单位代码：10293 密　级：

南京邮电大学

专 业 学 位 硕 士 论 文

论文题目：HDFS 高可用性方案的优化与实现

学　　号	1215043023
姓　　名	胡文龙
导　　师	王少辉
专业学位类别	工程硕士
类　　型	全　　日　　制
专业（领域）	计算机技术
论文提交日期	二〇一八年四月

万方数据

The Optimization and Implementation of HDFS High Availability Scheme

Thesis Submitted to Nanjing University of Posts and Telecommunications for the Degree of

Master of Engineering

By

Wenlong Hu

Supervisor:Prof.Shaohui Wang

April 2018

万方数据

南京邮电大学学位论文原创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南京邮电大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

本人学位论文及涉及相关资料若有不实，愿意承担一切相关的法律责任。

研究生学号：_____研究生签名：_____ 日期：_____

南京邮电大学学位论文使用授权声明

本人承诺所呈交的学位论文不涉及任何国家秘密，本人及导师为本论文的涉密责任并列第一责任人。

本人授权南京邮电大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档；允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入有关数据库进行检索；可以采用影印、缩印或扫描等复制手段保存、汇编本学位论文。本文电子文档的内容和纸质论文的内容相一致。论文的公布（包括刊登）授权南京邮电大学研究生院办理。

非国家秘密类涉密学位论文在解密后适用本授权书。

研究生签名：_____ 导师签名：_____ 日期：_____

万方数据

摘要

随着互联网的蓬勃发展，越来越多的数据在后台服务器中产生。如何科学地存储这些海量数据成了当前行业面临的挑战之一。近些年，随着大数据技术的迭代与发展，分布式文件存储系统 HDFS（Hadoop Distributed File System）得到了业界广泛的认可与应用。但当前版本的HDFS为了保证系统的高可用性所采用的主-从架构的多副本机制只能刚刚满足了基本功能需求，在应对单点故障和数据存储利用率这两个方面还存在着不少优化空间。

针对上述两个问题，本文做了以下工作：

（1）提出了一种基于局部校验纠删码算法的 HDFS 数据存储策略。

通过对 HDFS 当前版本的研究与分析，系统采用的是对原始数据创建副本的方式来避免因某些节点失效而导致的数据丢失问题。不难看出，在今天这个信息量俱增的互联网时代，副本策略需要消耗大量的底层硬件存储设备。所以本文提出一种基于局部校验纠删码算法的 HDFS 数据存储策略。该算法相较于副本策略能够显著降低磁盘的存储开销，而在对失效数据的重构过程又不像 RS 编码一样需要从各个网络节点中拉取所有剩余数据，与 EVENODD 编码与 X 编码这一类阵列码相比，改进算法在数据节点的个数上设置更加灵活。

（2）提出一种扁平化的高可用 NameNode 模型。

HDFS 文件系统采用主-从架构，集群主要由一个负责管理文件系统的 NameNode 节点和许多用来存储数据的 DataNode 节点组成。NameNode 节点维护着文件系统的命名空间，处理并响应来自客户端的请求。NameNode 能否正常工作关系到整个分布式文件系统的高可用性。本文分析了现有的几种保障 NameNode 高可用的性的方案，针对它们的不足与优势，提出了一种扁平化的高可用 NameNode 模型，该方案不仅缩短了 HDFS 在 NameNode 节点宕机后的崩溃恢复时间，还实现了 HDFS 集群在响应许多客户端请求时更好的负载均衡。

（3）实验与分析。

通过虚拟化软件来模拟真实集群环境来作为实验平台，在集群中部署了改进版 Hadoop 安装文件。实验首先针对本文所提出的局部校验纠删码算法与经典的 RS 编码进行了编码与重构原数据时的效率测试，之后对扁平化的 NameNode 模型的高可用性进行了验证测试。实验效果以图表和截图的形式展现出来。

关键词：大数据，HDFS，纠删码，高可用性

万方数据

Abstract

With the vigorous development of the Internet, more and more data are generated in the background server. How to scientifically store these massive data has become one of the challenges in the current industry. In recent years, with the iteration and development of big data technologies, the Hadoop Distributed File System(HDFS) has been widly confirmed and applied. However, the current version of HDFS, which adopts Master-Slave architecture and multiple-copy mechanism, just meets basic functional needs and still has the further optimizing space in the aspect of Single Point of Failure(SPOF) and storage utilization.

In view of the above two problems, the main works are as follows:

(1) A HDFS storage strategy based on locally repairable code is proposed.

Through the research and analysis of the current version of HDFS, filesystem using the way to create multiple copies to avoid data loss. It’s not hard to see that the replica strategy needs to consume a large number of storage devices in this era. Therefore, this paper proposes a HDFS data storage strategy based on locally repairable erasure code. This algorithm can significantly reduce the storage overhead of the disk compared with the replica strategy and the process of reconstructing the invalid data does not like RS code, which needs to pull all the remaining data from each network node. Compared with EVENODD code and X code, the improved algorithm is more flexible in the number of data nodes.

(2) A flat and high availability NameNode model is proposed.

The HDFS adopts the Master-Slave architecture, which is mainly composed of one NameNode and multiple DataNodes. NameNode is responsible for managing the whole set of metadata of the cluster and handling all requests from clients. DataNodes are responsible for storing data. Whether the NameNode is working properly is related to the high availability of the entire distributed file system. This article analyzes several existing proposals to guarantee the high availability of NameNode and summaries their advandages and disadvandages. Then, a flat and high availability NameNode model is proposed. This proposal Not only shortens the recovery time of HDFS when the NameNode collapses, but also achieves better load balancing when many clients request access.

(3) The implementation and analysis of the system.

The improved version of the Hadoop installation file is deployed in the cluster by using the virtual software to simulate the real cluster environment as an experimental platform. Experiments

万方数据

test the efficiency of encoding and reconstructing original data based on the proposed locally repairable erasure code and classic RS code firstly. After that, a series of experiments verify the high availability of the flat NameNode model. The experimental results are shown in the form of charts and screenshots.

Key words: big data, HDFS, erasure code, high availability

万方数据

目录

万方数据

万方数据

第一章 绪论

1.1 课题背景及意义

经过几年的概念热炒，大数据逐步走过了探索阶段、市场启动阶段，目前在接受度、技术、应用等各个方面已趋于成熟，开始步入产业的快速发展阶段[1]。大数据巨大的应用价值带动了大数据行业的迅速发展，行业规模增长迅速。根据中国信息通信研究院于 2017 年 3 月发布的《中国大数据发展调查报告(2017)》中的调查显示，截至 2016年，已有近六成的企业成立了数据分析相关部门，超过 1/3 的企业应用了大数据处理框架。中国大数据产业的市场规模已达到了 168亿元，预计 2017 年~2020 年仍将保持每年 30%以上的增长速度[2]。大数据相关技术的运用为企业带来最明显的效果是实现了智能决策和提高了运营效率。

随着 PC、手机、可穿戴设备等智能终端硬件的大批量生产，数据载体的不断增多，引发了全球数据的急剧增长。根据数据监测统计结果显示，2011 年全球数据总量已经达到 1.8ZB（1ZB=2^{30}TB，1.8ZB 相当于 18 亿个 1TB 的移动硬盘），而这个数值还在以每两年翻一番的趋势增长。2015 年全球的数据总量为 8.6ZB，目前全球数据的增长速度维持在每年 40%左右，预计到 2020 年全球的数据总量将达到 40ZB[3]。

我们正处于一个“数据洪流”时代，而大部分数据都被严密锁存在一些大型互联网公司、科研院所和金融机构中[4]。这就产生一个问题，在硬盘存储容量逐年大幅度提升的情形下，硬盘的读写速度却没有达到一个与之相匹配的程度。1990 年，一块普通的硬盘可以存储约 1370M 大小的数据，文件读取速度为 4.4MB/s，大约只需 5 分钟就可以读完整块硬盘中的数据。20 多年过去了，1TB 的硬盘已然成为主流，但数据传输速度提升确并不显著，约为 100MB/s，读完整个硬盘中的数据至少需要花费 2.5 个小时[5]。由此可见，传统的集中式的存储系统存在着性能的瓶颈。要想解决这个问题，一个简单易行的方法就是让一个客户端同时从多个硬盘上读取数据。试想，如果我们有 100 个硬盘，每个硬盘只存储一份文件 1%的数据量，采用并行读取的方式，那么不到两分钟就可以读完这份文件所包含的所有数据[6]。这种分布式的存储架构，采用了多个存储器同时处理一个读写请求的思想，不仅有效地降低了每个存储器的负荷，提高了系统的整体性能，还方便了在硬件层面的扩展。虽然分布式的存储架构在许多方面都要优于集中式的存储架构，但要同时对多个存储服务器中的数据进行并行读写，还有很多问题需要考虑。首当其冲就是个别硬件设备在遭遇到突发性故障时能否不影响整个系统的正常运行。一旦将许多硬件设备作为一个整体对外提供服务时，其中的个别硬件就很有可能

万方数据

发生故障。许多行业的存储服务系统都需要提供 7×24 小时不间断服务，任何服务的中断，无论是计划内的还是计划外的，都会造成巨大的损失[7]。为了保证集群中任何单台或多台服务器在发生故障时均不会导致系统对外提供的服务不可用，要求集群中的其他正常节点要能在几秒钟甚至更短时间内自动接管故障节点的资源和服务。这就要求我们需要构建一个高可用的分布式存储系统[8]。

Hadoop 是一个集合了众多分布式应用的框架，现由 Apache 基金会牵头开发并维护[9]。在用 Hadoop 框架做数据处理时，开发人员不需要知道分布式并行计算底层的实现细节，可以直接通过调用相应接口来开发基于分布式环境的应用程序，简单高效。Hadoop 框架最早起源于 Google 公司发表的三篇分布式领域的论文，它们分别是《The Google File System》[10]、《MapReduce》[11]以及《BigTable》[12]。Google 公司虽然没有将这些产品的代码开源，但在论文中给出了它们的详细设计，最后由 Yahoo 公司资助，依据这些论文，基于 Java 语言开发出了最初的 Hadoop 框架。虽然在性能上，当时版本的 Hadoop 不及 Google 内部使用的软件产品，但因为其开源的特点，目前它已经成为分布式数据处理领域最流行、应用最广泛、集各家之所长的大数据计算框架。

近年来随着云计算、大数据的兴起，也催生了一批分布式的存储应用，以 Hadoop 的 HDFS（Hadoop Distributed File System）最为流行[13]。HDFS 设计之初便考虑到要能部署在廉价的硬件之上，得益于这点，促使它在分布式存储领域得到了快速推广与应用。Hadoop 生态系统中的其他组件，诸如计算框架 MapReduce 和非关系型数据库 HBase 等，都需要借助 HDFS 为底层的数据存储提供保障[14]。

近些年，伴随着越来越多用户的请求访问，HDFS 上需要存储的数据也急剧增加，然而我们依然追求越来越短的请求响应时间和越来越高的存储利用率。而当前版本的 HDFS 为了保证其上海量数据的访问速度与容错能力采用了对原数据创建副本的方式来避免可能发生的数据丢失问题。存储于 HDFS 上的文件默认副本数量为 3 个，并分散保存于不同的数据节点中。个别重要文件还可以通过人工修改配置参数的方法来实现更多副本数。不难看出，在今天这个信息爆炸的时代，副本策略需要配备同样数量庞大的底层硬件存储设备，无论从时间成本还是经济效益角度来考量，副本策略都不可谓是一个理想的方案。针对这样的问题，有人提出了基于纠删码算法的 HDFS 容错技术，但现行的方案在重构丢失数据时的效率并不高。如何在保证数据可靠性的前提下降低存储开销是一个十分有意义，值得探索的研究方向，同时也是本文所研究的重点。

HDFS 除了要改进 3 副本机制所带来的存储利用率低的问题，其主-从模式的系统架构也存在着一定的单点故障隐患。单点故障是指当系统中的某个模块失效后，整个系统也会受其

万方数据

影响不能正常对外提供服务。至今为止，Apache 基金会所发布的各版本 Hadoop 中一直未能便捷有效地解决 HDFS 主节点 NameNode 的单点故障隐患。在一些需要借助后台数据来支撑公司业务的企业，存储系统一刻的宕机都会给企业造成不可估量的损失。因此研究 NameNode 的单点故障问题具有重要的理论价值和商业价值。

1.2 国内外研究现状

1.2.1 分布式存储研究现状

近年来，随着大数据、云计算等技术的推广及应用，为它们提供底层数据存储服务的分布式存储系统得到了蓬勃的发展[2,15]。传统的集中化存储还是存在着管理缺失和数据存储不安全等问题，而分布式文件系统将业务数据与元数据分开管理。业务数据存储于大量分散的、廉价的商用硬件上，而管理与监督业务数据的元数据信息则在另一台机器上单独维护。这为需要对海量数据进行高效管理和存储的互联网、电信等企业来说，方便了它们对整个存储系统的统一管理，并且降低了维护成本。

分布式存储系统越来越演变成存储领域的热点问题，当前国内外学术界主要的研究成果有麻省理工学院的 CFS[16]，加利福利亚大学伯克利分校的 OceanStore[17]和北京大学的燕星[18]等。

商业化的分布式存储系统同样也已诞生不少，Google、Amazon、阿里巴巴、中国移动等国内外 IT 企业或运营商都部署了各自定制化的存储服务系统。

Amazon 公司的云存储服务的核心关键技术 S3（Simple Storage Service）[19]。它是一个多功能、易扩展、能按需使用的分布式存储系统。用户可以将程序调用过程中所需要的相片，影像，音乐，文档等数据资料直接存储到 S3 上面。对于用户来说，使用 S3 服务就像使用带第三方硬盘驱动的本地硬盘一样。但实际上，他所使用的分布式文件系统是对散布在各个地理位置的许多机器上存储空间进行虚拟化后的存储服务。但到目前为止，受 Amazon 公司对 S3 的非公开影响，业界对 S3 分布式存储系统的内部原理知之甚少[20]。

Google 公司在搜索领域取得的成功除了它优秀的搜索算法，还依托于其稳定的存储平台。它的 GFS（Google File System）是专为大数据量、大访问量而设计的一款分布式存储系统。截止目前为止，Google 公司已经将 GFS 部署到全球超过 200 个的超大型的数据中心[21]。此外，它们还自行开发了基于 GFS 名为 MapReduce 的计算引擎来处理如索引、网页文字内容和 Google 地图的地理信息等各类数据[22]。

万方数据

开源大数据平台 Hadoop 的分布式文件系统组件 HDFS 参照 GFS 的模式开发，具有高容错、高可靠性、高扩展性和高吞吐率等特点，为海量数据存储提供了分布式的存储平台[23,24]。HDFS 集群采用主-从（Master-Slave）架构，由一个 NameNode 节点和若干个 DataNode 节点组成[25]。NameNode 负责保存并维护数据文件的元数据信息，DataNode 负责存储真正的数据，客户端通过与 NameNode 和 DataNode 的交互进行文件操作。如今 HDFS 作为云计算与大数据平台的重要组件，已应用到雅虎、 Facebook、 Linkedin 和 Twitter 等知名互联网企业的核心应用中，电信和传媒等传统行业也开始应用 HDFS 进行数据存储或作为其他应用平台的基础。

1.2.2 分布式存储容错机制研究现状

在海量数据存储中，数据容错是一项不可或缺的关键技术。目前存在两种比较常见的数据容错技术：一种是多副本策略，如图 1.1 所示，数据文件 Data1 与 Data2 分别以 3 个副本的数量分散存储在 4 台服务器中，即通过将文件复制多份来进行容错，；另一种是纠删码策略，通过编码生成可还原出原文件的校验块来实现容错。Weatherspoon 等人[26]在他们的研究中发现与副本策略相比，纠删码容错技术可以显著降低数据文件对磁盘存储空间的消耗，并且能够提供至少相同或更好的数据容错能力。

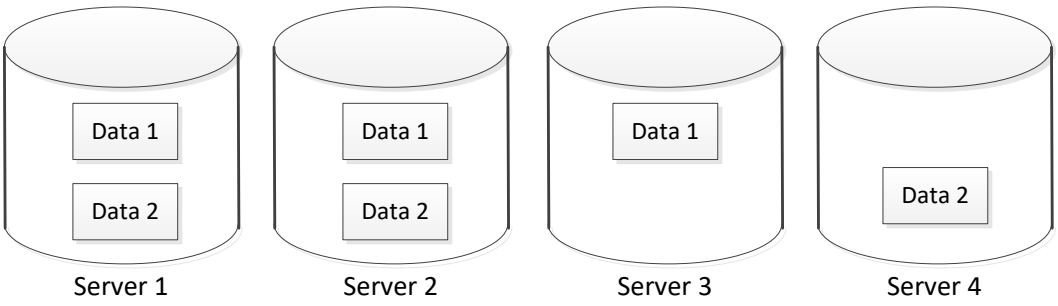


图 1.1 副本策略

目前，国内外关于纠删码容错技术的研究主要是与冗余磁盘阵列（Redundant Array of Independent Disks，RAID）存储系统有关，该存储系统最早由 Patterson，Gibson[27]等人于 1988 年提出。RAID 是将许多磁盘组成一个阵列，当做单一磁盘来对外提供服务。客户端对磁盘阵列的操作是并行的，有效的降低了读写数据的时间，提高了 I/O 效率。但与 RAID 存储系统所采用的固定数据布局和校验布局的策略不同，面向海量数据存储的分布式存储系统通常需要建立在规模庞大的集群之上，这就要求具有相关性的不同类型的数据文件尽量存储在网络距离较近的数据节点上，不同类型的数据文件要能够根据集群的网络拓扑结构采取随机灵活布局。

万方数据

目前，企业里为了分析处理大数据而部署最多的分布式应用就是 Hadoop，且以 2.x 版本最为流行。但该版本 Hadoop 中的 HDFS 在数据容错方面仍采用的是 3 副本模式。国内辽宁大学的宋宝燕[28]团队针对这一方面做了优化，提出了一种基于范德蒙码的 HDFS 优化存储策略，该策略可以节约 HDFS 约 30%的存储开销。而目前基于纠删码存储技术的 HDFS，Hadoop 官方还处于 Beta 测试阶段，但相信再过不久，正式发行版就会向广大开发者公开。

1.3 论文的研究内容

近些年，随着云计算、大数据技术的更新与发展，开源的分布式文件存储系统 HDFS 得到了业界广泛的认可与应用。随着信息的逐年递增，越来越多重要的数据需要依赖 HDFS 来实现存储与管理。如何保证 HDFS 上海量数据的可靠性和可用性是本篇论文思考优化的方向。而当前版本的 HDFS 为了保证系统的高可用性所采用的主-从架构的多副本机制只是刚刚满足了基本的功能需求，在应对主节点单点故障和磁盘空间利用率这两方面还存在着一定的优化空间。

针对上述两个问题，本文主要进行了以下研究：

（1）对于当前 HDFS 版本，系统采取的是对原文件创建 3 个副本的方式来解决由于某些节点失效或故障而造成的数据丢失问题，从而保证容错性。不难看出，当需要存储的数据量很大时，通过创建副本来保证文件存储系统容错性的方式需要消耗大量的底层硬件存储设备。为了提高磁盘存储利用率，本文提出一种基于局部校验纠删码算法的 HDFS 数据存储策略，通过编码生成局部校验块来对原数据文件进行容错。

（2）HDFS 集群采用主-从架构模型，由一个负责管理文件系统的 NameNode 节点和许多用来存储数据的 DataNode 节点组成。NameNode 节点能否正常工作关系到整个分布式文件系统的高可用性。本文分析研究了现有的几种保证 NameNode 高可用的性的方案，针对各个方案的不足与优势，提出了一种扁平化的高可用 NameNode 模型。

（3）最后，基于开源的 Hadoop 安装文件，修改了相应模块的源代码，设计并实现了融合以上两个创新点的原型系统。新系统在数据节点个数的设置上较同为纠删码算法的阵列码更加灵活，在编码与重构原数据时较 RS 码更加高效。同时，扁平化的 NameNode 模型不仅缩短了 HDFS 在 NameNode 节点故障后的崩溃恢复时间，还实现了 HDFS 集群在响应许多客户端请求时更好的负载均衡。

万方数据

1.4 论文章节安排

论文分为六章，每章的主题和主要内容如下：

第一章 绪论。主要介绍了分布式文件系统高可用性的研究背景及研究意义，同时简介了近几年国内外相关研究与实践的发展现状，引出本文的主要工作和创新点。

第二章 相关背景知识介绍。首先介绍了 Hadoop 生态系统的发展现状，简要地描述了其中各个组件的功能。接着着重分析了分布式文件系统 HDFS 的组成架构。最后介绍了计算机系统中高可用性的概念。

第三章 基于局部校验纠删码算法的 HDFS 存储策略。首先介绍了两种保障数据可靠性的冗余策略，分别是备份策略和纠删码策略。接着介绍了一种分布式存储领域中常用的 RS 纠删码算法。并在它们的基础上设计实现了一种适用于 HDFS 的局部校验纠删码算法。最后通过实验，比较分析了每种纠删码算法在编码与重构原数据时的速率。

第四章 扁平化 NameNode 模型。首先指出了采用主-从架构的 HDFS 文件系统所存在的单点故障问题，并介绍了现有的几种解决方案。针对这些方案的优缺点，本章在随后的小节中提出了一种新的解决 HDFS 单点故障隐患的方案——扁平化的 NameNode 模型，并通过实验分析，验证了新方案的优越性。

第五章 实验测试与分析。本章通过虚拟化软件模拟真实集群环境，部署了一个高可用的 HDFS 文件系统，并对系统的关键环节进行了测试与分析。测试效果表明改进版的 HDFS 文件系统能够很好的解决 NameNode 单点故障问题，提升系统的高可用性。

第六章 总结与展望。对本文所做的工作进行了总结，并且对如何进一步提高主-从架构的分布式文件系统的高可用性进行了分析与展望。

万方数据

第二章 相关背景知识介绍

2.1 Hadoop 生态系统简介

Hadoop 最早由 Apache Lucene 创始人 Doug Cutting 开发出来，Lucene 是一个应用广泛的全文搜索系统库[9]。Hadoop 起源于开源的网络搜索引擎 Apache Nutch，它本身也是 Lucene 项目的一部分。Nutch 项目开始于 2002 年，但后来开发者认为这一架构的灵活性不够，不足以解决数十亿网页的搜索问题，后经优化和版本更迭，成了最初的 HDFS[29]。基于这样的文件系统，Nutch 的开发人员又在其上实现了一个 MapReduce 系统来实现文本的搜索与处理功能。

尽管 Hadoop 因计算引擎 MapReduce 及分布式文件存储系统 HDFS 而出名，但经过不断的功能拓展，Hadoop 渐渐变成了大数据处理的一站式平台，后称之为 Hadoop 生态系统[30]。如图 2.1 所示，除了核心组件 MapReduce 和 HDFS 外，Hadoop 生态系统还包含分布式协调服务 Zookeeper、实时分布式数据库 Hbase、数据仓库 Hive、数据挖掘库 Mahout、日志收集工具 Flume 和数据库 ETL 工具 Sqoop。这些项目都使用这个基础平台进行分布式计算和海量数据处理。

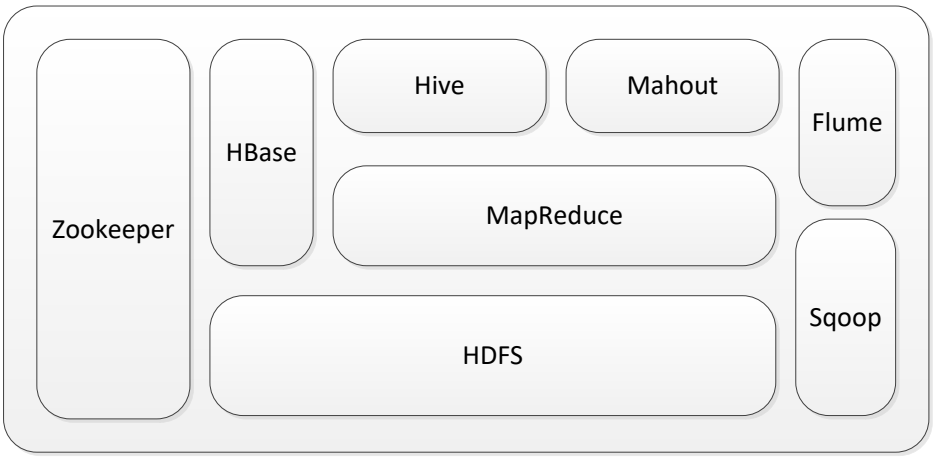


图 2.1 Hadoop 生态系统

万方数据

2.2 HDFS 架构分析

2.2.1 HDFS 架构概述

HDFS 是一个主-从架构的分布式文件系统，主要由一个 NameNode 节点和许多个 DataNode 节点组成。NameNode 节点负责处理客户端的请求和维护存储系统的元数据（metadata）信息[31]。DataNode 节点负责存储实际的数据文件，并且存储在 DataNode 上的数据文件被按照固定大小划分成若干个块（Block）。在 2.x 版本的 Hadoop 中，一个块默认大小为 128MB。为了保证存储在 DataNode 上数据的安全性，任意一个块默认会以 3 个副本的数量保存于不同 DataNode 节点上[32]。

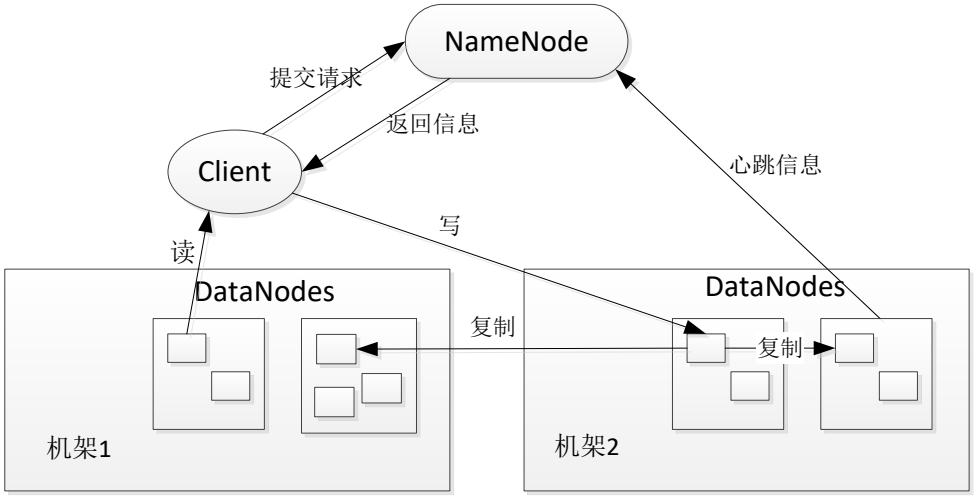


图 2.2 HDFS 架构图

当客户端向 HDFS 提交写文件请求时会首先与 NameNode 建立通信，NameNode 在确认路径合法后向客户端返回确认消息和允许接收上传文件的 DataNode 列表信息。随后，客户端按照一个 Block 块的大小将原文件切分并逐个发送到相应 DataNode 节点上进行保存，并且由接收到 Block 块的 DataNode 负责向其他 DataNode 复制该 Block 块的副本，直至达到系统配置文件中所约定的副本数量为止。

当客户端向 HDFS 提交读文件请求时，首先会将所读文件的路径发送给 NameNode，经 NameNode 验证无误后返回文件的元数据信息，元数据信息中包含着原文件所对应的 Block 块的位置信息。客户端根据返回结果定位到相应的 DataNode 节点并在客户端本地对获取到的 Block 块进行追加合并最终还原出完整的原文件。

2.2.2 HDFS 架构角色介绍

（1）客户端

万方数据

HDFS 客户端作为整个文件系统的接口程序，具有向 HDFS 发送各种文件操作指令的功能，包括创建目录、删除目录、上传文件和下载文件，但是不能修改存储在文件系统上的文件的内容，因为要符合 HDFS 一次写入，多次读取的设计原则。从 hadoop 0.20.2 版本开始，HDFS 的客户端程序既可以运行在集群内的任意节点也可以运行在与集群网络可达的其他互联网设备上。客户端主要是与 NameNode 节点和 DataNode 节点交互来访问 HDFS 文件系统，而存储在 HDFS 上文件的分块大小、副本数量和存储位置则无需客户端程序特别指定。NameNode 与 DataNode 对文件系统的管理细节对客户端保持透明，简化了开发人员操作的复杂性。

（2）NameNode 节点

NameNode 节点除了要与客户端交互以外还负责维护整个分布式文件系统的目录树。目录树信息又称元数据信息，它记录着每个文件中各个块信息以及块所在数据节点的信息。元数据以两个文件的形式保存在本地磁盘，如图 2.3 所示，一个是命名空间镜像文件 Fsimage，一个是客户端操作文件系统时的编辑日志文件 Edits。NameNode 进程在每一次启动后，会先将镜像文件 Fsimage 加载至内存。而此时，数据节点会以心跳的方式不断向 NameNode 上报各自所持有的 Block 文件块的信息。NameNode 整合镜像文件与 Block 块信息，最终在本机内存中生成一份完整的 HDFS 目录树[33]。图 2.3 为 NameNode 的目录树结构。除了上面介绍到的 Fsimage 文件与 Edits 文件外，Version 文件是 HDFS 的版本文件，记录了持久化到 HDFS 上的数据的版本信息。Fstime 文件则记录了该 NameNode 每次做 checkpoint 的时间戳。

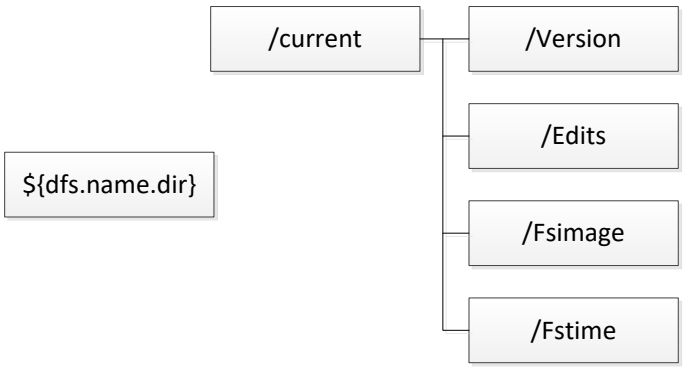


图 2.3 NameNode 目录树结构

（3）Secondary NameNode 节点

主-从架构的 HDFS 集群除了由一个 NameNode 节点与众多 DataNode 节点组成以外，在运行着 NameNode 进程的服务器节点上还存在着一个伴生进程——Secondary NameNode 进程。

Secondary NameNode 会周期性地从 NameNode 上下载 Fsimage 镜像文件和客户端的操作日志文件 Edits。Fsimage 镜像这是一份将 NameNode 内存中元数据信息持久化到磁盘上的本地文件。Secondary NameNode 会回滚 Edits 日志中的操作行为来更新 Fsimage 镜像文件，最

万方数据

终生成一份新的镜像文件，并回传给 NameNode。

（4）DataNode 节点

HDFS 中真正存储数据的地方是 DataNode，它会以块的形式保存客户端上传来的文件，块的大小可以通过修改集群配置文件来确定，在 Hadoop 2.x 版本中，一个块默认为 128MB。块的大小虽然能自定义，但不宜过大也不能太小，需要参考承载集群运行的硬件的配置。

一个 HDFS 集群可以含有少则几个多则上千的 DataNode 节点，当集群中 DataNode 数量十分庞大时，通常会将一定数量的 DataNode 编组放入一个机架，若干个机架组成一个集群，机架之间通过交换机相连。在生产环境中，DataNode 与 NameNode 都是部署在运行着 Linux 操作系统的服务器上，并运行于 Java 虚拟机中，这也使得 HDFS 具有很好的可移植性。

2.3 高可用性概述

一个高可用的系统可以通过合理的软件设计和硬件配置来避免宕机事件而导致的服务不可用状况。在评判一个系统是否具有高可用性时，通常要参考两个重要指标：平均无故障时间 MTTF（Mean Time To Failure）和平均修复时间 MTTR（Mean Time To Repair）[34]。平均无故障时间 MTTF 是指系统平均正常运行多长时间就会有可能发生一次故障，而平均修复时间 MTTR 是指在发生故障后系统平均需要多长时间修复才能又恢复平稳正常运行状态。由此可见，MTTF 值越高，表明系统的可靠性越好，而 MTTR 值越低，则说明系统的可维护性越高。MTTF 与 MTTR 之间的关系如图 2.4 所示。

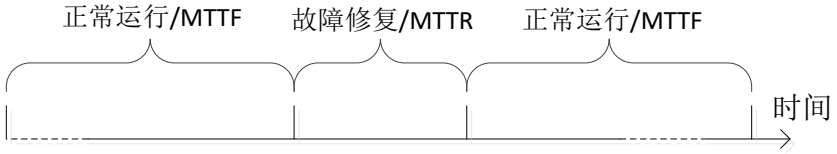


图 2.4 系统运行状态图

MTTF 与 MTTR 可以通过公式 2.1 计算出系统正常运行时间占系统总运行时间的百分比，这个结果就能够反应出系统的可用性。

$$\frac{\mathrm{MTTF}}{\mathrm{MTTF}+\mathrm{MTTR}}\times 100\% \tag{2.1}$$

理论上，不管怎样提升元器件的稳定性与可靠性，电子元器件都会出现老化的现象，而作为由电子元器件组成的计算机硬件系统就会存在使用寿命问题[35]。换而言之，任何一个计算机硬件系统在长期使用过程中都会产生故障。而发生故障的硬件机器将直接影响到部署在其上的软件服务。为了保证承载关键业务系统的硬件平台在发生故障后，将其对上层软件系

万方数据

统的影响降到最低，使软件系统依旧能稳定持续地对外提供服务，计算机界提出了高可用（HA，High Availability）的概念。

随着信息技术的发展，高可用性逐渐成为了一个广泛的概念。在底层硬件层面，抑或是操作系统层面，中间件层面均有成熟的高可用性方案。虽然各层的高可用性方案的实现方法上不同，但它们的核心思想是相通的。

2.4 本章小结

本章介绍了本文所涉及的相关领域与理论知识。首先介绍了 Hadoop 生态系统的发展与现状，并简要地描述了其中各个组件的功能。接着着重分析了分布式文件系统 HDFS 的组成架构。最后介绍了计算机系统中高可用性的概念。

万方数据

第三章　基于局部校验纠删码算法的 HDFS 存储策略

3.1 研究思路

在海量数据的分布式存储中，数据的可靠性是业界长期关注与研究的方向，而冗余策略一直都是保证数据可靠性最直接也是最重要的方法。广泛使用的数据冗余策略包括副本策略和纠删码策略。副本策略实现简单，在需要读取原始数据时响应速度快，但是较为占用存储空间。在数据量庞大的云存储中心势必会增加几倍的硬件投入成本[36]。为了保证在数据访问性能不降的情况下，同时能降低存储开销，基于纠删码的存储技术开始得到重视与推广。

本章着重研究了经典的 RS 纠删码的编码与译码过程，并在其基础上设计实现了一种能够代替 HDFS 副本策略的数据容错算法——局部校验纠删码算法（Locally Repairable Code，LRC）。该算法相较于 HDFS 传统的副本策略能够显著地降低磁盘的存储开销，而在对失效数据的重构过程又不需要像 RS 编码一样从各个网络节点中拉取所有剩余数据，降低了网络带宽的占用，提高了数据恢复的效率。

3.2 典型纠删码算法简介

纠删码作为一种纠错技术，最早源于通信领域，主要是为了解决数据在网络传输中的检错与纠错问题，后来被应用到存储领域并被逐步认识与推广[37]。纠删码技术主要是通过对原始数据进行编码得到冗余校验数据，以达到容错的目的。其基本思想是通过编码矩阵（又称为生成矩阵），对 k 块源数据进行矩阵运算，得到 m 块冗余校验数据。对于这 k+m 块数据文件，当其中任意不超过 m 块数据文件发生丢失或损坏情况时，均可通过相应的重构算法恢复出那 m 块数据文件[38]。经过长期的理论研究与实际操作，如今的纠删码算法已十分丰富。

基于冗余磁盘阵列的阵列码（Array Codes）可以说是目前应用于存储领域中较为广泛的纠删码算法，它的编码与重构过程只需要进行简单的异或运算，软硬件实现简单，计算复杂度低，也是最早被应用于存储领域的纠删码算法。阵列码可以分为横式阵列码和纵式阵列码，横式阵列码中具有代表性的纠删码算法为 EVENODD 码，而纵式阵列码中比较具有代表性的纠删码算法有 X 码。

除了上面所述的阵列码之外，基于范德蒙德矩阵的 RS（Reed-Solomon）编码也是经典的纠删码算法，而为了进一步降低范德蒙德 RS 编码的计算复杂度，提高 RS 码在编码与重构原

万方数据

数据时的效率，学术界又提出了改进的柯西 RS 编码[39,40]。

本节将对上述 4 种纠删码算法做简要介绍。

3.2.1 名词定义与说明

对于存储系统中有关纠删码的相关概念目前还没有一个统一的定义。为了方便后续阅读理解，本节将对一些在纠删码研究领域中常出现的术语做简要说明和定义：

（1）元素（Symbol）

元素是一定量的连续的数据信息或者冗余信息，它是编码理论中信息存储的最小单位。

（2）条带（Stripe）

一个 RAID 存储系统通常在逻辑上被划分为许多个条带，一个条带会跨越所有的数据磁盘和校验磁盘，它是经由一种纠删码算法计算所得的所有信息的集合，相当于算法的一个实例。条带与条带之间相互独立。

（3）分条（Strip）

处于单个磁盘上并属于某一个条带的数据集合称之为一个分条。一个条带上的每个分条与存储系统中的每个存储设备一一对应。分条的大小并不固定，由分条中所包含的元素大小和元素数目决定。

元素、分条和条带之间的关系如图 3.1 所示：

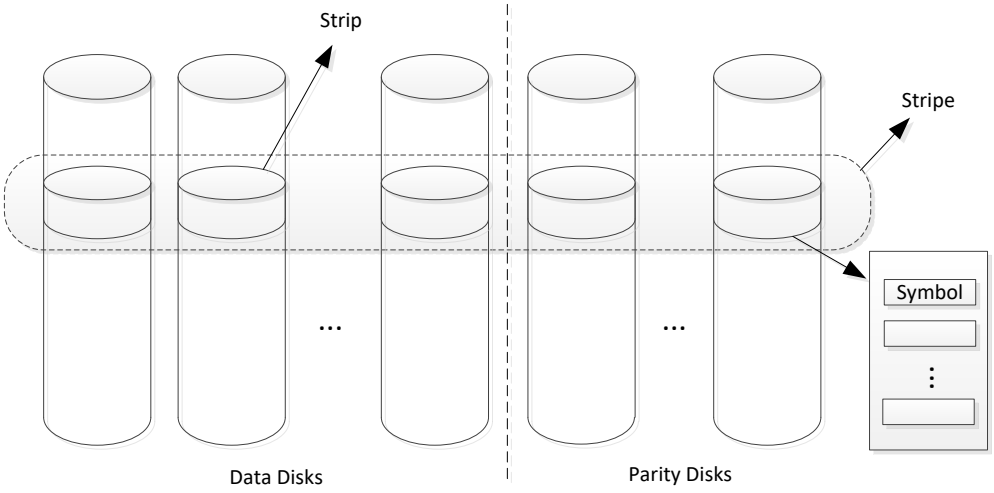


图 3.1　元素、分条和条带之间的关系图

（4）系统码（Systematic Code）

在编码理论中，系统码是指编码后的元素中包含原信息序列。换言之，存在 k 块原数据文件，经编码后生成 k+m 块数据文件，其中多出来的 m 块为校验块数据，而原始的 k 块数据

万方数据

的内容并不随编码而改变[41]。与之相反的就是非系统码，原始数据经过编码后将隐含在冗余数据中，需通过解码才能还原出来。系统码适用于对原始数据访问频率较高的存储系统。

（5）MDS 码（Maximum Distance Separable Codes）

MDS 码即最大距离分割码。如果 k 块原始数据通过编码算法计算产生 k+m 个编码分块中有任意 m 份编码分块出错时，均可通过相应的重构算法恢复出原始的 k 个数据块，纠错能力上限 $m \leqslant d_{min}-1$，其中 d_{min} 为纠删码的最小列距，当 $d_{min}=m+1$ 时，该纠删码就称为 MDS 码[42]。MDS 码可以从 k+m 个编码分块中任意选择 k 个分块，通过解码算法重构出 k 个原始数据块，而非 MDS 码需要多于 k 个编码数据块或者仅特定一些而非任意 k 个编码数据块的子集才能重构出原始数据。

3.2.2 EVENODD 编码

横式阵列码通常情况下能容忍数据块失效的个数不是很多，一般为两个。EVENODD 码是最具有代表性的横式阵列码[43]。它的原始数据与校验数据可以用一个(n-1)×(n+2)大小的阵列来表示，阵列的前 n 列存放原始数据，后 2 列存放校验数据，n 规定要是一个大于 2 的质数。

EVENOOD 码编码时，会将每一列原始数据分成 n-1 块，第一列校验数据按如下公式计算：

$$C_{\mathrm{i},0}=\sum_{t=0}^{p-1} D_{i,t} \quad (i=0,1,\ldots,p-2) \tag{3.1}$$

相比于第一列校验数据只需要对每个条带中的分条进行求和运算，第二列校验数据的构造过程稍显复杂，计算公式如下：

设

$$S=\sum_{t=0}^{p-1} D_{t,p-1-t} \tag{3.2}$$

则

$$\mathrm{C}_{i,1}=S+\sum_{t=0}^{p-2} D_{t,(i-t)\ mod\ p} \quad (i=0,1,\ldots,p-2) \tag{3.3}$$

图 3.2 演示了 n=5 时，EVENOOD 码的编码过程：

万方数据

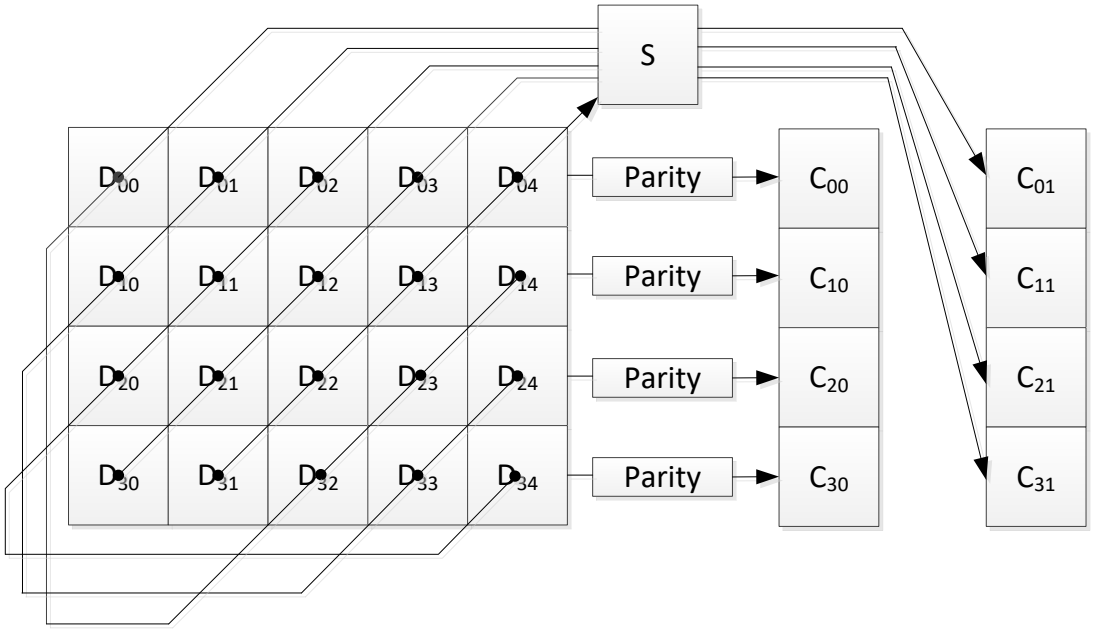


图 3.2　n=5 的 EVENOOD 码的编码示意图

从几何层面上分析，S 是第 n 列数据块在斜率为 1 的方向上所有数据块异或运算的结果。第一列校验数据中的每个校验块分别由各自对应条带上的分条数据异或计算得到，而第二列校验数据中的每个校验块是各自对应斜率为 1 的方向上的所有数据块再与 S 异或的结果。

EVENODD 码的解码过程也并不复杂，假设发生最坏情况，即出现了两列磁盘的数据因为硬件故障而发生丢失。由编码过程可以推导出至少存在一个斜率为 1 的对角上的数据只丢失 1 份，所以可以通过剩余对角数据和第二列校验数据中相应的校验块来恢复出丢失的 1 份数据。接着会出现存在某一行只丢失 1 份数据的情况，同样可以通过该行中剩余的数据块与第一列校验数据中对应的校验块来恢复出来。重复上述过程即可重构出所有丢失的数据。

3.2.3 X 编码

X 码是一种纵式阵列码，它同时也是 MDS 码。与横式阵列码不同的是，在纵式编码中每个条带的分条中既包含了原始数据元素，又包含了校验元素。在 X 码中，数据元素被置于一个(n-2)×n 的二维阵列中，而校验元素则在阵列的最后两行。校验元素的生成过程类似于其他阵列码，由某些行、列或者对角线上数据元素相异或而得。当矩阵中任意两列不可用时，可以通过剩下的 n ×(n-2)来重构原始数据。

编码时，假设第 i 行，第 j 列的原始数据元素用 $D_{i,j}$ 表示，倒数第二行的校验元素用 $C_{n-2,i}$ 表示，最后一行的校验元素用 $C_{n-1,i}$ 表示，则 X 码的校验元素可以通过下面公式生成：

$$C_{n-2,i} = \sum_{k=0}^{n-3} D_{k,(i+k+2)\ mod\ n}\ (i = 0,1,\dots,n-1) \tag{3.4}$$

万方数据

$$C_{n-1,i} = \sum_{k=0}^{n-3} D_{k,(i-k-2)\ mod\ n}\ (i = 0,1,\dots,n-1) \tag{3.5}$$

用表的形式来表示 n=5 的 X 码的编码过程如表 3.1 所示：

表 3.1　n=5 的 X 码的编码

$D_{0,0}$	$D_{0,1}$	$D_{0,2}$	$D_{0,3}$	$D_{0,4}$
$D_{1,0}$	$D_{1,1}$	$D_{1,2}$	$D_{1,3}$	$D_{1,4}$
$D_{2,0}$	$D_{2,1}$	$D_{2,2}$	$D_{2,3}$	$D_{2,4}$
$D_{0,2}+ D_{1,3}+ D_{2,4}$	$D_{0,3}+ D_{1,4}+ D_{2,0}$	$D_{0,4}+ D_{1,0}+ D_{2,1}$	$D_{0,0}+ D_{1,1}+ D_{2,2}$	$D_{0,1}+ D_{1,2}+ D_{2,3}$
$D_{0,3}+ D_{1,2}+ D_{2,1}$	$D_{0,4}+ D_{1,3}+ D_{2,2}$	$D_{0,0}+ D_{1,4}+ D_{2,3}$	$D_{0,1}+ D_{1,0}+ D_{2,4}$	$D_{0,2}+ D_{1,1}+ D_{2,0}$

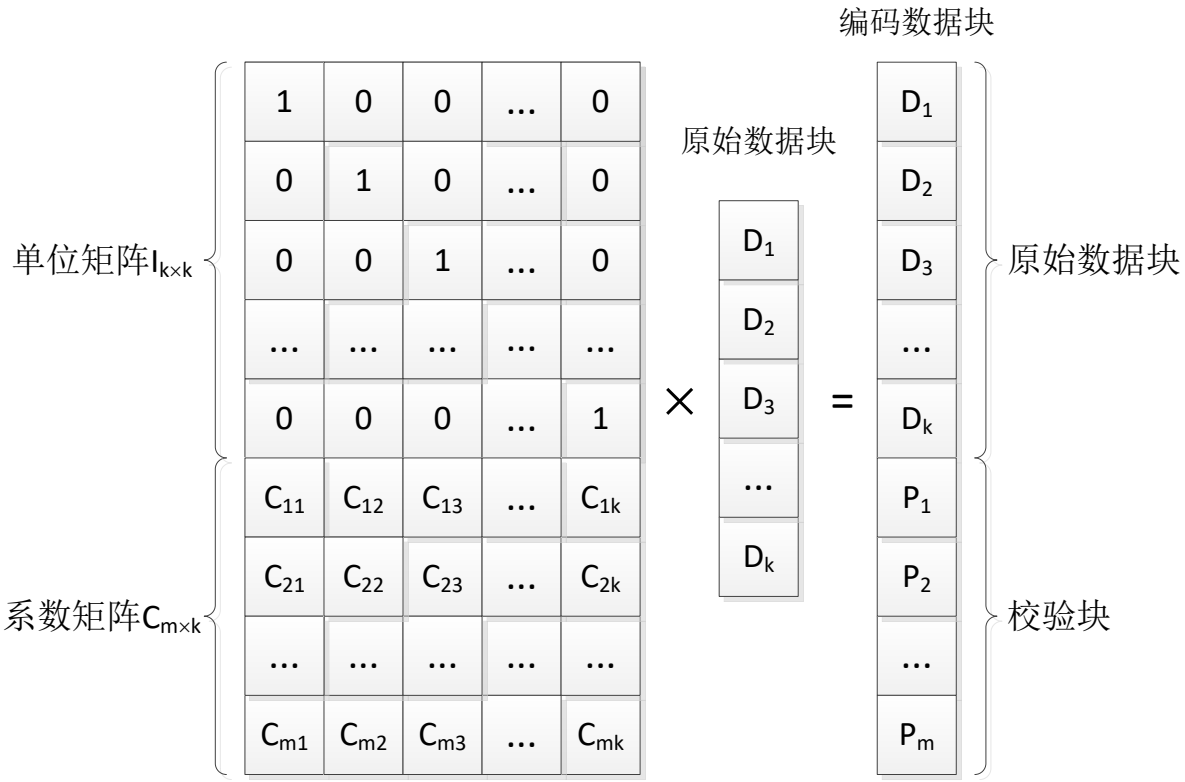
X 码与 EVENODD 码的解码过程类似，当数据阵列中出现不超过两列数据丢失的情况时，至少存在一个斜率为 1 或者为-1 的对角线上的数据只丢失一块，这样一来，即可通过剩余数据块与冗余块来重构出丢失的数据。

3.2.4 范德蒙德 RS 编码

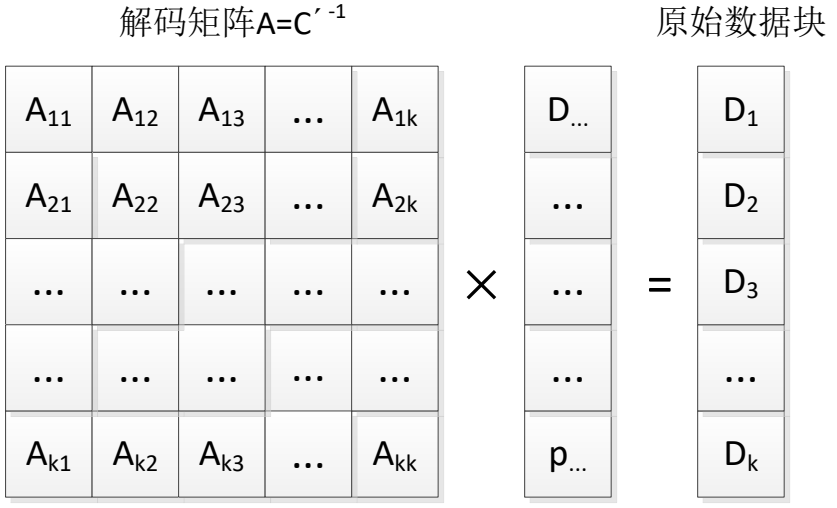
RS 纠删码最早由里德（Reed）和所罗门（Solomon）提出，故用他们的名字命名了该纠删码算法。经过多年的研究与完善，RS 纠删码已经发展成为一种可以满足任意磁盘数目（n）和冗余磁盘数目（m）的 MDS 码。RS 纠删码通常分为两类，一类是范德蒙德 RS 编码，另一类是柯西 RS 编码。顾名思义，范德蒙德 RS 编码使用的编码矩阵是范德蒙德矩阵，而柯西 RS 码使用的编码矩阵是柯西矩阵。

RS 纠删码的基本编码原理就是利用编码矩阵与数据列向量进行矩阵相乘，计算得到原始数据块和冗余校验数据块的过程。当发生个别数据块失效时，其重构过程也是通过未出错数据块所对应的残余编码矩阵的逆矩阵与未出错数据的列向量相乘来重构出失效数据[44]。图 3.3（a）展示了 RS 纠删码的编码过程，而图 3.3（b）展示了 RS 纠删码重构原数据的过程。需要注意的是在设计编码矩阵时，未出错数据块所对应的剩余编码矩阵需要在 GF（2^w）域上满足可逆条件，换句话讲，数据能够重构成功的必要条件是编码矩阵的任意 k 个行向量都要在 GF（2^w）域上是线性无关的。

万方数据



（a）



（b）

图 3.3　RS 编码的编码和重构过程

在数学上应用广泛的范德蒙德矩阵正好能够满足 RS 编码在编码和重构过程中对编码矩阵提出的要求。范德蒙德 RS 编码矩阵的表现形式为矩阵的前 k 行为单位矩阵，后 m 行为线性无关的范德蒙德矩阵。范德蒙德 RS 编码的编码时间复杂度为 O(km)，解码时间复杂度为 $O(m^3)$。

万方数据

3.2.5 柯西 RS 编码

在 GF（2^w）域上，加法操作实际上是通过异或运算实现的，而乘法则复杂得多，一般需要借助离散对数运算或者查表才能降低运算复杂度。因此，传统的基于范德蒙德矩阵的 RS 编码的计算开销还是稍显大。为了改善计算复杂度，提高 RS 编码的计算效率，后续又提出一种基于柯西矩阵的 RS 编码矩阵。

柯西 RS 编码矩阵将有限域中的每个数表示成一个二维位矩阵（bit matrix），使得有限域上的乘法运算转换成了只包含异或操作的简单运算，从而提高了运算效率。公式 3.6 为柯西 RS 编码的编码过程。公式 3.7 为在丢失数了据块 D_1 后，柯西 RS 编码重构原数据的过程。

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \frac{1}{x_1+y_3} & \cdots & \frac{1}{x_1+y_k} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \frac{1}{x_2+y_3} & \cdots & \frac{1}{x_2+y_k} \\ \vdots & \vdots & \vdots & & \vdots \\ \frac{1}{x_m+y_1} & \frac{1}{x_m+y_2} & \frac{1}{x_m+y_3} & \cdots & \frac{1}{x_m+y_k} \end{bmatrix} \times \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \end{bmatrix} = \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \\ P_1 \\ P_2 \\ \vdots \\ P_m \end{bmatrix} \tag{3.6}$$

$$\begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \frac{1}{x_1+y_3} & \cdots & \frac{1}{x_1+y_k} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \frac{1}{x_2+y_3} & \cdots & \frac{1}{x_2+y_k} \\ \vdots & \vdots & \vdots & & \vdots \\ \frac{1}{x_m+y_1} & \frac{1}{x_m+y_2} & \frac{1}{x_m+y_3} & \cdots & \frac{1}{x_m+y_k} \end{bmatrix}^{-1} \times \begin{bmatrix} D_2 \\ D_3 \\ \vdots \\ D_k \\ P_1 \\ P_2 \\ \vdots \\ P_m \end{bmatrix} = \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \end{bmatrix} \tag{3.7}$$

3.3 局部校验纠删码存储策略

尽管基于冗余磁盘阵列的阵列码算法能够为数据提供简单、高效、可靠的存储保证，但无论是 X 码还是 EVENODD 码，对于磁盘数目 n 都有着非常严格的限制，n 必须是一个大于 2 的素数，才能保证在数据阵列中任意 2 列数据丢失时通过残余数据重构出所有数据。这不免造成阵列码可扩展性较差的问题。而分布式文件系统 HDFS 通常包含少则十几多则上千的数据节点，硬件故障率随着集群规模的扩大而提高。下线故障节点，上线新的节点是经常发

万方数据

生的事情，很难保证集群中正常工作的节点个数始终是一个素数。所以，能否满足集群的高可扩展性成了选用哪种纠删码算法的重要标准之一。RS 纠删码恰好符合这样的标准，它对集群中的节点个数没有太严格的要求，但 RS 码也并不是完美的解决方案，与 EVENODD 码类似，RS 码在重构原数据的过程中需要通过网络汇集条带中的所有的分条数据。以 RS(6,3)编码为例，原始数据块的有 6 个，通过这 6 块原始数据块又生成了 3 块校验块，合计 9 个数据块。当出现这 9 个数据分块中任意不超过 3 个的数据分块丢失或错误的情况时，都需要汇总剩余所有数据块来进行重构，网络传输限制了 RS 码性能的进一步提高。

为了减少 RS 码在重构单个数据块时从不同存储服务器上拉取的剩余数据分块的个数，降低重构失效数据块的代价，以 RS 编码为基础，本章设计并实现了一种适用于 HDFS 的局部校验纠删码算法 HDFS-LRC。

HDFS-LRC（k，m，g）编码将一个文件按照固定大小划分为 k 个数据块，并以此计算出 m 个全局校验分块。k 个数据块将被分为 g 组，每组会计算得出一个局部校验分块，一共会生成 g 个局部校验分块。这里定义$\frac{k+m+g}{k}$为纠删码的空间冗余率，经过 HDFS-LRC（k，m，g）编码后的文件一共会产生 k+m+g 个文件块。如图 3.5 所示，HDFS-LRC（6，2，2）的空间冗余率为 1.67，RS(6,2)的空间冗余率为 1.3，而三副本冗余策略的空间冗余率则为 3。

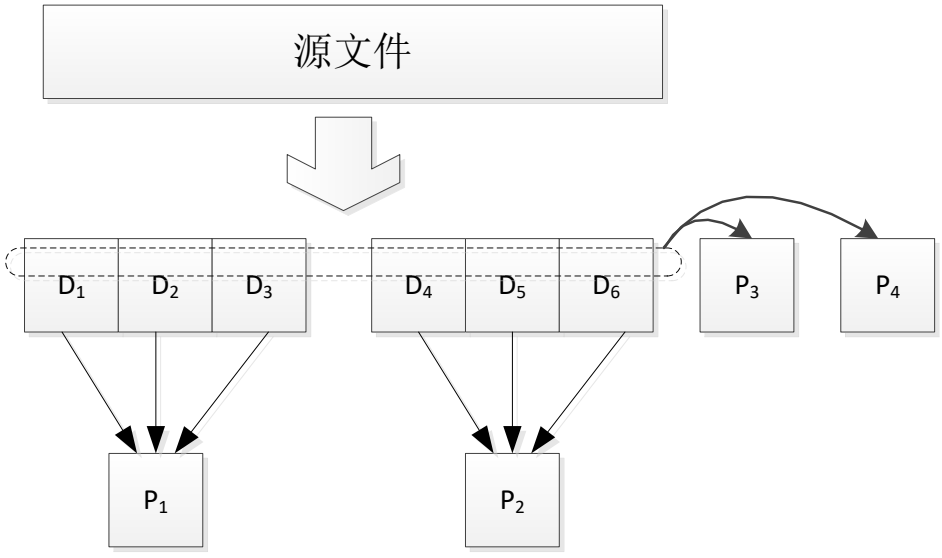


图 3.5　分组校验纠删码

LRC 与 RS 的不同在于 RS 只有数据块和全局校验块两种类型的文件块，而 LRC 除了这两种文件块外，还多了局部校验块。局部校验快和全局校验块的不同在于局部校验块是由部分数据块与编码矩阵运算所得到，只对组内数据块进行容错与恢复，而全局校验块是用所有的数据块与编码矩阵运算得到，对所有数据块进行容错与恢复。以 HDFS-LRC（4，2，2）为例，该纠删码将原文件切分为 4 个数据块，每两个数据块为一组共两组。假设 4 个数据块分

万方数据

别为 D_1、D_2、D_3 和 D_4，其中 G_1 组包含数据块 D_1 和 D_2，G_2 组包含数据块 D_3 和 D_4。则转化矩阵 M 的构造如下：

$$M=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\\ a_{11} & a_{12} & 0 & 0\\ 0 & 0 & a_{23} & a_{24}\\ b_{11} & b_{12} & b_{13} & b_{14}\\ b_{21} & b_{22} & b_{23} & b_{24}\end{bmatrix} \tag{3.8}$$

其中$A=(a_{ij})\ 1\le i\le g,\ 1\le j\le n$为局部编码矩阵，$B=(b_{ij})\ 1\le i\le m,\ 1\le j\le n$为全局编码矩阵。

HDFS-LRC（4，2，2）编码过程实际上是一个关于数据分块的线性运算过程。编码矩阵与数据分块相乘得到的结果就是校验分块。局部校验纠删码编码过程如公式 3.9：

$$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\\ a_{11} & a_{12} & 0 & 0\\ 0 & 0 & a_{23} & a_{24}\\ b_{11} & b_{12} & b_{13} & b_{14}\\ b_{21} & b_{22} & b_{23} & b_{24}\end{bmatrix}\times\begin{bmatrix}D_1\\ D_2\\ D_3\\ D_4\end{bmatrix}=\begin{bmatrix}D_1\\ D_2\\ D_3\\ D_4\\ P_1\\ P_2\\ P_3\\ P_4\end{bmatrix} \tag{3.9}$$

其中 P1、P2 为两个局部校验块，P3、P4 为两个全局校验块。

为了保证解码矩阵可逆，全局编码矩阵 B 通常选择范德蒙德矩阵或柯西矩阵。

3.4 实验对比测试

3.4.1 编码效率测试

编码时间除了与纠删码算法有关外，还与文件的大小，磁盘的读写速度相关，为了限定这些影响因素，实验采取在同一服务器环境下对内容为循环写入“helloworld”单词，大小分别为 100MB、200MB、300MB 和 500MB 的文本文件进行编码测试来比较 HDFS-LRC 编码算法、范德蒙德 RS 编码算法和柯西 RS 编码算法的编码效率。为了保证相同的容错性能，各纠删码算法的参数选择分别为 HDFS-LRC（k=6，m=2，g=2）、Vandermonde-RS（k=6，m=3）、Cauchy-RS（k=6，m=3）。

万方数据

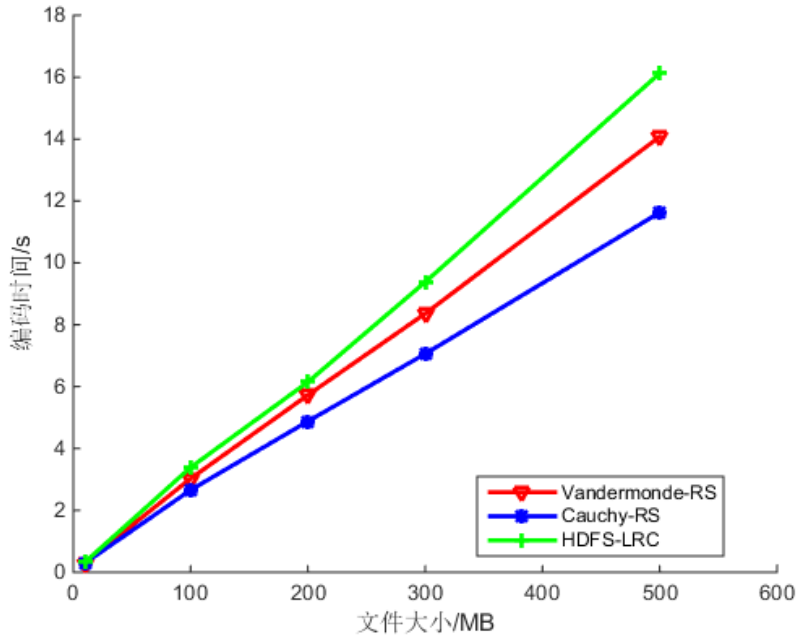


图 3.6　编码时间

在图 3.6 中，图的横坐标为文件大小，纵坐标为编码时间。从图中可以看出三种纠删码算法在对文件编码时所消耗的时间基本随文件大小的增大而呈线性增长趋势。柯西 RS 编码具有较高的编码效率，在每一轮的测试中均用时最短。通过用原始文件大小除以编码时间计算出柯西 RS 编码的平均编码速度约为 42MB/s，RS 编码速度为 36MB/s，而 HDFS-LRC 算法维持在 32MB/s 左右。考虑到 HDFS 的设计初衷为一次写入，多次读写，虽然 HDFS-LRC 算法在编码阶段速度不及另外两种算法，但是数据一旦写入文件系统，数据内容就不能被修改，接下去要面临的是长期存储在集群服务器上的个别数据块可能会因为一些意外状况而导致被损坏或者丢失。能不能保证数据的安全和对失效数据重构的速度是考量纠删码算法更为重要的一方面。

3.4.2 重构效率测试

当集群中某个存储服务器发生故障导致个别数据分块失效时，需要对残余数据进行重构来恢复出原始数据，即解码过程。解码速度主要由网络带宽、磁盘访问速度和译码速度决定。为了更好的比较几种算法之间的译码速度，所有的数据均放置在同一台服务器上。与编码测试稍有不同，在进行解码速度测试之前，先删除一个原始数据分块来模仿数据失效的情形，实验开始时将读取丢失的数据分块，通过计算最终读取到原数据所消耗的时间来比较每种算解码的快慢。

万方数据

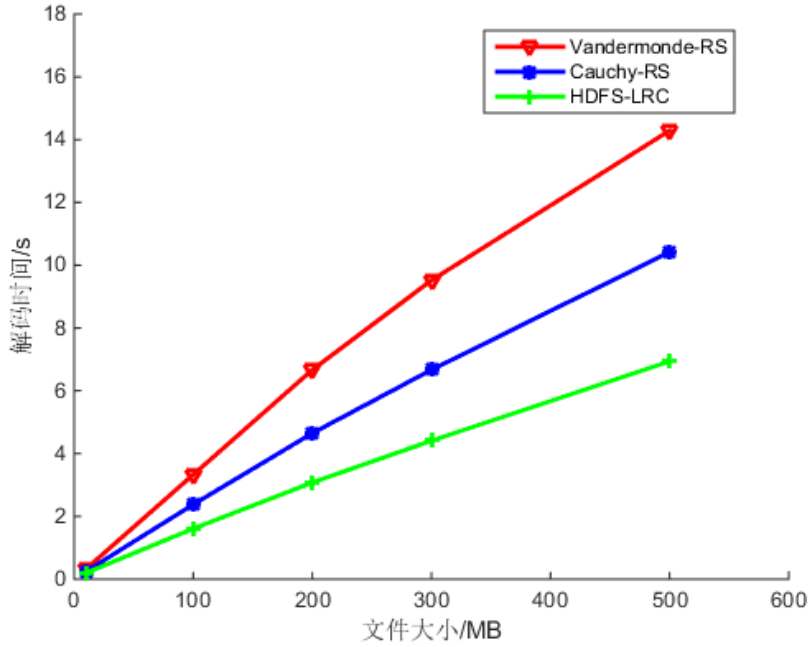


图 3.7　单个数据块丢失文件重构时间

由图 3.7 所示，范德蒙德 RS 编码在单个数据块丢失后重构原数据所耗费的时间最长。柯西 RS 编码相较于范德蒙德 RS 编码，有一定的优化。而得益于分组校验思想的 HDFS-LRC 算法由于在重构原数据时不需要从各个网络节点上拉取所有剩余文件分块，从而获得了更高的效率。

3.5 本章小结

本章先介绍了两种当前最常用的保障数据可靠性的冗余策略，分别是备份策略和纠删码策略。并解释了为什么云计算、大数据领域在底层数据存储方面更适合采用基于纠删码的数据冗余策略。接着介绍了一种在分布式存储领域中常用的纠删码算法——RS 纠删码。根据编码矩阵的不同，它又分为范德蒙德 RS 编码和柯西 RS 编码，并陈述了其各自的优缺点，并在它们的基础上设计实现了一种 HDFS-LRC 局部校验算法。该算法可以在分组中的某一个数据块失效后，直接通过局部校验块重构出失效的数据块，而当某一个分组内同时出现两个数据块丢失的情况时，该算法退化成普通的 RS 纠删码算法。本章的最后通过实验，比较分析了这三种算法在编码与重构原数据时的速率。

万方数据

第四章 扁平化的高可用 NameNode 模型

4.1 研究思路

HDFS 系统的高可用性可以理解为系统对外提供正常服务的能力[45]。由于在 HDFS 中元数据节点 NameNode 只有一个，而这一个 NameNode 节点既要处理来自客户端的所有读、写请求，又要承担集群中元数据的维护管理任务。这就可能存在因为用户的误操作或设备的故障而引起的单点故障问题，这将严重影响到系统的高可用性，使 HDFS 系统无法对外提供正常的服务[46]。

由于 HDFS 处于 Hadoop 的底层，其上层的分布式应用如 Hbase、Hive、Mahout 等都是建立在 HDFS 底层系统之上的。因此 HDFS 的可用性将对这些分布式应用的可用性构成直接影响，并最终影响到最上层云应用服务高可用性。在实际使用中，对于一个基于 Hadoop 的云计算系统来说，在大多数情况下都需要考虑其 HDFS 系统的高可用性问题。

为了解决这个问题，本章提出了一种扁平化的高可用 NameNode 模型，该模型不仅有效地解决了集群的单点故障问题，还实现了 NameNode 服务器处理客户端读请求时各节点的负载均衡，提升了系统整体性能。

4.2 HDFS 单点隐患解决方案

HDFS 系统的高可用性可以理解为系统对外提供正常服务的能力。NameNode 是否能保持长时间的正常工作，关系到整个分布式文件系统的可用性。行业中针对 NameNode 潜在的单点故障问题而采取的解决方案大致有 3 种，分别是 Secondary NameNode 机制[47]、Backup Node[48]机制和 Avatar 机制[49]。

4.2.1 Secondary NameNode 方案

构成HDFS的各种角色中，Secondary NameNode从命名方式上看会让人联想到NameNode的备份，但其实不然。为了保证查询效率，HDFS 的文件目录树是保存在 NameNode 的内存中的。为了防止生产环境下的突然断电而导致内存中数据的丢失，NameNode 会持久化一份内存中文件目录树的镜像文件到本地磁盘上作为备份，但备份并不是实时的，而内存中的目录树会随客户端对文件系统的操作而自动更新。为了保证 NameNode 在重启后所读入的本地

万方数据

磁盘上的镜像文件是一份能完整还原出集群关闭前真实状态的镜像文件，就需要借助 Secondary NameNode。Secondary NameNode 会定期地从 NameNode 上下载 Fsimage 镜像文件和 edits 编辑日志，并依据 edits 文件中的操作日志更新生成一份新的镜像文件，最后回传给 NameNode，并替换旧的镜像文件，整个过程如图 4.1 所示。这一过程就叫做 Checkpoint[50]。

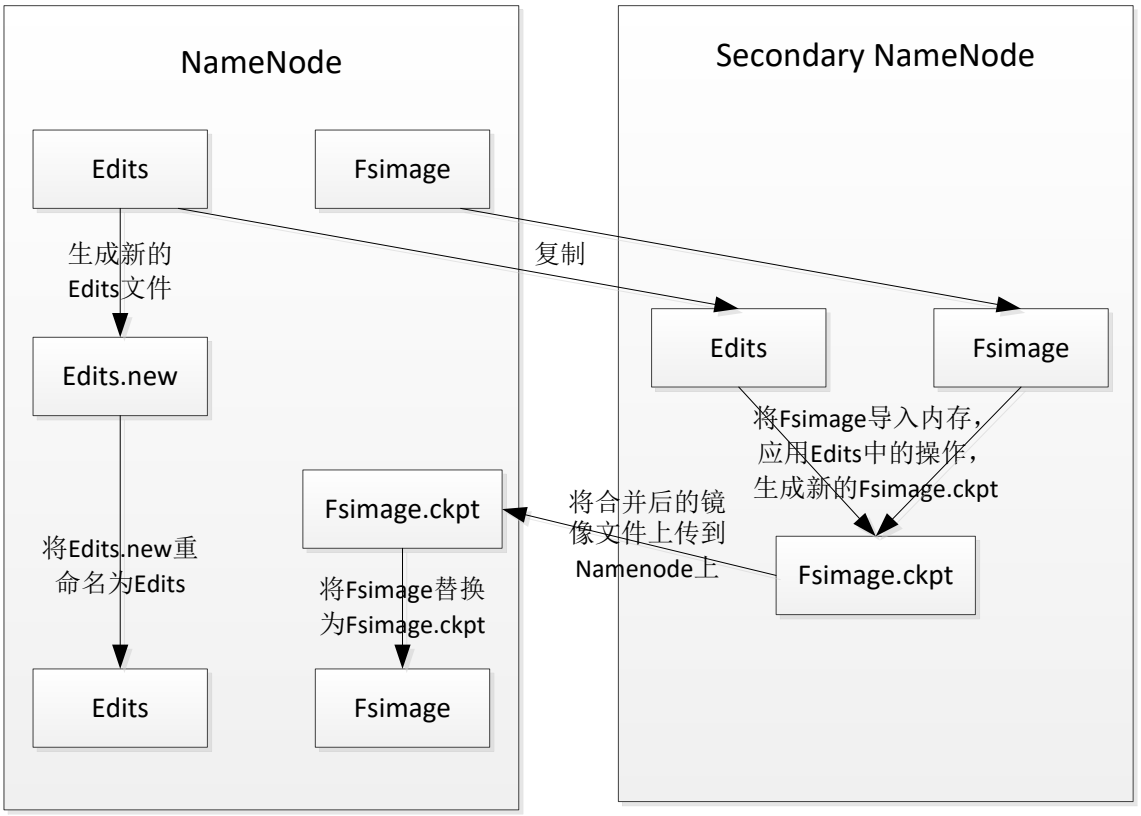


图 4. 1 Checkpoint 过程

Checkpoint 是 HDFS 的自带机制，成熟可靠，无需开发也无需配置，系统在启动后默认定时执行，减少了 NameNode 启动所需时间。但 Checkpoint 过程得到的元数据的镜像也只是准完整的，而且随着 Checkpoint 时间变长，数据丢失的风险也会加大。

4.2.2 Backup Node 方案

与 Secondary NameNode 方案不同，Hadoop 的 Backup Node 方案在防范主节点单点故障时，另外设置了一台独立的备份节点。备份节点会在自己的内存中维护着一份从 NameNode 上同步过来的 fsimage 镜像文件，同时实时地从 NameNode 接收 Edits 文件的日志流，并把它们持久化到磁盘，定期性地做 Checkpoint。这样，在备份节点的内存中就会维护一份与 NameNode 一样的元数据信息。当 NameNode 无法正常对外提供服务时，可以启动备份节点接管之前 NameNode 相应的服务，维持整个文件系统的可用性。Backup Node 方案的系统原理如图 4.2 所示。

万方数据

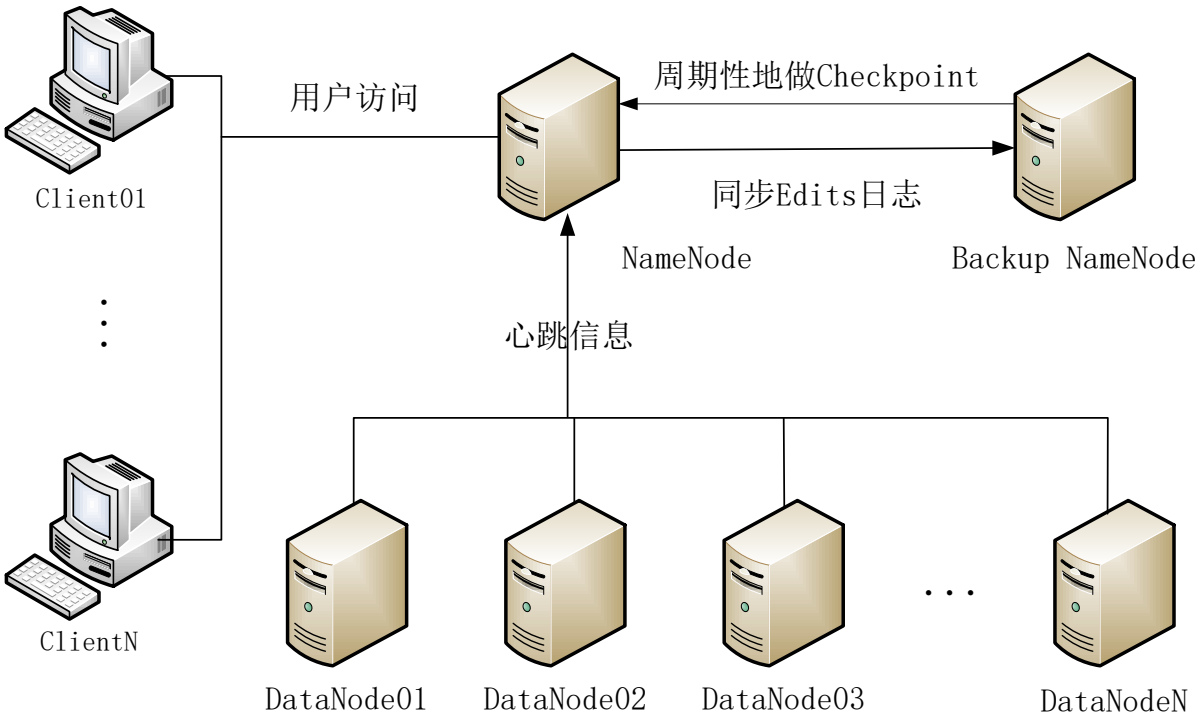


图 4.2 Backup Node 方案的系统原理图

与 Secondary NameNode 方案相同，Backup Node 方案也是 HDFS 自带的高可用性保障方案，但它无需定期从 NameNode 中下载镜像文件，只需同步操作日志，效率比 Checkpoint 更高。但该方案还不成熟，NameNode 无法提供服务时，Backup Node 还不能直接接替 NameNode 并提供服务，需要手动切换，操作并不方便。

4.2.3 Avatar 方案

Avatar 方案由社交媒体网站 FaceBook 提出[51,52]。它包含两个 NameNode 节点，一个 Primary NameNode，一个 Standby NameNode。另外，还有一台用来共享操作日志的 NFS 服务器。Primary NameNode 在源码层面对原生的 NameNode 进行了封装与扩展，它除了会记录客户端访问文件系统时的操作行为，还将操作日志同步到 NFS 日志服务器上供 Standby NameNode 共享。

Avatar 方案中的 DataNode 需要同时向 Primary NameNode 和 Standby NameNode 周期性地上报当前节点所含的 Block 块信息。由于 Standby NameNode 在内存中保存了一份最新的、完整的元数据信息，当 Primary NameNode 意外宕机时，集群的管理权限可以立刻切换到 Standby NameNode，并可立刻对外提供服务。Avatar 方案的系统原理如图 4.3 所示。

万方数据

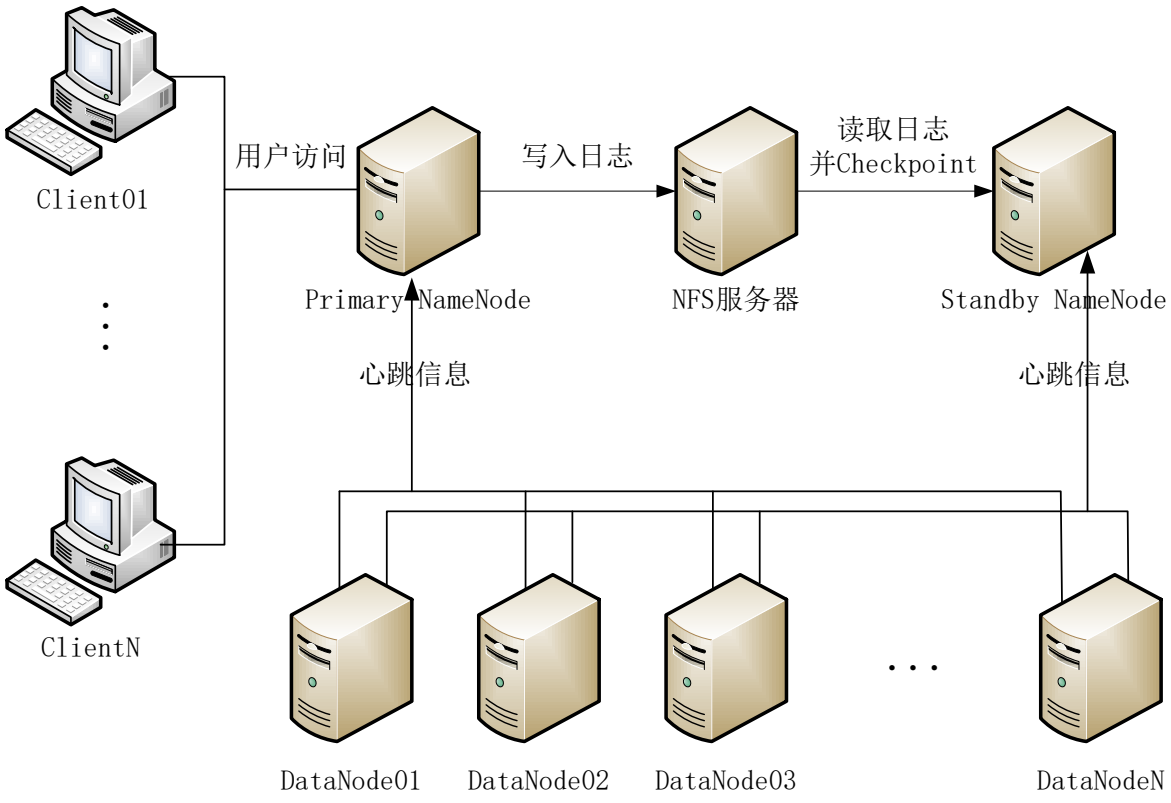


图 4.3 Avatar 方案系统原理图

Avatar 方案是真正的热备机制，实现了对元数据的实时备份，FaceBook 也已将其集成到了自己公司所维护的 HDFS 集群中。凭借 Avatar 方案的优越性，公司内基于 HDFS 的数据仓库在日常维护过程中减少了 10%的计划外停机和 50%的计划内的停机事件。

4.2.4 各方案优缺点比较

Secondary NameNode 方案采用的是定期 Checkpoint 操作来备份更新 NameNode 中的元数据信息，但 HDFS 的默认配置中限定了两次 Checkpoint 之间的时间间隔为 1 小时。这就导致一个问题，Secondary NameNode 中的元数据镜像文件可能与 NameNode 中实时的元数据信息差别很大，在下一次 Checkpoint 操作发生之前，Secondary NameNode 无法捕捉到上一次 Checkpoint 操作结束之后文件系统所发生的变化。当主节点宕机后，无法通过 Secondary NameNode 备份的镜像文件还原出一份完整真实的元数据[53]。除此之外，主备切换的操作还需人工手动完成，十分不方便。所以真实环境下很少采用 Secondary NameNode 方案来解决 NameNode 的单点问题。

Secondary NameNode 方案之所以要每隔 1 小时才进行一次 Checkpoint 操作，是因为当 HDFS 中存储了十分庞大的数据时，NameNode 在内存中维护的元数据信息也会非常大，通常 1TB 的数据文件对应的元数据大小约为 1.2GB。频繁地对 GB 级的文件进行下载、修改并上

万方数据

传将极大地占用网络带宽。Buckup Node 方案就是针对这点进行了优化，它的备份节点只从 NameNode 上下载一次元数据镜像文件，并依据从主节点上实时同步过来的 Edits 操作日志来实现自我更新。这种方案大大降低了网络带宽的占用，同时保证了元数据备份的实时性。但是 Buckup Node 方案备份的元数据并不包含 Block 块的位置信息，在主备切换后需要一段时间等待每个 DataNode 节点上报自己的块信息，造成切换时间较长[54,55]。

Avatar 方案又是一次对 Buckup Node 方案的补足[56]。在 Avatar 方案中，DataNode 需要向 Standby NameNode 发送 Block 块信息，节省了 Primary NameNode 宕机后 Standby NameNode 对元数据的补全过程，缩短了切换时间，且切换行为机器自动完成，无需人工介入，简单迅速。相较于 Secondary NameNode 方案与 Buckup Node 方案，Avatar 方案进一步提高了 HDFS 文件系统的可用性。但 AvatarNode 方案并不完美，单独设置的一台 NFS 日志服务器又引发了新的单点问题。同时，DataNode 开始同时向两个 NameNode 发送心跳信息，在一个节点规模较大的集群里，这样的方案无形中增加了一倍的网络通信量。

基于以上现有方案的优缺点，本文提出一种改进的防止 HDFS 发生单点故障问题的新方案——扁平化的 NameNode 模型。与传统的主/备模型不同，扁平化的 NameNode 模型由至少三台 NameNode 节点组成，节点之间通过多数派机制来保证元数据信息的一致性。NameNode 集群中的每台节点都能接收来自客户端的读写请求。该模型方案的特点是失败切换时间短，元数据一致性高，节点易于扩展，较 Avatar 方案而言，集群内的网络通信量也大大减少。

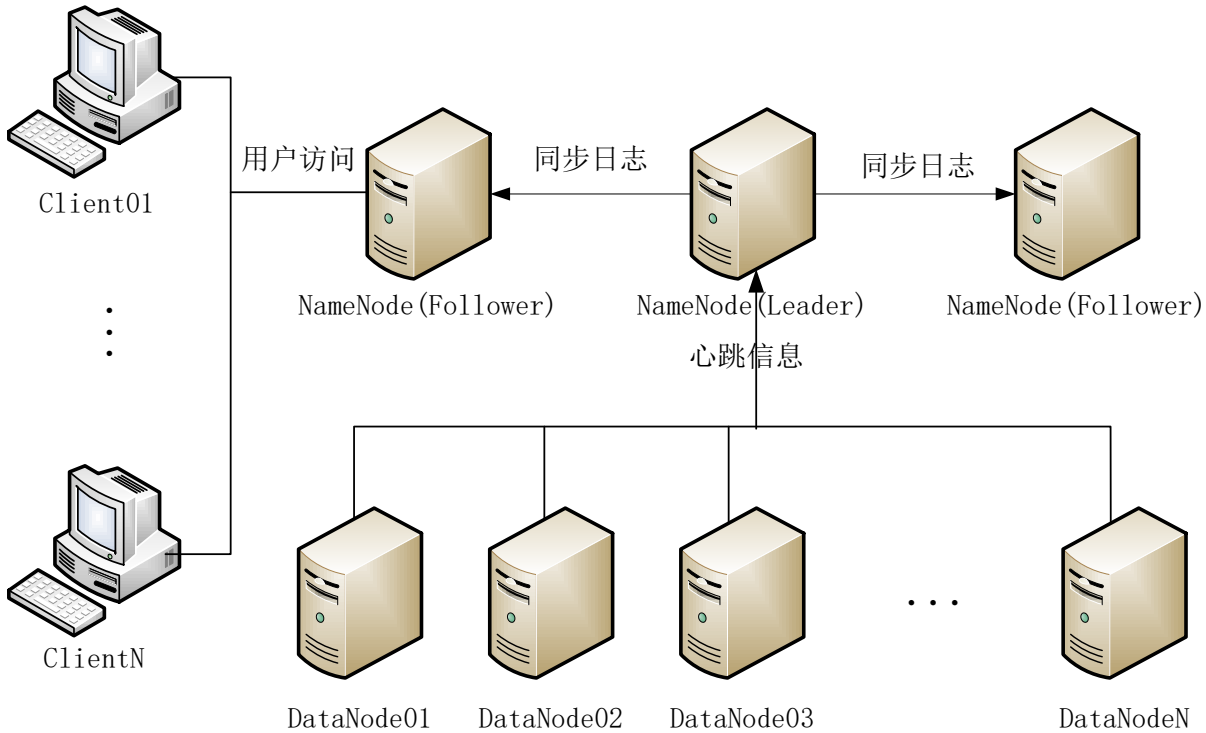


图 4.4 扁平化 NameNode 模型

现有方案与改进的扁平化 NameNode 方案的优缺点比较如表 4.1 所示。

万方数据

表 4.1 防止 HDFS 发生单点故障方案比较

方案名称	切换时间	元数据一致性	操作复杂度	是否热备	是否会自动切换
Secondary NameNode	长	不一致	高	否	否
Backup Node	中	不一致	中	否	否
Avatar	短	一致	中	是	是
扁平化 NameNode	短	一致	中	是	是

4.3 扁平化 NameNode 模型

在扁平化 NameNode 模型中需要三种角色的节点来协调工作：领导者 NameNode、候选者 NameNode 和跟随者 NameNode。一个节点在一段时间内可能扮演不止一种角色，但某一时刻只能处于一种角色。三种角色及相关术语介绍如下：

（1）领导者 NameNode：

处理客户端提交的读或写请求，并完成元数据同步。一个任期内只存在一个领导者。

（2）候选者 NameNode：

可以通过获得超过半数跟随者 NameNode 的选票成为一个任期内的领导者。

（3）跟随者 NameNode：

可以处理客户端提交的读请求。依据领导者的元数据来同步自己服务器上的元数据。

（4）任期：

从一轮选举开始到下一轮选举开始之间称作一个任期，每一个任期都有一个唯一的、递增的编号。

4.3.1 领导者 NameNode 的选举

当 HDFS 刚启动时，所有 NameNode 节点均处于跟随者状态，没有领导者。如果在 100ms 至 500ms 之间的任意时刻，跟随者 NameNode 没有接收到任何来自领导者 NameNode 的心跳消息（不含数据信息的远程过程调用消息），它就会假定此时集群内没有可达或可用的领导者，那么该跟随者 NameNode 就会发起选举，它首先会自增自己当前的任期号，设置一个比之前使用过的任何值都要大的新任期号。随后该跟随者 NameNode 进入候选者角色，尝试成为整个 NameNode 集群的领导者。

候选者 NameNode 会向其他 NameNode 服务器发送投票请求，同时自己会投给自己一票。

万方数据

如果获得集群中超过半数 NameNode 节点反馈的同意响应后，候选者 NameNode 就会将自己的角色转换为领导者，并立即向 NameNode 集群中其他服务器发送心跳信息，以建立领导者地位。三种状态的 NameNode 之间的转换条件如图 4.4 所示。

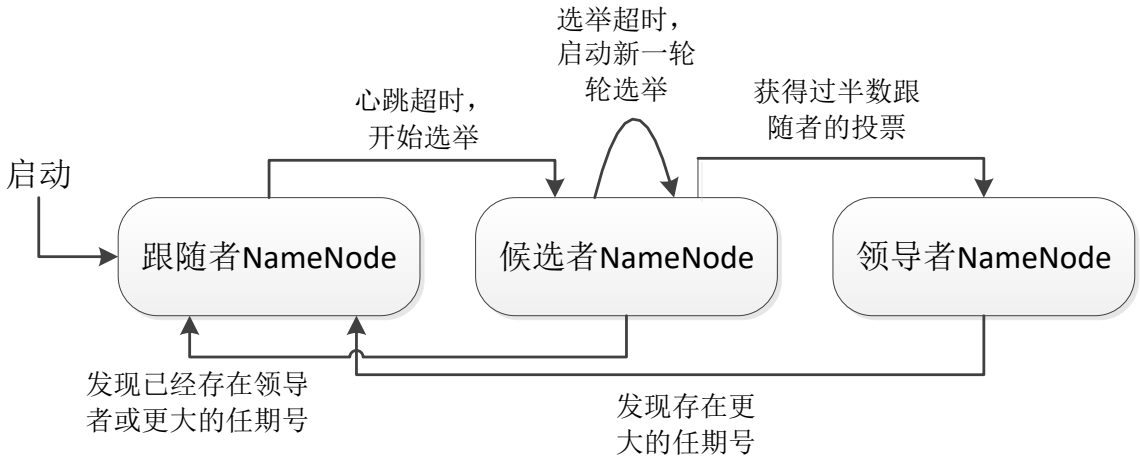


图 4.4 NameNode 状态转换图

上面描述的是正常选举领导者 NameNode 的过程，但在分布式环境中，可能会因为网络问题而产生一些异常状况，这种异常状况称之为中断事务。下面是两种常见的中断事务及其在扁平化 NameNode 方案中的解决方法：

（1）NameNode 集群中可能存在着其他候选者试图竞选领导者，并成功获取多数票当选为领导者。在扁平化 NameNode 方案中，当前候选者 NameNode 如果收到了来自于有效领导者 NameNode 的心跳信息，它就会立即放弃成为领导者的尝试，随即回到跟随者状态。

（2）由于 NameNode 集群中存在多个候选者，这些候选者 NameNode 分摊了来自跟随者的选票，造成谁都没有获得多数票，谁都无法当选领导者的情况。解决方法是，为每个候选者设置一个随机的选举超时时间。选举超时后，候选者会再次自增自己的任期号，然后重启新一轮的选举，直至集群最终产生领导者。

4.3.2 读文件过程

当领导者 NameNode 被选举出来后，就可以接收来自客户端的读请求。读请求的流程如图 4.5 所示，其主要过程描述如下：

（1）客户端向 NameNode 集群中任意一台服务器发送读请求；

（2）接收到来自客户端读请求的 NameNode 服务器随即去目录树中检查 HDFS 中是否存在该文件。如果存在，则返回该文件对应的 Block 块所在数据节点的列表信息，如果 HDFS 中不存在客户端要读的文件，则 NameNode 服务器返回文件不存在异常；

（3）客户端逐一向返回的 DataNode 列表信息中的 DataNode 服务器发送读文件请求。被

万方数据

请求的 DataNode 服务器向客户端传输数据文件。

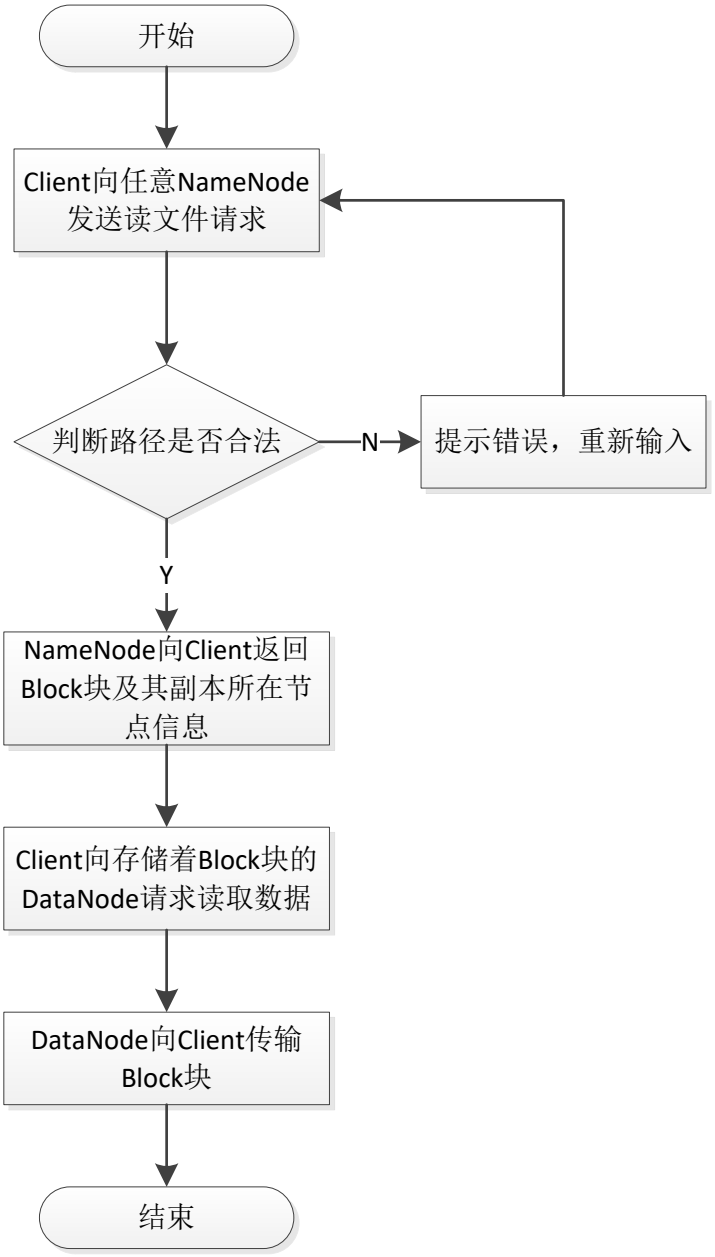


图 4.5 客户端向 NameNode 集群请求读文件

4.3.3 写文件过程

客户端向 NameNode 集群请求写文件的过程如图 4.6 所示，其主要过程可以描述为：

（1）客户端向领导者 NameNode 提交写一个数据块的请求；

（2）领导者 NameNode 首先去本机内存中维护的元数据的目录树中检查客户端所请求写入的文件的路径上是否已存在该文件，若没有，则会去 DataNode 信息池中挑选若干台 DataNode 服务器作为客户端可写入文件的数据节点，并将客户端申请写入 HDFS 的文件的元信息和所挑选出来的 DataNode 节点元信息作为一条日志发送给一致性模块；

万方数据

（3）领导者 NameNode 中的一致性模块向所有跟随者 NameNode 同步日志；

（4）日志同步完成后将之前挑选出来的 DataNode 数据节点列表信息返回给客户端。客户端在接收到领导者 NameNode 返回的 DataNode 列表信息后开始向这些 DataNode 上传文件。

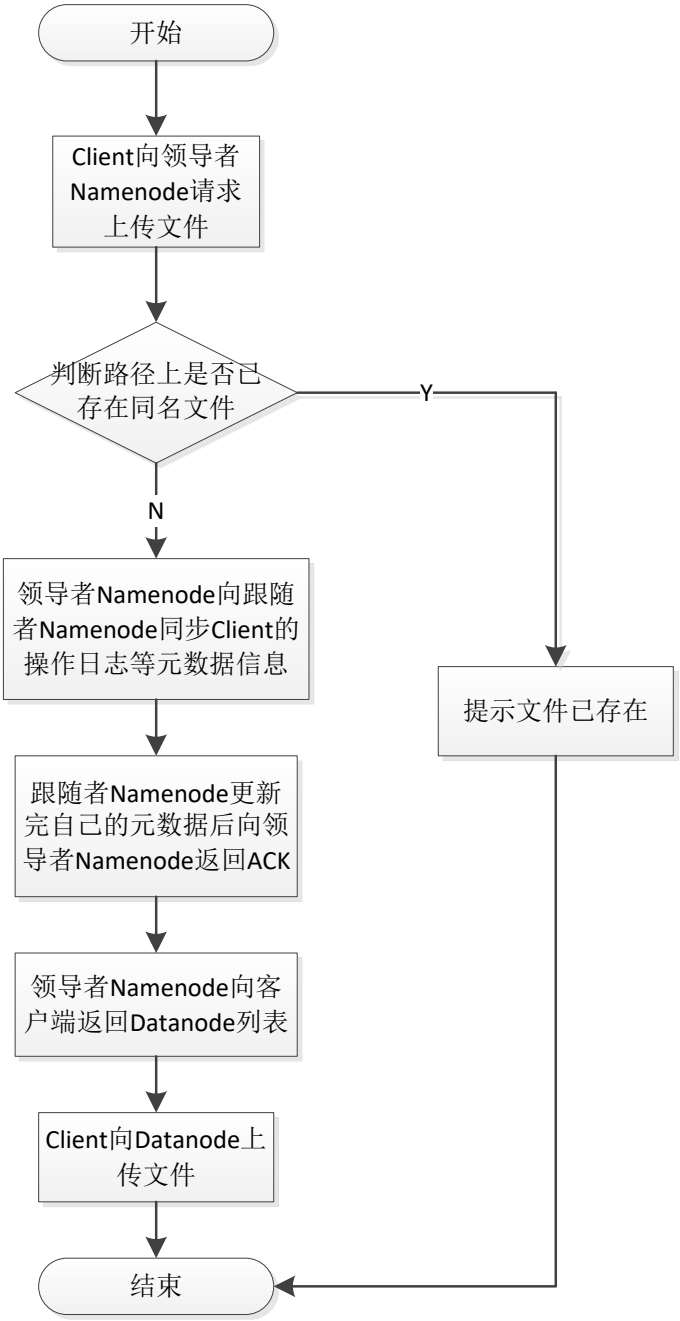


图 4.6 客户端向 NameNode 集群请求写文件

相较与处理客户端提交的读请求，写请求的过程会更复杂一些，遇到异常状况的概率也会更大。领导者 NameNode 可能会出现崩溃或者由于网络原因失去与过半跟随者 NameNode 的联系，为了保证日志在每台服务器节点上的完整性与一致性和整个 NameNode 集群的高可用性，此时 NameNode 集群会进入崩溃恢复状态，处于崩溃恢复状态的 NameNode 集群会暂停对外服务。此时，某些或某个跟随者 NameNode 会进入候选者状态，并向其他服务器发起投票请求，请求里会包含自身最后一条日志记录信息的索引（lastIndex）以及任期号（lastTerm）。

万方数据

当响应投票的服务器接收到请求，它会将候选者的日志信息与自己的日志信息进行比较，如果投票者（跟随者 NameNode）的日志更完整，它就会拒绝投票。日志更完整的判别条件如下：

$(lastTerm_{follower}>lastTerm_{candidate})||$

$((lastTerm_{follower}==lastTerm_{candidate})\&\&(lastIndex_{follower}>lastTerm_{candidate}))$

可以保证最终赢得选举的 NameNode 服务器拥有比大多数投票者更完整的日志记录。

经过上面步骤选举出领导者 NameNode 后，新的领导者 NameNode 会不断地向跟随者 NameNode 发送包含自己日志信息的心跳消息。跟随者 NameNode 根据接收到的心跳消息，删除所有跟领导者 NameNode 不同的日志记录，并将所有丢失的日志记录依照领导者的日志进行补足。

4.4 主备节点切换测试

在同一台服务器上，分别对 Secondary NameNode 方案、Buckup Node 方案、Avatar 方案和扁平化 NameNode 方案在 1000、5000、10000 和 15000 个不同数量级的文件上进行了多次主备节点切换测试，切换时间对比情况如图 4.7 所示。

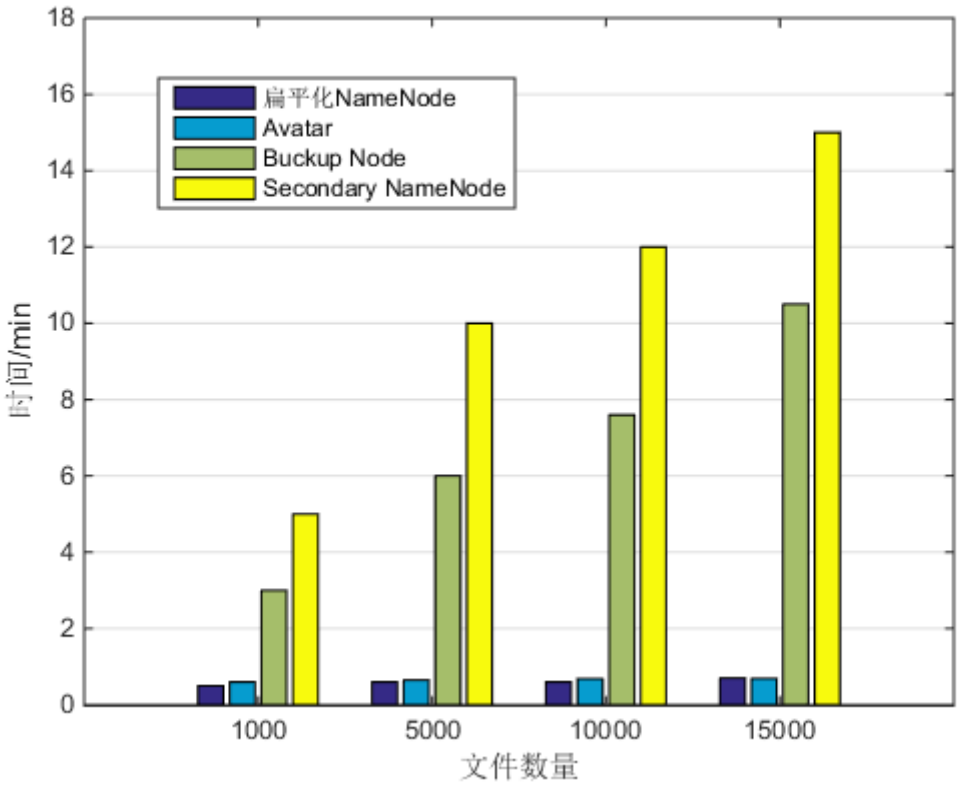


图 4.7 各种方案切换时间对比图

从图 4.7 可以看出，Avatar 方案和改进的扁平化 NameNode 方案在几轮主备切换测试实

万方数据

验中，使用的时间基本不随文件数量的增多而变长，均不到 1 分钟。原因是他们都属于热备机制，备用节点中的元数据时刻与主节点保持一致，在主节点宕机后，切换到备用节点消耗的时间只与切换方式有关，而与 HDFS 文件系统中所存储的文件数量无关。与之相对的，Buckup Node 和 Secondary NameNode 两个方案的主备节点切换时间会随着文件系统中文件数量的增长而呈现线性增长趋势。这是因为这两种方案所备份的元数据中并不包含 Block 块的位置信息，在复原主节点的过程中需要等待 DataNode 上报自己的 Block 块信息，而文件系统中存储的文件数量越多即 Block 块数量越多，意味着恢复主节点的时间就越长。且 Secondary NameNode 方案在人工恢复主节点的操作上较 Backup Node 方案更为繁琐，所以消耗时间也越多。

4.5 本章小结

本章首先指出采用主-从架构的 HDFS 所存在的单点故障隐患，并分别介绍了现有解决 HDFS 单点故障隐患的三种常用方案，它们分别是 Secondary NameNode、Buckup Node 和 Avatar。针对这些方案的优缺点，本章在随后的小节提出了一种新的解决 HDFS 单点故障隐患的方案——扁平化的 NameNode 模型。最后通过实验，对上述四个方案进行了对比分析。

万方数据

第五章 实验测试与分析

经过前面几个章节对 HDFS 文件系统的深入分析与优化，本章将具体讲述对从 Apache 官网下载的原生 Hadoop-2.7.1 版本安装文件经定制化后而具有高可用性的 HDFS 文件系统的实验与测试过程。

5.1 实验环境搭建

5.1.1 实验环境

改进版的高可用 HDFS 文件系统除了需要配置不少于 3 个的 NameNode 节点外，因为采用了基于纠删码算法的存储方案，在 DataNode 节点数量上的配置也有所规定。在 HDFS-LRC（4，2，2）编码中，会产生 4 个原始数据块，2 个局部校验块和 2 个全局校验块，总共 8 块冗余数据。为了在数据失效后获得最佳的容错能力与重构性能，这 8 个数据块需要分散存储在 8 台不同的数据节点上。

为了满足性能和节点数量的需求，实验选择在真实服务器上通过 VMWare 软件虚拟出总共 11 个节点。服务器的硬件配置如表 5.1 所示：

表 5.1 硬件配置

名称	配置
CPU 信息	Intel Xeon E5440 3.40GHz
核数	32 核
硬盘	4TB
内存	300GB

在 VMWare 软件虚拟化出的每台节点中所安装的软件及其版本号以及为每台虚拟节点所分配的硬件资源如表 5.2 所示：

表 5.2 系统环境

操作系统版本	CentOS 6.7（64 位）
JDK	JDK 1.7.0_79
Hadoop 源码版本	Hadoop 2.7.1
CPU 核数	2
硬盘	20G
内存	4G

实验环境中的 NameNode 节点用缩写字母 NN 表示，DataNode 节点用缩写字母 DN 表示，多台同类型的节点间用递增阿拉伯数字作为后缀加以区分。最终，集群内主机名与 IP 地址的

万方数据

对照关系如表 5.3 所示：

表 5.3 主机名与 IP 地址对照表

主机名	IP 地址
NN01	192.168.31.81
NN02	192.168.31.82
NN03	192.168.31.83
DN01	192.168.31.91
DN02	192.168.31.92
DN03	192.168.31.93
DN04	192.168.31.94
DN05	192.168.31.95
DN06	192.168.31.96
DN07	192.168.31.97
DN08	192.168.31.98

集群搭建完成后的拓扑图如下：

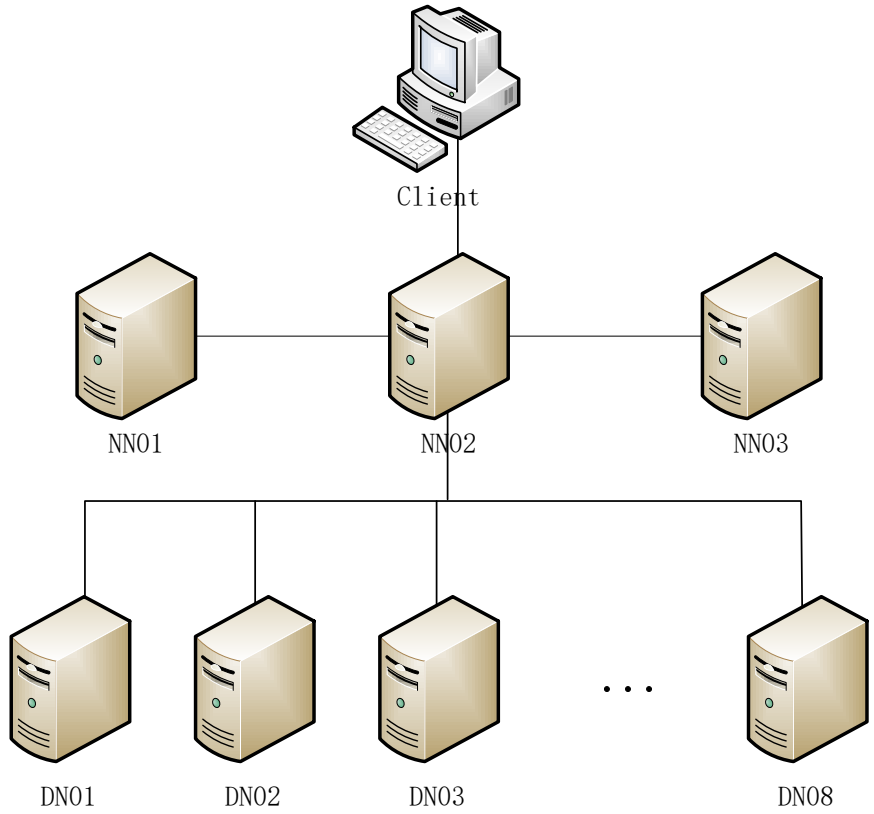


图 5.1 集群拓扑图

5.1.2 Hadoop 集群搭建

1.配置集群中各个节点的网络环境

通过修改每个节点上的 hosts 文件和网卡配置文件，将 11 台节点编入可以互相通信的同一个局域网中。局域网中每台节点的主机名和 IP 地址的映射关系如表 5.3 所示。

万方数据

2.在每台节点上安装 JDK，并配置相应的环境变量。

3.关闭所有节点的防火墙，并配置每台主节点（NN）到所有从节点（DN）的 SSH 免密登录

HDFS 在启动后，NameNode 需要通过 SSH 方式来控制 DataNode 的启停，因此，节点通信过程中不能有密码验证过程。

4.Hadoop 核心配置文件

在部署 Hadoop 的过程中，需要根据集群真实的网络环境来修改相应的配置文件，需要修改的配置文件有 6 个，分别为 hadoop-env.sh、core-site.xml、hdfs-site.xml、yarn-site.xml、mapred-site.xml 和 slaves。下面将对 core-site.xml、hdfs-site.xml 和 slaves 这 3 个与 HDFS 文件系统关系紧密的配置文件做简要介绍。

（1）core-site.xml

该配置文件为 Hadoop 的核心配置文件，用来指定 HDFS 文件系统采用哪个 NameNode 集群（fs.defaultFS）以及 Hadoop 运行过程中所产生的临时文件的存放目录（hadoop.tmp.dir）。

```
<configuration>
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://ns1/</value>
</property>
<property>
    <name>hadoop.tmp.dir</name>
    <value>/hwl/hadoop-2.7.1/tmp</value>
</property>
</configuration>
```

（2）hdfs-site.xml

该文件为 HDFS 的通用配置文件。主要指定 NameNode 集群的组成成员（dfs.ha.NameNodes.ns1），成员的 RPC 通信地址（dfs.NameNode.rpc-address.ns1.nn1）和 HTTP 通信地址（dfs.NameNode.http-address.ns1.nn1）。在扁平化的 NameNode 模型中需要至少三个 NameNode 节点，所以该配置文件中需要分别为 NN01、NN02 和 NN03 配置相应的通信地址。配置文件的最后要开启 NameNode 失败自动切换选项（dfs.ha.automatic-failover.enabled）。

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


万方数据

```
    <name>dfs.nameservices</name>
    <value>ns1</value>
</property>
<property>
    <name>dfs.ha.NameNodes.ns1</name>
    <value> NN01,NN02,NN03</value>
</property>
<property>
    <name>dfs.NameNode.rpc-address.ns1.nn1</name>
    <value> NN01:9000</value>
</property>
<property>
    <name>dfs.NameNode.http-address.ns1.nn1</name>
    <value> NN01:50070</value>
</property>
<!--  NN02 与 NN03 的通信地址配置同 NN01  -->
…
<property>
    <name>dfs.ha.automatic-failover.enabled</name>
    <value>true</value>
</property>
</configuration>
```

（3）slaves

slaves 文件配置的是 HDFS 集群中所有数据节点 DataNode 的主机名。

```
DN01
DN02
….
DN08
```

6.配置 Hadoop 环境变量

```
export JAVA_HOME=/app/jdk1.7.0_79
```


万方数据

```
export HADOOP_HOME=/hwl/hadoop-2.7.1
export PATH=$JAVA_HOME/bin:$HADOOP_HOME/bin:$PATH
```

7.将 JDK 和配置完成的 Hadoop 安装包分发到集群中所有节点上。

8.格式化 HDFS 文件系统

初次使用 HDFS 前，需要在每个 NameNode 节点上输入 shell 命令 hdfs NameNode –format 来格式化文件系统。当看到下图高亮部分日志信息时，表示格式化成功。

```
15/09/15 22:31:30 INFO namenode.FSImage: Allocated new BlockPoolId: BP-701922641-192.168.
8.91-1442327490762
15/09/15 22:31:30 INFO common.Storage: Storage directory /hwl/hadoop-2.7.1/tmp/dfs/name h
as been successfully formatted.
15/09/15 22:31:31 INFO namenode.NNStorageRetentionManager: Going to retain 1 images with
txid >= 0
15/09/15 22:31:31 INFO util.ExitUtil: Exiting with status 0
15/09/15 22:31:31 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
```

图 5.2 格式化 HDFS 结果图

5.2 系统测试与分析

5.2.1 启动测试

进入 Hadoop 安装路径下的 sbin 目录，执行 HDFS 的启动脚本（start-dfs.sh）。从图 5.3 中可以看出 NameNode 进程依次在主机 NN01、NN02 和 NN03 上启动，同时 DataNode 进程在主机 DN01 到 DN08 上启动。

```
[root@NN01 sbin]# pwd
/hwl/hadoop-2.7.1/sbin
[root@NN01 sbin]# ./start-dfs.sh
Starting namenodes on [NN01] [NN02] [NN03]
DN01: starting datanode, logging to /apps/hadoop-2.7.1/logs/hadoop-root-datanode-DN01.out
DN02: starting datanode, logging to /apps/hadoop-2.7.1/logs/hadoop-root-datanode-DN02.out
DN03: starting datanode, logging to /apps/hadoop-2.7.1/logs/hadoop-root-datanode-DN03.out
DN04: starting datanode, logging to /apps/hadoop-2.7.1/logs/hadoop-root-datanode-DN04.out
DN05: starting datanode, logging to /apps/hadoop-2.7.1/logs/hadoop-root-datanode-DN05.out
DN06: starting datanode, logging to /apps/hadoop-2.7.1/logs/hadoop-root-datanode-DN06.out
DN07: starting datanode, logging to /apps/hadoop-2.7.1/logs/hadoop-root-datanode-DN07.out
DN08: starting datanode, logging to /apps/hadoop-2.7.1/logs/hadoop-root-datanode-DN08.out
[root@NN01 sbin]# jps
2110 Jps
2140 NameNode
2201 SecondaryNameNode
[root@NN01 sbin]#
```

图 5.3 HDFS 启动效果图

打开本地浏览器，并在地址栏中输入 HDFS 监控页面的 URL（http://NN01:50070/）。当看到如图 5.4 所示页面时，表示 HDFS 启动成功。

万方数据

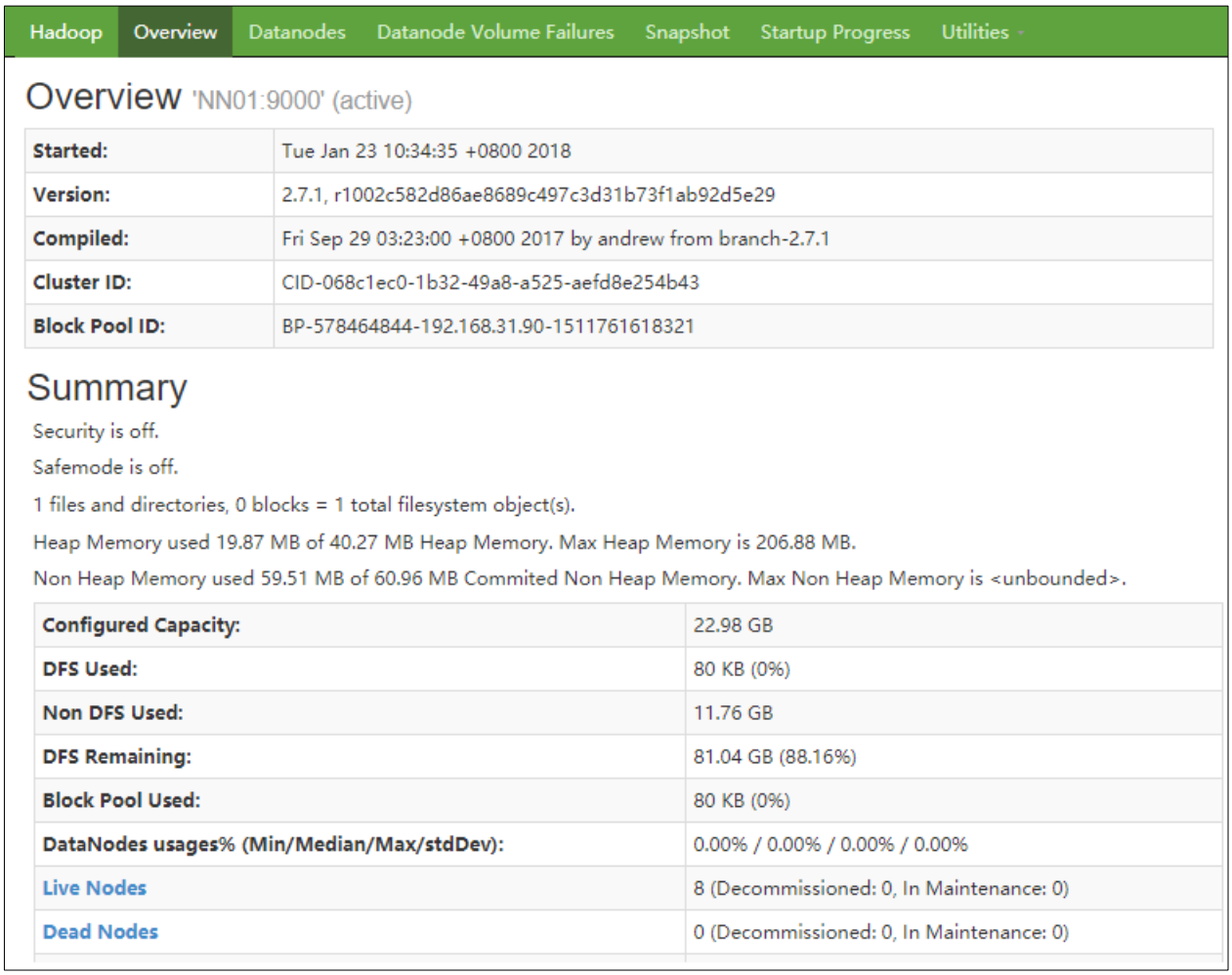


图 5.4 HDFS 监控 web 页面

点击 DataNodes 标签，可以查看当前集群中所有数据节点的状态信息。从图 5.5 中可以看出当前集群中共有 8 个 DataNode 节点，并处于可服务状态，每个节点可以用作存储的空间大小为 11.49GB。

万方数据

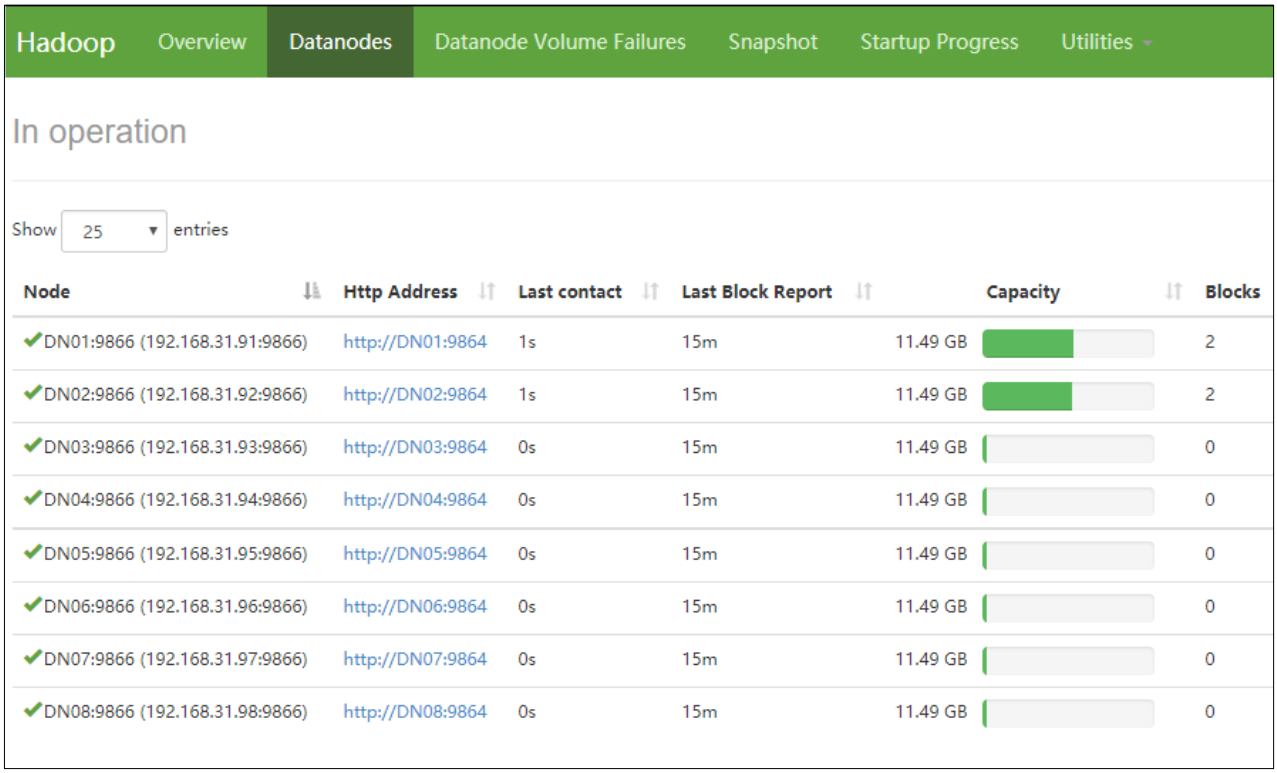


图 5.5 DataNode 状态信息监控页面

5.2.2 读写测试

对于文件系统而言，最基本的也是最重要的一点就是保证文件读写的正确性。当客户端上传文件时，文件要能正确的分块并存储于不同的数据节点上。当客户端下载文件时，文件系统要能还原出真实完整的数据。为了保证上述两项功能在运行过程中的无误，本文针对本系统进行了相关测试。测试的基本思路是向改进版 HDFS 文件系统上传文件并下载，将下载的文件与原始文件内容进行比对，如果相同，则代表读写过程成功，文件系统能够正常运行。

这里的测试输入为 NN01 节点上/home 目录中的 TestData 文本文件，TestData 文本文件大小为 256MB（268435456 byte），内容如图 5.6 所示，为循环写入的“helloworld”英文单词。因为每个英文字母只占 1 个字节，所以比较好控制最终文件的大小。

```
[root@NN01 home]# cat TestData
helloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhellow
orldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhe
lloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhellowor
ldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhell
oworldhelloworldhelloworldhlloworldhelloworldhelloworldhelloworldhelloworldhelloworldh
elloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhellowo
```

图 5.6 TestData 文件内容

万方数据

在终端窗口中输入文件上传命令，根据 HDFS-LRC（4，2，2）算法的设计原则，256MB 的 TestData 文件上传完毕后会按 64MB 一个块的大小被切分成 4 块，另外还会生成 2 个局部校验块和 2 个全局校验块，总共 8 个冗余数据块。

```
[root@NN01 home]# hdfs dfs -put /TestData /hdfs-lrc
2018-01-23 14:22:50,306 WARN erasurecode.ErasureCodeNative: ISA-L support is not available in your plat
form... using builtin-java codec where applicable
[root@NN01 home]# hdfs fsck /hdfs-lrc/TestData -files -blocks -locations
Connecting to namenode via http://NN01:9870/fsck?ugi=user&files=1&blocks=1&locations=1&path=%2Fhdfs-lrc
%2FTestData
FSCK started by user (auth:SIMPLE) from /192.168.31.91 for path /hdfs-lrc/TestData at Thu Jan 23 14:22:
59 EST 2018
/hdfs-lrc/TestData 268435456 bytes, erasure-coded: policy=hdfs-lrc, 4 block(s):  OK
0. BP-529485104-192.168.182.11-1511810134643:blk_-9223372036854775792_1065 len=268435456 Live repl=8  [
blk_-9223372036854775792:DatanodeInfoWithStorage[192.168.31.91:9866,DS-da58ee3e-adcc-4f6c-8488-c2a0b742
d8b9,DISK], blk_-9223372036854775786:DatanodeInfoWithStorage[192.168.31.92:9866,DS-c36de658-0f5a-42de-8
898-eab3b04c7016,DISK], blk_-9223372036854775785:DatanodeInfoWithStorage[192.168.31.93:9866,DS-a3569982
-de52-42b5-8543-94578f8b452a,DISK], blk_-9223372036854775784:DatanodeInfoWithStorage[192.168.31.94:9866
,DS-71be9468-c0c7-437c-8b59-ece27593b4c2,DISK],  [blk_-9223372036854775801:DatanodeInfoWithStorage[192.
168.31.95:9866,DS-8758ee3e-adcc-4f6c-8488-c2a0b742d8b9,DISK], blk_-9223372036854775875:DatanodeInfoWith
Storage[192.168.31.96:9866,DS-a96de658-0f5a-42de-8898-eab3b04c7016,DISK], blk_-9223372036854775888:Data
nodeInfoWithStorage[192.168.31.97:9866,DS-a1569982-de52-42b5-8543-94578f8b452a,DISK], blk_-922337203685
4775966:DatanodeInfoWithStorage[192.168.31.98:9866,DS-99be9468-c0c7-437c-8b59-ece27593b4c2,DISK]]
[root@NN01 home]#
```

图 5.6　文件上传

从图5.6终端输出的信息中可以看到，TestData 文件上传过程中采用了 hdfs-lrc 存储策略，原始文件被切分成 4 个 Block 块，总共生成了 8 个冗余文件，每个文件块分布在 IP 地址从 192.168.31.91 到 192.168.31.98 的 DataNode 节点上，符合实验预期结果。

在进行文件下载测试之前，为了防止下载文件覆盖上传目录下的原始文件，所以选择在 /home/test 目录下保存下载文件。从图 5.7 中可以看出下载文件 TestData 的文件大小和文本内容均与原始文件保持一致。

```
[root@NN01 test]# hdfs dfs -get /hdfs-lrc/TestData /home/test
2018-01-24 20:46:46,693 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platf
orm... using builtin-java classes where applicable
[root@NN01 test]# ll
总用量 1853
-rw-r--r--. 1 root root 268435456 1月  24 20:46 TestData
[root@NN01 test]# cat TestData
helloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhel
loworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhellow
orldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworl
dhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhl
loworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhellow
orldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworl
```

图 5.7　文件下载

为了验证领导者 NameNode 节点宕机后，文件系统依然能正常完成读写任务，我们可以先通过 kill 指令结束掉领导者 NameNode 进程来模拟实际使用中的宕机情形。根据选举法则，在按序依次启动 NN01、NN02 和 NN03 节点的过程中，NN02 会最先获得来自 NN01 和 NN02 的选票从而当选领导者，故我们在 NN02 节点上 kill 掉 NameNode 进程，实验操作如图 5.8 所示。

万方数据

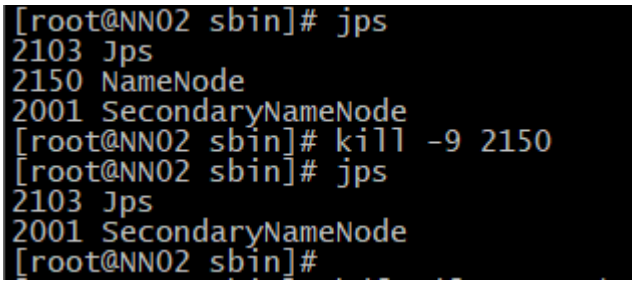


图 5.8 kill 领导者 NameNode 进程

当原先的领导者 NameNode 进程被结束掉后，我们选择在 NN03 节点上执行文件上传功能。操作截图 5.9 中，文本文件 TestData 被上传至新的目录/hdfs-lrc02 中，通过查看命令发现 TestData 文件上传成功，可见原先在 NN02 节点上的领导者 NameNode 进程的宕机并没有影响到改进版 HDFS 文件系统的高可用性。

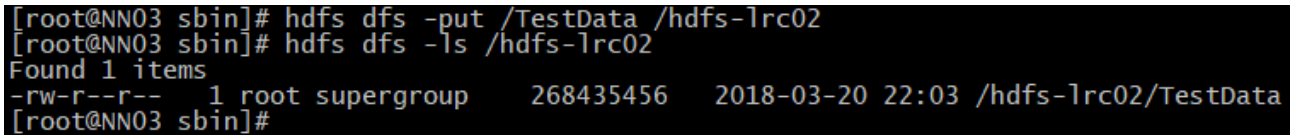


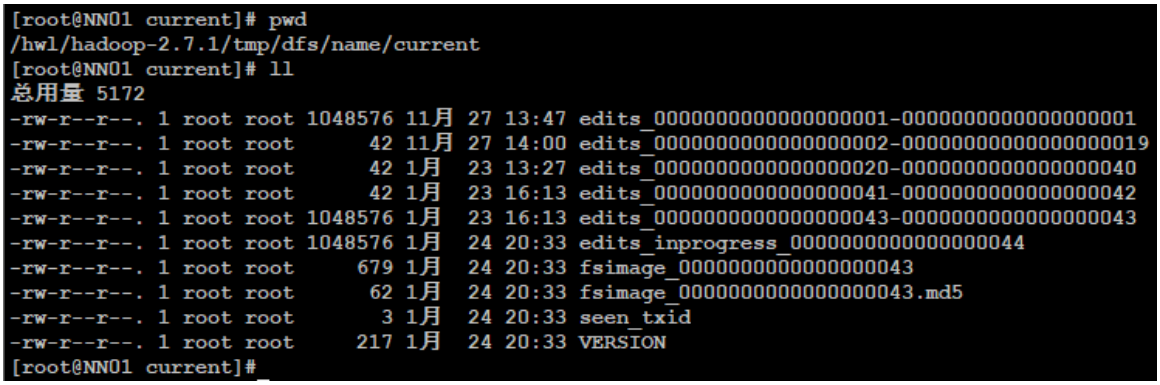
图 5.9 文件上传

综合上述实验步骤与实验结果，可以证明改进版的 HDFS 文件系统在文件读写方面运行正常。在领导者 NameNode 进程宕机后，系统依然能正常对外提供服务，具有较好的高可用性。

5.2.3 元数据一致性测试

高可用 HDFS 文件系统的高可用特性主要表现在解决了主-从架构系统的单点故障问题，而解决单点故障问题的关键是保证 NameNode 集群中所有节点中元数据的共享与同步。

经过几次如小节 5.2.2 所描述的读写操作，并查看每个 NameNode 节点上的元数据文件，从图 5.10 中可以看出，NN01、NN02 和 NN03 上的元数据文件均相同，可见元数据信息同步成功。



（a）

万方数据

```
[root@NN02 current]# pwd
/hwl/hadoop-2.7.1/tmp/dfs/name/current
[root@NN02 current]# ll
总用量 5172
-rw-r--r--. 1 root root 1048576 11月 27 13:47 edits_0000000000000000001-0000000000000000001
-rw-r--r--. 1 root root      42 11月 27 14:00 edits_0000000000000000002-0000000000000000019
-rw-r--r--. 1 root root      42 1月  23 13:27 edits_0000000000000000020-0000000000000000040
-rw-r--r--. 1 root root      42 1月  23 16:13 edits_0000000000000000041-0000000000000000042
-rw-r--r--. 1 root root 1048576 1月  23 16:13 edits_0000000000000000043-0000000000000000043
-rw-r--r--. 1 root root 1048576 1月  24 20:33 edits_inprogress_0000000000000000044
-rw-r--r--. 1 root root     679 1月  24 20:33 fsimage_0000000000000000043
-rw-r--r--. 1 root root      62 1月  24 20:33 fsimage_0000000000000000043.md5
-rw-r--r--. 1 root root       3 1月  24 20:33 seen_txid
-rw-r--r--. 1 root root     217 1月  24 20:33 VERSION
[root@NN02 current]#
```

（b）

```
[root@NN03 current]# pwd
/hwl/hadoop-2.7.1/tmp/dfs/name/current
[root@NN03 current]# ll
总用量 5172
-rw-r--r--. 1 root root 1048576 11月 27 13:47 edits_0000000000000000001-0000000000000000001
-rw-r--r--. 1 root root      42 11月 27 14:00 edits_0000000000000000002-0000000000000000019
-rw-r--r--. 1 root root      42 1月  23 13:27 edits_0000000000000000020-0000000000000000040
-rw-r--r--. 1 root root      42 1月  23 16:13 edits_0000000000000000041-0000000000000000042
-rw-r--r--. 1 root root 1048576 1月  23 16:13 edits_0000000000000000043-0000000000000000043
-rw-r--r--. 1 root root 1048576 1月  24 20:33 edits_inprogress_0000000000000000044
-rw-r--r--. 1 root root     679 1月  24 20:33 fsimage_0000000000000000043
-rw-r--r--. 1 root root      62 1月  24 20:33 fsimage_0000000000000000043.md5
-rw-r--r--. 1 root root       3 1月  24 20:33 seen_txid
-rw-r--r--. 1 root root     217 1月  24 20:33 VERSION
[root@NN03 current]#
```

（c）

图 5.10 NameNode 中保存在磁盘上的元数据文件

5.2.4 结果分析

从读写测试的实验中可以看出本文件系统在执行文件上传操作时，一个大小为 256MB 的测试文件经 HDFS-LRC（4，2，2）算法编码存储后，最终产生了 8 个冗余数据块，并分散存储在 8 台不同的 DataNode 节点上。与传统的三副本冗余策略相比，在需要存储的数据量较小时，基于纠删码算法的冗余策略反而需要用到更多的存储服务器。原因是如果将超过纠删码算法容错能力的分块数，在本系统中也就是超过 2 个的文件分块存储在同一服务器，如果该服务器意外宕机，将会造成不可复原的后果。所以在小数据量存储的情况下，副本冗余策略还是简单易行的。而当数据总量增长到需要用来作为存储的服务器个数超过了用纠删码算法存储一个文件所生成的分块数时，基于纠删码算法的存储策略就体现出了存储利用率高的优势，而海量数据的存储确实是云计算与大数据领域中更为常见的场景。

提高磁盘存储利用率从某种角度来说就是在保证原始数据完整性的前提下降低每次写入文件系统的数据量，从而提高文件写入速率，缩短任务处理时间，最终提升系统在这一方面的可用性。而本章针对高可用 HDFS 文件系统不仅进行了读写测试还对领导者 NameNode 与跟随者 NameNode 中的元数据进行了一致性测试实验。

万方数据

从元数据一致性测试实验中可以看出，在对文件系统进行读写操作时，客户端所提交的请求与系统状态的变化最终以 Edits 和 Fsimage 两个文件形式持久化到了每个 NameNode 节点本地磁盘的相应目录中。而从文件的修改时间可以判断，三个 NameNode 中的日志文件与镜像文件的内容几乎是同时被写入的，可以证明元数据信息的同步是实时的。当出现任意一个 NameNode 节点宕机失效时，如果该失效 NameNode 为领导者角色，那另外两个 NameNode 会立刻选举产生新的领导者，并对外提供服务；如果失效 NameNode 为跟随者角色，则不会影响集群功能，只需要在合适的时间动态扩展新节点即可，新节点会根据领导者 NameNode 的心跳消息来补全自己的元数据信息。

5.3 本章小结

本章通过虚拟化软件来模拟真实集群环境，基于修改过的 Hadoop 源码安装包，搭建了一个高可用的 HDFS 文件系统，并分小节对系统的关键环节，如启动环节、读写环节和元数据一致性环节进行了测试，最后对测试结果进行了描述分析。测试效果表明改进版的 HDFS 文件系统能够正常的完成文件的读写操作，很好的解决 NameNode 单点故障问题，提升系统的高可用性。

万方数据

第六章 总结与展望

6.1 总结

HDFS 作为作为 Hadoop 生态系统的核心组件之一，为生态圈中的其他组件提供了一个高可靠、易扩展的分布式存储服务。本文着重对分布式文件系统 HDFS 的高可用性进行研究。HDFS 采用的是单 NameNode 配备多 DataNode 的主-从架构模型，故而引发了潜在的单点故障隐患，降低了整个系统的可用性。除此以外，HDFS 默认的 3 副本容错机制在对海量数据进行存储时需要耗费大量的硬件存储设备。为了解决 HDFS 潜在的单点故障问题与降低磁盘存储消耗，论文做了如下几个方面的工作：

（1）着重探究了 HDFS 的组成架构，介绍了架构中的每种角色，除了最主要的 NameNode 与 DataNode 外，还介绍了客户端与 Secondary NameNode 在文件系统中发挥的作用，以及这些组成部分相互之间又是如何协调工作的。

（2）总结了分布存储领域在保障数据可靠性方面的两种数据冗余策略，分别为副本策略和纠删码策略。论文阐述了副本策略在海量数据存储场景中的局限性，提出了一种基于局部校验纠删码算法的 HDFS 优化存储策略。通过实验比较分析了其与另外两种传统纠删码算法的编码与重构性能。虽然在编码效率上不及柯西 RS 码，但在更为常见的重构原数据情景下获得了较其余两种算法更高的效率。

（3）介绍了现今的几种防止 HDFS 发生单点故障的方案，并分析总结了每种方案的优缺点。提出并实现了一种扁平化的 NameNode 模型，通过分布式一致性协议，对 HDFS 中原本单一的 NameNode 节点实现热备。当主节点意外宕机后，备份节点可以立刻接替主节点之前的服务，这一切换过程速度快，元数据丢失概率低。

（4）设计并实现了一种高可用的 HDFS 文件系统，将之前章节中基于局部校验纠删码算法的存储策略与扁平化的 NameNode 模型封装成模块替换掉了原生 HDFS 安装文件中相应位置的代码。经过试验，系统能够正常运行。相较于原生 HDFS 系统，改进版本提高了存储利用率，排除了主节点单点故障隐患。

6.2 展望

Hadoop 作为近几年最流行的大数据处理一栈式方案，已在越来越多相关企业与科研院所中得到应用。研究的下一步还将放在如何进一步提高其最重要的组件 HDFS 的可用性上。

万方数据

（1）在基于纠删码算法的分布式存储系统中，重构一块原始数据需要从分散的不同网络设备上拉取剩余原始数据与冗余校验块，虽然局部校验纠删码算法能够减少恢复失效数据时所用的数据块数量，但通过网络汇总数据的过程依然占据了重构过程的绝大多数时间，如何降低数据在网络传输中的时间将会是下一步的研究重点。

（2）HDFS 中的 NameNode 节点会在内存中维护整个文件系统的元数据信息。通常一个 Block 块所对应的 NameNode 中的一条元数据信息约占 150 字节，如果 DataNode 中存在着大量的小文件块将会消耗大量的 NameNode 内存，此时，NameNode 内存大小将成为制约集群规模的主要因素。如何优化 HDFS 文件系统对于小文件的存储，提高 NameNode 内存利用率也将是下一步研究的方向。

万方数据

参考文献

[1] Honnutagi P S. The Hadoop Distributed File System[J]. International Journal of Computer Science & Information Technolo, 2014:13-16.

[2] 张乐. 云计算环境下的分布存储关键技术研究[J]. 电子技术与软件工程, 2015(23):185-186.

[3] 范素娟, 田军锋, FANSu-juan,等. 基于Hadoop的云计算平台研究与实现[J]. 计算机技术与发展, 2016(7):127-132.

[4] Agrawal D, Abbadi A E, Antony S, et al. Data Management Challenges in Cloud Computing Infrastructures[C]// Databases in Networked Information Systems, International Workshop, Dnis 2010, Aizu-Wakamatsu, Japan, March 29-31, 2010. Proceedings. DBLP, 2010:1-10.

[5] 刘建矿, 于炯, 英昌甜,等. 基于内存云架构的带宽负载均衡算法[J]. 计算机工程与设计, 2015(11):2886-2891.

[6] 韩德志, 傅丰. 高可用存储网络关键技术的研究[M]. 科学出版社, 2009:101-108.

[7] Ford D, Popovici F I, Stokely M, et al. Availability in globally distributed storage systems[C]// Usenix Conference on Operating Systems Design and Implementation. USENIX Association, 2010:61-74.

[8] D'Souza, Jude Clement. D'Souza, J. C. Kthfs – A Highly Available and Scalable File System[J]. Teknik Och Teknologier, 2013:10-13.

[9] TomWhite. Hadoop权威指南.第3版[M]. 清华大学出版社:2015.68-71

[10] Ghemawat S, Gobioff H, Leung S T. The Google file system[J]. ACM SIGOPS Operating Systems Review, 2003, 37(5):29-43.

[11] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Commun. ACM, 2008, 51(1):10-15.

[12] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google file system[J]. Acm Sigops Operating Systems Review, 2003, 37(5):29-43.

[13] 韩佩. HDFS高可用性方案的研究与优化[D]. 西北大学, 2013:42-45.

[14] Oriani A, Garcia I C. From Backup to Hot Standby: High Availability for HDFS[C]// IEEE, Symposium on Reliable Distributed Systems. IEEE Computer Society, 2012:131-140.

[15] Dimakis A G, Ramchandran K, Wu Y, et al. A Survey on Network Codes for Distributed Storage[J]. Proceedings of the IEEE, 2011, 99(3):476-489.

[16] Dabek F, Kaashoek M F, Karger D, et al. Wide-area cooperative storage with CFS[C]// ACM, 2001:202-215.

[17] Kubiatowicz J, Wells C, Zhao B, et al. OceanStore[J]. Acm Sigops Operating Systems Review, 2000, 34(5):190-201.

[18] Introduction To YanXing File System. http://www.docin.com.

[19] Han Y. Cloud storage for digital preservation: optimal uses of Amazon S3 and Glacier[J]. Library Hi Tech, 2015, 33(2):261-271.

[20] Palankar M R, Iamnitchi A, Ripeanu M, et al. Amazon S3 for science grids:a viable solution?[C]// 2008:55-64.

[21] Li B, Wang M, Zhao Y, et al. Modeling and Verifying Google File System[C]// IEEE, International Symposium on High Assurance Systems Engineering. IEEE, 2015:207-214.

[22] Dean J, Ghemawat S. MapReduce: A Flexible Data Processing Tool[J]. Communications of the Acm, 2010, 53(1):72-77.

[23] Fredrick Ishengoma. HDFS+: Erasure-Coding Based Hadoop Distributed File System[J]. International Journal of Scientific & Technology Research, 2013, volume 2(Issue 9):11-12.

万方数据

[24] 鲁阳, 郑岩. 利用Zookeeper对HDFS中Namenode单点失败的改进方法[J]. 软件, 2012, 33(12):192-196.
[25] 邓鹏, 李枚毅, 何诚. Namenode单点故障解决方案研究[J]. 计算机工程, 2012, 38(21):40-44.
[26] Weatherspoon H, Kubiatowicz J D. Erasure Coding Vs. Replication: A Quantitative Comparison[M]// Peer-to-Peer Systems. Springer Berlin Heidelberg, 2002:328-337.
[27] Patterson D A, Gibson G, Katz R H. A case for redundant arrays of inexpensive disks (RAID)[C]// Proc. ACM SIGMOD Conference. 1988:109-116.
[28] 宋宝燕, 王俊陆, 王妍. 基于范德蒙码的 HDFS 优化存储策略研究[J]. 计算机学报, 2015(9):1825-1837.
[29] 李宽. 基于HDFS的分布式Namenode节点模型的研究[D]. 华南理工大学, 2011:21-22.
[30] Wang F, Qiu J, Yang J, et al. Hadoop high availability through metadata replication[C]// DBLP, 2009:37-44.
[31] Borthakur D. HDFS architecture guide[J]. 2008:22-26.
[32] 蔡斌, 陈湘萍. Hadoop技术内幕,深入解析Hadoop Common和HDFS架构设计与实现原理[M]. 机械工业出版社, 2013:23-30.
[33] 蒙安泰. 分布式文件系统中元数据管理机制的研究[J]. 电脑知识与技术, 2011, 07(35):9038-9040.
[34] Singh D, Singh J, Chhabra A. High Availability of Clouds: Failover Strategies for Cloud Computing Using Integrated Checkpointing Algorithms[C]// International Conference on Communication Systems and Network Technologies. IEEE Computer Society, 2012:698-703.
[35] Younge A J, Laszewski G V, Wang L, et al. Efficient resource management for Cloud computing environments[C]// Green Computing Conference, 2010 International. IEEE, 2010:357-364.
[36] 刘宏博. 基于纠删码的分布式文件系统恢复过程研究[D]. 中国科学院大学, 2015:12-13.
[37] Dimakis A G, Godfrey P B, Wu Y, et al. Network Coding for Distributed Storage Systems[J]. IEEE Transactions on Information Theory, 2010, 56(9):4539-4551.
[38] Wylie J J, Swaminathan R. System and method for determining the fault-tolerance of an erasure code: US, US 8127212 B2[P]. 2012:33.
[39] Ishengoma F R. The Art of Data Hiding with Reed-Solomon Error Correcting Codes[J]. International Journal of Computer Applications, 2015, 106(14):28-31.
[40] Kim S. Reversible Data-Hiding Systems with Modified Fluctuation Functions and Reed-Solomon Codes for Encrypted Image Recovery[J]. Symmetry, 2017, 9(5):61.
[41] Blaum M, Brady J, Bruck J, et al. EVENODD: an optimal scheme for tolerating double disk failures in RAID architectures[J]. Acm Sigarch Computer Architecture News, 1995, 22(2):245-254.
[42] KeesA.SchouhamerImmink, 伊明克, 徐端颐,等. 大容量数据存储系统编码[M]. 科学出版社, 2004:33-34.
[43] Xu L, Bruck J. X-code: MDS array codes with optimal encoding[J]. Information Theory IEEE Transactions on, 1999, 45(1):272-276.
[44] Wicker, Stephen B. Reed-Solomon Codes and Their Applications[J]. Communications of the Acm Cacm Homepage, 1994, 24(9):583-584.
[45] Wang Z, Wang D. NCluster: Using Multiple Active Name Nodes to Achieve High Availability for HDFS[C]// IEEE, International Conference on High PERFORMANCE Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing. IEEE, 2014:2291-2297.
[46] Deng P, Mei-Yi L I, Cheng H E. Research on Namenode Single Point of Fault Solution[J]. Computer Engineering, 2012, 38(21):40-44.
[47] 邓鹏, 李枚毅, 何诚. Namenode单点故障解决方案研究[J]. 计算机工程, 2012, 38(21):40-44.
[48] Calis G, Koyluoglu O O. On the maintenance of distributed storage systems with backup node for repair[C]// International Symposium on Information Theory and ITS Applications. IEEE, 2017:46-53.
[49] Borthakur D, Gray J, Sarma J S, et al. Apache hadoop goes realtime at Facebook[C]// ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June. DBLP, 2011:1071-1080.

万方数据

[50] 文艾, 王磊. 高可用性的HDFS-Hadoop分布式文件系统深度实践[M]. 清华大学出版社, 2012:56-68.
[51] Gupta T, Handa S S. An extended HDFS with an AVATAR NODE to handle both small files and to eliminate single point of failure[C]// International Conference on Soft Computing Techniques and Implementations. IEEE, 2016:67-71.
[52] Qu H, Lin M, Wang S C, et al. High availability using full memory replication between virtual machine instances on a network device:, US8806266[P]. 2014:33-36.
[53] Deng P, Mei-Yi L I, Cheng H E. Research on Namenode Single Point of Fault Solution[J]. Computer Engineering, 2012, 38(21):40-44.
[54] Khan M A, Memon Z A, Khan S. Highly Available Hadoop NameNode Architecture[C]// International Conference on Advanced Computer Science Applications and Technologies. IEEE, 2013:167-172.
[55] Soubhagya V N, Bhuvan N T. Highly Available Hadoop Name Node Architecture-Using Replicas of Name Node with Time Synchronization among Replicas[J]. 2014, 16(16):58-62.
[56] Khan M A, Memon Z A, Khan S. Highly Available Hadoop NameNode Architecture[C]// International Conference on Advanced Computer Science Applications and Technologies. IEEE, 2013:167-172.

万方数据

附录 1 攻读硕士学位期间申请的专利

（1）王少辉、胡文龙、肖甫、王汝传，一种扁平化的高可用 NameNode 模型的实现方法，2017108609986，2017.09；

万方数据

附录 2 攻读硕士学位期间参加的科研项目

（1）973、863 项目；

万方数据

致谢

时光荏苒，转眼间已经在南邮学习与生活了两年半。在读研究生的这段期间，实验室的老师和同学们都给予了我莫大的帮助，给我留下了难忘的记忆。离别之际，在此向所有曾经关心、帮助和指导过我的老师与同学致以我最诚挚的感谢！

首先我要感谢我的导师，同时也是我的论文的指导老师王少辉教授。王老师在平日的科研过程中给予了包括我在内的同学们极大的支持与鼓励。王老师特别注重培养我们的调研与分析问题能力，同时自己学术态度严谨，性格平易近人，但做事又坚决果断，给我们树立了非常好的榜样。特别珍惜与王老师的每次见面交流机会，探讨学术的过程中拓宽了我的视野，培养了我对科研浓厚的兴趣。在此，再一次向我的恩师表达我由衷的感谢之情！

同时要感谢同一教研室的肖甫老师和韩志杰老师，肖甫老师为我们教研室带来了国家863基金项目，让我们平日的科研更有方向性，更加聚焦。韩志杰老师作为项目全程的指导专家，在每一次例会中都能提出宝贵的指导意见与下一阶的段研究方向。在与两位老师相处合作的过程中使我学会了做工程的方法论，同时也为我的毕业论文打开了思路。

感谢304教研室的师兄师姐师弟师妹们，两年半的时间里，大家一起学习，共同生活，互帮互助。每当我在学习或者工作中遇到困难去请教你们的时候，你们总会耐心地帮我搜索资料寻找解决问题的办法。临近毕业找工作的那段时间，每天白天大家四处投简历，参加面试，晚上一起总结面试过程中需要注意的点供互相学习借鉴。除了学习与工作，有时我们也会在周末忙里偷闲的出去放松一下，让我见识到了你们的多才多艺。和你们相处的日子里，让我感觉到了自己的进步，同时也看到了彼此未来的可能性。

感谢我的父母，感谢你们一直在我身边陪伴我，感谢你们二十多年如一地养育我。成长的道路并不一帆风顺，没有你们的支持与理解，就没有我坚定走自己路的决心与勇气。今后的日子里我一定不辜负你们对我的殷切期望。

最后，感谢百忙之中还要抽空查阅我论文的老师、专家们，对你们表示我诚挚的谢意。

万方数据