中国科学技术大学

硕士学位论文

基于对象存储的分布式文件系统存储优化设计研究

作者姓名： 李子天

学科专业： 计算机软件与理论

导师姓名： 邢凯 副教授

完成时间： 二〇二〇年四月

University of Science and Technology of China

A dissertation for master's degree

Research on Storage Optimization Design of Distributed File System based on Object Storage

Author: Li Zitian

Speciality: Computer Software and Theory

Supervisor: Associate Prof. Xing Kai

Finished time: Apr, 2020

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等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。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

☑ 公开 ☐ 保密（____年） ☐ 控阅（3 年）

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

摘　要

对于目前的存储系统来说，为了实现指数级增长的大数据存储，需要提供大容量的存储空间以及快速的存储服务。在目前主流的分布式存储系统中，均可通过纠删码来节约数据中心磁盘成本并且保证数据的可靠性，从而满足应用程序和客户端的对存储系统的需求。然而由于纠删码的编码和解码需要大量的计算开销，因此如何提升基于纠删码编码的分布式文件系统的存储效率，是当前存储系统所研究的重点问题之一。对此，本文提出了如下两部分解决方案：

针对实际应用中数据往往重要程度并不相同，对数据可用性要求不一，且不同磁盘的故障率和可靠性动态不一的特点，需要我们对传统 RAID 存储方式包括基于纠删码的存储系统的冗余编码方案进行进一步研究。本文提出了一种面向数据可用性和磁盘可靠性动态要求的灵活自适应纠删码存储设计 On-demand ARECS（On-demand Availability and Reliability Oriented Adaptive Erasure Coded Storage System），综合考虑数据的可用性和存储设备的可靠性和性能等多项指标，通过计算确定纠删码编码的策略和数据存储的节点，从而减少存储系统的数据冗余度和延迟时间，进而同时提高存储系统的性能和数据安全性。我们在 Tahoe-LAFS 开源分布式文件系统中进行了实验，实验结果验证了我们的理论分析，在保证具有多样性要求的数据可用性和磁盘可靠性的前提下，明显减少了数据冗余度和存储延迟。

针对基于纠删码实现的分布式存储系统在数据进行修改时需要重新进行编码和解码，效率低下的特点，需要我们研究更加适合纠删码存储引擎的文件修改和追加算法。本文通过引入修改事件对象的概念，将文件的修改和追加操作抽象为单独的对象，进而保存于分布式对象存储系统中，从而避免了纠删码的重新编码和解码。我们通过应用一个无锁的事件队列来实现对修改事件队列的高效并发读写。针对修改事件过多可能导致的读取过慢的问题，我们设计了一种在系统空闲时进行数据合并的算法，从而避免数据碎片化的发生。我们在基于 On-demand ARECS 算法的 Tahoe-LAFS 分布式文件系统上进行了实验验证，实验结果显示基于修改事件的文件存储算法可以大大降低数据修改和数据追加时的纠删码编码和解码时间，提高数据存储速度。

关键词：分布式文件系统；纠删码；数据可用性；对象存储；文件存储

ABSTRACT

For the current storage system, in order to achieve exponential growth of big data storage, it is necessary to provide large-capacity storage space and fast storage services. In the current mainstream distributed storage systems, erasure codes can be used to save the cost of the disk in the data center and ensure the reliability of the data, thereby meeting the storage system requirements of applications and clients. However, because the encoding and decoding of erasure codes require a large amount of computational overhead, how to improve the storage efficiency of distributed file systems based on erasure codes is one of the key issues studied by current storage systems. In this regard, this article proposes the following two solutions:

In view of the fact that data is often of different importance in actual applications, the requirements for data availability are different, and the failure rates and reliability of different disks are dynamically different, we need to further study the redundant coding scheme for traditional RAID storage methods including erasure code-based storage systems. This paper proposes a flexible adaptive erasure code storage design for the dynamic requirements of data availability and disk reliability, which is called On-demand ARECS (On-demand Availability and Reliability Oriented Adaptive Erasure Coded Storage System), which comprehensively considers multiple indicators, such as data availability, storage devices reliability and performance to calculate the erasure coding strategy and data storage nodes, thereby reducing data redundancy and delay of the storage system, and thereby simultaneously improving the performance and data security of the storage system. We conducted experiments in the Tahoe-LAFS open source distributed file system, and the experimental results verified our theoretical analysis. On the premise of ensuring data availability and disk reliability with diverse requirements, data redundancy and storage delay were significantly reduced.

The distributed storage system based on erasure coding needs to re-encode and decode when data is modified. Due to its low efficiency, we need to study the file modification and append algorithms that are more suitable for erasure code storage engines. By introducing the concept of modification event objects, this paper abstracts the file modification and append operations into separate objects, and then saves them in a distributed object storage system, thereby avoiding the re-encoding and decoding of erasure codes. To solve the problem of too slow reading caused by too many

modification events, we designed an algorithm for data merging when the system is idle, so as to avoid the occurrence of data fragmentation. We conducted an experimental verification on the Tahoe-LAFS distributed file system based on the On-demand ARECS algorithm. The experimental results show that the file storage algorithm based on the modification event can greatly reduce the erasure code encoding and decoding time during data modification and data addition, and improve data storage speed.

Key Words: Distributed file system; Erasure code; Data availability; Object storage; File storage

目　录

第 1 章　绪　　论

1.1　研究背景与意义

近年来，随着信息技术的快速发展，互联网所产生和处理的数据量也在迅猛增长。有研究表明，过去两年内产生的数据量已经占世界数据总量的 90%[1]。因此，如何存储这些数据成为了存储系统相关研究的核心问题之一。传统上的集中式存储系统采用计算能力和 I/O 能力强悍的大型主机进行数据存储，然而对于 PB 量级甚至更高的数据量[2]，传统的集中式存储由于过高的设计和维护成本渐渐难以满足日益增长的存储需求。在目前的数据中心中，普遍采用大量的性价比更高的小型主机，通过分布式文件系统协同存储数据中心中的海量数据，其中常见的分布式文件系统有 Hadoop 分布式文件系统（Hadoop Distributed File System，即 HDFS）[3]和 Ceph 分布式文件系统[4]等，其架构如图 1.1 所示。在此背景下，如何在分布式文件系统中进行高效且可靠的数据存储，是目前存储领域研究的热点。

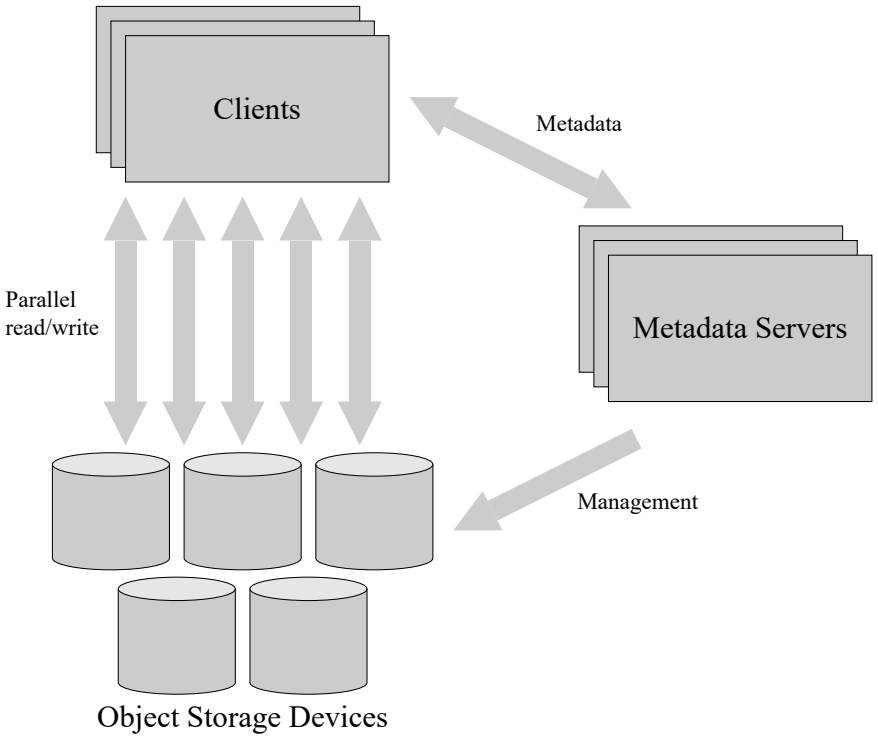


图 1.1　常见的分布式对象存储系统架构图

对于这些分布式存储系统来说，由于系统由大量的存储节点共同组成，其中每个节点均为不可靠的小型主机，因此发生存储节点故障是不可避免的。那

么如何在部分节点发生故障之后，保证分布式存储系统仍能够继续提供服务，并且有效地进行故障恢复，则是分布式存储系统研究中需要解决的重要问题之一。目前常用的分布式存储系统一般采用多副本备份技术来作为存储系统的容错性设计，但由于多副本备份技术引入了大量的数据冗余，导致数据的存储成本也被大大提高。近年来，有学者将纠删码技术应用于分布式存储系统的设计中[5–10]，以期在保障系统容错性的同时，通过数学计算大幅度减少恢复数据需要的冗余备份数据量。这些研究通过更好地使用纠删码进行数据的冗余备份存储，从而大大降低数据的冗余度，节约存储成本。

然而由于纠删码需要对一个数据块整体进行计算、备份与恢复，因此在数据块进行修改时，需要对整个块进行解码并重新编码，导致系统性能难以提高。因此，在目前的分布式文件系统中，纠删码的使用主要还是倾向于假设文件极少修改，或者文件内容修改的效率并不是分布式存储引擎主要优化目标。比如在 Ceph 等最新的分布式文件系统中，其纠删码引擎虽然实现了对象的修改，然而由于其直接将对象的修改操作映射为数据块的修改操作，因此在纠删码引擎上进行对象修改操作是较为低效的。然而，在分布式数据库或实时流计算等领域若应用此类存储系统，会导致性能急剧下降。故难以使用统一的存储结构实现分布式文件系统、分布式数据库系统和分布式流存储系统等各种不同类型的分布式存储系统。在此基础上，对于基于纠删码实现的对象存储引擎中数据的高效修改算法的研究，可以使基于纠删码的分布式存储系统能够更广泛地应用于不同存储领域，有利于在更多领域减少数据冗余备份所带来的额外开销。

1.2　国内外研究现状

在目前的分布式文件系统中，为了提高存储空间的利用率，普遍采用纠删码编码的方式代替多副本备份方式对数据的冗余进行存储。按照编码方式来分，在存储系统中应用的纠删码主要分为阵列码、低密度奇偶检查码和里德-所罗门码三类。按照存储空间利用率是否能达到理论上的最优来分，可以分为 MDS 编码（Maximum Distance Separable Codes）和非 MDS 编码两类。其中，里德-所罗门码是常用纠删码算法中唯一属于 MDS 编码且能任意选择数据块个数和冗余块个数的编码方案。因此，在分布式存储系统中，经常采用里德-所罗门码进行纠删码的编码。然而，目前大部分的分布式文件系统均采用了固定个数的数据块和冗余块，均为充分利用里德-所罗门码编码灵活的优势。如在 Ceph 和 HDFS 等分布式文件系统中，需要数据管理员在文件系统创建时分别制定数据块和冗余块的个数。针对这点，目前学术界提出了对编码策略进行改

进的几种方法，比如 Xiang 等[11]提出综合考虑磁盘的开销来确定纠删码编码的方案，Kadekodi 等[12]提出根据磁盘的可靠性来减少冗余数据，Abebe 等[13]提出根据工作负载访问模式进行数据移动的策略等。此外，由于里德-所罗门码相比于其他类型的纠删码需要更多的计算开销，Liu 等[14]提出在 GPU 上对矩阵运算进行并行化加速，Fan 等[15]提出了可以同时计算原始数据和冗余编码，Zhou 等[16]还提出了一种可以综合提升编码速度的方案等。通过这些方法，可以一定程度上提高里德-所罗门码纠删码的编码效率。

由于采用纠删码编码的文件系统数据是按块而非按字节进行存储，因此在数据修改或附加时，为了更新修改部分的数据，我们需要将包含原始数据的整个块进行纠删码的解码，然后使用修改数据进行修改后，再对新数据重新进行编码。这个过程相比于直接将新数据覆盖写入到存储介质上复杂的多。为了减少这个过程所引起的性能损失，我们需要改进基于纠删码编码的分布式文件系统中的元数据存储结构，使其更加适合按块存储数据的修改。元数据指的是数据属性的信息，如文件存储位置、大小、存储时间、访问授权等，从而使文件系统可以查找、记录并更新文件数据。针对分布式文件系统中文件数量巨大，元数据需要较高的吞吐量特点，LocoFS 提出了一个松散耦合的元数据服务[17]，Chen 等[18]提出了将小文件合并为块进行索引，这些研究均采用了新型的元数据结构代替原有文件系统中的固定字段的元数据结构，从而减少基于纠删码编码的分布式文件系统中的性能开销。

1.3　主要研究内容与贡献

本文在调研并了解当前已有的文件分布式文件系统和纠删码最新研究成果的基础上，研究了基于纠删码和对象存储的分布式存储系统，旨在进一步提高分布式存储系统的存储效率。当前基于纠删码的分布式存储系统在通用的对象存储上取得了不错的效果，然而在数据冗余度和对数据块的修改优化等方面还存在着一定的提升空间。针对大数据时代不断增长的存储需求，降低数据冗余可以减少数据备份所需要的存储介质数量，大幅降低数据的存储成本。针对数据库、实时流计算等对数据修改和附加要求更高的存储需求，优化数据修改算法能使更多类型的存储系统可以通过采用纠删码的方式来减少存储冗余。本文基于动态纠删码编码策略实现了一个统一支持对象存储和文件存储的分布式存储系统如图 1.2 所示。此系统分为基于动态纠删码编码策略的对象存储和基于对象存储的文件存储两大部分，下面将依次简要进行介绍。

本文首先实现了一个基于动态纠删码编码策略的对象存储系统。纠删码编码技术不同于多副本备份技术，它无需重复存储多份相同的原始数据，而

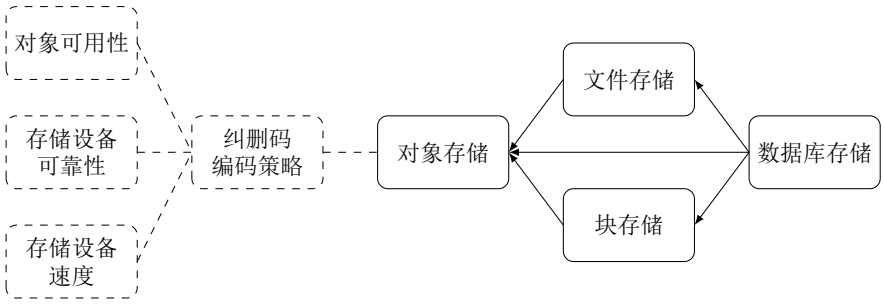


图 1.2　基于纠删码对象存储的分布式文件系统设计

是将原始数据分为 n 个原始数据单元，并通过纠删码编码计算，增加额外的 m 个冗余数据单元，通过这 $n+m$ 个原始数据单元和冗余数据单元中的任意至少 n 个数据单元，均可通过计算还原出原始数据。即基于 $n+m$ 个数据单元的纠删码的编码方案，最多可以容忍 m 个数据单元失效。然而在数据中心中，由于存储的数据量是不断变化的，因此分布式文件系统中的节点会动态地进行增加或者替换，这些增加或替换的存储介质的型号和批次往往不尽相同。因此，在分布式文件系统中，每一个节点之间的速度和可靠性也是有差异的。面对非一致速度和可靠性的节点，我们若采用传统的固定原始数据单元和冗余数据单元数量的方式计算纠删码编码，则会忽略节点之间的这些差异信息，从而难以得到最优的存储性能。同时，不同类型的文件根据其重要性的不同，也有着不同的可用性需求。因此，如果能充分利用待存储对象的可用性以及存储设备的可靠性等信息，来动态地确定纠删码的编码方案，则可以提高系统中存储设备的利用率，进而提升系统的整体性能，并降低存储成本。本文提出了 On-demand ARECS 算法（On-demand Availability and Reliability Oriented Adaptive Erasure Coded Storage System），其综合存储设备的类型、批次和使用状况等信息，对存储设备的可靠性进行预测，进而对存储于设备上的数据可靠性进行推断，并依据对象数据自身的可用性需求，动态地确定纠删码存储策略，并根据各个节点的实时状态计算得出满足可用性需求的最快节点集合，从而选取当前状况下存储速度最快的存储节点，解决了慢速节点的瓶颈问题，从而兼顾了存储效率和存储可靠性，实现了基于动态非一致存储设备可靠性和数据可用性的自适应纠删码对象存储系统。

在上述自适应纠删码对象存储系统的基础上，针对数据可能频繁地进行修改和附加的特点，我们研究了基于纠删码编码的数据修改算法。该算法将文件修改和附加操作的函数调用抽象为修改事件对象，并将修改事件对象单独存储于对象存储系统中。通过定义修改事件对象，我们将文件的随机修改和附加转化为对象的新增操作，从而避免了纠删码的解码与重新编码，大幅降低了基于纠删码的分布式存储系统的额外开销，一定程度上提升了基于纠删码实现的存

储引擎的文件存储性能。对于并发执行的文件修改和附加操作，我们采用了一个无锁的修改事件队列，从而避免了并发修改和附加时文件锁的开销，从而支持高并发的分布式文件系统操作。针对频繁修改的对象，本文提出在系统空闲时间进行修改事件整合的方法，从而解决过多的修改事件带来的读取性能损失。

本文相较于已有的分布式文件系统，针对非一致的对象可用性和存储设备可靠性灵活地定义纠删码策略，并针对存储设备的实时速度和可用性动态进行存储池规划和存储节点选择，从而高效地利用存储节点，降低数据冗余。针对基于纠删码的分布式文件系统的数据修改问题，本文提出了将文件修改和附加操作抽象为修改事件对象进行存储的算法，实现了无锁的高效并发写入，扩展了基于纠删码的分布式文件系统的应用场景。

1.4　本文组织结构

本文的主要内容分为以下六个章节：

第一章为绪论。本章首先介绍了本文的研究背景与研究意义，面对不断增长的存储需求，分布式存储系统渐渐替代了传统的集中式存储系统。为了兼顾数据的安全性和存储效率，在分布式文件系统中，常采用纠删码进行数据的冗余备份。在此基础上，简要概括了本文的主要研究内容与贡献，提出了一种针对非一致的存储设备可靠性和数据可用性进行纠删码编码的策略，从而提高基于纠删码的分布式存储系统的存储效率，并介绍了一种将文件修改和附加操作抽象为修改事件的方案，从而解决基于纠删码的存储系统在数据修改时需要解码和重新编码的效率问题。最后，介绍了本文的组织结构。

第二章为相关工作。本章首先介绍了目前存储系统中数据可能发生丢失的几种可能原因，并分别介绍了目前主流存储系统中的数据备份与恢复算法，其中存储利用率最高的备份与恢复算法便是基于纠删码的存储算法。之后，梳理了当前存储系统中使用的三类主流纠删码编码方案：阵列码、低密度奇偶校验码和里德-所罗门码。里德-所罗门码是唯一的可以使用任意原始数据单元个数和冗余数据单元个数的 MDS 编码（即存储效率达到理论上的最优），但是其存在着一定的计算开销。最后，总结了目前主流分布式文件系统中基于里德-所罗门码纠删码编码引擎的实现方案和数据修改算法。针对基于里德-所罗门码纠删码的计算效率问题，介绍了目前几种分布式存储系统中里德-所罗门码计算优化的最新研究成果。针对基于里德-所罗门码纠删码编码的存储系统在数据修改时解码和重新编码的性能开销问题，总结了目前分布式文件系统中针对文件存储结构的最新研究成果。

第三章为基于非一致可靠性存储的数据可用性评估策略。本章介绍了 On-demand ARECS 算法中存储设备可靠性和数据可用性的推断部分。针对不同重要性的数据，我们需要选取适当的数据可用性来达到更高的成本收益率。针对实际分布式存储系统中存储设备类型、批次和使用状况等特征，提出了一种对存储设备的可靠性进行动态估计的方法，并基于此存储设备的可靠性对其上存储的数据可用性进行估计，从而保证满足设定的数据可用性。针对实际存储系统中损坏节点的会被定期更换和重建的特点，我们在模型中引入了数据重建时间，从而保证存储系统在动态条件下能够满足数据所需的可用性。最后，本章对 On-demand ARECS 算法的磁盘故障率模型和成本收益模型进行了模拟实验验证。

第四章为面向动态数据可用性的纠删码对象存储策略。本章介绍了 On-demand ARECS 算法中的存储节点选择部分。根据数据本身的可用性需求以及基于存储设备推断的数据可用性保障，通过分析节点实时性能数据，最小化存储时所需要的读写时间，从而选取最优的存储设备组合，实现自适应的对象存储系统。针对过大的对象若直接进行纠删码计算时编码矩阵过大，导致计算开销过高的问题，我们通过数据分块的方式进行纠删码的计算，并假设各数据块大小均等，从而计算模型中的数据读写时间，保证存储系统高效地进行大文件的读写。最后，本章在开源的 Tahoe-LAFS 分布式文件系统上实现了基于 On-demand ARECS 算法的动态纠删码对象存储系统，对 On-demand ARECS 算法的存储效率和存储速度进行了实验验证。

第五章为基于纠删码对象存储的分布式文件存储。本章首先介绍了基于 On-demand ARECS 算法进行纠删码编码的对象存储系统的设计和物理存储组织方式，并通过 FUSE 技术设计了一种基于自适应对象存储的分布式文件系统中间件。其次，针对基于纠删码编码的对象存储在数据修改时需要解码并重新编码，导致数据修改开销过大的特点，我们将文件的修改和附加操作抽象为修改事件，存储于只读的对象存储系统中。为了保证并发状态下数据修改的效率，我们应用了基于数组的无锁队列来处理修改事件队列。针对数据多次修改后可能导致的读取速度减慢问题，我们提出了一种在系统空闲时间对文件修改对象进行合并的策略。最后，我们在基于 Tahoe-LAFS 的分布式文件系统上对此算法的文件修改和附加操作性能进行了测试。

第六章为结论与展望。本章首先回顾了全文的工作，介绍了基于 On-demand ARECS 算法和文件修改事件对象算法的分布式文件系统对存储速度进行的优化。之后，总结了目前基于纠删码的分布式文件系统研究中的热点和挑战，并针对数据损坏时的数据重建、纠删码编码和解码的开销和磁盘故障率的有效估计等问题展望了未来可能的研究方向。

第 2 章　相 关 工 作

2.1　引言

由于目前的分布式存储系统为使用数量非常庞大的存储节点所构成的大规模集群，因此其中的存储介质、服务器和网络等出现故障已经成为常态。根据 Rashmi 等在 Facebook 的数据中心中的研究[19]，其存储系统平均每天会发生 50 起节点不可用事件。因此分布式存储系统的数据备份与恢复是存储系统研究中的重要问题之一。本章首先对存储系统中数据损坏和恢复的方式进行介绍，总结了数据损坏的可能原因以及存储系统的几种数据备份和恢复方式。其次，针对传统数据备份与恢复需要的成本太高或数据冗余度过大等不足之处，介绍了纠删码的编码和解码算法，以及分布式存储系统中基于纠删码的数据备份与恢复算法。最后，针对在基于纠删码的分布式存储系统中数据的修改需要解码并重新编码，造成额外的计算和存储开销的特点，介绍了在基于纠删码的分布式存储系统中文件存储算法的优化。

2.2　存储系统中的数据损坏与恢复

2.2.1　存储系统中数据损坏的原因

在分布式存储系统中，使用了大批量的小型主机取代了集中式存储系统的大型主机，从而减少制造和维护大型主机的成本。然而，小型主机的可靠性和性能一般较大型主机存在一定的差距，也就是说分布式存储系统中节点发生故障的情况是难以避免的，以下是存储系统发生故障的几种常见原因。

1. 存储介质的物理损坏

对于软盘、机械硬盘或磁带等存储系统来说，其利用介质的磁性改变存储数据信息。在介质磁性被破坏，或者介质被划伤等等情况发生时，介质中存储数据的单元比如磁道或扇区等，会发生不可逆转的损坏，也就是我们常说的硬盘坏道。部分存储设备会将介质中的坏道进行屏蔽，但是损坏的数据是无法通过正常的数据读取过程读出的。

对于固态硬盘等采用闪存芯片的存储系统来说，其利用浮栅等结构存储电荷，用电荷的存储状态保存数据信息。在介质被多次擦写后，浮栅与沟道之间的氧化层被磨损地越来越严重，最终导致浮栅结构无法可靠地存储电荷，发生数据错误。

2. 存储设备发生的部件物理故障

对于硬盘等存储设备来说，除了用于存储数据的磁盘盘片或闪存芯片以外，硬盘内部还会有磁头、电机和电路板等部件，用于从盘片中读写信息，并通过 SATA 或 NVMe 等通信协议与主机通信。若硬盘中的磁头出现偏位或断开，或电路板中的电容发生击穿等，均会导致硬盘出现错误。

3. 存储设备发生的固件逻辑错误

对于硬盘等存储设备来说，为了屏蔽坏道以尽可能降低坏道造成的影响，或者对固态硬盘进行磨损平衡以延长使用寿命等目的，硬盘内部一般都会运行着数十万行代码构建的固件[20]，用以读取存储介质中的数据，并进行映射，然后再将数据传输给主机。如果由于固件 Bug 或者异常断电等原因造成固件逻辑发生错误，则有可能引起数据的丢失。

4. 文件系统发生的软件错误

除了存储设备的硬件故障导致数据损毁意外，软件故障也可能导致数据出错或丢失。目前计算机系统中的绝大部分数据均存储于 NTFS、APFS、XFS 或 ext4 等各种类型的文件系统中，由于异常断电、文件系统实现错误等等原因，文件系统也可能出现错误，导致数据发生损坏。

5. 存储系统使用者的数据误操作

除了存储系统故障导致的难以预测的数据损坏以外，上层应用或者用户的误操作比如误删除、误格式化或者错误分区等原因，可能导致数据被其使用者自身无意破坏，这也成为了数据丢失的重要原因之一。

2.2.2　存储系统中数据恢复的方法

综上所述，在存储系统中由于存储系统中的硬件故障、软件错误或人为失误等原因，数据损坏是难以避免的，因此在数据损坏前如何进行有效地备份，以及数据发生损坏后如何高效地进行数据恢复，便成为了存储系统研究的重要问题之一。以下我们将分别介绍目前常用的几种数据备份与恢复方法。

1. 存储设备的物理修复

针对在数据丢失前未进行有效备份，且数据由于存储设备本身发生故障导致丢失的情况，数据只能通过修复物理设备的方式进行恢复。当硬盘内部的部件发生物理损毁时，或者硬盘内固件发生逻辑错误时，虽然数据能完整地存储于存储介质中，但由于硬盘数据读取所需要的必要部件已经不能正常工作，此时一般只能寻求硬盘生产厂家或专业数据恢复公司的帮助，使用其提供的专用设备，通过开盘直接读取介质内的数据信息，或者通过特殊接口修复固件错误，从而修复损坏设备中的数据。这种方法由于需要较为专业的设备，因此数据恢复的成本较高，而且对于存储介质本身损坏的情形，这种方法不一定能完

全有效地恢复数据。

2. 独立硬盘冗余阵列

独立硬盘冗余阵列（Redundant Array of Independent Disks，即 RAID）是利用存储虚拟化的思想，将多个物理硬盘组合起来成为虚拟硬盘阵列组，其中除 RAID 0 外，均有不同数量的冗余备份，以保证存储系统中部分存储设备本身发生故障损坏时数据的安全性。RAID 是常用于集中式存储系统中的数据存储方案，操作系统一般将一个 RAID 阵列作为单一的逻辑硬盘操作。

（1） RAID 0

RAID 0 将两个或以上的存储设备组合起来，共同构成一个大容量的逻辑存储设备。对于 RAID 0 中存储的数据，RAID 控制器将数据分散切分为条块，直接存储于物理磁盘中。这样的结构在数据读写时可以同时在所有硬盘进行并行操作，因此 RAID 0 的读写速度是所有 RAID 级别中最高的。但是 RAID 0 没有存储任何冗余数据，因此不具备检错和纠错的功能。在 RAID 0 的阵列中由于所有数据均分散在所有磁盘上，因此只要其中一个存储设备损坏，整个阵列的所有数据都会丢失，数据安全性极差。

（2） RAID 1

RAID 1 将两个或以上的硬盘相互作为镜像备份，存储相同的原始数据，当一块硬盘发生损坏，仍可以使用另一块硬盘继续工作。这样的结构在读取数据时也可以同时并发地读取所有硬盘，因此读取性能和 RAID 0 相同，但数据写入时需要同时写入所有硬盘，较单个原始硬盘的写入速度有一定下降。RAID 1 级别的硬盘阵列只要其中一个硬盘正常，便可以正常进行读写，数据安全性最高。但由于 RAID 1 的硬盘之间均互为冗余镜像，因此 RAID 1 的容量为所有单硬盘容量的最小值，硬盘利用率是所有 RAID 级别中最低的。

（3） RAID 2

RAID 2 使用汉明码（Hamming Code）的方式，将原始数据额外编码，生成一段错误修正码（Error Correction Code，即 ECC），然后通过存储一份原始数据和其错误修正码，实现数据的冗余备份。因此，若要采用 RAID 2，至少需要三个或以上的硬盘设备。RAID 2 在目前的存储系统中较为罕见。

（4） RAID 3

RAID 3 类似 RAID 2 采用校验码来作为冗余备份数据，但 RAID 3 采用较为简单的异或运算进行校验码的计算，并将数据按比特分割存储于数据盘，校验码单独存放于一块硬盘上。RAID 3 在目前的存储系统中较为罕见。

（5） RAID 4

RAID 4 采用块交织技术，其采用异或方式生成奇偶校验信息，但在数据分割时按块分割而不是按比特分割，从而保证块的完整性。RAID 4 在目前的存储

系统中较为罕见。

（6） RAID 5

RAID 5 同样采用异或方式生成奇偶校验信息，并作为数据的冗余备份进行存储。其将奇偶校验信息和相对应的数据分别存储于不同的磁盘上，而非单独使用一块硬盘存储校验信息。RAID 5 至少需要三个硬盘，若其中任意一块硬盘发生损坏，均可以利用剩余其他硬盘中的数据和校验信息来恢复重建整个阵列。因此，我们可以认为 RAID 5 是在 RAID 0 的高硬盘利用率和 RAID 1 的高安全性之间权衡的解决方案。

（7） RAID 6

RAID 6 为了解决 RAID 5 中两块硬盘同时损坏时无法恢复数据的隐患，相比 RAID 5 增加了第二个独立的奇偶校验信息块。两种奇偶校验块通过采用不同的算法[21–24]，保证任意两块磁盘的同时失效均不会影响数据完整性，大大增加了硬盘阵列的可靠性。由于 RAID 6 增加了额外的奇偶校验，因此需要更大的空间存储校验信息，以及更多的计算资源实现校验码的计算，因此其相对于 RAID 5 来说有更大的 I/O 操作量和计算量，故通常采用专用的硬件 RAID 控制器实现相应计算。

（8） 混合 RAID

通过将不同算法的 RAID 组合使用，可以为存储系统提供更灵活的选择。

① RAID 01

RAID 01 它将所有的硬盘分为两个 RAID 0 阵列，然后令两个 RAID 0 阵列互为镜像，即 RAID 1。由于这种方式只要有一个硬盘损坏，同组 RAID 0 的其他硬盘也无法正常读写数据，只剩下其他组的硬盘可以工作，因此可靠性较低，在目前的存储系统中较为少用。

② RAID 10

RAID 10 和 RAID 01 的程序相反，其将所有硬盘分为若干个 RAID 1 阵列，然后将这些 RAID 1 阵列作为 RAID 0 的基本单元进行数据分割存储。这种方式单个硬盘损坏对于其他硬盘没有影响，所有硬盘均可以继续读写数据，阵列可靠性较高。

3. 文件系统的软件恢复

针对在数据丢失前未进行有效备份，但数据只是由于文件系统损坏或者使用者误操作引起的逻辑丢失，可以尝试通过数据恢复软件进行数据恢复。对于许多情况下的文件系统损坏和误操作，只是文件系统存储元数据的数据区发生了错误或者被误删除，而真正的用户数据很可能仍然存储于存储设备之上，我们可以通过全盘扫描等方法遍历整个存储设备，尝试越过文件系统的元数据服务，直接读取实际的数据块。这种方法要求数据块本身可以正常读取，也就是

说存储设备本身未发生故障，且数据块未被新数据覆盖才能进行数据恢复。

4. 数据的冗余备份与恢复

由于存储设备物理修复和文件系统的软件修复均有较为严格的限制条件，且成功率无法保证，因此为了防止数据由于节点故障引起丢失，在分布式存储系统中，一般均会在存储系统的软件层面采用不同类型的冗余备份策略。冗余备份即将部分数据或校验码等冗余存储于多个存储节点，从而保障数据的安全性。

（1） 数据的冗余备份机制

通常的数据冗余备份机制分为两类，即多副本方法和纠删码方法[25]。

多副本方法直接将原始数据拷贝多份，每份存储于不同的存储节点的方案，也就是多副本备份方案，便是其中最为简单也最具代表性的。在目前主流分布式存储系统，如 Ceph 分布式文件系统中，一般副本个数均选取为三份，即将三份相同的原始数据分别存储于不同的节点中。显然，三副本方案中冗余备份额外占用的存储空间为原始数据的两倍，即数据冗余度为 200%。这种方法不需要专门的编码和重建算法，因此性能较好，但存储利用率极低。

纠删码方法最早应用于通信传输领域中信息传输过程的检错和纠错问题，后来逐渐扩展到存储系统中，以解决数据存储中的冗余备份问题。纠删码方法相对于多副本方法极大的降低了数据的冗余度，提高了存储资源的利用率。

（2） 本地备份与异地备份

对于冗余备份的数据，若将全部数据均保存于相同的本地机房，假如数据所在机房由于自然灾害或设备损毁整体发生不可用的情况，那么备份数据将同时失效。因此在重要的存储系统中，一般均会采用异地备份进行容灾。异地备份可以选择多副本方法或纠删码方法构造冗余数据，然后将冗余数据存储于不同机房甚至不同城市的其他存储集群中，来保障数据的安全。

（3） 在线备份与离线备份

对于重要的数据来说，除了需要避免存储系统本身的故障导致的数据丢失以外，还需要保证在系统被攻击、应用程序出错或者使用者误操作等情况发生时的数据安全性。由于在线备份的数据无法保障上述情况发生时的数据安全，因此常用离线备份的策略，将存储系统中某个时间点的数据全部拷贝出来，生成一份数据镜像存储到外部的存储设备中，并将该备份存储设备脱机保存，以保证即使所有在线备份的数据都被破坏时，也可以从离线备份中进行数据恢复。由于在线备份无法实时进行，因此使用离线备份可能会导致部分最新产生的数据丢失，故从离线备份的副本中恢复一般被认为是存储系统数据安全的最后一道防线，仅在其他方法均无法有效地恢复数据时才会采用。

2.3　存储系统中的纠删码编码技术

2.3.1　阵列码

阵列码（Array Codes）是一种基于异或（XOR）运算的纠删码编码方式，分为横式阵列码和纵式阵列码两种。这类编码由于计算简便[26]，被广泛应用于 RAID 6 中[21–24]。

1. 横式阵列码

横式阵列码（Horizontal Parity Array Codes）是指冗余编码块和数据编码块相互独立，分别单独进行存储的阵列码编码方式。由于其冗余编码的存储是独立的，因而具有良好的可扩展性。典型的横式阵列码的容错率都不是很大，其中容错率为 2 的编码包括 EVENODD 编码[27]和 RDP 编码[28]等，容错率大于 2 的编码较少，主要包括 STAR 编码[29]等。

图 2.1 和图 2.2 分别给出了典型的两种横式阵列码 EVENODD 编码和 RDP 编码的示意图[30]，其中 EVENODD 编码是最早出现的阵列码编码方式。这些编码体系均要求数据块的个数必须与硬盘数量相匹配，且不能实现对任意数量硬盘的编码。在存储空间的利用率方面，由于这些编码的存储效率均达到了理论上的最优，因此均为 MDS 编码（Maximum Distance Separable Codes）。

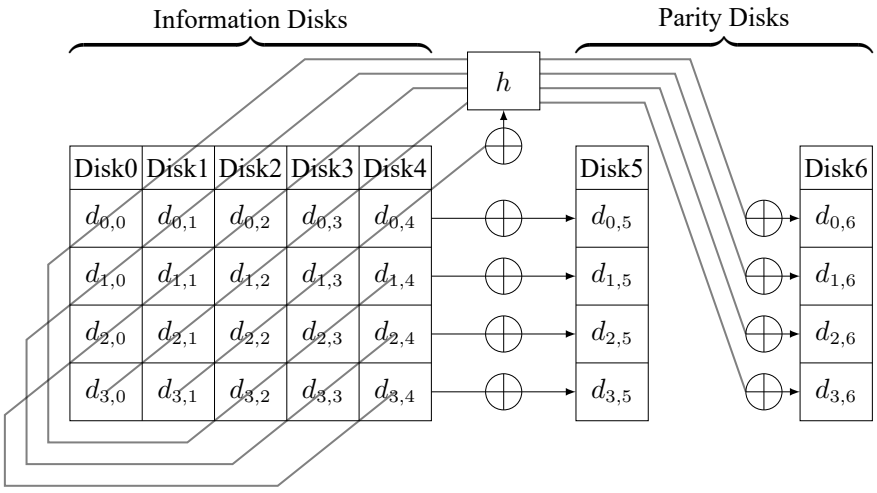


图 2.1　EVENODD 编码示意

2. 纵式阵列码

纵式阵列码（Vertical Parity Array Codes）与横式阵列码不同，其并不将冗余编码块与数据编码块单独存储于不同的硬盘上，而是直接将冗余数据存储于原始数据块的内部。在纵式阵列码的数据块中，既存储了原始数据，又存储了冗余编码。由于冗余数据分散存储与各个硬盘，因此不存在横式阵列码中冗余数据盘在数据写入时的瓶颈问题。但纵式阵列码中的硬盘之间依赖性强，可扩

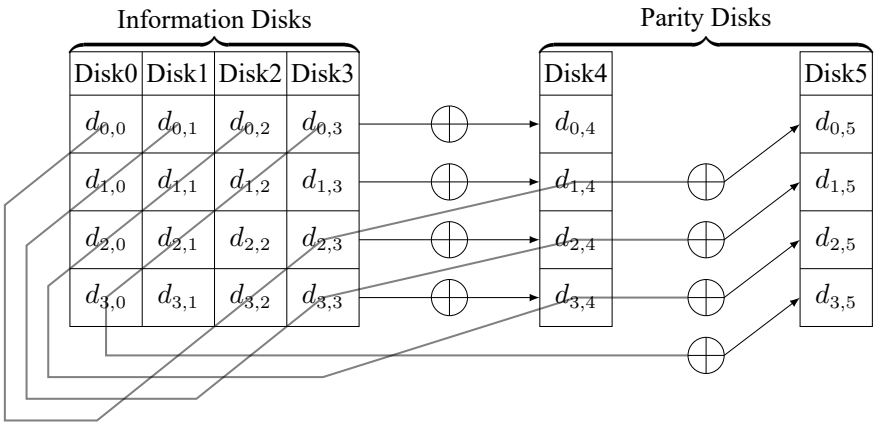
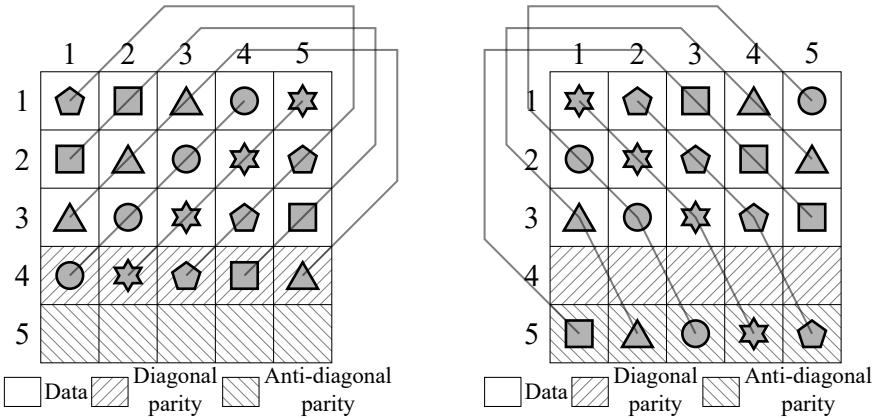


图 2.2　RDP 编码示意

展性较差。目前的纵式阵列码主要包括 X-code 编码[31]和 WEAVER 编码[32]等。

图 2.3 给出了 X-code 编码示意图[33]。在由 n 块磁盘组成的 X-code 的编码中，每块磁盘中包含 $n-2$ 个数据块和 2 个冗余编码块，其中要求 n 必须是一个大于 2 的素数。X-code 编码具有理论上最优的运算效率，且在存储利用率上 X-code 也是 MDS 的编码，但由于其中 n 必须为素数的限制导致其可扩展性很差。



(a)　The Diagonal Parity　　(b)　The Anti-diagonal Parity

图 2.3　X-code 编码示意

2.3.2　低密度奇偶检查码

低密度奇偶检查码（Low-Density Parity-Check Codes，即 LDPC 码）[34]同样是一类基于异或（XOR）运算的编码，但它的存储效率不属于 MDS 编码，常用于信道编码中[35]。利用 LDPC 码的校验矩阵 $\boldsymbol{H}$ 可以生成其编码矩阵 $\boldsymbol{G}$，进而得到 LDPC 编码，因此 LDPC 码的特点与性能主要通过校验矩阵 $\boldsymbol{H}$ 来体

现。也可以说，LDPC 码是用一个稀疏的校验矩阵 $\boldsymbol{H}$ 定义的线性分组码，构造 LDPC 码的核心就是构造 $\boldsymbol{H}$ 矩阵。常见的 $\boldsymbol{H}$ 矩阵构造方法有 Gallager 提出的方法[34]和 Mackay 提出的方法[36]等。

定义 $\boldsymbol{H}$ 矩阵的行重为矩阵每行中 1 的个数，列重为矩阵每列中 1 的个数。在规则 LDPC 码的校验矩阵中，各行的行重和各列的列重均是一致的。因此可以定义一个 (n,j,k) 的规则 LDPC 码的校验矩阵 $\boldsymbol{H}$ 有 n 列，m 行，列重为 j，行重为 k，其中 $m=n{\cdot}j/k$，$j<k$，$j{\ll}m$，$k{\ll}n$。我们以一个 $(6,2,4)$ 的规则 LDPC 码的校验矩阵为例，其行重为 4，列重为 2：

$$\boldsymbol{H}=\begin{bmatrix}1&1&0&1&1&0\\1&0&1&0&1&1\\0&1&1&1&0&1\end{bmatrix} \tag{2.1}$$

此 LDPC 码的校验矩阵的行对应着校验方程（校验节点），列对应着传输的比特（比特节点），它们之间的关系可以用 Tanner 图[37]来表示。上述 $\boldsymbol{H}$ 矩阵对应的 Tanner 图如图 2.4 所示，图的左边有 n 个节点，右边有 m 个节点。LDPC 码的校验矩阵可以通过矩阵的初等变换，把 $\boldsymbol{H}$ 矩阵变化成系统形式 $\boldsymbol{H'}=[\boldsymbol{P}^T,\boldsymbol{I}]$，得到该 $\boldsymbol{H}$ 矩阵对应的生成矩阵 $\boldsymbol{G}=[\boldsymbol{I},\boldsymbol{P}]$，用数据比特去乘编码矩阵 $\boldsymbol{G}$，就得到了编码结果。此算法的复杂度为 $O(n^2)$。

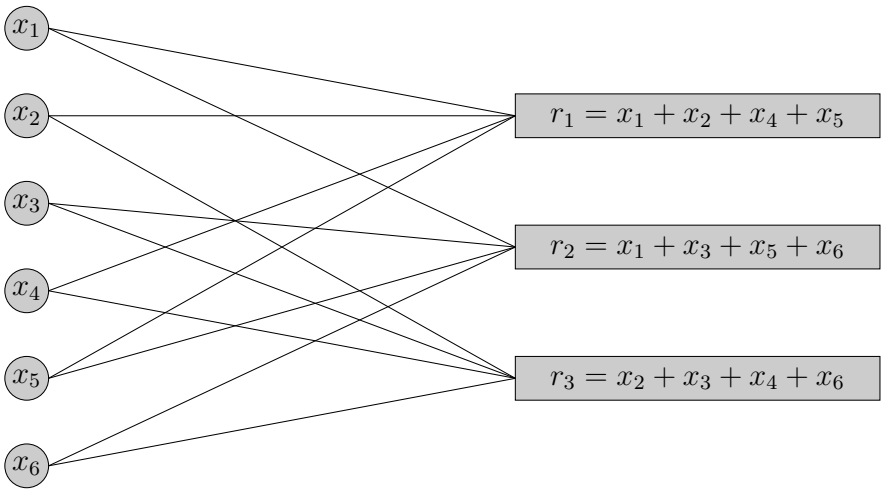


图 2.4　LDPC 编码的 Tanner 图示意

2.3.3　里德-所罗门码

里德-所罗门码（Reed-Solomon Codes）[38]是在存储系统中较为常用的一种纠删码算法，也是唯一一种可以使用任意数据块个数 n 和任意冗余块个数 m 进行编码的 MDS 编码方法。里德-所罗门码是一种基于有限域的编码算法，给定 n 个数据块（Data block）D_1、D_2、$\cdots$、D_n，和一个正整数 m，里德-所罗门码

根据 n 个数据块生成 m 个编码块（Code block）C_1、C_2、$\cdots$、C_m。在编码生成的全部 n 个数据块和 m 个编码块中，选取任意 n 个块均能解码出原始数据，即里德-所罗门码最多容忍任意 m 个数据块或者编码块同时丢失。上述里德-所罗门码中的变量 D_i、C_i 代表字长 w 为 8 位或者 16 位的字。较大的数据块需要先拆分为字，然后将字作为编码和解码的单位，对字进行编码和解码。

将原始数据记为向量 $\boldsymbol{D} = (D_1, D_2, \cdots, D_n)^T$，其经过里德-所罗门码编码生成的校验码为向量 $\boldsymbol{C} = (D_1, D_2, \cdots, D_n, C_1, C_2, \cdots, C_m)^T$，则里德-所罗门码的编码过程可视为矩阵运算 $\boldsymbol{G} \times \boldsymbol{D} = \boldsymbol{C}$，其中 $\boldsymbol{G}$ 是编码矩阵（Distribution Matrix，也可称为生成矩阵、分布矩阵）。由于解码是编码的逆运算，因此要求编码矩阵的任意 $n \times n$ 子方阵均为可逆矩阵。在里德-所罗门码的编码矩阵中，上部矩阵一般是 $n \times n$ 的单位阵，表示直接存储原始数据，下部矩阵是 $m \times n$ 的校验码矩阵。我们常选择范德蒙矩阵[38]或柯西矩阵[39]作为里德-所罗门码编码矩阵的下部矩阵。

范德蒙矩阵的各列间为等比级数关系，满足任意阶子方阵可逆的要求。一个 $m \times n$ 的范德蒙矩阵定义如下：

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ {a_1}^1 & {a_2}^1 & {a_3}^1 & \cdots & {a_n}^1 \\ {a_1}^2 & {a_2}^2 & {a_3}^2 & \cdots & {a_n}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ {a_1}^{m-1} & {a_2}^{m-1} & {a_3}^{m-1} & \cdots & {a_n}^{m-1} \end{bmatrix} \tag{2.2}$$

其中 a_i 互不相同，且均不为 0。

柯西矩阵的任意阶子方阵均为奇异矩阵，因此其任意阶子方阵均存在逆矩阵。而且由于柯西矩阵在迦罗华域上求逆矩阵的计算复杂度为 $O(n^2)$，相比于范德蒙矩阵求逆的计算复杂度 $O(n^3)$ 更低，运算速度更快。另外，在使用柯西矩阵进行矩阵计算时，可以通过将 $GF(2^w)$ 域的元素转换成二进制矩阵，从而将乘法转换为位运算，进而降低编码运算的复杂度。柯西矩阵的定义为：

$$\begin{bmatrix} \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \frac{1}{x_1+y_3} & \cdots & \frac{1}{x_1+y_n} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \frac{1}{x_2+y_3} & \cdots & \frac{1}{x_2+y_n} \\ \frac{1}{x_3+y_1} & \frac{1}{x_3+y_2} & \frac{1}{x_3+y_3} & \cdots & \frac{1}{x_3+y_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_m+y_1} & \frac{1}{x_m+y_2} & \frac{1}{x_m+y_3} & \cdots & \frac{1}{x_m+y_n} \end{bmatrix} \tag{2.3}$$

其中 x_i 和 y_i 都是有限域 $GF(2^w)$ 中的元素。

里德-所罗门码在任意不超过 m 个数据块丢失时均可进行数据恢复。如图 2.5 所示，假设其中的 D_2、D_5 发生损坏，我们可以从编码矩阵 $\boldsymbol{G}$ 中删掉

D_2、D_5 对应的行，组成新矩阵 $\boldsymbol{G_1}$。根据编码矩阵任意阶子方阵可逆可以证明 $\boldsymbol{G_1}$ 是可逆的，即存在逆矩阵 $(\boldsymbol{G_1})^{-1}$，满足 $\boldsymbol{G_1}\times(\boldsymbol{G_1})^{-1}=\boldsymbol{I}$，其中 $\boldsymbol{I}$ 为单位矩阵。将剩余未损坏的数据左乘 $(\boldsymbol{G_1})^{-1}$ 矩阵，便可以恢复出原始数据 $\boldsymbol{D}$。

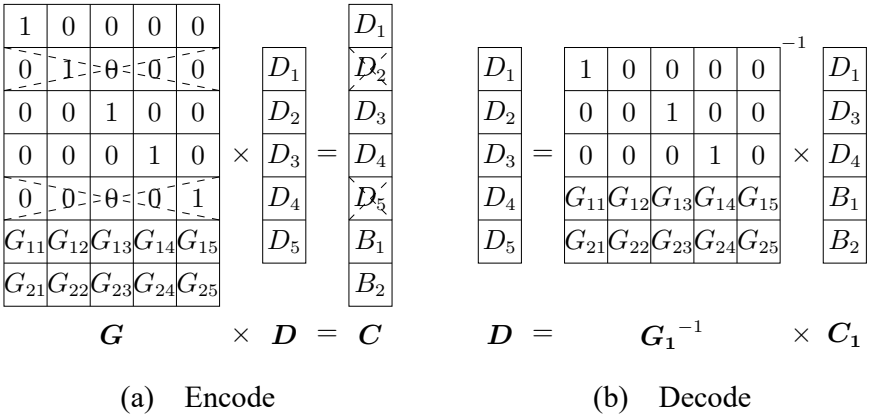


图 2.5　纠删码编码和解码示意

2.4　基于纠删码的分布式存储系统

2.4.1　常见分布式存储系统的纠删码实现

由于在分布式存储系统中应用多副本备份时数据冗余度较高，因此为了降低数据的冗余度，在保证数据安全的前提下降低存储成本，因此可以采用纠删码进行数据的存储。由于在常见的纠删码编码方案中，只有里德-所罗门码是可以基于任意数据块个数和校验块个数进行编码的纠删码方案，相比其他纠删码方案更加灵活，因此，在分布式存储系统中，一般采用里德-所罗门码进行数据的纠删码编码。

1. 里德-所罗门码在常见分布式存储系统中的应用

目前主流分布式存储系统均使用里德-所罗门码纠删码编码方案进行冗余备份。以 $n=4$，$m=2$ 的里德-所罗门码纠删码方案为例，其可以容忍最多两个数据块的丢失，因此具有与三副本备份方案相近的容错能力，但纠删码方案的数据冗余度仅为 50%，大大低于多副本备份的数据冗余度。但当使用纠删码编码的存储系统发生数据损坏时，存储系统需要通过至少 n 个块计算重建数据，相比多副本备份的直接拷贝需要额外的计算资源开销，也占用了更多的磁盘带宽和网络带宽。根据 Arafa 等人[40]的研究，在主流的分布式文件系统 HDFS 和 Ceph 中，使用纠删码引擎进行存储均有明显的性能损失。因此若要在存储系统中广泛应用纠删码技术，需要研究更加高效的方案，比如提高系统并行度等，以解决纠删码的性能瓶颈问题。

2. Ceph 分布式文件系统的纠删码存储方案

在 Ceph 分布式文件系统中，其纠删码引擎应用 Jerasure 函数库实现，当管理员创建基于纠删码的存储后端时，可以指定数据块数量 n 和编码块数量 m 作为系统的参数。在纠删码存储后端中，每个数据对象都被分成 n 个数据块和 m 个编码块，分别存放于 $n+m$ 个对象存储设备（Object Storage Device，即 OSD）中。因此在 Ceph 分布式文件系统中，对于所有对象均采用相同的数据块和编码块数量，没有动态地考虑对象的需求和存储设备的条件。

3. Ceph 分布式文件系统的纠删码数据的修改

在基于纠删码的数据修改方面，由于纠删码的编解码开销过大，导致 Ceph 难以简单地实现数据修改。Ceph 采用元数据服务器（Metadata Server，即 MDS）主要负责 CephFS 集群中文件和目录的管理，记录数据的属性，如文件存储位置、大小、存储时间等，同时负责文件查找、文件记录、存储位置记录、访问授权等。Ceph 的 Bluestore[41] OSD 在最新的 Luminous 版本增加了部分写入功能，可以将 Ceph 的文件数据存储在纠删码存储池中。然而由于覆盖写入需要对对象做出修改，增加了系统的复杂性，且由于需要进行解码和重新编码，写入效率不高。

4. Hadoop 分布式文件系统的纠删码数据的修改

在 HDFS 的早期版本中，为了与 MapReduce 配合，同样不支持文件附加或者文件修改。但在最新的 Hadoop 2.x 版本中，HDFS 新增支持了文件附加操作。

5. GFS 分布式文件系统的纠删码数据的修改

在 GFS 中，每个大文件被划分为若干个固定大小的块，其写操作主要分为写入和附加两种。其中写入操作是修改数据，附加操作则是在文件末尾添加数据。但 GFS 的前提假设是文件系统的写操作主要是对文件进行追加操作, 追加的数据是足够大且顺序追加的，在写入后文件就很少会被修改。虽然 GFS 支持小的随机写入，但并不是文件系统优化的目标。同时 GFS 的写入必须是原子操作，一次写入的数据必须是原子性的添加到文件结尾。若有两个独立的进程 A 和 B 同时附加数据到同一个文件里，由于需要寻找文件末尾的位置后才能添加数据，会产生互斥问题。

2.4.2　分布式文件系统中纠删码运算效率的研究

由于基于里德-所罗门码的纠删码编码和解码需要较高的额外资源开销，因此需要我们对编码运算和编码策略进行优化，从而实现高效的分布式文件系统。

在编码运算优化方面，最基础的便是对矩阵运算进行优化。由于柯西矩阵的求逆矩阵的计算复杂度更低，且在有限域 $GF(2^w)$ 上进行的运算更加容易优化，因此可以利用柯西矩阵代替范德蒙矩阵提高编码和解码效率。针对不同数据的纠删码编码可以并行计算的特点，Liu 等[14]提出了 G-CRS，其采用柯西矩阵进行纠删码的编码，并通过 GPU 加速基于柯西矩阵的里德-所罗门码纠删码的计算。通过在基于 Maxwell 和 Pascal 等架构的现代 GPU 上的实验结果显示，G-CRS 的数据吞吐量比目前大多数基于 CPU 的编码库快 10 倍。针对在纠删码解码过程中需要先通过解码矩阵计算原始数据，才能根据编码矩阵进行校验块的计算的问题，Fan 等[15]提出了一种新的计算方案，其可以同时计算原始数据块和冗余编码块。Zhou 等[16]还提出了一种结合改进编码矩阵、减少计算中相同的 XOR 操作、优化计算调度算法、高效管理 cache 和矢量化等技术的方法，来提高纠删码的编码效率。

此外，针对不同数据块和不同存储设备之间存在差异的特点，我们可以通过动态的纠删码编码策略来代替固定的纠删码编码策略，从而对系统进行优化。Abebe 等[13]提出了一种在纠删码分布式存储系统中，基于工作负载访问模式进行数据访问和数据移动的策略 EC-Store，并通过在两个基准测试上的评估，证明其可以显著地减少数据检索的开销。针对大规模集群存储系统通常为异构存储架构，常常是使用可靠性存在差异的不同存储设备混合组成的特点，Kadekodi 等[12]提出了 HeART，通过读取磁盘的属性信息，来估计磁盘的长期数据可靠性，并基于此尽可能地减少备份数据的存储。Xiang 等[11]提出了一种通过联合考虑存储系统延迟和开销来优化纠删码编码的方案。Plank 等[42]提出了 SD Codes 来区别对待整个磁盘故障和磁盘部分扇区故障，提高存储效率。Zhang 等[43]提出了 NADE 模型对节点数据读取进行优化。

2.4.3　分布式文件系统中的纠删码数据修改的研究

由于基于里德-所罗门码的纠删码的分布式文件系统在数据进行修改时需要重新编解码，因此一般均用于存储只读的对象数据。但是，在实际的数据存储需求中，对于数据的修改是很常见的。因此，如何在基于里德-所罗门码的纠删码分布式文件系统上实现一个快速的可读写的存储系统，是分布式文件系统领域研究的热点问题之一。

针对在对象存储上支持块存储，Gutierrez 等[44]提出了 uStorage 方案，作者认为块级存储广泛用于支持繁重的工作负载。它可以由操作系统直接访问，但它在分布式系统中面临一些持久性问题、硬件限制和性能下降。基于对象的存储设备（OSD）是一种广泛用于支持一次写入多次读取（WORM）系统的数据存储概念。由于 OSD 包含数据、元数据和唯一标识符，因此它变得非常强大且

可自定义。 OSD 是解决日益增长的数据增长和弹性需求问题，同时降低成本的理想选择。作者描述了一种可扩展的存储架构，该架构使用来自分布式 P2P 云存储系统的 OSD，并为用户提供块级存储层。该架构将商用硬件上 OSD 的备份、可靠性和可扩展性的优势与数据密集型工作负载的原始块的简单性相结合，展示了在后端使用 OSD 并基于原始块提供存储层的可能性，并为最终用户提供更好的性能，并基于缓存行为评估了所提出的架构，使用不同的高速缓存大小测量了两个存储层的繁重工作负载的高吞吐量性能。

在分布式文件系统的元数据存储方面，LocoFS 提出了一个分布式文件系统来提供松散耦合的元数据服务[17]，以弥合文件系统元数据和键值存储之间的性能差距。LocoFS 旨在通过两种技术来分离不同类型的元数据之间的依赖关系。首先，LocoFS 解耦目录内容和结构，它在平面空间中组织文件和目录索引节点，同时反向索引目录条目。其次，它将文件元数据分离，以进一步提高键值访问性能。评估显示，具有八个节点的 LocoFS 将元数据吞吐量提高了 5 倍，单个节点键值存储的吞吐量接近 93%，而最先进的 IndexFS 则为 18%。

随着互联网的发展，分布式文件系统中开始存储了大量的图片，且这些图片的大小通常不超过 1 MB。为了满足包含这么多图片的小文件存储系统的需求，Chen 等[18]提出了将小文件合并为块进行索引，并存储于纠删码后端的方案。实验表明，当系统容量为 1 PB 时，名称服务器中块信息的内存使用量小于 2 GB，显示了良好的可扩展性。同时，相比于两副本备份，使用纠删码降低了 25% 的存储开销，大大降低了存储成本。

2.5　小结

本章从数据常见的损坏方式出发，首先介绍了对应的几种数据恢复方法，如硬件恢复、多副本备份和纠删码备份等。针对于分布式存储系统中多副本备份冗余度高，存储资源浪费严重的特点，详细地介绍了常见的几种纠删码的定义和实现方式，并介绍了基于纠删码备份的分布式存储系统的优化方法。最后，针对分布式存储系统中纠删码实现在数据存储和恢复时需要解码并重新编码的特点，介绍了在基于纠删码实现的分布式存储系统中实现文件修改的方法以及文件存储结构相关的研究。

第 3 章　基于非一致可靠性存储的数据可用性评估策略

3.1　引言

在分布式存储系统中，由于数据的类型和用途等存在差异，因此重要性往往不尽相同。在对象存储系统中，我们可以定义对象持久性 p'_f，对于至关重要的数据，选取相对更高的对象持久性，对象数据发生损坏的概率更小，数据更加安全，但此时需要相应的冗余备份数增加，存储成本更高。相反，对于重要性一般的数据，选取适中的对象持久性，可以更加有效地利用存储资源，减少冗余数据。

同时，在存储系统中的数据可靠性通常受多种因素影响。首先，对于不同类型的存储设备，比如机械硬盘和固态硬盘，介质的存储机理不同，因此寿命通常是不同的。对于相同存储设备，不同批次间制造水平的差异以及制造过程中的误差也会导致存储介质寿命的差异性。对于同一介质，随着使用时间的不断增长，发生故障的可能性也会不断变化，存储于其中的数据的可靠性也会随之改变。在本章中，我们提出了一种基于存储设备实时状态对设备中数据可用性进行推断的算法，从而动态地估计数据损坏发生的可能性，为数据备份策略提供指导。

因此，我们提出了一种基于数据可用性和存储设备可靠性的动态要求，进行自适应纠删码编码的策略 On-demand ARECS。在纠删码存储系统中，对于从分布式存储系统中任意选出的 $n+m$ 块硬盘，将待存储的对象数据分为 n 份，同时增加 m 份纠删码校验数据，分别存储于这 $n+m$ 块磁盘上。我们需要根据每块磁盘 d_j 的型号和使用时间等要素推断出其当前发生故障的概率 p_{d_j}，从而计算得到在这 $n+m$ 块磁盘上存储的数据的正确率 $p_f(n,m)$。在此基础上，我们只需选择满足约束 $p_f(n,m) \geq p'_f$ 的纠删码存储方案，即可保障数据存储是安全且高效的。

此外，在实际的分布式数据中心中，通常均有专人负责损坏硬盘的更换和数据重建，因此硬盘更换和重建所需的时间也应是系统中一个重要的变量。我们假设硬盘更换和重建时间为 t_r 年，在 t_r 周期内计算对象的持久性需求 p'_f 和数据的正确率 $p_f(n,m)$ 间的关系，并对模型进行完善。

3.2　不同类型数据的可用性

3.2.1　数据重要性与数据可用性的关系

对于不同类型的数据，重要性通常并不相同[45]。对于重要性强、数据丢失后果严重的数据，如银行账户数据、业务合同数据、客户账号数据等，我们会通过增加备份数量、改进备份策略、异地备份和离线备份等多种方式尽可能地确保数据安全。而对于重要性一般、数据丢失影响较小的数据，如应用运行日志、下载的电影等，我们通过存储更少的备份数据，节约存储资源，实现对数据的分级处理。

3.2.2　对象存储系统中的数据可用性指标

在对象存储系统中，数据的可用性可以使用对象持久性指标来衡量，对象的持久性数值取决于数据的重要程度。在目前各大公有云的分布式存储服务中，其服务等级协议（Service Level Agreement，即 SLA）内均承诺了一年期对象持久性，具体数值如表 3.1 所示。

在亚马逊云的对象存储服务中，保证的一年期对象持久性为 $p'_f = 99.999999999\%$，也就是说，数据在一年之内出现错误的概率小于 10^{-11}。在微软的 Azure 对象存储服务[46]中，保证的一年期对象持久性与选取的备份方法有关。对于本地备份（Locally-Redundant Storage，即 LRS）来说，微软保证一年内错误率小于 10^{-11}，对于跨可用区备份（Zone-Redundant Storage，即 ZRS）来说此数值为 10^{-12}，对于异地备份（Geo-Redundant Storage，即 GRS）来说此数值为 10^{-16}。

表 3.1　云存储服务提供商保证的对象持久性

存储服务提供商	对象持久性
Amazon S3[47]	99.999999999%
Azure Blob Storage (LRS)[48]	99.999999999%
Azure Blob Storage (ZRS)[48]	99.9999999999%
Azure Blob Storage (GRS)[48]	99.99999999999999%

3.2.3　提高数据可用性的成本收益分析

对于重要的数据，我们需要保证更高的数据可用性，因此需要选择更高的对象持久性保证的存储系统。也就是说，数据的重要性与 p'_f 的选取是正相关的。但是，要想提高 p'_f，就需要更多的存储设备来存储数据的冗余备份，存储系统的成本也会随之上升。我们对数据存储过程的成本和收益进行分析可以发现，若假设一个单位的数据价值为 x 个单位，则可用性为 p'_f 时数据的收益为

xp'_f。假设存储系统的对象持久性达到 p'_f 时，每个单位数据需要占用 $s(p'_f)$ 单位的存储节点，每个单位的存储节点价格为 y 个单位，则数据存储的成本为 $ys(p'_f)$。因此可以得到系统在存储收益和存储成本相等时的平衡点为：

$$xp'_f = ys(p'_f) \tag{3.1}$$

在达到此平衡点后，若继续提高数据可用性，数据存储的边际成本会大于其边际收益，造成存储资源的浪费。因此，我们需要根据不同类型数据的重要程度，有针对性地选取对象持久性 p'_f，进而提升存储系统的成本收益率。

3.3　数据可用性和磁盘故障率的关系

3.3.1　固定磁盘故障率模型

在目前的分布式存储系统中，一般均采用固定的存储设备故障率的模型，来计算对象的持久性。固定磁盘故障率模型假设系统中共有 $n+m$ 块磁盘，每块硬盘故障率为 p_d，易得硬盘完好的概率为 $1-p_d$。假设硬盘发生损坏是独立随机事件，也就是不同硬盘的损坏之间没有关联，那么根据概率计算的乘法公式，可以推导得出，在 $n+m$ 块磁盘中，恰好有 i 块磁盘发生故障，另 $n+m-i$ 块磁盘完好的概率为：

$$\binom{n+m}{i} p_d^i (1-p_d)^{n+m-i} \tag{3.2}$$

其中 $\binom{n+m}{i}$ 为二项式系数，代表着从 n 块硬盘中任选 i 块硬盘的所有组合数。

对于有 n 个数据块，m 个校验块的纠删码编码，要想保障对象的持久性，需要至少有 m 块硬盘不发生故障，因此概率为恰好有 0 块磁盘故障到恰好有 m 块磁盘故障的所有可能性求和。即：

$$p_f(n,m) = \sum_{i=0}^{m} \binom{n+m}{i} p_d^i (1-p_d)^{n+m-i} \tag{3.3}$$

其中 p_f 即为对象的持久性，满足 $0 < p_f < 1$，数据块 n 和校验块 m 满足 $n > 0$ 且 $m > 0$，硬盘故障率 p_d 满足 $0 < p_d < 1$。

特别地，若取 $n = 1$，那么系统便退化为多副本备份，此时对象持久性为：

$$p_f(1,m) = \sum_{i=0}^{m} \binom{1+m}{i} p_d^i (1-p_d)^{1+m-i} \tag{3.4}$$

根据二项式定理：

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i} \tag{3.5}$$

我们有：

$$(p_d + (1-p_d))^{1+m} = \sum_{i=0}^{1+m} \binom{1+m}{i} p_d^i (1-p_d)^{1+m-i} \tag{3.6}$$

代入多副本备份的对象持久性公式中，得：

$$\begin{aligned} p_f(1,m) &= \sum_{i=0}^{m} \binom{1+m}{i} p_d^i (1-p_d)^{1+m-i} \\ &= \left(\sum_{i=0}^{1+m} \binom{1+m}{i} p_d^i (1-p_d)^{1+m-i} \right) \\ &\quad - \left(\binom{1+m}{1+m} p_d^{1+m} (1-p_d)^{1+m-(1+m)} \right) \\ &= (p_d + (1-p_d))^{1+m} - p_d^{1+m} (1-p_d)^0 \\ &= 1^{1+m} - p_d^{1+m} \\ &= 1 - p_d^{1+m} \end{aligned} \tag{3.7}$$

此公式即退化为在多副本备份中，要想保证数据不丢失，需要满足所有硬盘不同时损坏。对象的持久性为 1 减去所有磁盘同时发生损坏的概率。

3.3.2　动态磁盘故障率模型

但在实际情况中，每块磁盘型号和磁盘使用时间经常不尽相同，导致磁盘故障率也存在差异。因此，我们可以尝试根据磁盘的统计信息，预测得到不同磁盘当前的故障率。根据预测得到的差异化故障率，我们可以据此计算对象持久性，从而更大限度的降低纠删码编码的存储冗余。

1. 基于磁盘统计信息的磁盘故障率预测

根据 Ma 等[49]的研究，由于磁盘在扇区损坏时会重新分配完好的扇区来进行代替，因此磁盘的重分配的扇区数（Reallocated Sectors）与磁盘的故障率密切相关，其中磁盘的重分配扇区数可以通过其 S.M.A.R.T. 信息[50]获得。磁盘的故障率可以根据 S.M.A.R.T. 等信息来预测[51–57]。根据贝叶斯定理（Bayes' Theorem），我们可以使用数据中心完好磁盘和损坏磁盘的统计信息，预测磁盘故障率。

贝叶斯定理是概率论中的一个定理，描述在已知一些条件下某事件的发生概率。关于随机事件 A 和 B 的条件概率，贝叶斯定理给出了：

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{3.8}$$

其中：

- $P(A|B)$ 是已知 B 发生后，A 的条件概率。也由于得自 B 的取值而被称作 A 的后验概率

- $P(A)$ 是 A 的先验概率（或边缘概率）。之所以称为“先验”是因为它不考虑任何 B 方面的因素
- $P(B|A)$ 是已知 A 发生后，B 的条件概率。也由于得自 A 的取值而被称作 B 的后验概率
- $P(B)$ 是 B 的先验概率

在数据中心的所有完好磁盘和损坏磁盘中，我们定义：

- N_{RS} 是一块磁盘的重分配扇区数
- $P(fail|N_{RS})$ 重分配扇区数至少为 N_{RS} 的磁盘故障率
- $P(fail)$ 是磁盘发生故障的概率
- $P(N_{RS}|fail)$ 是一块故障磁盘重分配扇区数大于 N_{RS} 的概率
- $P(N_{RS})$ 是磁盘重分配扇区数大于 N_{RS} 的概率

根据贝叶斯定理，我们可以得到这块硬盘的故障率：

$$\begin{aligned} p_d &= P(fail|N_{RS}) \\ &= \frac{P(N_{RS}|fail)\times P(fail)}{P(N_{RS})} \\ &= \frac{\dfrac{\text{num. of failed disks with } N_{RS}}{\text{num. of failed disks}} \times \dfrac{\text{num. of failed disks}}{\text{num. of disks}}}{\dfrac{\text{num. of all disks with } N_{RS}}{\text{num. of disks}}} \\ &= \frac{\text{num. of failed disks with } N_{RS}}{\text{num. of all disks with } N_{RS}} \end{aligned} \tag{3.9}$$

因此，我们可以根据数据中心中所有完好磁盘和损坏磁盘的 N_{RS} 统计量，预测磁盘的故障率 p_d。

2. 基于动态磁盘故障率的对象可用性计算

为了定量地描述动态磁盘故障率模型，我们首先需要对磁盘组的几种数学表达进行定义。首先我们定义 n 块硬盘 $\{d_1, d_2, \cdots, d_n\}$，对于由 n 块硬盘组成的集合则为 $S'_d(n) = \{d_1, d_2, \cdots, d_n\}$，其中 $\|S'_d(n)\| = n$。其次定义集合 $S_a(n,k)$ 为 $\forall S_d \in S_a(n,k)$ 均满足下述性质的集合：

- $\forall d_i \in S_d$ 均有 $d_i \in S'_d(n)$
- $\|S_d\| = k$
- $\|S_a(n,k)\| = \binom{n}{k}$

也就是说，$S_a(n,k)$ 为从 $S'_d(n)$ 中任取 k 个元素组成的集合的全体，即：

$$S_a(n,k)=\{\{d_1,d_2,\cdots,d_k\},\{d_1,\cdots,d_{k-1},d_{k+1}\},\cdots,\{d_{n-k+1},d_{n-k+2},\cdots,d_n\}\} \tag{3.10}$$

基于上述定义，我们对动态磁盘故障率模型的对象持久性进行计算。易知，对于任意磁盘集合 $S_d{\subseteq}S'_d(n)$，其相对于全集 $S'_d(n)$ 的补集为 $S'_d(n)-S_d$。假设存在一块磁盘 d_j，该磁盘的故障率为 p_{d_j}，若对于磁盘损坏事件来说满足独立事件假设，即不同磁盘的损坏之间没有内在联系，那么根据概率论的乘法公式，可以得到 S_d 全部损坏，并且 $S'_d(n)-S_d$ 全部完好的概率为：

$$\left(\prod_{\forall d_j\in S_d}p_{d_j}\right)\left(\prod_{\forall d_j\in S'_d(n+m)-S_d}(1-p_{d_j})\right) \tag{3.11}$$

根据纠删码的解码原理，我们需要至少 n 块磁盘是完好的，或者说至多有 m 块磁盘是损坏的。因此，只要在存储数据块和编码块的全部磁盘 $S'_d(n+m)$ 中存在磁盘组 S_{good}，满足 S_{good} 中的所有磁盘均是可用的，$S_{good}{\subseteq}S'_d(n)$ 且 $\|S_{good}\|\geq n$，我们便可根据 S_{good} 磁盘组中的数据恢复出全部的原始数据。由此，我们可以得到 S_{good} 的取值范围 $\forall S_{good}{\in}S_a(n+m,i)$，其中 $0\leq i\leq m$。

由于满足对象持久性约束的情况为满足上述条件的全体，根据概率论的加法公式，求和得到概率为：

$$p_f(n,m)=\sum_{i=0}^{m}\sum_{\forall S_d\in S_a(n+m,i)}\left[\left(\prod_{\forall d_j\in S_d}p_{d_j}\right)\left(\prod_{\forall d_j\in S'_d(n+m)-S_d}(1-p_{d_j})\right)\right] \tag{3.12}$$

特别地，当公式中的 p_{d_j} 均取值 p_d 时，也就是各磁盘故障率相同，此公式便退化为固定磁盘故障率的对象持久性公式：

$$\begin{aligned}
p_f(n,m)&=\sum_{i=0}^{m}\sum_{\forall S_d\in S_a(n+m,i)}\left[\left(\prod_{\forall d_j\in S_d}p_d\right)\left(\prod_{\forall d_j\in S'_d(n+m)-S_d}(1-p_d)\right)\right]\\
&=\sum_{i=0}^{m}\sum_{\forall S_d\in S_a(n+m,i)}\left[\left({p_d}^{\|S_d\|}\right)\left((1-p_d)^{\|S'_d(n+m)-S_d\|}\right)\right]\\
&=\sum_{i=0}^{m}\sum_{\forall S_d\in S_a(n+m,i)}\left[\left({p_d}^{i}\right)\left((1-p_d)^{n+m-i}\right)\right]\\
&=\sum_{i=0}^{m}\|S_a(n+m,i)\|\left({p_d}^{i}(1-p_d)^{n+m-i}\right)\\
&=\sum_{i=0}^{m}\binom{n+m}{i}{p_d}^{i}(1-p_d)^{n+m-i}
\end{aligned} \tag{3.13}$$

3.4　损坏节点更换对模型的影响

在上述分析中，对于总节点数为 s 的存储系统，若待存储对象持久性为 p'_f，On-demand ARECS 算法需要在所有的节点中选取 $n+m$ 个节点，满足 $p_f(n,m) \geq p'_f$，从而做到在保证对象的持久性需求的基础上，选取最优的编码策略。然而，上述分析中的对象持久性和磁盘故障率均按照一年期进行计算，但在实际的数据中心中，存储系统的运行和维护人员每个工作日均会对节点状态进行检查，并对损坏节点进行替换与数据的恢复重建。我们假设一个节点从损坏到被发现且替换并重建数据的时间为 t_r 年，显然 $t_r \ll 1$，我们用在 t_r 周期内的对象持久性和磁盘故障率对上述公式进行替换：

$$p_f(n,m) = \sum_{i=0}^{m} \sum_{\forall S_d \in S_a(n+m,i)} \left[\left(\prod_{\forall d_j \in S_d} p'_{d_j} \right) \left(\prod_{\forall d_j \in S'_d(n+m)-S_d} (1-p'_{d_j}) \right) \right] \tag{3.14}$$

其中的 p'_f 和 p'_d 的取值为在 t_r 周期内的损坏概率，而非一年期的概率值。假设在一年内各个时点磁盘损坏的可能性相同且独立，则可以得到 $p'_d = 1-(1-p_d)^{t_r}$，$p'_f = p_f^{t_r}$。因此对于 On-demand ARECS 算法来说，需要选取的编码策略 $(n+m,n)$ 应该满足计算的节点可靠性大于等于对象的持久性。在 t_r 周期内，模型确定的节点可靠性和对象持久性的差值为：

$$\begin{aligned} f(n,m) = \sum_{i=0}^{m} \sum_{\forall S_d \in S_a(n+m,i)} \Bigg[& \left(\prod_{\forall d_j \in S_d} \left(1-(1-p_{d_j})^{t_r}\right) \right) \\ & \cdot \left(\prod_{\forall d_j \in S'_d(n+m)-S_d} (1-p_{d_j})^{t_r} \right) \Bigg] - p_f^{t_r} \end{aligned} \tag{3.15}$$

显然节点可靠性应达到或超过对象的持久性意味着此差值为非负值，即 $f(n,m) \geq 0$。由于编码的检验块数量 m 越少，数据冗余度越低，存储延迟越小，因此 m 应取满足约束 $f(n,m) \geq 0$ 的最小值：

$$m(n) = \underset{m}{\operatorname{argmin}}\, f(n,m), \text{ where } f(n,m) \geq 0 \tag{3.16}$$

3.5　实验与分析

3.5.1　模拟实验环境

在这一节的内容中，我们将通过模拟实验的方法，设计一个分布式存储系统实验环境，用以模拟并分析成本收益模型下的对象可用性选取，以及 On-demand ARECS 磁盘故障率模型的磁盘冗余度。根据 Pinheiro 等人的研究[58]，使用时间为一年的磁盘 $p_d = 1.7\%$，而使用时间为三年的磁盘 $p_d = 8.6\%$。我们

选取一个模拟实验环境，模拟实现了两种存储节点，节点 A 的磁盘故障率为 $p_{d_A} = 1.7\%$，节点 B 的磁盘故障率为 $p_{d_B} = 8.6\%$，存储系统中分别有 18 个节点 A 和节点 B。在通常的数据中心中，由于有专门人员负责损坏节点的检测与更换，因此损坏节点基本均能被及时发现。对于损坏节点的数据重建，因为目前的硬盘存取速度已经较快，因此正常情况下的用时也不会太长。基于此，我们不妨选取更换损坏硬盘并重建数据为 3 天，即取 $t_r = \dfrac{3}{365}$。基于此实验环境，我们进行了以下模拟与分析。

3.5.2　不同磁盘故障率模型与磁盘冗余度模拟与分析

我们首先模拟在不同的磁盘故障率模型下，不同编码方案对应的数据冗余度。根据前文的分析，目前各主要云存储服务提供商对于对象持久性的保证值一般均为 $p_f = 99.999999999\%$，我们在此实验中不妨也选取此值作为数据的可用性需求。对于数据冗余度来说，其定义为冗余数据量和原始数据量的比值，可记为 $\dfrac{m}{n}$。我们通过模拟实验可以得出，On-demand ARECS 的动态磁盘可靠性模型与传统的固定磁盘可靠性模型的数据冗余度对比如图 3.1。

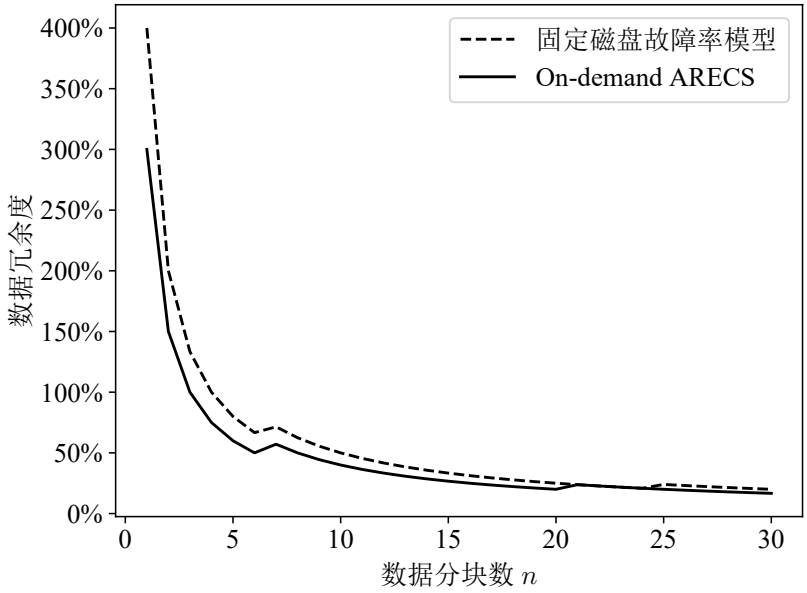


图 3.1　不同硬盘故障率模型的数据切分数与冗余度关系

从图 3.1 中我们可以看出，虽然在总体上 On-demand ARECS 动态磁盘故障率模型的磁盘冗余度相比固定磁盘故障率模型低，但是在一些特殊情况下，比如 $n = 21$ 到 $n = 24$ 时，On-demand ARECS 动态磁盘故障率模型和固定磁盘故障率模型得到了相同的结果，即均选取了 $m = 5$ 的编码方案。我们分析实验过程可以发现，如果单纯对于概率的计算来说，只要 n 值和 m 值选取相同，则数

据冗余度就是相同的。而在实际的分布式存储系统中，我们需要对编码块的数量，即 m 的值进行向上取整，对于我们的实验环境来说，我们根据公式：

$$m(n) = \underset{m}{\operatorname{argmin}} f(n,m), \text{ where } f(n,m) \geq 0 \tag{3.17}$$

计算 $n = 21$ 到 $n = 24$ 时得到的 m 值均满足 $4 < m < 5$，那么我们就需要选取 $m = 5$ 才能保障数据安全。换句话说，不管实际上我们需要 4.1 个校验块，还是实际上需要 4.9 个校验块，我们都需要向上取整为 5 个校验块。因此，m 值的选取为了具有实际意义，其公式应为：

$$m(n) = \left\lceil \underset{m}{\operatorname{argmin}} f(n,m) \right\rceil, \text{ where } f(n,m) \geq 0 \tag{3.18}$$

由于 On-demand ARECS 算法核心在于对于不同的节点状况动态选取合适的 n 值，也就是说，我们无法预先确定一个 n 值，然后计算其最优的存储方案，我们需要在算法中动态选择负载低、带宽高的节点，因而 n 值的最终确定需要在实际系统中动态确定。故在 On-demand ARECS 的动态磁盘故障率模型中，我们主要需要关注平均性能，也就是说对于所有 n 值来说，平均冗余度更低。

通过图 3.1 的实验结果中我们可以看出，On-demand ARECS 模型相比于传统算法，将数据冗余度平均降低了 18%，而最坏性能与传统固定磁盘故障率模型相当。

3.5.3　数据可用性与成本收益模拟与分析

根据 Liu 等人的研究[59]和 Tahoe 分布式文件系统[60]的默认实现，我们可以发现 n 一般均选取 $n = 10$ 左右较为合适。因此在我们本节的模拟中，我们不妨选取 $n = 10$ 来进行分析。对于具体 n 值的选取，我们将在下一章中通过对存储系统当前状态的分析，动态选取确定最优的编码方案和节点。

根据前文的分析，数据可用性的取值主要受制于数据本身的重要性，其简化的成本收益模型为：

$$xp'_f = ys(p'_f) \tag{3.19}$$

其中等式左半部分为存储系统的边际收益，右半部分为存储系统的边际成本。我们假设一份数据存储的成本为 80 个单位，数据本身的价值为 100 个单位，那么我们可以得到在不同磁盘故障率模型下，边际成本与边际收益的曲线图如图 3.2 所示。

从图 3.2 中可以看出，存储数据的边际成本相比于存储数据的边际收益上升更快，在固定磁盘故障率模型下，针对上述模拟实验环境，$p'_f \approx 99.995\%$ 时边际成本和边际收益相等。在 On-demand ARECS 浮动磁盘故障率模型下，针对上述模拟实验环境，$p'_f \approx 99.999995\%$ 时边际成本和边际收益相等。

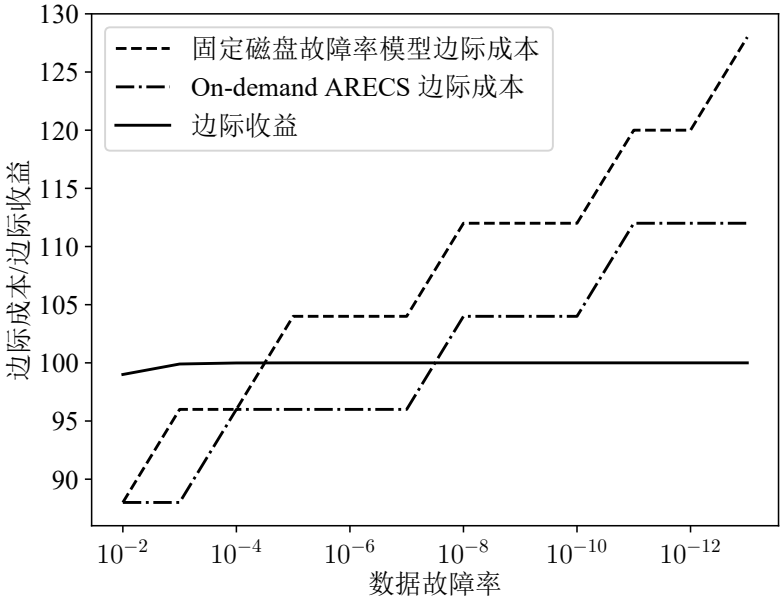


图 3.2　不同硬盘故障率模型的边际成本与边际收益分析

综上所述，On-demand ARECS 浮动磁盘故障率模型相比于固定磁盘故障率模型，达到边际成本收益平衡点时，对象持久性提高了三个数量级。在达到此平衡点后，若盲目增大对象的持久性 p'_f 对于存储系统来说是得不偿失的。

3.6　小结

在本章中，我们首先介绍了数据可用性的概念，说明了选取适当的数据可用性的意义。其次，我们针对存储系统中磁盘的特点，给出了一种推测磁盘可靠性的方法，并基于此计算对象的持久性。我们提出了 On-demand ARECS 动态磁盘故障率模型取代传统的固定磁盘故障率模型，以对实时推测的变化故障率的磁盘进行对象可靠性的计算。我们还分析了在数据中心中，定期维护、更换损坏磁盘和重建数据对对象可用性的影响。我们通过模拟实验的方式，对比分析了固定磁盘可靠性模型和 On-demand ARECS 动态磁盘可靠性模型。在模拟环境中，在达到预期对象可用性的条件下，On-demand ARECS 动态磁盘可靠性模型相比固定磁盘可靠性模型，将磁盘冗余度平均降低了 18%。在达到边际成本收益平衡点的条件下，On-demand ARECS 动态磁盘可靠性模型相比固定磁盘可靠性模型，将对象可用性提高了三个数量级。

第 4 章　面向动态数据可用性的纠删码对象存储策略

4.1　引言

现有的基于纠删码的分布式文件系统，如 Hadoop 分布式文件系统、Ceph 分布式文件系统和 Tahoe-LAFS 分布式文件系统等，均需要在存储系统进行初始化时确定纠删码的编码方案，在存储系统初始化后，除非重新创建新的存储池，并不能动态地调整数据块和校验块的数量。在上一章中，我们提出了基于不同类型的存储设备的状态进行数据可用性的推断策略，在本章中，我们将继续讨论 On-demand ARECS 算法，单独对每个具有不同的可用性需求的数据对象，根据系统实时状态计算数据块和校验块的编码方案，并选取最快的存储节点集合，从而实现基于动态数据可用性定义的高效对象存储。我们还将基于 Tahoe-LAFS 分布式文件系统实现 On-demand ARECS 算法，并部署一个分布式文件系统作为测试环境，对自适应纠删码对象存储策略 On-demand ARECS 的实际性能进行评估。

对于由 s 块硬盘组成的分布式存储系统，若待存储的对象持久性需求为 p'_f，对象使用纠删码的方式冗余存储于此分布式存储系统的 $n+m$ 块磁盘中，其存储方案的数据正确率为 $p_f(n,m)$。我们需要在能满足对象持久性需求，即满足约束 $p_f(n,m) \geq p'_f$ 的若干组纠删码方案中，寻找速度最快的一组方案完成存储操作。在本章中，我们针对所有可能的存储方案组合计算读写所需要的时间 $t_f(n+m)$，并选择读写时间 $t_f(n+m)$ 最小的磁盘组，这样选取的磁盘组便是可以满足对象持久性需求的速度最快的磁盘组。即通过动态计算 p_{d_j} 和 $t_f(n+m)$，选取满足 $p_f(n,m) \geq p'_f$ 且 $t_f(n+m)$ 最小的纠删码方案，作为分布式存储系统在当前的工作状况下最优的纠删码编码方案，从而在满足对象的可用性需求的所有存储方案中，选取当前状况下存储速度最快的方案，避免了传统算法确定性编码方案的使用，更加快速地将数据存储于分布式存储系统。

4.2　基于动态数据可用性的编码方案选择算法

对于 On-demand ARECS 算法来说，需要我们首先确定数据的可用性。根据上一章的分析我们可以得知，由于边际成本较边际收益增加幅度更快，因此选取合理的对象持久性，可以最大限度的提高成本收益率。On-demand ARECS 算法需要我们根据数据本身的特征选取合适的对象持久性，对于较为重要的数据，选取较高的 p'_f，对于一般的数据，选取较低的 p'_f，在此基础上进行算法的

优化选择。

其次 On-demand ARECS 算法根据磁盘的属性等信息预测磁盘的可靠性。我们通过读取磁盘的 S.M.A.R.T. 信息，获得磁盘的重分配扇区数 N_{RS}，并通过统计数据中心中所有磁盘中和所有损坏磁盘中 N_{RS} 的分布，依靠贝叶斯定理计算当前磁盘的故障率：

$$p_d = \frac{\text{num. of failed disks with } N_{RS}}{\text{num. of all disks with } N_{RS}} \tag{4.1}$$

在得出对象持久性和磁盘可靠性之后，我们便可以依据对象可用性和磁盘可靠性的关系计算存储的纠删码方案，通过上一章中的分析与计算，我们可以得知编码块数 m 的取值为：

$$m(n) = \left\lceil \underset{m}{\operatorname{argmin}} f(n,m) \right\rceil, \text{ where } f(n,m) \geq 0 \tag{4.2}$$

其中 $f(n,m)$ 为：

$$\begin{aligned} f(n,m) = \sum_{i=0}^{m} \sum_{\forall S_d \in S_a(n+m,i)} \Bigg[\Bigg(\prod_{\forall d_j \in S_d} \left(1-(1-p_{d_j})^{t_r}\right) \Bigg) \\ \cdot \Bigg(\prod_{\forall d_j \in S'_d(n+m)-S_d} (1-p_{d_j})^{t_r} \Bigg) \Bigg] - p_f^{t_r} \end{aligned} \tag{4.3}$$

因此，我们便可以基于此选择数据冗余度最低的纠删码编码方案 $(n+m,n)$ 及其对应的磁盘组 $S'_d(n+m)$。需要注意的是，由于不同磁盘可能存在故障率相同的情况，且 m 的选取对 $\underset{m}{\operatorname{argmin}} f(n,m)$ 进行了向上取整操作，因此此处的 $S'_d(n+m)$ 存在不止一组满足条件的组合。

4.3　基于存储速度的节点选择策略

4.3.1　单个数据块磁盘组读写时间的计算

由于在编码方案选择的过程中选取的 m 值和磁盘组 $S'_d(n+m)$ 依赖于 n 值的确定，且对于相同的 $(n+m,n)$ 编码方案，存在多种满足条件的磁盘组合 $S'_d(n+m)$，因此我们需要根据存储设备当前的负载等信息，从候选磁盘组中选取速度最快的一组，作为最终存储数据的磁盘组。我们可以用磁盘当前进行一个读写操作所需要的时间，来代表存储设备的速度。在本节中，我们首先对存储系统中单个数据块的读写时间进行计算。

对于分布式文件系统中的磁盘来说，其读写时间主要由以下三部分操作所需要的事件共同组成：

- 队列等待时间 t_{queue}：当存储系统负载过大时，磁盘的读写请求不能被实时处理，操作系统便会将请求放入磁盘读写队列中依次进行调度，在读写请求发出到操作系统实际调度执行之间的等待时间便是磁盘的队列等待时间，通过优化磁盘调度算法、平衡节点读写请求等方式可以有效地降低队列等待时间

- 读写延迟 $t_{latency}$：对于本地磁盘的读写，需要通过寻道等操作确定数据的位置，对于远程磁盘的读写，需要与远程磁盘建立连接并传输读写请求，这些操作均会产生磁盘的读写延迟，在分布式存储系统中，我们可以采用就近调度等方案来减少读写延迟

- 数据写入时间：数据实际的写入时间仅取决于待写入的数据量 $data\_size$ 和磁盘的数据传输带宽 $bandwidth$，对于分布式文件系统，可以通过增加节点间的传输带宽来提高数据写入时间

可以看出，磁盘的总读写时间 t_d 应为以上三个步骤所需要的时间之和，具体可以表示为：

$$t_d = t_{queue} + t_{latency} + \frac{data\_size}{bandwidth} \tag{4.4}$$

对于使用互联网传输的分布式存储系统来说，由于网络环境波动等原因，读写延迟和数据传输带宽是实时发生变化的。因此，我们在计算总的磁盘读写时间 t_d 时，需要动态采集读写延时和数据传输带宽。在 On-demand ARECS 算法中，我们在存储系统中的节点每次进行数据读写时，将本次操作的读写延时和数据传输带宽等关键参数进行保存，以便存储系统后续进行写入时使用。

对于分布式存储系统来说，不同节点之间的读写操作是相对独立的，因此可以认为数据在不同节点间是并行读写的。因此，总读写时间实际上就是所有节点均完成读写所需要的时间，即所有节点的读写时间 t_{d_i} 中的最大值 $\max\limits_{d_i \in S'_d(n+m)} t_{d_i}$。对于纠删码分布式存储系统来说，在数据进行写入之前，需要首先进行纠删码校验码的编码计算，因此需要再加上纠删码编码所耗费的 CPU 时间 t_{calc}。进而此纠删码分布式存储系统的总读写时间可以表示为：

$$t_f(n+m) = t_{calc} + \max_{d_i \in S'_d(n+m)} t_{d_i} \tag{4.5}$$

4.3.2　分块存储的对象所需总磁盘读写时间的计算

对于较大的对象，为避免直接采用纠删码编码时编码矩阵过大，计算效率过低的问题，我们可以将较大的数据分为多个块，不同块之间可以并行进行存储，从而提高存储效率。若将数据分为 r 块 $(b_1, b_2, \cdots, b_r)$，则每一块数据的读

写时间分别为：

$$\begin{cases} (t_f(n+m))_{b_1} = (t_{calc})_{b_1} + \left(\max\limits_{d_i \in S'_d(n+m)} t_{d_i} \right)_{b_1} \\ (t_f(n+m))_{b_2} = (t_{calc})_{b_2} + \left(\max\limits_{d_i \in S'_d(n+m)} t_{d_i} \right)_{b_2} \\ \vdots \\ (t_f(n+m))_{b_r} = (t_{calc})_{b_r} + \left(\max\limits_{d_i \in S'_d(n+m)} t_{d_i} \right)_{b_r} \end{cases} \tag{4.6}$$

我们不妨假设数据从 b_1 到 b_r 按照顺序依次传输，那么存储系统处理这些数据所用的总读写时间为：

$$t_f(n+m) = \sum_{x=1}^{r} (t_f(n+m))_{b_x} \tag{4.7}$$

即：

$$t_f(n+m) = \sum_{x=1}^{r} \left((t_{calc})_{b_x} + \left(\max_{d_i \in S'_d(n+m)} t_{d_i} \right)_{b_x} \right) \tag{4.8}$$

但由于数据的计算和数据的传输可以并行执行，在传输前一个数据块的内容的同时，可以同时并行进行后一个数据块的编码计算，即传输 b_i 和 b_{i+1} 块所用的时间不是：

$$(t_f(n+m))_{\{b_i,b_{i+1}\}} = (t_{calc})_{b_i} + \left(\max_{d_i \in S'_d(n+m)} t_{d_i} \right)_{b_i} + (t_{calc})_{b_{i+1}} + \left(\max_{d_i \in S'_d(n+m)} t_{d_i} \right)_{b_{i+1}} \tag{4.9}$$

而应该是：

$$\begin{aligned} (t_f(n+m))_{\{b_i,b_{i+1}\}} = (t_{calc})_{b_i} + \max & \left(\left(\max_{d_i \in S'_d(n+m)} t_{d_i} \right)_{b_i}, (t_{calc})_{b_{i+1}} \right) \\ & + \left(\max_{d_i \in S'_d(n+m)} t_{d_i} \right)_{b_{i+1}} \end{aligned} \tag{4.10}$$

因此，传输这 r 个数据块所需要花费的总时间可以表达为：

$$\begin{aligned} t_f(n+m) = (t_{calc})_{b_1} + & \left(\sum_{x=2}^{r} \max \left((t_{calc})_{b_{x-1}}, \left(\max_{d_i \in S'_d(n+m)} t_{d_i} \right)_{b_x} \right) \right) \\ & + \left(\max_{d_i \in S'_d(n+m)} t_{d_i} \right)_{b_r} \end{aligned} \tag{4.11}$$

由于在实际的分布式存储系统实现中，我们一般将数据切分为大小确定且均等的数据块，故对于上式来说，可以取 $(t_{calc})_{b_x} = t_{calc}$，$\left(\max\limits_{d_i \in S'_d(n+m)} t_{d_i} \right)_{b_x} = \max\limits_{d_i \in S'_d(n+m)} t_{d_i}$。因此我们可以得到实际的分布式存储系统中数据编码和传输所需要的总时间：

$$t_f(n+m) = t_{calc} + \max_{d_i \in S'_d(n+m)} t_{d_i} + (r-1) \left(\max \left(t_{calc}, \max_{d_i \in S'_d(n+m)} t_{d_i} \right) \right) \tag{4.12}$$

4.3.3　基于最小化磁盘读写时间的存储节点组合选择策略

根据 Liu 等人的研究[59]，在数据量一定的的前提下，随着数据分块数 n 的增大，虽然数据冗余度有所下降，但纠删码的编码时间迅速上升。因此，需要我们综合考虑数据冗余度和存储速度，选取数据分块数 n 相对较小且数据冗余度不过高的编码方案，无需考虑 n 值过大的存储策略所对应的存储方案，从而降低存储系统中纠删码编码所用的时间，提高存储系统的性能。

在此基础上，我们需要在所有满足上述条件的磁盘组 $S'_d(n+m)$ 中选取存储延时 $t_f(n+m)$ 最小的磁盘组，即选取的磁盘组为：

$$S'_d(n+m) = \underset{S'_d(n+m)}{\operatorname{argmin}} t_f(n+m) \tag{4.13}$$

代入上一节中得到的数据均等切分情况下数据编码和传输所需要的总时间 $t_f(n+m)$，得到选取的磁盘组为：

$$S'_d(n+m) = \underset{S'_d(n+m)}{\operatorname{argmin}} \left(t_{calc} + \max_{d_i \in S'_d(n+m)} t_{d_i} + (r-1) \left(\max \left(t_{calc}, \max_{d_i \in S'_d(n+m)} t_{d_i} \right) \right) \right) \tag{4.14}$$

4.4　实验与分析

4.4.1　实验环境

为了验证 On-demand ARECS 算法动态选择纠删码编码策略和存储节点的算法效果，我们基于 Tahoe-LAFS 分布式文件系统[60]实现了 On-demand ARECS 算法。Tahoe-LAFS 是一个免费、开源、安全、去中心化、容错性好的分布式存储和分布式文件系统，其围绕“最小权限原则”（Principle of Least Authority，即 POLA）进行设计和实现，严格使用加密传输数据，并针对每项功能仅提供能完成相应操作的最小特权集。由于数据均进行了加密和冗余备份，因此存储系统中的每个节点均无需保证其可靠性。Tahoe-LAFS 的纠删码引擎基于 zfec 纠删码编码库实现，其提供了三种不同类型的节点：

- 介绍节点：介绍节点负责连接分布式存储系统中所有的存储节点和客户节点，作为节点间的信息传输的一个桥梁，联系起分布式文件系统的所有节点
- 存储节点：存储节点负责实际的数据存储，由纠删码编码的原始数据块和冗余校验块均存储于存储节点
- 客户节点：客户节点负责提供分布式文件系统的用户接口，并将用户的读写请求转换为实际存储节点的操作指令传输给存储节点

在 Tahoe-LAFS 分布式文件系统的默认纠删码引擎配置中，针对所有数据均采用 $(n+m,n)=(10,3)$ 的纠删码编码策略。Tahoe-LAFS 分布式文件系统将文件先进行加密，然后分为若干数据单元，每个单元独立进行编码。Tahoe-LAFS 分布式文件系统将每个单元的数据切分为 n 个原始数据块，并通过纠删码编码矩阵计算得出 m 个冗余校验块。之后，Tahoe-LAFS 分布式文件系统随机选取 $n+m$ 个剩余存储空间足够的存储节点，并将这 $n+m$ 个块分别存储于这些存储节点中。

我们在 Tahoe-LAFS 分布式文件系统的基础上实现了 On-demand ARECS 算法，从而动态地确定纠删码编码策略和存储节点。我们在基于 KVM 虚拟化技术的分布式集群中，部署了一个由 37 个节点组成的分布式文件系统实验环境，具体信息如表 4.1 和图 4.1 所示。存储系统中一共有 3 种不同的节点：客户节点、A 类存储节点和 B 类存储节点。客户节点共有 1 个，充当 Tahoe-LAFS 分布式文件系统中的客户节点和介绍节点，负责进行实验操作，发出存储请求并进行测试。A 类存储节点为资源相对受限的存储节点，其计算资源、存储资源和存储可靠性均相对较低。B 类存储节点为相对优质的存储节点，其计算资源、存储资源和存储可靠性均相对较高。所有的存储节点平均分为三组，每组各有 6 个 A 类存储节点和 6 个 B 类存储节点。三组存储节点和客户节点分别放置在不同的机房中。

表 4.1　不同存储节点的配置

存储节点种类	Storage Server A	Storage Server B
vCPU 数量	1 vCPU	2 vCPU
内存容量	1 GB	2 GB
磁盘容量	25 GB	60 GB
磁盘的故障率 p_d	8.6%	1.7%

为了避免机房由于不可抗力的原因发生数据丢失，我们采用异地备份的方法，将三个存储节点的集群分别布置于荷兰阿姆斯特丹（以下简称 AMS）、德国法兰克福（以下简称 FRA）和英国伦敦（以下简称 LON），将客户节点布置于美国纽约（以下简称 NYC），这可以最大限度地满足在 On-demand ARECS 模型中的对磁盘损坏事件独立性的假设。由于 On-demand ARECS 算法考虑了到不同节点间的网络波动问题，因此我们使用真实的国际互联网在节点之间传输数据，以期得到更贴近于实际应用的实验数据。我们在实验过程中对各机房之间的平均通讯延迟（Round-Trip Time，即 RTT）和平均带宽的进行了多次测量，其平均值如图 4.1 所示。由于在国际互联网中不同机房间链路的拥挤程度不同，节点间的平均通讯延迟与节点间的数据传输带宽并不完全成负相关关系。在网络不发生拥堵的情况下，节点间的平均通讯延迟主要取决于节点间的地理

距离，而节点间的数据传输带宽主要取决于节点间光缆的建设容量。即使两个节点相距较远，由于光速的限制无法达到较低的通讯延迟，仍可以通过增大网络容量等方式达到较大的传输带宽。比如图 4.1 中 LON 节点到 FRA 节点的平均通讯延迟高于其到 AMS 节点的平均通讯延迟，但测得的数据传输带宽则是 LON 节点到 FRA 节点更高。

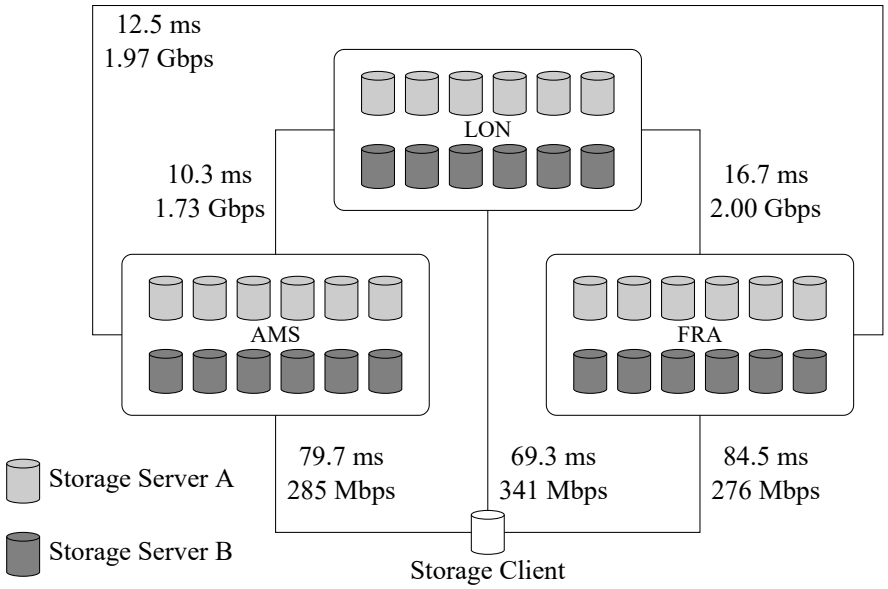


图 4.1　Tahoe-LAFS 实验集群示意

4.4.2　实验结果与分析

在上述实验环境中，我们使用随机生成的实验方法，创建了三种不同的测试数据对象文件，其大小分别为 10 MB、50 MB 和 200 MB。在实验系统中，我们分别针对这三种数据对象测试了 On-demand ARECS 算法的存储速度，得到了在不同分块情况 $(n+m, n)$ 下的文件传输时间如图 4.2 所示。

分析图 4.2 我们可以看出，当数据切分数 n 较小时，数据冗余度较高，因此需要传输的总数据量 $data\_size$ 更大，从而使得总文件传输时间更长。当数据切分数 n 过大时，由于纠删码的编码矩阵更加复杂，编码时间 t_{calc} 时间更长，同时过多的切分数需要选择更多的节点，因此选择到慢速节点的可能性更大，即数据读写延迟 t_{calc} 更大，数据传输带宽 $bandwidth$ 更小，同样使得文件传输时间更长。针对本文中选取的测试环境来说，在数据切分数 n 取 $n = 16$ 左右时存储系统的耗时最短。故若根据实际的传输情况选择数据切分数合适的编码方案，可以很大程度上降低存储系统的文件传输时间。

在 On-demand ARECS 算法中，我们通过动态地选取最优的编码方案和数据存储节点，从而减小数据的存储延时。我们在实现了 On-demand ARECS 算

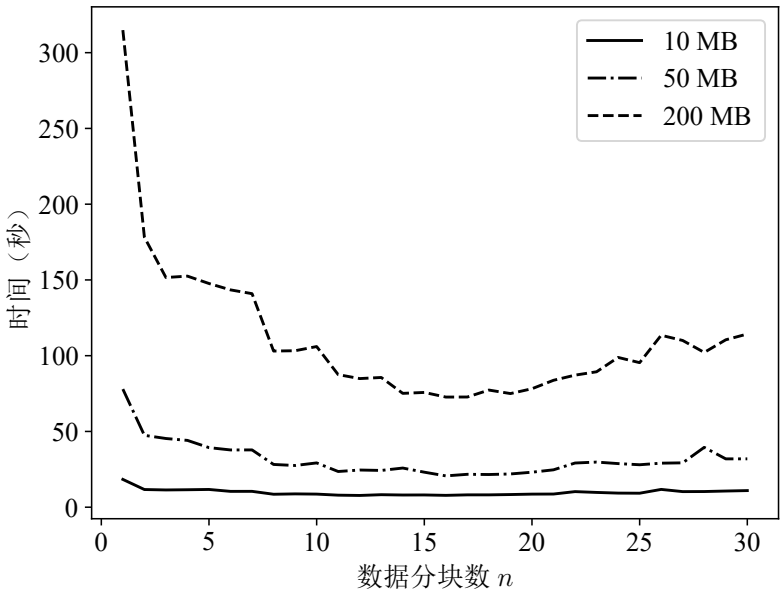


图 4.2　不同数据分块数的存储时间表现

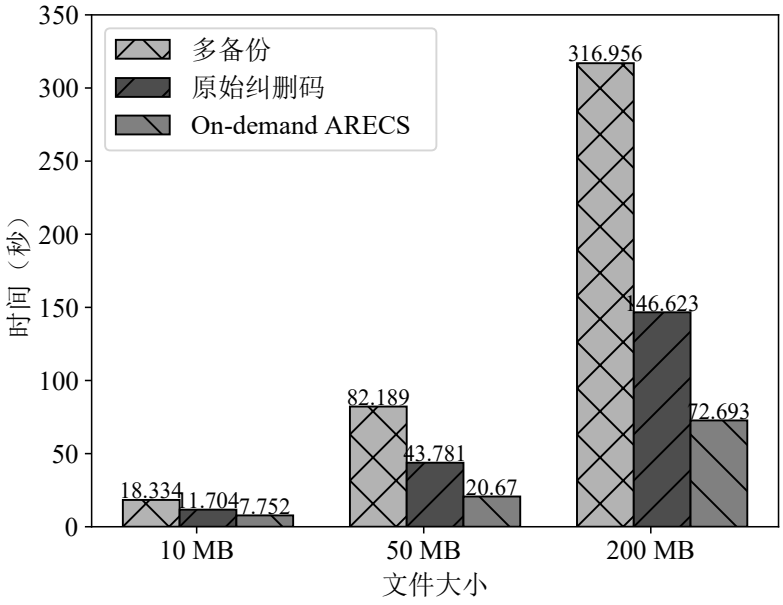


图 4.3　On-demand ARECS 算法和多备份、固定纠删码算法存储延时对比

法的 Tahoe-LAFS 分布式文件系统中，分别测试了原始的多备份方案以及基于 On-demand ARECS 算法的里德-所罗门码纠删码编码方案，针对上述生成的 10 MB、50 MB 和 200 MB 大小的测试数据对象文件进行了实验，实验结果如图 4.3 所示。在实验中，分析存储系统的记录可以得到，其中 On-demand ARECS 算法选取了 $(n+m, n) = (20, 16)$ 的纠删码编码策略，使用的存储节点占全部存储节

点的 56%。原始的纠删码算法则选取了 $(n+m,n)=(10,3)$ 的纠删码编码策略，使用的存储节点占全部存储节点的 28%。多副本备份方案则直接存储了 4 份原始数据作为冗余备份。

根据图 4.3 的实验结果可以看出，On-demand ARECS 算法通过在分布式存储系统中选取适当的纠删码编码方案和节点，最大限度地平衡系统负载，并提高系统的并行度，从而提高了系统的存储效率。其相比于目前的纠删码算法将文件传输时间平均降低了 46%，相比原始的多备份方案则有更加显著地提升。

4.5　小结

在本章中，我们首先介绍了 On-demand ARECS 算法基于动态对象持久性和磁盘可靠性选择纠删码编码方案的步骤，说明了通过最小化磁盘冗余度的方式选取满足可用性需求的候选磁盘组的方案。其次，On-demand ARECS 算法计算这些候选磁盘组编码并传输待存储数据所需要的时间，并选取所需要时间最小的纠删码编码方案和存储节点组合。我们基于 Tahoe-LAFS 分布式文件系统实现了 On-demand ARECS 算法，并部署了一个具有 37 个节点的分布式存储实验系统，测试了不同分块数对存储数据所需要时间的影响。我们还在此实验系统上测试了多备份方案、原始纠删码编码方案和 On-demand ARECS 算法存储文件实际所需要的时间，实验结果表明，On-demand ARECS 算法相比于目前的纠删码编码方案，将存储时间平均降低了 46%，显著地提升了存储系统的写入性能。

第 5 章　基于纠删码对象存储的分布式文件存储

5.1　引言

对于基于纠删码编码的分布式对象存储系统来说，由于纠删码编码和解码的计算开销很大，现有的分布式文件系统在对象存储上直接进行对象的修改操作，不仅会增加系统设计的复杂性，降低系统的通用性，而且由于在对象修改时需进行解码和编码，会导致系统的效率低下。因此，基于纠删码实现的对象存储一般假设对象是只读的，故并不能将存储系统中的对象直接封装为文件供分布式文件系统使用。

我们设计的基于纠删码的分布式文件系统以只读的对象存储系统为基础，将对文件的修改操作映射为对原始对象的修改事件，并将修改操作同样作为对象新增存储于对象存储引擎中，从而在逻辑上对原始对象进行了修改，简化了对象存储的设计，并减少了文件修改时的纠删码解码和重新编码的开销。此修改算法实现了文件的修改和附加等常见操作，从而可以基于此动态纠删码对象存储系统上实现典型的块存储、文件存储和数据库存储系统，以满足不同场景下的高性能高可用的分布式存储需求。

5.2　基于纠删码的分布式存储系统

5.2.1　基于纠删码编码的对象存储

在基于纠删码的分布式存储系统中，为了减少由于纠删码的编码和解码带来的计算开销，可以基于只读的分布式对象存储对数据进行存储。

对于待存储的对象，我们先使用 On-demand ARECS 算法选取校验码块数 m 满足下式的纠删码编码方案 $(n+m, n)$：

$$m(n) = \left\lceil \operatorname*{argmin}_{m} f(n,m) \right\rceil, \text{ where } f(n,m) \geq 0 \tag{5.1}$$

再选取其中总读写时间 $t_f(n+m)$ 取最小值时的磁盘组：

$$S'_d(n+m) = \operatorname*{argmin}_{S'_d(n+m)} \left(t_{calc} + \max_{d_i \in S'_d(n+m)} t_{d_i} + (r-1) \left(\max \left(t_{calc}, \max_{d_i \in S'_d(n+m)} t_{d_i} \right) \right) \right) \tag{5.2}$$

我们将待存储的数据对象分为 n 份原始数据块，并使用里德-所罗门码纠删码编码生成 m 份冗余备份块。对于这 $n+m$ 块数据，我们将其分别存放于选取的磁盘组 $S'_d(n+m)$ 中的 $n+m$ 块磁盘中。这样，我们便可以实现基于纠删码的快速的对象存储。

在数据的存储过程中，由于存储的数据块是只读的，因此每块数据块的长度是固定不变的。根据上一章中的研究，由于较大的对象在纠删码编码时存在由于编码矩阵过大，导致计算效率低下的问题，故在大对象存储时一般将其切分为较小的大小确定且均等的数据块。因此，我们在存储数据块时，可以不通过本地文件系统，直接将存储设备的线性地址空间分为若干个大小确定且均等的数据簇，每个数据簇存储一个数据块。这样，一个数据块就可以直接使用存储设备编号和数据簇编号来进行索引，避免了本地文件系统带来的性能损失，从而提高系统的并行性。

5.2.2　基于对象存储的分布式文件系统中间件

由于目前的分布式数据库以及云计算多需要一个支持修改操作的存储后端，上述只读的分布式对象存储不能满足这些存储领域的需求。因此，为了分布式存储系统的通用性，我们在基于纠删码实现的对象存储的基础上，设计了一个支持文件修改（Modify）和附加（Append）操作的存储引擎，来满足目前大型数据中心中多样化的存储需求。为了实现方便，此文件系统基于 Linux 的虚拟文件系统（Virtual File System，即 VFS）进行设计和实现。

VFS 子系统是 Linux 内核提供的一个灵活的文件系统框架[61]。在 Linux 操作系统上，所有和文件系统相关的系统调用，均被提交至 VFS 子系统，由 VFS 子系统转发相应的内核模块。实际的文件系统模块可以是 ext4 或 XFS 等基于磁盘存储的文件系统，也可以是 proc 或 sysfs 等提供系统服务的虚拟的文件系统。我们通过实现一个新的虚拟文件系统模块作为中间件，截获 Linux 操作系统中应用程序的文件系统调用，再把其转换为基于纠删码的分布式对象存储系统的读写请求，使得各种 Linux 应用程序可以和使用原生文件系统一样，不做修改地支持分布式文件存储系统。

在基于 Linux 虚拟文件系统的分布式文件系统接口实现中，我们采用了用户空间文件系统（Filesystem in Userspace，即 FUSE）框架，如图 5.1 所示。我们通过 FUSE 在用户态创建了一个分布式文件系统的服务进程，实现文件系统的标准 POSIX 接口，并将其转换为基于纠删码的分布式对象存储系统的读写请求调用。此文件系统主要分为两大部分，文件系统的元数据和文件内容数据。类似于分布式文件系统 Ceph 的方案，我们将元数据存储和文件内容数据存储分离，元数据采取单独的存储池和数据结构进行存储，文件内容数据作为对象存储于分布式对象存储系统中。

在分布式文件系统的元数据存储部分，我们存储的分布式文件系统元数据包含文件的修改时间和权限等属性信息，以及文件在分布式对象存储中的对象索引列表。由于分布式文件系统的元数据在文件的 I/O 操作中需要频繁读写，

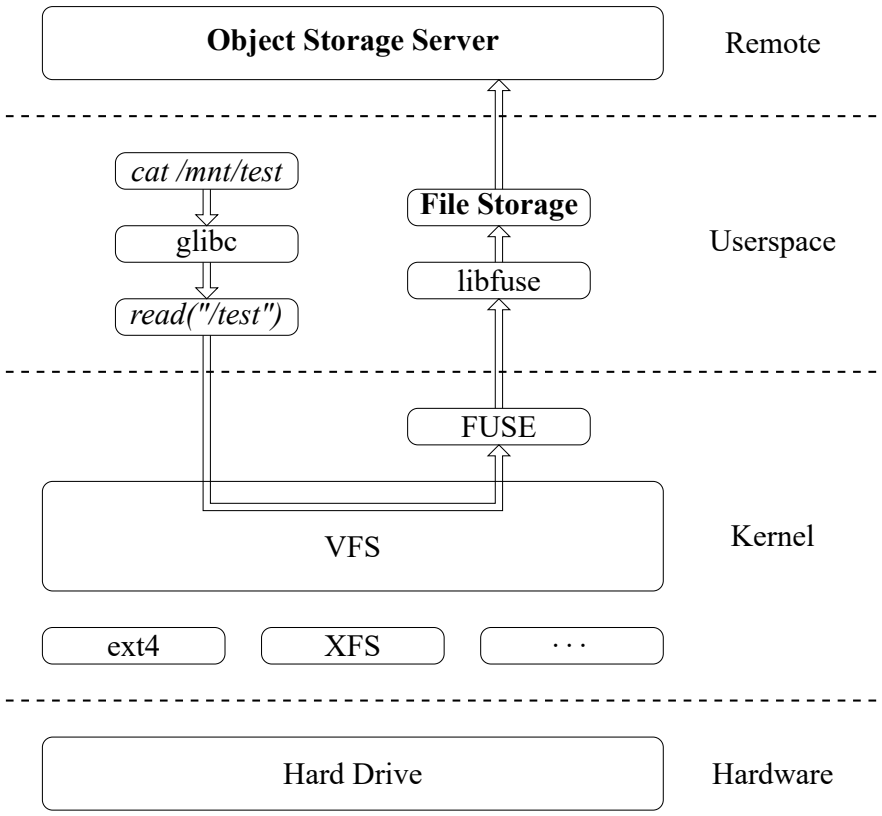


图 5.1　基于 FUSE 框架实现的文件系统架构图

因此在分布式文件系统进行挂载时，服务进程通过对文件系统的元数据进行本地缓存，从而优化文件的读写速度，保障文件系统元数据的快速读写。

在分布式文件系统的文件内容数据存储部分，我们使用修改事件存储文件的内容数据。在文件每次进行创建和修改时，将其新增或修改的内容均单独作为一个对象，给予编号存储于分布式对象存储中。在文件进行读取时，通过寻找读取位置最近的修改记录，确定待读取数据所位于的对象。通过这种方式，我们避免了在文件每次进行修改的重新编码和解码操作，从而提高系统效率。

5.3　基于修改事件的分布式文件修改算法

5.3.1　基于修改事件的文件数据存储结构

对于基于修改事件的分布式文件系统，在文件创建、修改或附加等事件发生时，我们将每个修改事件均描述为一个修改事件。下面我们详细介绍修改事件的存储方式。

1. 随机对象唯一编号的生成方案

首先，我们需要为每个修改事件均分配一个随机的对象唯一编号，在此记为 UUID。由于在文件读取时需要获知文件修改的先后顺序，从而确定最近的

修改记录，因此 UUID 需要包含修改事件的高精度时间。为了保证系统的通用性，我们不妨使用根据 RFC 4122 标准生成的 UUID，其结构如图 5.2 所示。在 UUID 的生成中，我们采用的版本号为 1，变体号为 1，时间戳是一个 60 比特的数据，记录从协调世界时（Coordinated Universal Time，即 UTC）1582 年 10 月 15 日 00 时 00 分 00 秒（即首次采用格里高利历的日期）至今所经过的 100 纳秒数，时钟序列是一个 14 比特的数据，在系统时间戳倒退时进行相应的增加，节点则是一个 48 比特的数据，存储当前节点编号，通常使用网卡的物理地址（Media Access Control Address，即 MAC 地址）。

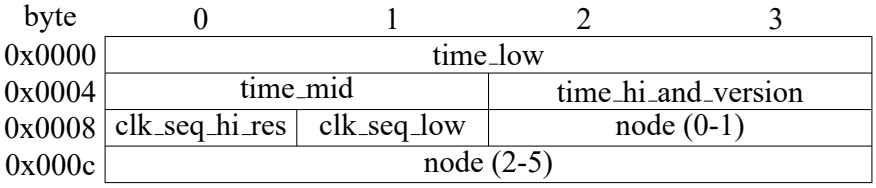
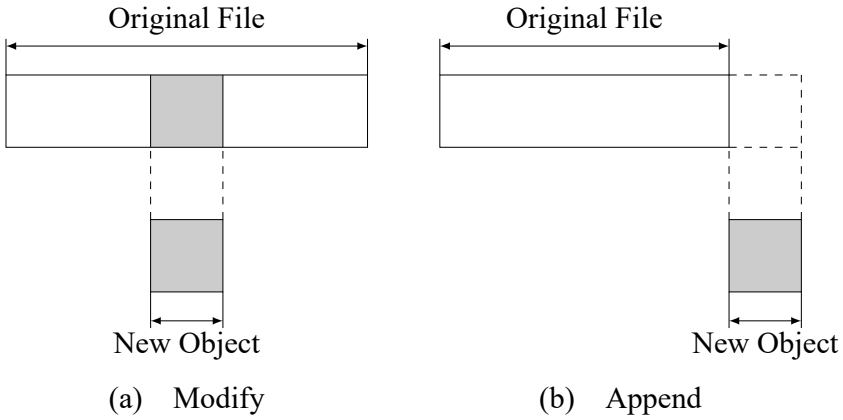


图 5.2　对象唯一编号结构示意图

2. 文件修改对象示意

在按照 RFC 4122 标准生成 UUID 作为对象的唯一编号之后，我们将修改的后的文件内容作为单独的对象，以生成的唯一编号 UUID 作为索引，存储于基于纠删码的对象存储系统中。对于文件修改和文件附加操作，存储的对象相对于原始文件示意如图 5.3 所示。



(a)　Modify　　(b)　Append

图 5.3　修改事件对象示意

3. 文件元数据的存储结构

对于存储的文件修改对象，我们需要在分布式文件系统的元数据中对其进行索引。我们首先需要在元数据中存储此次修改事件内容的编号 UUID，从而记录文件变更后的内容。此外，我们还需要存储此变更所参照的原始文件，即

编号 UUID_{prev}，从而确定文件修改的参照点。我们还需要记录此修改或者附加的起始位置 start = offset 和终止位置 end = offset + length，其中 length 为编号 UUID 所对应的对象大小。我们采用如图 5.4 的结构存储这些数据，从而得到一个完整的文件修改记录，并计算文件的最新内容。

byte	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
0x0000	UUID	
0x0010	UUID_{prev}	
0x0020	start = offset	end = offset + length
0x0030	UUID	
0x0040	UUID_{prev}	
0x0050	start = offset	end = offset + length
0x0060	···	

图 5.4　分布式文件系统的元数据结构

在读取文件时，我们可以遍历这个元数据结构，选择所有 UUID 编号中时间戳和时钟序列最新的对象，来确定当前最新版本的文件。若文件待读取的位置不在上述 UUID 对应记录的 start 和 end 之间，则递归地按上述步骤读取 UUID_{prev} 对应的记录，直至找到符合条件的记录，并返回符合条件的记录的 UUID 对应的对象中的相应位置。此算法流程图如 5.5 所示。

4. 文件的版本历史和并发写入

通过上述文件修改事件结构，我们可以维护一个文件的历史版本记录，在文件打开操作时，我们确定一个最新的文件版本，在此后对该文件的操作中，均针对此版本的文件进行修改，因此可以保证并发的写入请求不发生冲突。如果存在多个进程同时操作同一个文件，便会在文件系统中同时存储该文件的多个版本，文件系统将选择最新修改的文件版本。

5.3.2　文件修改事件元数据的无锁更新算法

在元数据的修改过程中，为了保证数组的一致性，若有多个线程同时对数组进行更新，则可能导致数据的不一致。为了防止对共享数据的非法访问，一般可以采用自旋锁等结构对并发访问进行限制。但采用锁机制会产生额外的性能开销，为了降低修改事件算法的性能开销，由于我们的数据结构和访问模式较为固定，在此处我们采用基于数组的无锁队列进行元数据的更新。

1. 无锁队列的结构

在基于数组的无锁队列中，我们采用数组下标 $read\_index$ 和 $write\_index$ 来分别表示当前数组的读下标和写下标。在数据初始化时，这两个下标均初始化为 0，即数据的第一个元素位置。在数据插入时，我们使用原子操作，将

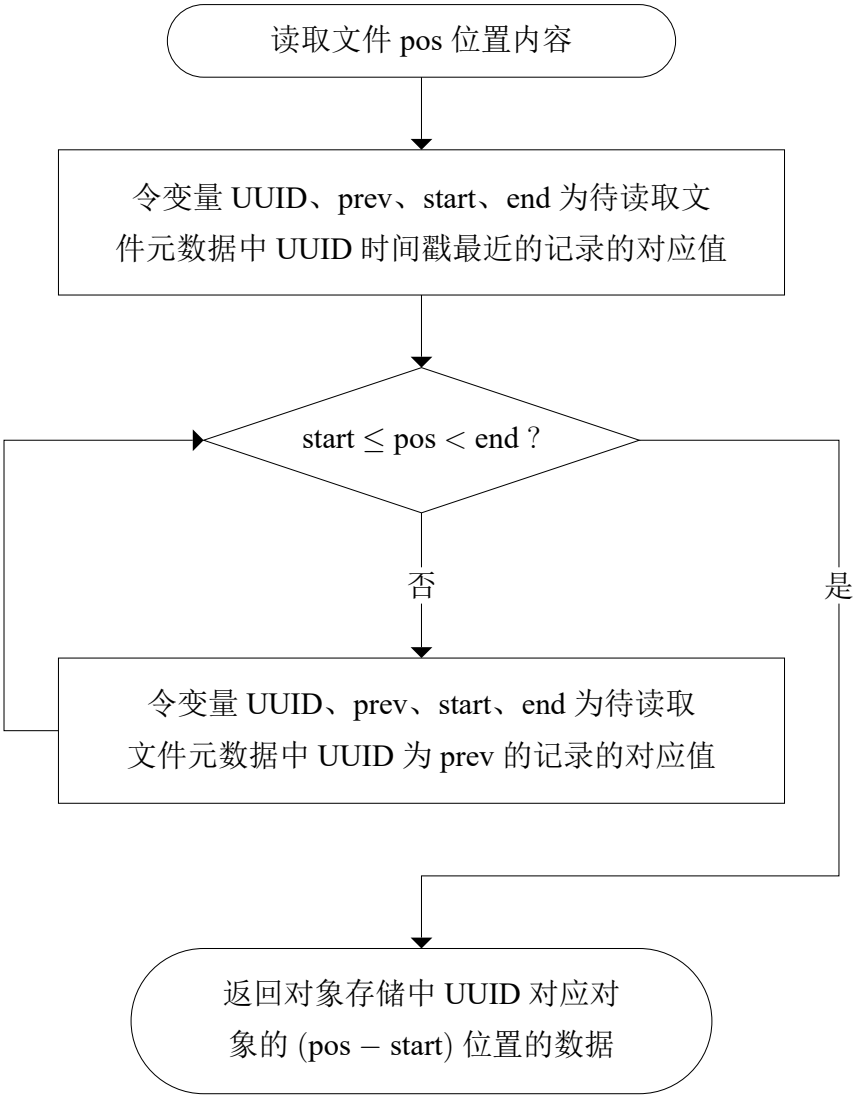


图 5.5　文件读取流程图

$write\_index$ 加 1，并返回原来的 $write\_index$ 值作为写入位置 $write\_current$。之后我们便可以向数组的 $write\_current$ 位置写入数据。在数据写入完成后，我们使用原子操作，比较若 $read\_index$ 与 $write\_current$ 相同，则将 $read\_index$ 改为 $write\_current + 1$，否则不进行任何操作。若执行此步骤后 $read\_index \leq write\_current$，则重新执行上面一步，直至 $read\_index > write\_current$。整个数据插入过程如图 5.6 所示。

2. 无锁队列的并发插入

通过采用此无锁队列，我们在更新对象元数据时，可以直接将修改对象的编号等信息插入到元数据末端。在多个进程进行并发写入时，此队列可以做到无锁的并发写入，如图 5.7 所示，在一个进程完成数据写入之前，又有另一个进程申请写入数据。对于这种情况，基于数组的无锁队列中的 $write\_current$ 向前继续移动一个元素，然后两个进程可以分别在对应位置写入，不会造成冲突。

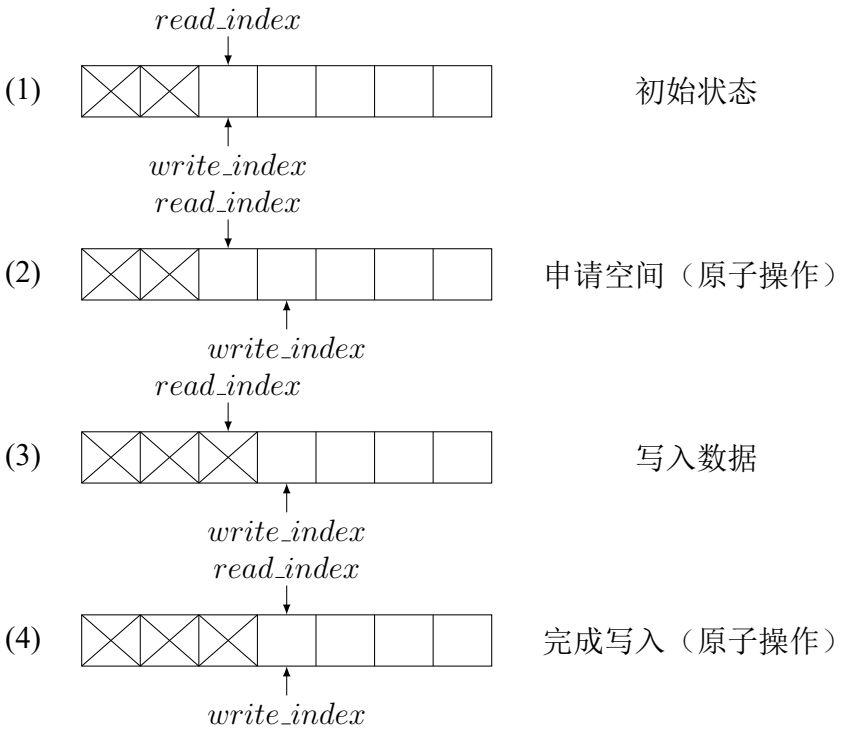


图 5.6　无锁队列的数据插入示意

在数据写入完成之后，在对 $read\_index$ 递增时，第二个写入进程必须等待第一个写入进程完成之后才能进行递增，因为我们必须保证数据完全被写入后才能被读取。基于此无锁队列的元数据存储在读取文件时，即进行元数据遍历等读取操作时，可以读取的数据便是无锁队列中已经完成写入的数据，也就是从上述队列头部开始，到 $read\_index$ 前的位置为止之间的数据。

5.3.3　文件修改事件对象的合并算法

基于文件修改对象的存储方式虽然能加速文件的写入速度和并行度，但当文件修改次数较多时，文件读取时会带来额外的计算需求，速度将无法保证，且需要更多的存储空间。在大部分情况下，我们不需要文件的修改历史，仅需要最新版本的文件内容。因此我们计划在系统空闲时，通过对修改历史较多的对象进行重新构建，防止在数据读取时需要遍历过多版本的数据。

1. 对象合并的方式

通过扫描文件系统中文件的未合并对象数，我们将未合并对象数大于一定阈值的文件进行整文件读取，然后将读取的对象作为一个全新的对象进行存储。在更换对象时，我们通过比对最新的记录与合并前的记录是否相同，确保对象在被读取到被替换这段时间未被修改，从而做到文件的无锁更新，并避免在更新过程中修改文件的操作导致对象不一致现象的产生。

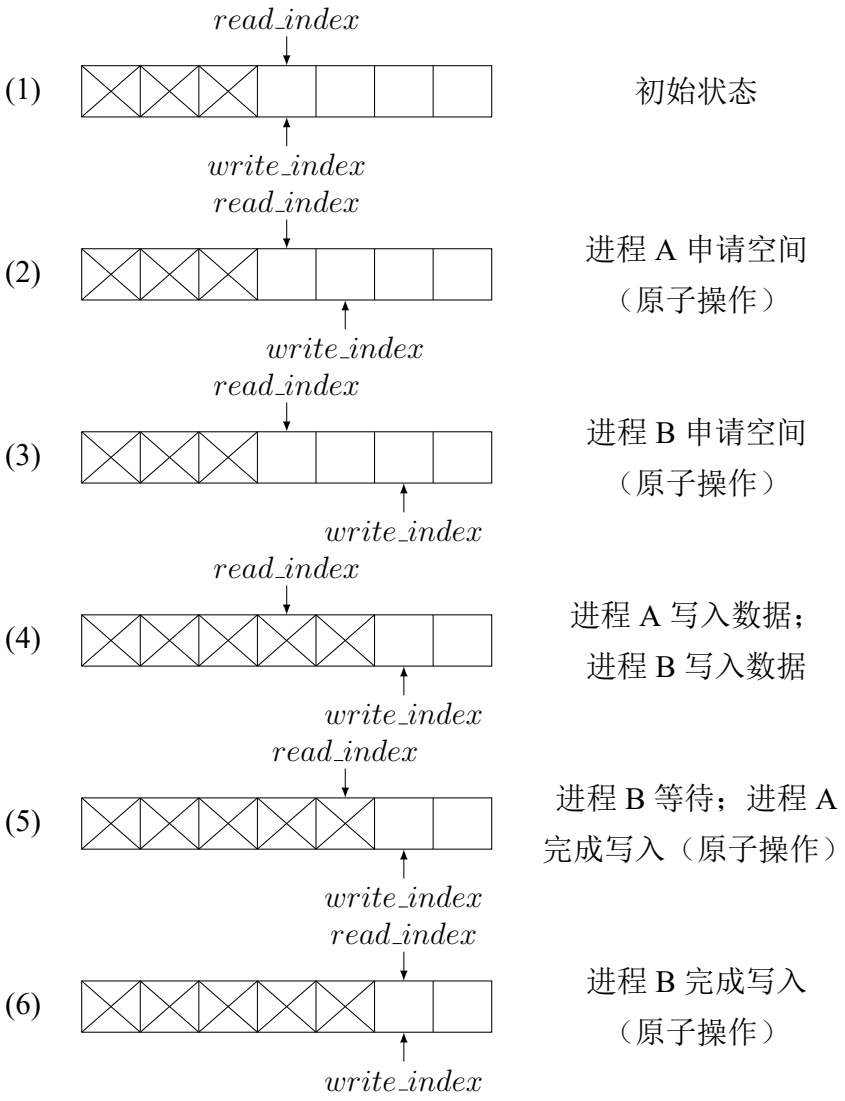


图 5.7　无锁队列中多进程同时写入示意

我们可以建立一个可删除对象列表，在每次合并算法执行过后，将早于合并算法执行时间的对象编号存储于此列表中，然后选取一个阈值，若磁盘占用率大于规定阈值，则自动清理可删除对象列表中的对象。在用户手动执行删除操作时，我们亦可以清理此可删除对象列表中的对象，从而避免空间浪费。

2. 对象合并的执行时机

由于文件修改事件的合并和回收算法均需要耗费较长的时间，我们希望此算法仅在系统空闲时才会执行。我们计划通过检测后端对象存储的读写时间 t_f 和实际的对象存储带宽 b_w，来获知后端存储的使用情况。在检测到后端存储空闲时，再进行数据写入，即：

- 根据对象存储的读写时间 t_f 设定阈值 t'_f，在 $t_f < t'_f$ 时发送数据整合请求，直至系统的 $t_f \geq t'_f$ 为止

- 根据系统的对象存储带宽 b_w 和最大带宽 $\max(b_w)$ 设定阈值 b'_w，$0 < b'_w < \max(b_w)$，在 $b_w < b'_w$ 时发送数据整合请求，直至系统的 $b_w \geq b'_w$ 为止

这两种方案均设定了一个硬阈值来代表系统是否空闲，但在实际应用中，系统的负载情况是动态变化的，因此此方案过于依赖阈值的设置，容易造成存储性能浪费或数据整合请求堆积。因此，我们可以通过系统状态动态地估算阈值，以此来平衡系统的读性能和写性能。我们定义对象 f 的写入列表的集合为 $S_w(f)$，数据整合的带宽需求为：

$$b(f) = \sum_{w \in S_w(f)} length(data)_w \tag{5.3}$$

全部对象的带宽需求总和为：

$$b_f = \sum_{f \in \{f \mid \|S_w(f)\| \neq 1\}} b(f) \tag{5.4}$$

如果考虑写入事件数为 $n_w = \|S_w(f)\|$，显然数据整合请求的优先级为 n_w 越高，对读性能的影响越大，因此数据整合的优先级越高，定义优先级可以表示为函数 $priority(n_w)$，因此定义：

$$b'_f = \sum_{f \in \{f \mid \|S_w(f)\| \neq 1\}} priority(\|S_w(f)\|) * b(f) \tag{5.5}$$

因此阈值应为函数 $t'_f = t'_f(b'_f)$ 和 $b'_w = b'_w(b'_f)$。我们需要平衡不同 $priority(n_w)$ 和 $t'_f(b'_f)$ 以及 $b'_w(b'_f)$ 的性能表现，从而动态平衡系统的读性能和写性能。

5.4　实验与分析

5.4.1　实验环境

在本章的实验中，我们继续采用如表 4.1 和图 4.1 所示的基于 KVM 虚拟化技术的由 37 个节点组成的分布式文件系统实验环境。在此分布式文件系统实验环境中，我们在上文的基于 On-demand ARECS 算法的 Tahoe-LAFS 分布式文件系统中，实现了基于修改事件的文件存储算法，并针对文件的修改和文件的附加两种操作，分别对算法进行了测试。

5.4.2　文件修改实验结果与分析

在文件修改操作的测试过程中，我们随机生成了一个大小为 50 MB 的文件作为未修改前的原始文件，然后让文件系统每次对该文件中的任意 10 MB 内容

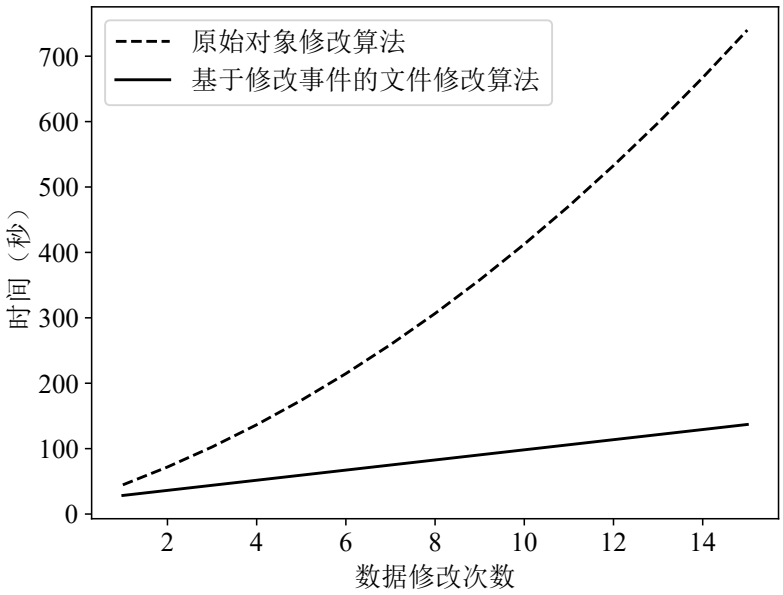


图 5.8　算法的执行时间随修改事件数变化

进行随机修改，测量并记录得到原始文件修改算法相比于我们的算法的总执行时间随修改数变化如图 5.8 所示。

通过图 5.8 可以看出，使用基于修改事件的文件存储算法，在文件修改操作中由于避免了文件的解码操作，并减少了文件重复部分的编码操作，从而大大降低了文件修改所需要的时间，对于进行 1 到 15 次的文件修改操作来说，将平均存储时间降低到了原来的 31%。

5.4.3　文件附加实验结果与分析

在文件附加操作的测试过程中，我们创建一个空的新文件，然后对该文件进行多次附加操作，文件的每次附加操作在文件末尾追加 10 MB 的随机内容，测量并记录得到原始文件修改算法相比于我们的算法的执行时间随附加事件数变化如图 5.9 所示。

通过图 5.9 可以看出，在文件附加操作中使用基于修改事件的文件存储算法，由于基本避免了原有文件的编码和解码操作，在大文件追加时相比于直接修改对象来说性能有很大提高。在小文件追加时，由于即使对象存储引擎对于原有对象进行了纠删码的解码与重新编码，由于数据量不大，因此性能提高幅度较小。总体来说，在进行 1 到 20 次附加操作的实验中，基于修改对象的文件存储算法将文件存储时间平均降低到了原来的 41%。

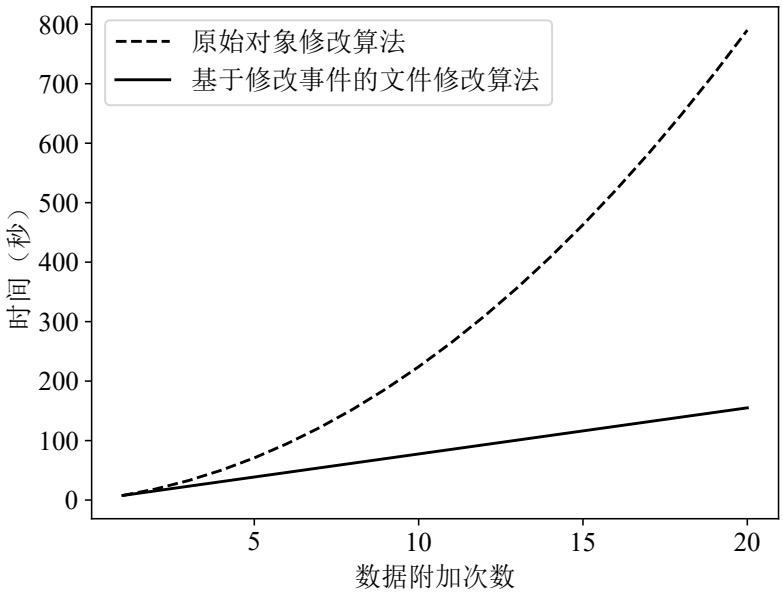


图 5.9　算法的执行时间随附加事件数变化

5.5　小结

在本章中，我们首先介绍了基于纠删码的对象存储及其物理存储方式，然后介绍了在 Linux 系统中通过 FUSE 技术实现一个基于对象存储的分布式文件系统中间件的方法。针对基于纠删码的对象存储在对象修改时需要解码和重新编码，效率低下的特点，我们采用修改事件表示文件修改操作，并介绍了相应的文件存储和读取算法。针对文件存储时元数据频繁更新的性能瓶颈问题，我们通过引入基于数组的无锁队列，避免元数据写入时锁的开销，提高系统的并发性能。针对多次修改后的文件读取时遍历比较次数过多的问题，我们介绍了在系统空闲时间对修改事件进行无锁合并的策略。我们在基于 On-demand ARECS 算法的 Tahoe-LAFS 分布式文件系统上实现了此修改算法并进行了对比实验，实验结果表明，基于修改事件的分布式文件存储算法相比于直接修改原始对象，在纠删码对象存储引擎上的文件修改和附加操作平均时间降低到了原来的 $30\% \sim 40\%$。

第 6 章　结论与展望

6.1　本文工作总结

随着计算机处理的数据量不断攀升，集中式存储系统逐渐被分布式存储系统所取代。然而在分布式存储系统中，由于采用大量低成本的节点共同进行数据的存储，因此节点发生损坏的风险大大增加，数据的冗余备份与恢复便成为了目前分布式存储系统研究的热点问题之一。本文通过对现有的分布式文件系统中的数据冗余和备份方案做出总结，明确了基于纠删码的数据冗余方案相比于多副本备份方案可以大大降低数据冗余度。紧接着通过对纠删码编码算法和纠删码在分布式存储系统中研究成果的分析，针对基于纠删码的分布式文件系统中编码策略方面和数据修改方面存在的问题，提出了一种基于纠删码和对象存储的分布式文件系统的实现方案，此方案主要分为以下两大部分：

针对不同数据具有不同可用性，且不同磁盘具有不同可靠性的特点，我们通过改进当前分布式文件系统中纠删码编码方案均为事先确定的方案，提出了 On-demand ARECS 算法，动态地选择编码策略和存储节点。On-demand ARECS 算法首先通过磁盘的 S.M.A.R.T 信息得到磁盘的重分配扇区数，并根据磁盘重分配扇区数的统计值估计磁盘的故障率。通过对存储于分布式存储系统中对象的组合情况分析，我们进一步推导得到了对应的对象可用性。针对数据中心中损坏磁盘的替换与数据重建，我们通过在模型中增加数据重建时间进行反应。在满足基于对象重要性确定的可用性约束的磁盘组中，我们通过最小化数据传输和存储延迟的方法动态选取最优的存储磁盘组。针对大文件直接进行纠删码编码时编码矩阵过大导致的运算效率低下的问题，我们提出了数据切分为多块分别存储时的节点选择模型。我们在 Tahoe-LAFS 分布式文件系统上实验了 On-demand ARECS 算法，相对于目前的纠删码编码策略，On-demand ARECS 算法将存储空间冗余平均降低了 18%，并将存储系统的文件传输时间降低了 46%，大大提高了分布式存储系统的存储空间利用率，并改善了分布式存储系统的存储效率。

针对基于纠删码的分布式存储系统在修改数据时需要重新编解码，导致数据修改的代价较高，难以用于可读写的数据存储的特点，本文提出了使用修改事件代替直接修改对象的方法。我们介绍了基于纠删码的分布式对象存储系统的存储结构，并设计了一个基于分布式对象存储的文件系统中间件，通过将修改事件内容单独存储为修改数据对象，并在分布式文件系统的元数据中对其进行索引的方法减少数据的解码和重新编码运算。针对元数据可能存在频繁读写

的情况，我们设计了基于数组的无锁队列结构对元数据进行更新，大大提高了系统的并发性能。针对此方法在数据存在大量修改记录的极端情况中，读取时最坏情况遍历比较次数可能过多的特点，设计了在系统空闲时进行对象数据合并的算法。我们在基于 On-demand ARECS 算法的 Tahoe-LAFS 分布式文件系统上实验了此文件修改算法，结果表明，基于修改事件的分布式文件存储算法大大降低了基于纠删码引擎的分布式文件系统中修改或附加数据所用的时间，数据的修改和附加操作平均耗时仅为原来的 30% ～ 40%，提升了基于纠删码的分布式文件系统在通用存储领域的可用性。

6.2　未来研究展望

尽管在分布式文件系统中测试的实验数据表明，上述方法在基于纠删码对象存储的分布式文件系统中的文件存储速度方面，相比于传统的方法有了很大改善，但距离基于纠删码的分布式存储系统完全替代基于多副本备份的分布式存储系统的目标，还有许多问题需要解决。我们通过总结，认为在当前基于纠删码的分布式文件系统中，可以通过以下几个方面加以改进：

一是在数据损坏时的数据重建问题。我们目前的研究主要基于数据存储时的性能开销，在面对存储设备损坏时，基于纠删码的存储方案需要通过矩阵计算恢复原始数据，相比于多副本备份方案，需要对原始数据进行解码和对冗余数据进行编码，进而消耗更多的网络带宽资源和节点计算资源。由于在分布式存储系统中节点损坏事件时有发生，数据重建所占用的计算和网络资源在数据中心中不能被忽视。未来的研究如果可以通过减小纠删码编码和解码的开销或优化重建传输策略等方式，更加高效地进行基于纠删码存储的分布式文件系统的数据重建，从而更大限度地减少数据中心内部的数据传输和节点的计算，将可以使极大地降低基于纠删码的分布式存储系统的额外性能开销。

二是纠删码编码和解码的开销问题。虽然目前的计算机系统已经能有效地计算纠删码的编码和解码，但相比于多副本备份的直接读取，对于热点数据的多次编码和解码还是有着较大的性能开销。此额外开销主要表现在数据写入时进行冗余数据编码的计算开销，在数据读取时通过冗余数据进行原始数据恢复的开销，以及在存储设备损坏时数据重建时原始数据的解码以及可能的冗余数据编码的开销等等。因此，研究如何通过改进纠删码编码方案，设计新的纠删码编码矩阵减少计算量，增加算法并行度并通过 GPU 或 ASIC 专用硬件等方案进行大规模的并行计算，实现更低代价的纠删码编码和解码，从而大大提升基于纠删码编码的分布式文件系统的数据写入、读取和重建性能，仍是分布式存储系统需要研究的重要问题。

三是磁盘故障率的有效估计问题。本文基于估计的磁盘故障率进行后续的编码方案选择，但通过引入硬盘 S.M.A.R.T. 信息，读取磁盘的重分配扇区数，并通过数据中心中大量磁盘的重分配扇区数统计量来估算磁盘的故障率的方法可能并不够准确。由于磁盘的重分配扇区数仅仅是反映硬盘健康程度的指标之一，这种估计的置信度可能并不够高，若能综合通过磁盘多种维度的信息进行磁盘故障率的估计，则磁盘故障率的估计值能更加准确。此外，部分存储介质可能不存在重分配扇区的操作，如何进行这部分介质的故障率估计也是需要研究的问题之一。若能更加准确的预计磁盘发生损坏的概率，则可以使本文估算的数据可用性置信度更高，进而提升存储空间利用率。

参 考 文 献

[1] BRADSHAW L. Big data and what it means[J/OL]. Business Horizon Quarterly, 2013(7): 32-35. https://www.uschamberfoundation.org/bhq/big-data-and-what-it-means.

[2] LI J, LI B. Erasure coding for cloud storage systems: a survey[J]. Tsinghua Science and Technology, 2013, 18(3):259-272.

[3] SHVACHKO K, KUANG H, RADIA S, et al. The Hadoop distributed file system[C]//2010 IEEE 26th Symposium on Mass Storage Systems and Technologies. Piscataway: IEEE, 2010: 1-10.

[4] WEIL S A, BRANDT S A, MILLER E L, et al. Ceph: a scalable, high-performance distributed file system[C]//Proceeding of the 7th Conference on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2006: 307-320.

[5] CHEN Y, ZHOU Y, TANEJA S, et al. aHDFS: an erasure-coded data archival system for Hadoop clusters[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(11): 3060-3073.

[6] XIE X, WU C, GU J, et al. AZ-code: an efficient availability zone level erasure code to provide high fault tolerance in cloud storage systems[C]//2019 35th Symposium on Mass Storage Systems and Technologies. Piscataway: IEEE, 2019: 230-243.

[7] WEI B, XIAO L M, WEI W, et al. A new adaptive coding selection method for distributed storage systems[J]. IEEE Access, 2018, 6:13350-13357.

[8] HADDOCK W, CURRY M L, BANGALORE P V, et al. GPU erasure coding for campaign storage[C]//High Performance Computing. Berlin: Springer, 2017: 145-159.

[9] DRUCKER N, GUERON S, KRASNOV V. The comeback of Reed Solomon codes[C]//2018 IEEE 25th Symposium on Computer Arithmetic. Piscataway: IEEE, 2018: 125-129.

[10] LI X, LI R, LEE P P, et al. OpenEC: toward unified and configurable erasure coding management in distributed storage systems[C]//17th USENIX Conference on File and Storage Technologies. Berkeley: USENIX Association, 2019: 331-344.

[11] XIANG Y, LAN T, AGGARWAL V, et al. Joint latency and cost optimization for erasure-coded data center storage[J]. IEEE/ACM Transactions on Networking, 2015, 24(4):2443-2457.

[12] KADEKODI S, RASHMI K, GANGER G R. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity[C]//17th USENIX Conference on File and Storage Technologies. Berkeley: USENIX Association, 2019: 345-358.

[13] ABEBE M, DAUDJEE K, GLASBERGEN B, et al. EC-Store: bridging the gap between storage and latency in distributed erasure coded systems[C]//2018 IEEE 38th International Conference on Distributed Computing Systems. Piscataway: IEEE, 2018: 255-266.

[14] LIU C, WANG Q, CHU X, et al. G-CRS: GPU accelerated Cauchy Reed-Solomon coding[J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 29(7):1484-1498.

[15] FAN D, XIAO F, TANG D. A new erasure code decoding algorithm[J]. International Journal of Network Security, 2019, 21(3):522-529.

[16] ZHOU T, TIAN C. Fast erasure coding for data storage: a comprehensive study of the acceleration techniques[C]//17th USENIX Conference on File and Storage Technologies. Berkeley: USENIX Association, 2019: 317-329.

[17] LI S, LU Y, SHU J, et al. LocoFS: a loosely-coupled metadata service for distributed file systems[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2017: 1-12.

[18] CHEN X, LIU J, XIE P. Erasure code of small file in a distributed file system[C]//2017 3rd IEEE International Conference on Computer and Communications. Piscataway: IEEE, 2017: 2549-2554.

[19] RASHMI K, SHAH N B, GU D, et al. A solution to the network challenges of data recovery in erasurecoded distributed storage systems: a study on the Facebook warehouse cluster[C]// 5th USENIX Workshop on Hot Topics in Storage and File Systems. Berkeley: USENIX Association, 2013.

[20] BAIRAVASUNDARAM L N, ARPACI-DUSSEAU A C, ARPACI-DUSSEAU R H, et al. An analysis of data corruption in the storage stack[J]. ACM Transactions on Storage, 2008, 4(3): 1-28.

[21] XIE P, YUAN Z, HUANG J, et al. N-code: an optimal RAID-6 MDS array code for load balancing and high I/O performance[C]//Proceedings of the 48th International Conference on Parallel Processing. New York: ACM, 2019: 1-10.

[22] PLANK J S. The RAID-6 liberation codes[C]//6th USENIX Conference on File and Storage Technologies. Berkeley: USENIX Association, 2008: 97-110.

[23] FENG J, CHEN Y, SUMMERVILLE D. EEO: an efficient MDS-like RAID-6 code for parallel implementation[C]//2010 IEEE Sarnoff Symposium. Piscataway: IEEE, 2010: 1-5.

[24] JIN C, JIANG H, FENG D, et al. P-code: A new RAID-6 code with optimal properties[C]// Proceedings of the 23rd International Conference on Supercomputing. New York: ACM, 2009: 360-369.

[25] LUO X, SHU J. Summary of research for erasure code in storage system[J]. Journal of Computer Research and Development, 2012, 49(1):1-11.

[26] YUAN W, YU Y, GAO Y, et al. Research on multi-fault-tolerant MDS array erasure code[C]// Journal of Physics: Conference Series: volume 1237. Bristol: IOP Publishing, 2019: 022015.

[27] BLAUM M, BRADY J, BRUCK J, et al. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures[J]. IEEE Transactions on Computers, 1995, 44(2):192-202.

[28] CORBETT P, ENGLISH B, GOEL A, et al. Row-diagonal parity for double disk failure correction[C]//3rd USENIX Conference on File and Storage Technologies. Berkeley: USENIX Association, 2004: 1-14.

[29] HUANG C, XU L. STAR: an efficient coding scheme for correcting triple storage node failures [J]. IEEE Transactions on Computers, 2008, 57(7):889-901.

[30] XIANG L, XU Y, LUI J C, et al. A hybrid approach to failed disk recovery using RAID-6 codes: algorithms and performance evaluation[J]. ACM Transactions on Storage, 2011, 7(3): 1-34.

[31] XU L, BRUCK J. X-code: MDS array codes with optimal encoding[J]. IEEE Transactions on Information Theory, 1999, 45(1):272-276.

[32] HAFNER J L. WEAVER codes: highly fault tolerant erasure codes for storage systems[C]// 4th USENIX Conference on File and Storage Technologies. Berkeley: USENIX Association, 2005: 211-224.

[33] SHEN Z, SHU J, FU Y. HV code: an all-around MDS code for RAID-6 storage systems[J]. IEEE Transactions on Parallel and Distributed Systems, 2015, 27(6):1674-1686.

[34] GALLAGER R. Low-density parity-check codes[J]. IRE Transactions on Information Theory, 1962, 8(1):21-28.

[35] KUMAR V A, NANDALAL V. A detailed study on LDPC encoding techniques[J]. Wireless Communication Technology, 2018, 2(2):1.

[36] MACKAY D J, NEAL R M. Near shannon limit performance of low density parity check codes[J]. Electronics letters, 1997, 33(6):457-458.

[37] TANNER R. A recursive approach to low complexity codes[J]. IEEE Transactions on Information Theory, 1981, 27(5):533-547.

[38] REED I S, SOLOMON G. Polynomial codes over certain finite fields[J]. Journal of the Society for Industrial and Applied Mathematics, 1960, 8(2):300-304.

[39] ROTH R M, LEMPEL A. On MDS codes via Cauchy matrices[J]. IEEE Transactions on Information Theory, 1989, 35(6):1314-1319.

[40] ARAFA Y, BARAI A, ZHENG M, et al. Evaluating the fault tolerance performance of HDFS and Ceph[C]//Proceedings of the Practice and Experience on Advanced Research Computing. New York: ACM, 2018: 1-3.

[41] AGHAYEV A, WEIL S, KUCHNIK M, et al. File systems unfit as distributed storage backends:

lessons from 10 years of Ceph evolution[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. New York: ACM, 2019: 353-369.

[42] PLANK J S, BLAUM M. Sector-disk (SD) erasure codes for mixed failure modes in RAID systems[J]. ACM Transactions on Storage, 2014, 10(1):1-17.

[43] ZHANG X, CAI Y, LIU Y, et al. NADE: nodes performance awareness and accurate distance evaluation for degraded read in heterogeneous distributed erasure code-based storage[J]. The Journal of Supercomputing, 2019:1-30.

[44] GUTIERREZ F O, GARCIA V C, CARDOSO J F S, et al. uStorage - a storage architecture to provide block-level storage through object-based storage[C]//Service-Oriented and Cloud Computing. Berlin: Springer, 2017: 213-228.

[45] ANDRONIKOU V, MAMOURAS K, TSERPES K, et al. Dynamic QoS-aware data replication in grid environments based on data “importance”[J]. Future Generation Computer Systems, 2012, 28(3):544-553.

[46] HUANG C, SIMITCI H, XU Y, et al. Erasure coding in Windows Azure Storage[C]//2012 USENIX Annual Technical Conference. Berkeley: USENIX Association, 2012: 15-26.

[47] AMAZON. Amazon S3 FAQs[EB/OL]. 2020. https://aws.amazon.com/s3/faqs/.

[48] MICROSOFT. Azure Storage redundancy[EB/OL]. 2020. https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy.

[49] MA A, TRAYLOR R, DOUGLIS F, et al. RAIDShield: characterizing, monitoring, and proactively protecting against disk failures[J]. ACM Transactions on Storage, 2015, 11(4): 1-28.

[50] ALLEN B. Monitoring hard disks with smart[J]. Linux Journal, 2004, 2004(117):9.

[51] XU Y, SUI K, YAO R, et al. Improving service availability of cloud systems by predicting disk error[C]//2018 USENIX Annual Technical Conference. Berkeley: USENIX Association, 2018: 481-494.

[52] MAHDISOLTANI F, STEFANOVICI I, SCHROEDER B. Proactive error prediction to improve storage system reliability[C]//2017 USENIX Annual Technical Conference. Berkeley: USENIX Association, 2017: 391-402.

[53] LU S, LUO B, PATEL T, et al. Making disk failure predictions SMARTer![C]//18th USENIX Conference on File and Storage Technologies. Berkeley: USENIX Association, 2020: 151-167.

[54] DOS SANTOS LIMA F D, AMARAL G M R, DE MOURA LEITE L G, et al. Predicting failures in hard drives with LSTM networks[C]//2017 Brazilian Conference on Intelligent Systems. Piscataway: IEEE, 2017: 222-227.

[55] CHAVES I C, DE PAULA M R P, LEITE L G, et al. Hard disk drive failure prediction method

based on a Bayesian network[C]//2018 International Joint Conference on Neural Networks. Piscataway: IEEE, 2018: 1-7.

[56] BOTEZATU M M, GIURGIU I, BOGOJESKA J, et al. Predicting disk replacement towards reliable data centers[C]//Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. New York: ACM, 2016: 39-48.

[57] ANANTHARAMAN P, QIAO M, JADAV D. Large scale predictive analytics for hard disk remaining useful life estimation[C]//2018 IEEE International Congress on Big Data. Piscataway: IEEE, 2018: 251-254.

[58] PINHEIRO E, WEBER W D, BARROSO L A. Failure trends in a large disk drive population[C]//5th USENIX Conference on File and Storage Technologies. Berkeley: USENIX Association, 2007.

[59] LIU Z. Implementation of QoE optimization design for real time streaming media system[D]. Hefei: University of Science and Technology of China, 2018.

[60] WILCOX-O'HEARN Z, WARNER B. Tahoe: the least-authority filesystem[C]//Proceedings of the 4th ACM International Workshop on Storage Security and Survivability. New York: ACM, 2008: 21-26.

[61] WETZEL A W, HOOD G, ROPELEWSKI A J. A virtual filesystem approach to storage, analysis and delivery of volumetric image data for connectomics[C]//2017 IEEE Applied Imagery Pattern Recognition Workshop. Piscataway: IEEE, 2017: 1-4.

致　谢

时光如白驹过隙，转眼间三年的研究生生涯即将结束，也意味着在科大七年的求学生涯接近尾声，其间有汗水，有挫折，有快乐，更有成长。在即将毕业离校之际，谨书此文，以感谢各位在成长路上的一路相随。

首先，我要特别感谢我的导师邢凯老师。邢老师不仅知识渊博，而且为人亲切，无论是在科研上还是生活上，均给予了我无微不至的指导和关怀。邢老师的辛勤培养不仅丰富了我的知识，更教会了我分析问题的方法和解决问题的思路，使我受益匪浅。从毕业论文的最初选题到最终定稿，邢老师均给予了我很多有价值的建议。在此，衷心地祝愿邢老师身体健康、工作顺利、阖家幸福！

其次，我要感谢我的母校中国科学技术大学。科大不仅为我们提供了优越的学习条件、科研氛围和生活环境，更为我们提供了一个优秀的平台，让我们见识到更加广阔的世界。在此，衷心地祝福母校越办越好，也祝愿学校的全体师生万事如意！

此外，我要感谢实验室的各位师兄、师姐、师弟和师妹们。特别地，我要感谢龚海华师兄对我的科研和生活等各方面均给予了无微不至的指导、帮助和关照；我还要感谢张孟涵同学在我研究生期间的陪伴和支持。在此，衷心地祝愿你们未来身体健康、前程似锦！

最后，我要感谢我的爸爸妈妈。感谢你们的无私关爱与支持，祝你们永远健康快乐！

在读期间发表的学术论文与取得的其他研究成果

已发表论文：

[1] GONG H, XING K, **LI Z**, et al. A new ranking based semantic hashing method for deep image retrieval[C]//Journal of Physics: Conference Series. Bristol: IOP Publishing, 2019, 1229(1): 012004.

已录用论文：

[1] **LI Z**, XING K, GONG H. Design of an adaptive erasure code storage strategy oriented to dynamic requirements of data availability and storage reliability[J]. Journal of Chinese Computer Systems, Accepted.