Hadoop 高可用方案（HA）

问题描述：

再当前我们配置的 hadoop 中存在 namenode 单点故障与 rm 单点故障， 如何来解决这个问题， 一旦挂机产生非常大的影响。

解决方案：

1 BackupNode 方案：

08 年时开源社区已经开始着手解决 Namenode 单点问题，随之出来的第一个方案是 BackupNode 方案。基于 0.20 版，并合并进入 0.21 版；参见 Apache JIRA HADOOP-4539 [1]

该方案思路为：将 NameNode 产生的 editLog（对文件系统元数据的修改）通过网络复写到 BackupNode 的内存中，再由 BackupNode 对接收的 editLog 重放操作，从而保持 BackupNode 与 NameNode 的 image 数据结构一致。

该方案的问题在于：

切换时间长；因为复写的 editLog 中不包含 block 信息，因而 BackupNode 内存中 blockMap 为空，在切换后需要等待 DataNode 重连并重传所有的 block 信息；需要时间在分钟级；

没有提供自动 failover 机制；BackupNode 是对 NameNode 的元数据进行实时备份，可以用来提供只读服务，却不能在 NameNode 失败后接替其工作；可以人工介入修改 ip 从而恢复服务；

注：

如果要减少切换时间，需要再增加逻辑以实现对 block 信息的转发，增加代码复杂性同时会遇到缓存、流控等问题，Facebook 的 AvatarNode 方案因这些原因考虑而放弃了 block 信息的转发；

2. AvatarNode 方案：

Facebook 内部使用的热备(host standby)HA 方案，在 10 年贡献到开源社区，参见 Apache JIRA HDFS-976[3]；当时 Facebook Hadoop 集群规模为 1200 节点 12PB

AvatarNode 方案基于 apache hadoop 0.20 版，其尽量不修改原有 NameNode 代码，在现有代码之上通过封装已有代码和通过成熟的技术实现高可用。

AvatarNode 方案的思路为：

使用一个共享的 NFS 服务来保存 NameNode（Primary Avatar）的 editLog，Standby Avatar 从 editLog 尾部读取最新的修改，重放进自己的内存数据结构；

AvatarDatanode 同时向 Primary 和 Standby 汇报 block 信息（包括 blockReport 和 blockReceived）；因 block 信息的转发需要解决缓存、流控等问题，会极大增加代码复杂度，因而放弃转发的实现；

客户端通过虚拟 ip 访问 NameNode 服务，当 Standby Avatar 与 Primary Avatar 进行切换时，通过配置该 ip 实现对客户端访问的透明；

该方案的优点为：切换时间很快，在秒级范围实现切换（<1 分钟）；

该方案的缺点为：

不实现自动 failover，切换由 OP 人工介入；作者对该点的解释为：Hadoop 集群的停机主要是由升级需要引起的，因而升级时由 OP 手工进行 failover 操作，从而也不需要担心脑裂问题；AvatarNode 方案不会应对意外故障导致的集群停机；

依赖 NFS 服务，

NFS 服务读写性能差；AvatarNode 方案作者对该点解释为：Facebook 使用了一台 NetApp 的 NAS 服务器保存 AvatarNode

的元数据，性能很高。

HA 风险转嫁给 NFS 服务器；NFS 服务器停机带来的后果未知；

大多 Linux 的 NFS 客户端实现有问题，如果不进行正确的配置，在某些意外情况下（NFS 服务器停机）NameNode 可能被卡住且无法恢复；

3 HDFS HA Branch 方案系列

11 年 HDFS HA 成为一条独立的分支进行讨论研究并最终合并进入 0.23 版主干，参见 Apache JIRA HDFS-1623 [5]；

该方案考虑了多种可选的解决途径，例如：使用共享 NFS 存储或 editLog 复写、使用 LinuxHA ResourceManager 或 Zookeeper FailoverController；甚至之前的 BackupNode 方案也被包含其中；

最被看好的是 BookKeeper[6]复写方案＋Zookeeper FailoverController；

BookKeeper 复写方案依赖 BookKeeper 团队实现的分布式日志服务来保存 NameNode 的 editLog；

Zookeeper FailoverController 可以实现对 Active/Standby NameNode 的状态监控和主从选举；

该方案对 block 信息的处理与 AvatarNode 类似：DataNode 都明确知道系统中所有 NameNode 的存在，向它们分别汇报 block 信息；

优点：社区主干代码，支持较好；可选择性丰富；

缺点：根据不同子方案需要单独评估缺点；引入模块较多，分析评估升级代价较大；

注：

CDH4 方案来源于 HA Branch 方案系列，在元数据存储上采用了共享 NFS 存储的方式，缺点与 AvatarNode 相同；

5. Quorum-based 方案（我们采用的方案）：

因为主干 HA 方案中对 NFS 或 BookKeeper 的依赖问题，12 年时开源社区又出现了一套基于 Quorum 的 HA 方案，

该方案的思想是：

- 集群中启动三个 JournalNode；
- 集群中每个 NameNode 上同时运行一个 QuorumJournalManager 组件；通过 Hadoop IPC 向 JournalNode 写入 editLog；
- 当 QJM 写 editLog 前，首先要保证没有其它 QJM 在写 editLog，从而保证当发生脑裂时，editLog 的写入依然是安全的；这一点的实现是通过 QJM 成为写者时（其 NameNode 成为 Active 节点时）分配唯一的 epoch 号，并广播给所有 JournalNode，JournalNode 在执行写 editLog 操作前对請求者 QJM 的 epoch 号进行检查；epoch 号的申请也是经过 JN 和 QJM 的仲裁同意的；
- 当 QJM 写 editLog 前，同样需要保证之前 editLog 在所有 JN 上一致；例如如果一个 QJM 写过程中发生失败，则几个 JN 的 editLog 的尾部很可能不同，新的 QJM 成为写者时，需要对这些不一致的 editLog 进行同步，仲裁后保证一致；
- 当 QJM 写 editLog 时，只要 JN 中的大部分（2/3）成功，就算成功；可以继续执行后续操作；
- 仅关心了 editLog 的存储问题，其它技术细节默认沿用主干 HA 方案；

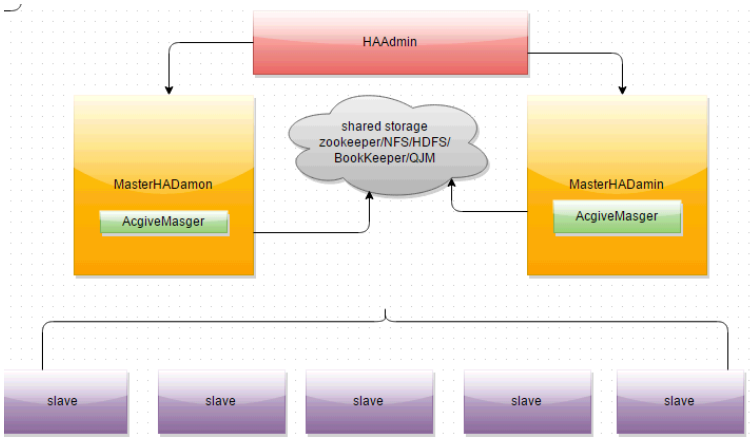
  优点：对第三方模块、特殊硬件无依赖；

QJM 解决 方案

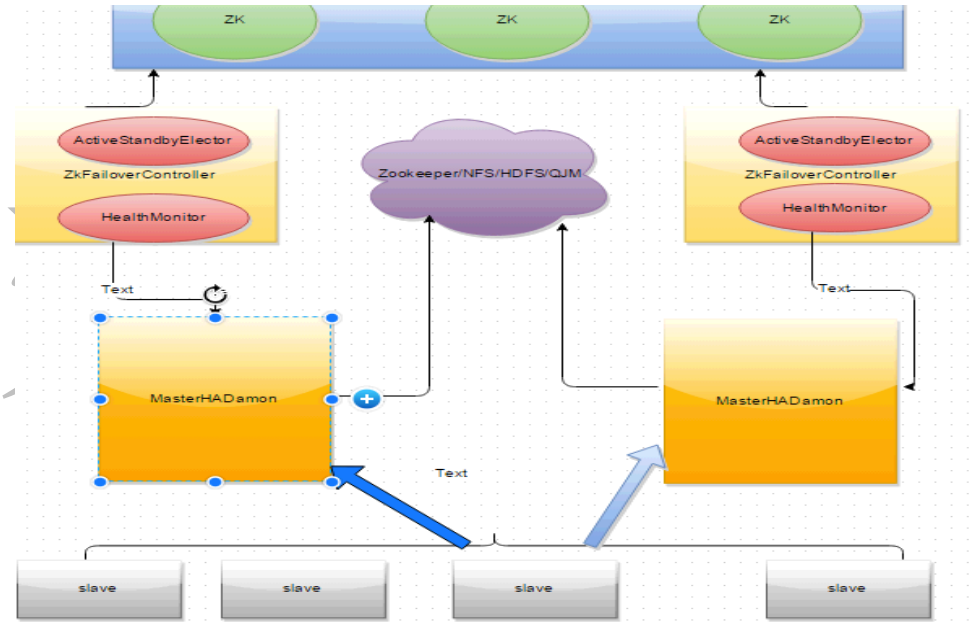
总体上说，Hadoop 中的 HDFS、MapReduce 和 YARN 的单点故障解决方案架构是完全一致的，分为手动模式和自动模式，其中

手动模式是指由管理员通过命令进行主备切换，这通常在服务升级时有用，自动模式可降低运维成本，但存在潜在危险。这两种模式下的架构如下。

手动模式：



自动模式：



在 Hadoop HA 中，主要由以下几个组件构成：

（1）MasterHADaemon：与 Master 服务运行在同一个进程中，可接收外部 RPC 命令，以控制 Master 服务的启动和停止；

（2）SharedStorage：共享存储系统，active master 将信息写入共享存储系统，而 standby master 则读取该信息以保持与 active master 的同步，从而减少切换时间。常用的共享存储系统有 zookeeper（被 YARN HA 采用）、NFS（被 HDFS HA 采用）、HDFS（被 MapReduce HA 采用）和类 bookeeper 系统（被 HDFS HA 采用）。

（3）ZKFailoverController：基于 Zookeeper 实现的切换控制器，主要由两个核心组件构成：ActiveStandbyElector 和 HealthMonitor，其中，ActiveStandbyElector 负责与 zookeeper 集群交互，通过尝试获取全局锁，以判断所管理的 master 进入 active 还是 standby 状态；HealthMonitor 负责监控各个活动 master 的状态，以根据它们状态进行状态切换。。

（4）Zookeeper 集群：核心功能通过维护一把全局锁控制整个集群有且仅有一个 active master。当然，如果 ShardStorge 采用了 zookeeper，则还会记录一些其他状态和运行时信息。

尤其需要注意的是，解决 HA 问题需考虑以下几个问题：

（1）脑裂（brain-split）：脑裂是指在主备切换时，由于切换不彻底或其他原因，导致客户端和 Slave 误以为出现两个 active master，最终使得整个集群处于混乱状态。解决脑裂问题，通常采用隔离(Fencing)机制，包括三个方面：

共享存储 fencing：确保只有一个 Master 往共享存储中写数据。

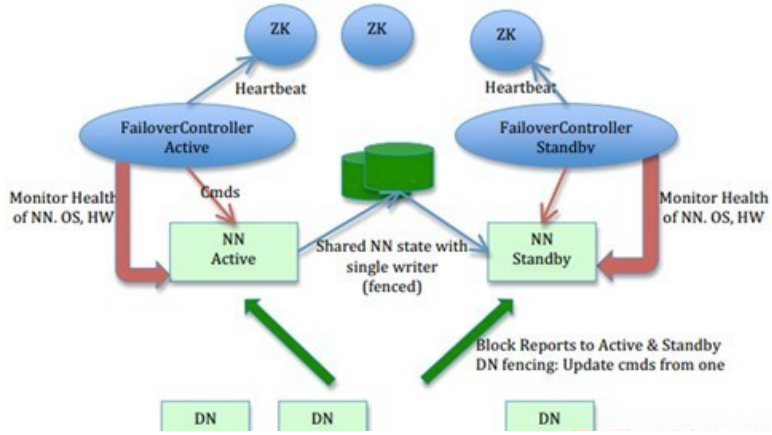
客户端 fencing：确保只有一个 Master 可以响应客户端的请求。

Slave fencing：确保只有一个 Master 可以向 Slave 下发命令。

Hadoop 公共库中对外提供了两种 fenching 实现，分别是 sshfence 和 shellfence（缺省实现），其中 sshfence 是指通过 ssh 登陆目标 Master 节点上，使用命令 fuser 将进程杀死（通过 tcp 端口号定位进程 pid，该方法比 jps 命令更准确），shellfence 是指执行一个用户事先定义的 shell 命令（脚本）完成隔离。

（2）切换对外透明：为了保证整个切换是对外透明的，Hadoop 应保证所有客户端和 Slave 能自动重定向到新的 active master 上，这通常是通过若干次尝试连接旧 master 不成功后，再重新尝试链接新 master 完成的，整个过程有一定延迟。在新版本的 Hadoop RPC 中，用户可自行设置 RPC 客户端尝试机制、尝试次数和尝试超时时间等参数。

目前 HDFS2 中提供了两种 HA 方案，一种是基于 NFS 共享存储的方案，一种基于 Paxos 算法的方案 Quorum Journal Manager（QJM），它的基本原理就是用 2N+1 台 JournalNode 存储 EditLog，每次写数据操作有大多数（>=N+1）返回成功时即认为该次写成功，数据不会丢失了。目前社区正尝试使用 Bookeeper 作为共享存储系统(还没有稳定的方案)，

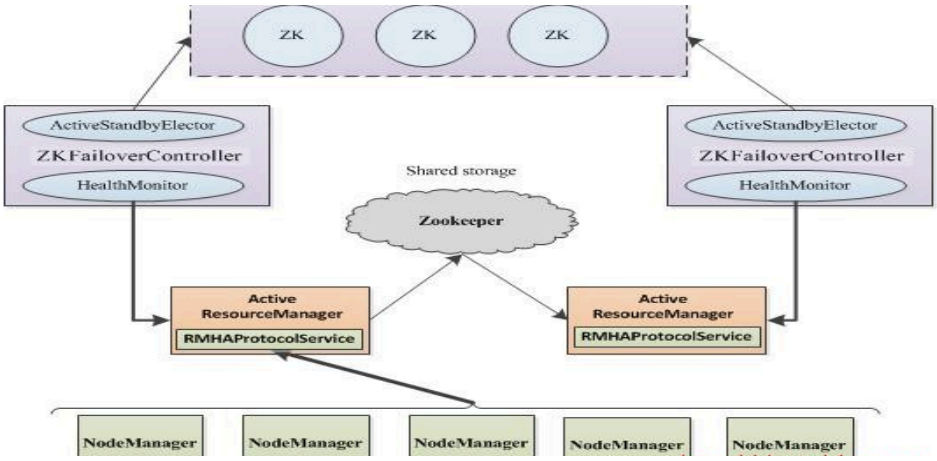


YARN 的单点故障指的是 ResourceManager 单点问题，ResourceManager 负责整个系统的资源管理和调度，内部维护了各个应用程序的 ApplictionMaster 信息，NodeManager 信息，资源使用信息等。考虑到这些信息绝大多数可以动态重构，因此解决 YARN 单点故障要比 HDFS 单点容易很多。与 HDFS 类似，YARN 的单点故障仍采用主备切换的方式完成，不同的是，备节点不会同步主节点的信息，而是在切换之后，才从共享存储系统读取所需信息。之所以这样，是因为 YARN ResourceManager 内部保存的信息非常少，大部分可以重构，且这些信息是动态变化的，很快会变旧。

目前单点故障解决方案仍是粗粒度的，它借助 Zookeeper 完成主备节点信息共享。它仅在 Zookeeper 上保存 Application ID 和 ApplicationAttempt ID，以便故障恢复后重新创建这些 Application，其他信息则动态重构或者丢弃，比如 NodeManager 信息（包括可用资源，健康状态等信息），需由 NodeManager 重新通过 RPC 汇报，而资源使用信息（每个节点资源使用情况，每个任务资源获取情况等）则全部重置，也就是说，故障恢复或者 ResourceManager 主备切换后，整个集群跟重启过一样，只不过是之前正在运行的应用程序不需要重新提交，但已经分配的资源信息则全部丢失。这意味着，NodeManager 重新跟新的 ResourceManager 连接后，ResourceManager 发送的第一个指令是让 NodeManager 重启，而 NodeManager 会杀死所有正在运行的 Container。

同 HDFS 和 MapReduce 一样，YARN HA 可存在两种模式，分别是手动模式和自动模式，在手动模式下，所有 ResourceManager 启动后将进入 standby 状态，需要由管理员通过命令将一个切换为 active 状态，YARN 集群才可对外提供服

务；而自动模式则基于 zookeeper 实现，基本的实现思想是，所有 ResourceManager 启动后，将创建 Zookeeper 下的同一个目录，谁创建成功，则谁进入 active 状态，其他的自动转入 standby 状态。YARN 的自动模式架构图如下所示，各个组件的作用已在“Hadoop 2.0 单点故障解决方案”一文中进行了详细的介绍，整个架构与 CDH4 中 MapReduce JobTracker 的 HA 解决思路一致，与该架构不同的是，ZKFailoverController（ZKFC）属于 RMHAProtocolService 中的一个线程，而 RMHAProtocolService 本身则变成 ResourceManager 内部的一个服务，这意味着你无需启动一个单独的 ZKFC（下图为了方便说明，将 ZKFC 独立画出来）。



此外，需要补充说明的是，YARN ResourceManager 只负责 ApplicationMaster 的状态维护和容错，ApplicationMaster 内部管理和调度的任务，比如 MapTask 和 ReduceTask，则需要由 AppicationMaster 自己容错，这不属于 YARN 这个系统管理的范畴。比如 MapReduce 的 ApplicationMaster—MRAppMaster 会在 HDFS 上记录 Task 运行日志，这样，当它运行失败重新被调度到另外一个节点运行时，会重新从 HDFS 上读取日志，恢复已经运行完成的 Task，而只需为那些未运行完成的任务申请资源和二次调度（注意，之前正在运行的 Task 会被杀死，重新执行）。

总体上讲，HA 解决的难度取决于 Master 自身记录信息的多少和信息可重构性，如果记录的信息非常庞大且不可动态重构，比如 NameNode，则需要一个可靠性与性能均很高的共享存储系统，而如果 Master 保存有很多信息，但绝大多数可通过 Slave 动态重构，则 HA 解决方法则容易得多，典型代表是 MapReduce 和 YARN。从另外一个角度看，由于计算框架对信息丢失不是非常敏感，比如一个已经完成的任务信息丢失，只需重算即可获取，使得计算框架的 HA 设计难度远低于存储类系统