HDFS 架构学习总结

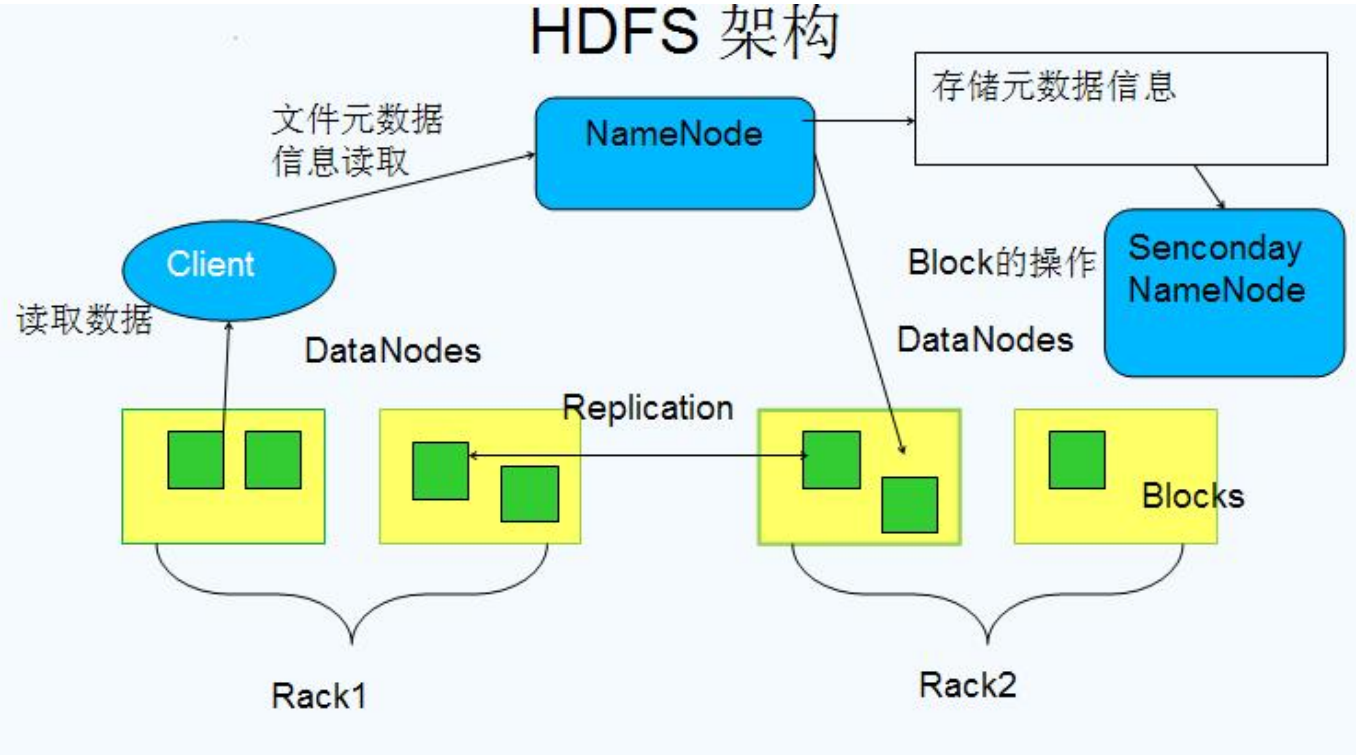
Jul 15, 2018 | 技术 | 4,533 字 | 1611 Times

HDFS（Hadoop Distributed File System）是一个分布式文件存储系统，几乎是离线存储领域的标准解决方案（有能力自研的大厂列外），业内应用非常广泛。近段抽时间，看一下 HDFS 的架构设计，虽然研究生也学习过相关内容，但是现在基本忘得差不多了，今天抽空对这块做了一个简单的总结，也算是再温习了一下这块的内容，这样后续再看 HDFS 方面的文章时，不至于处于懵逼状态。

文章目录

HDFS 1.0 架构

HDFS 采用的是 Master/Slave 架构，一个 HDFS 集群包含一个单独的 NameNode 和多个 DataNode 节点，如下图所示（这个图是 HDFS1.0的架构图，经典的架构图）：



HDFS 1.0 架构图

NameNode

NameNode 负责管理整个分布式系统的元数据，主要包括：

- **目录树结构；**
- **文件到数据库 Block 的映射关系；**
- **Block 副本及其存储位置等管理数据；**
- **DataNode 的状态监控，两者通过段时间间隔的心跳来传递管理信息和数据信息，通过这种方式的信息传递，NameNode 可以获知每个 DataNode 保存的 Block 信息、DataNode 的健康状况、命令 DataNode 启动停止等（如果发现某个 DataNode 节点故障，NameNode 会将其负责的 block 在其他 DataNode 上进行备份）。**

这些数据保存在内存中，同时在磁盘保存两个元数据管理文件：fsimage 和 editlog。

- **fsimage：是内存命名空间元数据在外存的镜像文件；**
- **editlog：则是各种元数据操作的 write-ahead-log 文件，在体现到内存数据变化前首先会将操作记入 editlog 中，以防止数据丢失。**

这两个文件相结合可以构造完整的内存数据。

Secondary NameNode

Secondary NameNode 并不是 NameNode 的热备机，而是定期从 NameNode 拉取 fsir e 和 editlog 文件，并对两个文件进行合并，形成新的 fsimage 文件并传回 NameNode，这样做的目的

是减轻 NameNod 的工作压力，本质上 SNN 是一个提供检查点功能服务的服务点。

DataNode

负责数据块的实际存储和读写工作，Block 默认是64MB（HDFS2.0改成了128MB），当客户端上传一个大文件时，HDFS 会自动将其切割成固定大小的 Block，为了保证数据可用性，每个 Block 会以多备份的形式存储，默认是3份。

文件写入过程

Client 向 HDFS 文件写入的过程可以参考HDFS写文件过程分析，整体过程如下图（这个图比较经典，最开始来自《Hadoop：The Definitive Guide》）所示：

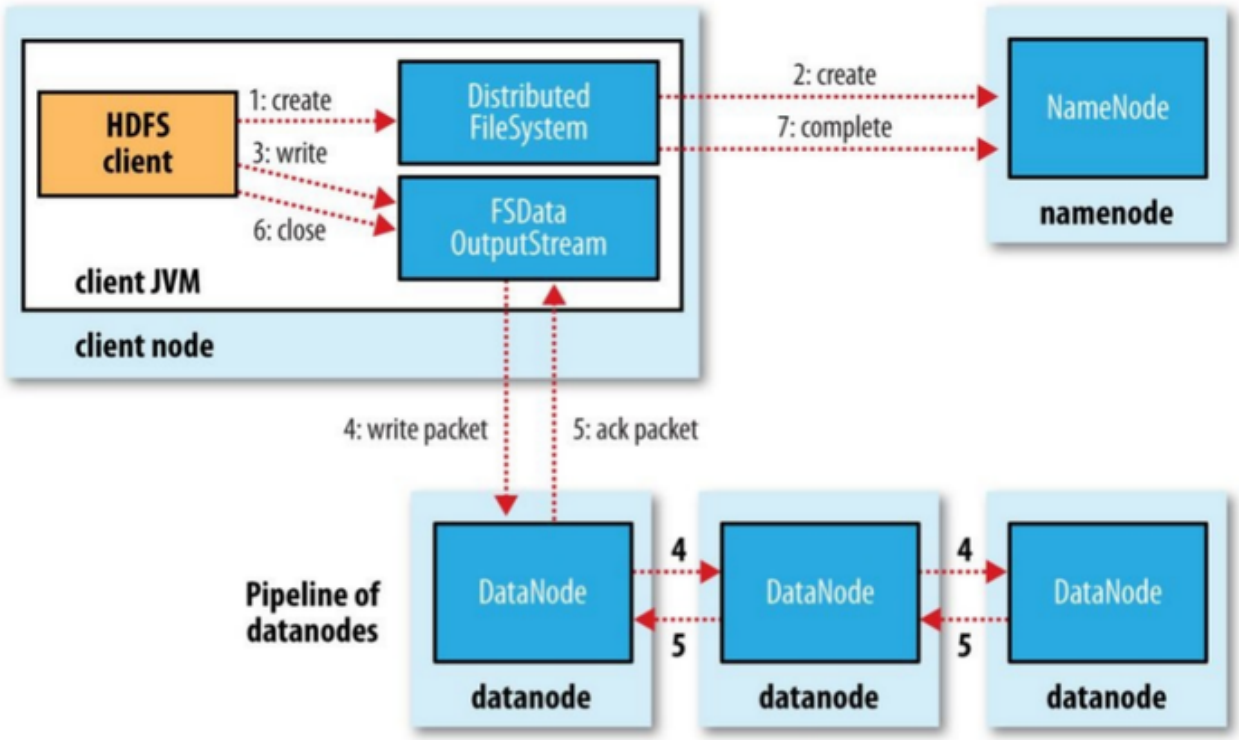


Figure 3-4. A client writing data to HDFS

HDFS 文件写入过程

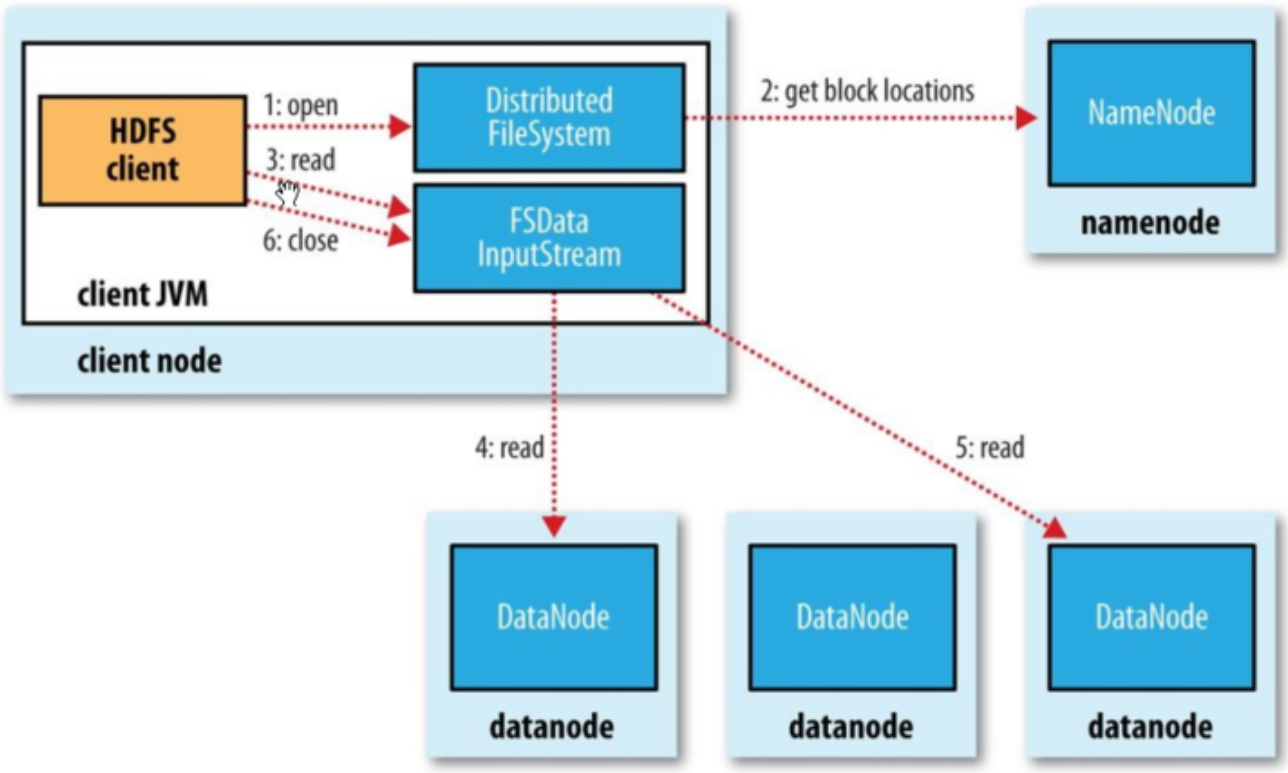
具体过程如下：

1. Client 调用 DistributedFileSystem 对象的 `create` 方法，创建一个文件输出流（FSDataOutputStream）对象；
2. 通过 DistributedFileSystem 对象与集群的 NameNode 进行一次 RPC 远程调用，在 DFS 的 Namespace 中创建一个文件条目（Entry），此时该条目没有任何的 Block，NameNode 会返回该数据每个块需要拷贝的 DataNode 地址信息；

3. 通过 FSDataOutputStream 对象，开始向 DataNode 写入数据，数据首先被写入 FSDataOutputStream 对象内部的数据队列中，数据队列由 DataStreamer 使用，它通过选择合适的 DataNode 列表来存储副本，从而要求 NameNode 分配新的 block；
4. DataStreamer 将数据包以流式传输的方式传输到分配的第一个 DataNode 中，该数据流将数据包存储到第一个 DataNode 中并将其转发到第二个 DataNode 中，接着第二个 DataNode 节点会将数据包转发到第三个 DataNode 节点；
5. DataNode 确认数据传输完成，最后由第一个 DataNode 通知 client 数据写入成功；
6. 完成向文件写入数据，Client 在文件输出流（FSDataOutputStream）对象上调用 `close` 方法，完成文件写入；
7. 调用 DistributedFileSystem 对象的 complete 方法，通知 NameNode 文件写入成功，NameNode 会将相关结果记录到 editlog 中。

文件读取过程

相对于文件写入，文件的读取就简单一些，流程如下图所示：



HDFS 文件读取过程

其具体过程总结如下（简单总结一下）：

1. Client 通过 DistributedFileSystem 对象与集群的 NameNode 进行一次 RPC 远程调用，获取文件 block 位置信息；
2. NameNode 返回存储的每个块的 DataNode 列表；
3. Client 将连接到列表中最近的 DataNode；
4. Client 开始从 DataNode 并行读取数据；

5. 一旦 Client 获得了所有必须的 block，它就会将这些 block 组合起来形成一个文件。

在处理 Client 的读取请求时，HDFS 会利用机架感知选举最接近 Client 位置的副本，这将会减少读取延迟和带宽消耗。

HDFS 1.0 的问题

在前面的介绍中，关于 HDFS1.0 的架构，首先都会看到 NameNode 的单点问题，这个在生产环境中是非常要命的问题，早期的 HDFS 由于规模较小，有些问题就被隐藏了，但自从进入了移动互联网时代，很多公司都开始进入了 PB 级的大数据时代，HDFS 1.0的设计缺陷已经无法满足生产的需求，最致命的问题有以下两点：

- NameNode 的单点问题，如果 NameNode 挂掉了，数据读写都会受到影响，HDFS 整体将变得不可用，这在生产环境中是不可接受的；
- 水平扩展问题，随着集群规模的扩大，1.0 时集群规模达到3000时，会导致整个集群管理的文件数目达到上限（因为 NameNode 要管理整个集群 block 元信息、数据目录信息等）。

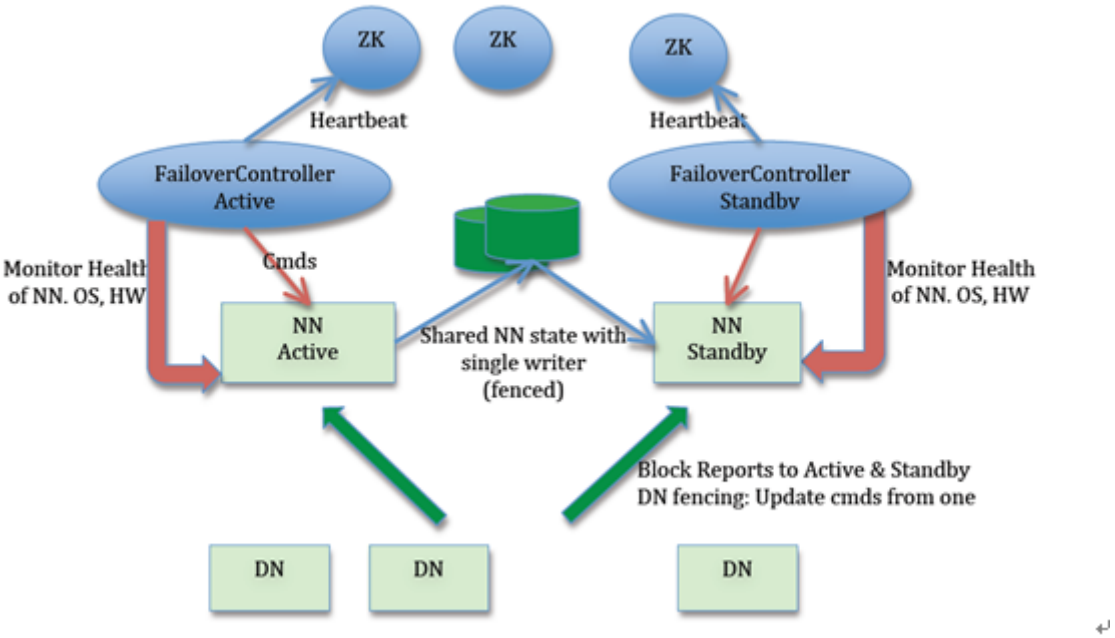
为了解决上面的两个问题，Hadoop2.0 提供一套统一的解决方案：

1. HA（High Availability 高可用方案）：这个是为了解决 NameNode 单点问题；
2. NameNode Federation：是用来解决 HDFS 集群的线性扩展能力。

HDFS 2.0 的 HA 实现

关于 HDFS 高可用方案，非常推荐这篇文章：Hadoop NameNode 高可用 (High Availability) 实现解析，IBM 博客的质量确实很高，这部分我这里也是主要根据这篇文章做一个总结，这里会从问题的原因、如何解决的角度去总结，并不会深入源码的实现细节，想有更深入了解还是推荐上面文章。

这里先看下 HDFS 高可用解决方案的架构设计，如下图（下图来自上面的文章）所示：



HDFS HA 架构实现

这里与前面 1.0 的架构已经有很大变化，简单介绍一下上面的组件：

1. Active NameNode 和 Standby NameNode：两台 NameNode 形成互备，一台处于 Active 状态，为主 NameNode，另外一台处于 Standby 状态，为备 NameNode，只有主 NameNode 才能对外提供读写服务；
2. ZKFailoverController（主备切换控制器，FC）：ZKFailoverController 作为独立的进程运行，对 NameNode 的主备切换进行总体控制。ZKFailoverController 能及时检测到 NameNode 的健康状况，在主 NameNode 故障时借助 Zookeeper 实现自动的主备选举和切换（当然 NameNode 目前也支持不依赖于 Zookeeper 的手动主备切换）；
3. Zookeeper 集群：为主备切换控制器提供主备选举支持；
4. 共享存储系统：共享存储系统是实现 NameNode 的高可用最为关键的部分，共享存储系统保存了 NameNode 在运行过程中所产生的 HDFS 的元数据。主 NameNode 和备 NameNode 通过共享存储系统实现元数据同步。在进行主备切换的时候，新的主 NameNode 在确认元数据完全同步之后才能继续对外提供服务。
5. DataNode 节点：因为主 NameNode 和备 NameNode 需要共享 HDFS 的数据块和 DataNode 之间的映射关系，为了使故障切换能够快速进行，DataNode 会同时向主 NameNode 和备 NameNode 上报数据块的位置信息。

FailoverController

FC 最初的目的是为了实现 SNN 和 ANN 之间故障自动切换，FC 是独立与 NN 之外的故障切换控制器，ZKFC 作为 NameNode 机器上一个独立的进程启动 ，它启动的时候会创建 HealthMonitor 和 ActiveStandbyElector 这两个主要的内部组件，其中：

1. HealthMonitor：主要负责检测 NameNode 的健康状态，如果检测到 NameNode 的状态发生变化，会回调 ZKFailoverController 的相应方法进行自动的主备选举；
2. ActiveStandbyElector：主要负责完成自动的主备选举，内部封装了 Zookeeper 的处理逻辑，一旦 Zookeeper 主备选举完成，会回调 ZKFailoverController 的相应方法来进行 NameNode 的主备状态切换。

自动触发主备选举

NameNode 在选举成功后，会在 zk 上创建了一个 `/hadoop-ha/${dfs.nameservices}/ActiveStandbyElectorLock` 节点，而没有选举成功的备 NameNode 会监控这个节点，通过 Watcher 来监听这个节点的状态变化事件，ZKFC 的 ActiveStandbyElector 主要关注这个节点的 NodeDeleted 事件（这部分实现跟 Kafka 中 Controller 的选举一样）。

如果 Active NameNode 对应的 HealthMonitor 检测到 NameNode 的状态异常时，ZKFailoverController 会主动删除当前在 Zookeeper 上建立的临时节点 /hadoop-ha/${dfs.nameservices}/ActiveStandbyElectorLock，这样处于 Standby 状态的 NameNode 的 ActiveStandbyElector 注册的监听器就会收到这个节点的 NodeDeleted 事件。收到这个事件之后，会马上再次进入到创建 /hadoop-ha/${dfs.nameservices}/ActiveStandbyElectorLock 节点的流程，如果创建成功，这个本来处于 Standby 状态的 NameNode 就选举为主 NameNode 并随后开始切换为 Active 状态。

当然，如果是 Active 状态的 NameNode 所在的机器整个宕掉的话，那么根据 Zookeeper 的临时节点特性，/hadoop-ha/${dfs.nameservices}/ActiveStandbyElectorLock 节点会自动被删除，从而也会自动进行一次主备切换。

HDFS 脑裂问题

在实际中，NameNode 可能会出现这种情况，NameNode 在垃圾回收（GC）时，可能会在长时间内整个系统无响应，因此，也就无法向 zk 写入心跳信息，这样的话可能会导致临时节点掉线，备 NameNode 会切换到 Active 状态，这种情况，可能会导致整个集群会有同时有两个 NameNode，这就是脑裂问题。

脑裂问题的解决方案是隔离（Fencing），主要是在以下三处采用隔离措施：

1. 第三方共享存储：任一时刻，只有一个 NN 可以写入；
2. DataNode：需要保证只有一个 NN 发出与管理数据副本有关的删除命令；
3. Client：需要保证同一时刻只有一个 NN 能够对 Client 的请求发出正确的响应。

关于这个问题目前解决方案的实现如下：

1. ActiveStandbyElector 为了实现 fencing，会在成功创建 Zookeeper 节点 hadoop-ha/${dfs.nameservices}/ActiveStandbyElectorLock 从而成为 Active NameNode 之后，创建另外一个路径为 /hadoop-ha/${dfs.nameservices}/ActiveBreadCrumb 的持久节点，这个节点里面保存了这个 Active NameNode 的地址信息；
2. Active NameNode 的 ActiveStandbyElector 在正常的状态下关闭 Zookeeper Session 的时候，会一起删除这个持久节点；
3. 但如果 ActiveStandbyElector 在异常的状态下 Zookeeper Session 关闭（比如前述的 Zookeeper 假死），那么由于 /hadoop-ha/${dfs.nameservices}/ActiveBreadCrumb 是持久节点，会一直保留下来，后面当另一个 NameNode 选主成功之后，会注意到上一个 Active NameNode 遗留下来的这个节点，从而会回调 ZKFailoverController 的方法对旧的 Active NameNode 进行 fencing。

在进行 fencing 的时候，会执行以下的操作：

1. 首先尝试调用这个旧 Active NameNode 的 HAServiceProtocol RPC 接口的 `transitionToStandby` 方法，看能不能把它转换为 Standby 状态；
2. 如果 `transitionToStandby` 方法调用失败，那么就执行 Hadoop 配置文件之中预定义的隔离措施。

Hadoop 目前主要提供两种隔离措施，通常会选择第一种：

1. sshfence：通过 SSH 登录到目标机器上，执行命令 fuser 将对应的进程杀死；
2. shellfence：执行一个用户自定义的 shell 脚本来将对应的进程隔离。

只有在成功地执行完成 fencing 之后，选主成功的 ActiveStandbyElector 才会回调 ZKFailoverController 的 `becomeActive` 方法将对应的 NameNode 转换为 Active 状态，开始对外提供服务。

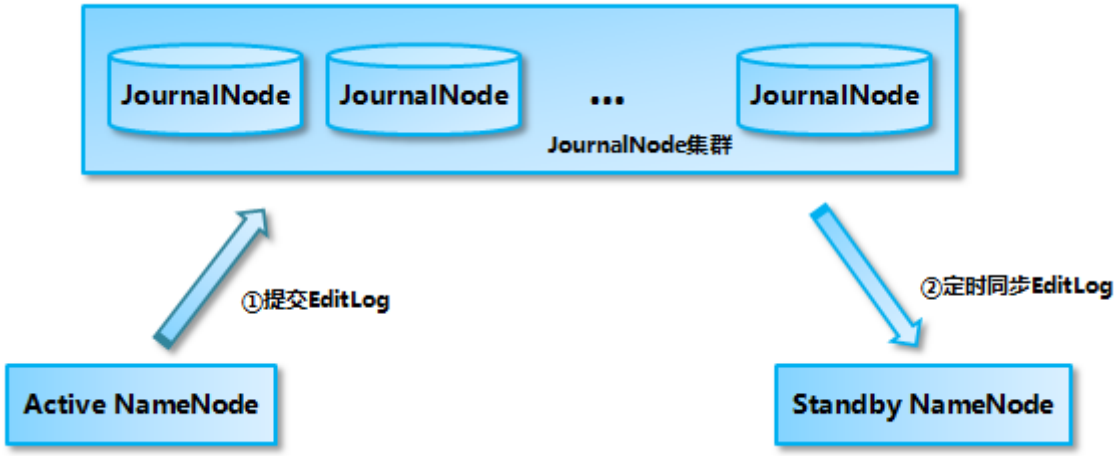
NameNode 选举的实现机制与 Kafka 的 Controller 类似，那么 Kafka 是如何避免脑裂问题的呢?

1. Controller 给 Broker 发送的请求中，都会携带 controller epoch 信息，如果 broker 发现当前请求的 epoch 小于缓存中的值，那么就证明这是来自旧 Controller 的请求，就会决绝这个请求，正常情况下是没什么问题的；
2. 但是异常情况下呢？如果 Broker 先收到异常 Controller 的请求进行处理呢？现在看 Kafka 在这一部分并没有适合的方案；
3. 正常情况下，Kafka 新的 Controller 选举出来之后，Controller 会向全局所有 broker 发送一个 metadata 请求，这样全局所有 Broker 都可以知道当前最新的 controller epoch，但是并不能保证可以完全避免上面这个问题，还是有出现这个问题的几率的，只不过非常小，而且即使出现了由于 Kafka 的高可靠架构，影响也非常有限，至少从目前看，这个问题并不是严重的问题。

第三方存储（共享存储）

上述 HA 方案还有一个明显缺点，那就是第三方存储节点有可能失效，之前有很多共享存储的实现方案，目前社区已经把由 Clouderea 公司实现的基于 QJM 的方案合并到 HDFS 的 trunk 之中并且作为默认的共享存储实现，本部分只针对基于 QJM 的共享存储方案的内部实现原理进行分析。

QJM（Quorum Journal Manager）本质上是利用 Paxos 协议来实现的，QJM 在 2F+1 个 JournalNode 上存储 NN 的 editlog，每次写入操作都通过 Paxos 保证写入的一致性，它最多可以允许有 F 个 JournalNode 节点同时故障，其实现如下（图片来自：Hadoop NameNode 高可用 (High Availability) 实现解析 ）：



基于 QJM 的共享存储的数据同步机制

Active NameNode 首先把 EditLog 提交到 JournalNode 集群，然后 Standby NameNode 再从 JournalNode 集群定时同步 EditLog。

还有一点需要注意的是，在 2.0 中不再有 SNN 这个角色了，NameNode 在启动后，会先加载 FSImage 文件和共享目录上的 EditLog Segment 文件，之后 NameNode 会启动 EditLogTailer 线程和 StandbyCheckpointer 线程，正式进入 Standby 模式，其中：

1. EditLogTailer 线程的作用是定时从 JournalNode 集群上同步 EditLog；
2. StandbyCheckpointer 线程的作用其实是为了替代 Hadoop 1.x 版本之中的 Secondary NameNode 的功能，StandbyCheckpointer 线程会在 Standby NameNode 节点上定期进行 Checkpoint，将 Checkpoint 之后的 FSImage 文件上传到 Active NameNode 节点。

HDFS 2.0 Federation 实现

在 1.0 中，HDFS 的架构设计有以下缺点：

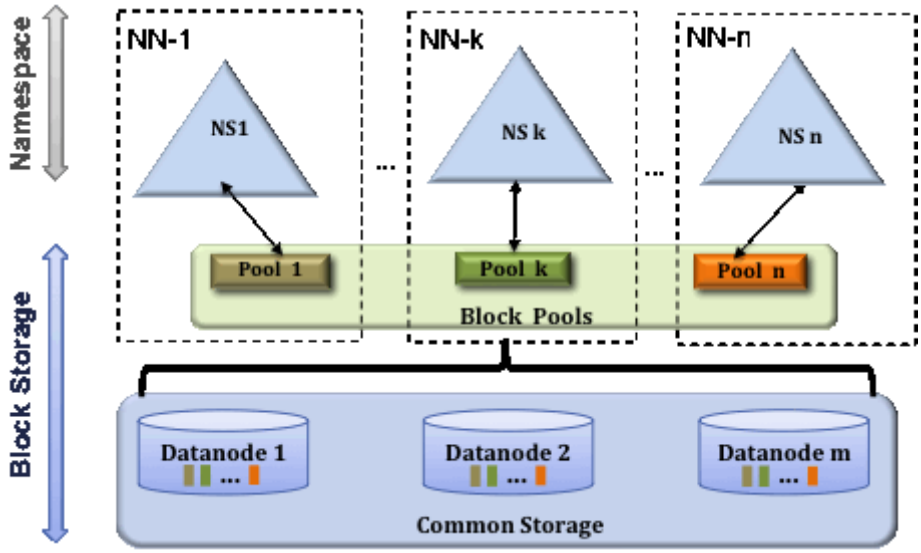
1. namespace 扩展性差：在单一的 NN 情况下，因为所有 namespace 数据都需要加载到 存，所以物理机内存的大小限制了整个 HDFS 能够容纳文件的最大个数（namespace 指的是 DFS 中树形目录和文件结构以及文件对应的 block 信息）；

2. 性能可扩展性差：由于所有请求都需要经过 NN，单一 NN 导致所有请求都由一台机器进行处理，很容易达到单台机器的吞吐；
3. 隔离性差：多租户的情况下，单一 NN 的架构无法在租户间进行隔离，会造成不可避免的相互影响。

而 Federation 的设计就是为了解决这些问题，采用 Federation 的最主要原因是设计实现简单，而且还能解决问题。

Federation 架构

Federation 的架构设计如下图所示（图片来自 HDFS Federation）：



HDFS Federation 架构实现

Federation 的核心设计思想

Federation 的核心思想是将一个大的 namespace 划分多个子 namespace，并且每个 namespace 分别由单独的 NameNode 负责，这些 NameNode 之间互相独立，不会影响，不需要做任何协调工作（其实跟拆集群有一些相似），集群的所有 DataNode 会被多个 NameNode 共享。

其中，每个子 namespace 和 DataNode 之间会由数据块管理层作为中介建立映射关系，数据块管理层由若干数据块池（Pool）构成，每个数据块只会唯一属于某个固定的数据块池，而一个子 namespace 可以对应多个数据块池。每个 DataNode 需要向集群中所有的 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 poo 流，如果一个 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/namespace 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 可以作为一个基本单元进行升级。

到这里，基本对 HDFS 这部分总结完了，虽然文章的内容基本都来自下面的参考资料，但是自己在总结的过程中，也对 HDFS 的基本架构有一定的了解，后续结合公司 HDFS 团队的 CaseStudy 深入学习这部分的内容，工作中，也慢慢感觉到分布式系统，很多的设计实现与问题解决方案都很类似，只不过因为面对业务场景的不同而采用了不同的实现。

参考：

- HDFS Architecture;
- HDFS 写文件过程分析;
- HDFS Router-based Federation;
- HDFS High Availability Using the Quorum Journal Manager;
- HDFS Commands Guide;
- Hadoop NameNode 高可用 (High Availability) 实现解析
- HDFS Federation;
- HDFS Federation设计动机与基本原理;
- 《大数据日知录：架构与算法》;
- HDFS NameNode重启优化;
- HDFS Federation在美团点评的应用与改进。

hadoop

JVM 之 ParNew 和 CMS 日志分析 Kafka Controller Redesign 方案