国防科学技术大学

硕士学位论文

面向PB级存储系统的元数据集群管理容错方法研究与实现

姓名：王涌

申请学位级别：硕士

专业：计算机科学与技术

指导教师：刘仲

20071101

摘 要

随着计算机技术，信息技术和互联网络的发展，高性能计算、商业计算、大规模数据处理、信息处理等技术得到广泛的应用。集群系统因其较高的性能价格比和较好的可扩展性而受到越来越多的青睐。与此同时，这些应用对分布式数据存储提出了更大容量，更高性能，更高可用性的要求。

新兴的对象存储结构能够利用现有的处理技术、网络技术和存储组件提供空前的可伸缩性和聚合吞吐量，为构建新一代的大规模并行存储系统提供了基础。

本文在全面深入了解对象存储体系结构与现有对象存储系统的基础上，对基于对象存储体系结构的大规模集群存储系统所涉及的元数据集群管理的容错问题进行了深入的研究，提出了新颖的思想和解决方法。主要的贡献如下：

（1）提出一种面向 PB 级存储系统的基于目录对象副本的高可用元数据管理模型。通过采用高效的、并发的目录对象副本放置、更新、迁移策略以及管理机制，既保证实现了目录对象数据的可靠性和可用性，又增加了读取数据时的聚合网络带宽，提高读操作的性能。同时采用马尔可夫激励模型进行了定量可用性分析。实际的功能测试与性能测试表明该模型方法能够有效保证存储系统的高可用性，提高元数据服务器集群的整体访问性能。

（2）提出了一种对面向基于日志的元数据管理方法进行检查点操作的思想方法。通过对基于日志的元数据管理方法实施检查点操作，既保证了系统存储空间的有效利用，同时实现了系统的快速恢复，保证了系统的高可用性。实际的测试表明该方法能够充分地利用磁盘空间，提高系统的恢复时间，保证元数据的高可用性，提高元数据的访问效率。

关键字：元数据管理，高可用性，目录对象副本，基于日志的元数据，检查点，对象存储，容错

ABSTRACT

With the development of computer technology, information technology and the Internet, it makes high-performance computing, business computing, large-scale data processing, information processing and other technologies being used widely. Cluster systems become more and more favored because of their higher cost-performance and better scalability. At the same time, these applications bring forward the requirements of greater capacity, higher performance and higher availability for distributed data storage.

Emerging object-based storage architecture provides unprecedented scalability and aggregation throughput by taking advantage of existing processing technologies, network technologies and storage components. It provides the foundation for the construction of a new generation's large-scale parallel storage system.

In this paper, on the basis of comprehensive and in-depth understanding of object-based storage architecture and existing storage systems, it has an in-depth study for the fault-tolerance of metadata cluster management which is involved in the object-based large-scale cluster storage system, and it puts forward some new ideas and solutions. The main contributions are as follows:

(1) A high availability metadata management model based on directory object replica is proposed for PB-scale storage system. By using with the high efficient and concurrent placing, updating and migration strategy and management mechanism of directory object replica, this model not only assures the directory object data reliability and availability but also improves reading aggregation network bandwidth and reading operating performance. We quantify the availability of storage system by using Markov Reward Model. The practical functional and performance testing results indicate that the model can effectively guarantee the high availability of storage system and improve the overall access performance of metadata server cluster.

(2) An efficient checkpointing scheme for journal-based metadata management is proposed. By implementing checkpointing to journal-based metadata management, the system will be assured the effective storage space utilization, while achieving a rapid system recovery, the high availability of system is guaranteed too. Actual testing results show that this approach can utilize disk space fully and improve the system's recovery time, and ensure the high availability of metadata, improve the access efficiency of metadata.

Keywords: Metadata Management, High Availability, Directory Object Replica, Journal-based Metadata, Checkpoint, Object-based Storage, Fault-tolerant

图目录

表目录

独 创 性 声 明

本人声明所呈交的学位论文是我本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和撰写过的研究成果，也不包含为获得国防科学技术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谢意。

学位论文题目：面向PB级存储系统的元数据集群管理容错方法研究与实现

学位论文作者签名：王 [illegible]　　日期：[illegible]年[illegible]月[illegible]日

学位论文版权使用授权书

本人完全了解国防科学技术大学有关保留、使用学位论文的规定。本人授权国防科学技术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、汇编学位论文。

（保密学位论文在解密后适用本授权书。）

学位论文题目：面向PB级存储系统的元数据集群管理容错方法研究与实现

学位论文作者签名：王 [illegible]　　日期：[illegible]年[illegible]月[illegible]日

作者指导教师签名：刘仲　　日期：2007年12月18日

第一章 绪论

§1.1 课题研究背景及意义

1.1.1 课题研究背景

在最近几年里，集群计算在高性能计算、商业应用、信息服务等领域得到了广泛的应用。大量的集群计算应用不仅是计算密集型，也是数据密集型应用。随着处理器和网络通讯技术的飞速发展，极大的提高了集群计算的处理和通讯能力，而受传统存储结构的限制，其计算能力不能得到充分体现。集群计算对可伸缩、高性能、高可用、安全、共享数据的存储需求对现有的存储结构提出了巨大的挑战。

新兴的对象存储结构[1][2]提供了基于对象的访问接口，有效地合并了网络附加存储系统（NAS，Network Attached Storage）和存储区域网（SAN，Storage Area Networks）存储结构的优势，是构建新一代存储系统的基础。对象存储结构无论是在存储容量，存取性能还是在高可用性方面都能得到良好的保证。

对象存储结构利用现有的处理技术、网络技术和存储组件提供空前的可伸缩性和聚合吞吐量，为构建 PB(10^{15} 字节)级规模的存储系统提供了基础[3][4]。

对象存储体系结构如图 1.1：

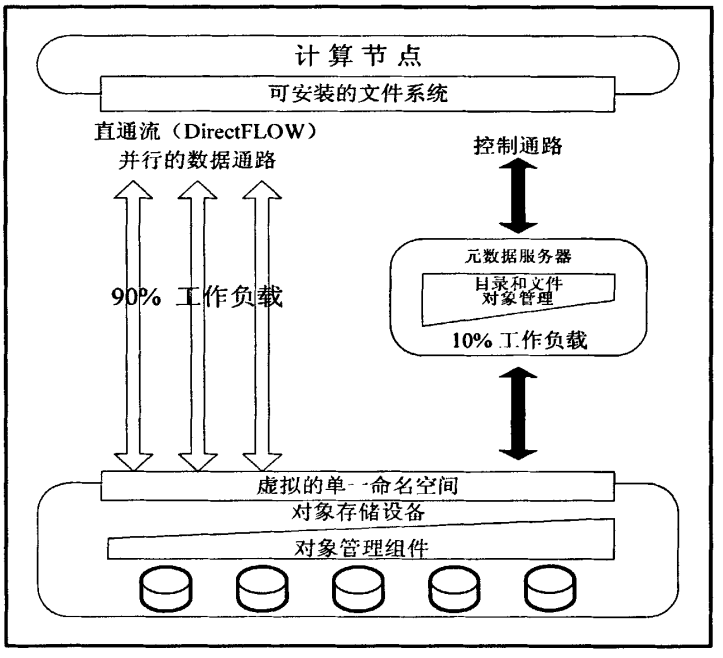

图 1.1 对象存储体系结构

文件是由两部分组成的：文件的元数据信息和文件实际的数据内容。在进行文件访

问时，文件的实际数据内容实际上占据了系统工作负载的大约 90%，而用于计算节点对文件系统进行浏览的文件和目录层次结构----文件元数据，占据了系统工作负载的剩余的大约 10%。

对象存储体系结构的核心就是采用专用服务器模式，将文件访问的数据流（文件实际数据的传输路径）与控制流（文件的元数据控制信息路径）进行有效的解耦分离，将文件的元数据控制信息集中到元数据服务器（MDS，Metadata Server）来管理，而实际的文件数据则分布到对象存储设备（OSD，Object-based Storage Devices）集群中。对象存储体系结构的主要优势就是增加了元数据服务器的可扩展性，由对象存储设备（OSD）具体负责文件的实际数据内容。每增加一个对象存储设备（OSD），就增加了 90%的文件数据管理资源，很好地支持了对象存储设备（OSD）的空间扩展。

由于对象存储体系结构采用灵活有效的机制（分布式文件系统）来管理系统存储资源，因而为系统客户提供高扩展能力（大容量）和高吞吐率（高性能）的数据访问。

Google 公司采用的 GFS[5]，Cluster File System / IBM Blue Gene 采用的 Lustre[4][6][7][8] 等分布式文件系统均采用对象存储体系结构并得到了非常成功的实施和应用。

1.1.2 课题研究意义

高可用性要求分布式系统中的部分节点失效时，其它节点能够尽可能最大限度地提供服务。数据高可用性是各种高可用性的基础，它要求一个或几个节点不能提供服务时，这些节点的数据能够在其它节点上得到，从而使整个系统能够提供一种持续的数据服务。

作为分布式系统中关键的组成部分：分布式文件系统，高可用性要求分布式文件系统中的部分节点失效时，其它节点能够尽可能最大限度地提供服务。

分布式文件系统中实现数据高可用性的基本原理是：在系统中引入数据冗余副本以及对系统实施检查点操作保存系统在某一时刻的状态。通过数据的冗余备份或校验以及保存的检查点状态实现发现错误并进行快速的数据恢复的目的。它能满足实际情况中一个或几个节点不能提供服务时，这些节点的数据能够在其它节点上得到，或者回滚到系统之前最近的时间点处的系统状态，从而使整个系统能够提供一种持续的数据服务。

§1.2 研究内容以及研究思路

1.2.1 研究内容

针对分布式文件系统中高可用性的实现目的，我们在基于对象存储的集群文件系统 OCFS（Object-based Cluster File System）[9]基础上，设计与实现一个高可用的元数据集群管理容错系统，它为分布式文件系统中元数据的高可用性提供支持。

它的基本思想是通过采用多个目录对象副本的方式以及对系统元数据实施检查点操

作的方法来实现文件系统元数据的高可用性，为元数据对象的高可用性提供支持。同时对原型系统的效果进行了测试。

1.2.2 研究思路

为实现系统的高可用性，本文拟对分布式文件系统中的元数据对象采用元数据对象副本的方法以及对系统的元数据对象实施检查点的方法来实现。

1.2.2.1 元数据对象副本方法

目前，高可用的元数据服务器集群的系统结构从是否共享存储的角度分为两类。一类是以 GFS，Lustre 为代表的非共享存储，元数据的可靠性是通过增加备份服务器实现的，系统的成本较高。另一类是通过分布式文件系统共享磁盘阵列或 SAN 来实现的共享存储，这种方法的不足是多台元数据服务器对共享元数据的并发冲突访问，容易形成 I/O 瓶颈，降低系统的整体性能。

本文的方法是在分布式文件系统的基础上通过对元数据采用多个元数据对象副本的方式来实现元数据的高可用性。这样既避免了通过增加备份服务器来实现元数据的可靠性而带来的较高的成本，同时又避免了多台元数据服务器对共享元数据的并发冲突访问而造成的系统整体性能的降低。

1.2.2.2 检查点方法

近年来，基于对象的存储体系结构在分布式存储领域，分布式文件系统中逐渐得到广泛应用。基于对象的存储使得文件数据块的分配被封装到智能的网络附加磁盘中，有效地在成千上万台对象存储设备（OSD）上进行数据块的分配。这种存储体系结构将文件元数据的访问从文件实际数据块的读写操作中解耦分离出来，使得系统具有更良好的以及更高的效率及可扩展性。随着系统逐渐扩展到 PB 级（10^{15} 字节）系统，保持文件系统的高效率的访问是十分重要的。而对元数据进行检查点操作以实现“在线（on-line）”的保存能力正逐渐成为健壮性系统、容错系统中的一个基本组成部分，同时也是保障实现高效率的系统访问的一种策略机制。

元数据对象检查点方法将文件系统的状态“在线（on-line）”保存集成到大型的基于对象的分布式文件系统中。基于检查点的概念，通过在确定的时间间隔内对文件系统的当前状态实施检查点操作来对文件系统的当前状态予以永久性的保存。当发生系统失效或其它故障后，系统可以回滚至最近一次保存的系统状态，根据该系统状态进行恢复，重新开始继续运行，而不是从头开始执行，从而实现保证了系统的健壮性、容错性及高可用性，同时也保证了系统的高效率访问。

§1.3 研究目标

对象存储文件系统的核心是将数据传输路径与控制信息路径分离，将控制信息（元数据）集中到元数据服务器（MDS）管理，而实际的数据对象分布到对象存储设备（OSD）集群中。元数据对象的处理机制与存储系统结构息息相关，所以，研究基于对象存储的元数据高可用性非常重要。

由于对象存储文件系统中一个潜在的系统性能瓶颈是对元数据的访问，所以在大规模的存储系统中，尽管元数据的数据量大小同整个存储系统的整体数据容量大小相比起来相对较小，但是统计数据表明，在所有的文件系统操作访问中，对元数据的访问操作约占所有文件访问操作的 50%－80%[10]。所以，元数据访问的高可用性对整个存储系统在提高性能和保证有一个良好的可扩展性、可伸缩性以及快速的恢复能力是至关重要的。

本文的研究目标是研究元数据集群管理的容错方法，包括：

- 高可用的元数据副本对象管理模型
- 高效的元数据对象检查点方法

§1.4 本文的工作

本文的研究内容包括：研究基于对象存储的元数据对象副本的高可用性；研究高效的基于日志的元数据对象管理的检查点方法。本文的具体工作和创新主要体现在：

（1）深入分析和研究了对象存储体系结构的对象模型、访问模型和优越特性，在现有对象存储体系结构的基础上，提出了基于元数据对象副本的高可用元数据管理模型。与现有的存储系统相比，该结构能够实现元数据对象的高可用性，支持元数据服务器集群的容错管理。

（2）提出对分布式文件系统的元数据对象实施检查点的方法：在应用程序的正常运行过程中，定期（或不定期）地设置检查点，保存系统当时的一致性状态（设置检查点）。当系统发生故障，进行回滚恢复，系统重新开始运行时，应用程序可以回滚到最近一个检查点处继续运行，而不必从头开始程序的执行，从而有效地提高系统的可用性，实现了元数据服务器集群的容错。

总之，本文以系统研究面向基于 PB 级存储系统的元数据集群管理容错方法为出发点，对其中的若干关键技术进行了深入的研究，着重探讨了元数据集群中元数据对象的高可用性，并通过原型系统验证了研究成果的有效性和正确性，达到了预期的研究目标，为实现大规模的元数据集群容错管理提供了可靠的技术基础和保障。

相关的研究工作已经整理为两篇论文（《面向 PB 级存储系统的高可用元数据管理模

型》，《一种面向基于日志的元数据管理的高效检查点方法》），分别发表在“2007 中国计算机（CNCC）大会（苏州）”会议论文集及“2007 年全国高性能计算学术年会（深圳）”会议论文集中。

§1.5 论文的组织结构

论文共分五章。各章的主要内容安排如下：

第一章“绪论”。首先分析本文的研究背景及意义，确定本文的研究内容和思路，接着介绍了本文的研究目标以及所做的研究成果及主要贡献，最后介绍本文的组织结构。

第二章“相关的研究工作”。对原型系统中所涉及的相关的研究技术进行了介绍，最后对元数据容错管理的研究现状进行了分析和总结。

第三章“一种容错的元数据集群管理系统结构”。首先通过对实际需求进行的分析，得出有元数据集群的需求，提出了本文的元数据集群系统设计的基本思想以及体系结构，对系统的各个组成部分以及本文的元数据集群系统的容错模块进行了说明，接着描述了对原型系统的设计与实现，最后对本章内容进行了总结。

第四章“基于目录对象副本的高可用元数据管理模型”。提出在元数据服务器集群中使用元数据对象副本来实现元数据的高可用性的思想，提出了基于元数据对象副本的高可用管理模型框架结构，对元数据对象副本管理策略的关键技术进行了描述和本文相应的解决方案，同时对模型使用马尔可夫激励模型进行了定量的高可用性分析，最后给出实验测试与性能分析的结果，结果说明原型系统具有高可用性以及较好的性能表现。

第五章“基于日志的元数据管理的高效检查点方法”。根据传统进程检查点的概念，提出对系统中的元数据对象实施检查点的思想，对版本文件系统的不足进行了分析，提出了相应的改进方法----实施元数据对象检查点操作，并对该方法的关键技术以及实施的过程步骤进行了描述和实现，最后给出实验测试与性能分析的结果，结果证明该思想方法能够保证系统具有高可用性以及快速的恢复能力。

第六章“结束语”。总结本文取得的主要成果，提出有待于进一步开展的研究工作方向并进行了展望。

第二章 相关的研究工作

§2.1 文件系统

文件系统[1]主要是用来组织文件和目录的。它是一种存储数据的方法，它采用分层目录的结构来存储文件，由一个根目录和许多子目录、文件组成。目录是存放一组文件的“容器”，也包括目录本身，由此形成一个大型的树形结构。程序文件、数据文件和其它目录甚至设备文件等都可视为目录中的实体。

在逻辑卷上创建文件系统之后，用户可以通过文件名按照文件的逻辑结构，使用简单、直观的操作存取所需要的信息，从而使用户摆脱了对物理卷、逻辑卷的 I/O 操作指令的细节和存储介质的特性的关心。从这个意义上来讲，文件系统给用户提供了一个操作外部存储设备的界面。

2.1.1 文件系统的概念

文件系统是文件、目录和其它数据结构的集合，记录着目录和文件的位置信息。一个文件系统是建立在一个逻辑卷上的，这个文件系统的所有目录和文件都保存在这个逻辑卷中，所以文件系统的大小最大不可能超过逻辑卷的大小。

文件系统是一个树形结构，因此，可把文件系统看成是一棵树。每个文件系统都有一个“树根”（称为文件系统的根），文件系统中的每个目录就成了这棵树的树枝，目录中的每一个文件就是树的叶子，因此，从文件系统的根（树根）到一个特定的文件（树叶）之间就存在一条唯一的路径，用户通过这条路径可以访问文件系统中的某个特定目录或文件。当系统成功启动之后，就会存在一个根文件系统（/），用户可以随时使用根文件系统中的任何文件或目录。除了根文件系统之外，其它的文件系统都是独立于根文件系统，用户要访问一个非根文件系统就必须把它安装在根文件系统的一个空目录上，或者其它某个非根文件系统（必须是已安装的）的一个空目录上，用户只有通过这个空目录才可以访问它。

非根文件系统的安装是指把一个文件系统的根链接到一个具体的空目录上，使得用户能够通过这个空目录来访问文件系统中的任何目录和文件。这个空目录称为安装点（Mount Point）。一个文件系统只有通过安装才能被用户使用，因此，在使用某个文件系统中的任何一个文件或目录之前，必须先安装这个文件系统。安装文件系统可用 mount 命令手工安装。mount 命令可以安装本地和远程的文件系统，只有在安装之后，文件系统才可以被访问。如果在/etc/fstab 中定义了文件系统，则系统启动时会自动安装它。当没有进程或用户使用文件系统的情况下，用 umount 命令可以卸载本地和远程文件系统。

安装和卸载文件系统的用户必须是root用户。

一般情况下，安装点必须是一个空目录，安装点是被安装的文件系统的根。如果安装点不是一个空目录，那么在这个安装点安装了一个文件系统之后，原来位于这个安装点中的文件或目录将暂时无法被访问，除非卸载了这个文件系统。

文件系统是一个逻辑上的概念，通常建立在一个逻辑卷上，存放文件系统的设备可以是硬盘，也可以是光盘，还可以是RAM Disk（内存磁盘）。文件系统之间不能相互覆盖，而是分散在不同的（逻辑）设备上。

除文件和目录之外，文件系统还包括超级块，索引节点（i节点，inode），数据块，分配位图和分配组，一个分配组包含磁盘i节点和碎片（Fragment）。

对文件系统的管理操作有：创建/删除文件系统、安装/卸载文件系统、校验文件系统、备份/恢复文件系统、显示文件系统和修改文件系统属性等。

2.1.2 采用文件系统结构的原因

（1）层次目录结构的文件系统易于扩充，即易于加大文件系统的空间；

（2）从特性上来讲，它可以放在磁盘的任何位置上，没有位置的限制；

（3）管理一个文件系统比管理这个文件系统中的每一个目录更有效，更方便；

（4）通过文件系统可以限制用户对存储空间的使用，防止用户无限制地使用存储空间；

（5）能够保证整个文件系统结构的完整性，当一个文件系统出现问题，不会影响到其它文件系统；

（6）文件系统按名存取文件，用户只提供文件名，文件系统会在相应的物理硬盘上建立一个文件，或者从物理硬盘上读出一个文件，用户不用知道文件在物理硬盘上的具体位置，只需要知道文件名和文件所在的目录；

（7）文件系统对文件有保护，保密措施，安全可靠；

（8）可以实现文件共享，节省空间和时间开销。

2.1.3 根文件系统的结构

UNIX的文件系统是一个包含目录和文件的分层结构（文件树结构），类似一个倒立的树，树根在顶部，树枝在底部，树枝表现为目录，树叶表现为文件。文件系统树用目录来组织大量的文件数据和程序，允许用户在同一时刻可以对多个目录或文件进行操作。

操作系统的整个文件系统分成基本文件系统和可装载卸载的文件系统两部分。基本文件系统就是一般所说的根文件系统，它是整个文件系统的基础，一般固定在内置的第一块可引导硬盘上。各个可装载卸载的文件系统一般存储在可移动的硬盘上。一旦启动

系统之后，根文件系统不能被卸载，其它文件系统可以随时被安装卸载。这种方式有利于扩充整个文件系统，使用灵活方便。

无论是根文件系统还是可装载卸载文件系统都有自己独立的目录结构，每个可装载卸载文件系统都有自己的根目录，它的根安装在根文件系统的一个目录上，这个目录称为可装载卸载文件系统的安装点。用户只有通过安装点才能够访问可装载卸载文件系统。图 2.1 是 root 文件系统的结构图：

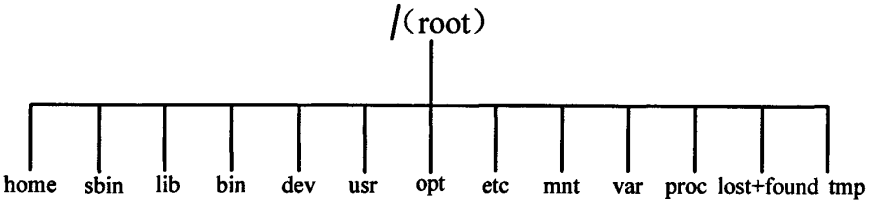

图 2.1 root 文件系统结构

一个文件系统只能有一个根文件系统，位于整个文件系统树结构的顶部，它包括系统运行所必备的文件和目录，其中有设备目录、系统引导程序，以及安装其它文件系统的安装点。

§2.2 索引节点

每一个文件（目录是一个特殊的文件）有一组控制信息，如文件的类型、访问权限、用户 ID、组 ID 和文件的链接数等。这组控制信息一般应保存在文件的目录项中。但是在 UNIX 系统中，为了加快对文件目录的搜索速度，便于实施文件共享，将它们从目录项中分离出来，单独构成一个数据结构----文件控制块或索引节点[11]（Index Node，简写为 i-node），一般称为磁盘 i 节点或 i 节点。在文件系统中开辟了许多个连续的区域，用来存放 i 节点，在每个区域中的 i 节点是按顺序编号。每个文件都有一个对应的 i 节点来存放其控制信息和数据块的位置信息。因此，从本质上来说，一个 i 节点是一个指向文件数据块的指针。

文件系统有一个管理所有空闲 i 节点的位图表和一个管理所有空闲数据块的位图表。i 节点只存放文件的控制信息，而文件的内容存储在实际的磁盘数据块中，在 i 节点中有一些指向这些磁盘数据块的指针，因此要访问文件的内容是通过 i 节点来寻找文件的数据块。

在文件系统中，文件的目录项只包括文件名和文件的 i 节点号，用户根据文件名来访问文件，系统首先根据文件名在目录中查找对应的 i 节点号，然后再根据 i 节点号确定 i 节点位置，最后根据 i 节点中的地址找到存放文件具体内容的数据块。

2.2.1 索引节点的结构

文件系统的索引节点（i 节点）是一个 128 字节的数据结构，在 i 节点列表中，由操作系统为每个 i 节点产生一个惟一的 i 节点号，用来标识每一个 i 节点。

每个文件都有一个 i 节点，i 节点包含着文件的存取权限、文件类型、创建、修改和访问文件的时间、文件的链接数、文件的大小和文件数据块在磁盘上的地址等。表 2.1 中列出的内容是一个 i 节点中的主要内容。

表 2.1 一个 i 节点包含的主要内容

i 节点字段	描述
i_generation	生成的编号，i 节点号
i_mode	文件的类型和访问权限，用位表示，一般可调用 stat()获得
i_nlink	所建立的文件链接的数量。如果是 0，则该 i 节点可用于分配给其它文件
i_uid	文件所有者的 ID，用户 ID
i_gid	文件所属组的 ID
i_size	文件的大小，以字节为单位
i_blocks	被该文件实际使用的块数
i_mtime	最近一次修改该文件内容的时间
i_atime	最近一次访问该文件的时间
i_ctime	最近一次修改 i 节点的时间
i_rdev	特殊文件（设备文件）的主、次设备号
i_rdaddr[8]	指向数据的实际磁盘地址，也称地址指针
i_rindirect	指向间接地址块的实际磁盘地址（如果需要用间接地址块）

磁盘 i 节点的 i_rdaddr 字段包含着 8 个磁盘地址，这些地址指向分配给这个文件的前 8 个数据块。i_rdaddr 字段中的磁盘地址指向一个间接地址块，这个间接地址块可以是单次间接地址块，也可以是二次间接地址块，因此有 3 种可能的寻址方法：直接寻址、单次间接寻址和二次间接地址寻址。

与设备相关的文件的 i 节点与一般普通文件的 i 节点所包含的信息有一些细微的不同差别。与设备有关的文件称为特殊文件或设备文件。在设备文件的 i 节点中不包含数据块的地址（i_rdaddr[8]和 i_rindirect），但是有设备的主号码和次号码（i_rdev），设备的主、次设备号保存在 i_rdev 字段中。

i 节点记录着文件的属性以及存储文件内容的磁盘块地址，因此修改文件的内容，一定也会修改它的 i 节点。只改变文件的内容而不改变文件的 i 节点是不可能的事，但是存在着只改变 i 节点而不改变文件内容的情况。

i 节点中不包括文件名和文件路径信息。文件名没有保存在 i 节点中，而是保存在文件所在的目录文件中。在目录类型的文件中存放着属于这个目录的每个文件的名字和 i 节点号，目录中的记录把文件名和它的 i 节点做了映射，因此目录中记录着文件名与 i 节点的对应关系。任何 i 节点可以用 link 命令或 symlink 函数把一个 i 节点链接给许多文件名，也就是在目录中添加多条关于一个 i 节点的记录。用 ls –i 命令可以查看文件的 i 节点号。

通常情况下，当 i 节点的链接数（i_nlink）等于 0 时，就会释放这个 i 节点，链接表示与这个 i 节点关联的文件名。当链接数为 0 时，由这个 i 节点指向的所有数据块都被释放掉，并在空闲数据块位图表中做登记，还在空闲 i 节点位图表中将已释放的 i 节点标记为空闲状态。

2.2.2 磁盘索引节点和内核索引节点

在操作系统中，当打开一个文件时，操作系统就会在内核中创建一个 i 节点，把这种 i 节点称为内核中的 i 节点（In-core i-node，简称为内核 i 节点，也可称为动态的内核 i 节点（In-core i-node)），这是相对于磁盘 i 节点（Disk i-node，也可称为静态的磁盘 i 节点（Disk i-node)）而言的。磁盘 i 节点保存在磁盘上，内核 i 节点位于系统的内核区，即文件系统的缓存区中。内核 i 节点包含这个文件对应的磁盘 i 节点中所有字段的内容，此外还增加了一些用于管理内核 i 节点的字段。

2.2.2.1 磁盘索引节点

在文件系统的内部，每个文件都表现为一个 i 节点。文件系统的 i 节点在磁盘上是以一种静态格式存在的，它包含着这个文件的访问信息，还指出这个文件的数据块的实际磁盘地址。一个文件系统中包含的可用 i 节点数取决于这个文件系统的大小、分配组的大小等。这些参数是在用 mkfs 命令创建文件系统时给定的。当一个文件系统中所有的可用 i 节点全部用完之后，就不能再创建一个文件或目录，即使这个文件系统还有很多空闲的存储空间。用 df –v 命令可以查看一个文件系统中可用的 i 节点数。在 /usr/include/linux/fs.h[45]头文件中定义了磁盘 i 节点的数据结构。

2.2.2.2 内核索引节点

无论是操作系统还是用户的应用程序打开一个文件，操作系统为了便于访问这个文件的 i 节点，就把这个文件的磁盘 i 节点中的信息复制到一个内核 i 节点中。在内核 i 节

点中还包含了一些用于跟踪内核 i 节点的字段。内核 i 节点包含的跟踪信息有：

（1）内核 i 节点的状态，下面简要地说明所包含的状态：

一个 i 节点锁

一个等待 i 节点被解锁的进程

在这个文件对应的 i 节点中所做的修改

对这个文件数据的修改

（2）包含这个文件的文件系统对应的逻辑设备号。

（3）用于标识这个 i 节点的 i 节点号。

（4）引用数（Reference Count）。当引用数的值等于 0 时，这个内核 i 节点就被释放。

当释放一个内核 i 节点时（例如调用了一个 close（）函数关闭了这个文件），这个内核 i 节点的参考数就会被减 1，直到这个内核 i 节点的参考数变为 0 时，才从内核 i 节点表中释放这个 i 节点。如果内核 i 节点 i 与磁盘 i 节点中的内容不相同，那么操作系统就会用内核 i 节点中的内容覆盖磁盘 i 节点。

§2.3 对象存储体系结构

2.3.1 对象存储体系结构简介

对象存储体系结构是基于数据对象的，数据对象将用户数据（文件）和该数据的属性进行封装。将数据和数据的属性进行封装的这种方法使得对象存储系统可以在基于每个文件的基础上对文件的布局和服务质量做出决定，这样提升了系统的灵活性和可管理性。对这些对象进行存储、检索和解释的设备是对象存储设备（OSD）。对象存储设备设计的独特特点在于它不同于其它传统标准存储设备（如光纤通道（Fiber Channel，FC）或 IDE）的传统基于块的接口。通过将底层的存储功能迁移到存储设备自身上并通过一个标准的对象接口来访问存储设备的方法，对象存储设备可以做到：

（1）在存储层进行智能的空间管理，这样可以使得对象存储设备在将数据分配至存储介质上时可以进行全局智能的考虑；

（2）可以进行“数据感知（Data-aware）”的预取并进行缓存。

最终对象存储系统有以下优点：

（1）多个客户端同时访问时可以拥有较好的共享访问能力和鲁棒性；

（2）通过数据路径的迁移可以实现很好的可扩展性；

（3）拥有很强的细粒度端到端的安全性。

所以基于对象存储设备（OSD）的存储体系结构能够为科学计算、工程应用提供必

要的文件共享能力，同时保证了较高的性能和良好的可扩展性。

2.3.2 对象存储体系结构的组成部分

对象存储体系结构主要由以下五部分组成：

（1）数据对象。它包含了数据的实际内容和额外的信息，这些额外的信息使得数据可以进行自治和自我管理；

（2）对象存储设备（OSD）。它是一种智能磁盘，能够对数据对象进行存储和服务，而不仅仅是将数据存放在磁道和扇区上；

（3）分布式文件系统（Distributed File System）。它是与计算节点集成在一起的，接收来自操作系统的 POSIX 文件系统命令和数据，直接访问对象存储设备（OSD）并将对象在多个对象存储设备（OSD）上进行分布；

（4）元数据服务器（MDS，Metadata Server）。在众多的计算节点中起着中间角色的作用，使得计算节点可以共享数据，同时负责在所有的节点上保证缓存的一致性；

（5）网络。它负责将计算节点连接到对象存储设备（OSDs）和元数据服务器（MDSs）上。

对象存储系统的体系结构逻辑视图[12]如图 2.2：

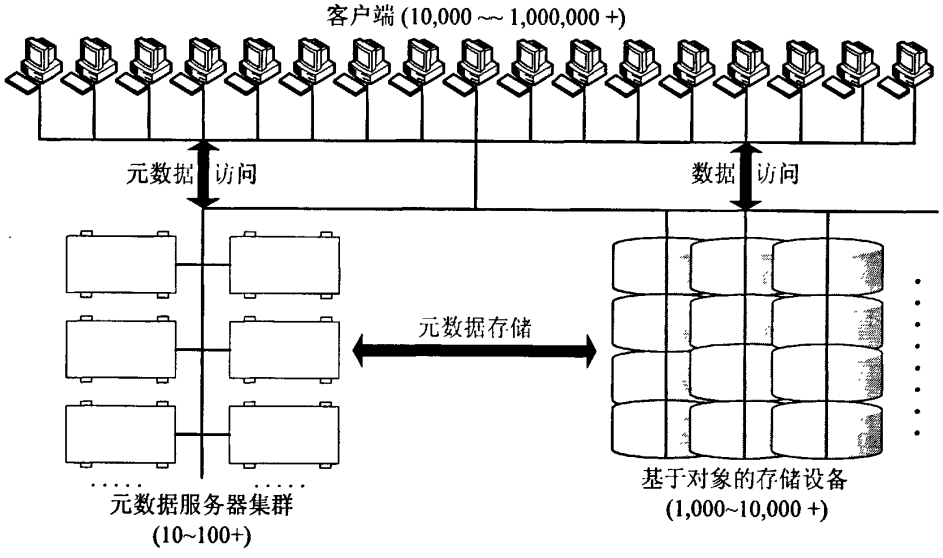

图 2.2 对象存储系统的体系结构

2.3.3 元数据服务器

元数据服务器（MDS）控制着计算节点和存储在对象存储设备（OSD）上的数据对象的交互，协调经过恰当认证的计算节点的访问，并且为用户（客户端）对相同的文件的访问保持缓存的一致性。在网络附加存储（NAS）系统中，元数据服务器是数据路径

的一个集成部分，这导致了随着文件访问流量的增加，元数据服务器成为显著的瓶颈。对象存储体系结构的方法是将元数据服务器从数据路径中分离出来，这样就实现了系统的高吞吐量和线性的扩展能力。这种线性的扩展能力的思想是来自于存储区域网(SAN)，它允许客户端同存储设备进行直接的交互。元数据服务器为对象存储体系结构提供以下服务：

（1）认证。元数据服务器的首要功能作用就是对加入存储系统的对象存储设备进行认证和标识。元数据服务器必须对对象存储设备提供认证证书并且周期性地对这些认证证书进行检查/更新，以确保对象存储设备都是有效的。同样地，当一个计算节点试图访问存储系统时，元数据服务器必须对计算节点进行验证和认证。

（2）文件和目录访问的管理。元数据服务器为计算节点提供存储系统的文件结构。当计算节点对一个特定的文件进行操作时，元数据服务器将对与文件相关的访问控制和访问权限进行检查，并为发出请求的计算节点提供文件映射和访问能力。文件映射是由对象存储设备列表和它们的 IP 地址组成，包括所需要访问数据的数据对象。访问能力是为计算节点提供的一个安全的、加密的令牌（token），该令牌在每次交互中都由对象存储设备进行检验。令牌（token）描述了允许计算节点以什么样的权限，多长的时间来访问哪些对象存储设备上的数据对象。

（3）缓存一致性。为取得很高的性能，计算节点通常会请求相关的数据对象，进而可能会请求超出本地缓存的数据。如果多个计算节点使用相同的文件，则必须采取相应的步骤措施，以保证当文件被任意一个计算节点变更时，本地的缓存也得到了更新。元数据服务器使用分布式对象锁机制或回调机制来提供这种服务。当一个计算节点向元数据服务器要求对一个文件或一个文件的一部分进行读或写要求时，计算节点就向元数据服务器注册一个回调函数。如果该文件的权限允许多个写用户，而且被另外一个计算节点更改了，那么元数据服务器就向所有打开这个文件的计算节点产生一个回调函数，回调函数使这些计算节点的本地缓存失效。这样，如果一个计算节点对已更新的文件进行读访问操作的话，那么它必须重新读取对象存储设备以刷新它的本地缓存的数据的拷贝，这样才保证了所有的计算节点访问的是相同的数据。

（4）空间管理。元数据服务器同时必须跟踪整个系统中对象存储设备的空间利用平衡情况和利用率的平衡情况，以确保整个系统最优地使用了有效的磁盘资源。当计算节点试图创建一个数据对象时，元数据服务器除向认证的对象存储设备写入新的数据对象之外，还必须确定如何对新文件进行最优的放置策略。由于计算节点在进行文件创建时不能确切知道文件的大小，所以元数据服务器将为计算节点提供空间配额（quota）服务。

这样就可以使计算节点将创建数据和写数据在一步操作中完成，从而使得写操作的性能得到了最大化。一旦文件进行了关闭操作，任何超出的磁盘空间配额都会得到恢复和释放，这样在进行关键写操作时就保证了系统的最大的性能。

（5）可扩展性。对试图在容量和性能方面进行扩展的存储系统而言，元数据的管理是一个关键的体系结构问题。由于对象存储体系结构将文件/目录的管理从块/扇区的管理中分离出来，所以它可以比其它的存储体系结构有更大更高层次的扩展性。它将块/扇区的管理分布到对象存储设备中（这占据了大约 90%的系统工作负载），将文件/目录的元数据管理（这占据了大约 10%的系统工作负载）分配到单独的服务器上，这些单独的服务器也可以以可扩展的集群方式来实现。元数据服务器的可扩展性是实现整个对象存储系统的可扩展性、实现对象存储设备在空间上和性能上的平衡的关键。

由以上所述可以看出：由于元数据是所有的计算节点和存储节点集中访问的内容，所以元数据管理能力对任何共享存储系统来说都是典型的瓶颈。而在对象存储体系结构中，对这一点做了很大的创新。

§2.4 元数据管理容错相关研究

分布式文件系统一般要提供与集中式文件系统同样的服务，用户不必关心系统中到底有多少个文件服务器以及各个服务器的物理位置和功能，这些对用户而言都是透明的，用户看到的文件系统与单个处理器上的文件系统应该没有丝毫差别。但由于性能，可靠性和分布式等原因，还要考虑的服务有：备份和多拷贝的一致性等问题。从而引入了元数据的容错问题研究。

目前对元数据的容错研究主要方法有采用多个备份的方法，对元数据实施擦除码（Erasure Code）[13]、纠错码方法以及采用基于日志的元数据管理方法等。

2.4.1 元数据备份方法相关研究

虽然已经实现了许多分布式文件系统，它们所采用的技术及基本成分也不尽相同。但仍可以提出一个通用的分布式文件系统的系统结构。在该通用的分布式文件系统结构中，其中的备份服务组成部分----数据一致性和多副本修改模块，它的主要功能作用是：共享文件时提供数据一致性（当多个用户访问同一个文件时必须采用并行控制机制以保证数据的一致性，特别是文件多个副本之间的数据一致性）和提供多重拷贝修改机制[14]。

由于性能和可靠性原因，文件服务的设计者通常使用备份文件。备份文件是指在不同的服务器中有多个拷贝的文件。使用备份文件可减少通信量而提高性能，通过向本地用户提供拷贝文件而提高响应时间，不同的服务器可使用几个用户访问同一个文件，因此提高了系统吞吐率。

文件备份提高了系统利用率，可以在几个服务器上访问同一个文件，从而提高了可靠性，还降低了服务器和通信故障的影响。

提供文件备份会带来另外一个问题，称为多拷贝修改问题。当修改备份文件时，维护多个副本的一致性是备份文件系统设计的一个主要问题。备份服务器之间的相互一致性是分布式文件系统的一个难题，它需要在备份服务器之间维持对象多重拷贝的一致性。（这意味着一个文件的每个拷贝都需要一个时间标记或版本号，而且访问和修改算法必须保证使用最新的版本。）

文件备份有两种可用的结构：主 / 从结构和分布式结构。

在使用主 / 从结构的系统中，对每一个备份文件都有一台明显的主服务器和几台从服务器。主服务器存储文件的主拷贝，并为所有的修改请求服务。每个文件的从拷贝通过从主拷贝处获得拷贝来进行修改，或从主服务器接收修改信息进行修改。此方法可用于文件修改不频繁的系统中，并且改变能被中心点所接受。

在有些应用中，文件会被不同位置的客户频繁地修改。如果采用主 / 从结构，在不同客户进行连续修改同一个文件时，主服务器必须不停地传播修改结果，占用了大量的网络传输带宽，因此需要分布式修改控制机制。分布式备份服务中存储文件拷贝的任何服务器都可接受对文件的改变请求，并将此改变通知其它拷贝，使用户对数据有一致性的视图。

经过 20 多年的研究，分布式文件系统的服务器结构发展到专有服务器系统占据主流的阶段。因而，目前大多数系统所使用的结构仍为主 / 从结构。主服务器为采用专门的服务器提供文件系统元数据的管理，从服务器为采用专门的服务器提供文件数据存储服务。系统的客户端（应用服务器）仅运行系统的各种应用服务。其典型的代表有：Google FS（Google），Storage Tank（IBM），Lustre（Cluster File System），Ceph（University of California，Santa Cruz）[15]等。而文件数据对象副本按配置规则保存在众多的从服务器上。

2.4.2 元数据擦除码相关研究

复制和校验（如 RAID）是提供数据高可用性的两种常用的方法。复制技术要求高传输带宽、高存储开销；校验技术不能提供高失效率下的容错保证。擦除码（Erasure Code）技术能够在低存储开销的情况下提供高可用性。Erasure Code 将数据对象（如文件）划分为 m 个对象，并且再将它们编码为 n 个对象，其中 n>m，称 r=m/n < 1 为编码率。Erasure Code 技术增加的存储开销为 1/r 倍，但是具有一个重要的特点，即原数据对象能够通过其中的任意 m 个数据对象进行重构来得到。事实上，Erasure Code 技术是复制和 RAID

的超集。例如：四个副本的系统可以描述为 Erasure Code（m=1，n=4），RAID 级 1 和 5 可以分别描述为 Erasure Code （m=1，n=2）和 Erasure Code （m=4，n=5）。

比较典型的采用 Erasure Code 方法的是采用基于 Reed-Solomon 编码的数据容错算法。它的主要思想是：第一步，运用范德蒙德（Vandermonde）矩阵计算和维护校验字符；第二步，运用高斯消元法（Gaussian Elimination）对系统进行恢复；第三步，运用伽罗瓦域（Galois Field）实现运算。

相关的研究也表明，在 PB（10^{15} 字节）级规模的存储系统中，提供两个数据对象副本或 Erasure Code 技术对保证存储系统的数据可靠性和可用性是必要的，三个或更多的数据对象副本使得系统的存储空间开销太大。在要求更高的可用性存储系统中，基于 Erasure Code 的容错机制是有必要的。它能够在提供更高可靠性和可用性的前提下，减少对存储空间的需求。

实际的研究测试结果表明基于 Erasure Code 编码的数据容错算法能够在增加 m/（m+n）的存储开销下，能够在任意不超过 m 个对象失效的情况下，仍然保证数据的高可用性，提高了存储系统的高可靠、高可用性。该方法尽管增加了计算开销，但这种能够容忍多个对象同时失效的方法为大规模的存储系统的数据恢复提供了一个很好的解决方法。

2.4.3 元数据检查点相关研究

检查点技术方法现有的主要应用是用于保存运行中的程序的状态，如进程数据段，用户栈内容，处理机状态字，当前活动的用户文件信息等。比较典型的系统有由美国威斯康星大学开发的 Condor 系统[16][17]，Rio de Janeiro 联邦大学（UFRJ）开发的可扩展操作系统 Nomad 中的一个基于检查点回滚恢复的进程迁移子系统 Epckpt[18]，AT&T 实验室的 libckp 库[19]，美国田纳西大学 Plank 等人编写的 libckpt[20][21][22]以及国内清华大学开发的 ChaRM 系统[23][24][25]。

然而这些检查点技术的应用均是在进程一级，对元数据进行检查点操作尚属新兴的起步阶段。目前国外以卡耐基-梅隆大学（CMU，Carnegie Mellon University）的 CVFS（Comprehensive Versioning File System）[26][27]和 S4（Self-Securing Storage Server）系统[28]以及加州大学圣-克鲁兹分校（University of California，Santa Cruz）的基于对象的文件系统（Object-based File System）[29][30][31][32]以及 Ceph[15]为代表。国内从已发表的论文来看，尚没有该方面的相关研究。

Ceph 这种可扩展的高性能分布式文件系统目前还没有实现 MDS 的恢复，它拟采用 MDS 日志记录的方法来实现 MDS 的失效恢复，其方法是在一台 MDS 失效时另一节点

快速地扫描日志来恢复失效节点内存缓存中的关键内容。

本文即拟采用这种在基于日志的元数据的思想基础上实施检查点操作来研究元数据的容错问题。

§2.5 小结

在本章中，介绍了文件系统、索引节点以及对象存储、当前元数据容错方法的相关研究技术。由于对象存储体系结构的诸多优越特性，使得它能够成为开发满足高性能及高可用性集群计算的新一代存储系统的基础。通过对目前在对象存储系统中元数据容错所采用的方法，以及目前主流的已经在现实实际中运行的系统进行的分析，以及通过对现有运行系统中元数据容错方法的现状及思想方法的分析，提出了高可用性的需求、一定的不足以及待改进的方面，为本文的第三章、第四章及第五章所进行的改进工作的引入做了铺垫。

第三章 一种容错的元数据集群管理系统结构

§3.1 引言

高性能计算在科学研究领域和关键业务处理方面占有举足轻重的地位，它在科学研究和工程计算等领域应用广泛。通常支持这类应用的解决方案多倾向于采用专用的昂贵的系统，但是随着用户对诸如 7x24 小时的高可用性以及复杂网络管理的需求日益提高，集群系统由于其高灵活性，高性能，高可用性，低成本等特点也日趋成熟，使得其逐渐得到广泛运用并逐渐在这些高端应用得到发展。

目前存在两种类型的网络存储系统，这两种系统的区别在于它们各自的命令集。第一种系统是 SCSI 块 I/O 命令集，用于存储区域网（SAN）。它通过在磁盘驱动层或光纤通道层直接访问存储的数据来提供很高的随机 I/O 和数据吞吐性能。第二种系统是网络附加存储系统（NAS），它使用 NFS 或 CIFS 命令集来访问数据。它的优点是当存储在存储介质上的元数据处于共享状态时，多个客户端节点可以同时访问数据。而新兴的对象存储体系结构则拥有二者的优点，既可以保证拥有直接访问磁盘而带来的很高的系统性能，又能通过简单的管理实现文件数据以及文件元数据的共享。

对象存储体系结构结合了当今这两种存储系统的优点：性能和文件共享，而消除了这两种存储系统的主要缺陷。第一，对象存储体系结构提供了计算节点可以直接和并行的方式来访问存储设备的方法，这样就为系统提供了很高的性能，第二，对象存储体系结构将系统的元数据进行了分布，这样就使得共享文件的访问不会成为系统的中心瓶颈。

这种基于对象存储的大规模存储系统最重要的一个特点是将文件的元数据与数据访问分离。而 PB 级规模的存储系统更是需要独立的元数据服务器（MDS，Metadata Server）集群承担元数据访问服务。

这是因为对象存储文件系统中一个潜在的系统性能瓶颈是对元数据的访问，所以在大规模的存储系统中，尽管元数据的数据量大小同整个存储系统的整体数据容量大小相比起来相对较小，但是统计数据表明，在所有的文件系统操作访问中，对元数据的访问操作约占所有文件访问操作的 50%－80%[10]。所以，一个高效的元数据访问对整个存储系统在提高性能和保证有一个良好的可扩展性，可伸缩性是至关重要的。而元数据服务器集群则可以做到的这一点。

以下以数据量化说明采用集群的处理方式的需求[33]：

假设每个元数据记录有 128Byte 的大小，每台元数据服务器（MDS）的内存大小为 4GB，同时假设内存 Cache 之间的重叠已经达到最小化，并且内存得到有效充分的利用，

则一台 MDS 服务器可以处理 4GB/128Byte ≈ $32*10^6$ ≈ $0.3*10^8$，即每台 MDS 服务器可以处理大约 0.3 亿（3 千万）个元数据记录项或目录项。而一个多 PB（Multi-PB）级的文件系统可能包含 10 亿多个文件，甚至上 100 亿多个文件。那么，为这样一个规模的存储系统应用则至少需要大约 30～330 多台（10 亿/0.3 亿 ≈ 33，100 亿/0.3 亿 ≈ 330）元数据服务器。所以，有对 MDS 采用集群的处理方式的需求。

因而，研究高效、可靠的元数据集群管理方法对实现 PB 级存储系统的高性能和高可用性至关重要[34]。

§3.2 元数据集群系统设计的基本思想与体系结构

当前，学术界和产业界都在致力于 PB 级规模的存储系统的研究与开发，其中，对象存储文件系统将成为重要的发展方向。根据当前加州大学（University of California），Lustre 等的研究的情况，对于一个 PB 级规模的存储系统而言：

（1） 如上所述，通常包含 100 台（10^2）规模左右的元数据服务器；

（2） 假设每个存储节点的存储容量为 TB 级（10^{12} 字节），那么一个 PB 级（10^{15} 字节）规模的存储系统则包含上千台（10^{15} 字节 / 10^{12} 字节 = 1,000）（10^3）规模的对象存储设备（OSD），对一个多 PB 级的存储系统而言，则包含上万台（10^4）规模的对象存储设备（OSD）；

（3） 系统应该能够支持上万个甚至是上百万个（10^6）规模的用户进程的并发访问。

以上元数据集群，对象存储设备（OSD），客户端的规模依次相差两个数量级。

因而对象存储系统的体系结构图如前图 2.2。

3.2.1 元数据集群系统设计的基本思想

在对象存储结构中，通过将文件的访问属性和数据对象的分开管理，将文件系统的元数据与实际数据读写分离，将元数据中的 inode 部分的工作负载分布到各个智能化的对象存储设备（OSD）上，从而大大减轻了系统的元数据工作负载，提高了系统的整体性能。

传统文件系统中，目录文件的内容以及文件的元数据(inode 层)是分开的，为了获得文件的元数据，需要多次磁盘 I/O，而将文件的元数据嵌入到目录文件中能够显著提高元数据的访问效率[35]。

在此基础上，本文对文件系统的元数据进行进一步的抽象，将目录路径属性与文件的元数据分开管理，目录路径属性包括文件所在的目录路径名及相应的路径访问控制属性，文件的元数据包括文件名、文件长度、创建时间、修改时间等访问属性。这种分割符合面向对象的管理方法，是对对象存储结构的进一步延伸[9]。

目录路径对象[36]类似于传统文件系统中的目录文件，包含该目录路径下的文件元数据列表。不同的是，传统目录文件中的登记项只包含文件名及其索引节点号，读取文件元数据需要先访问目录文件获得文件索引节点号，再根据索引节点号读取该文件的元数据；而目录路径对象的登记项包含文件的全部元数据，因此读取目录路径对象时直接获得文件的元数据。

目录路径对象的内部结构如下图 3.1：

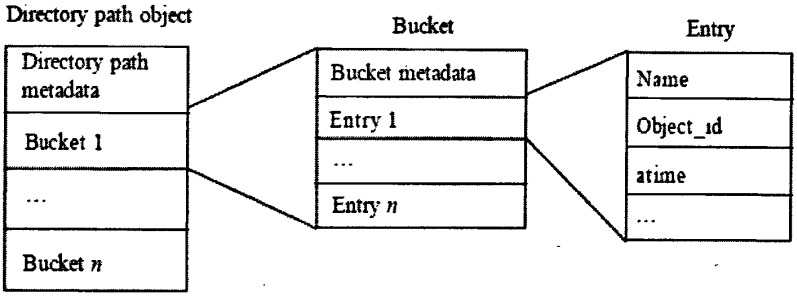

图 3.1　目录路径对象的组成结构

目录路径对象作为目录路径的数据对象由 MDS 管理，目录路径对象包含一个或多个固定大小的目录对象(Bucket)，每一个目录对象包含固定数量（可根据具体应用设定）的文件或目录文件元数据的登记项入口(Entry)，在目录路径对象的起始位置包含描述该目录路径对象属性的元数据。

3.2.2 元数据集群系统的体系结构

因而，依据上述的设计思想，其元数据集群系统的体系结构如下图 3.2 所示：

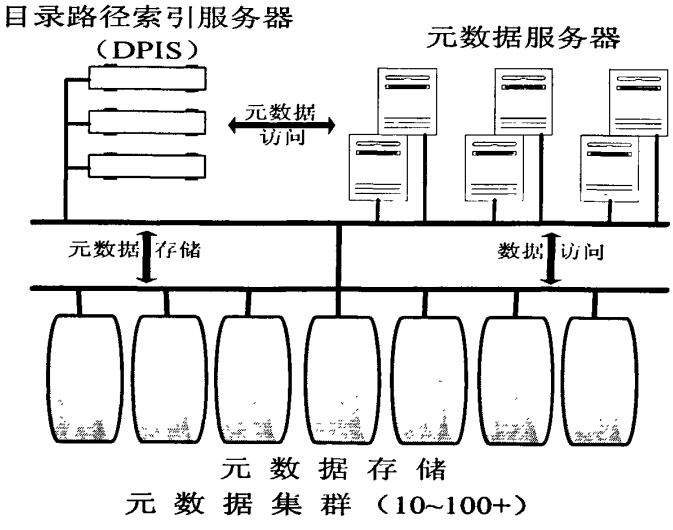

图 3.2　元数据集群系统的体系结构

据此，可以得到经采用本文提出的 OCFS 系统的体系框架后整个对象存储体系结构

图如下图 3.3 所示：

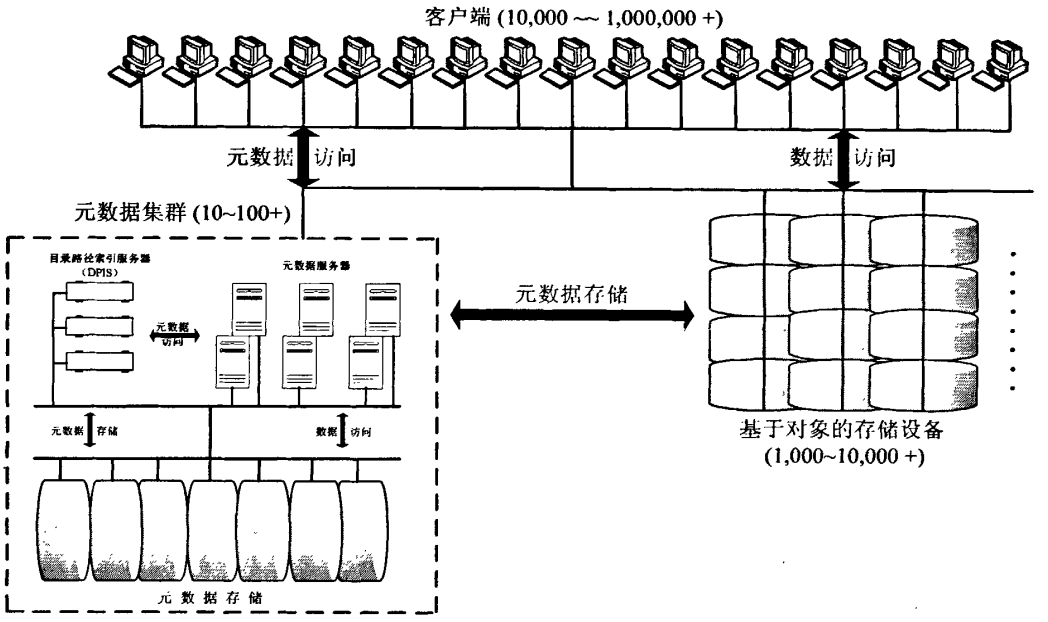

图 3.3 元数据集群系统体系结构扩展后的对象存储体系结构图

3.2.3 元数据集群系统小结

本文的基于对象存储的集群文件系统 OCFS（Object-based Cluster File System）[9]通过采用对象存储体系结构的方式将文件数据访问控制流和数据流有效地分离，为系统客户提供高吞吐率，可扩展的数据服务。同时，在此基础上，对对象存储体系结构进行了进一步的扩展和延伸，将 MDS 集群按照对象存储体系结构的方式予以扩展，将文件系统的元数据进行了进一步的抽象，将目录路径属性与文件的元数据进行进一步的分开管理。这种分割既符合面向对象的管理方法，又可以根据用户设定的要求，依据元数据间的关联关系和元数据服务器的负载情况动态地进行元数据的分布决策，保证了元数据服务器能够动态地加入到服务器集群中，使元数据以非常平稳的方式分布到新的元数据服务器上。

与此同时，还可以根据应用的需求情况，系统能够动态地决定参与应用请求处理的元数据服务器。系统的元数据服务具有很高的可扩展能力和可用性，并表现出能够根据用户需求动态分配系统资源的动态处理能力。

§3.3 元数据集群的系统组成部分

本文的 OCFS 集群文件系统主要由客户端文件系统（Client File System，CFS），目

录路径索引服务器（Directory Path Index Server，DPIS），元数据服务器（Metadata Server，MDS）和对象存储服务器（Object Storage Target，OST）四类子系统组成[9]。

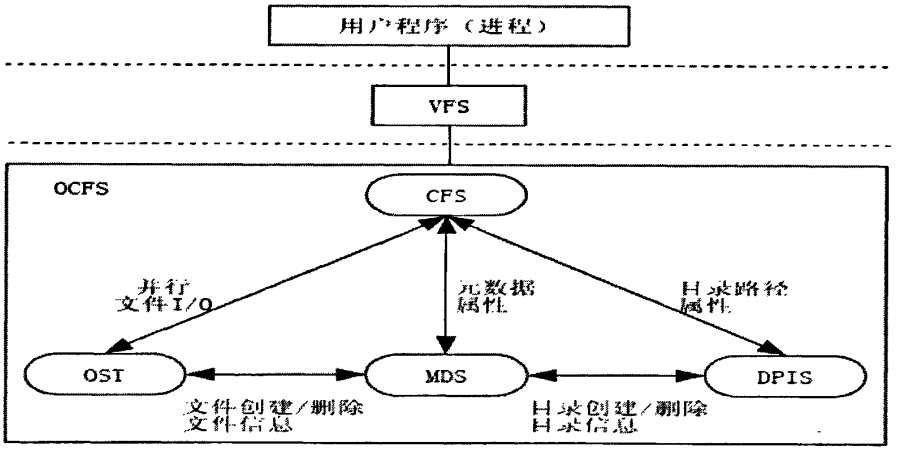

图 3.4　OCFS 的系统组成部分

- CFS　给上层应用提供树形目录文件结构和 POSIX 兼容的文件系统接口，支持上层应用执行标准的文件操作，如 Open（），Read（），Write（），Close（）等操作。
- DPIS　负责目录路径属性的存储管理，包括查询，创建，更新和删除。
- MDS　负责目录对象的存储管理，包括文件元数据的查询，创建，更新和删除；MDS 和 DPIS 协作为整个存储系统提供统一、一致的命名空间，负责存储系统的目录层次，权限管理，控制客户端对存储服务器的授权访问，协调不同客户对元数据 cache 的一致性。
- OST　是负责数据存储的目标服务器，负责底层的数据对象分配，布局，对客户的数据请求进行认证，响应，并提供基于对象接口的数据存储服务。

在 OCFS（基于对象存储的集群文件系统）中，任何文件的数据分为三部分：目录路径属性、文件元数据与数据对象。目录路径属性包括文件所在的目录路径名和路径访问控制属性，保存在目录路径索引服务器（DPIS）中；文件元数据包括文件的属性，如所有者、访问权限、文件长度、最近访问时间等，保存在 MDS 中；数据对象包含文件的实际数据，保存在 OSD 中。

于是， OCFS 中的数据管理分为两层，第一层是目录数据（高层元数据）管理，目录数据分割为目录路径属性和目录路径对象，目录路径对象包含多个桶（bucket）对象，每个对象具有唯一的 ID，文件元数据项包含在桶对象中；第二层是文件数据管理，文件数据分割为文件元数据和数据对象。文件的实际数据由多个数据对象构成，每个数据对

象具有唯一的 ID。桶对象和数据对象分别存储在 OSD 集群中，它们所分布的 OSD 是根据其对象 ID 计算确定的。这种根据对象 ID 自主计算分布位置的方法能够简化系统的存储管理，支持 MDS 和 OSD 的动态均衡扩展。

§3.4 元数据集群的容错模块组成部分

在本文中，元数据集群容错模块主要是由三部分组成，分别是：

- 元数据对象副本模块
- 基于日志的元数据对象模块
- 元数据对象擦除码模块

其结构类图如下图 3.5：

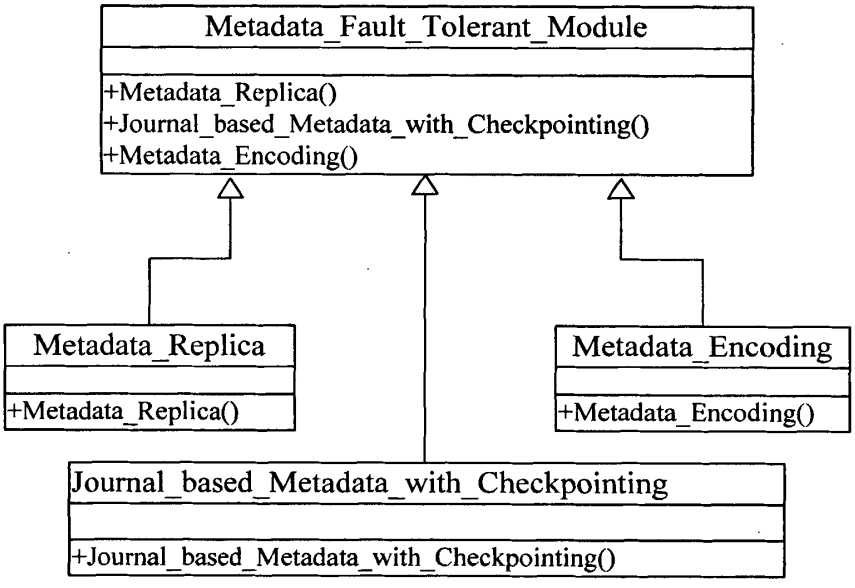

图 3.5　元数据容错模块结构类图

3.4.1 元数据对象副本模块

主要完成通过元数据对象副本的方法来实现元数据集群系统的容错。具体详细内容见本文第四章。

3.4.2 基于日志的元数据对象模块

主要完成通过对基于日志的元数据对象实施检查点操作的方法来实现系统元数据集群系统的容错。具体详细内容见本文第五章。

3.4.3 元数据对象擦除码模块

拟主要完成通过基于 Reed-Solomon 编码的方法来实现系统元数据集群系统的容错。

§3.5 系统设计与实现

3.5.1 元数据集群系统原型的设计

在本节中，对元数据集群系统原型做了总体设计。为了便于实现系统的可靠性以及遵循“分而治之（Divide and Conquer）”的思想，对系统整体框架进行了模块化的设计。系统模块及模块层次如下图 3.6：

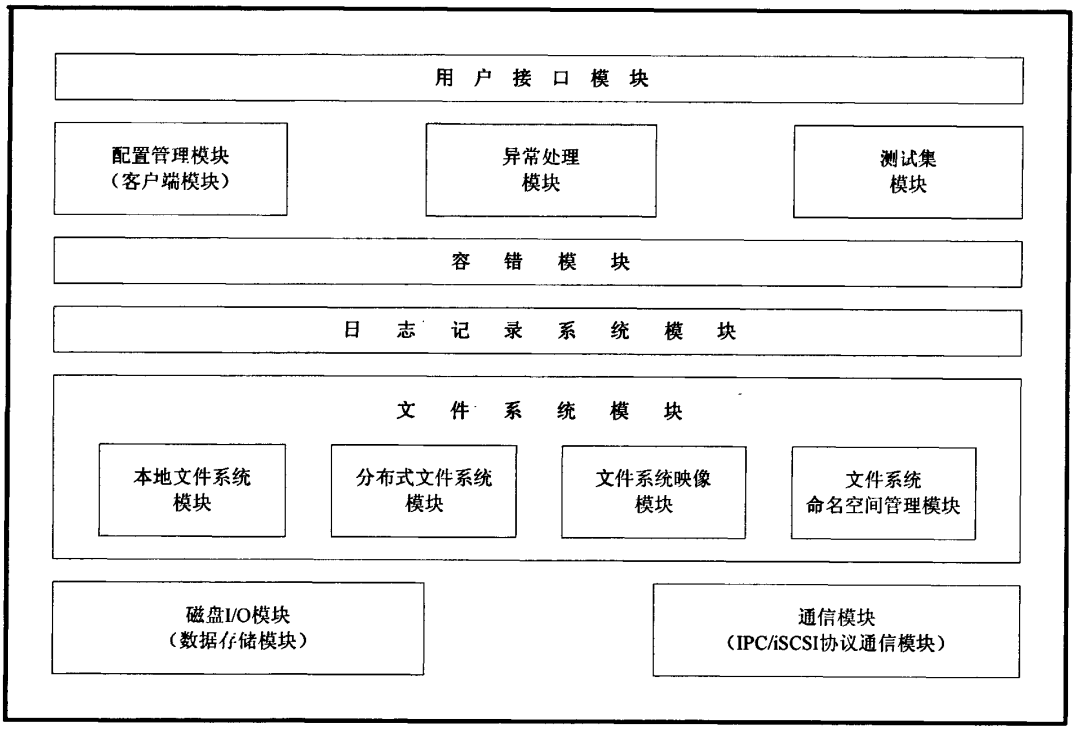

图 3.6 元数据集群系统原型结构框架图

下面分别对通信模块，文件系统模块及容错模块予以介绍：

3.5.1.1 通信模块

所有的通信协议都是基于 TCP/IP 协议之上的。

系统主要使用进程间通信（IPC，InterProcess Communication）协议，其原因是：

（1）IPC 是快速接口反射式 RPC（远程过程调用，Remote Procedure Call）系统，是实现 RPC 的一种方法，具有快捷、简单的特点。它不像 Sun 公司提供的标准 RPC 包，基于 Java 序列化。

（2）IPC 无需创建网络 skeletons 和 stubs。

（3）IPC 中的方法调用要求参数和返回值的数据类型必须是 Java 的基本类型，

String 和 Writable 接口的实现类，以及元素为以上类型的数组。接口方法应该只抛出 IOException 异常。

本文将目录路径索引服务器（DPIS）与其它目录路径索引服务器（DPIS）、元数据对象存储设备（OSD）以及客户端的通信分别采用经 IPC 抽象封装过的不同的协议，分别为 IPC_DPIS_DPIS_Protocol，IPC_DPIS_OSD_Protocol，IPC_DPIS_Client_Protocol 它们统一继承 IPC_DPIS_Protocol 并完成各自的功能。其关系如下图 3.7：

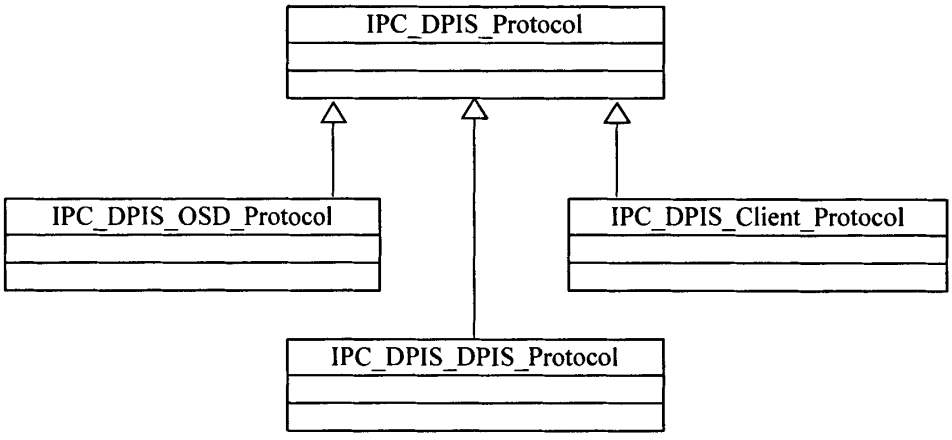

图 3.7　通信模块结构图

结合元数据集群系统的体系结构图 3.2，得到元数据集群系统的整体通信框图如图 3.8：

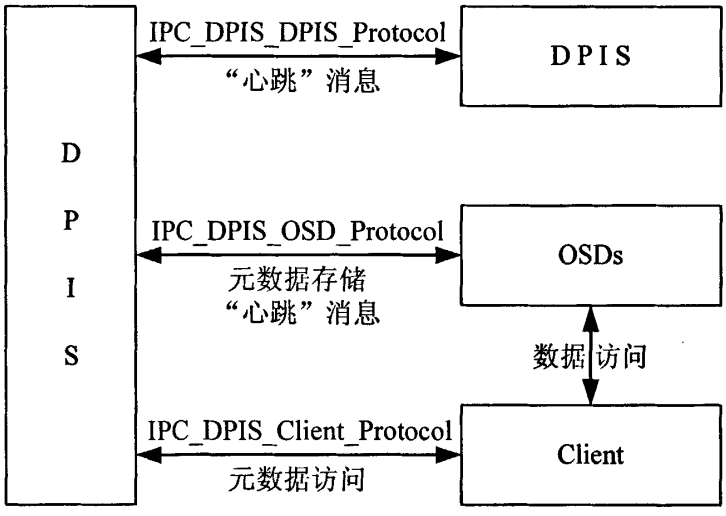

图 3.8　通信模块结构整体框图

3.5.1.2 文件系统模块

文件系统模块是整个元数据集群系统的核心。它由本地文件系统接口模块，分布式

文件系统模块，文件系统映像模块以及文件系统命名空间管理模块四部分组成。主要对整个集群系统负责维护文件系统所有的元数据，包括命名空间、访问控制信息、从文件到数据对象和从数据对象到块的映射以及块的当前位置。它也控制系统范围的活动，如数据对象的租约管理、孤儿数据对象的垃圾收集、对象存储设备（OSD）之间数据对象的迁移。DPIS 定期通过“心跳（HeartBeat）”消息来与每一个对象存储设备（OSD）通信，给对象存储设备（OSD）传递指令并收集它的状态。

3.5.1.3 容错模块

容错模块的设计见前述 3.4 节。

其组成如下图 3.9：

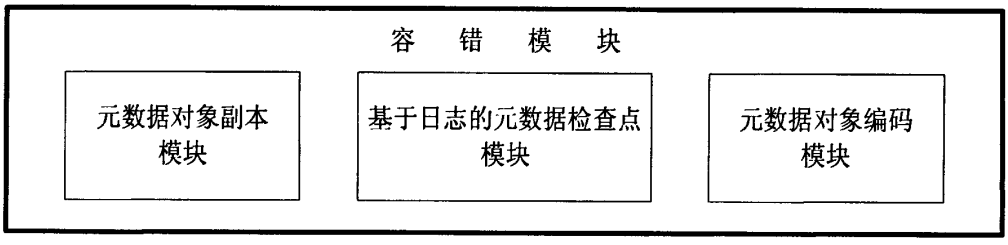

图 3.9 容错模块结构框架图

3.5.2 元数据集群系统原型的实现

3.5.2.1 通信模块的实现

```
ArrayList<OSD_Descriptor> heartbeats = new ArrayList<OSD_Descriptor> ();

public synchronized void register_OSD_node ( OSD_Registration OSD_nodeReg,
                                    String networkLocation)   throws IOException

public boolean gotHeartbeat( OSD_nodeID nodeID, long capacity, long remaining,
        xceiverCount, int xmitsInProgress, Object[] xferResults, Object deleteList[] )
        throws IOException

while (true) {
      try {
         this.taskDPIS_ReportServer = RPC.getServer(this, this.taskReportPort,
                                          maxCurrentTasks, false, this.Conf);
```

```
            this.taskReportServer.start();
            break;
          } catch (BindException e) {
                LOG.info("Could not open report server at" + this.taskReport + " ,
                        trying new port");
                this.taskReportPort + + ;
                }
              }
```

3.5.2.2 文件系统模块的实现

```
public DistributedFileSystem (InetSocketAddress DPIS , Configuration conf)
                                                        throws IOException
    {
      super(conf);
      this.dfs = new DFS_Client(DPIS, conf);
      this.name = DPIS.getHostName() + ":" + DPIS.getPort() ;
    }

public static MDS_Namesystem MDS_NamesystemObject ;
public static MDS_Namesystem getFSNamesystem() {
        return fsNamesystemObject;
    }

// 表示映像文件
MDS_FS_Image (File imageDir, String edits) throws IOException {
    this.imageDirs = new File[1];
    imageDirs[0] = imageDir;
    if (!imageDirs[0].exists()) {
      throw new IOException("File " + imageDirs[0] + " not found.");
    }
```

```
    this.editLog = new FSEditLog(imageDir, this, edits);
  }
```

3.5.3 元数据集群系统容错模块的实现

分别在第四章，第五章对相应的实现进行了描述和介绍。

§3.6 小结

本章首先通过对实际需求进行的分析，得出有元数据集群的需求，提出了本文的元数据集群系统设计的基本思想以及体系结构，对系统的各个组成部分以及本文的元数据集群系统的容错模块进行了说明，最后对原型系统的设计与实现进行了描述。

第四章 基于目录对象副本的高可用元数据管理模型

§4.1 引言

在 PB 级系统中，由于组件数目庞大，系统组件（磁盘，主存，连接器，网络，电源等）等硬件失效通常被看成是正常情况，而不再视为异常情况。整个分布式系统可能是由成千上万台的服务器机器组成，它们存放着文件系统数据片。事实是分布式系统拥有众多大量的组件，每个组件都有很大的失效概率。与此同时，人为的失误，操作系统，应用程序的 Bug 缺陷等软件因素而引起的失效也同样非常普遍。这就意味着分布式系统中的某些软硬件功能组件总是无法正常工作的。因而，失效的检测以及快速自动地从这些失效中恢复过来是分布式系统的体系核心目标。

作为分布式系统中关键的组成部分----分布式文件系统，高可用性要求分布式文件系统中的部分节点失效时，其它节点能够尽可能最大限度地提供服务。数据高可用性是各种高可用性的基础。实现数据高可用性的基本原理是在系统中引入数据冗余副本，通过数据的冗余备份或校验达到发现错误并进行数据恢复的目的。它能在一个或几个节点不能提供服务时，这些节点的数据能够在其它节点上得到，从而使整个系统能够提供一种持续的数据服务。

目前，高可用的元数据服务器集群的系统结构从是否共享存储的角度分为两类。一类是以 GFS，Lustre 为代表的非共享存储，元数据的可靠性是通过增加备份服务器实现的，系统的成本较高。另一类是通过分布式文件系统共享磁盘阵列或 SAN 来实现的共享存储，这种方法的不足是多台元数据服务器对共享元数据的并发冲突访问，容易形成 I/O 瓶颈，降低系统的整体性能。

本章的方法是在基于目录路径的元数据管理方法[36]的基础上，进一步考虑其高可用性，提出基于目录对象副本的高可用元数据管理模型，通过对元数据采用多个副本的方式来实现元数据的高可用性。通过高效的目录对象副本定位、放置、迁移和更新策略实现高性能、高可用的元数据集群管理。这样既避免了通过增加备份服务器来实现元数据的可靠性而带来的较高的成本，同时又避免了多台元数据服务器对共享元数据的并发冲突访问而造成的系统整体性能的降低。

§4.2 目录对象副本管理策略的关键技术

4.2.1 目录对象副本的放置策略

副本的放置策略对于分布式文件系统的可靠性和性能表现是非常关键的。

本章的目标是开发一个“机架感知（rack-aware）”的副本放置策略来提高改善数据

的可靠性、可用性以及网络带宽的利用率。假定元数据服务器集群是由分布在多个机架上的计算机所构成的集群。不同机架上的两台计算机之间的通信必须通过交换机。在大多数情况下，位于相同机架上的两台计算机之间的网络带宽要高于位于不同机架上的两台计算机之间的网络带宽。

在文件系统启动时，OCFS 中的每台 MDS 确定它所属的那个机架，并在向目录路径索引服务器(DPIS)注册时告知 DPIS 它的机架 ID 号（rack_id)。系统提供了用于确定每台机器的所属机架 id（rack_id）的接口，这些接口为用于确定每台机器的所属机架 id（rack_id）的可装配模块提供了便利。

一种最简单的策略就是在机架间随机放置对象副本。这种策略避免了由于整个机架失效而导致的数据丢失，而且实现了当进行数据读取时，可以充分利用多个机架的网络带宽。这种策略将对象副本在整个集群中进行均衡的分布。这样就使得在组件失效时很容易地实现负载均衡。但是，这种策略增加了写操作的开销，因为一次写操作需要向多个机架上的机器传输对象副本。

在大多数情况下，元数据对象副本的个数是两个。OCFS 的对象副本放置策略是：目录对象副本的个数是两个，首先在本地机架的一个节点上放置一个对象副本，而同时在另外一个不同的机架上的节点上放置另外一个对象副本。

这种策略增加了读取数据时的聚合网络带宽，提高改善了读操作的性能。因为整个机架失效的概率远小于节点失效的几率，提高了目录对象数据的可靠性以及可用性的保证。所以这种策略在保证目录路径对象数据可靠性或高可用性的条件下，提高改善了读操作的性能。但是这种策略增加了机架间的写操作的工作负载。当写入数据时，这种策略能够减少聚合的网络带宽。因为目录对象副本是放置在两个独立的机架上，在进行写操作时，需要向两个机架上的存储节点进行写操作。

目录路径对象副本放置算法：Dir_Path_Obj_Repli_Placmenting

输入：rack_id

输出：DPIS_xx

算法描述：

```
for rack_index=1 to 2
    selectRandomHash(rack_id)
    DPID_0[j]=selectRandomHash(BucketNode_id)
    selectNextRandomHash(rack_id)   //或 rack_id + 1
```

end for

4.2.2 目录对象副本的迁移策略

元数据服务器集群的高可用性要求在组件失效发生时仍能可靠地访问元数据。OCFS系统的首要目标是即使在失效发生时仍能可靠地对数据进行存储。考虑最常见的失效是MDS失效和网络连接失效。

OCFS系统中元数据服务器集群中的MDS周期性地向DPIS发送"心跳"消息。网络连接失效可能会导致一部分MDS与DPIS失去连接。DPIS检测到缺少这些心跳消息的情况后将这些MDS标记为"死亡"状态，并且不再向这些MDS转发任何新的I/O请求，位于这些MDS上的元数据对用户也不再有效，这时就会导致目录路径对象的副本系数低于指定的值。DPIS确定了所有需要重新复制的对象副本后就开始将它们重新迁移至其它的MDS节点上。对象副本迁移的原因可能是由于以下这些原因引起的：一个MDS失效了，一个冲突的对象副本，MDS上坏了硬盘或者是提高增加了部分对象的副本系数。

本文的OCFS原型系统能够支持目录路径对象的负载均衡。如果一个MDS上的剩余空间低于一个确定的阀值时，对象副本可由一台MDS自动地迁移至另外一台MDS上。与此同时，如果突然对某一个特定的文件有很大的访问需求时，OCFS系统能够动态地创建额外的对象副本并将这些对象在整个MDS集群中进行负载均衡。

4.2.3 目录对象副本的更新策略

为了充分有效地利用网络带宽，充分地使用每台机器的网络带宽，避免网络瓶颈以及高延时的链路，系统对进行数据访问的控制流和数据流进行解耦分离。客户端对数据进行访问时的控制流和数据流如图4.1。

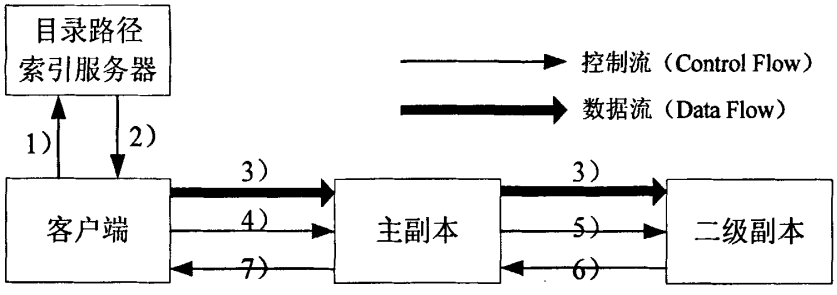

图4.1 写操作的控制流图和数据流图

控制流：

(1) 客户端向目录路径索引服务器（DPIS）询问哪个MDS节点持有当前请求目录对象的令牌以及该目录对象其它副本的位置。如果没有一个MDS持有令牌，则

DPIS 向其所选择的一个对象副本授权一个令牌；

(2) DPIS 用 primary 的标识符和其它副本的位置进行应答，客户端将这些目录对象进行缓存，为以后的变更进行准备。只有当主副本（primary 副本）不能访问或者 primary 副本返回它不再持有令牌的时候，客户端才有必要同 DPIS 再次取得联系；

(3) 转数据流；

(4) 一旦所有的副本都对收到的数据进行了应答，那么客户端就向 primary 副本发送一个写请求。该请求标识了早先发给所有副本的数据。primary 副本对它所接收到的可能是来自多个客户端的所有变更分配一系列连续的序列号，这些连续的序列号提供了必要的序列化的依据（将变更依次写至硬盘的依据）。primary 副本依照序列号的顺序对其本地状态实施变更；

(5) primary 副本将写请求转发至所有的二级副本（secondary replica）。每个二级副本依照由 primary 副本所分配的相同的序列号顺序对自身来实施变更；

(6) 所有的二级副本向 primary 副本进行应答，表示它们已完成了该项操作；

(7) primary 副本向客户端进行应答。

数据流：

(3) 客户端将数据发布给所有的副本。客户端可以以任意顺序来发布这些数据。每台 MDS 将会将数据保存在内部的 LRU 缓存中，直到这些数据被提交或者过期后才会从缓存中删除掉。

为了充分地利用每台机器的网络带宽，尽可能地避免网络瓶颈及高延时链路，数据是在 TCP 连接上以流水线的方式沿着一个 MDS 链线性地来进行传输的，而不是以其它形式的网络拓扑结构（比如树）将数据同时分发给其余的 MDS 来进行传输发布的。一旦 MDS 接收到某些数据，它就立即开始进行转发，将数据只传输转发给离自己最近的 MDS（距离的远近是通过 IP 地址来进行判断的），立即发送数据的同时不会减少接收数据的速度。这样，充分利用了全双工网络的带宽，每台机器的全部出口带宽都用于尽可能地传输数据，而不是在多个接收者之间进行分配。

通过将控制流和数据流进行解耦分离，系统可以通过基于网络拓扑结构来调度代价昂贵的数据流来提高改善性能，而无需考虑是哪个 MDS 拥有 primary 副本。

4.2.4 目录对象副本的一致性保证策略

目录对象的一致性包括目录对象自身数据的完整性、正确性以及多个对象副本之间的一致性。

4.2.4.1 目录对象自身正确性的检查

由于存储设备的失效，网络失效或软件本身的缺陷，MDS 中的目录对象副本可能失效，或者很有可能从一台 MDS 中获得的目录对象副本是损坏的。使用文件的其它副本可以使数据从损坏的文件中恢复过来。但是，通过频繁不断地比较位于不同服务器上的文件副本的方式来检测副本之间的冲突是不现实的。

一种简单的方法是通过比较位于不同服务器上的对象副本来检测副本之间的一致性，但是这种方法增加了网络带宽开销。因而本文采取的方法是，由每台服务器独立地通过维持一个特定的校验和（checksum）信息来验证它所拥有保存的对象副本拷贝的完整性和正确性。OCFS 系统每当创建一个目录对象副本时，对目录对象的内容实施校验和检查，计算每个对象副本的校验和并将这些校验和保存在一个单独的隐藏文件中。在对象副本之间的一致性比较之前，每个对象副本自身首先会通过校验和校验对其副本数据内容的完整性、正确性进行检查。

每个目录对象副本都有一个对应的 32 位的校验和。这些校验和信息是位于内存中的，并且它们随着日志系统而持久化地保存在磁盘系统上。这些信息与目录对象的实际数据是相分离的。在进行读操作时，对用户的请求返回任何数据之前，服务器将会检验这些目录对象的校验和。如果一个目录对象与其记录的校验和不相匹配，那么该服务器将向请求者返回一个错误信息，而不是返回一个不正确的数据，同时并将这种不匹配通知 DPIS，于是 DPIS 会从另外一个副本读取数据。与此同时，DPIS 会从该副本进行复制，当复制完成形成新的对象副本拷贝后，DPIS 将会通知 MDS 服务器对校验和不匹配的对象副本拷贝予以删除。

图 4.2 展示了该过程：

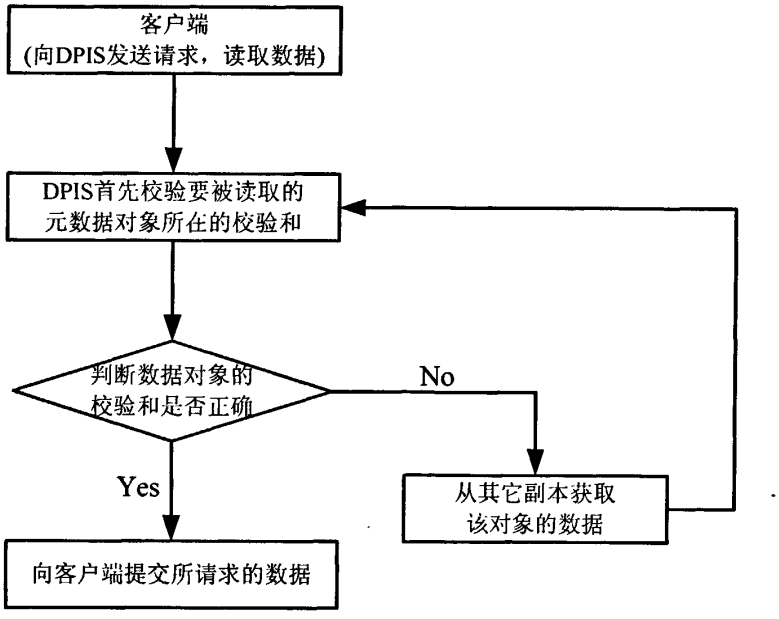

图 4.2　目录对象访问流程图

通过对校验和进行的校验，保证了用户请求者读到的是正确的目录对象副本。即使从服务器读取到的是损坏的对象副本，由于采用了这种机制，使得 MDS 服务器不会将损坏的对象副本传播给其它的存在相同对象副本的机器。

在每台 MDS 对其上的对象副本自身检查正确无误后，还需要对多个对象副本之间的一致性进行相应的检查。

4.2.4.2 目录对象副本之间一致性的检查

对于对象副本之间的一致性检查，通常有悲观复制策略和乐观复制策略。

悲观复制策略是阻止那些可能引起不一致的拷贝，然后将更新向其它服务器传播。它实现了非常严格的一致性语义。通过限制了可用性避免了不一致情形的发生。每台服务器对其它服务器的工作做出了最坏情况的假设，而且认为如果一个操作可能造成不一致，那么不一致必将发生。悲观策略为小概率事件付出了过分的代价。

事实表明，复制需要一个乐观的管理策略，所以系统采用乐观复制策略。乐观策略允许在任何一个拷贝中实施更新操作，当多个拷贝重新连接时，最新版本的拷贝自动更新其它的拷贝。但是，乐观策略必须解决更新冲突的检测问题。

乐观复制策略并不限制冲突的发生，但是操作冲突造成了文件的不一致。因此，采用乐观复制策略的系统必须具备冲突处理的能力。在冲突发生的时候，应该具备尽快检测到冲突的能力，系统必须容忍因为冲突而造成的损害，并且保存一定的修复信息，并利用这些修复信息提供冲突恢复的途径。

乐观复制策略为解决冲突提供了许多方法。目前的系统多采用版本向量方法（Version Vector）。一个文件 f 的每个拷贝都跟随一个版本向量，版本向量记录 f 的每个副本发出的更新的次数，它由 n 个二值对的序列组成（n 为 f 的拷贝的份数），其中第 i 个向量值（Si，Vi）表示由备份 Si 发出的更新的次数 Vi。可以通过比较版本向量检查是否存在多个拷贝发出的冲突更新。对于两个向量 V 和 V'的每个元素存在 Vi>=Vi'（i=1，2，..... n），则称 V 支配 V'。显然，如果 V 支配 V'，则具有向量 V 的拷贝是具有向量 V'的拷贝的数据超集。而且，如果 V 和 V'相互不支配，则存在更新冲突。

如上所述，系统对所有副本的每次变更都赋予了一个序列号，每次变更操作成功后，并都对数据对象赋予一个对象版本向量号。通过对副本之间对象版本向量号的比较来判断、维持对象副本之间的一致性。

副本的状态有下列四种状态：

- 一致的（consistent）：不论客户端是从哪个副本上读取数据，所有客户端看到的都是相同的数据。
- 确定的（defined）：不仅文件数据内容更改后是一致的，而且客户端可以看到每个副本都进行了哪些变更。
- 不确定的（undefined）：客户端无法看到每个副本都进行了哪些变更。
- 不一致的（inconsistent）：不同的客户端在不同的时间里看到的是不相同的数据。这种情况通常发生在变更操作失败时。

副本状态关系图及变更操作后对象副本的状态图如图 4.3：

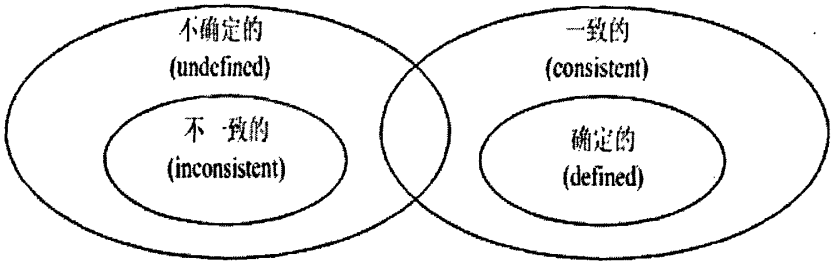

图 4.3　副本状态关系图

变更操作后对象副本的状态表如表 4.1：

表 4.1 变更操作后对象副本的状态

	写操作 （write）	记录追加 （record append）
单一操作成功 （single success）	确定的 （defined）	确定的 （defined）
并发操作成功 （concurrent success）	一致但非确定的 （consistent but undefined）	不一致的 （inconsistent）
操作失败 （failure）	不一致的 （inconsistent）	

对客户端来说，副本的各种状态对客户形成的视图仅分为最新的副本和过时的副本。从严格的语义上来讲，当且仅当副本的状态处于确定（defined）时，客户看到的该副本为最新的副本，而当副本处于其它三种状态时，客户看到的是过时的副本。

而当一台 MDS 失效时，其上的对象副本可能会成为过时的对象副本，而且当一台 MDS 宕机时，对象副本可能会错过对对象的变更操作。对每个对象来说，DPIS 都维持着一个对象版本向量号来区分最新的副本和过时的副本。

无论何时 DPIS 向一个对象授权一个新的令牌时，DPIS 将会增加对象版本向量号并通知最新的副本。DPIS 和所有的这些副本将最新的版本向量号在它们的持久化状态中进行记录，所有这些操作发生在通知任何客户端之前，因而这些操作发生在客户端可以对对象开始实施写操作之前。如果另一个副本当前是不可用的，那么它的对象版本号就不会增加。当这台 MDS 重新启动并向 DPIS 报告它所拥有的目录对象的集合以及与这些对象相关联的版本向量号时，DPIS 将会依据版本向量号检测到这台 MDS 拥有一个过时的副本。如果 DPIS 发现一个大于其记录中的版本号的话，DPIS 会认为它在授权令牌时失败了，于是会取这个较高版本的对象副本作为最新的对象副本。当客户端或 MDS 进行操作时都会对版本向量号进行校验，以确保它所访问的数据是最新的数据。

在进行了成功的数据变更之后，并经过了较长时间之后，组件的失效仍可以导致数据的损毁或损坏。OCFS 系统通过在 DPIS 和所有的 MDS 之间的心跳信息来标识失效的 MDS 并通过校验和操作（checksumming）来检测数据是否损坏。一旦发现问题，数据会从有效的对象副本中尽可能快地恢复过来，只有当 OCFS 系统不能在几分钟内对所有的副本都丢失的情况作出反应的情况下，一个对象才是被彻底地丢失掉了。即使是在这种情况下，也是数据块变得不可用了，而不是损坏了，客户端应用程序接收到的是明确的错误而不是损坏的数据。从而保证了用户看到的是一致性的数据。

§4.3 设计与实现

目录对象副本的交互图见图 4.1。

其实现如下：

```
// 初始化；
void init() {}
// 创建与主副本的链接；
public Secondary_DPIS_Node(Configuration conf)   throws IOException {}
// 为目录对象增加一个副本；
void add(Object dir_object, int numReplicas) {
    synchronized ()
}
void incrementReplicas(int increment) {}
// 减少一个目录副本；
void remove( dir_object) {
    synchronized ()
}
void decrementReplicas() {}
// 统计目录对象副本个数；
int getNumReplicas(Block block) {}
int getNumReplicas() {}
// 返回目录对象大小；
long size() {}
// 记录目录对象的时间向量；
Object[] getTimedofObjects() {}
long getTimeStamp() {}
void setTimeStamp() {}
// 副本之间“心跳”监控；
class PendingReplicationMonitor implements Runnable {
        pendingReplicationCheck();
        Thread.sleep(period);
    }
```

```
// 副本放置；
{
// 打开已知文件名的文件，返回数据对象及相应的 OSD 节点信息；
public LocatedObject[] open(String src) throws IOException;
// 创建新的文件，得到数据对象及相应的 OSD 节点信息；
public LocatedBlock create( String src, String clientName, boolean overwrite, short
                                    replication, long blockSize ) throws IOException;
// 向 OSD 节点写完数据对象的客户端向 DPIS 报告其已写完毕，客户端不能得到额外的数据对象，除非明确得到写完或放弃的消息；
public void reportWrittenObject(LocatedObject b) throws IOException;
// 如果没有报告写完，则放弃；
public void abandonObject (Object b, String src) throws IOException;
// 添加数据对象并返回 OSD 节点信息；
public LocatedObject addObject (String src, String clientName) throws IOException;
// 放弃当前正在写的文件；
public void abandonFileInProgress(String src, String holder) throws IOException;
// 返回所有对象副本的信息，不论对象是否得到正确的写操作；
public boolean complete(String src, String clientName) throws IOException;
// 报告 OSD 节点上指定位置的冲突对象；
public void reportBadObject (LocatedObject [] blocks) throws IOException;
// 获取所有数据对象的列表；
public DFSFileInfo[] getListing(String src) throws IOException;
// 返回所有数据对象的存储节点的主机信息；
public String[][] getHints(String src, long start, long len) throws IOException;
}
// 副本迁移；
{
// 对数据对象进行加锁；
public boolean obtainLock(String src, String clientName, boolean exclusive) throws
IOException;
// 释放锁；
```

```
public boolean releaseLock(String src, String clientName) throws IOException;
// DPIS 撤消客户端已认为处于“死亡”状态的 OSD 节点上数据对象的锁；
public void renewLease(String clientName) throws IOException;
// 获取当前数据对象的统计信息；
public long[] getStats() throws IOException;
// 获取系统当前所有有效的 OSD 节点的完整报告信息；
public DatanodeInfo[] getDatanodeReport() throws IOException;
// 为指定的文件获取数据对象大小；
public long getObjectSize(String filename) throws IOException;
// 通知 DPIS 重新读取新的 OSD 节点上的数据对象；
public void refreshNodes() throws IOException;
}
// 副本更新；
{
// 数据对象副本之间互相建立套接字连接；
public static InetSocketAddress createSocketAddr(String target) throws IOException {}
// 周期性的对数据对象进行更新；
public void doUpdates(MetricsContext unused) {
                synchronized void readBytes(int nbytes) {}
                synchronized void wroteBytes(int nbytes) {}
                synchronized void readObjects(int nobjects) {}
                synchronized void wroteObjects (int nobjects) {}
                synchronized void replicatedObjects (int nobjects) {}
                synchronized void removedObjects (int nobjects) {}
                }
}
// 副本自身一致性的检查；
{
// 获取原始的数据文件
public File getRawFile() {}
// 对数据对象进行校验和计算；
```

```
public ChecksumObjects(Object object){}
// 计算得到保存校验和的文件
public Path getChecksumFile(Path file) {}
// 对是否是校验和文件进行判断;
public static boolean isChecksumFile(Path file) {}
// 得到校验和文件的大小;
public long getChecksumFileLength(Path file, long fileSize) {}
// 对校验和文件进行检验;
private void verifySum(int delta) throws IOException {}
}
// 副本之间一致性的检查;
{
// 对对象副本之间的一致性进行检查;
        void isConsistent() throws IOException {
          if( objectTotal == -1 && objectSafe == -1 ) {
            return; // manual safe mode
          }
          int activeObjects = dir.activeObjects.size();
          if( objectTotal != activeObjects )
            throw new IOException( "objectTotal " + objectTotal + " does not match all
objects count. " + "activeObjects = " + activeObjects + ". safeObjects = " + objectSafe + "
safeMode is: " + ((safeMode == null) ? "null" : safeMode.toString()) );
          if( objectSafe < 0 || objectSafe > objectTotal )
            throw new IOException( "objectSafe " + objectSafe + " is out of range [0," +
objectTotal + "]. " + "activeObjects = " + activeObjects + " safeMode is: " + ((safeMode ==
null) ? "null" : safeMode.toString()) );
        }
}
```

§4.4 高可用性分析

对于系统的可靠性和可用性分析，一般采用马尔可夫激励（Markov Reward Model，MRM）模型[37]。MRM 模型把目标系统的状态 S 分为两类：

$S = A \cup N$

其中 A 表示可靠状态集合，N 表示不可靠状态集合。

若只考虑不可靠状态集合 N，可以通过公式 4-1 和 4-2 计算出系统到达不可靠状态前的平均时间。

$$L_N(\infty)Q_N = -\pi_N(0) \tag{4-1}$$

$$MTTDL = \sum_{i \in N} L_i(\infty) \tag{4-2}$$

其中，$L_N(\infty)$为到达不可靠状态前的平均时间向量，Q_N为状态转移概率矩阵，$\pi_N(0)$为初始时的状态概率向量，MTTDL 为系统的平均无数据丢失时间（Mean Time To Data Loss）。

冗余集包括同一数据对象的两个副本，被分布到不同的两个存储节点中。定义目标系统冗余集的系统状态如下：

- 状态 0：两个数据对象均可用，数据对象可用；
- 状态 1：其中一个数据对象失效，数据对象可用；
- 状态 2：两个数据对象同时失效，数据对象不可用；

图 4.4 显示冗余集的状态转移模型，其中 μ 和 ν 分别表示冗余集的失效率和修复率。系统的可靠状态集合包括状态 0 和 1；不可靠状态集合包括状态 2。

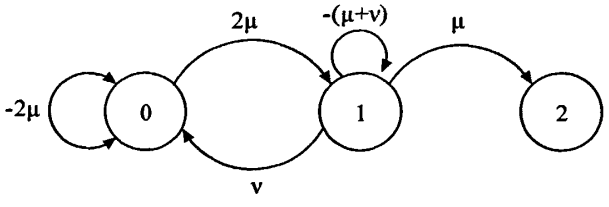

图 4.4　冗余集状态转移模型

不可靠状态集合 N 的状态转移概率矩阵和初始时的状态概率向量分别为：

$$Q_N = \begin{pmatrix} -2\mu & 2\mu \\ \nu & -(\mu+\nu) \end{pmatrix}, \quad \pi_N(0) = \begin{pmatrix} 1 & 0 \end{pmatrix}$$

由公式 4-1、4-2 求得：

$$MTTDL = \frac{3\mu + \nu}{2\mu^2}$$

表 4.2 存储系统参数

参数	意义	取值
T	存储系统总的存储数据容量	1PB
R	单个冗余集合的存储数据容量	可变
S	存储系统包含的冗余集合数量	T/R
$MTTF_{disk}$	磁盘的平均无故障时间	10^5小时
γ	故障修复率	100GB/小时

假定存储系统中的参数定义见表 4.2，在冗余集中，有：

$$\mu = \frac{1}{MTTF_{disk}}, \quad \nu = \frac{\gamma}{R}$$

由于 ν>>μ，所以有：

$$MTTDL_{冗余集} = \frac{3\mu + \nu}{2\mu^2} \approx \frac{\nu}{2\mu^2} = \frac{\gamma \cdot MTTF_{disk}^2}{2R}$$

存储系统包含的冗余集合数量为 $S = \frac{T}{R}$，并且 $\frac{1}{MTTDL_{冗余集}}$ 很小，简化计算后得到：

$$MTTDL_{存储系统} = \frac{1}{1-(1-\frac{1}{MTTDL_{冗余集}})^S} \approx \frac{MTTDL_{冗余集}}{S} = \frac{\gamma \cdot MTTF_{disk}^2}{2T}$$

按照表 4.2 的参数计算，基于对象副本的存储系统的平均无数据丢失时间大约为 57 年。

§4.5 实验测试与性能分析

4.5.1 实验测试方法

系统基于 JAVA 平台设计，开发实现了元数据对象副本管理的原型系统。所有节点使用 Red Hat 9.0 操作系统（内核版本为 2.4.20），开发环境是 J2SE SDK 1.5 + Eclipse 3.2。

4.5.2 功能测试

功能测试的环境是元数据服务器集群包括三台 MDSs，目录对象副本个数设置为两个。

当人为地将系统中某台 MDSs 存储设备关机以模拟该 MDS 设备宕机时，通过对 DPIS 上的日志信息的分析，可以看到 DPIS 丢失了来自于该 MDS 发出的“心跳”信息，并且将该 MDS 设备标注为“不活跃（dead）”状态，于是 DPIS 自动将该 MDS 设备上拥

有的目录对象元数据自动复制到其它另一个有效的 MDS 设备上。进一步，通过对该台 MDS 设备的数据访问，可确定该 MDS 上确有该副本。这些实验表明元数据服务器集群是高可用的，OCFS 文件系统可完成目录对象的自动容错。

4.5.3 性能测试

性能测试是在功能测试正确的基础上，对比采用传统方法与本章的对象副本方法在时间开销上的不同，通过记录关键参数在测试过程中的数据并进行分析。

以下记录了采用传统方法与 OCFS 副本方法实现目录对象副本在容错花费时间开销上的对比。

系统对不同大小的目录对象数据均采取了相同程度的冗余，分别对 1K，2K，4K，8K，16K，32K，64K 大小的目录对象进行了测试。副本拷贝操作的时间是对不同大小的纯文本文件的拷贝时间进行统计，两个副本的拷贝操作时间是该时间的两倍乘积。传统方法指的是向一台 MDS 拷贝完一个副本后再向另外一台 MDS 拷贝一个副本。其对象副本复制所花费的时间与传统方法所花费的时间比较如图 4.5：

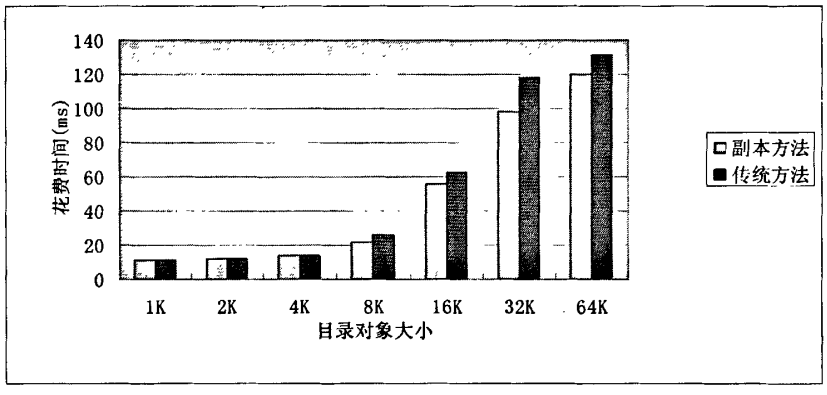

图 4.5 不同目录对象的副本花费时间比较

从图 4.5 可以看出，由于采用并发对象副本复制策略，本章的方法在时间开销上具有明显优势。这里需要说明的是，由于本章的 OCFS 系统在进行副本的实际创建过程中采用的是 4K 大小的缓冲区，所以对于 1K，2K，4K 大小的目录对象副本的创建，两种方法分别所花费的时间是相同的，但当对象副本的大小大于 4K 时，本章的副本方法就显示出比传统方法明显的优势。

4.5.4 实验结果分析

通过以上测试和分析可以得出：本系统具有高可扩展性和高可用性等特点，根据存储需要可以扩充存储节点，支持 PB 级的海量数据存储。同时，由于对每个文件的元数据采用了多副本的策略，并通过相应的机制和算法保证元数据副本间的负载均衡以及一

致性，从而实现了元数据集群的高可用性，系统的可靠性同时得到提高。如果某节点出现故障，系统会自动对失效节点的数据进行迁移，重新复制备份至其它节点中，整个系统的性能不会受到影响。

§4.6 小结

在本章中，在对象存储体系结构的基础上对其进行了扩展，提出一种基于对对象存储体系结构进行扩展的分布式文件系统元数据对象容错的管理方法。该方法将分布式文件系统中的元数据对象进行了进一步的抽象，同时对该元数据对象进行了多个副本的备份，并将它们存储在元数据服务器集群中。通过采用多个元数据对象副本的方法既避免通过增加元数据备份服务器带来的较高的成本又避免了多台元数据服务器对共享元数据的并发访问而造成的性能损失，从而很好地满足了元数据访问的高吞吐率和高可用性。

第五章 基于日志的元数据管理的高效检查点方法

§5.1 引言

随着集群系统及分布式系统逐渐被广泛地用作并行分布系统平台，组成一个计算机系统的软硬件日益复杂。伴随而之的是随着系统规模的扩大，故障率也是呈指数同步增长。而在使用集群系统或分布式系统进行计算的应用程序一般都是大规模的科学工程计算任务，这类任务的执行时间一般都比较长，一旦计算节点发生异常事件，将导致系统运行失败，任务不得不从头开始执行，要使集群系统及分布式系统在各个领域中得到广泛的应用，则必须为其提供容错功能和快速恢复功能。为了避免系统在发生上述事件后由于从头开始执行而引起计算上的大量浪费，充分提高集群系统的可用性，可以在系统正常运行的适当时刻设置检查点，保存系统当时的状态。检查点（checkpointing）设置[38]、回滚（roll-back）恢复技术是实现并行分布系统容错的一种重要的有效手段。它的主要目的是为了节省运行时间，即能够回滚到最近一次保留的结果处继续往下运行，而不是从头开始。其基本思想是在应用程序的正常运行过程中，定期（或不定期）地设置检查点，保存系统当时的一致性状态（设置检查点），当系统发生故障，回滚恢复重新运行时，应用程序可以回滚到最近一个检查点处继续运行，而不必从头开始程序的执行，从而有效地提高系统的可用性。

下图 5.1 展示了未设置检查点与设置检查点在恢复时间上的对比：

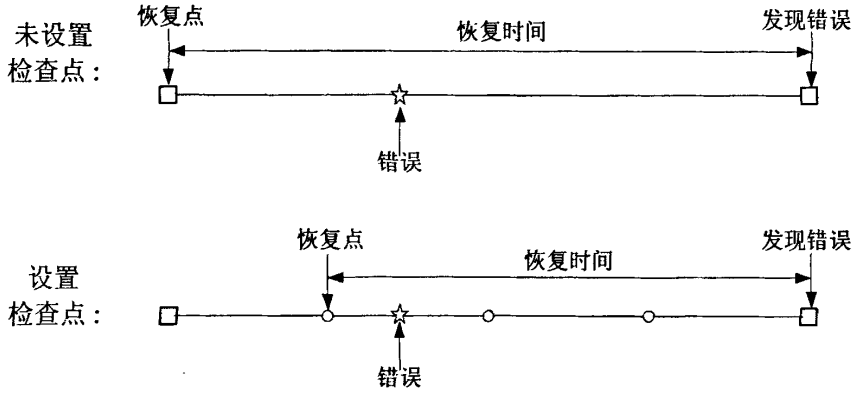

图 5.1 未设置检查点与设置检查点在系统恢复时间上的对比

检查点的快速恢复过程原理如下：

checkpoint i

file_operation (open，close，read，write，lseek，create，unlink)

failure occurs　　// roll-back to checkpoint i

checkpoint i+1

随着系统规模日益逐渐扩展到PB级（10^{15}字节）系统，保持文件系统的高可用性、快速恢复能力以及高效率的访问是十分重要的。而保障实现系统高可用性、快速恢复以及高效率的系统访问的一种策略机制就是对系统的元数据进行检查点操作，以实现系统状态“在线（on-line）”的保存能力，正逐渐成为健壮性系统、容错系统中的一个基本组成部分。

元数据对象检查点方法将文件系统的状态“在线（on-line）”地保存集成到大型的基于对象的分布式文件系统中。根据检查点的概念，通过在确定的时间间隔内对文件系统的当前状态实施检查点操作来对文件系统的当前状态予以永久性的保存。当发生系统失效或其它故障后，系统可以回滚至最近一次保存的系统状态，系统根据该系统状态进行恢复，重新开始继续运行，而不是从头开始执行，从而实现保证了系统的健壮性、容错性及高可用性，同时也保证了系统的高效率访问。

与此同时，对PB级系统中大量的元数据采取适当的策略来进行元数据数量数目的减少以及存储空间开销的节约也能为系统的快速恢复能力提供帮助。

以下以数据量化说明采取适当的策略来减少元数据存储空间开销的需求：

假设每个元数据记录为128字节，一个文件系统拥有一亿（$1*10^8$）个文件。由于每个文件（包含目录文件，在UNIX系统中，目录被视为一种特殊的文件）都需要一个元数据记录项或目录项，那么对这样一个规模的存储系统仅仅保存元数据就需要（128Byte*1*10^8）/（1024B/KB*1024KB/MB*1024MB/GB） ≈ 12GB。而对于一个多PB（Multi-PB）级的文件系统可能包含10亿多个文件，甚至是上100亿多个文件，那么，则分别需要大约12GB*10 = 120GB，12GB*100 = 1200GB ≈ 1TB的存储空间开销。因而对于一个包含有100亿多个文件的海量PB级（10^{15}字节）存储系统，仅元数据的存储空间开销就达到了TB （10^{12}字节）。

同时，如果对每个文件的每个版本都保留一份单独的拷贝，则该文件的每份拷贝都拥有一个单独的元数据记录项。现在假设每个文件拥有三份不同的拷贝，那么对于一个包含有100亿多个文件的海量PB级（10^{15}字节）存储系统则仅仅保存元数据就需要大约3TB的存储空间开销。而这种情况还仅仅只是不将额外的文件拷贝所带来的一级间接块，二级间接块等存储空间开销计算在内的理想情况，实际情况中的由于多个文件拷贝

而带来的元数据存储空间的开销则更是惊人。

由于元数据对象集合越小，越能很快地访问到客户端应用程序所需要的文件，所以如果能采取恰当有效的策略来减少元数据存储空间的开销，那么就将会有极大的实际应用价值。因为它不仅减少了元数据存储所需要的存储空间开销，与此同时，还可以实现元数据的快速访问，这样就既节省了空间又提高了访问效率，实现了系统的快速恢复，节省了时间，使得系统在空间和时间两方面都能得到良好的表现。

§5.2 基于日志的元数据管理的高效检查点方法

5.2.1 版本文件系统方法介绍

一个文件的每次修改就导致产生了该文件的一个新版本。与用新文件替换旧版本文件的方法不同的是版本文件系统（Versioning File System）会将新旧文件都保存起来。该系统的用户可以象访问最近的新文件一样去访问该文件的历史版本。传统的版本文件系统对每个版本的文件的元数据都保存了一份完整的拷贝。尽管这种方法简化了对不同版本的文件的访问，但是，它占用了大量的磁盘空间，因为即使是对文件数据或属性的一个很微小的变化，它也要对文件的元数据保留一份完整的拷贝。同时，在实际的工作中发现，版本文件系统设计的关键问题在于有效地对不同版本的文件的不同版本的元数据进行编码，因为不同版本的元数据是对实际的不同版本的数据进行跟踪的额外附加信息。尽管“写时拷贝（Copy-on-Write）”技术可以减少对不同版本的数据进行记录的开销，但是，传统的版本文件系统对不同版本的元数据的保存仍需要一个或多个元数据块，这样就占用了大量的磁盘空间开销。而据统计来说，不同版本的元数据通常需要与不同版本的数据大致相当的存储空间。所以，有效地减少不同版本的元数据的存储空间开销是非常必要的。

与此同时，版本文件系统为系统管理员和用户提供了从系统失效中进行恢复的便利和好处。当系统失效时，系统管理员通常对由于系统失效而造成的系统损害一无所知。由于这一点，他们可以用以往保存的“良好的”文件系统状态来对整个文件系统状态进行还原恢复。而这种方法通常所使用的技术便是实施检查点操作。检查点操作就像是为文件做了一个备份。检查点包含了在一个特定时刻，文件系统中每个文件状态的一个版本。这样，带有检查点的文件系统就为不同时间点的文件系统状态保存了一个系统映像，以便在文件系统失效时从最近的检查点处进行文件系统的恢复，从而节省了从文件系统的初始状态进行恢复的时间开销。

5.2.2 对版本文件系统进行的改进

在本章中，系统使用基于日志的元数据方法来实现对不同版本的元数据进行紧凑的

存储，以节省磁盘空间开销。

同时在本章中，系统采用对元数据实施检查点操作的方法来实现对不同时间点的元数据状态进行保存，以节省系统状态的恢复时间开销。

实施检查点的流程如图 5.2：

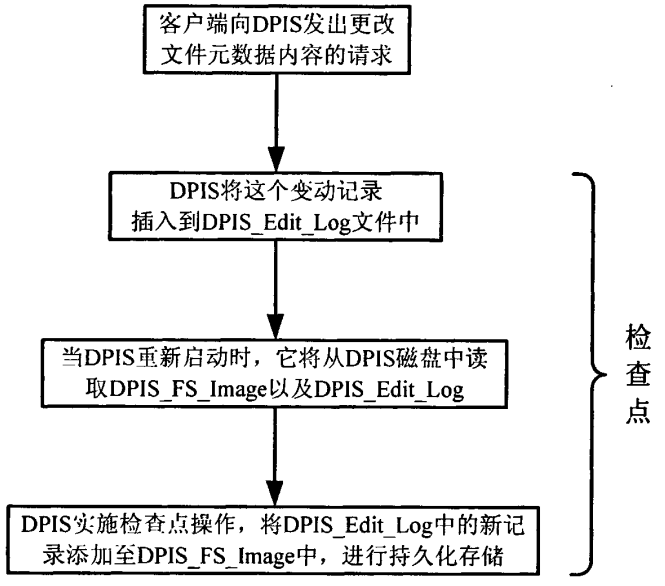

图 5.2　系统实施检查点的流程图

在日志文件中，为有效地保存元数据对象的各个版本， OCFS 系统使用了基于日志的元数据管理，它使用紧凑的日志条目来替换大多数的元数据实例。每次对数据对象的更新便生成了一个新版本的元数据，也就是一个新的元数据。例如，当由 inode 间接块所指向的数据对象发生修改变化时，那么间接块也必须进行新版本的记录。

5.2.3 基于日志的元数据对象检查点方法的关键技术描述

对元数据设置检查点的基本首要目标是在某一特定的时间点上对文件系统的状态予以保存。而在对象存储系统中实现检查点机制是非常具有挑战性的一项工作，因为一方面实施检查点操作是由 MDS（Metadata Server）集群所发起的，这将会影响到数据的 I/O；另一方面文件的 I/O 操作是以并行和分布的形式发生的，客户端同 MDS 进行交互获取访问权限，在获取相应的访问权限后，客户端即可不受任何控制地直接向 OSD 进行 I/O 操作，因为是由 OSDs 负责对之前写入的数据进行正确的存储以实现达到基本的“写时拷贝（Copy-on-Write）”行为。

5.2.4 对版本文件系统实施检查点操作方法的体系结构

为安全起见，文件系统中基于日志的元数据的更新必须及时很快地刷新保存到磁盘上，与此同时，由于 Cache 缺失而引起的对元数据的读操作必须非常高效以提高系统的

性能。为对较少的写操作提供高带宽和较多的读操作提供高效率的数据布局分布，MDS采用了一种两层的存储结构：（1）每台 MDS 对提交至准备持久存储至磁盘上的元数据的更新保留一份顺序记录的日志，（2）当插入日志中的条目数目达到设定的日志边界大小时，它们则被一次性地通过组提交（group committing）存储至持久存储的磁盘上，或者采用固定时间间隔、周期性地通过组提交存储至磁盘上。这种对众多的更新进行延迟写（delay writing），将众多的更新组成较大的组来一次进行持久化的存储的策略使得它们仅仅进行一次磁盘 I/O 操作就可以实现所有更新的提交。这样就可以显著地提高系统的读写性能，并使磁盘带宽得到最大化的使用。

同时，为减少元数据的更新而引起的磁盘 I/O 操作而带来的性能损失，系统所采用的是将元数据的变化记录在一个日志中的方法来进行增强和改进。在周期性的间隔时（或添满日志文件大小时），将日志文件中的内容进行持久化存储的提交，存储至磁盘上时，便对系统做一次检查点（CP，checkpoint，consistency point）操作，对系统的一个一致性状态进行记录，而保存的元数据则被写至元数据对象（metadata object）中，并保存至目录层次结构中相应的目录中。通过在日志中对元数据的变化进行持久化存储的保留，使得不同版本之间的元数据可以得到接合（conjunction），这样不论是单个的 inode 版本或者是间接块的版本，都可以从这个日志中重新生成。

图 5.3 展示了对元数据使用传统的版本文件系统方法与采用日志记录方法这两种方法在磁盘空间开销方面的对比：

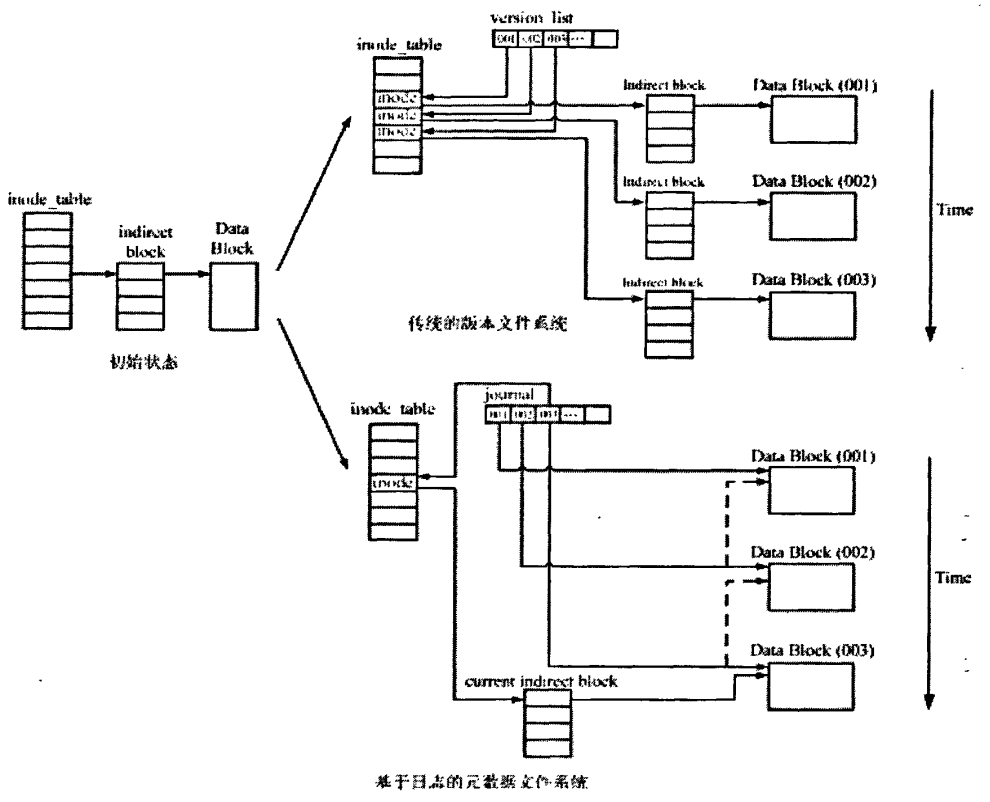

图 5.3　两种系统在磁盘存储空间开销上的对比

由于不同版本的数据块在磁盘上有不同的位置，所以，系统必须创建一个新版本的间接块来定位该位置。进一步，为定位新版本的间接块，系统必须创建一个新的 inode 节点来定位该间接块。由于数据或元数据的任何变化将会导致生成新的 inode 节点，新的间接块，以及版本列表中一个新的条目，所以将会有更多的元数据操作。

对此，解决办法是在不同版本的元数据之间运用差分方法。而基于日志的元数据方法即是对创建并保存的不同版本的元数据进行差分机制。

不同版本的元数据之间的一个显著的特点就是这些元数据之间的实际改变变动通常是非常细小细微的。在传统的版本文件系统中，尽管每个文件的新版本都会有一个新的 inode 节点及间接块写入，但是，对元数据的变化则仅仅只是一个块指针的更新变动而已。因而，新的系统可以利用这些细微的改变变化来节省不同版本的元数据的存储空间开销。

基于日志的元数据存储保存了当前版本的元数据的一份完整拷贝以及之前的元数据变化的记录日志。当需要重建历史版本的元数据时，只需要对记录元数据变化的记录日志向后遍历进行恢复操作，直到取到自己所期望版本的元数据。这种恢复元数据变化的过程即为元数据记录日志的回滚。如图 5.3 所示，基于日志的元数据通过将各个版本的

数据块记录在一个日志条目中来保存各个版本的数据块。每个条目指向当前的新块以及前一次被覆盖的块，系统仅保留当前版本的 inode 节点信息及间接块，这样，就显著地减少了元数据的存储空间开销。

虽然基于日志的元数据方法比传统的版本文件系统元数据方法在空间利用率上更有效，但是，这种方法在性能上有些损失，因为它需要重建历史版本的元数据。由于当对历史版本的元数据进行读操作及回滚操作时，需要对日志中的元数据的当前版本及所期望要访问的元数据版本之间的每个条目都进行读操作及回滚操作，这样，在对历史版本的元数据进行重建时，就会有一定的性能损失，而且一个文件变动的次数越频繁，那么这种性能损失就越高。

减少这种系统性能损失开销的方法就是对系统实施检查点操作。周期性地对文件系统元数据做一个完整的拷贝并保存至磁盘上。通过保存众多的检查点，系统可以从时间上最近的检查点进行日志的回滚操作，而不是从最初的状态重新开始。

通过日志段在日志中保存每个数据对象的变化，每个日志段包含了日志项，这些日志项指向了该段中某个数据对象的变化。日志段以时间的顺序向后链接在一起以方便进行各个版本的重建。日志中保存了许多依时间排序的，大小不等的日志项（日志条目）。OCFS 系统正是使用这个日志来实现基于日志的元数据管理策略。每一条日志项（日志条目）包含了针对某个文件某次改变变动的信息，指向该文件在日志中的前一项（前一条目）的位置指针，通过该指针，系统可以通过时间来跟踪某个文件的变动。该项目（条目）包括写入的时间，指向数据块的指针，数据块所占的逻辑块的范围大小，历史文件的大小，指向被重写的历史数据块（如果有的话）的指针等。

5.2.5 恢复至检查点的具体实施过程步骤

当访问当前的文件数据时，OCFS 系统查找最新的 inode 信息并直接读取指向数据块的数据指针，因为它们是最新的。当访问历史数据时，OCFS 使用检查点跟踪和日志回滚二者结合的方法来重建所请求版本的数据的数据指针。

二者结合的工作方式如下：假设用户想读取某个文件在时间 T 时刻的数据。第一步：OCFS 查找定位在满足 Tcp>=T 的条件下所做的所有检查点操作中 Tcp 最小的那个检查点，第二步：从那个检查点处，通过日志向后遍历查找对欲读取的数据进行的变更或改变，如果找到了历史版本，就直接使用它，否则的话（如果没有找到的话），就从做了元数据检查点的 Tcp 处读取数据。第一步，第二步分别如图 5.4，图 5.5，图 5.6 所示。

第一步：

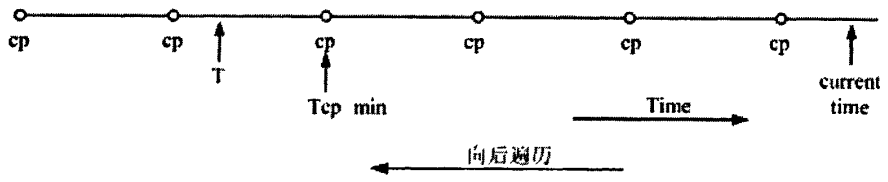

图 5.4 遍历查找最近的检查点

第二步：为简单起见，以 inode=7 的文件为例来说明问题。

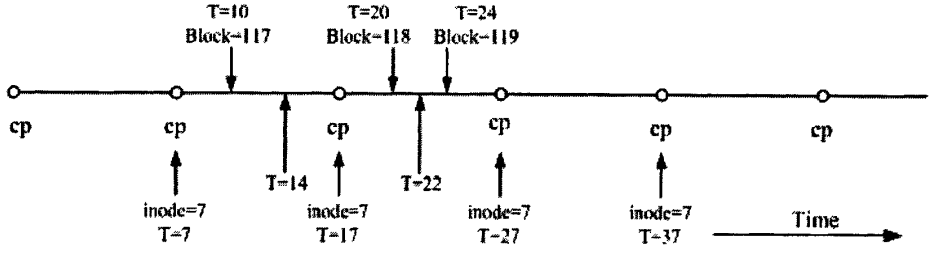

图 5.5 文件恢复示例一

1） 当欲读取 T=22 时刻 inode=7 的数据，OCFS 系统首先读取 T=27 时刻的检查点，然后读取日志条目，查看是否存在可用的数据块。在本例中，它发现日志条目中 T=24>T=22，即 inode=7 的文件数据内容在 T=24 时刻进行了重写，于是继续在日志条目中进行向后遍历的查询，返回至 T=20<T=22，于是读取到 T=20 时刻的文件数据。

2） 当欲读取 T=14 时刻 inode=7 的文件数据，首先读取 T=17 时刻的检查点，然后在日志条目中进行向后的遍历查询，发现在 T=14 时刻之后未对 inode=7 的数据进行修改变更（变动，改变）操作，于是从 T=17 时刻的检查点读取文件的数据。

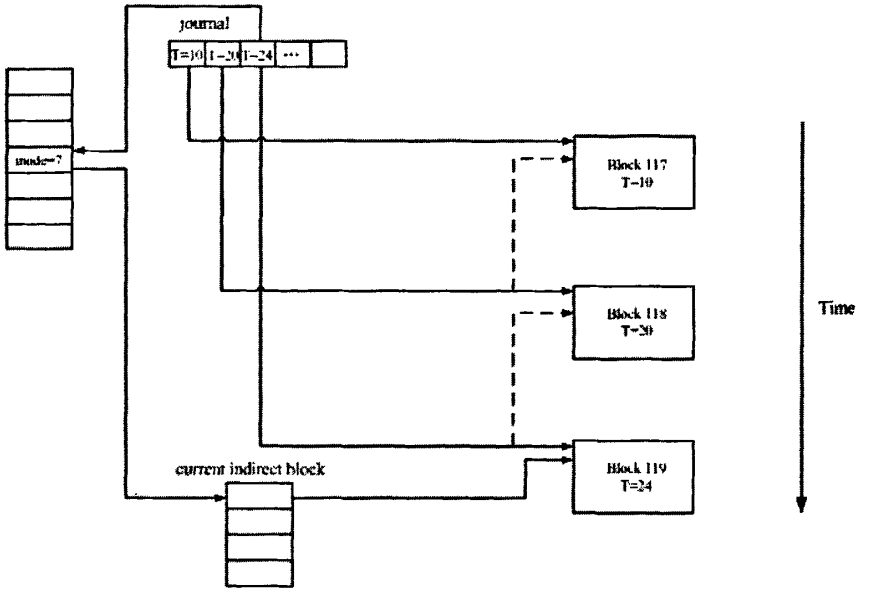

图 5.6 文件恢复示例二

§5.3 设计与实现

本章主要是对 OCFS 原型系统中元数据的持久化存储在文件系统层（OSD 层）采用了检查点设置技术，使得文件系统的元数据在容错性和高可用性方面保持良好的性能。它采用了一种低开销的日志记录策略，支持对元数据内容的检查点设置和回滚恢复，同时解决了磁盘写入过程中的故障恢复问题。

具体的实施过程步骤如下：

检查点设置流程：

1）保存非活跃文件的元数据当时的信息

2）保存活跃文件的元数据当时的信息

3）将活跃文件元数据的变更信息存入元数据日志文件中

4）将设置检查点时刻的元数据对象的数据段，栈段写入检查点

回滚恢复流程：

1）恢复元数据对象的数据段，栈段

2）恢复活跃文件的元数据信息

3）依据元数据日志文件恢复活跃文件元数据的变更信息

4）恢复非活跃文件的元数据信息

5.3.1 检查点方法的主要操作

对元数据集群系统中元数据实施检查点的主要操作有：

- OP_ADD：向日志文件中添加创建记录的操作；
- OP_RENAME：向日志文件中添加重命名记录的操作
- OP_DELETE：向日志文件中添加删除记录的操作；
- OP_MKDIR：向日志文件中添加创建目录记录的操作；
- OP_SET_REPLICATION：向日志文件中添加设置副本记录的操作；
- OP_OSD_NODE_ADD：在日志文件中对相应的新注册的对象存储节点创建一条记录；
- OP_OSD_NODE_REMOVE：在日志文件中对相应的对象存储节点的删除创建一条记录；

5.3.2 日志的主要内容及结构

日志文件的内容主要包含以下内容：

- 文件名
- 文件修改者
- 文件修改时间
- 文件修改操作
- 文件修改内容

其结构如表 5.1：

表 5.1 日志文件的内容结构

文件名	文件修改者	文件修改时间	文件修改操作	文件修改内容

5.3.3 基于日志的元数据检查点的具体实现

```
// 初始化输出流，以便进行日志的记录；
void create() throws IOException {}
// 如果不存在日志文件，则创建新的日志文件；
void createNewIfMissing() throws IOException {}
// 关闭存储的文件库；
void close() throws IOException {}
// 在进行任何日志操作时如果发生 I/O 错误，则将该目录从现有的目录列表中进行删除。如果不再存在有任何目录，则抛出一个异常，该异常声明该操作可能导致服务器退出；
void processIOError(int index) throws IOException {
```

```
        // 为新的日志文件保存相应的值，同时分配存储空间
        File[] editFiles1 = editFiles;
        File[] editFilesNew1 = editFilesNew;
        DataOutputStream[] editStreams1 = editStreams;
        // 将相应的值从旧的日志文件中拷贝至新的日至文件中，当遇到旧的日志文件中的错误时予以跳过；
        // 调用文件系统映像的 I/O 错误处理进程；
        MDS_FS_image.processIOError(index);
    }
    // 装载文件系统映像；
    void loadFSImage( Configuration conf ) throws IOException {
         // 检查是否存在检查点文件；
            if (ckptFile.exists()) {
                if (editLog.existsNew(idx)) {
                }
            }
     }
     boolean loadFSImage(Configuration conf, File curFile)   throws IOException {}
    // 保存相应的文件系统映像；
     void saveFSImage(String filename) throws IOException {}
     void saveFSImage() throws IOException {}
    // 更新时间
     void updateTimeFile(File timeFile, long timestamp) throws IOException {}
    // 删除指定的日志文件；
    void delete(int idx) throws IOException {}
    // 检查是否存在旧的日志文件，新的日志文件；
    boolean exists() throws IOException {}
    boolean existsNew() throws IOException {}
    boolean existsNew(int idx) throws IOException {}
    // 将文件系统映像的检查点以及日志中现有的内容保存到文件系统映像中，并打开一个新的日志文件
```

```
// 实施回滚至检查点的操作
void rollFSImage() throws IOException { rollFSImage(true); }
void rollFSImage(boolean reopenEdits) throws IOException {
    // 首先检查做过检查点操作的文件系统映像以及新的空日志文件是否存在于检查点目录中；
    if (!editLog.existsNew()) {}
    // 对新的文件系统映像文件进行重命名；
    // 更新对文件系统映像实施检查点操作的时间；
    long now = System.currentTimeMillis();
    updateTimeFile(timeFile, now);
    // 返回经过周期性实施检查点操作的文件系统映像的映像文件名；
    File getFsImageName() {}
    File[] getFsImageNameCheckpoint()
    // 装载记录的日志文件，并将日志中记录的变更内容周期性地应用（运用）实施到文件系统的内存结构中去。这一步是将之前持久化存储到磁盘空间上记录的日志内容应用到文件系统映像中；
    int loadFSEdits(Configuration conf, int index) throws IOException {}
    int loadFSEdits( Configuration conf, File edits) throws IOException {}
    // 向日志文件中进行相应的写操作；
    void logEdit(byte op, Writable w1, Writable w2) {}
    // 向日志文件中添加创建文件记录的操作；
    void logCreateFile( FSDirectory.INode newNode )
    // 向日志文件中添加创建目录记录的操作；
    void logMkDir( FSDirectory.INode newNode ) {}
    // 向日志文件中添加重命名记录的操作；
    void logRename( UTF8 src, UTF8 dst ) {}
    // 向日志文件中添加设置副本记录的操作；
    void logSetReplication( String src, short replication ) {}
    static short adjustReplication( short replication, Configuration conf) {}
    // 向日志文件中添加删除文件记录的操作；
    void logDelete( UTF8 src ) {}
```

```
// 在日志文件中对相应的新注册的对象存储节点创建一条记录;
void logAdd_OSD_node( OSD_node_Descriptor node ) {}
// 在日志文件中对相应的对象存储节点的删除创建一条记录;
void logRemove_OSD_node( OSD_nodeID nodeID )
// 关闭当前的日志文件，并打开一个新的日志文件；如果新的日志文件已经存在，则忽略此步或者返回一个错误;
void rollEditLogIfNeeded() throws IOException {}
或者是： void rollEditLog() throws IOException {}
// 删除旧的日志文件;
void purgeEditLog() throws IOException {}
void purgeEditLog(boolean reopenEdits) throws IOException {}
// 将新的日志文件更名为现时有效的日志文件名;
// 并对现时有效的这个日志文件进行重新打开;
// 返回日志文件名;
File getFsEditName() throws IOException {}
} // end of rollFSImage()，文件系统回滚结束
```

§5.4 实验测试与性能分析

5.4.1 实验测试方法

本系统是基于 JAVA 开发，开发实现了基于日志的元数据对象检查点方法的原型系统。测试平台是所有节点使用 Red Hat Linux 9.0 操作系统（内核版本为 2.4.20），开发环境是 J2SE SDK 1.5 + Eclipse 3.2。

实验以 1M 大小的纯文本文件为测试对象。假设对其做过两次修改，每次修改后的文件大小仍保持为 1M 的大小。现对该文件的三个不同的版本在采用不同的方法时，元数据在存储空间上的开销进行对比。

同时对检查点的设置策略是每一小时设置一个检查点，对设置检查点与不设置检查点两种条件下系统恢复的时间进行对比。

5.4.2 空间利用率测试

经过测试，得到在采用传统的方法时，该文件的三个不同版本的元数据的空间开销是 2743 个字节，在采用基于日志的元数据方法时，三个文件的元数据的空间开销合计是 567 个字节，如表 5.2。

表 5.2 磁盘占用空间对比

	传统文件元数据方法	基于日志的元数据方法
存储空间开销（Bytes）	2743	576

可节约的存储空间是：（1-567/2743）*100%=79.33%。

传统文件元数据记录方法与采用基于日志的元数据方法这两种方法在磁盘存储空间开销方面的对比图如图 5.7。

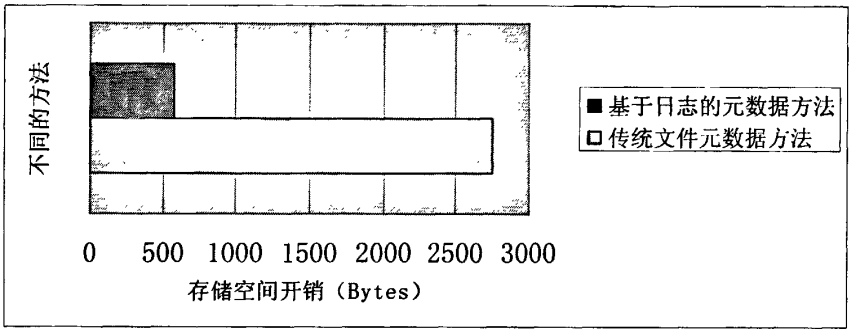

图 5.7 两种元数据方法在存储空间开销上的对比

5.4.3 系统恢复时间测试

实验对系统现有的文件、目录进行测试。测试环境对象是/home 分区，该分区的总容量是 4.55G，使用的空间大小是 1.97G，空间利用率是 46%。其下含有 5998 个目录，29630 个文件，目录层次最深的为 n=13 层，第 14 层的叶子节点是文件，目录层次最浅的为 n=1 层。

该分区的元数据总容量是 8.09M 的大小。

系统的检查点设置策略是每一小时设置一个检查点。

现在对设置检查点与不设置检查点两种条件情况下系统恢复的时间进行对比如表 5.3 及图 5.8。

表 5.3 系统恢复花费时间对比

	无检查点	有检查点
系统恢复花费时间（秒）	2470	1169

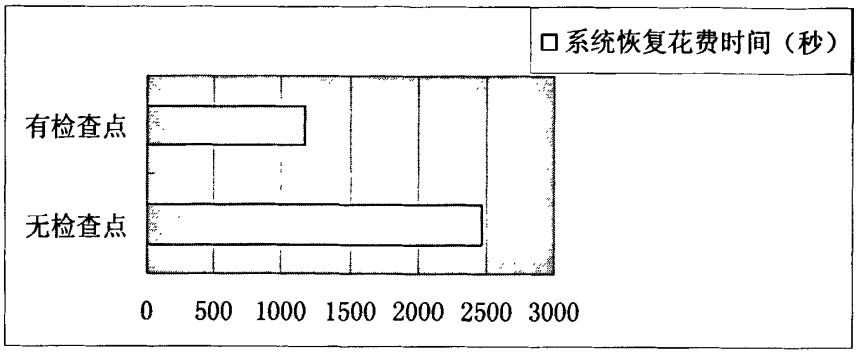

图 5.8 有无检查点的条件下系统恢复时间开销对比

5.4.4 实验结果分析

从以上测试结果可以看出：1）采用基于日志的元数据存储方法可以有效地节省磁盘存储空间开销，使得磁盘空间的利用更加充分，2）对基于日志的元数据进行恰当的检查点操作可以有效地减少系统失效时的恢复时间，从而保证了系统能够有一个较高的可用性。

§5.5 小结

本章讨论了 OCFS 系统采用基于日志的元数据管理方法有效地减少了系统对元数据的存储空间开销，使得磁盘空间的利用更加有效。同时，在基于日志的元数据基础上对元数据进行恰当的检查点操作，保存系统在当时的相关元数据状态信息，以便在系统出现故障时能够得到迅速的恢复，使得系统的恢复时间也更加有效。实际的实验测试表明该方法能够充分地利用磁盘存储空间，有效地减少系统失效时的恢复时间，在存储空间开销和恢复时间开销两方面均有良好的性能表现，从而保证了系统的高可用性。通过实际的实验结果证明了理论思想方法的正确性。

第六章 结束语

基于对象存储体系结构上的元数据集群研究正受到国内外学术界和产业界的广泛重视，同时也是成为国内外学术界和产业界的研究热点和难点问题。对它的研究具有非常重要的理论价值和实际应用价值，因而成为当前分布式系统领域的一个研究热点。基于对象存储体系结构上的元数据集群所涉及的问题也很广泛，本文对元数据集群中容错管理方法进行了一定的研究，主要包括采用基于元数据对象副本的方法和采用基于日志的元数据管理的检查点的方法实现元数据集群的高可用性，实现了高可用需求。

§6.1 工作总结

本文的主要研究内容和贡献如下：

（1）深入分析和研究了对象存储体系结构的对象模型、访问模型和优越特性，在现有对象存储系统的基础上，提出一种面向 PB 级存储系统的基于目录对象副本的高可用元数据管理模型。该模型通过采用高效的、并发的目录对象副本放置、更新、迁移策略以及管理机制，既保证实现了目录对象数据的可靠性和可用性，又增加了读取数据时的聚合网络带宽，提高读操作的性能。同时采用马尔可夫激励模型对该方法进行了定量可用性分析。实际的功能测试与性能测试，表明该方法能够有效保证存储系统的高可用性，提高元数据服务器集群的整体访问性能。与现有的存储系统相比，该结构模型能够实现元数据对象的高可用性，支持元数据服务器集群的容错管理。

（2）提出了一种对面向基于日志的元数据管理方法进行检查点操作的方法。通过对基于日志的元数据管理方法实施检查点操作，既保证了系统存储空间的有效利用，同时实现了系统的快速恢复，保证了系统的高可用性。当系统发生故障，进行回滚恢复，系统重新运行时，应用程序可以回滚到最近一个检查点处继续运行，而不必从头开始程序的执行，从而有效地提高系统的可用性，实现了元数据服务器集群的容错。实际的测试表明该方法能够充分地利用磁盘空间，提高系统的恢复时间，保证元数据的高可用性，提高元数据的访问效率。

总之，本文以系统研究面向基于 PB 级存储系统的元数据集群管理容错方法为出发点，对其中的若干关键技术进行了深入的研究，着重探讨了元数据集群中元数据对象的高可用性，并通过原型系统验证了研究成果的有效性和正确性，达到了预期的研究目标，为实现大规模的元数据集群容错管理提供了可靠的技术基础和保障。

§6.2 展望

基于对象存储体系结构上的元数据集群管理所涉及的研究范围很广，本文仅对其中

的元数据集群管理容错方法进行了一定深度的研究，提出了一点新的方法和思路。结合本人的研究工作进展和元数据集群存储技术的最新发展，在目前的基础上，今后的研究工作将在以下几个方面更进一步展开来进行：

（1）冗余机制

尽管元数据副本策略，基于日志的元数据策略已可以更好地满足实际应用的需求，但我们还在探索其它形式的机器间的冗余机制，比如奇偶校验码（parity code）或擦除码（erasure code）以满足我们日益不断增长的只读访问的存储需求。

（2）一致性模型

使用 MD5 码、数字签名等方法以较低的代价实现副本之间的一致性保证，以更好地实现元数据服务器集群的高可用性。

（3）元数据检查点策略

一是研究检查点的设置策略，是采用周期性的（固定间隔的）还是非周期性的（不固定间隔的）检查点设置操作，采用什么样的标准，而使系统能够在性能与可用性之间取得平衡。二是在一个做检查点的文件系统中，是否需要将历史的检查点予以删除，回收相应的存储空间以及何时将历史检查点予以删除。

（4）元数据组织、分配、放置、定位等管理算法

在元数据服务器集群中，元数据如何分布和管理对实现高效的元数据服务、元数据服务器集群的负载均衡和可伸缩性至关重要。高效的元数据表示、组织方法能够提供快速的元数据访问，支持大量用户同时对单个文件、目录或子目录的并发访问。

对子树分割（Subtree Partitioning）（静态（Static）、动态（Dynamic））、哈希（Hashing）（静态（Static）、动态（Dynamic）、文件（File）、目录（Directory））、延迟混合（LH，Lazy Hybrid）三种典型的方法仍有待进行更深一步的研究。

（5）负载均衡算法

研究负载均衡的元数据分布方法，支持元数据均衡地分布到不同的元数据服务器，避免访问瓶颈。

§6.3 小结

对本论文进行全文工作总结并做进一步的研究前景展望。

致　谢

在我的课题和硕士学位论文完成之际，谨向在我攻读硕士研究生学位的过程中曾经指导过我的老师，关怀过我的领导，关心过我的朋友，以及所有帮助过我的人们致以崇高的敬意和深深的感谢！

首先，衷心感谢我的导师刘仲教授！感谢刘老师在我的学习、工作和生活中所给予的悉心的指导、严格的要求和无私的关心。学习上，在我的课题研究期间，刘老师工作繁忙，但在百忙之中仍抽出时间对我进行全面、全方位的指导。从选题、研究、设计、直至撰写论文的各个阶段，刘老师以渊博精深的学识，严谨求实的作风，敏锐独到的洞察力，勇于开拓的思维，严谨、细致和求实的治学态度给予我悉心的指导和帮助，使我受益匪浅，同时，为我指明了方向，避免了不必要的弯路。刘老师高尚的品质和敬业的精神也是我学习的动力和楷模。工作中，刘老师的谆谆教诲和学术上的言传身教，为我将来的工作奠定了坚实的基础，将使我终身受益。刘老师的严格要求不仅将我由一名学化学的计算机门外人领进入了计算机科学的殿堂，而且更为重要的是还为我选取了具有一定技术壁垒的研究热点难点题目，使我个人在硕士研究生阶段的学习中具备了很强的竞争力。与此同时，生活中，刘老师还为我提供了良好的学术氛围，学习、实验环境和良好的硬件条件，解决了大量的昂贵书费以及多次在外参加会议的不菲费用，极大并非常有效地减轻了我的生活负担，缓解了我的后顾之忧。谢谢您，刘老师！

感谢学校研究生院，学院训练部教务处，学院政治部，学员大队的领导，硕士研究生队的领导等有关部门在我攻读硕士研究生期间给予的帮助和支持，是您们在日常生活中的关心、教育使我顺利地完成了硕士研究生阶段的学业；感谢硕士研究生队同队的全体同学，在过去的两年多时间里大家一起学习、工作，和我一起同甘共苦走过了硕士研究生阶段的美好时光，在这里既有技术上的支持，又能感受到大家庭的温暖，给我留下了许多美好的回忆。

感谢我敬爱的父亲母亲、兄嫂、姐姐姐夫在精神上和生活中给予我的鼓励、支持、关心和爱护。您们对我的支持和鼓励就是我不断前行、不断进取的巨大精神动力！您们教我的跌倒永不放弃使我勇敢、自信和坚强，您们教我的要拥有象大海一样宽广的胸怀使我坚持着自己的理想、信心和信念，使我时刻都能够勇敢地去面对迎面而来的海浪，迎面而来的困难和挑战。愿您们健康、快乐、开心！

感谢郭静茹女士，是你在精神上对我的鼓励和支持使我坚持并坚强着，使我能够有今天这样的成绩，是你在英语方面对我的提高和帮助，使我能够获得今天这样的成绩。愿你开心、快乐、幸福。

感谢我的好朋友胡志，盛刚李瑾夫妻，张健强，杜争平刘风云夫妻，是你们的鼓励使我重拾理想、信心与信念，让我更勇敢地去面对困难，迎接挑战。

感谢同宿舍的舍友郝鹏同学，丁烽祥同学，杨政军同学，我长期的、不规律的早起晚睡，怪异的行为与想法给你们带来了极大的不便，你们总是大度、宽容地对我。

感谢易乐天同学，赵克克同学，诸利勇同学，陈海坚同学，与你们在操作系统、文件系统方面的交流使我丰富了知识，有所长进。

衷心感谢所有给予过我帮助和支持，给予过我关心和问候的人！

最后，再次向所有在我攻读硕士研究生学位的过程中曾经指导我的老师，关怀我的领导，关心我的朋友和帮助我的人致以真挚的谢意！谢谢你们！

参考文献

[1] Panasas Inc. Object Storage Architecture. White Paper. http://www.panasas.com/objectbased_mgmt.html.

[2] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, Julian Satran. Object Storage: The Future Building Block for Storage Systems. IBM HRL (Haifa Research Laboratories).

[3] Nagle D, Serenyi D, Matthews A. The Panasas ActiveScale Storage Cluster Delivering Scalable High Bandwidth Storage. In: Benton V, ed. Proceedings of the ACM/IEEE SC2004 Conference. Washington: IEEE Computer Society, 2004. 53-62.

[4] Schwan P. Lustre: Building a File System for 1,000 Node Clusters. In: John WL, ed. Proceedings of the 2003 Ottawa Linux Symposium, Ottawa: Red Hat, Inc., 2003. 401-407.

[5] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google File System. In: Larry P, eds. Proceedings of the 19th ACM Symposium on Operating Systems Principles. New York: ACM Press, 2003. 19-22.

[6] Lustre. http://www.lustre.org. 2003.

[7] P.J.Braam. The Lustre Storage Architecture. Technical Report. Cluster File System, Inc. 2002. http://www.lustre.org/docs/lustre.pdf.

[8] Cluster File System, Inc. Lustre: A Scalable High-Performance File System. White Paper, 2003.

[9] 刘仲. 基于对象存储结构的可伸缩集群存储系统研究. 博士学位论文, 国防科学技术大学, 长沙, 2005.

[10] Drew Roselli, J. R. Lorch, and T.E. Anderson. A Comparsion of File System Workloads. In Proceedings of the 2000 USENIX Annual Technical Conference, San Diego, Calfornia, pages 41-54, June 2000.

[11] 于宁斌. IBM UNIX&Linux-AIX 5L系统管理技术. 电子工业出版社. 2007.

[12] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, Ethan L. Miller. Dynamic Metadata Management for Petabyte-scale File Systems. In: Benton V, ed. Proceedings of the ACM/IEEE SuperComputing (SC2004) Conference. Washington: IEEE Computer Society, 2004. 4-15.

[13] 王涌, 张宁, 彭建明, 刘仲. 基于Reed-Solomon编码的数据容错算法的设计与实现. 中国计算机学会第十一届计算机工程与工艺学术年会. 2007.

[14] 尹俊文, 邹鹏, 王广芳, 徐长梅. 分布式操作系统（修订版）. 长沙: 国防科技大学出版

社, 2004.

[15] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D.E.Long, Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06), USENIX Association Berkeley, CA, USA. November 2006. 22-35.

[16] Tanenbaum, T. Litzkow. The condor distributed processing system. Dr. Dobb's Journal, 1995, (2): 40-48.

[17] J Litzkow et al. Condor – A Hunter of Idle Workstation. In: Proc of the 8th IEEE Intl Conf on Distributed Computing Systems. Los Alamitos, CA: IEEE CS Press, 1988: 104-111.

[18] Pinheiro E, Bianchini R. Nomad. A scalable operating system for clusters of unix and multiprocessors. Proceedings of the 1st IEEE International Workshop on Cluster Computing, December, 1999.

[19] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, Chandra Kintala. Checkpointing and its applications. In: Proceedings of the IEEE 25th Symposium on Fault-Tolerant Computing. 1995: 22~31.

[20] James S Plank, Micah Beck, Gerry Kinsley et al. Libckpt: Transparent checkpointing under UNIX. In Proceeding of the 1995 USENIX Technical Conference. 1995: 213~223.

[21] Plank J S, Beck, M. Libckpt: transparent checkpointing for parallel programs. IEEE Trans. Par. Distr. Syst. Aug. 1994. 5: 874～879.

[22] Plank J S. Efficient checkpointing on MIMD Architectures. PhD Thesis. Dept. of Computer Science, Princeton University, June, 1993.

[23] 汪东升, 沈美明, 郑纬民. 一种基于检查点的卷回恢复与进程迁移系统. 软件学报. Jan 1999, 10(1): 68～73.

[24] Pei Dan, Wang Dong-sheng, Shen Mei-ming. Quasi-asynchronous migration: a novel migration protocol for PVM Tasks. Operating Systems Review. April 1999, 33(2): 5～14.

[25] Wang Dong-sheng, Zheng Wei-min, Wang Ding-xing, Shen Mei-ming. Checkpointing and rollback recovery for network of work tations. Science in China Series E-Technological Sciences APR, 1999, 42(2): 207～214.

[26] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, Gregory R. Ganger. Metadata Efficiency in a Comprehensive Versioning File System. Carnegie Mellon University. (2nd USENIX Conference on File and Storage Technologies, San Francisco, CA, Mar 31-Apr 2,

2003.)

[27] Craig A. N. Souls, Garth R. Goodson, John D. Strunk, Gregory R. Ganger. Metadata Efficiency in Versioning File System. Carnegie Mellon University.

[28] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Souls, and Gregory R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. Carnegie Mellon University. (In Proc. of the 4th Symposium on Operating Systems Design and Implementation, 2000.)

[29] Sage A. Weil. Scalable Archival Data and Metadata Management in Object-based File Systems. University of California, Santa Cruz. (CMPS 290s Project, Spring 2004.)

[30] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Kristal T. Pollack. Intelligent Metadata Management for a Petabyte-scale File System. University of California, Santa Cruz. Intelligent Storage Workshop, April 2004.

[31] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, Lan Xue. Efficient Metadata Management in Large Distributed Storage Systems. In: Miller E, Meter RV, eds. Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies. San Diego: IEEE Computer Society, 2003. 290-298.

[32] Teng Xu. Metadata Server Storage Management. University of California, Santa Cruz.

[33] 刘仲, 王涌. 面向PB级存储系统的高可用元数据管理模型. 2007中国计算机（CNCC）大会（苏州）（第四届）, 苏州. 2007.

[34] Xin Q, Ethan L. Miller, Darrell D. E. Long, Scott A. Brandt, Schwarz T, Litwin W. Reliability Mechanisms for very Large Storage Systems. In: Moore R, eds. Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies. Washington: IEEE Computer Society, 2003, 146–156.

[35] Ganger G. R, Kaashoek M. F. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In: Proceedings of the 1997 USENIX Annual Technical Conference. Anaheim: USENIX, 1997. 1-17.

[36] 刘仲, 周兴铭. 基于目录路径的元数据管理方法. 软件学报, 2007, Vol.18(2):2678-2687.

[37] Bolch G, Greiner S, Meer H, Trivedi K S. Queueing Networks and Markov Chains. John Wiley & Sons, Inc. Print ISBN 0-471-19366-6, 1998.

[38] Avi Ziv, Jehoshua Bruck. Performance Optimization of Checkpointing Schemes with Task Duplication. IEEE Transactions on Computers, Vol.46, No.12, December 1997.

[39] Zhong-wen Li, Yang Xiang, Hong Chen. Performance Optimization of Checkpointing Schemes with Task Duplication. Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06).

[40] Nith H. Vaidya. Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme. IEEE Transactions on Computers, Vol.46, No.8, August 1997.

[41] Elmootazbellah N. Elnozahy, James S. Plank. Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery. IEEE Transactions on Dependable and Secure Computing. Vol.1, No.2, April-June 2004.

[42] R. J. Honicky, Ethan L. Miller. Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution. Storage Systems Research Center, Jack Baskin School of Engineering, University of California, Santa Cruz. Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04).

[43] The Hadoop Distributed File System: Architecture and Design. http://www.hadoop.com.

[44] 刘仲, 王涌. 一种面向基于日志的元数据管理的高效检查点方法. 2007年全国高性能计算学术年会（深圳），深圳. 2007.

[45] Robert Love. Linux Kernel Development (Second Edition). Novell Press，机械工业出版社. 2006.4.

攻读硕士学位期间发表的论文

[1] 刘仲，王涌，章文嵩，邓鹏，王召福. OCFS：一种基于对象存储结构的可伸缩高性能集群文件系统. 通讯和计算机. Volume 4, No.6, Jun. 2007.

[2] 王涌，张宁，彭建明，刘仲. 基于Reed-Solomon编码的数据容错算法的设计与实现. 中国计算机学会第十一届计算机工程与工艺学术年会（合肥）（第八届），合肥. 2007.

[3] 刘仲，王涌. 面向PB级存储系统的高可用元数据管理模型. 2007中国计算机（CNCC）大会（苏州）（第四届），苏州. 2007.

[4] 刘仲，王涌. 一种面向基于日志的元数据管理的高效检查点方法. 2007年全国高性能计算学术年会（深圳），深圳. 2007.

面向PB级存储系统的元数据集群管理容错方法研究与实现

作者： 王涌
学位授予单位： 国防科学技术大学