

工程硕士学位论文

迅雷离线下载分布式文件系统的设计与实现

THE DESIGN AND IMPLEMENTATION
OF XUNLEI OFFLINE-DOWNLOAD
DISTRIBUTED FILE SYSTEM

吕磊



哈爾濱工業大學

2010 年 6 月

国内图书分类号：TP311

学校代码：10213

国际图书分类号：621.3

密级：公开

工程硕士学位论文

迅雷离线下载分布式文件系统的设计与实现

硕士研究生：吕磊

导师：马培军教授

副 导 师：贺鹏飞高级工程师

申 请 学 位：工程硕士

学 科 、 专 业：软件工程

所 在 单 位：软件学院

答 辩 日 期：2010 年 6 月

授予学位单位：哈尔滨工业大学

Classified Index: TP311

U.D.C.: 621.3

Dissertation for the Master's Degree in Engineering

**THE DESIGN AND IMPLEMENTATION
OF XUNLEI OFFLINE-DOWNLOAD
DISTRIBUTED FILE SYSTEM**

Candidate:	Lv Lei
Supervisor:	Professor Ma Peijun
Associate Supervisor:	Senior Engineer He Pengfei
Academic Degree Applied for:	Master of Engineering
Speciality:	Software Engineering
Affiliation:	School of Software
Date of Defence:	June, 2010
Degree-Conferring-Institution:	Harbin Institute of Technology

摘 要

随着计算机网络在社会生活中的广泛使用，分布式应用逐渐成为主流。传统的集中式服务器系统因其固有的，诸如单一故障点、可扩展性差等缺点，越来越制约着网络业务向多元化方向发展，为了满足众多应用服务对高性能、高可伸缩性，高可靠性以及高可用性的分布式服务器系统的需求，人们也慢慢地转向分布式服务器系统。

本文基于 Linux 内核开发了分布式并行文件系统，即迅雷离线下载分布式文件系统(Xunlei Offline-Download Distributed File System, 简称 XLFS)。该文件系统运行在普通的 Linux 机器上，集容错性、安全性、可扩展性于一身，能够同时为大量的客户机提供高性能的服务。

本文首先设计了分布式文件系统的整体架构，并针对迅雷离线下载项目的具体需求，采用了不同于以往文件系统的设计方案。首先，单一主服务器和多个块服务器的方案，既简化了文件系统的设计，又满足了文件系统的可扩展性。其次，针对离线下载数据追加多修改少的特点，文件系统特别提供了原子性的追加操作。再次，服务器间采用租约的形式交互，在数据块级别上保证数据的一致性，而不保证数据在字节级别上的一致性。最后，不同于以往大多数文件系统，该系统采用校验和的方式，在工程上提供数据的容错性。

本文的研究成果主要有以下几个方面：首先，掌握了分布式文件系统的相关技术，模拟实现了原子性追加、垃圾回收、租约等机制。其次，设计并实现了一个完整的分布式文件系统，包括主服务器、块服务器和客户端接口。最后，针对应用程序调用的问题，模拟 Linux 操作系统 API 实现了该系统的 FsShell 接口，并利用该接口对文件系统进行了测试。系统在测试环境下运行良好。

关键词：离线下载；分布式文件系统；原子性追加；租约

Abstract

With the widespread use of computer network in social living, distributed application becomes a mainstream gradually. Because of its inherent shortcomings, such as single fault point and low Scalability, traditional centralized server system more and more limits the development of network service to multi-aspect. In order to meeting the need of distributed server system, which has high performance, high extensibility, high reliability and high usability, people also turn to use distributed server system slowly.

A distributed file system named Xunlei Offline-download Distributed File System (XLFS) was developed based on Linux kernel. XLFS is a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

This thesis firstly designed the whole framework of distributed file system, and used a design scheme different from old ones for file system aiming at the specific requirements of Xunlei offline-download project. First, the scheme of single master server and several chunk servers has satisfied scalability for file system while simplified the designation. Second, in allusion to the characteristic of offline-download data that much appending and little amending, the file system specially provides atomic appending operation. Third, servers interact with each others in the form of lease, which could keep the consistency of data at the level of chunk, not byte. At last, different from most of old file systems, XLFS provides fault tolerance for data in engineering by verify code.

The research achievements are mainly in the following aspects: First of all, mastered relevant technologies of distributed file system, and simulately achieved the mechanisms such as atomic appending, rubbish callback and lease. Then designed and completed a distributed file system, including the master server, chunk servers and client interface. Last, aiming at the problem of application procedure call, achieved the FsShell interface of xunlei offline-download distributed file system similarly to linux operation system API. the FsShell interface was tested in this file system, and the system ran well in the testing environment.

Keywords: offline-download, distributed file system, atomic appending, lease

目 录

摘 要	I
Abstract	II
第 1 章 绪论	1
1.1 课题的背景及研究目的	1
1.2 与课题相关的国内外研究综述	1
1.2.1 NFS	2
1.2.2 AFS	2
1.2.3 Coda	3
1.2.4 xFS	3
1.2.5 Lustre	3
1.2.6 GPFS	4
1.2.7 其它分布式文件系统	4
1.2.8 Google file system	5
1.3 本论文的主要工作内容	5
第 2 章 分布式文件系统需求分析	7
2.1 离线下载项目简介	7
2.2 文件系统需求	8
2.2.1 功能需求	8
2.2.2 非功能需求	9
2.3 文件系统适用范围	11
2.4 本章小结	11
第 3 章 分布式文件系统的设计	12
3.1 文件系统设计概要	12
3.1.1 文件系统整体架构	12
3.1.2 单一主服务器设计方案	13
3.1.3 元数据设计	13
3.2 服务器交互设计	15
3.2.1 租约和变更	15

3.2.2 数据流设计方案	17
3.2.3 原子性记录追加	18
3.2.4 基本操作设计	19
3.3 主服务器设计	23
3.3.1 名称空间管理和锁	23
3.3.2 垃圾回收	24
3.3.3 主服务器类的设计	25
3.4 块服务器设计	26
3.4.1 块尺寸选择	26
3.4.2 完整性保证与校验和	26
3.4.3 块服务器类的设计	27
3.5 本章小结	28
第 4 章 分布式文件系统的实现	29
4.1 主服务器实现	29
4.1.1 主服务器功能分析	29
4.1.2 主服务器各功能实现	29
4.2 块服务器实现	36
4.2.1 块服务器功能分析	36
4.2.2 块服务器基本操作实现	36
4.3 客户端接口实现	39
4.3.1 文件系统接口	39
4.3.2 文件接口	41
4.3.3 多线程安全接口	42
4.4 本章小结	42
第 5 章 分布式文件系统的测试及评价	43
5.1 软件测试理论	43
5.1.1 常见测试流程	43
5.1.2 测试方法	43
5.2 功能测试	44
5.2.1 测试方案	44
5.2.2 测试结果	45
5.3 性能测试	49
5.3.1 测试方案	49

5.3.2 测试结果与分析	50
5.4 压力测试.....	51
5.5 本章小结.....	52
结 论	53
参考文献.....	54
哈尔滨工业大学硕士学位论文原创性声明	57
哈尔滨工业大学硕士学位论文使用授权书	57
致 谢	58
个人简历	59

第 1 章 绪论

1.1 课题的背景及研究目的

本课题来源于深圳市迅雷网络技术有限公司的离线下载项目。随着会员离线下载功能的普及，服务器端的压力越来越大。CPU 计算能力和带宽的迅速发展，使得服务器 I/O 能力明显成为服务器负载的瓶颈^[1]。

单服务器升级显然不能处理不断增长的负载。这种服务器升级方法有以下不足：一是升级过程烦琐，机器切换会使服务暂时中断，并造成原有计算机资源的浪费；二是越是高端的服务器，所花费的代价越大；三是一旦该服务器或应用软件失效，会导致服务的中断。

于是，通过高性能的网络与和局域网集群成为构建高性能、高可用的网络服务系统成为可行的解决方案。分布式/并行文件系统提供了一种 I/O 瓶颈的解决途径。它的设计目的在于将由本地操作系统完成文件系统独立出来以减轻本地 I/O 的负担。它将多个结点上的磁盘组织成为全局的存储系统并可通过网络实现共享。分布式/并行文件系统能提供更大的存储容量和聚集的 I/O 带宽，并可以随系统规模扩大而扩展^[2]。

XLFS(Xunlei File System)正是这样一个可扩展的分布式文件系统，用于大型的、分布式的、对大量数据进行访问的应用。它运行于廉价的普通硬件上，但可以提供容错功能，它可以给大量的用户提供总体性能较高的服务。

1.2 与课题相关的国内外研究综述

随着超级计算机的发展，尤其是大规模并行处理系统的出现，为了与可扩展的计算能力相匹配，并行文件系统的发展近年来逐渐成熟。利用硬件实现文件的并行存储，可靠性高，管理简便，但因为要使用特殊的硬件，受制于硬件的单价，投资巨大。而利用网络传输，也可以达到相同目的，费用却相对低廉。特别是近年来，网络的带宽不断扩大，而价格却不断下降，因而出现了许多利用网络拓展储存带宽的解决方案。它们是以提高文件存取的带宽、I/O 系统能力和容量以及可扩展性为目的设计的。在历史上曾出现过许多网络共享文档系统。下面对一些重要的分布式文件系统进行介绍，对其关键技术进行分析^[3,4]。

1.2.1 NFS

NFS(Network File System)^[5,6]现在已经发展到第四代,现在基本上已经成为通用分布式文件系统的标准。NFS 基于传统的客户/服务器服务模型,实际上是一种共享文档模型,单个节点既可以作为客户端也可以作为服务器端^[7]。NFS 利用 Unix 系统中的虚拟文件系统(VFS Virtual File System)机制,将客户机对文件系统的请求,通过规范的文件访问协议和远程过程调用,转发到服务器端进行处理;服务器端在 VFS 之上,通过本地文件系统完成文件的处理,实现了全局的分布式文件系统。在第四版中,NFS 提供了基于租赁的同步锁和基于会话的一致性。但是,NFS 实质上只是一个透明的存储远程文件的协议,它在写文件上的性能是比较差的^[8]。

1.2.2 AFS

AFS(Andrew-File-System)^[9,10],是卡内基-梅隆大学在 1983 设计开发的。它有诸多新的创举,比如第一个采用永久客户端缓存机制。它提供了对话语义,可扩展的,不依赖文件位置的名字空间。之所以采用这样的测率,是因为 AFS 将分布式文件系统的可扩展性放在了设计和实现的首要位置,并且着重考虑了在不安全的网路中实现安全访问的需求^[11],因此它在位置透明、用户迁移、与已有系统的兼容性等方面进行了特别设计。AFS 具有很好的扩展性,它能够很容易地支持数百个节点,甚至数千个节点的分布式环境^[12]。同时,在大规模的分布式文件系统中,AFS 利用本地存储作为分布式文件的缓存,在远程文件无法访问时,依然可以部分工作,提高了系统可用性^[13]。后来的 Coda File System^[14,15],Inter-mezzo File System^[16]都受到 AFS 的影响,更加注重文件系统高可用性(High Availability)和安全性,特别 coda 在支持移动计算方面做了很多的研究工作。

在 AFS 中,值得注意的是它的客户端永久缓存策略,它使用了回调机制来保证缓存的一致性。也就是说,客户端的读操作都是信赖客户端的缓存的,而缓存文件的更新则有服务器通过回调来维护。同时,它的缓存机制是永久性的,缓存文件会缓存在客户端磁盘上,这也就保证了在网络状况不理想情况或者客户端,服务器端重启下仍然可以提供服务的可能性。这也就是为什么 AFS 有如此出众的可用行。

同时,AFS 有强大的安全支持,客户机不被信任,它们必须通过授权从服务器得到 Kerberos tokens 形式的信用才能与服务器相连。

1.2.3 Coda

Coda^[14,15]系统尽管于 AFS 分布式文件系统存在很大渊源，但却拥有很多不同于以往的设计。在 Coda 中它的设计更多的考虑了对带宽的适应对断线情况的日志记录与恢复后的重做机制。

Coda 最吸引人的的是它的日志和版本戳算法，简单的说，就是在断线情况或者服务器带宽不足情况下，它在客户端采用日志机制来记下对于客户端缓存的写操作，在服务器端提供服务之后，服务器端通过对存储节点的版本和客户端缓存版本戳进行对比，如果版本不匹配，则通过客户端的日志记录在服务器端进行重做，这样既保证了缓存和服务器端文件的一致性，又提高了在故障情况下可用行。这是一个很好的设计思路。

1.2.4 xFS

xFS^[17,18]是由加利福尼亚大学伯克利分校设计的分布式文件系统。它不但可以运行在局域网上，而且它采用的一系列新技术，诸如多层次结构利用了文件访问的就近原则，通过在广域网上做缓存减少了大量的网络流量。同时，无效写回缓存一致性原则，包括对本地文件的有效利用，使它具有很好的性能。

xFS 最显著的特点是它是一种无集中式服务器(Sever less)模型的分布式文件系统，它通过并行协调式的元数据管理来达到这样的效果^[19]。同时，合作式缓存(collebrative cash)避免了单机缓存空间的限制。通过将传统的由服务器维护的如缓存一致性，资源定位等问题分步与各个存储节点之上，xFS 提高了相对的低延迟，高带宽的访问。

虽然研究表明 xFS 的可扩展性和性能均优于上面将到的 NFS 和 AFS^[20]，但是，由于无集中式服务器模型实现的复杂性，比如故障恢复功能，垃圾清除机制等^[21]，所以 xFS 现在仍然是一个处于研究状态的分布式文件系统。

1.2.5 Lustre

Lustre^[22]分布式文件系统是 Cluster File System 公司推出的面向下一代的存储的分布式文件系统。它采用基于对象的磁盘作为整个文件系统的存储设备。它使用带意图的锁，明显的减少了客户端和服务端的消息传递。

在 Lustre 中，系统中的资源的命名和组织通过树的形式，而锁就从这些资源中产生^[23]。将锁保持在 Cluster 成员上，是通过执行操作系统的一系列模式，包括独占的，保护的读与写，同步的读与写以及解锁。这六种模式之间的兼容性由一个简单的为标志。当一个成员在一个新的模式下请求锁，所有其它持有

这个锁的处于与这个模式不兼容的节点将被通知，而且一个回调函数被执行。为了得到一个锁，树的上级必须先被锁住。这样每一个锁的层次结构，比如在根节点下的树状资源，都有一个主节点，这个主节点通常是这棵树中的一个获取锁的系统。为了找到这个控制资源的主节点，可以通过资源名的 hash 值来请求一个分布资源目录。如果这个资源不存在，这个寻找资源的系统奖惩为主节点。

在每个 Lustre 节点中，都有大量的自治资源。对于这些资源的写操作，使用锁机制来实现回写缓冲；而对于读，则不使用锁机制。这也是由分布式文件系统的特性决定的。对读加锁会大大影响系统性能。

1.2.6 GPFS

General Parallel File System(GPFS)^[24]是 IBM 公司开发的高性能集群文件系统，从 1998 年开始首先应用于 AXI 集群，2001 年后应用于 LINUX 集群。在集群的环境中，GPFS 文件系统允许集群中所有的节点访问同一文件的数据，并提供统一的文件存储空间。应用可以使用标准的 UNXI 文件系统接口访问文件的内容。GPFS 支持 32 位和 64 位的应用，经过测试的文件系统的大小为 100TB。GPFS 可以动态的增加或减少文件系统的容量^[25]。

GPFS 直接对磁盘进行操作，它所采用的磁盘数据结构可以支持大容量的文件系统，支持大数据的文件，它采用了分片储存，大文件系统块，数据预取等方法获得较高的文件吞吐，采用扩展哈希技术来提高查找检索速度^[26]。GPRS 采用日志技术对系统进行灾难恢复，同时可以扩展在线的节点数^[27]。

GPFS 具有良好的性能，而这得益于它的几个设计方案诸如跨越多个节点的条带化数据，高效的客户端缓存和预读和后台写机制，总体上说，GPFS 的很多设计对于大文件系统具有良好的性能^[28]。

1.2.7 其它分布式文件系统

GDSS(Global Distributed Stroage System)是一个广域网分布式存储系统^[29,30]，其研究内容包括：多样的数据访问接口、全局范围内的数据共享与访问控制、全局统一的文件名字空间、文件缓存机制、灵活的安全认证机制、高性能数据传输技术和全局可视化统一管理。它的特点是实现了对广域范围内孤立存储资源的统一管理和数据共享，屏蔽了底层数据资源的分散性和异构性，为用户提供了统一逻辑视图，实现了数据的透明访问，减少了系统的管理开销。GDSS 主要包括存储服务提供者、客户端、存储代理 SA(Storage Agent)以及可视

化管理，其中存储服务提供者又可细分为存储服务接入点 SSP(Storage Service Provider)、命名服务器 NS(Name Server)、资源管理器 RM(Resource Manager)和缓存管理器 CM(Cache Manager)。

GFS (Global File System)^[31]是 RED HEAD 公司推出的新的分布式文件系统，它吸取了对称多处理器的优点，系统中的任一个客户机可以平等的访问系统的所有储存设备。更好的利用了系统的资源。

1.2.8 Google file system

Google File System^[32,33]，最后把它提出来是因为，它既不同于上面大部分通用分布式文件系统，它是专门为 Google 内部的需求而设计出的符合 Google 公司的一套分布式文件系统，而它的一个典型特点是它是建立在大量的普通 PC 机之上的，也就是利用现有资源完成的一套分布式文件系统。它的文件复制，一致性保证和容错机制都是简单却非常有效的，对于 XLFS 的系统设计来说，可以从它那里借鉴到很多。在后面具体技术的实现中会给出说明。

通过以上历史上到现在经典分布式文件系统的分析可以看到，尽管分布式文件系统可以有各种实现，但是有些是共通的问题，比如文件的复制，一致性的处理，多客户端对同一文件操作的锁等机制，XLFS 系统的设计中大量借鉴了前人的经验。在很多问题上，尽管这些分布式文件系统面对的问题不同提出了不同的解决方案，很多思路都是值得借鉴的。诸如，Coda 系统利用版本戳来保证缓存和服务端文件的一致性，XLFS 在热点文件缓存一致性保证中就使用了该算法；XLFS 借鉴了在多个文件系统中使用的日志机制来减小写锁的粒度算法；又比如，XLFS 借鉴了 Lusrt 中的对文件读写操作对锁机制的不同处理方案。同时，又专门为 XLFS 系统设计了许多专用的优化技术，比如相对于上述的通用分布式文件系统，XLFS 是特殊应用的专用分布式文件系统，在特定需求下系统的特定设计会显著的提高性能。在下面的系统结构中会有专门讲述。

1.3 本论文的主要工作内容

本文描述了一个分布式文件系统 XLFS 的设计和实现。通过实际工作和大量的文献阅读之后，对各个经典的文件系统模型有了一定的认识，同时对这些分布式文件系统对于各种问题的处理方式有了对比和参照。然后，根据 XLFS 项目的具体背景和需求，设计并实现了一个具体的特定环境下的分布式文件系统，本文就是对 XLFS 的设计和实现的描述和总结。

本论文写作主要分为以下几个部分：

论文第 1 章绪论，简要描述课题的来源、背景及研究目的，以及与课题相关领域的国内外研究现状。

论文第 2 章，简要介绍了离线下载项目的整体框架，明确 XLFS 实际应用环境。其次，针对离线下载项目的具体环境，提出该分布式文件系统的功能和性能上的具体需求。

论文第 3 章，对分布式文件系统 XLFS 实现进行了系统设计，着重论述了软件系统的架构设计，并对系统主要功能模块进行了详细的设计，为系统的实现打下了良好的基础。

论文第 4 章，描述了系统的实现方法，对系统的各个模块及对外接口进行了详细的阐述。

论文第 5 章，针对系统功能和性能两方面进行测试，并给予评价。

论文最后一章结论，总结了论文的研究成果，展望了今后的研究方向。

第 2 章 分布式文件系统需求分析

本章将分析离线下载分布式文件系统的需求，首先将介绍文件系统的运行环境离线下载项目的整体概况，明确 XLFS 在离线下载项目中的位置。其次，针对离线下载项目的具体环境，提出该分布式系统的功能和性能上的具体需求。

2.1 离线下载项目简介

如图 2-1 所示，离线下载的程序分别部署在双路机房和单路机房。其中，双路机房的服务器有两个网卡，分别连入电信和网通网络，所有用户登录 WEB 页面都登录双路机房。单路机房一共有 8 个，4 个在电信，4 个在网通，分别处理不同地域的离线下载任务。

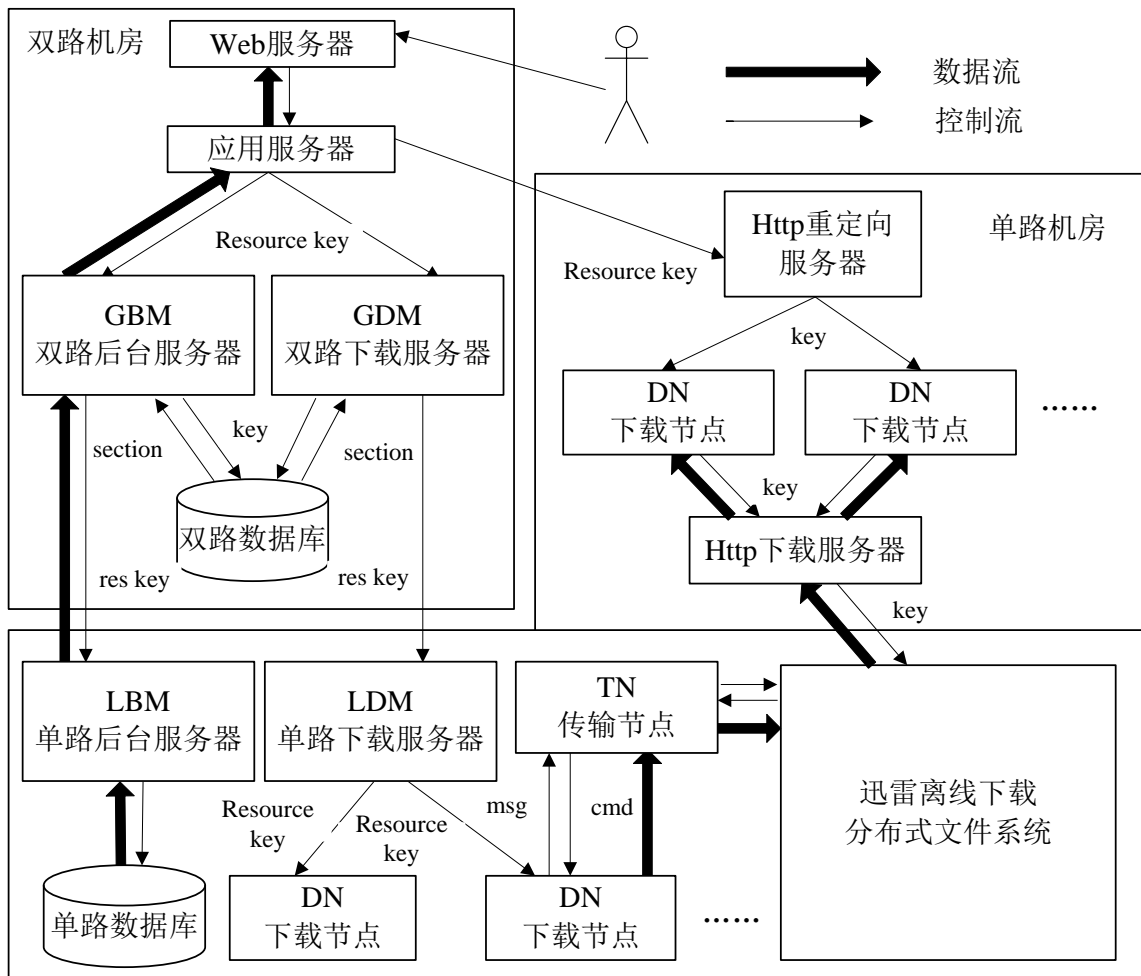


图 2-1 离线下载框图

当用户登录离线下载页面添加离线任务时，处理流程如下：

- (1)用户通过 WEB 服务器向应用服务器添加离线任务。
- (2)应用服务器提取离线任务资源关键字(url 或 gcid 或 res_id + res_type)并传给 GDM(Globe Download Manager)。
- (3)GDM 根据用户 ID 查询数据库，将该任务分配给用户所在的单路机房。
- (4)单路机房 LDM 接受资源关键字，并分配给某个无界面的客户端迅雷 DN(Download Node)下载。
- (5)下载完毕后由传输结点 TN(Transfer Node)将文件存储到迅雷分布式文件系统。

当用户登录离线下载页面下载离线任务时，处理流程如下：

- (1)用户通过 WEB 服务器向应用服务器发送请求，应用服务器根据用户 ID 将用户任务分配给相应的单路机房。
- (2)单路机房中 HTTP 重定向服务器将任务分配给具体的 DN。
- (3)DN 通过 HTTP 下载结点从 XLFS 读取文件并发送给客户的迅雷客户端。

2.2 文件系统需求

针对离线下载项目数据规模大、数据量大、增加数据多、改变原有数据少等特点，特提出了以下功能需求和非功能需求。

2.2.1 功能需求

该系统与许多以前的分布式文件系统拥有许多相同的目标，但它的设计还受到具体应用负载和技术环境观察的影响，它和早期文件系统的需求都有明显的不同，以下是该文件系统功能上的需求。

(1)**大文件管理** 系统保存一定数量的大文件。预期有几百万文件，尺寸通常是 100MB 或者以上。数 GB 的文件也很寻常，而且被有效的管理。小文件必须支持，但是不需要去优化。

(2)**流式读取和随机读取** 负载中主要包含两种读操作：大规模的流式读取和小规模随机读取。大规模的流式读取通常一次操作就读取数百 K 数据，更常见的是一次性读取 1MB 甚至等多。同一个客户机的连续操作通常是对一个文件的某个区域进行连续读取。小规模随机读取通常是在随机的位置读取几个 KB。对性能有所要求的程序通常把小规模的读批量处理并且排序，这样就不需要对文件进行时前时后的读取，提高了对文件读取的顺序性。

(3)**追加写入和随机写入** 负载中还包括许多大规模的顺序的写操作，追加数

据到文件尾部。一般来说这些写操作跟大规模读的尺寸类似。数据一旦被写入，文件就几乎不会被修改了。系统对文件的随机位置写入操作是支持的，但是不必有效率。

(4)**高效并行追加** 系统必须高效的实现良好定义的多客户端并行追加到一个文件的语意。文件经常用于"生产者-消费者"队列，或者多路文件合并。数百个生产者，一个机器一个，同时对一个文件进行追加。用最小的同步开销实现追加的原子操作是非常重要的。文件可能稍后被读取，也可能一个消费者同步的读取文件。

(5)**提供操作接口** 文件系统提供了一个类似传统文件系统的接口，虽然它并没有实现类似 POSIX 的标准 API。文件在目录中按照层次组织，用路径名来标识。支持常用的操作，如创建，删除，打开，关闭，读和写文件。

(6)**故障恢复功能** 系统由许多廉价易损的普通组件组成，它必须持续监视自己的状态，在组件失效作为一种常态的情况下，迅速地侦测、承担并恢复那些失效的组件。

2.2.2 非功能需求

离线下载项目是 XLFS1.0 的直接需求，具有实实在在的运营环境，这是 XLFS 的优势。为了满足离线下载项目的需要，有以下几个非功能性的需求值得注意。

2.2.2.1 容错性

在网络上运行的任务可能在任何时间开始或结束，释放或占有任务执行需要的相关资源，因此下载中的资源的使用是非常不稳定的，此外，网络平台的各主机在地理位置上的差异，造成数据传输也有一定的时延，并且这个时延是无法估计的。由于这个问题的产生是网络传输的根本性质，因此无法得到精确的解决。

在正常情况下，作业的执行情况不会频繁的变化，文件系统需求的容错性要求程序可以保证数据读取过程中，在服务器中存有不正确数据的情况下，能得到一份完整的没有被破坏的数据，并且在相对稳定的实验环境下，数据传输结果可以保证百分之百的准确性。

2.2.2.2 安全性

离线下载对该文件系统的应用日益增多，随之将至的是越来越重的压力。如果被恶意的应用程序使用该文件系统，定将会使其不堪重负。所以要确保文

件系统不被恶意的应用程序使用，以减轻其压力。

根据安全性的需求，每个客户端申请应用该文件系统时，由主服务器分配一个安全会话的 `SessionId`，作为元数据一并传送给客户端。客户端连接块服务器时，块服务器将检验这个 `SessionId`，如果不合事先预定的算法，将不为该客户端服务。

2.2.2.3 可扩展性

随着离线下载的用户与日俱增，离线下载服务器将不断扩充。与之相应，分布式系统的存储的资源也将越来越多，所以分布式系统的服务器将要不断扩展。而离线下载只是分布式的一个应用，随着以后该文件系统的成熟，应用程序将不断增多，因此文件系统需要提供的客户端接口也将需要扩展。

根据可扩展性的需求，本课题将文件系统分为两部分，一部分作为存储具体资源的介质，另一部分用于存储相关元数据并管理着第一部分的服务器。这样随着数据的增多，作为存储介质的服务器可以随时扩展。另外，我们将接口与服务器的实现相分离，以满足接口不断扩展的需求。

2.2.2.4 用户体验

离线下载面临着大量用户同时作业的情况，相应的分布式文件系统也将被大量应用实例所调用运行。为了提高用户的体验度，要求文件系统响应时间必须在客户可接受范围之内，一般不超过 3 秒。

根据用户体验需求，文件系统将客户端响应与任务运行线程相分离。应用程序添加任务时，确认数据的完整性与正确性后，将先反回结果，再由后台运行该任务。

2.2.2.5 其他非功能约束

由于分布式文件系统是在公司范围内为离线下载项目及后续应用开发的，为了遵循原系统的体系结构以及开发方式，统一编码规范，因此本课题还存在一些非功能性约束，包括：

- (1)限定 XSocket 开发框架，与项目组其它项目保持一致。
- (2)传输数据使用迅雷架构组的编解码流组件；
- (3)日志记录使用 Log4cplus 接口。

2.3 文件系统适用范围

迅雷离线下载分布式文件系统是一个分布式网络存储系统，产品内容包括相应的分布式网络存储策略、分布式文件系统规范、软件、硬件及相关用户手册等。

分布式文件系统面向迅雷离线下载项目，致力于为本公司、部门多个应用软件提供一套资源分布式存储解决方案，适用于拥有较大规模数据量的不可轻易丢失数据的应用程序软件。分布式文件系统通过将文件分割、重复存储等策略以保证存储文件的完整与安全性。

分布式文件系统对数据的管理有：确保数据的正确性；存储读取数据的权限控制；数据的副本范围控制；数据的存取历史记录；数据的完整性校验等。

2.4 本章小结

本章首先介绍了分布式文件系统运行环境，对离线下载项目进行简单的介绍。其次，介绍文件系统的需求，理解各项功能需求和非功能需求并进行分析。最后，说明文件系统的适用范围，该分布式文件系统并不适合所有应用服务，只对公司内部与离线下载项目类似的应用服务适用。

第 3 章 分布式文件系统的设计

3.1 文件系统设计概要

本节介绍 XLFS 的设计概要，根据第 2 章需求分析，首先设计 XLFS 的整体架构并给予说明。其次，提出单一主服务器设计方案，最后详细描述主服务器的元数据设计。

3.1.1 文件系统整体架构

一个 XLFS 集群包含一个主服务器和多个块服务器，被多个客户端访问，如图 3-1 所示。这些机器通常都是普通的 Linux 机器，运行着一个基于用户层的服务进程。如果机器的资源允许，而且运行多个程序带来的低稳定性是可以接受的话，可以很简单的把块服务器和客户端运行在同一台机器。

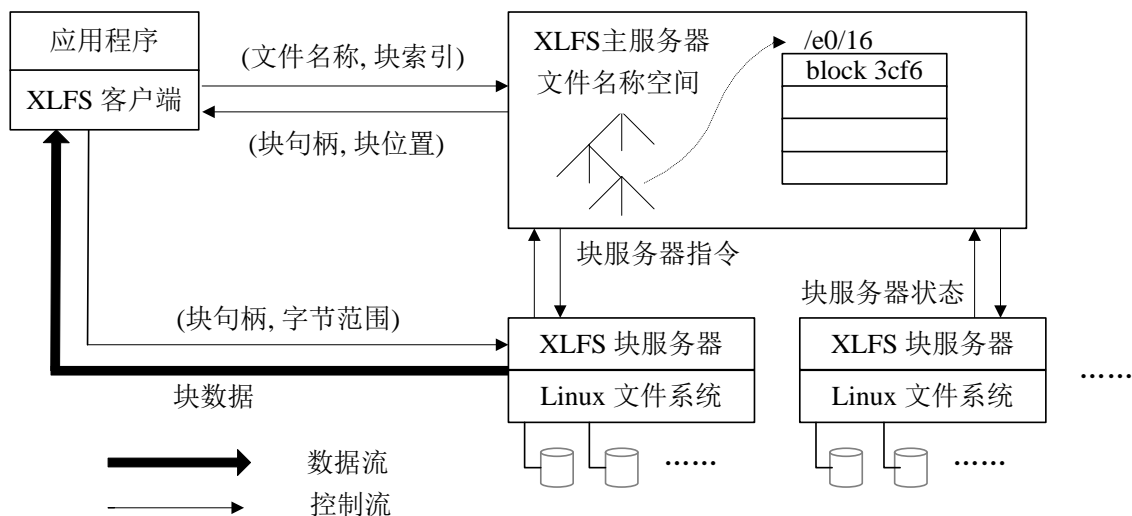


图 3-1 XLFS 架构图

离线下载绝大多数文件都是大文件，文件被分割成固定尺寸的块。在每个块创建的时候，服务器分配给它一个不变的、全球唯一的 64 位的块句柄对它进行标识。块服务器把块作为 linux 文件保存在本地硬盘上，并根据指定的块句柄和字节范围来读写块数据。为了保证可靠性，每个块都会复制到多个块服务器上。缺省情况下，保存三个备份，不过用户可以为不同的文件命名空间设定不同的复制级别。

如图 3-1 所示，主服务器管理文件系统所有的元数据。这包括名称空间，访问控制信息，文件到块的映射信息，以及块当前所在的位置。它还管理系统

范围的活动，例如块租用管理，孤儿块的垃圾回收，以及块在块服务器间的移动。主服务器用心跳信息周期地跟每个块服务器通讯，给他们以指示并收集他们的状态。

XLFS 客户端代码被嵌入到每个程序里，它实现了 XLFS 的 API，帮助应用程序与主服务器和块服务器通讯，对数据进行读写。客户端跟主服务器交互进行元数据操作，但是所有的数据操作的通讯都是直接和块服务器进行的。

不管是客户端还是块服务器都不缓存文件数据。客户端缓存几乎没什么好处，因为大部分程序读取巨大文件的全部，或者工作集太大无法被缓存。不进行缓存简化了客户端和整个系统，因为无需考虑缓存相关的问题。(不过，客户端会缓存元数据。)块服务器不需要缓存文件数据的原因是，块保存成本地文件形式，Linux 的缓冲器会把经常被访问的数据缓存在内存中。

3.1.2 单一主服务器设计方案

单一的主服务器的方案大大简化了设计，这样主服务器可以通过全局的信息精确确定块的位置以及进行复制决定。然而，必须减少主服务器对数据读写的影响，避免使主服务器成为系统的瓶颈。客户端不通过主服务器读写数据。反之，客户端向主服务器询问它应该联系的块服务器。客户端短期缓存这些信息，后续的操作直接跟块服务器进行。

一次简单读取的流程如下，首先，利用固定的块尺寸，客户端把文件名和程序指定的字节偏移转换成文件的块索引。然后，它把文件名和块索引发送给主服务器。主服务器回答相应的块句柄和副本们的位置。客户端用文件名和块索引作为键值缓存这些信息。

然后客户端发送请求到其中的有一个副本处，一般会选择最近的。这个请求指定了块的块句柄和字节范围。对同一块的更多读取不需要客户端和主服务器通讯，除非缓存的信息过期或者文件被重新打开。实际上，客户端通常在一次请求中查询多个块，而主服务器的回应也可以包含紧跟着这些请求块后面的块的信息。这些额外的信息实际上，在没有代价的前提下，避免了客户端和服务器的未来的几次通讯。

3.1.3 元数据设计

元数据主服务器保存三种主要类型的元数据：文件和块的命名空间，文件到块的映射，以及每个块副本的位置。所有的元数据都保存在主服务器的内存里。除此之外，前两种类型(命名空间和文件块映射)的元数据，还会用日志的方

式保存在主服务器的硬盘上的操作日志内，并在远程的机器内复制一个副本。使用日志，可以简单可靠的更新主服务器的状态，而且不用担心服务器崩溃带来的数据不一致的风险。主服务器不会持久化保存块的位置信息。主服务器在自己启动以及块服务器加入集群的时候，询问块服务器它所包含的块的信息。

3.1.3.1 内存内数据结构

因为元数据保存在内存中，所以主服务器操作的速度很快。而且，主服务器可以在后台，简单而高效的周期扫描自己的整个状态。这种周期性的扫描用来在块服务器间实现块的垃圾收集的功能，用来实现块服务器失效的时复制新副本的功能，用来实现负载均衡的块移动的功能，以及用来实现统计硬盘使用情况的功能等。

这种纯内存的机制一种可能的问题是，块的数量和整个系统的容量都受限于主服务器拥有的内存尺寸。不过实际上这不是一个严重的限制。对于每个 64MB 的块，服务器只需管理不到 64 字节的元数据。大多数的块是满的，因为每个文件只有最后一块是部分填充的。类似的，每个文件的命名空间通常在 64 字节以下，因为保存的文件名是用前缀压缩算法压缩过的。

如果需要支持更大的文件系统，为主服务器增加额外内存的花费是很少的，而把元数据保存在内容带来了系统的简洁性，可靠性，高性能和灵活性。

3.1.3.2 块位置

主服务器并不保存哪个块服务器有给定块的信息的持久化记录。它只是在启动的时候向块服务器要求这些信息。然后主服务器就可以保持它的信息最新，因为它控制了所有的块位置分配，而且用规律的心跳信息监控块服务器的状态。

开始的时候，试图把块位置信息持久化的保存在主服务器，但是发现在启动的时候查询块服务器，然后定期询问的方式更简单。这简化了在块服务器在加入和离开集群，改名，失效，重启等情况下，主服务器和块服务器的同步问题。在一个有数百台服务器的集群，这些事件发生的太频繁了。

理解这个设计思想的另外的途径是，理解块服务器才能最后确定一个块是否在它的硬盘上。系统没有打算在主服务器上面管理一个这些信息的全局视图，因为块服务器的错误可能会导致块自动消失，亦或者管理员可能会改变一个块服务器的名字。

3.1.3.3 操作日志

操作日志包含核心元数据变化的历史记录，这对 XLFS 很重要。这不仅是因为它是元数据唯一的持久化存储记录，而且因为它起到了定义同步操作顺序

的逻辑时间线的作用。文件和块以及他们的版本，都是唯一和持久地由他们创建时的逻辑时间标识的。

因为操作日志很重要，必须稳定的保存它，并保证只有在元数据的变化被持久化保存后，这种变化才对客户端可见。否则，即使块没有发生任何问题，仍有可能丢失整个文件系统，或者丢失近期的客户端操作。所以，把操作日志复制到多台远程机器，而且仅在把最近的日志记录写入本地以及远程机器的硬盘后，才会对客户端操作进行响应。写入之前主服务器批量处理数个日志记录，减少写入和复制的负载。

3.2 服务器交互设计

3.2.1 租约和变更

多个数据块副本写入不同块服务器时，为了解决数据的一致性，决定采用租约变更方案。变更就是一个会改变块内容或者元数据的操作，例如写入或者记录追加。每个变更执行在块的所有副本上。系统使用租约来保持多个副本间变更顺序的一致性。主服务器为其中一个副本签署一个块租约，暂把这个副本叫做主块。主块选择对块所有操作的一系列顺序。进行操作的时候所有的副本遵从这个顺序。这样全局的操作顺序首先由主服务器选择的租约生成顺序决定，然后由租约中主块分配的序列号决定。

如图 3-2 所示，可用如下这些数字步骤，展现写入操作的控制流程。

(1)客户机向主服务器询问哪一个块服务器保存了当前的租约，以及其他副本的位置。如果没有一个块服务器有租约，主服务器就选择一个副本给它一个租约(没有被显示出来)。

(2)主服务器回复主块的标识符以及其他副本的位置。客户机为了后续的操作缓存这个数据。只有主块不可用，或者主块回复说它已经不在拥有租约的时候，客户机才需要重新跟主服务器联络。

(3)客户机把数据推送到所有的副本上。客户机可以用任意的顺序推送。每个块服务器会把这些数据保存在它的内部的缓冲区内，直到数据被使用或者过期。通过把数据流和控制流分离，可以基于网络负载状况对昂贵的数据流进行规划，以提高性能，而不用去管哪个块服务器是主块。

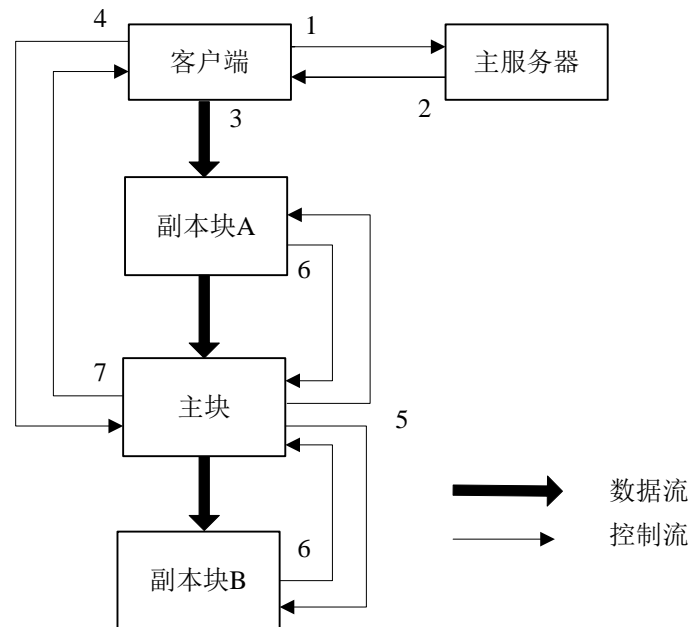


图 3-2 写的控制和数据流程

(4)所有的副本都被确认已经得到数据后，客户机发送写请求到主块。这个请求标识了早前推送到所有副本的数据。主块为收到的所有操作分配连续的序列号，这些可能来自不同的客户机。它依照序列号的顺序把这些操作应用到它自己的本地状态中。

(5)主块把写请求传递到所有的二级副本。每个二级副本依照主块分配的序列号的顺序应用这些操作。

(6)所有二级副本回复主块说明他们已经完成操作。

(7)主块回复客户机。任何副本产生的错误都会报告给客户机。错误的情况下，主块和一些二级副本可能成功的写入了数据。(如果主块写入失败，操作就不会被分配序列号，也不会被传递。)客户端请求被确认为失败，已经修改的区域保持一致的状态。客户机代码通过重复失败的操作来处理这样的错误。在完全从头开始写入之前，可以先从步骤 3 到步骤 7 进行几次尝试。

主块获取租约的方式采用状态机的设计模式，状态机有四个状态：初始状态(leaseInit)、活跃状态(leaseActive)、更新状态(leaseRenewing)、结束状态(leaseEnd)，如图 3-3 所示为租约的转换图，4 个状态间的转换关系如下。

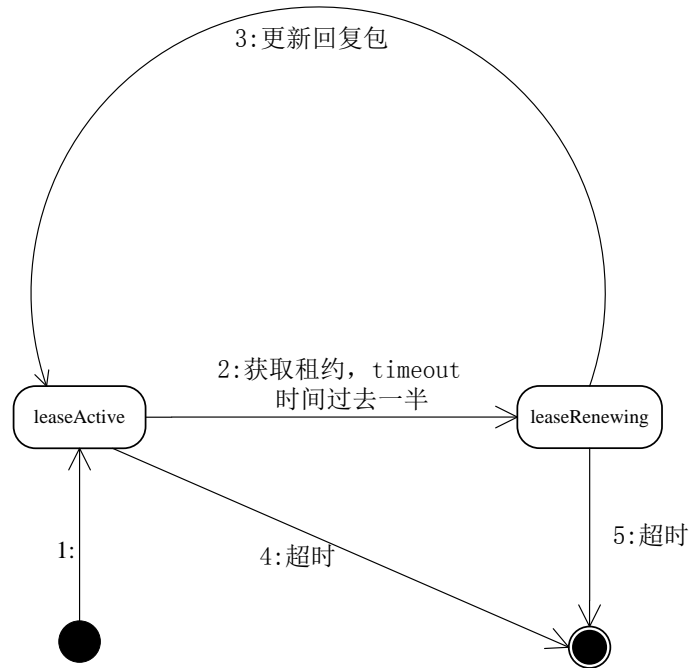


图 3-3 lease 状态转换图

(1)Lease 从初始状态转换为 LeaseActive 状态，此状态提供获取租约的 get_lease 方法，返回值是获得的序列号。

(2)Lease 当 get_lease 方法被调用，如果 timeout 时间已经过去一半，会给主服务器 Master 发请求延长 lease，从而进入 LeaseRenewing 状态，此状态一样提供 get_lease 方法。

(3)LeaseRenewing 状态下收到更新租约回复包，会进入 LeaseActive 状态并更新 timer 超时时间，此外，如果 LeaseRenewing 超时，会重复 3 次给 Master 发更新租约请求包。

(4)整个状态机超时都会进入终结状态。

3.2.2 数据流设计方案

为了提高网络的效率，可以把数据流和控制流分开。控制流从客户机到主块然后再到所有二级副本的同时，数据顺序推送到一个精心选择的管道形式的块服务器链。我们的目标是完全利用每个机器的带宽，避免网络瓶颈和延迟过长的连接，最小化推送所有数据的耗时。

为了完全利用每个机器的带宽，数据顺序推送到一个块服务器链，而不是分布在其他的拓扑下(例如，树)。这样，每个机器的带宽就会完全用于尽快的传输数据，而不是分散在多个副本间。

为了避免网络瓶颈和延迟过长的连接，每个机器都把数据传送到在网络拓扑中最近的机器。假设客户推送数据到块服务器 S1 至 S4。它把数据传送给最近的块服务器 S1。S1 把数据传递给 S2 到 S4 之间最近的机器 S2。同样的，S2 把数据传递给 S3 和 S4 之间更近的机器。系统的网络拓扑非常简单，简单到用 IP 地址就可以计算出节点的远近。

最后，可以利用在 TCP 连接上管道化数据传输来最小化延迟。块服务器一旦得到一些数据，马上开始传递它们。管道化对系统帮助极大，因为系统使用全双工连接的交换网络。马上发送数据不会降低接收的速度。没有网络拥塞的情况下，传送 B 字节的数据到 R 个副本的理想时间是 $B/T + RL$ ，T 是网络的吞吐量，L 是两台机器传输字节的延迟。系统的网络连接是 100Mbps(T)，L 远小于 1ms。这样，1MB 的数据理想情况下会在 80ms 左右分发出去。

3.2.3 原子性记录追加

XLFS 提供了一个原子性的记录追加操作。传统的写入操作，客户机指定数据写入的偏移位置。同意范围的并行写入是不可串行的，范围内会包括来自不同客户机的数据片段。而对于记录追加，客户机只能指定数据。XLFS 至少一次自动地把数据追加到文件中 XLFS 选定的偏移位置，然后把偏移位置返回给客户机。这类似于在 Unix 下，多个并行写入者在没有竞争条件下，对以 O_APPEND 模式打开的文件的写入^[34]。

记录追加在该分布应用中使用的非常频繁，许多客户机并行地追加数据到同一个文件。如果采用传统的写入方式，客户机需要额外的复杂和代价昂贵的同步机制，例如一个分布的锁管理器。在日常工作中，这样的文件经常服务于多生产者/单消费者队列，或者包含来自多个不同客户机的合并的结果。

记录追加是控制流程的一个操作，仅有主块有点额外的控制逻辑。客户机把数据推送给文件最后一个块的所有副本，然后发送请求给主块。主块检查对当前块的记录追加操作是否会造成块超过最大值(64MB)。如果是这样，它首先把块填充到最大值，告诉所有二级块做同样的操作，然后回复客户机说明需要对下一个块也进行操作。(一次记录追加操作的数据尺寸严格规定不超过 4 倍的最大块尺寸，这样即使最坏情况下存储碎片数量仍旧控制在可接受的级别。)如果记录不超过最大尺寸，这是比较常见的，主块把数据追加到自己的副本内，然后通知二级副本把数据写在跟主块一样的位置，最后通知客户机操作成功完成。

如果任何副本上追加失败，客户端重新进行操作。所以，同一块的不同副

本可能包含不同的数据，可能包括一个记录全部或者部分的重复。XLFS 并不保证所有的副本是字节级别一致的。它仅保证数据会作为原子性单元被至少写入一次。这个属性是由如下的简单观察推导而来的，如果操作报告成功，数据一定已经写入在同一块的所有副本的相同偏移位置上。此后，所有的副本至少达到了记录尾部的长度，任何额外的记录都会附加到更大的偏移或者不同的块，即使其他的块后来成为了主块。在一致性保障措施的术语中，成功被执行追加操作的区域的数据是已定义的(所以也是一致的)，反之则是不一致的(也就是未定义的)。

3.2.4 基本操作设计

本节将讨论 XLFS 几个关键步骤如写操作、读操作、追加操作，在考虑调用关系、参数、错误定义的基础上，给出详细设计。

3.2.4.1 write 操作

如图 3-4 所示为 XLFS 写操作的时序图，其过程如下。

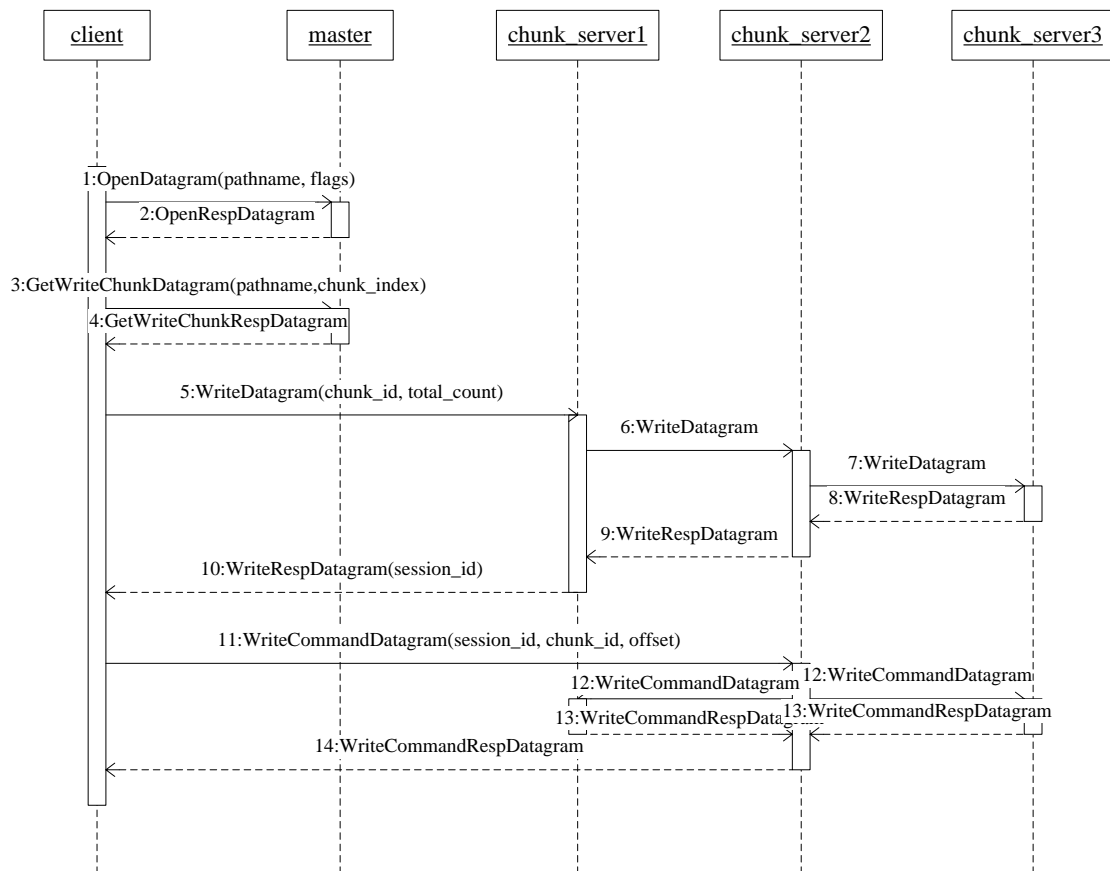


图 3-4 write 操作全局时序图

(1)应用程序调用 client 打开一个文件，以文件全路径名 `pathname`，以及打开标记(flags)作为参数，master 根据参数返回打开结果。

➤ flags 设置了 `O_CREAT|O_TRUNC`，文件存在，将文件截断为 0；文件不存在则创建文件；

➤ `O_CREAT|O_EXCL`，如果文件存在，返回错误(`ERR_EXIST`)；无，文件不存在则返回错误(`ERR_NOENT`)

➤ `O_CREAT`，如果文件存在，不截断文件也不创建；不存在则创建文件返回。

(2)返回值 `result` 合法返回 0，非法时 `result < 0`，同时设错误码 `ERR_EXIST`，`ERR_NOENT`，`ERR_LOCKED`(要修改 namespace 但其被锁定)。

(3)此步为可选的，假如缓冲中存在要写的块的信息，就无需向 master 请求获得要写的块的信息。client 以文件路径名、要获得块的索引号(即这个文件的第几块)为参数请求要写的块的信息，master 参看相应的索引号块，如果不存在，则会去 `chunk_server` 分配新块，分配成功后进入下一步处理。master 接着看是否有 lease 存在，如果不存在，会选定一个 `chunk_server` 作为主块，并通知他创建 lease。这里隐含着如果要写的块前面的块不存在，不影响该块的创建。

(4)Master 将 `primary` 块信息放在最前面，返回 client 请求块的信息。(假如前面分配好的 3 个块全部失效，这时候该如何处置？)。 `result < 0`，设错误码 `ERR_LOCKED`(master 锁定 namespace)；`ERR_NOSPACE`(已达最大可分配块，无块可以分配)；`ERR_BUSY`(该块正在做 copy、move 等操作，不能写)。

(5)Client 将要写入的数据发送至某个 `chunk_server`(根据 network topology)，`chunk_server` 计算一个 `session_id` 放入 `WriteDatagram`；然后 `chunk_server` 将接收到的数据转发给下一个 `chunk_server`，下一个 `chunk_server` 检查 `session_id` 是否为空，发现不为空就不用再计算 `session_id`。重复这一过程直到最后一个 `chunk_server` 接收到数据。(WriteDatagram 需不需要 `chunk_id` 依赖于 `session_id` 是否需要其参与计算)。

(6)同 5。

(7)同 5。

(8)同 10。

(9)同 10。

(10)Chunk_server 接受完数据后，确定不需要转发，就会给上一个 `chunk_server` 发送回复。由于 `chunk_server` 可能在等下一级 `chunk_server` 回复时超时，这时就需要返回给上一级发生错误的 `chunk_server`，并置错误码

ERR_TIMEOUT。(存不存在某个 chunk_server 无足够内存缓存数据的情况?置 ERR_NOSPACE?)。

(11)Client 接收到数据传输完成的确认后, 给 primary 发送写命令, 参数包括 session_id, 要写入的 chunk_id, 以及偏移位置。Primary 接收到写命令后, 首先检查是否存在 lease。如果不存在, 就会直接返回错误给 client, 同时有错误码 ERR_NOLEASE。如果存在 lease, 就由 lease 给当前的修改命令(write 或者 append)编号。

(12)然后给两个 secondary chunk_server 发送编了号的写命令, 并等待回复。

(13)Chunk_server 接收到命令后, 将对应的数据写入磁盘, 如果先收到了编号较大的命令, 在小编号命令到来之前不会执行。将执行的结果返回给 primary。

(14)Primary 将发生问题的 chunk_server 告知 client, 并置相应错误码(ERR_TIMEOUT:某 chunk_server 命令执行超时)。

(15)写失败时, client 会先重复 5-14 的序列 3 次, 如果失败, 再重复 3-14 的序列 3 次。如果还是失败, 会返回给应用程序错误码。

3.2.4.2 read 操作

如图 3-5 所示为 XLFS 读操作的时序图, 相对简单, 其过程如下。

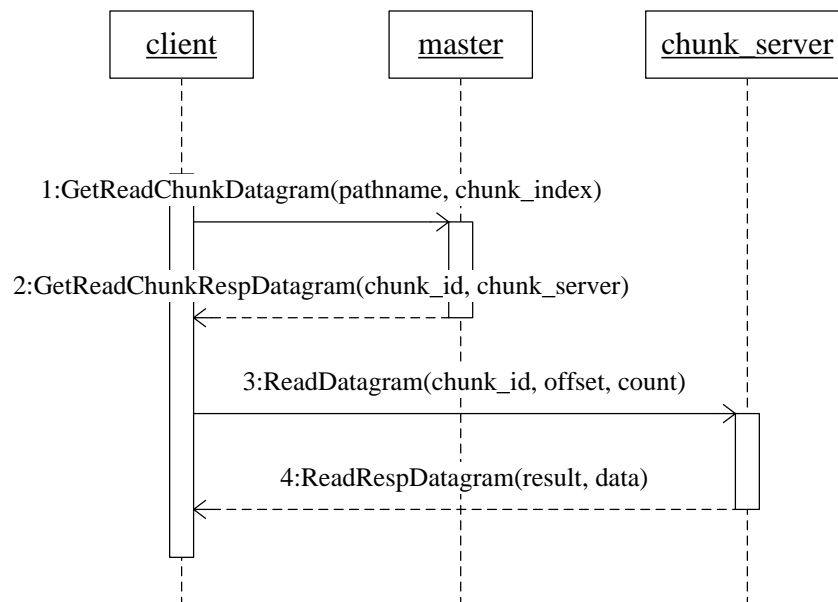


图 3-5 read 操作全局时序图

Read 过程相对简单, 1-2, 3-4 都会重试 3 次。

(1)如果块信息在缓冲中找不到, 会去 master 请求相应块的信息, 以文件路

径名和索引号以及预取数为参数。

(2) Master 返回给最大数目为预取数的块信息给 client，包括 chunk_id 和 chunk 所在的若干个 server 的 IP。如果块不存在，则置错误码 ERR_NOENT。

(3) Client 接到请求后，根据网络拓扑发 ReadDatagram 包去读取数据。以 chunk_id, 要读的字节数, 要读的 offset 为参数。

(4) Chunk_server 接到读请求后，会查看是否该块存在，如果不存在，则会返回 ERR_NOENT 错误。否则，会读取数据返回给 client。

3.2.4.3 append 操作

如图 3-6 所示为 XLFS 追加操作的时序图，与写操作类似，其过程如下。

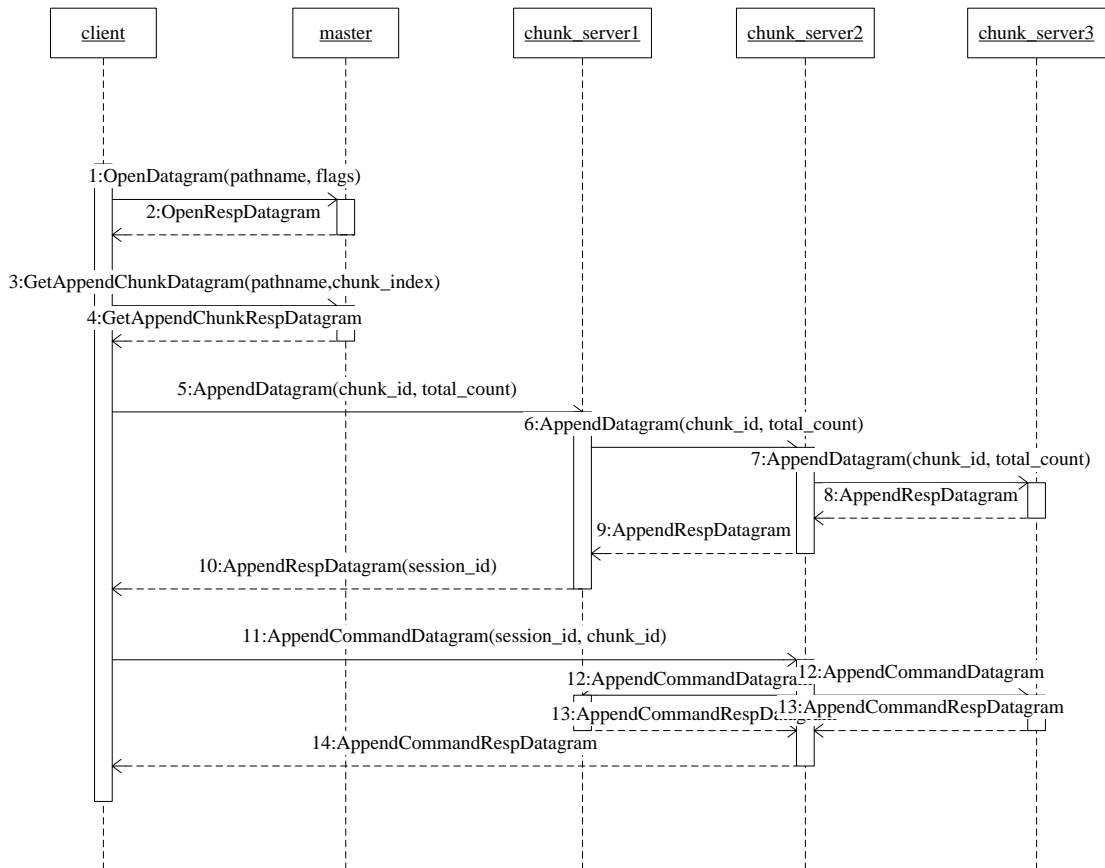


图 3-6 append 过程全局序列图

Append 过程跟 Write 很相似，主要区别如下：

(1) 流程第 3 步，申请的 chunk_index 只是作为参考值。如果 chunk_index ≤ 当前最后一个块索引值，返回最后一个块；chunk_index == (最后一个块索引值 + 1)，创建新块；其它情况下，返回错误，置错误码 ERR_NOENT。

(2) 流程第 4 步，错误码 ERR_LOCKED (master 锁定 namespace)；

ERR_NOSPACE(已达最大可分配块, 无块可以分配); ERR_BUSY(该块正在做 copy、move 等操作, 不能写); ERR_NOENT(索引值超出末尾+1)。

(3) 流程第 11 步, Primary 接收到 append 命令后, 先检查是否有足够空间写该块, 如果没有, 则直接转到 14 步返回 client 错误码 ERR_NOSPACE。Primary 计算该 record 应该写在该块的偏移位置, 生成类似写命令的命令, 并编号转发。

(4) 流程第 12 步, 略有不同, 可以考虑和 WriteCommandDatagram 一样的包。

3.3 主服务器设计

本节介绍主服务器内部的一些操作, 并在本章前两节的基础上, 给出主服务器设计类图。

3.3.1 名称空间管理和锁

主服务器的许多操作会花费较长的时间, 但不希望这些操作的运行延迟其他的主服务器操作。所以允许多个操作同时进行, 用名称空间的区域之上的锁来保证相应的秩序。

不同于许多传统文件系统的是, XLFS 没有一个用来列出目录内全部文件的, 每个目录的数据结构。而且不支持同一文件或者目录的别名(Unix 术语中的符号链接或者硬链接)。XLFS 展现名称空间的逻辑就像一个全路径映射到元数据的查找表。利用前缀压缩, 这个表可以高效的在内存中展现。名称空间树内的每个节点(绝对路径文件名和绝对路径目录名)都有一个与之对应的读写锁^[35]。

每个主服务器操作运行之前都需要获得一系列的锁。例如, 如果操作包含 /d1/d2/.../dn/conf, 首先获得目录/d1, /d1/d2, ..., /d1/d2/.../dn 的读取锁, 以及全路径/d1/d2/.../dn/conf 的读写锁。这里 conf 可以是一个文件也可以是一个目录, 取决于操作本身。

举个例子演示下锁机制, 例如/home/user 被剪切到/save/user 的时候, 为防止文件/home/user/lvlei 的创建。快照操作获得/home 和/save 的读取锁, 以及/home/user 和/save/user 的写入锁。文件创建操作获得/home 和/home/user 的读取锁, 以及/home/user/lvlei 的写入锁。这两个操作会顺序执行, 因为它们试图冲突地获取/home/user 的锁。文件创建操作不去请求父目录的写入锁, 因为这里没有"目录", 或者类似 inode 的数据结构, 用来防止修改。文件名的读取锁完全可以防止父目录被删除。

这种锁方案的好的属性是它支持对同一目录的并行操作。例如, 同一个目

录的多个文件创建操作可以并行执行：每一个操作获得目录名的读取锁和文件名本身的写入锁。目录名的读取锁完美的防止目录被删除或是改名等。文件名的写入锁保证不会创建同名文件两次。

因为名称空间可以有許多节点，所以读写锁需要的时候才会被分配，一旦不再使用就会被删除。锁的获取依据一个全局一致的顺序来避免死锁：首先由名称空间的层次决定顺序，统一层次内的锁顺序由字典顺序决定。

3.3.2 垃圾回收

当应用程序删除了一个文件，主服务器立刻把删除记录到日志里就像其他的改变一样。然而，它不是马上回收资源，而是把文件改成一个包含删除时间戳的隐藏的名字。当主服务器对文件系统命名空间进行常规扫描的时候，它会删除所有有如此隐藏文件名的文件，如果它们已经存在超过 3 天了(这个时间间隔是可以被设置的)。直到此时，文件仍旧可以用新的特殊的名字访问读取，可以被反删除，通过改名把它变为正常文件。如果隐藏文件从名称空间中移走，保存在内存中它的元数据就被删除了。这有效的服务于对所有块的连接^[36]。

在相似的对块命名空间的常规扫描，主服务器找到孤儿块(无法从任何文件到达的块)并擦除它们的元数据。在与主服务器交换数据的心跳信息中，每个块服务器报告它拥有的块的信息的一个子集，然后主服务器回复指出哪些块在主服务器的元数据中已经不存在了。块服务器上面就是任意删除这些块的副本了。

如果块服务器失效，或者块服务器当机的时候错过了一些操作，块副本会过期。对于每个块，主服务器保存一个块版本号，用来分辨当前副本和过期副本^[37]。

无论何时主服务器获得一个块的新租约，它增加块的版本号，然后通知当前副本。主服务器和这些副本都把新的版本号记录在它们的持久化存储的状态中。这发生在任何客户机得到通知以前，发生在它写入块之前。如果另外一个块服务器正好不可用，那么它的块版本号就不会被更新。当这个块服务器重新启动，向主服务器报告它拥有的块的集合以及相应的版本号的时候，主服务器就会检测出来块服务器包含过期的块。当主服务器看到一个比它们的记录更高的版本号，主服务器假定那些块服务器在获取租约的时候失效了，所以选择更高的版本作为当前的版本。

主服务器在周期的垃圾回收中移除所有的过期副本。在这之前，当回复客户机的块信息请求的时候，主服务器只是高效的当作那些过期的块不存在而已。作为另外的保护措施，主服务器通知客户机哪个块服务器持有租约的时候，当

主服务器指示块服务器从哪个块服务器读取数据进行克隆操作的时候，主服务器的信息都包含了块的版本号。客户机或者块服务器执行操作时会验证版本号，这样它们就可以总是访问当前版本的数据了。

3.3.3 主服务器类的设计

如图 3-7 所示为主服务器类图，其核心类是 Manager 类，在 Manager 类 impl_init 函数中将其它类进行初始化，循环执行的 run 函数调用各个类的 epoll 接口。

ChunkServerSystem 类负责管理各个块服务器，垃圾回收功能也在这个类中实现。LeaseManager 类负责块服务器租约的管理，包括租约的发放、变更等。其余各类也都类似，在 epoll 接口中处理各自的事务。

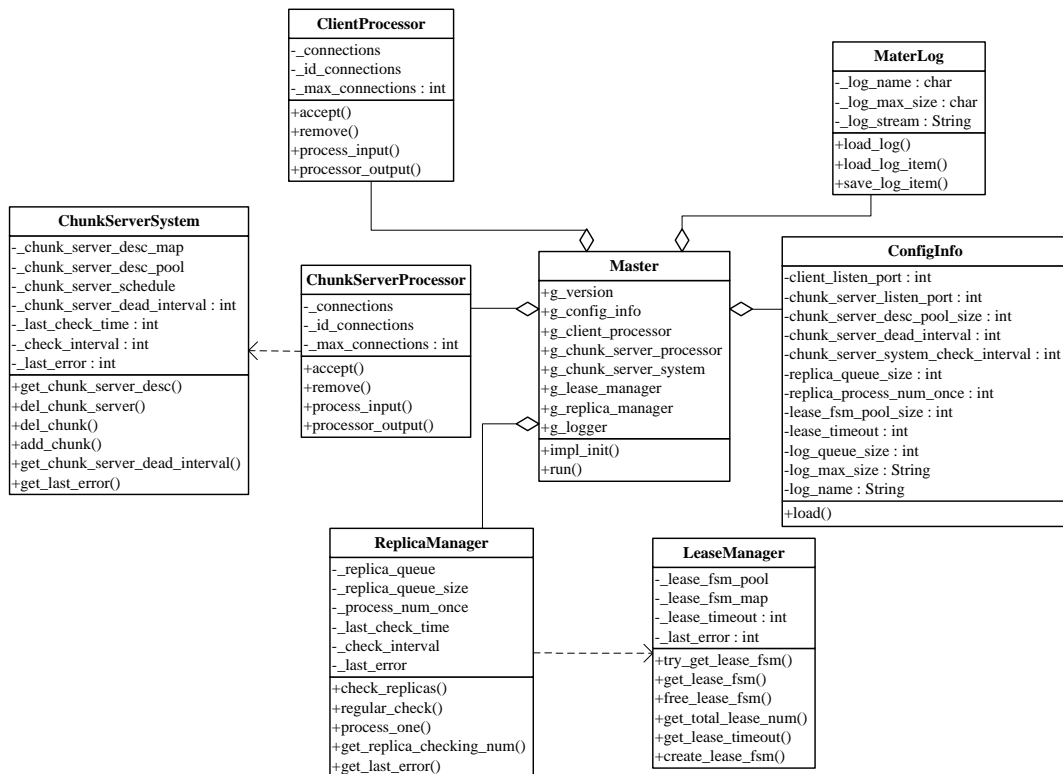


图 3-7 主服务器类图

3.4 块服务器设计

一个 XLFS 集群拥有许多块服务器，本着节约成本的原则，这些块服务器都运行在普通 linux 机器上，块服务器是 XLFS 存储文件的实际载体。

3.4.1 块尺寸选择

块尺寸是关键设计参数之一。该文件系统选择了 64MB，这远大于一般文件系统的块尺寸。每个块副本保存在块服务器上的 Linux 文件内，尺寸只在需要的时候才扩展变大。惰性空间分配避免了造成内部碎片带来的空间浪费，或许对巨大的块尺寸最没有争议的就是这点^[38]。

巨大的块尺寸有几个重要的好处。首先，它减少了客户端和主服务器通讯的需求，因为对同一个块的读写，只需要一次用于获得块位置信息的与主服务器的通讯。对工作负载来说，这种减少尤其明显，因为应用程序经常连续读写巨大的文件。即使是对小规模随机读取也有好处，因为客户端也可以轻松的缓存一个数 TB 工具集的所有块位置。其次，由于块尺寸很大，所以客户端会对一个给定的块进行许多操作，这样就可以通过跟块服务器保持较长时间的 TCP 连接来减少网络负载。第三，它降低了主服务器需要保存的元数据的尺寸。这就允许把元数据放在内存中，这样会带来一些其他的好处^[39]。

另一方面，巨大的块尺寸，即使配合惰性空间分配，也有它的缺点。小文件包含较少的块，甚至可能是一块。如果有许多的客户端访问同一个小文件，那么存储这些块的块服务器就会变成热点^[40]。在实践中，热点没有成为主要的问题，因为应用程序通常是连续的读取多个块的大文件。

然而，当 XLPS 用于一个批处理队列系统的时候，热点还是会产生的：一个执行文件被写为 XLFS 的一个单块文件，然后在数百个机器上同时启动。保存这个执行文件的几个块服务器被数百个并发请求访问到过载的地步。为了解决这个问题，系统采用更高的复制参数来保存这类的执行文件，以及让批处理队列系统错开程序的启动时间。另一个可能的长期解决方案是，在这样的情况下，允许客户端从其他客户端读取数据。

3.4.2 完整性保证与校验和

每个块服务器利用校验和来检查存储的数据是否损坏。一个拥有数百台机器数千个硬盘的 XLFS 集群，硬盘故障带来的数据损坏和丢失是非常频繁的。可以利用块的其他副本修复损坏，但是通过比较多台块服务器的数据来检查损坏是非常不可行的^[41]。而且，不一致的副本可能是合法的：XLFS 操作的语义，

特别是前面讨论过的原子性的记录添加，不保证产生完全一致的副本。所以，每个块服务器必须通过校验和，独立的验证它自己拥有的拷贝的数据完整性。

块被分为 64MB 的大小。每个块有一个对应的 32 位的校验和。和其他元数据一样，校验和保存在内存，持续化存储在日志里，与用户数据是分离的。

不管是对客户机还是其他的块服务器，在读取操作中，块服务器在返回任何数据之前验证读取范围的数据的校验和。这样块服务器就不会把损坏的数据传播给其他机器。如果块和记录的校验和不符，块服务器给请求者返回一个错误信息，然后把这个不匹配报告给主服务器。作为回应，请求者读取其他的副本，而主服务器从其他的副本克隆这个块。一个好的新块布置好后，主服务器指导报告不匹配的那台块服务器删除这个副本。

3.4.3 块服务器类的设计

如图 3-8 所示，块服务器核心类为 `ChunkServer` 类，在 `ChunkServer` 类的初始化函数中会启动两个线程池，线程池中的线程数由配置参数决定。其中一个线程池中的线程实现为 `RwDiskThread` 类，负责将数据块写到本地文件系统，另一组线程实现为 `MasterChunkThread` 类，负责和 `Master` 通信，上报本地数据块信息，更新数据块等。

`ClientProcessor` 类负责监听客户端消息请求，将消息请求放入请求队列，并循环查看结果队列，如果有结果队列不为空，则将处理结果返回给客户端。

`ConfigInfo` 类负责读取配置文件的信息，在程序启动时会调用该类的 `load` 函数，将配置文件中的数据读入内存。

`ChunkServerLog` 类实现 `Log4cplus` 接口，负责记录程序日志，便于开发与维护人员查找错误信息。

`LeaseState` 类是 4 个租约状态机类的基类，继承自它的 4 个状态类分别响应不同的消息事件，并在响应消息事件函数中处理事务逻辑，以实现块服务器租约的获取。

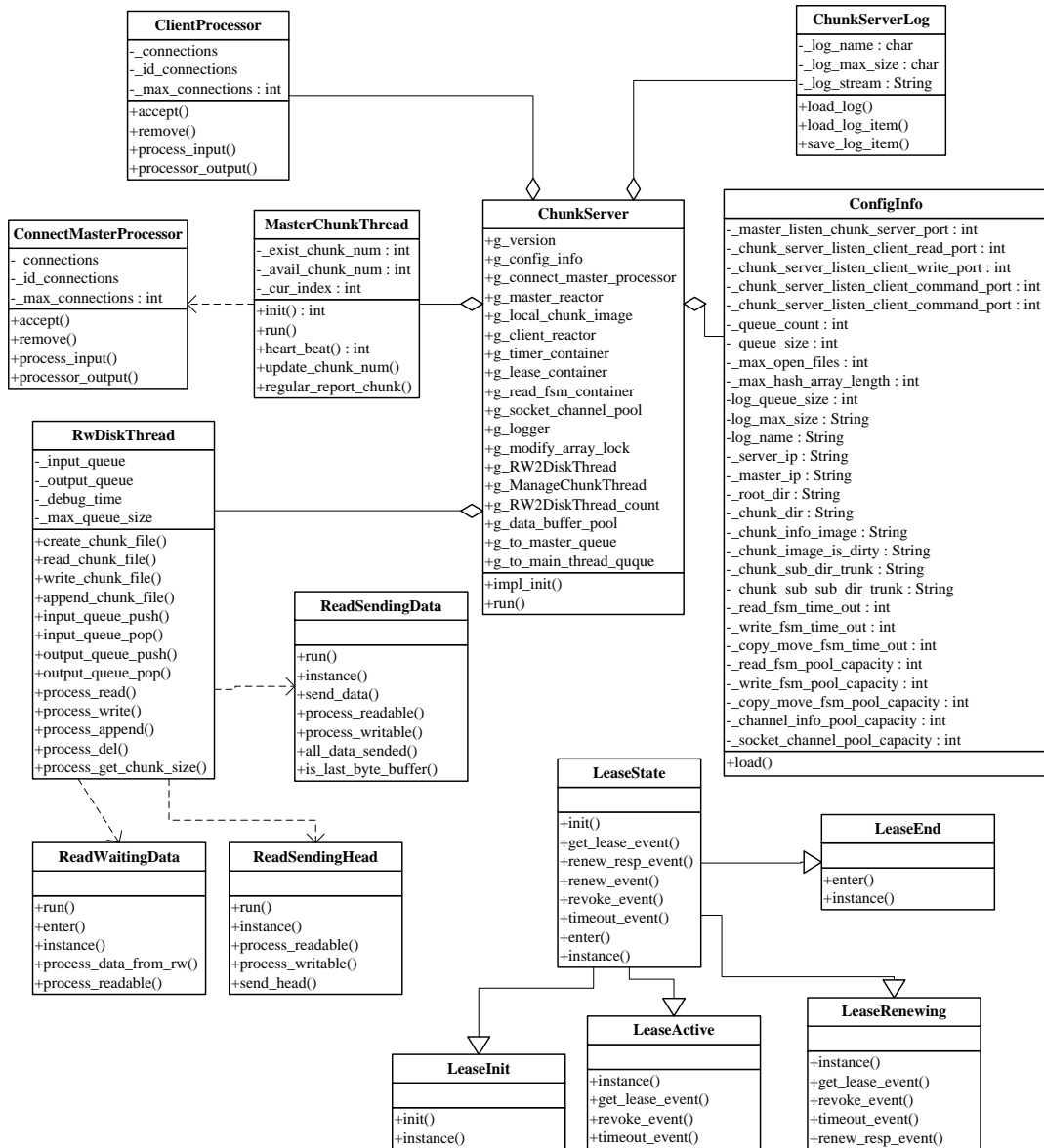


图 3-8 块服务器主要类的类图

3.5 本章小结

本章首先介绍 XLFS 的设计概要，包括整体架构与元数据。其次，对服务器间的交互过程给出详细设计。再次，主服务器和块服务器分别讨论，对主服务器的一些相关操作和块服务器存储方案进行分析与设计，并分别给出设计类图。

第 4 章 分布式文件系统的实现

4.1 主服务器实现

本节介绍主服务器框架和功能模块划分，并实现各个功能。

4.1.1 主服务器功能分析

主服务器主要负责块服务器和数据块复制的管理，并向客户端程序提供具体块服务器信息。具体功能见表 4-1 所示。

表 4-1 主服务器功能表

功能	详细描述
块服务器管理	主服务器通过心跳包与块服务器交互，查看块服务器是否还存活，并获取他们的状态
数据块复制管理	写入操作时，选择合适的块服务器写入。并定期检查块服务器，将数据块从忙服务器转移到闲服务器
数据块租约管理	对每一个数据块，选择一个块服务器保存它的租约。在读写操作时，将该块服务器作为主块
客户端接口	客户端向 XLFS 请求数据时连接主服务器，主服务器告知客户端具体块服务器位置

4.1.2 主服务器各功能实现

如图 4-1 所示，主服务器按功能划分为七个主要模块，每个功能模块由一个类实现。系统配置、日志模块在初始化时被主线程调用，租约管理模块在写操作时被调用，其余几个模块主线程在运行时会循环调用它们，以实现主服务器的功能。

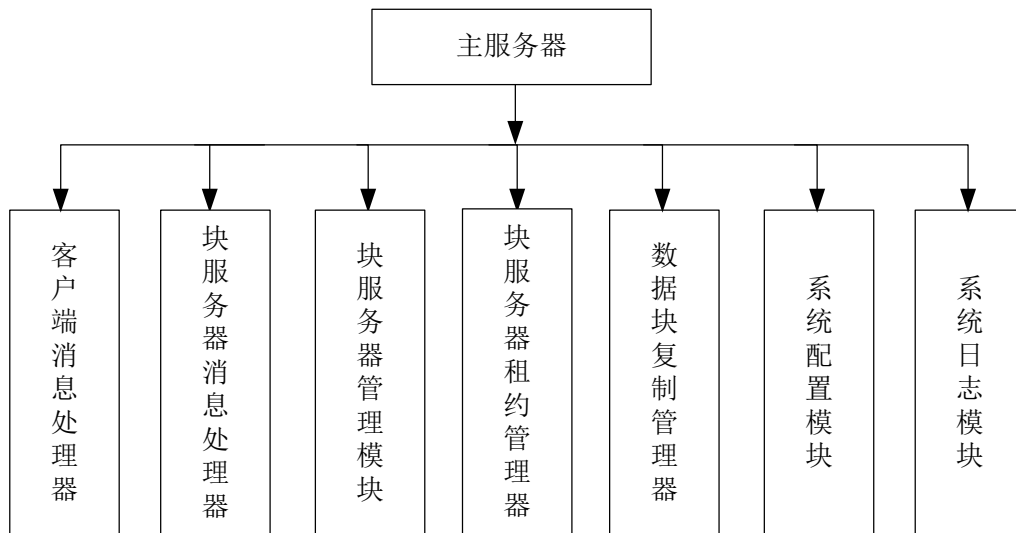


图 4-1 主服务器功能模块划分图

以下是各个功能模块实现简介。

(1)客户端消息处理 由 ClientProcessor 类实现，由该类的 processor_input 函数接收客户端请求包，解析请求类型并根据该类型将请求包转化为不同的协议包，放入请求队列。并待回复队列不空时，由 Manager 类循环调用的 poll 函数取出回复包调用 send_datagram 函数发送给客户端。

(2)块服务器消息处理 由 ChunkServerProcessor 类实现，负责与块服务的通信。与 client_processor 类类似，process_input 函数接收块服务器的消息包，解析类型放入请求队列，send_datagram 将回应包发送给块服务器。

如图 4-2 所示，为两个消息处理器处理消息的流程。实现该流程的关键代码，提炼如下。

```

void ClientProcessor::process_input(ConnectionInfo & conn, ByteBuffer & packet)
{
    uint8_t command = DatagramFactory::get_type(packet);
    LOG4CPLUS_DEBUG(g_logger, "One client connect, command : "<<(int)command);
    switch(command)
    {
        case Datagram:: GET_APPEND_CHUNK_CLIENT_TO_MASTER:
            get_append_chunk (conn,packet);
        default:
            break;
    }
}

void ClientProcessor::get_append_chunk(ConnectionInfo& conn, ByteBuffer& packet)
{
    GetAppendChunkDatagram datagram;
    
```

```

DatagramFactory::decode(packet,datagram);
TaskFsm* fsm = g_server->task_fsm_container()->create_fsm();
fsm->_command = Datagram:: GET_APPEND_CHUNK_CLIENT_TO_MASTER;
fsm->_append_chunk_datagram = datagram;
g_deal_msg_thread->_quest_queue.push_bak(fsm->fsm_no);
if(g_deal_msg_thread->_quest_queue.size()==1)
    g_deal_msg_thread.notify();
}
    
```

这两函数是主线程中的。其中，process_input 函数用于接收请求包，判断包的类型，并调用相应的处理函数。get_append_chunk 函数用于处理 append 请求，他将把请求消息放入请求队列，如果队列之前为空，则唤醒消息处理线程。

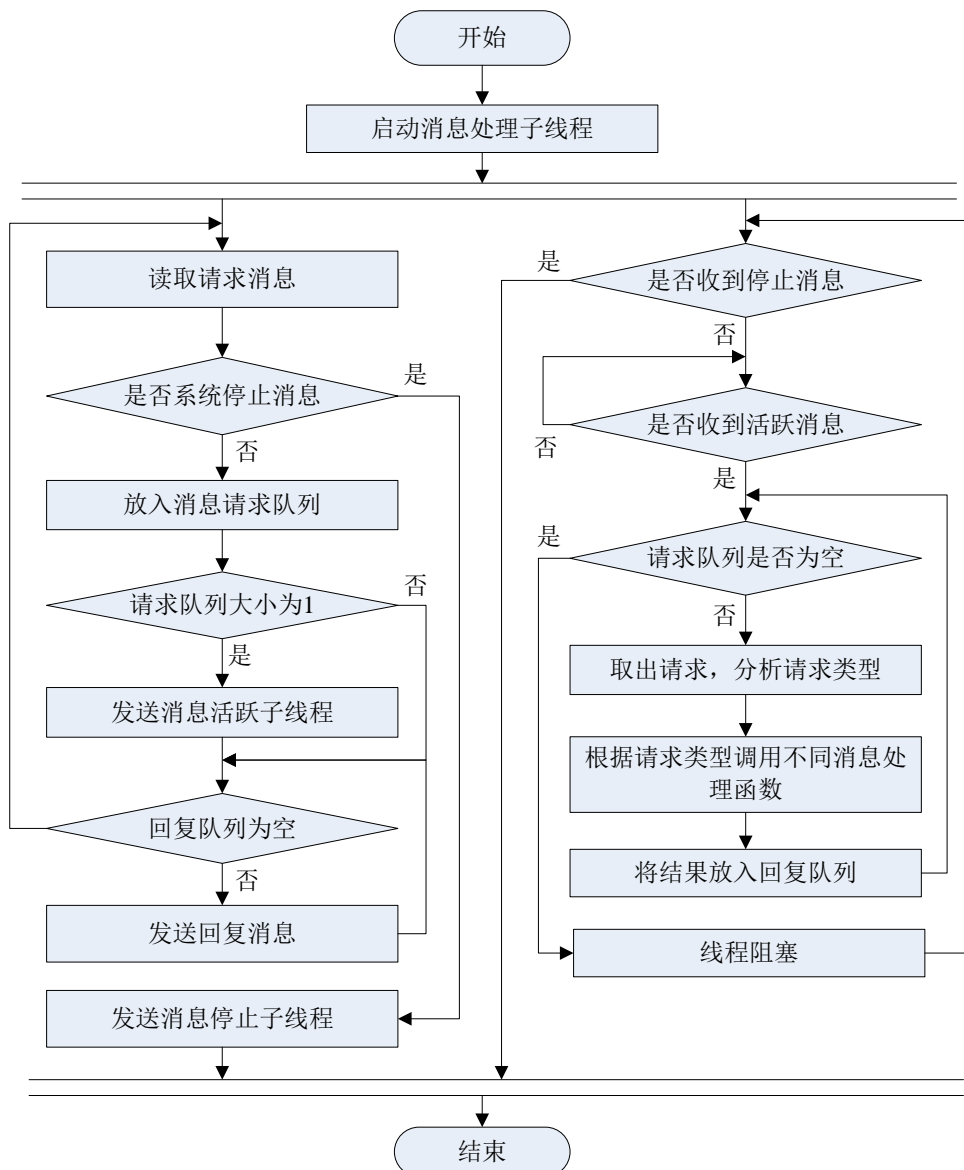


图 4-2 消息处理流程图

```

Void DealMsgThread::run()
{
    int fsm_no;
    _quest_queue.pop(fsm_no);
    TaskFsm* fsm = g_server->task_fsm_container()->find_fsm(fsm_no);
    switch(fsm->_command)
    {
        case Datagram:: GET_APPEND_CHUNK_CLIENT_TO_MASTER:
            get_append_chunk (conn,packet);
        default: break;
    }
    if(_quest_queue.empty())
        block();
}
    
```

DealMsgThread 类的 run 函数就是不断取出请求，判断请求类型、调用相应的处理函数。当请求队列为空时，线程阻塞。

(3)块服务器管理 由 ChunckServerSystem 类实现，主服务器通过心跳包与块服务器交互，查看块服务器获、取他们的状态，并负责新块的加入与坏块的退出。如图 4-3 是主服务器初始化计时器并处理块服务器心跳包的流程图。

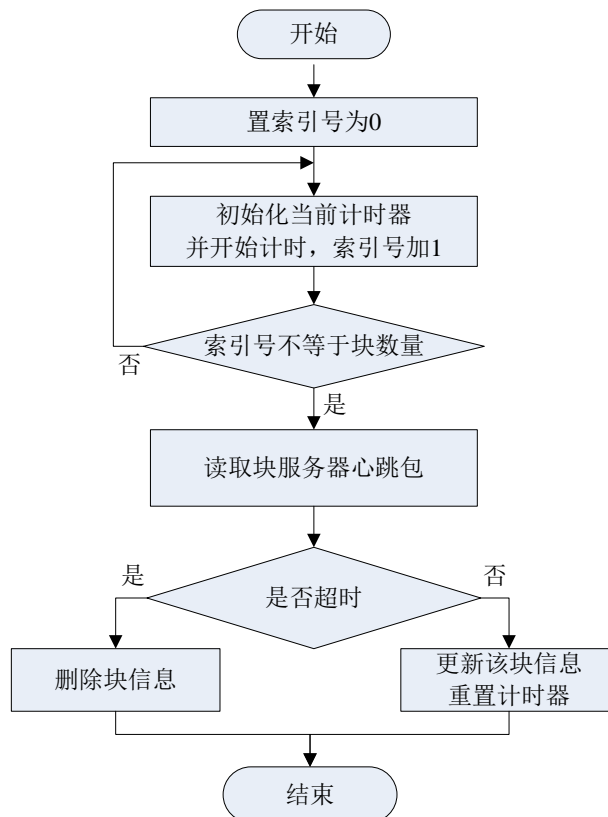


图 4-3 块服务器心跳处理流程图

主服务器初始化计时器在块服务器刚加入时已完成，处理心跳包代码片段如下。

```
Void ChunkServerSystem::check_heartbeat(CheckHeartbeat &d)
{
    ChunkDesc *chunk = find(d.chunk_id);
    if(chunk->time == 0)
        del_chunk(chunk_id);
    else
        chunk.reset_timer(g_config->_chunk_alive_time);
}
```

(4)块服务器租约管理 由 LeaseManage 类实现，对每一个数据块，选择一个块服务器保存它的租约，租约的选择办法有很多种，但实际应用中往往选择保存该块的第一个块服务器。在读写操作时，主服务器对每一个数据块用一个状态机保存，将该块服务器作为主块。关于租约，在介绍块服务器时有详细的说明。

(5)数据块复制管理 由 ReplicaManager 类实现，负责写操作时为数据块选择块服务器，创建租约状态机，先将数据拷贝到各个块服务器，通过主块向其余各块下达写命令，待包括主块在内的各块服务器写入数据完毕，则返回成功消息包，如图 4-4 和图 4-5 所示是主服务器复制数据块和检查数据块时的流程图。另外，该类还用类似的方式负责数据块的转移、孤儿块的回收。主服务器用于检查数据块的代码如下所示。

```
bool ReplicaManager::chech_chunk_data(HeartbeatDatagram &d)
{
    LOG4CPLUS_DEBUG(g_logger, " chech_chunk_data HeartbeatDatagram ");
    ChunkDesc *chunk = g_chunk_server_system->find(d.chunk_id)
    for(size_t index=0;i!=d.chunk_num;++index)
    {
        ChunkData *data = chunk->get_data(index);
        if(data->verify_code==::get_verify_code(*data))
        {
            NameSystem *ns = g_name_system->find(index)
            if(NULL == ns)
                date->del_data(index);
            else
                data->update_data(*data);
        }
        else
        {
            for(size_t replica_index = 0; replica_index!=chunk_num;++ replica_index)
```

```

{
    if(gcid(*data)==gcid(chunk->get_data(replica_index)))
    {
        copy_chunk_data(index, replica_index);
        break;
    }
}
If(replica_index == chunk_num)
    date->del_data(index);
}
}
}

```

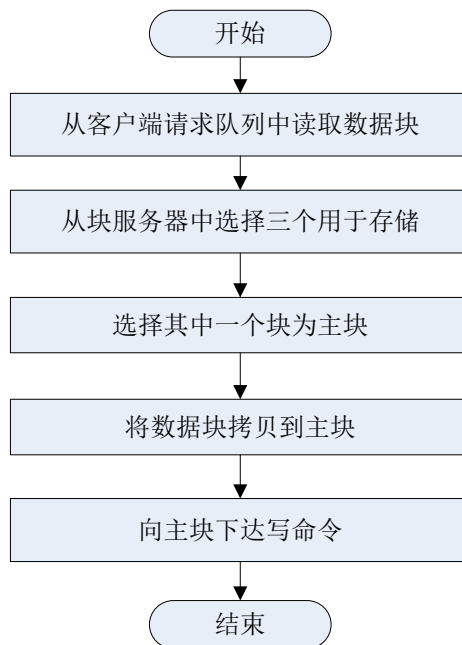


图 4-4 主服务器复制数据块流程图

(6)系统配置 由 MasterConfig 类实现，在 Master 类的 impl_init 函数中调用该类的 load 函数，读主服务器配置文件。之所以将参数写在配置文件里，而不写在代码里，是便于项目的运营。

(7)系统日志 由 MasterLogger 类实现，负责写系统日志，便于出错时查询。

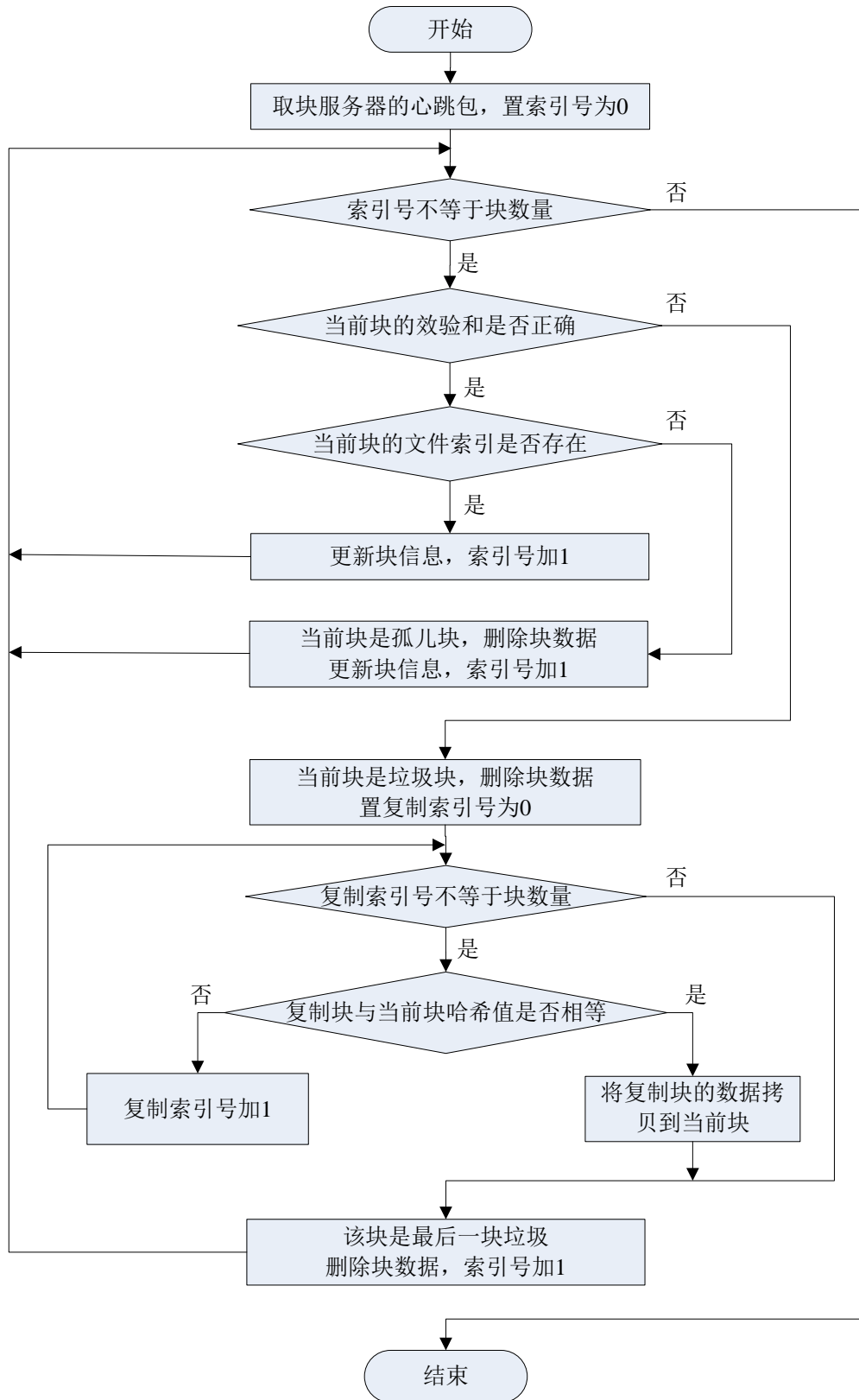


图 4-5 主服务器检查数据块流程图

4.2 块服务器实现

4.2.1 块服务器功能分析

块服务器的主要功能就是本地文件的存取，采用租约方式保证写数据正确。适时向主服务器上报数据块信息，并向客户端传输数据。具体功能见表 4-2 所示。

表 4-2 块服务器功能表

功能	详细描述
本地文件存取	块服务器接受客户端传来的数据块，写入本地文件系统，并在命名空间中写入操作信息
数据块信息保存	在内存中保存服务器上的数据块标识、大小等信息，用于数据块定位和与服务器的交互
数据块租约	为每一个数据块建一个租约状态机。通过 LeaseInit, LeaseActive, LeaseRenewing, LeaseEnd 四个状态确保写入数据正确
主服务器交互	通过心跳包与主服务器交互，向主服务器上报存储空间、块数据块元数据等信息
客户端交互	客户端与块服务器连接，读取或写入数据

块服务器和主服务器、客户端的通信的功能，读取配置文件的功能等，与主服务器的实现类似，数据块信息亦作为块的元数据保存。与主服务器实现不同的是块服务器作为数据的实际载体，它和块服务器之间，以及和主服务器、客户端之间怎样配合，将数据同步写入到磁盘的，以及怎样读出数据给客户端的。

4.2.2 块服务器基本操作实现

本节借绍块服务器 write、read、append 操作的具体实现流程，如图 4-6 所示是写操作的流程图。

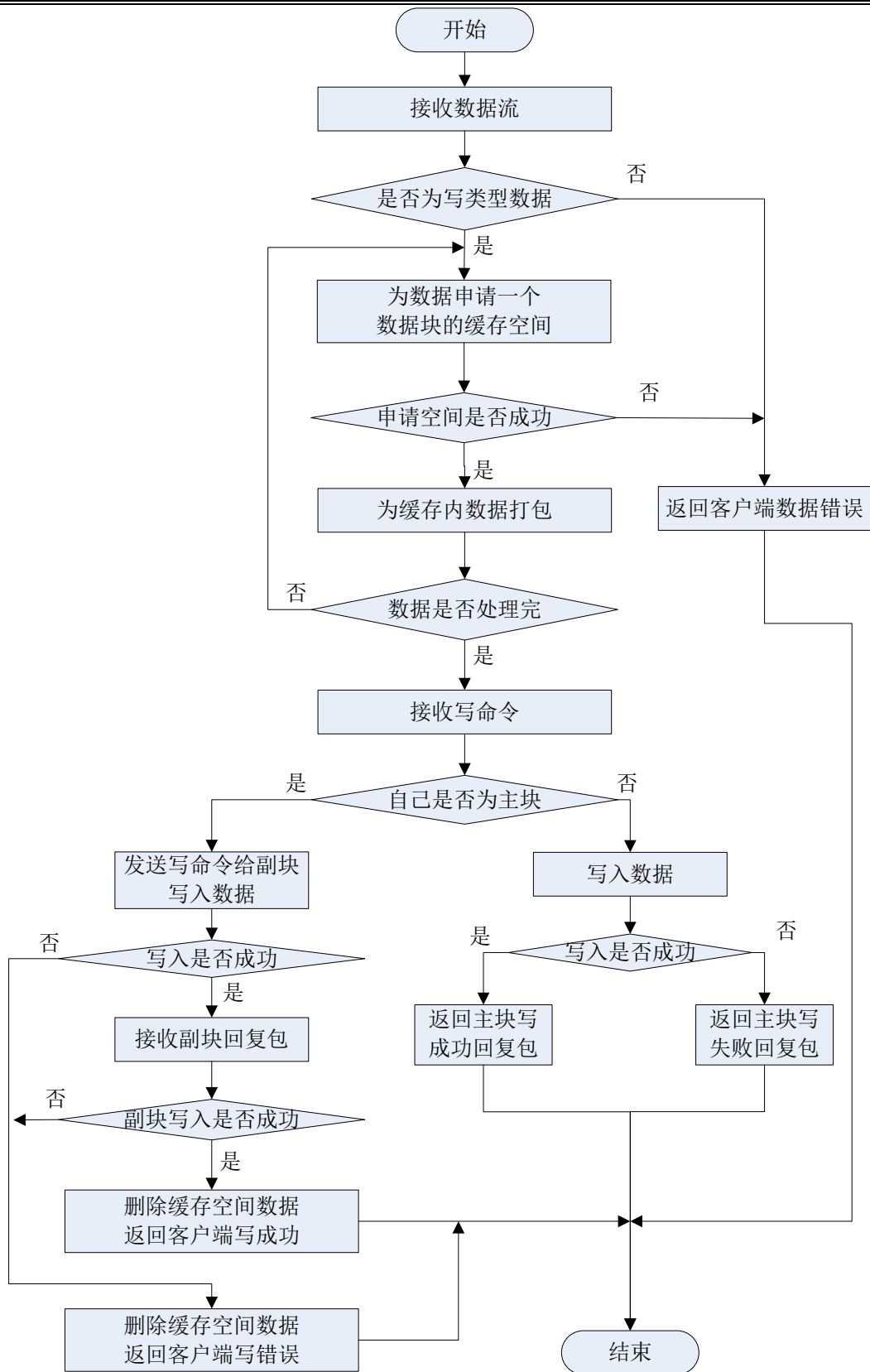


图 4-6 块服务器 write 操作流程

以上是写操作的程序框架流程图，它和读操作都是用状态机模式编写的程序，在不同的状态中每收到一个消息便触发一个事件，在处理该事件函数中处理具体事务逻辑，其代码框架如下。

```
class WriteFsmStateInit : public WriteFsmStateBase
{
    virtual void process_socket_event(WriteFsm& fsm, SelectionKey& key);
};
class WriteFsmStateWaitCommandRespAndLocalResp : public WriteFsmStateBase
{
    virtual void process_socket_event(WriteFsm& fsm, SelectionKey& key);
    virtual void process_write_disk_resp_event(WriteFsm& fsm, RwItem& rw_item);
};
class WriteFsmStateEnd : public WriteFsmStateBase
{
    virtual void enter(WriteFsm & fsm);
};
```

读操作的流程相对简单，块服务器上读操作流程如图 4-7 所示。追加操作的流程与写操作流程类似，但是写入数据时客户端不需要给出写入位置参数。因为是原子性操作，出错时返回的错误码不一样，客户端收到错误回复包会重新执行一遍操作。

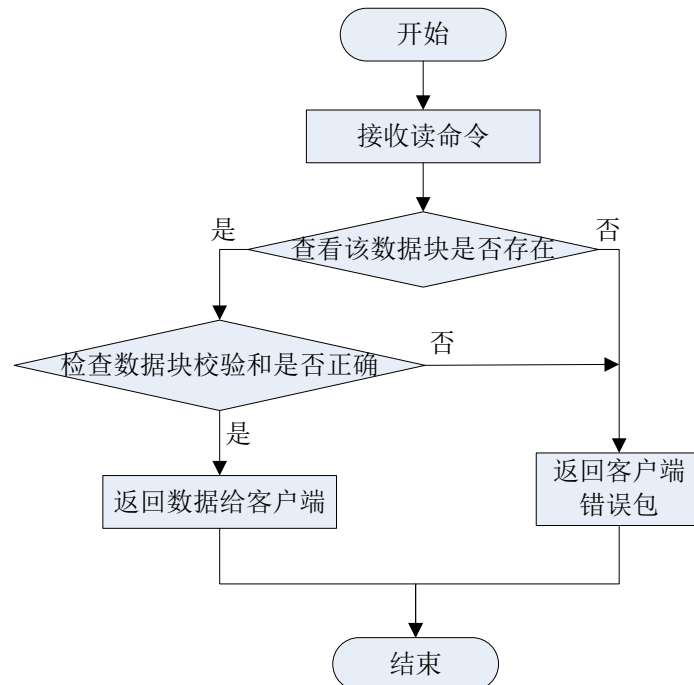


图 4-7 块服务器 read 操作流程

4.3 客户端接口实现

XLFS 的接口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^[42]。

4.3.1 文件系统接口

如表 4-3 所示是 FileSystem 的接口更表，其各个接口定义如下。

表 4-3 FileSystem 接口列表

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)		

(1)static FileSystem* get() FileSystem 是个单件，自身的创建通过调用静态方法 get()，无参数的 get 会启用默认配置，还可以将配置文件路径作为参数(配置文件里可设置 IP 与监听端口)，多线程用 multithread_get()。

(2)File* open(const char* pathname, int32_t flags, int32_t replica_num) 通过 open 方法，可以打开一个文件，如果 flags 置了 O_CREAT 标记，会转而调用 creat 方法。open 的时候可以设置只读(O_RDONLY)、只写(O_WRONLY)可读可写(O_RDWR)、O_CREAT、O_TRUNC、O_EXCL 标记。

(3)如果打开成功，返回指向 File 对象的指针，否则返回空指针。可以通过判断指针是否为空判别 open 是否成功。常用的标志位组合：

➤ O_CREAT | O_TRUNC | O_WRONLY 如果文件存在，文件长度截断为 0；不存在则创建文件，文件只写不能读。

➤ O_CREAT | O_EXCL 如果文件存在，返回错误(NULL 指针)；不存在则

创建文件。

➤ `O_RDONLY` 打开文件只读，文件不存在则返回错误(空指针)。

➤ `O_CREAT | O_WRONLY` 如果文件存在，则不创建，也不截断文件；不存在则创建，打开文件只写。

参数 `replica_num` 指定了文件的副本数，默认值为 3。

(1)`int32_t close(File* file)` 关闭一个打开的文件，参数为指向文件对象的指针，返回 0 代表 `close` 成功。

(2)`int32_t unlink(const char* pathname)` 删除一个文件，以要删除的文件名作为参数。返回 0 代表删除成功；-1 代表删除失败。

(3)`int32_t rename(const char* old_path, const char* new_path)` 重命名文件，第一个参数是要改变的文件名字，第二个参数是新的文件名。返回 0 代表重命名成功；-1 代表失败。

(4)`int32_t copy_from_local_file(const char* src, const char* dstf, bool delete_local = false)` 将本地文件拷贝至 GFS，`src` 是本地文件名，`dstf` 是 gfs 的文件名。通过设置 `delete_local` 参数可以实现是否删除本地文件(默认不删除)。返回值 0 代表拷贝成功；-1 代表拷贝失败。

(5)`int32_t copy_to_local_file(const char* srcf, const char* dst, bool delete_source = false)` 将 GFS 的文件拷贝至本地文件，`srcf` 是要拷贝的 GFS 文件名，`dst` 是本地文件。通过 `delete_source` 参数指定是否删除远程文件。返回值 0 代表拷贝成功；-1 代表拷贝失败。

(6)`Directory* opendir(const char* pathname)` 打开一个文件夹，打开成功返回一个指向 `Directory` 的指针，打开失败返回 `NULL`。

(7)`int32_t closedir(Directory* direc)` 关闭打开的文件夹。返回 0 代表关闭成功，-1 代表失败。

4.3.2 文件接口

如表 4-4 所示是文件接口列表，由于 File 的构造函数和析构函数都是私有的，无法通过直接 new 分配一个 File 对象或者直接 delete 析构一个 File 对象，只能通过 FileSystem 的相关调用实现(open, creat, close 等)。

表 4-4 文件接口列表

File
ssize_tread(void* buf, size_t count)
ssize_twrite(const void* buf, size_t count)
int64_t append(const void* buf, size_t count)
ssize_t writev(const struct iovec* vector, int32_t count)
int64_t lseek(int64_t offset, int32_t whence)

(1)ssize_t read(void* buf, size_t count) read 方法将 GFS 的文件数据读入 buf 所指的缓冲中，count 指出要读的字节数，返回值是实际读取的字节数。如果返回值为-1，表明读取失败。大于等于 0 代表读取的字节数。

(2)ssize_t write(const void* buf, size_t count) write 方法将 buf 指向缓冲区的 count 字节写入 XLFS 的文件，返回值代表实际写入的字节数，-1 代表写入失败。

(3)int64_t append(const void* buf, size_t count) append 方法将 buf 指向缓冲区的 count 字节当成一个记录写到文件尾端，返回这个记录在文件中的起始偏移位置。应用程序可以通过记录这个起始偏移位置和记录本身的长度，来控制对记录的读去操作。返回值小于 0 代表失败。

(4)ssize_t writev(const struct iovec* vector, int32_t count) writev 方法将 vector 指向的若干个缓冲区写入文件，count 是缓冲区数目，具体可 man writev 查看 struct iovec 定义等细节。返回值代表写入的字节数，小于 0 代表失败。

(5)int64_t lseek(int64_t offset, int32_t whence) whence 可以是 SEEK_CUR、SEEK_END、SEEK_SET 分别代表从当前位置、文件末尾、文件开头进行 lseek。返回值为 lseek 之后的文件当前偏移位置，小于 0 代表 lseek 出错。

4.3.3 多线程安全接口

如表 4-5 所示是多线程环境下得到文件系统的函数列表。在名字空间 multithread 下的 File, FileSystem 是线程安全的。

表 4-5 多线程函数表

<pre>#include <gfs_client/multithread_file_system.h> #include <gfs_client/multithread_file.h> multithread::FileSystem* fs = multithread::FileSystem::get() multithread::File* fd = fs->creat("/test/1") fs->close(fd)</pre>
--

multithread::FileSystem::get() 与 FileSystem::multithread_get() 的区别为：multithread::FileSystem::get() 和 FileSystem::multithread_get() 同样会得到一个独立的 FileSystem, 相当于 new FileSystem, 但 FileSystem::multithread_get() 得到的 FileSystem open 的文件不能在多线程中共享, 一个线程 open 的文件不允许另外一个线程读写, 否则线程安全性丢失。而 multithread::FileSystem::get() 得到的 multithread::FileSystem 打开的文件, 可以在多个线程中读写, 在对该文件对象加锁的情况下, 具备线程安全性。

4.4 本章小结

本章首先根据第三章文件系统的设计内容, 进行了系统实现, 包括主服务器各个功能的实现方案, 和块服务器各个基本操作的实现。此外, 为了满足上层应用程序的服务请求, 实现了可供调用的客户端接口, 可以不断地完善。

第 5 章 分布式文件系统的测试及评价

本章着重论述产品开发结束后测试人员所进行的功能测试和性能测试。功能测试和性能测试是产品质量保障的最后一道屏障。只有通过了严格的测试，产品才有可能经受住苛刻的用户运行环境的考验。

5.1 软件测试理论

5.1.1 常见测试流程

在软件工程中软件测试主要分为几个阶段：单元测试、组装测试、确认测试、系统测试^[43]。

(1)单元测试就是代码开发中，由开发人员进行的自测的部分。

(2)组装测试就是集成测试，主要是各个模块组合时的测试，测试各个模块是否相互影响，以及模块的接口之间数据的正确性的测试。

(3)确认测试，就是针对软件需求规格说明书进行的确认是否满足用户需求的测试。确认测试包括一有效性测试和软件配置审查两个部分。在确认测试中，有两个测试叫 α 测试和 β 测试，经过这两种测试的系统，就是大家常说的某个软件的 β 版本。 α 测试是在开发者内部进行的测试，交给开发机构内部的用户的测试，所以是不常见的。而 β 测试是给用户在实际环境下使用的测试。

(4)系统测试，是经过确认测试的软件的最终测试，全面的组装和确认，经过这关以后，就可以交给用户使用。

5.1.2 测试方法

软件测试的方法有黑盒和白盒两种测试方法^[44]，黑盒是对输入输出的一种测试，不关心内部结构。白盒是针对内部结构、逻辑结构进行的一种测试。

白盒测试也称结构测试或逻辑驱动测试，它是按照程序内部的结构测试程序，通过测试来检测产品内部动作是否按照设计规格说明书的规定正常进行，检验程序中的每条通路是否都能按预定要求正确工作。这一方法是把测试对象看作一个打开的盒子，测试人员依据程序内部逻辑结构相关信息，设计或选择测试用例，对程序所有逻辑路径进行测试，通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致。

黑盒测试也称功能测试，它是通过测试来检测每个功能是否都能正常使用。

在测试中，把程序看作一个不能打开的黑盒子，在完全不考虑程序内部结构和内部特性的情况下，在程序接口进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数据而产生正确的输出信息。黑盒测试着眼于程序外部结构，不考虑内部逻辑结构，主要针对换软件界面和软件功能进行测试。

5.2 功能测试

5.2.1 测试方案

功能测试为了检验系统各个命令能否运行成功，测试环境相对简单，基于 Linux 系统的一台主服务器、一台块服务器、一个客户端应用。可以沿分布式文件系统各个 FsShell 命令进行测试，待测试的命令分为四类，如表 5-1 所示。

表 5-1 待测命令表

类别	待测命令
目录对应的 Shell 命令	FsShell -mv <src> <dst> FsShell -cp <src> <dst> FsShell -rm <path> FsShell -mkdir <path>
Regular 文件对应的 Shell 命令	FsShell -mv <src> <dst> FsShell -cp <src> <dst> FsShell -put <localsrc> <dst> FsShell -putv <localsrc> <dst> FsShell -get [-crc] <src> <localdst> FsShell -cat <src> FsShell -touch <path>
Append 文件对应的 Shell 命令	FsShell -mv <src> <dst> FsShell -cp <src> <dst> FsShell -ap <localsrc> <dst> FsShell -apget [-crc] <src> <localdst> FsShell -apcat <src>
系统资源对应的 Shell 命令	FsShell -df

取 Regular 文件相关的 FsShell -put <localsrc> <dst> 命令举例，该命令调用客户端接口中的文件系统接口 `int32_t copy_from_local_file(const char* src, const char* dstf, bool delete_local = false)`，写一个简单的函数以实现这个功能，主要代码如图 5-1 所示。

当调用 FsShell -put <localsrc> <dst> 命令时，程序通过调用该函数以测试 put

功能的正确性。其他的 FsShell 命令和 put 命令类似，不再一一叙述。

```
int32_t FsShell::copy_from_local(const char* src, const char* dstf)
{
    int32_t pre_errno = xifs_errno;
    xifs_errno = 0;
    if (_file_system_impl->copy_from_local_file(src, dstf) < 0)
    {
        if (0 != xifs_errno)
        {
            char error_info[256];
            snprintf(error_info, 256, "copy from local file <%s> to xifs file <%s> failed, errno is %d", src, dstf, xifs_errno);
            error_info[255] = 0;
            XLFSError::instance()->perror(error_info);
        }
        xifs_errno = pre_errno;
        return -1;
    }
    return 0;
}
```

图 5-1 FsShell -put 命令调用代码

5.2.2 测试结果

如图 5-2 所示，是客户端命令接口支持选项示意图。

```
[root@so49 testdata]# FsShell
Usage: FsShell
[-fs <local | remote>]
[-conf <configuration file>]
[-D <[property=value]>]
[-ls <path>]
[-lsr <path>]
[-du <path>]
[-dus <path>]
[-mv <src> <dst>]
[-cp <src> <dst>]
[-rm <path>]
[-rmr <path>]
[-expunge]
[-put <localsrc> <dst>]
[-putv <localsrc> <dst>]
[-circleput <localsrc> <dst>]
[-ap <localsrc> <dst>]
[-append <localsrc> <dst>]
[-copyfromlocal <localsrc> <dst>]
[-movefromlocal <localsrc> <dst>]
[-get [-crc] <src> <localdst>]
[-apget <src> <dst> [start] [end]]
[-getmerge <src> <localdst> [addnl]]
[-cat <src>]
[-apcat <src>]
[-pa <src>]
[-copytolocal [-crc] <src> <localdst>]
[-movetolocal [-crc] <src> <localdst>]
[-mkdir <path>]
[-setrep [-R] <rep> <path/file>]
[-touch <path>]
[-test [-ezd] <path>]
[-stat [format] <path>]
[-df]
[-help [cmd]]
```

图 5-2 FsShell 支持选项示意图

XLFS 提供了类似 Shell 的工具，用于管理 XLFS 文件系统，并方便测试工作。用户可以执行 FsShell 看到所有的可能支持的选项，目前支持比较完善的有

ls、rm、get、put、cat、mv、append 等。

(1)put 如图 5-3 所示，FsShell -help put 可以看到其功能，就是将本地文件拷贝到 XLFS 文件系统中。例如：

```
[root@gougou client]# FsShell -put test.txt /usr/89898/1.txt
```

图 5-3 put 选项示意图

将当前目录下的 test.txt 拷贝到 XLFS 文件系统中，文件名字为 /usr/89898/1.txt.补充说明：-copyfromlocal 选项功能等价。

(2)get 如图 5-4 所示，和 put 相反，get 是将 GFS 文件系统内的文件拷贝至本地文件。

```
[root@gougou client]# FsShell -get /joy/rmnb/02.rmnb /data1/client/02.rmnb
```

图 5-4 get 选项示意图

这条命令将 GFS 文件系统内的/joy/rmnb/02.rmnb 拷贝到本地文件系统中，并命名为/data1/client/02.rmnb。补充：与-copytolocal 等价。

(3)ls 如图 5-5 所示，显示结果分别表示是否是文件夹(‘d’为文件夹，‘-’为文件)，包含的块数(文件夹包含的块数总为 0)，创建文件的时间，最后一列是文件名(不包含路径)。

```
[root@gougou client]# ./FsShell -ls /
d      0      Dec 13 11:28 2007      joy
d      0      Dec 13 11:35 2007      usr
[root@gougou client]# ./FsShell -ls /joy
-      1      Dec 13 13:20 2007      2.rmnb
-      29     Dec 13 11:40 2007      hdtv.avi
-      2      Dec 13 12:37 2007      1.rmnb
-      1      Dec 13 14:15 2007      3.rmnb
d      0      Dec 13 11:28 2007      rmnb
```

图 5-5 ls 选项示意图

(4)cat 如图 5-6 所示，不会分屏显示，只会显示全部文件内容。

```
[root@gougou client]# FsShell -cat /usr/154040/1.txt
i am root
慎行天下
流行前线

拉拉啊独守空房获得健康防护等级开发活动首付款
```

图 5-6 cat 选项示意图

则会将最后两个参数作为偏移位置，从文件中读取数据。不满足上述三条条件的会读出整个文件。

```
[root@so49 testdata]# Fsshell -apget /1m/1 1m.1
[root@so49 testdata]# ll -h 1m.1
-rwxr-xr-x 1 root root 3.0M 04-14 10:25 1m.1
[root@so49 testdata]# Fsshell -apget /1m/1 1m.2 1048576 2097152
[root@so49 testdata]# ll -h 1m.2
-rwxr-xr-x 1 root root 1.0M 04-14 10:26 1m.2
[root@so49 testdata]# Fsshell -apget /1m/1 1m.3 1048576
[root@so49 testdata]# ll -h 1m.3
-rwxr-xr-x 1 root root 3.0M 04-14 10:26 1m.3
```

图 5-11 apget 选项示意图

(10)df 如图 5-12 所示，查看系统资源状况。

```
[root@so49 testdata]# Fsshell -df
Master time      INodes  Chunks  Leases  checking replicas  Last backup
10:29:53         4       2       1       0                  Apr 14 09:39:58

total 3 chunk servers
IP          Used    Available    Use%    weight  Last heart beat
192.168.7.61 2      244         0%      2.859   10:29:48
192.168.7.62 2      252         0%      1.31    10:29:50
192.168.7.63 2      238         0%      1.343   10:29:52
```

图 5-12 df 选项示意图

(11)stat 如图 5-13 所示，查看文件目录属性。

```
[root@gougou client]# ./Fsshell -stat /t/txt
file name: /t/txt      Regular file
file length: 67108864
[root@gougou client]# ./Fsshell -stat /t
file name: /t          Directory
file length: 0
```

图 5-13 stat 选项示意图

(12)touch 如图 5-14 所示，同 linux touch 命令。

```
[root@so35 gfs]# Fsshell -touch /111
/111 is a directory
-touch excute failed.
-touch: error occurred.
[root@so35 gfs]# Fsshell -touch /111/1
[root@so35 gfs]# Fsshell -touch /111/1
[root@so35 gfs]# Fsshell -put client/test.txt /111/1
[root@so35 gfs]# Fsshell -touch /111/1
/111/1 must be a zero_length file
-touch excute failed.
-touch: error occurred.
```

图 5-14 touch 选项示意图

前几项显示了 master 的状况，Master time(当前 master 时间)，INodes(系统节点数)，Chunks(分配出去的 chunk 总数)，Leases(当前的 lease 数)，Checking replicas(正在做副本数目检查的 chunk 数，有可能引发 copy-move)，Last backup(Master 上次备份时间)。

下面显示了各 chunk_server 状况，Used(当前已用 chunk 数)，Available(可用

的 chunk 数), Use%(已用块的百分比), Weight(在 master 调度中的权值), Last heart beat(上次心跳时间)。

5.3 性能测试

5.3.1 测试方案

分布式文件系统最主要的功能是文件的存储与获取, 本系统并不实现用户文件存储与获取的透明性, 而将上传下载操作暴露给用户。在实际应用中本系统的海量文件传输包含两个方面, 大量的小文件传输以及大文件传输。作者为此次测试搭建如表 5-2 的运行环境, 整个运行环境处于一个局域网环境中。

表 5-2 数据传输测试运行环境机器列表

测试结点	操作系统	CPU	内存	磁盘	网卡
数据中心服务器	RH-Linux9	Pentium4-2.8MHz	2GB	160GB	100Mb/s
传输客户端 1	WindowsXP	Pentium4-1.6MHz	1GB	80GB	100Mb/s
传输客户端 2	WindowsXP	Pentium4-2.4MHz	1GB	80GB	100Mb/s
传输客户端 3	RH-Linux9	Pentium4-2.4MHz	1GB	80GB	100Mb/s
数据节点 1	RH-Linux9	Pentium4-2.4MHz	1GB	80GB	100Mb/s
数据节点 2	RH-Linux9	Pentium4-2.4MHz	1GB	80GB	100Mb/s
数据节点 3	RH-Linux9	Pentium4-2.4MHz	1GB	160GB	100Mb/s
数据节点 4	RH-Linux9	Pentium4-2.8MHz	2GB	160GB	100Mb/s
数据节点 5	RH-Linux9	Pentium4-2.8MHz	2GB	320GB	100Mb/s
数据节点 6	RH-Linux9	Pentium4-2.8MHz	2GB	320GB	100Mb/s

传输性能测试主要包括两部分, 分别是单个文件上传下载测试, 及多个文件(特别是小文件)上传下载测试。

在本测试中, 对单个文件的测试, 使用了八个测试文件, 大小分布于 5MB 到 600MB 之间的范围内。对于每个文件的传输, 使用十次传输并求取其中的平均值以最大程度避免服务器性能和网络状况的变化对单次传输造成的影响。

对于多个文件的测试, 同样也用了八个测试用例。为了与单个文件做对比, 这些文件夹的大小和前面的文件大小完全一致, 大小分布于 5MB 到 600MB 之间的范围内。方法与单个文件测试时一样, 使用十次传输并求取其中的平均值。

5.3.2 测试结果与分析

如表 5-3 至 5-6 单个文件和多个文件，上传下载分别的测试结果。

表 5-3 单个文件上传性能测试

文件名	(MB) 文件大小	(ms) 服务调用	(ms) 连接	(s) 传输
test1.rar	5.18	61.1	29.7	2.15
test2.rar	10.1	53.5	29.8	4.31
test3.rar	19.9	42.5	30.0	8.5
test4.rar	39.9	53.3	29.8	16.41
test5.rar	80.4	51.4	29.8	32.75
test6.rar	160	67.8	29.7	64.31
test7.rar	288	42.5	29.9	115.98
test8.rar	577	59.8	30.1	231.05

表 5-4 单个文件下载性能测试

文件名	(MB) 文件大小	(ms) 服务调用	(ms) 连接	(s) 传输
test1.rar	5.18	42.5	29.8	1.99
test2.rar	10.1	38.8	29.8	3.95
test3.rar	19.9	37.8	29.8	8.11
test4.rar	39.9	38.4	29.9	16.18
test5.rar	80.4	46.4	30.1	33.01
test6.rar	160	41.4	30.0	63.89
test7.rar	288	36.9	30.2	114.77
test8.rar	577	42.2	30.2	232.10

从表 5-3 和表 5-4 中可以看到，随着文件的增大连接时间没有改变仍然维持在 30 毫秒左右，而服务调用时间受具体文件影响虽然保持在十毫秒级，但仍有上下的波动。下载时的服务调用时间小于上传的时候，这是由于下载时需要请求的模块只有保存元数据的主服务器，不需要主服务器与块服务器的额外通信。

表 5-5 多个文件上传性能测试

文件名	(MB) 文件大小	文件数	(ms) 服务调用	(ms) 连接	(s) 传输
test1.rar	5.18	20	60.5	0.60	2.06
test2.rar	10.1	39	52.5	1.19	4.27
test3.rar	19.9	77	43.3	2.39	8.33
test4.rar	39.9	154	55.4	4.62	15.93
test5.rar	80.4	310	50.8	9.31	29.77
test6.rar	160	619	66.1	18.54	48.75
test7.rar	288	1113	57.2	33.40	82.39
test8.rar	577	2226	49.3	67.01	167.86

表 5-6 多个文件下载性能测试

文件名	(MB) 文件大小	文件数	(ms) 服务调用	(ms) 连接	(s) 传输
test1.rar	5.18	20	41.1	0.62	1.93
test2.rar	10.1	39	34.7	1.22	3.87
test3.rar	19.9	77	47.0	2.31	8.03
test4.rar	39.9	154	38.3	4.59	15.56
test5.rar	80.4	310	43.9	9.24	28.50
test6.rar	160	619	45.3	18.49	49.30
test7.rar	288	1113	35.5	33.38	81.98
test8.rar	577	2226	39.6	66.98	168.21

从表 5-5 和表 5-6 中可以看到,随着文件数的增大连接时间也呈线性变化成了性能的瓶颈。而服务调用时间受具体文件信息影响虽然保持在十毫秒级,但仍有上下的波动。下载时的服务调用时间小于上传,原因与单个文件类似。

对比下单文件与多文件传输性能,可以发现传输文件夹的时间一般都要大于传输单个文件的时间,这是由于传输文件夹将花费更多的时间在连接线上。而随着传输文件的个数变大,这一趋势却越来越不明显,这是由于传输文件夹的速度得到了提高。另一方面不管是文件夹传输也好,文件传输也好,上传总要比下载花费更多的时间。其原因在于,上传需要花费更多的时间进行服务请求,主服务器需要与块服务器交互得到上传源地址列表。

5.4 压力测试

压力测试是对系统不断施加压力的测试,是通过确定一个系统的瓶颈或不能接收用户请求的性能点,来获得系统能提供的最大服务级别的测试。其目的

是发现在什么条件下系统的性能变得不可接受，并通过对应用程序施加越来越大的负载，直到发现应用程序性能下降的拐点。

本文压力测试采用性能测试的环境，1 个数据中心主服务器、6 个数据节点块服务器，另外的客户端机器以循环的方式不断向文件系统发送请求。在主服务器和块服务器不同配置的情况下，测试文件系统最多能承载多少客户端同时发起服务请求，服务器配置与测试结果如表 5-7 所示。

表 5-7 压力测试结果列表

主服务器 内存(GB)	主服务器内 存分片(MB)	块服务器内 存(GB)	块服务器内 存分片(MB)	承载客 户端数
1	32	1	32	6w
1	64	2	64	7w
2	32	1	32	9w
2	64	1	64	10w
2	128	2	128	8w
2	64	2	64	12w
4	64	2	64	18w
4	128	2	128	14w
4	64	4	64	21w
4	128	4	128	18w

分析表中的数据可知，在主服务器和块服务器内存一定的条件下，服务器内存分片为 64M 时承载客户端数目最高，这是因为和文件系统的块大小相一致的原因，高于 64M 时因为每 64M 数据要占用 128M 的内存片，内存利用率低。低于 64M 时，每个数据块要申请多于 1 次内存片，量大时容易造成数据读写死锁。

纵观全表，内存提高时能承载客户端数相应上升。但由迅雷会员数目估计该系统同时使用的人数不会超过 20w，所以 4G 的服务器内存能满足我们的需求。

5.5 本章小结

本章对文件系统进行了测试，功能测试中利用 FsShell 对各个功能命令进行了测试，性能测试则通过单个文件与多个文件分别下载、上传的方法测试服务器功能。经过充分的测试，认可该分布式文件系统已达到离线下载项目的应用需求。

结 论

本文以离线下载项目为实际需求，在研究了已有的成熟的分布式文件系统后，设计并实现了迅雷离线下载分布式文件系统 XLFS。XLFS 为 WEB 环境下的文件存储提供了分布式的支持。它提供了对各种文件的上传、下载、添加和删除操作的支持。

单一主服务器的设计方案使得我们的文件系统非常简单，主服务器在内存中保存全局域下块服务器的信息，并且可以在后台，简单而高效的周期扫描自己的整个状态。这种周期性的扫描用来在块服务器间实现块的垃圾收集的功能，用来实现块服务器失效时的新副本复制功能，用来实现负载均衡的块移动的功能，以及用来实现统计硬盘使用情况的功能等。主服务器是整个分布式文件系统的核心，不但在内存中，也用操作日志的方式记录下各个文件和数据块、数据块和位置等信息，以便系统的操作与维护。

块服务器是数据存储的具体载体，它以原子性的记录追加操作代替了客户机复杂和代价昂贵的同步机制，并用校验码的方式保证的数据的完整性。和主服务器维持一个心跳包，周期性地向主服务器汇报自己的数据块信息。当然，它最主要最基本的功能还是处理客户端对数据的上传、下载、添加等操作。

客户端接口为上层应用程序提供了对文件系统的操作，FsShell 接口更是直观地展示了本分布式文件系统的功能，这两类接口仍可不断扩充完善。

分布式文件系统是一个复杂的工程，尽管 XLFS 可以满足现阶段的需求，但在设计上仍有进一步提高的余地。在本文的研究基础上，将来还可以在以下几个方面展开研究：

- (1) 绕过租约变更，采用更完善的技术，提高分布式文件系统的一致性。
- (2) 采用相关机制以提高文件的读写速度，例如缓存等。
- (3) 增加分布式文件系统的透明性，如操作透明性、存储透明性等。
- (4) 提升分布式文件系统与硬件无关，与操作系统无关性能。

参考文献

- 1 AS Tanenbaum, M Stee.分布式系统原理与范型.杨剑峰,常晓波,李敏.清华大学出版社.2008:13-24
- 2 Wang WY, Chen YW. Is Playing-as-downloading Feasible in an EMule P2P File Sharing System. Journal of Zhejiang University-Science C-Computers & Electronics.2010,11(6):465-475
- 3 AS Tanenbaum.分布式操作系统.陆丽娜,伍卫国,刘隆国.电子工业出版社.2008:279-282
- 4 NIIT.存储区域网概念与应用.周兆确 叶青.人民邮电出版社.2002:76-78
- 5 冯军.机群文件系统性能优化中的关键问题研究.中国科学院计算技术研究所硕士学位论文.2001:1-6
- 6 S Shepler, B Callaghan.RFC 3530:Network File System(NFS) Version 4 Protocol. The Internet Society, 2003:8-9
- 7 RC Jammalamadaka, R Gamboni, S Mehrotra, N Venkatasubramanian, KE Seamons. A Gmail Based Cryptographic Network file System. Barker S, Ahn GJ. 21st Annual Conference on Data and Applications Security Redondo, CA, 2007. Univ Calif Irvine.Data and Applications Security XXI, Proceedings. 2007: 30-34
- 8 T Igarashi, K Hayakawa, T Nishimura, T Ozawa, H Takizuka. Home Network File System for Home Network Based on IEEE-1394 Technology. IEEE Transactions on Consumer Electronics.1999,45(3):1000-1003
- 9 H Reuter. MR-AFS: A Global Hierarchical File-system. Fusion Engineering and Design. 2000,48(1-2):199-204
- 10 EB Nightingale, Chen PM, J Flinn. Speculative Execution in a Distributed File System. ACM Transactions on Computer System. 2006,24(4): 361-392
- 11 M Spasojevic, M Satyanarayanan. An Empirical Study of a Wide-area Distributed File System. ACM Transactions on Computer System. 1996,14(2): 200-202
- 12 F Garcia-Carballeira, A Calderon, J Carretero. The Design of The Expand Parallel File System. International of High Performance Computing Applications. 2003,17(1): 21-37
- 13 SK Hung, Y Hsu. Reliable Parallel File System With Parity Cache Table Support. IEICE Transactions on Information and Systems.2007,E90D(1):22-29
- 14 M Satyanarayanan, JJ Kistler, P Kumar, ME Okasaki, EH Siegel, DC Steere. Coda:A Highly Available File System for a Distributed Workstation

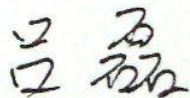
- Environment. Computers, IEEE Transactions.1990,39(4):447-459
- 15 H Inamura. Extending the Coda File System to Handle Cache Misses on Isolated Clients. IEEE. 17th IEEE Symposium on Reliable Distributed Systems (SRDS98), W LAFAYETTE, 1998. USA, IEEE Computer SOC, 1998:336-340
 - 16 B Peter, N Philip. Removing Bottlenecks in Distributed File System:Coda InterMezzo as Examples, <http://www.inter-mezzo.org>
 - 17 Wang RY, TE Anderson. xFS: A Wide Area Mass Storage File System. Workstation Operating Systems. 1993:71-78
 - 18 A Sweeney, D Doucette, Hu W, C Anderson, M Nishimoto, G Peck. Scalability in the XFS File System. USENIX ASSOC. USENIX 1996 Annual Technical Conference SAN DIEGO, CA, 1996. Canada, Proceedings of the USENIX 1996 Annual Technical Conference.1996:1-14
 - 19 Y Randolph, E Thomas, Anderson. xFS: A Wide Area Mass Storage File System In proceedings of the Fourth Workshop on Workstation Operation Systems. 2008:71-78
 - 20 焦利宝, 赵光星.浅谈 XFS 文件系统.电子商务.2009(1):1-3
 - 21 郭威.分布式文件系统 ZD-DFS 的设计与实现.浙江大学计算机学院.2006:8-9
 - 22 A Lakehal, I Parissis. Structural Coverage Criteria for LUSTRE/SCADE Programs. Software Testing Verification & Rellability.2009,19(2):133-154
 - 23 P Dickens, J Logan. Towards a High Performance Implementation of MPI-IO on the Lustre File System. Meersman R, Tari Z. On the Move Confederated International Conference and Workshops Monterrey, MEXICO,2008. USA, SPRINGER-VERLAG BERLIN, 2008:870-885
 - 24 F Schmuck, R Baskin.GPFS: A Shared-Disk File System for Large Computing Clusters. USENIX. Conference on File and Storage Technologies (FAST 02) MONTEREY, CA, 2002. USA, USENIX ASSOC, 2002:231-244
 - 25 杨昕.GPFS 文件系统原理和模式 IO 优化方法.气象科技.2006(S1):1-3
 - 26 JP Prost, R Treumann, R Hedges, AKoniges, A White. Towards a High-performance Implementation of MPI-IO on Top of GPFS. Bode A, Ludwig T, Karl W. 6th International Euro-Par 2000 Conference MUNICH, GERMANY, 2000. USA, SPRINGER-VERLAG BERLIN, 2000:1253-1262
 - 27 张艳.信息系统灾难备份和恢复技术的研究及实现.四川大学应用数学系.2006:3-7
 - 28 李国杰.大规模机群文件系统的关键技术研究.中国科学院研究生院计算技术研究所.2006:5-13
 - 29 L Rand, Jin H, Wang ZP. Architecture Design of Global Distributed Storage System for Data Grid. High Technology Letters. 2003,9(4):1-4

- 30 王为, 金海, 吴松, 熊慕舟. GDSS 系统中文件缓存副本策略及其性能研究. 华中科技大学学报.2005,33(S1):40-44
- 31 Liang S, Yu WK, DK Panda. High Performance Block I/O for Global File System (GFS) with InfiniBand RDMA. Feng WC. 35th International Conference on Parallel Processing Columbus, OH, 2006. USA, IEEE COMPUTER SOC, 2006:391-398
- 32 S Ghemawat, H Gobioff, ST Leung. The Google File System. Proceedings of the 19th ACM Symposium on Operating Systems Principles.2003:20-43
- 33 李柱. 分布式文件系统小文件性能优化技术研究. 国防科学技术大学计算机系.2008:3-7
- 34 WR Stevens, SA Rago. UNIX 环境高级编程(第 2 版). 尤晋元, 张亚英, 戚正伟. 人民邮电出版社.2006:37-38
- 35 Dai MB, T Matsui, Y Ishikawa. A Light Lock Management Mechanism for Optimizing Real-Time and Non-Real-Time Performance in Embedded Linux. Xu CZ, Guo M. 5th International Conference on Embedded and Ubiquitous Computing, Shanghai, 2008. USA, IEEE COMPUTER SOC, 2008:162-168
- 36 Huo YM, Tang K, Hu L. A Reliable Data Management Method for Parallel File System. T Miyazaki, I Paik, Wei D. 7th IEEE International Conference on Computer and Information Technology Aizu-Wakamatsu City, JAPAN,2007. USA, IEEE COMPUTER SOC,2007:328-332
- 37 陈晓宇, 苏中义. 具有副本透明性的分布式文件系统模型的讨论. 华东交通大学学报.2000,17(1):67-73
- 38 A McGregor, J Cleary. A Block-based Network File System. Proceedings of The 21th Australasian Computer Science Conference.1998,20(1):133-144
- 39 Zhang JW, Zhang JL, Zhang JG, Han XM, Xu L. A Storage Slab Allocator for Disk Storage Management in File System. IEEE International Conference on Networking, Architecture, and Storage, Zhang Jia Jie, 2009. USA, IEEE COMPUTER SOC,2009:295-302
- 40 Howard, Kazar. Scale and Performance in a Distributed File System. ACM Transactions on Computer Systems Feb. 1988,6(1):51-81
- 41 杨曙锋. 分布式并行文件系统的副本管理策略. 硕士学位论文, 电子科技大学.2003:6-11
- 42 喻锋荣. LINUX 内核解读入门. 软件工程师.2000(7):75-78
- 43 张向宠, 陈潞平. 软件测试理论与实践教程. 人民邮电出版社.2008,3(10):56-60
- 44 韩明军. 软件性能测试过程. 信息技术与标准化.2007,21(11):41-43

哈尔滨工业大学硕士学位论文原创性声明

本人郑重声明：此处所提交的硕士学位论文《迅雷离线下载分布式文件系统的设计与实现》，是本人在导师指导下，在哈尔滨工业大学攻读硕士学位期间独立进行研究工作所取得的成果。据本人所知，论文中除已注明部分外不包含他人已发表或撰写过的研究成果。对本文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。本声明的法律结果将完全由本人承担。

作者签名：



日期： 2010 年 6 月 29 日

哈尔滨工业大学硕士学位论文使用授权书

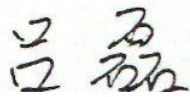
《迅雷离线下载分布式文件系统的设计与实现》系本人在哈尔滨工业大学攻读硕士学位期间在导师指导下完成的硕士学位论文。本论文的研究成果归哈尔滨工业大学所有，本论文的研究内容不得以其它单位的名义发表。本人完全了解哈尔滨工业大学关于保存、使用学位论文的规定，同意学校保留并向有关部门送交论文的复印件和电子版本，允许论文被查阅和借阅，同意学校将论文加入《中国优秀博硕士学位论文全文数据库》和编入《中国知识资源总库》。本人授权哈尔滨工业大学，可以采用影印、缩印或其他复制手段保存论文，可以公布论文的全部或部分内容。

本学位论文属于(请在以下相应方框内打“√”)：

保密☐，在 年解密后适用本授权书

不保密☒

作者签名：



日期： 2010 年 6 月 29 日

导师签名：



日期： 2010 年 6 月 29 日

致 谢

研究生的近二年时光转瞬即逝，亦苦亦乐的学习生活中，很多人给予我莫大的帮助和鼓励。值此论文完成和即将毕业之际，我要对各位老师、同事及同学们的无私的关心和帮助表示衷心的感谢！

首先感谢我的导师马培军老师，在毕业设计过程中，马老师从始至终都十分关心我的论文工作。从选题、研究的思路、开题、设计以及论文审核，马老师都帮我准确地把握课题发展方向和进度、保证课题质量，并提出许多独到的建议。正是在马老师那严谨的治学态度、扎实的科研作风和渊博的学识影响和激励了我，使我能够克服一切困难，毕业课题得以顺利完成。在此，谨向马老师表示最诚挚的谢意和敬意。

感谢实习单位迅雷网络技术有限公司会员部的领导和同事以及副导师贺鹏飞高级工程师，在项目开发和论文的撰写过程中给予了我极大的关心、帮助和支持，他丰富的工程实践经验、对计算机事业不懈的追求以及谦虚热情的为人态度，都是我学习的楷模。还有我的同事兼朋友的叶志辉为我的论文提供了许多珍贵的资料，对我在写论文过程中的疑难问题进行了详细解答。感谢项目组所有的同事，他们对我的论文都提供了很大的帮助，让我有机会去深入地接触网格计算中间件领域的知识。

感谢我的家人，在我这十几年的求学生涯中对我坚定的支持和鼓励，作我坚强的后盾，为我创造了很好的求学条件，使我能安心的完成我的学业。

感谢两年来教育我们的所有任课老师。感谢他们的谆谆教诲，是他们用渊博的知识和高水准的专业技能将我带进一个更高的知识的殿堂，他们敢于开拓的精神和高尚的人格魅力将永远影响着我。

个人简历

2004 年 8 月考入哈尔滨工业大学软件学院软件工程专业，2008 年 7 月毕业获工学学士学位。

2007 年 8 月到 2008 年 5 月在北京航天益来电子科技有限公司，完成电子文档安全管理软件文档转发模块开发。

2008 年 8 月推荐免试攻读哈尔滨工业大学软件学院软件工程硕士至今。

2009 年 7 月到 2010 年 5 月在深圳市迅雷网络技术有限公司实习，实习期间主要从事分布式网络存储的开发与维护，完成了离线下载分布式文件系统 XLFS 的设计与实现。

作者: [吕磊](#)
学位授予单位: [哈尔滨工业大学](#)

参考文献(9条)

1. [冯军](#) [机群文件系统性能优化中的关键问题研究](#)[学位论文]硕士 2001
2. [焦利宝, 赵光星](#) [浅谈XFS文件系统](#)[期刊论文]-[电子商务](#) 2009(01)
3. [杨昕](#) [GPFS文件系统原理和模式IO优化方法](#)[期刊论文]-[气象科技](#) 2006(z1)
4. [张艳](#) [信息系统灾难备份和恢复技术的研究及实现](#)[学位论文]博士 2006
5. [金海, Ran Longbo, Wang Zhiping, Huang Chen, Chen Yong, Zhou Runsong, Jia Yongjie](#) [Architecture Design of Global Distributed Storage System for Data Grid](#)[期刊论文]-[高技术通讯\(英文版\)](#) 2003(04)
6. [王为, 金海, 吴松, 熊慕舟](#) [GDSS系统中文件缓存副本策略及其性能研究](#)[期刊论文]-[华中科技大学学报\(自然科学版\)](#) 2005(z1)
7. [李柱](#) [分布式文件系统小文件性能优化技术研究及实现](#)[学位论文]硕士 2008
8. [陈晓宇, 苏中义](#) [具有副本透明性的分布式文件系统模型的讨论](#)[期刊论文]-[华东交通大学学报](#) 2000(01)
9. [韩明军](#) [软件性能测试过程](#)[期刊论文]-[信息技术与标准化](#) 2007(11)

引用本文格式: [吕磊](#) [迅雷离线下载分布式文件系统的设计与实现](#)[学位论文]硕士 2010