重庆大学本科学生毕业论文（设计）

基于 OpenHarmony 的分布式文件去重系统设计与实现

学　　生：李为

学　　号：20194303

指导教师：陈咸彰

专　　业：计算机科学与技术

重庆大学计算机学院

2023 年 6

Graduation Thesis (Design) of Chongqing University

Design and Implementation of Distributed File Deduplication System Based on OpenHarmony

Undergraduate: Wei Li

Supervisor: Associate Prof. Xianzhang Chen

Major: Computer Science and Technology

College of Computer Science

Chongqing University

June 202

摘　要

在分布式文件系统中，数据重复是一个常见的问题。由于分布式系统中可能存在多个节点和存储设备，不同节点上的文件可能会出现重复的情况。这种数据重复导致了存储资源的浪费，增加了数据的冗余性，并且可能降低系统性能。因此，本文设计并实现了基于 OpenHarmony 平台的分布式文件去重系统，目的是解决分布式中重复数据带来的存储开销问题。

为此，本文从分布式数据去重架构、文件远程读写设计和系统性能优化的角度展开研究。对比之前的研究工作，本课题致力于将 OpenHarmony 系统与分布式数据去重系统融合，充分利用国产操作系统完备的开发技术优势以降低设备存储开销。具体研究内容如下：

（1）针对分布式去重中心化架构的研究。提出带中心节点的分布式去重算法，通过中心节点存储文件全局唯一的指纹来定位文件，以达到重复文件删除和文件重定向的目的。

（2）针对分布式系统中远程文件读写交互的研究。利用库打桩机制拦截 C 库标准文件读写接口调用，使用多线程机制为读写接口添加网络访问功能，结合重定向表实现设备与设备之间文件的远程读写访问。

（3）针对系统网络传输优化的研究。使用 Socket 多线程方式实现中心节点服务端高并发元数据访问。文件传输过程利用 Openssl 加密传输特性，构建 Openssl 服务端，提高网络文件传输的安全性。

本文基于 OpenHarmony 操作系统实现了分布式文件级去重，并部署在润和大禹智能开发套件上，采用自拟文件数据集进行了测试验证。实验结果表明，本文的分布式文件去重系统能够有效地减少数据重复性，降低存储开销，提升分布式系统的整体性能。

关键词：分布式系统，数据去重，中心化架构，OpenHarmony 操作系统

ABSTRACT

Data duplication is a common and pervasive challenge encountered in distributed file systems. Given the existence of multiple nodes and storage devices within distributed systems, the occurrence of file duplications across different nodes is plausible. This duplication of data leads to wastage of storage resources, augmentation of data redundancy, and the potential degradation of system performance. Consequently, this paper introduces a novel approach by developing and deploying a distributed file deduplication system on the OpenHarmony platform. The primary objective of this system is to effectively mitigate the storage overhead resulting from duplicate data in distributed environments.

In order to address these objectives, this research paper delves into several key aspects, including the architecture of distributed data deduplication, the design of remote file read and write functionality, and system performance optimization. Notably, this study focuses on the integration of the OpenHarmony system and the distributed data deduplication system, leveraging the inherent advantages of a domestic operating system to effectively reduce equipment storage costs. The specific research areas covered in this paper are outlined as follows:

1) Investigation of distributed de-centralization architecture. This involves proposing a distributed deduplication algorithm that incorporates a central node. The algorithm operates by storing globally unique file fingerprints in the central node, enabling efficient file deduplication and redirection.

2) Design of remote file reading and writing interaction in a distributed system: To facilitate remote access to files across devices, a library piling mechanism is employed to intercept C library standard file read and write interface calls. The implementation includes integrating network access functions into the read and write interfaces using a multithreading mechanism. Additionally, a redirection table is utilized to enable seamless remote read and write operations.

3) Exploration of system network transmission optimization: The research focuses on achieving high-concurrency metadata access on the server side of the central node through the use of Socket multithreading. Moreover, to enhance the security of network file transmission, an Openssl server is built to leverage the encryption transmission feature provided by Openssl.

Building upon the OpenHarmony operating system, this paper has successfully implemented the distributed file-level deduplication mechanism and deployed it within the Runhe Dayu intelligent development suite. To validate the effectiveness of our system, a specially designed file dataset has been employed for rigorous testing and verification. The experimental results provide strong evidence of the system's ability to reduce data redundancy, alleviate storage overhead, and improve overall system performance.

Key words： distributed systems, data deduplication, centralized architecture, OpenHarmony operating system.

目 录

1 绪论

1.1 研究背景及意义

1.1.1 研究背景

分布式数据去重是当前分布式系统领域的一个重要研究方向，其目标是通过识别和消除数据中的重复副本，从而减少存储开销、提高存储效率和降低数据冗余。根据国际数据公司（IDC）发布的最新数据报告，全球数据量正以惊人的速度增长，到 2025 年预计将达到 180 Zettabytes。这一庞大的数据规模对分布式系统提出了巨大挑战。

IDC 的另一数据研究表明，在分布式环境中，重复数据占据了大量的存储空间。根据对企业和组织的调查和分析，平均而言，分布式系统中的数据重复率高达 30%至 50%。这种数据重复现象导致了存储资源的浪费和数据冗余的增加，严重影响了系统性能和效率。同时，由于分布式系统中的节点数量众多且分散，不同节点上的文件可能会出现重复的情况，进一步加剧了数据重复的问题。

因此，为了应对这一挑战，本课题旨在设计并实现基于 OpenHarmony 平台的分布式文件去重系统，以解决分布式环境中重复数据带来的存储开销问题。OpenHarmony 作为一个开源项目，由开放原子开源基金会（OpenAtom Foundation）孵化及运营，旨在构建一个智能终端设备操作系统的框架和平台，推动万物互联产业的繁荣发展。Harmony OS 已初步实现了这一目标，成为 OpenHarmony 的成功范例。在本文中，我们将通过分析 OpenHarmony 系统、开发技术、设计周期以及系统架构设计和代码实现等必要步骤，以及智能开发套件上的测试和后期验证，全面完成系统的开发和验证。

1.1.2 OpenHarmony 架构简介

OpenHarmony 采用分层设计，整体架构由内核层、系统服务层、框架层和应用层组成。在多设备部署场景下，OpenHarmony 支持根据实际需求裁剪非必要的组件，以实现更灵活的部署配置。OpenHarmony 技术架构[1]如图 1.1 所示。

内核层 OpenHarmony 采用多内核设计，包括 Linux 内核和 LiteOS，以适应不同资源受限设备的选择。内核抽象层（KAL）通过屏蔽多内核差异，为上层提供进程/线程管理、内存管理、文件系统、网络管理和外设管理等基础内核能力。

系统服务层 系统服务层是 OpenHarmony 的核心能力集合，通过框架层对应用程序提供服务。根据不同设备形态的部署环境，基础软件服务子系统集、增强软

件服务子系统集、硬件服务子系统集内部可以按子系统粒度裁剪，每个子系统内部又可以按功能粒度裁剪。

图 1.1 OpenHarmony 技术架构图

框架层 框架层为应用开发提供了 C/C++/JS 等多语言的用户程序框架和 Ability 框架，适用于 JS 语言的 ArkUI 框架，以及各种软硬件服务对外开放的多语言框架 API。根据系统的组件化裁剪程度，设备支持的 API 也会有所不同。

应用层 应用层包括系统应用和第三方非系统应用。应用由一个或多个 FA（Feature Ability）或 PA（Particle Ability）组成。其中，FA 具备 UI 界面，提供与用户交互的能力；而 PA 无 UI 界面，提供后台运行任务的能力和统一的数据访问抽象。基于 FA/PA 开发的应用能够实现特定的业务功能，支持跨设备调度和分发，为用户提供一致、高效的应用体验。

1.1.3 研究目的及意义

本课题具有很强实践意义和理论意义。课题迎合了当下由集中式存储转变为分布式存储的演变进程，以解决分布式存储中存在的实际问题，促进分布式存储的发展。其研究目的在于：

（1）减少分布式系统中的重复文件，提高磁盘的有效存储容量。

（2）降低分布式系统数据维护的难度。

其理论意义在于：

（1）提出了一种可行的分布式文件粒度去重的架构。

（2）提供了一种文件远程读写的实现方案。

（3）通过将课题的分布式文件去重系统与 OpenHarmony 结合，为国产操作系

统生态环境的维护与构建共享力量。

1.2 研究现状

在本节中，我们将介绍一些经典的分布式文件系统，并深入探讨分布式去重领域的研究成果，以期为进一步的研究和应用提供有价值的参考。

1.2.1 分布式存储系统研究现状

随着信息技术的迅猛发展，分布式系统在过去几十年中取得了显著的进展。这一领域的研究涵盖了分布式系统的设计、性能优化、容错机制和安全性等关键方面。研究人员通过引入新的架构、协议和算法，不断提升分布式系统的可扩展性、效率和可靠性。这些科研成果的取得为分布式系统的未来发展提供了坚实的基础，并为构建更强大、可靠和安全的分布式环境提供了重要的参考和指导。

回顾发展简史，大致可以把分布式存储分为四个发展阶段：

表 1.1 分布式存储发展阶段

阶段	研究重点	主要代表
1980s 网络文件系统	实现网络环境下的文件共享，解决客户端与服务器的交互问题	AFS 文件系统、SUN 公司的 NFS 文件系统
1990s 共享 SAN 文件系统[2]	解决存储系统的可扩展性和面向 SAN 的共享文件系统	IBM 研制的 GPFS[3]
2000s 面向对象并行文件系统	研究重点主要集中在对象存储技术，如何进行高效的元数据管理和提高数据访问的并发性	Ceph 文件系统[4,5]、Google 的 GFS 文件系统
2010s 云文件系统[6]	EB 级大规模存储系统，数据高可用性方法，高效智能存储技术，以及新型的计算存储融合系统和应用感知存储。	目前很多分布式文件系统都在往的云的方向发展，如：GlusterFS[7]、Ceph 等

2003 年 Google 开发的 Google File System[8]（以下简称 GFS）问世。GFS 是 Google 为满足自身需求开发的基于 Linux 的专有分布式文件系统，能够在廉价商务服务器上部署，既确保系统可靠性和可用性，又降低了成本。作为典型的中心化架构设计，GFS 为其他分布式系统提供了参考，其最大优势是便于管理，中心

服务器能够监控子服务器状态和负载情况。然而，该模式存在单点故障的致命缺点，当中心服务器发生故障时，整个系统将不可用。

表 1.2 经典分布式文件系统对比

名称	优点	缺点
GFS	架构简单；集群水平扩展 可以构建在廉价的服务器上 存储容量大；支持 PB 级存储规模；	适合大文件读写，不适合小文件存储；没有实现 POSIX 的标准文件接口；中心化架构易发生单点故障；
HDFS	多个副本容错性高；可以处理大数据（GB\TB 级）；可以构建在廉价的机器上；	不适合低延时数据访问； 无法高效对大量小文件进行存储；不支持并发写入；
Ceph	高扩展性:本身并没有主控节点，扩展起来比较容易;特性丰富:Ceph 支持对象存储、块存储和文件存储服务，故称为统一存储；	安装运维复杂； 加入了日志导致数据实际上会双写，性能不足； 整体代码质量一般；
FastDFS	主备 Tracker 服务，增强系统的可用性；支持在线扩容机制，增强了系统的可扩展性；支持主从文件，支持自定义扩展名；	不支持断点续传，对大文件不友好；不支持 POSIX 通用接口访问，通用性较低；同步机制不支持文件正确性校验，降低了系统的可用性；

HDFS[9](Hadoop Distributed File System)是基于 Google 发布的 GFS 论文设计开发的。HDFS 是 Hadoop 技术框架[10]中的分布式文件系统，对部署在多台独立物理机器上的文件进行管理。HDFS 会将数据自动保存多个副本，通过增加副本的形式，提高容错性。其中一个副本丢失以后，可以自动恢复。

Ceph 最早起源于 Sage 就读博士期间的工作、成果于 2004 年发表，并随后贡献给开源社区。Ceph 充分利用了存储节点上的计算能力，在存储每一个数据时，都会通过计算得出该数据存储的位置，尽量做到数据的负载均衡[16]。同时由于 Ceph 采用了 CRUSH 算法[11]、HASH 环等方法，使得它不存在传统的单点故障的问题，且随着规模的扩大性能并不会受到影响。

FastDFS[23]是一个开源的轻量级分布式文件系统。它解决了大数据量存储和负

载均衡等问题。特别适合以中小文件为载体的在线服务，如相册网站、视频网站等等。但是 FastDFS 不支持 POSIX 通用接口访问，通用性比较的低。

随着云计算[25]的发展以及新硬件，新算法的加持，一些创新的分布式文件系统也出现了，它们给这个方向提供了新的演进思路。

Amazon S3 是亚马逊提供的一种面向开发人员的云对象存储服务。它提供可扩展性高、高可用性、耐久性强的存储解决方案。用户可以通过简单的 API 调用在云中存储和检索任意数量的数据。Google Cloud Storage 是谷歌云平台提供的一种持久性、高可用性的对象存储服务。它具有可扩展性、安全性和可靠性，并提供了多种存储类别以满足不同的业务需求。Microsoft Azure Blob Storage 是微软 Azure 云平台提供的一种可伸缩的对象存储服务。它支持存储和访问大量非结构化数据，具有高可用性、可靠性和安全性。阿里云 OSS 是阿里巴巴云计算提供的一种海量、安全、低成本的云存储服务。它支持多种数据类型的存储和访问，并提供了高可用性和强大的数据保护功能。

1.2.2 分布式数据去重研究现状

分布式去重是分布式系统领域的一个重要研究方向。在分布式发展的历史长河中，重复数据所带来的系统性能开销问题一直备受关注。作为数据存储和管理领域的重要课题，分布式系统和分布式数据去重引发了广泛的研究兴趣，并衍生出许多相关研究成果。

关于数据去重的算法研究方面，一些经典的算法被广泛应用于分布式系统中。例如，基于指纹技术的算法可以通过计算文件内容的唯一标识来判断文件是否重复，如经典的 Rabin 指纹[12]和 SHA-1 算法等。此外，还有基于哈希函数和快速哈希查找表[21]的去重算法，如局部敏感哈希（LSH）和 Bloom 过滤器[22]。这些算法在不同的应用场景中展现出了良好的去重效果和性能。

分布式数据去重的系统设计和实现方面也取得了显著进展。研究人员提出了各种不同的系统架构和机制，以应对分布式环境中的去重挑战。例如，一些系统采用中心化架构[13]，它基于中心节点的设计，该中心节点负责协调和管理整个去重过程。该架构分为两个阶段：局部去重和全局去重。在局部去重阶段，每个节点使用局部去重算法（如哈希或指纹算法）对本地数据进行去重，识别并删除重复的数据块。然后，节点将保留的唯一数据块及其相关信息发送给中心节点。在全局去重阶段，中心节点接收来自各个节点的数据块信息，并进行全局去重操作。它使用全局去重算法（如索引或冗余检测算法）进一步识别和删除重复的数据块。中心节点维护全局数据块索引，记录已存在的数据块及其位置信息。

当前，许多分布式数据去重系统采用中心化架构进行实现。例如，德国研究机

构 SIT 开发的 Dedoop 是基于 Hadoop 的分布式数据去重系统，致力于解决大规模数据去重的挑战。此外，Dedoose 公司的 CloudDedupe 是专门为云存储环境设计的数据去重解决方案。该系统通过集中管理和协调各个存储节点的去重操作，有效减少了数据的存储开销和传输带宽消耗。这些系统的研发与应用为分布式数据去重领域带来了新的进展。但中心化架构始终存在如单点故障[14]问题。

而在另一些系统中，采用非中心化架构将数据去重任务分配到不同的节点上。例如，基于 P2P 网络[15]的去重系统充分利用对等网络的优势，将去重任务分散到各个节点进行处理。每个节点负责管理和去重自身持有的数据，通过协议和算法实现数据块的去重和存储。此外，基于比特币区块链技术[24]通过去中心化的方式实现数据的去重和存储。利用比特币区块链的共识机制，参与节点共同验证和存储交易块，确保数据的去重和存储的可靠性。当新的数据块添加到区块链时，节点对其中的交易进行验证，避免重复数据的存储。这些非中心化架构的数据去重系统充分利用分布式计算和通信技术，将去重任务分散到各个节点进行处理，实现高效的数据去重和存储，适用于分布式环境下的数据去重应用场景。

1.3 研究内容

本研究旨在从分布式数据去重架构、文件远程读写设计和系统性能优化的角度，对相关问题进行深入研究。与过去的研究工作相比，本课题的主要贡献在于将 OpenHarmony 系统与分布式数据去重系统有机融合，充分利用国产操作系统所具备的全面的开发技术优势，以降低设备存储开销。通过在系统架构设计中的创新，我们致力于提升分布式数据去重的效率和性能，以满足在大规模数据存储环境下的高效处理需求。通过该研究，我们期望为构建更高效、可靠和可扩展的分布式数据去重系统提供有价值的思路和方法，并为国内分布式系统领域的发展做出贡献。

1.3.1 针对分布式去重中心化架构的研究

本文提出了一种具有中心节点的分布式去重算法，旨在通过中心节点存储文件的全局唯一指纹来实现文件的定位，从而实现重复文件删除和文件重定向的目标。该算法的核心思想是利用中心节点作为全局指纹索引的存储单元，通过将文件的唯一指纹存储在中心节点上，实现了高效的文件定位和去重功能。通过中心节点的协调和管理，分布式系统可以有效地识别和删除重复的文件，减少存储资源的浪费，提高存储系统的利用率。

1.3.2 针对分布式系统中远程文件读写交互的研究

本文采用了库打桩机制来截取 C 库标准文件读写接口的调用。通过引入多线程机制，我们为读写接口添加了网络访问功能，并结合重定向表的应用，成功实现了设备与设备之间的远程文件读写访问。该设计方案旨在提供高效的通信机制，使得不同设备能够轻松地进行文件的读写操作，从而有效解决了跨设备间数据共享和通信的需求。我们的工作为分布式环境下的文件访问与交互提供了一种可行而有效的解决方案，为实现跨设备间的数据交流和共享打开了新的可能性。

1.3.3 针对系统网络传输优化的研究

本文中我们采用 Socket 多线程方式来实现中心节点服务端的高并发元数据访问。为了提高网络文件传输的安全性，我们利用 Openssl 加密传输特性构建了 Openssl 服务端。通过这种方式，我们能够有效地应对大量的元数据访问请求，并确保网络文件传输过程的安全性。这一设计方案为分布式系统中的元数据管理和文件传输提供了可靠的技术支持，为保护数据的机密性和完整性提供了有效的手段。

1.4 本章小结

本章详细介绍了课题的背景和研究动机，强调了在分布式系统中存在大量重复文件导致存储开销的问题，并对国内外相关研究进行了调研，确定了本课题的研究内容、目的和意义。此外，还补充介绍了 OpenHarmony 平台的相关知识，为后续章节的内容提供了必要的背景知识。

在下一章节中，我们将介绍系统的开发所使用的关键技术。这些技术包括 IPC 技术、库打桩技术和文件加密传输技术等，它们在系统的设计和实现过程中发挥了重要作用。通过对开发环境和关键技术的介绍，读者将更好地理解系统的实现过程，并为后续章节的详细讲解做好准备

2 关键开发技术

2.1 IPC 技术

本文所设计的 Data node(数据节点端)需要在扫描进程和远程文件请求进程间使用进程间通信[17]以共享文件重删表 进程间常见的通信方式有以下 5 种：管道 pipe、命名管道 FIFO、消息队列 MessageQueue、内存共享 SharedMemory 和信号量 Semaphore。本文主要采用了内存共享技术、管道技术和 Socket 网络通信技术实现进程间通信。

共享内存是一种将一段可被多个进程访问的内存映射出来的机制。该共享内存段由一个进程创建，但其他多个进程也可以通过映射来访问它。共享内存作为一种高效的进程间通信方式，旨在应对其他通信机制效率较低的问题。通常情况下，共享内存与其他通信机制（如信号量）结合使用，以实现进程间的同步和通信。在本文中，我们主要依靠进程间通信和共享内存机制来实现主要扫描进程和远程文件请求进程之间共享文件重删表的过程。具体的实现细节将在第四章中详细阐述。通过这种方法，我们能够有效地利用共享内存来满足进程间数据共享和通信的需求。

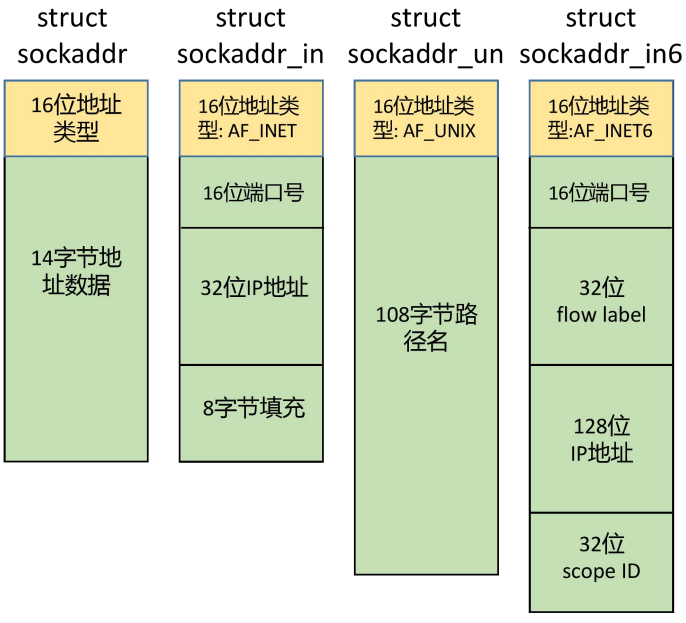

图 2.1 socket 中 TCP 通信数据结构字段

所谓管道通信，是指用于连接一个读进程和一个写进程，以实现它们之间通信

的共享文件，又称 pipe 文件。向管道（共享文件）提供输入的发送进程（即写进程），以字符流形式将大量的数据送入管道；而接收管道输出的接收进程（即读进程），可从管道中接收数据。由于发送进程和接收进程是利用管道进行通信的，故又称管道通信。在本课题中管道通信主要用于用户态执行系统调用命令文件。在 Data node 文件传输服务端需要使用到管道通信执行 find 系统调用命令以获取对应 inode 号的文件，再将该文件通过网络传输发送到对应的设备端点。具体实现将在第四章介绍。

UNIX Domain Socket 是在 Socket API 的框架上发展出的一种 IPC 机制，尽管 Socket 最初是为网络通信设计的。与网络 Socket 相比，UNIX Domain Socket 在同一台主机的进程间通信中更加高效。它不需要经过网络协议栈，无需进行打包拆包、计算校验和、维护序号和应答等操作，只需简单地将应用层数据从一个进程复制到另一个进程。这是因为 IPC 机制本质上是可靠的通信方式，而网络协议是为不可靠的通信设计的。UNIX Domain Socket 提供了面向流和面向数据包两种 API 接口，类似于 TCP 和 UDP。然而，即使是面向消息的 UNIX Domain Socket 也是可靠的，消息既不会丢失也不会顺序错乱。本地套接字的实现流程与网络套接字相似，通常采用 TCP 的通信流程。具体示意如图 2.1 所示。

2.2 库打桩技术

库打桩技术[18]，可以截获对共享库函数的调用。该技术为 Linux 环境 C 语言独有。应用上可以如可以控制函数调用的输入输出值，以自己的逻辑替换函数调用。其基本思想是：创建一个与目标函数相同原型的包装函数，通过编译时函数实现的搜索机制、或链接时函数符号解析搜索的机制、或运行时动态链接库的加载机制，将自定义的包装函数替换目标函数。另外，库打桩技术还用于单元测试。在传统的单元测试中，为了测试某个函数，需要确保该函数在被调用时能够返回预期的结果。库打桩的基本思想是在测试过程中，使用一个虚拟的替代函数（称为打桩函数或 mock 函数）来代替实际的库函数。这个打桩函数可以模拟库函数的行为，返回预期的结果或执行特定的操作。通过在测试中使用打桩函数，可以控制和验证函数的调用方式和结果，从而更容易编写全面的单元测试。实现库打桩技术的一种常见方式是使用测试框架或库，如 Google Test、Unity、CMock 等。这些框架提供了相应的工具和函数，用于创建和管理打桩函数，并与被测试的函数进行交互。然而，当函数依赖于外部库函数时，这可能会导致测试的困难，因为外部库函数的行为无法直接控制。这时，可以使用库打桩技术来解决这个问题。使用库打桩技术进行单元测试的步骤通常包括以下几个方面：（1）创建打桩函数：

根据需要，编写一个打桩函数，它的行为应与实际的库函数相似，但可以返回预期的结果或执行特定的操作。（2）打桩设置：在测试用例中，使用测试框架提供的函数或宏，将打桩函数与实际的库函数进行关联，以替代实际的函数调用。（3）执行测试：执行测试用例时，被测试的函数会调用打桩函数而不是实际的库函数。（4）验证结果：使用测试框架提供的断言函数或宏，验证被测试函数的行为是否符合预期。

本课题基于库打桩技术独立编写了一套可以远程文件访问的动态链接库，用户使用该动态链接库编译代码可以实现设备的远程文件访问功能。用户在使用系统时可以选择运行时打桩编译，将文件读写接口替换为远程文件访问接口，替换只需要在编译文件时携带库名即可，做到用户无感化远程文件访问。

2.3 文件加密技术

本课题在文件传输时使用了 openssl 库的文件加密传输技术[19]。openssl 是一个安全套接字层密码库，囊括主要的密码算法、常用密钥、证书封装管理功能及实现 ssl 协议。openssl 整个软件包大概可以分成三个主要的功能部分：ssl 协议库 libssl、应用程序命令工具以及密码算法库 libcrypto。

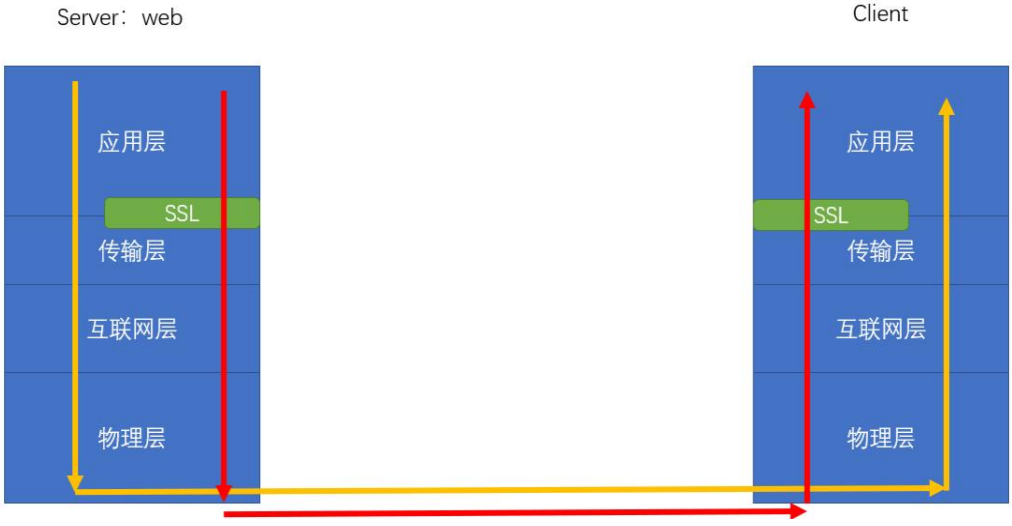

图 2.2 ssl 概念图

ssl 是在 TCP/IP 模型的应用层与传输层之间加上半层的 ssl 公共通用模块，调用这个模块 http 将会变成 https。图 3 中的黄线，表示没有调用 ssl 模块时的情况，红线代表调用 ssl 模块的情况，经过 ssl 时服务器端会使用算法对数据进行加密，并进行身份认证，使客户端访问服务器时更加安全。

此外 Data node（数据节点）会计算文件指纹，文件指纹使用的是 MD5 文件加密技术。 MD5 即 Message Digest -Algorithm 5（信息-摘要算法 5），用于确保信

息传输完整一致。是计算机广泛使用的杂凑算法之一（又译哈希算法），主流编程语言普遍已有 MD5 实现，并且目前也有许多与 MD5 增强算法[20]的相关研究。在分布式系统中，传统的文件去重中使用哈希算法标识文件内容，本课题依旧沿用传统文件去重方法，与分布式相结合从而形成独特的中心化架构分布式文件去重方法。

使用 MD5 算法作为文件计算指纹的原因如下：

- 压缩性：任意长度的数据，算出的 MD5 值长度都是固定的。
- 容易计算：从原数据计算出 MD5 值很容易。
- 抗修改性：对原数据进行任何改动，哪怕只修改 1 个字节，所得到的 MD5 值都有很大区别。
- 强抗碰撞：已知原数据和其 MD5 值，想找到一个具有相同 MD5 值的数据是非常困难的。

2.4 本章小结

在本章中我们介绍了在系统开发过程中所使用的关键技术。其中包括 IPC 技术（进程间通信），库打桩技术和文件加密传输技术。这些技术为后续系统的设计和实现提供了基础，并指明了我们的技术栈和开发路径。

在下一章节中，我们将详细介绍本系统的架构设计。这将包括系统的整体设计思路、各个模块之间的关系和功能，以及核心去重算法的实现原理。通过对系统架构的介绍，读者将更好地理解系统的设计思想和整体框架，并为后续章节的具体实现和测试部分做好准备。

3 系统设计

3.1 系统概述

本课题的重点之一在于系统架构的设计。课题实现的分布式文件去重系统借鉴了经典分布式系统 GFS 的中心化架构设计，基于 Linux 平台进行实验，自主编写文件读写接口，通过文件指纹进行对比以确定需要删除的重复文件。实验后期将采用华为自主研发的 OpenHarmony 平台进行上板测试。本系统主要涉及到的主干模块如下：Metadata node 模块、Data node 模块、自定义动态链接库。其中 Data node 模块可以拆分为设备扫描模块、文件收发模块和共享内存管理模块。

3.2 系统整体架构和分布式去重方案设计

本课题借鉴了 GFS 中心化架构设计，同样采取了元数据和文件数据分离存储的策略。这种将元数据和数据分离管理的策略有以下优点：简化元数据管理、提高系统灵活性、改善系统性能、增强系统可靠性和提高数据安全性。下图为本课题设计的中心化架构示意图：

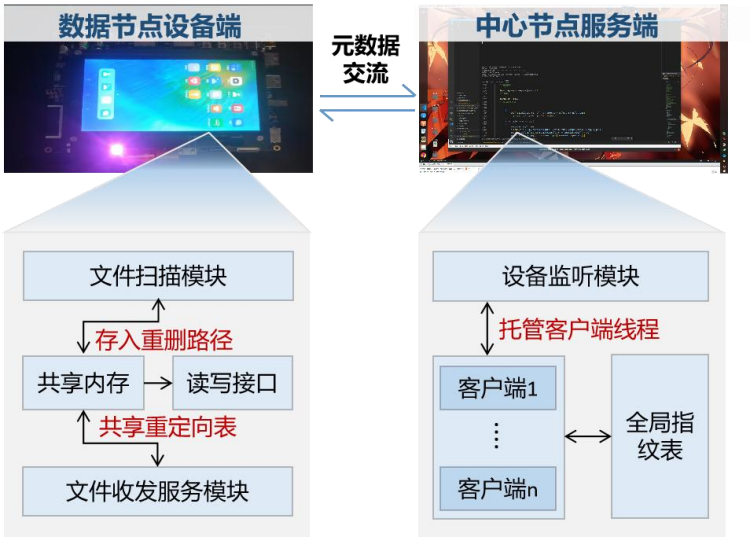

图 3.1 系统中心化架构图

（1）分布式去重算方案设计：

① 1 号、2 号和 3 号设备进行周期性扫描，获取在一个周期内的新增或发生修改的文件。计算新增文件的指纹并打包成新增文件指纹。

② 1 号、2 号和 3 号节点将各自的新增文件指纹表各自沿着上图中绿色元数据传输路径发送至 Metadata node（中心节点）。

③ Metadata node 监听来自设备节点的输入，解析新增文件指纹表，指纹表的解析结果可能有以下几种情况，四种情形的具体设计将在 3.3.1 中介绍：

- 新增文件地址唯一，且指纹全局唯一，将指纹添加到全局表中
- 新增文件地址存在，但指纹内容不同，则修改该地址上的指纹
- 新增文件地址存在，但指纹内容相同，不做任何修改
- 新增文件地址唯一，但指纹已经存在，则重定向到目标地址

④ Metadata node 将重定向文件打包并回复各个设备节点指纹重定向或者重删的情况。消息回复主要沿着上图绿色元数据传输路径。

⑤ 1 号、2 号和 3 号设备收到 Metadata node 的回复后依据回复消息进行文件的重删和重定向，删除系统中的重复文件。本轮去重结束。

（2）分布式文件远程访问方案设计：

这里以 1 号访问 2 号设备上的 test.txt 文本文件为例，该文件已经在周期扫描中被重删且重定向到了 2 号设备上的 test.txt 文本文件。流程如下：

① 1 号设备首先在本地访问 test.txt 文本，由于 test.txt 文本已经被重删导致无法正常访问，则告知设备节点访问本地重定向表。

② 设备访问本地重定向表，查明 test.txt 文本文件被重定向到了 2 号设备，此时 1 号设备通过上图中与 2 号直连的黑色文件传输路径建立连接。连接建立后发送 test.txt 文本元数据。

③ 2 号设备收到连接请求，依据文本元数据找到本地存储的 test.txt 文件并发送到 1 号设备。

④ 1 号设备收到 test.txt 文本文件，本次文件请求结束。

3.3 系统模块设计

本小节将介绍分布式文件去重系统中所包含的子模块设计方案。共包含两个主要模块，分别是：Metadata node 模块设计、Data node 模块设计。

3.3.1 Metadata node 模块设计

Metadata node 又称作元数据节点。在分布式文件去重系统中主要负责文件元数据管理和重删指令的发送。在元数据节点中只会存储文件的元数据，不会存储文件数据本身。元数据节点需要存储全局文件唯一的指纹和地址，当系统中出现新增文件时，元数据节点会在全局指纹表中进行比较，以此来推断文件是否重复。此外元数据节点还负责 Data node 地址存储，方便出现重复文件后发送重删指令到

特定地址的数据节点。

中心节点将设备节点发来的增文件表进行解析，分四种情况处理：

① 新增文件地址存在，指纹内容不同，则修改该地址上的指纹

② 新增文件地址存在，指纹内容相同，不做任何修改

③ 新增文件地址唯一，且指纹全局唯一，将指纹添加到全局表中

④ 新增文件地址唯一，但指纹已经存在，则重定向到目标地址

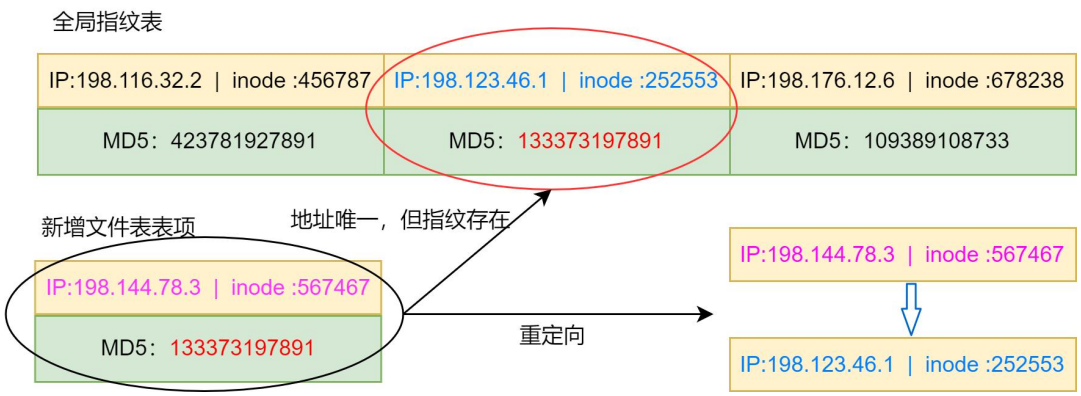

图 3.2 由上至下四种情形

3.3.2 Data node 模块设计

Data node 又称作数据节点。在分布式文件去重系统中数据节点主要负责保存文件数据和传输文件。数据节点会启动扫描线程定期扫描设备中的新增文件，计算新增文件的我指纹，并将新增文件的 inode 信息连同指纹打包成新增文件表发送至元数据节点。此外，数据节点既作为文件传输的服务器端，又作为文件接收的客户端，因此文件数据的传输与访问不会涉及到元数据节点。这样设计可以使得文件传输不经过中心节点中转，避免大量的文件传输使得中心节点成为文件系统的瓶颈。Data node 设计架构图如下：

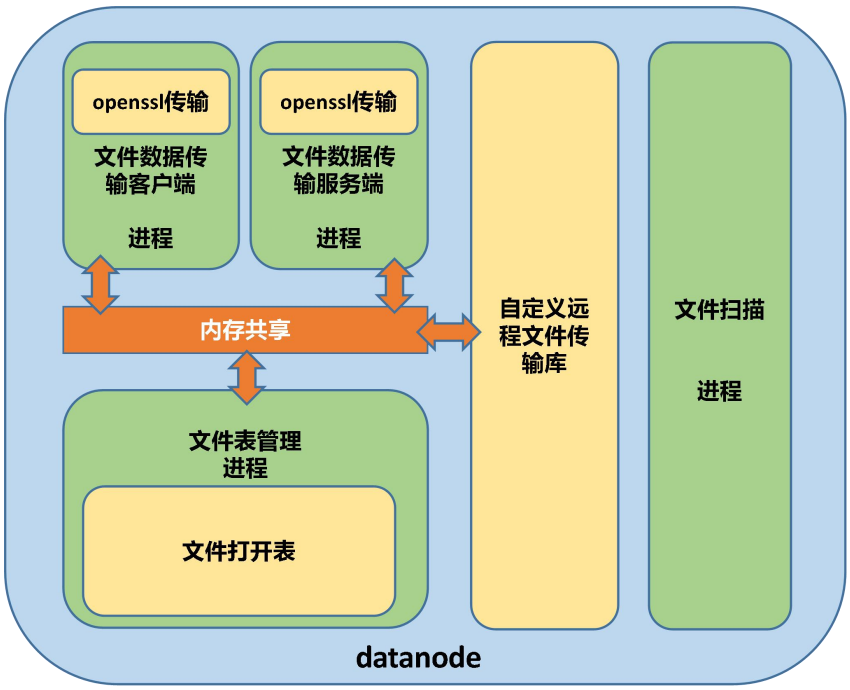

图 3.3 Data node 整体架构图

Data node 端出现读写文件的请求时需要调用文件读写接口，本文实现的自定义实现的接口符合 C 库标准读写接口规范，在 C 库标准读写功能基础上添加了远程读写的功能，可以实现多个 Data node 的远程文件读写需求。个别具体接口读写流程图和介绍如下：

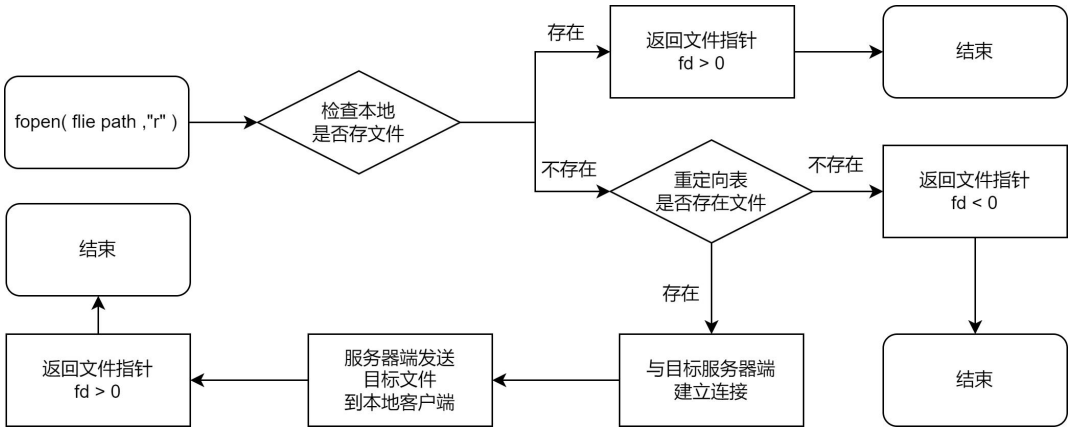

图 3.4 fopen 打开只读文件流程图

fopen 改写后具备了远程访问文件的功能。fopen 首先会检查传入路径的合法性并确定改路径文件的类别。如果该文件是本地文件则正常返回文件指针并退出；如果该路径对应的文件不存在，则 fopen 会检查本地重定向表以确定该路径对应的文件是否为重删文件。一旦确定为重删文件则 fopen 将开启子线程进行远程文件传输访问获取文件，最终返回远程文件指针。若上述两种情况均不存在，则 fopen 判定参数路径无效，返回空文件指针 NULL。

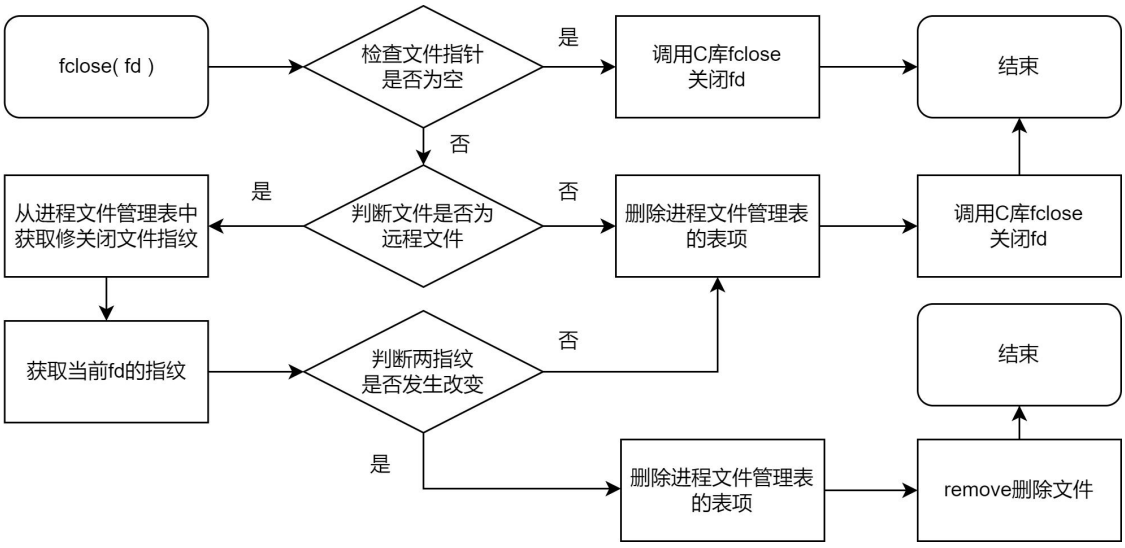

图 3.5 fclose 关闭只读文件流程图

fclose 接口改写后具备了删除扫描期间未被修改的文件功能。fclose 的设计初衷在于迎合本课题分布式去重的目标，尽量以最大化效果节约磁盘上的存储空间。为了达到这个目的，fclose 在关闭文件指针后会依据文件标记删除那些未被修改的文件。而在文件打开或使用期间被修改的文件，fclose 不会将其删除。

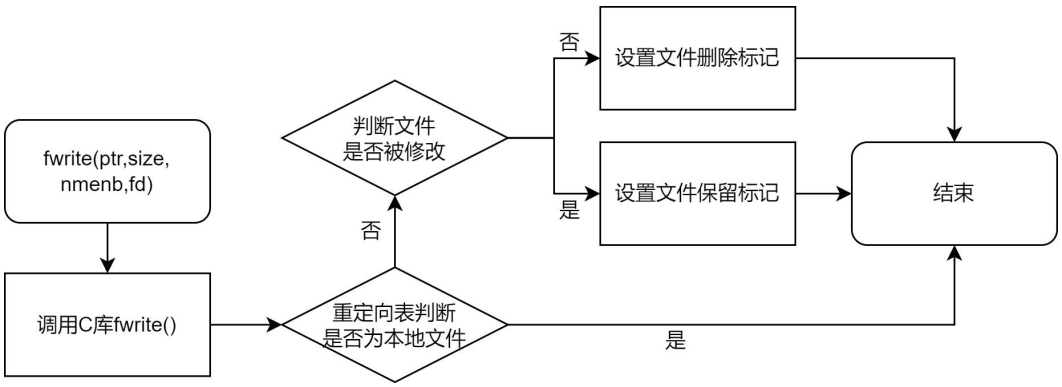

图 3.6 fwrite 写追加文件流程图

fwrite 作为文件写入接口之一，大体功能基本不变，唯一需要注意的是 fwrite 可以根据文件指纹判断文件是否在打开期间被修改。如果被修改则对文件做出标记，方便 fclose 的后续处理。

本文将在后续第 4 章节中详细介绍各个模块以及接口的实现方式，下一章节我们将介绍本课题所使用到的相关开发技术。

3.4 本章小结

本章详细介绍了分布式文件去重系统的整体架构设计方案。我们对系统的分布式去重原理和整体设计进行了详细说明，并给出了系统的功能和特点概述。此外，还对系统中各个模块的设计进行了简要的概述，以便读者对系统的整体结构有一个初步的了解。

在接下来的第四章中，我们将重点介绍本系统中各个模块的具体实现方式。通过深入的讲解，读者将能够全面了解每个模块的功能和工作原理。我们将涵盖系统的各个关键模块，为读者提供清晰的系统设计和实现的整体视角。

4 系统实现

在前面的章节中已经对本课题所设计和实现的系统进行了技术上探讨。本章将接着之前的论述，结合不同功能模块的功能，对系统进行进一步的分析和实现。

4.1 系统开发环境

在本小节中将介绍系统开发所用到的开发环境，对于 OpenHarmony 开发硬件设备规格将在第五章中详细介绍。课题所使用的开发环境如下表所示：

表 4.1 分布式文件去重系统开发环境

开发模式	Windows10 +Ubuntu20.24 混合开发模式
系统型号	宿主机：16G 内存, i5 CPU; 虚拟机：Linux 5.0 内核，8G 内存
开发工具	vscode 2023
硬件设备	OpenHarmony 智能开发套件 润和大禹系列 HH-SCDAYU200 RK3568 开发板
开发语言	C 语言

本课题于 Windows10 操作系统开发 + Ubuntu 20.24 虚拟机混合开发模式。代码编译主要在 Linux 虚拟机中进行，编译完毕后在 Windows 烧录上板测试。 混合开发流程如下：

（1）Windows 作为宿主系统，在其中安装虚拟机软件。本课题使用的是 VMware2023 作为虚拟机管理软件。

（2）在虚拟机中安装 Linux 发行版，网络采用 NAT 模式并绑定静态 IP。虚拟机根据编译需要分配 8G 内存和 120G 硬盘空间。

（3）在 Linux 中配置 sshd，git，安装所需的环境和软件。

（4）在 Windows 的 Hosts 文件中给 Linux 的静态 IP 添加解析。

（5）在 Windows 中安装 VS Code, 并安装 Remote - SSH 插件，然后配置 Remote - SSH 连接到 Linux VM。

混合开发模式有助于提高开发效率和开发灵活性。在 Ubuntu 操作系统中，可以通过使用一些强大的命令行工具和开发工具来提高开发效率。同时，Ubuntu 操作系统也支持更多的脚本语言和开发工具，这对于快速开发和快速迭代非常有帮助。此外，虚拟机也提供了更好的开发环境的隔离和管理。

4.2 Data node 模块实现

本小节将介绍 Data node 模块的运行流程与实现方式。

Data node 模块主要由文件扫描模块、自定义远程文件传输模块和文件收发服务模块组成，其中将各个模块关联起来的是内存共享技术。在内存共享中会存储各个子模块所需的数据结构，各个子模块在访问共享内存时会通过锁的方式访问，这样做的目的是防止，多个进程访问同一个数据结构时发生数据错误。

Data node 中的主体线程为 clock_filesearch ()线程。该线程流程中首先创建共享内存，然后将共享内存挂载至主体进程中，其次检测设备是否正常运行。如果设备正常运行扫描过程中会将新增文件计算指纹并打包发送，如果设不正常运行，则销毁共享内存，主线程停止执行。整体流程和关键调用栈如下图所示：

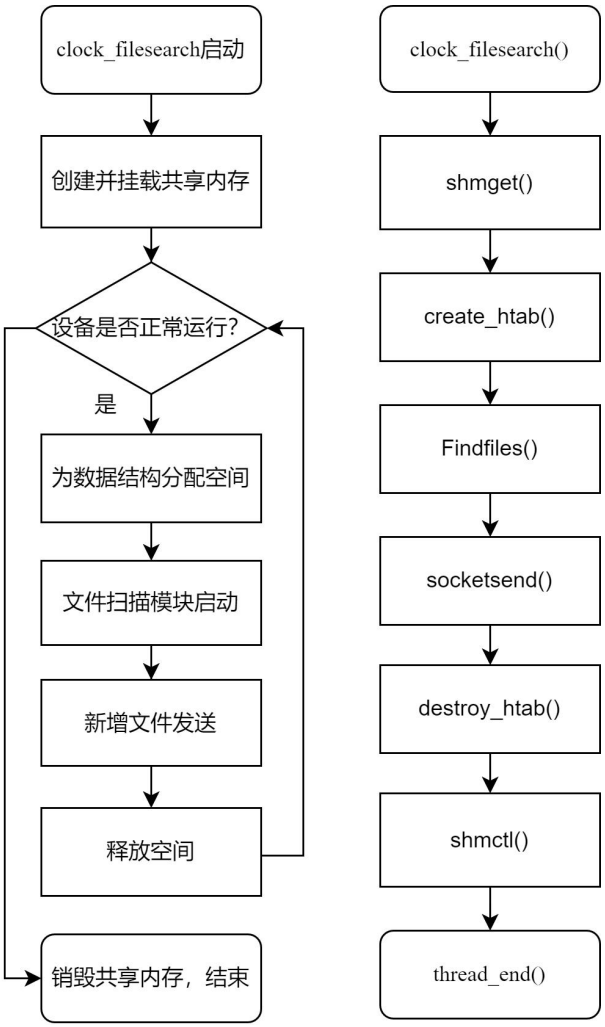

图 4.1 Data node 整体调用流程和调用栈

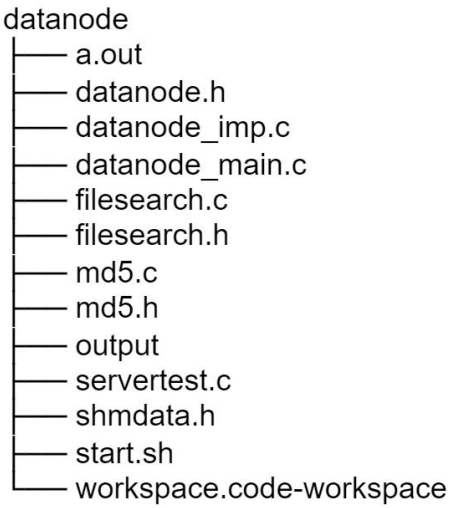

图 4.2 Data node 目录树

关于 data node 的目录树构成，其中，a.out 文件为编译后的可执行文件；datanode.h 包含了模块所需的主要接口定义以及数据结构；datanode_imp.c 为接口实现；datanode_main.c.为设备端启动起点；filesearch.c 为文件扫描子模块；md5.c 为文件指纹计算实现；shmdata.h 为内存共享树结构定义；servertest.c 为设备端文件收发服务模块实现测试文件；start.sh 为分布式测试脚本。

4.2.1 文件扫描子模块实现

文件扫描子模块的主要功能是扫描设备中的新增文件并发送新增文件到 Meta datanode(中心节点)。该模块会以文件的生成和修改日期作为文件新增的评判标准，如图 4.3 所示。

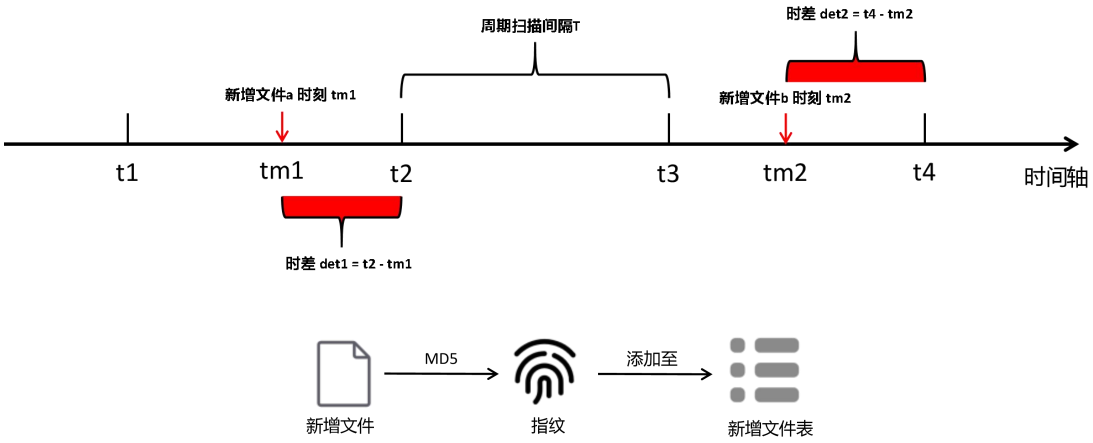

图 4.3 新增文件依据时间判定和文件指纹计算

t1、t2、t3、t4 为线程周期扫描时刻，周期扫描时间间隔为 T。tm1 和 tm2 分别

新增文件 a 和 b。在 t2 时刻，线程遍历目录发现文件 a 时差 det1 小于 T，则 a 在该轮扫描中被认为是新增文件，随后记录 a 文件状态并计算其指纹；在 t4 时刻，线程遍历目录发现文件 b 时差 det2 小于 T,文件 a 时差(t4-tm1)大于 T，因此判定 b 为新增文件，随后记录 b 文件状态并计算其指纹。

在文件扫描方式方面，本文采用经典的递归目录扫描的方式进行，一方面这样实现相对于其他算法来说较为简单，但是不足的是递归扫描的开销较大，后续的开销分析会在第五章节详细讲解，此处只作简单说明。采取递归扫描的另一个原因在扫描周期的可变性。由于扫描周期的设置可以远超扫描时间开销，当周期设置越长，开销对系统运作的损耗就越低。简单举例说明：假设扫描周期为 1h，意味着设备每经过 1h 就会进行一次分布式去重操作，而扫描周极限情况下耗费 5min，则扫描在系统运行周期内的时间开销为 5min/60min = 8.3% ；如果将扫描周期延长到 2h，扫描耗费时间不变，则扫描在系统运行周期内的时间开销为 4.15%。

4.2.2 新增文件发送子模块实现

新增文件发送模块的实现主要在 Socdket_Send()中完成。Socket_Send()是一个 Socket 线程，该线程首先会初始化 Socket TCP 结构（见第二章 IPC 技术），然后绑定本地客户端 IP 地址并请求连接 Meta datanode 服务端，随后将打包的新增文件发送到服务端并等待服务的收到回复，最后关闭本次 Socket 通信。

消息回复数据结构和新增文件表数据结构如下：

表 4.2 Instruction 结构字段表

字段	MSG	Count_ino	Ino_arr_del	Ino_arr_redir	IP_datanode
类型	int	int	ino_t 数组	ino_t 数组	字符串数组
功能	回复消息	计数	重删 inode	重定向 inode	目标 ip

表 4.3 Datasend 结构字段表

字段	Count_ino	File_ID	Ino_arr
类型	int	字符串数组	ino_r
功能	计数	存储新增指纹	存储新增 inode 号

内存共享主要发生在 Socket 线程收到中心节点回复后。根据表 5 可知中心节点会回复本轮扫描中重复文件的 inode 编号和目标重定向文件地址，该地址由目标节点的 IP 地址和目标文件的 inode 号构成。Socket 线程在收到消息回复数据结构后启动 store_memory()函数将重删文件地址保存在共享内存中，方便后续访问恢

复，然后 Socket 线程启动 del_inode()函数，该函数可以根据传入的文件 inode 号删除本地文件。

store_memory()函数的主要功能是内存共享存储，流程见图 13。其调用栈如下为：获取共内存锁 > 修改共享内存 > 释放锁 。在修改共享内存时会存储目标重定向文件的 inode 号、目标 IP 地址以及本地重删文件硬链接路径。这里存储路径的目的是方面后续远程访问恢复重删文件，但是出于文件存在硬链接的可能，因此存储路径需要存储重删文件的硬链接，确保文件恢复的正确性。关于锁的获取，如果获取失败则线程进入轮询状态，直到重新获取锁，这样可以保证共享内存区域数据结构的互斥访问。共享内存中具体存储的数据结构将会在 4.2.4 小节详细说明。

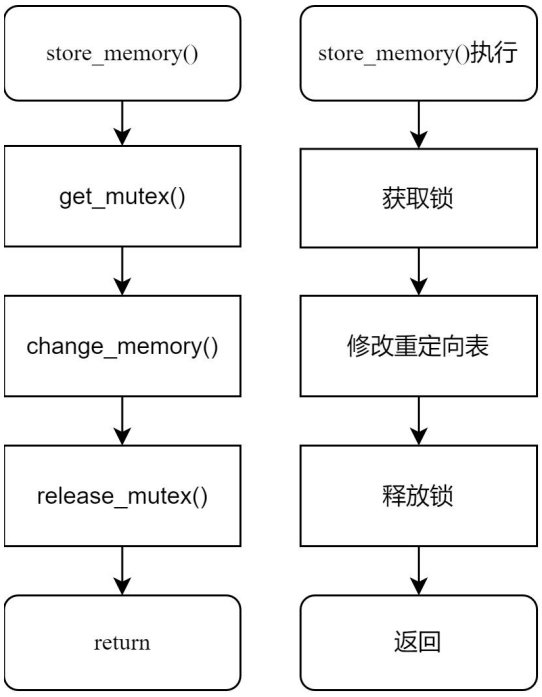

图 4.4 store_memory 流程以及调用栈

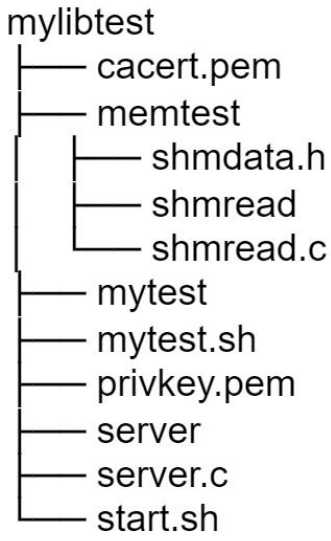

图 4.5 文件收发服务模块目录结构

4.2.3 文件收发服务模块实现

文件收发服务模块主要负责 Data node 端与端之间的远程文件传输。该模块更多的是接收其他端获取文件的请求信息，然后根据请求中携带的地址信息返回给对应端正确的文件。

在目录结构方面，cacert.pem 为 Openssl 生成的数字签名证书；memtest 为内存共享测试目录；privket.sh 为 Openssl 生成的私钥。

在技术实现方面，文件收发服务主要用到了 openssl 文件加密传输技术，此处

采用 OpenSSL 的开源加密算法。该传输建立在 socket TCP 传输框架之上增加 OpenSSL 加密系统，需要遵循如下的步骤：

客户端模型：1.初始化；2.创建 SSL；3.数据收发；4.关闭。

服务端模型：1.初始化；2.载入数字证书和私钥；3.创建 SSL；4.数据收发；5.关闭。

文件收发服务端生成私钥并使用私钥签名生成公钥的语句如下：

创建私钥：# openssl genrsa -out privkey.pem 2048 ； genrsa 表示创建私钥，privkey.pem 输出私钥文件名，2048 表示位数

创建公钥：# openssl req -new -x509 -key privkey.pem -out cacert.pem -days 1095；

下图为公私钥生成效果图：

图 4.6 公钥效果图（左）、私钥效果图（右）

下面详细叙述整个服务端的流程。步骤一：SSL 初始化。SSL 初始化使用 SSL_library_init()加载 SSL 库，然后调用 OpenSSL_add_all_algorithms()接口添加加密算法，接着加载错误返回信息，最后调用 SSL_CTX_new()新建 ssl 连接。步骤二：公私钥载入。步骤三：创建 socket，绑定 IP 地址和端口号并启动监听。步骤四：监听过程调用 SSL_read()获取客户端发送的文件请求信息。该文件请求信息主要为文件 inode 号。由于存在 find 系统调用，此处则通过管道技术传入文件 inode 号并使用 find 获取文件路径。步骤五：服务端将目标文件通过 SSL_write()发送到客户端。步骤六：断开 SSL 连接，本次通信结束。

4.2.4 自定义远程文件读写库

自定义远程文件读写库设计的初衷在于使用户能够在文件重删后依旧可以无感的访问重删文件。用户使用自定义远程文件读写库时只需要在编译时加入该库路径即可无感访问远程文件。这种与用户程序低耦合的设计同样是本课题的初衷之一。

表 4.4 alocate_info 结构字段表

字段	Shared	shm	shmid
类型	redirect_list	Void*	int
功能	重定向表体	内存分配指针	内存分配编号

表 4.5 redirect_list 结构字段表

字段	Mutex	list	Index_map	count
类型	int	redirect_addr 数组	Int 数组	int
功能	内存锁	重定向表项	位图索引	计数

表 4.6 redirect_addr 结构字段表

字段	Old_path	Old_path_number	inonumber	IP_addr
类型	字符串数组	int	Ino_t	字符串数组
功能	存储重删路径	硬链接计数	目标文件 inode 号	目标 IP 地址

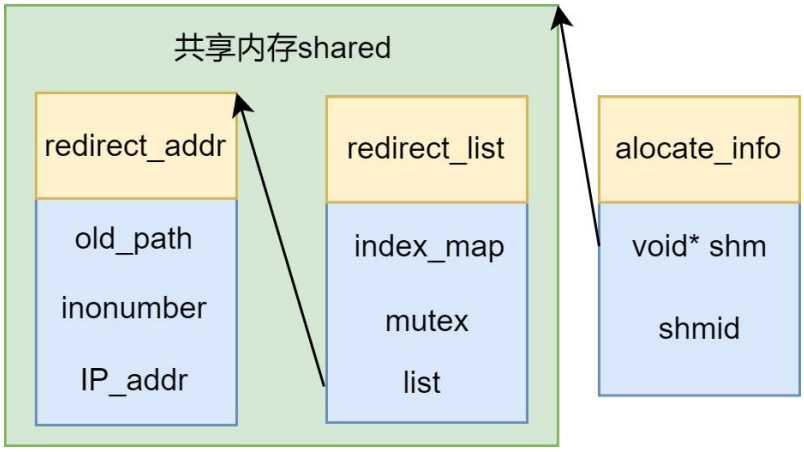

图 4.7 共享内存结构 alocate_info、redirect_list 和 redirect_addr 关系示意图

该库的核心在于与共享内存的交互。共享内存中存储了以下数据结构：redirect_addr, redirect_list 和 alocate_info。alocate_info 表示内存共享数据结构，是各个模块之间产生联系的桥梁。redirect_list 结构存储所有的重定向地址，也就是前文提到的重定向表。重定向表的表项为 redirect_addr 结构，表项由重删地址 old_path、重定向 inode 号和目标 IP 地址构成。这三个结构为包含关系：alocate_info > redirect_list > redirect_addr 见图 4.7。结构的具体信息见表 4.4、表 4.5 和表 4.6 所示。

（1）fopen 接口实现

我们通过库打桩技术来进行 fopen 函数的自定义实现，使得其具备远程文件传输功能。首先，fopen 会进行一些检查，确保进程未超过最大文件指针数和最大打开文件数。然后，它使用动态链接库的 dlsym 函数获取了系统中实际的 fopen 函数的指针，并将其保存在 fopenp 变量中。接下来，fopen 创建了一个子线程，并将 fopen_r 函数作为线程的入口点，我们将在下一个段落详细介绍 fopen_r 的实现。这个函数的作用是在子线程中执行实际的 fopen 函数，并将返回的文件指针存储在结构体 pstru 的 fback 成员中。主线程使用 pthread_join 函数来等待子线程的执行结束，并获取子线程的返回结果。在子线程执行完毕后，它将返回的文件指针存储到进程的文件指针表和文件打开表中，并释放之前分配的内存。最后，它返回文件指针 fl。总的来说，上述流程是对标准的 fopen 函数进行扩展，以添加额外的功能或记录文件操作的信息。通过创建子线程来执行实际的 fopen 函数，可以在子线程中处理一些耗时的操作，而不会阻塞主线程。同时，通过存储文件指针和相关信息到进程的表中，可以进行进一步的处理或跟踪文件操作。

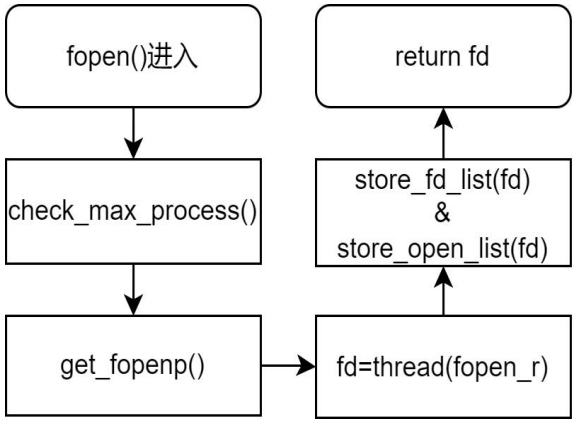

图 4.8 fopen 调用栈

对于子线程函数 fopen_r，首先打印一条消息，提示进入 fopen_r 函数，以方便后续的函数调试和调用栈打印。然后将传递给子线程的参数 arg 转换为结构体指针 pstru，并进行相应的赋值操作。使用 access 函数检查文件是否存在。通过对文件进行 F_OK 访问权限检查，如果返回值为 0，则表示文件存在。在这种情况下，直接在本地打开文件，调用 pstru->fopenp 函数，并将打开模式__modes 传递给 fopenp 函数。将打开的文件指针存储在 pstru->fback 中，并将 pstru->is_local 标志设置为 true，表示文件在本地存在。

如果文件不存在，则查找重定向表。为了实现文件的远程传输，首先分配共享

内存，使用 allocate_memory 函数进行内存分配。然后使用 is_in_list 函数在重定向表中查找文件。如果找到匹配的文件，表示需要通过网络进行文件传输。然后使用 Socket_Client_Fopen_r 函数将文件传输到远程位置，随后再次调用 pstru->fopenp 函数在本地打开文件，并将打开的文件指针存储在 pstru->fback 中。将 pstru->is_local 标志设置为 false，表示文件通过远程传输获取。

如果文件不在重定向表中，表示文件是新创建的或者无法访问的文件。根据需求，可以根据自定义的处理逻辑进行处理，例如创建新文件或返回 NULL 指针。在远程传输和文件处理完成后，释放共享内存和相关资源，以避免内存泄漏。在子线程函数的末尾，调用 pthread_exit 函数退出子线程。

在 Socket_Client_Fopen_r 函数中同样使用了前文提到的文件收发模块的 SSL 文件加密传输技术。通过创建 SSL 上下文和 SSL 对象，使用 SSL 库提供的函数进行加密通信。连接服务器后，调用 download_file 函数进行文件下载，并在下载完成后关闭连接并释放相关资源。

（2）fclose 接口实现

fclose 函数是 C 标准库中的一个函数，用于关闭打开的文件流，此处我们对其进行修改，增添文件未修改从而删除的功能。

fclose 函数的作用是关闭一个打开的文件流。它首先调用 close_fd_list 函数，该函数用于处理文件指针的相关操作，包括从 fd_list 和 open_list 中删除相应的条目。然后，它调用实际的 fclose 函数 fclosep 来关闭文件流。fclose 函数最终返回 fclosep 函数的返回值。

首先，初始化变量 back，用于 fclose 返回值。使用 while 循环检查 fd_mutex 是否为 0，如果是，则等待 1 秒。将 fd_mutex 设置为 0，以确保在使用 fd_list 期间不会被其他线程修改。调用 find_fd_list_by_stream 函数找到文件指针 stream 在 fd_list 中的索引。调用 del_fd_list_by_index 函数通过索引从 fd_list 中删除文件指针。将 fd_mutex 设置为 1，表示释放对 fd_list 的访问控制。使用 while 循环检查 open_file_mutex 是否为 0，如果是，则等待 1 秒。将 open_file_mutex 设置为 0，以确保在使用 open_file_list 期间不会被其他线程修改。调用 find_file_list_by_name 函数找到文件名在 open_list 中的索引。将该文件的 link 值减 1，并将结果存储在 link 变量中。打印文件指针 stream 的值和 link 的值。

如果 link 为 0，表示该文件已被关闭：

a. 将 open_list_index 中对应索引位置的值设置为 0，以标记该位置为空闲。

b. 将 open_list_count 减 1，表示打开的文件数目减少。

c.检查 stream 是否为 NULL，如果不为 NULL，则表示该文件存在。

d.如果该文件是本地文件且指纹未发生变化，删除该文件。

e. 如果指纹发生变化，不删除该文件。

将 open_file_mutex 设置为 1，释放对 open_file_list 的访问控制。调用 fclosep 函数，实际上关闭文件流。使用 dlsym 函数获取 fclosep 函数的地址。检查地址获取是否成功，如果失败，则输出错误消息并退出程序。调用 close_fd_list 函数，传递文件流__stream 和 fclosep 函数的地址。将 close_fd_list 函数的返回值存储在 back 变量中。打印一条消息，指示 fclose 函数执行成功。最后返回 back 值，调用结束。

在 close_fd_list 函数中，除了处理文件指针相关的操作外，还会检查文件是否为本地文件，并根据文件的指纹是否发生变化来决定是否删除文件。

（3）其他文件写入接口实现

其他文件写入接口与原 C 库的实现相比没有进行修改，原因是，fopen 函数用于远程文件传输后会将文件保存在本地。由于 C 语言中对任何文件的读写操作都需要先打开文件，然后再进行读写操作，因此文件写入操作无需关心文件是来自本地还是远程，只需要根据对应路径找到文件进行读写即可。因此，在这里的文件写入接口与原始 C 库的实现方式相同，没有进行修改。

4.3 Metadata node 模块实现

元数据节点（本文中又称中心节点）主要负责存储管理全局文件指纹表和发送相关重删指令到 Data node（设备节点）。

中心节点在系统中充当服务器的角色。服务器通过创建和管理线程来处理每个客户端连接，从而实现对多个客户端请求的并发处理，从而提高了服务器的并发性能。每个线程承担特定的处理逻辑，包括接收元数据、查询表格、发送指令等操作。此外，服务器使用互斥锁确保对全局指纹表的访问安全。

在并发处理能力方面，通过创建线程来处理客户端请求，服务器能够同时处理多个客户端连接，从而提高了服务器的并发处理能力。每个线程独立执行特定的逻辑，彼此之间不会相互阻塞，因此可以加快请求的响应速度。

在资源管理方面，中心节点的服务器采用动态管理客户端连接和线程的方式，避免了资源的浪费和泄露。每当有新的客户端连接时，会创建一个线程来处理该连接，线程完成任务后会被回收和释放。这种方式可以合理利用系统资源，提高服务器的稳定性和可靠性。

在并发安全性方面，代码采用互斥锁确保线程对共享资源的并发访问安全。线程在访问共享资源之前先获取互斥锁，在完成操作后释放锁，从而保证了线程之间的互斥访问，防止了数据竞争和冲突的发生。

最后，服务器还支持调试和监控功能，通过打印 client_sockets 和 thread_ids 数

组的内容，可以方便地查看当前活动的客户端连接和线程。这提供了一种简便的调试和监控手段，有助于诊断问题和监测服务器的状态。

尽管当前的服务端架构已经能够实现一定的并发性能和资源管理，但还存在改进的空间。目前的实现方式是每当有新的客户端连接时就创建一个线程来处理，任务完成后再回收线程。这种方式可能导致线程频繁地创建和销毁，造成一定的资源开销。改进的方法是引入线程池管理机制，预先创建一定数量的线程，然后根据需要将任务分配给线程进行处理。这样可以减少线程的创建和销毁次数，提高线程的重用性和效率。在并发数据结构保护方面，互斥锁被用于保护对全局指纹表的并发访问。然而，互斥锁在高并发场景下可能成为瓶颈，限制了系统的并发性能。一种改进的方法是使用并发数据结构，例如并发哈希表或并发队列，来替代互斥锁。这些数据结构能够提供更细粒度的并发控制，从而提高系统的并发性能。通过引入线程池管理和并发数据结构等改进措施，可以进一步提升代码架构的并发性能、资源管理能力和可扩展性。这些改进方向将有助于构建更高效、稳定和可靠的服务器系统。

4.4 本章小结

本章详细介绍了系统中的 Metadata Node 模块和 Data Node 模块，以及它们下面的子模块的实现细节。我们通过功能流程图和调用栈的形式，清晰地展示了系统各个模块之间的功能和交互方式，帮助读者更好地理解系统的工作原理。

在 Metadata node 模块中，我们介绍了元数据的管理和维护过程，包括文件信息的存储和索引，以及元数据的更新和查询操作。

在 Data node 模块中我们介绍了文件扫描和新增文件打包发送的过程，并着重介绍了自定义库的实现方式。

接下来，我们将进行系统功能的测试，并展示系统的文件去重效果和性能开销。我们将使用不同类型和大小的文件进行测试，验证系统的去重功能是否正常工作。同时，我们将测试系统在文件扫描时的性能表现，评估系统周期扫描的开销。

通过测试和性能评估，我们将全面了解系统的性能优劣和可扩展性。同时，我们也将发现系统中可能存在的问题和改进的空间。这些测试结果和分析将有助于我们进一步优化和完善系统，提供更好的文件去重服务。

5 系统测试

在本章节中，我们将所实现的系统迁移至 OpenHarmony 开发板串口调试，并且展示去重效果和部分去重过程中的系统开销。

5.1 系统上板调试流程

本课题所使用到的硬件设备为 OpenHarmony 润和大禹系列 HH-SCDAYU200 套件，所用的开发板为 RK3568 开发板。RK3568 开发板基于 Rockchip RK3568 芯片，集成双核心架构 GPU 以及高效能 NPU；搭载四核 64 位 Cortex-A55 处理器，采用 22nm 先进工艺，主频高达 2.0GHz；支持蓝牙、Wi-Fi、音频、视频和摄像头等功能，拥有丰富的扩展接口，支持多种视频输入输出接口；配置双千兆自适应 RJ45 以太网口，可满足 NVR、工业网关等多网口产品需求。开发板样例如图 5.1 所示，开发板详细参数如表 5.1 所示：

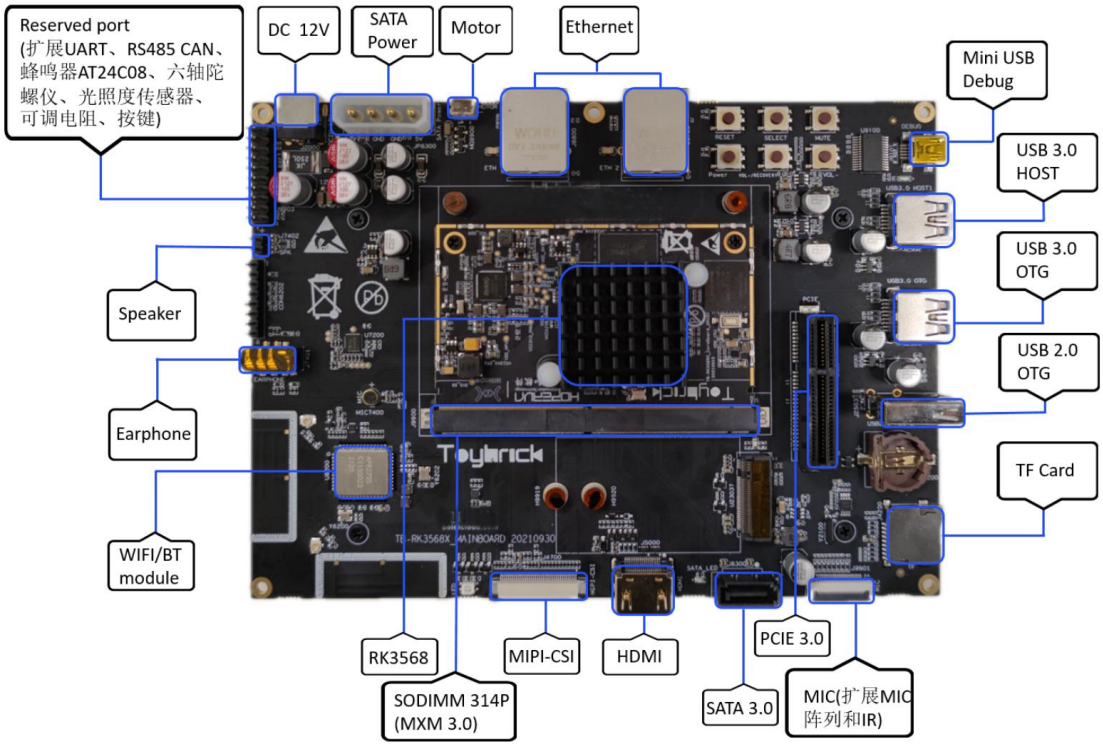

图 5.1 RK3568 开发

表 5.1 RK3568 开发板详细参数表

Model	HH-SCDAYU200
CPU	RK3568
RAM	2BG LPDDR4x
Storage	32GB EMMC
Port	SODIMM-314P

5.1.1 编译与烧录

OpenHarmony 系统编译使用华为开发的 Device-tool 工具自动化编译。下载 OpenHarmony3.1 源码后使用上述工具导入工程，依照：Build > Upload >Monitor 顺序完成操作系统自动化编译烧录。

课题所实现的分布式文件去重系统编译需要使用交叉编译。由于开发板为 arm64 架构，在 linux 虚拟完成编写后需要使用 arm64 编译器交叉编译上板。编译生成的 64 位二进制文件使用 PC 端工具 HCD 发送至开发板。 烧录完成后效果如图 5.2 所示。

图 5.2 烧录后开机

5.1.2 调试

本课题使用 IPOP v4.1 工具进行串口调试。 IPOP 软件是一个华为 IP 工具的集合，最原始的功能是 IP 地址动态绑定，后续在此基础上不断的进行了扩充，就形

成了现在的版本。IPOP 是一款实用强大的网络设置工具。 在调试过程中，IPOP 主要用到终端工具模块，终端工具用于 telnet 远程设备管理网口/连接串口使用。

这里使用终端工具绑定 COM3 端口，带开发板重启开机后，终端显示如下：

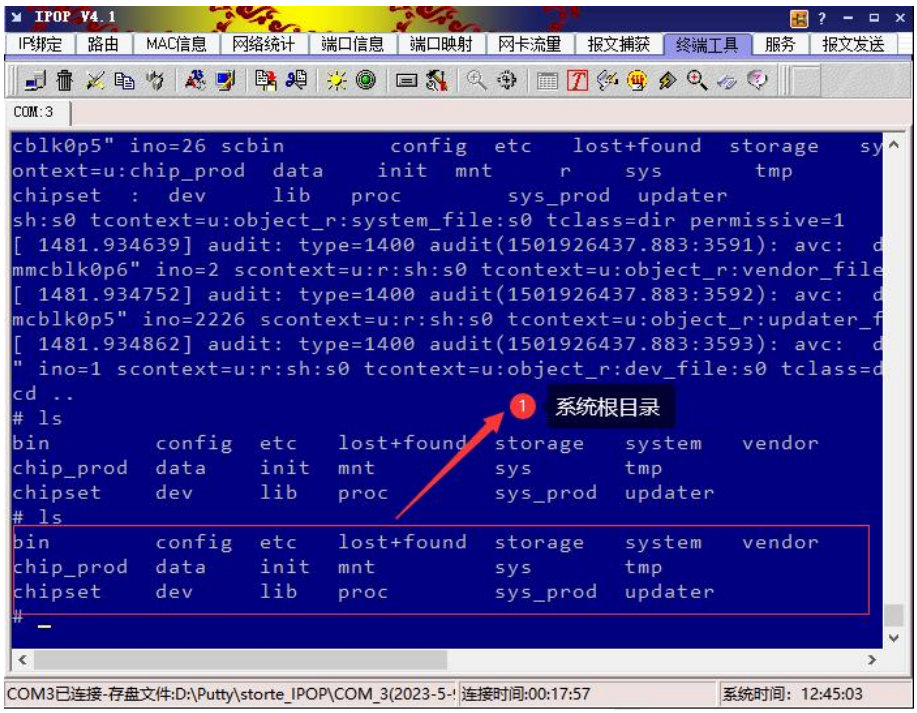

图 5.3 串口调试终端显示系统根目录

这里可以看到 OpenHarmony 操作系统的根目录情况，我们的分布式去重系统将存在 data 目录下运行。

5.2 系统去重效果测试

本实验使用了 3 个 Data node 和 1 个 Metadata node 来验证去重效果。其中，Metadata node 作为中心节点服务器运行在主机上，而 Data node 则使用 OpenHarmony 开发套件进行模拟。以 node1 为例，其挂载目录下的文件组成如图 20 所示。系统按照如下步骤启动执行：

① 启动 Metadata node（中心节点），监听数据节点消息，见图 5.4；

② Data node(数据节点)载入配置文件；

③ 文件扫描开启，见图 5.6。

系统启动完成后，按照第二章节中讲述的去重方式进行去重。

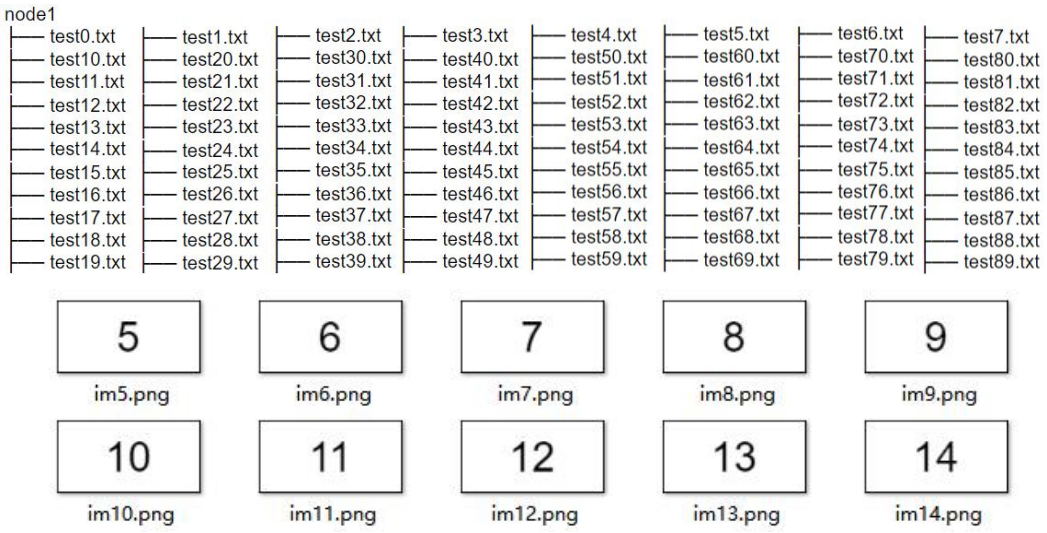

图 5.4 node1 挂载目录下的 100 个随机文本文件和图片文件

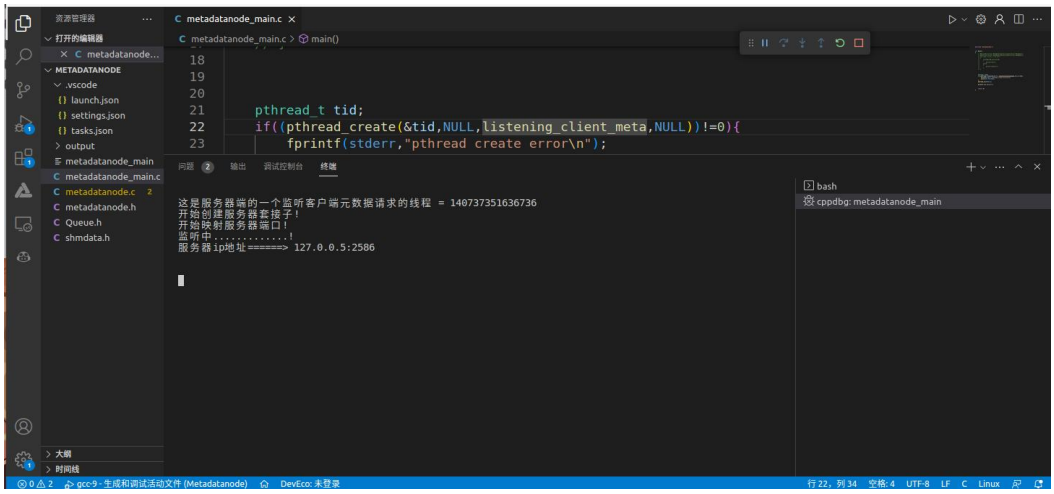

图 5.5 中心节点启动，监听设备节点

图 5.6 node1 文件扫描开启

经过一个周期的去重后，各个节点挂载目录下的文件对比：

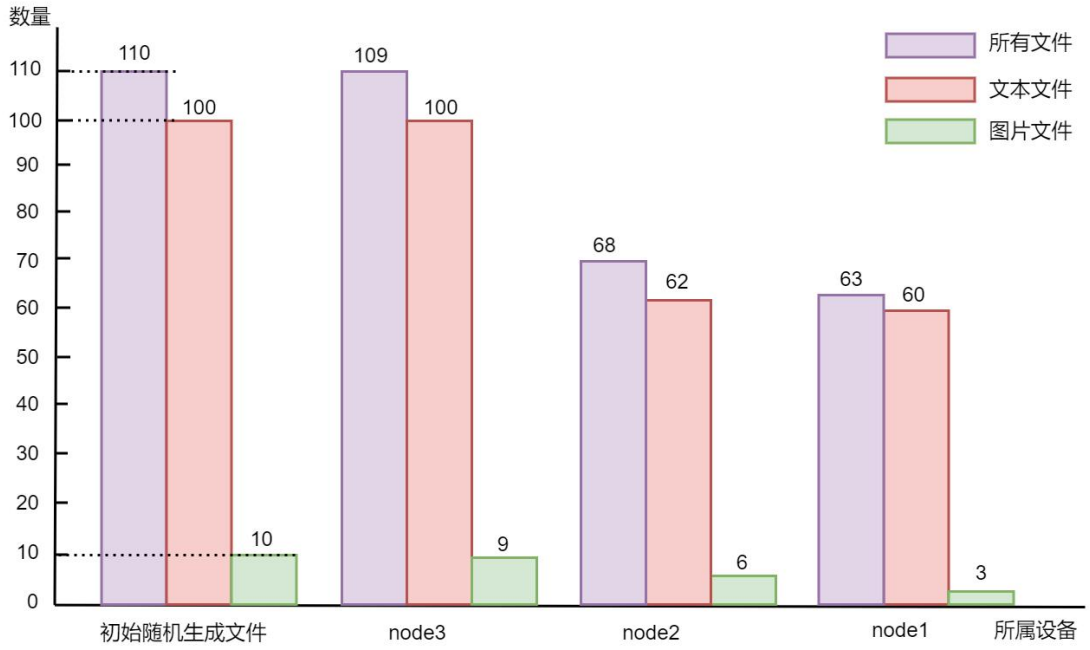

图 5.7 各个挂载节点去重效果对比

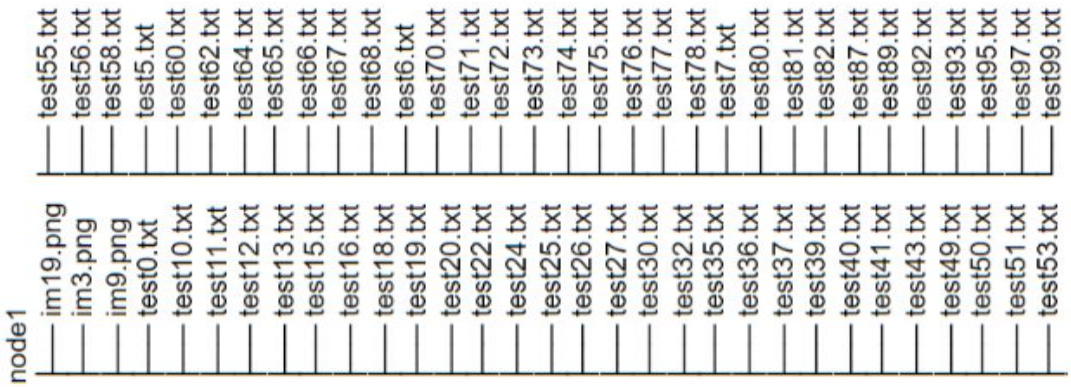

图 5.8 node1 去重后挂载目录文件情况

根据图 5.7 和图 5.8 的数据，我们可以观察到在一轮重复文件删除后，node1 和 node2 的重复文件数量大幅减少。在重复文件删除之前，node1 的总文件数量为 110 个，其中文本文件数量为 100 个，图片文件数量为 10 个。经过重复文件删除后，总文件数量减少到 63 个，文本文件数量减少到 60 个，图片文件数量减少到 3 个，分别相比之前减少了 40.0%和 70.0%，总体减少了 42.7%。对于 node2，在重复文件删除后，总文件数量减少到 68 个，文本文件数量减少到 62 个，图片文件数量减少到 6 个，分别相比之前减少了 38.0%和 40.0%，总体减少了 38.2%。由于 node3 作为 node1 和 node2 的重定向节点，因此文件数量基本保持不变。从整个系

统的角度来看，在一轮扫描后，系统中总共产生了 330 个文件，经过重复文件删除后剩余 240 个文件，总体减少了 27.3%。

为了模拟更真实的应用场景，我们构建了一个规模较小的分布式文件数据集，专门用于评估在分布式系统中数据去重的效果。该数据集由多个文件类型组成，且在实际运行时更具有随机性。具体而言，我们包含了 100 个随机内容的 txt 文本文件、20 个标识数字 1~20 的 PNG 图片文件、20 个视频文件、20 个 bin 二进制文件和 20 个图片 zip 压缩文件。为了进行测试，我们采用了一个中心节点和三个设备节点的配置来模拟实际环境。测试过程采用了一轮扫描的方式，通过多次检查每个设备节点的平均去重效果来评估系统性能。见表格 5.2 和表格 5.3。

表 5.2 一轮扫描去重前系统中数据情况

文件类别	所有文件	node1	node2	node3
txt	300	100	100	100
bin	60	20	20	20
mp4	60	20	20	20
pgn	60	20	20	20
zip	60	20	20	20
合计	540	180	180	180

表 5.3 一轮扫描去重后系统中数据情况

文件类别	所有文件	node1	node2	node3
txt	236	68	70	98
bin	47	13	14	20
mp4	43	11	13	19
pgn	44	12	13	19
zip	42	12	10	20
合计	412	116	120	176

表 5.2 和表 5.3 展示了经过一轮扫描去重前后系统中的数据情况。表格分为两个部分：去重前系统中数据情况和去重后系统中数据情况。每个部分都包含了文件类别和所属设备的信息，并统计了相应的文件数量。

在去重前的系统中，包括了不同类型的文件（txt、bin、mp4、png 和 zip）以及它们在各个设备节点上的分布情况。总共有 540 个文件，其中 txt 文件有 300 个，

bin、mp4、png 和 zip 文件各有 60 个。这些文件在 Node1、Node2 和 Node3 上均匀分布，每个设备节点上都有相同数量的文件。

经过一轮扫描后，系统中的文件数量发生了变化，表明成功去重了一些重复的文件。总文件数量减少至 412 个。在各个文件类别中，每个设备节点上的文件数量也发生了变动。txt 文件减少至 236 个，bin 文件减少至 47 个，mp4 文件减少至 43 个，png 文件减少至 44 个，zip 文件减少至 42 个。针对每个设备节点和整体系统的去重操作效果，我们进行了去重率的计算与评估。

首先，我们计算了每个设备节点的去重率。去重率是通过计算去重后的文件数量与去重前文件数量之间的比值来获得。

对于 node1 设备节点，去重率为 35.56%，计算方式为 (180 - 116) / 180 * 100%

对于 node2 设备节点，去重率为 33.33%，计算方式为 (180 - 120) / 180 * 100%

对于 node3 设备节点，去重率为 2.22%，计算方式为 (180 - 176) / 180 * 100%

接着，我们计算了整体的去重率。整体的去重率是通过计算去重后的文件数量与去重前文件数量之间的比值来获得的。

整体去重率为 23.70%，计算方式为 (540 - 412) / 540 * 100%。

根据计算结果，node1 设备节点的去重率为 35.56%，Node2 设备节点的去重率为 33.33%，node3 设备节点的去重率为 2.22%。整体的去重率为 23.70%。

根据上述数据，我们可以观察到 node1 和 node2 在去重效果上表现出明显的优势，而 node3 的去重效果较差。这可以归因于系统架构设计中需要使用重定向节点的原因，而 node3 恰好扮演了 node1 和 node2 的重定向节点的角色。重定向节点的存在表明在系统进行数据重定向时需要采用负载均衡策略。一个良好的负载均衡策略能够分担文件传输负载，避免某个节点成为系统性能瓶颈的作用。

但基于提供的数据，我们可以可靠地得出结论：经过一轮扫描的去重操作，在测试的分布式文件系统中，成功减少了系统中的重复文件数量，并且不同设备节点表现出不同的去重效果。整体而言，去重操作在系统中取得了一定的效果，提高了数据的去重性能。

5.3 系统时间开销分析

本小节的重点是对设备节点中的新增文件扫描模块进行开销测试。我们自行创建了文件数据集，并进行了挂载测试，其中目录中的大部分文件都是普通文件。在测试过程中，我们对不同文件数量的目录进行扫描，并记录挂载所需的时间。测试分为两种方式：不计算指纹的扫描和计算指纹的扫描，我们对比了两种方式在时间上的开销。测试结果是通过对四个文件数据集进行多次测量并取平均值得出的，详见图 5.9。

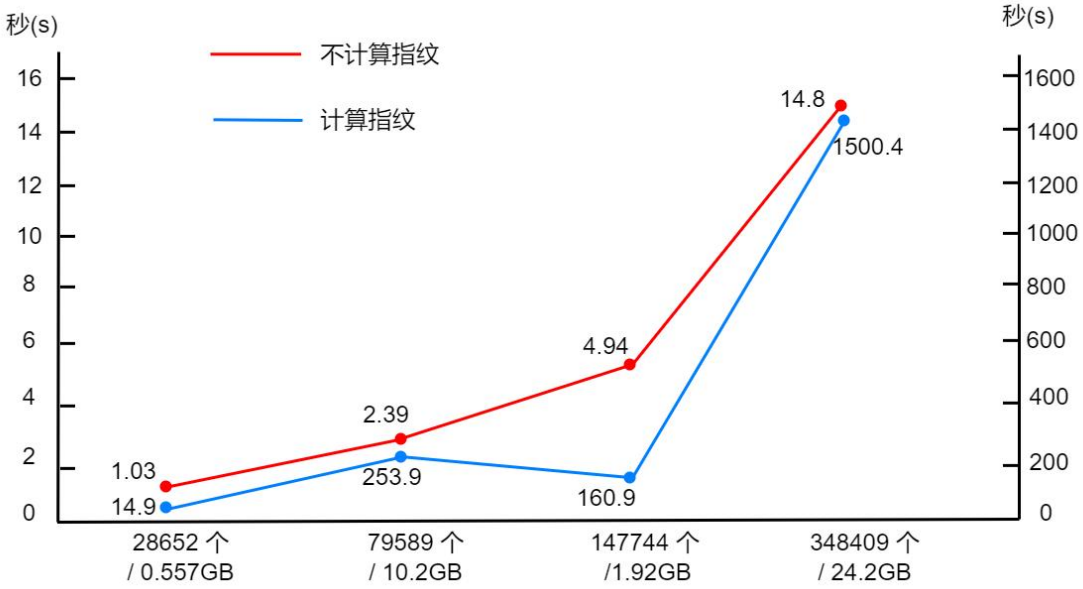

图 5.9 扫秒计算指纹和不计算指纹开销对比

根据图表分析可得出以下结论：在不计算指纹的情况下，扫描模块的总体开销较低，即使对 24.2GB 的文件进行扫描，所需时间仅为 14.8 秒。在极限情况下，假设挂载目录下的文件全都是新增文件且需要计算指纹，则开销的花费如蓝色折线所示。

在扫描周期为 1 小时的前提下，针对四个文件数据集，扫描开销的占比分别为 0.41%、7.12%、4.53%和 41.68%。这表明我们的系统在少量新增文件的场景中时间开销很低，最多不超过 10%。这个实验结果符合我们在小型设备网络内进行小批量文件去重的初衷。以上结果证明了我们系统的高效性和良好的性能表现。

5.4 本章小结

本章介绍了系统编译、上板调试的流程，并展示了分布式文件去重的效果。首先，我们介绍了系统的编译和上板调试过程，确保系统在目标设备上能够正常运行。

接着，我们展示了系统的分布式文件去重功能。通过在客户端使用共享内存和重定向表，实现了多个设备之间的文件去重，并展示了文件去重的效果。通过减少重复文件的存储，系统可以节省大量的存储空间，提高存储效率。

最后，我们对系统的扫描模块进行了性能分析。通过测试小型设备网络内执行小批量文件去重任务的时间开销，我们发现系统的性能表现良好，开销较低。这说明系统在实际应用中具有很高的可行性和实用性。

下一章中，我们将对整个项目进行总结与展望。我们将回顾项目的成果和亮点，并探讨可能的改进和扩展方向。通过总结与展望，我们可以为后续的项目开发和优化提供有益的指导和思路。

6 项目总结与展望

在本项目中，我们设计并实现了一个基于 OpenHarmony 的分布式文件去重系统，该系统采用了中心化分布式架构，包括 Metadata 节点和多个 Data 节点。系统通过计算文件指纹并比对实现文件去重的功能。以下是对项目的总结：

总结部分，本项目的分布式文件去重架构具有以下优点：

（1）高效的存储利用率：通过去重技术，系统能够检测和删除重复的文件，大大减少了存储开销。重复的文件只保留一个副本，节约了存储空间。

（2）分布式处理能力：系统采用中心化分布式架构，可以在多个 Data 节点上同时进行文件指纹计算和比对操作，提高了系统的处理能力和效率。同时，Metadata 节点负责协调和管理各个 Data 节点的工作，实现了任务的分发和结果的汇总。

在项目实施过程中，我们运用了多种关键技术，包括 IPC 技术、库打桩技术和文件加密传输技术。这些技术为系统的设计与实现提供了坚实的基础，保证了系统的可靠性和安全性。

实验结果部分，通过对实验结果的分析，我们得出以下结论：

系统的文件扫描模块表现出了较低的时间开销。即使在扫描大规模文件的情况下，系统的扫描处理时间占扫描周期时长比重较低，表明系统在小型设备网络内执行小批量文件去重任务的效果良好。

在重删过程中，系统成功减少了大量的重复文件数量。通过一轮重删操作，节点的重复文件数量明显减少，总体上降低了存储开销。

展望部分，尽管系统在文件去重方面取得了良好的效果，但在分布式负载均衡方面还有所欠缺。在未来的改进中，可以考虑引入一种可靠的负载均衡算法，以平衡各个节点的负载并提高系统的整体性能。

另外，还可以进一步优化系统的并发处理能力，以提高系统的吞吐量和响应速度。通过合理的任务调度和资源管理策略，可以进一步提升系统的性能和效率。

最后，本系统基于 OpenHarmony 操作系统开发，并且充分利用了 OpenHarmony 提供的架构原理和智能开发套件来实现分布式文件去重功能。

参 考 文 献

[1] OpenHarmony 开源项目技术架构[EB/OL][2023-5-23].https://gitee.com/openharmony#:~:text=Open-Harmony.

[2] CLEMENTS, A. T., AHMAD, I., VILAYANNUR, M., & LI, J . Decentralized deduplication in SAN cluster file systems. In USENIX Annual Technical Conference , 2009, (p. 8) https://www.usenix.org/legacy/events/usenix09/tech/slides/clements.pdf

[3] SCHMUCK F B, HASKIN R L. GPFS: A Shared-Disk File System for Large Computing Clusters[C]//FAST. 2002, 2(19).

[4] WEIL S A, BRANDT S A, MILLER E L, et al. Ceph: A scalable, high-performance distributed file system[C]//Proceedings of the 7th symposium on Operating systems design and implementation. 2006: 307-320.

[5] 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. 2019: 353-369.

[6] OUSTERHOUT J, GOPALAN A, GUPTA A, et al. The RAMCloud storage system[J]. ACM Transactions on Computer Systems (TOCS), 2015, 33(3): 1-55.

[7] ANANTHANARAYANAN G, GHODSI A, SHENKER S, et al. Effective Straggler Mitigation: Attack of the Clones[C]//NSDI. 2013, 13: 185-198.

[8] GHEMAWAT S, GOBIOFF H, LEUNG S T. The Google file system. ACM SIGOPS Operating Systems Review[J]. 2003,37(5):29-43.

[9] SHVACHKO K, KUANG H, RADIA S, et al. The hadoop distributed file system[C]//2010 IEEE 26th symposium on mass storage systems and technologies (MSST). Ieee, 2010: 1-10.

[10] KARUN A K, CHITHARANJAN K. A review on hadoop—HDFS infrastructure extensions[C]//2013 IEEE conference on information & communication technologies. IEEE, 2013: 132-137.

[11] WEIL S A, BRANDT S A, MILLER E L, et al. CRUSH: Controlled, scalable, decentralized placement of replicated data[C]//Proceedings of the 2006 ACM/IEEE conference on Supercomputing. 2006: 122-es.

[12] ALEXI W, CHOR B, GOLDREICH O, et al. RSA and Rabin functions: Certain parts are as hard as the whole[J]. SIAM Journal on Computing, 1988, 17(2): 194-209.

[13] 周炳. 海量数据的重复数据删除中元数据管理关键技术研究[D].清华大学,2015.

[14] 黄逸翔. 基于区块链的解决 DNS 体系单点故障问题研究[D].郑州大学 2021.DOI:

:10.27466/d.cnki.gzzdu.2021.004675.

[15] STOICA I, MORRIS R, KARGER D, et al. Chord: A scalable peer-to-peer lookup service for internet applications[J]. ACM SIGCOMM computer communication review, 2001, 31(4): 149-160.

[16] RAI, S., SHARMA, G., BUSCH, C., & HERLIHY, M. (2021). Load balanced distributed directories. Information & Computation, 285, 104700. https://doi.org/10.1016/j.ic.2021.104700

[17] BOVET D, CESATI M.Understanding the Linux Kernel[M].O'Reilly Media , 2005-11.

[18] BRYANT R, O'HALLARON D. Computer Systems A Programmer's Perspective (深入理解计算机系统 修订版) [M].龚奕利，雷迎春,译.中国电力出版社.

[19] VIEGA, J., MESSIER, M., & CHANDRA, P [M] . Network Security with OpenSSL, 2002 , https://cds.cern.ch/record/2013302?ln=fr.

[20] ABD, A. A., & FARHAN, A. K. A Novel Improvement With an Effective Expansion to Enhance the MD5 Hash Function for Verification of a Secure E-Document[J]. IEEE Access, 2020,8,80290–80304. https://doi.org/10.1109/access.2020.2989050.

[21] ARASU, A., BABCOCK, B., BABU, S., DATAR, M., & MOTWANI, R. The CQL continuous query language: semantic foundations and query execution [J]. The VLDB Journal, 2006,15(2), 121-142.

[22] YAN, N., CHEN, H., LIN, K., NI, Z., LI, Z., & XUE, H. BFSearch: Bloom filter based tag s earching for large-scale RFID systems. Ad Hoc Networks, 2022,139, 103022. https://doi.org/10.1016/j.adhoc.2022.103022.

[23] SHI, A., HUANG, T., & RUAN, G. Optimizing the File Synchronization for New FastDFS Storage Nodes Based on Load Balance [J]. ICAIIS 2021: 37:1-37:5

[24] KASAHARA, S. Performance Modeling of Bitcoin Blockchain: Mining Mechanism and Transaction-ConfirmationProcess[J].IEICE Transactions on Communications, 2021,E104.B(12), 1455–1464. https://doi.org/10.1587/transcom.2021iti0003.

[25] MANA, S. ,DARAKHSHAN, S. , NOMAN, I., et al.A Fast Converging and Globally Optimized Approach for Load Balancing in Cloud Computing [J]. IEEE Access,2023, 11: 11390-11404 .

致　　谢

本科四年的学习生涯即将结束，论文结笔的那一刻让我有一种说不出的感觉，仿佛心中有千言万语但又无处诉说，唯有将心绪化作笔墨。在这篇论文的完成过程中，有太多的人值得我一一感谢。他们的帮助和支持是我前进的动力和信心的源泉。他们与我一起分享喜悦和困难，共同见证了我的成长和进步。没有他们的支持和鼓励，我无法顺利完成这篇论文。

我首先要感谢陈咸彰导师。感谢他在整个研究过程中给予我的指导和支持。他的专业知识、丰富经验和耐心指导使我能够深入理解研究领域，并提供了宝贵的建议和意见。他的悉心指导和严谨的态度对我的研究和论文起到了至关重要的作用。我还要感谢实验室的师兄师姐们。感谢他们在实验和技术方面的指导和帮助。他们的经验和知识对我的研究起到了重要的启发和指引。在本科毕业后我将正式加入陈老师的科研团队，我愿怀着满腔的热情和对科研的无限向往，将全身心地投入到这个科研团队，与优秀的研究者们共同探索知识的辽阔领域。

我要感谢我的家人。感谢他们对我学业上的支持和鼓励，让我能够专注于学习并克服困难。他们一直是我最坚实的后盾，无论我遇到什么困难和挑战，他们总是在我身边给予我无尽的爱和支持。

此外，我还要感谢我的同学们。感谢他们在学习和研究中的合作与支持。我们一起度过了忙碌的日子，共同面对着学术和技术的挑战。我们相互鼓励、相互帮助，在思想碰撞中不断成长和进步。

我还要特别感谢 OpenHarmony 操作系统的开发团队和社区。感谢他们为国产操作系统的生态环境的构建和社区维护所做的努力。OpenHarmony 为我提供了一个创新的平台，使我能够深入研究和开发分布式文件去重系统。在这个过程中，我得以学习和应用最新的技术，拓宽了我的视野和技能。

最后，我要感谢身边的朋友和所有帮助过我的人。感谢他们在我低落时的鼓励和支持，感谢他们在我困惑时给予的帮助和建议。他们的存在使我感到温暖和勇气，给予我前进的动力。

在未来的学术和职业道路上，我将继续努力学习和成长，为科学研究和技术发展做出贡献。我希望能够利用所学知识和技能，不断创新和探索，为社会的发展和进步贡献自己的力量。

原创性声明

郑重声明：所呈交的论文（设计）《基于 OpenHarmony 分布式文件去重系统设计与实现》，是本人在导师的指导下，独立进行研究取得的成果。除论文（设计）中已经标注引用的内容外，本论文（设计）不包含其他人或集体已经发表或撰写过的作品成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果，并承诺因本声明而产生的法律结果由本人承担。

论文（设计）作者签名：李为

日期：2023.06.11

版权使用授权书

本论文（设计）作者完全了解学校有关保留、使用论文（设计）的规定，同意学校保留并向国家有关部门或机构送交论文（设计）复印件和电子版，允许论文（设计）被查阅和借阅。本人授权重庆大学将本论文（设计）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、缩印或扫描等复制方式保存和汇编本论文（设计）。

本论文（设计）属于：

保　密 □　在____年解密后适用本授权书

不保密 ☑

论文（设计）作者签名：李为　　指导教师签名：陈成彩

日期：2023.06.11　　日期：2023.06.11