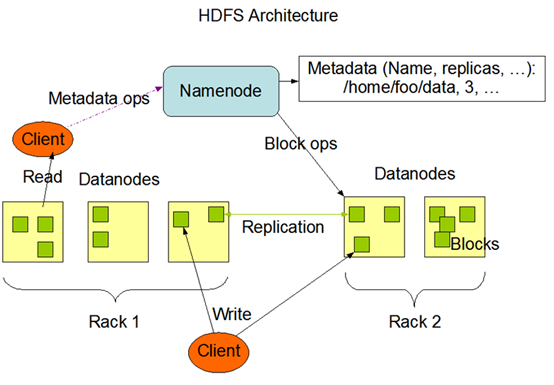
# HDFS 学习笔记

# HDFS 1.X

## 1 HDFS简介

         HDFS是Hadoop Distribute File System 的简称，也就是Hadoop的一个分布式文件系统. HDFS被设计成适合运行在通用硬件(commodity hardware)上的分布式文件系统。HDFS是一个高度容错性的系统，适合部署在廉价的机器上。HDFS能提供高吞吐量的数据访问，非常适合大规模数据集上的应用。HDFS放宽了一部分POSIX约束，来实现流式读取文件系统数据的目的。HDFS在最开始是作为Apache Nutch搜索引擎项目的基础架构而开发的。HDFS是Apache Hadoop Core项目的一部分。

## 2 HDFS基本架构图



## 3 HDFS相关术语

HDFS采用master/slave架构。一个HDFS集群是由一个NameNode和一定数目的DataNodes组成。

NameNode: NameNode是一个中心服务器，负责管理文件系统的命名空间(namespace)以及客户端对文件的访问。其将所有的文件和文件夹的元数据保存在一个文件系统树中。这些信息也会在硬盘上保存成以下文件：命名空间镜像(namespace image)及修改日志(edit log) 。其中fsimage镜像文件包含了整个HDFS文件系统的所有目录和文件的indoe信息。对于文件来说包括了数据块描述信息、修改时间、访问时间等；对于目录来说包括修改时间、访问权限控制信息(目录所属用户，所在组等)等。edit文件主要是在NameNode已经启动情况下对HDFS进行的各种更新操作进行记录，HDFS客户端执行所有的写操作都会被记录到edit文件中。NameNode 还保存了一个文件包括哪些数据块，分布在哪些数据节点上。然而这些信息并不存储在硬盘上，而是在系统启动的时候从数据节点收集而成的. NameNode是所有HDFS元数据的仲裁者和管理者，这样，用户数据永远不会流过NameNode。

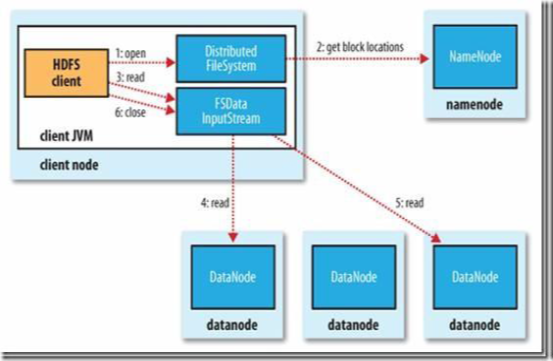
DataNode:  数据节点是文件系统中真正存储数据的地方。集群中的DataNode一般是一个节点一个，负责管理它所在节点上的存储。HDFS暴露了文件系统的名字空间，用户能够以文件的形式在上面存储数据。从内部看，一个文件其实被分成一个或多个数据块，这些块存储在一组DataNode上。NameNode执行文件系统的名字空间操作，比如打开、关闭、重命名文件或目录。它也负责确定数据块到具体DataNode节点的映射。DataNode负责处理文件系统客户端的读写请求。在NameNode的统一调度下进行数据块的创建、删除和复制。

Block(数据块)：   HDFS默认的最基本的存储单位是64M的数据块。和普通文件系统相同的是，HDFS中的文件是被分成64M一块的数据块存储的。不同于普通文件系统的是，HDFS中，如果一个文件小于一个数据块的大小，并不占用整个数据块存储空间。当然，数据块的大小是可以调整的，可以根据系统需求自定义，在Hadoop 1.X 系列版本中默认为64M,在Hadoop 2.X 系列版本中默认为128M

secondary NameNode (从元数据节点) ： 从元数据节点并不是元数据节点出现问题时候的备用节点，它和元数据节点负责不同的事情。其主要功能就是周期性将元数据节点的命名空间镜像文件和修改日志合并，以防日志文件过大。合并过后的命名空间镜像文件也在从元数据节点保存了一份，以防元数据节点失败的时候，可以恢复。

Namespace（文件系统的名字空间）：HDFS支持传统的层次型文件组织结构。用户或者应用程序可以创建目录，然后将文件保存在这些目录里。文件系统名字空间的层次结构和大多数现有的文件系统类似：用户可以创建、删除、移动或重命名文件。NameNode负责维护文件系统的名字空间，任何对文件系统名字空间或属性的修改都将被NameNode记录下来。应用程序可以设置HDFS保存的文件的副本数目。文件副本的数目称为文件的 **副本系数** ，这个信息也是由NameNode保存的。

## 4 HDFS读取文件的流程



•使用HDFS提供的客户端开发库Client，向远程的NameNode发起RPC请求；

•NameNode会视情况返回文件的部分或者全部block列表，对于每个block，NameNode都会返回有该block拷贝的DataNode地址；

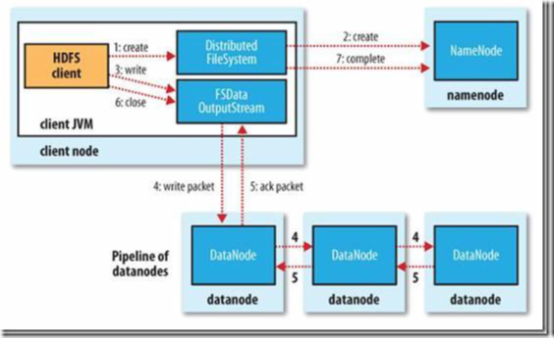
•客户端开发库Client会选取离客户端最接近的DataNode来读取block；如果客户端本身就是DataNode,那么将从本地直接获取数据.

•读取完当前block的数据后，关闭与当前的DataNode连接，并为读取下一个block寻找最佳的DataNode；

•当读完列表的block后，且文件读取还没有结束，客户端开发库会继续向NameNode获取下一批的block列表。

•读取完一个block都会进行checksum验证，如果读取DataNode时出现错误，客户端会通知NameNode，然后再从下一个拥有该block拷贝的DataNode继续读。

## 5 HDFS写文件的流程



•使用HDFS提供的客户端开发库Client，向远程的NameNode发起RPC请求；   
•NameNode会检查要创建的文件是否已经存在，创建者是否有权限进行操作，成功则会为文件创建一个记录，否则会让客户端抛出异常；   
•当客户端开始写入文件的时候，会将文件切分成多个packets，并在内部以数据队列"data queue"的形式管理这些packets，并向NameNode申请新的blocks，获取用来存储replicas的合适的DataNodes列表，列表的大小根据在NameNode中对replication的设置而定。

•开始以pipeline（管道）的形式将packet写入所有的replicas中。把packet以流的方式写入第一个DataNode，该DataNode把该packet存储之后，再将其传递给在此pipeline中的下一个DataNode，直到最后一个DataNode，这种写数据的方式呈流水线的形式。   
•最后一个DataNode成功存储之后会返回一个ack packet，在pipeline里传递至客户端，在客户端的开发库内部维护着"ack queue"，成功收到DataNode返回的ack packet后会从"ack queue"移除相应的packet。

•如果传输过程中，有某个DataNode出现了故障，那么当前的pipeline会被关闭，出现故障的DataNode会从当前的pipeline中移除，剩余的block会继续剩下的DataNode中继续以pipeline的形式传输，同时NameNode会分配一个新的DataNode，保持replicas设定的数量。

### 5.1 流水线复制

当客户端向 HDFS 文件写入数据的时候，一开始是写到本地临时文件中。假设该文件的副本系数设置为 3 ，当本地临时文件累积到一个数据块的大小时，客户端会从 NameNode 获取一个 DataNode 列表用于存放副本。然后客户端开始向第一个 DataNode 传输数据，第一个 DataNode 一小部分一小部分 (4 KB) 地接收数据，将每一部分写入本地仓库，并同时传输该部分到列表中 第二个 DataNode 节点。第二个 DataNode 也是这样，一小部分一小部分地接收数据，写入本地 仓库，并同时传给第三个 DataNode 。最后，第三个 DataNode 接收数据并存储在本地。因此， DataNode 能流水线式地从前一个节点接收数据，并在同时转发给下一个节点，数据以流水线的 方式从前一个 DataNode 复制到下一个

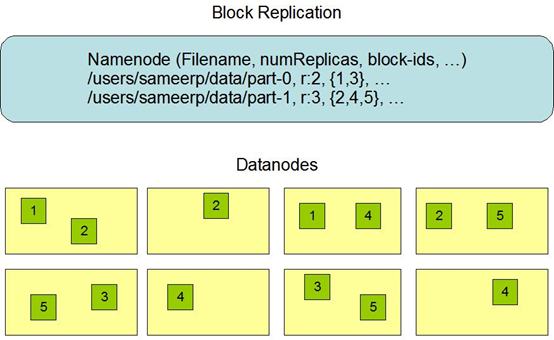
### 5.2 更细节的原理

客户端创建文件的请求其实并没有立即发送给 NameNode ，事实上，在刚开始阶 段 HDFS 客户端会先将文件数据缓存到本地的一个临时文件。应用程序的写操作被透 明地重定向到这个临时文件。当这个临时文件累积的数据量超过一个数据块的大小 ，客户端才会联系 NameNode 。 NameNode 将文件名插入文件系统的层次结构中，并 且分配一个数据块给它。然后返回 DataNode 的标识符和目标数据块给客户端。接着 客户端将这块数据从本地临时文件上传到指定的 DataNode 上。当文件关闭时，在临 时文件中剩余的没有上传的数据也会传输到指定的 DataNode 上。然后客户端告诉 NameNode 文件已经关闭。此时 NameNode 才将文件创建操作提交到日志里进行存储 。如果 NameNode 在文件关闭前宕机了，则该文件将丢失。

## 6 HDFS 数据复制

HDFS设计成能可靠地在集群中大量机器之间存储大量的文件，它以块序列的形式存储文件。文件中除了最后一个块，其他块都有相同的大小。属于文件的块为了故障容错而被复制。块的大小和复制数是以文件为单位进行配置的，应用可以在文件创建时或者之后修改复制因子。HDFS中的文件是一次写的，并且任何时候都只有一个写操作。

NameNode负责处理所有的块复制相关的决策。它周期性地接受集群中数据节点的心跳和块报告。一个心跳的到达表示这个数据节点是正常的。一个块报告包括该数据节点上所有块的列表。



副本位置：第一小步

块副本存放位置的选择严重影响HDFS的可靠性和性能。副本存放位置的优化是HDFS区分于其他分布式文件系统的的特征，这需要精心的调节和大量的经验。机架敏感的副本存放策略是为了提高数据的可靠性，可用性和网络带宽的利用率。副本存放策略的实现是这个方向上比较原始的方式。短期的实现目标是要把这个策略放在生产环境下验证，了解更多它的行为，为以后测试研究更精致的策略打好基础。

HDFS运行在跨越大量机架的集群之上。两个不同机架上的节点是通过交换机实现通信的，在大多数情况下，相同机架上机器间的网络带宽优于在不同机架上的机器。

在开始的时候，每一个数据节点自检它所属的机架id，然后在向名字节点注册的时候告知它的机架id。HDFS提供接口以便很容易地挂载检测机架标示的模块。一个简单但不是最优的方式就是将副本放置在不同的机架上，这就防止了机架故障时数据的丢失，并且在读数据的时候可以充分利用不同机架的带宽。这个方式均匀地将复制分散在集群中，这就简单地实现了组建故障时的负载均衡。然而这种方式增加了写的成本，因为写的时候需要跨越多个机架传输文件块。

默认的HDFS block放置策略在最小化写开销和最大化数据可靠性、可用性以及总体读取带宽之间进行了一些折中。一般情况下复制因子为3，HDFS的副本放置策略是将第一个副本放在本地节点，将第二个副本放到本地机架上的另外一个节点而将第三个副本放到不同机架上的节点。这种方式减少了机架间的写流量，从而提高了写的性能。机架故障的几率远小于节点故障。这种方式并不影响数据可靠性和可用性的限制，并且它确实减少了读操作的网络聚合带宽，因为文件块仅存在两个不同的机架， 而不是三个。文件的副本不是均匀地分布在机架当中，1/3在同一个节点上，1/3副本在同一个机架上，另外1/3均匀地分布在其他机架上。这种方式提高了写的性能，并且不影响数据的可靠性和读性能。

副本的选择

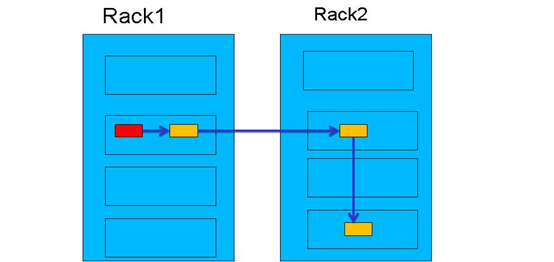
为了尽量减小全局的带宽消耗读延迟，HDFS尝试返回给一个读操作离它最近的副本。假如在读节点的同一个机架上就有这个副本，就直接读这个，如果HDFS集群是跨越多个数据中心，那么本地数据中心的副本优先于远程的副本。

## 7 HDFS 副本机制

### 7.1 特点

1. 数据类型单一   
2. 副本数比较多   
3. 写文件时副本的放置方法   
4. 动态的副本创建策略   
5. 弱化的副本一致性要求

### 7.2 ****副本摆放策略****



### ****7.3 修改副本数****

1.集群只有三个DataNode，hadoop系统replication=4时，会出现什么情况？   
        对于上传文件到hdfs上时，当时hadoop的副本系数是几，这个文件的块数副本数就会有几份，无论以后你怎么更改系统副本系统，这个文件的副本数都不会改变，也就说上传到分布式系统上的文件副本数由当时的系统副本数决定，不会受replication的更改而变化，除非用命令来更改文件的副本数。因为dfs.replication实质上是client参数，在create文件时可以指定具体replication，属性dfs.replication是不指定具体replication时的采用默认备份数。文件上传后，备份数已定，修改dfs.replication是不会影响以前的文件的，也不会影响后面指定备份数的文件。只影响后面采用默认备份数的文件。但可以利用hadoop提供的命令后期改某文件的备份数：hadoop fs -setrep -R 1。如果你是在hdfs-site.xml设置了dfs.replication，这并一定就得了，因为你可能没把conf文件夹加入到你的 project的classpath里，你的程序运行时取的dfs.replication可能是hdfs-default.xml里的 dfs.replication，默认是3。可能这个就是造成你为什么dfs.replication老是3的原因。你可以试试在创建文件时，显式设定replication。replication一般到3就可以了，大了意义也不大。

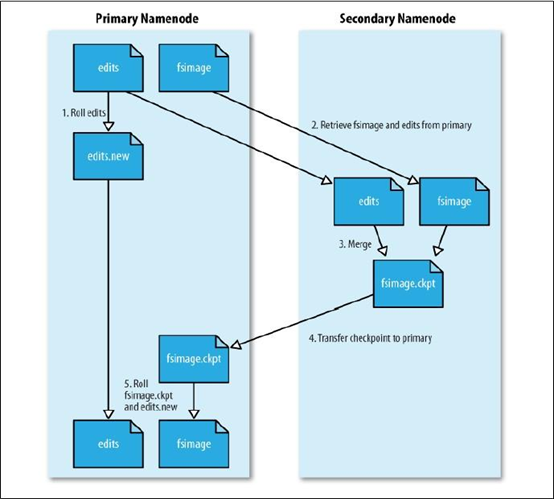
## ****8 HDFS负载均衡****

HDFS的数据也许并不是非常均匀的分布在各个DataNode中。一个常见的原因是在现有的集群上经常会增添新的DataNode节点。当新增一个数据块（一个文件的数据被保存在一系列的块中）时，NameNode在选择DataNode接收这个数据块之前，会考虑到很多因素。其中的一些考虑的是：   
•将数据块的一个副本放在正在写这个数据块的节点上。   
•尽量将数据块的不同副本分布在不同的机架上，这样集群可在完全失去某一机架的情况下还能存活。   
•一个副本通常被放置在和写文件的节点同一机架的某个节点上，这样可以减少跨越机架的网络I/O。   
•尽量均匀地将HDFS数据分布在集群的DataNode中。

## ****9 NameNode简单启动过程****

在HDFS中，任何一个文件，目录和block，在HDFS中都会被表示为一个object存储在NameNode的内存中，每一个object占用150 bytes的内存空间。当NameNode启动的时候，首先会将fsimage里面的所有内容映像到内存中，然后再一条一条地执行edits中的记录，然后等待各个DataNode向自己汇报块的信息来组装blockMap，从而离开安全模式。在这里涉及到BlockMap结构，所谓的BlockMap结构就是记录着block的元数据（加载在NameNode的内存中）和其对应的实际数据（存储在各个DataNode中）的映射关系。真正每个block对应到DataNodes列表的信息在hadoop中并没有进行持久化存储，而是在所有DataNode启动时，每个DataNode对本地磁盘进行扫描，将本DataNode上保存的block信息汇报给NameNode,NameNode在接收到每个DataNode的块信息汇报后，将接收到的块信息，以及其所在的DataNode信息等保存在内存中。HDFS就是通过这种块信息汇报的方式来完成 block -> DataNodes list的对应表构建。DataNode向NameNode汇报块信息的过程叫做blockReport，而NameNode将block -> DataNodes list的对应表信息保存在一个叫BlocksMap的数据结构中。因此，我们可以得出一个非常重要的结论，NameNode不会定期的向各个DataNode去”索取“块的信息，而是各个DataNode定期向NameNode汇报块的信息。当组装完NameNode组装完BlockMap的信息后基本上整个HDFS的启动就完成了，可以顺利地离开安全模式了。分析到这里，我们就可以很清楚地知道整个HDFS的启动速度是由上面决定的了，第一：执行各个edits文件，这个也是我这篇blog重点讨论的。第二：各个DataNode向NameNode汇报块信息的进度（当99.9%的block汇报完毕才会离开安全模式）。

## 10 Secondary NameNode处理流程



(1) NameNode 响应 Secondary NameNode 请求，将 edit log 推送给 Secondary NameNode，开始重新写一个新的edit log。

(2)Secondary NameNode 收到来自 NameNode 的 fsimage 文件和 edit log 。

(3)Secondary NameNode 将 fsimage 加载到内存，应用 edit log ， 并生成一 个新的 fsimage 文件。

(4)Secondary NameNode 将新的 fsimage 推送给 NameNode 。

(5)NameNode用新的fsimage取代旧的fsimage,在fstime文件中记下检查 点发生的时

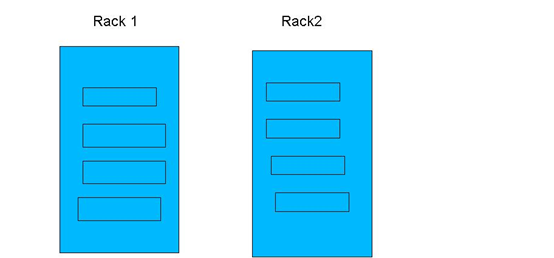
## 11 HDFS 的安全模式

 NameNode 启动后会进入一个称为安全模式的特殊状态。处于安全模式 的 NameNode 是不会进行数据块的复制的。 NameNode 从所有的 DataNode 接收心跳信号和块状态报告。块状态报告包括了某个 DataNode 所有的数据 块列表。每个数据块都有一个指定的最小副本数。当 NameNode 检测确认某 个数据块的副本数目达到这个最小值，那么该数据块就会被认为是副本安全 (safely replicated) 的；在一定百分比（这个参数可配置）的数据块被 NameNode 检测确认是安全之后（加上一个额外的 30 秒等待时间）， NameNode 将退出安全模式状态。接下来它会确定还有哪些数据块的副本没 有达到指定数目，并将这些数据块复制到其他 DataNode 上。

## ****12 HDFS 机架感知****

通常，大型 Hadoop 集群是以机架的形式来组织的，同一个机架上不同节点间的网络状况比不同机架之间的更为理想。另外，NameNode设法将数据块副本保存在不同的机架上以提高容错性。

而 HDFS不能够自动判断集群中各个DataNode的网络拓扑情况Hadoop 允许集群的管理员通过配置 dfs.network.script 参数来确定节点所处的机架。文件提供了IP->rackid的翻译。 NameNode 通过这个得到集群中各个DataNode机器的rackid 。如果 topology.script.file.name 没有设定，则每个IP都会翻译成 / default-rack 。



有了机架感知，NameNode就可以画出上图所示的 DataNode 网络拓扑图。 D1,R1 都是交换机，最底层是 DataNode 。 则 H1 的 rackid=/D1/R1/H1 ， H1的parent是R1，R1的是D1。 这些 rackid 信息可以通过 topology.script.file.name 配置。有了这些 rackid 信息就可以计算出任意两台 DataNode 之间的距离。

distance(/D1/R1/H1,/D1/R1/H1)=0  相同的 DataNode

distance(/D1/R1/H1,/D1/R1/H2)=2  同一 rack 下的不同 DataNode

distance(/D1/R1/H1,/D1/R1/H4)=4  同一 IDC 下的不同 DataNode

distance(/D1/R1/H1,/D2/R3/H7)=6   不同 IDC 下的 DataNode

## ****13 HDFS 健壮性****

HDFS 的主要目标就是即使在出错的情况下也要保证数据存储的可靠性。 常见的三种出错情况是： NameNode 出错 , DataNode 出错和网络割裂 ( network partitions) 。

### ****13.1 磁盘数据错误，心跳检测和重新复制****

每个 DataNode 节点周期性地向 NameNode 发送心跳信号。网络割裂可能 导致一部分 DataNode 跟 NameNode 失去联系。 NameNode 通过心跳信号的缺 失来检测这一情况，并将这些近期不再发送心跳信号 DataNode 标记为宕机 ，不会再将新的 IO 请求发给它们。任何存储在宕机 DataNode 上的数据将不 再有效。 DataNode 的宕机可能会引起一些数据块的副本系数低于指定值， NameNode 不断地检测这些需要复制的数据块，一旦发现就启动复制操作。 在下列情况下，可能需要重新复制：某个 DataNode 节点失效，某个副本遭 到损坏， DataNode 上的硬盘错误，或者文件的副本系数增大。

### ****13.2 数据完整性****

         从某个 DataNode 获取的数据块有可能是损坏的，损坏可能是由 DataNode 的存储设备错误、网络错误或者软件 bug 造成的。 HDFS 客户端软件实现了对 HDFS 文件内容的校验和 (checksum) 检查。当客户端创建一个新 的 HDFS 文件，会计算这个文件每个数据块的校验和，并将校验和作为一个单独的隐藏文件保存在同一个 HDFS 名字空间下。当客户端获取文件内容后 ，它会检验从 DataNode 获取的数据跟相应的校验和文件中的校验和是否匹配，如果不匹配，客户端可以选择从其他 DataNode 获取该数据块的副本。

### 13.3元数据磁盘错误

         FsImage 和 Editlog 是 HDFS 的核心数据结构。如果这些文件损坏了，整个 HDFS 实例都将失效。因而， NameNode 可以配置成支持维护多个 FsImage 和 Editlog 的副本。任何对 FsImage 或者 Editlog 的修改，都将同步到它们的副 本上。这种多副本的同步操作可能会降低 NameNode 每秒处理的名字空间事 务数量。然而这个代价是可以接受的，因为即使 HDFS 的应用是数据密集的 ，它们也非元数据密集的。当 NameNode 重启的时候，它会选取最近的完整 的 FsImage 和 Editlog 来使用。

       NameNode 是 HDFS 集群中的单点故障 (single point of failure) 所在。如果 NameNode 机器故障，是需要手工干预的。目前，自动重启或在另一台机器 上做 NameNode 故障转移的功能还没实现。

## ****14 HDFS 文件删除恢复机制****

当用户或应用程序删除某个文件时，这个文件并没有立刻从 HDFS 中删 除。实际上， HDFS 会将这个文件重命名转移到 /trash 目录。只要文件还在 /trash 目录中，该文件就可以被迅速地恢复。文件在 /trash 中保存的时间是可 配置的，当超过这个时间时， NameNode 就会将该文件从名字空间中删除。 删除文件会使得该文件相关的数据块被释放。注意，从用户删除文件到 HDFS 空闲空间的增加之间会有一定时间的延迟。

      只要被删除的文件还在 /trash 目录中，用户就可以恢复这个文件。如果 用户想恢复被删除的文件，他/她可以浏览/trash目录找回该文件。/trash目录仅仅保存被删除文件的最后副本。/trash目录与其他的目录没有什么区别，除了一点：在该目录上HDFS会应用一个特殊策略来自动删除文件。目前的默认策略是删除/trash中保留时间超过 6小时的文件。将来，这个策略可以通过一个被良好定义的接口配置。

### ****14.1 开启回收站****

hdfs-site.xml

<configuration>

       <property>

               <name>fs.trash.interval</name>

                <value> 1440</value>

                <description>Number ofminutes between trash checkpoints.

                        If zero, the trashfeature is disabled.

                </description>

       </property>

</configuration>

1, fs.trash.interval参数设置保留时间为 1440 分钟(1天)

2, 回收站的位置：在HDFS上的 /user/$USER/.Trash/Current/

## ****15 HDFS 分布式缓存（DistributedCache ）****

（1）在HDFS上准备好要共享的数据(text、archive、jar)，你拼路径的时候必须加前缀"file://"说明是本地路径，否则hadoop默认访问的路径是hdfs。

（2）DistributedCache 在 Mapper 或者 Reducer 启动时会被 copy to local，然后被 DistributedCache.getLocalCacheFiles() 调用，运行完 job 后 local cache file 会被删掉，如果另一个 job 也需要这样一份文件，需要重新添加、重新缓存，因为在分布式场景下 task 并不知道该 node 是否存在 cache file。如果在同台机器已经有了dist cache file,不会再次download，DistributedCache 根据缓存文档修改的时间戳进行追踪。 在作业执行期间，当前应用程序或者外部程序不能修改缓存文件，所以分布式缓存一般用来缓存只读文件。

（3）DistributedCache 在添加的时候注意要添加具体的文件，如果你添加目录，DistributedCache 将不会自动遍历、识别目录下的文件。

## ****16 HDFS 缺点****

### ****16.1 大量小文件****

因为 NameNode 把文件系统的元数据放置在内存中，所以文件系统所能 容纳的文件数目是由 NameNode 的内存大小来决定。一般来说，每一个文件 、文件夹和 Block 需要占据 150 字节左右的空间，所以，如果你有 100 万个文件，每一个占据一个 Block ，你就至少需要 300MB 内存。当前来说，数百万的文件还是可行的，当扩展到数十亿时，对于当前的硬件水平来说就没法实 现了。还有一个问题就是，因为 Map task 的数量是由 splits 来决定的，所以用 MR 处理大量的小文件时，就会产生过多的 Maptask ，线程管理开销将会增加作业时间。举个例子，处理 10000M 的文件，若每个 split 为 1M ，那就会有 10000 个 Maptasks ，会有很大的线程开销；若每个 split 为 100M ，则只有 100 个 Maptasks ，每个 Maptask 将会有更多的事情做，而线程的管理开销也将减小很多。

**改进策略**：要想让HDFS能处理好小文件，有不少方法。

利用SequenceFile、MapFile、Har等方式归档小文件，这个方法的原理就是把小文件归档起来管理，HBase就是基于此的。对于这种方法，如果想找回原来的小文件内容，那就必须得知道与归档文件的映射关系。

1）横向扩展，一个Hadoop集群能管理的小文件有限，那就把几个Hadoop集群拖在一个虚拟服务器后面，形成一个大的Hadoop集群。google也是这么干过的。

2）多Master设计，这个作用显而易见了。正在研发中的GFS II也要改为分布式多Master设计，还支持Master的Failover，而且Block大小改为1M，有意要调优处理小文件啊。

3）附带个Alibaba DFS的设计，也是多Master设计，它把Metadata的映射存储和管理分开了，由多个Metadata存储节点和一个查询Master节点组成。

### 16.2 不适合低延迟数据访问

         如果要处理一些用户要求时间比较短的低延迟应用请求，则HDFS不适合。HDFS是为了处理大型数据集分析任务的，主要是为达到高的数据吞吐量而设计的，这就可能要求以高延迟作为代价。

**改进策略**：对于那些有低延时要求的应用程序，HBase是一个更好的选择。通过上层数据管理项目来尽可能地弥补这个不足。在性能上有了很大的提升，它的口号就是goes real time。使用缓存或多master设计可以降低client的数据请求压力，以减少延时。还有就是对HDFS系统内部的修改，这就得权衡大吞吐量与低延时了，HDFS不是万能的银弹。

# HDFS 2.X 新特性

## 1 HDFS 快照

HDFS快照是一个只读的,基于时间点的文件系统拷贝。

快照可以是整个文件系统，也可以是一部分。

常用来作为数据备份，防止用户错误和容灾。

HDFS实现了：

1.Snapshot 创建的时间 复杂度为O(1)，但是不包括INode 的寻找时间

2.只有当修改SnapShot时，才会有额外的内存占用，内存使用量为O(M),M 为修改的文件或者目录数

3.在DataNode上面的blocks不会复制，做Snapshot的文件是纪录了block的列表和文件的大小，但是没有数据的复制

4.Snapshot并不会影响HDFS 的正常操作：修改会按照时间的反序记录，这样可以直接读取到最新的数据。快照数据是当前数据减去修改的部分计算出来的。

## 2 NameNode HA和Federation（联邦）出现背景

### 2.1 ****单点故障****

在Hadoop 2.0之前，也有若干技术试图解决单点故障的问题，我们在这里做个简短的总结

·Secondary NameNode。它不是HA，它只是阶段性的合并edits和fsimage，以缩短集群启动的时间。当NameNode(以下简称NN)失效的时候，Secondary NN并无法立刻提供服务，Secondary NN甚至无法保证数据完整性：如果NN数据丢失的话，在上一次合并后的文件系统的改动会丢失。

· Backup NameNode ([HADOOP-4539](https://issues.apache.org/jira/browse/HADOOP-4539))。它在内存中复制了NN的当前状态，算是Warm Standby，可也就仅限于此，并没有failover等。它同样是阶段性的做checkpoint，也无法保证数据完整性。

·手动把name.dir指向NFS。这是安全的Cold Standby，可以保证元数据不丢失，但集群的恢复则完全靠手动。

· [Facebook AvatarNode](http://hadoopblog.blogspot.tw/2010/02/hadoop-namenode-high-availability.html)。Facebook有强大的运维做后盾，所以Avatarnode只是Hot Standby，并没有自动切换，当主NN失效的时候，需要管理员确认，然后手动把对外提供服务的虚拟IP映射到Standby NN，这样做的好处是确保不会发生脑裂的场景。其某些设计思想和Hadoop 2.0里的HA非常相似，从时间上来看，Hadoop 2.0应该是借鉴了Facebook的做法。

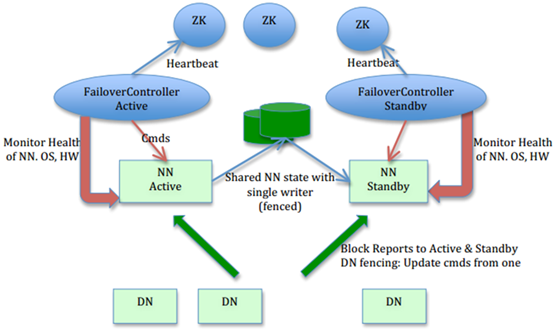
·还有若干解决方案，基本都是依赖外部的HA机制，譬如[DRBD](http://zh.wikipedia.org/zh/DRBD)，[Linux HA](http://www.linux-ha.org/wiki/Main_Page)，[VMware的FT](http://www.vmware.com/products/datacenter-virtualization/vsphere/fault-tolerance.html)等等。

### ****2. 2 集群容量和集群性能****

单NN的架构使得HDFS在集群扩展性和性能上都有潜在的问题，当集群大到一定程度后，NN进程使用的内存可能会达到上百G，常用的估算公式为1G对应1百万个块，按缺省块大小计算的话，大概是64T (这个估算比例是有比较大的富裕的，其实，即使是每个文件只有一个块，所有元数据信息也不会有1KB/block)。同时，所有的元数据信息的读取和操作都需要与NN进行通信，譬如客户端的addBlock、getBlockLocations，还有DataNode的blockRecieved、sendHeartbeat、blockReport，在集群规模变大后，NN成为了性能的瓶颈。Hadoop 2.0里的HDFS Federation就是为了解决这两个问题而开发的。

## 3 HDFS NameNode HA

### 3.1 NameNode HA架构



NameNode HA包括两个NN，主（active）与备（standby），ZKFC，ZK，share editlog。流程：集群启动后一个NN处于active状态，并提供服务，处理客户端和DataNode的请求，并把editlog写到本地和share editlog（可以是NFS，QJM等）中。另外一个NN处于Standby状态，它启动的时候加载fsimage，然后周期性的从share editlog中获取editlog，保持与active的状态同步。为了实现standby在sctive挂掉后迅速提供服务，需要DN同时向两个NN汇报，使得Stadnby保存block to DataNode信息，因为NN启动中最费时的工作是处理所有DataNode的blockreport。为了实现热备，增加FailoverController和ZK，FailoverController与ZK通信，通过ZK选主，FailoverController通过RPC让NN转换为active或standby。

### 3.2 ZKFC的设计

1). FailoverController实现下述几个功能

  (a) 监控NN的健康状态

  (b) 向ZK定期发送心跳，使自己可以被选举。

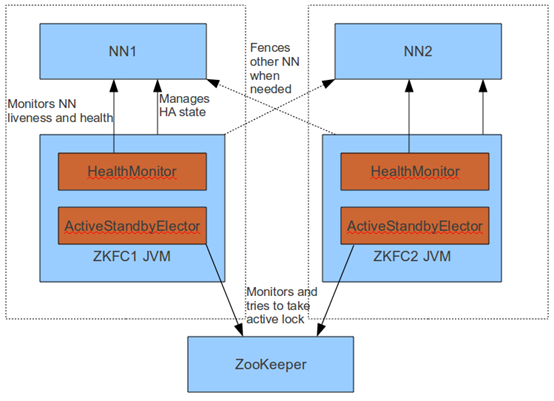
  (c) 当自己被ZK选为主时，active FailoverController通过RPC调用使相应的NN转换为active。

2). 为什么要作为一个deamon进程从NN分离出来

  (1) 防止因为NN的GC失败导致心跳受影响。

  (2) FailoverController功能的代码应该和应用的分离，提高的容错性。

  (3) 使得主备选举成为可插拔式的插件。



3). FailoverController主要包括三个组件，

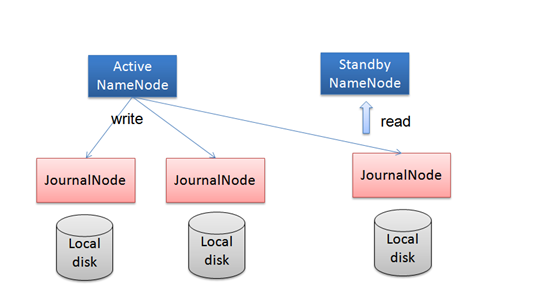
  (1) HealthMonitor 监控NameNode是否处于unavailable或unhealthy状态。当前通过RPC调用NN相应的方法完成。

  (2) ActiveStandbyElector 管理和监控自己在ZK中的状态。

  (3) ZKFailoverController 它订阅HealthMonitor 和ActiveStandbyElector 的事件，并管理NameNode的状态。

### 3.3 QJM的设计

NameNode记录了HDFS的目录文件等元数据，客户端每次对文件的增删改等操作，NameNode都会记录一条日志，叫做editlog，而元数据存储在fsimage中。为了保持Stadnby与active的状态一致，standby需要尽量实时获取每条editlog日志，并应用到FsImage中。这时需要一个共享存储，存放editlog，standby能实时获取日志。这有两个关键点需要保证， 共享存储是高可用的，需要防止两个NameNode同时向共享存储写数据导致数据损坏。 是什么，Qurom Journal Manager，基于Paxos（基于消息传递的一致性算法）。这个算法比较难懂，简单的说，Paxos算法是解决分布式环境中如何就某个值达成一致，（一个典型的场景是，在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点都执行相同的操作序列，那么他们最后能得到一个一致的状态。为保证每个节点执行相同的命令序列，需要在每一条指令上执行一个'一致性算法'以保证每个节点看到的指令一致）



(1) 初始化后，Active把editlog日志写到2N+1上JN上，每个editlog有一个编号，每次写editlog只要其中大多数JN返回成功（即大于等于N+1）即认定写成功。

(2) Standby定期从JN读取一批editlog，并应用到内存中的FsImage中。

(3) 如何fencing： NameNode每次写Editlog都需要传递一个编号Epoch给JN，JN会对比Epoch，如果比自己保存的Epoch大或相同，则可以写，JN更新自己的Epoch到最新，否则拒绝操作。在切换时，Standby转换为Active时，会把Epoch+1，这样就防止即使之前的NameNode向JN写日志，也会失败。

(4) 写日志：

  (a) NN通过RPC向N个JN异步写Editlog，当有N/2+1个写成功，则本次写成功。

  (b) 写失败的JN下次不再写，直到调用滚动日志操作，若此时JN恢复正常，则继续向其写日志。

  (c) 每条editlog都有一个编号txid，NN写日志要保证txid是连续的，JN在接收写日志时，会检查txid是否与上次连续，否则写失败。

(5) 读日志：

  (a) 定期遍历所有JN，获取未消化的editlog，按照txid排序。

  (b) 根据txid消化editlog。

(6) 切换时日志恢复机制

  (a) 主从切换时触发

  (b) 准备恢复（prepareRecovery），standby向JN发送RPC请求，获取txid信息，并对选出最好的JN。

  (c) 接受恢复（acceptRecovery），standby向JN发送RPC，JN之间同步Editlog日志。

  (d) Finalized日志。即关闭当前editlog输出流时或滚动日志时的操作。

  (e) Standby同步editlog到最新

(7) 如何选取最好的JN

  (a) 有Finalized的不用in-progress

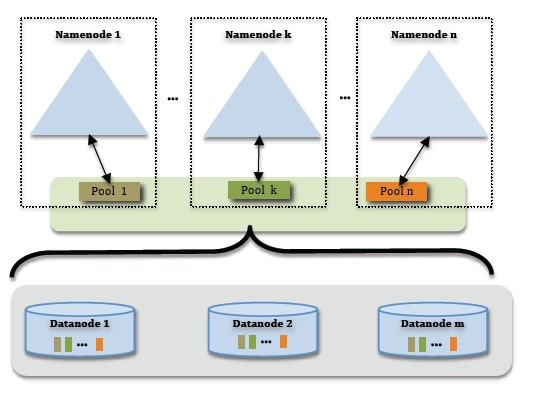
  (b) 多个Finalized的需要判断txid是否相等

  (c) 没有Finalized的首先看谁的epoch更大

  (d) Epoch一样则选txid大的。

## 4 HDFS Federation(联邦)

### 4.1 HDFS Federation 架构图



为了水平扩展NameNode，federation使用了多个独立的NameNode/namespace。这些NameNode之间是联合的，也就是说，他们之间相互独立且不需要互相协调，各自分工，管理自己的区域。分布式的DataNode被用作通用的数据块存储存储设备。每个DataNode要向集群中所有的NameNode注册，且周期性地向所有NameNode发送心跳和块报告，并执行来自所有NameNode的命令。

一个block pool由属于同一个namespace的数据块组成，每个DataNode可能会存储集群中所有block pool的数据块。

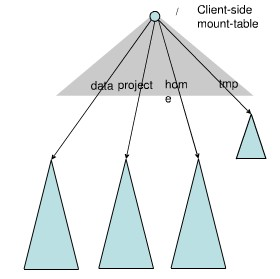
每个block pool内部自治，也就是说各自管理各自的block，不会与其他block pool交流。一个NameNode挂掉了，不会影响其他NameNode。

某个NameNode上的namespace和它对应的block pool一起被称为namespace volume。它是管理的基本单位。当一个NameNode/nodespace被删除后，其所有DataNode上对应的block pool也会被删除。当集群升级时，每个namespace volume作为一个基本单元进行升级。

### 4.2 Federation关键技术点

【命名空间管理】

Federation中存在多个命名空间，如何划分和管理这些命名空间非常关键。在Federation中并未采用“文件名hash”的方法，因为该方法的locality非常差，比如：查看某个目录下面的文件，如果采用文件名hash的方法存放文件，则这些文件可能被放到不同namespace中，HDFS需要访问所有namespace，代价过大。为了方便管理多个命名空间，HDFS Federation采用了经典的Client Side Mount Table。



如上图所示，下面四个深色三角形代表一个独立的命名空间，上方浅色的三角形代表从客户角度去访问的子命名空间。各个深色的命名空间Mount到浅色的表中，客户可以访问不同的挂载点来访问不同的命名空间，这就如同在Linux系统中访问不同挂载点一样。这就是HDFS Federation中命名空间管理的基本原理：将各个命名空间挂载到全局mount-table中，就可以做将数据到全局共享；同样的命名空间挂载到个人的mount-table中，这就成为应用程序可见的命名空间视图。

【Block Pool】

所谓Block pool(块池)就是属于单个命名空间的一组block(块)。每一个DataNode为所有的block pool存储块。DataNode是一个物理概念，而block pool是一个重新将block划分的逻辑概念。同一个DataNode中可以存着属于多个block pool的多个块。Block pool允许一个命名空间在不通知其他命名空间的情况下为一个新的block创建Block ID。同时，一个NameNode失效不会影响其下的DataNode为其他NameNode的服务。

当DataNode与NameNode建立联系并开始会话后自动建立Block pool。每个block都有一个唯一的标识，这个标识我们称之为扩展的块ID（Extended Block ID）= BlockID+BlockID。这个扩展的块ID在HDFS集群之间都是唯一的，这为以后集群归并创造了条件。

DataNode中的数据结构都通过块池ID（BlockPoolID）索引，即DataNode中的BlockMap，storage等都通过BPID索引。

在HDFS中，所有的更新、回滚都是以NameNode和BlockPool为单元发生的。即同一HDFS Federation中不同的NameNode/BlockPool之间没有什么关系。

### 4.3 Federation 主要优点

【扩展性和隔离性】

支持多个NameNode水平扩展整个文件系统的namespace。可按照应用程序的用户和种类分离namespace volume，进而增强了隔离性。

【通用存储服务】

Block Pool抽象层为HDFS的架构开启了创新之门。分离block storage layer使得：

<1> 新的文件系统（non-HDFS）可以在block storage上构建

<2> 新的应用程序（如HBase）可以直接使用block storage层

<3> 分离的block storage层为将来完全分布式namespace打下基础

【设计简单】

Federation 整个核心设计实现大概用了4个月。大部分改变是在DataNode、Config和Tools中，而NameNode本身的改动非常少，这样 NameNode原先的鲁棒性（健壮性）不会受到影响。虽然这种实现的扩展性比起真正的分布式的NameNode要小些，但是可以迅速满足需求，另外Federation具有良好的向后兼容性，已有的单NameNode的部署配置不需要任何改变就可以继续工作

### 4.4 Federation不足

【单点故障问题】

HDFS Federation并没有完全解决单点故障问题。虽然NameNode/namespace存在多个，但是从单个NameNode/namespace看，仍然存在单点故障：如果某个NameNode挂掉了，其管理的相应的文件便不可以访问。Federation中每个NameNode仍然像之前HDFS上实现一样，配有一个secondary NameNode，以便主NameNode挂掉一下，用于还原元数据信息。

【负载均衡问题】

HDFS Federation采用了Client Side Mount Table分摊文件和负载，该方法更多的需要人工介入以达到理想的负载均衡。

