HDFS NameNode内存全景

# 一、概述

从整个HDFS系统架构上看，NameNode是其中最重要、最复杂也是最容易出现问题的地方，而且一旦NameNode出现故障，整个Hadoop集群就将处于不可服务的状态，同时随着数据规模和集群规模地持续增长，很多小量级时被隐藏的问题逐渐暴露出来。所以，从更高层次掌握NameNode的内部结构和运行机制尤其重要。除特别说明外，本文基于社区版本Hadoop-2.4.1[1][2]，虽然2.4.1之后已经有多次版本迭代，但是基本原理相同。

NameNode管理着整个HDFS文件系统的元数据。从架构设计上看，元数据大致分成两个层次：Namespace管理层，负责管理文件系统中的树状目录结构以及文件与数据块的映射关系；块管理层，负责管理文件系统中文件的物理块与实际存储位置的映射关系BlocksMap，如图1所示[1]。Namespace管理的元数据除内存常驻外，也会周期Flush到持久化设备上FsImage文件；BlocksMap元数据只在内存中存在；当NameNode发生重启，首先从持久化设备中读取FsImage构建Namespace，之后根据DataNode的汇报信息重新构造BlocksMap。这两部分数据结构是占据了NameNode大部分JVM Heap空间。

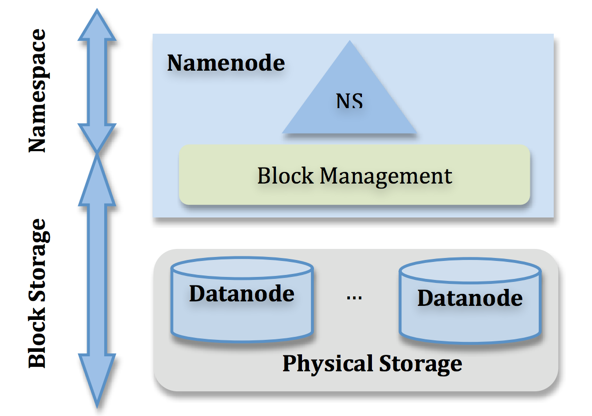


图1 HDFS结构图

除了对文件系统本身元数据的管理之外，NameNode还需要维护整个集群的机架及DataNode的信息、Lease管理以及集中式缓存引入的缓存管理等等。这几部分数据结构空间占用相对固定，且占用较小。

测试数据显示，Namespace目录和文件总量到2亿，数据块总量到3亿后，常驻内存使用量超过90GB。

# 二、内存全景

如前述，NameNode整个内存结构大致可以分成四大部分：Namespace、BlocksMap、NetworkTopology及其它，图2为各数据结构内存逻辑分布图示。

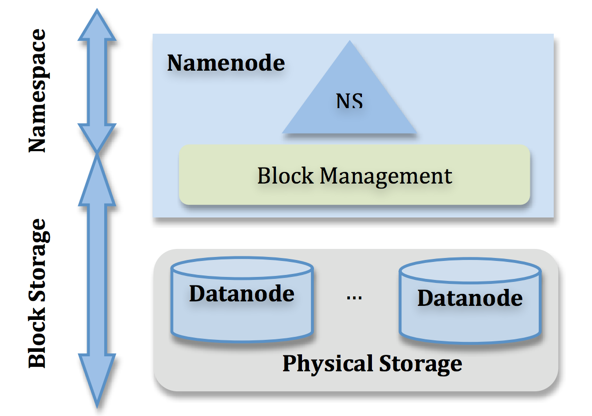


图2 NameNode内存全景图

Namespace：维护整个文件系统的目录树结构及目录树上的状态变化；

BlockManager：维护整个文件系统中与数据块相关的信息及数据块的状态变化；

NetworkTopology：维护机架拓扑及DataNode信息，机架感知的基础；

其它：

LeaseManager：读写的互斥同步就是靠Lease实现，支持HDFS的Write-Once-Read-Many的核心数据结构；

CacheManager：Hadoop 2.3.0引入的集中式缓存新特性，支持集中式缓存的管理，实现memory-locality提升读性能；

SnapshotManager：Hadoop 2.1.0引入的Snapshot新特性，用于数据备份、回滚，以防止因用户误操作导致集群出现数据问题；

DelegationTokenSecretManager：管理HDFS的安全访问；

另外还有临时数据信息、统计信息metrics等等。

NameNode常驻内存主要被Namespace和BlockManager使用，二者使用占比分别接近50%。其它部分内存开销较小且相对固定，与Namespace和BlockManager相比基本可以忽略。

# 三、内存分析

## 3.1 Namespace

与单机文件系统相似，HDFS对文件系统的目录结构也是按照树状结构维护，Namespace保存了目录树及每个目录/文件节点的属性。除在内存常驻外，这部分数据会定期flush到持久化设备上，生成一个新的FsImage文件，方便NameNode发生重启时，从FsImage及时恢复整个Namespace。图3所示为Namespace内存结构。前述集群中目录和文件总量即整个Namespace目录树中包含的节点总数，可见Namespace本身其实是一棵非常巨大的树。

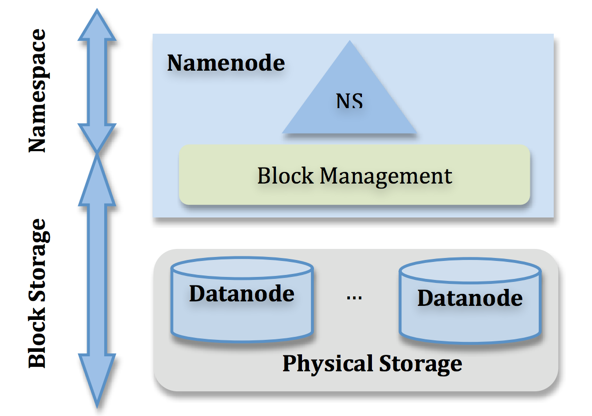


图3 Namespace内存结构

在整个Namespace目录树中存在两种不同类型的INode数据结构：INodeDirectory和INodeFile。其中INodeDirectory标识的是目录树中的目录，INodeFile标识的是目录树中的文件。由于二者均继承自INode，所以具备大部分相同的公共信息INodeWithAdditionalFields，除常用基础属性外，其中还提供了扩展属性features，如Quota、Snapshot等均通过Feature增加，如果以后出现新属性也可通过Feature方便扩展。不同的是，INodeFile特有的标识副本数和数据块大小组合的header（2.6.1之后又新增了标识存储策略ID的信息）及该文件包含的有序Blocks数组；INodeDirectory则特有子节点的列表children。这里需要特别说明children是默认大小为5的ArrayList，按照子节点name有序存储，虽然在插入时会损失一部分写性能，但是可以方便后续快速二分查找提高读性能，对一般存储系统，读操作比写操作占比要高。具体的继承关系见图4所示。

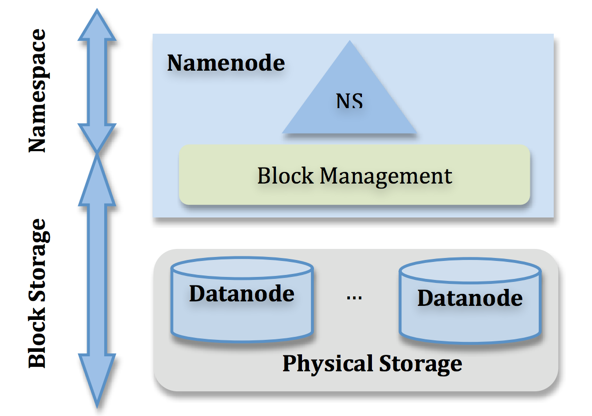


图4 INode继承关系

## 3.2 BlockManager

BlocksMap在NameNode内存空间占据很大比例，由BlockManager统一管理，相比Namespace，BlockManager管理的这部分数据要复杂的多。Namespace与BlockManager之间通过前面提到的INodeFile有序Blocks数组关联到一起。图5所示BlockManager管理的内存结构。

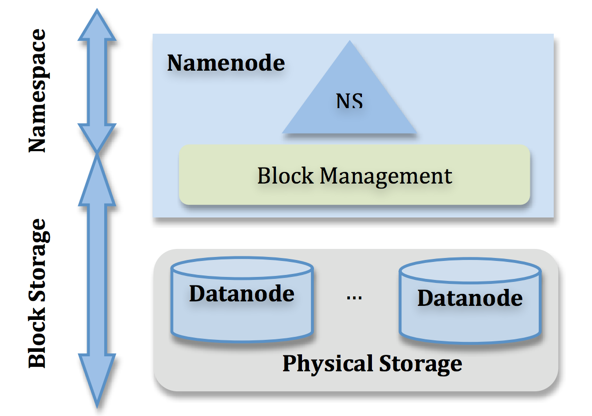


图5 BlockManager管理的内存结构

每一个INodeFile都会包含数量不等的Block，具体数量由文件大小及每一个Block大小（默认为64M）比值决定，这些Block按照所在文件的先后顺序组成BlockInfo数组，如图5所示的BlockInfo[A~K]，BlockInfo维护的是Block的元数据，结构如图6所示，数据本身是由DataNode管理，所以BlockInfo需要包含实际数据到底由哪些DataNode管理的信息，这里的核心是名为triplets的Object数组，大小为3\*replicas，其中replicas是Block副本数量。triplets包含的信息：

triplets[i]：Block所在的DataNode；

triplets[i+1]：该DataNode上前一个Block；

triplets[i+2]：该DataNode上后一个Block；

其中i表示的是Block的第i个副本，i取值[0,replicas)。

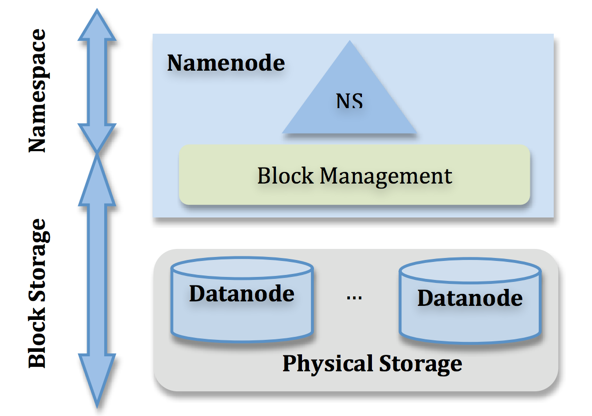


图6 BlockInfo继承关系

从前面描述可以看到BlockInfo几块重要信息：文件包含了哪些Block，这些Block分别被实际存储在哪些DataNode上，DataNode上所有Block前后链表关系。

如果从信息完整度来看，以上数据足够支持所有关于HDFS文件系统的正常操作，但还存在一个使用场景较多的问题：不能通过blockid快速定位Block，所以引入了BlocksMap。

BlocksMap底层通过LightWeightGSet实现，本质是一个链式解决冲突的哈希表。为了避免rehash过程带来的性能开销，初始化时，索引空间直接给到了整个JVM可用内存的2%，并且不再变化。集群启动过程，DataNode会进行BR（BlockReport），根据BR的每一个Block计算其HashCode，之后将对应的BlockInfo插入到相应位置逐渐构建起来巨大的BlocksMap。前面在INodeFile里也提到的BlockInfo集合，如果我们将BlocksMap里的BlockInfo与所有INodeFile里的BlockInfo分别收集起来，可以发现两个集合完全相同，事实上BlocksMap里所有的BlockInfo就是INodeFile中对应BlockInfo的引用；通过Block查找对应BlockInfo时，也是先对Block计算HashCode，根据结果快速定位到对应的BlockInfo信息。至此涉及到HDFS文件系统本身元数据的问题基本上已经解决了。

前面提到部分都属于静态数据部分，NameNode内存中所有数据都要随读写情况发生变化，BlockManager当然也需要管理这部分动态数据。主要是当Block发生变化不符合预期时需要及时调整Blocks的分布。这里涉及几个核心的数据结构：

excessReplicateMap：若某个Block实际存储的副本数多于预设副本数，这时候需要删除多余副本，这里多余副本会被置于excessReplicateMap中。excessReplicateMap是从DataNode的StorageID到Block集合的映射集。

neededReplications：若某个Block实际存储的副本数少于预设副本数，这时候需要补充缺少副本，这里哪些Block缺少多少个副本都统一存在neededReplications里，本质上neededReplications是一个优先级队列，缺少副本数越多的Block之后越会被优先处理。

invalidateBlocks：若某个Block即将被删除，会被置于invalidateBlocks中。invalidateBlocks是从DataNode的StorageID到Block集合的映射集。如某个文件被客户端执行了删除操作，该文件所属的所有Block会先被置于invalidateBlocks中。

corruptReplicas：有些场景Block由于时间戳/长度不匹配等等造成Block不可用，会被暂存在corruptReplicas中，之后再做处理。

前面几个涉及到Block分布情况动态变化的核心数据结构，这里的数据实际上是过渡性质的，BlockManager内部的ReplicationMonitor线程（图5标识Thread/Monitor）会持续从其中取出数据并通过逻辑处理后分发给具体的DatanodeDescriptor对应数据结构（3.3 NetworkTopology里会有简单介绍），当对应DataNode的心跳过来之后，NameNode会遍历DatanodeDescriptor里暂存的数据，将其转换成对应指令返回给DataNode，DataNode收到任务并执行完成后再反馈回NameNode，之后DatanodeDescriptor里对应信息被清除。如BlockB预设副本数为3，由于某种原因实际副本变成4（如之前下线的DataNode D重新上线，其中B正好有BlockB的一个副本数据），BlockManager能及时发现副本变化，并将多余的DataNode D上BlockB副本放置到excessReplicateMap中，ReplicationMonitor线程定期检查时发现excessReplicateMap中数据后将其移到DataNode D对应DatanodeDescriptor中invalidateBlocks里，当DataNode D下次心跳过来后，随心跳返回删除Block B的指令，DataNode D收到指令实际删除其上的Block B数据并反馈回NameNode，此后BlockManager将DataNode D上的Block B从内存中清除，至此Block B的副本符合预期，整个流程如图7所示。

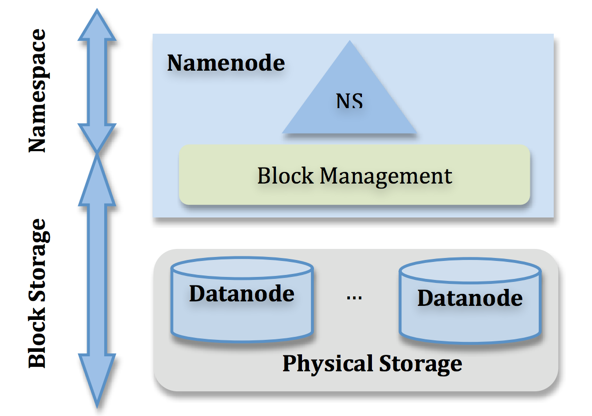


图7 副本数异常时处理过程

## 3.3 NetworkTopology

前面多次提到Block与DataNode之间的关联关系，事实上NameNode确实还需要管理所有DataNode，不仅如此，由于数据写入前需要确定数据块写入位置，NameNode还维护着整个机架拓扑NetworkTopology。图8所示内存中机架拓扑图。

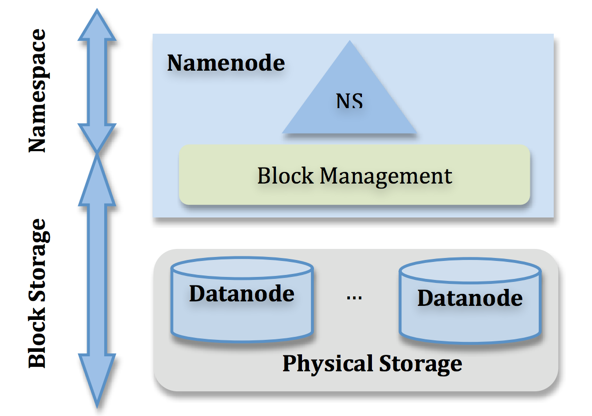


图8 NetworkTopology内存结构

从图8可以看出这里包含两个部分：机架拓扑结构NetworkTopology和DataNode节点信息。其中树状的机架拓扑是根据机架感知（一般都是外部脚本计算得到）在集群启动完成后建立起来，整个机架的拓扑结构在NameNode的生命周期内一般不会发生变化；另一部分是比较关键的DataNode信息，BlockManager已经提到每一个DataNode上的Blocks集合都会形成一个双向链表，更准确的应该是DataNode的每一个存储单元DatanodeStorageInfo上的所有Blocks集合会形成一个双向链表，这个链表的入口就是机架拓扑结构叶子节点即DataNode管理的DatanodeStorageInfo。此外由于上层应用对数据的增删查随时发生变化，随之DatanodeStorageInfo上的Blocks也会动态变化，所以NetworkTopology上的DataNode对象还会管理这些动态变化的数据结构，如replicateBlocks/recoverBlocks/invalidateBlocks，这些数据结构正好和BlockManager管理的动态数据结构对应，实现了数据的动态变化由BlockManager传达到DataNode内存对象最后通过指令下达到物理DataNode实际执行的流动过程，流程在3.2 BlockManager已经介绍。

这里存在一个问题，为什么DatanodeStorageInfo下所有Block之间会以双向链表组织，而不是其它数据结构？如果结合实际场景就不难发现，对每一个DatanodeStorageInfo下Block的操作集中在快速增加/删除（Block动态增减变化）及顺序遍历（BlockReport期间），所以双向链表是非常合适的数据结构。

## 3.4 LeaseManager

Lease 机制是重要的分布式协议，广泛应用于各种实际的分布式系统中。HDFS支持Write-Once-Read-Many，对文件写操作的互斥同步靠Lease实现。Lease实际上是时间约束锁，其主要特点是排他性。客户端写文件时需要先申请一个Lease，一旦有客户端持有了某个文件的Lease，其它客户端就不可能再申请到该文件的Lease，这就保证了同一时刻对一个文件的写操作只能发生在一个客户端。NameNode的LeaseManager是Lease机制的核心，维护了文件与Lease、客户端与Lease的对应关系，这类信息会随写数据的变化实时发生对应改变。

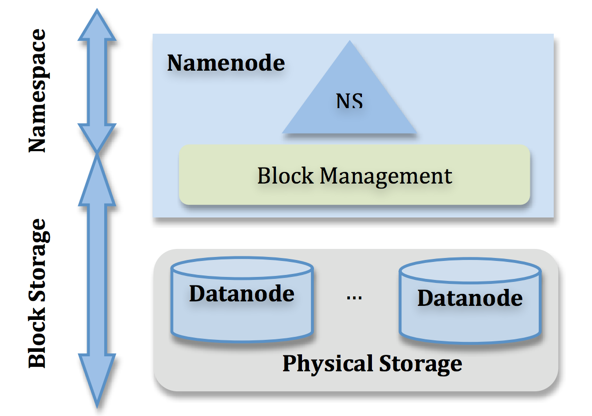


图9 LeaseManager的内存数据结构

图9所示为LeaseManager内存结构，包括以下三个主要核心数据结构：

sortedLeases：Lease集合，按照时间先后有序组织，便于检查Lease是否超时；

leases：客户端到Lease的映射关系；

sortedLeasesByPath：文件路径到Lease的映射关系；

其中每一个写数据的客户端会对应一个Lease，每个Lease里包含至少一个标识文件路径的Path。Lease本身已经维护了其持有者（客户端）及该Lease正在操作的文件路径集合，之所以增加了leases和sortedLeasesByPath为提高通过Lease持有者或文件路径快速索引到Lease的性能。

由于Lease本身的时间约束特性，当Lease发生超时后需要强制回收，内存中与该Lease相关的内容要被及时清除。超时检查及超时后的处理逻辑由LeaseManager.Monitor统一执行。LeaseManager中维护了两个与Lease相关的超时时间：软超时（softLimit）和硬超时（hardLimit），使用场景稍有不同。

正常情况下，客户端向集群写文件前需要向NameNode的LeaseManager申请Lease；写文件过程中定期更新Lease时间，以防Lease过期，周期与softLimit相关；写完数据后申请释放Lease。整个过程可能发生两类问题：（1）写文件过程中客户端没有及时更新Lease时间；（2）写完文件后没有成功释放Lease。两个问题分别对应为softLimit和hardLimit。两种场景都会触发LeaseManager对Lease超时强制回收。如果客户端写文件过程中没有及时更新Lease超过softLimit时间后，另一客户端尝试对同一文件进行写操作时触发Lease软超时强制回收；如果客户端写文件完成但是没有成功释放Lease，则会由LeaseManager的后台线程LeaseManager.Monitor检查是否硬超时后统一触发超时回收。不管是softLimit还是hardLimit超时触发的强制Lease回收，处理逻辑都一样：FSNamesystem.internalReleaseLease，逻辑本身比较复杂，这里不再展开，简单的说先对Lease过期前最后一次写入的Block进行检查和修复，之后释放超时持有的Lease，保证后面其它客户端的写入能够正常申请到该文件的Lease。

NameNode内存数据结构非常丰富，这里对几个重要的数据结构进行了简单的描述，除了前面罗列之外，其实还有如SnapShotManager/CacheManager等，由于其内存占用有限且有一些特性还尚未稳定，这里不再展开。

# 四、问题

随着集群中数据规模的不断积累，NameNode内存占用随之成比例增长。不可避免的NameNode内存将逐渐成为集群发展的瓶颈，并开始暴漏诸多问题。

1、启动时间变长。NameNode的启动过程可以分成FsImage数据加载、editlogs回放、Checkpoint、DataNode的BlockReport几个阶段。数据规模较小时，启动时间可以控制在~10min以内，当元数据规模达到5亿（Namespace中INode数超过2亿，Block数接近3亿），FsImage文件大小将接近到20GB，加载FsImage数据就需要~14min，Checkpoint需要~6min，再加上其它阶段整个重启过程将持续~50min，极端情况甚至超过60min，虽然经过多轮优化重启过程已经能够稳定在~30min，但也非常耗时。如果数据规模继续增加，启动过程将同步增加。

2、性能开始下降。HDFS文件系统的所有元数据相关操作基本上均在NameNode端完成，当数据规模的增加致内存占用变大后，元数据的增删改查性能会出现下降，且这种下降趋势会因规模效应及复杂的处理逻辑被放大，相对复杂的RPC请求（如addblock）性能下降更加明显。

3、NameNode JVM FGC（Full GC）风险较高。主要体现在两个方面：（1）FGC频率增加；（2）FGC时间增加且风险不可控。针对NameNode的应用场景，目前看CMS内存回收算法比较主流，正常情况下，对超过100GB内存进行回收处理时，可以控制到秒级别的停顿时间，但是如果回收失败被降级到串行内存回收时，应用的停顿时间将达到数百秒，这对应用本身是致命的。

4、超大JVM Heap Size调试问题。如果线上集群性能表现变差，不得不通过分析内存才能得到结论时，会成为一件异常困难的事情。且不说Dump本身极其费时费力，Dump超大内存时存在极大概率使NameNode不可服务。

针对NameNode内存增长带来的诸多问题，社区和业界都在持续关注并尝试不同的解决方案。整体上两个思路：（1）扩展NameNode分散单点负载；（2）引入外部系统支持NameNode内存数据。

从2010年开始社区就投入大量精力持续解决，Federation方案[3]通过对NameNode进行水平扩展分散单点负载的方式解决NameNode的问题，经过几年的发展该方案逐渐稳定，目前已经被业界广泛使用。除此之外，社区也在尝试将Namespace存储值外部的KV存储系统如LevelDB[4]，从而降低NameNode内存负载。

除社区外，业界也在尝试自己的解决方案。Baidu HDFS2[5]将元数据管理通过主从架构的集群形式提供服务，本质上是将原生NameNode管理的Namespace和BlockManagement进行物理拆分。其中Namespace负责管理整个文件系统的目录树及文件到BlockID集合的映射关系，BlockID到DataNode的映射关系是按照一定的规则分到多个服务节点分布式管理，这种方案与Lustre有相似之处（Hash-based Partition）。Taobao HDFS2[6]尝试过采用另外的思路，借助高速存储设备，将元数据通过外存设备进行持久化存储，保持NameNode完全无状态，实现NameNode无限扩展的可能。其它类似的诸多方案不一而足。

尽管社区和业界均对NameNode内存瓶颈有成熟的解决方案，但是不一定适用所有的场景，尤其是中小规模集群。结合实践过程和集群规模发展期可能遇到的NameNode内存相关问题这里有几点建议：

合并小文件。正如前面提到，目录/文件和Block均会占用NameNode内存空间，大量小文件会降低内存使用效率；另外，小文件的读写性能远远低于大文件的读写，主要原因对小文件读写需要在多个数据源切换，严重影响性能。

调整合适的BlockSize。主要针对集群内文件较大的业务场景，可以通过调整默认的Block Size大小（参数：dfs.blocksize，默认128M），降低NameNode的内存增长趋势。

HDFS Federation方案。当集群和数据均达到一定规模时，仅通过垂直扩展NameNode已不能很好的支持业务发展，可以考虑HDFS Federation方案实现对NameNode的水平扩展，在解决NameNode的内存问题的同时通过Federation可以达到良好的隔离性，不会因为单一应用压垮整集群。

# 五、总结

NameNode在整个HDFS系统架构中占据举足轻重的位置，内部数据和处理逻辑相对复杂，本文简单梳理了NameNode的内存全景及对其中几个关键数据结构，从NameNode内存核心数据视角对NameNode进行了简单的解读，并结合实际场景介绍了随着数据规模的增加，NameNode内存可能遇到的问题及业界各种可借鉴的解决方案

# 六、参考

[1] Apache Hadoop, 2016, <https://hadoop.apache.org/> .

[2] Apache Hadoop Source Code, 2014, <https://github.com/apache/hadoop/tree/branch-2.4.1/> .

[3] HDFS Federation, 2011, <https://issues.apache.org/jira/browse/HDFS-1052> .

[4] NemeNode Scalability, 2013, <https://issues.apache.org/jira/browse/HDFS-5389> .

[5] Baidu HDFS2, 2013, <http://static.zhizuzhefu.com/wordpress_cp/uploads/2013/04/a9.pdf>.

[6] Taobao HDFS2, 2012, <https://github.com/taobao/ADFS> .

来自：http://tech.meituan.com/namenode.html