(19)中华人民共和国国家知识产权局

(12)发明专利申请

(10)申请公布号 CN 103473337 A
(43)申请公布日 2013.12.25

(21)申请号 201310431658.3

(22)申请日 2013.09.22

(71)申请人 北京航空航天大学
地址 100191 北京市海淀区学院路 37 号

(72)发明人 王鲁俊 龙翔 王雷

(51)Int.Cl.
G06F 17/30(2006.01)

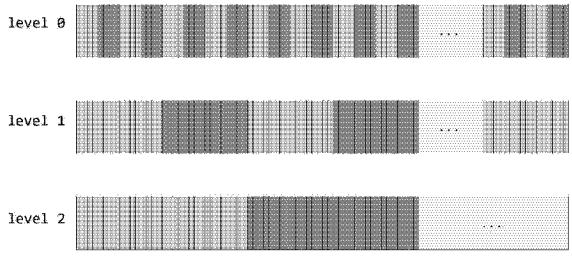
权利要求书1页 说明书4页 附图2页

(54)发明名称

一种分布式存储系统中处理面向海量目录和文件的方法

(57)摘要

本发明提出了一种分布式存储系统中处理面向海量目录和文件的方法。在集群中，将所有的用于存放元数据的 MetaServer 用一致性哈希方案组织。对于每一个 MetaServer，其上运行的元数据服务会将元数据组织成不同的文件。每个元数据都是一个元数据项，记录了各种元数据信息，如时间、权限等。本发明将目录打包成一个较大的块，存放在本地文件系统中，块由系统预分配。为了适应不同大小的目录，并适应目录所包含元数据不断增长的应用需求，因此将不同大小的目录放在不同级别的块中(低阶别的块存放小目录，高级别的块存放打得目录)，如图 1，对不断增大的目录通过移动其元数据项移动到更高级别的块。



CN 103473337 A

1. 本发明设计了自适应的目录存放方法，其特征在于：能够针对不同大小的目录，自适应存放，一个元数据操作的平均磁盘数据略高于 1 次。

一种分布式存储系统中处理面向海量目录和文件的方法

技术领域

[0001] 本发明涉及分布式存储和海量小文件存储领域，具体涉及一种分布式存储系统中处理面向海量目录和文件的方法。

背景技术

[0002] 近年来，随着大数据时代的来临，分布式文件系统的研究逐渐增多，云计算时代。对于大多数分布式文件系统而言，元数据的管理是个十分复杂的问题，元数据的管理部件的设计架构直接影响到系统的性能、可扩展性、系统可用性等。分布式存储系统最根本的是将大量普通商用机器组织成高性能、低延迟、对用户层屏蔽失效和不可用事件的集群，这要求分布式存储系统必须能够进行良好的横向扩展(Scale-Out)，而不能仅仅通过更换更快的 CPU，增大内存容量，增大磁盘存储容量，更换 SSD 来进行纵向的扩展(Scale-Up)。

[0003] 横向扩展(Scale-Out)最主要考虑的问题的就是处理记录数据逻辑与物理位置的映像关系即数据元数据，另外包括诸如属性和访问权限等信息。尤其对于面向海量小文件的应用中，元数据问题是个非常棘手的挑战。

[0004] 分布式文件系统中，数据会分布在多个独立的物理节点上，对数据的 IO 请求通常也会被分散到相应的物理节点中。这也是分布式文件系统横向扩展(Scale-Out)所要求的。现有的分布式文件系统中，如果以文件为单位进行调度，则不同的文件会存储在不同的节点上，当文件比较大时，一些分布式文件系统将文件分块(例如 GFS，HDFS 将比较大的文件系统分为 64M 的块)。因此带来的问题就是如何确保对数据的定位，也即如何确定一个文件(或者文件的每一个数据块)在哪一台物理节点上。分布式系统中往往会提供元数据服务来解决这个问题解决这个问题的。元数据服务记录数据与其对应的存储位置的映射关系，另外还包含文件访问控制所需要的所有元数据(访问时间、访问权限、属主用户等)。对文件进行访问时，先向元数据服务请求查询对应的元数据，然后通过获得的元数据进行后续的文件读写等 I/O 操作。

[0005] 目前主流的分布式文件系统的元数据管理方式大致可以分为三种模型，即

[0006] 1. 中心化元数据服务模型：

[0007] 中心化元数据服务模型是比较简单的一种模型。出于设计上简单、实现容易等因素的考虑，同时由于历史遗留问题，很多分布式文件系统采用了中心化的元数据服务。例如 Google 的 GFS，Apache 的 HDFS，Lustre，PVFS，StorNext，等。

[0008] 在中心化元数据服务模型中，通常设置一个中心化的元数据服务器来支持元数据的存储和客户端查询请求，例如 GFS 和 HDFS 中的 NameServer 节点。中心化的元数据服务器节点提供统一的文件系统命名空间，并处理名字解析和数据定位，访问权限控制，元数据查询等功能。因此，中心化元数据服务模型的最大优点就是设计实现简单，实际上相当于一个单机的服务，独立地维护目录和文件，对外提供网络访问接口即可。元数据服务设计实现的关键考量是节点的 OPS 吞吐量，即单位时间处理的请求数。为了优化 OPS，中心化元数据服务模型对 CPU、内存、磁盘要求较高，条件允许的情况下尽量使用高性能 CPU、大内存和高

速磁盘，甚至后端存储可考虑使用高端磁盘阵列或 SSD。

[0009] 2. 分布式元数据服务模型

[0010] 与中心化元数据服务模型相对应的是分布式元数据服务模型。

[0011] 目前常见的分布式元数据服务模型主要有四种设计方案：

[0012] (1) 基于文件系统的设计：由于磁盘文件系统本身就是树状结构视图，因此可以利用这现成的机制在元数据服务器上实现名字空间。对于分布式文件系统中的每一个目录或文件，在元数据服务器的本地文件系统之上一一对应创建一个目录或文件（以下称为元目录和元文件）。元目录用来表示 DFS 中的目录，其元目录属性保存 DFS 目录属性；元文件用来表示 DFS 中的文件，元文件属性保存 DFS 文件属性，元文件内容则用来保存元数据，包括更详细的文件属性、访问控制信息、数据分片信息、数据存储位置等信息。由此，基于现有的本地文件系统构建了 DFS 的名字空间，设计简单实现容易。元文件仅用来存储数据文件的元数据，一般都是小于 1KB 的小文件，如果文件目录数量比较大，本地文件系统性能会急剧下降。

[0013] (2) 位于内存的分层设计：Apache HDFS 采用了这种方案。与基于文件系统的实现不同，名字空间完全在元数据服务器节点的内存中，使用层次结构来表示，具体实现可以使用树结构或者层次化的数组结构。在层次结构中，每个结点表示 DFS 的一个目录或文件，结点的孩子结点理论上没有数量限制（取决于内存可用量），孩子结点使用动态数组或者链表来表示。

[0014] (3) 位于内存的 Hash 设计：这种方式与 Google GFS 实现相仿。GFS 论文中指出其名字空间采用了全内存设计、偏平式组织、前缀压缩算法、二分查找算法、没有支持 ls 的数据结构，论文中还指出 ls 操作的效率较低。GFS 没有开源，从论文中可以看出位于内存的 Hash 设计可能比较接近其设计。这种设计采用 Hash 和二分查找相结合的来实现，即目录以完整的绝对路径进行 hash 定位，该目录下的孩子结点使用二分查找进行定位。它与分层设计的主要不同在于，只需要一次 hash 和一次二分查找，而分层设计需要多次的二分查找，在性能上更优。我们仅对目录进行 Hash，名字空间具有一定的偏平性，但没有达到 GFS 的完全偏平；子文件目录不包括父路径部分，相当于作了前缀压缩，但不如分层前缀压缩彻底。

[0015] (4) 位于内存的双重 Hash 设计：这种方式是对基于全内存 hash 设计的改进。它先对目录进行第一次 hash 运算，然后对子文件目录进行第二次 hash 运算，从而将查找时间复杂性从 log(n) 进一步降低至 O(2)。目录 Hash 表是全局的，而目录结点的 Hash 表是局部的，每一个目录结点都包含一个 Hash 表，仅用来存储本目录下的子文件目录信息。

[0016] 3. 无元数据服务模型：理论上，无元数据服务模型是可行的，只要寻找到元数据查询定位的替代方法即可。目前，基于无元数据服务模型的分布式文件系统非常少，比较具有代表性的是 Glusterfs。Glusterfs 使用弹性哈希算法代替传统分布式文件系统中的集中或分布式元数据服务，使用算法进行数据定位，集群中的任何服务器和客户端只需根据路径和文件名就可以对数据进行定位和读写访问。

[0017] 三种元数据服务模型比较

[0018] 传统分布式存储系统使用集中式或布式元数据服务来维护元数据，集中式元数据服务会导致单点故障和性能瓶颈问题，而分布式元数据服务存在性能开销、元数据同步一

致性和设计复杂性等问题。无元数据服务模型，消除了元数据访问问题，但同时增加了数据本身管理的复杂性，缺乏全局监控管理功能，并增加了客户端的负载。由此可见，这三种模型都不是完美的，分别有各自的优点和不足，没有绝对的优劣与好坏之分，实际选型要根据具体情况选择合适的模型，并想方设法完善其不足之处，从而提高分布式文件系统的扩展性、高性能、可用性等特性。

发明内容

[0019] 本发明提出了一种分布式存储系统中处理面向海量目录和文件的方法。

[0020] 在集群中，将所有的用于存放元数据的 MetaServer 在逻辑上组织成环。系统采用一致性哈希方案，按照指定的哈希算法对 MetaServer 的 ID 进行哈希，并根据哈希值将各个 MetaServer 分布在整个哈希值域的环上。

[0021] 每个 MetaServer 提供元数据服务，即在 MetaServer 上启动元数据功能服务，对于每一个元数据操作请求(例如，readdir，create)，首先将该请求中的目录名用上述哈希算法进行哈希，根据哈希值确定一个(在元数据服务配置备份时可能有多个)处理该请求的 MetaServer。

[0022] 对于每一个 MetaServer，其上运行的元数据服务会将元数据组织成不同的文件。

[0023] 每个元数据都是一个元数据项，如图 1，d/f 标记是文件还是目录，文件 / 目录名字，创建时间，修改时间，属主和组，访问权限位。

[0024] 每个目录都包含了一组元数据项，这组元数据项记录了该目录下的文件和子目录的元数据信息。

[0025] 由于分布式文件系统中，目录数目可能非常多。因此将目录打包成一个较大的块，存放在本地文件系统中，块由系统预分配。如图 2。

[0026] 由于不同目录所包含的元数据项数目差别比较大，例如有些目录只包含少量的文件和子目录，因此这样的目录包含的元数据项就比较少，有些目录可能包含成千上万个元数据项。为了适应不同大小的目录，并适应目录所包含元数据不断增长的应用需求，每个目录在最初创建时被打包到 level0 的块中，level0 的块中每个目录可以存放 4 个元数据项。若该目录下创建更多的文件或子目录，则在超过 4 个后，将该目录对应的所有元数据项全部移动到 level1 的块中，level1 中的块中，每个目录可以存放 8 个元数据项。如果该目录下继续创建文件或子目录，则类推移动到下一个 level 的块中，如图 3 所示。

[0027] 由于块是预分配的，并且移动元数据项是批量移动，顺序读顺序写，磁盘 IO 效率很高。另外，如果连续向同一目录创建 1000000 个文件，实际只需要移动 18 次。

[0028] 每个块都有一个检索文件 Menifest 描述每一个目录在块中的位置和已经写入的元数据项个数，例如，图 4 中‘/home’目录从 0 开始存放，当前有 6 个文件或子目录。系统将每个检索文件 Menifest 的信息在内存用哈希表重建，加速查找。

附图说明

[0029] 图 1 是元数据项示意图

[0030] 图 2 是目录打包成块示意图

[0031] 图 3 是不同 level 的块示意图

[0032] 图 4 是 Menifest 文件信息示意图

具体实施方式

[0033] 步骤 1 ：

[0034] 每一个 MetaServer 配置一个 id，选定一个哈希函数，如 Murmurhash，将所有的 MetaServer 的 id 进行哈希。

[0035] 步骤 2 ：

[0036] 对于任意一个元数据请求，例如在 /home/zhang/ 下建立一个文件 myfile，则先对目录字符串进行哈希（使用同样的哈希函数 Murmurhash），按照一致性哈希算法选定一个 MetaServer 来处理这个元数据请求。

[0037] 步骤 3 ：

[0038] 分别针对不同的元数据请求，执行不同的动作 ：

[0039] 1) 创建目录请求 ：

[0040] 首先判断该请求是否能够执行，如果该 MetaServer 上已经有了这个目录，则返回目录已存在消息。否则在 level0 块中查找一个空闲的位置，在 Menifest 中添加该记录。

[0041] 2) 在目录下创建文件请求 ：

[0042] 首先从内存中记录的检索信息文件 Menifest 中找到该目录，如果找不到，则返目录不存在消息。如果该目录所在的 level 块中还能添加新的元数据项条目，则增加这个文件元数据信息，如果没有空闲空间，则在下一个级别的 level 块中找到一个空闲位置，将该目录对应的所有元数据项移动到新位置（元数据清除），然后添加请求中这个文件的元数据信息。

[0043] 3) 在目录下创建子目录请求 ：

[0044] 首先从内存中记录的检索信息文件 Menifest 中找到该目录，如果找不到，则返目录不存在消息。根据一致性哈希算法，找到应该处理子该目录的 MetaServer，然后向该 MetaServer 发送创建该子目录的请求。如果成功，需要向本地 level 块中添加新条目。如果该目录所在的 level 块中还能添加新的元数据项条目，则增加这个子目录元数据信息，如果没有空闲空间，则在下一个级别的 level 块中找到一个空闲位置，将该目录对应的所有元数据项移动到新位置（元数据清除），然后添加请求中这个子目录的元数据信息。

[0045] 4) 删除目录下某个元数据项 ：

[0046] 首先从内存中记录的检索信息文件 Menifest 中找到该目录，如果找不到，则返目录不存在消息。从该目录所在的块查找这一个元数据项，标记为已删除。如果该目录下的元数据项少于当前 level 块中目录存放元数据项最大数目的一半，则在上一个级别的 level 块中找到一个空闲位置，将该目录对应的所有元数据项移动到新位置（元数据清除）。

[0047] 5) 删除目录请求 ：

[0048] 首先找到该目录，如果该目录下有文件或者子目录，则返回错误。根据一致性哈希找到其父目录对应的 MetaServer，向该 MetaServer 发送删除该元数据项的请求。

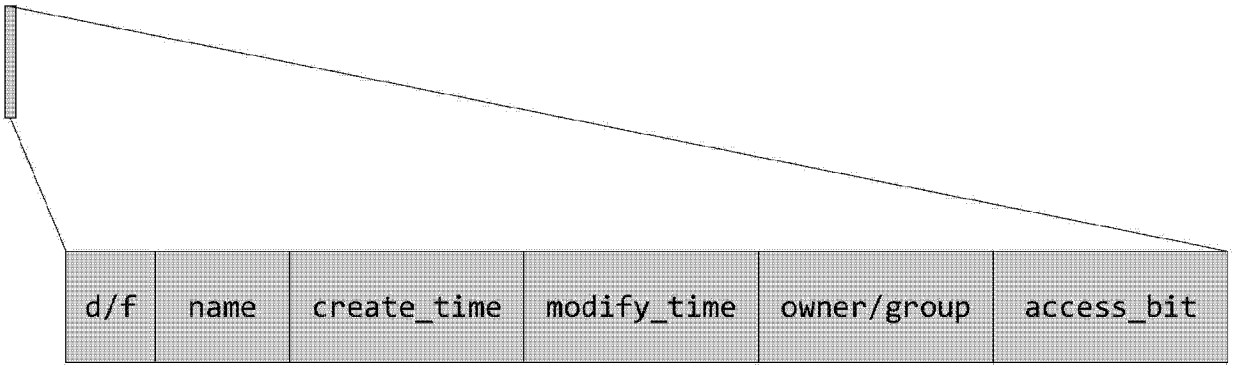

d/f
name
create_time
modify_time
owner/group
access_bit

图 1

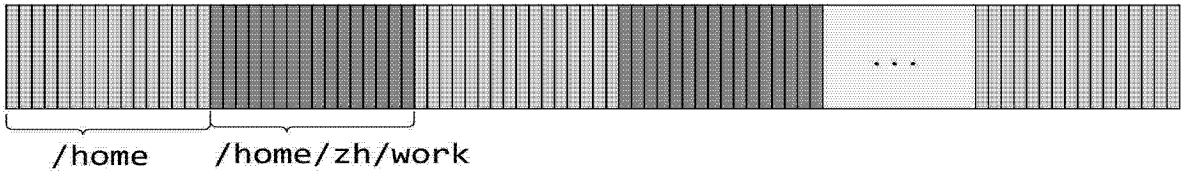

...
/home
/home/zh/work

图 2

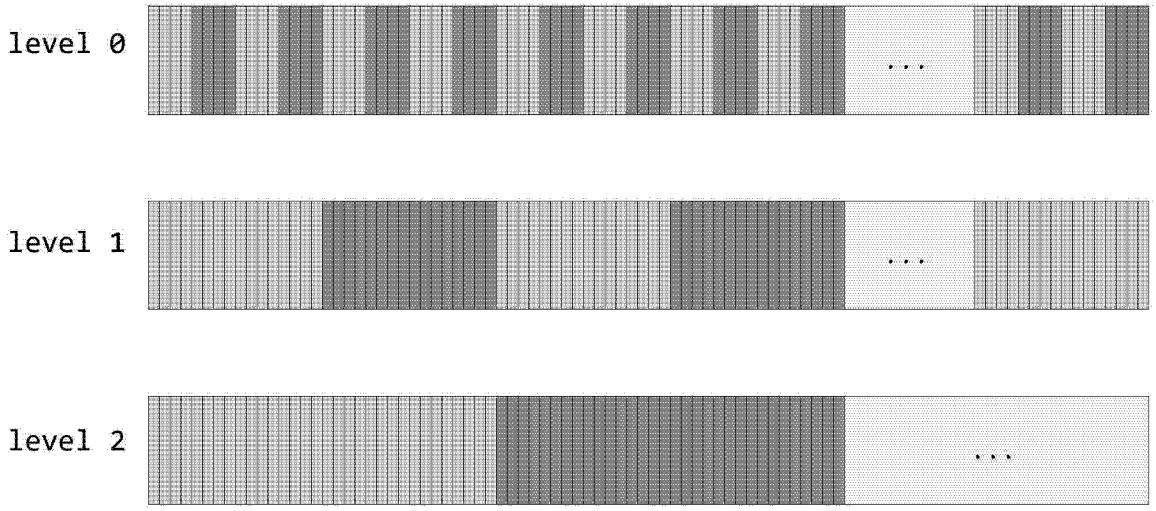

level 0
...
level 1
...
level 2
...

图 3

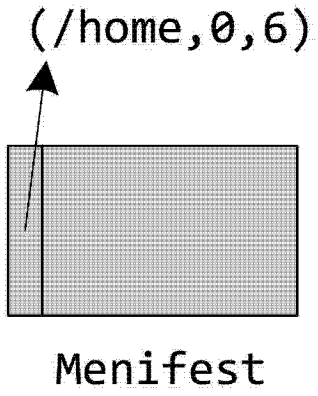


图 4