地址为host的OSD上
 11. addToBloomfilter(o, host, Info_j)
 12. **return** host
 13. **end if**
-

G-Diffloc 中选择 OSD 存储对象的过程描述如算法 4.2 所示。

- (1). 首先根据存储系统中的设备层次描述信息采用分布式算法 A1 将对象映射到确定的子集群中；
- (2). 在该确定的子集群内：
 - (a) 如果该对象为小对象，则采用分布式算法 A2 来确定该对象在该子集群

的哪个 OSD 上;

- (b) 如果该对象为大对象, 则在该子集群内选择负载较轻的 OSD 来存放该对象, 并更新 Bloom Filter Array 中与该 OSD 对应的相应位置。

G-Diffloc 中查找对象所在 OSD 的过程描述如算法 4.3 所示。

- (1). 首先根据存储系统中的设备层次描述信息采用分布式算法 A1 确定对象存储的子集群 j;
- (2). 在该子集群 j 内:
 - (a) 如果该对象为小对象, 则采用分布式算法 A2 来确定该对象在该子集群的哪个 OSD 上;
 - (b) 如果该对象为大对象, 则将对象查询请求随机路由到该子集群内的某个 OSD 上, 在该 OSD 中查询 Bloom Filter Array 以确定该大对象在该子集群内的哪个 OSD 中。

算法 4.3

查找对象所在 OSD 算法

1. 输入: o: 对象标识符; Info: 存储系统中的设备层次描述信息
 2. 输出: 对象 o 所在的 OSD 地址
 3. 功能: 返回存放对象 o 的 OSD 地址
 4. Subcluster $j \leftarrow A1(o, \text{Info})$ //定位对象o所在的子集群j
 5. **if** o.size < threshold **then** //小对象
 6. host $\leftarrow A2(o, \text{Info}_j)$ //小对象采用分布式算法A2定位
 7. **return** host
 8. **else** //大对象
 9. host $\leftarrow \text{lookup_Bloomfilter}(o, \text{Info}_j)$ //查询Bloom Filter Array得知存储o的OSD
 10. **return** host
 11. **end if**
-

下面分别给出每个阶段采用算法或方法的详细说明。

- 对象到子集群的映射算法

大规模存储系统中, 一般是根据存储系统需求一次性的加入若干个新的 OSD 或一次性的淘汰掉一批旧 OSD。对象放置策略除了需要能够按照子集群的存储能力来分配对象外, 还应能在系统的子集群数量发生变化时为了维护算法的一致性, 在子集群之间迁移的对象数近似最优。Straw bucket 算法满足该特点, 因此采用 straw

bucket 算法来完成对象到子集群的映射。

采用 straw bucket 算法来将对象映射到子集群的公式表示为：

$$\text{Subcluster}(x) = \max_i(f(W_i) \times \text{hash}(x, i))$$

其中 i 是子集群的标识符， x 是对象标识符， W_i 是该子集群 i 的总权重。它以对象标识符 x 和子集群标识符 i 为输入计算哈希值，将对象分配到哈希值与子集群权重函数乘积最大的子集群中。

Straw bucket 算法能够在子集群间按子集群权重(即存储能力)均匀分配对象，并能在子集群数量发生改变时迁移最少量的对象，其计算速度随存储系统包含的子集群总数线性增长。由于存储系统中子集群的数量远比 OSD 总数小很多，且子集群规模的增长也比较缓慢，因此在对子集群的选择中用该算法很有效。

● 子集群内的放置

在子集群内对不同类型的对象采用不同的放置策略，对大对象采用启发式方法从子集群中选取负载较轻的 OSD 放置，并采用 Bloom Filter Array 来记录该大对象在子集群内 OSD 的位置，避免采用分布式算法在子集群内 OSD 总数变化时为维护算法一致性导致的大对象迁移浪费大量的网络带宽。由于子集群内部 OSD 总数可能发生变化，采用改进哈希算法来分布和定位小对象，避免采用启发式方法导致的大量的内存开销，改进的哈希算法的计算开销相比 Diffloc^[87]中采用的一致性哈希算法可忽略不计。

a) 子集群内小对象的映射

在 Diffloc 中，小对象的映射采用了一致性哈希算法。一致性哈希算法能在 OSD 之间均匀分配对象，在存储规模发生改变时迁移最少量的对象，但其计算开销随 OSD 总数呈线性增长。

在 G-Diffloc 中，子集群内部 OSD 的配置相同，子集群内 OSD 总数的增量变化不会太大(OSD 失效是经常发生事件，新的 OSD 很少被单独加入到已经存在的子集群中)。简单哈希算法虽能在 OSD 之间均匀分布对象，但当子集群内 OSD 总数发生改变时，简单哈希算法会招致大量的对象迁移。针对子集群内 OSD 规模的变化规律，采用改进哈希算法来处理小对象在子集群内的映射。改进哈希算法既能继承简单哈希算法的计算开销小和均匀分配对象的优点，又能保证在子集群内 OSD 总数增量不大的情况下有近似最优的对象迁移。改进哈希算法描述及性能论证见 4.3 节。

b) 子集群内大对象的映射

在子集群内，采用 Bloom Filter Array 来跟踪大对象存储的 OSD。每个子集群中有一个“主 OSD”来周期性统计该子集群内所有 OSD 的存储空间使用信息，在

剔除一些剩余空间小于均值的 OSD 后，以与剩余 OSD 的剩余空间成正比的概率选择 OSD 来存放对象。假定选择了 ID 为 i 的 OSD 来存放该对象，Bloom Filter Array 将以对象标识符为输入通过 k 个哈希函数映射到位串向量 $V[i]$ 中的 k 个位的值置为 1；进行对象查询时，以对象标识符为输入计算这 k 个哈希函数，判断位串向量 $V[j]$ 的相对应的这 k 个比特位是否都为 1，如果都为 1，则认为该对象被存储在 ID 为 j 的 OSD 上，否则认为该对象没有在 ID 为 j 的 OSD 上。Bloom Filter 的描述见 4.4.1 节。

在每个 OSD 中，跟踪属于该 OSD 本地存储的大对象的 Bloom Filter (BF) 叫做 Master BF (MBF)，它的副本 Slave BF (SBF) 也同时存储在与该 OSD 同属于一个子集群的其他 OSD 上。在子集群内的每个 OSD 中采用一个 Bloom Filter Array (由该 OSD 的 MBF 和该子集群内的其他的所有 OSD 的 SBF 组成，简称 BFA) 来跟踪属于该子集群的大对象所在的 OSD 的位置。大对象的定位请求能在定位到确定的子集群 i 后，在该子集群 i 内随机选择一个 OSD 在其上根据 BFA 来执行成员查询。其示意图如图 4.11 所示，在图中根据 BFA 来确定对象 10060 存储在哪个 OSD 上，根据 BFA 中每个 BF 可得知，该对象仅可能被存储在 ID 为 $j-1$ 的 OSD 上 (因只有 ID 为 $j-1$ 的 OSD 的 BF 中相应的哈希函数映射的位置中每位为 1)。

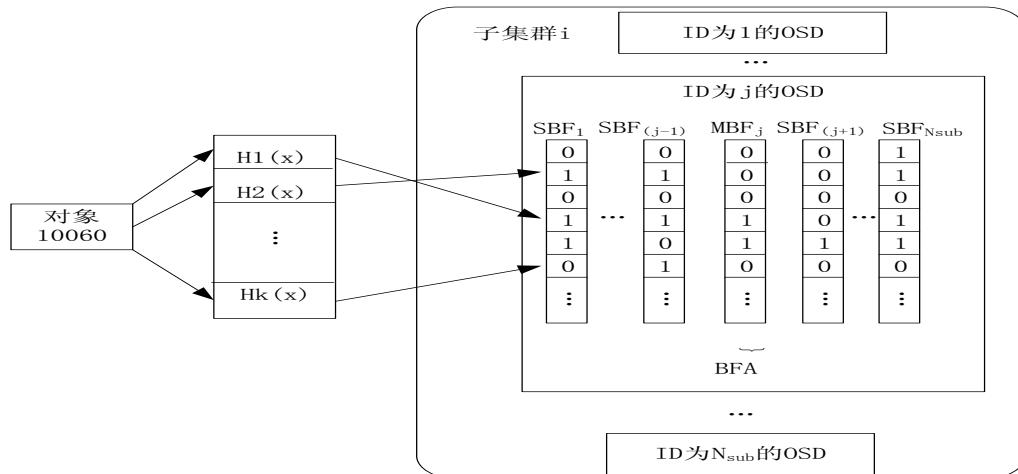


图4.11 在子集群内根据BFA来查找对象所在的OSD

假定系统中总共包含 N 个 OSD，设每个子集群内包含的 OSD 个数为 N_{sub} ，则系统包含的总子集群数为 $S = (N/N_{sub})$ 。设 M_{G-S} 是 G-Diffloc 中每个 OSD 的 MBF 消耗的空间开销， M_S 是 Diffloc 中每个 OSD 的 MBF 消耗的空间开销， $M_{G-Diffloc}$ 是在 G-Diffloc 中所有 OSD 上所有 BFA 消耗的总空间开销， $M_{Diffloc}$ 是在 Diffloc 中所有 OSD 上所有 BFA 消耗的总空间开销，G-Diffloc 与 Diffloc 中由 BFA 消耗的总空间开销之比为 α 。则：

$$M_{\text{Diffloc}} = (M_S \times N) \times N;$$

$$M_{\text{G-Diffloc}} = ((M_{\text{G-S}} \times N_{\text{sub}}) \times N_{\text{sub}}) \times S = M_{\text{G-S}} \times N_{\text{sub}} \times N$$

假定在任意一个 OSD 的 MBF 中，每个对象占用位数为 x ，对象数在存储系统中的所有 OSD 上均匀分配，设系统总共有 y 个对象，则 $M_S = (y/N) \times x$ ； $M_{\text{G-S}} = ((y/S)/N_{\text{sub}}) \times x = (y/N) \times x = M_S$ 。则有：

$$\alpha = (M_{\text{G-Diffloc}}/M_{\text{Diffloc}}) = N_{\text{sub}}/N$$

在 Diffloc 中，每个 OSD 上的 BFA 维护了系统中所有 OSD 的 BF 的全局图。G-Diffloc 中，子集群内的每个 OSD 上仅仅需要维护该子集群内的 OSD 相应的 BF，大大减少了维护这些 BF 的内存和带宽要求。

在 ID 为 j 的 OSD 中，设其 MBF 消耗的空间开销的总位数为 m_j ，该 OSD 上存储了 n 个对象，采用 k 个哈希函数，设 p_j 为其 BF 中某位为 1 的概率，则 $1-p_j$ 为该位为 0 的概率。假设哈希函数取值服从均匀分布，则 p_j 为^[112]：

$$p_j = 1 - (1 - \frac{1}{m_j})^{kn} \approx 1 - e^{-kn/m_j}$$

则误判一个对象属于该 OSD 的概率为 $f_j = p_j^k$

通过求导可得出当 $k = (\ln 2) \times (m_j/n)$ 时误判率 f_j 的值最小为 $(0.6185)^{m_j/n}$ 。

当查找一个对象时，BFA 中的任意一个 BF 的误判都会导致在 BFA 中多次命中，从而导致查找失效，需要继续在子集群内的每个 OSD 上查找。对象的正确查找概率是 BFA 中的所有 BF 都没有误判的概率，即：

在 G-Diffloc 中，系统中对象的正确查找率 $R_{\text{G-Diffloc}}^{\text{BFA}}$ 能被表示为：

$$R_{\text{G-Diffloc}}^{\text{BFA}} = (1 - f_j)^{N_{\text{sub}}} = (1 - (0.6185)^{m_j/n})^{N_{\text{sub}}}$$

在 M_{Diffloc} 中，系统中对象的正确查找率 $R_{\text{Diffloc}}^{\text{BFA}}$ 能被表示为：

$$R_{\text{Diffloc}}^{\text{BFA}} = (1 - f_j)^N = (1 - (0.6185)^{m_j/n})^N$$

由上面两个公式可看出，当 m_j/n 比值一定时，在 G-Diffloc 中子集群内的正确查找率随子集群的规模的变大而减少，在 Diffloc 中，由于 $N > N_{\text{sub}}$ 因此其正确查找率远小于 G-Diffloc 中的对象的正确查找率，因此 G-Diffloc 相比 Diffloc 节省了在 OSD 之间广播查找的网络查找开销。

从以上分析中可以看出，G-Diffloc 相比 Diffloc 既减少了内存开销和网络带宽要求，又大大提高了正确查找率。

真实测试环境中 G-Diffloc 和 Diffloc 的内存消耗与正确查找率的对比见 4.5.1 节。

- 注意事项

当子集群内 OSD 个数发生变化时会导致该子集群的总权重发生轻微变化，由于按照 straw bucket 算法，对象到子集群的映射与每个子集群的总权重相关，需要在子集群之间移动一些对象来达到负载均衡，更改后与更改前的负载差别并不大，但是导致了迁移开销。因此在 G-Diffloc 中，当子集群的总权重变化在一定范围内时，仍按其原来的总权重进行映射。

4.5 基于组的区分定位策略性能评估

G-Diffloc 的设计旨在能在存储系统的 OSD 之间有效的负载均衡，能够快速定位对象，当系统规模发生变化时能尽可能少的迁移对象，同时具有对象分布的灵活性和很好的扩展性。

本节旨在论证 G-Diffloc 的有效性，下面来评估 G-Diffloc 的性能。改进哈希算法的详细性能评估已在 4.3 节中给出，在该节中着重比较 G-Diffloc 和 Diffloc 的整体性能比较，具体包括存储开销以及正确查找率的比较，在存储系统具有不同配置时的定位性能(即计算开销)以及可扩展性的比较。

由于没有现成可用的元数据快照和负载，故只能根据现有系统所收集的元数据快照的分布特性来用人工合成的办法得到应用负载。合成负载中文件大小所依据的分布特性是在 2001 年到 2004 年之间 4 年中从上万个文件系统中获取的元数据快照中获得的^{[11][106]}。由于文件到对象的映射与文件到对象映射策略的选择相关，不同的选择可能使大对象总数与小对象总数之比发生变化。这里旨在对比 G-Diffloc 和 Diffloc 的性能，只需在这两种策略中有着相同应用负载即可，这里考虑简单情况：文件到对象的映射方式是一个文件映射为一个对象。合成的应用负载中包含文件总数有 1×10^5 个，涉及到的文件大小分布特性来源于元数据快照^{[11][106]}。

在该快照中，文件大小小于 512KB 的文件占了文件总数的 90%，这些文件占用的总存储空间却只有所有文件所占存储空间的 20%，测试中我们采用 512KB 作为区分大小对象的界限。

4.5.1 存储开销与正确查找率

由于对象到子集群的映射以及小对象的放置均采用分布式算法，分布式算法具有确定性，只需用到存储系统中的设备层次描述信息，相对于大对象需要的 BFA，其内存开销可忽略不计。由于小对象采用分布式算法来分布和定位，分布式算法的确定性保证其正确查找率为 100%。因此本节主要评估大对象消耗的内存开销以及

其采用 BFA 定位的正确查找率。

前面已经从理论上对比分析了 G-Diffloc 和 Diffloc 中内存开销和正确查找率，下面给出当存储系统包括 2048 个 OSD 时存储系统的内存开销和正确查找率，实验结果分别如图 4.12 和图 4.13 所示，为了便于比较，将内存开销的值正规化到[0,1]区间。具体实现中，将 BFA 的每个大对象占用的位数比(m/n)设为 16，采用 $k=(\ln 2) \times (m/n)=11$ 个哈希函数，测试在不同子集群规模配置时两种放置策略的内存开销以及正确查找率。X 轴为子集群内的 OSD 总数。由于 Diffloc 中没有子集群的概念，其内存开销和正确查找率仅与系统 OSD 总数相关，而与子集群总数以及子集群内 OSD 总数无关，图中它的测试结果仅仅是存储系统包含 OSD 总数为 2048 时的结果。从实验结果中可以看出，当存储系统包含的 OSD 总数一定时，每个子集群内的 OSD 总数越多(即系统中子集群数越少)需要的内存开销越大，它随子集群内的 OSD 数呈线性增长，正确查找率也逐渐降低，当子集群内的 OSD 总数等于系统 OSD 总数时，即系统子集群数为 1 时，G-Diffloc 的内存开销和正确查找率退化到与 Diffloc 一样。从图 4.13 可以看出，在系统 OSD 总数为 2048 个、当子集群内的 OSD 数小于 64 时，G-Diffloc 的正确查找率大于 90%；当子集群内包含的 OSD 数目较多时，正确查找率较低，导致很多查找不能立即定位到正确的 OSD 上，而需要在子集群之间广播或多播，导致大量的网络开销。从该实验中论证了 Diffloc 在大规模存储系统中定位大对象的无效性，同时也说明为了维护定位的有效性和减少内存开销，G-Diffloc 系统中子集群内包含的 OSD 数不要过多。

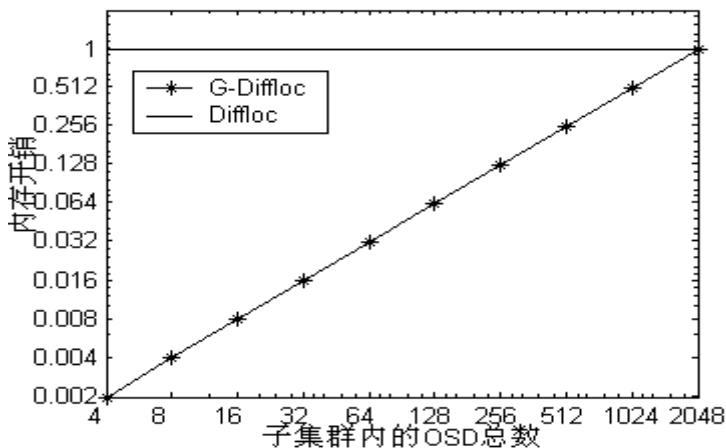


图4.12 G-Diffloc和Diffloc的总内存消耗与每个子集群内OSD总数的关系

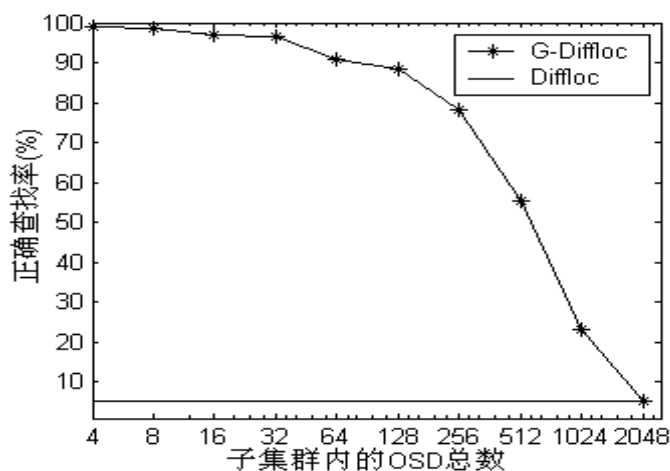


图4.13 G-Diffloc和Diffloc的正确查找率与每个子集群内OSD总数的关系

4.5.2 计算开销与可扩展性

下面给出在不同系统配置和子集群规模中，G-Diffloc 与 Diffloc 在定位大、小对象的平均时间(即计算开销)比较，为了便于比较，将其值正规化到[0,1]区间。

下面主要从两个方面比较 G-Diffloc 和 Diffloc 的定位性能。首先在系统包含 OSD 总数固定的情况下，给出随子集群内 OSD 总数的不同 G-Diffloc 和 Diffloc 的性能比较；其次假定系统规模不断扩展，最初系统中包含 OSD 总数为 32，每次一次性增加 32 个 OSD 到该系统中(在 G-Diffloc 中，则认为每次扩充的 32 个 OSD 属于一个新的子集群)，总共增加了 63 次，最终系统中包含 OSD 总数为 2048 的情况下，分析在系统规模逐渐扩大情况下 G-Diffloc 和 Diffloc 的性能比较，分析了这两种策略的可扩展性。

- 系统包含 OSD 总数固定

设存储系统中包含 2048 个 OSD，在图 4.14 中给出 G-Diffloc 的大、小对象计算开销随子集群内包含 OSD 数目不同的变化情况，Diffloc 中大、小对象的计算开销仅仅与存储系统中 OSD 总数相关，图中它的测试结果仅仅是 OSD 总数为 2048 时的结果。从实验结果中可以看出，G-Diffloc 中的小对象的计算开销总是小于 Diffloc 中的计算开销；在绝大多数情况下，G-Diffloc 中的大对象的计算开销小于 Diffloc 中的计算开销，当每个子集群内包含的 OSD 数目非常少即存储系统中子集群数目非常多时(如每个子集群中仅仅包含 1 或 2 个 OSD 时)，G-Diffloc 中的大对象的计算开销超过了 Diffloc 中的计算开销，这是因为当子集群中包含的 OSD 数非常少时，系统中包含的子集群数非常多，导致大对象定位到子集群的计算开销很大，G-Diffloc 在子集群内的 BFA 中的查找开销虽小于 Diffloc 的 BFA 的开销，但

G-Diffloc 中大对象的总计算开销超过了 Diffloc。G-Diffloc 中，小对象的计算开销随子集群内的 OSD 数的增加而减少，大对象的计算开销开始随子集群内的 OSD 数的增加而减少，后来又继续增加，这是因为当子集群内 OSD 数很少时，系统包含的子集群数就很大，straw bucket 算法的计算开销较大；当子集群内 OSD 数很大时，虽然 straw bucket 算法的计算开销变小，但 BFA 的查找开销变大，因此当在子集群内包含 OSD 数比较适中时，G-Diffloc 中大对象的定位有更优的性能。

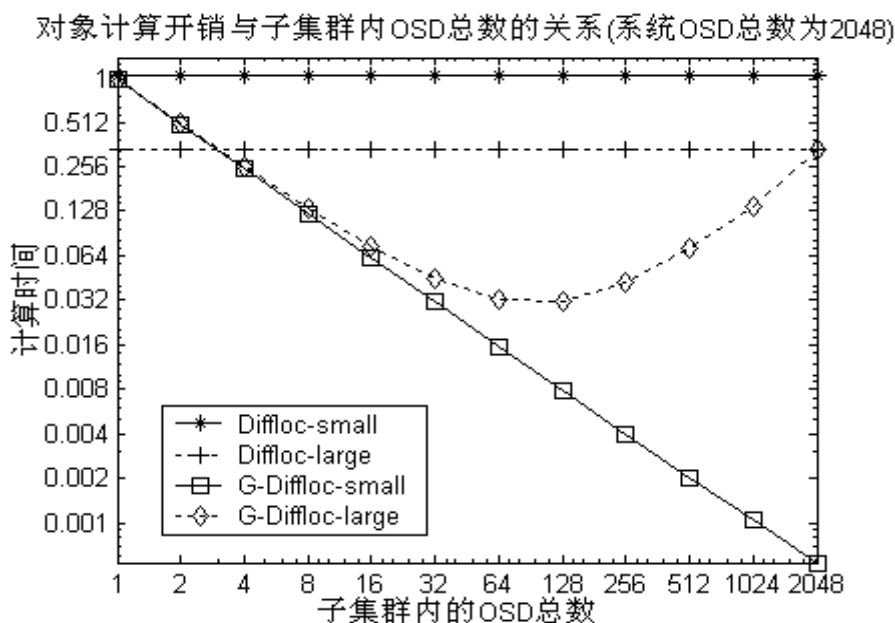


图4.14 当存储系统包含OSD总数固定(2048)时G-Diffloc和Diffloc的性能比较

● 系统规模逐渐扩大

下面给出系统规模逐渐增大时，G-Diffloc 和 Diffloc 的性能比较。设最初系统中包含 OSD 总数为 32，假定每次一次性增加 32 个 OSD 到该系统中(在 G-Diffloc 中，则认为每次扩充的 32 个 OSD 属于一个新的子集群)，总共增加了 63 次，最终系统中 OSD 总数为 2048 个，给出 G-Diffloc 和 Diffloc 在系统规模逐渐变大的情况下性能比较，分析了两种策略的可扩展性，其结果如图 4.15 所示。从结果中可以看出，G-Diffloc 定位大、小对象的性能始终优于 Diffloc 的性能。Diffloc 中定位小对象采用一致性哈希算法，其计算开销随系统中 OSD 总数线性增长，在 G-Diffloc 中定位小对象先采用 straw bucket 算法定位其所在的子集群、再在子集群内采用改进哈希算法定位所在 OSD，straw bucket 算法计算开销随系统包含的子集群数线性增长，改进哈希算法计算开销是常数级(并且是纳秒级)，因此 G-Diffloc 中小对象计算开销随系统包含子集群数线性增长，因此其增量比率远小于 Diffloc 中小对象的增长比率。Diffloc 中定位大对象采用 BFA，时间开销也随系统包含的 OSD 总数线性增

长；在 G-Diffloc 中由于子集群内包含的 OSD 数目固定，子集群内的 BFA 定位开销相对固定，大对象定位开销主要由 straw bucket 算法开销决定，因此其大对象的计算开销也随系统包含的子集群数线性增长，其增量比率也小于 Diffloc 中大对象的增长比率。从实验结果中可以看出，G-Diffloc 算法的可扩展性要远好于 Diffloc。

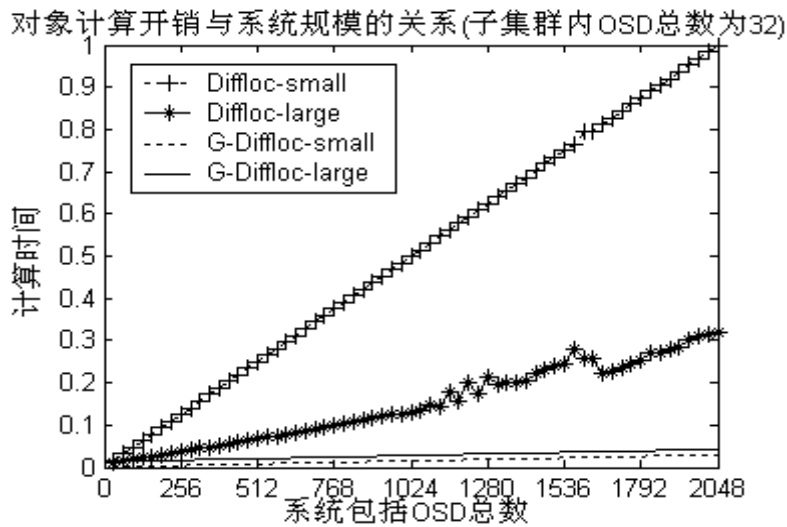


图4.15 当系统规模逐渐扩展时G-Diffloc和Diffloc的性能比较

从上述实验中可以发现在子集群内包含 OSD 数目非极端情况(如每个子集群内只包含 1 或 2 个 OSD)下，G-Diffloc 定位大、小对象性能总是优于 Diffloc 的性能。并且随着系统规模的扩展，G-Diffloc 的可扩展性要高于 Diffloc。

4.6 本章小结

针对存储系统不同规模提出两种不同的放置策略。

静态放置策略用于在存储系统规模较小、OSD 总数固定的情况下利用遗传算法根据文件的不同特性求解文件放置，将文件分割为一个或多个对象存放到不同的 OSD 上，在利用了 OSD 并行性带来的好处的同时也减少了不必要的文件分割引入的系统开销，寻求系统开销与性能的近似最优解。通过实验结果表明，采用遗传算法求解文件分布问题是行之有效的。

在大规模存储系统中，OSD 数量成百上千，存储系统规模动态多变，文件访问特性也灵活多变。基于存储系统规模的变化趋势以及大、小对象的不同特性，提出一种动态放置策略——基于组的区分定位策略。首先根据 OSD 加入存储系统的不同批次将其组织成子集群，先采用分布式算法 straw bucket 将对象映射到存储系统的某个子集群中，再在子集群内部根据不同类型的对象采用不同的放置方法，对大对象采用启发式方法从子集群中选取负载较轻的 OSD 放置，并采用 BF 来记录该大

对象在子集群内 OSD 的位置，对小对象采用改进哈希算法来放置。改进哈希算法是基于子集群内 OSD 规模的变化规律提出的一种新的分布式算法，它既能继承简单哈希算法的计算开销小和均匀分配对象的优点，又能保证在子集群内 OSD 总数增量不大的情况下有近似最优的对象迁移。最后评估了基于组的区分定位策略的性能，从测试结果中论证了该放置策略的有效性。

在本章中提出的动态放置策略没有考虑对对象副本的放置，这是因为实际上可以通过考虑对象副本号的方式将对象的副本分别映射到不同的子集群中，从而解决了对象副本的放置。

5 元数据可靠性研究

大多数存储系统通过采用备份 MDS 来提供失效接管元数据服务来避免单点失效，每条元数据的不同副本被存储在不同的 MDS 上。在常见的存储系统中，通常每条元数据至少拥有两个副本来保证元数据的可靠性。

在对象存储系统中，充分利用富余的对象接口，提出名为 EAP^[113](扩展属性页, Extending Attributes Page)的方案，通过增加用户对象文件信息属性页来提高元数据可靠性。它不需要额外的硬件配置，且不排斥其他的提高存储系统元数据可靠性的方法，它为提供更高的元数据可靠性提高了很好的补充。

5.1 可靠性设计

在对象存储系统中，存储空间管理的任务交由 OSD^[94]负责，MDS 仅仅负责维护存储系统的全局名字空间，存储文件的描述信息以及空间组织信息(文件到用户对象的映射信息)。

5.1.1 访问流程

对象存储系统主要由 MDS、OSD 和客户端组成。应用程序要访问的文件被分割为一个或多个用户对象，对象标识符由 MDS 在该用户对象被创建时根据该用户对象所属文件的全局标识符生成并分配给该用户对象(用户对象的 128 位对象标识符的高 112 位是文件的全局标识符，后 16 位根据用户对象在该文件内的创建顺序由小到大生成)，在 OSD 中用来标识用户对象。OSD 中存储用户对象数据以及其相关的属性。

MDS 维护的元数据包括：目录转换元数据、文件元数据、文件到对象的映射元数据，对象到 OSD 的映射元数据(若是采用静态放置策略，则 MDS 中维护有该元数据；若采用的是动态放置策略，用户对象所在的 OSD 通过分布式算法或在存储系统子集群的 OSD 中查找 BFA 得到，MDS 中不需存储该元数据；在下面的讨论中若不包含该元数据，则相应的 OSD_IP 字段为空即可)。当客户端需要访问文件时，客户端最先与 MDS 交互来获取它要访问的文件包含的用户对象信息以及存放用户对象的 OSD 信息。MDS 将文件操作转换为一个或多个用户对象操作，并通知客户端该文件包括的用户对象的对象标识符(Object ID)以及这些用户对象所在 OSD 的 IP 地址(OSD_IP)，然后客户端与相应的 OSD 直接交互来完成后续的对象读写操作，

访问文件的操作流程图如图 5.1 所示。

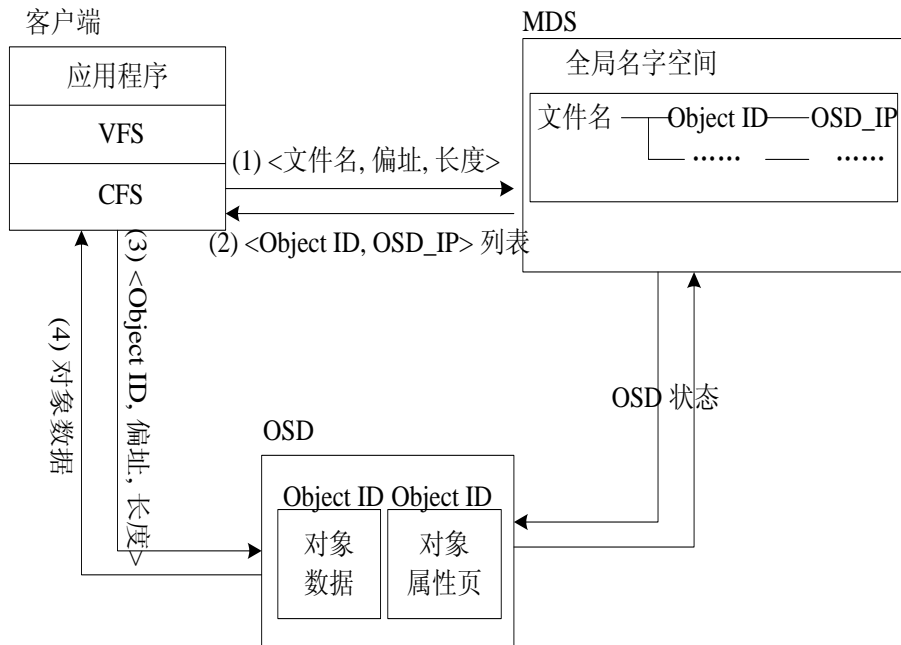


图5.1 访问文件的操作顺序流程图

文件到用户对象的映射过程由 MDS 来完成。在 OSD 中，用户对象仅仅由对象 ID 来标识，它并不知道与该用户对象相关的文件和目录信息，因此一旦 MDS 由于软硬件或网络失效，存储系统就不能提供存储服务，即使此时所有的 OSD 和存储在它们上面的用户对象数据都是完整无缺的。

5.1.2 EAP 方案

由于只有 MDS 知道文件和用户对象之间的映射信息，MDS 是至关重要的，它容易成为单点失效，因此需要寻求一种方法来提供高可靠的元数据服务。大部分存储系统通过提供失效接管元数据服务来消除单点失效，它能够提高存储系统的可靠性。在对象存储系统中可利用用户对象属性来进一步提高元数据的可靠性。

在 EAP 方案中，为每个用户对象增添了一个用户对象文件信息属性页(OFAP)，它定义了与用户对象相关的基本的文件和目录信息属性。EAP 方案是对当前的 OSD 规范的扩充。

表 5.1 用户对象文件信息属性页

属性号	长度 (bytes)	属性
0h	40	页标识
1h	2	分片标识符
2h	2	对象数
3h	2	文件到对象的映射策略
4h	8	第一个对象 ID
5h	4	第一个 OSD IP
...
(2+2×n)h	8	第 n 个对象 ID
(3+2×n)h	4	第 n 个 OSD IP
(4+2×n)h	14	文件父目录的目录标识符
(5+2×n)h	2	文件名长度
(6+2×n)h	可变的	文件名
(7+2×n)h 到 FFFFFFFh		保留

OFAP 中定义的属性如表 5.1 所示，并被描述如下。

1. 分片标识符字段指出该用户对象属于文件的哪一个分片。例如，假定文件 /root/1.avi 被顺序分割映射为三个用户对象：O1, O2, O3，他们被分别存储在 OSD1, OSD2 和 OSD3 上。因此用户对象 O2 的 OFAP 的分片标识符应该是 2（因为 O2 属于该文件的第二个分片）。
2. 对象数字段指定该文件被映像为多少个用户对象。对于上述例子，对象数为 3。
3. 文件到对象的映射策略指定文件如何被映射为用户对象。由此才能根据该属性将相关的用户对象重构为文件。对于以上的例子，文件到对象的映射策略为顺序分割模式。
4. 第一个对象 ID 字段指明该用户对象所属文件的第一个用户对象的对象标识符。对于以上的例子，O2 的 OFAP 的第一个对象 ID 是用户对象 O1 的对象 ID。
5. 第一个 OSD IP 字段用来说明该用户对象所属文件的第一个用户对象所在的 OSD 的 IP 地址信息。在上述例子中，O2 的 OFAP 的第一个 OSD IP 是 OSD1 的 IP 地址。

6. 文件父目录的目录标识符字段用来指出该文件父目录的全局标识符。例如假定该字段值为 6，该文件的全路径名为/root/1.avi，则由此可知目录/root 的全局标识符为 6。
7. 文件名长度字段用来指出文件名字段所占的字节数。例如全路径名 /root/1.avi 的文件名长度为 5。
8. 文件名字段描述了该用户对象所属文件的名称。对于以上的例子，用户对象 O2 的 OFAP 的文件名为 1.avi。该字段的长度可变，由前面的文件名长度字段指定长度，这是因为不同的文件名的长度不同，如果文件名字段长度固定则会浪费存储空间。

表 5.1 给出了 OFAP 的必要字段，用户也能根据需要定义一些选择字段。例如，在支持多种文件类型(如支持符号链接文件)的存储系统中可以定义文件类型字段。这样，一个链接文件的全路径名或者文件名可以被存储在其用户对象数据中，该用户对象的 OFAP 的文件类型为符号链接。

当存储系统的 MDS 失效后，文件元数据、文件到对象的映射元数据，和对象到 OSD 的映射元数据能够通过 OFAP 被访问或重建，提高了存储系统的元数据的可靠性。重构名字空间的过程比较复杂，在重构期间可以采用只读操作来确保名字空间的强一致性。

有人可能认为 EAP 方案的同步开销和存储开销比较大，而事实并非如此。维护扩展属性页需要的同步开销比较小，这是因为仅仅当该用户对象所属的文件有新的用户对象被创建或存在的用户对象被删除时，该用户对象的 OFAP 才需要通过设置属性命令被修改，而在实际系统中绝大多数的对象操作是读写操作。每个用户对象新增的 OFAP 占用的存储空间相比该对象的数据和当前 SNIA OSD 规范^[94]中已经定义的属性页来说是非常小的。

5.2 马尔可夫可靠性模型分析

由于单台机器的平均无故障时间 (Mean Time To Failure, MTTF) 和平均修复时间 (Mean Time To Repair, MTTR) 服从指数分布，因此可以采用连续时间马尔可夫链来分析存储系统元数据的可靠性。

存储系统的平均元数据无故障时间 (Mean Time To Metadata Loss, MTTML) 能采用不同的方法来计算和分析。在下面的分析中，仅仅采用 MDS 来提供元数据服务的方法简称为多 MDS 互备方法，元数据副本从 MDS 中获取。一旦存储某个文件的元数据副本的这些 MDS 均失效，该文件的用户对象就不能被访问到，即使在 OSD

中保存的该文件的用户对象完整无缺。采用 EAP 方案的方法简称为 EAP 方法，在这种环境中，即使所有的 MDS 均失效，系统仍能提供存储服务。

5.2.1 多 MDS 互备方法的 Markov 模型

通常存储系统采用 $P (P \geq 2)$ 个不同的 MDS 来存储元数据的 P 个副本以确保高可靠的元数据服务。一个 MDS 作为元数据的主 MDS 来处理元数据的请求，同时其他的 $P-1$ 个 MDS 作为备份 MDS 来提供失效元数据服务接管并在主 MDS 失效时用做恢复源。

假定存储在 MDS 上的元数据副本的 MTTF 是 $MTTF_1$ ，MTTR 是 $MTTR_1$ ，存储在 MDS 的元数据副本的失效率 λ_1 和恢复率 μ_1 分别为 $1/MTTF_1$ 和 $1/MTTR_1$ 。

当存储某个文件的所有元数据副本的 P 个 MDS 均失效时，即使该文件的所有用户对象均完整无缺也不能被访问到。

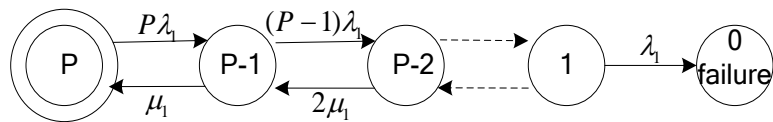


图5.2 多MDS互备方法的Markov模型

图 5.2 中的多 MDS 互备方法的 Markov 模型有 $P+1$ 个状态， P 是元数据副本总数。最初的状态是 State P ，这时有 P 个副本可用；State $P-1$ 时有 $P-1$ 个副本可用、一个副本失效；State 1 时仅仅有一个副本可用；State 0 意味着所有副本均失效，MDS 不能再提供元数据服务。当有 MDS 失效或恢复时引起副本的失效或恢复，从而导致状态之间相互转换。在 State P ， P 个副本均有可能失效，因此从 State P 到 State $P-1$ 的转换概率为 $P\lambda_1$ ；在 State $P-1$ 时，从 State $P-1$ 转换到 State P 的概率是 μ_1 （由于单个失效副本的恢复率是 μ_1 ），从 State $P-1$ 转换到 State $P-2$ 的概率是 $(P-1)\lambda_1$ （由于剩下的 $P-1$ 个副本均有可能失效）；在 State 1 时，从 State 1 转换到 State 0 的概率是 λ_1 （由于只有剩下的 1 个副本可能失效）；State 0 不能转换到 State 1 （由于没有可用的副本用做恢复源）。状态能被分为两类：正常服务状态集 $\{P, P-1, \dots, 1\}$ 和失效服务状态集 $\{0\}$ 。正常服务状态集能提供正常的元数据服务，失效服务状态集不能提供元数据服务。

假定 State P , State $P-1, \dots$, State 0 的概率分别是 $P_{P(t)}$, $P_{P-1(t)}$, \dots and $P_{0(t)}$ ，多 MDS 互备方法的平均元数据无故障时间 $MTTML_1$ 能根据正常服务状态集的状态的概率计算得到，如公式 5.1 所示。

$$\int_0^{\infty} (P_P(t) + P_{P-1}(t) + P_1(t)) dt \quad (5.1)$$

通过求解如下公式 5.2 中的差分方程^{[114][115]}能够得到 $P_{P(t)}$, $P_{P-1(t)}$, \dots 和 $P_{0(t)}$ 的值。

$$[P'_P(t), P'_{P-1}(t), \dots, P'_0(t)] = [P_P(t), P_{P-1}(t), \dots, P_0(t)] \cdot Q_1 \quad (5.2)$$

其中矩阵 Q_1 的值可以从图 5.2 中得到。

矩阵 Q_1 中的元素 q_{ij} ($0 \leq i, j \leq P$) 描述如下。

$$q_{ij} = \begin{cases} i\mu_1 & (j = i - 1, 1 \leq i \leq P - 1) \\ -i\mu_1 - (P - i)\lambda_1 & (j = i, 0 \leq i \leq P - 1) \\ (P - i)\lambda_1 & (j = i + 1, 0 \leq i \leq P - 1) \\ 0 & (\text{otherwise}) \end{cases} \quad (5.3)$$

当值 P 已知时, 通过带入等式 5.3 和状态的初始值 $P_P(0)=1$, $P_{P-1}(0)=\dots=P_0(0)=0$ 能求解出差分方程 5.2。最后由公式 5.1 计算出 $MTTML_1$ 。

5.2.2 EAP 方法的 Markov 模型

当一个文件被映射成 Q 个用户对象时, 假定对该文件提供了 P 个元数据副本(由 P 个 MDS 提供)和 Q 个 OFAP(由 Q 个 OSD 提供)。当所有的 P 个 MDS 均失效时, 只要存储该文件的用户对象的 Q 个 OSD 仍能被访问, 则该文件仍然能被访问到。

假定存储在 MDS 上的元数据副本的失效率和恢复率分别为 λ_1 和 μ_1 , 存储在 OSD 上的 OFAP 的失效率和恢复率分别为 λ_2 和 μ_2 。

图 5.3 中的 EAP 方法的 Markov 模型有 $(P+1) \times (Q+1)$ 个状态, 每个状态由 $\langle x, y \rangle$ ($0 \leq x \leq P, 0 \leq y \leq Q$) 来描述, 其中 x 代表仍有 x 个元数据副本可用, y 表示仍有 y 个 OFAP 可用。最初的状态是 State $\langle P, Q \rangle$, 这时所有的元数据副本和所有的 OFAP 均可用; State $\langle 0, 0 \rangle$ 表示所有的元数据副本和 OFAP 均不可用。当有 MDS 或者 OSD 失效或恢复时引起元数据副本或 OFAP 的失效或恢复, 从而导致状态之间相互转换。在 State $\langle x, y \rangle$ 中, 从 State $\langle x, y \rangle$ 转换到 State $\langle x-1, y \rangle$ 的概率为 $x\lambda_1$ (由于剩下的 x 个元数据副本均有可能失效); 从 State $\langle x, y \rangle$ 转换到 State $\langle x, y-1 \rangle$ 的概率为 $y\lambda_2$ (由于剩下的 y 个 OFAP 均有可能失效); 从 State $\langle x, y \rangle$ 转换到 State $\langle x+1, y \rangle$ 的概率为 $(P-x)\mu_1$ (由于失效的 $P-x$ 个元数据副本均有可能恢复); 从 State $\langle x, y \rangle$ 转换到 State $\langle x, y+1 \rangle$ 的概率为 $(Q-y)\mu_2$ (由于失效的 $Q-y$ 个 OFAP 均有可能恢复)。由于没有可用的副本或 OFAP 用做恢复源, State $\langle 0, 0 \rangle$ 不能转换到任何其他状态。

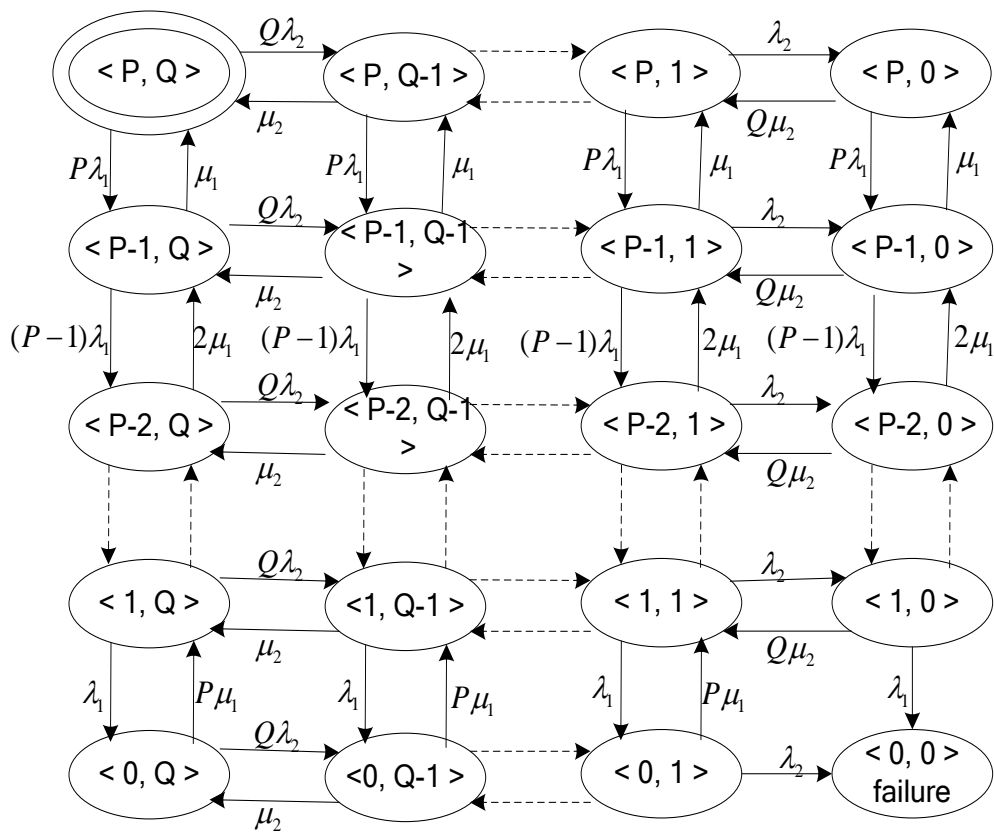


图5.3 EAP方法的Markov模型

采用同样的方法，能计算出 EAP 方法的平均元数据无故障时间 $MTTML_2$ 。

5.3 可靠性分析

由于表达的复杂性和空间的限制，表 5.2 中仅仅给出了 P 和 Q 比较小时的 $MTTML_1$ 和 $MTTML_2$ 的一些结果。

表 5.2(a) 多 MDS 互备方法的 $MTTML_1$

P	$MTTML_1$
1	$\frac{1}{\lambda_1}$

2	$\frac{3\lambda_1 + u_1}{2\lambda_1^2}$
3	$\frac{11\lambda_1^2 + 7\lambda_1\mu_1 + 2\mu_1^2}{6\lambda_1^3}$

表 5.2(b) EAP 方法的 MTTML₂

P\Q	0	1
0	---	$\frac{1}{\lambda_2}$
1	$\frac{1}{\lambda_1}$	$\frac{1}{\lambda_1} + \frac{(\lambda_1 + u_1)(\lambda_1 + u_2)}{\lambda_1\lambda_2(\lambda_1 + \lambda_2 + u_1 + u_2)}$

相对于多 MDS 互备方法，EAP 方法 Markov 模型中的状态转换更复杂一些。实际上，能够看出多 MDS 互备方法 Markov 模型是 EAP 方法 Markov 模型中当 Q 等于 0 的特例。

采用 Markov 模型中计算出的不同方法的 MTTML，能对他们的结果进行比较，如图 5.4 所示。在每种配置中，MTTR₁=MTTR₂=4 小时，MTTML 的结果随 MTTF₁=MTTF₂=MTTF 变化。从图中可以看出，EAP 方法的 MTTML 远远大于多 MDS 互备方法的 MTTML，并随着 Q 值得增加而增加(当 P 固定时)。

由图 5.4(a)可以看出，在多 MDS 互备方法中当仅仅只有一个元数据副本时 MTTML 的值较小，当有多个元数据副本或者采用 EAP 方案时 MTTML 的值会大大增加。

在实际系统中，虽然 MTTML 能随着 P 和 Q 的增大而增加，但是当元数据副本数或 OFAP 数增大时，维护元数据一致性的开销也需要增加，因此 P 和 Q 的值不应太大。从图 5.4(a)和 5.4(b)能看出，当 P=2、Q=1 或 P=1、Q=2 时 MTTML 已经能够满足需求。

根据以上的分析，当某个文件被映射到 n (n≥2)个用户对象时，OFAP 仅仅被包含在该文件的前两个对象中。

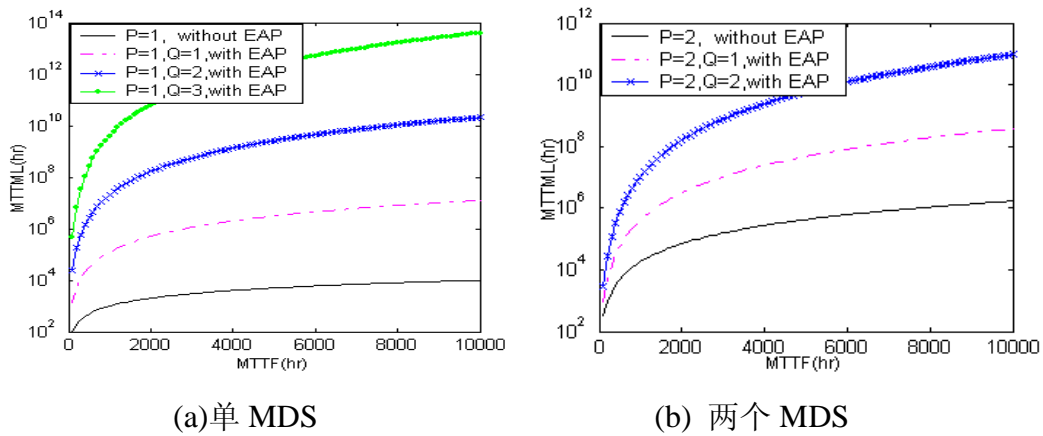


图 5.4 不同配置中 MTTML 随 MTTF 的变化曲线(MTTR₁=MTTR₂=4 小时)

在图 5.4(a)和 5.4(b)中，假定了 MTTR₁ 和 MTTR₂ 相等。而实际上，MTTR₂ 比 MTTR₁ 短，这是由于元数据被集中存放在 MDS 中，而 OFAP 被分散存放在不同的 OSD 中，因此在 EAP 方案中有更多的恢复源和恢复目标来加速元数据恢复的过程。图 5.5 给出了在不同方法中 MTTML 随 MTTR₂/MTTR₁ 的变化情况，这里假定 MTTR₁=4 小时，MTTF₁=MTTF₂=5×10⁴ 小时。从图 5.5 中也可以看出，当 MTTR₁ 等于 MTTR₂ 时，在多 MDS 互备方法中 P=2 时的 MTTML 与在 EAP 方法中 P=1、Q=1 时的 MTTML 相等；当 MTTR₂ 小于 MTTR₁ 时，EAP 方法的 MTTML 通常要比多 MDS 互备方法的 MTTML 大。

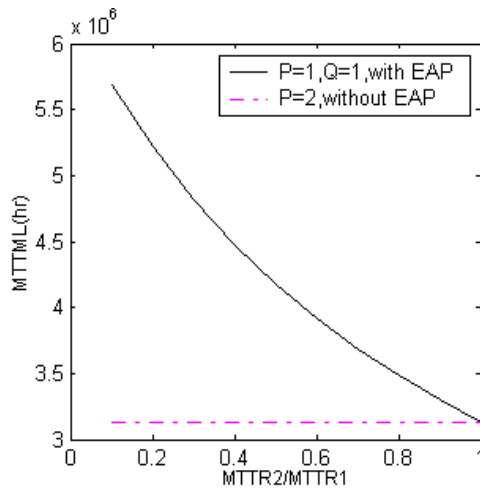


图 5.5 不同配置中 MTTML 随 MTTR₂/MTTR₁ 的变化曲线(MTTR₁=4 小时)

由于 EAP 方法使用了更少的 MDS(即更少的硬件需求) 得到了与多 MDS 互备方法相似的 MTTML 性能，因此 EAP 方法更有优势一些。

在一般情况下，为了提高用户对象的可靠性，存储系统中的用户对象副本数通常不会少于 2，此时如果采用 EAP 方案的话，元数据可靠性也能得到相应的提高。

若不采用 EAP 方案，元数据的可靠性仅仅依赖于元数据所在的 MDS 的个数，在这种情况下，虽然用户对象副本数增加，但是元数据可靠性并没有从中获益。

5.4 本章小结

本章提出采用 EAP 方法来提高元数据可靠性，它不需要额外的硬件配置，而充分利用了对象存储系统的对象接口，通过增加用户对象文件信息属性页来提高元数据可靠性。在 EAP 方法中，当 MDS 失效时，用户对象文件信息属性页临时提供元数据服务接管，大大提高了存储系统元数据的可靠性。

本章中，分别为采用 MDS 以及采用 EAP 方法来提高文件元数据的可靠性建立了 Markov 模型进行分析。从这两种不同方法的 MTTML 的比较中可以发现采用 EAP 方法的存储系统元数据可靠性要好于多 MDS 互备方法的元数据可靠性。

总之，通过增加用户对象文件信息属性页，存储系统元数据的可靠性能够大大提高。同时，EAP 方案并不排斥其他的提高存储系统元数据可靠性的方法，它为提供更高的元数据可靠性提供了很好的补充。

6 全文总结

对象存储系统 (Object Based Storage System, OBSS) 是一种具有高性能、高可扩展性、跨平台以及安全数据共享的存储体系结构, 它采用了一种全新的接口——对象接口, 有效综合了块接口与文件接口二者的优点, 并能提供比其他任何一种接口更为丰富的语义。对象接口访问的基本单位是对象, 对象除了包含用户数据外, 还包含能描述对象特征的属性, 其中对象的属性可使 OBSS 更好地组织数据和提供服务, 使得系统的灵活性和可管理性得以提高。

文件系统主要由两大部分组成: 用户组件部分和存储管理组件部分。在 OBSS 中, 存储管理组件部分由对象存储设备(Object-based Storage Device, OSD)负责管理, 元数据服务器(MetaData Server, MDS)主要对 OBSS 的用户组件部分进行管理, MDS 的负担已经大大减轻, 但随着系统规模的扩大, MDS 仍会成为系统访问瓶颈, 需要研究高性能、可扩展的元数据管理方法。在 OBSS 中, 数据放置策略负责将文件映射为对象, 并为新创建的对象选择合适的 OSD 存放, 客户端访问对象时, 需要快速定位到对象所在的 OSD, 数据放置策略对系统访问性能有关键性的影响, 需要根据系统的规模选择合适的数据放置策略。数据放置策略引入的文件到对象的映射元数据由 MDS 集中管理; 对象到 OSD 的映射元数据根据采用的数据放置策略的不同, 可能由 MDS 集中维护或者分布存放在 OSD 上的映射表构成, 也有可能通过分布式算法直接计算获得。MDS 中的元数据记录了文件和目录信息与对象之间的关系, 元数据的丢失将导致数据无法访问, 因此 MDS 中的元数据的可靠性维护至关重要。本文的工作主要围绕 OBSS 中元数据管理展开, 取得研究成果如下:

1. 提出一种分布式元数据管理方案, 提供高性能和可扩展的元数据访问。

MHS (Mimic Hierarchical directory Structure) 方案采用四种方法和技术来提供高性能和可扩展的元数据服务, 分别是: 仿层次目录结构, 目录转换元数据, 针对不同类型元数据的不同访问特性提供不同的在 MDS 集群中灵活分布元数据的方法, 以及高效的元数据存储方式。在深入分析传统文件系统中与用户组件部分相关的元数据组成结构的基础上, 结合数据库能提供高处理事务吞吐量的特点, 提出一种改进的元数据存储和管理方法, 提高访问速度; 在系统中不再用持久存储(如磁盘)来存储记录文件名到索引节点号映射关系的目录数据, 而是采用一种间接的方案来模拟层次目录结构, 避免层次目录结构自身成为热点, 从而提供高性能、可扩展的元数据访问; 引入目录转换元数据来避免子树分割方案中的目录遍历和哈希方案中的重命名目录导致的大量元数据迁移, 提高元数据总体访问性能; 针对每种元数据

自身的访问特性，采用不同的分割方法将其分布在 MDS 集群中，方便系统规模扩展，如对目录转换元数据按照记录关键字的字典序分割，对文件元数据采用元数据查找表来分割。在实验中采用详细的 micro-benchmark 评估 MHS 方案在不同操作中的元数据访问性能，并将其与 LH 方案进行性能比较，论证了 MHS 方案的有效性；然后给出了 MHS 方案在不同 MDS 集群规模下的元数据性能，论证了 MHS 方案的可扩展性。

2. 提出一种应用于 MDS 集群的负载均衡算法，在 MDS 之间均衡负载。

为了在 MDS 集群中提供高性能、可扩展的元数据服务，需要在 MDS 之间均衡负载。提出一种以文件元数据请求的响应时间为衡量标准、应用于 MDS 集群的负载均衡算法。该算法首先从映射算法上实现 MDS 集群的静态负载均衡。由于元数据访问负载随时间动态变化，元数据在 MDS 集群中的静态分配可能会导致某一时刻某个 MDS 成为系统元数据访问性能的瓶颈，此时引入动态负载均衡，将负载重的 MDS 上的部分文件元数据转移到负载轻的 MDS 上。试验结果表明，在文件元数据访问频率差别较大的情况下，启动动态负载均衡算法能很大程度地减少 MDS 集群中的 MDS 的响应时间的差别，大大减少系统的响应时间方差。

3. 针对不同的应用环境提出合适的数据放置策略。

针对系统规模较小、OSD 总数固定的应用环境，提出一种利用遗传算法根据文件的不同特性求解数据放置的策略，寻求系统性能的近似最优解。

针对系统规模较大且 OSD 总数可能发生变化的应用环境，提出 G-Diffloc（基于组的区分定位策略）。由于系统中数量众多的小对象占据少量存储空间、数量较少的大对象占据大量存储空间，小对象更适合采用分布式算法、大对象更适合采用启发式方法来放置。基于系统规模的变化趋势，G-Diffloc 首先根据 OSD 加入系统的不同时期将每个 OSD 划分到不同的存储子集群，先采用分布式算法将对象映射到系统的某个子集群中，再在子集群内部根据不同类型的对象采用不同的映射方法，对新创建的大对象采用启发式方法来选择负载较轻的 OSD 存放，对小对象采用改进哈希算法来直接计算出其所在的 OSD，兼顾了对象分布的灵活性和系统可扩展性。在实验中，将 G-Diffloc 与 Diffloc 的整体性能进行比较，具体包括存储开销以及正确查找率的比较，各种不同系统配置（OSD 总数固定而子集群数变化，或子集群内 OSD 总数固定而系统规模不断扩展）时的定位性能（即计算开销）以及可扩展性的比较，实验结果论证了 G-Diffloc 的有效性，表明其具有很好的性能和可扩展性。

改进哈希算法是基于子集群内 OSD 规模的变化规律提出的一种新的分布式算法。实验表明，改进哈希算法继承了简单哈希算法的计算开销小和均匀分配对象的

优点。改进哈希算法在当存储集合内 OSD 数目添加较少或 OSD 数目减少时有非常好的性能，它能以近似最优的对象迁移开销有效支持子集群内的 OSD 规模的变化。

4. 提出一种采用扩展属性页来提高元数据可靠性的方法，它利用 OBSS 富有表达力的对象接口来提高系统元数据可靠性。

在 OBSS 中，对象采用 128 位的对象标识符标识，它不了解与其相关的文件或目录信息，一旦 MDS 失效，系统就不能提供存储服务，即使此时所有的 OSD 和存储在它们上面的对象都是完整无缺的。MDS 中的元数据的可靠性的维护至关重要。利用 OBSS 富有表达力的对象接口设计采用扩展属性页来提高元数据可靠性的方法能够充分利用对象存储的优势来保障元数据的高可靠性。采用 Markov 模型对其可靠性进行分析，实验结果表明该方法能大大提高存储系统元数据的可靠性。该方法不需要额外的硬件配置，且不排除其他的提高存储系统元数据可靠性的方法，为提供更高的元数据可靠性提供了一种补充方案。

本文对 OBSS 中元数据管理进行相关研究，结合本人的研究工作进展，今后的研究工作将在以下几个方面进行：

1. 元数据预取技术

在 OBSS 中，当访问某个文件时，客户端须先访问 MDS 获取文件元数据，再从相应的 OSD 读写文件数据。对于小文件来说，获取文件元数据的时间和获取文件数据的时间在同一数量级，使得用户访问延迟加倍。由于文件访问具有一定的相关性，可以在访问某个文件元数据的时候将与其关联的一组文件元数据一起预取到客户端缓存，这样既减少了客户端与 MDS 之间的频繁通信，减轻了 MDS 的负载，又减少了用户平均访问延迟。

2. 文件元数据缓存一致性

为提高文件元数据访问性能，客户端缓存有部分文件元数据，需要维护文件元数据多副本之间的一致性，避免客户端访问到无效的文件元数据。

3. 元数据查询技术

在大规模 OBSS 中，MDS 承担着面向 PB 级海量信息的查询任务。伴随着互联网应用的蓬勃发展，MDS 所需完成的任务已不再是简单地获取文件元数据，而是要实现面向多维属性元素的多种信息查询服务。在面向 PB 级海量信息的 OBSS 中，元素通常是具有多维属性的，元素属性是多维的特点就必然要求 MDS 也要能够提供面向多维属性的信息查询服务，而且这种信息查询服务应当是快速的、准确的和个性化的，即能够根据用户个性化的查询请求提供相应的查询服务。

参考文献

- [1] John F. Gantz, Christopher Chute, Alex Manfrediz, et al. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. IDC white paper, sponsored by EMC, March 2008.
- [2] 张江陵, 冯丹. 海量信息存储. 北京: 科学出版社, 2003. 2~6
- [3] P Xia, D Feng, H Jiang, et al. FARMER: A novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale file system performance. In: Proceedings of the 17th international symposium on High performance distributed computing. 2008. 185~196
- [4] DK Gifford, P Jouvelot, MA Sheldon, et al. Semantic file systems. ACM SIGOPS Operating Systems Review, 1991, 25(5): 16~25
- [5] Y Padioleau, B Sigonneau, O Ridoux. Lisfs: a logical information system as a file system. In: Proceedings of the 28th international conference on Software engineering. 2006. 803~806
- [6] PA Chirita, S Costache, W Nejdl, et al. Beagle++:Semantically enhanced searching and ranking on the desktop. In: Proceedings of The Semantic Web: Research and Applications: 3rd European Semantic Web Conference. 2006. 348~362
- [7] D Ellard, J Ledlie, P Malkani, et al. Passive NFS Tracing of Email and Research Workloads. In: Proceedings of the 2nd USENIX Conference on File and Storage Technologies. 2003. 203~216
- [8] D Ellard, M Mesnier, E Thereska, et al. Attribute-based prediction of file properties. Technical Report TR-14-03, Harvard, 2004
- [9] M Mesnier, E Thereska, GR Ganger, et al. File classification in self-* storage systems. In: Proceedings of the First International Conference on Autonomic Computing. 2004. 44~51
- [10] AW Leung, S Pasupathy, G Goodson, et al. Measurement and analysis of large-scale network file system workloads. In: USENIX 2008 Annual Technical Conference on Annual Technical Conference table of contents. 2008. 213~226
- [11] N Agrawal, WJ Bolosky, JR Douceur, et al. A five-year study of file-system metadata. In: Proceedings of the 5th USENIX conference on File and Storage Technologies. 2007. 31~45
- [12] M. Rosenblum, JK. Ousterhout. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems, 1992, 10(1):26~52
- [13] D. Hitz, J. Lau, M. Malcom. File system design for an NFS file server appliance. In: Proceedings of the Winter 1994 USENIX Conference. 1994. 235~246
- [14] MK McKusick, WN Joy, SJ Leffler, et al. Fsck-The UNIX File System Check Program. Unix System Manager's Manual-4.3 BSD Virtual VAX-11 Version. 1986

- [15] V Henson, A van de Ven, A Gud, et al. Chunkfs: using divide-and-conquer to improve file system reliability and repair. In: Proceedings of the 2nd conference on Hot Topics in System Dependability. 2007. 7~12
- [16] HS Gunawi, A Rajimwale, AC Arpaci-Dusseau, et al. SQCK: A Declarative File System Checker. In: 8th USENIX Symposium on Operating Systems Design and Implementation. 2008 . 131~146
- [17] 叶卫东, 曾哲昱. SCSI 技术及接口设计方法. 测控技术, 2002, 21(3): 58~60
- [18] F.Schmidt. SCSI 总线和 IDE 接口: 协议、应用和编程(第二版). 精英科技译.北京: 中国电力出版社, 2001. 63~76, 129~161
- [19] David Sacks. Demystifying DAS, SAN, NAS, NAS Gateways, Fibre Channel, and iSCSI. IBM Storage Consultant, March 2001: 6~10
- [20] D. Nagle, G. Ganger, J. Butler, et al. Network Support for Network-Attached Storage. In: Proceedings of Hot Interconnects 1999. 1999. 18~20
- [21] Luo Xinguo, Jiangling Zhang. Study on a Network Storage System. In: Proceedings of International Symposium on Multidisciplines, China, 1992. 151~155
- [22] Robert Gray, Bill North, Vernon Turner. Storage Network Management and Virtualization. IDC, August 2002: 1~5
- [23] T. Clark. Designing Storage Area Networks: A Practical Reference for Implementing Fibre Channel and IP SANS (Second Edition). Addison-Wesley Networking Basics Series, 2003.
- [24] Mesnier M, Ganger G R, Riedel E. Object-based storage. Communications Magazine, IEEE, 2003, 41(8): 84~90
- [25] Mesnier M, Ganger G R, Riedel E. Object-based storage: pushing more functionality into storage. Potentials, IEEE, 2005, 24(2): 31~34
- [26] A. Azagury, V. Dreizin, M. Factor, et al. Towards an object store. In: Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies. 2003. 165~176
- [27] 覃灵军. 基于对象的主动存储关键技术研究: [博士学位论文]. 武汉: 华中科技大学图书馆, 2006.
- [28] Y Lu, DHC Du, T Ruwart. Qos provisioning framework for an osd-based storage system. In: Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies. 2005. 28~35
- [29] 赵水清, 冯丹. 对象存储设备上的服务质量研究. 计算机科学, 2006, 33(9): 89~92
- [30] Peter J Braam. The Lustre Storage Architecture. Cluster File Systems, Inc. Whiter Paper. <http://www.clusterfs.com>. 2003
- [31] B Welth, M Unangst, Z Abbasi, et al. Scalable performance of the panasas parallel file system. In: Proceedings of the Sixth USENIX Conference on File and Storage Technologies. 2008. 17~33

- [32] S.A.Weil, S.A. Brandt, E.L. Miller, et al. Ceph: a scalable, high performance distributed file system. In: Proceedings of the Seventh USENIX Symposium on Operating Systems Design and Implementation. 2006. 307~320
- [33] S Ghemawat, H Gobioff, ST Leung. The Google file system. ACM SIGOPS Operating Systems Review, 2003, 37(5): 29~43
- [34] B Pawlowski, C Juszczak, P Staubach, et al. NFS version 3: Design and implementation. In: Proceedings of the Summer 1994 USENIX Conference.1994. 137~151
- [35] JH Morris, M Satyanarayanan, MH Conner, et al. Andrew: A distributed personal computing environment. Communications of the ACM, 1986, 29(3): 184~201
- [36] M Satyanarayanan, JJ Kistler, P Kumar, et al. Coda: A highly available file system for a distributed workstation environment. IEEE Transactions on Computers, 1990, 39(4): 447~459
- [37] JK Ousterhout, AR Cherenon, F Dougliis, et al. The Sprite network operating system. IEEE Computer, 1988, 21(2): 23~36
- [38] SA Weil, KT Pollack, SA Brandt, et al. Dynamic metadata management for petabyte-scale file systems. In: Proceedings of the 2004 ACM/IEEE conference on Supercomputing. 2004. 523~534
- [39] PF Corbett, DG Feitelson. The Vesta parallel file system. ACM Transactions on Computer Systems, 1996, 14(3): 225~264
- [40] PJ Braam, M Callahan, P Schwan, et al. The intermezzo file system. In: Proceedings of the 3rd of the Perl Conference.1999
- [41] EL Miller, RH Katz. RAMA: An easy-to-use, high-performance parallel file system. Parallel Computing, 1997, 23(4): 419~446
- [42] O Rodeh, A Teperman. zFS - a scalable distributed file system using object disks. In: Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies. 2003. 207~218
- [43] SA Brandt, L Xue, EL Miller, et al. Efficient metadata management in large distributed file systems. In: Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies. 2003. 290~298
- [44] 刘仲, 周兴铭. 基于目录路径的元数据管理方法. 软件学报, 2007, 18(2): 236~245
- [45] Y Zhu, H Jiang, J Wang. Hierarchical bloom filter arrays (HBA): a novel, scalable metadata management system for large cluster-based storage. In: Proceedings of 2004 IEEE International Conference on Cluster Computing. 2004. 165~174
- [46] B Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970, 13(7): 422~426
- [47] Gremilion L L. Designing a Bloom filter for differential file access. Communications of ACM, 1982, 25(9): 600~604

- [48] Mullin J k. A Second Look at Bloom Filters. *Communications of ACM*, 1983, 26(8): 570~571
- [49] Bonomi F, Mitzemacher M, Panigraphy R, et al. Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines. In: *Proceedings of ACM SIGCOMM 2006*. 2006. 315~326
- [50] Y Hua, Y Zhu, H Jiang, et al. Scalable and adaptive metadata management in ultra large-scale file systems. In: *Proceedings of the 28th International Conference on Distributed Computing Systems*. 2008. 403~410
- [51] CK Wong. Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems. *Computing Surveys*, 1980, 12(2): 167~178
- [52] Keith A. Smith, Margo Seltzer. A comparison of FFS disk allocation policies. In: *Proceedings of the USENIX 1996 Annual Technical Conference*. 1996. 15~25
- [53] G. R. Ganger, M. F. Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. In: *The 1997 USENIX Annual Technical Conference*. 1997. 1~17
- [54] D.Patterson, G. Gibson and R. Katz. A Case for Redundant arrays of Inexpensive Disks (RAID). In: *Proceedings of the ACM SIGMOD'88*. 1988. 109~116
- [55] P.M.Chen, E.K.Lee, G.A.Gibson, et al. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 1994, Vol.26(2): 145~185
- [56] H. Jin, K. Hwang. Stripped Mirroring Disk Array. *Journal of Systems Architecture, Elsevier Science*, 2000, 46(6): 543~550
- [57] M Holland, GA Gibson. Parity Declustering for Continuous Operation in Redundant Disk Arrays. In: *Proceedings of the 5th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.1992. 23~35
- [58] P. M. Chen, E. K. Lee, A. L. Drapeau, et al. Performance and Design Evaluation of the RAID-II Storage Server. *Journal of Distributed and Parallel Databases*, 1994, 2(3): 243~260
- [59] Ann L. Drapeau, Ken W. Shirrif, John H. Hartman, et al. RAID-II: a high-bandwidth network file server. In: *Proceedings of the 21st Annual International Symposium on Computer Architecture*. 1994. 234~244
- [60] P. Cao, S. B. Lim, S. Venkataraman, et al. The TickerTAIP Parallel RAID Architecture. *ACM Transactions on Computer Systems*, 1994, 12(3): 236~269
- [61] W. Dowdy, D. Foster. Comparative Models of the File Assignment Problem. *ACM Computing Surveys*, 1982, 14(2): 287~313
- [62] R.L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal Applied Math*,1969, 7(2): 416~429
- [63] L.W. Lee, P. Scheuermann, R. Vingralek. File assignment in parallel I/O systems

- with Minimal Variance of Service Time, IEEE Transactions on Computers, 2000, 49(2): 127~140
- [64] Michael Rabinovich, Irina Rabinovich, Rajmohan Rajaraman, et al. A dynamic object replication and migration protocol for an Internet hosting service. In: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems. 1999. 101~113
- [65] T Xie. SOR: A Static File Assignment Strategy Immune to Workload Characteristic Assumptions in Parallel I/O Systems. In: Proceedings of the 2007 International Conference on Parallel Processing. 2007.32~39
- [66] 吴松, 金海, 邹德清. 一种流媒体文件的分块放置方法. 计算机学报, 2006, 29(3): 500~507
- [67] 曾碧卿. 分布式计算中并行 I/O 调度策略研究:[博士学位论文]. 长沙:中南大学图书馆, 2005.
- [68] 曾碧卿, 陈志刚, 刘安丰等. 一种集群计算系统中并行 I/O 文件存储分配策略. 小型微型计算机系统, 2005, 26(5): 873~876
- [69] 曾碧卿, 陈志刚, 谭璐等. 分布式计算中可扩展的并行 I/O 数据分配策略研究. 小型微型计算机系统, 2005, 26(10): 1799~1802
- [70] W Litwin, MA Neimat, DA Schneider. LH*-a scalable, distributed data structure. ACM Transactions on Database Systems, 1996, 21(4): 480-525
- [71] W Litwin, MA Neimat, G Levy, et al. LH*S: a high-availability and high-security scalable distributed data structure. In: Proceedings of the 7th International Workshop on Research Issues in Data Engineering. 1997. 141~150
- [72] W Litwin, T Risch. LH*g: a high-availability scalable distributed data structure by record grouping. IEEE Transactions on Knowledge and Data Engineering, 2002, 14(4): 923~927
- [73] W Litwin, T Schwarz. LH*RS: A high-availability scalable distributed data structure using Reed Solomon codes. In: Proceedings of the 2000 ACM SIGMOD of International Conference on Management of Data. 2000. 237~248
- [74] Steve Heller. Extensible hashing. Dr. Dobb's Journal, 1989, 14(11): 66~70
- [75] Brinkmann A, Salzwedel K, Scheideler C. Efficient, distributed data placement strategies for storage area networks. In: Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA). 2000. 119~128
- [76] Brinkmann A, Salzwedel K, Scheideler C. Compact, adaptive placement schemes for non-uniform capacities. In: Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA). 2002. 53-62
- [77] C Wu, R Burns. Handling heterogeneity in shared-disk file systems. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing. 2003.7~7
- [78] 谈华芳. 基于共享对象存储设备的并行文件系统研究: [博士学位论文]. 北京: 中国科学院计算技术研究所图书馆, 2005.

- [79] 刘仲, 周兴铭. 基于动态区间映射的数据对象布局算法. 软件学报, 2005, 16(11): 1886~1893
- [80] 刘仲. 基于对象存储结构的可伸缩集群存储系统研究 :[博士学位论文]. 长沙: 国防科技大学图书馆, 2005
- [81] Choy DM, Fagin R, Stockmeyer L. Efficiently extendible mappings for balanced data distribution. *Algorithmica*, 1996, 16(2): 215~232
- [82] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995.
- [83] Renuga Kanagavelu, Yong Khai Leong. A Bit-Window based Algorithm for Balanced and Efficient Object Placement and Lookup in Large-scale Object Based Storage Cluster. In *Proceedings of the 23th IEEE Conference on Mass Storage Systems and Technologies*, 2006.
- [84] Honicky RJ, Miller EL. A fast algorithm for online placement and reorganization of replicated data. In: *Proceedings of the 17th International Parallel & Distributed Processing Symposium*. 2003.
- [85] Honicky RJ, Miller EL. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In: *Proceedings of the 18th International Parallel & Distributed Processing Symposium*. 2004. 1357~1366
- [86] S. A. Weil, S. A. Brandt, E. L. Miller, et al. CRUSH: Controlled, scalable, decentralized placement of replicated data. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 2006.
- [87] Hong Tang, Tao Yang. An Efficient Data Location Protocol for Self-organizing Storage Clusters. In: *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. 2003.53~66
- [88] D. Karger, E. Lehman, T. Leighton, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In: *Proceedings of the 29th ACM Symposium on Theory of Computing*. 1997. 654~663
- [89] D. Roselli, J. Lorch, T. Anderson. A comparison of file system workloads. In: *Proceedings of the 2000 USENIX Annual Technical Conference*. 2000. 41~54
- [90] J. K. Ousterhout, H. D. Costa, D. Harrison, et al. A trace-driven analysis of the Unix 4.2 BSD file system. In: *Proceedings of the 10th ACM Symposium on Operating Systems Principles*. 1985. 15~24
- [91] D. A. Patterson, J. Hennessy. *Computer Architecture A Quantitative Approach*. 北京: 机械工业出版社, 1999. 14~39
- [92] Austin Group. *Draft Standard for Information Technology—Portable Operating System Interface(POSIX)*. 2001. 3~48
- [93] Zhongying Niu, Ke Zhou, Dan Feng, et al. Implementing and Evaluating Security Controls for an Object-Based Storage System. In: *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*. 2007. 87~99

- [94] Lohmeyer, J.B., Evans, M., 2008. Information Technology-SCSI Object-based Storage Device Commands-2 (OSD-2). INCITS T10 Working Draft. Available from: <http://www.t10.org/ftp/t10/drafts/osd2/osd2r04.pdf>
- [95] Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel, 3rd Edition. O'Reilly, 2005.
- [96] J KATCHER. Postmark: A new file system benchmark, Technical Report. TR3022 (Oct.1997). Network Appliance.
- [97] Feng Dan, Wang Juan, Wang Fang, Xia Peng. DOIDFH: An effective distributed metadata management scheme. In: Proceedings of 2007 International Conference on Computational Science and its Applications. 2007. 245~250
- [98] Juan Wang, Dan Feng, Fang Wang, Chengtao Lu. MHS: A Distributed Metadata Management Strategy. The Journal of Systems and Software. 2009, 82(12), 2004~2011
- [99]<http://www.oracle.com/technology/products/berkeley-db/index.html>
- [100]L Mummert, M Satyanarayanan. Long term distributed file reference tracing: implementation and experience. Software-Practice and Experience(SPE), 1996, 26(6), 705~736
- [101]王娟, 冯丹, 王芳, 廖振松. 一种元数据服务器集群的负载均衡算法. 小型微型计算机系统, 2009, 30(4): 757~760
- [102]孙荣恒, 李建平. 排队论基础. 北京: 科学出版社, 2002.
- [103]Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer widom. 数据库系统实现. 北京: 机械工业出版社, 2001.
- [104]熊劲, 范志华, 马捷等. DCFS2 的元数据一致性策略. 计算机研究与发展, 2005, 42(6), 1019~1027
- [105] 熊劲. 大规模机群文件系统的关键技术研究. 北京: 中国科学院计算技术研究所图书馆, 2006.
- [106]Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. In: Proceedings of the 7th USENIX Conference on File and Storage Technologies. 2009. 125~138
- [107]Davis L.D. Handbook of Genetic Algorithms. Van Nostrand Remhold, 1991
- [108]Xiaoyu Yao, Jun Wang. RIMAC: A Novel Redundancybased Hierarchical Cache Architecture for Energy Efficient, High Performance Storage Systems. In: Proceedings of the 2006 EuroSys conference. 2006. 249~262
- [109]F. Wang, Q. Xin, B. Hong, et al. File system workload analysis for large scale scientific computing applications. In: Proceedings 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies. 2004. 139~152
- [110]M. Baker, J. Hartman, M. Kupfer, et al. Measurements of a distributed file system. In: Proceedings of the 13th ACM symposium on Operating systems principles. 1991. 198~212

- [111]W. Vogels. File system usage in Windows NT 4.0. In: Proceedings of the 17th ACM symposium on Operating systems principles. 1999. 93~109
- [112]Mitzenmacher M. Compressed Bloom filters. IEEE/ACM Transactions on Networking, 2002, 10(5): 604-612
- [113]Juan WANG, Dan FENG, Fang WANG, Cheng-tao LU. Extending attributes page: a scheme for enhancing the reliability of storage system metadata. Journal of Zhejiang University SCIENCE A, 2009, 10(8): 1106~1113
- [114]Ross, S., 1983. Stochastic Processes. John Wiley Press, New York, USA.
- [115] Billinton, R., Allan, R.N., 1992. Reliability Evaluation of Engineering System: Concepts and Techniques. Plenum Press, New York.