在云端 云计算最佳实践

高可用性的HDFS

王磊 文艾 著

Hadoop分布式文件系统深度实践

清华大学出版社
北京

内 容 简 介

本书专注于 Hadoop 分布式文件系统（HDFS）的主流 HA 解决方案，内容包括：HDFS 元数据解析、Hadoop 元数据备份方案、Hadoop Backup Node 方案、AvatarNode 解决方案以及最新的 HA 解决方案 Cloudrea HA Name Node 等。其中有关 Backup Node 方案及 AvatarNode 方案的内容是本书重点，尤其是对 AvatarNode 方案从运行机制到异常处理方案的步骤进行了详尽介绍，同时还总结了各种异常情况下 AvatarNode 的各种处理方案。

本书从代码入手并结合情景分析、案例解说对 HDFS 的元数据以及主流的 HDFS HA 解决方案的运行机制进行了深入剖析，力求使读者在解决问题时做到心中有数，不仅知其然还知其所以然。

本书可操作性强，所有案例都经过验证并附有详细的步骤说明和视频教程，无论是对云计算的初学者，还是想进一步深入学习云计算技术的研发人员，或是云计算的研究人员都有很好的参考价值。

本书光盘包含本书部分操作的视频教程以及所有源代码、脚本等开发文件。

本书读者主要为云计算相关领域的研发人员、云计算系统管理维护人员，也适合作为高校研究生和高年级本科生的专业课辅助教材。

图书在版编目(CIP)数据

高可用性的 HDFS—Hadoop 分布式文件系统深度实践/文艾，王磊编著.—北京：清华大学出版社，2012.5

ISBN 978-7-302-28258-7

Ⅰ.①高… Ⅱ. ①文… ②王… Ⅲ. ①分布式文件系统—研究 Ⅳ. ①TP316

中国版本图书馆 CIP 数据核字（2012）第 040268 号

责任编辑：栾大成
装帧设计：
责任校对：徐俊伟
责任印制：
出版发行：清华大学出版社

网　　址：http://www.tup.com.cn，　http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：188mm×230mm　　**印　张**：24　　**插　页**：1　　**字　数**：433 千字
附 DVD 1 张
版　　次：2012 年 5 月第 1 版　　**印　次**：2012 年 5 月第 1 次印刷
印　　数：1~5000
定　　价：59.00 元

产品编号：044551-01

前 言

Hadoop 是目前主流的开源云计算系统，它实现了一个高可扩展的分布式文件系统——HDFS（Hadoop Distributed File System），HDFS 作为 Hadoop 底层基础设施，为云计算提供高可靠、高性能的存储服务。HDFS 在很大程度上借鉴了 Google GFS 文件系统的设计思想，具有高度容错、支持大数据集等诸多特性。这些特性曾让我们欢欣鼓舞，一度以为 HDFS 是一个可以解决数据密集型应用的海量数据存储难题的完美方案。随着研究的深入，我们在不断叹服 HDFS 设计构思巧妙的同时，也深深地认识到仅有以上的特性还不足以构建一个实用的分布式文件系统，还需要一些其他特性进行支撑，高可用性则是其中最为关键的一点。

高可用性是指系统正常服务时间所占的百分比，它是衡量系统对外正常服务能力的重要指标。对于 HDFS 来说，每一份数据可以有多个副本，因此文件数据的可靠性可以由副本来解决。然而，对于元数据管理来说，只有一个节点 NameNode，它的好坏直接决定了 HDFS 能否正常服务，因此 NameNode 的高可用性决定了整个 HDFS 系统的高可用性。不同应用对 HDFS 高可用性有不同要求，目前 Hadoop 自身包括其他一些开源组织提供了一些相应的高可用性机制以满足不同的需要，如 Backup Node 方案以及 AvatarNode 方案等。

根据我们的实践经验，在使用以上方案解决实际问题时，需要具备以下几点基础：

- 首先是对 NameNode 元数据机制有较深的理解和把握；
- 其次要对各种解决方案的运行机制及使用方法有个全面掌握；
- 再次就是要有较强的实践操作经验。

然而，就现实情况而言，要在以上任何一点取得一点进展都需要付出相当大的努力，回顾我们的团队在接触 HDFS 的高可用性之初，由于资料和经验的匮乏，每

掌握一个知识点，都需经历资料查找、邮件列表搜索、邮件请教、代码查看、实验验证等多个环节，其间的付出可想而知，正是基于这点，我们也深切地感觉有必要将我们前期的经验和心得与大家分享，姑且也算做是我们团队对于开源软件事业的一点小小的回馈吧。

本书内容

本书一共 8 章，分为 4 个部分。

其中第 1 部分为第 1 章，主要介绍当前 HDFS 主流的 HA 方案以及相关概念，使读者能够有一个宏观上的认识，同时通过方案的比较，遴选出 3 种具有代表性的 HA 方案。

第 2 部分为第 2 章，围绕 HDFS HA 的重点关注对象元数据，对内存元数据结构、磁盘元数据文件、文件系统格式化场景以及元数据在 HA 中的应用场景进行了深入剖析。

第 3 部一共 5 章（3、4、5、6、7），主要介绍 3 种经典的 HA 解决方案：**Hadoop 元数据备份方案、BackupNode 方案**以及 **AvatarNode 方案**，从代码入手，分别从运行机制、使用方法等方面进行说明，每种方案都有详细的使用说明并配以视频，便于读者掌握。

第 4 部分为第 8 章，介绍目前最新 HA 解决方案 Cloudrea HA NameNode。

适合读者

如果您是一位 Hadoop 集群管理维护人员，请阅读本书，它将向您展示当前主流的 HDFS HA 解决方案，通过文字说明和视频展示这些方案的实现机制和操作细节，使您能够在最短的时间内消化和吸收这些技术，您可以根据自己的需要选择和部署实施最合适的 HA 方案。

如果您是一位 Hadoop 应用开发者，请阅读本书，您将会在此找到如何结合 HDFS 的 HA，编写出更为健壮的应用程序。

如果您是一位分布式文件系统研发人员，请阅读本书，它将向您深入剖析 HDFS 这

一最有影响力的开源云计算分布式存储系统的各种 HA 方案及其实现机制。

如果您是一位云计算技术的爱好者，请阅读本书，本书会从零开始，一步一步地带您掌握云计算相关技术，并加深概念的理解，为您日后深入接触云计算技术打下基础。

本书由文艾和王磊共同编著而成。文艾负责总体设计、内容把握以及写作组织，独立完成第 1、2、3、8 章，并与王磊共同完成第 4、5、6、7 以及实验的视频设计和制作。

感谢中国电子学会云计算专家委员会专家刘鹏教授的大力支持；感谢我的家人，你们是我奋斗前进的最大动力；

最后，希望大家从书中找到需要的东西。

时间紧，任务急，错误在所难免，敬请各位批评指正。请发送邮件到 hdfsha@126.com。

目 录

第 1 章　HDFS HA 及解决方案

HDFS[1]（Hadoop Distributed File System）即 Hadoop 分布式文件系统，它为 Hadoop 这个分布式计算框架提供高性能、高可靠、高可扩展的存储服务。

[1] http://hadoop.apache.org/hdfs/

1.1 HDFS 系统架构

HDFS 的系统架构如图 1.1 所示，它是一个典型的主/从架构，包括一个 NameNode 节点（主节点）和多个 DataNode 节点（从节点），并提供应用程序访问接口。**NameNode 是整个文件系统的管理节点，它负责文件系统名字空间（Namespace）的管理与维护，同时负责客户端文件操作的控制以及具体存储任务的管理与分配；DataNode 提供真实文件数据的存储服务**。

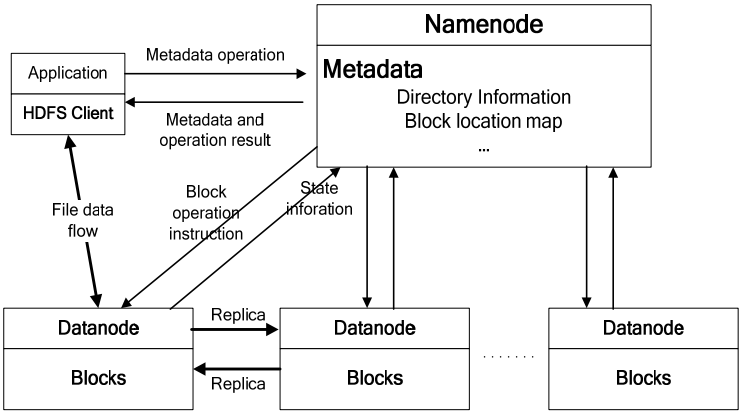

图 1.1 HDFS 系统架构

HDFS 有两种数据：文件数据和元数据。

1．文件数据

文件数据是指用户保存在 HDFS 上的文件的具体内容，HDFS 将用户保存的文件按照固定大小(用户可设置,通常是 64MB)进行分块(每一块简称为一个 Block)，保存在各个 DataNode 上，每一个块可能会有多个副本（Replica），具体个数可以由用户指定（通常是 3 个），相同块对应的副本通常保存在不同的 DataNode 上，通过副本机制，可以有效保证文件数据的可靠性。

2．元数据

所谓元数据（Metadata）是指数据的数据，HDFS 与传统的文件系统一样，提

供了一个分级的文件组织形式，维护这个文件系统所需的信息（除了文件的真实内容）就称之为 HDFS 的元数据。

元数据由 NameNode 进行维护和管理，NameNode 在启动时，会从磁盘加载元数据文件到内存，并且等待 Data Node 上报其他元数据信息，形成最终的元数据结构。由于 NameNode 是单节点，一旦 NameNode 无法正常服务，将导致整个 HDFS 无法正常服务。

1.2　HA 定义

HA 的英文全称是 High Availability，中文翻译为高可用性。什么是 HA？HA 与我们平时常说的高可靠性又有什么关系，下面我们一起来看下 HA 的定义。

HA 的定义为**系统对外正常提供服务时间的百分比**。具体来说，HDFS 的可靠性可用平均无故障时间（MTTF[2]）来度量，即 HDFS 正常服务的平均运行时间，HDFS 的可维护性用平均维修时间（MTTR[2]）来度量，即 HDFS 从不能正常服务到重新正常服务所需要的平均维修时间。因此 HDFS 的 HA 可精确定义为：

MTTF/(MTTF+MTTR) * 100%

由上面的定义我们可以很清楚地将 HA 与高可靠性区分开来，高可靠性更多的是对于系统自身而言，它是系统可靠程度的一个指标。而 HA 则更多地是从系统对外的角度来说的，除了包含系统正常工作的能力，它还强调系统中止服务后迅速恢复的能力：一个可靠性很高的系统，如果其中止服务后，修复时间很长，那么它的可用性也不会很高，而一个可靠性不是特别高的系统，如果发生中止服务后，可迅速恢复，那么其可用性也可能会很高。因此只有 HA 才能准确度量系统对外正常服务的能力。

HDFS HA 的应用场景有很多，我们可以从正常和异常两种情况来分析 HDFS 对外无法正常服务的情景：

[2] http://baike.baidu.com/view/849879.htm

- 首先是正常使用情况，最常见的应用场景就是 NameNode 节点软、硬件的升级与维护，由于 NameNode 只有一个，当 NameNode 节点软、硬件的升级与维护操作需要 NameNode 进行重启时，HDFS 将无法服务。
- 其次是异常情况，常见的场景有：用户的误操作导致 NameNode 系统崩溃或 HDFS 发生故障、或者是硬件故障等。在实际使用过程中，软硬件维护、软件故障、错误操作等因素是造成 HDFS 无法提供正常服务的主要原因，而大家普遍关注的硬件故障并不是主要原因。

雅虎的数据表明：在雅虎运行的 15 个集群中，三年时间内，只有 3 次 NameNode 的故障与硬件问题有关。

此外，由于 HDFS 处于 Hadoop 的底层，上层的其他分布式处理框架如 Mapreduce、HBase、Hive、Pig 等都依赖于 HDFS 提供的基础服务，因此 HDFS 的 HA 将对这些分布式处理框架的 HA 构成直接影响，并最终影响到最上层分布式应用的 HA。因此对于一个实用的系统来说，在大多数情况下都需要考虑 HDFS 的 HA 问题。

1.3 HDFS HA 原因分析及应对措施

根据定义，影响 HDFS HA 的因素从可靠性和可维护性两方面进行分析。

1.3.1 可靠性

我们知道，HDFS 由 NameNode 和 Data Node 两类节点组成，由于 NameNode 只有 1 个，且负责整个 HDFS 文件系统的管理和控制，因此当 NameNode 不能提供正常服务时，会直接导致 HDFS 不能对外正常服务，因此 NameNode 的可靠性是影响 HDFS 可靠性的重要因素。

由于 NameNode 只有一个，它的正常运行与否直接决定了 HDFS 能否正常服务，因此它也就成为了 HDFS 系统的一个单一故障点（single point of failure ——SPOF），DataNode 负责存储真实文件数据，每个文件可以指定副本个数，因此同一个文件可在多个 DataNode 上进行存储，当有 DataNode 发生故障时，客户端可

以访问其他 DataNode 上的副本，因此 DataNode 发生故障并不会影响 HDFS 对外正常服务。

1.3.2　可维护性

当 NameNode 不能正常服务时，通常需要重新启动 NameNode 来恢复服务，NameNode 启动时需要加载磁盘上的元数据文件：如果此时元数据没有损坏，那么直接启动 NameNode 就可以恢复 HDFS 对外正常服务；如果元数据损坏，将导致 NameNode 无法启动，无法再对外正常服务，也就是说平均维护时间是无限大。

因此元数据的可靠性决定了 HDFS 的可维护时间。当 DataNode 无法正常工作时，HDFS 会自动启动该 DataNode 上所有数据的复制任务，将丢失的数据重新分布到其他 DataNode 上，因此 DataNode 并不影响 HDFS 的可维护性。

综上所述，HDFS 的 HA 主要由 NameNode 的 HA 决定，NameNode 的可靠性主要取决于自身计算机硬件系统的可靠性、系统软件以及 HDFS 软件的可靠性；NameNode 的可维护性则取决于元数据的可靠性以及 NameNode 服务恢复时间。

1.4　现有 HDFS HA 解决方案

HDFS 的 HA 的解决方案，主要是从使用者的角度出发，提高元数据的可靠性，减少 NameNode 服务恢复时间。提高元数据的可靠性措施主要是对元数据进行备份，HDFS 自身就具有多种机制来确保元数据的可靠性。

减少 NameNode 服务恢复时间的措施有两种思路：

- 第 1 种基于 NameNode 重启恢复服务的方式，对 NameNode 自身的启动过程进行分析，优化加载过程，减少启动时间；
- 第 2 种则是启动一个 NameNode 的热备（Warm standby）节点，当主节点不能正常服务时，由热备节点进行接替，此时主备切换时间成为服务恢复时间。

从效率上分析，第 1 种思路尽管进行了优化，但 NameNode 的启动时间仍受文件系统规模的限制，第 2 种则突破了这种限制。

现有比较成熟的 HA 解决方案有：

- Hadoop 的元数据备份方案
- Hadoop 的 Secondary NameNode 方案[3]
- Hadoop 的 Checkpoint Node 方案[4]
- Hadoop 的 Backup Node 方案[5]
- DRDB 方案[6]
- Facebook 的 Avatarnode 方案[7]

下面将依次介绍。

1.4.1 Hadoop 的元数据备份方案

该方案利用 Hadoop 自身的 Failover 措施（通过配置 dfs.name.dir），NameNode 可以将元数据信息保存到多个目录。通常的做法，选择一个本地目录、一个远程目录（通过 NFS 进行共享），当 NameNode 发生故障时，可以启动备用机器的 NameNode，加载远程目录中的元数据信息，提供服务。

优点

- Hadoop 自带机制，成熟可靠，使用简单方便，无需开发，配置即可。
- 元数据有多个备份，可有效保证元数据的可靠性，并且内容保持最新状态。
- 元数据需要同步写入多个备份目录，效率低于单个 NameNode。

[3] http://hadoop.apache.org/common/docs/stable/hdfs_user_guide.html#Secondary+NameNode
[4] https://issues.apache.org/jira/browse/HADOOP-4539
[5] http://www.drbd.org/
[6] https://issues.apache.org/jira/browse/HDFS-976

缺点

- 该方案主要是解决元数据保存的可靠性问题，但没有做到热备，HDFS 恢复服务时，需要重新启动 NameNode，恢复时间与文件系统规模成正比。
- NFS 共享的可靠性问题，如果配置的多个目录中有任何一个目录的保存因为异常而阻塞，将会导致整个 HDFS 的操作阻塞，无法对外提供正常服务。

1.4.2　Hadoop 的 Secondary NameNode 方案

该方案启动一个 Secondary NameNode 节点，该节点定期从 NameNode 节点上下载元数据信息（元数据镜像 fsimage 和元数据库操作日志 edits），然后将 fsimage 和 edits 进行合并，生成新的 fsimage（该 fsimage 就是 Secondary NameNode 下载时刻的元数据的 Checkpoint），在本地保存，并将其推送到 NameNode，同时重置 NameNode 上的 edits。

优点

- Hadoop 自带机制，成熟可靠，使用简单方便，无需开发，配置即可。
- Secondaryary NameNode 定期做 Checkpoint，可保证各个 Checkpoint 阶段的元数据的可靠性，同时，进行 fsimage 与 edits 的合并，可以有效限制 edits 的大小，防止其无限制增长。

缺点

- 没有做到热备，当 NameNode 无法提供服务时，需要重启 NameNode，服务恢复时间与文件系统规模大小成正比。
- Secondary NameNode 保存的只是 Checkpoint 时刻的元数据，因此，一旦 NameNode 上的元数据损坏，通过 Checkpoint 恢复的元数据并不是 HDFS 此刻的最新数据，存在一致性问题。

1.4.3　Hadoop 的 Checkpoint Node 方案

Checkpoint Node 方案与 Secondary NameNode 的原理基本相同，只是实现方

式不同。该方案利用 Hadoop 的 Checkpoint 机制进行备份，配置一个 Checkpoint Node。该节点会定期从 Primary NameNode 中下载元数据信息（fsimage+edits），将 edits 与 fsimage 进行合并，在本地形成最新的 Checkpoint，并上传到 Primary NameNode 进行更新。

当 NameNode 发生故障时，极端情况下（NameNode 彻底无法恢复），可以在备用节点上启动一个 NameNode，读取 Checkpoint 信息，提供服务。

优点

- 使用简单方便、无需开发、配置即可。
- 元数据有多个备份。

缺点

- 没有做到热备、备份节点切换时间长。
- Checkpoint Node 所做的备份，只是最后一次 Check 时的元数据信息，并不是发生故障时最新的元数据信息，有可能造成数据的不一致。

1.4.4 Hadoop 的 Backup Node 方案

利用新版本 Hadoop 自身的 Failover 措施，配置一个 Backup Node，Backup Node 在内存和本地磁盘均保存了 HDFS 系统最新的名字空间元数据信息。如果 NameNode 发生故障，可用使用 Backup Node 中最新的元数据信息。

优点

- 简单方便、无需开发、配置即可使用。
- Backup Node 的内存中对当前最新元数据信息进行了备份（Namespace），避免了通过 NFS 挂载进行备份所带来的风险。
- Backup Node 可以直接利用内存中的元数据信息进行 Checkpoint，保存到本地，与从 NameNode 下载元数据进行 Checkpoint 的方式相比效率更高。

- NameNode 会将元数据操作的日志记录同步到 Backup Node，Backup Node 会将收到的日志记录在内存中更新元数据状态，同时更新磁盘上的 edits，只有当两者操作成功，整个操作才成功。这样即便 NameNode 上的元数据发生损坏，Backup Node 的磁盘上也保存了 HDFS 最新的元数据，从而保证了一致性。

缺点

- 高版本（0.21 以上）才支持。
- 许多特性还处于开发之中，例如：当 NameNode 无法工作时，Backup Node 目前还无法直接接替 NameNode 提供服务，因此当前版本的 Backup Node 还不具有热备功能，也就是说，当 NameNode 发生故障，目前还只能通过重启 NameNode 的方式来恢复服务。
- Backup Node 的内存中未保存 Block 的位置信息，仍然需要等待下面的 DataNode 进行上报，因此，即便在后续的版本中实现了热备，仍然需要一定的切换时间。
- 当前版本只允许 1 个 Backup Node 连接到 NameNode。

1.4.5　DRDB 方案

利用 DRDB 机制进行元数据备份。

当 NameNode 发生故障时，可以启动备用机器的 NameNode，读取 DRDB 备份的元数据信息，提供服务。

优点

- DRDB 是一种较为成熟的备份机制。
- 元数据有多个备份、并且保持最新状态。
- 由于备份的工作交由 DRDB 完成，对于一条新的日志记录，NameNode 无需同步写入多个备份目录，因而 NameNode 在效率上优于 Hadoop 的元数

据备份方案。

缺点

- 没有做到热备、备份节点切换时间长。
- 需要引入新的机制、由此带来一定的可靠性问题。

1.4.6 FaceBook 的 AvatarNode 方案

利用 FaceBook 提出的 Avatar Node机制。

Active Node 作为 Primary NameNode 对外提供服务。Standby Node 处于 Safe mode 模式，在内存中保存 Primary NameNode 最新的元数据信息。Active Node 和 Standby Node 通过 NFS 共享存储进行交互。DataNode 同时向 Active Node 和 Standby Node 发送 Block location 信息。当管理员确定 Primary NameNode 发生故障后，将 Standby Node 切换为 Primary NameNode。由于 Standby Node 内存中保存了所有元数据的最新信息，因此可直接对外提供服务，大大缩短了切换时间。

优点

- 提供热备、切换时间大大缩短。
- FaceBook 已将其集成到自己维护的 Hadoop 代码中，并部署到了自己的集群使用。

缺点

- 修改了部分源码、增加了一定的复杂性，并在软件的维护性上带来一定问题。
- 参考资料较少。
- 只提供一个备份节点。

1.5 方案优缺点比较

综上所述，HDFS HA 方案比较如表 1.1 所示。

表 1.1　HDFS HA 方案比较

方案名称	切换时间	元数据一致性	是否做 checkpoint	使用复杂度	成熟度	相关资料
元数据备份	长	一致	否	低	高	较多
Secondary NameNode	长	不一定	是	中	高	较多
Checkpoint Node	长	不一定	是	中	高	较少
Backup Node	中	一致	是	中	中	较少
DRDB	长	一致	否	高	高	多
AvatarNode	短	一致	是	高	高	少

其中“元数据备份方案”不能单独使用，因为在系统运行期间，没有相应的 Checkpoint 机制，会造成日志的无限制增长，因此需要和 Secondary NameNode、Checkpoint Node或 Backup Node 配合使用。

“DRDB 方案”同样如此。

而对于 Secondary NameNode、Checkpoint Node 机制，它们只有 Checkpoint 的功能，而不能保存实时的元数据，因此需要在 Namenode 上配置元数据备份路径来保存实时元数据。

对于 Backup Node，虽然它可以实时保存元数据，但为防止 Backup Node 成为一个单点，也需要在 NameNode 上配置元数据备份路径，保存在本地进行备份。

总结

- 元数据备份方案使用简单方便，在功能上可替代 DRDB；
- Backup Node 是 Checkpoint Node 的升级版，效率更高；
- Secondary NameNode 在低版本的 Hadoop 中就已存在。

因此用户实际上可选择的 HA 组合方案为：

（1）元数据备份+ Secondary NameNode

这种方案适用于目前 Hadoop 的所有版本，属于冷备，切换时间长。由于

Secondary NameNode 自身并不实时保存元数据，一旦 NameNode 上的元数据损坏，将无法恢复到最新的元数据，因此采用元数据备份机制，在 NameNode 上需要配置多个目录进行备份，常见的做法是再配置一个 NFS 节点，共享一个目录进行备份。

（2）元数据备份+ Backup Node

这种方案只有在 0.21.0 以上版本才支持，目前的实现只支持冷备，切换时间长，自身实时保存元数据，不需要 NFS 节点。

（3）元数据备份+ AvatarNode

这种方案需要打 Patch（补丁包），而且只支持特定的版本（0.20）或者使用 FaceBook 自身的 Hadoop 版本，切换时间短，需要一个 NFS 节点作为 Active 节点和 Standby 节点的数据交互节点。

> 总之，如果当前 Hadoop 的版本较低，同时也不允许升级版本的话，可以选择第 1 种方案；如果版本较新（在 0.21.0 以上），第 2 种方案优于第 1 种；如果对 HA 切换时间有严格要求的话，则需要选择第 3 种方案。第 1 种方案比较简单，资料较多，本书将不再说明，后面着重讲述第 2 种和第 3 种方案。

第 2 章　HDFS 元数据解析

前面已经说过，HDFS 中的 HA 主要解决 NameNode 的 HA，NameNode 的 HA 最重要的目的就是维护元数据的高可用性，因此，有必要对 HDFS 的元数据进行深入了解。

2.1 概 述

所谓元数据[1]（Metadata）就是指数据的数据。HDFS 的元数据就是指维护 HDFS 文件系统中的文件和目录所需要的信息。

> 需要注意的是，具体的文件内容不是元数据，元数据是用于描述和组织具体的文件内容，如果没有元数据，具体的文件内容将变得没有意义。元数据的作用十分重要，它的可用性直接决定了 HDFS 的可用性。

从形式上讲，元数据可分为**内存元数据**和**元数据文件**两种。其中 NameNode 在内存中维护整个文件系统的元数据镜像，用于 HDFS 的管理；元数据文件则用于持久化存储。

从类型上讲，元数据有三类重要信息：

- 第 1 类是文件和目录自身的属性信息，例如文件名、目录名、父目录信息、文件大小、创建时间、修改时间等；
- 第 2 类记录文件内容存储相关信息，例如文件分块情况、副本个数、每个副本所在的 Data Node 信息等；
- 第 3 类用来记录 HDFS 中所有 Data Node 的信息，用于 Data Node 管理。

从来源上讲，元数据主要来源于 NameNode 磁盘上的元数据文件（它包括元数据镜像 fsimage 和元数据操作日志 edits 两个文件）以及各个 Data Node 的上报信息。

下面我们结合源码进行分析，源码的版本为 0.21.0。

2.2 内存元数据结构

从 HDFS 自身实现的角度来看，文件和目录是文件系统的基本元素，HDFS 将这些元素抽象成 INode，每一个文件或目录都对应一个唯一的 INode，INode 存储

[1] http://baike.baidu.com/view/107838.htm

了名字信息（分别对应文件或目录的名字）同时还存储了创建时间、修改时间、父目录等信息，这些都是目录和文件的一些公共属性。

> 需要注意的是：有了 INode 信息，HDFS 就可以构建整个文件系统的层次结构，并保存每个文件或目录的属性信息，用户就可以对文件和目录进行创建、删除等操作，唯一不能完成的操作是读取和写入文件的内容。

2.2.1　INode

在具体实现上，类 FSNamesystem 的成员变量 dir 实现了对整个 HDFS 中 INode 的组织和操作，它是一个 FSDirectory 类，由于所有 INode 的信息完全位于内存，因此可有效提高元数据的服务性能。

正因为 INode 信息完全位于内存，一旦掉电将不再存在，因此需要将 INode 信息保存到磁盘，这个功能是由类 FSImage 完成的，类 FSImage 是构架在内存元数据与磁盘元数据文件之间的桥梁，在 HDFS 初始化时，它负责将磁盘元数据文件中的记录转化为内存元数据中的 INode；在需要持久化存储时，它负责将元数据操作转换为日志记录并保存。

由于所有的元数据都位于内存，其大小随文件系统的规模增大而增大，如果每次都将整个内存元数据导出到磁盘，将会带来很大的系统开销，HDFS 在实现时，没有采用定期导出元数据的方法，而是采用元数据镜像文件（FSImage）+日志文件（edits）的备份机制，其中镜像文件是某一时刻内存元数据的真实组织情况，而日志文件则记录了该时刻以后所有的元数据操作。

HDFS 启动时会读取元数据镜像文件和日志文件到内存，进行合并形成最新的内存元数据，随后将该元数据保存到磁盘，形成新的磁盘镜像文件和日志文件；在运行过程中，HDFS 不再从内存导出元数据，而只是将元数据操作记录到日志文件中。

该机制的优点是：既保证了元数据内容不丢失，同时又最大程度地降低了备份元数据的开销；缺点是：在 HDFS 启动加载后，进行合并会消耗一定的时间。

具体实现时，所有与元数据镜像文件有关的操作由类 FSDirectory 的成员变量

fsImage 完成，它是一个 FSImage 类，所有与日志文件有关的操作由类 FSImage 的成员变量 editLog 完成，它是一个 FSEditLog 类。

2.2.2 Block

至此，我们讲到了和 INode 有关的几个类，通过 INode 我们可以知道文件的名称、创建时间、大小等属性，但这些信息对于访问某一个具体文件的内容来说是不够的。在此，我们引入 HDFS 中的另外一个概念——Block。

Block 是对于文件内容组织而言的，我们假设一个文件的长度大小为 size，那么从文件的 0 偏移开始，按照固定的大小，顺序对文件进行划分并编号，划分好的每一个块就称之为一个 Block，HDFS 默认的 Block 大小为 64MB，以一个 256MB 大小的文件为例，该文件一共有 256/64=4 个 Block。

对于每一个 Block，HDFS 将其内容复制多份，以文件的形式保存到各个 Data Node 上，这些文件就称之为该 Block 的副本（Replica），HDFS 中默认的副本数为 3。对于 1 个 256MB 的文件来说，它有 4 个 Block，每个 Block 对应有 3 个副本。因此，如果要访问文件指定偏移量的数据，首先可以根据 Block 的大小计算出该偏移所在的 Block 以及在该 Block 内的偏移，然后还要知道该 Block 对应的副本信息，其中最重要的信息是该副本所在的 Data Node 的信息。

类 INodeFile 中设置了一个 Block 的数组（protected BlockInfo blocks[] = null;）来保存该文件所有的 Block 信息，并提供了相应的 Block 操作方法。每一个 Block 的信息由类 BlockInfo 来表示，类 BlockInfo 中的成员变量（private Object[] triplets;）保存了该 Block 副本所在的 Data Node 的信息。每一个 Data Node 信息通过类 DatanodeDescriptor 进行描述，具体包括：容量、空间使用率、剩余容量、更新时间、主机名等。

至此，当一个客户端访问某一文件特定偏移量的内容时，HDFS 首先根据路径信息找到该文件对应的 INode，根据偏移计算出 Block 位置，在“protected BlockInfo blocks[] = null;”中找到相应的 BlockInfo，然后在“private Object[] triplets;”找到副本所在 Data Node 的信息，然后选择其中的一个 Data Node 进行

连接，获取相应的内容。同样，Block 信息也是位于内存的，但是它不需要导出到磁盘进行存储，这是因为 Block 信息来源于底下各个 Data Node 的上报信息，这些信息由 Data Node 维护，因而不需要在 NameNode 上再进行保存。

2.2.3　BlockInfo 和 DatanodeDescriptor

综上所述，我们可知道类 INode、BlockInfo、DatanodeDescriptor 是 HDFS 元数据的三个关键类，分别代表了三种管理对象：

- INode 代表的是文件系统基本元素：文件和目录；
- BlockInfo 代表的是文件内容对象；
- DatanodeDescriptor 代表的是具体存储对象。

前面所述的类 FSDirectory 实现了 INode 的管理，下面将对类 BlockManager 和 NavigableMap<String, DatanodeDescriptor> datanodeMap 进行说明，它们分别实现了对 BlockInfo 和 DatanodeDescriptor 的管理。

类 BlockManager 包含了一个成员变量“final BlocksMap blocksMap;”，类 BlocksMap 实现了 HDFS 中所有 Block 的管理，它构建了一个 Hash 表“private Map<BlockInfo, BlockInfo> map;”来存储所有的 BlockInfo，并实现 Block 的快速检索及其他操作，如：以 Block 为输入参数，获取该 Block 对应的 INode、设置该 Block 对应的 INode、删除该 Block 对应的 INode、删除该 Block 对应的 BlockInfo、获取该 Block 对应的 BlockInfo 等。

Hash 表“NavigableMap<String, DatanodeDescriptor> datanodeMap”实现了 HDFS 中所有 Data Node 的管理，它存储了 HDFS 中所有的 Data Node，通过键值 StorageID 可以快速定位到相应的 Data Node。

2.2.4　小结

至此我们从下向上对元数据的结构及相应的实现进行了说明，现在再从上到下梳理一遍：在 HDFS 中，由类 FSNamesystem 来代表总的元数据，由它对外提供统

一的元数据操作接口，而这些接口的具体实现则是由它的成员变量及相关方法组合实现的，其中最重要的三个成员变量分别是：

- public FSDirectory dir
- BlockManager blockManager
- NavigableMap<String, DatanodeDescriptor> datanodeMap

下面我们结合具体的代码对元数据结构进行分析。

2.2.5 代码分析——元数据结构

先来看类 FSNamesystem 的三个重要成员变量。如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
FSNamesystem.java
public class FSNamesystem implements FSConstants, FSNamesystemMBean,
FSClusterStats {
    ……
    public FSDirectory dir;
    BlockManager blockManager;
    NavigableMap<String, DatanodeDescriptor> datanodeMap =
        new TreeMap<String, DatanodeDescriptor>();
    ……
}
```

类 FSDirectory 实现了 INode 的管理，并且通过成员变量“FSImage fsImage”实现了元数据信息的加载以及持久化存储。如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
FSDirectory.java
class FSDirectory implements FSConstants, Closeable s {

    final INodeDirectoryWithQuota rootDir;
```

```
    FSImage fsImage;
    ……
    INodeFileUnderConstruction addFile(……){……};
    boolean mkdirs(……){ ……};
    ……
}
```

类 FSImage 是实现 INode 信息与元数据镜像文件 fsimage 以及日志文件 edits 之间相互转换的桥梁，其中与日志有关的功能由其成员变量“FSEditLog editLog”来实现。如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
FSImage.java
public class FSImage extends Storage {
    ……
    protected FSEditLog editLog = null;
    ……
    boolean loadFSImage() throws IOException {……}
    ……
    int loadFSEdits(StorageDirectory sd){……}
    ……
    void saveFSImage() throws IOException {……}

    public void format() throws IOException {……}
}
```

FSEdit 类定义了日志操作的类型，如 OP_INVALID=-1；OP_ADD = 0；OP_RENAME = 1；OP_DELETE = 2；OP_MKDIR = 3 等。如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
FSEditLogjava
public class FSEditLog {
```

```
    ……
    private ArrayList<EditLogOutputStream> editStreams = null;
    ……
    static int loadFSEdits(EditLogInputStream edits) throws IOException {……}

    synchronized void logEdit(byte op, Writable ... writables) {……}
    synchronized void logSyncAll() throws IOException {……}
    public void logSync() throws IOException {……}
    public void logOpenFile(String path, INodeFileUnderConstruction newNode)
        throws IOException {……}
    public void logCloseFile(String path, INodeFile newNode) {……}
    public void logMkDir(String path, INode newNode) {……}

    ……
}
```

HDFS 提供了一个分层的目录结构，最顶端是根目录“/”，文件系统中的每一个元素，如目录、文件等都称之为 INode，每一个 INode 有一些公共属性和方法，如名字（protected byte[] name;），父目录（protected INodeDirectory parent;），修改时间（rotected long modificationTime;），访问时间（protected volatile long accessTime;）等。

HDFS 中一共有 4 种类型的 INode，每一种类型由一种 INode 子类来实现，其中：

- 类 INodeDirectory 表示目录类型的 INode；
- 类 INodeFile 表示文件类型 INode；
- 类 INodeDirectoryWithQuota 是类 INodeDirectory 的子类，它表示有配额限制的目录；
- 类 INodeFileUnderConstruction 是类 INodeFile 的子类，它表示正在写入的文件，当用户创建一个文件时，HDFS 首先会实例化类 InodeFileUnder

Construction 的一个对象来表示该文件，接下来用户向文件写入数据，写入结束后，HDFS 会实例化一个 INode 对象来取代原来的类 INodeFileUnderConstruction 对象。

如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
INodejava
abstract class INode implements Comparable<byte[]>, FSInodeInfo {
    protected byte[] name;
    protected INodeDirectory parent;
    ……
}
package org.apache.hadoop.hdfs.server.namenode;

class INodeFile extends INode {protected byte[] name;
    ……
    protected BlockInfo blocks[] = null;
}
```

类 BlockManager 通过成员变量“final BlocksMap blocksMap;”实现 HDFS 所有 Block 的管理。

类 BlocksMap 通过 Hash 表“private Map<BlockInfo, BlockInfo> map;”来存储 HDFS 中所有的 Block，并且提供了相应的 Block 操作方法。

类 BlockInfo 是类 BlocksMap 的内部类，它包括“private INodeFile inode;”和“Object [] triplets;”等成员变量，其中：

“private INodeFileinode;”表示该 Block 所对应的 INode 节点；triplets[3*i]代表的是第 i 个副本所在的 Data Node，Data Node 维护了一个它所存储的副本的列表，以该 Block 所在副本列表位置为基准，triplets[3*i+1]代表的是副本列表的前一个 Block，triplets[3*i+2] 代表的是后一个 Block。

通过 Hash 表“private Map<BlockInfo, BlockInfo> map;”HDFS 可以快速地找到当前 Block 的 BlockInfo，然后通过 BlockInfo 中的 triplets[3*i+1]和 triplets[3*i+2]，HDFS 可以很方便地遍历该 Block 所在 Data Node 的所有 Block，通过 BlockInfo 中 inode 的“protected BlockInfo blocks[] = null;”可以很方便地遍历该 Block 对应文件的所有 Block。如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
BlocksMap.java
public class BlockManager {
    ……
    final BlocksMap blocksMap;
    ……
}
package org.apache.hadoop.hdfs.server.namenode;
BlocksMap.java
public class BlocksMap {
    ……
    private Map<BlockInfo, BlockInfo> map;
    boolean addNode(Block b, DatanodeDescriptor node, int replication) {……}
    boolean removeNode(Block b, DatanodeDescriptor node) {……}
    ……
}
package org.apache.hadoop.hdfs.server.namenode;
BlocksInfo.java
class BlockInfo extends Block {
    ……
    private INodeFile inode;
    ……
    private Object[] triplets;
```

```
}
```

类 DatanodeDescriptor 是类 DatanodeInfo 的子类，类 DatanodeInfo 的成员变量“private volatile BlockInfo blockList = null;”维护了该 Data Node 上所有 Block 的一个链表，其中：blockList 是链表的头，“protected long capacity;”、“protected long dfsUsed;”、“protected long remaining;”、“protected long lastUpdate;”、“protected String hostName = null;”分别记录了 Data Node 节点容量、空间使用率、剩余空间、更新时间、主机名等信息。如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
DatanodeDescriptor.java
public class DatanodeDescriptor extends DatanodeInfo {
    ……
    private volatile BlockInfo blockList = null;
    ……
    boolean addBlock(BlockInfo b) {……}
    ……
    boolean removeBlock(BlockInfo b) {……}
    ……
}
package org.apache.hadoop.hdfs.protocol;
DatanodeInfo.java
public class DatanodeInfo extends DatanodeID implements Node {
    protected long capacity;
    protected long dfsUsed;
    protected long remaining;
    protected long lastUpdate;
    ……
    protected String hostName = null;
    ……
}
```

2.3 磁盘元数据文件

NameNode 启动时，所有的元数据是位于内存进行管理的，一旦掉电，内存中的数据将不复存在，为此，需要将内存中的元数据保存到磁盘永久存储。

磁盘元数据文件包括以下四个：

- fsimage：元数据镜像文件，它存储的是某一时刻 NameNode 内存元数据信息，包括所有的 INode 信息、正在写入的文件信息以及其他的一些状态信息等；
- edits：日志文件，它保存了该时刻以后元数据的操作记录；
- fstime：保存了最近一次 Checkpoint 的时间；
- VERSION：是一个标志性文件，它最后被创建，它的存在表明前三个元数据文件的创建成功。

下面我们通过一个简单的例子来查看下具体的元数据文件。

1. 解压

```
$tar xzf  hadoop-0.21.0.tar.gz
```

2. 配置 hadoop-env.sh

```
$cd  hadoop-0.21.0
```

在最末尾加入下面一行，配置 JAVA_HOME 为 jdk 的安装路径：

```
$export JAVA_HOME=/home/blfs/hdfs/soft/jdk1.6.0_26
```

3. 设置环境变量

```
$source bin/hadoop-config.sh
$echo $HADOOP_HOME
```

如果可以看到相应的路径，则说明设置成功。

4. 配置元数据备份路径

```
$cp hdfs/src/java/hdfs-default.xml conf/hdfs-site.xml
$vi conf/hdfs-site.xml
```

配置 dfs.namenode.name.dir 的 value，它将作为 fsimage 的保存路径，可以配置多个，以“，”隔开。配置 dfs.namenode.edits.dir 的 value，它将作为 edits 的保存路径，可以配置多个，以“，”隔开。本例中 dfs.namenode.name.dir 的 value 配置为/tmp/dfs/name，dfs.namenode.edits.dir 的 value 配置为/tmp/dfs/log。如下所示。

```
<property>
    <name>dfs.namenode.name.dir</name>
    <value>/tmp/dfs/name</value>
    <description>Determines where on the local filesystem the DFS name node
    should store the name table(fsimage).  If this is a comma-delimited list
    of directories then the name table is replicated in all of the
    directories, for redundancy. </description>
</property>
<property>
    <name>dfs.namenode.edits.dir</name>
    <value>/tmp/dfs/log</value>
    <description>Determines where on the local filesystem the DFS name node
    should store the transaction (edits) file. If this is a comma-delimited list
    of directories then the transaction file is replicated in all of the
    directories, for redundancy. Default value is same as dfs.name.dir

</property>
```

5. 格式化

```
$bin/hdfs namenode format
```

6. 检查

```
$ls /tmp/dfs/name/
```

应该有 current 和 image 两个目录。

```
$ls /tmp/dfs/name/current
```

应该有“fsimage”、“fstime”、“VERSION”三个文件。

```
$ls /tmp/dfs/name/image
```

应该有“fsimage”一个文件。

```
$ls /tmp/dfs/log/
```

应该有 current 和 image 两个目录。

```
$ls /tmp/dfs/log/current
```

应该有“edits”、“fstime”、“VERSION”三个文件。

```
$ls /tmp/dfs/log/image
```

应该有“fsimage”一个文件。

由上述例子我们可知：

- fsimage 和 edits 的保存路径可以单独设置，如果不设置 edits 的保存路径，那么 edits 默认的保存路径是 fsimage 的保存路径；
- fsimage 保存在 fsimage 保存路径的 current 目录下，edits 保存在 eidts 保存路径的 current 目录下；
- fsimage 的保存路径以及 edits 的保存路径都可以设置多个，以“，”作为分隔符。

此外，fsimage 文件存在两个状态，以文件名进行区分，分别是：fsimage 和 fsimage.ckpt。fsimage 表示正常状态，fsimage.ckpt 表示 Checkpoint 过程中，上传到 NameNode 的 fsimage 文件，Checkpoint 结束时会将 fsimage.ckpt 重命名为

fsimage。

edits 也有两个状态，同样以文件名进行区分，分别是：edits 和 edits.new。在 HDFS 正常运行的过程中，元数据操作记录都会写入 edits 文件，当开始 Checkpoint 时，日志记录转而写入 edits.new，而 edits 则作为此时刻日志的一个快照，供 Checkpoint 进行元数据合并，当 Checkpoint 结束后，会将 edits.new 重命名为 edits。

2.4　Format 情景分析

和我们平时使用本地文件系统一样，HDFS 在使用之前也需要格式化。所谓格式化就是对文件系统进行初始化，形成一个用户未写入数据前的初始系统。格式化操作在 NameNode 进行，由用户发起，具体命令调用如下。

```
$bin/hdfs namenode format
```

format 主要分为以下几个步骤。

（1）确定能否格式化；

（2）创建元数据文件在内存中的镜像；

（3）对内存镜像中的数据结构进行初始化；

（4）将内存镜像写入元数据备份目录。

下面结合代码对以上步骤进行分析。

1. 确定能否格式化

format 命令的入口是类 NameNode 的 main 函数，Main 函数中的主要方法是 createNameNode，该方法根据输入的参数来决定执行相应的动作，对于格式化操作将调用 format 方法，在 format 方法中，遍历所有的元数据存储目录，提示用户是否允许对其格式化，如果用户不允许，则中断返回，只有用户确定允许所有目录格式化后才向下执行。

```
package org.apache.hadoop.hdfs.server.namenode;
NameNode.java
public class NameNode implements NamenodeProtocols, FSConstants {
    ……
    public static void main(String argv[]) throws Exception {
        try {
           StringUtils.startupShutdownMessage(NameNode.class, argv, LOG);
            NameNode namenode = createNameNode(argv, null);
            if (namenode != null)
                namenode.join();
        } catch (Throwable e) {
            LOG.error(StringUtils.stringifyException(e));
            System.exit(-1);
        }
    }
}
package org.apache.hadoop.hdfs.server.namenode;
NameNode.java
public static NameNode createNameNode(String argv[], Configuration conf)
throws IOException {
    // 解析并设置启动参数，并实例化类 Configuration 的对象 conf

    // 根据不同的启动参数，采用不同的处理方法
    StartupOption startOpt = parseArguments(argv);
    if (startOpt == null) {
        printUsage();
        return null;
    }

    switch (startOpt) {
```

```
        case FORMAT:
            boolean aborted = format(conf, true);
            System.exit(aborted ? 1 : 0);
            return null; // avoid javac warning
        case FINALIZE:
            aborted = finalize(conf, true);
            System.exit(aborted ? 1 : 0);
            return null; // avoid javac warning
        case BACKUP:
        case CHECKPOINT:
            return new BackupNode(conf, startOpt.toNodeRole());
        default:
            return new NameNode(conf);
        }
    }
```

package org.apache.hadoop.hdfs.server.namenode;

NameNode.java

```
    private static boolean format(Configuration conf, boolean
isConfirmationNeeded)
    throws IOException {
        // 与用户交互来决定是否允许进行 format
        Collection<URI> dirsToFormat = FSNamesystem.getNamespaceDirs(conf);
        Collection<URI> editDirsToFormat =
            FSNamesystem.getNamespaceEditsDirs(conf);
        for(Iterator<URI> it = dirsToFormat.iterator(); it.hasNext();) {
            File curDir = new File(it.next().getPath());
            if (!curDir.exists())
                continue;
            if (isConfirmationNeeded) {
```

```
            if (!(System.in.read() == 'Y')) {
                System.err.println("Format aborted in "+ curDir);
                return true;
            }
            while(System.in.read() != '\n'); // discard the enter-key
        }
    }
    FSNamesystem nsys = new FSNamesystem(new FSImage(dirsToFormat,
            editDirsToFormat), conf);
    nsys.dir.fsImage.format();

}
```

2. 创建元数据文件在内存中的镜像

元数据的内存镜像包括：类 FSNamesystem 实例化对象，类 FSDirectory 的实例化对象，类 FSImage 的实例化对象，类 FSEdit 的实例化对象。

- 类 FSNamesystem 代表的是整个 HDFS 的 Namespace；
- 类 FSDirectory 代表了 HDFS 中所有的目录结构及其属性；
- 类 FSImage 的实例化对象对应元数据的磁盘文件 fsImage；
- 类 FSEdit 的实例化对象对应元数据的磁盘文件 edits。

下面我们依次进行说明。

首先是创建类 FSNamesystem 的对象，其构造函数主要完成 3 个功能：

第 1 是创建类 BlockManager 的对象；

第 2 是读取配置内容，对 FSNamesystem 的成员变量进行初始化；

第 3 是创建类 FSDirectory 对象。

```
package org.apache.hadoop.hdfs.server.namenode;
FSNamesystem.java
FSNamesystem(FSImage fsImage, Configuration conf) throws IOException {

    setConfigurationParameters(conf);
    this.dir = new FSDirectory(fsImage, this, conf);
    dtSecretManager = createDelegationTokenSecretManager(conf);
}
```

类 FSImage 的实例化对象作为参数传入类 FSNamesystem 的构造函数。类 FSImage 的构造函数主要完成 3 个动作：

第 1 是成员变量的初始化，类 FSImage 是类 Storage 的子类，而类 Storage 又是类 StorageInfo 的子类，在类 FSImage 的构造函数中依次调用了父类的构造函数，最终在 StorageInfo 的构造函数中对 layoutVersion、namespaceID、cTime 这几个成员变量赋初始值 0；

第 2 是实例化类 FSEditLog 的对象 EditLog；

第 3 是对备份目录进行分类，加入到 storageDirs，便于后续的目录操作。

```
package org.apache.hadoop.hdfs.server.namenode;
public class FSImage extends Storage {
    ……
    FSImage() {
        this((FSNamesystem)null);
    }
    FSImage(FSNamesystem ns) {
        super(NodeType.NAME_NODE);

        setFSNamesystem(ns);
    }
```

```
      FSImage(Collection<URI> fsDirs, Collection<URI> fsEditsDirs) throws
IOException {
          this();
          setStorageDirectories(fsDirs, fsEditsDirs);
      }
      void setStorageDirectories(Collection<URI> fsNameDirs,
Collection<URI> fsEditsDirs)
      throws IOException {
          this.storageDirs = new ArrayList<StorageDirectory>();
          this.removedStorageDirs = new ArrayList<StorageDirectory>();
          // Add all name dirs with appropriate NameNodeDirType
          for (URI dirName : fsNameDirs) {
              checkSchemeConsistency(dirName);
              boolean isAlsoEdits = false;
              for (URI editsDirName : fsEditsDirs) {
                  if (editsDirName.compareTo(dirName) == 0) {
                      isAlsoEdits = true;
                      fsEditsDirs.remove(editsDirName);
                      break;
                  }
              }
          NameNodeDirType dirType = (isAlsoEdits) ?
          NameNodeDirType.IMAGE_AND_EDITS :
          NameNodeDirType.IMAGE;
          // Add to the list of storage directories, only if the
          // URI is of type file://
          if(dirName.getScheme().compareTo(JournalType.FILE.name().toLow
erCase())
          == 0){
```

```
File(dirName.getPath()),
                dirType));
            }
          }
          // Add edits dirs if they are different from name dirs
          for (URI dirName : fsEditsDirs) {
             checkSchemeConsistency(dirName);
             // Add to the list of storage directories, only if the
             // URI is of type file://
             if(dirName.getScheme().compareTo(JournalType.FILE.name().to
LowerCase())
                == 0)
              this.addStorageDir(new StorageDirectory(new File(dirName.getPath()),
                NameNodeDirType.EDITS));
          }
       }
       }
```

package org.apache.hadoop.hdfs.server.common;

Storage.java

```
 public abstract class Storage extends StorageInfo {
     ……
     protected Storage(NodeType type) {
         super();
         this.storageType = type;
     }
}
```

package org.apache.hadoop.hdfs.server.common;

StorageInfo.java

```
public class StorageInfo {
```

```
    public StorageInfo () {
        this(0, 0, 0L);
    }
    public StorageInfo(int layoutV, int nsID, long cT) {
        layoutVersion = layoutV;
        namespaceID = nsID;
        cTime = cT;
    }
    ……
}
```

package org.apache.hadoop.hdfs.server.namenode;

FSEditLog.java

```
public class FSEditLog {
        ……
    FSEditLog(FSImage image) {
        fsimage = image;
        isSyncRunning = false;
        metrics = NameNode.getNameNodeMetrics();
        lastPrintTime = FSNamesystem.now();
    }
```

在类 FSDirectory 的构造函数中，创建了 HDFS 根目录的 INode 节点 rootDir，根目录是 HDFS 的最顶层目录，存在于 HDFS 生命周期的所有阶段。

```
package org.apache.hadoop.hdfs.server.namenode;
class FSDirectory implements Closeable {
    FSDirectory(FSImage fsImage, FSNamesystem ns, Configuration conf) {
        ……
        rootDir = new INodeDirectoryWithQuota(INodeDirectory.ROOT_NAME,
            ns.createFsOwnerPermissions(new FsPermission((short)0755)),
```

```
            Integer.MAX_VALUE, UNKNOWN_DISK_SPACE);
        ……
    }
}
```

3. 对内存镜像中的数据结构进行初始化

初始化工作主要是由类 FSImage 的 format 函数完成的，初始化的对象也主要是类 FSImage 的成员变量，包括：

- layoutVersion：表示的是当前软件所处的版本，以此版本来决定是否允许升级，layoutVersion 的默认值为-18；
- namespaceID：是 NameSpace 的一个重要属性，是 HDFS 的身份标识，它在 NameNode 格式化的时候产生，在 NameNode 的整个生命周期内都不改变，当 Data Node 注册到 NameNode 后，会获得该 NameNode 的 NamespaceID，并作为后续与 NameNode 通信的身份标识，对于未知身份的 Data Node，NameNode 会拒绝其通信请求，namespaceID 值的产生有一个专门的函数 newNamespaceID，它以 FSNamesystem 的时间（以毫秒形式表示的当前系统时间）为种子产生随机数，取随机数低 31 位作为 namespaceID；
- cTime：表示 FSImage 文件的创建时间；
- checkpointTime：表示 Namespace 的第 1 次 Checkpoint 时间，取当前时间值作为第 1 次 Checkpoint 时间。

```
package org.apache.hadoop.hdfs.server.namenode;
FSImage.java
public void format() throws IOException {
    this.layoutVersion = FSConstants.LAYOUT_VERSION;
    this.namespaceID = newNamespaceID();
```

```
        this.checkpointTime = FSNamesystem.now();
        for (Iterator<StorageDirectory> it =  dirIterator(); it.hasNext();) {
            StorageDirectory sd = it.next();
            format(sd);
        }
    }
    private int newNamespaceID() {
        Random r = new Random();
        r.setSeed(FSNamesystem.now());
        int newID = 0;
        while(newID == 0)
            newID = r.nextInt(0x7FFFFFFF);  // use 31 bits only
        return newID;
    }
```

4. 将内存镜像写入元数据备份目录

类 FSImage 初始化成员变量后，会遍历所有的元数据存储目录，以存储目录作为参数，依次调用 format 方法，format 方法采用了重载的方式，可以根据输入参数的个数和类型确定所调用的方法，此处调用的方法为 format(StorageDirectory sd)。

该方法首先调用“sd.clearDirectory();”删除当前存储目录下[配置的 fsimage 路径（edits 路径）/current]的所有内容；

然后对传入的目录类型进行判断，如果是存储 FSImage 文件的目录，则调用 saveFSImage 保存 FSImage，如果是存储 Edits 日志文件的目录，则调用 editLog.createEditLogFile，在该目录下创建 Edits 文件；

最后调用 sd.write()方法在存储目录下创建 fstime 和 VERSION 文件，VERSION 通常是在存储目录更新的最后写入，VERSION 的存在表明存储目录下其他的文件已成功写入，因此该存储目录有效无需恢复，VERSION 文件的内容为：layoutVersion、storageType、namespaceID、cTime。

来看下面的代码。

```
package org.apache.hadoop.hdfs.server.namenode;
FSImage.java
void format(StorageDirectory sd) throws IOException {
    sd.clearDirectory(); // create currrent dir
    sd.lock();
    try {
        saveCurrent(sd);
    } finally {
        sd.unlock();
    }
    LOG.info("Storage directory " + sd.getRoot() + " has been successfully
formatted.");
}
static File getImageFile(StorageDirectory sd, NameNodeFile type) {
    return new File(sd.getCurrentDir(), type.getName());
}
protected void saveCurrent(StorageDirectory sd) throws IOException {
    File curDir = sd.getCurrentDir();
    NameNodeDirType dirType = (NameNodeDirType)sd.getStorageDirType();
    // save new image or new edits
    if (!curDir.exists() && !curDir.mkdir())
        throw new IOException("Cannot create directory " + curDir);
    if (dirType.isOfType(NameNodeDirType.IMAGE))
        saveFSImage(getImageFile(sd, NameNodeFile.IMAGE));
    if (dirType.isOfType(NameNodeDirType.EDITS))
       editLog.createEditLogFile(getImageFile(sd, NameNodeFile.EDITS));
    // write version and time files
```

```
    }
    void saveFSImage(File newFile) throws IOException {
        FSNamesystem fsNamesys = getFSNamesystem();
        FSDirectory fsDir = fsNamesys.dir;
        long startTime = FSNamesystem.now();
        DataOutputStream out = new DataOutputStream(new BufferedOutputStream(
            new FileOutputStream(newFile)));
        try {
            out.writeInt(FSConstants.LAYOUT_VERSION);
            out.writeInt(namespaceID);
            out.writeLong(fsDir.rootDir.numItemsInTree());
            out.writeLong(fsNamesys.getGenerationStamp());
            byte[] byteStore = new byte[4*FSConstants.MAX_PATH_LENGTH];
            ByteBuffer strbuf = ByteBuffer.wrap(byteStore);
            // save the root
            saveINode2Image(strbuf, fsDir.rootDir, out);
            // save the rest of the nodes
            saveImage(strbuf, 0, fsDir.rootDir, out);
            fsNamesys.saveFilesUnderConstruction(out);
            fsNamesys.saveSecretManagerState(out);
            strbuf = null;
        } finally {
            out.close();
        }
        LOG.info("Image file of size " + newFile.length() + " saved in "
            + (FSNamesystem.now() - startTime)/1000 + " seconds.");
    }

node,DataOutputStream out)
    throws IOException {
```

```
      int nameLen = name.position();
      out.writeShort(nameLen);
      out.write(name.array(), name.arrayOffset(), nameLen);
      if (node.isDirectory()) {
         out.writeShort(0);  // replication
         out.writeLong(node.getModificationTime());
         out.writeLong(0);   // access time
         out.writeLong(0);   // preferred block size
         out.writeInt(-1);   // # of blocks
         out.writeLong(node.getNsQuota());
         out.writeLong(node.getDsQuota());
         FILE_PERM.fromShort(node.getFsPermissionShort());
         PermissionStatus.write(out, node.getUserName(),
node.getGroupName(),
            FILE_PERM);
      } else if (node.isLink()) {
         out.writeShort(0);  // replication
         out.writeLong(0);   // modification time
         out.writeLong(0);   // access time
         out.writeLong(0);   // preferred block size
         out.writeInt(-2);   // # of blocks

         FILE_PERM.fromShort(node.getFsPermissionShort());
         PermissionStatus.write(out, node.getUserName(),
node.getGroupName(),
            FILE_PERM);
      } else {
         INodeFile fileINode = (INodeFile)node;

         out.writeLong(fileINode.getModificationTime());
```

```
            out.writeLong(fileINode.getAccessTime());
            out.writeLong(fileINode.getPreferredBlockSize());
            Block[] blocks = fileINode.getBlocks();
            out.writeInt(blocks.length);
            for (Block blk : blocks)
               blk.write(out);
            FILE_PERM.fromShort(fileINode.getFsPermissionShort());
            PermissionStatus.write(out, fileINode.getUserName(),
fileINode.getGroupName(),
               FILE_PERM);
         }
      }
      private static void saveImage(ByteBuffer parentPrefix, int prefixLength,
         INodeDirectory current, DataOutputStream out) throws IOException {
         int newPrefixLength = prefixLength;
         if (current.getChildrenRaw() == null)
            return;
         for(INode child : current.getChildren()) {
            // print all children first
            parentPrefix.position(prefixLength);
            parentPrefix.put(PATH_SEPARATOR).put(child.getLocalNameBytes());
            saveINode2Image(parentPrefix, child, out);
         }
         for(INode child : current.getChildren()) {
            if(!child.isDirectory())
               continue;
            parentPrefix.position(prefixLength);

            newPrefixLength = parentPrefix.position();
            saveImage(parentPrefix, newPrefixLength, (INodeDirectory)child, out);
```

```
        }
        parentPrefix.position(prefixLength);
    }
    package org.apache.hadoop.hdfs.server.namenode;
    FSNamesystem.java
    void saveFilesUnderConstruction(DataOutputStream out) throws IOException {
        synchronized (leaseManager) {
            out.writeInt(leaseManager.countPath()); // write the size
            for (Lease lease : leaseManager.getSortedLeases()) {
                for(String path : lease.getPaths()) {
                    // verify that path exists in namespace
                    INode node;
                    try {
                        node = dir.getFileINode(path);
                    } catch (UnresolvedLinkException e) {
                     throw new AssertionError("Lease files should reside on this FS");
                    }
                    if (node == null) {
                       throw new IOException("saveLeases found path " + path +
                                  " but no matching entry in namespace.");
                    }
                    if (!node.isUnderConstruction()) {
                        throw new IOException("saveLeases found path " + path
+ " but is not
                            under construction.");
                    }
                    INodeFileUnderConstruction cons =
(INodeFileUnderConstruction) node;
                    FSImage.writeINodeUnderConstruction(out, cons, path);
```

```
        }
    }
}
package org.apache.hadoop.hdfs.server.namenode;
FSImage.java
static void writeINodeUnderConstruction(DataOutputStream out,
    INodeFileUnderConstruction cons, String path)  throws IOException {

    out.writeShort(cons.getReplication());
    out.writeLong(cons.getModificationTime());
    out.writeLong(cons.getPreferredBlockSize());
    int nrBlocks = cons.getBlocks().length;
    out.writeInt(nrBlocks);
    for (int i = 0; i < nrBlocks; i++) {
        cons.getBlocks()[i].write(out);
    }
    cons.getPermissionStatus().write(out);
    writeString(cons.getClientName(), out);

    out.writeInt(0); //  do not store locations of last block
}
```

format 在写入 FSImage 文件或 Edits 文件成功后调用类 StorageDirectory 的 wirte 方法写入 fstime 和 VERSION 文件，类 StorageDirectory 是类 Storage 的内部成员类，可以访问类 Storage 所声明的成员变量和方法，wirte 方法首先将类 Storage 自身的属性信息（layoutVersion、storageType、namespaceID、cTime）写入类 Properties 的实例化对象 props 中，通过类 FSImage 重载 setFields 方法，写入 fstime 文件，然后实例化类 RandomAccessFile 的对象 file，用于对 VESRION 文件的操作，最后通过 file 写入属性信息并更新 VERSION 文件长度。

```
package org.apache.hadoop.hdfs.server.namenode;
Storage.java
public abstract class Storage extends StorageInfo {
    ……
    public class StorageDirectory {
        ……
        public File getVersionFile() {
            return new File(new File(root, STORAGE_DIR_CURRENT),
                STORAGE_FILE_VERSION);
        }
        public void write() throws IOException {
            corruptPreUpgradeStorage(root);
            write(getVersionFile());
        }
        public void write(File to) throws IOException {
            Properties props = new Properties();
            // 类 StorageDirectory 是类 Storage 的内部类，它可以直接调用类
            // Storage 的方法 setFields。类FSImage 继承自类Storage,因此在类FSImage

            //的setFields 方法。类 FSImage 的 setFields 方法将写入 fstime 文件。
            setFields(props, this);
            RandomAccessFile file = new RandomAccessFile(to, "rws");
            FileOutputStream out = null;
            try {
                file.seek(0);
                out = new FileOutputStream(file.getFD());
                props.store(out, null);
                file.setLength(out.getChannel().position());
            } finally {
                if (out != null) {
```

```
                    out.close();
                }
                file.close();
            }
        }
    }
```

类 FSImage 重载了父类 Storage 的 setFields 方法，在其方法中，将 checkpointTime 写入 fstime 文件。如下所示。

```
    package org.apache.hadoop.hdfs.server.namenode;
    FSImage.java
    protected void setFields(Properties props, StorageDirectory sd) throws
IOException {
        super.setFields(props, sd);
        boolean uState = getDistributedUpgradeState();
        int uVersion = getDistributedUpgradeVersion();
        if(uState && uVersion != getLayoutVersion()) {
            props.setProperty("distributedUpgradeState",
Boolean.toString(uState));
            props.setProperty("distributedUpgradeVersion",
Integer.toString(uVersion));
        }
        writeCheckpointTime(sd);
    }
    void writeCheckpointTime(StorageDirectory sd) throws IOException {
        if (checkpointTime < 0L)
            return; // do not write negative time
        File timeFile = getImageFile(sd, NameNodeFile.TIME);
```

```
            LOG.error("Cannot delete chekpoint time file: "+
timeFile.getCanonicalPath());
        }
        DataOutputStream out = new DataOutputStream(new
FileOutputStream(timeFile));
        try {
            out.writeLong(checkpointTime);
        } finally {
            out.close();
        }
    }
    package org.apache.hadoop.hdfs.server.namenode;
    Storage.java
    protected void setFields(Properties props, StorageDirectory sd ) throws
IOException {
        props.setProperty("layoutVersion", String.valueOf(layoutVersion));
        props.setProperty("storageType", storageType.toString());

        props.setProperty("cTime", String.valueOf(cTime));
    }
```

2.5　元数据应用场景分析

至此，我们对 HDFS 的元数据进行了详细介绍，从 HA 的角度，我们需要重点关注元数据所出现的应用场景，确保它在任何时刻的可用性。下面我们按照元数据的生命周期顺序，列举元数据所出现的各个应用场景。

场景 1

HDFS 在第 1 次使用前，需要进行格式化，格式化的结果是：在元数据镜像文件备份路径的 current 目录下，产生元数据文件：fsimage、fstime、VERSION 等；在日

志文件备份路径的 current 目录下，产生日志文件：edits、fstime、VERSION 等。

场景 2

启动 HDFS，HDFS 将 fsimage 和 edits 文件内容读入内存，进行合并，填充与 INode 相关的元数据结构，并将新的元数据内存镜像导出，在磁盘上形成新的 fsimage 和 edits 文件。HDFS 同时还接收 Data Node 的心跳信息，填充与 Block 和 Data Node 相关的元数据结构。

场景 3

HDFS 正常运行，此时如果有元数据更新操作，HDFS 会更新对应的内存元数据结构，并将该元数据操作日志记录写入磁盘的日志文件 edits。

场景 4

如果 NameNode 与 Secondary NameNode、Backup Node 或 Checkpoint Node 配合使用，那么，一定间隔后会进行 Checkpoint 操作，Checkpoint 操作会形成当前某一时刻的元数据镜像文件 fsimage，以该文件替换 NameNode 上原有的 fsimage，并以最新 fsimage 对应时刻之后的日志记录文件 edits 替换 NameNode 上原有的 edits。该机制可以有效限制日志文件的大小，防止其无限制增长，同时也降低了 HDFS 启动时的合并时间。

第 3 章　Hadoop 的元数据备份方案

Hadoop 元数据备份方案为 HDFS 元数据提供多个备份，防止因 NameNode 的元数据损坏，而导致整个文件系统无法正常服务。HDFS 保存在磁盘上的元数据文件包括：文件系统镜像文件 fsimage 以及日志记录文件 edits。用户可以配置多个元数据的存储目录，HDFS 会将元数据文件依次在这些目录中保存一份，并通过相应的机制保证这些数据的一致性。

3.1 运行机制分析

Hadoop 元数据备份机制相对比较简单，下面我们结合代码，对其运行机制进行分析。

新旧版本中 name.dir 的名称

我们采用的源码版本为 0.21.0，该版本通过 hdfs-site.xml 中的元数据备份目录项 dfs.namenode.name.dir[1]和 dfs.namenode.edits.dir[1]来配置多个元数据保存目录，老版本 hdfs-site.xml 中的元数据备份目录项名字有所不同，为 dfs.name.dir[2]以及 dfs.name.edits.dir[2]。

```
   package org.apache.hadoop.hdfs;
   HdfsConfiguration.java
   private static void addDeprecatedKeys() {
      ……
      deprecate("dfs.name.dir",
DFSConfigKeys.DFS_NAMENODE_NAME_DIR_KEY);
      ……
      deprecate("dfs.name.edits.dir",DFSConfigKeys.DFS_NAMENODE_EDITS_DI
R_KEY);

   }
```

元数据备份目录项的默认配置保存在 hdfs-default.xml 中，如果需要修改元数据备份目录项，可以从 hdfs-default.xml 中将相应的选项复制到 hdfs-site.xml 中进行配置，hdfs-default.xml 中的选项包括：

- dfs.namenode.name.dir：用来指明 Fsimage 的保存路径，默认值是

[1] http://hadoop.apache.org/hdfs/docs/r0.21.0/hdfs-default.html
[2] http://hadoop.apache.org/common/docs/r0.20.2/hdfs-default.html

file://${hadoop.tmp.dir}/dfs/name，路径之间使用“，”进行分隔。

- dfs.namenode.edits.dir：用来指明 Edits 的保存路径，其默认路径是 dfs.namenode.name.dir，也就是说，如果不配置 dfs.namenode.edits.dir，HDFS 将选择 dfs.namenode.name.dir 来保存 Edits，路径之间同样使用“，”进行分隔。

代码如下所示。

```
hdfs-default.xml
<property>
    <name>dfs.namenode.name.dir</name>
    <value>file://${hadoop.tmp.dir}/dfs/name</value>
    <description>Determines where on the local filesystem the DFS name node
    should store the name table(fsimage).  If this is a comma-delimited list
    of directories then the name table is replicated in all of the
    directories, for redundancy. </description>
</property>
<property>
    <name>dfs.namenode.edits.dir</name>
    <value>${dfs.namenode.name.dir}</value>
    <description>Determines where on the local filesystem the DFS name node

    of directories then the transaction file is replicated in all of the
    directories, for redundancy. Default value is same as dfs.name.dir

</property>
```

元数据备份目录项主要涉及到几个场景：

（1）首先是 NameNode 启动时，从 hdfs-site.xml 的元数据备份目录项中检查最新的 fsimage 和 edits，读取到内存进行合并，然后将 fsimage 写回

dfs.namenode.name.dir 指定的目录，并重置 edits；

（2）其次是有元数据更新时，NameNode 将日志记录写入 dfs.namenode.edits.dir 指定的目录；

（3）最后是做 Checkpoint 时，NameNode 将 Checkpoint 好的 fsimage 写回 dfs.namenode.name.dir 指定的目录，并重置 edits。

下面将对这几个场景进行分析。

3.1.1 NameNode 启动加载元数据情景分析

NameNode 定义了一个静态的初始化模块，在模块中调用 Configuration 的静态函数 addDefaultResource，将 hdfs-default.xml 和 hdfs-site.xml 加入到 Configuration 的 DefaultResource 中。由于该模块是静态初始化模块，当第 1 次实例化 NameNode 的时候，JVM 就会先于构造函数来调用该模块，且只调用一次。

```
package org.apache.hadoop.hdfs.server.namenode
NameNode.java
public class NameNode implements NamenodeProtocols, FSConstants {
    static{
        Configuration.addDefaultResource("hdfs-default.xml");
        Configuration.addDefaultResource("hdfs-site.xml");
    }
    ……
}
```

NameNode 的构造函数，主要调用的是 initialize 方法。

```
package org.apache.hadoop.hdfs.server.namenode
NameNode.java
protected NameNode(Configuration conf, NamenodeRole role) throws
IOException {
```

```
    try {
        initialize(conf);
    } catch (IOException e) {
        ……
    }
}
```

NameNode 在 initialize 方法中调用 loadNamesystem 来加载元数据到内存，loadNamesystem 方法创建了一个 FSNamesystem 类。HDFS 通过 FSNamesystem 来存储和操作内存中的元数据，FSNamesystem 的主要类成员变量包括：

- FSDirectory dir：dir 存储了整个 HDFS 文件系统的文件目录信息；
- BlockManager blockManager：存储了 Block 的相关信息，包括：

  1）Block 与元数据信息的映射（元数据包括 INode 信息 InodeFile 以及 Block 所在的 DataNode 信息 DatanodeDescriptor）；

  2）NavigableMap<String, DatanodeDescriptor> datanodeMap = new TreeMap<String, DatanodeDescriptor>()，实现了一个按照 Storage ID 排序的 Hash 列表，其中 Key 是 Storage ID，Value 是 DatanodeDescriptor，每个 DatanodeDescriptor 代表了一个 DataNode，记录了诸如当前可用磁盘空间、最后一次的更新时间等信息，同时还维护了该 DataNode 上所有 Block 的一个集合，通过该列表，实现了 DataNode 与 Block 的一个映射关系。

代码如下所示。

```
package org.apache.hadoop.hdfs.server.namenode
NameNode.java
protected void initialize(Configuration conf) throws IOException {

    loadNamesystem(conf);
```

```
        ……
    }
    //loadNamesystem 中创建 FSNamesystem 类
    protected void loadNamesystem(Configuration conf) throws IOException {
        this.namesystem = new FSNamesystem(conf);
    }
    package org.apache.hadoop.hdfs.server.namenode
    FSNamesystem.java
    public class FSNamesystem implements FSConstants, FSNamesystemMBean,
FSClusterStats
    {
        ……
        public FSDirectory dir;
        ……
        BlockManager blockManager;
        NavigableMap<String, DatanodeDescriptor> datanodeMap =
            new TreeMap<String, DatanodeDescriptor>();
        ……
        FSNamesystem(Configuration conf) throws IOException {
            try{
                initialize(conf, null);
            } catch(IOException e) {
                ……
            }

    }
```

FSNamesystem 构造函数的关键函数是 initialize 方法，initialize 方法首先创建 FSDirectory 的实例化对象 dir，然后调用 dir 的 loadFSImage 方法从磁盘上加载元数据到内存。其中 getNamespaceDirs 和 getNamespaceEditsDirs 用来分别对

hdfs-site.xml 中 dfs.namenode.name.dir 和 dfs.namenode.edits.dir 下的字符串进行解析，以“，”作为分隔符，将路径解析出来，返回字符串集合，作为 loadFSImage 方法的输入参数。代码如下所示。

```
   package org.apache.hadoop.hdfs.server.namenode
   FSNamesystem.java
   private void initialize(Configuration conf, FSImage fsImage)throws
IOException {
       ……
       if(fsImage == null) {
           this.dir = new FSDirectory(this, conf);
           StartupOption startOpt = NameNode.getStartupOption(conf);
           this.dir.loadFSImage(getNamespaceDirs(conf),
getNamespaceEditsDirs(conf),
               startOpt);
           ……
       } else {
           this.dir = new FSDirectory(fsImage, this, conf);
       }
       ……
   }
```

获取 dfs.namenode.name.dir 对应的值。

```
   FSNamesystem.java
   public static Collection<URI> getNamespaceDirs(Configuration conf) {
       return getStorageDirs(conf,
DFSConfigKeys.DFS_NAMENODE_NAME_DIR_KEY);
```

获取 dfs.namenode.edits.dir 对应的值。

```
   FSNamesystem.java
   public static Collection<URI> getNamespaceEditsDirs(Configuration conf) {
       return getStorageDirs(conf,
DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_KEY);
   }
```

Configuration 类中 getStorageDirs 方法，解析 xml 文件，获取相应名称下的数值。

```
   package org.apache.hadoop.hdfs.server.namenode
   FSNamesystem.java
   public static Collection<URI> getStorageDirs(Configuration conf, String
propertyName) {
       Collection<String> dirNames =
conf.getStringCollection(propertyName);
       ……
   }
   package org.apache.hadoop.conf;
   Configuration.java
   public Collection<String> getStringCollection(String name) {
       // 获得 name 下对应的字符串值
       String valueString = get(name);
       // 对字符串值进行解析，返回解析后元素的集合
       return StringUtils.getStringCollection(valueString);
   }
   package org.apache.hadoop.util;
   StringUtils.java
   public static Collection<String> getStringCollection(String str){

       if (str == null)
           return values;
```

```
        // 以", "为分隔符进行解析，返回路径的集合
        StringTokenizer tokenizer = new StringTokenizer (str,",");
        values = new ArrayList<String>();
        while (tokenizer.hasMoreTokens()) {
            values.add(tokenizer.nextToken());
        }
        return values;
    }
```

FSDirectory 的 loadFSImage 方法主要实现了元数据的检查、加载、内存合并以及保存。

关键函数是 FSImage 类的 recoverTransitionRead 和 saveNamespace。其中 recoverTransitionRead 主要实现了元数据的检查、加载及内存合并，saveNamespace 主要实现了元数据的持久化存储。

recoverTransitionRead 的输入参数 dataDirs 和 editsDirs 分别代表 fsimage 文件的存储路径和 edits 的存储路径，recoverTransitionRead 首先实例化类 StorageDirectory 的对象列表 storageDirs，然后对 dataDirs 和 editsDirs 中的路径进行过滤，分成 3 种类型：

- 第 1 种是既存储 fsimage 又存储 Edits 的路径；
- 第 2 种是专门存储 fsimage 的路径；
- 第 3 种是专门存储 edits 的路径。

在后面会涉及到大量的备份目录操作，将其合理分类便于迅速定位和查找，分类好的路径将加入到 storageDirs 中[fsimage（由父类 Storage 实现）实现了 1 个基于 storageDirs 的 Iterator，可以遍历 storageDirs 中的所有路径]，具体的实现函数为 setStorageDirectories；

然后 recoverTransitionRead 对 storageDirs 中的路径进行检查，计算状态，判断是否符合迁移的条件，对不符合条件的目录进行恢复；

接下来 recoverTransitionRead 会对 storageDirs 中未格式化的目录进行格式化；

最后 recoverTransitionRead 会加载最新的 fsimage 文件和 edits 到内存进行合并，并判断是否需要将内存中的元数据进行保存，具体实现函数为 loadFSImage。

saveNamespace 的功能是将内存中的元数据写回磁盘，保存为 fsimage 文件，同时创建空的 edits 文件。具体包括以下几个步骤：

（1）首先将备份目录下的 current 目录重命名为 lastcheckpoint.tmp；

（2）然后在备份目录下创建新的 current 目录，将元数据保存为 fsimage 文件，并创建空的 edits 文件；

（3）最后将 lastcheckpoint.tmp 目录重命名为 previos.checkpoint 目录。

由于重命名操作可迅速完成，且各阶段的元数据文件都有备份，根据目录的名字可以很清楚地判断 saveNamespace 所处的阶段，可以方便地回滚或前进到另一阶段，因此，以上机制可以有效地保证 saveNamespace 的可靠性以及各个目录下元数据的一致性。

saveNamespace 在访问备份目录的过程中，会将无法正常访问的备份目录加入到一个集合中，最后将其从 storageDirs 中去除，但是，如果由于备份目录的异常导致访问调用函数阻塞，那么整个 saveNamespace 过程就会阻塞。如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
FSDirectory.java
void loadFSImage(Collection<URI> dataDirs, Collection<URI> editsDirs,

startOpt) throws IOException {
    ……
    try {
       if (fsImage.recoverTransitionRead(dataDirs, editsDirs, startOpt)) {
           // 根据返回值确定是否需要将内存中的元数据写回磁盘
```

```
            // 只要任何一个目录需要保存元数据，则都需要保存
            fsImage.saveNamespace(true);
        }
        ……
    } catch(IOException e) {
        ……
    }
    ……
}
```

package org.apache.hadoop.hdfs.server.namenode;

FSImage.java

```
boolean recoverTransitionRead(Collection<URI> dataDirs, Collection<URI>

StartupOption startOpt){
    ……
    // dataDirs 为 FSImage 的保存路径，editsDirs 为 Edit logs 的保存路径
    // 将 dataDirs 和 editsDirs 中的路径保存到 storageDirs 中，并按三种类型存储
    setStorageDirectories(dataDirs, editsDirs);
    ……
    // 遍历 storageDirs 中的路径，加载最新的 image 和 edit logs 到内存，进行合并

    boolean needToSave = loadFSImage();

}
```

遍历 fsNameDirs 和 fsEditsDirs，将以下三种类型的路径加入到 storageDirs：

（1）既作为 FSImage 又作为 Edit logs 的路径。

（2）只作为 FSImage 的路径。

（3）只作为 Edit logs 的路径。

代码如下所示。

```
   package org.apache.hadoop.hdfs.server.namenode;
   FSImage.java
   void setStorageDirectories(Collection<URI> fsNameDirs, Collection<URI>
fsEditsDirs)
   throws IOException {
       // 创建 storageDirs
       this.storageDirs = new ArrayList<StorageDirectory>();
       this.removedStorageDirs = new ArrayList<StorageDirectory>();
       // 将所有的 fsNameDirs 加入到 storageDirs，并设置相应的类型，类型包括：
       // 1)NameNodeDirType.IMAGE_AND_EDITS，既作为 FSImage 又作为 Edit logs
       // 的路径；2) NameNodeDirType.IMAGE，只作为 FSImage 的路径


          ……
          for (URI editsDirName : fsEditsDirs) {
              if (editsDirName.compareTo(dirName) == 0) {
                  ……
                  fsEditsDirs.remove(editsDirName);
                  break;
              }
          }
          NameNodeDirType dirType = (isAlsoEdits) ?
              NameNodeDirType.IMAGE_AND_EDITS : NameNodeDirType.IMAGE;
          // Add to the list of storage directories, only if the
          // URI is of type file://
          if(dirName.getScheme().compareTo(JournalType.FILE.name().toLow
erCase())


                      this.addStorageDir(new StorageDirectory(new
```

```
File(dirName.getPath()),
                        dirType));
                }
        }
        // fsEditsDirs 中剩下的路径只作为 Edit logs 的路径，将其加入到 storageDirs
        for (URI dirName : fsEditsDirs) {
            ……
            if(dirName.getScheme().compareTo(JournalType.FILE.name().toLow
erCase())== 0)
                this.addStorageDir(new StorageDirectory(new
File(dirName.getPath()),
                    NameNodeDirType.EDITS));
        }
```

FSImage 的父类 Storage 中包含了一个 StorageDirectory 数组的成员变量 storageDirs，用于存储备份路径，同时还实现了一个基于 storageDirs 的 Iterator，用于对 storageDirs 进行操作。代码如下所示。

```
    package org.apache.hadoop.hdfs.server.common;
    Storage.java
    public abstract class Storage extends StorageInfo {
        ……
        protected List<StorageDirectory> storageDirs = new
ArrayList<StorageDirectory>();
        ……
        private class DirIterator implements Iterator<StorageDirectory> {
            // 实现了一个基于 storageDirs 的 Iterator
            ……
        }
        ……
```

```
        // 将路径加入到 storageDirs
        protected void addStorageDir(StorageDirectory sd) {
            storageDirs.add(sd);
        }
        ……
    }
```

package org.apache.hadoop.hdfs.server.namenode;

FSImage.java

```
    boolean loadFSImage() throws IOException {
        ……
        // 遍历所有的元数据保存路径，找出最新的 checkpoint 所在的路径，并判断当前路径下
        // 的元数据文件是否刚格式化，如果是，则在后面需要将内存中的元数据进行保存，保
        // 证一致性。如果 checkpoint 的时间无效，则也需要保存元数据
        for (Iterator<StorageDirectory> it = dirIterator(); it.hasNext();) {
            ……
        }
        // 对保存最新元数据文件的路径进行检查，确保 FSImage 和 Edit logs 在同一个目录
        // 且有效
        ……
        // 判断 checkpoint 时间的有效性，如果无效，也需要保存元数据
        needToSave |= checkpointTimes.size() != 1;

        // 如果存在被中断的 checkpoint，进行恢复
        needToSave |= recoverInterruptedCheckpoint(latestNameSD,
latestEditsSD);
        ……
        // 加载最新的 FSImage
        needToSave |= loadFSImage(getImageFile(latestNameSD,


        ……
```

```
    // 加载最新的 Edit logs
    if (latestNameCheckpointTime > latestEditsCheckpointTime)
        // the image is already current, discard edits
        needToSave |= true;
    else
        // latestNameCheckpointTime == latestEditsCheckpointTime
        needToSave |= (loadFSEdits(latestEditsSD) > 0);

    return needToSave;
}
```

recoverInterruptedCheckpoint 函数的主要功能是检查备份目录下元数据文件的状态，看是否有在 Checkpoint 时发生中断的情况，如果有则采取相应的措施，确保数据的一致性。通常来说一个 Checkpoint 的过程是：

（1）首先 Secondary NameNode(Checkpoint Node)会通知 NameNode 上产生一个新的 Edit log 文件 edits.new，之后所有的日志更新将会写入 edits.new 之中；

（2）接下来，Secondary NameNode(Checkpoint Node)会从 NameNode 下载 fsimage 和 edits 文件，进行合并，产生新的文件 fsimage.ckpt；

（3）然后，Secondary NameNode 会将 fsimage.ckpt 上传到 NameNode；

（4）最后，NameNode 会将 edits.new 重命名为 edits，将 fsimage.ckpt 重命名为 fsimage。

recoverInterruptedCheckpoint 通过检查备份目录下文件的状态，来判断当前目录所处 Checkpoint 的状态，由此做出相应的处理措施，具体代码如下。

```
package org.apache.hadoop.hdfs.server.namenode;

boolean recoverInterruptedCheckpoint(StorageDirectory nameSD,
```

```
StorageDirectory editsSD) throws IOException {
    boolean needToSave = false;
    File curFile = getImageFile(nameSD, NameNodeFile.IMAGE);
    File ckptFile = getImageFile(nameSD, NameNodeFile.IMAGE_NEW);
    if (ckptFile.exists()) {
        // 如果 fsimage.ckpt 存在，说明 checkpoint 没有成功，需要进行处理

        needToSave = true;
        if (getImageFile(editsSD, NameNodeFile.EDITS_NEW).exists()) {
            // 如果 edits.new 存在，说明此时只是上传了 fsimage.ckpt，但无法确定

            if (!ckptFile.delete()) {
                throw new IOException("Unable to delete " + ckptFile);
            }
        } else {
           // 如果 edits.new 不存在，说明 NameNode 已经将 edits.new 重命名为 edits，
           // 但还没来得及将 fsimage.ckpt 重命名为 fsimage，因此，可以将
           // fsimage.ckpt 重命名为 fsimage，来完成之前的 Checkpoint
            if (!ckptFile.renameTo(curFile)) {
                if (!curFile.delete())
                  LOG.warn("Unable to delete dir " + curFile + " before rename");
                if (!ckptFile.renameTo(curFile)) {
                  throw new IOException("Unable to rename " + ckptFile +" to "
                        + curFile);
                }
            }
        }

    return needToSave;
}
```

```
   package org.apache.hadoop.hdfs.server.namenode;
   FSImage.java
   void saveNamespace(boolean renewCheckpointTime) throws IOException {
      ……
      ArrayList<StorageDirectory> errorSDs = new
ArrayList<StorageDirectory>();
      // 1.将备份目录下的 current 目录重命名为 lastcheckpoint.tmp
      for (Iterator<StorageDirectory> it = dirIterator(); it.hasNext();) {
         StorageDirectory sd = it.next();
         try {
            // 将当前所有的存储目录下的 current 目录重命名为 lastcheckpoint.tmp，并
            // 重新创建这些目录，如果目录无法访问，则抛出异常如果异常导致操作阻塞，
            //那么将导致整个 saveNamespace 阻塞，后续操作无法进行
             moveCurrent(sd);
         } catch(IOException ie) {
             // 对于无法正常访问的目录，将其加入到 errorSDs，在最后进行处理
             LOG.error("Unable to move current for " + sd.getRoot(), ie);
             errorSDs.add(sd);
         }
      }
      ……
      // 2.在 Image 的备份目录下创建新的 Current 目录，将 Image 保存到该目录下，
      // 同样将无法正常访问的目录加入到 errorSDs
      ……
      // 3.在 Edit 备份目录中创建新的 Current 目录，在该目录下创建新的 Edits 文件，
      // 同样将无法正常访问的目录加入到 errorSDs
      ……
      // 4.将备份目录下的 lastcheckpoint.tmp 重命名为 previous.checkpoint，将无法正常


      ……
```

```
        // 5.处理无法正常访问的备份目录
        processIOError(errorSDs, false);
        ……
    }
```

3.1.2 元数据更新及日志写入情景分析

本节以客户端 Mkdir 操作为例，分析元数据的数据流过程，并对其中涉及到备份目录部分进行重点分析。

Mkdir 操作由客户端发起，具体实现是调用 DFSClient.java 中的 mkdirs 方法，mkdirs 又通过 RPC 远程调用 NameNode 所实现的 Mkdirs 接口。如下所示。

```
    package org.apache.hadoop.hdfs;
    DFSClient.java
    public boolean mkdirs(String src, FsPermission permission, boolean
createParent)
    throws IOException, UnresolvedLinkException {
        ……
        try {
            return namenode.mkdirs(src, masked, createParent);
        } catch(RemoteException re) {
            ……
        }
    }
```

NameNode 的 mkdirs 方法调用了类 FSNamesystem 的 mkdirs 方法。如下所示。

```
    package org.apache.hadoop.hdfs.server.namenode
    NameNode.java
    public boolean mkdirs(String src, FsPermission masked, boolean
createParent)
```

```
        ……
        return namesystem.mkdirs(src,
            new
            PermissionStatus(UserGroupInformation.getCurrentUser().getShor
tUserName(),
                null, masked), createParent);
    }
```

类 FSNamesystem 的 mkdirs 方法首先调用 mkdirsInternal，创建目录，然后调用 FSEditlog 的 logSync，将日志记录写入 Edits 文件。如下所示。

```
    package org.apache.hadoop.hdfs.server.namenode;
    FSNamesystem.java
    public boolean mkdirs(String src, PermissionStatus permissions, boolean
createParent)
    throws IOException, UnresolvedLinkException {

        boolean status = mkdirsInternal(src, permissions, createParent);
        getEditLog().logSync();
        ……
        return status;
    }
```

mkdirsInternal 方法调用 FSDirectory 的 mkdirs 方法来创建目录，并将日志记录写入相应的输出流。如下所示。

```
    package org.apache.hadoop.hdfs.server.namenode;
    FSNamesystem.java
    private synchronized boolean mkdirsInternal(String src, PermissionStatus
permissions,

        ……
```

```
        if (!dir.mkdirs(src, permissions, false, now())) {
            throw new IOException("Invalid directory name: " + src);
        }
        return true;
    }
```

FSDirectory 的 mkdirs 方法通过 synchronized 关键字对 rootDir 进行锁定，保证元数据操作原子性，mkdirs 在创建完目录后，调用 FSEditlog 的 logMkdir 方法，将 mkdir 的日志记录写入日志输出流中，此时日志记录还并未写入磁盘文件。如下所示。

```
    package org.apache.hadoop.hdfs.server.namenode;
    FSDirectory.java
    boolean mkdirs(String src, PermissionStatus permissions, boolean
inheritPermission, long now) throws FileAlreadyExistsException,
QuotaExceededException, UnresolvedLinkException {
        ……
        synchronized(rootDir) {
            ……
            // create directories beginning from the first null index
            for(; i < inodes.length; i++) {
                // 创建目录
                ……
                // 将日志记录写入输出流
                fsImage.getEditLog().logMkDir(cur, inodes[i]);
                ……
            }

        return true;
    }
    package org.apache.hadoop.hdfs.server.namenode;
```

```
FSEditLog.java
public void logMkDir(String path, INode newNode) {
    // 创建要写入的日志信息
    ……
    // 将信息写入输出流
    logEdit(OP_MKDIR, new ArrayWritable(DeprecatedUTF8.class, info),
        newNode.getPermissionStatus());
}
```

每个 Edit 备份目录下的 Edits 文件对应一个 EditLogFileOutputStream，它是 EditLogOutputStream 的子类，editStreams 是所有 EditLogFileOutputStream 的集合，logEdit 方法会将 Mkdir 的日志记录依次写入 editStreams 中的每个 EditLogFileOutputStream，对于无法正常访问的 EditLogFileOutputStream，将把它加入到 errorStreams 中，在最后统一处理。同样的道理，在将日志记录写入 EditLogOutputStream 的时候发生阻塞，将导致整个 Mkdir 操作阻塞。如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
FSEditLog.java
synchronized void logEdit(byte op, Writable ... writables) {
    if(getNumEditStreams() == 0)
        throw new
java.lang.IllegalStateException(NO_JOURNAL_STREAMS_WARNING);
    ArrayList<EditLogOutputStream> errorStreams = null;
    long start = FSNamesystem.now();
    for(EditLogOutputStream eStream : editStreams) {
        FSImage.LOG.debug("loggin edits into " + eStream.getName()  + "
stream");
        if(!eStream.isOperationSupported(op))
            continue;
        try {
```

```
        } catch (IOException ie) {
           FSImage.LOG.warn("logEdit: removing "+ eStream.getName(), ie);
            if(errorStreams == null)
                errorStreams = new ArrayList<EditLogOutputStream>(1);
                errorStreams.add(eStream);
        }
    }
    processIOError(errorStreams, true);
    recordTransaction(start);
}
```

所有的元数据操作最终都会调用 logEdit 方法写入日志记录，类 FSEditLog 中将每个写日志记录的动作称为一次事务（Transaction），用一个递增的变量 txid 来唯一标识此次事务，该 ID 保存在调用 logEdit 方法的线程的变量中，当该线程后续调用 logSync 时，使用该 ID 来确定自己之前写入的日志事务，并且通过与当前正在 Sync 的事务 ID 进行比较，确保 flush 到磁盘文件的日志记录的顺序性。

```
package org.apache.hadoop.hdfs.server.namenode;
FSEditLog.java
private void recordTransaction(long start) {
    // get a new transactionId
    txid++;
    // record the transactionId when new data was written to the edits log
    TransactionId id = myTransactionId.get();
    id.txid = txid;
    // update statistics
    long end = FSNamesystem.now();
    numTransactions++;
    totalTimeTransactions += (end-start);
    if (metrics != null)
        metrics.transactions.inc((end-start));
```

```
    }
    }
```

EditLogFileOutputStream 中设计了双缓冲 bufCurrent 和 bufReady，其中 bufCurrent 用于接收日志流的输入，而 bufReady 则用于将日志流输出到磁盘文件，双缓冲的设计可以使得日志的写入和输出可以并行完成，提高效率。EditLogFileOutputStream 的 write 方法会将数据写入 bufCurrent 缓冲中，但此时并不写入磁盘。如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
EditLogFileOutputStream.java
void write(byte op, Writable... writables) throws IOException {
   write(op);
   for (Writable w : writables) {
      w.write(bufCurrent);
   }
```

Mkdir 日志记录写入 EditLogFileOutputStream 后，并未写入到磁盘的 Edits 文件，沿调用返回至类 FSNamesystem 的 mkdirs 方法后，将调用 getEditLog().logSync()，最终将日志从 EditLogOutputStream 写入磁盘文件。

类 FSEditLog 的 logSync 方法分为 3 个主要步骤。

第 1 步

调用线程使用 synchronized(this)，获取对象锁，然后判断能否进行 sync，判断的条件是：当前线程的事务 ID(mytxid)小于正在处理的事务 ID(synctxid)，或者当前没有进行 sync(isSyncRunning)。

如果条件不满足，则说明正在处理之前的日志记录，那么线程调用 wait，释放对象锁，进入等待状态，直到其他线程完成 sync 后，调用 notifyAll 通知其结束等待，wait 还设置了超时时间，防止无限期地等待。

当条件满足后，如果当前线程的事务 ID(mytxid)小于正在处理的事务 ID(synctxid)，则说明该线程写入的日志记录已经处理完毕（多个线程都是调用 logEdit 方法，将日志记录写入 bufCurrent，随后的任何一次 logSync 调用都将会把 bufCurrent 中所有的日志记录 flush 到磁盘），因此无需再处理直接返回。

否则，向下执行 Sync，主要是设置 3 个标志：

- syncStart = txid;
- isSyncRunning = true;
- sync = true;

其中 syncStart 表示正在进行 Sync 的事务 ID，isSyncRunning = true 表示当前正在做 Sync，sync = true 表示 Sync 确实在进行。

接下来交换 bufCurrent 和 bufReady，这样原来的 bufReady 将作为 bufCurrent，接受新的日志写入，而原来的 bufCurrent 将作为 bufReady，其内容将在第 2 步中被 flush 至磁盘，退出 synchronized 代码段，释放对象锁。

第 2 步

具体执行 Sync，在第 2 步中并没有加锁，这样日志的写入和日志的 Sync 可以并行进行。由于在第 1 步将 isSyncRunning 已经置为 true，因此即便此时对象锁已经释放，也可以保证只有一个线程进入第 2 步。

第 2 步中，同样采用了一个 EditLogOutputStream 数组 errorStreams，将无法使用正常方法的 Stream 加入到该数组中，并在最后进行处理。Sync 的操作主要是调用各个 EditLogOutputStream 中的 flush 操作完成的，flush 方法将 bufReady 中的内容追加到该 EditLogOutputStream 对应的磁盘文件上。

第 3 步

将 synctxid 置为当前处理完毕的事务 ID，表示当前进度，然后置 isSyncRunning 为 false，表示 Sync 结束，然后调用 notifyAll 唤醒所有在该对象中等待的线程，当然，最后只有一个获得了对象锁的线程才能继续执行下去。

整个 logSync 方法采用了 try-finally 的结构，第 3 步处于 finally 部分，这样可以确保进入第 2 步的线程一定可以执行到第 3 步，防止在前面出现异常退出，而不能释放对象锁，最终造成死锁。

代码如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
FSEditLog.java
public void logSync() throws IOException {
    ……
    boolean sync = false;
    try {
        // 第 1 步
        synchronized (this) {
            ……
        }
        while (mytxid > synctxid && isSyncRunning) {
               try {
                   wait(1000);
               } catch (InterruptedException ie) {
               }
        }
        if (mytxid <= synctxid) {
            numTransactionsBatchedInSync++;
            if (metrics != null)
                metrics.transactionsBatchedInSync.inc();
            return;
        }
        // now, this thread will do the sync
        syncStart = txid;
```

```
            sync = true;
            for(EditLogOutputStream eStream : editStreams) {
                eStream.setReadyToFlush();
            }
            streams = editStreams.toArray(new
EditLogOutputStream[editStreams.size()]);
        }
            // 第 2 步
            long start = FSNamesystem.now();
            for (int idx = 0; idx < streams.length; idx++) {
                EditLogOutputStream eStream = streams[idx];
                try {
                    eStream.flush();
                } catch (IOException ie) {
                    if (errorStreams == null) {
                        errorStreams = new ArrayList<EditLogOutputStream>(1);
                    }
                    errorStreams.add(eStream);
                }
            }
            long elapsed = FSNamesystem.now() - start;
            processIOError(errorStreams, true);

           if (metrics != null) // Metrics non-null only when used inside name node
                metrics.syncs.inc(elapsed);
        } finally {
            // 第 3 步
            synchronized (this) {
                synctxid = syncStart;
```

```
                isSyncRunning = false;
            }
            this.notifyAll();
        }
    }
```

3.1.3　Checkpoint 过程情景分析

Checkpoint 将内存中最新的元数据以文件形式存储到各个备份目录之下，同时清除备份目录下原有的 fsiamge 文件和 edits 文件，这样可以定期的对 Fsiamge 文件和 Edits 文件进行合并，产生最新的 Fsiamge 文件，减少 NameNode 重启时的合并时间，同时又防止 Edits 的无限制增长。

Checkpoint 的功能可以由 Secondary NameNode、Checkpoint Node或 Backup Node 来完成，下面将以 Backup Node 为例，结合代码对 Checkpoint 的过程进行分析，重点关注其中涉及到备份目录的部分。整个 Checkpoint 的过程由 Backup Node 发起并结束，其中还通过 RPC 调用 NameNode 的部分接口，具体步骤描述如下。

（1）Backup Node RPC 调用 NameNode 的 startCheckPoint 方法，Name Node 遍历当前的日志输出流，并将其重新定位到其对应的备份目录下的新文件 edits.new 上，并创建一个指向 Backup Node 的输出流，用于向其发送日志记录，Backup Node 则会创建一个 journal spool，用于接收 Checkpoint 期间 NameNode 发送过来的日志记录。

（2）Backup Node 根据需要从 NameNode 下载最新的 Fsimage 和 Edits 文件。

（3）Backup Node 将最新的 Fsimage 和 Edits 文件加载到内存合并。

（4）Backup Node 将合并后的元数据保存到磁盘，并创建新 Edits。

（5）Backup Node 根据需要，通过 http 上传合并后的 Fsimage，Name Node 接收 Fsimage，将其命名为 Fsimage.ckpt，依次保存在各个备份目录下。

（6）Backup Node RPC 调用 NameNode 的 endCheckpoint 方法，NameNode 将所有备份目录下的 edits.new 重命名为 edits，将 Fsimage.ckpt 重命名为 Fsimage，并重定向输出流到 edits。

（7）Backup Node 将 journal spool 的日志合并到内存，并销毁 journal spool。

下面进行具体代码分析。

Backup Node 的 Checkpoint 由 doCheckpoint 方法完成。如下所示。

```
package org.apache.hadoop.hdfs.server.namenode;
Checkpointer.java
void doCheckpoint() throws IOException {
    long startTime = FSNamesystem.now();
    // RPC 调用 NameNode 的 startCheckpoint 方法

        getNamenode().startCheckpoint(backupNode.getRegistration());
    ……
    if(cpCmd.isImageObsolete()) {
        ……
        // 根据需要下载最新的 Fsimage 和 Edits。

    }
    BackupStorage bnImage = getFSImage();
    // 加载、合并 FSImage 和 Edits。

    sig.validateStorageInfo(bnImage);
    // 将内存中的元数据保存到磁盘

    // 根据需要上传最新的 FSImage
    if(cpCmd.needToReturnImage())
        uploadCheckpoint(sig);
```

```
        // RPC 调用 NameNode 的 endCheckpoint，结束 Checkpoint
        getNamenode().endCheckpoint(backupNode.getRegistration(), sig);
        // 将 journal spool 中的日志合并到内存，销毁 journal spool

        ……
    }
```

第一步

Backup Node RPC 调用 NameNode 的 startCheckpoint 方法，该方法完成以下几个任务：

（1）首先对 Checkpoint 进行验证，判断是否允许进行 Checkpoint；

（2）其次，决定 BackupNode 是否需要下载 FSImage 和 Edits；

（3）再次，决定 Backup Node 是否需要将合并后的 FSImage 文件上传回 NameNode；

（4）接下来，在所有 NameNode 的 Edits 备份目录下创建 edits.new 文件，关闭原有的文件输出流，创建指向 edits.new 的文件输出流，在 Checkpoint 过程中，所有的日志记录都将写入 edits.new 文件；

（5）接下来，创建一个指向 Backup Node 的输出流，同时在 Backup Node 端创建 journal spool，用于接收 Checkpoint 期间 NameNode 发送过来的日志记录。

```
    package org.apache.hadoop.hdfs.server.namenode;
    NameNode.java
    public NamenodeCommand startCheckpoint(NamenodeRegistration
registration)
    throws IOException {
        verifyRequest(registration);
        if(!isRole(NamenodeRole.ACTIVE))
```

```
            throw new IOException("Only an ACTIVE node can invoke
startCheckpoint.");
        return namesystem.startCheckpoint(registration, setRegistration());
    }
    package org.apache.hadoop.hdfs.server.namenode;
    FSNamesystem.java
    synchronized NamenodeCommand startCheckpoint(
        NamenodeRegistration bnReg, // backup node
        NamenodeRegistration nnReg) // active name-node
    throws IOException {
        ……
        NamenodeCommand cmd = getFSImage().startCheckpoint(bnReg, nnReg);
        // 日志记录写入 edits.new
        getEditLog().logSync();
        return cmd;
    }
    package org.apache.hadoop.hdfs.server.namenode;
    FSImage.java
    NamenodeCommand startCheckpoint(NamenodeRegistration bnReg, // backup node
    NamenodeRegistration nnReg) // active name-node
    throws IOException {
        // 验证是否允许进行 Checkpoint
        ……
        // 决定 Backup Node 是否需要下载 FSImage 和 Edits

        if(bnReg.getLayoutVersion() == this.getLayoutVersion()
            && bnReg.getCTime() == this.getCTime()
            && bnReg.getCheckpointTime() == this.checkpointTime)

        // 决定 Backup Node 是否上传合并后的 FSImage
```

```
        boolean needToReturnImg = true;
        // 如果没有配置备份目录，则 Backup Node 不需要上传
        if(getNumStorageDirs(NameNodeDirType.IMAGE) == 0)
            // do not return image if there are no image directories
            needToReturnImg = false;
        // 创建新的 edits.new，将备份目录的输出流重定向到 edits.new
        CheckpointSignature sig = rollEditLog();
        // 增加一个 EditLogBackupOutputStream 到 editStreams，用于向 Backup Node

        getEditLog().logJSpoolStart(bnReg, nnReg);
        return new CheckpointCommand(sig, isImgObsolete, needToReturnImg);
    }
    package org.apache.hadoop.hdfs.server.namenode;
    FSImage.java
    CheckpointSignature rollEditLog() throws IOException {
        getEditLog().rollEditLog();
        ckptState = CheckpointStates.ROLLED_EDITS;
        // If checkpoint fails this should be the most recent image, therefore
        incrementCheckpointTime();
        return new CheckpointSignature(this);
    }
    package org.apache.hadoop.hdfs.server.namenode;
    FSEditLog.java
    synchronized void rollEditLog() throws IOException {
        waitForSyncToFinish();
        Iterator<StorageDirectory> it =
fsimage.dirIterator(NameNodeDirType.EDITS);
        if(!it.hasNext())

        // 验证备份目录下 edits.new 的一致性，要么都存在，要么都不存在，否则抛出异常
```

```
        boolean alreadyExists = existsNew(it.next());
        while(it.hasNext()) {
            StorageDirectory sd = it.next();
            if(alreadyExists != existsNew(sd))
                throw new IOException(getEditNewFile(sd)
                + "should " + (alreadyExists ? "" : "not ") + "exist.");
        }
        // 如果 edits.new 存在，则不需要做任何处理
        if(alreadyExists)
            return;
        // 检查之前访问失败的备份目录是否可用，如果可以，把它加入回来
        fsimage.attemptRestoreRemovedStorage();
        // 将输出流重定向到 edits.new
        divertFileStreams(
            Storage.STORAGE_DIR_CURRENT + "/" +
NameNodeFile.EDITS_NEW.getName());
    }
```

类 FSEditLog 的 divertFileStreams 方法完成重定向输出流的功能，具体步骤是：

（1）检查当前输出流的路径与配置的存储备份目录是否一致；

（2）关闭原来的输出流；

（3）创建指向 edits.new 的输出流；

（4）以新创建的输出流替代当前输出流。

在对存储备份目录进行遍历的过程中，同样设置了一个 EditLogOutputStream 数组 errorStreams，将无法正常访问的备份目录对应的输出流加入到 errorStreams，并在最后进行错误处理。

```
package org.apache.hadoop.hdfs.server.namenode;
```

```
   synchronized void divertFileStreams(String dest) throws IOException {
       waitForSyncToFinish();
       assert getNumEditStreams() >= getNumEditsDirs() :"Inconsistent number
of streams";
       ArrayList<EditLogOutputStream> errorStreams = null;
       EditStreamIterator itE =
           (EditStreamIterator)getOutputStreamIterator(JournalType.FILE);
       Iterator<StorageDirectory> itD =
fsimage.dirIterator(NameNodeDirType.EDITS);
       while(itE.hasNext() && itD.hasNext()) {
           EditLogOutputStream eStream = itE.next();
           StorageDirectory sd = itD.next();
           if(!eStream.getName().startsWith(sd.getRoot().getPath()))
             throw new IOException("Inconsistent order of edit streams: " + eStream);
           try {
               // 关闭原来的 stream
               closeStream(eStream);
               // 创建一个新的 stream
              eStream = new EditLogFileOutputStream(new File(sd.getRoot(), dest),
                   sizeOutputFlushBuffer);
               eStream.create();
               // 使用新的 stream 替换原来的 stream
               itE.replace(eStream);
           } catch (IOException e) {
             FSNamesystem.LOG.warn("Error in editStream " + eStream.getName(), e);

                 errorStreams = new ArrayList<EditLogOutputStream>(1);
               errorStreams.add(eStream);

       }
```

```
        processIOError(errorStreams, true);
    }
```

logJSpoolStart 创建一个指向 Backup Node 的输出流 EditLogBackupOutputStream，它是 EditLogOutputStream 的子类，并且复写（override）了父类的相关接口。

至此，可以总结一下：

对于 NameNode 来说，它的所有日志记录都通过一组 EditLogOutputStream 进行输出，而在具体实例化的时候，这一组 EditLogOutputStream 中包括多个 EditLogFileOutputStream 和 1 个 EditLogBackupOutputStream：

- 每个 EditLogFileOutputStream 将日志记录最终输出到对应的备份目录下的 edits 文件或 edits.new 文件；
- 1 个 EditLogBackupOutputStream 对应的是注册上来的 Backup Node，EditLogBackupOutputStream 将日志通过网络发送到 Backup Node

> 对于 NameNode 来说，通过统一的 EditLogOutputStream 的接口进行操作，无须关心具体的实例对象是什么类型，屏蔽了差异，便于扩展，而对于 EditLogOutputStream 的实例来说，对父类接口的调用，通过多态可以直接转到相应子类的实现。

logJSpoolStart 最后调用 logEdit 方法，写入一条 OP_JSPOOL_START 的日志记录，其作用是通知 Backup Node 创建一个日志池（journal spool），该日志池可以用来缓存 NameNode 发送过来的日志记录，以供 Backup Node 后续处理。

> logEdit 只是将日志记录写入到输出流的缓存中，也就是 bufCurrent 中，此时还并未发送到 Backup Node，真正的发送需要等到后续调用 getEditLog().logSync()。

```
package org.apache.hadoop.hdfs.server.namenode;
FSEditLog.java
synchronized void logJSpoolStart(NamenodeRegistration bnReg, // backup node
```

```
throws IOException {
    // NameNode 上没有指向 checkpoint node 的输出流
    if(bnReg.isRole(NamenodeRole.CHECKPOINT))
        return;
    if(editStreams == null)
        editStreams = new ArrayList<EditLogOutputStream>();
    EditLogOutputStream boStream = null;
    // 查找是否有指向该 Backup Node 的输出流
    for(EditLogOutputStream eStream : editStreams) {
        if(eStream.getName().equals(bnReg.getAddress())) {
            boStream = eStream; // already there
            break;
        }
    }
    //  如果没有，则创建新的指向该 Backup Node 的输出流

        boStream = new EditLogBackupOutputStream(bnReg, nnReg);
        editStreams.add(boStream);
    }
    logEdit(OP_JSPOOL_START, (Writable[])null);
}
```

logEdit 方法遍历输出流，依次调用输出流的 write 方法，写入日志记录，由于 EditLogOutputStream 的子类复写了 write 方法，因此，实际上调用的是各个子类复写的 write 方法。

```
package org.apache.hadoop.hdfs.server.namenode;
FSEditLog.java
synchronized void logEdit(byte op, Writable ... writables) {


        throw new
```

```
java.lang.IllegalStateException(NO_JOURNAL_STREAMS_WARNING);
       ArrayList<EditLogOutputStream> errorStreams = null;
       long start = FSNamesystem.now();
       // 遍历所有的输出流
       for(EditLogOutputStream eStream : editStreams) {
         FSImage.LOG.debug("loggin edits into " + eStream.getName()  + " stream");


             continue;
          try {
             eStream.write(op, writables);
          } catch (IOException ie) {
             // 将访问异常的输出流加入到 errorStreams
             FSImage.LOG.warn("logEdit: removing "+ eStream.getName(), ie);
             if(errorStreams == null)
                errorStreams = new ArrayList<EditLogOutputStream>(1);
             errorStreams.add(eStream);
          }
       }
       // 处理所有访问异常的输出流
       processIOError(errorStreams, true);
       recordTransaction(start);
    }
```

EditLogBackupOutputStream 的子类复写了 write 方法，它将创建一条日志记录 JournalRecord，将其加入到 bufCurrent 缓存汇中。

```
package org.apache.hadoop.hdfs.server.namenode;
EditLogBackupOutputStream.java
void write(byte op, Writable ... writables) throws IOException {


}
```

沿调用依次返回至 FSNamesystem.java 3917，调用 NamenodeCommand cmd = getFSImage().startCheckpoint(bnReg,nnReg)结束后，将调用 getEditLog().logSync();，logSync 中也是遍历输出流，调用输出流的 flush 方法，在父类的 flush 方法中调用了 flushAndSync 方法，输出流的子类复写了 flushAndSync 方法，因此对父类接口 flushAndSync 的调用将直接转移至相应子类的实现，我们下面看下 logSync 中对 EditLogBackupOutputStream 的调用。

```
package org.apache.hadoop.hdfs.server.namenode;
FSEditLog.java
public void logSync() throws IOException {
    ……
    try {
        synchronized (this) {
            ……
        }
        ……
        // do the sync
        long start = FSNamesystem.now();
        // 遍历所有的输出流
        for (int idx = 0; idx < streams.length; idx++) {
            EditLogOutputStream eStream = streams[idx];
            try {
                eStream.flush();
            } catch (IOException ie) {
                // 输出流异常处理
                ……
                if (errorStreams == null) {
                    errorStreams = new ArrayList<EditLogOutputStream>(1);
                }
                errorStreams.add(eStream);
```

```
            }
        }
    } finally {
                ……
    }
}
```

package org.apache.hadoop.hdfs.server.namenode;

EditLogOutputStream.java

```
public void flush() throws IOException {
    ……
    // 子类复写了父类方法
    flushAndSync();
    long end = FSNamesystem.now();
    totalTimeSync += (end - start);
    }
```

EditLogBackupOutputStream 中复写了 flushAndSync 方法，它依次取出 bufReady 的每条日志记录 JournalRecord，对其解析，对于普通日志，将其记录内容写入 DataOutputBuffer 中，最后调用 send 一次发送；对于之前写入的 OP_JSPOOL_START 记录，由于它是一个特殊的操作，不与其他日志合并，可以单独将该日志将放入 DataOutputBuffer，进行一次 send 调用。

package org.apache.hadoop.hdfs.server.namenode;

EditLogBackupOutputStream.java

```
protected void flushAndSync() throws IOException {
    assert out.size() == 0 : "Output buffer is not empty";
    int bufReadySize = bufReady.size();
    for(int idx = 0; idx < bufReadySize; idx++) {
        JournalRecord jRec = null;
        for(; idx < bufReadySize; idx++) {
```

```
            // 普通的日志记录可以合并后一次发送
            // 特殊的日志操作需要单独发送
            if(jRec.op >= FSEditLog.OP_JSPOOL_START)
                break;
            jRec.write(out);
        }
        if(out.size() > 0)
            send(NamenodeProtocol.JA_JOURNAL);
        if(idx == bufReadySize)
            break;
        // OP_JSPOOL_START 操作单独发送
        jRec.write(out);
        send(jRec.op);
    }
    // 清除 buffer 中所有的数据
    bufReady.clear();
    // reset buffer 至初始位置
    out.reset();
}
```

EditLogBackupOutputStream send 方法通过 RPC 调用 Backup Node 的 journal 方法，Backup Node 的 journal 方法对不同的日志类型采用不同的处理，其中 JA_JSPOOL_START 将调用 bnImage.startJournalSpool(nnReg)方法。

```
package org.apache.hadoop.hdfs.server.namenode;
EditLogBackupOutputStream.java
private void send(int ja) throws IOException {
    try {
        int length = out.getLength();
        out.write(FSEditLog.OP_INVALID);
```

```
            backupNode.journal(nnRegistration, ja, length, out.getData());
        } finally {
            out.reset();
        }
    }
```

package org.apache.hadoop.hdfs.server.namenode;

BackupNode.java

```
    public void journal(NamenodeRegistration nnReg,int jAction,int
length,byte[] args) throws IOException {
        ……
        BackupStorage bnImage = (BackupStorage)getFSImage();
        // 根据不同的操作类型采取不同的处理
        switch(jAction) {
        case (int)JA_IS_ALIVE:
            return;
        case (int)JA_JOURNAL:
            bnImage.journal(length, args);
            return;
        case (int)JA_JSPOOL_START:
            bnImage.startJournalSpool(nnReg);
            return;
        case (int)JA_CHECKPOINT_TIME:
            bnImage.setCheckpointTime(length, args);
            setRegistration(); // keep registration up to date
            return;
        default:
            throw new IOException("Unexpected journal action: " + jAction);
         }
```

BackupStorage 的 startJournalSpool 方法将在 Backup Node 备份目录下创建 jspool 目录，并创建 edits.new 文件，将输出流指向 edits.new，后面的日志记录都将写入 edits.new。

至此，Backup Node 的 journal spool 已经建立起来，在整个 Checkpoint 过程中，都将会接收 NameNode 发送过来的日志记录，直到 Checkpoint 结束时，Backup Node 会将 journal spool 中的日志进行合并，然后销毁 journal spool。

```
   package org.apache.hadoop.hdfs.server.namenode;
   BackupStorage.java
   synchronized void startJournalSpool(NamenodeRegistration nnReg)throws
IOException {
       ……
       // 创建 journal spool 目录

dirIterator(NameNodeDirType.EDITS);
       it.hasNext();) {
           StorageDirectory sd = it.next();
           File jsDir = getJSpoolDir(sd);
           if (!jsDir.exists() && !jsDir.mkdirs()) {
              throw new IOException("Mkdirs failed to create " +
jsDir.getCanonicalPath());
           }
           // create edit file if missing
           File eFile = getEditFile(sd);
           if(!eFile.exists()) {
              editLog.createEditLogFile(eFile);
           }

       if(!editLog.isOpen())
           editLog.open();
```

```
        // 将输出流指向备份目录下的 jspool/edits.new 文件
        editLog.divertFileStreams(STORAGE_JSPOOL_DIR + "/" +
STORAGE_JSPOOL_FILE);
        setCheckpointState(CheckpointStates.ROLLED_EDITS);
        // 设置 spooling 输入流

            backupInputStream = new
EditLogBackupInputStream(nnReg.getAddress());
        jsState = JSpoolState.INPROGRESS;
    }
```

第二步

Backup Node 根据需要从 NameNode 下载最新的 fsimage 和 edits 文件。

具体的实现是 Checkpointer 的 downloadCheckpoint 方法，NameNode 与 Backup Node 之间文件的上传与下载是通过 http 协议进行传输的，在 NameNode 和 Backup Node 各自都启动了一个 httpServer，传输时由一方先发起请求，另一方相应请求并执行相应 java servlet 完成功能。

以 fsimage 文件的下载为例：

第 1 步在 Backup Node 方发起请求下载文件，具体实现是由 TransferFsImage 的 getFileClient 方法实现的，传入的参数有：

- nnHttpAddr 表示 NameNode 的 http 地址
- fileid 表示传入的参数
- getimage=1 表示下载
- flies 表示下载的 fsimage 保存在本地的路径

```
package org.apache.hadoop.hdfs.server.namenode;
Checkpointer.java
private void downloadCheckpoint(CheckpointSignature sig) throws IOException {
```

```
        // 下载 FSImage 文件
        String fileid = "getimage=1";
        Collection<File> list = getFSImage().getFiles(NameNodeFile.IMAGE,
            NameNodeDirType.IMAGE);
        File[] files = list.toArray(new File[list.size()]);
        assert files.length > 0 : "No checkpoint targets.";
        String nnHttpAddr = backupNode.nnHttpAddress;
        TransferFsImage.getFileClient(nnHttpAddr, fileid, files);
        LOG.info("Downloaded file " + files[0].getName() + " size " +
            files[0].length() + " bytes.");
        // 下载 Edits 文件
        fileid = "getedit=1";
        list = getFSImage().getFiles(NameNodeFile.EDITS,
NameNodeDirType.EDITS);
        files = list.toArray(new File[list.size()]);
        assert files.length > 0 : "No checkpoint targets.";
        TransferFsImage.getFileClient(nnHttpAddr, fileid, files);
        LOG.info("Downloaded file " + files[0].getName() + " size " +
            files[0].length() + " bytes.");
    }
```

getFileClient 方法首先根据传入的参数 fsName，形成 url 字符串，与 NameNode 的 httpServer 建立连接，请求 NameNode 的 httpServer 执行名字为 getimage 的 javaservlet，传入的参数为 getimage=1。

第 2 步，NameNode 响应请求，发送数据。具体实现是 NameNode 的 httpServer 接收到请求后进行解析，执行名字为 getimage 的 javaservlet GetImageServlet，GetImageServlet 是 HttpServlet 的子类，doGet 处理 get 请求，在 doGet 方法中，对输入的参数进行判断，对于 FSImage 下载请求，则调用 TransferFsImage.getFileServer()方法，向 Backup Node 发送 FSImage 文件。

getFileServer 方法的输入参数有 OutputStream outstream、File localfile，其中 localfile 表示的是 FSImage 的路径，getFileServer 先将 localfile 读入 FileInputStream infile，然后通过 OutputStream outstream 发送给 Backup Node。

第 3 步，Backup Node 接收数据保存文件。Backup Node 根据 localPath 打开相应的 FileOutputStream，设置一个 InputStream stream 用来读取 NameNode 的 httpServer 发送过来的数据流，将数据流写入 FileOutputStream，形成相应的 Fsimage 文件。

```
   package org.apache.hadoop.hdfs.server.namenode;
   TransferFsImage.java
   static void getFileClient(String fsName, String id, File[] localPath)
   throws IOException {
       // 形成 url 字符串
       byte[] buf = new byte[BUFFER_SIZE];
       StringBuilder str = new
StringBuilder("http://"+fsName+"/getimage?");
       str.append(id);
       URL url = new URL(str.toString());
       // 与 server 建立连接，发送 url 请求
       URLConnection connection = url.openConnection();
       long advertisedSize;
       String contentLength = connection.getHeaderField(CONTENT_LENGTH);
       if (contentLength != null) {
           advertisedSize = Long.parseLong(contentLength);
       } else {
          throw new IOException(CONTENT_LENGTH + " header is not provided " +
              "by the namenode when trying to fetch " + str);
       }
       long received = 0;
```

```
FileOutputStream[] output = null;
try {
    if (localPath != null) {
        // 如果本地路径不为空，则创建输出流，用于保存本地数据
        output = new FileOutputStream[localPath.length];
        for (int i = 0; i < output.length; i++) {
            output[i] = new FileOutputStream(localPath[i]);
        }
    }
    int num = 1;
    while (num > 0) {
        num = stream.read(buf);
        if (num > 0 && localPath != null) {
            received += num;
            for (int i = 0; i < output.length; i++) {
                output[i].write(buf, 0, num);
            }
        }
    }
} finally {
    // 不管前面是否出现异常，都会执行到这里
    stream.close();
    if (output != null) {
        for (int i = 0; i < output.length; i++) {
            if (output[i] != null) {
                output[i].close();
            }
        }
    }
```

```
            throw new IOException("File " + str + " received length " + received +
              " is not of the advertised size " + advertisedSize);
          }
        }
        }
```

package org.apache.hadoop.hdfs.server.namenode;

GetImageServlet.java

```
public void doGet(HttpServletRequest request,HttpServletResponse
response
) throws ServletException, IOException {
    Map<String,String[]> pmap = request.getParameterMap();
    try {
        ServletContext context = getServletContext();
        FSImage nnImage =
(FSImage)context.getAttribute("name.system.image");
        TransferFsImage ff = new TransferFsImage(pmap, request, response);

            response.setHeader(TransferFsImage.CONTENT_LENGTH,
                String.valueOf(nnImage.getFsImageName().length()));
            // 发送 fsImage
            TransferFsImage.getFileServer(response.getOutputStream(),
                nnImage.getFsImageName());
        } else if (ff.getEdit()) {
            response.setHeader(TransferFsImage.CONTENT_LENGTH,String.
                valueOf(nnImage.getFsEditName().length()));
            // 发送 edits
            TransferFsImage.getFileServer(response.getOutputStream(),
                nnImage.getFsEditName());

            // 发送一个 HTTP get 请求来下载 Fsimage
```

```
            nnImage.validateCheckpointUpload(ff.getToken());
           TransferFsImage.getFileClient(ff.getInfoServer(), "getimage=1",
               nnImage.getFsImageNameCheckpoint());
            nnImage.checkpointUploadDone();
         }
      } catch (Exception ie) {
         String errMsg = "GetImage failed. " +
StringUtils.stringifyException(ie);
         response.sendError(HttpServletResponse.SC_GONE, errMsg);
         throw new IOException(errMsg);
      } finally {
         response.getOutputStream().close();
      }
   }
```

package org.apache.hadoop.hdfs.server.namenode;

TransferFsImage.java

```
static void getFileServer(OutputStream outstream, File localfile)
throws IOException {
   byte buf[] = new byte[BUFFER_SIZE];
   FileInputStream infile = null;
   try {
      infile = new FileInputStream(localfile);
      ……
      int num = 1;
      // 读入本地文件进行发送
      while (num > 0) {
         num = infile.read(buf);
         if (num <= 0) {

         }
```

```
            outstream.write(buf, 0, num);
        }
    } finally {
        if (infile != null) {
            infile.close();
        }
    }
}
```

第三步

Backup Node 将最新的 Fsimage 和 Edits 文件加载到内存合并。

对于 Backup Node 来说，如果是第 1 次启动，此时内存中的 NameSpace 为空，此时需要加载 Fsimage 和 Edits 文件加载到内存合并，如果是启动之后进行的 Checkpoint，Backup Node 的内存中已经形成 NameSpace，因而不需要再从磁盘加载。对于 Checkpoint Node，由于它并不在内存中维护 NameSpace，因此每次 Checkpoint 都需要加载。

下面我们假设 Backup Node 是第 1 次启动，分析其加载 Fsimage 和 Edits 文件到内存进行合并的过程。

```
package org.apache.hadoop.hdfs.server.namenode;
BackupStorage.java
void loadCheckpoint(CheckpointSignature sig) throws IOException {
    // 如果内存中没有 NameSpace，则从本地加载
    if(!editLog.isOpen())
        editLog.open();
    FSDirectory fsDir = getFSNamesystem().dir;


        Iterator<StorageDirectory> itImage =
dirIterator(NameNodeDirType.IMAGE);
```

```
          Iterator<StorageDirectory> itEdits =
dirIterator(NameNodeDirType.EDITS);
          if(!itImage.hasNext() || ! itEdits.hasNext())
             throw new IOException("Could not locate checkpoint
directories");
          StorageDirectory sdName = itImage.next();
          StorageDirectory sdEdits = itEdits.next();
          synchronized(getFSDirectoryRootLock()) {
             // 加载 fsimage

NameNodeFile.IMAGE));
          }
          // 加载 edits
          loadFSEdits(sdEdits);
       }
       // set storage fields
       setStorageInfo(sig);

    }
```

loadCheckpoint 方法通过类 FSDirectory 的 isEmpty 判断内存中的 NameSpace 是否为空，

isEmpty 方法调用 isDirEmpty 方法，传入参数“/”即根目录，isDirEmpty 方法首先判断传入的参数“/”是否为目录，如果是，再判断该目录是否为空。对于 Backup Node 第 1 次初始化，“/”下面没有加入 INode，为空，需要加载，之后“/”下面就不为空了。

```
   package org.apache.hadoop.hdfs.server.namenode;
   FSDirectory.java
   boolean isEmpty() {
```

```
    try {
        // 判断内存中根目录下是否为空
        return isDirEmpty("/");
    } catch (UnresolvedLinkException e) {
        NameNode.stateChangeLog.debug("/ cannot be a symlink");
        assert false : "/ cannot be a symlink";
        return true;
    }
}
```

FSDirectory.java

```
boolean isDirEmpty(String src) throws UnresolvedLinkException {
    boolean dirNotEmpty = true;
    // 首先判断输入参数类型是否是目录
    if (!isDir(src)) {
        return true;
    }
    synchronized(rootDir) {
        INode targetNode = rootDir.getNode(src, false);
        assert targetNode != null : "should be taken care in isDir() above";

        if (((INodeDirectory)targetNode).getChildren().size() != 0) {
            dirNotEmpty = false;
        }
    }
    return dirNotEmpty;
}
```

FSDirectory.java

```
boolean isDir(String src) throws UnresolvedLinkException {

        INode node = rootDir.getNode(normalizePath(src), false);
```

```
            return node != null && node.isDirectory();
        }
    }
```

从磁盘加载 Fsimage 文件。

```
    package org.apache.hadoop.hdfs.server.namenode;
    FSImage.java
    boolean loadFSImage(File curFile) throws IOException {
        assert this.getLayoutVersion() < 0 : "Negative layout version is
expected.";
        assert curFile != null : "curFile is null";
        FSNamesystem fsNamesys = getFSNamesystem();
        FSDirectory fsDir = fsNamesys.dir;
        boolean needToSave = true;
        DataInputStream in = new DataInputStream
            (new BufferedInputStream(new FileInputStream(curFile)));
        try {
            // read image version: first appeared in version -1
            int imgVersion = in.readInt();
            // read namespaceID: first appeared in version -2
            this.namespaceID = in.readInt();
            // read number of files
            long numFiles;
            if (imgVersion <= -16) {
                numFiles = in.readLong();
            } else {
                numFiles = in.readInt();
            }
            this.layoutVersion = imgVersion;
```

```
if (imgVersion <= -12) {
    long genstamp = in.readLong();
    fsNamesys.setGenerationStamp(genstamp);
}
needToSave = (imgVersion != FSConstants.LAYOUT_VERSION);
// read file info
short replication = fsNamesys.getDefaultReplication();
LOG.info("Number of files = " + numFiles);
String path;
String parentPath = "";
INodeDirectory parentINode = fsDir.rootDir;
……
for (long i = 0; i < numFiles; i++) {
    long modificationTime = 0;
    long atime = 0;
    long blockSize = 0;
    path = readString(in);
    replication = in.readShort();
    replication = editLog.adjustReplication(replication);
    modificationTime = in.readLong();
    if (imgVersion <= -17) {
        atime = in.readLong();
    }
    if (imgVersion <= -8) {
        blockSize = in.readLong();
    }
    int numBlocks = in.readInt();
    Block blocks[] = null;
    // for older versions, a blocklist of size 0
```

```
if ((-9 <= imgVersion && numBlocks > 0) ||
    (imgVersion < -9 && numBlocks >= 0)) {
    blocks = new Block[numBlocks];
    for (int j = 0; j < numBlocks; j++) {
        blocks[j] = new Block();
        if (-14 < imgVersion) {
            blocks[j].set(in.readLong(), in.readLong(),
            GenerationStamp.GRANDFATHER_GENERATION_STAMP);
        } else {
            blocks[j].readFields(in);
        }
    }
}
if (-8 <= imgVersion && blockSize == 0) {
    if (numBlocks > 1) {
        blockSize = blocks[0].getNumBytes();
    } else {
     long first = ((numBlocks == 1) ? blocks[0].getNumBytes(): 0);
     blockSize = Math.max(fsNamesys.getDefaultBlockSize(), first);
    }
    long nsQuota = -1L;
  if (imgVersion <= -16 && blocks == null  && numBlocks == -1) {
        nsQuota = in.readLong();
    long dsQuota = -1L;
  if (imgVersion <= -18 && blocks == null && numBlocks == -1) {
        dsQuota = in.readLong();
    }
    // Read the symlink only when the node is a symlink
    String symlink = "";
```

```
                    symlink = Text.readString(in);
                }
                PermissionStatus permissions =
fsNamesys.getUpgradePermission();
                if (imgVersion <= -11) {
                    permissions = PermissionStatus.read(in);
                }
                if (path.length() == 0) {
                    // it is the root
                    // update the root's attributes
                    if (nsQuota != -1 || dsQuota != -1) {
                        fsDir.rootDir.setQuota(nsQuota, dsQuota);
                    }
                    fsDir.rootDir.setModificationTime(modificationTime);
                    fsDir.rootDir.setPermissionStatus(permissions);
                    continue;
                }
                // check if the new inode belongs to the same parent
                if(!isParent(path, parentPath)) {
                    parentINode = null;
                    parentPath = getParent(path);
                }
                // add new inode
                parentINode =
fsDir.addToParent(path,parentINode,permissions,
                    blocks,symlink,replication,modificationTime, atime,
nsQuota, dsQuota,
                    blockSize);
            } // for end
```

```
            this.loadDatanodes(imgVersion, in);
            // load Files Under Construction
            this.loadFilesUnderConstruction(imgVersion, in, fsNamesys);
            this.loadSecretManagerState(imgVersion, in, fsNamesys);
        } finally {
            in.close();
        }

    }
```

第四步

Backup Node 将合并后的元数据保存到磁盘，并创建空的 Edits。代码略。

第五步

Backup Node 根据需要，通过 http 上传合并后的 Fsimage，Name Node 接收 Fsimage，将其命名为 Fsimage.ckpt，依次保存在各个备份目录下。如果 NameNode 上配置了备份目录，则需要上传，具体步骤如下，源码分析请参考前面文件下载部分的源码。

（1）Backup Node 通知 NameNode 有上传请求，具体实现是 Backup Node 的 uploadCheckpoint 调用 TransferFsImage.getFileClient 与 NameNode 建立连接，请求 getimage servlet，并传入参数 putimage=1。

（2）NameNode 响应请求，向 Backup Node 发出下载文件请求。具体实现是 NameNode 的 httpsServer 响应请求，运行 getimage 的 servlet GetImage Servlet，解析输入参数为 putimage，运行 TransferFsImage 的 getFile Client 与 BackupNode 建立连接，请求 getimage servlet，传入参数 getimage，然后等待回应。

（3）Backup Node 发送文件。具体实现是 Backup Node 的 httpServer 响应请求，调用 getimage 的 servlet GetImageServlet，GetImageServlet 对参数

进行解析，得知是 getimage，因此运行 TransferFsImage 的 getFileServer 函数，读取本地的 Fsimage 文件，然后写 outstream 并发送到 NameNode。

（4）NameNode 接收并保存文件到备份目录。具体实现是 NameNode 将输入流接收的数据写入 getFsImageNameCheckpoint 所命名的目录，getFsImageNameCheckpoint 会加入所有的备份目录，并将文件名命名为 FSImage.ckpt。也就是说，NameNode 在接收 FSImage 的时候，会将接收到的 FSImage 依次写入各个备份目录，当然在写入的时候有可能不成功，但是没有关系，因为写入的文件名为 FSImage.ckpt，在后续的过程中可根据文件名来判断 Checkpoint 所在的目录，然后进行相应的恢复动作。

第六步

Backup Node RPC 调用 NameNode 的 endCheckpoint 方法，NameNode 将所有备份目录下的 edits.new 重命名为 edits，将 FSImage.ckpt 重命名为 FSImage，并重定向输出流到 edits。

```
   package org.apache.hadoop.hdfs.server.namenode;
   NameNode.java
   public void endCheckpoint(NamenodeRegistration
registration,CheckpointSignature sig) throws IOException {
       verifyRequest(registration);
       if(!isRole(NamenodeRole.ACTIVE))
          throw new IOException("Only an ACTIVE node can invoke
endCheckpoint.");
       namesystem.endCheckpoint(registration, sig);
   }
   package org.apache.hadoop.hdfs.server.namenode;
   FSNamesystem.java
   synchronized void endCheckpoint(NamenodeRegistration registration,
```

```
        LOG.info("End checkpoint for " + registration.getAddress());
        getFSImage().endCheckpoint(sig, registration.getRole());
    }
    package org.apache.hadoop.hdfs.server.namenode;
    FSImage.java
    void endCheckpoint(CheckpointSignature sig, NamenodeRole remoteNNRole)
    throws IOException {
        sig.validateStorageInfo(this);
        boolean renewCheckpointTime =
remoteNNRole.equals(NamenodeRole.CHECKPOINT);
        rollFSImage(renewCheckpointTime);
    }
    void rollFSImage() throws IOException {
        rollFSImage(true);
    }
    void rollFSImage(boolean renewCheckpointTime) throws IOException {
        // 状态检查
        if (ckptState != CheckpointStates.UPLOAD_DONE
        && !(ckptState == CheckpointStates.ROLLED_EDITS
        && getNumStorageDirs(NameNodeDirType.IMAGE) == 0)) {
    throw new IOException("Cannot roll fsImage before rolling edits log.");
        }
        // 遍历备份目录，检查 FSImage.ckpt 文件是否存在

dirIterator(NameNodeDirType.IMAGE);
        it.hasNext();) {
            StorageDirectory sd = it.next();

            if (!ckpt.exists()) {
    throw new IOException("Checkpoint file " + ckpt +" does not exist");
```

```
            }
        }
        //遍历存储备份目录，将 edits.new 重命名为 edits 文件，并重定向输出
        editLog.purgeEditLog();
        LOG.debug("rollFSImage after purgeEditLog: storageList=" +
listStorageDirectories());
        // 遍历存储备份目录，将 FSImage.ckpt 文件重命名为 FSImage

        resetVersion(renewCheckpointTime);
    }
    package org.apache.hadoop.hdfs.server.namenode;
    FSEditLog.java
    synchronized void purgeEditLog() throws IOException {
        waitForSyncToFinish();
        revertFileStreams(
            Storage.STORAGE_DIR_CURRENT + "/" +
NameNodeFile.EDITS_NEW.getName());
    }
    synchronized void revertFileStreams(String source) throws IOException {
        ……
        // 遍历所有的 Edit 存储目录和输出流
        while(itE.hasNext() && itD.hasNext()) {
            EditLogOutputStream eStream = itE.next();
            StorageDirectory sd = itD.next();
            ……
            try {
                // 关闭原来的输出流
                closeStream(eStream);

                File editFile = getEditFile(sd);
```

```
            File prevEditFile = new File(sd.getRoot(), source);
            if(prevEditFile.exists()) {
               if(!prevEditFile.renameTo(editFile)) {
                  if(!editFile.delete()
|| !prevEditFile.renameTo(editFile)) {
                     throw new IOException("Rename failed for " +
                     sd.getRoot());
                  }
               }
            }
            // 打开新的输出流
            eStream = new EditLogFileOutputStream(editFile,
sizeOutputFlushBuffer);
            // 替换成新的输出流
            itE.replace(eStream);
         } catch (IOException e) {
            ……
         }
      }
      processIOError(errorStreams, true);
   }
   package org.apache.hadoop.hdfs.server.namenode;
   FSImage.java
   void renameCheckpoint() {
      ArrayList<StorageDirectory> al = null;
      // 遍历所有的 FSImage 存储目录，将 FSImage.ckpt 重命名为 FSImage

dirIterator(NameNodeDirType.IMAGE);

         StorageDirectory sd = it.next();
```

```
        File ckpt = getImageFile(sd, NameNodeFile.IMAGE_NEW);
        File curFile = getImageFile(sd, NameNodeFile.IMAGE);
        if (!ckpt.renameTo(curFile)) {
            if (!curFile.delete() || !ckpt.renameTo(curFile)) {
                LOG.warn("renaming  " + ckpt.getAbsolutePath() + " to "  +

                if(al == null) al = new ArrayList<StorageDirectory> (1);
                al.add(sd);
             }
        }

    if(al != null) processIOError(al, true);
}
```

第七步

Backup Node 将 journal spool 的日志合并到内存，并销毁 journal spool。

（1）首先查找 Edits 的备份目录下的 jspool 目录下 edits.new 是否存在，如果存在，则将 edits.new 中的内容读入到内存进行合并，并将 journal spool 的状态置为 wait，此后，所有 journal writer 将阻塞；

（2）接下来调用 FSEdit 的 revertFileStreams，关闭原来指向 edits.new 的输出流，将 jspool/edits.new 重命名为 current/edits 文件，即销毁 journal spool，创建新的输出流指向 edits 文件，后续的日志记录将写入 edits 文件，revertFileStreams 采用了 synchronized 关键字，FSEdit 其他 synchronized 修饰的日志操作将会被阻塞，从而有效保证日志写入的一致性，此外，将 jspool/edits.new 重命名，可以有效防止从第 1 次读入 jspool/edits.new 后到当前这一段时间内写入的日志记录丢失；

（3）最后，将 journal spool 的状态置为 off，并唤醒所有等待的 journal writer。

```
   package org.apache.hadoop.hdfs.server.namenode;
   BackupStorage.java
   void convergeJournalSpool() throws IOException {
      Iterator<StorageDirectory> itEdits =
dirIterator(NameNodeDirType.EDITS);
      if(! itEdits.hasNext())
         throw new IOException("Could not locate checkpoint directories");
      StorageDirectory sdEdits = itEdits.next();
      int numEdits = 0;
      File jSpoolFile = getJSpoolFile(sdEdits);
      long startTime = FSNamesystem.now();
      if(jSpoolFile.exists()) {
         // 加载 edits.new
         EditLogFileInputStream edits = new
EditLogFileInputStream(jSpoolFile);
         DataInputStream in = edits.getDataInputStream();
         numEdits += editLog.loadFSEdits(in, false);
         // first time reached the end of spool
         jsState = JSpoolState.WAIT;
         numEdits += editLog.loadEditRecords(getLayoutVersion(), in, true);
         getFSNamesystem().dir.updateCountForINodeWithQuota();
         edits.close();
      }
      // rename spool edits.new to edits making it in sync with the active node
      // subsequent journal records will go directly to edits
      editLog.revertFileStreams(STORAGE_JSPOOL_DIR + "/" +
STORAGE_JSPOOL_FILE);
      // write version file
      resetVersion(false);
```

```
        synchronized(this) {
            jsState = JSpoolState.OFF;
            notifyAll();
        }
        // 将 lastcheckpoint.tmp 目录重命名为 previous.checkpoint

            moveLastCheckpoint(sd);
        }
    }
    package org.apache.hadoop.hdfs.server.namenode;
    FSEditLog.java
    synchronized void revertFileStreams(String source) throws IOException{
        waitForSyncToFinish();
        assert getNumEditStreams() >= getNumEditsDirs() :
            "Inconsistent number of streams";
        ArrayList<EditLogOutputStream> errorStreams = null;
        EditStreamIterator itE =
            (EditStreamIterator)getOutputStreamIterator(JournalType.FILE);
        Iterator<StorageDirectory> itD =
fsimage.dirIterator(NameNodeDirType.EDITS);
        while(itE.hasNext() && itD.hasNext()) {
            EditLogOutputStream eStream = itE.next();
            StorageDirectory sd = itD.next();
            if(!eStream.getName().startsWith(sd.getRoot().getPath()))
               throw new IOException("Inconsistent order of edit streams: " +
                eStream + " does not start with " + sd.getRoot().getPath());
            try {
                // close old stream

                // rename edits.new to edits
```

```
                File editFile = getEditFile(sd);
                File prevEditFile = new File(sd.getRoot(), source);
                if(prevEditFile.exists()) {
                    if(!prevEditFile.renameTo(editFile)) {
                        ……
                    }
                }
                // open new stream
                eStream = new EditLogFileOutputStream(editFile,
sizeOutputFlushBuffer);
                // replace by the new stream
                itE.replace(eStream);
            } catch (IOException e) {
                ……
            }

        processIOError(errorStreams, true);
    }
```

3.1.4　元数据可靠性机制

Hadoop 的元数据备份方案以及 HDFS 的自身机制从多个方面保证了元数据的可靠性：

- 首先，可以配置多个备份路径，NameNode 在更新日志或进行 doCheckpoint 过程中，会将元数据文件保存到所有配置的备份目录，这样即便是某个备份路径对应的存储出现问题，也不会导致整个 HDFS 彻底无法工作；
- 其次对于每一个需要保存的元数据文件，都创建一个输出流，对访问过程中出现异常的输出流进行处理，将其移出，并在后续合适的时机再次检查移出的输出流是否恢复正常，这样可以有效保证备份过程中出现异常的输

出流不影响其他输出流的正常操作；

> 注意：如果输出流异常造成调用操作无法返回时，将导致整个输出流操作阻塞。

- 此外，还采用了多种机制来保证元数据存储的可靠性，例如在 Checkpoint 保存元数据时，将整个过程分为几个阶段，通过不同的文件名来标识当前所处的状态，为存储失败后进行恢复提供了可能。

3.1.5 元数据一致性机制

Hadoop 的元数据备份方案以及 HDFS 的自身机制从多个方面保证了元数据的一致性：

- 首先从 NameNode 启动时，对每个备份目录是否格式化、目录下的元数据文件名是否正确等进行检查，确保元数据文件间状态的一致性，然后选取最新的元数据加载到内存，这样可以确保 HDFS 当前状态与最后一次关闭时状态的一致性；
- 其次，通过对异常输出流的处理，可以确保正常输出流之间数据的一致性；
- 此外，运用了同步机制确保数据写入的一致性。

3.2 使用说明

Hadoop 的元数据备份使用简单，在 hdfs-default.xml 中有默认的配置选项，如下所示。dfs.namenode.name.dir 用来配置 Fsimage 的备份路径，路径间用“，”进行分隔，dfs.namenode.edits.dir 用来配置日志文件 Edits 的备份路径，路径间用“，”进行分隔。在使用时，可以将 hdfs-default.xml 中默认的配置选项复制到 hdfs-site.xml 进行相应的配置，在配置时通常配置两个路径，一个为本地路径，另一个为远程路径，如 NFS 共享路径。

Hadoop 的元数据备份机制通常与 Secondary NameNode，Checkpoint Node 或 BackupNode 结合使用，在后续章节中将详细介绍。

```
   <property>
      <name>dfs.namenode.name.dir</name>
      <value>配置 FSImage 的备份路径，用"，"进行分隔 </value>

           should store the name table(fsimage).  If this is a
comma-delimited list
         of directories then the name table is replicated in all of the
         directories, for redundancy.
      </description>
   </property>
   <property>
      <name>dfs.namenode.edits.dir</name>
      <value>配置 Edits 的备份路径，用"，"进行分隔</value>
      <description>Determines where on the local filesystem the DFS name node
        should store the transaction (edits) file. If this is a
comma-delimited list
        of directories then the transaction file is replicated in all of the

      </description>
   </property>
```

读书笔记

第 4 章　Hadoop 的 Backup Node 方案

Hadoop 的 Backup Node 方案的最终目标是为 HDFS 提供 NameNode 的热备节点，减少服务恢复时间。其主要的实现原理是：通过同步更新机制，在 Backup Node 节点中保存一份与 NameNode 完全一致的内存镜像，并且当 NameNode 无法提供服务时，能够自动接替，对外提供服务。

4.1 Backup Node 概述

Backup Node 目前正处于开发之中，按照开发计划， Backup Node 需要经历 3 个阶段[1]，才能最终成为 NameNode 的热备节点。

第 1 阶段

替换 Secondary Name Node。完成的主要功能是：定期将名字空间元数据内容导出并保存到磁盘、并且定期使 NameNode 裁减日志的大小，限制其无限制增长。此阶段的 Backup Node 称之为 Checkpoint Node。

第 2 阶段

改进 Checkpoint node 的 Checkpoint 机制，在 Backup Node 的内存中保存一份和 NameNode 元数据完全一致的镜像，当元数据发生变化时，NameNode 进行同步更新，此时 Backup Node 可直接利用自身的镜像进行 Checkpoint，无需再从 NameNode 进行下载。此阶段的 Backup Node 就称之为 Backup Node。

Backup Node 相对于 Checkpoint Node 的优势有两点：

首先是 Checkpoint 的效率更高，其次是元数据同步更新，恢复时可以保证与最新的元数据一致。

> 在后面的表述中，Backup Node 泛指广义上的 Backup Node，而第 2 阶段的 Backup Node 则会注明为“第 2 阶段 Backup Node”。

第 3 阶段

对第 2 阶段 Backup Node 的功能进行扩展，使之能够在 NameNode 无法正常服务后，接替 NameNode 的所有工作，对外正常服务。此阶段的 Backup Node 称之为 Standby Node。

> 目前 Hadoop 的 0.21.0 版本代码已经实现了上述第 1 、 2 个阶段。

[1] https://issues.apache.org/jira/secure/attachment/12400631/StreamEditsToBN.pdf

4.1.1　系统架构

使用 Backup Node 的 HDFS 系统架构如图 4.1 所示，与传统的 HDFS 不同的是增加了一个节点作为 Backup Node。

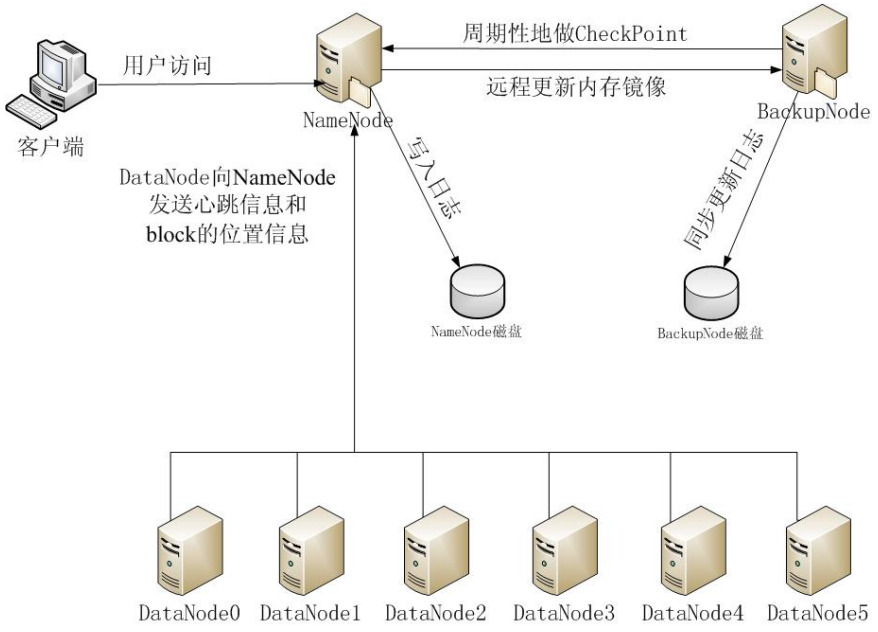

图 4.1　使用 Backup Node 的 HDFS 系统架构图

根据目前代码的实现情况，Backup Node 可配置成两种角色分别运行：Checkpoint Node 或第 2 阶段的 Backup Node，目前还不支持 Standby Node。此外，由于 Backup Node 实际负责元数据的持久化存储，因此在 NameNode 上可以不需要永久保存元数据文件的存储介质。

4.1.2　使用原则

不同阶段的 Backup Node 具有不同的特性，因而使用方法也不尽相同。总的来说：Checkpoint Node 或第 2 阶段 Backup Node 是冷备的方式，当 NameNode 无法正常服务时，需要重启 Backup Node 来恢复服务；第 2 阶段 Backup Node 由于改进了 Checkpoint 的实现方式，与 Checkpoint Node 相比，它具有以下几个优点：

- 实时备份、解决了 Checkpoint Node 的备份一致性问题。
- 直接在内存合并输出，不需要从 NameNode 再下载镜像文件。
- NameNode 可以无需存储，解决了 Checkpoint Node 需要通过 NFS 进行元数据备份的问题。
- 第 3 阶段的 Standby Node 已具备热备功能，可直接切换，恢复服务时间最短。

4.1.3 优缺点

Backup Node 的优缺点在第 1 章中已进行了细致分析，下面从其他方面进行简要论述。

首先从机制本身，按照 Backup Node 的三阶段计划，Backup Node 可实现名字空间元数据的热备，这种方式相对之前的冷备方式，无疑能大大节约恢复时间，但是，Backup Node 并未对另一种元数据（块位置映射信息）做热备，因此即便是成功接替后，也需要等待下面的 Data Nodes 上报该信息后，才能正常提供服务。

以 Face Book 的统计为例，他们的集群存储了 1.5 亿个文件，加载名字空间元数据和等待块位置映射信息上报这两个阶段各自耗费的时间大概都在 20 分钟左右，基本相等，因此，Backup Node 的恢复时间并未达到优化的极限。

> 从目前实现的情况来看，Hadoop 的最新版本中 Backup Node 可完全替代 Secondary Node，但遗憾的是目前的代码还未实现 Warm Standby（具体分析见 4.2.4 节故障切换机制），因此，目前 Backup Node 实现 HDFS 的 HA 还是一种冷备的方式，这就是理想与现实之间的差距。

4.2 运行机制分析

由前面的论述可知，Checkpoint Node 的运行机制和 Secondary Node 相似，Secondary Node 会定期地从 NameNode 下载镜像文件和日志，然后在内存中合并，产生新的镜像文件，并将其上传到 NameNode。NameNode 会将日志实时更新到

dfs.namenode.name.dir 配置的目录下。

对于第 2 阶段 Backup Node 而言，NameNode 会将注册上来的 Backup Node 也视为一个存储对象，以同样的方式存储日志。

下面我们将结合代码进行详细分析，不仅知其然，还要知其所以然。在论述的过程中，将采用情景分析的方式，选取一些重要场景，分析这些场景以及所涉及的代码。

> 我们采用的源码版本为 0.21.0，Backup Node 的代码主要位于如下目录：
> hadoop-0.21.0/hdfs/src/java/org/apache/hadoop/hdfs/server/namenode。

Backup Node 相关的文件有：

- BackupNode.java
- BackupStorage.java
- Checkpointer.java
- CheckpointSignature.java
- NameNode.java
- EditLogBackupInputStream.java
- EditLogBackupOutputStream.java
- ……

4.2.1　启动流程

Backup Node 启动的流程如下：

（1）创建初始化

（2）握手（Handshake）

（3）NameNode 初始化

（4）注册（Registration）

（5）启动后台线程，定期创建 Checkpoint 检查点

1. 创建初始化

Backup Node 主要由类 BackupNode 实现，BackupNode 是 NameNode 的子类，

Backup Node 的启动和 NameNode 的启动是一个入口，Main 函数会在启动的时候，根据输入的参数，决定创建哪个类，并传入不同的初始化参数。

Backup Node 会在构造函数中进行初始化，初始化主要完成四个步骤，也就是我们后面描述的：握手、NameNode 初始化、注册和启动 Checkpoint 后台线程。

下面我们来分析初始化具体代码。

Backup Node 的启动入口。

NameNode.java

```
public static void main(String argv[]) throws Exception {
    try {
        ……
        NameNode namenode = createNameNode(argv, null);
        ……
    } catch (Throwable e) {
        ……
    }
}
```

根据不同的输入参数创建不同的 NameNode 对象，由如下代码可知 BACKUP、CHECKPOINT 的实现是同一个类 Backup Node。

NameNode.java

```
public static NameNode createNameNode(String argv[], Configuration conf)
throws IOException {
    ……
    switch (startOpt) {
    case FORMAT:
        boolean aborted = format(conf, true);
        System.exit(aborted ? 1 : 0);
```

```
    case FINALIZE:
        aborted = finalize(conf, true);
        System.exit(aborted ? 1 : 0);
        return null; // avoid javac warning
    case BACKUP:
        case CHECKPOINT:
        return new BackupNode(conf, startOpt.toNodeRole());
    default:
        return new NameNode(conf);
    }
}
```

类 Backup Node 继承自 NameNode。

BackupNode.java

```
public class BackupNode extends NameNode {
    ……
}
```

类 BackupNode 的构造函数，调用父类的构造方法。

BackupNode.java

```
BackupNode(Configuration conf, NamenodeRole role) throws IOException {
    // 对于 Checkpoint Node 和 Backup Node 分别传入不同的 role


}
```

父类 NameNode 的构造方法。

NameNode.java

```
protected NameNode(Configuration conf, NamenodeRole role)


    ……
```

```
        this.role = role;
        try {
            initialize(conf);
        } catch (IOException e) {
            ……
        }
    }
```

类 BackupNode 中重写（override）了父类 NameNode 的 initialize 方法，initialize 中执行四个重要步骤：

（1）握手（handshake）

（2）NameNode 初始化

（3）注册（registration）

（4）启动 Checkpoint 线程

```
NameNode.java
protected void initialize(Configuration conf) throws IOException {
    ……
    NamespaceInfo nsInfo = handshake(conf);
    super.initialize(conf);

    registerWith(nsInfo);
    runCheckpointDaemon(conf);
}
```

2. Handshake

Handshake 的主要功能是在 Backup Node 上建立与 NameNode 的通信机制（创建 RPC 调用的 Proxy），对 Backup Node 的信息和 NameNode 上的信息进行验证，只有验证通过了，程序才继续向下执行。

Backup Node 通过 NameNode 的 RPC 调用，获取 NameNode 的 NamespaceInfo

信息，NamespaceInfo 信息除了用于和 Backup Node 的自身信息进行验证、比较，同时还将作为后续函数的参数。

NamespaceInfo 包括 Build Version 和 Layout Version。 Handshake 主要验证的是 NameNode 与 Backup Node 双方的 Build 版本是否一致，也就是说这两个程序是否是一起构建出来的，以及各自 Layout 的版本是否一致。这是从大的方面对 Backup Node 进行验证，后面的步骤中，还要从其他的方面对 Backup Node 进行验证。

Handshake 方法通过 RPC 调用 NameNode 的接口，获取 NamespaceInfo，如果获取失败，将休眠一段时间后，重新获取，获取成功后在本地进行验证。

BackupNode.java

```
private NamespaceInfo handshake(Configuration conf) throws IOException {
    // 与 name node 建立连接，获得 name node 的 RPC 调用 proxy

    this.namenode = (NamenodeProtocol) RPC.waitForProxy(NamenodePro
tocol.class,
        NamenodeProtocol.versionID, nnAddress, conf);
    // 通过 RPC 调用从 name-node 获取 version 和 id info
    NamespaceInfo nsInfo = null;
    while(!isStopRequested()) {
        try {
            nsInfo = handshake(namenode);
            break;
        } catch(SocketTimeoutException e) {
            ……
        }

    return nsInfo;
}
```

handshake 中 RPC 调用 NameNode 中的 versionRequest 方法，获取

NamespaceInfo，对 Build Version 和 Layout Version 进行验证。

```
BackupNode.java
private static NamespaceInfo handshake(NamenodeProtocol namenode)
throws IOException, SocketTimeoutException {
    NamespaceInfo nsInfo;
    nsInfo = namenode.versionRequest();
    ……
    // 如果 build version 与本地的不一致，则抛出异常
    if( ! nsInfo.getBuildVersion().equals( Storage.getBuildVersion())) {
        ……
        throw new IOException(errorMsg);
    }
    // 如果 layout verison 不一致，则退出
    assert FSConstants.LAYOUT_VERSION == nsInfo.getLayoutVersion() :
       "Active and backup node layout versions must be the same. Expected: "
        + FSConstants.LAYOUT_VERSION + " actual "+
nsInfo.getLayoutVersion();
    return nsInfo;
}
```

NameNode 中的 versionRequest 方法。

```
NameNode.java
public NamespaceInfo versionRequest() throws IOException {
    //调用 namesystem 的 getNamespaceInfo 方法
    //namesystem 是在创建 NameNode 时的 initialize 方法中创建的，
    //是一个 FSNamesystem 类，具体创建函数是 loadNamesystem(conf)
    return namesystem.getNamespaceInfo();
}
```

FSNamesystem 中的 getNamespaceInfo 方法调用，返回 NamespaceInfo：包括

NamespaceID、CTime 以及 Version 信息。CTime 是指文件系统状态的创建时间，它在升级（upgrade）的时候被更新。

```
FSNameSystem.java
NamespaceInfo getNamespaceInfo() {
    return new NamespaceInfo(dir.fsImage.getNamespaceID(),
dir.fsImage.getCTime(),
        getDistributedUpgradeVersion());
}
```

3. Initialize

Handshake 验证成功后，Backup Node 会调用父类的 initialize 方法。initialize 中的关键方法是 loadNamesystem。Backup Node 重写（override）了 loadNamesystem 方法，因此，实际调用的是 Backup Node 的 loadNamesystem 方法。在 NameNode 的 loadNamesystem 方法中，NameNode 会从磁盘加载元数据到内存，而 Backup Node 的 loadNamesystem 并不加载元数据到内存。具体分析代码如下：

handshake 执行完后，接下来调用父类 NameNode 中的 initialize 方法。

```
NameNode.java
protected void initialize(Configuration conf) throws IOException {
    ……
    loadNamesystem(conf);
    ……
}
```

Backup Node 重写（override）了父类 NameNode 的 loadNamesystem 方法，不同之处在于 FSNamesystem 的构造函数。

```
BackupNode.java
protected void loadNamesystem(Configuration conf) throws IOException {
    BackupStorage bnImage = new BackupStorage();
```

```
        this.namesystem = new FSNamesystem(conf, bnImage);
        bnImage.recoverCreateRead(FSNamesystem.getNamespaceDirs(conf),
            FSNamesystem.getNamespaceEditsDirs(conf));
    }
    NameNode.java
    protected void loadNamesystem(Configuration conf) throws IOException {
        this.namesystem = new FSNamesystem(conf);
    }
```

FSNamesystem 类中维护了几个重要的表：

- 维护了文件名与 block 列表的映射关系；
- 有效 block 的集合；
- block 与节点列表的映射关系；
- 节点与 block 列表的映射关系；
- 更新的 heartbeat 节点的 LRU cache。

```
    FSNamesystem.java
    FSNamesystem(Configuration conf, BackupStorage bnImage) throws
IOException {
        try {
            //Backup Node 中调用的 initialize 参数非空
            initialize(conf, bnImage);
        } catch(IOException e) {
            ……
        }
    }
    FSNamesystem(Configuration conf) throws IOException {
        try {
```

```
            initialize(conf, null);
        } catch(IOException e) {
            ……
        }
    }
    private void initialize(Configuration conf, FSImage fsImage)
    throws IOException {
        ……
        if(fsImage == null) {
            // NameNode 的执行路径 FSDirectory 负责存储文件系统的目录状态，
            //也就是维护文件系统的名字空间
            this.dir = new FSDirectory(this, conf);
            StartupOption startOpt = NameNode.getStartupOption(conf);
            // 加载元数据文件并合并，形成内存镜像
            this.dir.loadFSImage(getNamespaceDirs(conf),
            getNamespaceEditsDirs(conf), startOpt);
            ……
        } else {
            // Backup Node 的执行路径



        }
        ……
    }
```

NameNode 调用 loadFSImage，设置 ready 标志；而 Backup Node 不调用 loadFSImage，因此 ready 为初始值 false。ready 标志非常重要，许多客户端对文件的操作，都需要检查 ready 值，只有在 ready=true，才允许操作继续，因此，Backup Node 在 ready=true 之前，都无法响应客户端的操作。

```
FSDirectory.java
void loadFSImage(Collection<URI> dataDirs, Collection<URI> editsDirs,
StartupOption startOpt) throws IOException {
    ……
    try {
        // 选择最新的 FSImage 和 Edits 加载到内存，合并形成内存镜像
       if (fsImage.recoverTransitionRead(dataDirs, editsDirs, startOpt)) {
           fsImage.saveNamespace(true);
        }
        ……
    } catch(IOException e) {
        ……
    }
    synchronized (this) {
        // 设置 ready 标志



    }
}
```

4. Register

Backup Node 向 NameNode 注册之前，需要先验证双方的 NameSpaceID 是否一致，如果不一致，则会抛出异常，退出程序。如果一致，则通过 RPC 远程调用 NameNode 的 register 接口，并传入自身的注册信息，注册信息包括：rpcAddress、httpAddress、FSImage、Role、CheckpointTime 等。

NameNode 根据传入的 Backup Node 注册信息进行判断，看是否允许该 Backup Node 进行注册，确保任何时候都只有一个活动的 Backup Node 注册在 NameNode 上。

注册成功后，会返回 NameNode 的注册信息，Backup Node 会保存该信息，用于后续的通信与处理。

Backup Node 通过 RPC 调用 NameNode 的 register 方法，向 NameNode 进行注册，同时获得 NameNode 自身的注册信息。

BackupNode.java

```
// 传入的参数是 handshake 时获取的 nsInfo
private void registerWith(NamespaceInfo nsInfo) throws IOException {
// 1. 验证本地镜像的 namespaceID 和 NameNode 的 namespaceID，如果不一致，抛出异常
    ……
// 2. 注册
    setRegistration();
    NamenodeRegistration nnReg = null;
    while(!isStopRequested()) {
        try {
            // NameNode RPC 调用
            nnReg = namenode.register(getRegistration());
            break;
        } catch(SocketTimeoutException e) {
            ……
        }
    }
// 3. 注册结果处理
    ……
    nnRpcAddress = nnReg.getAddress();
}
// 产生 Backup Node 的注册信息
NamenodeRegistration setRegistration() {
    nodeRegistration = new
NamenodeRegistration(getHostPortString(rpcAddress),
        getHostPortString(httpAddress), getFSImage(), getRole(),
```

```
        return nodeRegistration;
    }
    NamenodeRegistration getRegistration() {
        return nodeRegistration;
    }
```

NameNode register 方法返回 NameNode 自身的 NamenodeRegistration 信息。

```
    NameNode.java
    public NamenodeRegistration register(NamenodeRegistration registration)
    throws IOException {
        verifyVersion(registration.getVersion());
        namesystem.registerBackupNode(registration);
        return setRegistration();
    }
```

NameNode 对注册信息进行验证，确定是否允许注册。

```
    FSNamesystem.java
    synchronized void registerBackupNode(NamenodeRegistration registration)
    throws IOException {
        // 验证 BackupNode 的 NamespaceID 与 NameNode 的 NamespaceID

            // 抛出异常
        }
          // 检查 Backup Node 的有效性，确定是否允许注册
        boolean regAllowed =
getEditLog().checkBackupRegistration(registration);
        if(!regAllowed)
            throw new IOException("Registration is not allowed. " +

        }
```

```
    FSEditLog getEditLog() {
        return getFSImage().getEditLog();
    }
    FSImage getFSImage() {
        return dir.fsImage;
    }
    FSImage.java
    public FSEditLog getEditLog() {
        return editLog;
    }
```

FSEditLog 的 checkBackupRegistration 方法。

```
    FSEditLog.java
    synchronized boolean checkBackupRegistration(NamenodeRegistration
registration) {
        // 获取 BACKUP 类型的输出日志流
        Iterator<EditLogOutputStream> it =
getOutputStreamIterator(JournalType.BACKUP);
        // 如果没有输出流，则 regAllowed=true
        boolean regAllowed = !it.hasNext();
        NamenodeRegistration backupNode = null;
        ArrayList<EditLogOutputStream> errorStreams = null;
        while(it.hasNext()) {
        // 如果有 EditLog 输出流，则判断该输出流对应的 Backup Node 和注册的是否相同
            EditLogBackupOutputStream eStream =
(EditLogBackupOutputStream)it.next();
            backupNode = eStream.getRegistration();
            if(backupNode.getAddress().equals(registration.getAddress()) &&

        // 如果是同一个，则 regAllowed = true
```

```
                regAllowed = true; // same node re-registers
                break;
            }
            if(!eStream.isAlive()) {
        // 如果 EditLog 输出流对应的 Backup Node 不再存在，regAllowed = true
                if(errorStreams == null)
                    errorStreams = new ArrayList<EditLogOutputStream>(1);
                errorStreams.add(eStream);
                regAllowed = true; // previous backup node failed
            }
        }
        // 如果 backupNode 存在且是 BACKUP，则报警退出
        assert backupNode == null || backupNode.isRole(NamenodeRole.BACKUP) :
        "Not a backup node corresponds to a backup stream";
        processIOError(errorStreams, true);
        // 输出流不存在，返回 true
        // 输出流存在且该输出流对应的 Backup Node 和注册的是同一个，返回 true
        // 输出流存在且该输出流对应的 Backup Node 不存在，返回 true
        // 其他情况，返回 false
        return regAllowed;
    }
```

5. Checkpoint 后台线程

注册成功后，Backup Node 会启动一个后台线程，专门用于做 Checkpoint，Checkpoint Node和第 2 阶段的 Backup Node 都是通过该线程实现 Checkpoint 的，只是各自执行的步骤稍有差异。该线程是一个无限循环，每 5 分钟检查一次，只要当前时间距离上一次 Checkpoint 的时间超过 5 分钟，或者 Edits 的大小超过设定值，就会触发 Checkpoint 任务。

每次 Checkpoint 任务执行以下几个步骤。

（1）在 NameNode 上进行 Checkpoint 的验证工作，如判断 Backup Node 的 FSImage 是否过期等，如果符合 Checkpoint 的条件，则在 NameNode 上进行相关的准备工作，如在日志保存目录下，创建新的 Edits 文件，将日志输出流重新定位，输出到新的日志文件等，最后将返回一个类（用于标识本次 Checkpoint）。

（2）Backup Node 根据验证结果进行相应的动作，如果允许 Checkpoint，对于 Checkpoint Node 则总是从 NameNode 下载新的 FSImage 文件和 Edits 文件，对于第 2 阶段的 Backup Node，则是根据验证返回的结果判断本地的元数据是否过期，如果过期，也将从 NameNode 下载新的 FSImage 文件和 Edits 文件。

（3）在内存中合并产生新的元数据，对于 Checkpoint Node 来说，需要先从本地磁盘读取 FSImage 文件和 Edits 文件，再进行合并，而对于第 2 阶段的 Backup Node，则直接在内存进行合并，因而效率更高。

（4）将合并后的元数据输出到磁盘。

（5）如果需要上传，则将合并后的元数据上传到 NameNode，替换原来的元数据文件。具体实现见下面的代码分析。

BackupNode 的 runCheckpointDaemon 方法，启动后台线程。

BackupNode.java

```
private void runCheckpointDaemon(Configuration conf) throws IOException {
    checkpointManager = new Checkpointer(conf, this);


}
```

Checkpointer 的主线程。

Checkpointer.java

```
public void run() {
```

```
    // 1. 设置检查时间 periodMSec 为 5 分钟
    ……
    // 2. 计算最后一次 check point 时间
    ……
    while(shouldRun) {
        try {
            // 如果当前时间距离最后一次 check point 时间超过 periodMSec
            // 或者 Edits 大小超过 check point 设定值，则进行 check point

                doCheckpoint();
                lastCheckpointTime = now;
            }
        } catch(IOException e) {
            ……
        } catch(Throwable e) {
            ……
        }
        try {
            // check point 结束后，休眠 periodMSec
            Thread.sleep(periodMSec);
        } catch(InterruptedException ie) {}
    }
}
```

Checkpointer 的 doCheckpoint 方法。

Checkpointer.java

```
void doCheckpoint() throws IOException {
    long startTime = FSNamesystem.now();
    // Namenode 验证 Checkpoint 的条件，如可以，做相关准备，
    // 如创建新的 EditLog 文件，将日志输出流重定向到新的 EditLog 文件等
```

```
        NamenodeCommand cmd =
            getNamenode().startCheckpoint(backupNode.getRegistration());
        CheckpointCommand cpCmd = null;
        switch(cmd.getAction()) {
        case NamenodeProtocol.ACT_SHUTDOWN:
            shutdown();
            throw new IOException("Name-node " + backupNode.nnRpcAddress
            + " requested shutdown.");
        case NamenodeProtocol.ACT_CHECKPOINT:
            cpCmd = (CheckpointCommand)cmd;
            break;
        default:
            throw new IOException("Unsupported NamenodeCommand:
"+cmd.getAction());
        }
        // 获取本次 Checkpoint 的标识 sig

        assert FSConstants.LAYOUT_VERSION == sig.getLayoutVersion() :
            "Signature should have current layout version. Expected: "
            + FSConstants.LAYOUT_VERSION + " actual "+ sig.getLayoutVersion();
        assert !backupNode.isRole(NamenodeRole.CHECKPOINT) ||
            cpCmd.isImageObsolete() : "checkpoint node should always download
image.";
        backupNode.setCheckpointState(CheckpointStates.UPLOAD_START);
        if(cpCmd.isImageObsolete()) {
            // First reset storage on disk and memory state
            backupNode.resetNamespace();

            downloadCheckpoint(sig);
        }
```

```
    ……
}
```

Checkpointer 的 doCheckpoint 方法。

```
Checkpointer.java
void doCheckpoint() throws IOException {
    ……
    BackupStorage bnImage = getFSImage();
    // 加载 FSImage、Edits 等元数据文件，并在内存进行合并
    bnImage.loadCheckpoint(sig);
    sig.validateStorageInfo(bnImage);
    // 将最新的元数据保存到磁盘
    bnImage.saveCheckpoint();
    // 如果需要 upload，则将元数据 upload 到 NameNode

        uploadCheckpoint(sig);
    getNamenode().endCheckpoint(backupNode.getRegistration(), sig);
    bnImage.convergeJournalSpool();
    backupNode.setRegistration(); // keep registration up to date
    if(backupNode.isRole(NamenodeRole.CHECKPOINT))
        getFSImage().getEditLog().close();
    LOG.info("Checkpoint completed in " +
        (FSNamesystem.now() - startTime)/1000 + " seconds."+

}
```

NameNode 的 startCheckpoint 方法，用来验证注册的 BackupNode 信息：LAYOUT_VERSION、RegistrationID，同时还验证 NameNode 自身是否处于 ACTIVE 状态。

NameNode.java

```
public NamenodeCommand startCheckpoint(NamenodeRegistration
registration)
throws IOException {
    // 验证
    verifyRequest(registration);
    if(!isRole(NamenodeRole.ACTIVE))
       throw new IOException("Only an ACTIVE node can invoke startCheckpoint.");
    return namesystem.startCheckpoint(registration, setRegistration());
}
public void verifyRequest(NodeRegistration nodeReg) throws IOException {
    verifyVersion(nodeReg.getVersion());
    if
(!namesystem.getRegistrationID().equals(nodeReg.getRegistrationID()))
        throw new UnregisteredNodeException(nodeReg);
}
public void verifyVersion(int version) throws IOException {
    if (version != LAYOUT_VERSION)
        throw new IncorrectVersionException(version, "data node");
}
```

FSNamesystem.java

```
synchronized NamenodeCommand startCheckpoint(
    NamenodeRegistration bnReg, // backup node
    NamenodeRegistration nnReg) // active name-node
throws IOException {
    LOG.info("Start checkpoint for " + bnReg.getAddress());
    NamenodeCommand cmd = getFSImage().startCheckpoint(bnReg, nnReg);
    getEditLog().logSync();

}
```

NameNode 的 startCheckpoint 方法将会判断 Backup Node 本地镜像与 active name-node 镜像，如果不兼容或者有更新，则关闭 Backup Node；如果 Backup Node 本地镜像比 active name-node 镜像老，则从 active name-node 下载新的镜像；如果两者镜像相同，则使用本地镜像作为当前镜像。这样可以确保 Backup Node 的镜像与 active name-node 镜像一致，并且以 active name-node 镜像为准。判断的依据是 NamespaceID 和 LayoutVersion。

FSImage.java

```
NamenodeCommand startCheckpoint(NamenodeRegistration bnReg, // backup
node
NamenodeRegistration nnReg) // active name-node
throws IOException {
    String msg = null;
    // Verify that checkpoint is allowed
    if(bnReg.getNamespaceID() != this.getNamespaceID())
        msg = "Name node " + bnReg.getAddress()+
        " has incompatible namespace id: " + bnReg.getNamespaceID()+
        " expected: " + getNamespaceID();
    else if(bnReg.isRole(NamenodeRole.ACTIVE))
        msg = "Name node " + bnReg.getAddress() +
        " role " + bnReg.getRole() + ": checkpoint is not allowed.";
    else if(bnReg.getLayoutVersion() < this.getLayoutVersion() ||
        (bnReg.getLayoutVersion() == this.getLayoutVersion() &&
        bnReg.getCTime() > this.getCTime()) ||
        (bnReg.getLayoutVersion() == this.getLayoutVersion() &&
        bnReg.getCTime() == this.getCTime() &&
        bnReg.getCheckpointTime() > this.checkpointTime))
        // remote node has newer image age

        " has newer image layout version: LV = " +
```

```
            bnReg.getLayoutVersion()+ " cTime = " + bnReg.getCTime() +
            " checkpointTime = " + bnReg.getCheckpointTime() +
            ". Current version: LV = " + getLayoutVersion() +
            " cTime = " + getCTime() + " checkpointTime = " + checkpointTime;
        if(msg != null) {
            LOG.error(msg);
            // 如果不符合条件，则返回 ACT_SHUTDOWN，将关闭 Backup Node
            return new NamenodeCommand(NamenodeProtocol.ACT_SHUTDOWN);
        }
        ……
    }
```

NameNode 的 startCheckpoint 方法。

```
    FSImage.java
    NamenodeCommand startCheckpoint(NamenodeRegistration bnReg, // backup
node
    NamenodeRegistration nnReg) // active name-node
    throws IOException {
        ……
        // 判断 Backup Node 上的 FSImage 是否过期
        // 如果是，Backup Node 需要从 NameNode 下载最新的 FSImage
        boolean isImgObsolete = true;
        if(bnReg.getLayoutVersion() == this.getLayoutVersion()
        && bnReg.getCTime() == this.getCTime()
        && bnReg.getCheckpointTime() == this.checkpointTime)
            isImgObsolete = false;

        // Backup Node 进行 Checkpoint 后，需要将元数据 upload 回 NameNode

        if(getNumStorageDirs(NameNodeDirType.IMAGE) == 0)
```

```
        // do not return image if there are no image directories
        needToReturnImg = false;
    CheckpointSignature sig = rollEditLog();
    // 创建 Backup Node 的 edit output stream,它会将 NameNode 的日志记录以 stream
    // 的方式发送到指定的 Backup Node。对于 Checkpointer 则不需要创建
    getEditLog().logJSpoolStart(bnReg, nnReg);
    // 向 Backup Node 返回 startCheckpiont 的结果
    return new CheckpointCommand(sig, isImgObsolete, needToReturnImg);
}
```

FSImage 的 rollEditLog 方法。

```
FSImage.java
CheckpointSignature rollEditLog() throws IOException {
    getEditLog().rollEditLog();
    ckptState = CheckpointStates.ROLLED_EDITS;
    // 增长 checkpoint 的 age。checkpointTime 表示 FSImage 的保存时间

    return new CheckpointSignature(this);
}

public FSEditLog getEditLog() {
    return editLog;
}
```

FSEditLog 的 rollEditLog 方法。

```
FSEditLog.java
synchronized void rollEditLog() throws IOException {

    // 如果 NameNode 本地不保存 EditLogs，不需要任何操作，直接返回
    Iterator<StorageDirectory> it =
```

```
fsimage.dirIterator(NameNodeDirType.EDITS);
       if(!it.hasNext())
           return;
       // 所有存储目录下的 edits.new 文件必须一致，要不全部存在，要不全部不存在
       boolean alreadyExists = existsNew(it.next());
       while(it.hasNext()) {
           StorageDirectory sd = it.next();
           if(alreadyExists != existsNew(sd))
               throw new IOException(getEditNewFile(sd)
                   + "should " + (alreadyExists ? "" : "not ") + "exist.");
       }
       // 如果 edits.new 存在，则不需要任何操作，直接返回
       if(alreadyExists)
           return;
       // 检查之前移除的存储目录，是否还能使用
       fsimage.attemptRestoreRemovedStorage();
       // 关闭之前的 EditLog 输出流，重定向到 edits.new 上

           Storage.STORAGE_DIR_CURRENT + "/" +
NameNodeFile.EDITS_NEW.getName());
   }
```

FSEditLog 的 divertFileStreams 方法。

```
   FSEditLog.java
   synchronized void divertFileStreams(String dest) throws IOException {
       waitForSyncToFinish();

of streams";
       ArrayList<EditLogOutputStream> errorStreams = null;
       EditStreamIterator itE =
```

```
            (EditStreamIterator)getOutputStreamIterator(JournalType.FILE);
        Iterator<StorageDirectory> itD =
fsimage.dirIterator(NameNodeDirType.EDITS);
        while(itE.hasNext() && itD.hasNext()) {
            EditLogOutputStream eStream = itE.next();
            StorageDirectory sd = itD.next();
            if(!eStream.getName().startsWith(sd.getRoot().getPath()))
        throw new IOException("Inconsistent order of edit streams: " + eStream);
            try {
                // 关闭之前的 stream，并创建一个新的 stream 进行替换
                closeStream(eStream);
        eStream = new EditLogFileOutputStream(new File(sd.getRoot(), dest),
                    sizeOutputFlushBuffer);
                eStream.create();
                itE.replace(eStream);
            } catch (IOException e) {
                ……
            }
        }
        processIOError(errorStreams, true);
    }
```

CheckpointSignature 用来标识一次 Checkpoint。

```
    CheckpointSignature.java
    public class CheckpointSignature extends StorageInfo


        ……
    }
    CheckpointSignature(FSImage fsImage) {
        // 在父类中设置 layoutVersion、namespaceID、cTime
```

```
    super(fsImage);
    // 再增加 editsTime、checkpointTime
    editsTime = fsImage.getEditLog().getFsEditTime();
    checkpointTime = fsImage.getCheckpointTime();
}
StorageInfo.java
public StorageInfo(StorageInfo from) {
    setStorageInfo(from);
}
public void  setStorageInfo(StorageInfo from) {
    layoutVersion = from.layoutVersion;
    namespaceID = from.namespaceID;

}
```

4.2.2　元数据操作情景分析

本节以用户操作为主线，以元数据为研究对象，分析 NameNode 和 Backup Node 的内部运行流程和交互机制。

图 4.2 是用户数据操作的一个示意图，首先由用户运行客户端命令发起操作，接下来 NameNode 会更新内存中的元数据镜像，并将日志记录更新到日志文件，通过输出流将日志记录同步到 Backup Node 的内存，Backup Node 将日志记录与内存中元数据进行合并，形成新的内存镜像，并将日志记录更新到本地的日志文件。

接下来以一个用户的具体操作 mkdir 为例，结合代码，分析其整个流程，总的流程可以分为以下几个步骤。

（1）客户端执行命令。

（2）NameNode 更新内存镜像。

（3）NameNode 更新日志。

（4）Backup Node同步更新内存镜像。

（5）Backup Node 更新磁盘上的日志。

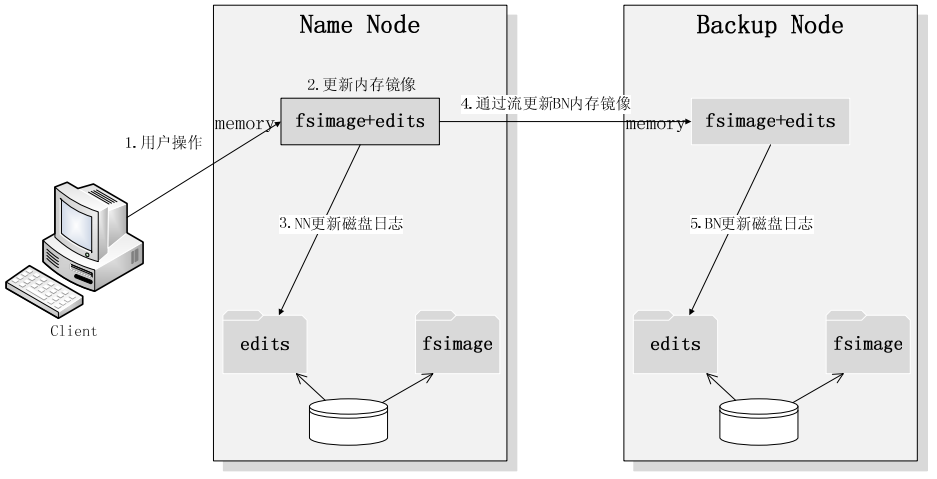

图 4.2　用户操作流程图

代码分析如下：

1. 客户端执行命令流程

FsShell 的 mkdir 方法，用户执行控制台命令时会调用此方法。

FsShell.java

```
void mkdir(String src) throws IOException {
    ……
    try {
        ……
    } catch(FileNotFoundException e) {
        if (!srcFs.mkdirs(f)) {
            throw new IOException("failed to create " + src);
        }
```

```
}
```

FsShell 的 mkdir 方法，用户执行控制台命令时会调用此方法。

FsShell.java

```
void mkdir(String src) throws IOException {
    ……
    try {
        ……
    } catch(FileNotFoundException e) {
        if (!srcFs.mkdirs(f)) {
            throw new IOException("failed to create " + src);
        }
    }
}
```

DistributedFileSystem 的 mkdirs 方法，此方法被 FileSystem.mkdirs(Path f)方法所调用。

DistributedFileSystem.java

```
public boolean mkdirs(Path f, FsPermission permission) throws IOException {
    return dfs.mkdirs(getPathName(f), permission, true);
}
```

DFSClient 的 mkdirs 方法，通过 RPC 远程调用 NameNode 所实现的接口。

DFSClient.java

```
public boolean mkdirs(String src, FsPermission permission, boolean


throws IOException, UnresolvedLinkException {
    ……
    try {
        // RPC 远程调用 NameNode 的方法
```

```
            return namenode.mkdirs(src, masked, createParent);
        }
        ……
    }
```

NameNode 端方法调用流程。

NameNode 的 mkdirs 方法。

```
    NameNode.java
    public boolean mkdirs(String src, FsPermission masked, boolean
createParent)
throws IOException {
    ……
    return namesystem.mkdirs(src, new
        PermissionStatus(UserGroupInformation.getCurrentUser().getShortUse
rName(), null, masked), createParent);
    }
```

FSNamesystem 的 mkdirs 方法。

```
    FSNamesystem.java
    public boolean mkdirs(String src, PermissionStatus permissions,boolean
createParent)
    throws IOException, UnresolvedLinkException {
        boolean status = mkdirsInternal(src, permissions, createParent);
        ……
    }
    private synchronized boolean mkdirsInternal(String src,

    throws IOException, UnresolvedLinkException {
        ……
        if (!dir.mkdirs(src, permissions, false, now())) {
```

```
            throw new IOException("Invalid directory name: " + src);
        }
        return true;
    }
```

2. NameNode 更新内存镜像

FSDirectory 的 mkdirs、unprotectedMkdir 和 addChild 方法。

```
    FSDirectory.java
    boolean mkdirs(String src, PermissionStatus permissions,boolean
inheritPermission,
    long now)
    throws FileAlreadyExistsException, QuotaExceededException,
UnresolvedLinkException{
        // create directories beginning from the first null index
        for(; i < inodes.length; i++) {
            ……
            unprotectedMkdir(inodes, i, components[i],
               permissions,inheritPermission || i != components.length-1, now);
            if (inodes[i] == null) {
                return false;
            }
            ……
        }
        ……
    }
    private void unprotectedMkdir(INode[] inodes, int pos, byte[] name,
    PermissionStatus permission, boolean inheritPermission,long timestamp)

        // 在内存目录树中添加了一个节点
        inodes[pos] = addChild(inodes, pos, new INodeDirectory(name,
```

```
permission, timestamp),-1, inheritPermission );
    }
    private <T extends INode> T addChild(INode[] pathComponents, int pos,T
child,
    long childDiskspace, boolean inheritPermission)throws
QuotaExceededException {
        return addChild(pathComponents, pos, child,
            childDiskspace,inheritPermission, true);
    }
```

3. NameNode 将日志写入日志文件

将日志写入 buffer。

FSDirectory 的 mkdirs 方法。

```
    FSDirectory.java
    boolean mkdirs(String src, PermissionStatus permissions,boolean
inheritPermission,
    long now)
    throws FileAlreadyExistsException, QuotaExceededException,

        ……
        fsImage.getEditLog().logMkDir(cur, inodes[i]);
        ……
    }
```

FSEditLog 的 logMkDir、logEdit 方法。

```
    FSEditLog.java
    public void logMkDir(String path, INode newNode) {
        ……
        logEdit(OP_MKDIR, new ArrayWritable(DeprecatedUTF8.class, info),
```

```
            newNode.getPermissionStatus());
    }
    synchronized void logEdit(byte op, Writable ... writables) {
        ……
        try {
            //将日志写入缓存中
            eStream.write(op, writables);
        } catch (IOException ie) {
            ……
        }
    }
```

将日志通过流写入日志文件。

FSNamesystem 的 mkdirs 方法。

```
    FSNamesystem.java
    public boolean mkdirs(String src, PermissionStatus permissions,boolean
createParent)
    throws IOException, UnresolvedLinkException {
        ……
        EditLog().logSync();
        ……
    }
```

FSEditLog 的 logSync 方法。

```
    FSEditLog.java
    public void logSync() throws IOException {

        for (int idx = 0; idx < streams.length; idx++) {
            EditLogOutputStream eStream = streams[idx];
            try {
```

```
            //将缓存中的日志写入本地文件或发送到 BN
            eStream.flush();
        } catch (IOException ie) {
            ……
        }
        ……
    }
}
```

4. NameNode 通过日志输出流更新 Backup Node 内存镜像

NameNode 通过流将日志发送给 Backup Node。

FSEditLog 的 logSync 方法以及 EditLogBackupOutputStream 相关方法。

FSEditLog.java

```
public void logSync() throws IOException {
    ……
    for (int idx = 0; idx < streams.length; idx++) {
        EditLogOutputStream eStream = streams[idx];
    try {
        //将缓存中的日志写入本地文件或发送到 BN
        eStream.flush();
    } catch (IOException ie) {
        ……
    }

}
```

EditLogBackupOutputStream.java

```
public void flush() throws IOException {
    ……
    flushAndSync();
```

```
        ……
    }
    protected void flushAndSync() throws IOException {
        ……
        if(out.size() > 0)
            send(NamenodeProtocol.JA_JOURNAL);
        ……
    }
    private void send(int ja) throws IOException {
        try {
            ……
            //RPC 远程调用 BN 的方法
            backupNode.journal(nnRegistration, ja, length, out.getData());
        } finally {
            out.reset();
        }
    }
```

Backup Node 更新本地内存镜像。

BackupNode 的 journal 方法。

```
    BackupNode.java
    public void journal(NamenodeRegistration nnReg,int jAction,int
length,byte[] args)
    throws IOException {
        ……
        switch(jAction) {

        case (int)JA_JOURNAL:
            bnImage.journal(length, args);
```

```
            return;
        ……
        }
    }
    synchronized void journal(int length, byte[] data) throws IOException {
        ……
        try {
            switch(jsState) {
            case WAIT:
            case OFF:
                ……
                // update NameSpace in memory
                backupInputStream.setBytes(data);
                //此方法用于将从 NN 获取的日志信息（即一个包含 byte 数组）的输入
                //流解析，并进行相应的操作更新内存目录树
                editLog.loadEditRecords(getLayoutVersion(),
                    backupInputStream.getDataInputStream(), true);
                ……
            }
            ……
        } finally {
            backupInputStream.clear();
        }
    }
```

5. Backup Node 更新磁盘上的日志文件

BackupStorage 的 journal 方法。

```
BackupStorage.java
synchronized void journal(int length, byte[] data) throws IOException {
```

```
try {
    switch(jsState) {
    ……
    }
    // write to files
    editLog.logEdit(length, data);
    editLog.logSync();
} finally {
    backupInputStream.clear();
}
```

4.2.3　日志池（journal spool）机制

在第 2 阶段的 Backup Node 中，采用了日志池（journal spool）的机制，用于缓存 NameNode 发送过来的来不及处理的日志记录。通常情况下，Backup Node 可以实时地处理接收的日志记录，将其合并更新到内存元数据之中，但在一些情况下，Backup Node 的负载较重，如正在做 Checkpoint 合并元数据，或者将元数据输出到磁盘时，此时如果接收到新的日志记录，就有可能来不及处理。为保证数据的一致性，有必要先将日志缓存起来，待负载较轻时再进行处理。

总的来说，日志池机制包括三个对象：

- 日志生产者：日志生产者由 NameNode 担任，当它发现 Backup Node 不能及时处理数据时，就会通知 Backup Node 创建日志池，并将日志记录以流的方式追加到日志池的尾部；
- 日志池：日志池是一个“先进先出”的结构，可以有效保证日志记录顺序的一致性，并且具有多种状态，可以实现动态创建与销毁；
- 日志消费者：日志的消费者由 Backup Node 担任，它从日志池依次取出日志记录，将其合并更新到内存元数据中，并将更新写入磁盘上的日志文件。

由于日志的产生速度是有限的，而日志处理可快速进行，因此，日志池最终会变空，此时日志池也就失去了存在的意义，Backup Node 将和 NameNode 进行协商销毁日志池，并最终切换到日志实时处理模式。

Journal spool 的实现代码在 BackupStorage.java 中，并未单独形成一个类。Journal spool 通过一个名为 edit.new 的文件来实现的数据存储。Journal spool 有三种状态：

- OFF 表示未创建 Journal spool，此时，日志正常写入 Edits。
- INPROGRESS 表示 Journal spool 可用，此时日志将被写入 Journal spool，即 edit.new。
- WAIT 表示 Backup Node 正在对 Journal spool 操作，需要等待。

状态转换分析如下：

（1）初始化时，未创建 Journal spool，因此状态为 OFF。

（2）当需要创建 Journal spool 时，会调用 startJournalSpool 进行创建，将创建 edit.new 文件作为日志文件，此时状态转为 INPROGRESS。

（3）写入 Journal 时，会进行判断，如果此时状态是 OFF，则写入 Edit log 文件，如果状态是 INPROGRESS，则写入 Journal spool，即 edit.new 文件。

（4）当 checkpoint 结束后，Backup Node 会将 Journal spool 中的记录更新到内存的 NameSpace。它首先会读入 edit.new 文件内容，将状态置为 WAIT，然后进行更新，最后将 edit.new 重命名为 edits，销毁 Journal spool，状态置为 OFF。

代码分析如下：

Journal spool 初始状态为 OFF。

BackupStorage.java

```
BackupStorage() {
```

```
    jsState = JSpoolState.OFF;
}
```

startJournalSpool 方法，用来创建 Journal Spool。

BackupStorage.java

```
synchronized void startJournalSpool(NamenodeRegistration nnReg)
throws IOException {
    ……
    // 1.状态判断，如果已经创建了，则直接返回
    ……
    // 2.创建 Journal Spool 目录
    ……
    // 3.创建 Edits 文件
    ……
    // 4.创建指向 Journal Spool 文件的流
    ……
    // 5.状态置为 INPROGRESS
    jsState = JSpoolState.INPROGRESS;
}
```

Journal 方法，写入日志。

BackupStorage.java

```
synchronized void journal(int length, byte[] data) throws IOException {

    try {
        switch(jsState) {
        case WAIT:
        case OFF:
            // 如果是 WAIT，则一直等待到 Journal Spool 销毁
```

```
                // 更新内存中的 NameSpace
                backupInputStream.setBytes(data);
                editLog.loadEditRecords(getLayoutVersion(),
                    backupInputStream.getDataInputStream(), true);
                getFSNamesystem().dir.updateCountForINodeWithQuota();
                break;
            case INPROGRESS:
                break;
            }
            //如果已经创建了 Journal Spool，将日志记录写入文件

            // 如果状态是 INPROGRESS，则写入 edit.new 文件
            editLog.logEdit(length, data);
            // 日志内容同步，防止“并发写”出现问题
            editLog.logSync();
        } finally {
            backupInputStream.clear();
        }
    }
```

convergeJournalSpool 方法，写入日志。

```
    BackupStorage.java
    void convergeJournalSpool() throws IOException {
        ……
        if(jSpoolFile.exists()) {
            // 加载 edits.new 文件
            EditLogFileInputStream edits = new
EditLogFileInputStream(jSpoolFile);
            DataInputStream in = edits.getDataInputStream();
            numEdits += editLog.loadFSEdits(in, false);
```

```
            // 第 1 次将 edits.new 的内容读完，置 WAIT
            jsState = JSpoolState.WAIT;
            numEdits += editLog.loadEditRecords(getLayoutVersion(), in,
true);
            getFSNamesystem().dir.updateCountForINodeWithQuota();
            edits.close();
        }
        // 销毁 edits.new，将 edits.new 重命名为 edits，后面的日志记录将写入 edits
        editLog.revertFileStreams(STORAGE_JSPOOL_DIR + "/" +
STORAGE_JSPOOL_FILE);
        ……
        // 唤醒 Journal writer，置 OFF
        synchronized(this) {
            jsState = JSpoolState.OFF;
            notifyAll();
        }
        ……
    }
```

convergeJournalSpool 方法调用时机。

```
    Checkpointer.java
    void doCheckpoint() throws IOException {
        ……
        getNamenode().endCheckpoint(backupNode.getRegistration(), sig);
        bnImage.convergeJournalSpool();

    }
```

4.2.4 故障切换机制

按照 Backup Node 的开发计划，Backup Node 需要经过三个阶段，最后才能成为 Warm Standby，目前最新代码 0.21.0 中的 Backup Node 处于第 2 个阶段，即只支持 Checkpoint，不支持热备切换。具体代码分析如下：

Backup Node 启动过程（部分代码）

```
BackupNode.java
protected void initialize(Configuration conf) throws IOException {
    ……
    super.initialize(conf);
    ……
}
NameNode.java
protected void initialize(Configuration conf) throws IOException {
    ……
    loadNamesystem(conf);
    ……
}
protected void loadNamesystem(Configuration conf) throws IOException {
    this.namesystem = new FSNamesystem(conf);
}
FSNamesystem.java
FSNamesystem(Configuration conf) throws IOException {
    try {
        initialize(conf, null);
    } catch(IOException e) {
        ……
    }
}
```

```
    throws IOException {
        if(fsImage == null) {
            ……
            this.dir.loadFSImage(getNamespaceDirs(conf),getNamespaceEditsD
irs(conf),
            startOpt);
            ……
        }
    }
```

Backup Node 启动过程调用的是 FSNamesystem 的 FSNamesystem(Configuration conf，BackupStorage bnImage)构造方法，所以在 FSNamesystem.initialize(Configuration conf，FSImage fsImage)中 fsImage 不为 null，this.dir.loadFSImage(getNamespaceDirsconf)未被调用，而在此方法中 FSDirectory 实例的全局变量 this.ready = true，并且只有此处 this.ready 被赋值为 true，其初值为 false。所以 BN 的生命周期中 this.ready 的值始终为 false，而 NameNode 调用的是 FSNamesystem 的 FSNamesystem(Configuration conf)构造方法，在 FSNamesystem.Initialize(Configuration conf，FSImage fsImage)中 fsImage 为 null，所以其 FSDirectory 实例的全局变量 this.ready 为 true。

大多数文件操作方法首先都会对 FSDirectory 类的 waitForReady()方法进行调用，因此，当 NameNode 无法正常服务时，客户端连接到 Backup Node 所进行的操作都会阻塞在 waitForReady()，而那些没有调用 waitForReady()的命令（如 mkdir）则能执行。

```
FSDirectory.java
void waitForReady() {
    if (!ready) {
        synchronized (this) {
            while (!ready) {
```

```
                this.wait(5000);
            } catch (InterruptedException ie) {}
        }
    }
}
```

4.3 实验方案说明

Backup Node 方案的优缺点在前面已作过详细分析，第 2 阶段的 Backup Node 方案在各个方面都要优于 Checkpoint Node 方案，完全可以替代 Checkpoint Node 方案，而第 3 阶段 Standby Node 还未实现，因此实验采用第 2 阶段的 Backup Node 方案。

> 在实验描述中，我们将尽量详细描述实验细节，便于各种层次读者能够准确地还原实验场景，读者可根据需要选择性阅读。

实验方案的思路如下：

首先，我们在前面理论分析的基础上，对 NameNode 各种可能的异常情况进行分析；然后提出相应的解决方案；最后，在实验环境中模拟各种异常情况，应用相应的解决方案进行验证。

具体实施上，我们首先将构建一个实验环境，其机器采用虚拟机来构建，这样便于部署与重现，实验环境的构建描述将尽量详细；然后，在该环境下模拟各种异常情况和操作场景，应用相应的解决方案；最后检查实验结果，验证解决方案的有效性。

4.4 构建实验环境

如前所述，实验环境采用虚拟机 VMware 进行构建，通常情况下，读者在一台

物理机器上就可以完成整个实验。

> 关于 VMware 的使用，请参考《虚拟智慧——VMware Vsphere 运维实录》，清华大学出版社。

4.4.1　网络拓扑

实验环境的网络拓扑如图 4.3 所示，包括一个 NameNode，一个 Backup Node，4 个 Data Node 以及 4 个 Client。

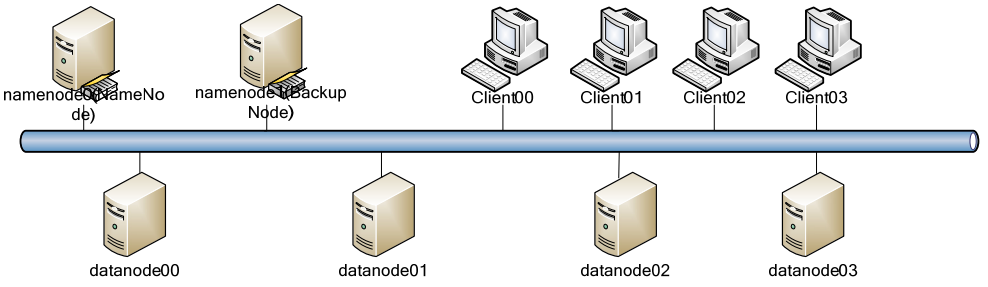

图 4.3　网络拓扑图

其中，每个 Client 和相应的 Data Node 共用一个虚拟机，因此，整个实验环境需要 6 个虚拟机，读者在具体部署时，可根据实际情况减少 Data Node 和 Client 的个数，但至少要各保持 1 个，具体配置如表 4.1 所示。

表 4.1 节点配置表

节点名	角 色	IP 地址	软件配置
namenode0	NameNode	真实 IP：192.168.1.11 虚拟 IP：192.168.1.9	Centos5.6 32 位 Ucarp-1.5.2 Hadoop-0.21.0 JDK:build 1.6.0_24-b07
namenode1	Backup Node 节点	真实 IP：192.168.1.12 虚拟 IP：192.168.1.9	同上
datanode00~ datanode03	DataNode 节点 客户端	192.168.1.13~192.168.1.16	Centos5.6 32 位 Hadoop-0.21.0 JDK:build 1.6.0_24-b07

4.4.2 系统安装及配置

本节我们将创建一个虚拟机，在虚拟机上安装 Linux 操作系统并进行相应配置，以此作为实验环境中的基准系统。虚拟机以文件的形式在磁盘上存储，因此，我们在安装其他节点的时候，就可以复制已经制作好的虚拟机文件，无需重新安装。

步骤描述如下。

（1）准备一台安装了 Windows 操作系统（最好是 Windows 7，可以利用 4GB 以上的内存）的主机，主机的 CPU 没有特殊要求，内存最好在 2GB 以上，硬盘在 160GB 以上。

> 提示：后续所有的实验都将在这台主机上运行。

（2）安装虚拟机软件 VMware 7，它可以在一台物理机器上创建多个虚拟机，对于用户来说，每一个虚拟机就如同一台真实的物理机器。

（3）使用 VMware 7 创建网络拓扑中的虚拟机。

（4）在创建的虚拟机上安装 Linux 操作系统，并进行相应的配置。

详细步骤描述如下。

1. 安装虚拟机

> 视频参见：\视频\4 BackupNode\1 vmware 安装流程
> 我们采用的虚拟机软件版本为 7.1，安装程序为：
> vmware-workstation-full-7.1.4-385536.exe

（1）双击安装程序。如图 4.4 所示。

（2）自定义安装。如图 4.5 所示。

图 4.4　安装进度界面

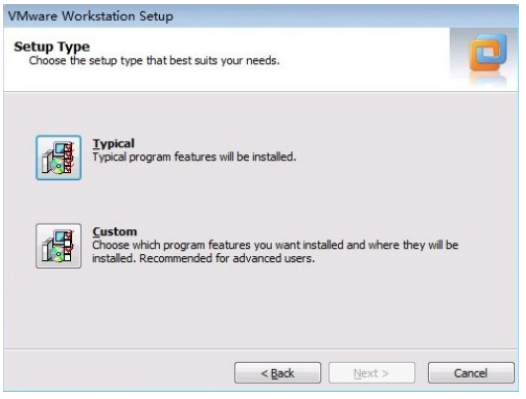

图 4.5　自定义安装界面

（3）选择安装路径。如图 4.6 所示。

（4）一路 next。如图 4.7 所示。

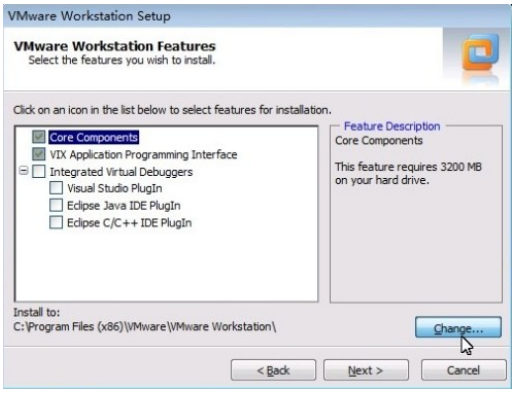

图 4.6　安装路径选择界面

图 4.7　安装选择界面

（5）开始安装。如图 4.8 所示。

（6）安装完毕。如图 4.9 所示。

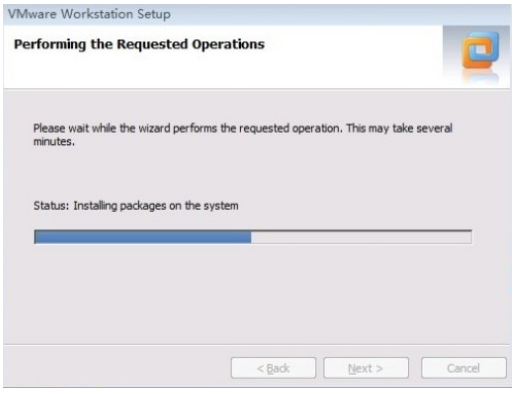

图 4.8　安装进度界面

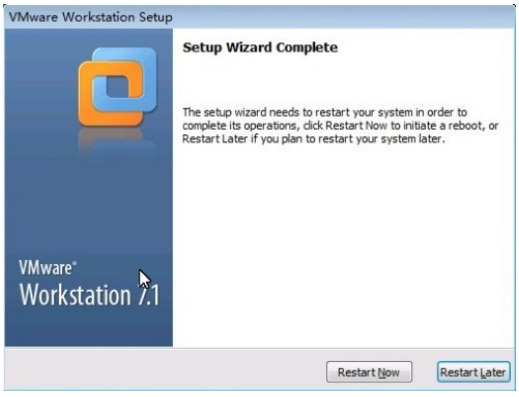

图 4.9　安装结束界面

（7）可以看到两个虚拟网卡 VMwnet1、VMwnet8 已经安装。如图 4.10 所示。

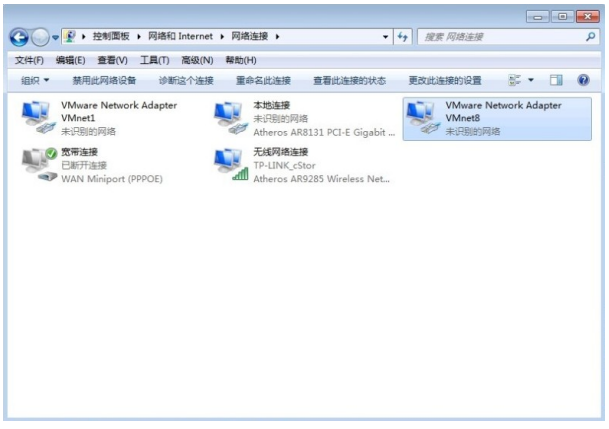

图 4.10　虚拟网卡

2. 创建虚拟机与安装操作系统

视频参见：\视频\4 Backup Node\2 CentOS5.6(32)安装流程.exe

（1）新建一个虚拟机，自定义安装。如图 4.11 所示。

（2）先设置虚拟机硬件环境，之后再安装 CentOS5.6。如图 4.12 所示。

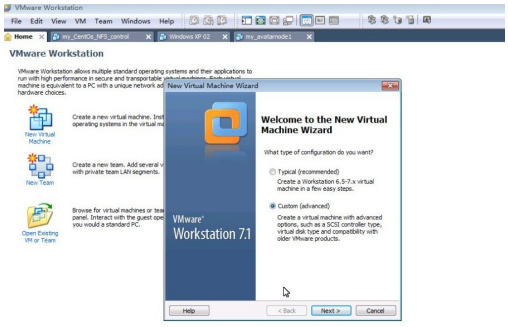

图 4.11　虚拟机自定义向导界面

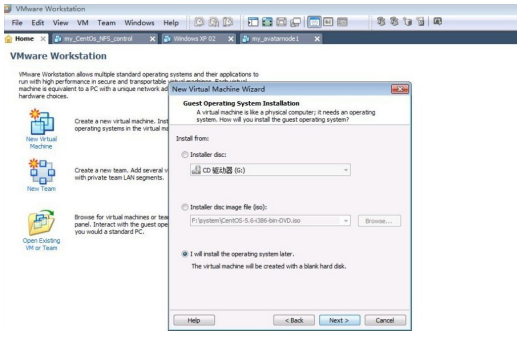

图 4.12　虚拟机硬件设置界面

（3）选择要安装的操作系统类型。如图 4.13 所示。

（4）选择虚拟机存储目录。用户创建的虚拟机的所有信息，如配置、系统信息、虚拟机磁盘内容等，都以文件的形式存储，在此，选择这些虚拟机文件的存储路径。如图 4.14 所示。

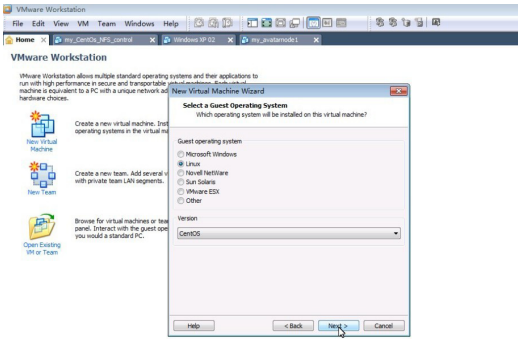

图 4.13　操作系统选择界面

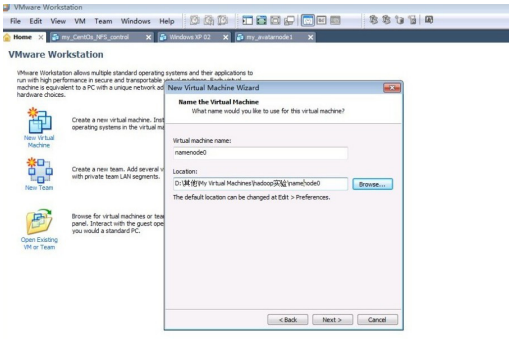

图 4.14　虚拟机存储目录选择界面

（5）设置虚拟机内存大小。如图 4.15 所示。

（6）设置虚拟机与 host 主机的网络连接方式。桥接（Bridge）方式下，虚拟机与 Host 主机之间通过物理的交换设备进行通信，也就是说，在该方式下 Host 主机的网卡必须连接到交换机；host-only 模式下，虚拟机和 Host 主机直接通信，Host 主机无需连接交换机。如图 4.16 所示。

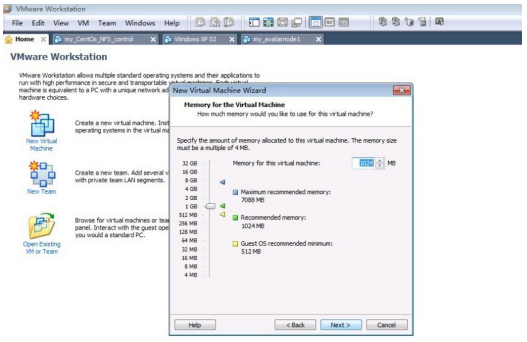

图 4.15　虚拟机内存设置界面

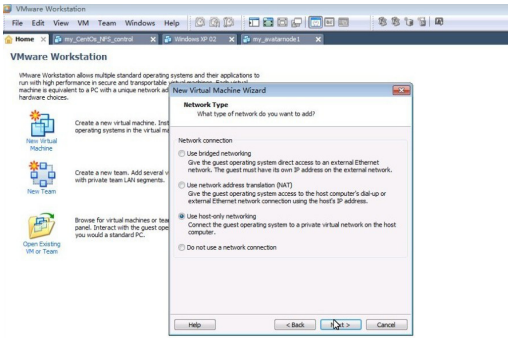

图 4.16　虚拟机网卡方式选择界面

（7）设置虚拟机引导启动的镜像文件。虚拟机可以使用 ISO 镜像文件作为其引导启动的文件，这就好比我们在使用物理机器时，将引导光盘插入光驱。如图 4.17 所示。

至此虚拟机创建完毕，我们可以在之前设置的“虚拟机存储目录”下看到虚拟机文件，接下来在创建好的虚拟机上安装 Linux 操作系统。

（8）启动虚拟机。虚拟机启动时，可以按 F2 键查看虚拟机的引导方式，确保从光盘引导。如果虚拟机从光盘镜像引导，将出现图 4.18 所示界面，按回车键进入下一步。

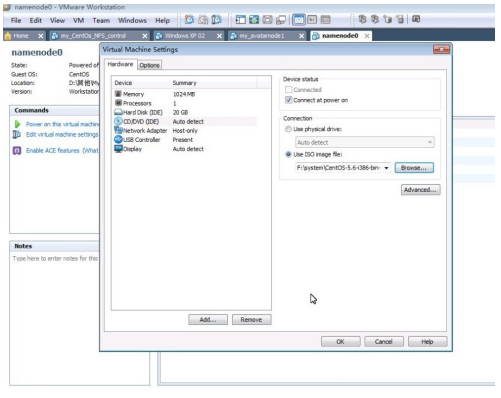

图 4.17　虚拟机镜像文件选择界面

图 4.18　centos 安装界面

（9）跳过 CD 检测。如图 4.19 所示。

（10）这里选择安装英文版本。如图 4.20 所示。

图 4.19　CD 检测界面

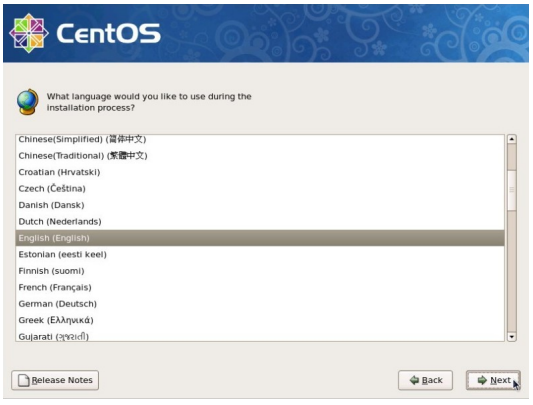

图 4.20　安装语言选择界面

（11）初始化磁盘。如图 4.21 所示。

（12）DHCP 方式动态获取 IP。如图 4.22 所示。在此先配置 IP 地址的获取方式为 DHCP。后续实验，我们将根据网络的实际情况来配置 IP 地址的获取方式。

图 4.21　磁盘分区界面

图 4.22　网络配置界面

（13）设置 root 用户的密码。如图 4.23 所示。

（14）根据需要选择要安装的软件包，在这里不要安装 JDK，其他一些软件包

可以都选上。如图 4.24 所示。

图 4.23　密码设置界面

图 4.24　软件安装包选择界面

（15）安装完毕，重启。如图 4.25 所示。

（16）CentOS 设置。如图 4.26 所示。

图 4.25　安装结束界面

图 4.26　设置界面

（17）自己做实验的话，可以关掉防火墙。如图 4.27 所示。

（18）创建普通用户，设置密码。在此，我们创建一个名为 user 的普通用户，密码设置为 123456。如图 4.28 所示。

图 4.27　防火墙配置界面

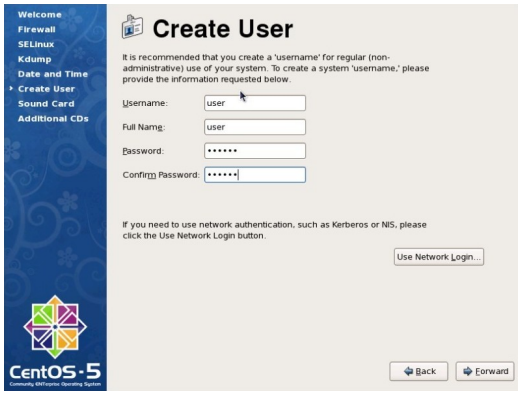

图 4.28　密码设置界面

（19）初步设置完毕，可以登录了。如图 4.29 所示。

（20）安装结束。在 Username 中输入用户名为 user，在接下来的密码框中输入之前设置好的密码 123456 就可以登录系统了。如图 4.30 所示。

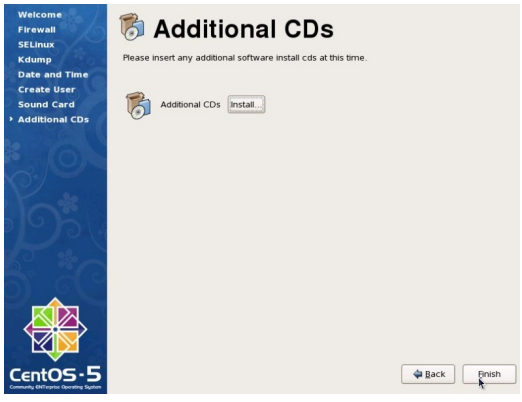

图 4.29　设置结束界面

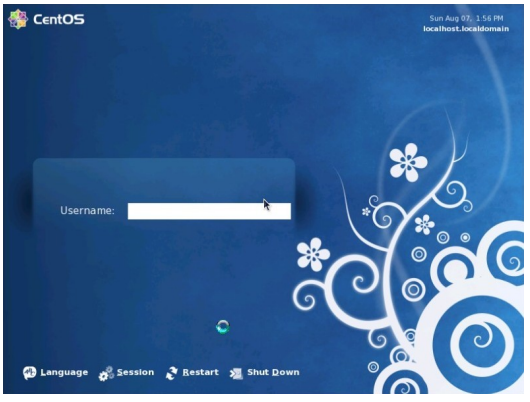

图 4.30　登录界面

3．配置操作系统

（1）vi 的基本使用。

视频参见：\视频\4 Backup Node\3 vi 的基本使用.exe

在配置操作系统之前，首先介绍一下 Linux 下最常用的文本编辑器 vi 的简单使

用方法，后续所有配置文件的修改都可以通过 vi 完成，使用说明如下。

① 在桌面上点击右键，选择 Open Terminal 启动一个终端。

② 在终端提示符“$”下输入“vi 文件名”，直接进入 vi 的命令模式。例如编辑一个名字为 test 的文件：

```
$vi test
```

- 按 A 键进入编辑模式，此时可进行字符的编辑与输入。
- 按 Esc 键退出编辑模式。
- 按 Shift+Z 组合键退出，就是按 Shift 键的同时按两下 Z 键保存退出，或者按 Shift+Q 组合键，进入命令模式，再输入 q，不保存退出。

（2）网络设置。

视频参见：\视频\4 Backup Node\4 CentOS5.6(32)设置流程.exe

将虚拟机的网卡设置为 Host-only 模式，这样虚拟机和 Host 主机可直接进行网络连接，而不需要借助外部设备。物理机器的虚拟网卡 VMwnet1 的 IP 设置为 192.168.1.1，以后建立的虚拟集群都在这个网段。

（3）初始化配置。

CentOS 刚安装完毕要进行初始化设置，在默认搜索路径 PATH 中加入一些自定义路径，这样使得一些基本命令可以直接使用，不用再输入绝对路径。

配置/etc/profile 文件，在文件末加入以下两行：

```
PATH=$PATH:/sbin                #在 PATH 变量后追加/sbin 目录
```

配置/home/user/.bash_profile 文件，将末尾处的 PATH 修改为：

```
PATH=$PATH:$HOME/bin:/sbin:/usr/sbin:usr/sbin:/usr/local/sbin:/usr/kerberos/sbin
```

（4）编辑/etc/sudoers 文件，使得普通用户可以以 root 权限执行命令。

- 添加/etc/sudoers 文件的写权限：

```
#chmod u+w /etc/sudoers
```

- 编辑/etc/sudoers 文件：

在 root ALL=(ALL) ALL 下面添加下面一行，这里的 user 是用户名。

```
user ALL=(ALL) ALL
```

- 撤销文件的写权限：

```
#chmod u-w /etc/sudoers。
```

（5）配置 IP 地址。

```
#vi /etc/sysconfig/network-scripts/ifcfg-eth0
```

设置为静态获取 IP，修改 BOOTPROTO、IPADDR 以及 NETMASK 的配置。

```
DEVICE=eth0
BOOTPROTO=static
HWADDR=00:0C:29:ED:51:9E
IPADDR=192.168.1.11
NETMASK=255.255.255.0
GATEWAY=192.168.1.1
ONBOOT=yes
```

（6）设置主机名（hostname）。

```
#vi /etc/sysconfig/network
```

将主机名设置为 namenode0，在后续其他节点的设置中，需要将主机名进行相应的修改。

```
HOSTNAME=namenode0
```

（7）编辑/etc/hosts。

在网络访问时，可使用相应的主机名来代替 IP 地址。

```
127.0.0.1    localhost
192.168.1.11 namenode0
192.168.1.12 namenode1
192.168.1.13 datanode00
192.168.1.14 datanode01
192.168.1.15 datanode02
192.168.1.16 datanode03
```

（8）重启虚拟机。

如果设置成功，重启后，相应的 sudo 权限、IP 地址、主机名、用户名等都将按要求修改了。

4.4.3 安装 JDK

虚拟机启动后，使用 SecureCRT 工具登录到虚拟机的 Linux 系统。

> SecureCRT 是一个 Windows 下的终端控制台图形程序，后续的操作都将在 SecureCRT 的终端下进行。我们使用的是免安装的 SecureCRT.exe，具体使用参见视频：\视频\4 Backup Node\5 JDK 安装.exe

JDK 安装步骤如下。

（1）准备 JDK 文件。实验中使用的 JDK 文件为 jdk1.6.0_24.tar，下载地址为：http://www.vdisk.cn/down/index/8993973A3659

（2）上传 JDK 文件。使用 SecureFX 工具上传 jdk1.6.0_24.tar 到 Linux 系统中/home/user/路径。

（3）解压。在文件管理器中双击压缩包，将其解压到当前路径。

（4）编辑/etc/profile 文件：

```
$sudo vi /etc/profile
```

加入以下几行：

```
export JAVA_HOME=/home/user/jdk1.6.0_24
export JRE_HOME=/home/user/jdk1.6.0_24/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH
```

（5）执行/etc/profile 中的内容：

```
$source /etc/profile
```

（6）查看 java 版本。如果可以看到 java 的版本信息，则说明安装成功。

```
$java -version
```

4.4.4　虚拟机集群架设

视频参见：\视频\4 Backup Node\6 虚拟集群架设流程.exe

1. 关闭不必要的启动服务

（1）查看开机启动的服务项。

```
$chkconfig --list
```

（2）关闭服务。在 SecureRCT 中，直接右键粘贴下列命令关闭下列服务，可有效提高开机速度。

```
echo "123456" | sudo -S chkconfig sendmail off
echo "123456" | sudo -S chkconfig bluetooth off
echo "123456" | sudo -S chkconfig NetworkManager off
echo "123456" | sudo -S chkconfig acpid off
echo "123456" | sudo -S chkconfig apmd off

echo "123456" | sudo -S chkconfig pand  off
```

```
echo "123456" | sudo -S chkconfig capi  off
echo "123456" | sudo -S chkconfig cups off
echo "123456" | sudo -S chkconfig iptables off
echo "123456" | sudo -S chkconfig ip6tables off
echo "123456" | sudo -S chkconfig irda off
echo "123456" | sudo -S chkconfig isdn off
echo "123456" | sudo -S chkconfig kudzu off
echo "123456" | sudo -S chkconfig lm_sensors off
echo "123456" | sudo -S chkconfig mdmonitor off
```

以上命令利用了管道命令“|”，将 root 用户的密码 123456 作为后续 sudo 命令的标准输入。因此，在执行 sudo 命令时，不再需要输入密码。

2. 上传软件包

准备需要上传的软件包，如表 4.2 所示。

表 4.2　软件包说明

软件包名称	说　明	获取位置
libpcap-0.9.4-14.el5.i386.rpm	Ucarp 依赖包	CentOS 安装光盘
libpcap-devel-0.9.4-14.el5.i386.rpm	Ucarp 依赖包	CentOS 安装光盘
ucarp-1.5.2-1.el5.i386.rpm	Ucarp 安装包	http://download.fedora.redhat.com/pub/epel/5/i386/ucarp-1.5.2-1.el5.i386.rpm
apache-ant-1.8.1.tar.gz	Ant 编译工具	http://www.vdisk.cn/down/index/8993876A5660
AvatarNode.20.patch	Avatar 补丁	https://issues.apache.org/jira/secure/attachment/12447060/AvatarNode.20.patch
facebook-hadoop-20-append-b6449e4.tar.gz	facebook 版 hadoop	http://www.vdisk.cn/down/index/8993830A5545
hadoop-0.20.2.tar.gz	hadoop 0.20.2 版	http://mirror.bjtu.edu.cn/apache/hadoop/common/hadoop-0.20.2/hadoop-0.20.2.tar.gz

续表

软件包名称	说　明	获取位置
hadoop-0.21.0.tar.gz	hadoop 0.21.0 版	http://mirror.bjtu.edu.cn/apache/hadoop/common/hadoop-0.21.0/hadoop-0.21.0.tar.gz
hbase-0.90.3.tar.gz	hbase 0.90.3 版	http://mirror.bjtu.edu.cn/apache/hbase/hbase-0.90.3/hbase-0.90.3.tar.gz
jdk1.6.0_24.tar.gz	JDK1.6	http://www.vdisk.cn/down/index/8993973A3659

使用 SecureFX 工具将其上传到虚拟机 Linux 的/home/user/soft 目录下。

3. 关闭 Linux

4. 关闭 VMware 程序

此时的虚拟机已作为实验的基准系统配置完毕，在后续的实验中，直接复制该虚拟机的镜像文件，修改配置，就可构建出其他节点。

5. 复制虚拟机镜像文件

将基准系统的镜像文件分别复制到 6 个不同的目录。

虚拟机集群需要较大的硬盘空间以及物理内存空间，实验中的物理机内存为 8GB，读者可根据自己的实验环境情况，调整虚拟机的个数或是调整虚拟机的配置。

注意：复制完一个，需要打开该虚拟机，修改虚拟机磁盘文件的存储路径，这点非常重要，不然虚拟机会将复制前的镜像文件作为它的硬盘。

4.4.5　NameNode 安装及配置

由于基准系统是按照 NameNode(namenode0)配置的，因此，复制过来的镜像文件基本无需修改，只需要注意修改 MAC 地址。

4.4.6　Backup Node 安装及配置

视频参见：\视频\4 Backup Node\6 虚拟机集群架设流程

（1）打开新的镜像文件。

（2）修改虚拟机名称。将虚拟机名称修改为 namenode1

（3）检查虚拟机硬盘的路径。

（4）启动系统。

（5）修改配置文件中的 IP 地址及 MAC 地址。

```
$sudo vi /etc/sysconfig/network-scripts/ifcfg-eth0
```

虚拟机镜像被复制后网卡的 MAC 地址会被重新分配，所以在复制后的 Linux 系统中，要修改其 MAC 地址，可以将 HWADDR=后的地址替换为新的网卡 MAC 地址，同时设置新的 IP 地址。

```
DEVICE=eth0
BOOTPROTO=static
HWADDR=00:0C:29:FC:22:F1
IPADDR=192.168.1.12
NETMASK=255.255.255.0
GATEWAY=192.168.1.1
ONBOOT=yes
```

（6）修改主机名。

```
$sudo vi /etc/sysconfig/network
```

将主机名设置为 namenode1：

```
HOSTNAME=namenode1
```

4.4.7 Data Node 安装及配置

> 具体操作参见 4.4.6 节“Backup Node 安装及配置”。

- 虚拟机名称为：datanode00～datanode03
- MAC 地址修改为自身虚拟机的 MAC 地址

- IP 地址为：192.168.1.12～192.168.1.16
- 主机名为：datanode00～datanode03

4.4.8　Clients 安装及配置

由于 Clients 与 Data Node 共用一个虚拟机，无需其他设置。

4.5　异常解决方案

4.5.1　异常情况分析

由图 4-1 网络拓扑图可知，HDFS 系统中的异常情况主要有：

- NameNode 无法提供服务
- Backup Node 无法提供服务

在总体架构上，我们采用 NameNode 与 Backup Node 进行元数据的实时同步，并且在 NameNode 与 Backup Node 上均进行元数据的保存，这样可以确保任何一方发生不可恢复性异常时，在另一方都有相应的备份；此外，采用 Ucarp 虚拟 IP 软件，可实现 IP 地址的自动切换，减少恢复时间。

具体来说，当 NameNode 无法提供服务时，由于 Backup Node 上的元数据和 NameNode 完全一致，因此，基于 Backup Node 的实现情况（目前无法提供热备），可以去掉损坏的 NameNode 节点，以 NameNode 角色重启 Backup Node，接替之前的 NameNode，然后准备一个新的节点，用作新的 Backup Node。

当 Backup Node 发生故障时，由于是 NameNode 对外提供服务，因此不会影响到整个系统的对外服务，只需准备一个新的节点，用作新的 Backup Node，重新注册到 NameNode 即可。

4.5.2　NameNode 配置

视频参见：\视频\4 Backup Node\7 hadoop-0.21.0 部署流程.exe

1. ssh 无密码登录本地或远程节点

HDFS 是一个集群系统，因此很多情况下需要通过 ssh 远程控制其他节点，将集群中的节点设置成无密码的 ssh 登录，便于实现远程节点的自动控制。

（1）namenode0 无密码登录本机。

```
$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
```

直接回车后会在~/.ssh/中生成两个文件：id_dsa 和 id_dsa.pub。这两个是成对出现，类似钥匙和锁。再把 id_dsa.pub 追加到授权 key 里面（当前并没有 authorized_keys 文件）。

```
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

完成后可以测试无密码访问本机。

```
$ ssh localhost hostname
```

执行以上命令，如果设置成功，将会自动显示本机的 Hostname。

（2）namenode0 无密码登录其他节点。

下面以 namenode0 无密码登录 datanode00 为例，其他节点间的无密码登录可以参照其步骤。

远程复制 namenode0 的 id_dsa.pub 文件到 datanode00 的/home/user/目录：

```
$ scp ~/.ssh/id_dsa.pub 192.168.1.13:/home/user/
```

登录 192.168.1.13 并执行如下命令，把 namenode0 上的 id_dsa.pub 文件追加到 datanode00 的 authorized_keys 内。

```
$ cat ~/id_dsa.pub >> ~/.ssh/authorized_keys
```

修改 datanode00 的 authorized_keys 权限，authorized_keys 权限应为 600，其父目录和祖父目录应为 755。

```
$ chmod 600 authorized_keys
```

关闭防火墙。

```
$ sudo ufw disable
```

> 注意：这步非常重要。如果不关闭，会出现找不到 datanode 的问题。

（3）验证。

重启 namenode0，在 namenode0 上执行。

```
$ ssh datanode00 hostname
```

如果设置成功，将会自动显示 datanode00 的 Hostname。

2. Ucarp 安装与配置

> 视频参见：\视频\4 Backup Node\7-1 ucarp 安装流程.exe

Ucarp 是一个虚拟 IP 软件，在 NameNode 和 Back Node 上均安装有 Ucarp。NameNode 启动的时候，将以 Ucarp 上配置的虚拟 IP 地址对外提供服务，一旦 NameNode 宕机，Back Node 上的 Ucarp 将自动接替 NameNode 上的虚拟 IP 对外提供服务，所有的一切对于访问虚拟 IP 的客户端来说是透明的。下面以在 namenode0(Name Node)上安装 Ucarp 为例进行说明。

（1）安装 rpm 包。

在 namenode1(Backup Node)上执行，Ucarp rpm 包的默认安装目录为 /usr/sbin/ucarp。

```
$sudo rpm -ivh ucarp-1.5.2-1.el5.i386.rpm
```

（2）复制 Ucarp 脚本。

将相关的三个 Ucarp 配置脚本/etc/ucarp.sh，/etc/vip-up.sh，/etc/vip-down.sh 复制到/etc/下。

/etc/ucarp.sh 为 Ucarp 的启动脚本。192.168.1.11 为 namenode0(NameNode)的真实 IP，192.168.1.9 为对外的虚拟 IP 。

> 如果在 namenode1（Backup Node）上安装 Ucarp，需要将“--srcip”后面的值修改为 namenode1（Backup Node）的真实 IP：192.168.1.12，这是这两个节点安装 Ucarp 的唯一区别。

```
#! /bin/sh
echo "123456" | sudo -S ucarp --interface=eth0 --srcip=192.168.1.11
--vhid=1 --pass=mypassword --addr=192.168.1.9 --preempt --neutral
```

其中，script=本机 IP。

在 Ucarp 中有 master 和 slave 两种角色，当节点竞争到 master 角色时，将执行/etc/vip-up.sh，在本实验中，NameNode 先启动 Ucarp，毫无疑问它将竞争到 master，/etc/vip-up.sh 脚本将给 NameNode 的网卡添加 IP 地址 192.168.1.9。

```
#! /bin/sh
echo "123456" | sudo -S /sbin/ip addr add 192.168.1.9/24 dev eth0
```

当 master 节点无法连接，将降级为 slave 节点，此时将执行/etc/vip-down.sh，去除原来添加的虚拟 IP 地址 192.168.1.9。

```
#! /bin/sh
echo "123456" | sudo -S /sbin/ip addr del 192.168.1.9/24 dev eth0
```

> 注意：123456 为用户密码，192.168.1.12 为物理 IP，192.168.1.9 为虚拟 IP，读者在实际使用中可以换成自定义的 IP。

3. 安装 Hadoop

Hadoop 的安装非常简单，将 Hadoop 的 tar 包解压后，进行相应配置即可，具体步骤描述如下。

（1）复制 Hadoop 安装包。

首先修改/usr/local 目录的权限，使得普通用户可以进行读写。

```
$sudo chmod 777 /usr/local
```

接下来在 user 用户下将/home/user/soft 目录下的 hadoop-0.21.0.tar.gz 复制到/usr/local 目录下。

（2）解压 Hadoop。

Hadoop 的解压目录为/usr/local，在文件管理器中进入该目录，双击 hadoop-0.21.0.tar.gz 即可完成解压。主要文件和目录如下所示。

- jar 包：Hadoop 运行所需的 class 文件打成的 jar 包。
- bin 目录：HDFS、MapReduce、balancer 工具的启动或关闭脚本。
- conf 目录：HDFS 和 MapReduce 的相关配置文件。
- common 目录：common 源码包。
- hdfs 目录：HDFS 源码包。
- mapred 目录：MapReduce 源码包。
- lib 目录：程序依赖的 jar 包。

（3）添加 Hadoop 路径信息。

在/etc/profile 末尾加入：

```
export HADOOP_HOME=/usr/local/hadoop-0.21.0
```

（4）配置$HADOOP_HOME/conf/下的 masters 和 slaves 文件。

master 配置文件

```
192.168.1.11
```

slaves 配置文件

```
192.168.1.13
192.168.1.14
192.168.1.15
```

```
192.168.1.16
```

（5）配置/usr/local/hadoop-0.21.0/conf/hadoop-env.sh。

在末尾加入下面一行。

```
export JAVA_HOME=/home/user/jdk1.6.0_24
```

（6）配置/usr/local/hadoop-0.21.0/conf/core-site.sh。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
       <name>fs.default.name</name>
       <value>hdfs://0.0.0.0:9000</value>
</property>
<property>
       <name>hadoop.tmp.dir</name>
       <value>/usr/local/hadoop/tmp</value>
</property>
</configuration>
```

（7）配置/usr/local/hadoop-0.21.0/conf/hdfs-site.sh。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<!-- special parameters for avatarnode -->
<configuration>
<property>
       <name>dfs.http.address</name>
```

```
</property>
<property>
       <name>dfs.name.dir</name>
       <value>/usr/local/hadoop/local/namenode</value>
</property>
<property>
       <name>dfs.name.edits.dir</name>
       <value>/usr/local/hadoop/local/editlog</value>
</property>
<property>
       <name>dfs.data.dir</name>

</property>
</configuration>
```

注意：fs.default.name 与 dfs.http.address 的 IP 均配置为 0.0.0.0，在本地所有 IP 地址上监听，客户端可以通过本机的实际 IP 和虚拟 IP 访问 Active NameNode。

（8）创建相关目录。

```
/usr/local/hadoop/tmp                    //hadoop 临时目录
/usr/local/hadoop/local/namenode         //镜像存储目录
/usr/local/hadoop/local/editlog          //日志存储目录
/usr/local/hadoop/block                  //数据块存储目录
```

3. 格式化

```
$/usr/local/hadoop-0.21.0/bin/hadoop namenode -format
```

4. 启动测试

运行下面的命令测试 NameNode(namenode0)能否正常启动。

```
$/usr/local/hadoop-0.21.0/bin/hadoop namenode
```

启动截图如 4.31 所示。

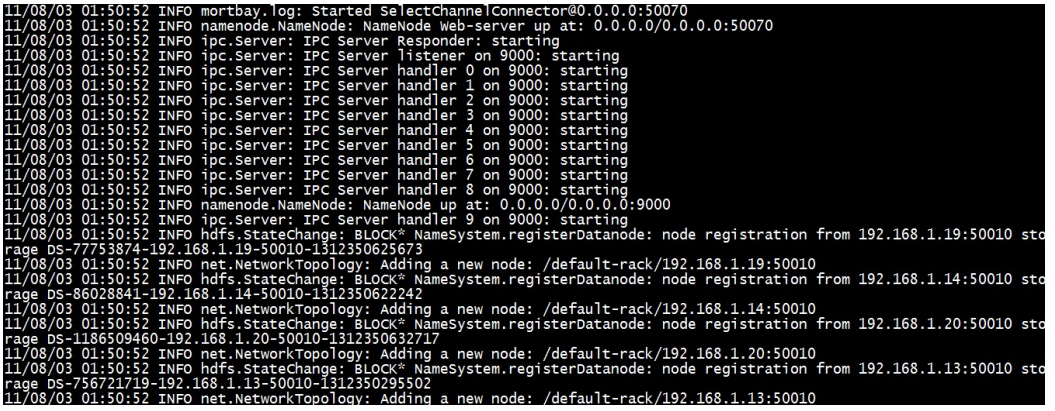

图 4.31　Active NameNode 启动图

使用 jps 命令查看 NameNode 的启动情况。

```
$jps
```

4.5.3　Backup Node 配置

1. ssh 无密码登录本地或远程节点

操作步骤请参考 4.5.2 中第 1 节“ssh 无密码登录本地或远程节点”。

2. Ucarp 安装与配置

Backup Node 上 Ucarp 的安装与配置与 NameNode 大致相同，唯一不同的地方需要将/etc/ucarp.sh 脚本--srcip 后面的值修改为 namenode1(Backup Node)的真实 IP：192.168.1.12 ，具体可参见 4.5.2 中“Ucarp 安装与配置”。

3. 安装 Hadoop

（1）远程复制 Hadoop。

从 NameNode(namenode0)远程复制 Hadoop 到 Backup Node(namenode1)。

首先修改/usr/local 目录的权限，使得普通用户可以进行读写。

```
$sudo chmod 777 /usr/local
```

接下来在 user 用户下将 NameNode 的 Hadoop 复制到 Backup Node。

```
$cd /usr/local
$scp -r hadoop-0.21.0 192.168.1.12:/usr/local/
```

（2）添加 Hadoop 路径信息，配置/etc/profile，在最后加入下面两行：

```
export HADOOP_HOME=/usr/local/hadoop-0.21.0
export PATH=$HADOOP_HOME/bin:$PATH
```

（3）配置 masters 和 slaves 文件。

masters 配置文件

```
192.168.1.12
```

slaves 配置文件

```
192.168.1.13
192.168.1.14
192.168.1.15
192.168.1.16
```

（4）配置/usr/local/hadoop-0.21.0/conf/core-site.sh。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
       <name>fs.default.name</name>
       <value>hdfs://192.168.1.11:9000</value>
</property>
<property>
       <name>hadoop.tmp.dir</name>
```

```
</property>
</configuration>
```

（5）配置/usr/local/hadoop-0.21.0/conf/hdfs-site.sh。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<!-- special parameters for avatarnode -->
<configuration>
<property>
       <name>dfs.http.address</name>
       <value>192.168.1.11:50070</value>
</property>
<property>
       <name>dfs.name.dir</name>
       <value>/usr/local/hadoop/local/namenode</value>
</property>
<property>
       <name>dfs.name.edits.dir</name>
       <value>/usr/local/hadoop/local/editlog</value>
</property>
<property>
       <name>dfs.data.dir</name>
       <value>/usr/local/hadoop/block</value>
</property>
```

注意：fs.default.name 与 dfs.http.address 的 IP 均配置为 NameNode 的 IP 地址 192.168.1.11，使得 Backup Node 可以访问 NameNode。

（6）创建相关目录。

```
/usr/local/hadoop/tmp                  //hadoop 临时目录
/usr/local/hadoop/local/namenode       //镜像存储目录
/usr/local/hadoop/local/editlog        //日志存储目录
```

4. 格式化

Backup Node 不应被格式化，否则会报 namespaceID 不一致的错误。

5. 启动测试

在 Backup Node(namenode1)的终端上执行。

```
$/usr/local/hadoop-0.21.0/bin/hadoop namenode -backup
```

启动过程如图 4.32 所示。

```
11/08/09 08:57:51 INFO namenode.NameNode: Backup Node Web-server up at: namenode1/192.168.1.12:50105
11/08/09 08:57:51 INFO ipc.Server: IPC Server Responder: starting
11/08/09 08:57:51 INFO ipc.Server: IPC Server listener on 50100: starting
11/08/09 08:57:51 INFO ipc.Server: IPC Server handler 0 on 50100: starting
11/08/09 08:57:51 INFO ipc.Server: IPC Server handler 1 on 50100: starting
11/08/09 08:57:51 INFO ipc.Server: IPC Server handler 2 on 50100: starting
11/08/09 08:57:51 INFO ipc.Server: IPC Server handler 3 on 50100: starting
11/08/09 08:57:51 INFO ipc.Server: IPC Server handler 4 on 50100: starting
11/08/09 08:57:51 INFO ipc.Server: IPC Server handler 5 on 50100: starting
11/08/09 08:57:51 INFO ipc.Server: IPC Server handler 6 on 50100: starting
11/08/09 08:57:51 INFO ipc.Server: IPC Server handler 7 on 50100: starting
11/08/09 08:57:51 INFO ipc.Server: IPC Server handler 8 on 50100: starting
11/08/09 08:57:51 INFO namenode.NameNode: Backup Node up at: namenode1/192.168.1.12:50100
11/08/09 08:57:51 INFO ipc.Server: IPC Server handler 9 on 50100: starting
11/08/09 08:57:51 INFO namenode.Checkpointer: Checkpoint Period : 3600 secs (60 min)
11/08/09 08:57:51 INFO namenode.Checkpointer: Log Size Trigger  : 67108864 bytes (65536 KB)
11/08/09 08:57:52 INFO common.Storage: Number of files = 187
11/08/09 08:57:52 INFO common.Storage: Number of files under construction = 0
11/08/09 08:57:52 INFO common.Storage: Edits file /usr/local/hadoop/local/editlog/current/edits of siz
e 4 edits # 0 loaded in 0 seconds.
11/08/09 08:57:52 INFO namenode.FSNamesystem: Number of transactions: 0 Total time for transactions(ms
): 0Number of transactions batched in Syncs: 0 Number of syncs: 1 SyncTimes(ms): 6
11/08/09 08:57:52 INFO common.Storage: Image file of size 18069 saved in 0 seconds.
11/08/09 08:57:52 INFO namenode.Checkpointer: Posted URL namenode0:50070putimage=1&port=50105&machine=
192.168.1.12&token=-24:1015070777:0:1312894668000:1312848921092
11/08/09 08:57:53 INFO common.Storage: Edits file /usr/local/hadoop/local/editlog/jspool/edits.new of
size 4 edits # 0 loaded in 0 seconds.
11/08/09 08:57:53 INFO namenode.Checkpointer: Checkpoint completed in 1 seconds. New Image Size: 18069
```

图 4.32　Backup Node 启动图

4.5.4　Data Node 配置

以 DataNode00 为例说明，DataNode01～DataNode03 以及 Client00～Client03

配置同 DataNode00。

1. ssh 无密码登录本地或远程节点

操作步骤请参考 4.5.2 中“ssh 无密码登录本地或远程节点”相关内容。

2. 安装 Hadoop

（1）远程复制 Hadoop。

从 NameNode(namenode0)远程复制 Hadoop 到 DataNode00。

首先修改/usr/local 目录的权限，使得普通用户可以进行读写。

```
$sudo chmod 777 /usr/local
```

接下来在 user 用户下将 NameNode 的 Hadoop 复制到 DataNode00。

```
$cd /usr/local
```

（2）添加 Hadoop 路径信息，配置/etc/profile，在最后加入下面两行：

```
export HADOOP_HOME=/usr/local/hadoop-0.21.0
export PATH=$HADOOP_HOME/bin:$PATH
```

（3）配置 masters 和 slaves 文件。

masters 配置文件

```
192.168.1.12
```

slaves 配置文件

```
192.168.1.13
192.168.1.14
192.168.1.15
192.168.1.16
```

（4）配置/usr/local/hadoop-0.21.0/conf/core-site.sh。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
       <name>fs.default.name</name>
       <value>hdfs://192.168.1.9:9000</value>
</property>
<property>
       <name>hadoop.tmp.dir</name>
       <value>/usr/local/hadoop/tmp</value>
</property>
</configuration>
```

（5）配置/usr/local/hadoop-0.21.0/conf/hdfs-site.sh。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
       <name>dfs.http.address</name>
       <value>192.168.1.9:50070</value>
</property>
<property>
       <name>dfs.name.dir</name>
       <value>/usr/local/hadoop/local/namenode</value>
</property>
<property>
       <name>dfs.name.edits.dir</name>
```

```
</property>
<property>
        <name>dfs.data.dir</name>
        <value>/usr/local/hadoop/block</value>
</property>
```

> 注意：fs.default.name 与 dfs.http.address 的 IP 均配置为虚拟 IP 地址 192.168.1.9，当 NameNode 宕机后，Backup Node 节点会接替 NameNode 的虚拟 IP 地址，这些对于 Data Node 是透明的，因此，NameNode 发生故障时，DataNode 无需作任何修改。

（6）创建相关目录。

```
/usr/local/hadoop/tmp                  //hadoop 临时目录
/usr/local/hadoop/local/namenode       //镜像存储目录
/usr/local/hadoop/local/editlog        //日志存储目录
/usr/local/hadoop/block                //数据块存储目录
```

3. 启动测试

执行下面的命令，启动 DataNode。

```
$/usr/local/hadoop-0.21.0/bin/hadoop datanode
```

结果如图 4.33 所示。

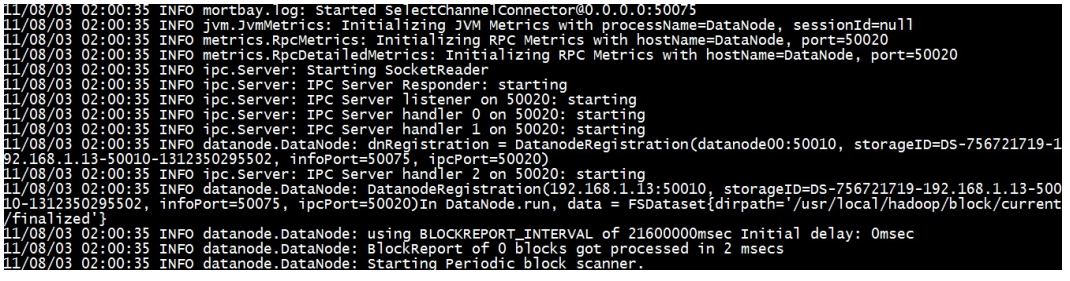

图 4.33　Data Node 启动图

4.5.5　NameNode 宕机切换实验

当运行 NameNode 的节点宕机时，需要在运行 Backup Node 的节点上终止 Backup Node，修改配置文件后，将 Backup Node 以 NameNode 方式启动，此时虚拟 IP 将被此节点获得，集群中的 DataNode 不需改变配置文件，也不需要重启就可以到运行在此节点之上的 NameNode 注册，之后整个集群就恢复正常。

此时，可以在一个新节点上启动 Backup Node 作为备份。具体步骤如图 4.34 所示。

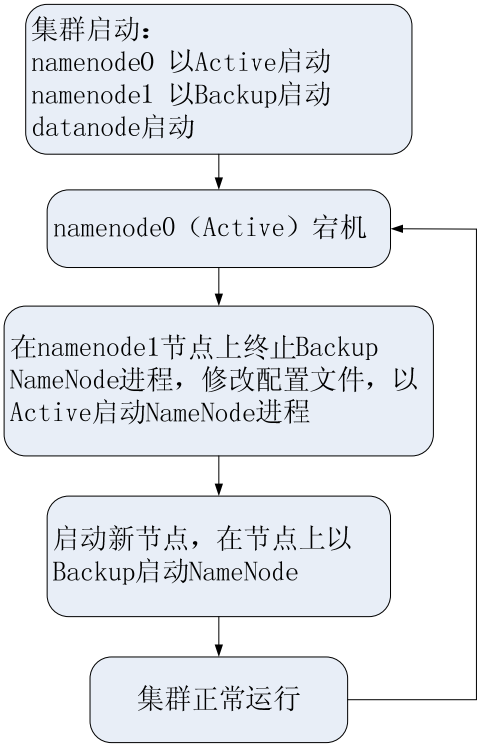

图 4.34　HDFS 集群启动流程

视频参见：\视频\4 Backup Node\8 hadoop-0.21.0 切换实验.exe

1. 启动虚拟机

依次启动 namenode0、namenode1、datanode00、datanode01、datanode02 和 datanode03。使用 SecureCRT 分别登录到各节点。

2. Hadoop 集群测试

（1）启动 Hadoop。在 namenode0(NameNode)上执行，该脚本将启动 namenode0，datanode00～datanode03。

```
$cd /usr/local/hadoop-0.21.0
$bin/start-dfs.sh
```

依次在 namenode0、datanode00～datanode03 上检查节点是否正常运行。

```
$jps
```

（2）关闭 Hadoop。

```
$bin/stop-dfs.sh
```

依次在 namenode0，datanode00～datanode03 上检查节点是否正常运行。

```
$jps
```

3. 启动虚拟 IP

在 namenode0(NameNode)上执行下列命令，namenode0 将获取到 Ucarp 的 master 角色，添加虚拟 IP 地址 192.168.1.9 对外服务。

```
$nohup /etc/ucarp.sh &
```

测试虚拟 IP 是否工作，如显示 namenode0，则说明 namenode0(NameNode)的虚拟 IP 已开始工作。

```
$ssh 192.168.1.9 hostname
```

在 namenode1(Backup Node)上执行下列命令，此时 namenode1(Backup Node)将获得 Ucarp 的 slave 角色，当 master 无法工作时，namenode1(Backup Node)上

的 Ucarp 将调用/etc/vip-up.sh 脚本，为本机添加虚拟 IP 地址 192.168.1.9。

```
$nohup /etc/ucarp.sh &
```

4. 启动 Hadoop

在 namenode0(NameNode)上执行，该脚本将启动 namenode0，datanode00～datanode03。

```
$cd /usr/local/hadoop-0.21.0
$bin/start-dfs.sh
```

依次在 namenode0，datanode00～datanode03 上检查节点是否正常运行。

```
$jps
```

查看集群状态，在 namenode0(NameNode)上执行：

```
$cd /usr/local/hadoop-0.21.0
$bin/hadoop dfsadmin -report
```

正常情况下，可以看到 4 个 Data Node 都启动了。

5. 启动 Backup Node

在 namenode1(Backup Node)上执行，该命令可以将 Backup Node 的日志输出为文件。

```
$cd /usr/local/hadoop-0.21.0
```

6. 查看 NameNode 与 Backup Node 元数据是否一致

（1）查看 NameNode 上的元数据。

在 namenode0(NameNode)上执行，根据配置，该命令将访问 namenode0(NameNode)上的元数据，以获取 HDFS 信息。

```
$bin/hadoop dfs -lsr /
```

（2）查看 Backup Node 上的元数据。

在 namenode1(Backup Node)执行，根据配置，该命令将访问 namenode1(Backup Node)上的元数据，以获取 HDFS 信息。

```
$bin/hadoop dfs -D fs.default.name=hdfs://192.168.1.12:50100 -lsr /
```

可以看到两个客户端命令所列举的 HDFS 信息完全相同，namenode0(NameNode)和 namenode1(Backup Node)元数据一致。

（3）复制文件。

前面步骤（1）、（2）验证了 NameNode 和 Backup Node 的元数据一致，下面再验证元数据的动态同步。

在 namenode0(NameNode)上执行客户端命令，将当前目录下所有 txt 结尾的文件复制到 HDFS 的 111 目录下，该命令更新的是 namenode0(NameNode)上的元数据信息。

```
$bin/hadoop dfs -copyFromLocal *.txt 111
$bin/hadoop dfs -lsr /
```

查看 Backup Node 上的元数据是否同步。

```
$bin/hadoop dfs -lsr /
```

可以看到 namenode0(NameNode)和 namenode1(Backup Node)元数据一致。

7. 模拟 NameNode 无法服务

关闭 namenode0(NameNode)。

```
$sudo shutdown -h now
```

8. Backup Node 切换准备

尽管此时 Backup Node 的内存元数据是与 NameNode 的内存元数据同步的，而且虚拟 IP 也可以切换过来，但是，由于第 2 阶段的 Backup Node 还不支持热备

（具体原因参见 4.2.4“故障切换机制”），因此，需要以重启的方式来恢复服务。namenode1(Backup Node)之前是配置成 Backup Node 启动的，现在角色变换，需要以 NameNode 启动，因此需要修改如下配置文件。

（1）修改 core-site.xml。

在 namenode1(Backup Node)上执行。

```
$vi conf/core-site.xml
```

将 fs.default.name 的值修改为 hdfs://0.0.0.0:9000，在所有 IP 地址上监听，由于 192.168.1.9 是其中的一个 IP，因此可以接收所有访问 192.168.1.9 的请求，如来自客户端或 Data Node 的请求。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
       <name>fs.default.name</name>
       <value>hdfs://0.0.0.0:9000</value>
</property>
<property>
       <name>hadoop.tmp.dir</name>
       <value>/usr/local/hadoop/tmp</value>
</property>
```

（2）修改 hdfs-site.xml。

将 dfs.http.address 的值修改为 0.0.0.0:50070，在所有 IP 地址上监听，这样外部通过虚拟 IP：192.168.1.9 就可以访问到 namenode1 的 Web。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
       <name>dfs.http.address</name>
       <value>0.0.0.0:50070</value>
</property>
<property>
       <name>dfs.name.dir</name>
       <value>/usr/local/hadoop/local/namenode</value>
</property>
<property>
       <name>dfs.name.edits.dir</name>
       <value>/usr/local/hadoop/local/editlog</value>
</property>
<property>
       <name>dfs.data.dir</name>
       <value>/usr/local/hadoop/block</value>

</configuration>
```

9. 关闭 namenode1(Backup Node)的 NameNode 进程

在 namenode1(Backup Node)上执行，查看 NameNode 的进程号。

```
$jps
```

杀死进程。

```
$kill -9 NameNode 的进程号
```

10. 以 NameNode 模式启动 namenode1 节点

在 namenode1 上执行。

```
$cd /usr/local/hadoop-0.21.0
$bin/hadoop-daemon.sh start namenode
```

查看 HDFS。

```
$bin/hadoop dfs -lsr /
```

可以看到，所有文件都在，之前的元数据并未丢失。

写操作测试。

```
$bin/hadoop dfs -copyFromLocal conf/*.xml 111
```

可以看到 namenode1 切换过来后，读写操作可顺利进行。

查看集群状态。

```
$bin/hadoop dfsadmin -report
```

正常情况下，可以看到 4 个 Data Node 都注册上来了。

11. 启动新的 Backup Node

选择之前的 namenode0 作为新的 Backup Node 启动，在 namenode0 上执行。

（1）重启 namenode0。

（2）启动虚拟 IP。

```
$nohup /etc/ucarp.sh &
```

测试虚拟 IP 是否工作。

```
$ssh 192.168.1.9 hostname
```

正常情况下，虚拟 IP 应指向 namenode1。

（3）修改配置文件。

修改 core-site.xml。

```
$vi conf/core-site.xml
```

将 fs.default.name 的值修改为 hdfs://192.168.1.12:9000。

在 namenode0 上修改 hdfs-site.xml。

```
$vi conf/hdfs-site.xml
```

将 dfs.http.address 的值修改为 hdfs://192.168.1.12:50070。

（4）启动。

```
$bin/hadoop-daemon.sh start namenode -backup
```

（5）查看 namenode0(Backup Node)的元数据。

```
$bin/hadoop dfs -D fs.default.name=hdfs://192.168.1.11:50100 -lsr /
```

可以看到元数据已同步。

4.5.6 NameNode 宕机读写测试

当 NameNode 宕机时，需要在运行 Backup Node 的节点上终止 Backup Node，修改配置文件后以 NameNode 方式启动。NameNode（原 Backup Node）启动后将获得虚拟 IP，集群中的 DataNode 无需改动配置，也不需要重启就可以到新的 NameNode 注册，整个集群恢复正常，接下来，选择一个新的节点启动 Backup Node。本节实验模拟以上场景，测试其对读写操作的影响。具体步骤如下。

视频参见：\视频\4 Backup Node\9 hadoop-0.21.0 数据读写实验.exe

1. 启动虚拟机

依次启动 namenode0、namenode1、datanode00、datanode01、datanode02 和 datanode03。使用 SecureCRT 分别登录到各节点。如果节点已经启动，则重启

节点，恢复到初始状态。

2. 启动虚拟 IP

（1）在 NameNode(namenode0)上执行。

```
$ nohup /etc/ucarp.sh &
```

Ucarp 启动后，NameNode(namenode0)将增加一个虚拟 IP（192.168.1.9）对外工作，在其他节点上，访问 192.168.1.9 可直接访问到 NameNode(namenode0)。

在 datanode00 终端上运行。

```
$ ssh 192.168.1.9 hostname
```

运行结果为 namenode0，说明通过虚拟 IP（192.168.1.9）直接访问到了 NameNode(namenode0)。

（2）在 Backup Node(namenode1)上执行。

```
$ nohup /etc/ucarp.sh &
```

当 NameNode(namenode0)无法访问时，Backup Node(namenode1)将接替 NameNode(namenode0)的虚拟 IP（192.168.1.9）对外服务。

3. 启动 Hadoop

（1）启动 HDFS 集群。

在 NameNode(namenode0)的终端上执行。

```
$/usr/local/hadoop-0.21.0/bin/start-dfs.sh
```

查看启动情况。

```
$/usr/local/hadoop-0.21.0/bin/hadoop dfsadmin -report
```

HDFS 集群状态如图 4.35 所示。

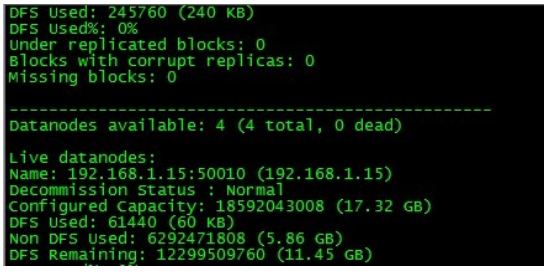

图 4.35　HDFS 集群状态图

（2）启动 Backup Node(namenode1)。

在 Backup Node(namenode1)的终端上执行，该命令可使得日志输出到文件。

```
$/usr/local/hadoop-0.21.0/bin/hadoop-daemon.sh start namenode -backup
```

查看启动情况。

```
$jps
```

查看日志。

```
$ls -l /usr/local/hadoop-0.21.0/logs
```

（3）检查 HDFS。

在 NameNode(namenode0)上执行，通过 NameNode(namenode0)上的元数据，查询 HDFS 的情况。

```
$/usr/local/hadoop-0.21.0/bin/hadoop dfs -lsr /
```

可以看到在/tmp 目录下有没有 192.168.1.13～192.168.1.164 四个目录，如果没有，则手工创建。

在 Backup Node(namenode1)上执行：

```
$/usr/local/hadoop-0.21.0/bin/hadoop dfs -D
```

查询 Backup Node(namenode1)上的元数据信息。可以看到两者显示一样，说

明元数据已经同步。

4. 启动写操作

调用脚本，从 datanode00～datanode03 四个客户端节点向 HDFS“/tmp”下的 192.168.1.13～192.168.1.164 四个目录中写入文件，每个客户端写入 50 个文件，编号为 10000～10050，每个文件大小为 5MB，写入间隔 10s。

在 datanode00～datanode03 上依次执行：

```
$/usr/local/hadoop-0.21.0/bin/write_to_hdfs.sh 50 50000000 10
```

注意：write_to_hdfs.sh 脚本源码参见 4 Backup Node\write_to_hdfs.sh，其中调用/usr/local/hadoop-0.21.0/bin/makefile 程序用于创建文件。

5. 准备切换

由于下面将模拟 NameNode 无法正常服务，在此，修改 Backup Node(namenode1)的配置，为其切换为 NameNode 做准备。

（1）修改 hdfs-site.xml。

将 dfs.http.address 的值修改为 0.0.0.0:50070，在所有 IP 地址上监听，这样外部通过虚拟 IP（192.168.1.9）就可以访问到 namenode1 的 Web。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
       <name>dfs.http.address</name>
       <value>0.0.0.0:50070</value>
</property>
<property>
       <name>dfs.name.dir</name>
```

```
</property>
<property>
        <name>dfs.name.edits.dir</name>
        <value>/usr/local/hadoop/local/editlog</value>
</property>
<property>
        <name>dfs.data.dir</name>
        <value>/usr/local/hadoop/block</value>
</property>
</configuration>
```

（2）修改 core-site.xml。

```
$vi conf/core-site.xml
```

将 fs.default.name 的值修改为 hdfs://0.0.0.0:9000，在所有 IP 地址上监听，192.168.1.9 是其中的一个 IP，因此可以接收所有访问 192.168.1.9 的请求，如来自客户端或 Data Node 的请求。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
        <name>fs.default.name</name>
        <value>hdfs://0.0.0.0:9000</value>
</property>
<property>
        <name>hadoop.tmp.dir</name>
        <value>/usr/local/hadoop/tmp</value>
</property>
```

6. 模拟 NameNode 无法提供服务

重启 NameNode(namenode0)之前，先查看一下 HDFS 上文件的写入情况，在 NameNode(namenode0)上执行：

```
$/usr/local/hadoop-0.21.0/bin/hadoop dfs -lsr /
```

可以看到，此时每个客户端的 50 个文件并未全部写完，说明此时还在进行写入操作。

重启 NameNode(namenode0)。

```
$sudo reboot
```

7. 切换 Backup Node

（1）关闭 Backup Node(namenode1)上的 NameNode 进程。

在 Backup Node(namenode1)上执行：

```
$jps
```

查看 NameNode 的进程号，将其 Kill 掉。

（2）在 Backup Node(namenode1)上启动新的 NameNode。

```
$/usr/local/hadoop-0.21.0/bin/hadoop-daemon.sh start namenode
```

（3）查看集群启动情况。

```
$/usr/local/hadoop-0.21.0/bin/hadoop dfsadmin -report
```

（4）查看 HDFS。

```
$/usr/local/hadoop-0.21.0/bin/hadoop dfs -lsr /
```

连续执行上面的命令，可以看到有新的文件创建，说明切换成功，等待文件写入结束，再次查看，发现一些文件没有成功复制到 HDFS 中，以本次实验为例：

```
192.168.1.13  10013~10017
```

```
192.168.1.14  10013~10016
192.168.1.15  10013~10016
192.168.1.16  10013~10016
```

以上文件的操作发生在切换期间，因此需要有额外的机制来确保其写入成功。

8. 启动新的 Backup Node

选择刚才重启的 namenode0 节点作为新的 Backup Node。

（1）启动虚拟 IP。

```
$nohup /etc/ucarp.sh &
```

测试虚拟 IP 是否工作。

```
$ssh 192.168.1.9 hostname
```

正常情况下，虚拟 IP 应指向 namenode1。

（2）修改配置文件。

修改 hdfs-site.xml。

```
$vi conf/hdfs-site.xml
```

将 dfs.http.address 的值修改为 hdfs://192.168.1.12:50070。

修改 core-site.xml。

```
$vi conf/core-site.xml
```

将 fs.default.name 的值修改为 hdfs://192.168.1.12:9000。

（3）启动 Backup Node。

```
$bin/hadoop-daemon.sh start namenode -backup
```

（4）查看 namenode0(Backup Node)的元数据。

```
$bin/hadoop dfs -D fs.default.name=hdfs://192.168.1.11:50100 -lsr /
```

可以看到元数据已同步。

9. 启动读操作

调用脚本 write_to_hdfs.sh，从 datanode00～datanode03 四个客户端节点向 HDFS“/tmp”下的 192.168.1.13～192.168.1.16 四个目录中写入文件，每个客户端写入 50 个文件，编号为 10000～10050，每个文件大小为 5MB，写入间隔 10s。

在 datanode00~datanode03 客户端调用脚本 read_from_hdfs_IP.sh（脚本源码参见 4 Backup Node\read_from_hdfs_IP.sh）从 HDFS 上复制对应目录下的文件到本地的/tmp/192.168.1.1x 目录下。在 datanode00～datanode03 上执行：

```
$nohup /usr/local/hadoop-0.21.0/bin/read_from_hdfs_IP.sh 192.168.1.13 &
$nohup /usr/local/hadoop-0.21.0/bin/read_from_hdfs_IP.sh 192.168.1.14 &
$nohup /usr/local/hadoop-0.21.0/bin/read_from_hdfs_IP.sh 192.168.1.15 &
$nohup /usr/local/hadoop-0.21.0/bin/read_from_hdfs_IP.sh 192.168.1.16 &
```

查看执行情况。

```
$ps -an
```

查看复制情况。

```
$ls -l /tmp/192.168.1.13
$ls -l /tmp/192.168.1.14
$ls -l /tmp/192.168.1.15
```

10. 准备切换

此次切换是将 namenode0(Backup Node)切换为 NameNode。在 namenode0 上修改配置文件。

（1）修改 hdfs-site.xml。

```
$vi conf/hdfs-site.xml
```

将 dfs.http.address 的值修改为 hdfs://0.0.0.0:50070。

（2）修改 core-site.xml。

```
$vi conf/core-site.xml
```

将 fs.default.name 的值修改为 hdfs://0.0.0.0:9000。

11. 模拟 NameNode(namenode1)无法服务

在 namenode1 上执行：

```
$sudo reboot
```

12. 切换 Backup Node

具体操作参见步骤 7，在 namenode0 上执行。

在 datanode00 上查看复制情况，可以看到有新的文件复制到本地，说明切换成功，但也有部分文件没有复制成功，以本次实验为例。

```
192.168.1.13  10016~10025
192.168.1.14  10015~10024
192.168.1.15  10013~10023
192.168.1.16  10015~10024
```

以上未成功的文件发生在切换期间，需要其他的机制来确保其复制成功。

13. 启动新的 Backup Node

选择刚才重启的 namenode1 节点作为新的 Backup Node，具体操作参见步骤 9，在 namenode1 上执行。

第 5 章　AvatarNode 运行机制

NameNode 是一个单一故障点，而 Hadoop 自身的 HA 机制恢复耗时较长，以第 2 阶段的 Backup Node 为例，一个 1.5 亿个文件规模的 HDFS 系统，NameNode 重启恢复服务至少需要 45 分钟[1]，即使是将来使用第 3 阶段 Backup Node，HDFS 恢复正常的时间也至少需要 20 分钟[1]，这对于 24 × 7 uptime 的应用来说是难以接受的，在这样的前提下，Facebook 提出并实现了 HDFS 的热备（Hot standby）机制 AvatarNode，力图达到秒级的迁移（Fail over）时间。

---

[1] Apache Hadoop Goes Realtime at Facebook. Dhruba Borthakur and Joydeep Sen Sarma and Jonathan Gray. Sigmod 2011

5.1 方案说明

5.1.1 系统架构

AvatarNode 的系统架构如图 5.1 所示，它包括一个 Primary AvatarNode、一个 Standby AvatarNode、一个 NFS 服务器、多个 Data Nodes 以及多个 Clients。

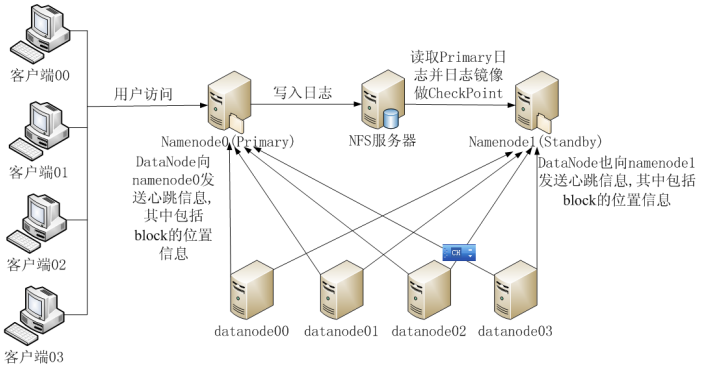

图 5.1 系统架构图

- Primary AvatarNode 是对外提供服务的 NameNode；
- Standby AvatarNode 也运行一个 NameNode 进程，与 Primary AvatarNode 的内存元数据保持同步，当 Primary AvatarNode 无法对外服务时，它将接替 Primary AvatarNode 对外服务；
- Primary AvatarNode 和 Standby AvatarNode 通过 NFS 服务器进行元数据同步，Primary AvatarNode 向 NFS 共享目录写入日志记录，Standby AvatarNode 定期读入 NFS 共享目录中的日志记录至内存进行合并，NFS 协议的自身机制可以确保 Standby AvatarNode 在接替 Primary AvatarNode 时，内存中的元数据与之前 Primary AvatarNode 的内存元数据完全一致；
- Data Nodes 将 Primary AvatarNode 和 Standby AvatarNode 视之为两个 NameNode，将分别向上报 Block 信息等；

- Clients 是 Facebook 提供的 AvatarNode 客户端。

AvatarNode 在设计和实现上有许多值得借鉴的地方，它们确保了 AvatarNode 机制的高效可靠。

首先 AvatarNode 采用了 Standby AvatarNode 与 Primary AvatarNode 进行元数据同步，同时 Data Nodes 分别向两个 AvatarNode 上报同样的心跳、Block 位置等信息，这样可以确保 Standby AvatarNode 的内存元数据与 Primary AvatarNode 完全一致，因此切换成功后便可对外服务，切换时间短。

其次 AvatarNode 使用 NFS 作为元数据同步的共享存储，NFS 在理论上是一个新的单一故障点，但在实际使用中，造成 HDFS down time 的主要因素是软件的升级与维护，硬件因素所占的比例极小。

> 据统计，Facebook 团队在 4 年的时间内，仅有一次由于硬件故障造成 down time。

因此 AvatarNode 可以解决绝大部分 down time，加之 NFS 软件自身非常成熟可靠，因此 NFS 单点造成 HDFS down time 的几率极小，而且我们在后面的实验中也会证明，即便是 NFS 无法服务，恢复的时间也非常短，大大小于 NameNode 直接从硬盘启动进行恢复所需时间。此外 NFS 自身的许多机制可直接应用于维护元数据之间的一致性，大大简化了开发。

再次 AvatarNode 在性能上优于 Hadoop 的 Backup Node 机制。对于 Backup Node 机制，NameNode 需要将新的日志记录同步更新到 Backup Node，这也意味着，NameNode 每产生一条日志记录，除了自身的同步，还必须等待 Backup Node 接收，先进行内存合并更新内存中元数据，然后将日志记录写入磁盘，所有步骤完成后才算结束。对于 AvatarNode，Primary AvatarNode 将日志写入 NFS 的共享目录即结束，不用同步 Standby 写入，因此 AvatarNode 的效率更高。

此外，在实现上 AvatarNode 采用了 wrapper 模式，尽可能地继承 NameNode 已有的功能和特性，在此基础上增加新的内容，既简化了开发，又提高了系统的可靠性，而且也便于 HDFS 与 AvatarNode 各自代码的维护。

当然，AvatarNode 也存在一些不足之处，其中主要的一点是相关资料太少，只能通过查看源码来获取信息，加之其机制又相对复杂，限制了它的应用普及。

5.1.2 思路分析

在后续章节中，我们将结合源码对 AvatarNode 的机制进行分析，方法是按照 AvatarNode 的一个生命周期内的运行顺序进行说明，具体顺序是：

（1）Primary AvatarNode 先启动；

（2）接下来 Standby AvatarNode 启动；

（3）再接下来 Primary AvatarNode 启动接收 Client 请求，对外正常服务；

（4）最后 Primary AvatarNode 停止服务，Standby AvatarNode 进行切换接替服务。

> 本章原理性部分较多，需要反复仔细阅读，对于初次接触 AvatarNode 的读者，建议先对本章进行泛读，了解了 AvatarNode 的基本概念后，转去阅读下一章“AvatarNode 使用”，通过动手实践加深印象，然后再返回来阅读本章，这样从理论到实践，再回到理论进行循环，效果更好。

AvatarNode 的源码以两种方式提供：

- 第 1 种是 Patch 方式，可直接作用于 Hadoop 的官方版，目前支持的 Hadoop 版本是 0.20，第 1 种方式适合目前已经部署使用 Hadoop 官方版的应用，使用时在已有源码上打上 Patch 并重新 Build；
- 第 2 种方式是直接集成进 FaceBook 维护的 Hadoop 分支，其版本也是基于官方的 0.20 版本进行修改而成的。第 2 种方式适合对当前 Hadoop 分支没有特殊严格要求的应用，该方式使用更为简单。两种方式除了第 1 种需要 Build 外，其原理和使用方法基本相同。

后面我们将采用 FaceBook Hadoop 分支源码对 AvatarNode 机制进行分析。

我们分析的 FaceBook Hadoop 分支的版本是 facebook-hadoop-20-append-

eaec342.tar.gz，下载地址是：

```
https://github.com/facebook/hadoop-20-append
```

该版本是在 Hadoop 官方版本 0.20-append 基础上，加入自身特性发展起来的。AvatarNode 的代码位于：

```
facebook-hadoop-20-append-eaec342\facebook-hadoop-20-append-eaec342\s
```

具体分析如下。

5.1.3　性能数据

1. 公开资料信息

5000 万个文件，AvatarNode 的切换时间不超过 60 秒，与此相对的采用冷备方式的切换需要 1 个小时[2]。

2. 实际测试

10 万个文件切换时间不超过 15 秒。

5.2　元数据分析

AvatarNode 的 Primary 节点和 Standby 节点都继承自类 NameNode，其元数据结构和 NameNode 是一致的。

> 我们分析采用的 FaceBook Hadoop 分支源码是：facebook-hadoop-20-append-eaec342.tar.gz，它是在 Hadoop 官方版本 0.20-append 基础上修改而来的，而第 2 章中分析所采用的版本是 Hadoop 官方版本 0.21，版本的差异会导致实现上也会有所不同，在此我们对 FaceBook Hadoop 分支源码的 HDFS 元数据进行简要分析。

[2] http://hadoopblog.blogspot.com/2010/02/hadoop-namenode-high-availability.html

5.2.1 类 FSNamesystem

AvatarNode 的内存元数据由类 FSNamesystem 来统一表示，类 FSNamesystem 包括三个最重要的成员变量：

- “public FSDirectory dir;”：管理整个 HDFS 的 INode 信息，也就是维护整个 HDFS 的层次目录信息，同时还负责元数据的加载以及持久化存储；
- “final BlocksMap blocksMap=newBlocksMap(DEFAULT_INITIAL_MAP_CAPACITY, DEFAULT_MAP_LOAD_FACTOR);”：负责 HDFS 中所有 Block 信息的管理，它维护了从 Block 到 INode 以及 DataNode 的映射；
- “NavigableMap<String,DatanodeDescriptor>datanodeMap=new TreeMap <String, DatanodeDescriptor>();”：负责 HDFS 中所有 DataNode 的管理，它维护了从 DataNode 到 Block 的映射。

5.2.2 类 FSDirectory

对于类 FSDirectory，它包括“final INodeDirectoryWithQuota rootDir;”以及“FSImage fsImage;”两个重要的成员变量。

“final INodeDirectoryWithQuota rootDir;”表示整个 HDFS 的根目录，它是一个配额类型的 INode 节点，INode 是 HDFS 基本元素的抽象，其子类包括：

- InodeFile：表示 HDFS 中的文件；
- InodeDirectory：表示目录；
- InodeDirectoryWithQuota：表示有配额限制的目录；
- InodeFileUnderConstruction：表示还未完全写入成功的文件。

FSImage

负责内存元数据与磁盘元数据文件之间的转换，它包括一个“FSEditLog editLog = null;”的成员变量。

- 类 FSImage 自身负责内存元数据与 fsimage 文件之间的转换，包括读取 fsimage 文件内容，进行解析后形成 FSDirectory 中的 INode 组织信息，以及将 FSDirectory 中的 INode 组织信息转换为 fsimage 文件中的记录进行存储；
- “FSEditLog editLog = null;”则负责将元数据操作记录同步到磁盘日志文件 edits，或读取磁盘日志文件 edits 的日志记录至内存，并与原有的 fsimage 信息进行合并，形成最新的内存元数据。

5.2.3　AvatarNode 的磁盘元数据文件

AvatarNode 的磁盘元数据文件包括：

- fsimage：其中 fsimage 代表某一时刻的内存元数据镜像，最初由 format 命令产生，在每次 HDFS 启动的时候，NameNode 会将当前的 fsimage 以及 edits 日志合并到内存，形成最新的元数据，然后导出到磁盘形成新的 fsimage，此外，通过其他机制（Secondary NameNode、Backup Node 或 AvatarNode）也可以在 Checkpoint 时，产生新的 fsimage 文件；
- edits：edits 文件最初由格式化产生，在每次 HDFS 启动时或进行 Checkpoint 时，会对 edits 文件进行重置；
- VERSION：VERSION 文件最后写入，它的写入标志着该目录下其他文件写入成功，它的存在与否表明了当前操作所处的状态；
- fstime：fstime 文件主要用于存储上次 CheckPoint 的时间。

它们位于元数据路径的 current 目录下。

5.3　AvatarNode Primary 启动过程

AvatarNode Primary 启动后会判断本节点角色，然后将 Avatar 的相关属性（在 core-site.xml 和 hdfs-site.xml 中属性名结尾为 0 或 1 的属性）转换为对应的 Hadoop

属性。

接着产生一个 AvatarNode 实例，并对父类 NameNode 进行初始化，在初始化时会启动 Web 服务器（默认地址为 0.0.0.0:50070）、RPC 远程调用服务器和相关服务线程。

在此之前 NameNode 会做一次 CheckPoint。即系统将磁盘上的镜像文件加载到内存生成命名空间即目录树，日志文件也会被加载到内存的命名空间中，日志文件的每条日志记录被解析为对元数据的操作，当日志文件被加载结束也就完成了镜像与日志的合并，在内存中生成前一次系统关闭时的目录树。

接下来，内存中的最新镜像会被序列化保存到磁盘的目录镜像存储路径下的 current/fsimage 中，同时日志目录镜像存储路径下的 current/edits 会被清空，两个路径下的 fstime 文件会存储此次 CheckPoint 的时间，之后会等待用户请求和 DataNode 的心跳信息。

以下是对相关代码的简要分析。

（1）首先是从入口方法 main(String argv[])进入，其中 createAvatarNode(argv, null)方法返回一个 AvatarNode 实例。

```
$HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/hdfs/server/namenode/AvatarNode.java
package org.apache.hadoop.hdfs.server.namenode;
public static void main(String argv[]) throws Exception {
    ……
    do {
        ……
        try {
           StringUtils.startupShutdownMessage(AvatarNode.class, argv, LOG);
            avatarnode = createAvatarNode(argv, null);
            ……
        } catch (Throwable e) {
```

```
            ……
          }
      } while (doRestart == true);
      ……
  }
```

（2）createAvatarNode(String argv[],Configuration conf)方法的调用步骤如下。

方法 parseArguments(argv)对入口参数进行了解析。

方法 copyFsImage(startupConf,startOpt)将配置文件中有关 Avatar 的配置项转换为 Hadoop 类型，并当启动参数中有“-sync”参数时，此方法将本节点与另一个 AvatarNode 保持元数据同步。

实例化一个 AvatarNode。如下所示。

```
  $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/server/namenode/AvatarNode.java
  package org.apache.hadoop.hdfs.server.namenode;
  public static AvatarNode createAvatarNode(String argv[],
  Configuration conf) throws IOException {
      ……
      StartupOption startOpt = parseArguments(argv);
      ……
      conf = copyFsImage(startupConf, startOpt);
      ……
      return new AvatarNode(startupConf, conf, startOpt.toAvatar());
  }
```

（3）copyFsImage(Configuration conf,StartupOption opt)方法的执行顺序如下。

若本节点的 Avatar 以 Standby 启动，则调用 setStartCheckpointTime(conf)方法读取另一个 AvatarNode 的 checkpoint time 并存储在内存变量中。

当启动参数中有“-sync”参数时，将 NFS 上另一个 AvatarNode 的镜像、日志等元数据文件存储目录覆盖 NFS 上和本地的镜像文件目录，同时清空日志文件。

将配置文件中 Avatar 的参数项的值在以“，”连接后，设置为 Hadoop 相关参数项的值。

$HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/hdfs/server/namenode/AvatarNode.java

```
package org.apache.hadoop.hdfs.server.namenode;
static Configuration copyFsImage(Configuration conf, StartupOption opt)
throws IOException {
    ……
    if (opt == StartupOption.STANDBY) {
        setStartCheckpointTime(conf);
    }
    ……
    if (src.exists() && syncAtStartup) {
        ……
    }
    if (srcedit.exists() && syncAtStartup) {
        ……
        createEditsFile(destedit.toString());
        ……
    }
    ……
    Configuration newconf = new Configuration(conf);
    StringBuffer buf = new StringBuffer();
    if (instance == InstanceId.NODEONE) {
        buf.append(img1);
    } else if (instance == InstanceId.NODEZERO) {
```

```
        }
        for (String str : namedirs) {
            buf.append(",");
            buf.append(str);
        }

        newconf.set("dfs.name.dir", buf.toString());
        ……
    }
```

（4）AvatarNode 的构造方法。

首先调用父类 NameNode 的构造方法。

调用初始化方法进行 AvatarNode 的初始化。

```
    $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/server/namenode/AvatarNode.java
    package org.apache.hadoop.hdfs.server.namenode;
    AvatarNode(Configuration startupConf, Configuration conf, Avatar avatar)

        super(conf);

        ……
    }
```

（5）NameNode 的构造方法只调用了一个初始化方法：initialize(Configuration Conf)，其执行顺序如下。

实例化一个 FSNamesystem 类，在此类的实例化过程中做了一次 CheckPoint。

启动一个 http 服务器，以使得客户端可以通过 Web 查看 HDFS。

```
   $HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/server/namenode/namenode
.java
   package org.apache.hadoop.hdfs.server.namenode;
   private void initialize(Configuration conf) throws IOException {
       this.conf = conf;
       ……
       this.namesystem = new FSNamesystem(this, conf);
       this.startDNServer();
       ……
       startHttpServer(conf);
   }
```

（6）AvatarNode 的初始化方法 initialize(Configuration conf)。

此方法主要建立并启动了一个 RPC 服务器，以使客户端可以远程调用 AvatarNode 中的方法。

```
   $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/server/namenode/AvatarNode.java
   package org.apache.hadoop.hdfs.server.namenode;
   private void initialize(Configuration conf) throws IOException {
       InetSocketAddress socAddr = AvatarNode.getAddress(conf);
       int handlerCount = conf.getInt("hdfs.avatarnode.handler.count", 3);
       this.server = RPC.getServer(this, socAddr.getHostName(),
socAddr.getPort(),
           handlerCount, false, conf);
       this.serverAddress = this.server.getListenerAddress();

       this.server.start();
   }
```

5.4　AvatarNode Standby 启动过程

AvatarNode Standby 相对于 AvatarNode Primary 来说，其不同之处在于 AvatarNode 的构造方法中多启动了一个 Standby 线程，此线程周期性地作 CheckPoint，并且在此线程中启动了一个 Ingest 线程，其主要作用是读取 NFS 上 AvatarNode Primary 的磁盘日志文件，周期性地读取日志文件中自上次读取结束时产生的新日志记录，并将这些记录转换成元数据操作更新 AvatarNode Standby 的内存目录树。这样就使得 AvatarNode Standby 与 AvatarNode Primary 的元数据保持了一致性。

集群在运行期间进行元数据同步，在集群启动时，需要一个节点以 Primary 方式先启动，另一个节点以 Standby 方式后启动。后启动的 AvatarNode Standby 就需要对元数据进行初始化同步，因此在 AvatarNode Standby 启动时，要加一个“-sync”参数，有了这个参数，AvatarNode Standby 就会在启动时将 AvatarNode Primary 的镜像和日志文件全部复制到本地，并将其加载到内存中，使得两个节点的元数据同步。这一点在上一节的 AvatarNode Primary 启动过程中的 AvatarNode.copy FsImage()方法的分析中有描述。

5.4.1　AvatarNode 的构造方法

（1）首先调用父类 NameNode 的构造方法。

（2）接着调用初始化方法进行 AvatarNode 的初始化。

（3）若此节点以 AvatarNode Standby 启动，则进入保护模式。

（4）若此节点以 AvatarNode Standby 启动，则启动 Standby 线程。

```
   $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/server/namenode/AvatarNode.java
   package org.apache.hadoop.hdfs.server.namenode;
   AvatarNode(Configuration startupConf, Configuration conf, Avatar avatar)
```

```
        super(conf);
        initialize(conf);
        ……
        if (avatar == Avatar.STANDBY) {
            ……
            setSafeMode(SafeModeAction.SAFEMODE_ENTER);
            ……
            standby = new Standby(this, startupConf, confg);
            standbyThread = new Thread(standby);
            standbyThread.start();
            ……
        }
    }
```

5.4.2 Standby 线程的 run()方法

首先调用 hasStaleCheckpoint()方法，通过对比远程 AvatarNode Primary 和本地的 CheckPoint 时间判断远程 AvatarNode Primary 是否在重启或者有其他守护线程在对 AvatarNode Primary 的元数据做 CheckPoint，如果是，则重启本节点的 AvatarNode Standby 来保证与 AvatarNode Primary 磁盘镜像文件的一致性。

如果是线程的第 1 次启动或当前已经过去了一个周期的时间或是当前的日志文件的大小达到了一个阈值，则调用 doCheckpoint()方法做 CheckPoint。

启动 Ingest 线程读取 AvatarNode Primary 的日志。

```
   $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/server/namenode/Standby.java

   public void run() {
       ……
       while (running) {
```

```
          try {
              ……
              if (hasStaleCheckpoint()) {
                  backgroundThread = null;
                  quiesce();
                  break;
              }
              if (lastCheckpointTime == 0 ||
                  (lastCheckpointTime + 1000 * checkpointPeriod < now) ||
                  (earlyScheduledCheckpointTime < now) ||
                  avatarNode.editSize(confg) > checkpointSize)
{earlyScheduledCheckpointTime = now + CHECKPOINT_DELAY;
                  doCheckpoint();
                  earlyScheduledCheckpointTime = Long.MAX_VALUE;
                  lastCheckpointTime = now;
                  AvatarNode.setStartCheckpointTime(startupConf);
              }
              ……
              if (ingest == null) {
                  ingest = new Ingest(this, confg,
                      avatarNode.getRemoteEditsFile(startupConf));
                  ingestThread = new Thread(ingest);
                  ingestThread.start(); // start thread to process
transaction logs
              }
          } catch (IOException e) {
              ……
          }
      }
```

5.4.3 Ingest 线程的 run()方法

Ingest 线程的 run()方法，默认每 3 秒读取一次 NFS 上 AvatarNode Primary 的日志文件，将读取到的每条日志应用到内存中的命名空间，以使与 AvatarNode Standby 上的元数据保持一致。

```
   $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/server/namenode/Ingest.java
   package org.apache.hadoop.hdfs.server.namenode;
   public void run() {
       ……
       while (running) {
           try {
               // keep ingesting transactions from remote edits log.
               loadFSEdits(ingestFile);
           } catch (Exception e) {
               ……
           } finally {
   LOG.warn("Ingest: Processing transactions has a hiccup. " + running);
           }
       }
       success = true;     // successful ingest.
   }
```

5.4.4 Ingest 线程的 ingestFSEdits ()方法

Ingest 线程的 run()方法中循环调用 loadFSEdits()方法，此方法所调用的 ingest FSEdits()方法主要用来读取 NFS 上 AvatarNode Primary 的日志，并将日志应用到内存中。在此方法的 while (running)的循环中对每条日志进行处理，根据不同类型的日志调用不同的方法更新命名空间。

```
$HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/

package org.apache.hadoop.hdfs.server.namenode;
int ingestFSEdits(File fname, DataInputStream in, int logVersion)
throws IOException
{
    ……
    // Process all existing transactions till end of file
    while (running) {
        currentPosition = fc.position(); // record the current file offset.
        ……
        switch (opcode) {
            ……
        }
    }
    return numEdits; // total transactions consumed
}
```

5.4.5　Standby 线程的 doCheckpoint()方法

首先远程调用 AvatarNodePrimary 的方法 rollEditLog()，使 AvatarNode Primary 关闭当前的日志文件 edits，建立临时日志文件 edits.new 来存储日志。此方法返回的 CheckpointSignature 实例存储了进行一次 CheckPoint 的信息。

调用 ingest.quiesce(sig)方法对 AvatarNode Primary 的日志进行一次读取后 Ingest 线程就会退出，其主要作用是对在 CheckPoint 之前没有读取的那部分日志进行读取。

在语句 fsImage.saveFSImage()中 FSImage 的 saveFSImage()方法主要是将日志加载到内存的镜像中并用此镜像更新磁盘镜像文件。在此操作期间元数据被加上了写入锁。

此后，通过调用 putFSImage(sig)方法将更新后的镜像文件上传给 AvatarNode Primary，并远程调用其 rollFsImage()方法将磁盘镜像更新为从 AvatarNode Standby 上传的镜像文件同时将日志文件 edits.new 重命名为 edits。

```
   $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/server/namenode/Standby.java
   package org.apache.hadoop.hdfs.server.namenode;
   private void doCheckpoint() throws IOException {
      ……
      CheckpointSignature sig =
(CheckpointSignature)primaryNamenode.rollEditLog();
      ……
      ingest.quiesce(sig);
      try {
         ingestThread.join();
   LOG.info("Standby: finished quitting ingest thread just before ckpt.");
      } catch (InterruptedException e) {
         LOG.info("Standby: quiesce interrupted.");
         throw new IOException(e.getMessage());
      }
      ……
      fsnamesys.writeLock();
      try {
         ……
         LOG.info("Standby: Save fsimage on local namenode.");
         fsImage.saveFSImage();
      } finally {
         fsnamesys.writeUnlock();
      }
```

```
        putFSImage(sig);
        // make transaction to primary namenode to switch edit logs
        LOG.info("Standby: Roll fsimage on primary namenode.");
        primaryNamenode.rollFsImage();
        ……
    }
```

5.5　用户操作情景分析

5.5.1　创建目录情景分析

创建目录的过程就是 mkdir 命令的执行流程：

首先从 org.apache.hadoop.fs. FsShell.java 的入口方法 main(String argv[])开始，中间经过 RPC 远程调用了 NameNode 的 mkdirs(String src, FsPermission masked)方法，到最后的 org.apache.hadoop.hdfs.server.namenode.FSDirectory.java 中的方法<T extends INode> T addChild(INode[] pathComponents, int pos, T child, long childDiskspace, boolean inheritPermission,boolean checkQuota)在目录树上添加一个节点，完成整个流程。

下面对$HADOOP_HOME/bin/hadoop dfs -mkdir /usr/root/mydir 命令的执行流程进行详细分析。

1．org.apache.hadoop.fs.FsShell.java 的入口方法 main(String argv[])

此方法首先将 FsShell 实例化，并在其 run()方法中处理输入参数，在 run()方法中，输入参数被指定到 doall(String cmd, String argv[], int startindex)方法中处理，之后参数被送入 mkdir(String src)方法。

```
$HADOOP_HOME/src/core/org/apache/hadoop/fs/FsShell.java

public static void main(String argv[]) throws Exception {
    FsShell shell = new FsShell();
```

```
    int res;
    try {
        res = ToolRunner.run(shell, argv);
    } finally {
        shell.close();
    }
    System.exit(res);
}
```

2．org.apache.hadoop.fs.FsShell.java 的 mkdir(String src)方法

在此方法中首先判断要建立的目录是否已经存在，如果不存在则调用 org.apache.hadoop.hdfs.DistributedAvatarFileSystem.java 类的 mkdirs(Path f, FsPermission permission)方法，此方法调用了 org.apache.hadoop.hdfs.DFSClient 类的 mkdirs(String src, FsPermission permission)方法。DistributedAvatarFileSystem 类是抽象类 org.apache.hadoop.fs. DistributedFileSystem 的子类，在初始化时获取了 AvatarNode Primary 的地址。

```
$HADOOP_HOME/src/core/org/apache/hadoop/fs/FsShell.java
package org.apache.hadoop.fs;
void mkdir(String src) throws IOException {
    Path f = new Path(src);
    FileSystem srcFs = f.getFileSystem(getConf());
    FileStatus fstatus = null;
    try {
        fstatus = srcFs.getFileStatus(f);
        ……
    } catch(FileNotFoundException e) {
        if (!srcFs.mkdirs(f)) {
            throw new IOException("failed to create " + src);
        }
```

```
    }
```

3．org.apache.hadoop.hdfs.DistributedAvatarFileSystem 类的 initialize(URI name, Configuration conf)方法

在 DistributedAvatarFileSystem 类的初始化方法中首先将 core-site.xml 中属性 fs.default.name 的值中的“.”、“:”替换为“/”形成 ZooKeeper 集群上 AvatarNode Primary 的地址存储路径，例如 hdfs://0.0.0.0:9000 替换为/0/0/0/0/9000，接下来会实例化一个 AvatarZooKeeperClient 实例作为 ZooKeeper 的客户端对 ZooKeeper 集群进行操作。最后会调用 initUnderlyingFileSystem(boolean failover)方法从 ZooKeeper 集群上获取 AvatarNode Primary 的 IP 地址 addrBytes。

```
   $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/DistributedAvatarFileSystem.java
   package org.apache.hadoop.hdfs;
   public void initialize(URI name, Configuration conf) throws IOException {
      ……
      zNode = "/" + fsAddress.replaceAll("[.:]", "/");
      zk = new AvatarZooKeeperClient(connection, zkTimeout, watchZK,
watcher,conf,zNode);
      initUnderlyingFileSystem(false);
   }

   private boolean initUnderlyingFileSystem(boolean failover) throws
IOException {
      try {
         byte[] addrBytes = zk.getNodeData(zNode, activeStat);

         }
   }
```

4．org.apache.hadoop.hdfs.DFSClient 类的 mkdirs(String src, FsPermission permission)方法

此方法中的 namenode 实例就是 NameNode 实例的 RPC 远程代理，通过语句 namenode.mkdirs(src, masked)调用远程实例的方法来执行 mkdir 操作。

```
$HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/DFSClient.java
package org.apache.hadoop.hdfs;
public boolean mkdirs(String src, FsPermission permission)throws
IOException{
      ……
      try {
          return namenode.mkdirs(src, masked);
      } catch(RemoteException re) {
          ……
      }
      }
```

5．org.apache.hadoop.hdfs.server.namenod.NameNode 类的 mkdirs(String src, FsPermission masked)方法

之前的流程均是在客户端执行，从此方法开始流程进入 NameNode，在此方法中首先检查要建立的目录的绝对路径的长度，之后调用 org.apache.hadoop.hdfs.server.namenode.FSNamesystem 类的 mkdirs(String src, PermissionStatus permissions)方法。

```
$HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/server/namenode/namenode
.java
package org.apache.hadoop.hdfs.server.namenode;
public boolean mkdirs(String src, FsPermission masked) throws IOException {

      if (!checkPathLength(src)) {
          throw new IOException("mkdirs: Pathname too long.  Limit "
```

```
            + MAX_PATH_LENGTH + " characters, "
            + MAX_PATH_DEPTH + " levels.");
    }
    return namesystem.mkdirs(src, new
PermissionStatus(UserGroupInformation.getCurrentUGI().getUserName(),
        null, masked));
}
```

6. org.apache.hadoop.hdfs.server.namenode.FSNamesystem 类的 mkdirs(String src, PermissionStatus permissions)方法

此方法调用了内部私有方法 mkdirsInternal(String src,PermissionStatus permissions)，在此私有方法所调用的其他方法中一条建立目录的日志被记录到了内存里，所以语句 getEditLog().logSync()的作用就是将内存中的日志写入到磁盘日志文件中。

```
$HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/server/namenode/FSNamesy

package org.apache.hadoop.hdfs.server.namenode;
public boolean mkdirs(String src, PermissionStatus permissions
) throws IOException {
    boolean status = mkdirsInternal(src, permissions);
    getEditLog().logSync();
    ……
    return status;
}
```

7．org.apache.hadoop.hdfs.server.namenode.FSNamesystem 类的 mkdirsInternal(String src，PermissionStatus permissions)方法

在此方法中首先加上了写入锁，表明此操作为互斥操作，同一时刻只能有一个 mkdir 请求被执行。

之后就是一系列的检查，包括：目录是否已建立、NameNode 是否处于保护模

式、路径名是否有效、客户端是否有写权限、inode 节点数是否已经达到上限等。

再下来就是调用 org.apache.hadoop.hdfs.server.namenode.FSDirectory.java 类的 mkdirs(String src, PermissionStatus permissions,boolean inheritPermission, long now)方法，最后解除写入锁。

```
$HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/server/namenode/FSNamesy

package org.apache.hadoop.hdfs.server.namenode;
private boolean mkdirsInternal(String src,PermissionStatus permissions)
throws IOException {
    try {
        writeLock();
        ……
        if (dir.isDir(src)) {
        return true;
        }
        if (isInSafeMode())
throw new SafeModeException("Cannot create directory " + src, safeMode);
        if (!DFSUtil.isValidName(src)) {
            throw new IOException("Invalid directory name: " + src);
        }
        if (isPermissionEnabled) {
            checkAncestorAccess(src, FsAction.WRITE);
        }
        checkFsObjectLimit();
        if (!dir.mkdirs(src, permissions, false, now())) {
            throw new IOException("Invalid directory name: " + src);
        }
        return true;
    } finally {
```

```
            writeUnlock();
        }
    }
```

8．org.apache.hadoop.hdfs.server.namenode.FSDirectory.java 类的 mkdirs (String src, PermissionStatus permissions,boolean inheritPermission, long now)方法

在此方法中，首先将要建立的目录的绝对路径上的所有目录名存放到一个 names 的 String 数组中，再将此数组解析成 byte 二维数组。

在 rootDir.getExistingPathINodes(components, inodes)方法调用中将元数据中已经有的目录名对应的 INode 实例返回，在之后第 1 个 for 循环中将元数据中已存在 INode 实例的目录名组成路径 pathbuilder，第 2 个 for 循环中对元数据中不存在 INode 实例的目录名调用 unprotectedMkdir(INode[] inodes, int pos,byte[] name, PermissionStatus permission, boolean inheritPermission,long timestamp)方法以此建立目录的 INode 实例，就是说即使要建立的目录的父目录或是其更上层目录没有建立的话，也可以同时被建立。

在循环中，路径 pathbuilder 被添加了新建立的目录，并且当新建一个目录的 INode 实例时，fsImage.getEditLog().logMkDir(cur, inodes[i])就被调用，即建立此目录的操作被写入到了日志中。此操作在磁盘日志文件中生成了一条日志记录，同时 AvatarNode Standby 节点上的 Ingest 线程会读取到这条记录并将其加载到内存的命名空间中，这样 AvatarNode Standby 与 AvatarNode Primary 的元数据就同步了。此外，此方法的整个过程是互斥的，加了写入锁。

```
    $HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/server/namenode/FSDirect
ory.java
    package org.apache.hadoop.hdfs.server.namenode;
    boolean mkdirs(String src, PermissionStatus permissions,
    boolean inheritPermission, long now)
    throws FileNotFoundException, QuotaExceededException {
```

```
    src = normalizePath(src);
    String[] names = INode.getPathNames(src);
    byte[][] components = INode.getPathComponents(names);
    INode[] inodes = new INode[components.length];
    writeLock();
    try {
       rootDir.getExistingPathINodes(components, inodes);
       // find the index of the first null in inodes[]
       StringBuilder pathbuilder = new StringBuilder();
       int i = 1;
       for(; i < inodes.length && inodes[i] != null; i++) {
          pathbuilder.append(Path.SEPARATOR + names[i]);
          if (!inodes[i].isDirectory()) {
throw new FileNotFoundException("Parent path is not a directory: "
             + pathbuilder);
          }
       }
       for(; i < inodes.length; i++) {
          pathbuilder.append(Path.SEPARATOR + names[i]);
          String cur = pathbuilder.toString();
          unprotectedMkdir(inodes, i, components[i], permissions,
             inheritPermission || i != components.length-1, now);
          if (inodes[i] == null) {
             return false;
          }
          if (namesystem != null)
             NameNode.getNameNodeMetrics().numFilesCreated.inc();

          NameNode.stateChangeLog.debug(
             "DIR* FSDirectory.mkdirs: created directory " + cur);
```

```
            }
        } finally {
            writeUnlock();
        }
        return true;
    }
```

9. org.apache.hadoop.hdfs.server.namenode.FSDirectory.java 类的 unprotectedMkdir(INode[] inodes, int pos,byte[] name, PermissionStatus permission, boolean inheritPermission,long timestamp)

最终调用 addChild(INode[] pathComponents, int pos,T child, long childDiskspace, boolean inheritPermission,boolean checkQuota)方法在元数据的目录树上添加了一个节点。

到此整个 mkdir 操作流程结束。

5.5.2　创建文件情景分析

在这里，我们以从本地复制文件到 HDFS 中为例，说明一下文件创建的过程。同创建目录一样，入口方法也是 org.apache.hadoop.fs.FsShell.java 的 main(String argv[])方法，除了一些系统管理命令，所有对 HDFS 系统中文件或目录操作的入口均是此方法。

在本节中，设定命令为$HADOOP_HOME/bin/hadoop dfs –copyFromLocal /tmp/test.iso /usr/root/，其中/tmp/test.iso 是本地磁盘文件。

在此期间，AvatarNode Primary 将日志写入日志文件。每一个用户操作都会被分解为多条日志，并被写入到日志文件中，例如 copyFromLocal（向 HDFS 复制文件）的命令，就被分解为 openFile、addBlock 等多条日志。在 AvatarNode Primary 节点上，这些日志不但会写入本地日志文件，而且通过在配置文件里配置日志文件目录，也会写入到 NFS 上的日志文件。同时，AvatarNode Standby 读取 AvatarNode Primary 在 NFS 上的日志，更新内存的命名空间。

> 提示: 具体日志读取与 AvatarNode Standby 的内存命名空间的更新可以参考 5.4 节 AvatarNode Standby 启动过程中的 Ingest 线程的运行机制。

（1）此命令进入 org.apache.hadoop.fs.FsShell.java 的 main(String argv[])方法后，经过几个简单的方法调用到了此类的 copyFromLocal(Path[] srcs, String dstf)方法。

在此方法中，语句 FileSystem dstFs = dstPath.getFileSystem(getConf())返回的是一个继承 org.apache.hadoop.fs DistributedFileSystem.java 类的 org.apache.hadoop.hdfs. DistributedAvatarFileSystem.java 类，接下来若是标准输入则调用 copyFromStdin(Path dst, FileSystem dstFs)方法，如果是文件的 copy 操作则调用 org.apache.hadoop.fs.FileSystem.java 类的 copyFromLocalFile(boolean delSrc, boolean overwrite,Path[] srcs, Path dst)方法。

$HADOOP_HOME/src/core/org/apache/hadoop/fs/FsShell.java

```
void copyFromLocal(Path[] srcs, String dstf) throws IOException {
    Path dstPath = new Path(dstf);
    FileSystem dstFs = dstPath.getFileSystem(getConf());
    if (srcs.length == 1 && srcs[0].toString().equals("-"))
        copyFromStdin(dstPath, dstFs);
    else
        dstFs.copyFromLocalFile(false, false, srcs, dstPath);
}
```

（2）org.apache.hadoop.fs.FileSystem.java 类的 copyFromLocalFile(boolean delSrc, boolean overwrite,Path[] srcs, Path dst)方法。

此方法调用了 org.apache.hadoop.fs.FileUtil.java 类的 copy(FileSystem srcFS, Path[] srcs, FileSystem dstFS, Path dst, boolean deleteSource, boolean overwrite, Configuration conf)方法，该方法只是对参数中的多个文件中的每个文件循环调用 copy(FileSystem srcFS, Path src,FileSystem dstFS, Path dst, boolean deleteSource,

boolean overwrite,Configuration conf)方法，用于处理单个文件的拷入。

```
   $HADOOP_HOME/src/core/org/apache/hadoop/fs/FileSystem.java
   package org.apache.hadoop.fs;
   public void copyFromLocalFile(boolean delSrc, boolean overwrite, Path[]
srcs, Path dst)
   throws IOException {
       Configuration conf = getConf();
       FileUtil.copy(getLocal(conf), srcs, this, dst, delSrc, overwrite, conf);
   }
```

（3）org.apache.hadoop.fs.FileUtil.java 类的 copy(FileSystem srcFS, Path src, FileSystem dstFS, Path dst, boolean deleteSource, boolean overwrite, Configuration conf)方法。

在此方法中，若源文件为目录则进行递归调用，否则建立对应源文件的输入流和对应目的文件的输出流。

这里的重点在于输出流的建立，即 org.apache.hadoop.fs. DistributedAvatarFileSystem.java 类的父类的父类 org.apache.hadoop.fs.FileSystem 的 create(Path f, boolean overwrite)方法，该方法最终调用的是 org.apache.hadoop.fs. DistributedFileSystem 类的 create(Path f, FsPermission permission, boolean overwrite, int bufferSize, short replication, long blockSize, Progressable progress)方法。

```
   $HADOOP_HOME/src/core/org/apache/hadoop/fs/FileUtil.java
   package org.apache.hadoop.fs;
   public static boolean copy(FileSystem srcFS, Path src, FileSystem dstFS,
Path dst,
   boolean deleteSource,boolean overwrite,Configuration conf) throws
IOException {
       dst = checkDest(src.getName(), dstFS, dst, overwrite);
```

```
        ……
        FileStatus contents[] = srcFS.listStatus(src);
        for (int i = 0; i < contents.length; i++) {
            copy(srcFS, contents[i].getPath(), dstFS,
                new Path(dst, contents[i].getPath().getName()),
                deleteSource, overwrite, conf);
        }
    } else {
        InputStream in=null;
        OutputStream out = null;
        try {
            in = srcFS.open(src);
            out = dstFS.create(dst, overwrite);
            IOUtils.copyBytes(in, out, conf, true);
        } catch (IOException e) {
            ……
        }

    ……
}
```

（4）org.apache.hadoop.fs.DistributedFileSystem.java 类的 create(Path f, FsPermission permission, boolean overwrite, int bufferSize, short replication, long blockSize, Progressable progress)方法。

此方法返回一个 FSDataOutputStream 输出流，此流的构造方法中有个 DFSOutputStream 流的参数，也就是说在这里 FSDataOutputStream 实例是对 DFSOutputStream 实例的一个包装。而 DFSOutputStream 流的实例是由 org.apache.hadoop.hdfs.DFSClient 类的 create(String src, FsPermission permission,

boolean overwrite, short replication, long blockSize, Progressable progress, int buffersize)方法实例化的。

```
$HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/DistributedFileSystem.java
package org.apache.hadoop.hdfs;
public FSDataOutputStream create(Path f, FsPermission permission,boolean
overwrite,
int bufferSize, short replication, long blockSize,Progressable progress)
throws IOException {
return new FSDataOutputStream (dfs.create(getPathName(f), permission,
overwrite, replication, blockSize, progress, bufferSize),statistics);
}
```

（5）org.apache.hadoop.hdfs.DFSClient 类的内置类 DFSOutputStream 的构造方法 DFSOutputStream(String src, FsPermission masked, boolean overwrite, short replication, long blockSize, Progressable progress,int buffersize, int bytesPerChecksum)。

在此方法中进行了对 NameNode 的远程 RPC 调用，调用了 create(String src, FsPermission masked, String clientName, boolean overwrite, short replication, long blockSize)方法用来在 NameNode 元数据的目录树上添加一个文件节点，此节点是 INodeFileUnderConstruction 类的实例。最后在此方法中启动了一个 DataStreamer 线程用于数据发送。

```
$HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/DFSClient.java
package org.apache.hadoop.hdfs;
DFSOutputStream(String src, FsPermission masked, boolean overwrite,
short replication, long blockSize, Progressable progress,int buffersize,
int bytesPerChecksum) throws IOException {
    ……
    try {
```

```
blockSize);
       } catch(RemoteException re) {
           ……
       }
       streamer.start();
   }
```

（6）org.apache.hadoop.hdfs.DFSClient 类的内置线程类 DataStreamer 的 run() 方法。

此方法主要用于将数据发送到 DataNode，被发送的数据被组织成了 Packet 类的实例，这个类包括了一个 Packet 实例中的数据在一个 Block 块中的起始位置、是否为最后一个 Packet 等信息。

代码中线程作为“消费者”一直在 dataQueue 队列中等待被添加数据，当数据到来时，此类的 nextBlockOutputStream(String client)方法被调用，建立了一个指向特定的 DataNode 的输出流，之后 ResponseProcessor 线程被启动，用于接收 DataNode 的一个 Packet 传输成功的返回信息。

接下来就是用 java.io 的 DataOutputStream 输出流将数据发送到 DataNode。而数据的接收主要在 DataNode 端的 org.apache.hadoop.hdfs.server.datanode.DataXceiverServer.java 线程类中进行。

```
$HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/DFSClient.java

public void run() {
    ……
    Packet one = null;
    synchronized (dataQueue) {
        ……
       while ((!closed && !hasError && clientRunning && dataQueue.size() == 0)
```

```
            try {
                dataQueue.wait(1000);
            } catch (InterruptedException  e) {
            }
            doSleep = false;
        }
        ……
        try {
            one = dataQueue.getFirst();
            long offsetInBlock = one.offsetInBlock;
            if (blockStream == null) {
                LOG.debug("Allocating new block");
                nodes = nextBlockOutputStream(src);
this.setName("DataStreamer for file " + src +" block " + block);
                response = new ResponseProcessor(nodes);
                response.start();
            }
            ……
            ByteBuffer buf = one.getBuffer();
            ……
blockStream.write(buf.array(), buf.position(), buf.remaining());
            if (one.lastPacketInBlock) {
                blockStream.writeInt(0); // indicate end-of-block
            }
            blockStream.flush();
            ……
        } catch (Throwable e) {
            ……
        }
```

```
        ……
    }
```

（7）org.apache.hadoop.io.IOUtils.java 类的 copyBytes(InputStream in, OutputStream out, Configuration conf, boolean close)方法。

此方法又调用了 copyBytes(InputStream in, OutputStream out, int buffSize, boolean close)方法，在该方法中读取客户端输入流的数据，用 org.apache.hadoop.hdfs.DFSClient.java 的内置输出流类 DFSOutputStream 的 write(byte b[], int off, int len)方法将数据发送到 Data Node。

```
    $HADOOP_HOME/src/core/org/apache/io/IOUtils.java
    package org.apache.hadoop.io;
    public static void copyBytes(InputStream in, OutputStream out, int
buffSize, boolean close)
    throws IOException {
        PrintStream ps = out instanceof PrintStream ? (PrintStream)out : null;
        byte buf[] = new byte[buffSize];
        try {
            int bytesRead = in.read(buf);
            while (bytesRead >= 0) {
                out.write(buf, 0, bytesRead);
                if ((ps != null) && ps.checkError()) {
                    throw new IOException("Unable to write to output stream.");
                }
                bytesRead = in.read(buf);
            }
        } finally {
            if(close) {
                out.close();
```

```
        }
    }
}
```

（8）org.apache.hadoop.hdfs.DFSClient.java 的内置输出流类 DFSOutputStream 的 close()方法。

此方法调用了 closeInternal()方法，closeInternal()方法又调用了此类的 closeFile(String src)方法，在该方法中 RPC 远程调用了 NameNode 的方法 complete(String src, String clientName)。之后的调用路径是：

NameNode.complete(String src, String clientName)→FSNamesystem.completeFile(String src, String holder) →FSNamesystem.completeFileInternal (String src,String holder) →FSNamesystem.finalizeINodeFileUnderConstruction (String src,INodeFileUnderConstruction pendingFile)

在此类中将 INodeFileUnderConstruction 实例 pendingFile 转换成了 INodeFile，即此节点的状态由“创建中”转换到了“已创建”。至此一个文件的数据传送以及相对应的元数据目录树上的节点的建立就完成了。

```
$HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/DFSClient.java
package org.apache.hadoop.hdfs;
private void finalizeINodeFileUnderConstruction(String src,
INodeFileUnderConstruction pendingFile) throws IOException {
    ……
    INodeFile newFile = pendingFile.convertToInodeFile();
    dir.replaceNode(src, pendingFile, newFile);
    // close file and persist block allocations for this file
    dir.closeFile(src, newFile);

}
```

5.6 AvatarNode Standby 故障切换过程

AvatarNode Standby 的故障切换过程就是命令$HADOOP_HOME/bin/hadoop org.apache.hadoop.hdfs.AvatarShell -setAvatar primary 的执行流程。入口方法就是 org.apache.hadoop.hdfs.AvatarShell 类的 main(String argv[])方法，之后新建 AvatarShell 类的实例，再调用该类的 run()方法，在 run()方法中调用了 setAvatar(String cmd, String argv[], int startindex)方法。具体流程如下。

（1）org.apache.hadoop.hdfs.AvatarShell 类的 setAvatar(String cmd, String argv[], int startindex)方法。

在此方法中，用 AvatarNode 的 getAvatar()方法返回 AvatarNode 实例的角色，当实例的角色与要切换的角色不同时才进行切换，现在只支持由 Standby 切换到 Primary，即 RPC 远程调用 AvatarNode 的 setAvatar(Avatar avatar)方法。在此之后将调用 updateZooKeeper()方法将 ZooKeeper 集群上存储的 AvatarNode Primary 地址更新为本机的 IP。

```
   $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/AvatarShell.java
   package org.apache.hadoop.hdfs;
   public int setAvatar(String cmd, String argv[], int startindex)throws
IOException {
       ……
       Avatar current = avatarnode.getAvatar();
       if (current == dest) {
   System.out.println("This instance is already in " + current + " avatar.");
       } else {
          avatarnode.setAvatar(dest);

       }
       return 0;
```

```
    }
```

（2）org.apache.hadoop.hdfs.server.namenode.AvatarNode.java 类的 setAvatar(Avatar avatar)方法。

在此方法的代码中我们可以看到与 Standby 角色相关的两个线程实例 standby 和 cleaner 被停止了，角色由 Standby 转换为了 Primary，整个切换过程就结束了。

```
    $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/server/namenode/AvatarNode.java
    package org.apache.hadoop.hdfs.server.namenode;
    public synchronized void setAvatar(Avatar avatar) throws IOException {
        ……
        if (avatar == Avatar.STANDBY) {
            // ACTIVE to STANDBY
           String msg = "Changing state from active to standby is not allowed." +
            "If you really want to pause your primary, put it in safemode.";
                LOG.warn(msg);
                throw new IOException(msg);
        } else {
            // STANDBY to ACTIVE
            ……
            standby.quiesce();
            cleaner.stop();
            cleanerThread.interrupt();
            try {
                cleanerThread.join();
            } catch (InterruptedException iex) {

            }
```

```
            clearInvalidates();
            super.namesystem.setSafeModeManualOverride(false);
            setSafeMode(SafeModeAction.SAFEMODE_LEAVE);
        }
        LOG.info("Changed avatar from " + currentAvatar + " to " + avatar);
        currentAvatar = avatar;
    }
```

5.7 元数据一致性保证机制

5.7.1 元数据目录树信息

AvatarNode Standby 与 AvatarNode Primary 的元数据中的目录树保持一致的主要机制是：AvatarNode Standby 节点的 Standby 线程中启动的 Ingest 线程在周期性读取 AvatarNode Primary 在 NFS 服务器上的日志，每读取一条日志便将其加载到内存元数据中，这样就保持了两个 NameNode 内存中的元数据目录树的一致。

org.apache.hadoop.hdfs.server.namenode.Ingest.java 类的 run()方法中调用了 loadFSEdits(File edits)方法，该方法建立了一个 AvatarNode Primary 在 NFS 服务器上的日志文件输入流，从此流中读取数据并解析之后调用 org.apache.hadoop.hdfs.server.namenod. FSDirectory.java 的相关方法更新内存元数据的目录树。

```
    $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/server/namenode/Ingest.java
    package org.apache.hadoop.hdfs.server.namenode;
    public void run() {
        while (running) {
            try {
                loadFSEdits(ingestFile);
            } catch (Exception e) {
```

```
                running + ") " + StringUtils.stringifyException(e));
                throw new RuntimeException("Ingest: failure", e);
            } finally {
        LOG.warn("Ingest: Processing transactions has a hiccup. " + running);
            }
        }
        success = true;      // successful ingest.
    }
```

5.7.2　Data Node 与 Block 数据块映射信息

在内存的元数据中，除了目录树还有 DataNode 与 Block 数据块的映射信息。目录树保存在磁盘的镜像文件中，在 AvatarNode 节点启动时，会读取此文件，将整个镜像加载到内存中形成目录树。而 AvatarDataNode 与 Block 数据块的映射信息是在 Avatar Data Node 节点启动后，周期性地将本地存储的 Block 数据块的信息上传给两个 AvatarNode，在 AvatarNode 接收到此信息后，更新内存中的映射关系。由于 AvatarDataNode 是将此信息上传给两个 AvatarNode，这就保持了两个 AvatarNode 内存中的 vatarDataNode 与 Block 数据块映射信息的一致性。

（1）org.apache.hadoop.hdfs.server.datanode.AvatarDataNode.java 类的 run() 方法会在 AvatarDataNode 启动时被调用，在此方法中启动了两个 OfferService 线程，分别服务于两个 AvatarNode。

```
    $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/server/datanode/AvatarDataNode.java
    package org.apache.hadoop.hdfs.server.datanode;
    public void run() {
        ……
        while (shouldRun) {
            try {
                ……
```

```
            if (namenode1 != null && !doneRegister1 &&
            register(namenode1, nameAddr1)){
                doneRegister1 = true;
    offerService1 = new OfferService(this, namenode1, nameAddr1,
                avatarnode1, avatarAddr1);
    of1 = new Thread(offerService1, "OfferService1 " + nameAddr1);
                of1.start();
            }
            if (namenode2 != null && !doneRegister2 &&
            register(namenode2, nameAddr2)) {
                    doneRegister2 = true;
                    offerService2 = new OfferService(this, namenode2,
                        nameAddr2, avatarnode2, avatarAddr2);
    of2 = new Thread(offerService2, "OfferService2 " + nameAddr2);
                    of2.start();
            }
            ……
        } catch (Exception ex) {
    LOG.error("Exception: " + StringUtils.stringifyException(ex));
        }
        ……
    }

}
```

（2）org.apache.hadoop.hdfs.server.datanode.OfferService.java 线程类的 run()方法调用了 offerService()方法，此方法中经过 RPC 远程调用了 NameNode 的 sendHeartbeat(DatanodeRegistration nodeReg, long capacity, long dfsUsed, long remaining, int xmitsInProgress,int xceiverCount)和 blockReport(Datanode Registration nodeReg,long[] blocks)方法，分别

用来周期性发送心跳信息和 Block 的存储信息。

$HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/hdfs/server/datanode/OfferService.java

```
package org.apache.hadoop.hdfs.server.datanode;
public void offerService() throws Exception {
    ……
    while (shouldRun) {
        try {
            ……
            long startTime = anode.now();
            if (startTime - lastHeartbeat > anode.heartBeatInterval) {
                lastHeartbeat = startTime;
        DatanodeCommand[] cmds = namenode.sendHeartbeat(dnRegistration,
                  data.getCapacity(),data.getDfsUsed(),data.getRemaining(),
                   anode.xmitsInProgress.get(),anode.getXceiverCount());
                myMetrics.heartbeats.inc(anode.now() - startTime);
                //LOG.info("Just sent heartbeat, with name " + localName);
                if (!processCommand(cmds))
                    continue;
            }
            ……
           if (startTime - lastBlockReport > anode.blockReportInterval) {
                long brStartTime = anode.now();
                Block[] bReport = data.getBlockReport();
                DatanodeCommand cmd = namenode.blockReport(dnRegistration,
                    BlockListAsLongs.convertToArrayLongs(bReport));
                long brTime = anode.now() - brStartTime;
                myMetrics.blockReports.inc(brTime);
```

```
            }
        } catch(RemoteException re) {……}
        } catch (IOException e) {…… }
    }
}
```

5.8 Block 更新同步问题

5.8.1 问题描述

（1）客户端写入一个文件 a。

（2）写入成功后，Data Node 分别向 AvatarNode Primary 和 AvatarNode Standby 报告 Block 位置信息。

（3）此时，由于同步的周期时间未到，AvatarNode Standby 里并没有文件 a 的信息，因此，AvatarNode Standby 是否会将上报的 Block 信息视为无效的块，然后通知 Data Node 删除文件数据。

5.8.2 结论

AvatarNode Standby 不会删除 Block，它会循环检查对应关系，直到 AvatarNode Standby 节点 Ingest AvatarNode Primary 的日志收到其中的关于文件 a 与 Block 的对应关系信息。

5.8.3 源码分析

Data Node 启动时会启动一个 OfferService 线程用来发送 Heartbeat 和 Blockreport，另外会启动 Block 接收发送线程 DataXceiverServer() ，在此线程 run() 方法中创建一个 Socket 进行监听，等待来自客户端的 Block 写入请求，当客户端写入文件时，就会向 Data Node 发送 Block 写入请求，Data Node 接受请求后，会开启一个新的线程 DataXceiver()用来将 Block 写入 Data Node 磁盘。具体步骤如下。

（1）DataXceiverServer.java 线程类的 run()方法通常在等待数据接收请求，当客户端发送数据到 DataNode，则此线程的 run()方法的 while 循环中就会开启一个 DataXceiver 线程来接收数据块。

```
   $HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/server/datanode/DataXcei
verServer.java
   package org.apache.hadoop.hdfs.server.datanode;
   public void run() {
       while (datanode.shouldRun) {
          try {
              ……
              Socket s = ss.accept();//等待请求
              s.setTcpNoDelay(true);
              //若有请求，开启 DataXceiver 线程
              new Daemon(datanode.threadGroup,
              new DataXceiver(s, datanode, this)).start();
              ……
          } catch()
          {
              ……
          }
    }
```

（2）DataXceiver.java 线程类的 run()方法接收到数据块，调用 writeBlock()将数据块写入磁盘。此后 OfferService.java 类的 notifyNamenode ReceivedBlock()方法会被调用。

```
   $HADOOP_HOME/src/hdfs/org/apache/hadoop/hdfs/server/datanode/DataXcei

   package org.apache.hadoop.hdfs.server.datanode;
   public void run() {
```

```
      ……
      try {
         ……
         switch ( op ) {
            ……
            case DataTransferProtocol.OP_WRITE_BLOCK:
               writeBlock( in );
              datanode.myMetrics.writeBlockOp.inc(DataNode.now() - startTime);
               if (local)
                  datanode.myMetrics.writesFromLocalClient.inc();
              else
                 datanode.myMetrics.writesFromRemoteClient.inc();
               break;
               ……
           default:
         throw new IOException("Unknown opcode " + op + " in data stream");
         }
      } catch (Throwable t) {
         ……
      } finally {
         ……
      }
   }
   private void writeBlock(DataInputStream in) throws IOException {
    ……
       long totalReceiveSize = blockReceiver.receiveBlock(mirrorOut,
mirrorIn, replyOut,
                              mirrorAddr, null, targets.length);

   }
```

（3）在 org.apache.hadoop.hdfs.server.datanode.OfferService 线程类的 notify NamenodeReceived 方法中实例化一个 BlockInfo 存储类，并将其添加到 OfferService 线程类的 receivedBlockList 中，以便其他方法从中取出该实例。

$HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/hdfs/server/datanode/OfferService.java

```
package org.apache.hadoop.hdfs.server.datanode;
void notifyNamenodeReceivedBlock(Block block, String delHint) {
    if (block==null || delHint==null) {
        throw new IllegalArgumentException(block==null?"Block is null":"delHint is null");
    }
    synchronized (receivedBlockList) {
        receivedBlockList.add(new BlockInfo(block, delHint));
        receivedBlockList.notifyAll();
    }
}
```

（4）此时的 OfferService 线程中的变量 receivedBlockList 不为空，在 OfferService 线程的 run()方法中第 203 行：远程调用 avatarnode.block ReceivedNew()，当有 Block 无法对应 AvatarNode 上的文件时，此方法返回一个 failed 数组，存储没有对应关系的 Block，OfferService 线程将循环调用 avatarnode.blockReceivedNew()处理 failed 数组，直到 AvatarNode 上有对应关系建立时，有对应关系的 Block 被 failed 数组删除，直到 failed 数组为空。OfferService 线程主方法如下所示。

$HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/

```
package org.apache.hadoop.hdfs.server.datanode;
public void offerService() throws Exception {
```

```
        ......
        Block[] failed = avatarnode.blockReceivedNew(dnRegistration, blist,
delHintArray);
        ......
    }
```

（5）AvatarNode 类的 blockReceivedNew()方法接收到的数据块信息实例数组中的每个 block 在内存目录树中与文件的对应关系，并将没有对应关系的 block 集合返回给 DataNode。

```
    $HADOOP_HOME/src/contrib/highavailability/src/java/org/apache/hadoop/
hdfs/server/namenode/AvatarNode.java
    package org.apache.hadoop.hdfs.server.namenode;
    public Block[] blockReceivedNew(DatanodeRegistration nodeReg,Block
blocks[], String delHints[]) throws IOException {
        super.blockReceived(nodeReg, blocks, delHints);
        List<Block> failed = new ArrayList<Block>();
        for (int i = 0; i < blocks.length; i++) {
            Block block = blocks[i];
            synchronized(namesystem) {
                BlockInfo storedBlock =
namesystem.blocksMap.getStoredBlock(block);
                //从对应关系中查找block
                if (storedBlock == null || storedBlock.getINode() == null)  {
                    // If this block does not belong to anyfile, then record it.
                    LOG.info("blockReceived request received for "
                    + block + " on " + nodeReg.getName()+ " size " +
block.getNumBytes()+ " But it does not belong to any file."+ " Retry later.");


            }
```

```
    }
    return failed.toArray(new Block[failed.size()]);
```

读书笔记

第 6 章　AvatarNode 使用

第 5 章我们了解了 AvatarNode 的原理和运行机制，本章通过实例讲解 AvatarNode 的具体应用。

6.1 方案说明

6.1.1 网络拓扑

AvatarNode 是 HDFS 上的一个补丁包，因此其部署和使用都在 HDFS 之上进行，我们首先来看一下 AvatarNode 的网络拓扑结构，实验环境的网络拓扑如图 6.1 所示。

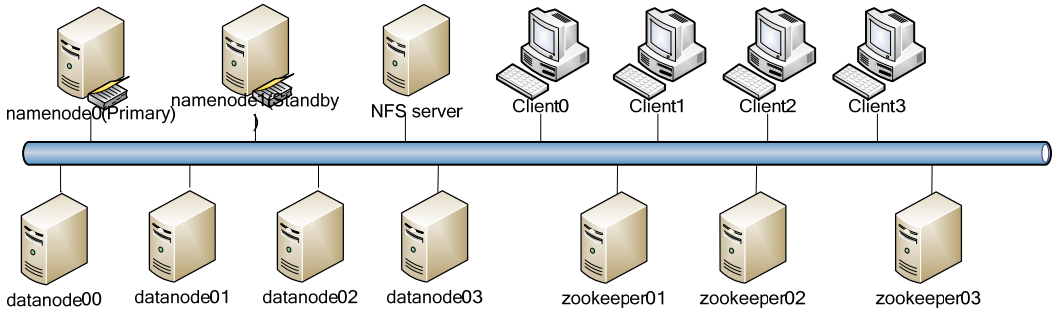

图 6.1 网络拓扑图

整个系统逻辑上有 Primary NameNode、Standby NameNode、NFS Server、Data Node、Client、Zookeeper 共 6 种类型的节点，限于单台物理机的性能，在 datanode00 至 datanode03 上，每一个节点同时部署 Client 和 Data Node，同时还在 datanode01 至 datanode03 上，每一个节点部署一个 Zookeeper，总共需要 7 个虚拟机节点，具体情况如表 6.1 所示。

表 6.1 节点配置表

节点名	角 色	IP 地址	软件配置
namenode0	Primary NameNode	192.168.1.11	Centos5.6 32 位 Ucarp-1.5.2 hadoop-0.20.3-dev facebook-hadoop-20-append JDK:build 1.6.0_24-b07 NFS 客户端
namenode1	Standby NameNode	192.168.1.12	同上

续表

节点名	角 色	IP 地址	软件配置
datanode00	DataNode Client	192.168.1.13	Centos5.6 32 位 hadoop-0.20.3-dev　JDK:build 1.6.0_24-b07
datanode 01 datanode 02 datanode 03	DataNode Client Zookeeper	192.168.1.14~192.168.1.16	同上
NFS	NFS Server	192.168.1.10	Centos5.6 32 位 nfs-utils-1.0.9-50.el5

6.1.2　操作系统安装及配置

虚拟机以及操作系统的安装在第 4 章已经准备好了，在此无需重新安装，在配置具体节点时，复制一份镜像，然后修改相应配置或者直接使用之前的节点都可以。

6.2　使用 Avatar 打补丁版本

当前的 AvatarNode 有两种使用方式，第 1 种是在原 Hadoop-0.20.2 版的 HDFS 上打 Avatar 补丁，另一个是 Hadoop 的 FaceBook 版本，其中已包含了 Avatar 包。我们首先来看一下 Hadoop-0.20.2 打补丁版本的使用，后续我们将介绍 FaceBook Hadoop 版本的使用。

官方提供的 Hadoop 版本上打上特定的 Avatar 补丁后，需要重新 Build 之后方可使用。本节介绍两种 Hadoop 源码 Build 方法：

- 第 1 种是联机 Build，每次 Build 都需要借助互联网下载所需的软件包，该方法改动小，适合于有互联网连接的环境；
- 第 2 种是本地 Build，该方法以第 1 种方法构建的环境为基础，之后每次 Build 时，不再需要从网络下载软件包，适合于无互联网连接的环境。

1. 操作环境

（1）主机要求

- 主机能够连接互联网

（2）虚拟机环境

- 1G 内存
- 16G 硬盘
- 网络连接方式为桥接（Bridge）

（3）软件环境

- CentOS 5.6 32 位
- Hadoop 的版本为 0.20.2
- JDK 版本为 1.6.0_16
- Ant 的版本为 1.8.1

6.2.1 Hadoop 源码联机 Build

参见视频：\视频\6 AvatarNode 使用\1 Avatar 有网络打补丁并编译.exe

大致步骤描述如下。

1. 配置该虚拟机上网

（1）复制第 4 章中基准系统的虚拟机镜像文件到本地目录。

（2）修改虚拟机网络连接方式为桥接（Bridge），此时，虚拟机与主机（Host）共同连接在一个物理网络上。

（3）启动虚拟机。

（4）修改网络配置。

```
$sudo vi /etc/sysconfig/network-scripts/ifcfg-eth0
```

> 我们实验的网络是通过 DHCP 分配 IP 地址的，因此在虚拟机配置成 DHCP 方式，具体步骤参见视频。

如果读者网络 IP 是事先设置好的，没有 DHCP 服务器，那么在此就要将虚拟机系统的 IP 地址设置好，设置的参数包括：HWADDR（MAC 地址），IPADDR（IP 地址），GATEWAY（网关地址）。此外，还要设置 DNS 服务器地址。

```
$sudo vi /etc/resolv.conf
```

DNS 通常设置为网关地址即可。

```
nameserver 192.168.1.1
```

配置完毕后，重启网卡。

```
$sudo service network restart
```

网卡重启完毕后。

检查 IP 地址是否设置正确。

```
$ip addr show
```

ping 网关地址、DNS 地址，看是否能够连接。如果可以连接，启动浏览器，输入网址，再次验证，本实验截图如图 6.2 所示。

图 6.2　网络访问界面

2．上传软件包

（1）准备以下软件包：Hadoop-0.20.2.tar.gz，AvatarNode.20.patch， Apache-ant-1.8.1.tar.gz。下载地址请见第 4 章 4.4.4 节。

（2）使用 SecureFX 将以上软件包复制到虚拟机的/home/user 目录下，如图 6.3 所示。

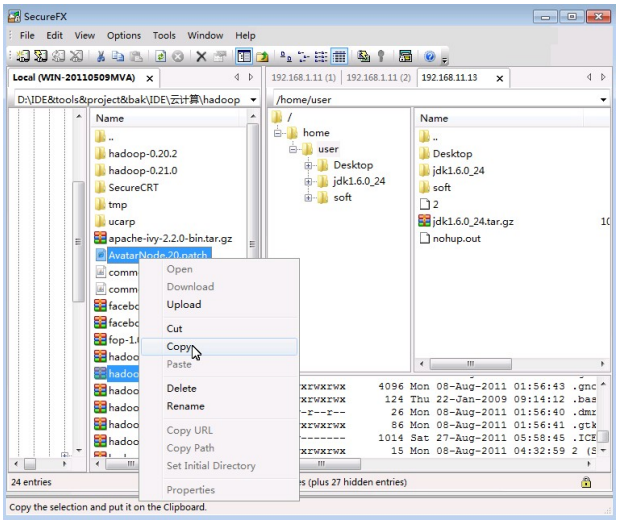

图 6.3　文件复制界面

3．安装 ant

ant 是一个 Build 工具，类似于 make。

（1）将 Apache-ant-1.8.1.tar.gz 从/home/user 复制到/usr/local 目录下。

（2）双击解压，Apache-ant-1.8.1.tar.gz 将解压到当前目录。

（3）编辑/etc/profile，在末尾添加：

```
export ANT_HOME=/usr/local/apache-ant-1.8.1
```

（4）执行命令使配置生效：

```
$source /etc/profile
```

4．解压 hadoop-0.20.2

（1）将 Hadoop-0.20.2.tar.gz 从/home/user 复制到/usr/local 目录下。

（2）双击解压，Hadoop-0.20.2.tar.gz 将解压到当前目录，目录名为 hadoop-0.20.2。

5．将补丁(AvatarNode.20.patch)复制到/usr/local/hadoop-0.20.2/

6．打补丁

```
$cp /home/user/ AvatarNode.20.patch /usr/local/hadoop-0.20.2/
$cd /usr/local/hadoop-0.20.2/
$patch -p0 -E <  AvatarNode.20.patch
```

7．修改源码

（1）修改 Standby.java。

将/usr/local/hadoop-0.20.2/src/contrib/highavailability/src/java/org/apache/hadoop/hdfs/server/namenode/Standby.java 的第 341 行：

```
fsImage.saveNamespace(true);
```

修改为：

```
fsImage.saveFSImage();
```

（2）修改 AvatarNode.java。

在/usr/local/hadoop-0.20.2/src/contrib/highavailability/src/java/org/apache/hadoop/hdfs/server/namenode/AvatarNode.java 中 import 三个类：

```
import org.apache.hadoop.security.UnixUserGroupInformation;
import org.apache.hadoop.security.UserGroupInformation;
```

并在第 210 行 initialize(Configuration conf)方法内部添加语句：

```
   try{
   UserGroupInformation.setCurrentUser(UnixUserGroupInformation.login(co
nf));
   } catch (LoginException e){

   }
```

（3）删除/usr/local/hadoop-0.20.2/build.xml 中的内容。

删除 build.xml 中 Documentation 部分的 target 标签对之间的所有内容（target 标签本身不删）。

> 去除的内容负责生成 Hadoop 的 doc，如果不去除这部分内容，在 Build 的时候会报错，去除后， Build 之后将原来的 doc 复制过去即可，不影响使用。

```
   <!-- ================================== -->
   <!-- Documentation                      -->
   <!-- ================================== -->
   <target name="docs" depends="forrest.check" description="Generate
forrest-based documentation. To use, specify -Dforrest.home=&lt;base of
Apache Forrest installation&gt; on the command line." if="forrest.home">
   </target>
   <target name="cn-docs" depends="forrest.check, init"
   description="Generate forrest-based Chinese documentation. To use,
specify -Dforrest.home=&lt;base of Apache Forrest installation&gt; on the

   if="forrest.home">
   </target>
   <target name="forrest.check" unless="forrest.home"
depends="java5.check">
   </target>
```

```
    <target name="java5.check" unless="java5.home">
    </target>
    <target name="javadoc-dev" description="Generate javadoc for hadoop
developers">
    </target>
    <target name="javadoc" depends="compile, ivy-retrieve-javadoc"
description= "Generate javadoc">
    </target>
    <target name="api-xml"

    </target>
    <target name="write-null">
    </target>
    <target name="api-report" depends="ivy-retrieve-jdiff,api-xml">
    </target>
    <target name="changes-to-html" description="Convert CHANGES.txt into an
html file">
    </target>
```

8．复制 doc

由于在第 7 步中注释掉了 doc 的产生，因此新建/usr/local/hadoop-0.20.2/build/目录，将/usr/local/hadoop-0.20.2/目录下的 docs 复制到/usr/local/hadoop-0.20.2/build/下。

9．编译打包

```
    $cd /usr/local/hadoop-0.20.2/
    $ ant -Djava5.home=$Java5Home compile tar
```

如图 6.4 所示的编译过程中会从网上下载一些依赖包，之后做一些小的修改再编译的话就不用连上网了。

```
[ivy:resolve] ........................... (37kB)
[ivy:resolve] .. (0kB)
[ivy:resolve]   [SUCCESSFUL ] commons-logging#commons-logging;1.0.4!commons-logging.jar (1896
ms)
[ivy:resolve] downloading http://repo1.maven.org/maven2/log4j/log4j/1.2.15/log4j-1.2.15.jar .
..
[ivy:resolve] ..........................................................................
..........................................................................
.......
[ivy:resolve] ..........................................................................
.......... (382kB)
[ivy:resolve] .. (0kB)
[ivy:resolve]   [SUCCESSFUL ] log4j#log4j;1.2.15!log4j.jar (2991ms)
[ivy:resolve] downloading http://repo1.maven.org/maven2/commons-httpclient/commons-httpclient
/3.0.1/commons-httpclient-3.0.1.jar ...
[ivy:resolve] ..........................................................................
..........................................................................
..................... (273kB)
```

图 6.4　编译过程

10．生成可使用的 tar 包

编译后生成的压缩包为/usr/local/hadoop-0.20.2/build/hadoop-0.20.3-dev.tar.gz，解压后将：

```
/usr/local/hadoop-0.20.2/build/hadoop-0.20.3-dev/contrib/highavailabi
```

复制到：

```
/usr/local/hadoop-0.20.2/build/hadoop-0.20.3-dev/lib
```

再把 hadoop-0.20.3-dev 压缩成 tar 包，这个包就是可用来部署的 Avatar 包了。

6.2.2　Hadoop 源码本地 Build

Hadoop 源码本地 Build 需要用到上节中搭建好的环境，也就是说，如果要实现 Hadoop 源码本地 Build，首先要完成上节中的步骤。具体视频参见“\视频\6 AvatarNode 使用\2 Avatar 无网络编译.exe”。后续步骤如下。

（1）准备虚拟机环境

- 关闭上节中虚拟机
- 修改虚拟机的网卡模式为 host-only，使其与互联网断开

- 启动虚拟机

（2）复制源码包

从 hadoop-0.20.2 中将通过网络编译生成的用于部署的源码包 hadoop-0.20.3-dev.tar.gz 解压后 copy 到“/usr/local/”下。

（3）复制 build 目录和 build.xml

从 hadoop-0.20.2 中将 build 目录和 build.xml copy 到/usr/local/hadoop-0.20.3-dev/下。

（4）复制 ivy-2.0.0-rc2.jar

将/usr/local/hadoop-0.20.3-dev/ivy/ivy-2.0.0-rc2.jar copy 到/usr/local/hadoop-0.20.3-dev/lib/。

（5）修改 xml 文件

- 修改 build.xml

修改/usr/local/hadoop-0.20.3-dev/build.xml，把 build.xml 中 154 行注释掉，改为：

```
<property name="ivy_repo_url" value="file://${basedir}/
lib/ivy-2.0.0-rc2.jar"/>
```

- 修改 build-contrib.xml

修改/usr/local/hadoop-0.20.3-dev/src/contrib/build-contrib.xml 注释掉 68、69 行，改为：

```
<property name="ivy_repo_url" value="file://${hadoop.
```

（6）编译打包

在/usr/local/hadoop-0.20.2/下，执行：

```
$ant -Djava5.home=$Java5Home compile tar
```

编译后生成的压缩包为/usr/local/hadoop-0.20.2/build/hadoop-0.20.3-dev.tar.gz，解压后将/usr/local/hadoop-0.20.2/build/hadoop-0.20.3-dev/contrib/highavailability/hadoop-0.20.3-dev-highavailability.jar 复制到 /usr/local/hadoop-0.20.2/build/hadoop-0.20.3-dev/lib 下，再把 hadoop-0.20.3-dev 压缩成 tar 包，这个包就是可用来部署的 Avatar 包了。

6.2.3 NFS 服务器构建

详细操作请参考视频：\视频\6 AvatarNode 使用\3 建立 NFS 服务器

1. 准备虚拟机

（1）复制虚拟机镜像。

NFS 对虚拟机系统没有特殊要求，我们在此选择第 4 章中构建的 datanode03 虚拟机作为 NFS 服务器的镜像源文件，将其整个目录复制到另外的目录下，重命名为 NFS。

（2）设置网络为 host-only。

（3）修改虚拟机名称为 NFS。

（4）修改存储路径，将虚拟机磁盘文件的存储路径修改为当前镜像文件的存储路径。

（5）启动虚拟机。

2. 修改网络配置

（1）修改 MAC 地址和 IP 地址，其中 IP 地址设置为 192.168.1.10。

```
$vi /etc/sysconfig/network-scripts/ifcofg-eth0
```

（2）重启网卡。

```
$sudo service network restart
```

（3）检查设置。

```
$ifconfig
```

3. 检查 NFS 是否是开机启动

```
$chkconfig  -- list | grep nfs
```

如果 NFS 没有开机启动，就需要设置 NFS 服务为开机启动。

```
$sudo chkconfig nfs on
```

4. 配置/etc/exports

```
$sudo vi /etc/exports
```

内容如下：

```
/usr/local/hadoop/avatarshare *(rw,sync,no_root_squash)
```

其中/usr/local/hadoop/avatarshare 为 NFS 共享目录。

5. 重启 NFS 服务器节点

6. 修改主机名

在 NFS 服务器上执行：

```
$sudo vi /etc/sysconfig/network
```

修改主机名为 NFS：

```
HOSTNAME=NFS
```

7. 修改/etc/hosts

在 NFS 服务器上修改：

```
$sudo vi /etc/hosts
```

内容如下：

```
127.0.0.1    localhost
```

```
192.168.1.10 NFS
192.168.1.11 namenode0
192.168.1.12 namenode1
192.168.1.13 datanode00
192.168.1.14 datanode01
192.168.1.15 datanode02
192.168.1.16 datanode03
```

8. 重启 NFS 服务器

9. 启动 namenode0 节点

10. 在 namenode0 上挂载 NFS 目录

```
$mount -v -t nfs -o tcp,soft,rsize=32768,wsize=32768
```

> 注意：在 NFS 服务器和 namenode0 上都要创建/usr/local/hadoop/avatarshare 目录。

其中 192.168.1.10:/usr/local/hadoop/avatarshare 是作为 NFS 服务器上的共享目录存在的，如果没有此目录就需要新建一个。/usr/local/hadoop/avatarshare 是挂载点，即在 namenode0 挂载 NFS 后，对/usr/local/hadoop/avatarshare 目录中的所有操作，实际是对 NFS 服务器上/usr/local/hadoop/avatarshare 目录的操作。在这里 NFS 服务器上的共享目录与 namenode0 上的映射目录使用了同样的绝对路径名，但这是两个不同节点上的目录。

11. 验证

如果挂载成功，使用下面的命令查看挂载信息：

```
$mount
```

或者：

```
$df -h
```

12. NFS 操作实例

（1）查看 namenode0 节点上的/usr/local/hadoop/avatarshare 目录内容。图 6.5 显示只有一个子目录“111”。

```
[user@namenode0 ~]$ ls /usr/local/hadoop/avatarshare/
111
```

图 6.5　namenode0 挂载点内容

（2）在 namenode0 节点上/usr/local/hadoop/avatarshare 目录下新建目录“222”，如图 6.6 所示。

```
[user@namenode0 ~]$ mkdir /usr/local/hadoop/avatarshare/222
[user@namenode0 ~]$ ls /usr/local/hadoop/avatarshare/
111  222
```

图 6.6　namenode0 挂载点内容

（3）在 NFS 节点上，可以看到共享目录/usr/local/hadoop/avatarshare 下的子目录与 namenode0 节点上映射目录中的相同，如图 6.7 所示。

```
[user@NFS ~]$ ls /usr/local/hadoop/avatarshare/
111  222
```

图 6.7　NFS 共享目录内容

（4）在 NFS 节点上/usr/local/hadoop/avatarshare 目录下新建目录 NFS，如图 6.8 所示。

```
[user@NFS ~]$ ls /usr/local/hadoop/avatarshare/
111  222  NFS
```

图 6.8　NFS 共享目录内容

（5）在 namenode0 节点上，可以看到共享目录/usr/local/hadoop/avatarshare 下的子目录与 NFS 节点上的映射目录中的相同，如图 6.9 所示。

```
[user@namenode0 ~]$ ls /usr/local/hadoop/avatarshare/
111  222  NFS
```

图 6.9　NFS 共享目录内容

6.2.4　Avatar 分发与部署

操作视频请参考：\视频\6 AvatarNode 使用\ 4 Avatar 分发与部署.exe

1. 准备虚拟机

（1）准备 namenode0 虚拟机。在此，我们直接选择第 4 章中构建的 namenode0 虚拟机，作为 namenode0 的镜像源文件。

（2）准备 namenode1 虚拟机。在此，我们直接选择第 4 章中构建的 namenode1 虚拟机，作为 namenode1 的镜像源文件。

（3）NFS 服务器。NFS 服务器在上节中已准备好，直接使用。

（4）准备 datanode00 虚拟机。在此，我们选择第 4 章中构建的 datanode00 虚拟机，作为 datanode00 的镜像源文件，将其整个目录复制到另外的目录下。

（5）准备 datanode01 虚拟机。在此，我们选择第 4 章中构建的 datanode01 虚拟机，作为 datanode01 的镜像源文件，将其整个目录复制到另外的目录下。

（6）准备 datanode02 虚拟机。在此，我们选择第 4 章中构建的 datanode02 虚拟机，作为 datanode02 的镜像源文件，将其整个目录复制到另外的目录下。

（7）准备 datanode03 虚拟机。在此，我们选择第 4 章中构建的 datanode03 虚拟机，作为 datanode03 的镜像源文件，将其整个目录复制到另外的目录下。

2. 启动虚拟机集群

3. 使用 SecureCRT 连接到虚拟机

4. 将编译好的 Avatar 包远程复制到各个节点（NFS Server 除外）

```
$scp -r hadoop-0.20.3-dev 192.168.1.12:/usr/local/
```

192.168.1.12 只是 namenode1 的 IP 地址， 依次用其他节点的 IP 地址替换，完成复制。

6.2.5　Primary（namenode0）节点配置

1. Ucarp 在第 4 章中已配置好

2. 配置/etc/profile

```
export HADOOP_HOME=/usr/local/hadoop-0.20.3-dev
```

使设置生效：

```
$ source /etc/profile
```

3. 配置/etc/hosts

```
127.0.0.1    localhost
192.168.1.10 NFS
192.168.1.11 namenode0
192.168.1.12 namenode1
192.168.1.13 datanode00
192.168.1.14 datanode01
192.168.1.15 datanode02
192.168.1.16 datanode03
```

4. 配置 masters 和 slaves 文件

这两个文件主要在执行/usr/local/hadoop-0.20.3-dev/bin/start-all.sh 等脚本时被调用，用来远程调用其他节点的脚本。

Masters 文件：

```
$vi /usr/local/hadoop-0.20.3-dev/conf/masters
```

内容如下：

```
192.168.1.11
```

Slaves 文件：

```
$vi /usr/local/hadoop-0.20.3-dev/conf/slaves
```

内容如下：

```
192.168.1.13
192.168.1.14
192.168.1.15
192.168.1.16
```

5. 修改 hadoop-env.sh

管理员可在/usr/local/hadoop-0.20.3-dev/conf/hadoop-env.sh 脚本内对 Hadoop 守护进程的运行环境做特别指定。

```
$vi /usr/local/hadoop-0.20.3-dev/conf/hadoop-env.sh
```

文件末尾添加 JDK 的绝对路径：

```
export JAVA_HOME=/home/user/jdk1.6.0_24
```

将 hadoop-env.sh 远程复制到其他节点的/usr/local/hadoop-0.20.3-dev /conf 目录下。

6. 配置 core-site.xml

/usr/local/hadoop-0.20.3-dev/conf/core-site.xml 文件主要用于配置 HDFS 系统基本属性，例如：全局参数、IO 参数、日志参数、IPC 参数等。

```
$vi /usr/local/hadoop-0.20.3-dev/conf/core-site.xml
```

内容如下：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<!-- special parameters for avatarnode -->
  <property>
```

```
        <value>hdfs://0.0.0.0:9000</value>
  </property>
  <property>
        <name>fs.default.name1</name>
        <value>hdfs://192.168.1.12:9000</value>
  </property>
<!-- special parameters for avatarnode -->
<property>
        <name>fs.default.name</name>
        <value>hdfs://192.168.1.9:9000</value>
</property>
<property>
        <name>hadoop.tmp.dir</name>
        <value>/usr/local/hadoop/tmp</value>
</property>
```

- fs.default.name0 表示 Primary 监听地址，0.0.0.0 表示在所有地址上监听。
- fs.default.name1 表示 Standby 监听地址。
- fs.default.name 表示客户端访问 HDFS 的 NameNode 的地址。

7. 配置 hdfs-site.xml

```
$vi /usr/local/hadoop-0.20.3-dev/hdfs-site.xml
```

配置选项说明如下。

- dfs.name.dir.shared0 ： NameNode (primary) 镜像存储路径。
- dfs.name.edits.dir.shared0：NameNode (primary) 日志存储路径。
- dfs.http.address0：NameNode (primary) http 服务器地址。
- dfs.name.dir.shared1：NameNode (standby) 镜像存储路径。

- dfs.name.edits.dir.shared1：NameNode (standby) 日志存储路径。
- dfs.http.address1：NameNode (standby) http 服务器地址。
- dfs.http.address：NameNode http 服务器地址。
- dfs.name.dir：NameNode 镜像存储路径。
- dfs.name.edits.dir：NameNode 日志存储路径。

配置文件内容如下。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<!-- special parameters for avatarnode -->
<configuration>
<property>
       <name>dfs.name.dir.shared0</name>
       <value>/usr/local/hadoop/avatarshare/share0/namenode</value>
</property>
<property>
       <name>dfs.name.edits.dir.shared0</name>
       <value>/usr/local/hadoop/avatarshare/share0/editlog</value>
</property>
<property>
       <name>dfs.http.address0</name>
       <value>0.0.0.0:50070</value>
</property>
<property>
       <name>dfs.name.dir.shared1</name>
       <value>/usr/local/hadoop/avatarshare/share1/namenode</value>
```

```
<property>
       <name>dfs.name.edits.dir.shared1</name>
       <value>/usr/local/hadoop/avatarshare/share1/editlog</value>
</property>
<property>
       <name>dfs.http.address1</name>
       <value>192.168.1.12:50070</value>
</property>
<property>
       <name>dfs.http.address</name>
       <value>0.0.0.0:50070</value>
</property>
<property>
       <name>dfs.name.dir</name>
       <value>/usr/local/hadoop/local/namenode</value>
</property>
<property>
       <name>dfs.name.edits.dir</name>

</property> </configuration>


$vi /usr/local/hadoop-0.20.3-dev/conf/core-site.xml
```

6.2.6　Standby（namenode1）节点配置

1. Ucarp 在第 4 章中已配置好
2. 配置/etc/profile（同 namenode0）
3. 配置/etc/hosts（同 namenode0）

4. 配置 masters 和 slaves 文件（同 namenode0）
5. 配置 hadoop-env.sh（同 namenode0）
6. 配置 core-site.xml

内容如下：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<!-- special parameters for avatarnode -->
   <property>
        <name>fs.default.name0</name>
        <value>hdfs://192.168.1.11:9000</value>
  </property>
  <property>
        <name>fs.default.name1</name>
        <value>hdfs://0.0.0.0:9000</value>
  </property>
<!-- special parameters for avatarnode -->
<property>
       <name>fs.default.name</name>
       <value>hdfs://192.168.1.9:9000</value>
</property>
<property>
       <name>hadoop.tmp.dir</name>
       <value>/usr/local/hadoop/tmp</value>
</property>
```

7. 配置 hdfs-site.xml

```
$vi /usr/local/hadoop-0.20.3-dev/hdfs-site.xml
```

内容如下：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<!-- special parameters for avatarnode -->
<configuration>
<property>
       <name>dfs.name.dir.shared0</name>
       <value>/usr/local/hadoop/avatarshare/share0/namenode</value>
</property>
<property>
       <name>dfs.name.edits.dir.shared0</name>
       <value>/usr/local/hadoop/avatarshare/share0/editlog</value>
</property>
<property>
       <name>dfs.http.address0</name>
       <value>192.168.1.11:50070</value>
</property>
<property>
       <name>dfs.name.dir.shared1</name>
       <value>/usr/local/hadoop/avatarshare/share1/namenode</value>
</property>
<property>
       <name>dfs.name.edits.dir.shared1</name>
       <value>/usr/local/hadoop/avatarshare/share1/editlog</value>
</property>
```

```
        <name>dfs.http.address1</name>
        <value>0.0.0.0:50070</value>
</property>
<property>
        <name>dfs.http.address</name>
        <value>0.0.0.0:50070</value>
</property>
<property>
        <name>dfs.name.dir</name>
        <value>/usr/local/hadoop/local/namenode</value>
</property>
<property>
        <name>dfs.name.edits.dir</name>
        <value>/usr/local/hadoop/local/editlog</value>
</property>
```

6.2.7 Data Node 节点配置

datanode00~datanode03 的配置相同，下面以 datanode00 为例进行说明，配置结束后，可用 scp 命令将修改好的配置文件复制到其他 Data Node 节点。

1. 配置/etc/profile（同 namenode0）
2. 配置/etc/hosts（同 namenode0）
3. 配置 hadoop-env.sh（同 namenode0）
4. 配置 core-site.xml

```
$vi /usr/local/hadoop-0.20.3-dev/conf/core-site.xml
```

内容如下：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
```

```
<!-- Put site-specific property overrides in this file. -->
<configuration>
<!-- special parameters for avatarnode -->
  <property>
       <name>fs.default.name0</name>
       <value>hdfs://192.168.1.11:9000</value>
  </property>
  <property>
       <name>fs.default.name1</name>
       <value>hdfs://192.168.1.12:9000</value>
  </property>
<!-- special parameters for avatarnode -->
<property>
       <name>fs.default.name</name>
       <value>hdfs://192.168.1.9:9000</value>
</property>
<property>
       <name>hadoop.tmp.dir</name>
       <value>/usr/local/hadoop/tmp</value>
</property>
</configuration>
```

5. 配置 hdfs-site.xml

```
$vi /usr/local/hadoop-0.20.3-dev/conf/hdfs-site.xml
```

内容如下：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
```

```
<configuration>
<property>
       <name>dfs.http.address0</name>
       <value>192.168.1.11:50070</value>
</property>
<property>
       <name>dfs.http.address1</name>
       <value>192.168.1.12:50070</value>
</property>
<!-- special parameters for avatarnode -->
<property>
       <name>dfs.http.address</name>
       <value>192.168.1.9:50070</value>
</property>
<property>
       <name>dfs.replication</name>
       <value>3</value>
</property>
<property>
       <name>dfs.data.dir</name>
       <value>/usr/local/hadoop/block</value>
</property>
```

6.2.8 Client 节点配置

Client 与 Data Node 共用 1 个虚拟机节点，Client 访问 NameNode 的 IP 地址由 $HADOOP_HOME/conf/core-site.xml 中键 fs.default.name 所对应值所设置，该选项在 Data Node 已配置好，无须另外配置。

6.2.9　创建目录

1. namenode0(Primary)

创建以下目录：

```
/usr/local/hadoop/avatarshare/share0/namenode
/usr/local/hadoop/avatarshare/share0/editlog
/usr/local/hadoop/local/namenode
/usr/local/hadoop/local/editlog
/usr/local/hadoop/tmp
```

- /usr/local/hadoop/avatarshare/share0/namenode: namenode0 的 NFS 上磁盘镜像存储目录。
- /usr/local/hadoop/avatarshare/share0/editlog：namenode0 的 NFS 上日志存储目录。
- /usr/local/hadoop/local/namenode：namenode0 的磁盘镜像存储目录。
- /usr/local/hadoop/local/editlog：namenode0 的 NFS 上日志存储目录。
- /usr/local/hadoop/tmp：namenode0 的临时目录。

2. namenode1(Standby)

创建以下目录：

```
/usr/local/hadoop/avatarshare/share1/namenode
/usr/local/hadoop/avatarshare/share1/editlog
/usr/local/hadoop/local/namenode
/usr/local/hadoop/local/editlog
```

所建立的目录的作用与 namenode0 类似。

3. datanode00~datanode03

```
/usr/local/hadoop/block
/usr/local/hadoop/tmp
```

- /usr/local/hadoop/block：DataNode 的数据块存储路径。
- /usr/local/hadoop/tmp：DataNode 的临时目录。

4. NFS

```
/usr/local/hadoop/avatarshare/share0/namenode
/usr/local/hadoop/avatarshare/share0/editlog
/usr/local/hadoop/avatarshare/share1/namenode
```

这些目录分别与 namenode0 和 namenode1 上相同路径的目录对应。

6.2.10 挂载 NFS

在 namenode0(Primary)和 namenode 1(Standby)分别挂载 NFS，命令如下：

```
$sudo mount -v -t nfs -o tcp,soft,rsize=32768,wsize=32768
192.168.1.10:/usr/local/hadoop/avatarshare /usr/local/hadoop/avatarshare
```

挂载后在 namenode0 和 namenode1 上使用下面的命令检查挂载结果：

```
$df -h
```

或者：

```
$mount
```

6.2.11 启动 Ucarp

参考第 4 章 4.5.5 中第 3 节。

6.2.12　格式化

1. 在 namenode0(Primary)执行格式化命令

```
$cd /usr/local/hadoop-0.20.3-dev/
$rm -rf ../hadoop/local/*
$/usr/local/hadoop-0.20.3-dev/bin/hadoop namenode -format
```

将正常格式化后的 local 目录下 namenode、editlog 复制到 NFS 共享目录/usr/local/hadoop/avatarshare/share0/和/usr/local/hadoop/avatarshare/share1/中：

```
$cp -r ../hadoop/local/* ../hadoop/avatarshare/share1/
$cp -r ../hadoop/local/* ../hadoop/avatarshare/share0/
```

2. 在 namenode0(Primary)执行格式化命令

```
$bin/hadoop org.apache.hadoop.hdfs.server.namenode.AvatarNode -zero
```

6.2.13　系统启动

AvatarNode 集群有两种启动方式：

- 有日志后台启动
- 无日志前台启动

> 注意：启动前要保证所有节点的系统时间是同步的。

1. 有日志后台启动

（1）启动 namenode0(Primary)。

```
$/usr/local/hadoop-0.20.3-dev/bin/hadoop-daemon.sh start
org.apache.hadoop.hdfs.server
.namenode.AvatarNode -zero
```

（2）namenode1(Standby)。

```
   $/usr/local/hadoop-0.20.3-dev/bin/hadoop-daemon.sh start
org.apache.hadoop.hdfs.server
   .namenode.AvatarNode -one -standby -sync
```

（3）Data Node。依次启动 datanode00～datanode03。

```
   $/usr/local/hadoop-0.20.3-dev/bin/hadoop-daemon.sh start
org.apache.hadoop.hdfs.server
   .datanode.AvatarDataNode.AvatarDataNode
```

2. 无日志前台启动

（1）namenode0(Primary)。

```
   $/usr/local/hadoop-0.20.3-dev/bin/hadoop
org.apache.hadoop.hdfs.server.namenode
   .AvatarNode -zero
```

（2）namenode1(Standby)。

```
   $/usr/local/hadoop-0.20.3-dev/bin/hadoop
org.apache.hadoop.hdfs.server.namenode
   .AvatarNode -one -standby -sync
```

（3）Data Node。

```
   $/usr/local/hadoop-0.20.3-dev/bin/hadoop
org.apache.hadoop.hdfs.server.datanode.
```

6.2.14 检查

查看启动情况。

```
   $/usr/local/hadoop-0.21.0/bin/hadoop dfsadmin -report
```

HDFS 集群状态如图 6.10 所示。

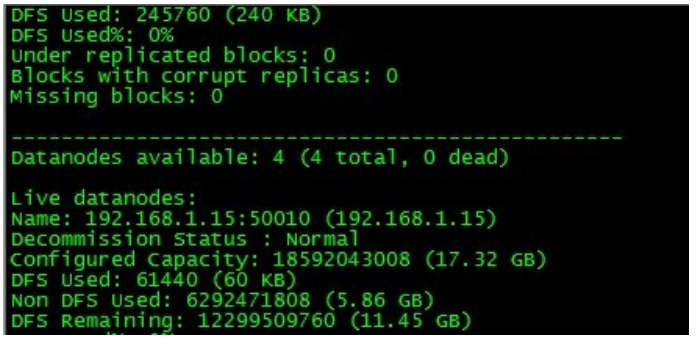

图 6.10　HDFS 集群状态

6.2.15　NameNode 失效切换写文件实验

具体操作参见：\视频\6 AvatarNode 使用\7 Avatar 写实验 Primary 宕机.exe

1. 启动虚拟机

启动的虚拟机包括：namenode0、namenode1、datanode00～03、NFS。如果虚拟机已经启动，则重启各个虚拟机恢复到初始状态。

2. 用 SecureCRT 连接到虚拟机

3. 启动 Ucarp

（1）先在 namenode0 上启动：

```
$nohup /etc/ucarp.sh &
```

（2）后在 namenode1 上启动：

```
$nohup /etc/ucarp.sh &
```

启动成功后，namenode0 上的虚拟 IP（192.168.1.9）应该工作，可使用 ssh 登录该 IP 进行验证。

4. 挂载 NFS

挂载之前，检查 NFS 服务器上的 NFS 服务是否已启动。然后分别在 namenode0 和 namenode1 上执行。下列命令会将 NFS 服务器的/usr/local/hadoop/avatarshare

目录分别挂载到 namenode0 和 namenode1 的/usr/local/hadoop/avatarshare 目录下。

在/usr/local/hadoop/avatarshare 有 share0 和 share1 两个目录。其中：

- 按照配置，Primary(namenode0) 会将 fsimage 文件保存在 share0/namenode 目录下，edits 文件保存在 share0/editlog 目录下；
- Standby(namenode1)fsimage 文件保存在 share1/namenode 目录下，edits 文件保存在 share1/editlog 目录下；
- 此外，Standby 还将开启一个线程，定期读取 share0/editlog 下的日志文件，用于与 Primay 进行同步。

```
   $echo "123456" | sudo -S mount -v -t nfs -o
tcp,soft,rsize=32768,wsize=32768
192.168.1.10:/usr/local/hadoop/avatarshare /usr/local/hadoop/avatarshare
```

5. 格式化 HDFS

（1）在 namenode0 上执行：

```
   $/usr/local/hadoop-0.20.3-dev/bin/hadoop
```

根据配置，格式化产生的元数据文件如表 6.2 所示。

表 6.2 元数据文件表

文 件	作 用
/usr/local/hadoop/local/namenode/current/fsimage /usr/local/hadoop/avatarshare/share0/namenode/current/fsimage	空镜像文件，内容包括：namespaceID、root 的子树的数目、命名空间的时间戳、目录树等信息
/usr/local/hadoop/local/editlog /current/edit /usr/local/hadoop/avatarshare/share0/editlog /current/edits	空日志文件，内容包括：版本信息、文件结束标志

续表

文 件	作 用
/usr/local/hadoop/local/namenode/current/fstime /usr/local/hadoop/local/editlog/current/fstime /usr/local/hadoop/avatarshare/share0/namenode/current/fstime /usr/local/hadoop/avatarshare/share0/editlog /current/fstime	存储 CheckPoint 时间
/usr/local/hadoop/local/namenode/current/VERSION /usr/local/hadoop/local/editlog/current/VERSION /usr/local/hadoop/avatarshare/share0/namenode/current/VERSION /usr/local/hadoop/avatarshare/share0/editlog /current/VERSION	版本文件，存储基本信息

（2）清除 Block 文件。在 datanode00~datanode03 上执行。

```
$rm -r /usr/local/hadoop/block/*
```

6. 启动 HDFS

（1）在 namenode0 以 Primary 启动：

```
$/usr/local/hadoop-0.20.3-dev/bin/hadoop-daemon.sh start
```

AvatarNode Primary 启动后判断本节点角色后会将 Avatar 的相关属性（在/usr/local/hadoop-0.20.3-dev/conf/core-site.xml 和/usr/local/hadoop-0.20.3-dev/conf/hdfs-site.xml 中属性名结尾为 0 或 1 的属性）转换为对应的 Hadoop 属性。

接着产生一个 AvatarNode 实例，并对父类 NameNode 进行初始化，在初始化时会启动 Web 服务器（默认地址 0.0.0.0:50070）、RPC 远程调用服务器和相关服务线程。

在此之前 NameNode 会做一次 CheckPoint。

即系统通过判断 /usr/local/hadoop/local/namenode/current/fstime 与 /usr/local/hadoop/avatarshare/share0/namenode/current/fstime 文件中的时间将磁盘上的镜像文件 /usr/local/hadoop/local/namenode/current/fsimage 与

/usr/local/hadoop/avatarshare/share0/namenode/current/fsimage 中最新的加载到内存生成命名空间即目录树。

同样将日志文件中/usr/local/hadoop/local/editlog/current/edits 与/usr/local/hadoop/avatarshare/share0/editlog/current/edits 最新的也会被加载到内存的命名空间中。

日志文件的每条日志记录被解析为对元数据的操作，当日志文件被加载结束也就完成了镜像与日志的合并，在内存中生成前一次系统关闭时的目录树。

接下来，内存中的最新镜像会被序列化保存到磁盘的目录镜像存储路径下的/usr/local/hadoop/local/namenode/current/fsimage 与/usr/local/hadoop/avatarshare/share0/namenode/current/fsimage 中，同时日志目录镜像存储路径下的/usr/local/hadoop/local/editlog/current/edits 与/usr/local/hadoop/avatarshare/share0/editlog/current/edits 会被清空，两个路径下的 fstime 文件会存储此次 CheckPoint 的时间。之后会等待用户请求和 DataNode 的心跳信息。

（2）在 nameode1 以 Standby 启动：

```
$/usr/local/hadoop-0.20.3-dev/bin/hadoop-daemon.sh start
```

- 参数“-one -standby”：表示本节点的角色是 Standby，如果是“-zero”或空则表示本节点的角色是 Primary。
- 参数“-sync”：表示与另外一个 AvatarNode 节点同步。

AvatarNode Standby 的启动过程相对于 AvatarNode Primary 的启动过程来说，其不同之处有两点。

首先，由于有参数“-sync”，AvatarNode Standby 启动时，会将 NFS 上 AvatarNode Primary 的镜像文件/usr/local/hadoop/avatarshare/share0/namenode/current/fsimage 和日志文件/usr/local/hadoop/avatarshare/share0/editlog/current/edits 等元数据文件覆盖 NFS 上 AvatarNode Standby 的和本地的镜

像文件/usr/local/hadoop/local/namenode/current/fsimage 同时清空日志文件/usr/local/hadoop/local/editlog/current/edits。

其次，AvatarNode 的构造方法中多启动了一个 Standby 线程，此线程周期性地作 CheckPoint，并将 CheckPoint 后产生的最新的磁盘镜像文件/usr/local/hadoop/local/namenode/current/fsimage 上传到 AvatarNode Primary 上并覆盖对应的文件。并且在此线程中启动了一个 Ingest 线程，其主要作用是读取 NFS 上 AvatarNode Primary 的磁盘日志文件/usr/local/hadoop/avatarshare/share0/editlog/current/edits，周期性地读取日志文件中自上次读取结束时产生的新的日志记录，并将这些记录转换成元数据更新 AvatarNode Standby 的内存目录树。这样就使得 AvatarNode Standby 与 AvatarNode Primary 的元数据保持一致。

（3）在 datanode00～datanode03 上依次执行：

```
   $/usr/local/hadoop-0.20.3-dev/bin/hadoop-daemon.sh start
org.apache.hadoop.hdfs.server.datanode.AvatarDataNode
```

AvatarDataNode 启动后会首先收集数据块存储目录/usr/local/hadoop/block/下的数据块文件的信息，在内存中形成数据块信息集合。之后会分别向 AvatarNode Standby 与 AvatarNode Primary 发送心跳信息，并会将数据块信息集合上报给 AvatarNode Standby 与 AvatarNode Primary。

（4）检查是否启动成功。

在 namenode1 上执行，通过 Primary(namenode0)查看 HDFS 状态。

```
   $/usr/local/hadoop-0.20.3-dev/bin/hadoop dfsadmin -report
```

通过 Standby(namenode1)查看 HDFS 状态。

```
   $/usr/local/hadoop-0.20.3-dev/bin/hadoop dfsadmin
-Dfs.default.name=hdfs://192.168.1.12:9000 -report
```

通过 Web 访问 192.168.1.11:50070 以及 192.168.1.12:50070 两个地址，查看 HDFS 状态。

7. 复制文件

（1）删除已有文件。

```
$cd /usr/local/hadoop-0.20.3-dev/bin/

$bin/hadoop -rm -skipTrash /tmp/192.168.1.*/*
```

（2）复制文件。

在 NFS（192.168.1.10）上使用脚本/usr/local/hadoop-0.20.3-dev/bin/test04_IO.sh，该脚本远程调用 datanode00（192.168.1.13）和 datanode01（192.168.1.14）上的脚本/usr/local/hadoop-0.20.3-dev/bin/write_to_hdfs.sh，复制 80 个 10MB 的文件到 HDFS 上的/tmp/192.168.1.13/和/tmp/192.168.1.14/路径下，同一个节点上两个文件复制的时间间隔为 10 秒。其中 10MB 的文件由/usr/local/hadoop-0.20.3-dev/bin/makefile 程序生成为本地/tmp/test.iso。

> 相关脚本代码参见光盘目录“脚本\6 AvatarNode 使用”。

在 namenode1(Standby)上查看已经复制到 HDFS 上的文件，如图 6.11 所示。

```
[user@namenode1 ~]$ /usr/local/hadoop-0.20.3-dev/bin/hadoop dfs -Dfs.default.name=hdfs://192.168.1.12:9000 -lsr /
drwxr-xr-x   - user supergroup          0 2011-08-28 13:33 /tmp
drwxr-xr-x   - user supergroup          0 2011-08-28 13:34 /tmp/192.168.1.13
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.13/10000
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.13/10001
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.13/10002
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.13/10003
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:34 /tmp/192.168.1.13/10004
drwxr-xr-x   - user supergroup          0 2011-08-28 13:34 /tmp/192.168.1.14
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.14/10000
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.14/10001
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.14/10002
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.14/10003
```

图 6.11　复制结果图

Copy 命令在 HDFS 会创建新的文件，从而在 Primary 上产生新的元数据记录，其记录也会同步更新到 Standby。以 Copy 命令的元数据更新为例，Primary 和 Standby 的元数据同步过程描述如下。

① 在客户端，用户会对 HDFS 进行操作，包括对数据的操作，例如文件的写入和读出，还有对元数据的操作，例如对目录的建立和查看，这些都是用户操作。

② 每一个用户操作都会被分解为多条日志，并被写入到日志文件中，例如

copyFromLocal（向 HDFS 复制文件）的命令，就被分解为 openFile、addBlock 等多条日志。在 AvatarNode Primary 上，日志不但会写入本地日志文件 /usr/local/hadoop/local/editlog/current/edits，而且通过在配置文件里配置日志文件目录，也会写入到 NFS 上的日志文件：

```
/usr/local/hadoop/avatarshare/share0/editlog/current/edits
```

③ AvatarNode Standby 节点上的 Ingest 线程，默认每 3 秒读取一次 NFS 上 AvatarNode Primary 的日志文件，将读取到的每条日志应用到内存中的命名空间，以使与 AvatarNode Primary 上的元数据保持一致。

8. 模拟 Primary(namenode0)失效

重启 namenode0。这时 copy 的过程中会报两种错误，如图 6.12 和图 6.13 所示。

```
copying tmp/test.iso to /tmp/192.168.1.14/10018

copying tmp/test.iso to /tmp/192.168.1.13/10018
copyFromLocal: Call to 192.168.1.9/192.168.1.9:9000 failed on local exception: java.io.IOException: Connection reset by peer
copyFromLocal: Call to 192.168.1.9/192.168.1.9:9000 failed on local exception: java.io.IOException: Connection reset by peer
```

图 6.12　复制时 NameNode 无连接报错

```
copying tmp/test.iso to /tmp/192.168.1.14/10019

copying tmp/test.iso to /tmp/192.168.1.13/10019
copyFromLocal: org.apache.hadoop.hdfs.server.namenode.SafeModeException: Cannot create file/tmp/192.168.1.13/10019. Name node
 is in safe mode.
copyFromLocal: org.apache.hadoop.hdfs.server.namenode.SafeModeException: Cannot create file/tmp/192.168.1.14/10019. Name node
 is in safe mode.
```

图 6.13　复制时 NameNode 处于保护模式报错

9. 切换 Standby(namenode1)为 Primary

（1）在 namenode1 执行 saveNamespace 命令

Standby 平时不更新 share1 下面的日志文件，镜像文件依靠定期的 Checkpoint 进行更新，share1 下面的日志文件始终为空，镜像文件为最后一次 Checkpoint 的内存镜像，和当前内存的镜像有时间差。所以，需要通过 saveNamespace 将内存元数据导出到磁盘形成最新的元数据文件。最新的元数据文件有两个作用。

- 首先，防止 Standby 切换前失效，造成元数据丢失。如果不保存的话，Standby 在切换前失效，那么从最后一次 CheckPoint 的时刻到失效的时刻之间所产生的元数据将丢失。

- 其次，供切换后新的 Standby 启动使用。新的 Standby 启动时因为“-sync”参数要从 NFS 上 Primary 的镜像存储目录中 copy 镜像到本地，所以在做 saveNamespace 之后，可以确保新的 Standby 加载的镜像将是最新的。

```
$/usr/local/hadoop-0.20.3-dev/bin/hadoop dfsadmin -saveNamespace
```

（2）执行切换命令。

```
$/usr/local/hadoop-0.20.3-dev/bin/hadoop
org.apache.hadoop.hdfs.AvatarShell -setAvatar primary
```

namenode1 切换为 Primary 的过程有以下 3 个主要步骤。

① 管理员手工执行命令将 namenode1 切换为 Primary。

② namenode1 上 Ingest 线程最后一次从 NFS 中的 namenode0 对应日志的目录中读取 namenode0 的日志。

③ namenode1 上结束 Ingest 线程、结束 Standby 线程。

（3）检查切换结果如图 6.14 所示。

```
$/usr/local/hadoop-0.20.3-dev/bin/hadoop
org.apache.hadoop.hdfs.AvatarShell -showAvatar
```

```
[user@namenode1 /]$ /usr/local/hadoop-0.20.3-dev/bin/hadoop org.apache.hadoop.hdfs.AvatarShell -setAvatar primary
11/10/31 18:39:52 INFO hadoop.AvatarShell: AvatarShell connecting to /192.168.1.9:9001
[user@namenode1 /]$ /usr/local/hadoop-0.20.3-dev/bin/hadoop org.apache.hadoop.hdfs.AvatarShell -showAvatar
11/10/31 18:40:08 INFO hadoop.AvatarShell: AvatarShell connecting to /192.168.1.9:9001
The current avatar of /192.168.1.9:9001 is Primary
[user@namenode1 /]$ ssh 192.168.1.9 hostname
namenode1
```

图 6.14　切换状态图

10. 在 namenode0 上以 Standby 启动

执行命令如下。

```
$/usr/local/hadoop-0.20.3-dev/bin/hadoop


.AvatarNode -zero -standby -sync
```

11. 实验结果

在测试过程中，2 个 Write Client（192.168.1.13、192.168.1.14）分别向 HDFS 写入 80 个文件，在切换过程中，一共有 8 个文件写入失败，测试结果如表 6.3 所示。

表 6.3　写测试结果

写客户端的 IP	namenode0 失效 namenode1 切换为 primary 期间出现的文件错误	
	namenode0 失效时正在复制的文件（HDFS 上文件大小为 0）	虚拟 IP 地址切换到 namenode1 时，由于保护模式被 HDFS 拒绝 copy 的文件
192.168.1.13	/tmp/192.168.1.13/10018	/tmp/192.168.1.13/10019 ~ /tmp/192.168.1.13/10032
192.168.1.14	/tmp/192.168.1.14/10018	/tmp/192.168.1.14/10019 ~ /tmp/192.168.1.14/10032

分析：

- 写入操作处于切换期间时，会写入失败。
- 提交写入操作请求，热备节点处于保护模式时，会写入失败。
- 其余阶段的写操作都能正常进行。
- 写操作失败时，会报错退出，可靠性需要由上层应用程序来保证。

6.2.16　NameNode 失效切换读文件实验

具体视频参见：\视频\6 AvatarNode 使用\6 Avatar 读实验 Primary 宕机.exe

1. Avatar 启动

确定 HDFS 系统已启动，如未启动，参见 6.2.15 中启动部分。

2. 复制文件

（1）删除已有文件。

```
$cd /usr/local/hadoop-0.20.3-dev/bin/
```

（2）复制文件。

在 NFS（192.168.1.10）上使用脚本/usr/local/hadoop-0.20.3-dev/bin/test04_IO.sh，该脚本远程调用 datanode00（192.168.1.13）和 datanode01（192.168.1.14）上的脚本/usr/local/hadoop-0.20.3-dev/bin/write_to_hdfs.sh，复制 80 个 10MB 的文件到 HDFS 上的/tmp/192.168.1.13/和/tmp/192.168.1.14/路径下，同一个节点上两个文件复制的时间间隔为 10 秒。

具体内容参见光盘目录：\脚本\6 AvatarNode 使用

在 namenode1（Standby）上查看已经复制到 HDFS 上的文件，如图 6.15 所示。

```
[user@namenode1 ~]$ /usr/local/hadoop-0.20.3-dev/bin/hadoop dfs -Dfs.default.name=hdfs://192.168.1.12:9000 -lsr /
drwxr-xr-x   - user supergroup          0 2011-08-28 13:33 /tmp
drwxr-xr-x   - user supergroup          0 2011-08-28 13:34 /tmp/192.168.1.13
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.13/10000
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.13/10001
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.13/10002
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.13/10003
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:34 /tmp/192.168.1.13/10004
drwxr-xr-x   - user supergroup          0 2011-08-28 13:34 /tmp/192.168.1.14
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.14/10000
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.14/10001
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.14/10002
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:33 /tmp/192.168.1.14/10003
-rw-r--r--   3 user supergroup   10000000 2011-08-28 13:34 /tmp/192.168.1.14/10004
```

图 6.15　复制结果图

3. 读取文件

等待步骤 2 中所有文件复制完毕，执行如下命令。

（1）从 datanode00（192.168.1.13）复制 HDFS 上的文件到本地。HDFS 的目录为/tmp/192.168.1.13/，本地目录为/tmp/192.168.1.13/目录下。在 datanode00（192.168.1.13）上执行脚本 read_from_hdfs_IP.sh。

```
$mkdir -p /tmp/192.168.1.13
```

如图 6.16 所示，在 datanode00(client00)上，可以看到文件正在从 HDFS 复制到本地。

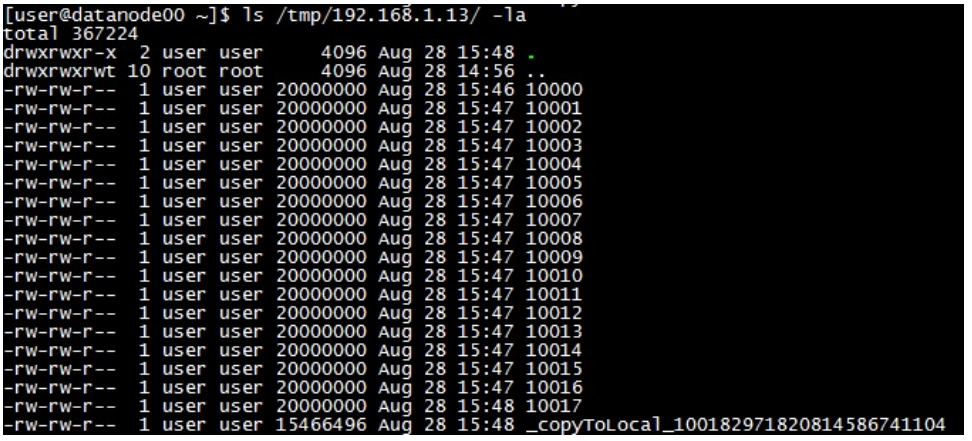

图 6.16　复制过程图

（2）在 datanode01(client01)执行：

```
$mkdir -p /tmp/192.168.1.14
```

4. 重启 namenode0(Primary)，模拟失效

此时，datanode00(client00)和 datanode01(client01)正在复制 HDFS 文件到本地，当 namenode0 无法提供服务时，当前正在读取的文件操作会失败，在实验脚本中，每个文件的复制操作对应了一个进程，因此，namenode0 无法提供服务只会影响当前操作，当虚拟 IP 接替后，由于 Standby 可以响应读取操作，因此后续的复制操作无需切换 Standby 到 Primary 就可以继续进行。

5. namenode1(Standby)切换为 Primary

（1）在 namenode1(Standby)执行 saveNamespace 命令：

```
$/usr/local/hadoop-0.20.3-dev/bin/hadoop dfsadmin -saveNamespace
```

（2）执行切换命令：

```
$/usr/local/hadoop-0.20.3-dev/bin/hadoop
org.apache.hadoop.hdfs.AvatarShell -setAvatar primary
```

（3）检查切换结果，查看本机的角色是 Primary 还是 Standby。

```
   $/usr/local/hadoop-0.20.3-dev/bin/hadoop
```

6. 在 namenode0 上以 Standby 启动。执行命令如下

```
   $usr/local/hadoop-0.20.3-dev/bin/hadoop
org.apache.hadoop.hdfs.server.namenode
   .AvatarNode -zero -standby -sync
```

7. 结论

- 当 Primary 无法提供服务时，当前的读取操作会出错跳出。
- 当 Standby 节点接替其虚拟 IP 后，后续新的读取操作可进行，因此在切换中间，有可能造成读取文件的丢失，其可靠性需要由读取程序来保证。
- 对于读取操作，Standby 无需切换到 Primary。
- 对于写操作，由于 Standby 处于保护模式，因此必须切换到 Primary 才能支持。

6.3 Avatar FaceBook 版本的使用

Avatar FaceBook 版本的源码位于 FaceBook 维护的 Hadoop 包内，无需打补丁，此外，该版本还自带了改进版的客户端，该客户端配合 Zookeeper 使用，可以配置 2 个 NameNode 的地址，当第 1 个地址不可用时，自动访问第 2 个 IP 地址，这样，无需再使用虚拟 IP 工具。

> 具体视频参见: \视频\6 AvatarNode 使用\5 Avatar 使用 zookeeper 做切换的过程.exe

6.3.1 Hadoop FaceBook 版本安装

Hadoop FaceBook 版本的安装很简单，将安装包解压编译后即可使用。

1. 准备软件包

我们所使用的版本为 facebook-hadoop-20-append，下载地址为：

```
http://www.vdisk.cn/down/index/8993830A5545
```

压缩包文件为：

```
facebook-hadoop-20-append-b6449e4.tar.gz
```

图 6.17　下载页面

2. 解压编译

假设安装目录为/usr/local/，将 facebook-hadoop-20-append-b6449e4.tar.gz 解压到此目录下。

> 编译过程与打 Avatar 补丁的版本类似，只是不用修改源码，具体步骤参考 6.2 节。

6.3.2　节点配置

Avatar FaceBook 版本使用 6.2 节中相同的节点，具体步骤如下。

1. 启动虚拟机

启动的节点包括：namenode0、namenode1、datanode00～datanode03、NFS。

2. 使用 secureCRT 登录到各个节点

3. 安装 Zookeeper

Zookeeper 在这里主要用于存储当前 NameNode(Primary)的地址，Primary 启动后用如下命令将地址存储到 Zookeeper 上：

```
   $/usr/local/hadoop-0.20.1-dev-facebookbin/hadoop
org.apache.hadoop.hdfs.AvatarShell -zero –updateZK
```

当 Standby 切换为 Primary 时，Zookeeper 上的 Primary 的地址被更新（手工切换之前，Zookeeper 上的 NameNode 的地址不会更新）。客户端访问 HDFS 时，首先会到 Zookeeper 上取得 Primary 的地址，之后再去访问。

由于 hbase 中包含的 Zookeeper 软件使用更方便，因此这里使用 hbase-0.90.3.tar.gz 自带的 Zookeeper 软件。将 hbase-0.90.3.tar.gz 上传到 datanode03 的/usr/local 目录下，并解压缩。

Zookeeper 只在 datanode01～datanode03 上部署，由于 Zookeeper 的选举算法需要奇数个节点，因此 Zookeeper 集群至少需要 3 个节点。

4. 修改 hbase-site.xml

在 datanode01 上修改：

```
   $vi /usr/local/hbase-0.90.3/con/hbase-site.xml
```

内容如下：

```
   <?xml version="1.0"?>
   <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
   <configuration>
   <property>
```

```
<value>/usr/local/hbase-0.90.3/zookeeper</value>
</property>
<property>
<name>hbase.zookeeper.property.clientPort</name>
<value>2181</value>
</property>
<property>
<name>hbase.zookeeper.quorum</name>
<value>192.168.1.14,192.168.1.15,192.168.1.16</value>
</property>
</configuration>
```

修改完毕后，将 hbase-site.xml 依次复制到 192.168.1.15 和 192.168.1.16。

5. 启动 Zookeeper

依次在 datanode01～datanode03 上启动 Zookeeper。

```
$cd /usr/local/hbase-0.90.3
$bin/hbase zookeeper
```

启动完毕后，在 datanode01 上查看 Zookeeper 的状态，如图 6.18 所示。

```
$cd /usr/local/hbase-0.90.3
$bin/hbase zkcli
```

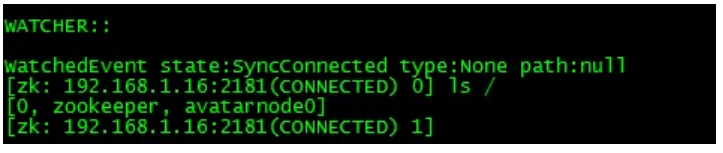

图 6.18　Zookeeper 状态图

6. 配置所有节点的 core-site.xml 和 hdfs-site.xml

（1）在 namenode0(Primary) 和 namenode1(Standby) 上修改 6.2.5 的

core-site.xml 内容如下：

```
<!--下面这个 namenode0 为 192.168.1.11 的机器名-->
  <property>
       <name>fs.default.name0</name>
       <value>hdfs://namenode0:9000</value>
  </property>
<!--下面这个 namenode1 为 192.168.1.12 的机器名-->
  <property>
       <name>fs.default.name1</name>
       <value>hdfs://namenode1:9000</value>
  </property>
<!--下面这个 0.0.0.0 为自定义的地址，一般为 0.0.0.0，在 zk 中 primary 的地址存在
/0/0/0/0/9000 节点下 -->
<property>
       <name>fs.default.name</name>
       <value>hdfs://0.0.0.0:9000</value>
</property>
```

增加以下内容：

```
<!--下面这个地址是可以访问的 zk 集群的地址 -->
<property>
       <name>fs.ha.zookeeper.quorum</name>
       <value>192.168.1.14:2181</value>
</property>
```

（2）在 namenode0(Primary)和 namenode1(Standby)上修改 6.2.5 节的 hdfs-site.xml。

```
<property>
  <name>dfs.permissions</name>
```

```
</property>
<property>
       <name>dfs.http.address0</name>
       <value>namenode0:50070</value>
</property>
<property>
       <name>dfs.http.address1</name>
       <value>namenode1:50070</value>
</property>
<property>
       <name>dfs.http.address</name>
       <value>0.0.0.0:50070</value>
</property>
```

（3）在 datanode01～datanode03 上修改 6.2.7 的 core-site.xml。

```
<property>
       <name>fs.default.name0</name>
       <value>hdfs://namenode0:9000</value>
</property>
<property>
       <name>fs.default.name1</name>
       <value>hdfs://anamenode1:9000</value>
</property>
<property>

       <value>hdfs://0.0.0.0:9000</value>
</property>
```

增加以下内容：

```
<property>
<name>fs.hdfs.impl</name>
<value>org.apache.hadoop.hdfs.DistributedAvatarFileSystem</value>
<description>The FileSystem for hdfs: uris.</description>
</property>
<property>
<name>fs.ha.zookeeper.quorum</name>
<value>192.168.1.14:2181</value>
</property>
<property>
<name>fs.ha.zookeeper.watch</name>
<value>true</value>
</property>
```

（4）在 datanode01～datanode03 上修改 hdfs-site.xml，内容与 namenode0 相同。

6.3.3 启动 HDFS

1. 挂载 NFS

挂载前，检查 NFS 服务器的 NFS 服务是否已启动，登录 namenode0 和 namenode1 执行以下命令：

```
$echo "123456"| sudo mount -v -t nfs -o tcp,soft,rsize=32768,wsize=32768
```

命令执行后，使用以下命令检查：

```
$mount
```

或：

```
$df -h
```

2. 格式化

在 namenode0 执行：

```
   $cd /usr/local/hadoop-0.20.1-dev-facebook
   $bin/hadoop org.apache.hadoop.hdfs.server.namenode.AvatarNode -zero
-format
```

3. 清空 Data Node 上的 Block

依次登录 datanode00～datanode03，执行：

```
   $rm -rf /usr/local/hadoop/block/*
```

4. 启动 Primary

在 namenode0(Primary)上执行：

```
   $cd /usr/local/hadoop-0.20.1-dev-facebook
   $bin/hadoop org.apache.hadoop.hdfs.server.namenode.AvatarNode -zero
```

5. 启动 Standby

在 namenode1(Standby)上执行：

```
   $cd /usr/local/hadoop-0.20.1-dev-facebook
   $bin/hadoop org.apache.hadoop.hdfs.server.namenode.AvatarNode -one
-standby -sync
```

6. 启动 Data Node

依次登录 datanode00～datanode03，执行：

```
   $cd /usr/local/hadoop-0.20.1-dev-facebook
```

7. 启动 Zookeeper

注：Zookeeper 在 6.3.2 第 5 步中已启动，在此无需再操作。

8. 向 Zookeeper 注册 Primary 地址

在 namenode0 上执行：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook
$bin/hadoop org.apache.hadoop.hdfs.AvatarShell -zero -updateZK
```

9. 在 datanode01 上查看 Zookeeper 状态

进入 Zookeeper 命令行：

```
bin/hbase zkcli
```

结果如图 6.19 所示。

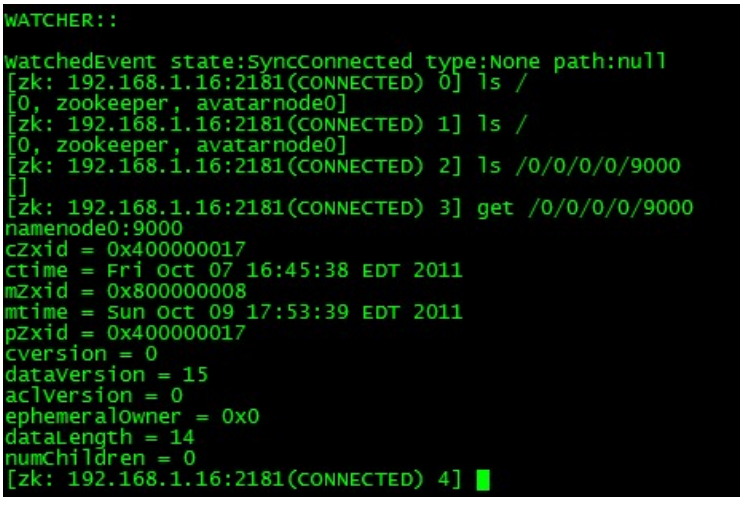

图 6.19　Zookeeper 状态图

6.3.4　NameNode 失效切换

1. 关闭 namenode0 的 NameNode，模拟失效

登录 namenode0，按下 Ctrl + C 快捷键中止当前运行的 NameNode 进程。

在 datanode00 使用客户端访问 HDFS。

```
$bin/hadoop dfs -lsr /
```

此时 HDFS 无法访问。

2. 在 namenode1 上手工切换 Standby 到 Primary

```
$bin/hadoop org.apache.hadoop.hdfs.AvatarShell -one -setAvatar primary
```

namenode1 切换为 Primary 的过程有以下几个步骤。

（1）管理员手工执行命令将 namenode1 切换为 Primary。

（2）namenode1 上 Ingest 线程最后一次从 NFS 中的 namenode0 对应日志的目录中读取 namenode0 的日志。

（3）namenode1 上结束 Ingest 线程、结束 Standby 线程。

（4）将 Zookeeper 上/0/0/0/0/9000 路径下的 namenode0 的 IP 地址 192.168.1.11:9000 更新为 namenode1 的 IP 地址 192.168.1.12:9000。

在 datanode00 上访问 HDFS。

```
$bin/hadoop dfs -lsr /
```

此时，可以访问 HDFS，说明 Starndby 已切换为 Primary 开始工作了。

3. 查看 Zoopkeeper 的状态

在 datanode01 上进行查看：

```
[zk: 192.168.1.16:2181(CONNECTED) 4]get /0/0/0/0/9000
```

显示为 namenode1 即 192.168.1.16 的地址。

4. 重启 namenode0 作为 Standby

登录 namenode0：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook
$bin/hadoop org.apache.hadoop.hdfs.server.namenode.AvatarNode -one
```

至此，在 HDFS 集群中，Primary 和 Standby 又同时正常工作，如果 Primary 无法正常服务，则重复上面的步骤。

第 7 章　AvatarNode 异常解决方案

本章以 AvatarNode 机制为基础，对各种异常情况下的处理机制进行验证。

7.1 测试环境

测试环境采用 6.3 节搭建的环境，即 Avatar FaceBook 版本，在实验过程中，主要模拟以下几种常见的异常现象，并且模拟相应的读写操作，最后检查结果以验证各种解决方案。

单节点失效

- Primary 节点失效
- Standby 节点失效
- NFS 节点失效

双节点失效

- Primary 节点先失效
- NFS 节点后失效
- NFS 节点先失效
- Primary 节点后失效等

7.2 Primary 失效

7.2.1 解决方案

Primary 节点失效解决方案流程如图 7.1 所示，我们将在 t1、t2、t3 阶段分别模拟读写操作，进行验证。

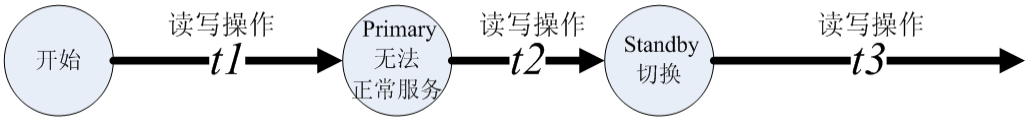

图 7.1 Primary 失效解决方案流程图

7.2.2　写操作实验步骤

> 具体操作视频请参考：\视频\7 AvatarNode 异常解决方案\ 1 Primary 失效，写操作.exe
>
> 脚本内容参见：\脚本\7 AvatarNode 异常解决方案

步骤描述如下。

1. 启动虚拟机集群

启动的节点包括：

- namenode0(Primary)
- namenode1(Standby)
- NFS（NFS 服务器）
- datanode00（包括：Data Node 节点 1，Client 节点 1）
- datanode01～datanode03（包括：Data Node 节点 2～4、Zookeeper 集群节点 1～3、Client 节点 2～4）。

2. 用 SecureCRT 连接到虚拟机

3. 启动 Zookeeper

在 datanode01（192.168.1.14）、datanode02（192.168.1.15）、datanode03（192.168.1.16）三个节点上启动 Zookeeper。依次在 datanode01～datanode03 的终端执行：

```
$cd /usr/local/hbase-0.90.3
```

在 datanode01（192.168.1.14）上查看 Zookeeper 启动情况。

```
$cd /usr/local/hbase-0.90.3
$bin/hbase zkcli
```

4. 挂载 NFS 目录

（1）挂载之前先检查 NFS（NFS 服务器）上的 NFS 服务是否已启动。

（2）依次登录 namenode0(Primary)、namenode1(Standby)执行：

```
$echo "123456" | sudo -S mount 192.168.1.10:/usr/local/hadoop/avatarshare
/usr/local/hadoop/avatarshare
```

> 注意：“123456”是以 sudo 执行命令时输入的 root 用户密码。

5. HDFS 格式化

（1）在 namenode0(Primary)上执行：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook/
$bin/hadoop org.apache.hadoop.hdfs.server.namenode.AvatarNode -zero
-format
```

（2）清空所有 Data Node 数据。

按照配置，Data Node 会将真实文件的数据以 Block（通常 64MB）的形式存储在本机的/usr/local/hadoop/block 目录下。依次在 datanode00～datanode03 上执行：

```
$rm -rf /usr/local/hadoop/block/*
```

6. 启动 Avatar

（1）启动 Primary。在 namenode0 上执行：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook/
```

（2）更新 Zookeeper 存储的 NameNode 地址。在 namenode0 上执行：

```
$bin/hadoop org.apache.hadoop.hdfs.AvatarShell -zero -updateZK
```

在 datanode01 上执行：

```
$cd /usr/local/hbase-0.90.3/
$bin/hbase zkcli
[zk: 192.168.1.16:2181(CONNECTED) 0] ls /
[zk: 192.168.1.16:2181(CONNECTED) 1] get /0/0/0/0/9000
```

（3）启动 Standby。在 namenode1(Standby)上执行：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook/
$bin/hadoop org.apache.hadoop.hdfs.server.namenode.AvatarNode -one
-standby -sync
```

（4）启动 DataNode。依次在 datanode00～datanode03 上执行：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook/
$bin/hadoop-daemon.sh start
org.apache.hadoop.hdfs.server.datanode.AvatarDataNode
```

（5）检查。在浏览器中输入：

```
192.168.1.11:50070/dfshealth.jsp
```

7. 模拟 t1 阶段写操作

（1）向 HDFS 写入文件（以脚本方式在前台写入，确保 Primary 宕机时，文件写入还没有结束）。在 datanode00（192.168.1.13）上复制 bin/write_to_hdfs.sh 脚本，将其修改为前台串行写入文件，新的脚本名为：write_to_hdfs_2.sh。运行脚本，向 HDFS 写入 10 个 10MB 文件，测试写入脚本：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook/
```

在 Primary(namenode0)上查看文件写入情况：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook/
$bin/hadoop dfs -lsr /
```

可以看到在 HDFS 的/tmp/192.168.1.13 目录下会产生新的文件，序号为：10000～10009。

清除写入的文件：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook/
$bin/hadoop dfs -rm -skipTrash /tmp/192.168.1.13/*
```

再写入 10 个 1GB 的文件，在 datanode00（192.168.1.13）上执行：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook/
$bin/write_to_hdfs_2.sh 10 1000000000 1
```

（2）查看文件的写入情况。分别连接到 Primary 和 Standby，查看两个节点的元数据情况。连接 Primary(namenode0)，查询文件写入情况，在 Primary(namenode0)上执行：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook/
```

可以看到在 HDFS 的/tmp/192.168.1.13 目录下，已经写入成功 1 个文件 10000，而文件 10001 正在写入。

连接 Standby(namenode1)，查询文件写入情况，在 Standby(namenode1)执行。

```
$cd /usr/local/hadoop-0.20.1-dev-facebook/
$bin/hadoop dfs -dfs.default.name=hdfs://192.168.1.12:9000 -lsr /
```

可以看到，Standby 上的元数据信息滞后于 Primary(namenode0)上的元数据信息，一段时间后，可以和 Primary 保持一致。

8. 模拟 Primary 失效

（1）在 Primary(namenode0)上执行

```
$sudo shutdown -h now
```

（2）检查 t1 阶段的写入情况。Primary 失效后，可以看到 t1 阶段的写入操作将无法连接 Primary，客户端重试 10 次后将报错。

（3）模拟 t2 阶段写操作。首先从 datanode00 上复制 write_to_hdfs_2.sh 脚本文件到 datanode01 的/usr/local/hadoop-0.20.1-dev-facebook/bin 目录下。在 datanode00（192.168.1.13）上执行：

```
   $cd /usr/local/hadoop-0.20.1-dev-facebook/
   $scp bin/write_to_hdfs_2.sh
192.168.1.14:/usr/local/hadoop-0.20.1-dev-facebook/bin/
```

在 datanode01（192.168.1.14）上向 HDFS 写入 10 个 1GB 的文件：

```
   $cd /usr/local/hadoop-0.20.1-dev-facebook/
   $bin/write_to_hdfs_2.sh 10 1000000000 10
```

写入结果显示，t2 阶段的写操作也因为连接不上 Primary 而报错。

9. 将 Standby(namenode1)切换为 Primary

（1）保存元数据。在 Standby(namenode1)上执行：

```
   $cd /usr/local/hadoop-0.20.1-dev-facebook
   $bin/hadoop dfsadmin -saveNamespace
```

（2）切换。在 Standby(namenode1)上执行：

```
   $bin/hadoop org.apache.hadoop.hdfs.AvatarShell -one -setAvatar primary
```

（3）查看 Zoookeeper 上的地址更新信息。在 datanode01 上执行：

```
   $cd /usr/local/hbase-0.90.3/
   $bin/hbase zkcli

   [zk: 192.168.1.16:2181(CONNECTED) 1] get /0/0/0/0/9000
```

（4）看是否已经切换到 namenode1（192.168.1.12）。连接 Primary

(namenode1)，看能否访问。在 Primary(namenode1)上执行：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook
$bin/hadoop dfs -lsr /
```

可以看到，HDFS 可以访问，且只有 t1 阶段写入的若干文件，未见 t2 阶段写入的文件。

10. 模拟 t3 阶段写操作

向 HDFS 写入 3 个 100MB 文件（脚本方式在前台写入），在 datanode00 上执行。文件将写入 HDFS 的/tmp/192.168.1.13 目录，写入之前，先清空目录。

```
$cd /usr/local/hadoop-0.20.1-dev-facebook
$bin/hadoop dfs -rmr -skpTrash /tmp/192.168.1.13
```

11. 检查

查看 HDFS 写入结果，在 datanode00 上执行：

```
$cd /usr/local/hadoop-0.20.1-dev-facebook
$bin/hadoop dfs -lsr /
```

12. 实验结果（见表 7.1 所示）

表 7.1　Primary 失效写操作测试结果表

	T1	T2	T3
写入序号	10000~10009	10000~10009	10000~10002
成功序号	10000~10001	无	10000~10002
失败序号	10002~10009	10000~10009	无

以上数据说明：在 Primary 失效前，已经结束的文件写入操作不受影响；Primary 失效后，继续进行的文件写入操作将失败；切换成功后，新的文件写入操作将成功。

7.2.3　改进写操作机制

由上面的测试可知，在 Primary 发生宕机、Standby 切换到 Primary 的过程中，元数据修改、文件的写入等操作将会失败。因此，需要一定的机制来保证上述操作的最终成功，而不受切换操作的影响。考虑到上述操作均由客户端调用 HDFS API发起以及修改 AvatarNode 自身代码所带来的可靠性、不可维护性等问题，因此我们通过上层应用程序机制来保证上述操作的可靠性。以文件复制为例，具体机制描述如下。

（1）在应用程序中，产生一个专门的线程进行文件复制操作，因为如果按照通常的调用方式在主线程中直接调用文件复制操作的话，当发生切换时，文件复制操作会抛出异常，直接导致整个程序崩溃；而在单独的线程中调用文件复制操作，当发生切换导致写入操作失败时，只会导致该线程退出。

（2）由于切换时线程会直接退出，无法通过 HDFS 文件复制 API 返回值判断操作是否成功，因此需要其他方法来判断此时是否发生切换。我们采用的方法是，在产生一个文件复制任务的同时，产生一个任务监控线程，定期调用 HFS 的 FileSystem.listStatus()方法，检查写入文件的长度是否在增长，若没有达到源文件长度并在 15 秒内一直不增长，或者长度减少，则认为此次文件复制任务失败，启动一个新的文件复制任务。如果写入文件长度达到源文件长度，则认为复制成功并退出。

7.2.4　读操作实验步骤

> 具体操作视频请参考：\视频\7 AvatarNode 异常解决方案\ 2 Primary 失效，读操作.exe

步骤描述如下。

1. 启动 HDFS 和 Zookeeper

实验环境和 7.2.2 节完全一样，具体参见 7.2.2 节中的步骤 1～6。

2. 准备要读取的文件

从 datanode00（192.168.1.13）和 datanode01（192.168.1.14）上复制 6 个 500MB 大小的文件到 HDFS 的/tmp/192.168.1.13 和/tmp/192.168.1.14 目录下，文件编号为 10000～10005。在 datanode00 和 datanode01 上分别执行：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/wite_to_hdfs_2.sh 6
500000000 100
```

此脚本与 write_to_hdfs.sh 唯一的不同是将 copy 的执行语句由后台变成了前台，在执行前确保该脚本已上传。

3. 模拟 t1 阶段读操作

从 HDFS 上复制一批文件到 datanode00 和 datanode01，下面的脚本会将 HDFS 的/tmp/192.168.1.1x 目录下的内容复制到本地的/tmp/192.168.1.1x 目录下。

在 datanode00 上执行脚本：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_W.sh
192.168.1.13
```

在 datanode01 上执行脚本：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_W.sh
192.168.1.14
```

此脚本与 read_from_hdfs_IP.sh 唯一不同是将 get 的执行语句由后台变成了前台，执行前确保该脚本在对应的目录下存在。

4. 模拟 Primary(namenode0)失效

（1）重启 Primary(namenode0)：

```
   $sudo reboot
```

（2）检查第 t1 阶段读操作结果。

可以看到，在 Primay 失效前就开始的读取操作可以继续进行。在 Primary 失

效后开始的读取操作由于连接不上 Primary，会报错，重试 10 次后退出。

5. 模拟 t2 阶段读操作

在 datanode00 上执行脚本，复制 HDFS 的/tmp/192.168.1.13 目录下的文件到本地的/tmp/192.168.1.13 目录。

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_2.sh
192.168.1.13
```

在 datanode01 上执行脚本，复制 HDFS 的/tmp/192.168.1.14 目录下的文件到本地的/tmp/192.168.1.14 目录。

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_2.sh
192.168.1.14
```

可以看到，t2 阶段的读取操作会尝试连接 Primary，失败 10 次后退出。

6. Standby 切换

（1）在 Standby(namenode1)上导出最新的元数据，供后续的 Standby 使用：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/hadoop dfsadmin
```

（2）切换 Standby(namenode1)为 Primary。在 Standby(namenode1)上执行：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/hadoop
org.apache.hadoop.hdfs.AvatarShell -one -setAvatar primary
```

（3）查看切换结果。在 datanode01（192.168.1.14）上查看 Standby 切换是否成功：

```
   $cd /usr/local/hbase-0.90.3/
   $bin/hbase zkcli
   [zk: 192.168.1.16:2181(CONNECTED) 0] ls /
```

```
[zk: 192.168.1.16:2181(CONNECTED) 1] get /0/0/0/0/9000
```

7. 模拟 t3 阶段读操作

（1）从 HDFS 上复制一批新的文件到本地（前台脚本）。具体操作参见步骤 5。

（2）检查复制结果。在 datanode00 上执行：

```
$ls /tmp/192.168.1.13/ -la
```

在 datanode01 上执行：

```
$ls /tmp/192.168.1.14/ -la
```

可以看到 t3 阶段的读取操作可以顺利进行。

8. 检查

检查 t1～t3 阶段的读取情况。

- 客户端 1（192.168.1.13）实验结果（见表 7.2 所示）。

表 7.2 Primary 失效读操作测试结果表

	T1	T2	T3
读取序号	10000~10005	10000~10005	10000~10005
失败序号	10003~10005	10000~10005	无

- 客户端 2（192.168.1.14）实验结果（见表 7.3 所示）。

表 7.3 Primary 失效读操作测试结果表

	T1	T2	T3
读取序号	10000~10005	10000~10005	10000~10005
失败序号	10001~10005	10000~10005	无

结果显示，在 t1 阶段发起的读取操作，不受 Primary 失效的影响，可以顺利完成，如 datanode00 上的 10000～10002 文件，都是在 t1 阶段发起操作的，其中 10002 在 Primary 失效时还没有读取结束，后续的文件 10003～10005，则是在

Primary 失效后发起的；在 t2 阶段发起的读取操作全部失败；t3 阶段的读取操作全部成功，说明切换成功后，读取操作恢复正常。

7.2.5　小结

当 Primary 无法正常服务时，采用以上解决方案，其恢复时间和可靠性分析如下。

1．恢复时间

由上述分析和实验可知，HDFS 在 Primary 失效到切换成功这一段间隔内无法正常服务。因此，恢复时间为：

恢复时间 = Primary 失效到切换成功的间隔时间

2．读操作影响

在 t1 阶段发起的读操作不受影响；t2 阶段发起的读操作将由于连接不上 Primary，进行重试；t3 阶段的读操作不受影响。

3．写操作影响

在 t1 阶段发起的写操作，如果在 Primary 失效前完成，则不受影响；在 t2 阶段发起的写操作将由于连接不上 Primary，进行重试；t3 阶段的写操作不受影响。

7.3　Standby 失效

由于 Standby 是一个热备节点，并不对外提供服务，因此当其无法工作时，并不影响整个 HDFS 的服务。具体解决方案也很简单，选择一个新的节点，进行相应的配置，以 Standby 启动即可。

7.4　NFS 失效（数据未损坏）

7.4.1　解决方案

NFS 节点失效（数据未损坏）解决方案流程如图 7.2 所示，我们将在 t1、t2、

t3 阶段分别模拟读写操作，进行验证。

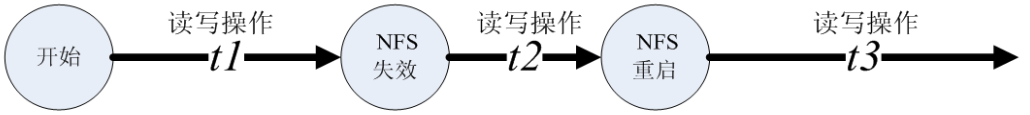

图 7.2　NFS 失效解决方案流程图

7.4.2　写操作实验步骤

在读写操作场景下，重启 NFS 服务器，模拟宕机以及服务恢复现象。

> 具体操作视频请参考：7 AvatarNode 异常解决方案\ 3 NFS 失效（数据未损坏），写操作.exe

步骤描述如下。

1. 启动 HDFS 和 Zookeeper

实验环境和 7.2.2 节完全一样，具体参见 7.2.2 节中的步骤 1～6。

2. 模拟 t1 阶段写操作

（1）向 HDFS 写入文件（前台脚本，确保后面 NFS 失效时，写操作还没有完成）。分别在 datanode00 和 datanode01 上执行：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/write_to_hdfs_2.sh 40
20000000 10
```

> 注意：以上脚本将从 datanode0x 上分别复制 40 个 20MB 大小的文件到 HDFS 的 /tmp/192.168.1.1x 目录。

（2）查看写入情况。在 Primary(namenode0)上执行：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/hadoop dfs -lsr
```

3. 模拟 NFS 失效

（1）关闭 NFS 服务器。

（2）检查 t1 阶段写入情况。按照分析，所有的写入操作应该阻塞，对照实验结果检查。

（3）模拟 t2 阶段的写操作。在 HDFS 上创建目录，在 datanode00 执行：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/hadoop dfs -mkdir
192.168.1.15
```

在 datanode01 上执行：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/hadoop dfs -mkdir
192.168.1.16
```

可以看到以上操作均阻塞。

4. 启动 NFS

NFS 启动后，可以看到之前阻塞的操作继续进行，t1 和 t2 阶段的操作全部成功。

5. 模拟 t3 阶段写操作

（1）清除 HDFS 上原有文件。

在 datanode00 上执行：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/hadoop dfs -rmr -skipTrash /tmp
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/hadoop dfs -rmr -skipTrash /usr
```

（2）向 HDFS 写入新文件。

在 datanode00 上执行：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/write_to_hdfs_2.sh 40
```

（3）检查写入结果：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/dfs -lsr /
```

可以看到 t3 阶段的操作顺利进行。

6. 检查

检查 t1～t3 阶段的写入情况。按照分析，所有写入操作应该继续进行。

7. 实验结果（见表 7.4 所示）

表 7.4 NFS 失效写操作测试结果表

	T1	T2	T3
写入序号	10000~100039	创建 1 个目录	10000~100039
失败序号	无	无	无

7.4.3 读操作实验步骤

> 具体操作视频请参考：\视频\7 AvatarNode 异常解决方案\ 3 NFS 失效（数据未损坏），读操作.exe

1. 接上面的步骤继续

2. 产生读取文件

要读取的文件在 7.5.2 的 t3 阶段已产生，为/tmp/192.168.1.13 和/tmp/192.168.1.14 目录下，序号为 10000～10039 的文件。

3. 模拟 t1 阶段读操作

从 HDFS 复制文件到本地。在 datanode00 上执行：

```
   $rm -rf /tmp/192.168.1.13/*
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_2.sh
192.168.1.13
   $ls -l /tmp/192.168.1.13
```

在 datanode01 上执行：

```
   $rm -rf /tmp/192.168.1.14/*


192.168.1.14
```

```
$ls -l /tmp/192.168.1.14
```

4. 模拟 NFS 失效

（1）关闭 NFS 节点。在 NFS（192.168.1.10）上执行：

```
$sudo shutdown -h now
```

（2）检查 t1 阶段读操作情况。在 datanode00 上执行：

```
$ls -l /tmp/192.168.1.13
```

在 datanode01 上执行：

```
$ls -l /tmp/192.168.1.14
```

可以看到 t1 阶段的读取操作继续进行。

5. 模拟 t2 阶段读操作

在 datanode00 上执行：

```
$rm -rf /tmp/192.168.1.13/*
$/usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_2.sh
192.168.1.13
$ls -l /tmp/192.168.1.13
```

在 datanode01 上执行：

```
$rm -rf /tmp/192.168.1.14/*

192.168.1.14
$ls -l /tmp/192.168.1.14
```

可以看到 t2 阶段的读取操作可以顺利进行，不受 NFS 失效的影响。

6. 重启 NFS

7. 模拟 t3 阶段读操作

（1）检查。在 datanode00 和 datanode01 上检查 t2 阶段的读取情况，可以看到 t2 阶段的读取操作顺利进行，等待结束。

（2）模拟读操作。在 datanode00 上执行：

```
   $rm -rf /tmp/192.168.1.13/*
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_2.sh
192.168.1.13
   $ls -l /tmp/192.168.1.13
```

在 datanode01 上执行：

```
   $rm -rf /tmp/192.168.1.14/*


192.168.1.14
   $ls -l /tmp/192.168.1.14
```

8. 检查

可以看到 t3 阶段的读取操作可以顺利进行。

9. 实验结果（见表 7.5 所示）

表 7.5　NFS 失效读操作测试结果表

	T1	T2	T3
读取序号	10000~10039	10000~10039	10000~10039
失败序号	无	无	无

7.4.4　小结

当 NFS 无法正常服务时，采用以上解决方案，其恢复时间和可靠性分析如下。

1．恢复时间

读取操作不受 NFS 失效的影响，因此恢复时间为零。

写入操作在 NFS 失效到恢复服务这段时间内阻塞，恢复时间为：

NFS 的重启时间

2．读操作影响

读操作不受 NFS 失效影响。

3．写操作影响

写操作在 NFS 失效期间阻塞，待 NFS 恢复服务后，解除阻塞，继续进行。

7.5　NFS 失效（数据已损坏）

7.5.1　解决方案

NFS 节点失效（数据损坏）解决方案流程如图 7.3 所示，我们将在 t1、t2、t3 阶段分别模拟读写操作，进行验证。

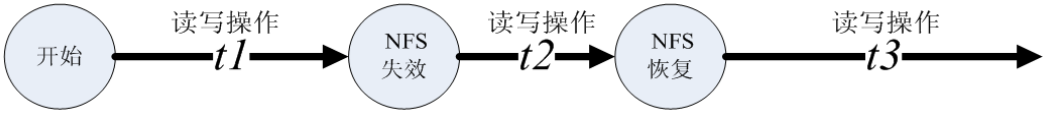

图 7.3　NFS 失效解决方案流程图

其中 NFS 恢复的步骤如下。

（1）准备一个 NFS 节点。

（2）将 Primary(namenode0)和 Standby(namenode1)的元数据复制到 NFS 节点。

（3）重启 NFS。

7.5.2 写操作实验步骤

具体操作视频请参考：\视频\7 AvatarNode异常解决方案\ 3 NFS 失效（数据损坏）.exe

1. 启动 HDFS 和 Zookeeper

（1）实验环境和 7.2.2 节完全一样，具体参见 7.2.2 节中的步骤 1～6。

（2）创建 3 个脚本。创建以下 3 个脚本：

- /usr/local/hadoop-0.20.1-dev-facebook/bin/write_to_hdfs_t1.sh
- /usr/local/hadoop-0.20.1-dev-facebook/bin/write_to_hdfs_t2.sh
- /usr/local/hadoop-0.20.1-dev-facebook/bin/write_to_hdfs_t3.sh

每个脚本会将文件写入 HDFS 不同的目录，分别对应：

- /tmp/192.168.1.x/t1
- /tmp/192.168.1.x/t2
- /tmp/192.168.1.x/t3

具体脚本内容参见：\脚本\7 AvatarNode异常解决方案\

2. 模拟 t1 阶段写操作

（1）从 datanode00（192.168.1.13）和 datanode01（192.168.1.14）向 HDFS 写入 10 个 200MB 的文件。在 datanode00 和 datanode01 上运行脚本：

```
$/usr/local/hadoop-0.20.1-dev-facebook/bin/write_to_hdfs_t1.sh  40
```

该脚本会向 HDFS 的/tmp/192.168.1.x/t1 目录下写入 10 个 200MB 的文件。

（2）查看写入情况。在 datanode02 上运行：

```
$/usr/local/hadoop-0.20.1-dev/bin/hadoop dfs -lsr /
```

3. 模拟 NFS 失效

（1）修改 NFS 的 IP 地址。在 NFS 上执行：

```
$sudo vi /etc/sysconfig/network-scripts/ifcfg-eth0
```

将原来的 IP 地址由 192.168.1.10 修改为 192.168.1.8。

重启网卡服务，使修改后的 IP 地址生效：

```
$sudo service network restart
```

（2）清除共享目录。在 NFS 上，清除共享目录 share0 和 share1 下面的内容，模拟 NFS 上的元数据丢失，无法恢复。

```
$cd /usr/local/hadoop/avatarshare/share0
$rm -rf *
$cd /usr/local/hadoop/avatarshare/share1
$rm -rf *
```

4. 模拟 t2 阶段写操作

在 datanode00 和 datanode01 上运行脚本。

```
$/usr/local/hadoop-0.20.1-dev-facebook/bin/write_to_hdfs_t2.sh  40
20000000 10
```

该脚本会向 HDFS 的/tmp/192.168.1.x/t2 目录下写入 10 个 200MB 的文件。

查看文件写入情况，发现 NFS 失效后，t1、t2 阶段的所有写入操作阻塞。

5. 恢复 NFS 元数据

分别将 Primary 和 Standby 的元数据文件复制到 NFS 的共享目录中。

（1）在 Primary(namenode0)上执行：

```
$cd /usr/local
```

```
$rm -rf hadoop/local/*:*
$ls hadoop/local/
$scp -r /usr/local/hadoop/local/* 192.168.1.8:
                /usr/local/hadoop/avatarshare/share0
```

（2）在 Standby(namenode1)上执行：

```
$cd /usr/local
$ls hadoop/local/
$rm -rf hadoop/local/*:*
$ls hadoop/local/
$scp -r /usr/local/hadoop/local/* 192.168.1.8:
                  /usr/local/hadoop/avatarshare/share1
```

6. NFS 恢复服务

将 NFS 的 IP 地址修改为原来的 IP 地址，模拟其恢复服务。

在 NFS 上执行：

```
$sudo vi /etc/sysconfig/network-scripts/ifcfg-eth0
```

将原来的 IP 地址由 192.168.1.8 修改为 192.168.1.10。

重启网卡服务，使修改后的 IP 地址生效。

```
$sudo service network restart
```

7. 模拟 t3 阶段写操作

在 datanode00 和 datanode01 上运行脚本。

```
$/usr/local/hadoop-0.20.1-dev-facebook/bin/write_to_hdfs_t3.sh  40
20000000 10
```

该脚本会向 HDFS 的/tmp/192.168.1.x/t3 目录下写入 10 个 200MB 的文件。

8. 检查

查看文件写入情况，发现 NFS 恢复后，t1、t2、t3 阶段的所有写入操作继续进行。

- 实验结果（见表 7.7 所示）。

表 7.6　NFS 失效写操作测试结果表

	T1	T2	T3
写入序号	10000~10019	10000~10019	10000~10019
失败序号	无	无	无

7.5.3　读操作实验步骤

1. 接上面的步骤继续向下执行

2. 产生读取文件

（1）创建 3 个脚本。在 datanode00 的/usr/local/hadoop-0.20.1-dev-facebook/bin/目录下创建以下 3 个脚本：

- /usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_t1.sh
- /usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_t2.sh
- /usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_t3.sh

以上 3 个脚本分别模拟 t1、t2、t3 阶段的读操作，从 HDFS 的/tmp/192.168.1.1x/上读取文件到本地目录，分别对应：

- /tmp/192.168.1.x/t1
- /tmp/192.168.1.x/t2
- /tmp/192.168.1.x/t3

将以上脚本复制到 datanode01 的/usr/local/hadoop-0.20.1-dev-facebook/bin/目录下。

（2）产生读取所需的文件

在 datanode00（192.168.1.13）和 datanode01（192.168.1.14）上执行：

```
$/usr/local/hadoop-0.20.1-dev-facebook/bin/write_to_hdfs_t2.sh  40
```

上述脚本将在 HDFS 的 192.168.1.1x/t2 目录下产生 10 个 200MB 的文件。

（3）创建目录。分别在 datanode00（192.168.1.13）和 datanode01（192.168.1.14）上创建下列目录：

- /tmp/192.168.1.x/t1
- /tmp/192.168.1.x/t2
- /tmp/192.168.1.x/t3

3. 模拟 t1 阶段读操作

（1）从 HDFS 复制文件到本地。

在 datanode00（192.168.1.13）上执行：

```
$/usr/local/hadoop-0.20.1-dev-facebook/bin/ read_from_hdfs_IP_t1
192.168.1.13
```

在 datanode01（192.168.1.14）上执行：

```
$/usr/local/hadoop-0.20.1-dev-facebook/bin/ read_from_hdfs_IP_t1
192.168.1.14
```

注意：上述脚本将从HDFS读取文件复制到本地的/tmp/192.168.1.1x/t1目录下。

（2）检查复制情况。

在 datanode00（192.168.1.13）上执行：

```
$ls -l /tmp/192.168.1.13/t1
```

在 datanode01（192.168.1.14）上执行：

```
$ls -l /tmp/192.168.1.14/t1
```

可以发现，T1 阶段的读取操作可以顺利进行。

4. 模拟 NFS 失效

（1）步骤同 7.5.2 节中步骤 3。

（2）检查 t1 阶段读取情况。

5. 模拟 t2 阶段读操作

（1）从 HDFS 复制文件到本地。

在 datanode00（192.168.1.13）上执行：

```
$/usr/local/hadoop-0.20.1-dev-facebook/bin/ read_from_hdfs_IP_t2
192.168.1.13
```

在 datanode01（192.168.1.14）上执行：

```
$/usr/local/hadoop-0.20.1-dev-facebook/bin/ read_from_hdfs_IP_t2
192.168.1.14
```

> 注意：上述脚本将从 HDFS 读取文件复制到本地的/tmp/192.168.1.1x/t2 目录下。

（2）检查复制情况。检查 t1、t2 阶段的读取操作可以顺利进行，不受 NFS 失效影响。

6. NFS 恢复

步骤同 7.5.2 节中步骤 5、6。

7. 模拟 t3 阶段读操作

在 datanode00（192.168.1.13）上执行：

```
$/usr/local/hadoop-0.20.1-dev-facebook/bin/ read_from_hdfs_IP_t3
192.168.1.13
```

在 datanode01（192.168.1.14）上执行：

```
   $/usr/local/hadoop-0.20.1-dev-facebook/bin/ read_from_hdfs_IP_t3
192.168.1.14
```

注意: 上述脚本将从 HDFS 读取文件复制到本地的/tmp/192.168.1.1x/t3 目录下。

8. 检查

检查 t1、t2、t3 阶段的读取情况，可以看到每个阶段的读取操作都可以顺利进行，且数据恢复和切换并未造成元数据丢失。

9．实验结果（见表 7.7 所示）

表 7.7　NFS 失效读操作测试结果表

	T1	T2	T3
读取序号	10000~10019	10000~10019	10000~10019
失败序号	无	无	无

7.5.4　小结

当 NFS 无法正常服务时，采用以上解决方案，其恢复时间和可靠性分析如下。

1．恢复时间

读取操作不受 NFS 失效的影响，因此恢复时间为零。

写入操作在 NFS 失效到恢复服务这段时间内阻塞，恢复时间为：

NFS 数据恢复时间+重启时间

2．读操作影响

读操作不受 NFS 失效影响。

3．写操作影响

写操作在 NFS 失效期间阻塞，待 NFS 恢复服务后，解除阻塞，继续进行。

7.6　Primary 先失效，NFS 后失效（数据未损坏）

7.6.1　解决方案

Primary 先失效，NFS 后失效（数据未损坏）解决方案流程如图 7.4 所示，我们将在 t1、t2、t3、t4、t5 阶段分别模拟读写操作，进行验证。

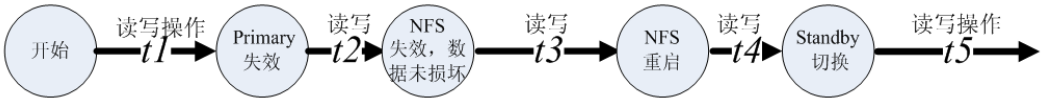

图 7.4　Primary 先失效，NFS 后失效（数据未损坏）解决方案流程图

7.6.2　写操作实验步骤

> 具体操作视频请参考：\视频\7 AvatarNode 异常解决方案\6 Primary 先失效，NFS 后失效（数据未损坏）.exe

步骤描述如下。

1. 启动 HDFS 和 Zookeeper

实验环境和 7.2.2 节完全一样，具体参见 7.2.2 节中的步骤 1～6。

创建两个脚本，分别模拟 t4、t5 两个阶段的操作。脚本路径：

```
/usr/local/hadoop-0.20.1-dev-facebook/bin/write_to_hdfs_t4.sh
```

以上脚本将会产生文件，写入 HDFS 的/tmp/192.168.1.x/t4、/tmp/192.168.1.x/t5 目录中。

2. 模拟 t1 阶段写操作

（1）从 datanode00（192.168.1.13）和 datanode01（192.168.1.14）向 HDFS 分别写入 10 个 100MB 文件。

（2）查看写入情况。

3. 模拟 Primary 失效

（1）关闭 Primary。

（2）查看 t1 阶段写入情况。

4. 模拟 t2 阶段写操作

5. 模拟 NFS 失效

（1）关闭 NFS 服务器。

（2）检查 t1、t2 阶段写入情况。

6. 模拟 t3 阶段写操作

7. 启动 NFS

（1）启动 NFS 服务。

（2）检查 t1、t2、t3 阶段写入情况。

8. 模拟 t4 阶段写操作并检查

9. Standby 切换

（1）切换：

```
$bin/hadoop dfsadmin -saveNamespace
```

（2）检查 t1、t2、t3、t4 阶段写入情况。

10. 模拟 t5 阶段写操作

11. 检查

检查 t1～t5 阶段的写操作情况，以客户端 192.168.1.13(datanode00)为例，统计实验结果（见表 7.8 所示）：

表 7.8　写操作测试结果表

	T1	T2	T3	T4	T5
写入序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
成功序号	10000~10001	10004~10009	10004~10009	10000~10009	10000~10009
失败序号	10002~10009	10000~10003	10000~10003	无	无

7.6.3　读操作实验步骤

1. 接上面的步骤继续

（1）接上面的步骤。

（2）创建新的脚本，用于准备读取的文件。脚本的名字和目录如下：/usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_t4.sh/usr/local/hadoop-0.20.1-dev-facebook/bin/read_from_hdfs_IP_t5.sh

2. 产生读取文件

（1）清空 HDFS 上已有文件。

（2）准备读取所需要的文件。从 datanode00（192.168.1.13）和 datanode01（192.168.1.14）向 HDFS 分别写入 10 个 100MB 文件

3. 模拟 t1 阶段读操作

（1）从 HDFS 复制文件到本地（前台脚本，确保后面 NFS 失效时，读操作还没有完成）。

（2）检查复制情况。

4. 模拟 Primary 失效

5. 模拟 t2 阶段读操作并检查

6. 模拟 NFS 失效

（1）关闭 NFS 节点。

（2）检查 t1、t2 阶段读取情况。

7. 模拟 t3 阶段读操作并检查

8. 重启 NFS

（1）重启 NFS 服务。

（2）检查 t1～t3 阶段读取情况。

9. 模拟 t4 阶段读操作并检查

10. Standby 切换

（1）切换。

（2）检查 t1～t4 阶段读取情况。

11. 模拟 t5 阶段读操作

12. 检查

（1）检查 t1、t2、t3、t4、t5 中读取情况。

（2）按分析，读操作应该继续进行，对照实验结果进行检查。

13. 实验结果

客户端 1（192.168.1.13）读取情况如表 7.9 所示。

表 7.9　读操作测试结果表

	T1	T2	T3	T4	T5
读取序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
失败序号	10003~10009	10000~10009	10000~10009	10000~10009	无

7.6.4　小结

采用以上解决方案，其恢复时间和可靠性分析如下。

1. 恢复时间

由测试可知，该机制下，HDFS 的恢复时间为：

NFS 重启时间 + Standby 手工切换时间

2．读、写操作影响

对于写操作来说，客户端和 Standby 都需要连接 Primary 进行通信，因此，在 Primary 失效后，手工切换前，读、写操作无法成功。

7.7　Primary 先失效（数据未损坏），NFS 后失效（数据损坏）

7.7.1　解决方案

本节讨论 Primary 先失效（数据未损坏），接着 NFS 后失效且数据损坏无法恢复的情况。具体解决方案如图 7.5 所示。

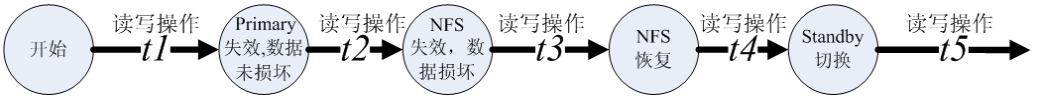

图 7.5　Primary 先失效（数据未损坏），NFS 后失效（数据损坏）解决方案流程图

其中 NFS 恢复具体步骤如下。

（1）准备一个 NFS 节点。

（2）复制 Primary、Standby 的元数据文件到 NFS。

（3）重启 NFS。

7.7.2　写操作实验步骤

具体步骤请参考视频: \视频\7 AvatarNode 异常解决方案\7 Primary 先失效（数据未损坏），NFS 后失效（数据损坏）.exe

1. 启动 HDFS 和 Zookeeper

实验环境和 7.2.2 节完全一样，具体参见 7.2.2 节中的步骤 1～6。

2. 模拟 t1 阶段写操作

（1）从 datanode00（192.168.1.13）和 datanode01（192.168.1.14）分别写入 10 个 100MB 文件。

（2）查看 t1 阶段写入情况。

3. 模拟 Primary 失效

（1）关闭 Primary。

（2）检查 t1 阶段写入情况。

4. 模拟 t2 阶段写操作

5. 模拟 NFS 失效，且数据损坏

（1）模拟失效，步骤同 7.5.2 中步骤 3。

（2）检查 t1、t2 阶段写入情况。

6. 模拟 t3 阶段写操作

具体步骤参见 7.4.2 中步骤 3。

7. NFS 恢复

（1）恢复 NFS 服务。分别将 Primary 和 Standby 的元数据文件复制到 NFS 的共享目录中。

（2）在 namenode0(Primary)上执行：

```
$cd /usr/local
$ls hadoop/local/
$rm -rf hadoop/local/*:*
$ls hadoop/local/
$scp -r /usr/local/hadoop/local/*
```

（3）在 namenode1(Standby)上执行：

```
   $cd /usr/local
   $ls hadoop/local/
   $rm -rf hadoop/local/*:*
   $ls hadoop/local/
   $scp -r /usr/local/hadoop/local/*
192.168.1.8:/usr/local/hadoop/avatarshare/share1
```

（4）恢复 NFS 的 IP。

（5）检查 t1、t2、t3 阶段写入情况。

8. 模拟 t4 阶段写操作

9. Standby 切换

（1）切换。在 Standby 上执行：

```
   bin/hadoop dfsadmin -saveNamespace
```

（2）检查 t1、t2、t3、t4 阶段写入情况。

10. 模拟 t5 阶段写操作

11. 检查

检查 t1～t5 阶段的写入情况

- 客户端 1（192.168.1.13）实验结果如表 7.10 所示。

表 7.10　写操作测试结果表

	T1	T2	T3	T4	T5
写入序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
失败序号	10001~10009	10000~10009	10000~10008	无	无

- 客户端 2（192.168.1.14）实验结果如表 7.11 所示。

表 7.11　写操作测试结果表

	T1	T2	T3	T4	T5
写入序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
失败序号	10005~10009	10000~10009	10000~10008	无	无

7.7.3　读操作实验步骤

1. 接上面的步骤继续

2. 产生读取文件

（1）清空 HDFS。

（2）从 192.168.1.13 和 192.168.1.14 向 HDFS 分别写入 10 个 100MB 文件。

3. 模拟 t1 阶段读操作

（1）准备读取所需的文件。

（2）从 HDFS 复制文件到本地，注意先清空本地目录。

（3）检查复制情况。

4. 模拟 Primary 失效

（1）模拟失效。

（2）检查 t1 阶段读取情况。

5. 模拟 t2 阶段读操作

6. 模拟 NFS 失效，且数据损坏

（1）模拟失效，步骤同 7.6.2 节中步骤 3。

（2）检查 t1、t2 阶段读取情况。

7. 模拟 t3 阶段读操作

8. NFS 恢复

（1）恢复。

（2）检查 t1、t2、t3 阶段读取情况。

9. 模拟 t4 阶段读操作

10. Standby 切换

（1）恢复。

（2）检查 t1、t2、t3、t4 阶段读取情况。

11. 模拟 t5 阶段写操作

12. 检查

检查 t1～t5 阶段的读情况。

- 客户端 1（192.168.1.13）实验结果如表 7.12 所示。

表 7.12　读操作测试结果表

	T1	T2	T3	T4	T5
读取序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
成功序号	10000~10003	无	无	10000~10009	10000~10009
失败序号	10004~10009	10000~10009	10000~10009	无	无

- 客户端 2（192.168.1.14）实验结果如表 7.13 所示。

表 7.13　读操作测试结果表

	T1	T2	T3	T4	T5
读取序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
成功序号	10000~10002	无	无	10000~10009	10000~10009
失败序号	10003~10009	10000~10009	10000~10009	无	无

7.7.4　小结

采用以上解决方案，其恢复时间和可靠性分析如下。

1. 恢复时间

由测试可知，该机制下，HDFS 的恢复时间：

NFS 恢复时间+Standby 手工切换时间

2．读、写操作影响

对于写操作来说，客户端和 Standby 都需要连接 Primary 进行通信，因此，在 Primary 失效后，手工切换前，读、写操作无法成功。

7.8 NFS 先失效（数据未损坏），Primary 后失效

7.8.1 解决方案

本节讨论 NFS 先失效（数据未损坏），Primary 后失效的异常情况，具体解决方案如下。

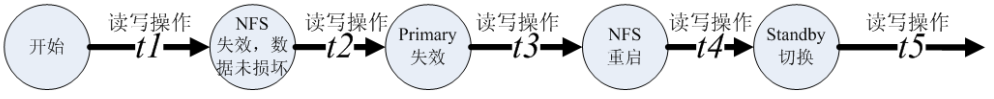

图 7.6 NFS 先失效（数据未损坏），Primary 后失效解决方案流程图

7.8.2 写操作实验步骤

具体步骤请参考视频：\视频\7 AvatarNode 异常解决方案\ 8 NFS 先失效（数据未损坏），Primary 后失效.exe

1. 启动 HDFS 和 Zookeeper

实验环境和 7.2.2 节完全一样，具体参见 7.2.2 节中的步骤 1～6。

2. 模拟 t1 阶段写操作

（1）从 192.168.1.13 和 192.168.1.14 向 HDFS 分别写入 10 个 100MB 文件。

（2）查看 t1 阶段写入情况。

3. 模拟 NFS 失效（数据未损坏）

（1）模拟失效、步骤同 7.5.2 节中步骤 3。

（2）检查 t1 阶段写入情况。

4. 模拟 t2 阶段写操作

5. 模拟 Primary 失效

（1）关闭 Primary。

（2）检查 t1、t2 阶段写入情况。

6. 模拟 t3 阶段写操作

具体步骤参见 7.4.2 节中步骤 3。

7. 启动 NFS

（1）NFS 恢复服务。

（2）恢复 NFS 的 IP。

（3）检查 t1、t2、t3 阶段写入情况。

8. 模拟 t4 阶段写操作

9. Standby 切换

（1）切换命令。

（2）检查 t1、t2、t3、t4 阶段写入情况。

10. 模拟 t5 阶段写操作

11. 检查

检查 t1～t5 阶段的写入情况。

- 客户端 1（192.168.1.13）实验结果如表 7.14 所示。

表 7.14　写操作测试结果表

	T1	T2	T3	T4	T5
写入序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
失败序号	10003~10004	10000~10001	10000~10001	无	无

- 客户端 2（192.168.1.14）实验结果如表 7.15 所示。

表 7.15　写操作测试结果表

	T1	T2	T3	T4	T5
写入序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
失败序号	10003~10004 10006	10000 10003	10000 10003	无	无

7.8.3　读操作实验步骤

1. 接上面的步骤继续

2. 产生读取文件

（1）清空 HDFS。

（2）从 192.168.1.13 和 192.168.1.14 向 HDFS 分别写入 10 个 100MB 文件。

3. 模拟 t1 阶段读操作

（1）准备需要读取的文件。

（2）从 HDFS 复制文件到本地，注意先清空本地目录。

（3）检查复制情况。

4. 模拟 NFS 失效（数据未损坏）

步骤同 7.4.2 节中步骤 3。

5. 模拟 t2 阶段读操作

6. 模拟 Primary 失效

（1）关闭 Primary。

（2）检查 t1、t2 阶段读取情况。

7. 模拟 t3 阶段读操作

具体步骤参见 7.5.2 节中步骤 3。

8. NFS 重启

（1）NFS 恢复服务。

（2）检查 t1～t3 阶段读取情况。

9. 模拟 t4 阶段读操作

10. Standby 切换

（1）切换。

（2）检查 t1～t4 阶段读取情况。

11. 模拟 t5 阶段写操作

12. 检查

检查 t1～t5 阶段的读情况。

- 客户端 1（192.168.1.13）实验结果如表 7.16 所示。

表 7.16　读操作测试结果表

	T1	T2	T3	T4	T5
读取序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
失败序号	10007~10009	10002~10003	10000~10009	10000~10009	无

- 客户端 2（192.168.1.14）实验结果如表 7.17 所示。

表 7.17　读操作测试结果表

	T1	T2	T3	T4	T5
读取序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
失败序号	10006~10008	10001~10003	10000~10009	10000~10009	无

7.8.4　小结

采用以上解决方案，其恢复时间和可靠性分析如下。

1．恢复时间

由测试可知，该机制下，HDFS 的恢复时间：

NFS 重启时间+Standby 手工切换时间

2．读、写操作影响

对于写操作来说，客户端和 Standby 都需要连接 Primary 进行通信，因此，在 Primary 失效后，手工切换前，读、写操作无法成功。

7.9 NFS 先失效（数据损坏），Primary 后失效（数据未损坏）

7.9.1 解决方案

本节讨论 NFS 先失效，接下来 Primary 无法服务且数据损坏无法恢复的情况。具体解决方案如下。

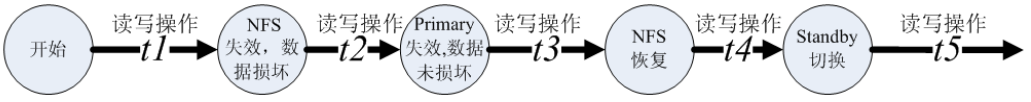

图 7.6 NFS 先失效（数据损坏），Primary 后失效（数据未损坏）解决方案流程图

其中 NFS 恢复的具体步骤如下。

（1）准备一个 NFS 节点。

（2）复制 Primary、Standby 的元数据文件到 NFS。

（3）重启 NFS。

7.9.2 写操作实验步骤

具体步骤请参考视频：\视频\7 AvatarNode 异常解决方案\ 9 NFS 先失效（数据损坏），Primary 后失效（数据未损坏）.exe

1. 启动 HDFS 和 Zookeeper

实验环境和 7.2.2 节完全一样，具体参见 7.2.2 节中的步骤 1～6。

2. 模拟 t1 阶段写操作

（1）从 192.168.1.13 和 192.168.1.14 向 HDFS 分别写入 10 个 100MB 文件。

（2）查看 t1 阶段写入情况。

3. 模拟 NFS 失效，且数据损坏

（1）删除元数据、关闭 NFS。

（2）检查 t1 阶段的写入情况。

4. 模拟 t2 阶段写操作

5. 模拟 Primary 失效

（1）关闭 Primary。

（2）检查 t1、t2 阶段的写入情况。

6. 模拟 t3 阶段写操作

7. NFS 恢复

（1）准备一个 NFS 节点。

（2）复制 Primary、Standby 的元数据文件到 NFS。

（3）重启 NFS。

（4）检查 t1～t3 阶段的写入情况。

8. 模拟 t4 阶段写操作

9. Standby 切换

（1）切换命令。

（2）检查 t1～t4 阶段的写入情况。

10. 模拟 t5 阶段写操作

11. 检查

检查 t1～t5 阶段的写入情况。

- 客户端 1（192.168.1.13）实验结果如表 7.18 所示。

表 7.18　写操作测试结果表

	T1	T2	T3	T4	T5
写入序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
失败序号	10003	无	无	无	无

- 客户端 2（192.168.1.14）实验结果如表 7.19 所示。

表 7.19　写操作测试结果表

	T1	T2	T3	T4	T5
写入序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
失败序号	10003~10004	无	无	无	无

7.9.3　读操作实验步骤

1. 接上面的步骤继续

2. 产生读取文件

（1）清空 HDFS。

（2）产生文件。

3. 模拟 t1 阶段读操作

4. 模拟 NFS 失效，且数据损坏

（1）关闭 NFS、删除数据。

（2）检查 t1 阶段读取情况。

5. 模拟 t2 阶段读操作

6. 模拟 Primary 失效（数据未损坏）

（1）关闭 Primary。

（2）检查 t1～t2 阶段读取情况。

7. 模拟 t3 阶段读操作

8. NFS 恢复

（1）恢复。

（2）检查 t1～t3 阶段读取情况。

9. 模拟 t4 阶段读操作

10. Standby 切换

（1）切换。

（2）检查 t1～t4 读取情况。

11. 模拟 t5 阶段写操作

12. 检查

检查 t1～t5 阶段的读情况。

- 客户端 1（192.168.1.13）实验结果如表 7.20 所示。

表 7.20　读操作测试结果表

	T1	T2	T3	T4	T5
读取序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
失败序号	10005~10009	10001~10009	10000~10009	10000~10009	无

- 客户端 2（192.168.1.14）实验结果如表 7.21 所示。

表 7.21　读操作测试结果表

	T1	T2	T3	T4	T5
读取序号	10000~10009	10000~10009	10000~10009	10000~10009	10000~10009
失败序号	10005~10009	10002~10009	10000~10009	10000~10009	无

7.9.4　小结

采用以上解决方案，其恢复时间和可靠性分析如下。

1．恢复时间

由测试可知，该机制下，HDFS 的恢复时间：

NFS 恢复时间+Standby 手工切换时间

2．读、写操作影响

对于写操作来说，客户端和 Standby 都需要连接 Primary 进行通信，因此，在 Primary 失效后，手工切换前，读、写操作无法成功。

7.10　实验结论

实验结果表明，以上异常情况的解决方案能够较好地解决 HDFS 系统元数据的 HA 问题。首先以上机制能够确保读写操作在切换过程中的可靠性；其次切换时间短，最差情况下也只需要机器重启的时间+手工切换时间，相比 NameNode 重启时消化 image 以及等待下面的 Data Node 来 Report block location 的时间来说，前者所需的切换时间最坏情况下也只需要几分钟，不受文件系统规模的影响，而后者的时间随文件系统的规模线性增长，以 fsimage 文件 12GB，2000 个 DataNode 大小规模的 HDFS 系统为例，Namenode 重启的时间大概需要 1 个小时。

第 8 章　Cloudera HA NameNode 使用

HA NameNode 的目标是为 HDFS 的 NameNode 提供热备节点，于 2012 年 2 月发布了测试计划，目前已完成了部分测试，其余的尚在进行之中，目前尚未见到有该机制的大规模、成熟的应用，其稳定性和性能有待进一步检验。

8.1 HA NameNode 说明

HA NameNode 的原理及架构与 AvatarNode 类似：包括 2 个 NameNode，1 个共享存储以及若干 DataNode 和 Client。

一个 NameNode 对外提供服务，处于 Active 状态，另外一个 NameNode 则通过共享存储与 Active NameNode 进行元数据同步，处于 Standby 状态。当 Active NameNode 失效时，可以通过 failover 机制将 Standby NameNode 切换为 Active 状态，接替失效的 Name Node 对外服务。

> HA NameNode 的实现目前还只支持手工的 failover，也就是说系统并不会自动监测 Active Name Node 的故障完成切换。下一步的开发计划是实现 Active Name Node 的自动检测，并自动完成 failover 切换。
>
> HA NameNode 依托于 Apache Hadoop 的主干版本进行开发，其主要工作围绕 HDFS-1623 和 HADOOP-7454 进行。2012 年 3 月 2 日，该项目的代码已合并到 Apache Hadoop 主干版本。社区计划将这部分工作合并到 0.23 分支，然后作为 0.23 的更新版进行发布，但目前 0.23 最新的发行版中还尚未合并该部分内容。Cloudera 于 2012 年 2 月发布的 CDH4 beta1 版本中包含了 HA Name Node 项目的内容，我们下面的 HA NameNode 实验也将使用该版本。

本章实验的主要目的是熟悉 HA NameNode 的使用方法，我们首先来看一下 HA NameNode 的网络拓扑结构，实验环境的网络拓扑如图 8.1 所示，整个系统从逻辑上有 NameNode、NFS Server、DataNode、Client 四种类型的节点，限于单台物理机的性能，在 dn1 至 dn3 上，每一个节点同时部署 Client 和 DataNode，总共需要 6 个虚拟机节点，具体情况如表 8.1 所示。

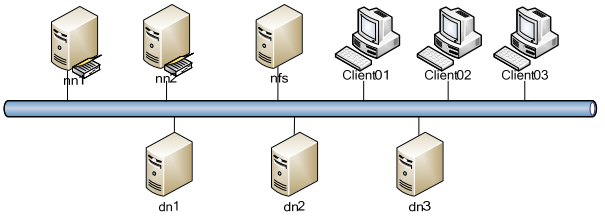

图 8.1　网络拓扑图

表 8.1 节点配置表

节点名	角 色	IP 地址	软件配置
nn1	NameNode	192.168.219.11	centos 6.2 for 32bit jdk1.6.0_24.tar.gz hadoop-0.23.0-cdh4b1.tar.gz
nn2	NameNode	192.168.219.12	
dn1~dn3	DataNode、Client	192.168.219.13 ~192.168.219.15	
nfs	NFS Server	192.168.219.16	

8.2　CDH4B1 版本 HDFS 集群配置

8.2.1　虚拟机安装

虚拟机的安装步骤请参考前面章节中描述，本章实验我们采用的操作系统为：centos 6.2 的 32 位版本，安装时注意选择安装 SSH 服务器以及 NFS 服务器，并创建一个用户名为“user”的普通用户。

8.2.2　nn1 配置

首先复制 8.2.1 节中安装好的虚拟机文件作为 nn1，然后对 nn1（第 1 个 Name Node，nn1 既是虚拟机名称又是主机名）进行配置，nn1 配置完毕后，以此为基础，复制 nn1 的虚拟机文件作为其他节点并配置，主要步骤描述如下。

（1）配置虚拟机内存大小为 256MB。

（2）修改硬盘文件路径（虚拟机复制后，其虚拟磁盘还是使用的复制时的路径，因此需要将其修改为当前虚拟机磁盘所在路径）。

（3）修改虚拟机名字为 nn1，如图 8.2 所示。

图 8.2　修改虚拟机名称

（4）修改网卡的连接方式为 Host-only，如图 8.3 所示。

图 8.3　修改网卡连接方式

（5）系统配置。

① 以用户 user 登录。

② 切换到 root 用户，删除/etc/udev/rules.d/70-persistent-net.rules 里面的内容。

> 由于虚拟机复制后会产生新的 Mac 地址，而 rules.d/70-persistent-net.rules 中仍然记录了复制之前虚拟机的 Mac 地址，这样会导致网卡的编号不是从 0 开始，删除这部分内容后，可以使得网卡从 0 开始编号。

③ 查看当前网卡的 Mac 地址。

```
$ip addr show
```

④ 修改网卡配置。

```
$sudo vi /etc/sysconfig/network-scripts/ifcfg-eth0
```

需要修改的地方包括 HWADDR，HWADDR 以机器实际 Mac 地址为准；ONBOOT 修改为 Yes；IPADDR 修改为 192.168.219.11，NETMASK 修改为 255.255.255.0，具体如下。

```
DEVICE="eth0"
HWADDR="00:0c:29:23:40:21"
NM_CONTROLLED="yes"
ONBOOT="yes"

NETMASK=255.255.255.0
```

⑤ 修改主机名为 nn1。

```
$sudo vi /etc/sysconfig/network
```

⑥ 修改 hosts，内容如下：

```
192.168.219.11  nn1
192.168.219.12  nn2
192.168.219.13  dn1
```

```
192.168.219.14  dn2
192.168.219.15  dn3
192.168.219.16  nfs
```

⑦ SSH 免密码登录。

```
$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
$ chmod 600 ~/.ssh/authorized_keys
$sudo /etc/rc.d/init.d/sshd restart
$ssh nn1
```

⑧ 将 user 添加到 sudoers。

```
#chmod +w /etc/sudoers
#vi /etc/sudoers
```

在“root ALL=(ALL) ALL”下增加一行“user ALL=(ALL) ALL”。

```
#chmod -w /etc/sudoers
#su - user
```

⑨ 关闭防火墙的自启动。

```
$sudo chkconfig --del iptables
$sudo chkconfig --list | grep iptables
```

⑩ 重启，使配置生效。

（6）上传软件。

将 jdk1.6.0_24.tar.gz 和 hadoop-0.23.0-cdh4b1.tar.gz 上传到 nn1 的/home/user 目录下，并解压。

```
$cd /home/user


$tar xf hadoop-0.23.0-cdh4b1.tar.gz
```

（7）配置 JAVA_HOME。

```
$sudo vi /etc/profile
```

在最后一行追加如下内容。

```
export JAVA_HOME=/home/user/jdk1.6.0_24
```

（8）配置 hadoop-env.sh。

```
$cd  /home/user/hadoop-0.23.0-cdh4b1/
$cp share/hadoop/common/templates/conf/hadoop-env.sh etc/hadoop
$vi etc/hadoop/ hadoop-env.sh
```

将 JAVA_HOME 和 HADOOP_LOG_DIR 的定义修改如下。

```
export JAVA_HOME=/home/user/jdk1.6.0_24
export HADOOP_LOG_DIR=/home/user/hadoop/log
```

（9）关机。

```
$sudo shutdown -h now
```

8.2.3　dn1~dn3 配置

复制虚拟机 nn1 文件，分别作为 dn1～dn3，复制后对每个虚拟机进行相应的配置，具体见参见 8.2.2 节中步骤 1 到步骤 5 的描述，在此不再重复。下面分别描述 dn1、dn2、dn3 、nn1 有关 HDFS 集群的配置。

（1）dn1 配置。

① 清除 DataNode 数据。

```
$mkdir -p /home/user/hadoop/datanode
```

② 配置 core-site.xml。

```
$vi /home/user/hadoop-0.23.0-cdh4b1/etc/hadoop/core-site.xml
```

内容如下：

```
<property>
<name>fs.defaultFS</name>
<value>hdfs://nn1:8020</value>
<description>The name of the default file system.  A URI whose
   scheme and authority determine the FileSystem implementation.  The
   uri's scheme determines the config property (fs.SCHEME.impl) naming
   the FileSystem implementation class.  The uri's authority is used to

</description>
</property>

<property>
<name>hadoop.tmp.dir</name>
<value>/home/user/hadoop/tmp</value>
<description>A base for other temporary directories.</description>
</property>
```

其中，fs.defaultFS 为默认文件系统名字的配置项，nn1 为 NameNode 的主机名。hadoop.tmp.dir 为 hadoop 临时目录的配置项。

③ 配置 hdfs-site.xml：

```
$vi /home/user/hadoop-0.23.0-cdh4b1/etc/hadoop/hdfs-site.xml
```

内容如下：

```
<property>
   <name>dfs.datanode.data.dir</name>
   <value>/home/user/hadoop/datanode</value>
   <description>Determines where on the local filesystem an DFS data node
   should store its blocks.  If this is a comma-delimited
```

```
    list of directories, then data will be stored in all named
    directories, typically on different devices.
    Directories that do not exist are ignored.
    </description>
</property>
```

dfs.datanode.data.dir 配置了 DataNode 的数据存储路径。

（2）dn2 配置。

dn2 的步骤与 dn1 完全相同。

（3）dn3 配置。

dn3 的步骤与 dn1 完全相同。

（4）nn1 的 HDFS 配置

① 配置 hdfs-site.xml。

在 nn1 上运行如下命令：

```
$vi /home/user/hadoop-0.23.0-cdh4b1/etc/hadoop/hdfs-site.xml
```

内容如下，其中 dfs.namenode.name.dir 配置了 NameNode 元数据存储路径。

```
<configuration>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>/home/user/hadoop/namenode</value>
  </property>
```

② core-site.xml 的配置文件与 dn1 的 core-site.xml 相同。

③ 修改 slaves，配置 3 个 DataNode 节点。

```
$vi /home/user/hadoop-0.23.0-cdh4b1/etc/hadoop/slaves
```

内容如下：

```
dn1
dn2
dn3
```

8.2.4 HDFS 集群构建

（1）格式化。

在 nn1 上运行。

```
$cd /home/user/hadoop-0.23.0-cdh4b1
$bin/hdfs namenode -format
```

（2）启动集群。

在 nn1 上运行以下脚本，该脚本将启动 nn1 上的 NameNode、dn1～dn3 上的 DataNode，以及一个 secondary namenode。

```
$sbin/start-dfs.sh
```

正常情况下，在 nn1 的 putty 客户端将显示如图 8.4 所示的信息。

图 8.4　HDFS 启动信息显示

在 dn3 上查看启动情况。

```
$bin/hdfs dfsadmin -report
```

如果正常启动，将显示如图 8.5 所示的信息。

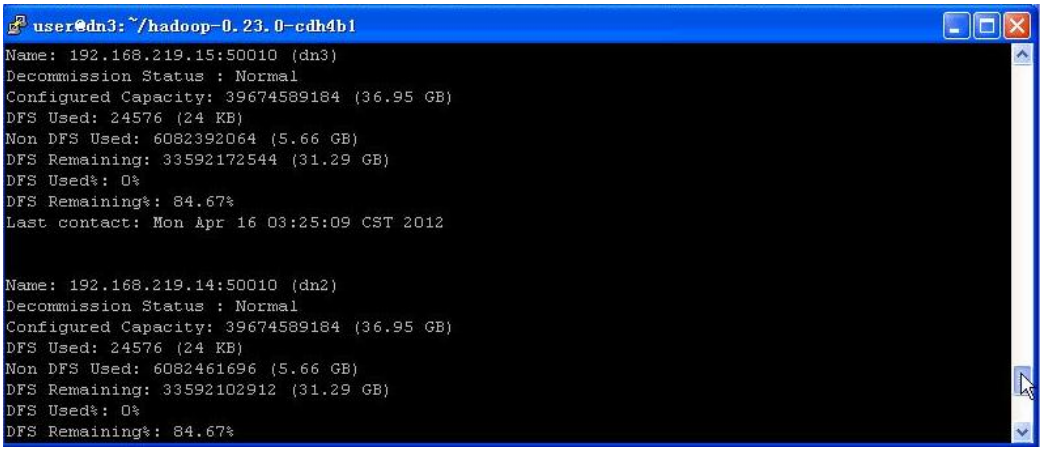

图 8.5　HDFS 文件系统信息显示

也可以通过 Web 访问进行验证。如果使用 Windows 主机访问 HDFS 的 Web，需要修改其 hosts 文件，位于 C:\WINDOWS\system32\drivers\etc，内容如下：

```
192.168.219.11  nn1
192.168.219.12  nn2
192.168.219.13  dn1
192.168.219.14  dn2
192.168.219.15  dn3
192.168.219.16  nfs
```

修改完毕后，在浏览器中输入 http://192.168.219.11:50070，正常情况下将显示如图 8.6 所示的信息。

图 8.6　HDFS Web 显示

（3）写入文件。

在 dn3 上产生文件，然后写入 HDFS，命令如下：

```
$cd /home/user/hadoop-0.23.0-cdh4b1
$dd if=/dev/zero of=/tmp/32M bs=1M count=32
```

写入 HDFS。

```
$bin/hdfs dfs -mkdir data
```

8.3　HA NameNode 配置

8.3.1　nn1 配置

（1）配置 core-site.xml。

```
$vi /home/user/hadoop-0.23.0-cdh4b1/etc/hadoop/core-site.xml
```

内容如下：

```
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://hacluster</value>
    <description>The name of the default file system.  A URI whose
    scheme and authority determine the FileSystem implementation.  The
    uri's scheme determines the config property (fs.SCHEME.impl) naming
    the FileSystem implementation class.  The uri's authority is used to


  </property>


  <property>
  <name>hadoop.tmp.dir</name>


  <description>A base for other temporary directories.</description>
</property>
```

注意：相对前面的集群的 nn1 配置，fs.defaultFS 的值由 hdfs://nn1:8020 变成 hdfs://hacluster，nn1 是 NameNode 的主机名，只代表一个单独的节点，hacluster 是 NameNode 名字服务的逻辑名，可以配置多个 NameNode 来提供名字服务，目前支持的个数为 2。对于 nn1 来说，对于所有的操作，客户端默认只会与 nn1 这个 Name Node 联系；而对于 hacluster 来说，hacluster 对应多个 Name Node，这样就为客户端实现 NameNode 访问的自动切换提供了可能。

（2）配置 hdfs-site.xml。

```
$vi /home/user/hadoop-0.23.0-cdh4b1/etc/hadoop/hdfs-site.xml
```

内容如下：

```
<property>
   <name>dfs.namenode.name.dir</name>
   <value>/home/user/hadoop/namenode</value>
</property>
```

dfs.namenode.name.dir 的内容没有改变，设置了元数据存储路径。

```
<property>
   <name>dfs.namenode.secondary.http-address</name>
   <value> </value>
   <description>
   The secondary namenode http server address and port.
   If the port is 0 then the server will start on a free port.
   </description>
</property>
```

按照官方文档的说法，由于 Standby Name Node 包含了做 Checkpoint 的功能，因此 HA NameNode 中也不需要 Secondary NameNode，CheckpointNode，or BackupNode，如果配置了将引起错误。

> 注意：dfs.namenode.secondary.http-address 的值在此配置成一个空格符，表示不启动 SecodaryNameNode。

```
<property>
   <name>dfs.datanode.data.dir</name>
   <value>/home/user/hadoop/datanode</value>
   <description>Determines where on the local filesystem an DFS data node
```

```
    list of directories, then data will be stored in all named
    directories, typically on different devices.
    Directories that do not exist are ignored.
    </description>
</property>
```

dfs.datanode.data.dir 和原来一样，配置的是 DataNode 的保存路径。

```
<!-- for ha. -->
<property>
    <name>dfs.federation.nameservices</name>
    <value>hacluster</value>
</property>

<property>
    <name>dfs.ha.namenodes.hacluster</name>
    <value>nn1,nn2</value>
</property>
```

在 HDFS federation 中，一个 HDFS 集群可以有多个互相隔离的名字空间，每个名字空间对应一个名字服务的逻辑名，dfs.federation.nameservices 配置了 HDFS 名字服务的逻辑名，名字之间以“，”隔开。在本例中，HDFS 只有一个名字空间，对应的逻辑名为 hacluster。在 HA NameNode 中，一个逻辑名对应多个 NameNode，其中一个作为 Active Name Node 提供服务，其余的作为 Standby Name Node，具体的配置在 dfs.ha.namenodes.hacluster 中。本例中，hacluster 对应的 NameNode 为 nn1 和 nn2。

```
<property>
    <name>dfs.namenode.rpc-address.hacluster.nn1</name>

</property>
```

```
   <property>
      <name>dfs.namenode.rpc-address.hacluster.nn2</name>
      <value>nn2:8020</value>
   </property>
```

分别配置 NameNode nn1 和 nn2 上完整的 RPC 地址。

```
   <property>
      <name>dfs.namenode.http-address.hacluster.nn1</name>
      <value>nn1:50070</value>
   </property>

   <property>
      <name>dfs.namenode.http-address.hacluster.nn2</name>
      <value>nn2:50070</value>
   </property>
```

分别配置 NameNode nn1 和 nn2 上的 HDFS 的 Web 地址。

```
   <property>
      <name>dfs.namenode.shared.edits.dir</name>
      <value>file:///home/user/hadoop/nfs</value>
   </property>
```

配置 NameNode nn1 和 nn2 元数据共享路径。

```
   <property>


<value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxy
Provider</value>
   </property>
```

配置 HDFS 客户端用来与 Active Name 通信的 Java 类，该 Java 类用来确定哪一个是 Active Name Node，目前 Hadoop 只包含了一个唯一的实现 ConfiguredFailoverProxyProvider。

```
<property>
   <name>dfs.ha.fencing.methods</name>
   <value>sshfence</value>
</property>

<property>

   <value>/home/user/.ssh/id_dsa</value>
</property>
```

在 HA NameNode 中，为了确保 HDFS 的正确性，需要一种机制来确保在任何时候只有一个 Active Name Node。dfs.ha.fencing.methods 可以指定具体的机制，这些机制可以是脚本，也可以是 Java 类，也可以有多个，以回车分隔。

在进行 failover 时，HA NameNode 将按照 dfs.ha.fencing.methods 所配置的方法，依次尝试，直到其中一个成功为止。

在 Hadoop 中包含了两个 fencing 方法：<shell>和<sshfence>。本例选择<sshfence>方法，同时还需要配置 dfs.ha.fencing.ssh.private-key-files，用于无密码登陆所需要操作的节点。

8.3.2　其他节点配置

（1）配置 nn2。

复制 nn1 虚拟机文件，作为新的节点 nn2，具体步骤请参 8.2.2 节中步骤 1 到步骤 5 的描述。

（2）分发配置文件。

使用 Scp 将 nn1 的 core-site.xml 和 hdfs-site.xm 分发到 dn1～dn3 对应的目录下，同时清空/home/user/hadoop/namenode 以及/home/user/hadoop/datanode 等目录。

（3）配置 nfs。

① 复制一个新的节点 nfs，具体步骤请 8.2.2 节中步骤 1 到步骤 5 的描述。

```
$mkdir -p /home/user/hadoop/nfs
$rm -rf /home/user/hadoop/nfs/*
```

② 修改 nfs 的配置文件。

```
$sudo vi /etc/sysconfig/nfs
```

增加如下内容，用于解决 NFS 的挂载问题。

```
RPCNFSDARGS="-N 2 -N 3"
# Turn off v4 protocol support
RPCNFSDARGS="-N 4"
```

③ 修改 nfs 的 exports 文件。

```
$sudo vi /etc/exports
```

内容如下，我们将设置/home/user/hadoop/nfs 作为共享路径。

```
/home/user/hadoop/nfs   *(rw,sync,no_root_squash)
```

④ 启动 NFS 服务。

```
$sudo /etc/rc.d/init.d/nfs start
```

⑤ 在 nn1 上挂载 NFS 目录。

```
$mkdir -p /home/user/hadoop/nfs
$rm -rf /home/user/hadoop/nfs/*
$sudo umount /home/user/hadoop/nfs
```

```
$mount
```

如果成功，将显示如图 8.7 所示的信息。

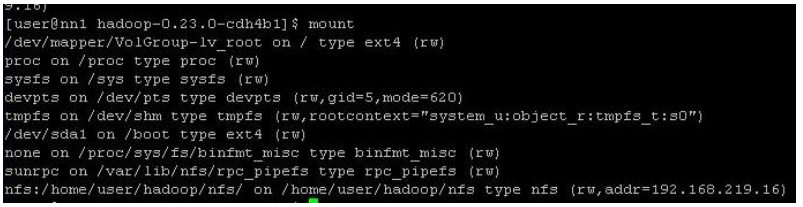

图 8.7　NFS 挂载信息

在 nn2 上执行同样的命令，挂载 NFS 目录。

8.4　HA NameNode 使用

8.4.1　启动 HA HDFS 集群

（1）清空 nn1、nn2/home/user/hadoop/namenode 下的数据，以及 dn1～dn3/home/user/hadoop/namenode 下的数据。

（2）格式化 nn1。在 nn1 上执行：

```
$cd  /home/user/hadoop-0.23.0-cdh4b1/
```

（3）复制格式化的元数据至 nn2。在 nn1 上执行：

```
$scp -r /home/user/hadoop/namenode/* nn2:/home/user/hadoop/namenode/
```

（4）启动集群。在 nn1 上执行：

```
$sbin/start-dfs.sh
```

nn1 和 nn2 启动初始都是 Standby 状态，此时使用 bin/hdfs dfsadmin -report 无法查询 HDFS 的信息，DataNode 也无法注册，需要将其中一个 NameNode 切换为 Active 状态，才能正常工作。

8.4.2 第 1 次 failover

切换，在 nn1 上执行：

```
$bin/hdfs haadmin -failover nn2 nn1
```

切换成功后，nn1 将处于 Active 状态，nn2 仍为 Standby 状态。此时，DataNode 可以注册，HDFS 的状态也可以正常查询。

8.4.3 模拟写操作

向 HDFS 写入文件，在 dn3 上执行。

```
$bin/hdfs dfs -mkdir data
```

正常情况下，可以在浏览器中看到如图 8.8 所示的信息。

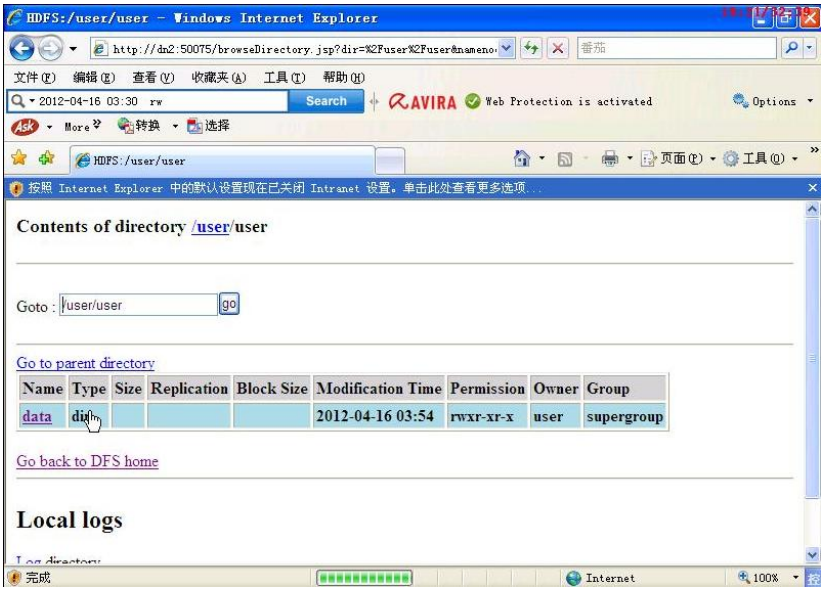

图 8.8 HDFS 文件系统信息 Web 界面

在 dn3 上复制文件至 HDFS。

```
$bin/hdfs dfs -copyFromLocal /tmp/32MB ./data/
```

```
$bin/hdfs dfs -copyFromLocal /tmp/32MB ./data/32MB-1
```

8.4.4　模拟 Active Name Node 失效，第 2 次 failover

（1）模拟 nn1 故障，切换至 nn2。

在 nn1 上执行：

```
$bin/hdfs haadmin -failover nn1 nn2
```

该命令可以使得 nn2 完成从 Standby 到 Active 的切换，同时，通过调用 fencing 机制，确保 nn1 不再处于 Acitive 状态。如果使用 transitionToActive 进行 State 转换时，将不会调用 fencing 机制来确保只有一个 Active Name Node，因此应尽量少使用，而应使用 failover 命令。切换后的结果如图 8.9 所示。

```
[user@nn1 hadoop-0.23.0-cdh4b1]$ bin/hdfs haadmin -getServiceState nn2
active
[user@nn1 hadoop-0.23.0-cdh4b1]$ bin/hdfs haadmin -getServiceState nn1
standby
```

图 8.9　Name Node 状态

（2）查看 nn2：

```
$bin/hdfs dfs -ls ./data
```

如图 8.10 所示。

```
-rw-r--r--   3 user supergroup   32000000 2012-04-16 03:55 data/32MB
-rw-r--r--   3 user supergroup   32000000 2012-04-16 03:56 data/32MB-1
-rw-r--r--   3 user supergroup   32000000 2012-04-16 03:57 data/32MB-2
```

图 8.10　写入文件信息

结果显示，没有文件丢失。

（3）向 HDFS 写入文件。

```
$dd if=/dev/zero of=/tmp/16MB bs=1MB count=16
$bin/hdfs dfs -copyFromLocal /tmp/16MB ./data/
$bin/hdfs dfs -ls ./data
```

结果显示如图 8.11 所示，可以正常写入。

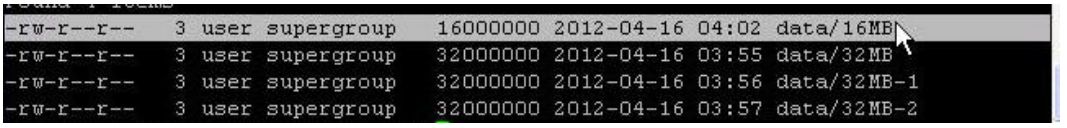

图 8.11　写入文件信息

8.3.5　模拟新的 Standby NameNode 加入

模拟 nn1 作为新节点加入。在 nn1 上执行：

```
$sudo reboot
```

重启后，登录 nn1，执行：

```
$sudo mount -t nfs nfs:/home/user/hadoop/nfs /home/user/hadoop/nfs
$cd /home/user/hadoop
$rm -rf namenode/*
$scp -r nn2:/home/user/hadoop/namenode/* namenode/
$cd /home/user/hadoop-0.23.0-cdh4b1
```

查看状态，如图 8.12 所示，结果显示，新节点 nn1 已加入成功。

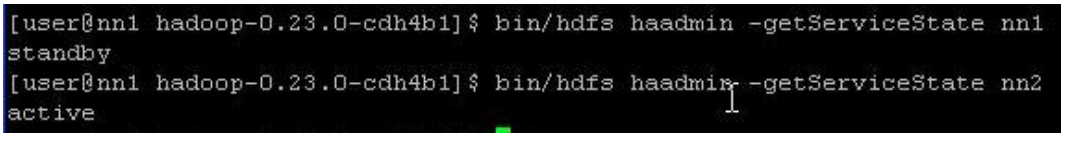

图 8.12　Name Node 状态

8.5　小 结

HA NameNode 在功能和机制上和 AvatarNode 类似，都提供了 Name Node 的热备机制，两者的使用也大同小异。目前 HA NameNode 还只支持手工 failover，处于测试阶段，功能和性能还有待进一步检验，与之相比，AavtarNode 已经过大规模、长时间的测试和实际使用，可靠性和性能都有保证。从长远来看，HA Name Node 已合并到社区 HDFS 的主干版本之中，社区 release 版本的发布只是时间问题，而 AvatarNode 更多的是关注 Facebook 所维护开发的 HDFS 版本，用户可以根据自己的实际情况灵活选择。