東南大學

毕业设计(论文)报告

题　目 基于 GFS 的分布式文件系统设计和实现

软件学院　院（系）　软件工程　专业

学　　号 71112318

学生姓名 黄　鑫

校内导师 徐造林

企业导师 孙永跃

起止日期 2016 年 2 月到 2016 年 5 月

设计地点 深圳市迅雷网络技术有限公司

摘要

Google 公司发表的论文《The Google File System》是业界著名的分布式文件系统理论支撑之一。根据此论文和其它相关开源项目、研究课题，许多互联网公司相继开发出自己的分布式文件系统，不同的针对性优化使得这些分布式文件系统无论是在可用性、扩展性、性能，还是在数据管理、数据存储方面都各具优势。理论上而言，在不同的应用场景，不同的存储架构，不同的性能考量下不可能存在一个通用的性能优化方案，所以根据实际需求对一个系统进行针对性性能优化是非常必要的。

本论文首先介绍了包括 Google File System 在内的一些国内外具有实践基础的分布式文件系统，并根据《The Google File System》论文对分布式文件系统做了一个相对完整的研究，在元数据管理方案上作了详细的分析，然后结合研究成果和实际项目需求，重点介绍了系统的设计细节，对基于 GFS 的分布式文件系统进行了小文件支持、副本压缩、缓解单点问题、存储引擎改进等针对性优化，最后简要说明了系统的实现。

关键词：分布式文件系统；性能优化；Google File System；副本压缩

Abstract

The Google File System is one of the most famous theory of distributed File System. According to the paper and other related open source projects, many internet company have developed their own distributed File System. The different targeted optimization made these distributed File Systems have different advantages in the data management, data storage, availability, scalability or performance. In theory, there is no way to find a common performance optimization scheme for different application scenarios, different storage architecture, or different performance consideration. Therefore, it is necessary to do the specific performance optimization according to the actual demand.

This paper firstly introduces some distributed File Systems with practical foundation both in China and abroad, including Google File System, then makes a relatively complete research of distributed File System according to the paper *The Google File System.* Then, an analysis in metadata management solution is made for detail. And then, combined with the research results and the actual demand, the system design details with performance optimization in replica compression, single point problem, storage engine and small file support is mainly proposed. Finally, the paper simply explains the implementation of the system.

Keywords: distributed File System; performance optimization; Google File System; replica compression

目录

第 1 章 前言

1.1 课题背景

随着互联网服务在社会生活中越来越全面的渗透，互联网时代也正式迈向大数据时代，无数的数据由用户产生，服务提供商也意识到这些数据的价值，爆炸式增长的数据怎么保存下来，如此庞大的数据怎么有效的组织利用，普通企业如何获得低成本且足以匹敌大型主机的计算能力来挖掘这些数据，成为了互联网从业者们不断研究的课题。

分布式一直都是计算机领域的一个很前沿的课题，分布式的出现，使得一些只能在传统大型主机上做的事能用很廉价的集群实现。对于企业应用而言，分布式文件系统最大的优势是可扩展性，可以通过增加或减少集群中机器数量来增加整个系统的计算性能或降低运营成本，以应对业务的变化。

Google File System 正是 Google 针对自己的业务需求研发出来的一个分布式文件系统。它运行于普通硬件集群而不是大型机上，具有良好的扩展性和容错性，为全世界用户提供高质量的网络服务。像 Google File System 一样的分布式文件系统的应用非常广泛，它是属于一类产业的基础，像是云计算、云存储、CDN，甚至未来的人工智能也离不开分布式文件系统对它们海量数据的支撑。

Google 公司发表的论文《The Google File System》算是业界著名的分布式文件系统理论支撑之一，业界也有很多基于该论文的实践，比如著名的 HDFS。通过此论文和各种相关开源项目，很多互联网公司也相继开发出自己的分布式文件系统，不同的针对性优化使得这些分布式文件系统无论是在可用性、扩展性、性能，还是在数据管理、数据存储方面都各具优势。理论上而言，在不同的应用场景，不同的存储架构，不同的性能考量下并不可能存在一个通用的性能优化方案，所以对于一个互联网企业来说，根据实际需求设计和定制分布式文件系统是非常必要的。

1.2 分布式文件系统现状

随着整个互联网行业的发展，实现大规模并行计算成为了各项互联网服务的门槛，传统的利用硬件实现文件的并行存储，虽然管理简便，但由于要受限于硬件的单价，成本太高，普通小型企业或是初创公司并不能负担，而利用互联网交换数据来构建廉价集群也可以达到

相同目的。随着不断提升的网络带宽和不断下降的网络价格，许多利用网络带宽和速度，以提高系统存储容量、并发 I/O 性能和可扩展性为目的而设计的分布式文件系统相继出现。

分布式文件系统由来已久，最早可追溯到上个世纪八十年代的 NFS 和 AFS，发展至今，各种需求驱动下的实践与研究使得分布式文件系统的理论不断成熟。Google File System 虽然是一类很经典的分布式文件系统理论，但距今已有十多年的历史，目前谷歌公司早已过渡到了 Colossus，又被称作 GFS2 的新一代分布式文件系统。不同体系的分布式文件系统理论被相继提出，适用于不同领域、不同需求。一些还处于验证阶段，一些却早已经历过实践，投入了生产。下面将会介绍一些国内外具有实践基础的分布式文件系统。

1.2.1 GFS

《The Google File System》论文发表于 2003 年[1]，对当时的分布式文件系统的设计思路是一个比较大的启发。正如 Google 工程师所言，他们重新审视了传统文件系统在设计上的折衷选择，基于对自己的应用的负载情况和技术环境的观察，衍生出了完全不同的设计思路[1]。GFS 的设计思路与以下实际问题相辅相成：由于 GFS 被设计在普通硬件上，组件失效被认为是常态事件；文件数量非常巨大，文件一般是大于 100M 的大文件；文件一旦写入不需要修改；客户端程序和文件系统接口需要协同设计以便不同业务的应用。GFS 采用了一种控制与业务相分离的架构，这种架构在许多流行的软件产品上使用得较为广泛，也较为成功。具体细节将在第 2 章详述。

1.2.2 HDFS

HDFS(Hadoop Distribute File System)作为《The Google File System》论文的一个最重要的实现，设计目标和 GFS 是高度一致的，并且在整体架构、元数据服务、副本策略上都与 GFS 大致相同[2]。不过，在一些功能上还是跟 GFS 有所差别。如：

快照：GFS 是提供快照功能的，利用写时复制的机制快速生成一个文件或者目录的复制，对性能开销很小。而最初的 HDFS 版本并不支持快照，在 HDFS2.x 版本开始才新增了快照功能。

记录追加操作：HDFS 不支持 GFS 论文中描述的记录追加和并行写操作。

垃圾回收：GFS 的垃圾回收策略是用标记的方式记录文件是否被删除，这是一种惰性回收策略，被删除标记的文件不能被普通用户访问。系统会定期回收垃圾，删除一个

规定时间前的被删除标记的文件。这样可以减小开销和防止误删。最初版本的 HDFS 并没有垃圾回收机制，而是直接删除。HDFS 在之后的版本加入了这个功能，但是实现方式与 GFS 有一定的出入，HDFS 会把删除的文件路径改在/trash 目录下，类似 windows 的回收站，而不是像 GFS 一样采用标记的方式。

1.2.3 XFS

XFS 是腾讯台风云计算平台中的分布式文件系统，也是《The Google File System》论文的一个实现。不同于 GFS 的是，XFS 根据实际需求，设计了一套主备机制，通过维持多个 Master 副本，使得系统可以在主 master 故障时迅速切换到备用服务，提升了分布式文件系统的可靠性，缓解了 GFS 与 HDFS 面临的单点故障问题，但是也付出了性能的代价，主要是在主备 master 的一致性维护上。

GFS 的单 Master 架构虽然很大程度上简化了系统，这也让 Master 成了单点：只要 Master 发生故障，整个集群就会停止工作，并且单点的性能也可能会成为瓶颈。HDFS 沿袭了 GFS 的设计，也有单点问题。

XFS 的主备机制由三套协议驱动：Replication 协议、failover 协议、learning 协议[3]。

Replication 协议负责对所有的写操作执行两段式提交，确保主备 master 的元数据一致性。Failover 协议负责在主 master 故障时主导从备份 master 中选举出新 master，并完成文件系统向应用服务提供的 server 的平稳迁移，保证了系统容错性。Learning 协议负责新加入的 master 的状态同步，以及自动修复发生故障的 master。

1.2.4 TFS

Taobao File System 是淘宝开发的一个分布式文件系统，针对海量小文件的随机读写做了特殊优化，承载着淘宝主站所有图片、商品描述等数据存储[4]。由于 TFS 前端有两层缓存系统来保证用户展示页面的速度，最终能到达 TFS 的请求大约只有总请求数量的 2%左右，随着前端缓冲的效率不断提升，这个比例还会继续下降，TFS 面临着完全随机、基本上没有热点的数据访问。

为了解决随机访问、小文件问题，TFS 采用直接基于块设备的自建文件系统，而不是 Linux 文件系统，减少 EXT3 文件系统的性能损耗；抛弃了目录结构，采用完全扁平化的数据组织；对每块磁盘都用一个单独进程管理，以保证 I/O 的并发性能。

1.2.5 Ceph

Ceph[5]是一个典型的非集中式的分布式文件系统，它的 master 节点采用集群的方式有效的避免了单点问题，并且一般的类 GFS 的分布式文件系统通常是采用两层命名空间，第一层是文件系统和文件到 Chunk 块的映射信息，第二层是 Chunk 块对应的副本位置信息，这两层命名空间尤其是第二层副本位置信息占据了大量元数据内容。在 Ceph 中的元数据只有第一层命名空间和文件映射信息，通过文件信息和 CRUSH 算法，Ceph 可以在任何地方包括客户端计算出全部副本到数据节点的映射关系，这样为整个系统节省了大量元数据存储与传输，提供了高扩展性。

1.2.6 BWFS

蓝鲸分布式文件系统（BWFS）是国家高性能计算机工程技术研究中心基于对国内外现有研究成果的分析和研究，自主设计实现的分布式文件系统。它着重于大容量、高 I/O 吞吐率和高扩展能力等方面特性[6]。

BWFS 抛弃传统集中式元数据管理方案，实现了自定义的分布式分层模型[7]，元数据管理方案中采用了创新的分布存储、集中控制的元数据管理架构[8]，多个元数据服务器和一个应用服务器、一个绑定服务器共同组成了元数据服务器集群。所有元数据分布存储在元数据服务器中，由绑定服务器统一决策管理，元数据分布以用户访问数据为参数，采用了动态子树划分策略，数据节点和元数据服务器之间的元数据交换只在元数据服务器集群中的一个特殊的应用服务器与数据节点之间进行，所有元数据服务器通过应用服务器这个中介与数据节点进行数据交互。

1.2.7 GFS2

GFS2(Google File System 2)代号 Colossus，是 Google File System 的第二代产品，但是目前 Google 并没有像 GFS 一样公开其技术细节。根据一些对 Google 技术人员的访谈资料可以了解到，Colossus 已通过分布式 master 服务解决了 GFS 的单点问题，通过把 chunk 块大小由 64M 缩小到 1M 解决了对小文件不友好的问题。但是理论上来讲，chunk 块大小缩小 64 倍，相应的元数据大小会增加到接近 64 倍，而元数据全部缓存在内存中，虽然 master 服务已集群化，但对于内存消耗而言依然是个很大的问题，不过 Google 工程师提到他们已经解决了这个问题。

1.3 本文结构安排

本文由四章组成：

第一章主要介绍了课题背景、课题内容和课题关注点，并重点分析了一些目前流行的分布式文件系统。

第二章根据实际情况分析了基于 GFS 的分布式文件系统的设计需求，简要介绍了整个系统的架构，详细介绍了设计细节，包括元数据管理方案，元数据服务的设计，副本管理服务的设计。

第三章简要说明了基于 GFS 的分布式文件系统的实现，包括元数据管理服务、数据服务、客户端接口的实现。

第四章总结了基于 GFS 的分布式文件系统的设计成果，并提出今后的展望。

第 2 章 分布式文件系统的设计细节

2.1 分布式文件系统的设计需求

需求对设计起决定作用，设计的方案对整个系统的应用范围和效果有着很大的影响。在需求分析的过程中，会对整个系统的各个细节进行一次全面的分析，考虑各种综合因素，以此来选择最合适的方案，确保系统的有效性与针对性。

目前分布式文件系统流行的搭建平台一般都是 Linux，主要用途是对分布存储的文件进行统一管理。Linux 平台对于此类服务器式应拥有独天得厚的优势：稳定、高效。所以在设计与实现过程中，本文也会考虑到 Linux 系统的特性[9]。

就本系统而言，主要用于大规模文件存储，涉及到多个业务，许多系统会使用本分布式文件系统作为底层存储工具，是作为互联网服务的一体式分布式存储服务解决方案，具体需求可以分为两个层面：功能需求与非功能需求。对功能需求来说，最主要的即是分布式系统的功能和文件系统的功能。非功能需求则更多的着眼于性能、成本、安全等方面。

2.1.1 功能需求

在功能需求上，要注意的主要是对系统本身的参考以及对业务需求的参考。实际中，系统会涉及到多个业务，出于成本的考虑，倾向设计一个通用式的分布式文件系统，由配置文件可控作出一些针对性的性能优化，就实际业务而言，小文件会与大文件共存，类似 TFS 一样的全局小文件优化并不适用，否则将会降低对大文件的支持度。因此功能需求从系统需求和业务需求两方面入手。

系统功能需求是对项目最基本需求的描述，通过参考典型的分布式系统和文件系统的应用场景，可以得到以下的需求模型：

（1）数据节点的无缝加载与卸载

当有新的数据节点希望加入集群时，系统能在不改变现有内部结构的基础上无缝地、没有任何影响的让新节点参与到数据存储的工作中，并且能够平滑地控制新加入节点上的数据量过渡到平均值。当有节点希望退出集群时，这个过程也一样，系统会控制节点的数据平滑的均衡的复制到其他节点。

（2）加载路径的统一管理

集群中需要统一每个数据节点的加载路径，确保每个节点加载路径的一致性。

（3）内部资源的管理

节点的加入与退出会带来整个集群的资源变化，统一有效的管理能保证集群的资源不会泄露。

（4）全局文件的统一管理

无数的文件被分配到许多数据节点进行存储，这时候就需要一个全局统一的唯一标识来分辨每一个文件，使得所有文件可以只需要路径名称就能被正常访问。

（5）基本的文件系统操作

系统需要提供类似 Linux 的文件系统操作，整个文件空间以文件树的方式组织。具体的操作包括但不限于：目录相关操作（创建、删除、修改、拷贝、移动），文件相关操作，工作路径切换与获取等等。

业务需求是抛开系统本身最基本的需求外，根据系统的实际用途来看的。通过前文所述的用途分析，可以得到以下的需求模型：

（1）支持海量文件存储

系统涉及到的数据量可能高达数百 PB，这比十多年前 Google 预计的数百 TB 直接上升了一个数量级。

（2）大文件管理

大文件是系统中的常规存储方式，在这里定义为大于 64M 的文件，数 GB 的文件非常普遍。

（3）支持小文件

部分小于 64M 的小文件也应该采用一些特殊的方式优化存储，但不应该对正常的大文件存储产生过多的负面影响。

（4）流式读取和随机读取

读取方式主要包含大规模的流式读取和小规模随机读取。小规模的随机读取通常是读取 1M 以下的数据大小，客户端还可以把小规模的随机读取进行排序后批量处理，来优化读取效率。

（5）节点负载均衡

由于用户行为的随机性，数据节点上的不均匀负载情况十分常见，要想获得良好的性能，必须解决这个问题，不然某个负载过重的节点会影响整个系统的性能。

（6）文件备份

集群由普通的设备，文件的安全性必然是首位的，通过文件备份可以有效地规避因自然

灾害或者系统故障造成的数据损坏或丢失。

（7）故障恢复

硬件的故障无法挽回，但是程序的故障必然是常见的，并且可以挽回，必须被很快处理。无论是元数据服务还是数据节点都应该能在很短的时间内重启，并且恢复到故障前的状态。

（8）冷热数据的区分对待

正是因为用户行为的随机性，有些文件可能会被频繁地访问，而有些文件却几乎不被访问，这两种文件数据不应该被同等对待，这无论是对系统的性能或者运营成本都是不利的。

（9）提供客户端编程接口

为了适应不同业务需求，系统并不提供客户端，但是会向上层应用程序提供统一的文件系统编程接口，如文件的打开关闭，读写，权限管理，等等，各种各样的自定义客户端可以通过 API 调用对分布式文件系统进行操作，当然，是在权限允许的范围内。

（10）系统资源统计

为了方便维护，系统需要实时统计各个资源的使用状况，比如说数据节点还有多少空间可以用，其中元数据占用了多少，文件数据占用了多少，目前整个系统有多少文件，多少是冷数据，多少是热数据等等。

（11）日志管理

日志是任何系统必备的。系统的维护与修正离不开对日志的分析。一般倾向于尽可能的存储日志文件，除非是其占用的空间已达到一个可能会产生负面影响程度。

2.1.2 非功能需求

非功能需求是项目需求在除功能需求外的一个很重要组成部分，描述了对系统性能、安全、维护方面的期望，在本系统中，主要的非功能需求有以下几个方面：

（1）并行处理性能

并行处理性能在描述在大量用户（可以是不同业务）同时作业的情况下，大量客户端对系统产生大量请求，系统必须及时处理这些请求，在用户可等待时间限度内，以保证用户体验需求。

（2）最大响应延迟

最大相应延迟指对任意请求处理而言，客户端都要有一定的响应时间需求，因为客户端不可能无限等待，必须规定一个最大时限，超出时限没有处理完成，则视为请求失败。

（3）系统高可用性

高可用性是指在系统一些部件不可用时，也不会对整个系统的运行造成影响，一般通过快速恢复和备份复制等方式将出故障的部件的工作转移到其他正常部件。由于分布式文件系统的集群规模可能会达到一个很大的数量级，所以它必须将组件失效作为一种常态，能够迅速地侦测、冗余并恢复失效的组件。

（4）数据容错性

网络本身就是极不稳定的，但是整个系统是运行在互联网之上的，机群与机群之间也是通过互联网通信，所以数据传输的延迟甚至失败将会非常普遍，在这种情况下，数据的完整性也会受到相应的影响，如何保证服务器中存在不完整数据的情况下，能得到一份完整数据，是通过容错性指标来判断的。

（5）客户端安全性

在网络日益繁荣的当下，各种网络恶意攻击也无比频繁。一个合格的系统必须考虑权限相关的问题，避免被恶意客户端破坏。

（6）集群可扩展性

可扩展性主要是指系统应对不确定增长的能力。包括数据节点的增加，客户端的增加，新功能的加入等等。

（7）可靠性

可靠性描述的是整个分布式文件系统的无故障时间占比，业界一般是四九原则，即无故障时间占比 99.99%，也就是一年中故障时间不超过 0.876 个小时。

2.2 分布式文件系统的主体架构

本论文所述的分布式系统设计基于《The Google File System》论文，整体架构如图 2-1 所示：

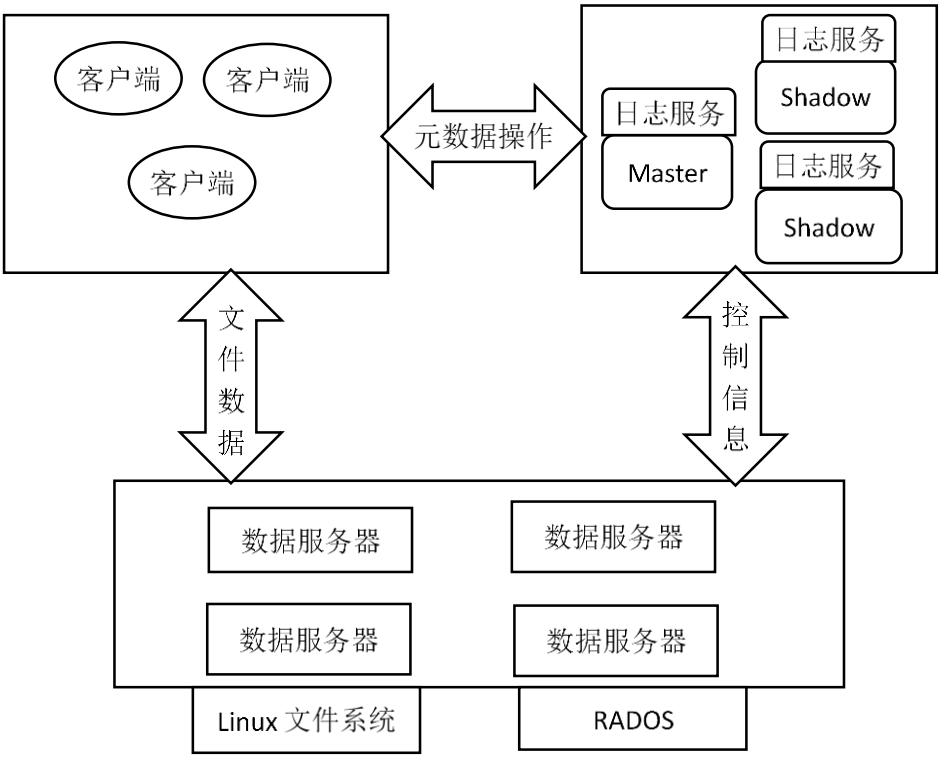

图 2-1 基于 GFS 的分布式文件系统主体架构

系统运行在 Linux 操作系统上，大体上遵从客户端/服务端架构，服务端是一个集群，为了实现控制流与数据流分离，分为数据服务(chunk server)和元数据管理服务(master server)，管理节点负责处理由客户端发来的请求，包括但不限于创建文件、复制文件、删除文件、重命名文件、读取文件、写入文件等等操作，将元数据根据请求作相应的处理，并且负责决策这些请求应该指向哪些数据节点，并把相应的元数据信息和数据节点位置返还给客户端，由客户端直接与数据节点进行数据的交换。不同于管理节点，数据节点则进行物理上的数据读写，并且数据节点之间也会进行数据的交换以达到副本的移动复制需求，保证文件系统中文件数据不会经过管理节点而对其造成额外的带宽负担。另外，日志服务(log server)和影子服务(shadow master)的存在也使得一些针对性优化能够进行。

2.3 元数据管理方案

元数据服务是整个分布式文件系统的决策中枢，也是系统最复杂关键的地方，就如同人的大脑一样。分布式文件系统的元数据服务的设计方案直接决定了整个系统的适用范围、效率、可用性等等。元数据管理主要有这几个问题：副本管理、名字空间、可用性、一致性。

一般而言，元数据分为两个部分，名字空间映射信息与副本映射信息。名字空间由目录、文件和块组成，是文件和目录组成的层次结构。支持文件系统相关的所有名字空间操作，比如创建、删除、修改、列出文件和目录。副本映射信息则包含数据块的多个副本到数据节点的位置映射信息。

元数据的管理方案通常而言主要有三种：集中式元数据管理、分布式元数据管理、无中心元数据管理[10]。当然，为了应对各种针对性优化需求，也有这三种方式混用的实践，在这里定义为第四种方案：混合式元数据管理。

集中式元数据管理方案是目前常用的方案，所有元数据的存储与操作都在一台元数据服务器上进行，控制流高度集中，GFS、HDFS、XFS、TFS 等一批商用的分布式文件系统都采用这种方式。这种方式在实践中具有很针对性的特点：开发成本低，应用效果比较好。集中式元数据管理方案简化了许多同步一致性问题，并且类似垃圾回收、读写调度、崩溃恢复等等分布式文件系统的常规功能都能很容易实现。

分布式元数据管理方案的提出是为了解决集中式元数据管理中的单点问题，将元数据分散到多个元数据服务器，从而提升元数据服务的扩展性，在这种方案下，整个分布式文件系统的扩展性得到了充分的提升，理论上来说，整个系统的存储容量可以无限增加，不再受单个元数据服务器内存容量的制约。利用一些常规手段更可以轻松实现元数据访问的负载均衡。但是这种方案的实现及其复杂，元数据集群的同步开销，性能与扩展性之间需要找到一个平衡点。GFS2 就是采用的这种方案。

无中心元数据管理方案主要基于 p2p 技术，无中心管理节点和当然也无单点问题。支持无限扩展。就目前的技术而言，这种方案的商用实践几乎不现实，因为数据一致性的问题非常复杂，并且数据安全也得不到很好的保障。

混合式元数据管理方案是一种目前很流行的研究课题，并已有比较好的实践基础，这种方案一般是根据一些已有方案的痛点问题，以及对一些负面开销的平衡判断，针对性的结合上述三种方案的设计思路，形成一种混合式的具有特殊适用性的方案。比如说 Ceph 结合了分布式元数据管理方案和无中心元数据管理方案，蓝鲸分布式文件系统则结合了集中式元数据管理方案和分布式元数据管理方案。

下面具体分析这几种方案。

2.3.1 集中式元数据管理

最早期的分布式文件系统，元数据和文件数据是统一存储的，在这样的架构下元数据管理和文件数据 IO 共同消耗一台机器的资源，系统的性能、存储能力有着很大的局限性，可扩展性也大大受到了制约。后来人们根据商业系统中控制与数据分离的思想，发现在分布式文件系统中将数据流与控制流分离会带来更好的性能以及扩展性，集中式元数据管理方式孕育而生。在集中式元数据管理方案中，所有元数据的操作与存储都集中在一台元数据服务器上，它管理调度着多台数据服务器，文件 IO 也只在客户端与数据服务器之间发生。集中式元数据管理方案的通用架构如图 2-2 示：

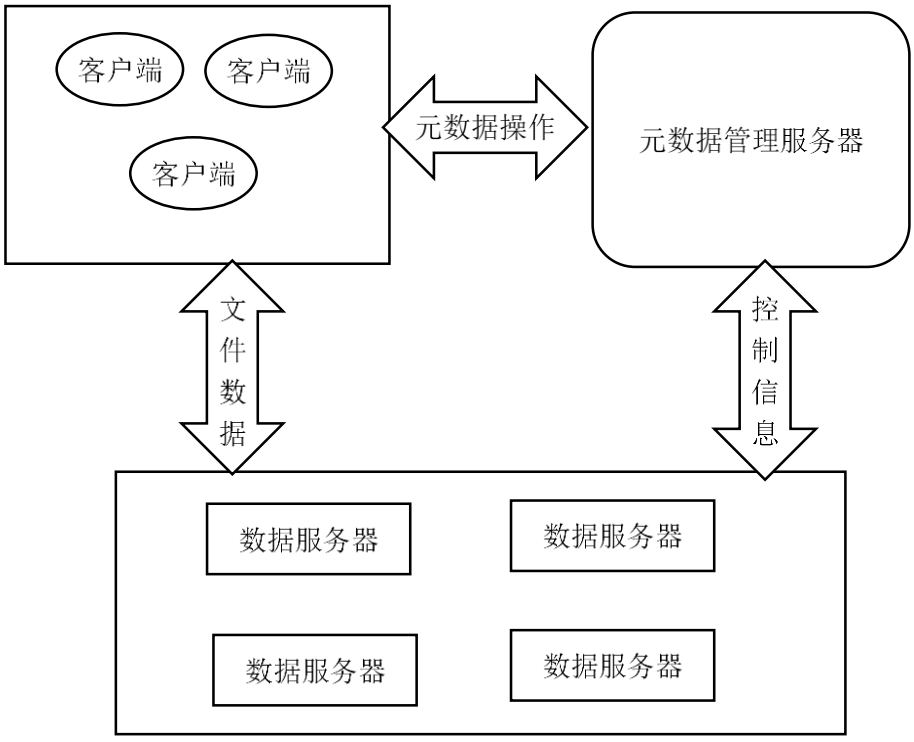

图 2-2 集中式元数据管理方案架构

当然，集中式元数据管理方案最大的问题就是单点问题。单点问题包括单点故障和性能瓶颈。

单点故障：在集中式元数据管理方案的系统架构中可以看到，元数据服务器承接着接受客户端元数据操作请求，向数据服务器发送控制信息，向客户端返回元数据信息等操作，并且元数据服务器是一个单点。无论这个单点的硬件或者软件质量再高，都有故障的可能性，并且只要一故障，整个系统就如同失去了头部一般。这是毁灭性的。

性能瓶颈：虽然说单点故障是毁灭性的，但是它并没有那么严格，并且目前来看有许多针对性方案比如说双机热备去缓解这个问题，所以性能瓶颈才是最主要的问题，也是集中式元数据管理方案中无法回避的问题。由于集中式元数据管理方案为了效率考虑，一般都会将全部元数据缓存在内存，所以单机内存的容量上限直接决定了元数据的上限，也决定了整个系统的存储上限。此外，单机的性能上限也直接影响了整个系统的并发处理计算能力。在越来越庞大的数据增长趋势下，性能瓶颈所带来的制约不下于单点故障。

目前，常见的集中式元数据管理方式有基于目录树的名字空间管理和基于哈希映射的名字空间管理两种。

2.3.2 分布式元数据管理

在集中式元数据管理方案中，性能瓶颈和单点故障这两个问题都是因为元数据服务器只有一台导致。所以，后来人们提出了用集群解决这个问题。即是将元数据分布到多台机器上，多台元数据服务器协同完成客户端请求和数据服务器调度，这样一来，性能瓶颈和单点故障这两个问题也就不复存在。分布式元数据管理方案的通用架构如图 2-3 示：

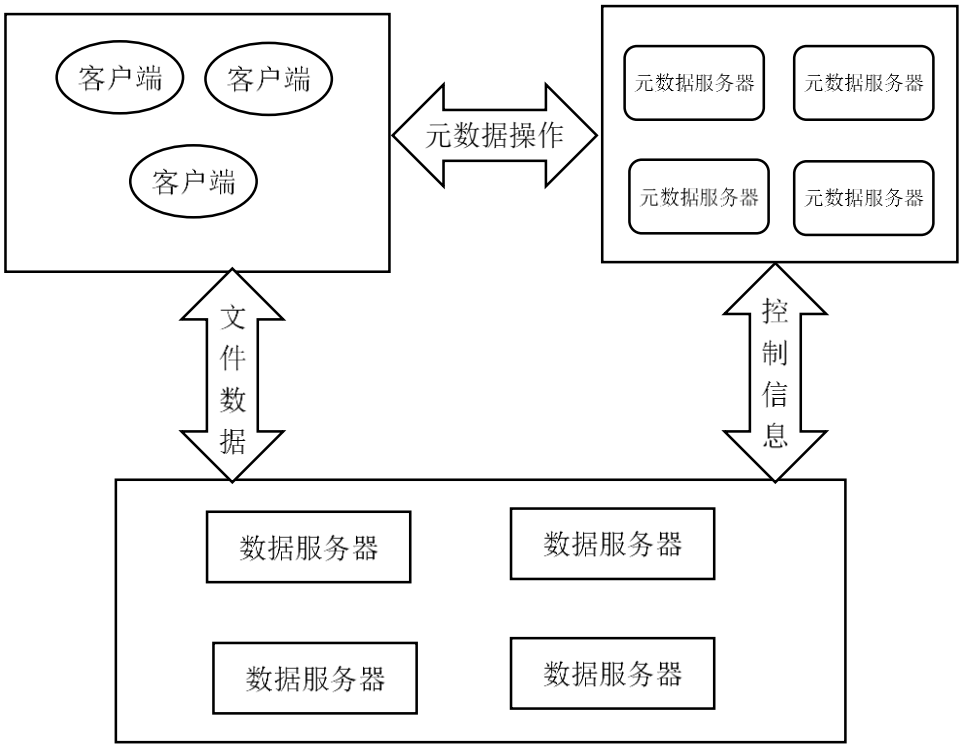

图 2-3 分布式元数据管理方案架构

在分布式元数据管理方案中，元数据服务器之间的关系有两种：1、元数据服务器之间

角色对等，互相的元数据信息可能会有重叠；2、元数据服务器之间相互协作，各自负责一部分元数据，元数据之间没有重叠。

分布式元数据管理方案虽然解决了集中式元数据管理方案中的单点问题，但是它也引入了一致性和性能等问题。

性能开销：分布式元数据管理方案中，元数据服务器之间会因为元数据的同步造成额外开销，无论是带宽或者 CPU，都会造成性能损耗。此外，由于要保证元数据的一致性，还会使用各种锁来保证同步，所以对性能的额外要求在总体上要比集中式元数据管理方案大。

一致性问题：元数据服务器之间的除了要同步，它们也要面对一致性问题。在分布式元数据管理方案中，元数据服务器之间的数据有对等和协作两种关系，无论哪一种都需要保证元数据的一致性，否则会出现“数据分身”的现象，干扰整个系统的运作。

目前，常见的分布式元数据管理方案中名字空间的划分方式有静态子树划分和动态子树划分两种。

2.3.3 无中心元数据管理

无中心元数据管理方案主要是 p2p 技术的发展产物。它没有数据节点和管理节点之分，每个节点都是对等的。在无中心元数据管理方案中，一般会将整个名字空间分成很多段，每一段会分给一个或多个节点，这样可以保证系统的可靠性。无中心元数据管理方案的通用架构如图 2-4 示。

无中心元数据管理方案充分解决了单点问题，系统可靠性也得到了提升，更保障了稳定性和可扩展性。但是这种方案带来的一致性问题比分布式元数据管理方案更为复杂，且对范围查询的支持十分欠缺，对整个系统的监控能力较前两种方案有一定的不足。

无中心元数据管理方案为了在没有中心节点的条件下能够有效地管理到每个节点，一般采用一致性哈希或者一些特殊算法，通过文件名就可以直接计算出文件所有块的存储信息，包括对应的映射关系和块存储点位置等等。

在无中心元数据管理方案中，只要很少的机器分布表，加上一致性哈希算法，即使没有中心管理节点，在无中心元数据管理方案中，在任一个节点都可以有效的管理到每一个文件。

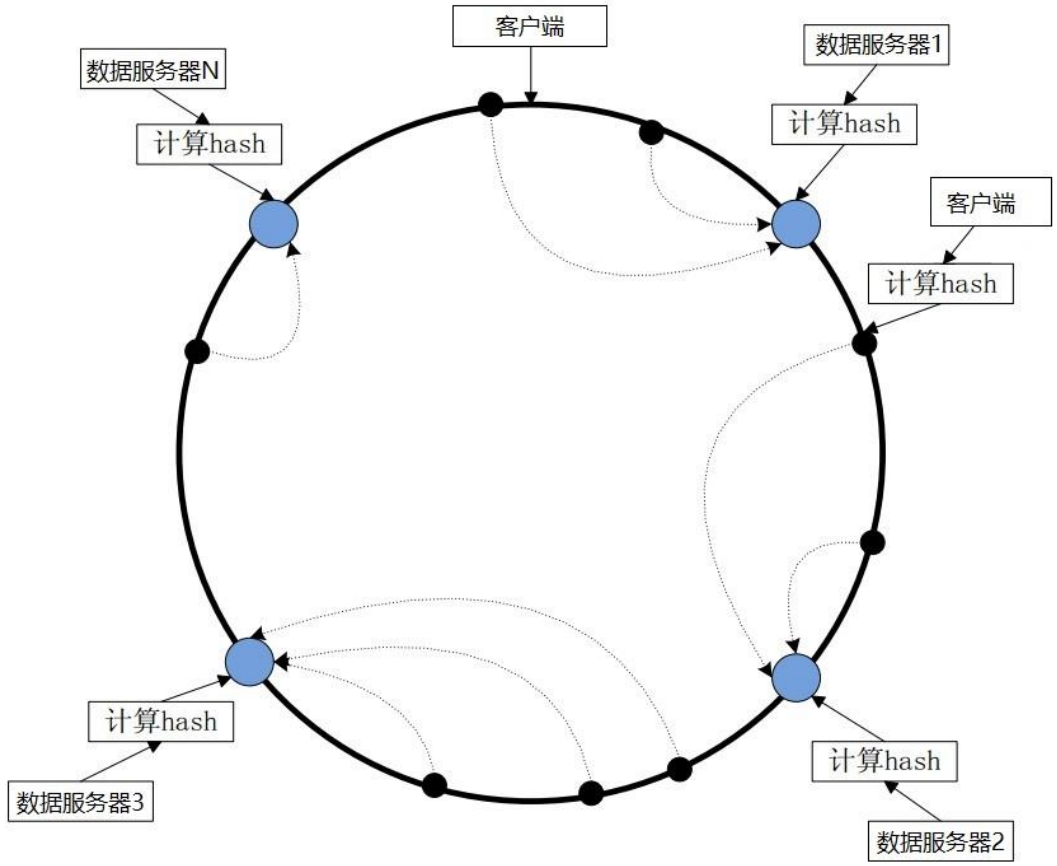

图 2-4 无中心元数据管理方案架构

2.3.4 混合式元数据管理

混合式管理是在前面三种管理方式的基础上，根据具体情况做的针对性优化，往往具有一些综合优势和缺陷。由于在实际应用中，系统的有些缺陷可以妥协，有些需求又是必要的，所以出现了混合式元数据管理方案。

理论上而言，集中式元数据管理方案中元数据是集中控制、集中存储的，而分布式元数据管理方案中元数据是分布控制，分布存储的。集中控制有不需要同步开销、一致性简化等优点，但是会有性能瓶颈，集中存储也会造成单点问题和容量上限。分布式控制虽然没有性能瓶颈，但是同步开销会增大，一致性问题也会很复杂，分布式存储也有效缓解单点问题。所以蓝鲸分布式文件系统创新的将二者特点结合，采用了分布存储、集中控制的混合式方案，使得单点问题得到一定缓解，同步开销明显减小很多，一致性问题也没有那么复杂。

而 Ceph 则结合了无中心元数据管理方案和分布式元数据管理方案的思路，用分布式集群管理文件系统的名字空间，而独创的 CRUSH 算法取代了副本映射信息，使得整个存储系统没有像 GFS 一样基于现有的 Linux 文件系统，而是直接操作硬盘，通过读写顺序优化以及

存取方式优化使其高效化。

2.3.5 基于 GFS 的元数据管理方案

考虑到设计需求和实现难度，本论文所述的分布式系统选择类似 GFS 的单元数据服务器的集中式元数据管理方案，这样就可以避免元数据的一致性问题和分布式元数据服务器之间同步的额外性能开销，另外，为了解决单点故障问题，对元数据服务器采用了 shadow 机制，即一个元数据服务器配备两个 shadow，不同于普通的冷热备机制，shadow 与元数据服务器的功能在读操作(read)上完全一致，但是 shadow 是只读的，这样就能保证在元数据服务器故障的时候整个文件系统的读操作能正常进行，写操作会失效；另外 shadow 与数据服务器的元数据保持懒同步，采用了弱一致性的方案，每个元数据服务器和 shadow 会配备相应的一个日志服务(log server)，元数据服务器会在所有操作之前先记录在日志服务器上，配置在 shadow 上的日志服务会向元数据服务器的日志服务同步数据，然后 shadow 会根据元数据服务器的操作日志回放操作，使得 shadow 上的元数据与元数据服务器保持一致，并且，shadow 也会定期和元数据服务器握手来确保他们的状态一致。当然这个过程有一定的延迟，但这个延迟在理论上是属于可接受范畴的，这种建立在 shadow 只读的基础上针对性优化而来的懒同步机制可以有效避免强一致性带来的巨大的性能损耗。另外，根据实际情况的分析与统计，发现对元数据的操作大部分是读操作，所以通过将元数据的读操作分流到两个 shadow 上可以在很大程度上缓解单点性能瓶颈问题。另外，实践中发现文件的名称信息、路径信息等等的前缀重复信息比较多，所以利用前缀压缩技术可以压缩元数据，从而缓解容量上限问题。

2.4 元数据服务

元数据一般会有三种类型：文件的命名空间，文件到数据块的映射信息，每个数据块副本与数据服务器的位置映射关系[11]。所有的元数据都会缓存在元数据管理服务器的内存里，这样能保证整个系统的性能。

2.4.1 名字空间管理方案

选择把元数据全部缓存在内存中具体而言有两个好处：一是元数据管理服务器的请求处理速度会很快。二是元数据管理服务器可以定时在后台扫描自己保存的全部状态信息，这种

操作将会十分高效并且实现简单。这样一来，垃圾回收、数据服务器故障是迁移数据、存储空间使用状态统计和跨数据服务器的负载均衡等这些基于系统全量状态信息分析的功能实现起来也会轻松许多。

名字空间的数据结构采用了 B+树。这有三个好处，一是能高效的处理读写请求，因为如果遇到特别深的目录，树结构也能在 log n 的时间内找到文件，且插入删除的时间复杂度也会控制在 log n。二是方便做 Checkpoint。三是很容易支持 COW（写时拷贝）机制，实现快照的功能，在批量复制的应用场景上能对整个系统的性能会起到很大的提升。关于 Checkpoint 机制将会在 2.4.4 节中详述。

2.4.2 元数据一致性保障

为了使应用程序和文件系统 API 的协同设计更加灵活，整个系统决定支持一个宽松的一致性模型。由于元数据管理采用了集中式方案，整个系统所要解决的元数据一致性问题基本都在多个客户端同时进行的修改操作上。名字空间锁确保了名字空间的修改是原子性的。名字空间是一个树结构，锁的粒度不会是整个目录树，而是会在整个路径。

2.4.3 影子服务器

为了解决单点故障和并发性能问题，引入了影子服务器这个概念。影子服务器(shadow master)实为元数据管理服务器(master)的影子，理论上来讲，影子服务器上面的元数据和元数据管理服务器一致（当然会有很小一段时间的延迟），影子服务器设计为只读的，用于分担元数据管理服务器的读请求。并且当元数据管理服务器故障重启时，影子服务器还能继续为系统提供元数据访问服务，使得业务不至于完全中断。

影子服务器与元数据管理服务器的同步有两种，一种是通过日志服务(log server)的懒同步，这是实时的，虽然有延迟。由于元数据管理服务器的每一步元数据操作前都会先把操作写入日志，通过日志服务管理并同步到其他的日志服务器上，而元数据管理服务器、影子服务器与日志服务器的部署是一一对应的关系，所以影子服务器会通过本机上部署的日志服务读取元数据变更日志，并进行回放，达到与元数据管理服务器元数据同步的效果。这样的好处是同步不需要额外损耗元数据管理服务器的资源。第二种是影子服务器会定期向元数据管理服务器发起同步请求，然后元数据管理服务器会将自己的元数据做一个 Checkpoint，把 Checkpoint 文件传给影子服务器。

2.4.4 故障恢复

元数据服务的故障恢复主要靠 Log server 和 Checkpoint 两个机制来保障。

但是，元数据管理服务器并不保存持久化 chunk 块的副本位置映射信息。这些信息将会由数据服务器在与元数据管理服务器成功连接上时全部上报。

Log server

日志服务管理着元数据管理服务器所有元数据操作的日志记录。一般是每一台部署了管理服务或影子服务的服务器上都是部署一个相应的日志服务。日志服务采取主动同步的机制，部署了管理服务的服务器上的日志服务会主动分发最新的元数据操作日志记录到部署影子服务的服务器上。

Checkpoint

在元数据管理服务器故障重启后，系统会读取操作日志并且仅在元数据管理服务层重新执行这些操作使元数据恢复到故障前的状态。为了优化元数据管理服务器启动的时间，Checkpoint 机制会将日志控制得足够小。Checkpoint 机制会在日志文件增长到一定量时对名字空间中的元数据状态做一次 Checkpoint，将所有元数据状态信息写入到硬盘上的一个 Checkpoint 文件。在元数据管理服务器故障重启时，系统会读取这个 Checkpoint 文件，并仅在元数据管理服务层重新执行少量的操作日志就能成功使元数据恢复到故障前的状态。由于名字空间中的元数据的数据结构是 B 树，所以 Checkpoint 文件也以压缩 B 树的结构存储。这样，读取时 Checkpoint 文件就可以直接载入内存，不需要作额外的解析，很大程度上提升了重启速度。

由于元数据管理服务器的故障恢复只需要最新的 Checkpoint 文件和在这个 Checkpoint 点之后的少许操作日志，过期的 Checkpoint 文件和操作日志可以被清理，不过为了保障数据的安全，一般都会保留一些过期文件。此外，Checkpoint 的过程如果意外中断，系统不会受到任何影响，因为 Checkpoint 会在一个额外的线程执行，而且重启时的恢复代码也会自动进行完整性校验来跳过中断的 Checkpoint 文件。

2.4.5 负载均衡

系统通过影子服务器的读请求分摊来实现一个最简单的负载均衡，但这是很有效果的，因为根据设计需求，系统在稳定运行时期超过 50%来自客户端的请求都是读请求。

另外，在副本层面，元数据管理服务器会定期检测并将热点数据从热点数据服务器上迁移到非热点数据服务器上。

2.5 副本管理

2.5.1 Chunk 块管理

对于在普通硬件设备上部署的分布式文件系统，硬件故障必须视作常态事件。为了方便管理和容灾，文件并不以文件的方式进行存储，而是被分割时多个大小一致的 Chunk 块，分别存储到不同的数据服务器。为了保证数据安全，每一个 Chunk 块会有多个完全相同的副本，副本数默认为 3，但是支持自定义。元数据管理服务会保证 3 个副本中至少有一个在不同机架。

根据 GFS 的观点，Chunk 块大小被设置为 64M，远超过普通文件系统的 Block 块大小。这样的设置大幅减小了元数据的规模，提升了整个系统的容量上限。

2.5.2 数据完整性

每个数据服务器都会使用 Checksum 来校验数据块的完整性。Checksum 是数据校验中很常用的手段，每个 64M 的 Chunk 块会分成 1024 个 64K 的小块，每个小块对应一个 Checksum，Checksum 数据一般存在数据服务器本地内存和硬盘。

在任何读取操作之前，数据服务器都会先校验涉及到的 Chunk 块的 Checksum，一旦校验失败，数据服务器会让读取操作返回失败，并通知元数据管理服务器创建一个新的副本，然后删除不完整的副本。

作为一个优化，Checksum 的校验可以和文件 IO 同步进行，这样一来，Checksum 的性能影响几乎可以忽略不计。

另外，在数据服务器负载不高的时候，它会对不活跃的数据块进行扫描检验。因为这些数据块很可能因为未被读取而无法被发现损坏。这个机制能有效地避免不活跃块成为完整性检测的落网之鱼，保障了系统整体数据的完整性。

2.5.3 副本一致性

当数据服务器故障时，可能正好有一些修改操作命中到故障的数据服务器，这会导致当数据服务器故障重启后，服务器上存储的一些副本可能因为错过了修改操作而过期。

过期副本由版本号机制识别。元数据中每个 Chunk 块都有一个版本号信息，用来检测数据服务器中副本是否过期。只要元数据管理服务器和数据服务器建立了一个租约，元数据管

理服务器就会增加相应 Chunk 块的版本号，另外，在数据服务器进行数据变更时也会增加每个副本的版本号，如果有一些操作不成功导致副本不一致，那么副本的版本号也会不一致，这样就可以在垃圾回收时检测出过期副本并作出相应的处理。除了变更操作时，元数据服务器也会在数据服务器全量上报副本位置信息时进行一次检测。

元数据管理服务器会在定期的垃圾回收任务中通过向数据服务器发送删除请求来处理过期副本。在过期副本被处理之前，元数据管理服务器在进行元数据交互时会直接认为过期副本不存在，这样能简化交互流程。另外，在任何发送至数据服务器的元数据中都会包括数据块的版本号信息，数据服务器在任何操作之前都是先验证版本号信息，以确保副本的一致性。

租约(lease)是副本变更操作中保证副本一致性的重要手段。租约其实就是数据服务器与元数据管理服务器之间一个合同，数据服务器向元数据管理服务器申请某个 Chunk 块的租约，在合同期限内，元数据管理服务器不会对这个 Chunk 块进行修改操作，而在这段时间内，多个数据服务器之间会进行这个 Chunk 块所有副本的有序更改。当然，如果时间不够，数据服务器可以申请租约延期，或者等到合同到期，元数据管理服务器会自动解除租约，相应的 Chunk 块的状态变为可修改。租约机制在保证副本一致性的同时，通过期限限制，有效地防止了忙等，因为数据服务器可能会因为网络延迟等原因在副本更改未结束、租约到期的时候又没有成功执行续约操作，这时，就算数据服务器未连接上元数据管理服务器，他自己也知道租约过期，副本修改失败，不会再继续执行修改操作，除非重新申请租约。

2.5.4 副本压缩策略

当数据达到一个很大规模时，比如说 120PB，如果按照现有副本策略（每个 Chunk 块存 3 份副本），备份量将达到 80PB，数据有效率只有 33%，这对成本而言无疑是一个很大的压力。但在实践中，可以发现绝大多数数据不会被经常使用，热点数据总是只占据一小部分。一个典型的适应性操作就是增加热点数据的副本数，减小冷数据的副本数，但是 3 个副本已经是能接受在数据安全行约束下的最小程度。所以需要一个有效的策略来提升占比为大部分的冷数据的有效存储率，并且由于是冷数据，还可以放宽对读取性能的要求。

通过分析，发现纠删码技术[12]能有效应对这个需求。表 2-1 是三副本数据存储方案和纠删码方案对比。

表 2-1 三副本数据存储方案和纠删码方案对比

	三副本	纠删码
存储成本	较高（3X）	经济（<3X）
容错能力	2	可配置（>2）
实现复杂度	简单	较复杂
系统性能	相对无延迟	需编码解码

通过纠删码技术，选择 10 冗余 4 策略，每 10 个数据块产生 4 个数据校验块，总共 14 个数据块，容忍丢失 4 个副本，而按照三副本策略 10 个数据块会产生 30 份数据块，容忍丢失 2 个备份，在容错能力提升的情况下，整体的数据有效率从 33.33%提升到了 71.42%！当然，纠删码方案需要编码解码过程，对性能有一定的影响，但是由于只对冷数据进行纠删码编码优化，所以性能的损失可忽略不计。

2.5.5 小文件聚集块

由于小文件通常不足 1M，远远达不到一个 Chunk 块 64M 的大小，如果按照现有的数据存储机制，一个小文件就会占据 64M 的存储空间。并且，元数据是按文件为单位存储的，同样容量比如说 1G，如果全部是 1M 的小文件，需要 1024 条元数据，而如果是 100M 的大文件，只需要 10 条。小文件带来的问题是元数据容量和文件存储容量的大量消耗，所以针对小文件，可以采取一种文件聚集策略[13]，将多个小文件统一存在一个 64M 的 Chunk 块里，但是这个块是特殊的块，分为索引和数据两块空间，一般索引会在块的头部，保存着每个文件的偏移量，实际访问就可以通过先访问索引，得到具体文件的偏移量，在找到具体文件内容。在元数据方面，由于多个小文件共用一个 Chunk 块，可以大幅压缩位置映射信息。

2.5.6 RADOS 存储引擎

RADOS(Reliable Autonomic Distributed Object Storage)[14]是专门为 Ceph 设计的一个分布式对象存储系统。CRUSH 算法是 RADOS 最大的创新，它通过一个算法而不是元数据就维护起了存储对象与服务器之间的对应关系，并采用了无中心管理节点的设计，数据迁移、对象复制、故障检测、集群扩容直接由数据节点集群提供，这样一来，避免了单点问题，扩展性也非常不错，更支持海量存储对象。而 GFS 的存储引擎并没有做特殊优化，所有的数据块都直接存在普通 Linux 文件系统上。本系统的数据块存储引擎采用双引擎，提供自由选择，默

认将数据块存储在 Linux 文件系统上，但根据需要优化的程度存储大文件，为集群提供更好的扩展性，还可以缓解单点问题，也可以将小文件存储到 RADOS 上，这样就不需要采用小文件聚集策略，也不需要元数据块地址映射信息，因为这个地址可以根据 CRUSH 算法计算而来。由于 Ceph 是开源项目，并且直接提供了 RADOS 库供开发者直接调用，所以实现起来比较简单。

2.6 客户端接口设计

一般来说，一个分布式文件系统可以有很多个进行着不同业务的客户端共同访问。作为分布式文件系统本身而言，不提供直接的客户端，客户端代码以库的形式被提供。客户端接口模仿 Linux 传统文件系统的细节，实现了一些基本的文件操作，当然，这些文件操作却不同于传统文件系统，因为其中涉及到与不同服务器之间的通信和协调。客户端和元数据管理服务器之间只有元数据的交换，正真的数据流只在客户端和数据服务器集群之间传输。为了规避缓存一致性问题并且得益于 Linux 文件系统的缓存机制，客户端不需要缓存文件数据，不过，通常而言，客户端会缓存元数据，在从元数据管理节点获取元数据的时候，客户端一般会再获取目标元数据接下来的部分元数据放入缓存，因为根据经验和业务需求，客户端执行的业务很大可能会进行连续读取。

第 3 章 分布式文件系统的实现概述

本章简单介绍基于 GFS 的分布式文件系统的元数据管理服务的实现、数据服务器的实现以及客户端接口的实现。

3.1 元数据管理服务实现

元数据管理服务器主要负责元数据的管理、数据服务器集群的调度、租约管理和 Chunk 复制，并向客户端提供元数据信息。具体功能见表 3-1。

表 3-1 元数据管理服务功能表

功能	详细描述
数据服务器管理	元数据管理服务器与数据服务器通过心跳包交互，判断数据服务器是否正常运行，并接受数据服务器上报的状态信息。
Chunk 复制管理	在写入数据时，会根据数据服务器负载情况将 chunk 块复制为设定好的数量分布在不同的机器上。
租约管理	管理副本租约，进行变更操作时，对每一个副本建立一个租约，选择一个主副本，由主副本所在数据服务器决定修改顺序，并执行操作，这能有效的减轻元数据管理服务器的负担。
客户端接口	客户端向分布式文件系统请求或写入数据时会先连接元数据管理服务器，元数据管理服务器根据负载状况和集群状况选取合适的几个数据服务器并告知客户端位置，客户端直接与数据服务器发生数据交换。
元数据管理	包括 Checkpoint，操作日志管理、Shadow 等一系列保障元数据的机制。

如图 3-1 所示，元数据管理服务器主要分为 7 个模块，其中系统配置模块和系统日志模块会在初始化时调用，租约管理器在写操作时创建，可以同时有多个，其余每个模块由一个或多个线程循环调度，以提供稳定的系统服务。

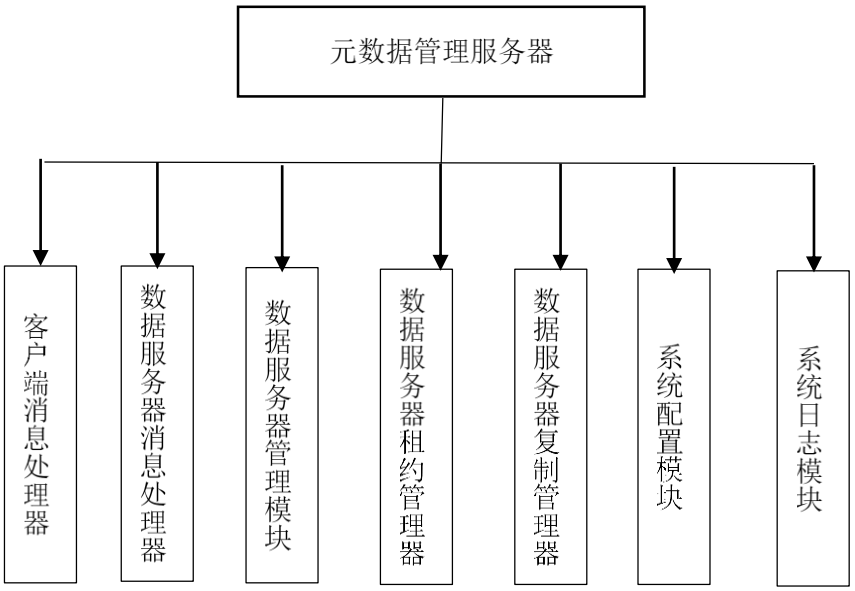

图 3-1 **元数据管理服务器模块划分**

3.2 数据服务实现

数据服务器主导本地文件的存取的功能，利用租约保证数据一致性。按需上报 chunk 信息到元数据管理服务器，并与客户端交换文件数据。具体功能如表 3-2 所示。

表 3-2 **数据服务功能表**

功能	详细描述
本地文件存取	数据服务器收到客户端发来的数据或者将数据发送给客户端，并将成功信息发给元数据管理服务器。
数据块信息管理	数据服务器会将本地管理的数据块信息缓存在内存中。
副本租约	在涉及到修改操作时，为每一个副本建一个租约，来保持多个副本间变更顺序的一致性。
元数据管理服务器交互	定期与元数据管理服务器通过心跳包联系，并上报状态。
客户端交互	与客户端建立连接，执行客户端发来的操作请求，如读写数据块、复制数据块、移动数据块、删除数据块等。

3.3 客户端接口实现

为了保证客户端的可编程性，相似于传统文件系统的 API，提供了一套 API 接口函数，大体上分为文件系统接口和文件接口两类，都是模仿 Linux 的系统调用接口，包括参数和返回值类型、标志位的设置等。文件标识是基于路径名的分层目录树结构，支持常用的文件系统操作，如创建文件目录、重命名、创建链接、删除文件目录、打开文件、关闭文件、读写文件、获取文件信息。如图 3-2 示：

FileSystem
File* open(const char* pathname, int32_t flags, int32_t replica_num = DEFAULT_REPLICA_NUM)
File* creat(const char* pathname, int32_t replica_num = DEFAULT_REPLICA_NUM)
int32_t unlink(const char* pathname)
int32_t rename(const char* old_path, const char* new_path)
int32_t mkdir(const char* pathname)
Directory* opendir(const char* pathname)
int32_t closedir(Directory* direc)
int32_t exists(const char* pathname)
int32_t stat(const char* pathname, FileStatus* file_status, bool get_exact_len = false)
int32_t copy_from_local_file(const char* src, const char* dstf, bool delete_local = false)
int32_t copy_to_local_file(const char* srcf, const char* dst, bool delete_source = false)
int32_t close(File* file)
static FileSystem* get()
static FileSystem* get(const char* pathname)

图 3-2 文件系统接口

另外，系统还提供了快照和记录追加操作。快照以很低的性能消耗复制一个文件或者目录，得益于 Linux 中写时拷贝的机制。记录追加操作不需要额外同步锁，支持多客户端同时对同一个文件进行原子性的数据追加操作，这对实现多路合并的并行模型十分有效。

第 4 章 结束语

本论文基于《The Google File System》论文，结合现有的相关研究成果与实例，对分布式文件系统进行了一个相对完整的研究，最终完成了基于 GFS 的分布式文件系统的设计和实现。

在设计上，本文充分参考了一些国内外具有实践基础的分布式文件系统，例如作为《The Google File System》最重要实践的 HDFS、针对海量小文件存储的 TFS、采用主备机制缓解单件问题的 XFS、采用算法代替位置映射表的 Ceph、在元数据管理方案中采用创新思路集中服务分布存储的 BWFS 以及采用分布式元数据服务的 GFS 进化版本 GFS2。正如前言所述，对于一个企业级应用，面对不同的应用场景，最佳方案是做出针对性的优化，而非寻求通用解决方案。商业化需求中成本和收益始终是放在第一位的，这与科研活动有一定的区别。而本文所论述的基于 GFS 的分布式文件系统的定位是一个应对公司业务需求、商业化运营下的，为大量网民提供高质量互联网服务的分布式文件系统。在详细分析了设计需求后，发现 GFS 的设计理念与本系统的设计需求最为契合。此外还通过一些业内的研究成果与实践基础的研究，对基于 GFS 的分布式文件系统做了一定的针对性优化，包括但不限于小文件支持、副本压缩、缓解单点问题、存储引擎改进。

在实现上，由于企业机密等因素，在这里并没有做过多的叙述，只是介绍了大概的实现思路以及功能划分。

对于本系统而言，功能已相对完善，从实际运营经验中看到的成本和收益也是有了一定的平衡，但始终会存在单点问题，在单点问题上已经做出的针对性优化也只能是缓解，并不能完全解决。所以今后的工作还可以从单点问题入手，可以尝试将集中式元数据管理方式往分布式元数据管理方式上迁移，另外，为了进一步降低运营成本，还可以尝试例如集群节电策略、节省存储空间的策略等等的优化。

致谢

毕业设计论文的完成为我大学四年的学习生涯划下了句号，在毕设过程中很多人给了我无私的帮助，在此表示由衷感谢。

首先要感谢我的校内指导老师徐造林徐老师，在毕设的开题阶段给了我指导性的建议，及时的纠正了我选题中不合理的地方，让我之后的毕业设计能顺利进行。

同时也要感谢我的企业指导老师兼团队 Leader 孙永跃，在毕设项目开始阶段给我讲解了很多相关知识，给我了很多建议。

感谢我的同事廖晨歌、任亚坤、潘燕，在项目编码阶段给了我很多帮助。

最后感谢我的父母，是他们养育我，教育我，为我无私付出，让我能健康成长。

再次感谢所有在毕设上给予我帮助和关心的人！

参考文献

[1] Ghemawat S, Gobioff H, Leung S T. The Google file system[C]//ACM SIGOPS operating systems review. ACM, 2003, 37(5): 29-43.

[2] 郝向涛. 基于 Hadoop 的分布式文件系统技术分析及应用 [D]. 武汉理工大学, 2013.

[3] XFS Master 主备机制：高性能的分布式文件系统的元数据单点高可用方案. http://djt.qq.com/article/view/322.

[4] 揭秘淘宝自主研发的文件系统——TFS. http://www.infoq.com/cn/articles/tao-tfs.

[5] Weil S A, Brandt S A, Miller E L, et al. Ceph: a scalable, high-performance distributed file system[C]// 7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA. 2010:307--320.

[6] 杨德志, 黄华, 张建刚, 等. 大容量, 高性能, 高扩展能力的蓝鲸分布式文件系统[J]. 计算机研究与发展, 2005, 42(6): 1028-1033.

[7] 黄华, 张建刚, 许鲁. 蓝鲸分布式文件系统的分布式分层资源管理模型[J]. 计算机研究与发展, 2005, 42(6): 1034-1038.

[8] 杨德志, 许鲁, 张建刚. 蓝鲸分布式文件系统元数据服务[J]. 计算机工程, 2008, 34(7):4-6.

[9] 王亮. 基于 Linux 的分布式文件系统的设计与实现[D]. 华中科技大学, 2013.

[10] 陈云云. 分布式文件系统名字空间管理[D]. 华中科技大学, 2013.

[11] 冯幼乐. 分布式文件系统元数据管理技术研究与实现[D]. 中国科学技术大学, 2010.

[12] 王敬轩. 分布式文件系统存储效率优化研究[D]. 华中科技大学, 2013.

[13] 付松龄, 廖湘科, 黄辰林, 等. FlatLFS: 一种面向海量小文件处理优化的轻量级文件系统[J]. 国防科技大学学报, 2013, 35(2): 120-126.

[14] Weil S A, Leung A W, Brandt S A, et al. Rados: a scalable, reliable storage service for petabyte-scale storage clusters[C]//Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07. ACM, 2007: 35-44.

[15] 田怡萌, 李小勇, 刘海涛. 分布式文件系统副本一致性检测研究[J]. 计算机研究与发展, 2012 (S1): 276-280.