分布式存储系统架构调研

金孟哲

基础架构组

主要问题：

1. 元数据管理
2. 数据可用性保证
3. 数据持久性保证
4. 故障恢复，手动自动，恢复时间
5. 垂直分层 vs 全能型
6. 负载均衡，热点迁移
7. 小文件：磁盘寻道，元数据查找，Locality
8. 一致性探究
9. Snapshot，重复数据删除
10. 元数据管理方案

元数据管理是分布式存储系统的重中之重，操作复杂，对可靠性的要求高，实现难度大。首先列举一下目前常见的一下元数据管理方案：

1. GFS，HDFS，MooseFS，BladeStore。单一元数据服务器

最大的好处是，实现简单，出现问题容易定位，通常使用B-tree作为内存的数据结构。最大的问题是：单点失效，性能瓶颈，内存瓶颈。

**单点失效：**只有一个元数据服务器，挂了就不能提供服务了，早起的GFS和HDFS需要手工恢复，后来都加入了自动恢复的机制，bladestore中的实现是用一台或多台热备来降低恢复延时，通过zookeeper同步来提供Master挂掉的通知。延时较大？

**性能瓶颈：**单一的元数据服务器，能够提供的元数据操作能力是有限的，再加上复杂的同步机制之后，单台机器想达到上万次每秒的服务能力有相当难度。

**内存瓶颈：**元数据需要高效的处理，通常是将所有信息都放在内存中，那在集群规模变大的时候，单台机器的内存显得太小了。如果放在磁盘上，那单一的元数据服务器的性能瓶颈会更严重。比如10亿个文件，每个150字节的元数据的话，就需要150G的内存。

1. Xfs，主+备

在单一元数据服务器的基础上，实现了一致性的热备，读操作可以通过读取Slave来降低Master的压力。而且也降低了Master挂掉之后的恢复时间。

内部实现了journal，减少了使用第三方的WAL（比如bookkeeper）组件的开销。直接将Master和Slave作为journal的写入单元，都只需要写本地磁盘即可。[[1]](#endnote-1)

部分解决了单点失效，优化了一定的性能瓶颈，内存瓶颈依然存在。

1. HDFS2.0 b-tree静态拆分，NFS,AFS,Coda,Sprite

依然使用B-tree作为数据结构，但人工将其拆分为几个子树，类似于Linux文件系统中挂载点的方案。好处是实现简单，在原来的元数据管理方案的基础上不需要做太大的改动，拆分也是静态的，不涉及迁移等过程。缺点也很明显，过于依赖静态拆分本身，如果在运行过程中某一个子树访问很多或迅速变大，仍然面临单一元数据服务器的所有问题。

1. Ceph，b-tree动态拆分

依然使用B-tree作为数据结构，但通过算法来动态的拆分b-tree，实现相对复杂，但能够解决好读写热点的问题，提供了比较好的读写性能。同时保留了B-tree的结构，查找定位的速度比较快。[[2]](#endnote-2)

问题：实现复杂度较高，一致性保证

1. Hashing, Vesta,Intermezzo,RAMA,zFS,Lustre

使用路径或inode number的哈希值来分配节点

1. GIGA+ 一致性哈希 OrangeFS

一致性Hash：在普通hash的基础上，改进了加入和退出节点时的对应关系改变。在增加和删除节点的时候，保证最小数量的元素对应关系改变。

这种方案中，直接将每一级目录根据inode哈希到一台机器上，然后找出下一级的目录对应的inode，继续计算哈希寻找。

这种方案的完全的分布式，在集群规模极大的情况下，线性可扩展性最好。问题：损失了hierarchical locality，在目录结构较深的情况下，查找定位的延时较大。一致性保证。

1. 数据库

使用KV数据库来管理元数据。其中使用方式也有很多种，比如直接将全路径作为key保存在Cube这种数据库中，问题在于：无法实现目录的重命名，一致性的保证需要跨行事务的支持。比较适合于对象存储（没有目录树结构），不太适合传统的文件系统结构。

还有将inode之类的信息作为key，连续的保存该目录下的文件和目录信息的方式，同样存在跳转次数多，一致性保证等诸多问题。

同时使用单独的数据库服务的话，会有部署复杂，依赖关系负载等问题。使用嵌入式的BerkeleyDB的方式比较流行。

1. 其他

Colossus用了什么方式还不清楚，但根据透漏出来的信息，能够轻松扩展到上百台元数据服务器，应该不是静态拆分B-tree的方法。

BFS这种小文件存储方案，直接采用了fileid包含元数据信息的方式，没有传统的目录树结构。实际上相当于用Cube作为元数据服务器，是对象存储。

1. 可用性保证

Availbility，可用性对于分布式系统来说非常重要，CAP的其中之一。

提供高可用性的服务有以下几个难题：

1. 足够高的性能。
2. 故障的容忍程度
3. 故障的恢复速度

性能这里不谈了。

针对发生故障的容忍，有三种处理方案，a）允许部分失败的情况下成功返回b）失败后能够重新选择目标继续完成，c）失败后需要等待故障恢复。可用性依次降低。

a的典型系统：bookkeeper，quorum journal manager, Dynamo。

这种处理方案对于高可写性的系统很重要，比如journal，Dynamo是用在用户添加购物车时的，也要求尽量不失败。[[3]](#endnote-3)在这种方案下，一次写直接发送到N台目标机器，其中只需要W台返回成功信息，即认为成功，不仅能够保证高可用，还能保证低延时。因为即使部分目标机器延时较大，也不影响整体延时。

这里涉及到NWR原则，如果一共写到N台机器中，只需要写成功W台就认为成功，读的时候需要从R台中读取最新的数据，要保证强一致性，需要N<W+R，要保证高可写，需要N>W，要保证高可读，需要R尽量小，也就是=1。系统设计可以根据需求来选择方案。

b的典型系统：BladeStore，HDFS。同时向三个目标写入数据，失败任意一个，这次写都会失败，但是客户端有机制重新向NameNode申请新的目标，然后重新尝试写。这样虽然会有一定的延时，但是通常能够成功完成写入，也是目前比较常见的方式。

c的典型系统:BladeCube，这种方式一般不会用在分布式文件系统中，但是在Cube和HBase这种系统中，这是一种相对折中的设计方案，避免了太复杂的错误处理。因为如果承载对应Region的机器挂了，必须要等相应的Region重新在某一台机器上线之后，才能够继续写入和读出数据，而这个错误恢复时间通常在秒级到分钟级之间。相比于前两种，可用性要低一个档次。但是HBase的火爆也说明，某些时候这种取舍是值得的。据说新的BigTable设计中， 做了表级的副本，也就有了在这方面改进的可能性。

1. 持久性保证和冗余管理

数据的持久性，主要是指在各种意外情况下（比如掉电，磁盘损坏，死机，机房起火），数据能够完成保存下来的保证，亚马逊的标准存储支持11个9的年持久性。

持久性的保证主要靠冗余来实现，这样在磁盘损坏，机器损害等各种情况下，都能够恢复出原始数据。

1. 最简单的冗余处理：复制（replication），比如BladeStore， BFS中目前采用的都是三副本的冗余方案，这样可以容忍两块磁盘同时损坏。

优点：实现简单，可以提供读的负载均衡，性能好。

缺点：数据冗余度大，浪费多，持久性一般。

1. 校验码XOR，这种方案能够提供相对较小的冗余度，但持久性仍然不高。以前最主要的优势是计算很快，但是目前ReedSolomon的计算性能也已经有保证了，所以单纯用XOR做数据冗余的方案已经不是主流了。
2. 目前更加先进的管理方式：纠删码，其中最流行的是Reed-Solomon编码，可以在不高的冗余度的情况下，提供远高于三备份的持久性保证。具体RS编码相关的内容请参见严林同学做的总结。

优点：冗余度小，持久性非常好。

缺点：恢复带宽大，实现相对复杂，无法提供读的负载均衡，在出现故障时需要边算边读，性能不如复制的方式。

1. 在纠删码的基础上，为了解决恢复带宽过大的问题，最近出现了一种LRC的方案，能够增加不太大的冗余度（14%）的情况下，将恢复带宽降低一半。



除此之外，持久性的保证还有一个很重要的特性，跨机房跨地域备份，只有跨机房备份才能防止火灾，地震等意外情况。

目前跨机房备份还没有很好的用纠删码实现的方案，因为在公网上使用纠删码的恢复带宽会完全无法接受，但是有LRC之后，这部分也有了一定的可能性，比如下面这种方案。

机房1：D1-D5，S1

机房2：D6-D10，S2

机房3：P1-P4，S3

使用1.7个备份数，保证在任何一块磁盘损坏的情况下，都可以在数据中心内部完成恢复，只有坏2块以上的情况下才需要用到大量的跨数据中心的数据传输。

1. 故障恢复

目前分布式系统中的常见故障类型：程序bug导致程序宕机，网络无法连通，机器死机重启，磁盘有坏道，磁盘永久损坏，机器关机。

故障的恢复速度，有几个级别，手工恢复，本机自动恢复，迁移恢复，热备恢复。

手工恢复最慢，目前在我们这种使用量的情况下，应该不允许任何核心的模块需要等待手工恢复。

本机自动恢复，是指在宕机重启之后，能够自动恢复所有数据并上线服务。数据保存在本地磁盘上，在完成重启之后能够恢复服务，但无法应对机器关机的情况，需要人工干预。

迁移恢复，是指通过zookeeper心跳之类的机制，发现故障之后，将服务迁移到其他的机器上，目前我们的系统更多的是采用这种方式。主要的问题是，zookeeper的超时时间难以把握。太短容易误报，太长则发现问题并迁移的时间变长。

热备恢复，是指本身服务就有多个完全一致的热备，在原机器没有恢复的情况下也能够提供服务。

1. 垂直分层 vs 一站式解决

HBase是典型的垂直分层的解决方案。

底层HDFS本身就是一个成熟的系统，然后KV数据库HBase架构在HDFS上，省去了管理数据备份以及数据可靠性的工作，从而可以集中处理高性能的KV读取等。

Facebook应该是HBase的最大线上服务用户，因此Facebook针对HBase的分层模式进行了分析。

1. 分层模式最大的好处是容易实现，能够利用其他系统的优势来帮助自己的系统。同时系统更加容易稳定。
2. 坏处主要是性能问题。底层系统不可能完全符合上次系统的需要，因此在性能，网络带宽，磁盘IO等方面无法针对性的进行优化。
3. 针对每一点的性能问题，都可以进行穿透性的改造来优化性能，但是同时就会损失掉低耦合的好处。这时候就要权衡改进和代价是否匹配。两个例子：compact的时候浪费store三个备份的带宽， 写log的时候无法利用到磁盘的连续写。

放在文件系统层面，也有是否使用分层方案的选择。比如BladeStore目前的Master的日志存储，目前是放在Bookkeeper中，也是分层设计。但就遇到了比较大的性能问题，需要进行改进。

1. 负载均衡
2. 解决数据读写热点。
3. 元数据操作热点
4. 磁盘使用率

数据读热点的负载均衡，主要可以靠多副本的分配，将不同的读请求分散到不同的机器上。而对于持续性的读热点，可以通过元数据服务器对数据块进行移动来解决。而写的负载均衡，主要靠block分配时的算法，将写请求均匀分散到不同的机器上。

除此之外还有一种数据分布方式是Ceph，通过对文件id的哈希来分配存储节点，写操作的均衡是可以达到的，但如果出现读热点，没有办法进行这一类的迁移操作。

磁盘使用率是指在扩容之后，或经过长时间的运行出现大量删除操作之后，某些磁盘空间很大，某些则很小，在进行写负载均衡的时候受限制。两种数据分布方式的系统都能够较好的处理这个问题，但是哈希方式的不够灵活，在增加容量的时候会自动的完成迁移，而通过元数据服务器保存对应关系的，可以根据运维状况自行决定迁移策略。

元数据操作热点的问题，在元数据一部分已经涉及到了，静态子树拆分的方式无法进行良好的负载均衡，其他方式则各有利弊。

1. 小文件主要问题

小文件很重要的就是随机读性能，大量的不同的读请求如何在限定时间内完成是主要问题。

1. temporal locality, spatial locality, and sequentiality，大意是时间局部性，空间局部性，连续性。

在某些模式下，一次磁盘访问之后，在不长的时间之后会再次访问这部分内容，就是指时间局部性。

预取通常依据的是空间局部性，访问了一块内容后，知道应用将会有很大概率访问附近的内容，就可以将附近的数据都读取出。

连续性是指应用将会连续的读取相邻的数据，比如读取log的时候，可以设置较大的缓冲区来提高吞吐量。

一个系统是否满足这几种局部性对于系统的设计非常重要，因此在需求调研阶段就需要对这部分特性进行一定的分析。在优化性能的时候，这部分也是相当重要。在Facebook的message应用中，他们发现大部分数据一次都不会被访问到，而有一部分数据会被反复的读取非常多次。因此通过增加一个SSD的缓存，将这部分内容保存起来，可以大大的提升性能。[[4]](#endnote-4)

总之在存储系统中，尤其是小文件的存储系统中，性能优化的重点是减少磁盘寻道的次数，方式有

1. 改进内容在磁盘上的排布，使元数据和元数据，元数据和内容数据在磁盘上连续。
2. 一次读取更多内容，但前提是多读出来的内容是以后会用得到的。比如预计到程序有较大概率连续读取后面的内容时，直接一次性多读取一部分。
3. 预读，预测接下来将会读取的位置，提前放入缓存。依赖于上层应用，可以更好的降低延迟。
4. 独占磁盘，多个对磁盘压力大的系统不要共用一块磁盘，比如日志写入的盘，最好不要同时承担随机读的压力。
5. 用SSD做缓存，把元数据完全放在内存中等。
6. 大量的元数据管理，针对方案：使用数据库保存元数据，文件系统内采用自解释的fileid。

也可以采用分布式的元数据服务器来保存文件名和文件元信息的关系。

1. 底层的本地文件系统的支持，需要将小文件整合成大文件来存储。

本地文件系统也无法承受海量的小文件，因此需要将小文件合并成大文件存储。涉及到的主要问题是文件的位置索引。

1. 一致性问题

一致性问题几乎是分布式系统中最头疼的问题。第一是受制于CAP原理，在提高可用性和网络问题容忍性的时候，必然要牺牲一部分的一致性。第二是在和性能权衡的时候，一致性有些时候会被牺牲。第三是实现良好的一致性代码开发的难度也较大，很多时候即使没有权衡的需求，也没有实现出较好的一致性。

1. 强弱一致性

强一致性是在任意时刻，从每个节点读取的值应该是相同的。

弱一致性包括：

最终一致性，因果一致性，Read your write, Session Consistency, Monotonic Read, Monotonic Write

一致性越强的系统，实现难度越大，性能越会受到影响，但是架构在之上的服务逻辑会越简单越可靠。

1. 事务，日志，回滚

实现比较好的一致性的方式，通常是通过两段式提交，三段式提交，Paxos算法等。

对于很多系统，已经发生的修改回滚是非常困难的。比如：

Cube和HBase，回滚比较困难，Append型的系统。

文件系统的元数据管理，回滚比较困难，比如删除等操作。

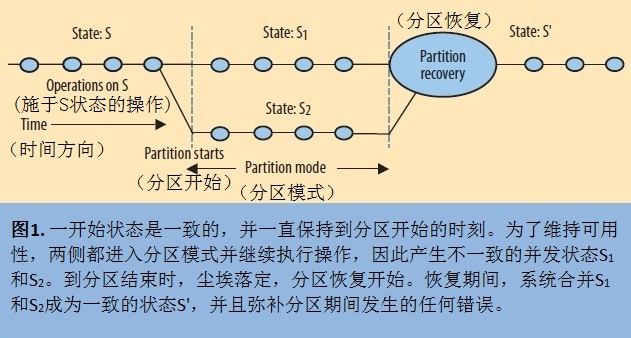
文件的追加写，回滚较容易，只需要将文件长度信息回滚即可。

对于回滚困难的系统，在进行分布式的协调时，通常是先对操作达成共识，并写入各自的日志，该步骤成功之后，再各自对真正的数据结构进行修改。写入日志成功之后发生的失败，在恢复的时候都会恢复出来，并且保持一致。

1. 时间戳？Spanner

Spanner是Google的全球关系型数据库，没有开源，只能通过论文看一些设计思路。通过一个true time API，进行全球的时间同步，所有的操作根据他们的时间戳来确定顺序。

1. 如何通过显式区分网络故障来提高一致性和可用性。[[5]](#endnote-5)



1. Snapshot，重复数据删除。

快照是一个比较实用的附加功能，可以将系统当前的状态做一个备份，比如BladeStore 的checkpoint，其使用的就是元数据信息的快照。更加新的快照技术，能够对整个文件系统用很低的代价做一个快照，并且在快照可以和主分支的系统继续分别作改动，互不影响，有点像git的操作。这是一个非常好的附加功能，但是目前尚未看到特别重要的应用场景。

重复数据删除，主要是用在备份系统中，比如传统的数据备份，每天进行增量备份，每周进行全量备份的操作，会在文件系统中留下大量的冗余数据，如何检测到这些冗余数据并且进行标记，只保留一份，是重复数据删除的核心技术。

目前较为先进的是data domain的content based segment，summary vector等技术，能够达到很高的写入吞吐量，和极高的压缩率，该技术在公有云中如何应用，仍有待进一步开发。

1. http://djt.qq.com/article/view/322 [↑](#endnote-ref-1)
2. Dynamic Metadata Management for Petabyte-scale File System [↑](#endnote-ref-2)
3. Dynamo: Amazon’s Highly Available Key-value Store [↑](#endnote-ref-3)
4. Analysis of HDFS Under HBase: A Facebook messages Case Study [↑](#endnote-ref-4)
5. http://www.infoq.com/cn/articles/cap-twelve-years-later-how-the-rules-have-changed/ [↑](#endnote-ref-5)