华南理工大学

South China University of Technology

硕士学位论文

基于 FUSE 的云数据访问与存储优化研究

作 者 姓 名	邱双旭
学 科 专 业	计算机科学与技术
指 导 教 师	韩国强 教授
所 在 学 院	计算机科学与工程学院
论文提交日期	2018 年 4 月

Research on the Optimization of Cloud Data Access and Storage Based on FUSE

A Dissertation Submitted for the Degree of Master

Candidate：Qiu Shuangxu

Supervisor：Prof. Han Guoqiang

South China University of Technology

Guangzhou, China

分类号：TP3 **学校代号：10561**

学　号：201520130783

华南理工大学硕士学位论文

基于 FUSE 的云数据访问与存储优化研究

作者姓名：邱双旭　　指导教师姓名、职称：韩国强 教授

申请学位级别：工学硕士　　学科专业名称：计算机科学与技术

研究方向：计算机应用技术

论文提交日期：2018 年 4 月 20 日　　论文答辩日期：　年　月　日

学位授予单位：华南理工大学　　学位授予日期：　年　月　日

答辩委员会成员：

主席：　齐德昱

委员：　李拥军　沃焱　周杰　梅登华

华南理工大学
学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，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由本人承担。

作者签名：邱文旭　　日期：2018 年 6 月 3 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属华南理工大学。学校有权保存并向国家有关部门或机构送交论文的复印件和电子版，允许学位论文被查阅（除在保密期内的保密论文外）；学校可以公布学位论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存、汇编学位论文。本人电子文档的内容和纸质论文的内容相一致。

本学位论文属于：

□保密，在_____年解密后适用本授权书。

☑不保密，同意在校园网上发布，供校内师生和与学校有共享协议的单位浏览；同意将本人学位论文提交中国学术期刊(光盘版)电子杂志社全文出版和编入 CNKI《中国知识资源总库》，传播学位论文的全部或部分内容。

（请在以上相应方框内打“√”）

作者签名：邱文旭　　日期：2018.6.3

指导教师签名：　　日期：2018.6.4

作者联系电话：　　电子邮箱：

联系地址(含邮编)：

摘要

传统互联网模式中，企业所需求的计算和存储资源需要提前进行预估。不同企业业务规模大小不同，对资源需求量也存在较大差别。同时，企业都会储备一定额外的资源，用于高峰期可能发生的业务增长，但这些资源往往无法得到充分的利用，增加了企业的隐形成本。云计算时代，随着云存储技术的出现，资源利用率得到了极大的提升。

本文基于用户空间文件系统 FUSE，使用 HDFS 作为底层存储方式，提出了一种新的云数据访问与存储方案，并对其进行了一系列优化，主要包括：1）针对云存储系统中大规模数据并发访问的问题，设计了一种新的负载均衡算法，该算法根据后端服务器集群中节点的 CPU、内存、磁盘资源占用等指标来计算其负载程度，同时将其实时反馈给调度器，调度器随后对该节点的权值进行动态调整，以实现服务器资源的高效利用。2）针对 FUSE 访问 HDFS 存在性能瓶颈问题，本文通过分析 FUSE 源代码，结合 Linux 文件读写过程，分别从减少 CPU 上下文切换时间、采用直接 I/O 代替缓存 I/O、开启部分元数据缓存三个角度对云存储方案的读写性能进行优化。实验表明，提出的优化方案带来了约 20%的性能提升。3）为了提升云存储方案安全性，本文基于文件属性，实现了细粒度的文件访问控制机制。系列实验结果表明，优化以后的方案，从性能、可用性和安全性方面来看都有更好的表现。

除了对已有方案进行优化外，基于优化后的方案，本文还实现了一个完整的云存储系统，并应用于大数据分析平台中，处理日均数百 TB 级别数据的访问和存储。总体上看，本文所提出的方案，在实际应用过程中，充分解决了业务方在数据存储方面的痛点。同时，也适用于各种不同的应用场景。

关键字：云存储；FUSE；负载均衡；性能优化

Abstract

In the traditional Internet model, companies need to estimate the required computing and storage resources in advance. Due to the different scales of different businesses, the demand for resources is often quite different. At the same time, companies will reserve certain additional resources for the business growth that may occur during the peak period, and these resources are often not fully utilized, which increases the hidden costs of the company. In the era of cloud computing, with the advent of cloud storage technology, resource utilization has been greatly improved.

Based on the user space file system FUSE, using HDFS as the underlying storage mode, this paper proposes a new cloud data access and storage technology scheme, and performs a series of optimizations, which include: 1) For the problem of large-scale data concurrent access in cloud storage system, this paper designs a new load balancing algorithm. The algorithm calculates the degree of load based on the CPU, memory, and disk usage of the nodes in the back end server cluster. At the same time, the algorithm feedback it to the scheduler in real time. The scheduler then dynamically adjusts the weight of the node to achieve efficient utilization of the server resources. 2) Aiming at the performance bottleneck of using FUSE to access HDFS, by analyzing the FUSE source code, combining with the Linux file reading and writing process, this paper optimizes the reading and writing performance of the scheme from three angles, which include reducing the CPU context switching time, using direct I/O instead of cache I/O, and caching some metadata. Experiments show that this article's optimization brings about 20% performance improvement. 3) In order to improve the security of the cloud storage scheme, this paper implements fine-grained file access control mechanism based on file attributes. A series of experimental results show that the optimized scheme does better in terms of performance, availability and security.

In addition to the analysis and optimization of existing solutions, this paper also implements a complete cloud storage system based on the optimized solution, and applies it to a big data analysis platform to deal with the daily data access and storage at hundreds of TB level. In general, the solution proposed in this paper fully resolves the pain points of the

business side in data storage during the actual application process. And it is also suitable for a variety of different application scenarios.

Keywords: Cloud Storage; FUSE; Load Balancing; Performance Optimization;

目录

第一章 绪论

1.1 研究背景和意义

当前，主流的云计算[1]厂商几乎全都有自研的存储技术，并不是直接使用业内所熟知的开源分布式存储系统。之所以会有这样一种局面出现，是因为各个云计算厂商的业务形态不一，技术路线不同，各自所专注的细分领域也不一样。云存储[2]的目的，就是将一系列廉价的服务器集中起来，从而对外提供存储服务或是计算服务。一个完整的云存储系统，往往是由不同的组件构成，主要包括资源调度组件、网络分发组件、数据管理组件、数据存储组件等几个大的方面。相比传统的数据存储方式，云存储所具有的优势很多，这里主要列出如下几个方面：

1）节约成本。使用云存储技术可以大幅降低企业的运营成本，它使得企业不需要去购置服务器或是硬盘等存储设施。企业的数据都存储在虚拟的服务器上，所以，也就不需要聘请专人对服务器等硬件进行维护。此外，云存储使得企业可以根据自己的实际需求购买相应规模的资源，从而达成资源的高利用率。

2）安全性更高。一般来讲，专业的云服务厂商都会提供相当可靠的数据保护服务。企业的数据在未经授权时绝不会轻易的被访问或是修改，而在私有云领域数据的保护更加严密。此外，传统的本地存储，直接将数据存储在硬盘或是其它可移动存储介质上，但是硬件或系统总会有出错的概率，会导致数据的丢失或是损坏。而在使用云存储技术以后，由于其完善的数据备份和恢复机制，将这种风险降到了最低。

3）可扩展性更好。由于企业的业务规模可能是处于一个不断增长的过程之中，因而，当现有的存储资源不能满足需求时，在传统的模式下，重新购置新的设备并对其进行安装部署等往往是一个即耗时又费力的过程。而使用云存储，则可以将这个过程简化。只需要向云服务厂商购买更多的资源，至于资源本身如何进行调配和管理无需用户关心。

4）数据访问更加方便。使用云存储，普通用户只要连接到了互联网，就可以使用不同的设备访问存储在云端的数据。对于企业来讲，这使得它们的员工可以在任何地点工作，通过相应应用程序，同步保存在云端的工作记录，就像在固定办公地点一样。并且，云存储还可以在不同的设备之间同步数据，并自动更新，使得其非常适合跨平台的数据迁移。

具体来讲，本文使用了 Hadoop 的分布式文件系统 HDFS 来作为底层的数据存储方式[3, 4]。Hadoop 自 2007 年推出以后，在大规模数据的计算和存储领域应用非常广泛，成为进行大数据处理必不可少的计算和存储平台。伴随着 Hadoop 所取得的成功，其底层文件系统 HDFS 也同样得到了广泛使用，无论是在工业界还是学术界，都对其进行了大量的研究和改进。当前 HDFS 已经成为了大数据处理的一大重要标签，因而，本文使用 HDFS 来作为云存储技术的实现基础，既有理论意义，也有较大的实际应用价值。

1.2 国内外研究现状

与传统存储方案相比，由于云存储技术所具有的明显优势，国内外都进行了大量的研究。在学术界，近年来主要研究内容有：

- 文献[5]重点研究了云存储服务中如何在保证数据完整性的前提下，尽量降低云存储系统上的数据占用空间。通过对云存储系统中的重复数据进行删除，并使用完善的安全审计操作，使得用户对数据的动态操作不会对数据本身造成任何损失。这种方法兼顾了云环境下数据存储的安全性和性能，降低了云服务提供商的运营成本。
- 文献[6]则针对云环境下大文件的存储问题进行了深入研究。文中提出了一种基于键值存储的大文件云存储系统 BFC（Big File Cloud），通过设计新的存储架构和算法，解决了云存储系统中的大规模数据并发访问问题。具体来讲，通过设计较低复杂度，并且大小固定的元数据来达到目标。
- 文献[7]则针对云存储系统中的数据安全和用户隐私保护问题，提出了一种新的云存储模型，这种模型融合了私有云、公有云等多种服务方式。它将元数据信息存储在私有云中，从而防止了黑客进行未经授权的数据检索或是其它不合法操作。而具体的存储数据保存则是在基于公有云模型的存储集群上完成，这样既保证了系统提供数据存储的服务能力，又保证了数据的安全性。
- 文献[8]则是专注于云存储系统上的数据分配问题，提出了一种高效的数据分配和负载均衡策略。文中提出的策略基于一致性哈希算法的思想，引入虚拟化的概念，使用虚拟节点来改善云存储系统的数据分配能力。此外，文中还加入了能够实时的根据节点可用空间和节点资源利用率来动态调整数据分布的方法，进一步提升了云存储系统的性能。
- 文献[9]则研究了在大规模的云存储集群中，使用不同分布式文件系统来作为后

端的存储技术对于文件访问性能所带来的影响。该文中比较了四种不同的分布式文件系统的性能表现，并结合医疗图像存储的实际应用需求，证明了存储层文件系统的不同对整个云存储系统的性能可能带来的影响。

在工业界，主流的云计算厂商都使用自己的存储技术:

- Amazon 的 AWS（Amazon Web Services）使用的是 Dynamo，这一技术主要用来实现一个增量扩展，并且拥有高可用性的 KV（Key-Value）存储系统[10]。这种技术综合权衡了云存储系统的成本、数据一致性和数据访问性能，与此同时也保证系统的高可用性。
- Microsoft 的 Microsoft Azure 使用 Azure Storage 来作为底层存储，这种存储方案重点是保证存储服务的高可用性和不同 IDC（Internet Data Center）之间数据的一致性[11]。
- Google 的云服务使用了自研开源的 GFS（Google File System），这种分布式文件系统面向的是大规模数据吞吐量的应用程序，拥有可伸缩的特点，主要运行在廉价的硬件设备上[12, 13, 14]。GFS 有很多开源实现，已经成为了云计算领域至关重要的一种基础技术，本文所使用的 HDFS 就是基于这一概念设计而来。
- EMC 公司推出的 Atoms 是一种横向扩展的对象存储平台，它是专门为存储、归档和访问非结构化数据以及在云计算平台上支持大规模的应用程序和服务来构建的[15, 16]。Atoms 的特点在于其在保持大规模的同时，最大限度地提升存储效率。其提供了完善的存储访问方法，如 HTTP、Web 服务、内容寻址存储和基于文件的访问方式[17]。
- 国内最大的云计算厂商阿里云同样有自己的分布式存储技术，即盘古平台[18]。盘古平台在设计之初在高性能、合理成本、高性价比方面做出了一定的妥协，其目的是提供易用、服务化、方便用户接入、运维完善的云存储平台。

云存储技术已经得到了广泛的关注和研究，这些研究对于本课题的研究工作，都有一定的借鉴意义。经过分析，现有的方案往往都是以通用性为目标，针对某些特定应用场景中复杂网络环境下的数据安全性问题和服务稳定性问题，以及大规模的数据并发访问的高效处理问题没有一个很完善的解决方案。本文的研究目的，就是要对现有的技术进行优化研究，研究一种满足大规模数据并发访问需求，同时保障数据安全性，以及服务高可用性的云数据访问与存储方案。

1.3 主要工作内容和创新点

本文的主要工作内容如下：

1）为了提升云数据访问和存储过程的高可用性和负载均衡，本文从多个层面进行了研究与优化。首先是云存储数据访问层面处理来自客户端请求时进行了高可用性和负载均衡的设计，对存储集群处理客户端请求并进行负载均衡的算法进行了研究和应用；其次，研究了高可用的 HDFS 集群，使用多 NameNode 同时运行的机制，实现了主备 NameNode 的切换。

2）为了提升云环境下的数据访问能力，对 HDFS 的挂载组件进行了优化，提升了其读写性能和安全性;为了方便用户对存储集群进行访问和管理,本文使用前沿的 REST（REpresentational Status Transition）软件设计架构，设计了一套访问 API（Application Program Interface）[19]。此外，从本文所提出的方案实际出发，设计和实现了一系列云存储的管理组件。相比开源组件，本文设计的组件与本方案更加契合。

3）基于优化过后的云存储技术方案实现了一个完整的云存储系统，并在大数据分析平台中进行了应用，支撑了日均数百 TB 级别的数据访问量。

主要的创新点有：

- 在处理来自客户端的请求时，本文从 FUSE 访问 HDFS 对 CPU、内存、磁盘等需求较高的实际情况出发，设计了动态调整的负载均衡算法，相比常用的负载均衡算法，经过实验验证，其有更好的性能表现。
- 针对在 FUSE 访问 HDFS 在实际应用过程中所表现出来的文件读写性能问题，本文通过对 Linux 文件系统读写原理和 FUSE 运行机制的分析，对 HDFS 的挂载组件进行优化，显著提升了云存储方案的读写性能。
- 针对云存储安全性的问题，本文不使用传统的 Web Service 身份认证方式，而是从 Linux 文件系统本身出发，设计了基于属性的细粒度访问控制方式[20]。

1.4 论文组织结构

本文共分为六章，各章的主要内容介绍如下：

第一章主要是介绍了本课题的研究背景和研究意义，同时对当前业内的研究现状进行了调研，确定了本文的研究方向，最后还对本文所进行的研究工作相比常规方案的创新之处进行了说明。

第二章详细介绍了在研究过程所涉及到的相关理论基础以及现有技术的不足之处。首先解释了云计算和云存储的概念，确定了云存储技术的实际需求和在进行设计时应该遵循的基本原则。在此基础上对分布式文件系统 HDFS 的工作原理和基本架构进行了介绍，接下来对 FUSE 访问 HDFS 的过程进行了描述，最后分析了通过 FUSE 访问 HDFS 的不足之处并对计划进行的优化进行了说明。

第三、四两章主要是依照第二章末尾提出来的现有技术的不足之处以及本文的优化方案，详细地阐述了本文的优化过程。其中，第三章对来自客户端访问请求调度算法进行了研究。这一章首先从数学角度分析现有的负载均衡算法所存在的缺点，结合其在实际场景中表现出来的不足，确定了自行设计调度算法的思路。在此基础上，对本文所设计的动态负载均衡算法进行了阐述。最后，通过设计对比实验，对本文所提出的负载均衡算法与加权轮询算法在实际场景中的具体表现进行了对比。

第四章则是通过对 HDFS 挂载组件源代码的分析，对其访问 HDFS 集群上文件的过程进行了优化。这一章首先介绍了数据访问过程的优化方案，主要是从提升性能和安全性两个方面来入手。在此基础上，分析了 FUSE 内核模块和用户空间的数据结构设计，分别介绍了其在用户空间和内核空间的运行流程。此后，结合 Linux 中文件系统的读写原理，从减少 CPU 上下文切换、使用直接 I/O 技术、缓存文件的 inode 号和 dentry 信息三个角度对组件的读写性能进行了优化。除此之外，考虑到云存储方案的安全性问题，使用基于文件属性的方式，通过对组件源代码的优化，实现了细粒度的访问控制。本章最后设计了一系列对比实验，分别从大文件和小文件的角度验证了优化后读写性能的提升。

第五章描述了优化后的云存储技术在实际场景中的具体应用。这一章首先介绍了本方案的总体框架，然后基于该框架，详细介绍了数访问层和数据存储层的设计与实现。在此基础上，对云存储方案进行了进一步的优化和完善，设计和实现了本方案特有的监测组件和日志管理组件。

最后是本文的总结与展望部分。首先对本文所做的工作和最后的成果进行了总结，此后提出了本文的研究成果所存在的不足之处，并对课题以后的研究方向进行了思考。

第二章 云存储技术理论分析

本章内容主要是对本文中所使用到的关键技术和涉及的理论进行解释。2.1 小节介绍云计算和云存储的基本概念，并提出了云存储技术的基本需求和设计原则，引入论文研究主题。2.2 小节介绍了分布式文件系统 HDFS。2.3 小节介绍了用户空间文件系统 FUSE，同时对本文所使用的 FUSE 访问 HDFS 的过程进行了分析。2.4 小节提出了现有技术存在的不足之处，以及本文的优化计划。2.5 小节为本章小结。

2.1 云计算和云存储

2.1.1 云计算分类

从总体上来看，云计算可以分为云平台（Cloud Platform）和云服务（Cloud Service）。云平台是指基于硬件的一种服务，向外提供数据的存储和计算等功能。云服务是指以底层的硬件设施为基础，可以方便地进行扩展的服务。云计算可以进行不同的分类，例如按照是否向外提供服务可以分为公有云、混合云和私有云；按照向外提供的服务种类来看，可以分为基础设施即服务层（IaaS，Infrastructure as a Service）[21]、平台即服务层（PaaS，Platform as a Service）[22]、软件即服务层（SaaS，Software as a Service）[23]。图 2-1 对云计算的分类进行了说明。

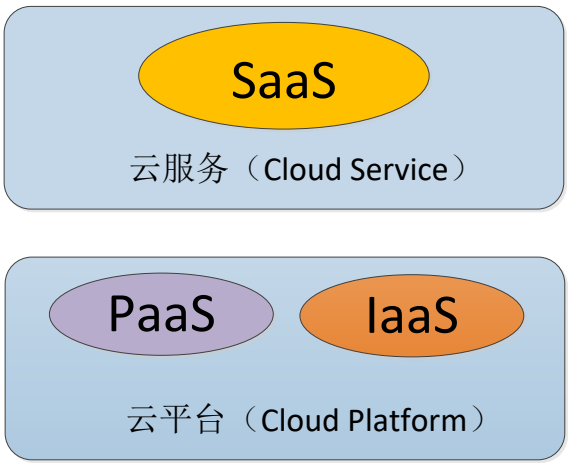


图 2-1 云计算分类

2.1.2 云存储技术需求研究与基本原则

根据上一节对云计算所进行的分类，云存储技术主要是指云计算中的基础架构即服务层所衍生而来的一个基本概念。在如今的大数据时代，每天都会产生海量各种不同类

型的数据。人们每天的衣食住行等各种活动都会产生数据，这些数据随着时间的迁移往往是呈指数级增长的。在数据规模增长如此快速的情况下，传统的本地存储技术自然也就无法满足存储需求。例如本文将要讲到的优化后方案的应用场景，是一个基于对用户行为数据进行分析来创造商业价值的平台，其每天会产生近 400TB 的离线数据。如果试图将每天的数据都进行保存在本地的单机存储设备上，几乎不可能。

云存储技术的源起，正是由于工业界在实际应用方面的需求推动而来。这种技术，一般情况下，是由大量的普通 PC（Personal Computer）服务器组成。它融合了分布式存储、数据安全、高可用性等概念，目的是为用户提供稳定高效的海量存储服务。图 2-2 说明了云存储技术对外提供服务的基本模式。

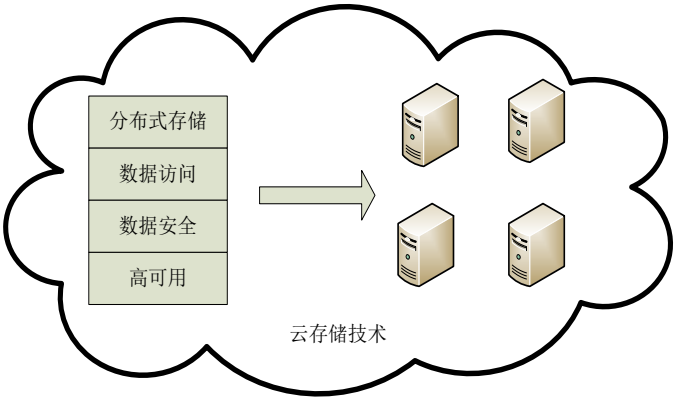


图 2-2 云存储技术

根据对国内外工业界和学术界对于云存储技术的研究与应用成果，结合具体的应用场景，分析总结了对云存储技术的以下几大需求：

1）较低的构建和运行成本。由于云存储技术需要对外提供服务，相比使用具体的服务器来进行部署，在使用云存储技术之后，必须能够显著的降低用户的开发和运维成本。

2）良好的扩展性。由于业务规模和资源需求可能是不断变化的，因而，云存储技术必须保证能够随时方便的对存储和计算能力进行扩展，也就是说，必须能够随时加入新的节点，或者是对集群中的节点进行替换，这将取决于具体的业务需求。

3）高可用性。由于云存储系统上往往同时保存着很多不同的企业应用程序数据，而这些应用程序能否经受住长时间的检验，是评价一个云存储系统的关键指标。同时，所设计的云存储方案必须能够迅速的发现故障和进行自愈处理。完善的监控和运维体系，是保证云存储系统高容错率的一个重要因素。此外，必须考虑到在大规模集群中，硬件故障将常态化这一实际情况，在云存储技术方案的不同层面，都必须有高可用的设

计。

5）安全性。由于云存储系统上保存了很多用户数据，如何保证这些数据不被窃取，也是一个重要的设计考虑。因此，必须引入安全审计机制，对数据访问权限进行限制，保证数据安全性。

根据前文对云存储技术的需求研究结果，在设计本次提出的基于 FUSE 的云存储技术方案时，主要满足以下几个基本原则：

- 底层存储技术。所使用的底层存储技术必须是成熟稳定的，其对运行环境的要求不能太高，这样能够保证对各种廉价资源的充分利用。
- 数据访问方式。访问数据必须要方便直观，通过统一风格的接口来对外提供服务。这样设计的好处在于不同客户端访问时，业务方不需要调整自己的代码。
- 访问控制机制。对云存储系统中的数据访问必须进行审计，通过完善的访问控制来对数据的安全性予以保证。
- 读写性能。在保证不提升服务器负载的前提下，尽量优化存储系统的读写性能，对外提供高效稳定的服务。
- 采用分层级模块化的方式进行设计，以方便方案的具体实施和部署。
- 合理地使用相对较为成熟的方案。由于云存储技术已经得到了较为广泛的研究，所以，为了提高本课题研究成果的实用价值和稳定性，尽量在保证原创性的前提下，适当借鉴这些现有的方案，吸取他们的优点，对不完善或者不合理的地方进行改善，从而使得本次提出的方案在鲁棒性上有很好的保证。

2.2 HDFS 原理及运行机制

HDFS 与其他常见的分布式文件系统有一些共有的属性，同时，也有很多自己独有的特征。HDFS 的容错能力非常高，因而为了降低成本，一般都是被部署在一些成本低廉的硬件设备上。HDFS 所支持的数据吞吐量规模很大，能够满足一些数据密集型的应用程序的要求。在 HDFS 中，为了实现对数据的流式访问，以更加契合实际应用场景中应用程序的需求，它并不完全依照在 POSIX 标准中文件系统设计的一些硬性指标[24, 25]。

2.2.1 HDFS 的设计目标

HDFS 的设计是源于工业界对分布式文件系统在实际应用的需要，因而其设计目标是以实用为原则，并非是单纯理论上的想象。总体来讲，对几个关键的指标列出如下：

1）硬件故障将成为常态。HDFS 集群的规模往往非常大，节点数目一般都会达到上百甚至是上千。很显然，在如此大规模的集群中，单个节点或组件的小概率事件放大后将成为常态，这会导致 HDFS 可能会始终存在无法正常工作的节点或者组件。HDFS 在设计时考虑到了这个问题，并提供了完善的数据恢复和容灾机制。

2）应用程序以流式访问的方式来访问 HDFS 上的数据。对访问 HDFS 的应用程序来讲，最关键的是要达到大规模的数据访问，也就是高吞吐量，至于访问大规模数据时所带来的延时，在设计时做了一定的折中。HDFS 并不保证数据访问的时效性，这不是其关注的重点。因而，在 POSIX 中的一些文件系统标准在这里也不再适用，所以，被大量抛弃，以达到更高的数据吞吐量。

3）迁移计算而非迁移数据。根据局部性原理，如果下一次需要进行计算的数据在当前计算节点所处位置附近，计算效率会高很多。所以，为了遵循这个原则，HDFS 中一般会根据当前计算所需的数据所处位置，将计算工作进行迁移，使其离数据存储位置更近。这样一来，降低了由于网络阻塞或是其它不可控因素带来的影响，使得计算工作得以更快的完成。

4）HDFS 具有高可移植性。HDFS 本身使用的是 Java 来进行开发，由于语言特性，其跨平台的移植能力很强，对于硬件或软件环境本身的依赖并不高。

2.2.2 NameNode 与 DataNode 的交互

典型的 HDFS 集群往往是由一个 NameNode 和多个 DataNode 组成。NameNode 上保存了集群中文件的元数据信息，包括文件名，文件路径，所属块等。HDFS 上的文件以数据块的方式存储在 DataNode 上，NameNode 则管理了文件与组成文件的块到具体 DataNode 的映射。当客户端访问请求到达 NameNode 时，NameNode 会根据当前所保存的元数据信息，将数据访问工作分配给特定的 DataNode 节点，来对外提供服务。由于 NameNode 上保存了 HDFS 集群中文件的元数据信息，所以一般来讲 NameNode 对内存资源的占用会比较高，并且，NameNode 的数据处理能力很可能成为整个集群的性能瓶颈。

DataNode 上保存了集群中的所有文件，真正的文件打开、关闭等工作在这里完成，同时完成数据块的新增、删除和复制工作。鉴于 HDFS 上的所有数据都保存在这里，所以 DataNode 对硬盘资源的需求往往比较大。图 2-3 为 HDFS 中 NameNode 和 DataNode 的交互方式。

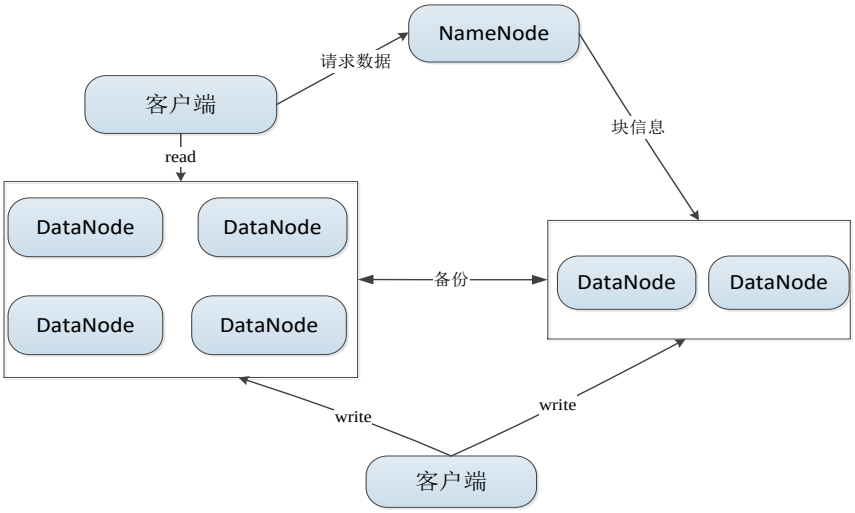


图 2-3 HDFS 中 NameNode 与 DataNode 的交互

2.2.3 DataNode 中的数据复制

HDFS 可以在大型集群中对大规模文件进行存储，并且稳定性很高。NameNode 会将一个文件分成多个相同大小的块来进行存储，将这些块保存在多个 DataNode 上，NameNode 上则会保存一份文件块到指定 NameNode 的映射。相比之下，文件块的复制容错率较高，单个块的大小，以及需要进行多少个备份都是可以根据用户的实际需求来进行修改的，应用程序可以按需进行修改。名称节点即 NameNode 会定期对数据节点即 DataNode 进行心跳检测，从而确保节点是处于正常运行的状态。这个过程中数据节点中上的数据块信息也会被送到名称节点。以下以一个具体实例说明 HDFS 中数据块的复制过程。

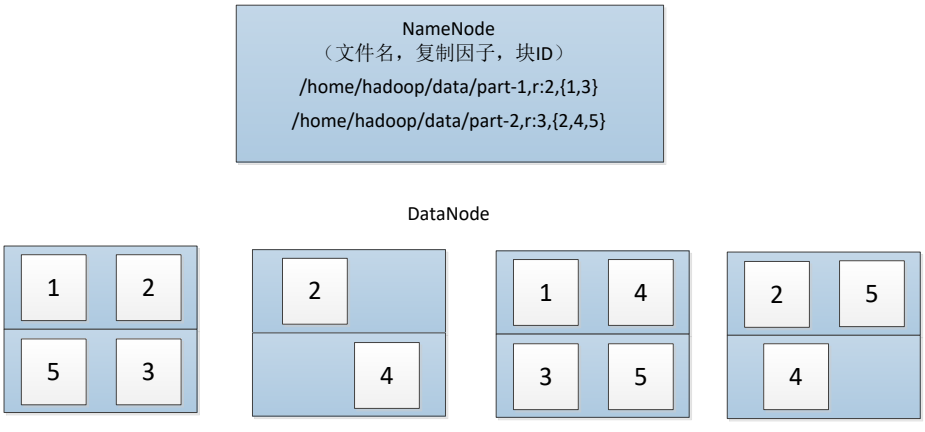


图 2-4 DataNode 的数据复制过程

图 2-4 中，part-1 这个文件的复制因子为 2，所以，在 DataNode 中 1，3 两个数据块分别出现了两次；而 part-2 这个文件的复制因子为 3，因而 2，4，5 这三个数据块分别

在节点中出现了 3 次。

2.3 FUSE 访问 HDFS

2.3.1 FUSE 原理分析

FUSE 是一种用户空间的文件系统，它的功能是通过编写用户空间程序，获取文件信息以后以用户空间文件系统的方式将目录挂载到 Linux 上。FUSE 主要包含了两个组件：内核模块 FUSE kernel，开发者库 libfuse。libfuse 是内核模块与用户空间进行通信的一个接口。在 libfuse 模块中，向开发者提供了相对上层的 API，在具体使用时，只需要对 FUSE 中所定义的各种操作进行功能实现[26]。

图 2-5 为一个简单的基于 FUSE 的文件系统的工作过程。

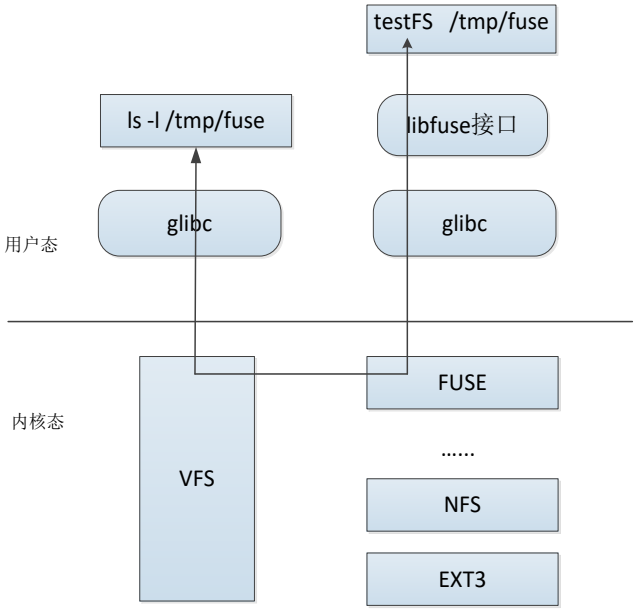


图 2-5 FUSE 工作过程

可以看到，所实现的文件系统名为 testFS，它被挂载在系统目录中的/tmp/fuse 上。当用户在/tmp/fuse 上进行任何操作时，会导致 VFS（Virtual File System）[27]中的接口函数被具体调用，此后，FUSE 内核模块会接收到这个操作信息。通过 FUSE 内核模块中已经定义好的机制，会调用相应的回调函数来响应用户的操作。

FUSE 本身是基于 VFS 的，相比 EXT2、EXT3 这些文件系统，FUSE 的身份是一个转换器，将来自 VFS 的请求，转换成用户级的函数来进行处理。使用 FUSE 有几个优势：

1）FUSE 运行在用户空间，有效地避免了用户和内核直接打交道，从而保证了系统的稳定性。

2）开发文件系统的工作难度被大大降低。开发者不需要去理解文件系统与内核之间交互的流程和原理，可以仅关注文件系统中各操作功能的实现。

本文所研究的云存储技术就是通过 FUSE 来访问 HDFS 的。目前，国内外对 FUSE 的研究比较广泛，也推出了一系列基于 FUSE 的产品。

2.3.2 FUSE 访问 HDFS 过程分析

Hadoop 针对 C 语言开发提供了 libhdfs 这个库，这个库不仅可以访问 HDFS 文件系统，也可以访问整个 Hadoop 上的数据。libhdfs 的本质还是在调用 Hadoop 的 Java 接口，通过 JNI（Java Native Interface）来实现，并且使用上也基本一致。Hadoop 的二进制包中已经带有 32 位的 libhdfs 组件，如果需要使用其它版本，需要自行编译。

Fuse-DFS 是 Hadoop 下的一个组件，使用该组件可以将任何一个 HDFS 实例挂载到本地文件系统中，访问起来就像是通用文件系统一样。从而，就可以直接通过普通的文件系统命令，如 ls、cat 等命令来操作该虚拟文件系统，当然也可以使用不同的编程语言通过调用相应的文件系统接口来进行访问。Fuse-DFS 通过调用 libhdfs 接口来连接并访问 HDFS 中的数据，相比于使用 Hadoop 的其它接口，如使用 HTTP 来进行访问的方式，这种方式更加适合在实际的工程中来使用，特别是使用 C 语言来编写的程序需要访问 HDFS 时，如本平台中运行在 Linux 上的服务端程序。一次客户端通过 FUSE 访问 HDFS 上文件的过程如图 2-6 所示：

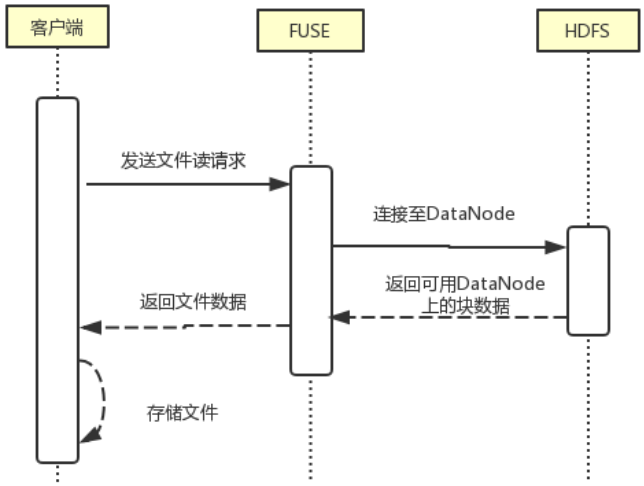


图 2-6 FUSE 写入文件到 HDFS

Fuse-DFS 的主程序首先通过 libhdfs 的接口访问到 HDFS 中的数据，然后，FUSE

内核模块通过 libfuse 接口来对获取到的数据进行处理，用户在 Linux 上执行的每一个命令都对应到通过 libfuse 实现的一些回调函数中，FUSE 内核模块随后与 Linux 的文件系统进行通信，在本地目录上进行挂载，这样一来，就完成了远程 HDFS 存储集群上的文件和目录到本地的映射。

通过使用 FUSE 访问 HDFS，解决了之前访问 HDFS 中的数据时往往不太直观，需要使用 HDFS 提供的 Java 版 API 或者其他语言的接口来进行读写的问题，使得具体应用程序的开发过程变得简单。此外，在实际的云存储应用场景中，为了更加方便的快速读取数据，以提升访问效率，业务方可能会将 HDFS 上的数据在本地进行一个备份，而鉴于 HDFS 主要用来做大规模数据的存储，这毫无疑问会带来相当规模的额外资源浪费。FUSE 访问 HDFS 的方式，同样非常契合这种应用场景。

2.4 现有技术的不足与优化方案

尽管 HDFS 拥有很多优势，但是当前的数据存储与访问方式在实际的应用过程中存在以下几个问题：

1）使用 FUSE 来访问 HDFS 时，由于云存储方案面对的是大规模并发数据访问请求，而 Hadoop 本身的负载均衡机制是用来解决不同 DataNode 中数据块的分配和调度问题，并非是用于解决 Fuse-DFS 挂载组件并发访问的负载均衡问题。而使用常见的负载均衡算法[28]，与 Fuse-DFS 组件本身的工作特点并不契合，无法真实反映后端集群中各节点的负载情况。

2）研究发现，相比使用 Java 版本的 HDFS 读写 API 来实现文件读写，使用 FUSE 来访问 HDFS 的方式，会带来一定的性能损失，因而，需要对其读写效率进行优化。

3）HDFS 本身并没有提供细粒度的访问控制机制，这会对云存储的安全性带来较大的影响。由于托管在云存储平台上的数据往往是十分敏感的信息，特别是对一些政府部门来讲，更是如此。如果不对用户访问进行限制，可能会导致用户信息泄露或是其他严重后果。

4）FUSE 的高可用 HA（High Availability）方案不够完善。HDFS 通常是由多个 NameNode 来组成，一般情况下，只有一个处于激活状态的主节点，而其它的都是处于未激活状态的备用节点，而当前的 FUSE 挂载组件在主备节点发生切换后，其并不会自动去切换，这会导致整个云存储系统的高可用性大大降低。

针对以上问题，本文设计了如图 2-7 所示的数据访问和存储架构：

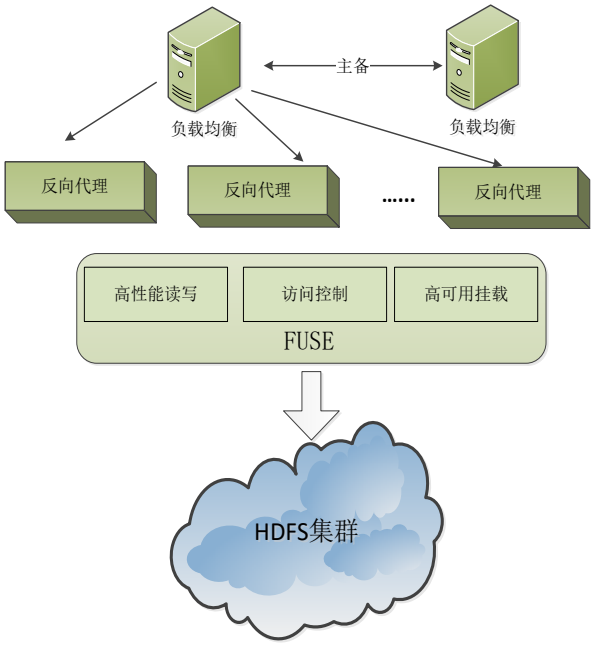


图 2-7 数据访问与存储架构设计

上图中，多台部署有负载均衡算法程序的服务器互为主备，接受用户的读写请求，并对其进行调度，直接为用户提供服务。此外，还使用了多台服务器来提供具体的数据访问服务，在这些具体的数据访问服务器中，通过反向代理的方式来返回实际的数据。参考常见数据库中的主备服务器同时运行的方式，这里设计了主备的资源调度服务器，从而保证同一时间至少有一个服务器能够正常对外提供服务，充分保证了集群设计中的高可用性。

在所设计的架构基础上，确定了具体需要对现有技术进行优化的几个方面是：

- 在资源调度这一层面，通过对现有负载均衡算法的分析，结合 FUSE 访问 HDFS 的实际资源消耗特点，设计性能表现更好，更契合本方案的负载均衡算法，并在云存储方案中进行应用。这一部分在第三章进行介绍。
- 在 FUSE 挂载这一层面，分析 FUSE 访问 HDFS 的原理，结合 Linux 文件系统中的文件读写过程，对读写性能进行优化；从文件属性的角度来出发，来设计细粒度的访问控制机制。这一部分在第四章进行介绍。
- 同样在 FUSE 挂载这一层面，设计轮询挂载方式，充分考虑 NameNode 可能出现的故障，实时的监测当前 NameNode 是处于激活状态或是作为备用节点运行，并选择合适节点进行挂载。这一部分在 5.2.3 节中进行介绍。

2.5 本章小结

本章内容主要是对云存储平台的研究过程中使用到的关键技术进行了介绍。其中，2.1 小节介绍了云计算和云存储的概念。2.2 小节对 HDFS 的设计目标以及 NameNode 和 DataNode 的工作原理进行了说明，同时也对 DataNode 中的数据复制过程进行了分析。2.3 小节先分析 FUSE 的工作原理，进而引出通过 FUSE 结合 HDFS 的 libhdfs 来访问 HDFS 的方式。2.4 小节分析了现有方案所存在的不足之处，并确定了本文所要进行的研究和优化工作。2.5 小节为本章小结。

第三章 云存储负载均衡算法研究

本章针对第二章末尾提出来的现有技术的不足，对云存储方案的负载均衡算法进行了研究。3.1 小节分析了常见的负载均衡算法工作原理，以及其在实际应用中的缺陷。3.2 小节提出了动态调整的负载均衡算法，对节点的权值计算过程以及算法中涉及到的参数选取进行了详细说明，并描述了算法的执行过程。3.3 小节通过实验对比了常用的加权轮询算法与本算法在请求调度中的实际表现，并对实验结果进行了分析。3.4 小节为本章小结。

3.1 现有负载均衡算法研究

3.1.1 典型负载均衡算法分析

在典型的分布式系统中，如何将大规模的请求分发到集群中，选择恰当的节点来处理请求，对于整个集群的运行效率和稳定性来讲是至关重要的。在当前的分布式系统中，常用的算法主要包括以下几种[29]：

1）轮询调度（Round Robin）

在轮询调度算法中，负载均衡调度器将来自用户的访问请求轮流的发送到后端服务器池中的每一个节点上。这种算法较为简单，完全不考虑服务器本身的性能和其当前的负载，集群中的每一个节点对于调度器来讲都是一样的。

2）加权轮询（Weighted Round Robin）

加权轮询算法会在初始时为后端的每一个节点赋予一个权值初始值，这个值往往是根据节点自身的提供服务的能力来确定，其能力越强，则权值越高。相应的，调度器在调度时，同一请求会优先调度到权值高的节点上。这种算法应用较为广泛，有稳定的性能表现。加权轮询算法的缺点在于，由于权值固定，往往无法实时反映后端集群中节点的真实服务能力，导致服务器资源利用率受限。

3）随机分配（Random）

这种算法在实现上来讲，一般是通过生成一个随机数，根据这个随机数，以及之前定义好的某种映射关系，将请求发送到后端的服务器池中的某个服务器上来进行具体处理。这种方法实现起来简单，应用也较为广泛。但是当考虑到服务器池中出现某个节点宕机时，这个算法可能无法正常工作。

4）最快响应算法（Least Response Time）

一般来讲，这种算法是通过记录历史访问请求到达服务器中每个节点的处理时间，对其求平均值，然后可以获得一个平均响应时间最快的节点。具体实现上，一般使用 ICMP 包或者自定义的 UDP 数据包来主动对后端服务器池中的每个节点进行探测，获取响应时间。这种方法有一个很明显的缺点，由于通过主动检测获取来的数据是一个历史记录的平均值，因而其存在一定的滞后性，无法实时反映集群中的节点具体运行状态。

5）最少连接数（Least Connections）

调度器上记录了后端服务器池中每一个节点的具体连接数，该连接数表明了当前该节点正在处理的请求数目。在此基础上，调度器会从后端节点中选择一个连接数最小的节点，用于提供服务。

3.1.2 现有算法的不足之处

在大规模的分布式系统中，集群的运行环境是相当复杂的。不同集群中的网络环境，节点的硬件性能，集群所依赖的软件环境都不一样。在这种情况下，集群中出现故障的可能性非常大。上一节中的算法都是从理论上对集群的负载均衡策略进行研究，但是没有综合考虑到这些问题。

为了便于分析，本文模拟了如图 3-1 所示的一个集群，集群中一共有 5 台服务器，分别为 Node1，Node2，Node3，Node4，Node5。

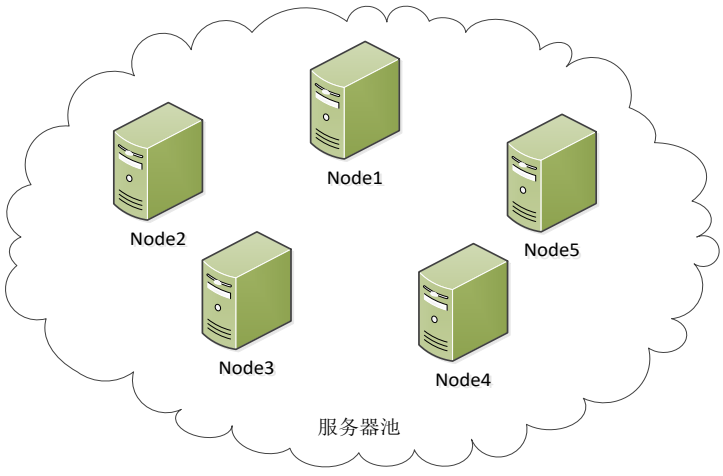


图 3-1 模拟的服务器池

假设在某一时刻，该集群接收到一个请求，而这个请求如果需要被完整的处理，一共需要完成 5 次服务器的成功处理和数据返回。如果，在某个时刻，集群中的 Node1 所代表的服务器因为网络抖动或是节点本身的软硬件错误导致无法向外提供服务，那么最

理想的情况下，到达该集群中的所有请求的$\frac{4}{5}$都可以正常处理，即处理成功的理想概率$p_{ideal}=\frac{4}{5}$。

若集群采用了轮询算法，很显然，与服务器的单次交互被分配到能够正常提供服务的节点上的概率为$p_{once}=\frac{4}{5}$，总体上来讲，这个请求被成功处理的概率为：

$$p_{success}=(p_{once})^3=\left(\frac{4}{5}\right)^3=\frac{64}{125}$$

远低于理论上应该达到的请求成功的概率p_{ideal}，如果放大到更大规模的集群，那么出错的概率会更大。同样，上述的计算也适用于使用随机分配算法的集群。

通过上述描述，下文对集群中可能出现的某些更加一般的情况进行分析。假设集群中单个节点发生故障的概率为p，那么可以得到单个节点处理请求时成功的概率为1-p。若某一个到达集群的请求需要 t 次正确的操作和返回才能正确返回数据，可以计算得到某个请求成功处理的概率为$f(p)=(1-p)^t$。这里，为了更加直观的说明请求成功处理的实际概率与理想状态之间的差距，将 t 取值为 5，在图 3-2 中，反映了这种情况下两者之间的趋势：

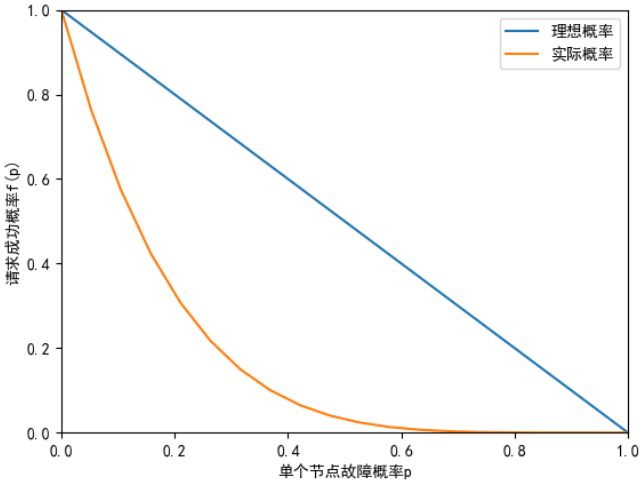


图 3-2 理想情况和实际概率之间的差别

从图 3-2 中可以很明确的看出，随着单个节点故障率的上升，会导致整个集群向外提供可靠服务的概率大幅下降。因而，可以得到一个结论，轮询算法并不适合使用在对稳定性要求高的大规模集群中。而使用随机分配算法，其结果与轮询算法类似。所以，单纯的使用这两种策略，很明显会导致单个节点的错误被扩大到整个集群中，无法满足

设计需求。

再假设集群中当前使用的策略是最小连接数算法。此时可以将集群中的各个节点的请求处理能力进行量化，认为故障节点处理该请求的能力 $L_{malfunction}=1$，而正常节点处理该请求的能力 $L_{normal}=10$。那么根据最小连接数算法的定义，可以对该请求被发送到一个正常节点的概率 p_{once} 如下计算： $p_{once}=\frac{L_{normal}*normal}{L_{normal}*normal+L_{malfunction}*malfunction}=\frac{40}{41}$。这里 normal=4，malfunction=1，分别表示正常节点和故障节点的数目。

参考前文所述计算方法，一个请求被成功处理的概率 $p_{success}$ 可以计算为：

$$p_{success}=(p_{once})^3=\left(\frac{40}{41}\right)^3\approx 0.92$$

在这种情况下，其结果超过理想状态的概率 p_{ideal}。不失一般性，将上述推导引申到更加通用的情况，使用 n 来表示集群中的节点数目，$1/q$ 表示在某个节点出现异常的时候，它向外提供服务的能力会下降为正常情况下能力的比例。那么，可以得到如下结果，对于某个集群来讲，当它采用最小连接数的算法时，在某一段时间内其向外提供服务的能力可以量化为：$n(1-p)q+np$，显然可以得到，对于到达集群的某一次操作，其可以得到正常响应的概率是：

$$p_{once}=1-\frac{np}{n(1-p)q+np}=1-\frac{p}{q-pq+p} \tag{3-1}$$

总体上来讲，对于某一次请求，其成功的概率可以用如下式子来表示：

$$f(p)=(p_{once})^t=\left(1-\frac{p}{q-pq+p}\right)^t \tag{3-2}$$

类似前文的假设，在这里分别令 t 分别取 5，8，10，q=10，那么可以得到实际情况下单次请求的成功概率与理想情况下概率的对比图：

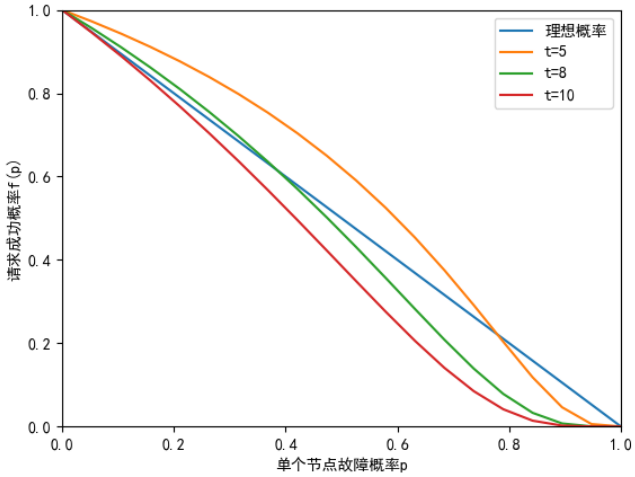


图 3-3 不同 t 值下最小连接数算法请求成功概率

从图 3-3 中可以初步的得出一个结论，当 t 值越大，也就是单个请求较为复杂，需要多次与集群中的服务器进行交互的时候，会导致实际的成功概率与理想情况下的概率差距越来越大。当 t 为 5 时，两者之间的差距并不是特别明显；但是 t 为 8 时，则在多数情况下，实际成功概率都要低于理想概率；而 t 为 10 时，很明显，这意味着几乎不可能出现达到理想概率的情况。

进一步地，为了分析 q 值对于成功概率的影响，这里假设某一时间段内发生故障的机器数是确定的，也就是说在概率函数 $f(p)$ 中，其 p 值是确定的。这种情况下，分别取 p 为 0.1，t 分别取 3、5、7，而 q 值可以理解成为集群对于节点出现异常时所设置的超时时间，时间越长，代表着 q 值越大。这样可以得到成功概率 $f(q)$ 的趋势如图 3-4 所示：

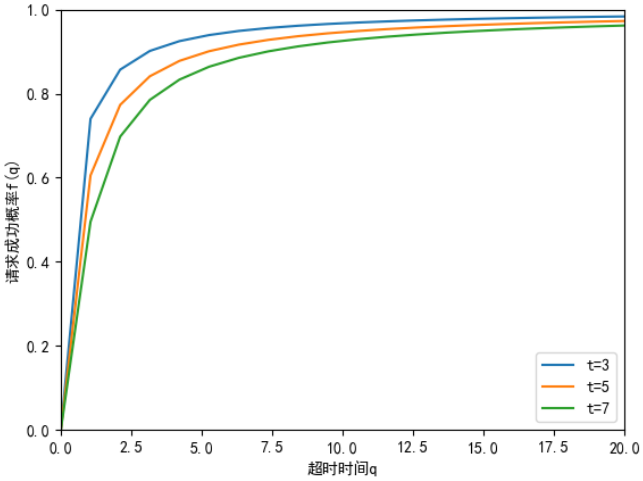


图 3-4 不同 q 值下请求成功的概率

针对图 3-4，首先，对于单条曲线来讲，随着 q 值的增大，该曲线迅速上升，即请求成功的概率迅速上升，这意味着 q 值是否合适对于整个集群对外成功服务的概率影响非常大。在一个大规模的集群中，这往往是不合理的，因为在实际应用场景中往往很难量化单个节点在出现故障以后其提供服务的能力到底应该如何定义。如果 q 值定义的不合理，则会对整个算法的可用性带来很大的影响。此外，横向对比三条曲线，可以看到，当来自外部的请求较为复杂，需要多次与集群进行交互时，其成功率往往较低，这同样是最少连接数在理论上所存在的一个短板。

3.2 动态调整的负载均衡算法

3.2.1 算法设计

综合前两节的分析，上文所提到的负载均衡算法，在某些场景下，都存在性能表现不好或是稳定性不足的问题，因而，在本文所提出的云存储方案中，不能直接套用这些现有的算法。于是，针对本文所提出的使用 FUSE 访问 HDFS 方案的特点，本文设计了了一种新的数据访问请求调度算法[30, 31]。在集群向外提供数据访问服务的节点上，使用本文设计的请求调度算法，将任务分配到具体的 FUSE 挂载节点上去，来提供访问服务。

在大规模的集群中，每个节点所承受的负载在不同的时间段内会发生很大的变化。单纯的使用节点当前的连接总数来判断服务器的负载，是不合理的。因为不同的任务请求，其对服务器资源的要求往往也有很大的差别，连接总数并不能完全反映服务器的资源消耗情况。通过对影响服务器性能的因素进行分析，CPU（Central Process Unit）的资源占用是影响服务性能的首要指标，因为服务器上所有的计算都是在这里完成；而内存占用和磁盘的 I/O 效率则往往决定了单个节点提供服务的上限[32]。在本文所提出的数据访问和存储方案中，使用 FUSE 访问 HDFS 时，由于其本身是通过在 C 语言代码中使用 JNI 来调用 Java 接口来实现对数据的访问，一次完整的请求读写过程涉及到各种组件和系统服务的协同工作，因而其对服务器的 CPU 和内存资源要求是比较高的。在此基础上，磁盘效率则决定了在数据从 HDFS 中缓存到本地目录以后向外传输的能力。根据以上分析，确定了以 CPU、内存、磁盘占用率为核心的三个衡量节点提供服务能力的指标。

在初步确定几个主要的指标以后，需要考虑如何将这些指标量化，具体的反映出节点的实际运行状态。这里，本文定义 W 为根据这几个指标所计算出来的一个权重，权

重越大则相应节点的任务处理能力越强。但是，在实际的大规模集群中，节点的权值不应该是始终固定的，因为单个服务器上处理的任务数量和复杂程度都在不断地变化，如果权值不及时进行调整，会导致数据存在滞后性，不能实时地反映节点的服务能力。因此，这里定义了一个时间周期 T，在该周期内，调度器会主动发送心跳检测，获取每个节点当前的各个指标，重新为其计算权值。图 3-5 描述了本文定义的这种负载均衡模型。

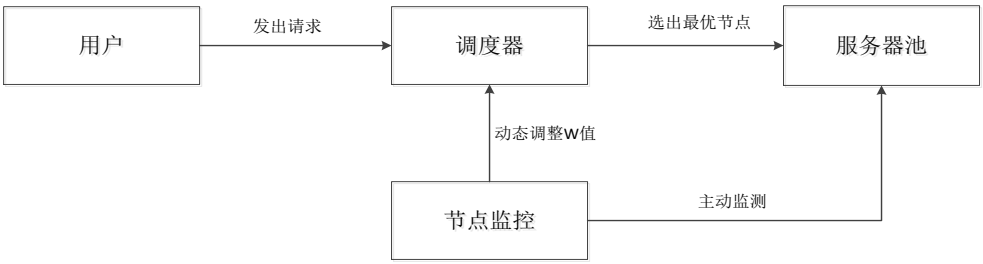


图 3-5 动态负载均衡算法模型

3.2.2 节点的权值定义与参数选取

定义后端的服务器池为$S=\{S_0,S_1,....S_{n-1}\}$，集群中以$S_i$来表示第 i 个节点。定义权值集合$W=\{W_0,W_1,...W_{n-1}\}$其中，$W_i$为集群中第 i 个节点$S_i$的权值。定义集群中第 i 个节点的 CPU 占用率为C_i，内存资源占用率为M_i，磁盘 I/O 占用率为D_i。对于这三个指标，由于实际应用中的存储集群中使用的是装有 Linux 操作系统的服务器，因而，本文主要是根据 Linux 服务器/proc 目录下的一些文件和常见的开源工具来具体的计算这三个指标。

- 对于 CPU 占用率，在 Linux 中，通过读取/proc/cpuinfo 文件进行分析。由于 CPU 是时刻都在高速运转的，其状态时刻都在变化。因而，如果想要准确计算，那么不能只是获取某个时间点的 CPU 占用率。在所读取的文件中，CPU 数据主要分为以下几个部分：user，nice，system，idle，iowait，irq，softirq，steal，guest，guest_nice。那么某一个时间点总的 CPU 时间为：

  $$c_i=user_i+nice_i+system_i+iowait_i+irq_i+softirq_i+steal_i+guest_i+guest\_nice_i$$

  定义某一个时间段位t_1~t_2，其 CPU 占用总时间分别为c_{1i}，c_{2i}，空闲时间分别记为$idle_{1i}$，$idle_{2i}$，那么：

  $$C_i=(c_{2i}-c_{1i})/(idle_{2i}-idle_{1i}) \quad (3\text{-}3)$$

- 对于内存占用率，通过读取/proc/meminfo 来获取信息。获取当前已经占用的内存记为 $memused_i$，其空闲内存记为 $memidle_i$，则对内存占用率的定义如下:

$$M_i=memused_i/(memused_i+memidle_i) \quad (3\text{-}4)$$

- 对于磁盘占用率，使用开源工具 iostat 来获取信息。iostat 工具提供了丰富的参数，可以直接通过其中的 util 项来获取当前的磁盘占用情况，记为 D_i。

根据前文的分析，本文使用如下公式来计算节点权值：

$$W_i = \frac{1}{k_1C_i + k_2M_i + k_3D_i} \quad (3\text{-}5)$$

在公式（3-5）中，k_1，k_2，k_3 分别代表 CPU 占用率、内存占用率、磁盘 I/O 占用率在权值计算中的重要性比例，对其有如下规定：$i \in \{1,2,3\}, k_i \in (0,1), \sum_{i=1}^{3} k_i = 1$。针对不同集群实际的硬件配置和应用场景，可以对 k_i 设定不同的系数。将本文对 C_i, M_i, D_i 三个指标的计算代入公式（3-5）中，即可得到服务器池中的某一个节点的权重。可以得出，在某一周期 T 内，为来自客户端的某一次访问请求提供服务的节点 S_T 可以如下表示：

$$S_T = \arg\min_{S_i \in S} \left(\frac{1}{k_1C_i + k_2M_i + k_3D_i} \right), i \in \{1,2,3\}, k_i \in (0,1), \sum_{i=1}^{3} k_i = 1 \quad (3\text{-}6)$$

在对每个节点权值根据已经确定的三个指标进行计算的基础上，还需要考虑到，无论是 CPU 资源，内存资源还是磁盘 I/O 占用情况，都不能任其无限制增长，达到 100%。长时间资源占用如此高，可能会对服务器的硬件造成不可逆的损伤，而且，在资源占用率如此高的情况下，往往也无法对外提供高效的服务。因而，针对这三个指标，本算法都定义了能够提供服务的阈值，如果达到或者是超过该阈值，调度器会将其从当前周期内可以提供对外服务的节点集合中移除。

在将算法应用到实际的生产环境中之前，需要对本文提出的动态调整的负载均衡算法中未确定的相关参数进行取值。首先，在公式（3-6）中，k_1，k_2，k_3 取值的不同，将对算法的性能产生较大的影响。在不同的应用场景下，由于处理数据访问对资源占用的的情况不一，应该对具体情况进行分析，来决定取值大小。为了提升算法在不同系统中的实用性，规定 $\sum_{i=1}^{3} k_i = 1$。算法的使用者可以根据不同的应用场景来对参数做出调整，

不断修正，以达到最佳的性能表现。在本文中，经过对 FUSE 访问 HDFS 的过程进行分析，发现其性表现主要由几个因素来决定：1）FUSE 读写 HDFS 过程中会有频繁的 CPU 上下文切换，导致对 CPU 计算能力要求较高；2）考虑到后文的优化过程需要将部分文件信息缓存到内存中，所以内存资源占用也很大程度上决定了性能上限。在确定了几个指标的重要性之后，本文使用 TOPSIS 算法[33]来辅助精确确定参数的值。TOPSIS 算法用于在多目标决策分析中选择最优解，在本文中，只需要输入三个指标的重要程度对比表，就可以将各自在节点权值计算过程中的重要性进行量化。TOPSIS 算法中提出了理想解和负理想解的概念。这里的理想解是所设想出来各方面评价指标都达到最优的值，而负理想解是设想的最差的值，算法的目标就是要找到一个距离最优值最近并且距离最差值最远的解，这就是所要求的最优方案。该算法的流程如图 3-6 所示。

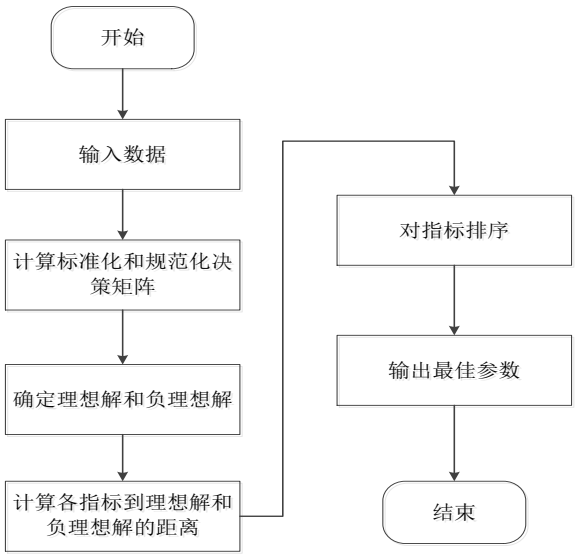


图 3-6 TOPSIS 算法执行过程

具体来讲，主要分为以下几个步骤：

1）假设在某一个应用场景中，共有 m 个参数的权值需要确定，分别记为 $D_1,D_2,...,D_m$，而当前算法的性能是由 n 个评价指标来进行衡量的，分别记为 $X_1,X_2,...,X_n$。首先需要输入参数权值与其具体性能表现所组成的矩阵，将该矩阵记为 D，则有：

$$D=\begin{bmatrix} x_{11} & \cdots & x_{1j} & \cdots & x_{1jn} \\ \cdots & & \cdots & & \cdots \\ x_{i1} & \cdots & x_{ij} & \cdots & x_{in} \\ \cdots & & \cdots & & \cdots \\ x_{m1} & \cdots & x_{mj} & \cdots & x_{mn} \end{bmatrix}=\begin{bmatrix} D_1(x_1) \\ \cdots \\ D_i(x_j) \\ \cdots \\ D_m(x_n) \end{bmatrix}=\left[X_1(x_1),...,X_j(x_i),...,X_n(x_m)\right] \quad (3\text{-}7)$$

2）一般来讲，在具体进行计算时，需要对矩阵进行标准化，记标准化后的值为 r_{ij}，

过程如下所示：

$$r_{ij}=\frac{x_{ij}}{\sqrt{\sum_{i=1}^{m}x_{ij}^{2}}},i=1,2,...,m,j=1,2,...,n \quad (3\text{-}8)$$

3）定义集合 w 来表示第一步中各个指标的重要性，w_j 表示的是第 j 个指标的重要性。矩阵 w 中的值需要根据这 n 个指标的相对重要性来进行确定，往往需要结合实际的场景来选用不同的方法来进行规范化，其具体过程如下：

$$v_{ij}=w_j r_{ij},i=1,2,...,m,j=1,2,...,n \quad (3\text{-}9)$$

4）确定理想解 A^+，这里的理想解即在某个指标上达到最佳的一个取值。J_1 为第 i 个指标上的最优解，J_2 为最差解。

$$A^+=(\max_i v_{ij}|j\in J_1),(\min_i v_{ij}\, j\in J_2),|i=1,2,...,m=v_1^+,v_2^+,...v_j^+,...,v_n^+ \quad (3\text{-}10)$$

5）确定负理想解 A^-，与理想解相对的，这里的负理想解就是在某个指标上达到最差的取值。J_1 为第 i 个指标上的最优解，J_2 为最差解。

$$A^-=(\min_i v_{ij}|j\in J_1),(\max_i v_{ij}\, j\in J_2),|i=1,2,...,m=v_1^-,v_2^-,...v_j^-,...,v_n^- \quad (3\text{-}11)$$

6）对于每一个待确定权值的参数，计算其到理想解和负理想解的距离，分为记为 S^+ 与 S^-。这里的距离计算方式可以为多种，多维欧氏距离较为常用。具体如下：

$$S^+=\sqrt{\sum_{j=1}^{n}(V_{ij}-v_j^+)^2}\ ,\ S^-=\sqrt{\sum_{j=1}^{n}(V_{ij}-v_j^-)^2}\ ,\ i=1,2,...m\ 。 \quad (3\text{-}12)$$

7）对于每一个参数，使用公式（3-13）计算出一个比率。将该比率从大到小进行排序，就表示了每个参数在实际情况中的重要性比例，以此来确定每个参数权值大小。

$$R=\frac{S^-}{S^-+S^+} \quad (3\text{-}13)$$

经过理论推导，结合经验值，本文在实际应用场景中确定的系数值分别为 k_1=0.5，k_2=0.3，k_3=0.2。具体的计算过程较为繁琐，这里仅说明方法。

此外，由于对节点负载信息的采集是每个周期 T 内进行一次，T 值的选取不能太大。如过每次权值更新的周期比较长，调度器所计算出来的权值将无法反映集群最新的负载情况，存在一定的滞后性，最终会对负载均衡算法的调度效率造成一定的影响。但是，如果过于频繁的去查询集群中各个节点的信息，一方面会给集群中的节点带来不必要的

负担；另一方面，鉴于在实际应用中，集群中的节点数目规模往往是非常大的，如果频繁的传输集群负载信息，并计算整个集群所有节点的权值，可能会造成网络堵塞或是负载均衡调度器本身资源占用过高。因而，对实际情况进行分析以后，这个值一般设置为5 秒至 15 秒之间比较合适。

3.2.3 动态负载均衡算法执行过程

在本文所提出的方案中，由于服务端采用的是 RESTful 风格 API 的方式来向外提供文件访问服务，所以在调度器实现的时候，本文通过 Unix 操作系统中的 IPC（Inter-Process Communication）通信方式与调度器进行通信，使用 socket 编程中的 I/O 多路复用技术，让请求访问的客户端与服务端建立起 TCP 连接。在这里，socket 本身提供了多种事件模型，如 epoll、poll、select 等。为了提升调度器的并发处理能力，调度器中使用的是 epoll 事件模型。该模型的优势在于其相比另外两种事件模型，在占用相同的服务器资源时，epoll 可以处理更多的任务。并且，当有空闲资源来处理新的任务时，其会主动通知处理等待队列中的事件，这让其相比其它两种事件模型在处理等待事件时更加高效。图 3-7 描述了本文中的动态负载均衡算法执行过程。

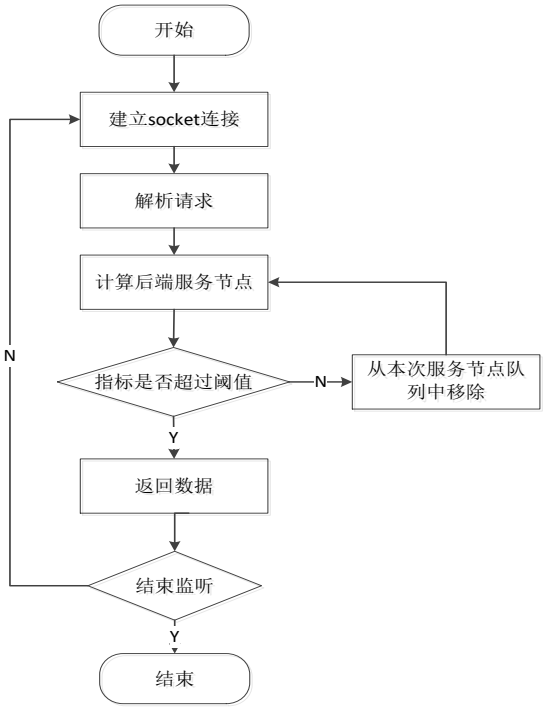


图 3-7 动态负载均衡算法执行过程

如图 3-7 所示，当一个数据访问请求通过 RESTful 风格 API 到达负载均衡调度器时，具体的处理过程如下：

1）客户端和服务器建立一个 TCP 连接，在本方案中，服务端直接监听 80 端口来

获取 HTTP 请求；

2）所传输过来的请求，会包括各种不同的访问操作，在后文的云存储方案实现中会详细介绍。这里需要对其进行解析，来判断是何种操作；

3）调度器通过这一周期 T 内主动访问后端服务器集群所计算出来的节点权值，选取当前最优的服务节点，将访问请求发送给该节点对应的服务器；

4）被选中的服务器判断自身是否有指标已经超过了定义的阈值，这里全部定义为 90%，如果是，则向调度器返回失败的消息，调度器同时将该节点中该周期内的可用服务节点中移除，然后跳到步骤 3；

5）如果被选中的服务器当前运行情况正常，那么该服务器将通过 FUSE 访问 HDFS，返回所需要的数据，完成一次成功的请求；

6）调度器继续监听在 80 端口，等待下一个数据访问请求的到来。

3.3 实验结果与分析

3.3.1 实验环境

系统测试环境如表 3-1 所示：

表 3-1 软件环境

类型	版本
操作系统	CentOS 6.5
服务软件	Nginx，Keepalived

服务器的硬件配置如表 3-2 所示：

表 3-2 硬件信息

CPU	内存	SSD	硬盘
Intel E5-2660	16G	128G	1T

实验环境网络拓扑结构如图 3-8 所示，部署了两台负载均衡调度器互为主备，采用本文设计的调度算法来进行请求分发。在后端的服务器池中，配置了三台用于向外提供服务的 Linux 服务器，通过 Nginx 反向代理向外提供 Web 访问服务。同时，使用 Apache Benchmark 基准测试工具来模拟请求。

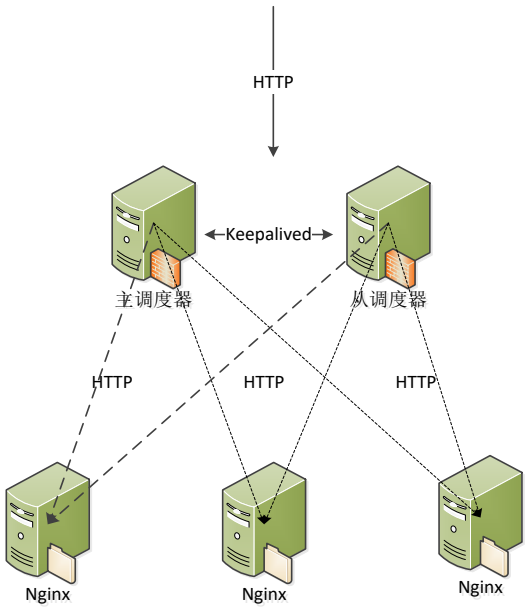


图 3-8 负载均衡集群的网络拓扑结构

3.3.2 实验方案设计

在设计对比实验时，为了更加真实的模拟在大规模集群中用户并发访问请求的到达情况，本文假设到达调度器的数据访问请求是服从 Poisson（泊松）分布的，那么可以得到请求 r 到达的具体时间间隔：

$$P(X=r)=\frac{\lambda^r}{r!}e^{-\lambda}, r\in N^+ \tag{3-14}$$

在具体实现过程中，取 λ=150，可得到函数图像如图 3-9 所示[34, 35]。

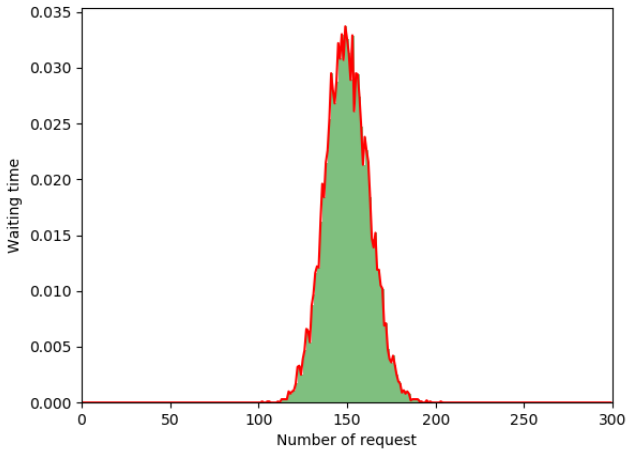


图 3-9 泊松分布图像

为了真实模拟存储集群中大规模用户并发访问时调度器的实际处理情况，假设其处

理某个请求的时间是服从 Pareto（帕累托）分布[36]，其分布函数如下：

$$P(X > x) = \left(\frac{x}{x_{\min}} \right)^{-k} \tag{3-15}$$

其中 x 是任何一个大于 $x_{\min}$ 的数，$x_{\min}$ 是 X 最小的可能值，k 是为正的一个参数。通过 Pareto 分布函数，生成一个随机数，用来表示某一事件具体的处理时间。

由于本文设计的实验需要准确地评估本文所设计的算法在实际应用过程中的表现，本文选用了在负载均衡算法中常用的三个指标来评估算法的表现，分别是单位时间吞吐率、请求处理时间、请求等待时间。为了更加方便的对实验结果进行数学上的分析，这里定义集合 $P = \{P_1, P_2, ...P_{n-1}\}$ 为不同请求数下的单位时间吞吐率，其中 P_j 表示的是请求数为 j 时的吞吐率；定义集合 $Q = \{Q_1, Q_2, ...Q_{n-1}\}$ 为不同请求数下的请求处理时间，其中 Q_j 为请求数为 j 时的请求处理时间；定义集合 $R = \{R_1, R_2, ...R_{n-1}\}$ 为不同访问请求数下的请求等待时间，其中 R_j 为请求数为 j 时的请求等待时间。

对这三个指标具体说明如下：

1）吞吐率数据即集合 P。对于 Web 服务器来讲，单位时间内能够处理的网络数据量是衡量其性能的第一标准。吞吐率说明了服务器所能向外提供的最高负载能力。本质上来讲，它衡量的是通网络所传输的数据。

2）请求处理时间数据即集合 Q。实际上，请求处理时间从数值上来讲衡量了服务器的整体服务质量。对于单个用户来讲，请求处理时间衡量的是单个请求从调度器开始处理到处理完成的时间差。

3）请求等待时间数据即集合 R。请求等待时间描述的是 Web 服务器在处理来自大量用户的并发访问请求时，由于自身性能的限制，单位时间能处理的请求量是有限的，因而必然会有请求运行在后端，等待调度器的调度。这个指标，则是从总体上衡量了单个请求所需等待的时间，往往是最能够反映用户实际体验的。

3.3.3 实验结果

相比文献[30]以及文献[31]中所提出的负载均衡算法，本文所设计的算法主要有几个优点：

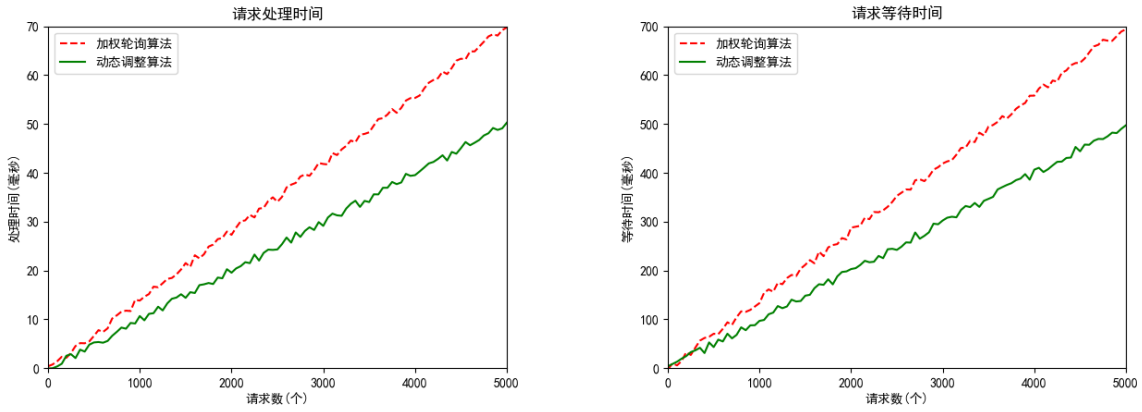
1）本文使用的算法的所考虑的影响服务器负载的指标更加全面，能够更加准确的

反映集群中单个节点的实际负载情况，因而能带来更高的调度效率；

2）本文所选取的指标要更加契合实际的应用场景，在实际的使用中性能更好；

3）本文所设计的算法在参数确定时，使用 TOPSIS 算法来辅助确定参数值，理论依据更加充分，并非完全是依赖经验值。

本文设计的实验，主要是对比了加权轮询算法和本文所设计的算法在具体的存储集群中的实际表现。选择加权轮询算法的原因在于，在实际的大规模集群中，由于综合考虑到算法的性能和稳定性，往往需要选择一些有较好性能表现，同时又能保证整个系统高可用性的算法，用来做实际的访问请求调度，而加权轮询算法由于其良好地性能和可靠的表现，在实际场景中应用广泛。本算法可以理解为对简单加权算法的改进，因而，与其进行对比，能够较好的反映本文所做优化带来的实际效果。经过对比，得到的实验结果如下：



a)请求处理时间　　b)请求等待时间

图 3-10 处理时间与等待时间

在图 3-10 中，可以看到，a、b 两幅子图分别表示的是不同请求数的平均处理时间和平均等待时间。两幅图在走势上是一致的，这是因为本文模拟了多个线程并发进行数据访问，因而等待时间与处理时间存在倍数关系，该倍率接近单位时间内并发数。图中可以看到，当请求数规模不大时，调度算法的不同对于具体的请求处理时间和等待时间差异并不大。这是由于此时后端集群的负载并不高，因而反映到各个节点的资源占用上，差异并不大。所以，此时使用本文设计的动态调节算法，调度器对于权值的调整并不明显，最后的效果类似于加权轮询算法。当请求数量开始增大以后，此时，后端集群中节点的负载程度开始增大，相应的通过本文所定义的公式计算出来的权值也开始出现明显的波动，反馈到调度器后，节点的服务器权值也会有相应的调整。因而，处理相同请求的时间相比加权轮询算法也有了较大的降低。

实验测得不同连接数下的吞吐率如下图所示：

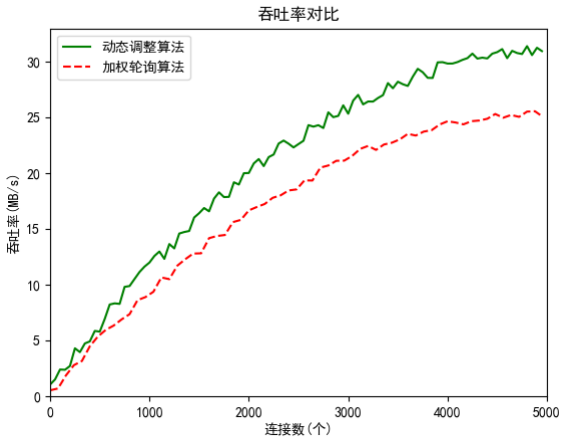


图 3-11 不同连接数下的吞吐率

图 3-11 反映了不同请求数下分别使用本文设计的算法与加权轮询算法时服务器的吞吐率的变化曲线。与请求处理时间和等待时间类似，随着连接数的增长，本文所设计的算法相比加权轮询算法同样有更好的表现。需要注意的是，在前文中已经提到过，本文所设计的算法，相比普通的加权轮询算法，会带来一定的权值计算代价，而事实上，由于算法的计算过程是轮询后端集群中每一个节点，来计算负载程度，因而其复杂度为 $O(n)$。并且，本文使用的权值更新周期 T 经过反复验证，以减少权值计算过程所带来的额外代价。总的来讲，本文所设计的算法在处理文件读写请求时，相比传统的负载均衡算法，在增加少量的权值计算代价以后，应用到云存储系统中，能够带来更快的文件上传或下载速度。

3.4 本章小结

本章主要描述了云存储方案中的请求调度算法的设计与实现过程。首先调研和分析了常见负载均衡算法的不足，然后针对云存储方案的特点，设计了动态负载均衡算法。最后对算法进行了测试，结果表明其相比常用算法要有更好的性能表现。

第四章 云存储数据访问优化研究

本章的主要内容是对云数据访问过程进行优化，通过分析 FUSE 挂载 HDFS 的过程，从提升 FUSE 读写性能和安全性两个方面进行了优化。4.1 小节介绍了云数据访问过程的优化方案，主要从性能优化和安全性优化两个层面入手；4.2 小节对 HDFS 的挂载组件 Fuse-DFS 的挂载和文件读写过程进行了详细的分析，并对 FUSE 读写性能进行了优化；4.3 节对云数据访问过程的安全性进行了优化；4.4 小节通过设计相关实验对 FUSE 访问 HDFS 在优化前后读写性能进行了对比；4.5 小节为本章小结。

4.1 云存储数据访问优化方案设计

在本文 2.4 节中，提出了当前使用 FUSE 来访问 HDFS 存在的不足之处，主要有两个方面的问题：

- 使用 FUSE 来访问 HDFS 时，没有对访问者的权限进行限制，任何用户只要连接到 FUSE 就可以对 HDFS 上的数据进行修改，这会带来安全性的问题。
- 相比于使用 HDFS 所提供的 Java 版文件读写 API 来访问 HDFS 集群上的文件，使用 FUSE 会带来一定的性能损失。这主要是由于：1）FUSE 运行过程中在内核态与用户态之间进行大量切换；2）FUSE 没有对文件的元数据信息进行缓存，包括 inode 号和 dentry 信息；3）FUSE 使用的是缓存 I/O 方式，并不是直接将数据写入到磁盘，而是先缓存到内存中，然后再进行读写。

在进行优化之前，使用 FUSE 访问 HDFS 上的数据流程可以用图 4-1 来表示。

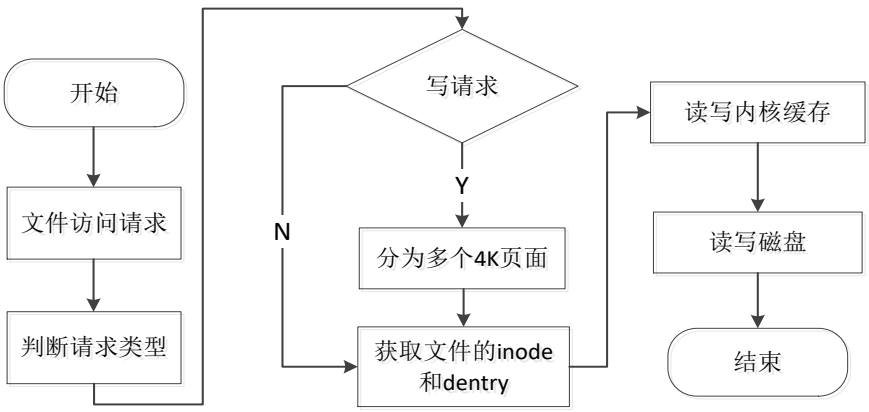


图 4-1 优化前的数据访问流程

针对前文提出的问题，在深入研究了优化之前使用 FUSE 访问 HDFS 的过程所存在

的问题之后，本文提出了如图 4-2 所示的优化方案（注：图中与优化前方案存在的差异用不同颜色做了区分）：

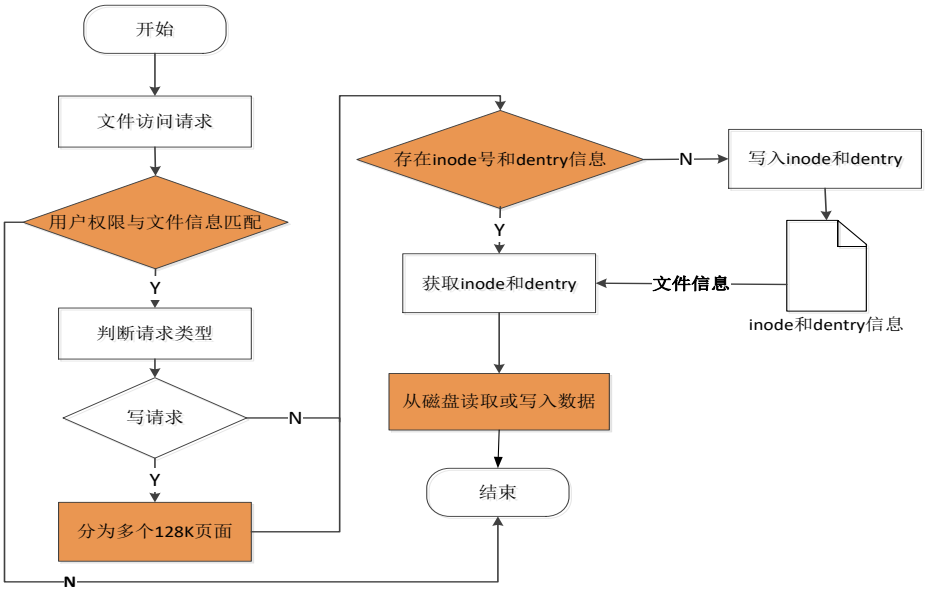


图 4-2 优化后的数据访问流程

优化后的方案，在接收到一次数据访问请求时，具体的执行过程如下：

1）首先，HDFS 的挂载组件 Fuse-DFS 会对文件的权限与当前挂载进程执行者的权限进行比对，如果验证通过则会继续执行，否则返回错误信息；

2）在此基础上，程序会判断请求类型，如果是写请求，则会将其分为多个 128KB 大小的页面，来与内核进行交互，否则直接进入步骤 3；

3）在内核中的 inode 和 dentry 信息中查找是否存在文件的缓存信息，如果存在则直接读取，如果不存在则调用相关接口来获取信息，再写入内核缓存中；

4）获取到文件数据以后，直接向磁盘中读取或者写入数据，无需经过内核缓存。

可以看到的是，相比原有的 FUSE 访问 HDFS 过程，优化后的方案，相比原有方案主要有以下几个不同：

- 首先，在保证数据访问安全性方面增加了细粒度的访问控制，对发起数据访问的用户进行权限验证。
- 其次，在提升性能方面：将内核数据页面大小由原来的 4KB 修改为 128KB；对 inode 和 dentry 进行缓存；直接从磁盘上读取数据，不经过内核缓存。

后文将针对所提出的优化方案，在分析 FUSE 源代码以及 HDFS 挂载组件 Fuse-DFS 的基础上，对其进行具体的实现，并分析本文进行优化的依据。4.2 节介绍了性能优化的实现，4.3 节介绍了访问控制优化过程。

4.2 FUSE 读写性能优化

4.2.1 VFS 文件读写过程分析

在第二章中已经介绍过，Fuse-DFS 挂载组件是基于 FUSE 来开发的。在 Linux 中，同时支持多种文件系统，如 EXT，EXT2，FAT32 等，其实现基础是 VFS 技术。VFS 向外提供了一个统一的操作接口，用户的操作如 ls、cat、rm 等会通过 VFS 进行转换以后，传递给具体的文件系统来进行处理。这些文件系统相互之间是独立的，互不影响。

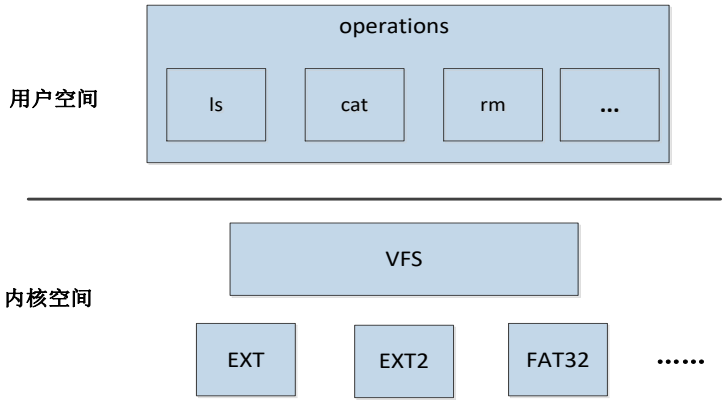


图 4-3 VFS 多文件系统支持

如图 4-3 所示，在上层，普通用户所执行的命令运行于用户空间，这一层面的操作即使是不同的文件系统也都是一样的。之后通过 VFS 提供的接口，将其转换为特定文件系统中对应的方法。以下以一个来自用户空间的读取命令为例，具体描述其在 VFS 中的完整生存周期。

首先，应用程序会以如下方式调用 read()函数：

```
ret = read(fd, buf, len);
```

这个操作所带来的效果是打开文件描述符 fd 所表示的文件，读取 len 个字节，然后放入 buf 数组中进行缓存。底层的实现过程是，read()会调用 VFS 中的 sys_read()接口，然后通过各文件系统的实现，来调用其中具体的读取文件处理逻辑，访问磁盘，完成整个数据读取过程。整个过程可以如图 4-4 来表示。

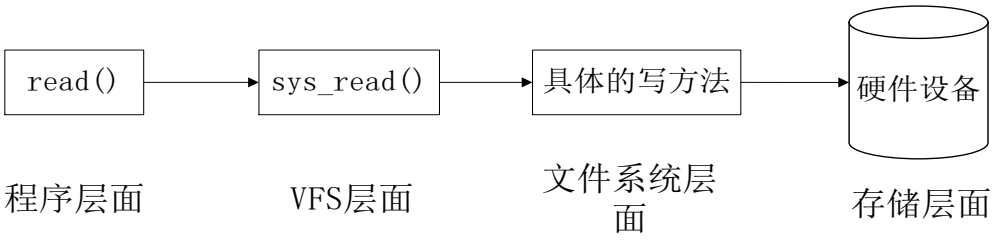


图 4-4 VFS 中的文件读取过程

4.2.2 FUSE 中文件读写过程分析

在 FUSE 中，总体上来讲，按照各部分职能的不同，可以分为两个模块：

- 运行于操作系统内核空间的 FUSE 内核模块。当 FUSE 内核模块被加载以后，会通过 VFS 注册一个文件系统。但是，在这个文件系统中，具体的文件处理方法没有实现。FUSE 内核模块本身可以看成是一个用户空间自定义函数与 VFS 进行通信的代理。
- 运行于用户空间的编程接口 libfuse。为了方便实际文件系统的开发，libfuse 提供了两组不同的 API（Application Programming Interface），上层 API 和底层 API。底层 API 主要完成以下几件事情：接收来自内核的请求并进行解析；对请求以某种固定的格式进行响应；管理文件系统的配置和挂载；将用户空间和内核空间可能存在的版本差异进行隐藏。对于上层 API，它的设计是以底层 API 为基础，在操作文件时，直接通过路径来确定文件位置，而不需要像底层 API 一样完成文件 inode 到路径之间的映射。在实际情况中，这往往是一个很耗时的过程。相比之下，上层 API 要更加方便于文件系统的开发。

除以上两个模块外，FUSE 还使用到了将文件系统挂载到本地目录的工具 fusermount。由于在 Linux 中，普通用户是无法执行文件系统挂载操作的，所以，FUSE 在实现时，借助于 fusermount 这个工具，来实现目录挂载。

其中，内核模块与用户空间的数据传输通过/dev/fuse 这个块设备来完成。内核读取来自该设备的请求，然后将返回的数据写回，返回给用户空间的程序。在 FUSE 中，其内核模块完成的功能是读取请求，处理完成后将数据写入/dev/fuse 并返回给用户空间程序。来自/dev/fuse 的所有请求都是通过 request（请求）队列来发送到内核。那么 FUSE 内核在处理这些请求时，其过程有点类似于常见的消息中间件中的生产者——消费者模型[37]。这一过程大致可以如下划分：

1）首先，使用文件系统的用户在文件系统的具体挂载目录下执行一些操作，这些

操作被映射为相应的系统调用；

2）所产生的系统调用，通过生产者将其保存到 request 队列中，放入相应结构体中的 pending 列表里；

3）位于内核空间的消费者，读取/dev/fuse 内的 request 对象，去 request 队列中取出处于 pending 状态的请求，进行处理。当检测到 request 中处于 pending 状态的请求为空时，FUSE 内核将不再占用 CPU。

这里以一个 write 系统调用所触发的系统调用来进行说明。当 write 系统调用被触发以后，会首先通过在挂载过程中所传入的函数指针，来调用具体的写函数。在具体的写函数中则会调用 FUSE 连接创建方法，创建一个到 FUSE 的连接，以一个结构体成员 fc 来表示。这个结构体成员作为一个私有变量则是保存在块结构体中，而块结构体本身，则是保存在具体文件的 inode 中。这样，一个系统调用就和具体的文件关联起来了。在此基础上，数据将通过 pages 结构体从用户空间拷贝到内核，这一过程是循环执行的。整个过程如图 4-5 所示：

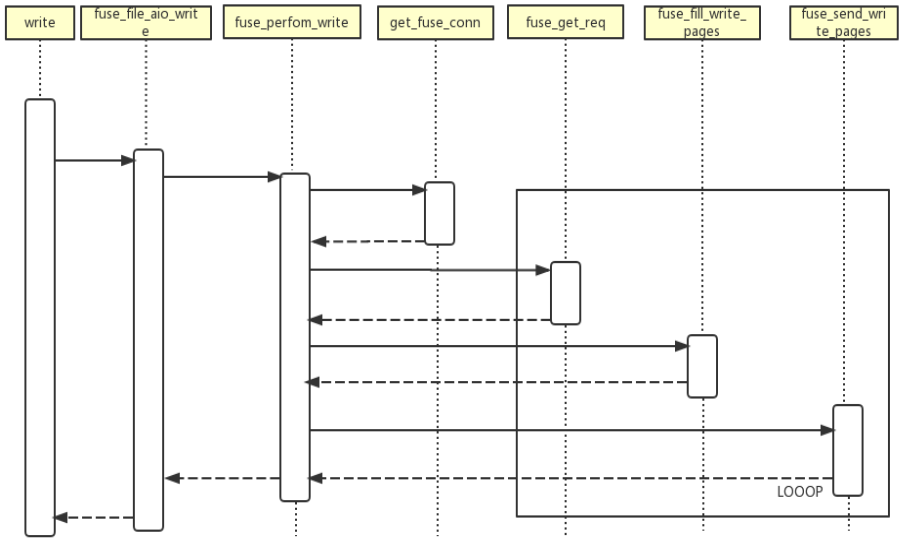


图 4-5 write 调用时序图

Fuse-DFS 挂载组件本身是基于 FUSE 来实现和运行的，当用户在具有 HDFS 访问权限的机器上执行了对远程 HDFS 的挂载命令以后，就可以将 HDFS 当成一个本地目录来访问。具体的操作过程中，首先需要在向外提供服务器的机器上执行命令：

```
fuse_dfs_wrapper.sh -oserver=master -oport=9000 /home/hadoop/mountPoint/ -d
```

fuse_dfs_wrapper.sh 是一个 shell 编写的脚本，其主要功能是调用 Fuse-DFS 的主程序 fuse-dfs 二进制可执行文件，并将命令行参数传递给 Fuse-DFS 主程序。在执行该程序

之前，脚本会首先对程序运行需要的环境变量进行设置，主要包括 JRE 路径，程序依赖的 Hadoop 的 jar 包路径等。其中，oserver 参数是 NameNode 所在的主机名，oport 是 HDFS 的访问端口，/home/hadoop/mountPoint/是指定的挂载路径，-d 选项表示使用 Debug 模式来运行程序，会在控制台输出一些调试信息。

在 Fuse-DFS 主程序开始执行以后，会从 fuse_dfs.c 中的入口函数开始执行。首先，会对 fuse_options.h 中定义的全局变量 options 进行赋值，也就是将已经传递给 fuse_dfs 主程序的参数传递给自定义类型 options 的实例 options 变量。然后，位于 fuse_init.c 文件中的 dfs_init()函数会开始执行，这个函数首先会将传递过来的 options 变量输出到控制台，在此基础上，程序会根据 options 对象的具体属性，来初始化 fuse_dfs 主程序整个生存周期中一直存在的上下文环境，也就是定义于 fuse_context_handle 中的 dfs_context 结构体的实例 dfs 变量。dfs_context 的结构体定义如图 4-6 所示：

```
typedef struct dfs_context_struct {
    int debug;
    int usetrash;
    int direct_io;
    char **protectedpaths;
    size_t rdbuffer_size;
} dfs_context;
```

图 4-6 dfs_context 结构体定义

在 dfs_context 结构体中，debug 表示是否开启调试模式；usetrash 代表使用 fuse-dfs 删除文件以后，是否要放入文件系统的/Trash 目录；direct_io 表示是否使用直接 IO 方式，默认为缓存 IO；protectedpaths 字符串表示受保护的路径，即不允许写入的路径；rdbuffer_size 表示当 FUSE 在读取 HDFS 中的内容时，应该使用多大的缓存。这个上下文环境变量非常关键，在整个程序的运行过程中都会被用到。

在完成变量初始化之后，会调用 libfuse 中的入口函数，进入内核态的操作。在这里，fuse_main()有三个参数，分别为命令行传入的参数个数 argc，参数数组 argv，以及用户自定义的 FUSE 文件操作方法结构体 dfs_oper。这个结构体是整个 FUSE 文件系统运行的核心。dfs_oper 结构体中保存了内核函数到用户函数的映射关系，基本包含了常见的

文件操作可能要用到的各种内核函数。例如，getattr 类似于 Linux 下的 stat 命令，用于获取文件属性；access 用于控制文件的操作权限；readdir 用于获取目录信息等等，这里不再一一赘述。当用户在客户端挂载执行一个文件或者目录操作以后，内核接受到来自 FUSE 的信息，解析完之后，会根据 dfs_oper 结构体中相应用户自定义的方法，获取数据，然后返回给内核，这个过程中内核的具体执行过程可以参考图 4-5 所示的 write 系统调用过程在内核中的执行步骤。然后，内核将获取的数据返回给 VFS，就可以顺利地在 Linux 中将文件或是目录的操作信息进行展示了。上述过程可以通过图 4-7 来进行表示。

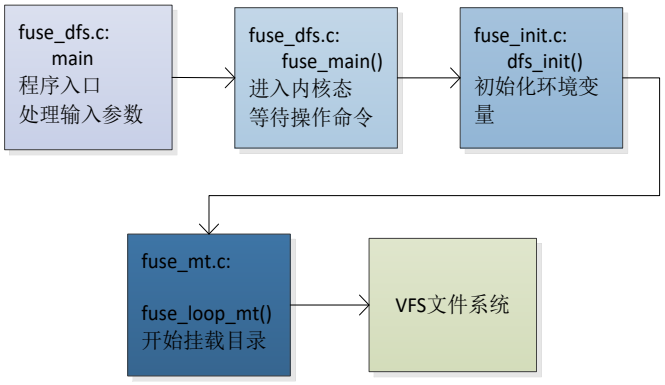


图 4-7 Fuse-DFS 总体流程

前面描述了挂载组件主程序从处理参数，到接受文件系统命令再到挂载至本地的大体过程，为了进一步分析主程序在进入到操作系统内核后的具体执行过程，本文使用了一些辅助手段来对系统调用进行跟踪。这里使用的是 strace[38]工具来对挂载和读写过程中进行的系统调用进行了跟踪。具体过程如下。

在终端执行以下命令：

```
strace -f -e open fuse_dfs_wrapper.sh -oserver=master -oport=9000 /mnt/fuse
```

上述命令中，“-f”选项代表的是同时跟踪程序中运行的子进程。因为前文已经介绍过，fuse_dfs_wrapper.sh 只是对 fuse_dfs 二进制可执行文件的一个封装，其本身只是对程序运行的环境变量进行一些设置。“-eopen”选项的含义是仅显示 open 系统调用相关的信息，这主要是到为了使所展现的信息更加明确，方便查看。执行命令后，终端输出的结果如图 4-8 所示。

```
[pid 18597] open("/etc/ld.so.cache", O_RDONLY) = 3
[pid 18597] open("/lib64/libc.so.6", O_RDONLY) = 3
[pid 18597] open("/dev/fuse", O_RDWR)    = 3
[pid 18597] open("/etc/fuse.conf", O_RDONLY) = 5
[pid 18597] open("/etc/mtab", O_RDONLY|O_CLOEXEC) = 5
[pid 18597] open("/mnt/fuse", O_RDONLY) = 5
[pid 18597] open("/etc/nsswitch.conf", O_RDONLY|O_CLOEXEC) = 6
[pid 18597] open("/etc/ld.so.cache", O_RDONLY) = 6
[pid 18597] open("/lib64/libnss_files.so.2", O_RDONLY) = 6
[pid 18597] open("/etc/passwd", O_RDONLY|O_CLOEXEC) = 6
[pid 18597] open(".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 6
```

图 4-8 strace 跟踪结果

libfuse 内核模块除了会对用户自定义的处理方法进行注册之外，还会创建一个/dev/fuse 的块设备。图中可以看出，使用的 fd 即文件描述符是 3。/dev/fuse 这个设备的作用是用户空间进程的 FUSE 进程和内核之间沟通的桥梁。前文已经提到过，内核中的 session 在处理完来自用户空间的请求以后，会将返回信息传输给上一层的 VFS，就是通过这里的/dev/fuse 设备来实现的。当应用程序调用操作系统的接口时，一般都会传入一个路径，但是在内核中，每一个具体的请求都需要包含一个具体文件的 inode 号。而从文件路径到 inode 号的转化，则是由内核来完成的。这个过程需要解析文件路径中的每一层的 inode 号，最后进行拼接。因而，如果对 inode 号不进行缓存，那么每次内核在处理请求的时候，都要重新进行 inode 的查找，这是非常耗时的一个过程。

4.2.3 性能优化实现

通过前文对挂载过程的分析，结合 Linux 上的文件读写过程，在对挂载组件进行优化时，主要是从三个方面来进行，优化前后文件读写过程对比如图 4-9 所示。

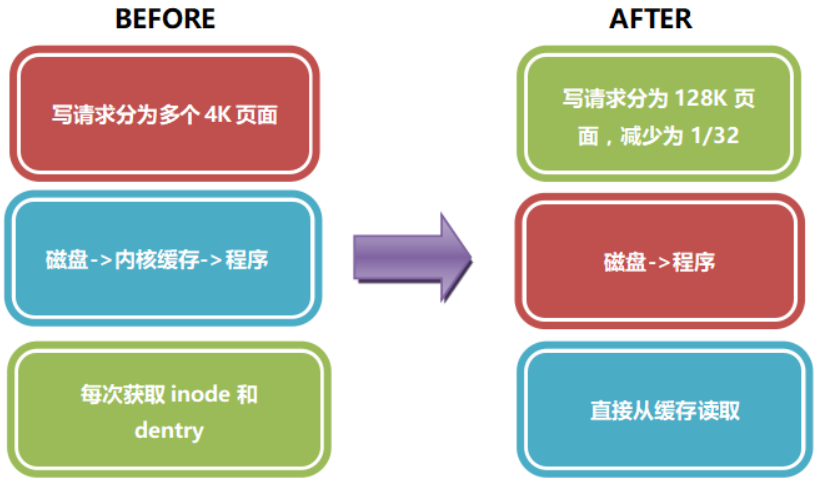


图 4-9 优化前后读写过程对比

图中可以看到，本文主要的优化工作，首先是将写请求分为更大的页面，其次是使

得数据流直接从磁盘到达应用程序，不再经过内核缓存，最后是通过 inode 和 dentry 的缓存减少内核处理时间。接下来对优化原理进行具体分析：

1）减少上下文切换。对 FUSE 性能进行优化时，需要考虑到程序运行过程中比较重要的两个概念，即模式切换（Mode Switch）和上下文切换（Context Switch）[39, 40]。在 Linux 上，几乎所有的程序都可以在两种模式下运行，即内核模式和用户模式。而这里的模式切换就是指从内核模式切换到用户模式，或者是从用户模式切换到内核模式。相对于模式切换，上下文切换指的是 CPU 控制权在不同进程之间的转移，正是由于上下文切换的存在，使得 Linux 操作系统看起来像是在单个 CPU 上同时运行多个进程，并且还相互不干扰。在上下文切换的过程中，一方面要保存交出 CPU 控制权的进程状态，以便后续进行状态恢复，另一方面，还要对获得 CPU 控制权的进程进行状态恢复。同时，在这个过程中，CPU 中的寄存器信息将会被保存然后重新载入新的数据，TLB（Translation Lookaside Buffer）表会重新载入。在实际运行过程中，影响上下文切换效率的因素非常多，包括系统的负载、CPU 的性能等。从上述分析可以看出，相比模式切换，上下文切换带来的资源消耗代价会高很多。而在 FUSE 中，相比普通的 EXT3，FAT32 等内核空间的文件系统，其在运行过程中会有更多的上下文切换。普通的操作系统，在一次写过程中，会产生两次模式切换，即进程会先从用户模式切换为内核模式，然后再从内核模式切换回用户模式，这个过程只会产生模式切换，而不会有上下文切换发生。但是对于基于 FUSE 的用户态文件系统来讲，一次写过程，首先是会包含上述的两次模式切换过程，因为其本身是基于 VFS 的。其次，根据前文的分析，在完成一次读写过程时，首先用户态的应用程序进程会切换到用户态中的 libfuse 进程，通过/dev/fuse 设备写入数据后，再从 libfuse 进程切换到应用程序进程。这样一来，一次写请求会耗费相当多的时间在上下文切换上。而在 FUSE 中，一次写请求往往会被分成多个 page，默认的 page 大小是 4K。事实上，在本文所提出的云存储方案中，根据对其应用场景的分析，所存储的单个文件大小往往远远大于 4K，所以，这会导致上传文件的时候，会有大量的上下文切换，以至于写文件过程既耗时，又占用 CPU 资源。所以，通过修改 Fuse-DFS 的组件源代码，将其单个 page 大小修改为 128K。这样一来，对于相同大小的文件，所需要的上下文切换次数将会减少到原来的 1/32，大大减少了文件写入的时间。这里之所以将其设置在 128K，是因为 Linux 最大只支持 128K 的页面大小，设置为更大，也没有意义。

2）使用直接 I/O。在一些情况下，由于不同应用程序的具体需求不一，有些时候，

应用程序或许需要对自行管理读写过程中产生的缓存信息，而不经过操作系统，也就是直接 I/O 技术。相比于缓存 I/O，数据必须要经过内核缓存来和磁盘进行 I/O 操作，直接 I/O 允许应用程序直接将数据写入到存储设备中，这一点对于诸如数据库系统之类的软件如 Redis、Mongodb 等是至关重要的，因为这些软件基本都有自己的缓存机制。同样的，进行读取操作时，也可以考虑让 FUSE 不直接从磁盘上获取数据，而是将数据缓存在内核中，直接从内核缓冲区内读取数据[41]。此外，在 Linux 内核中，应用程序的读请求，在实际被执行时，往往会有一个预读机制，也就是说，会将所请求资源附近的文件也一起进行读取。这样的设计是为了提高 CPU 中 cache 的命中率，提升性能。

3）缓存文件的 inode 号和 dentry 信息。在 VFS 中，不管是什么类型的文件系统，文件在内存中的描述都是它的 inode 和 dentry 信息。在 Linux 中，打开一个文件的过程，事实上就是要在内存中建立 inode 和 dentry 结构，并让它们和进程关联起来，图 4-10 描述了进程打开文件的过程。而根据前文的分析，FUSE 中每一次请求处理，都需要有文件 inode 号支持。在 FUSE 中，传入的参数事实上是文件名，所以需要进行转化。在这里，FUSE 使用的是 stat 接口。这意味着内核中会大量调用 stat 接口来获取文件的属性，其目的就是通过 stat 接口来获取文件的 inode 和 dentry，通过这两个信息来具体的描述一个文件，而 stat 操作本身是非常耗时的，所以，可以得出的结论是，如果没有对 inode 和 dentry 进行缓存，那么每次对文件和目录的操作都需要经过 libfuse 内核，在一个具体的基于 libfuse 的文件系统中，会使得服务器后端压力大幅增加。

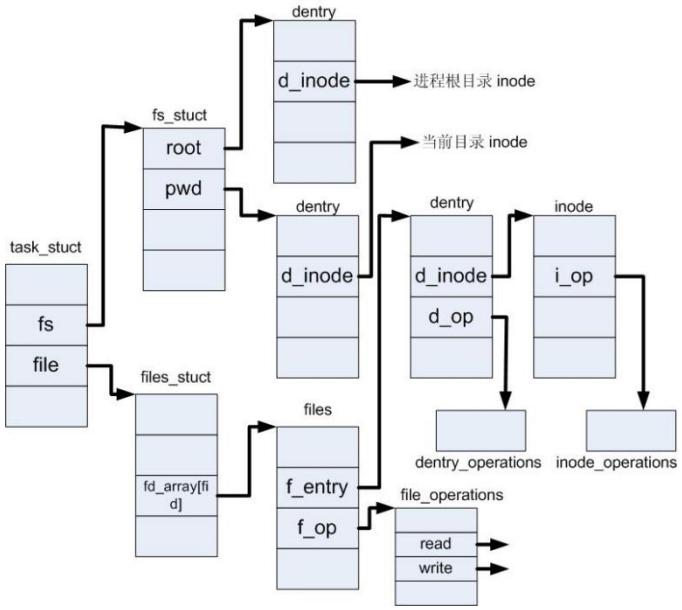


图 4-10 进程打开文件过程

具体来讲，由于客户端执行挂载时的所有参数最后都会通过 fuse_dfs.c 文件中的入

口函数传递给内核，因而，要实现挂载参数选项，需要在挂载时对相关的参数进行指定，并在源代码中通过 FUSE 接口来具体实现。在指定参数以后，通过 libfuse 中提供的 API 函数 fuse_opt_add_arg()来将参数传递给 FUSE 环境。

4.3 FUSE 安全性优化

4.3.1 HDFS 连接过程分析

在前文中已经提到过，FUSE 内核模块会根据 dfs_oper 变量传入的值，来确定内核中的某一个操作所对应的用户空间的函数。本小节会具体介绍回调函数是如何通过 Hadoop 所提供的 C 语言版本 API，即 libhdfs 库来实现对 HDFS 连接以及文件和目录访问与读写操作。由于在 dfs_opr 结构体中定义的多个回调函数中，大体的工作流程是类似的，所以这里以比较典型的 readdir 函数为例，来进行介绍。

readdir 对应的函数实现位于 fuse_impl_readdir.c 文件中，具体定义如下：

```
int dfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset,
                struct fuse_file_info *fi)
```

在该函数中，首先会初始化一个空的 HDFS 连接，在 Fuse-DFS 中，是以 hdfsConn 这个结构体来表示，图 4-11 为该结构体的定义：

```
struct hdfsConn {
    RB_ENTRY(hdfsConn) entry;
    int64_t refcnt;
    char *usrname;
    char *kpath;
    time_t kPathMtime;
    long kPathMtimeNs;
    hdfsFS fs;
    int condemned;
    int expirationCount;
};
```

图 4-11 hdfsConn 结构体定义

在以上结构体中，有几个比较关键的属性。entry 属性是一个自定义的红黑树对象，用于保存在整个组件生存周期中产生的 hdfsConn 对象；refcnt 用于保存连接到该 hdfsConn 对象的数目；kpath 是 HDFS 连接过程中的 Kerbeos 证书缓存路径；fs 是 libhdfs 中定义的 hdfsFS 对象；condemned 为当前连接是否应该被遗弃；expirationCount 表示在丢弃当前连接之前应该执行几次过期验证函数。

在初始化连接之后，会调用 fuse_connect.c 中的 fuseConnectAsThreadUid()函数来初始化 FUSE 环境，以及连接到 HDFS。建立连接之前会通过 hdfsConn 对象中的验证信息来进行身份验证。在 HDFS 连接建立成功以后，会使用 libhdfs 库中的 hdfsConnGetFs()函数获取 fs 对象，获取到该对象以后，就可以开始获取 fs 具体的文件和目录信息了。在 readdir 函数中，使用的是 hdfsListDirectory()返回一个 hdfsFileInfo 对象，在此基础上，通过 fill_stat_structure()函数将该对象中的值赋值给一个 stat 结构体，该 stat 结构体就是 VFS 中可以具体进行操作的对象。至此，从本地建立连接到 HDFS 并获取文件和目录信息最后经由 VFS 返回上层的过程结束。以上过程可以由图 4-12 来表示。

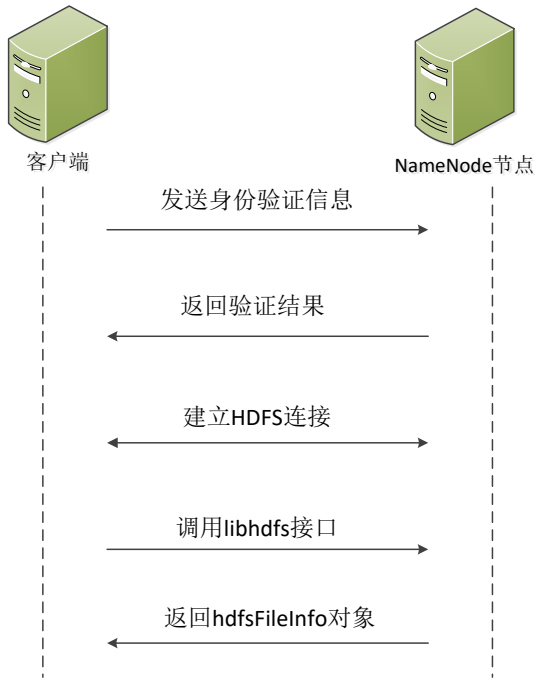


图 4-12 Fuse-DFS 访问 HDFS 过程

可以发现，挂载组件本身是没有 ACL（Access Control List），即访问控制机制的。在一些 FUSE 的应用场景中，较为典型的方案是直接从 Web Service 接口层面来进行访问控制。这种方式直接通过上层的代码逻辑来与数据库或是其它存储设备中保存的用户身份信息比对来进行权限认证，但是这样设计的问题在于无法实现细粒度的访问控制，无法满足云存储系统对安全性的需求。

4.3.2 安全性优化实现

在本文中，想要达到的目的是，挂载 HDFS 的用户和具体的访问用户分离。当前的挂载组件中，其并没有提供这样的实现。也就是说，只要有用户去访问 HDFS 中的数据，

就会被授予所有的文件操作权限。很显然，这是不符合实际需求的。

但是，在上层的 HDFS 本身并没有合适的访问控制方法。试想，如果存在这样一个进程，伪装成 HDFS 的 client，来发起数据访问数据请求，这会对存储在集群中的数据的安全性带来很大的损害。本文重新定义了自己的访问控制流程，其执行过程如图 4-13 所示：

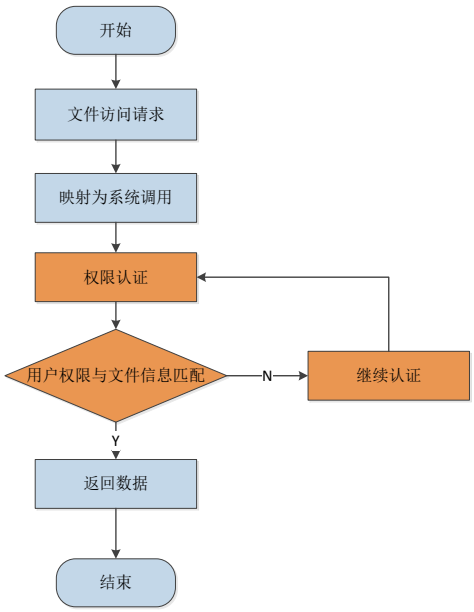


图 4-13 Fuse-DFS 访问控制机制实现

具体实现来讲，首先，在挂载程序开始的时候，获取用户访问信息，保存在如下图所示的一个结构体中：

```
struct passwd {
    char * pw_name; /* Username, POSIX.1 */
    char * pw_passwd; /* Password */
    __uid_t pw_uid; /* User ID, POSIX.1 */
    __gid_t pw_gid; /* Group ID, POSIX.1 */
    char * pw_gecos; /* Real Name or Comment field */
    char * pw_dir; /* Home directory, POSIX.1 */
    char * pw_shell; /* Shell Program, POSIX.1 */
};
```

图 4-14 passwd 结构体定义

在此基础上，本文替换了组件中原有的连接方法，以自定义的连接函数来建立到 HDFS 的连接。函数的定义如下：

```
int hdfsConnectAsUser(host, port, char *user, char *groups[])
```

在这个函数中，会将之前保存下来的用户信息结构体中的用户信息和所属组信息传递给 libhdfs。然后，在进行连接的时候，所使用的则是当前发起访问请求的用户，而非执行挂载程序的用户，这样一来，在每一次进行文件操作时，通过所定义的权限比对机制，对用户的操作进行限制。

本文所使用的访问控制方法，是基于 Linux 的 DAC（Discretionary Access Control），即自主访问控制机制。DAC 机制的核心原理是进程对文件系统的访问权限是与执行该进程的用户保持一致的。为此，首先在 Fuse-DFS 中设计了一个回调函数 getUserAttr()，这个函数会在用户执行 read、write、chmod、chown 等敏感操作时被调用。该函数通过访问 Linux 内核中用于保存用户信息的结构体，获取当前挂载进程的文件访问权限。此外，还定义了 isAccessible()这个回调函数。该函数主要通过 libhdfs 的接口获取文件的属性，然后与前文的 getUserAttr()函数中获取的进程权限信息对比，验证是否有操作文件或者目录的权限。

4.4 性能优化测试

本节所设计的测试方案，重点是针对 Fuse-DFS 源代码优化前后，其读写性能之间的差异进行证明，以进一步验证本文的研究成果。

4.4.1 测试环境

系统测试环境如表 4-1 所示：

表 4-1 软件环境

类型	版本
操作系统	CentOS 6.5
Hadoop 版本	Hadoop 2.6.3

服务器的硬件配置如表 4-2 所示：

表 4-2 硬件信息

CPU	内存	SSD	硬盘
Intel E5-2660	16G	128G	1TB

实验环境结构如图 4-15 所示：

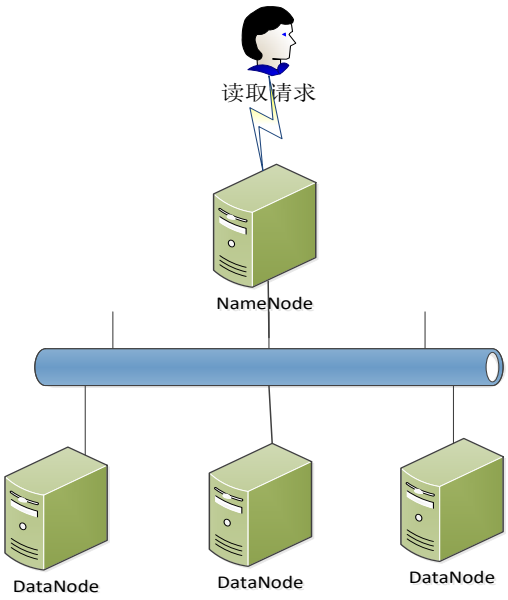


图 4-15 对比实验网络拓扑

4.4.2 测试方案设计

根据常见云数据访问和存储场景，为了模拟实际应用中的 I/O 需求，从多个角度对优化过的方案 I/O 性能进行测试，准确地反映优化前后 Fuse-DFS 组件的实际表现，本文设计了如图 4-16 所示的测试方案。

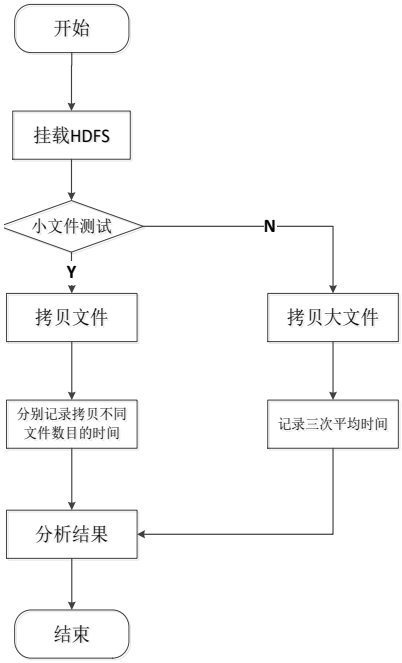


图 4-16 性能测试方案

对数据读写性能测试主要分为小文件创建测试和大文件的拷贝测试。这是由于在 Hadoop 中，小文件的读写是很容易成为性能瓶颈的一个因素，对大文件的拷贝测试，

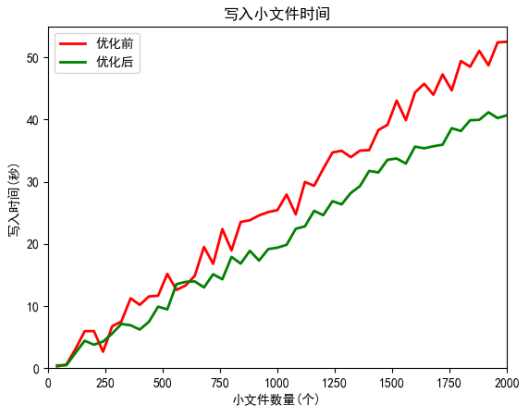
则主要是对系统连续读写的能力进行测试。

在小文件测试中，通过编写的 Python 脚本，在挂载目录中分别写入和读取 40~2000 个 1MB 大小的文件，模拟实际网络环境中常见的文件大小，并通过命令执行开始和结束的时间戳来记录花费的时间。

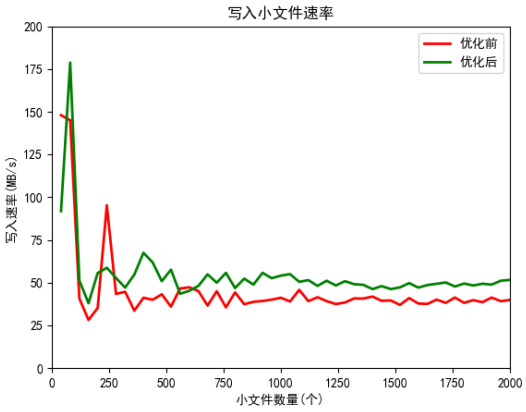
对于大文件的测试，分别记录了 1GB~5GB 的文件拷贝时间，共选择了五种不同大小的文件（每次增加 1GB）。对于每一组实验，重复三次，对其求平均值，以使得数据更具有一般性。

4.4.3 测试结果

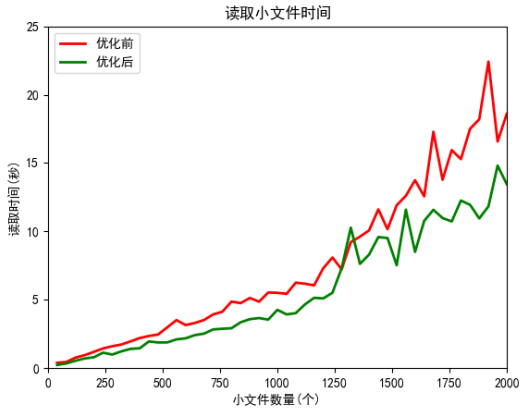
如图 4-17 所示，为使用一个进程通过 FUSE 向 HDFS 读取或写入不同数目小文件所耗费的时间和单位时间的速率，其优化前后的实验结果如下：



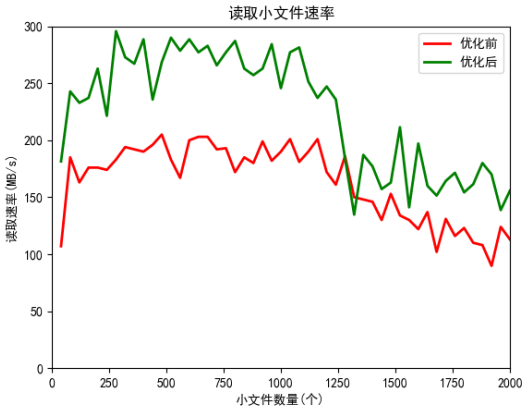
a)写入小文件时间对比



b)写入小文件速率对比



c)读取小文件时间对比



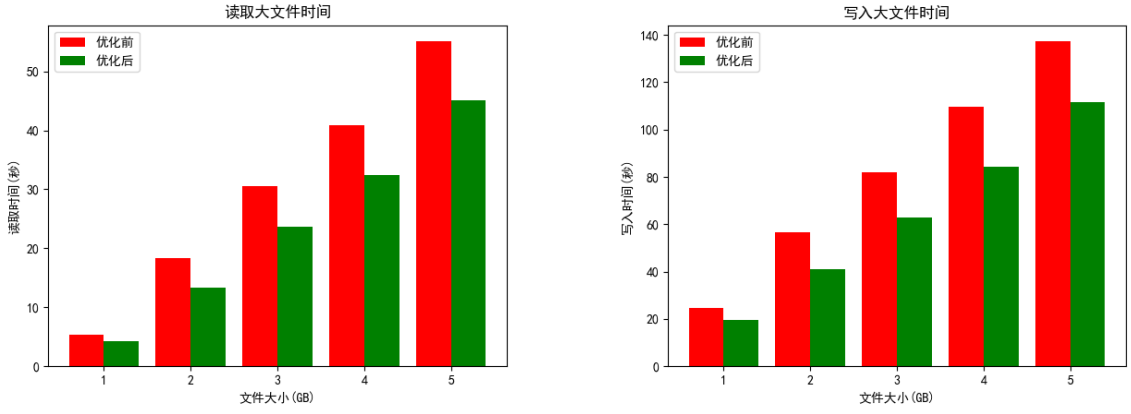
d)读取小文件速率对比

图 4-17 优化前后小文件读写对比

首先从图 a 和图 b 两幅图可以看出，在优化前后，写相同文件的时间有明显的缩短，相应的，其读写速度有较大提升。从图 a 中可以看出，随着小文件数目的增多，优化前

后的时间差由逐渐增大的趋势。结合前文分析可知，由于在扩大了内核读取文件的页面大小之后，单个文件写请求，所会产生的上下文切换时间会大幅减少，这说明本文的分析与实际情况吻合。此外，由于本文所研究的云存储技术方案，面向的是超大规模的数据存储，因而，本文的优化，应用到实际场合中，在用户上传数量规模较大，而单个文件大小较小的文件如图片，文档等类型文件到 HDFS 中时，其所能体现出来的优势将更加明显。观察图 b 可以看到，在一开始，文件读写的速率有较大的波动，而随着文件数量的增多，其写速率逐渐趋向稳定。分析波动的原因，推测是由于在测试时，使用的是 Linux 原生的 dd 命令来生成文件和写入文件，而该命令，本身并没有考虑到内存缓存因素的影响，所以可能在某一瞬间，写速率会有较大的波动。

图 c 和图 d 反映了优化前后读取相同数目小文件的时间和读取速率之间的差异。从总体数据上来看，读取小文件的性能提升同样非常明显。从图 c 中可以看到，类似于写入文件的操作，读取文件在优化前后的时间差异随着文件规模的增大，也在不断扩大。从图 d 中可以看到，写入文件的速率在一开始处于 100MB/s 左右，但是到了中期，其速度开始提升。而随着文件规模的进一步扩大，文件读取速率又有了一定的下降，并趋于稳定。经过分析，一开始，由于内存和磁盘中都没有相关数据的缓存，因而其速度较能反映真实读取速率。随后，在一段时间的文件读写之后，开始有部分文件的缓存，因而其速度有较大提升。而当文件规模过大以后，缓存所带来的影响开始被弱化，导致其速度开始下降，并趋于稳定。



a)读取大文件时间对比　　b)写入大文件时间对比

图 4-18 优化前后大文件读写对比

对于大文件的读写，主要是为了反映 FUSE 向 HDFS 中连续读取文件的性能特点。从图 4-18 子图 a 中可以看到的是，随着文件大小的从 1GB 增长到 5GB，优化后读取相

同大小文件的时间明显低于优化前，这证明，当用户从云存储系统中下载单个大文件时，其耗费时间会明显降低。图 b 反映的是随着文件大小的增长，写入单个大文件的时间变化。优化后的写入时间同样有着明显降低，这证明，当用户向云存储系统中上传大文件时，所耗费的时长会减少。横向对比两幅图，可以发现，读取文件的时间会比写入相同大小文件的时间低很多，这是由于写入操作需要先将磁盘现有的数据抹除，再进行数据复制，而读取操作则省去了这个步骤。总体来看，本文的优化使得读写性能提升了大约 20%。

4.5 本章小结

本章主要是针对 FUSE 访问 HDFS 来进行优化，分别从读写性能和安全性方面入手。首先对云存储数据访问的优化方案进行了设计，然后分别针对两个不同的方面，进行具体的实现。在此基础上，本文设计了一系列的实验来对比优化前后 FUSE 的读写性能，证明了本文的优化带来的实际性能提升。最后，对本章的工作进行了总结。

第五章 基于 FUSE 的云存储技术应用

基于本文第二章确定的云存储系统需求和设计时应该遵循的基本原则，结合本文对现有技术所进行的优化，本章具体的实现了一个云存储系统，并应用到了腾讯 TEG 的大数据分析平台中，该平台主要是通过收集海量 APP 的运营数据，为开发者提供实时的数据统计与分析功能，以达到个性化、精细化的运营目标。需要注意的是，本文的描述为实际应用系统的一个原型，实际应用过程中在软硬件环境上有一定的差异。其中，5.1 小节介绍了系统的总体架构；5.2 小节描述了数据存储层的设计，介绍了后端 HDFS 集群的设计以及轮询挂载的实现；5.3 小节对数据访问层的实现进行了说明，主要包括负载均衡设计以及文件访问 API 的实现；5.4 小节对云存储方案中涉及到的监测组件进行了实现；5.5 小节介绍了日志管理组件的实现；5.6 小节为本章小结。

5.1 云存储技术方案总体架构

依照本文第二章中已经确定的云存储技术需求和需要满足的设计原则，对基于 FUSE 的云存储技术方案基本框架设计如图 5-1 所示。可以看到的是，该方案大体可以分为四个层次，即用户访问层，数据服务层，数据访问层，数据存储层。其中，每一层又细分为多个功能模块，这样的好处是使得各模块功能明确，符合高内聚，低耦合的软件工程基本原则[42]。此外，对功能模块的划分还方便了开发过程，使得每阶段的任务非常明确，提升了开发效率。

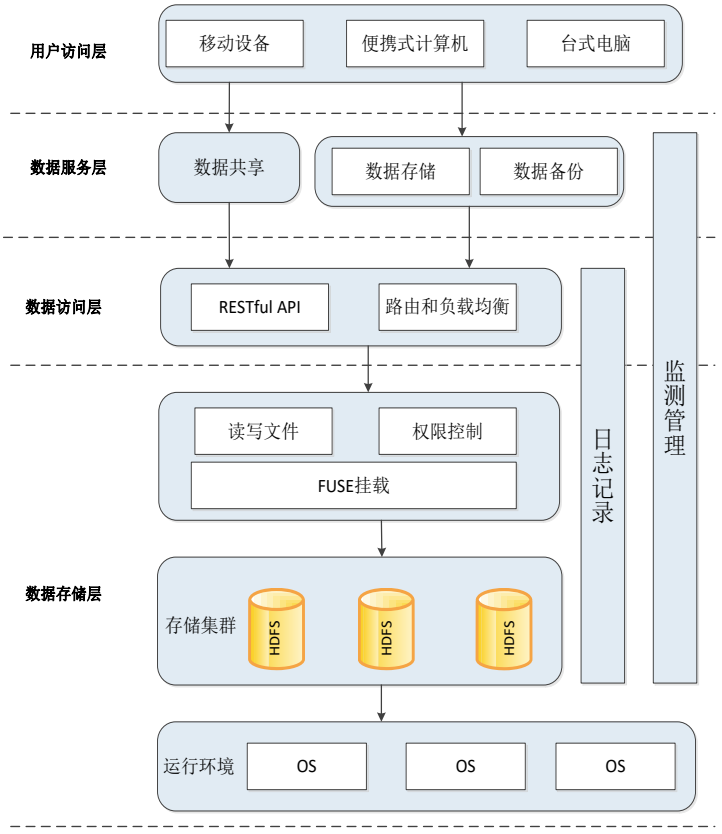


图 5-1 云存储技术方案总体框架设计

接下来对每一层的具体功能和结构进行说明。

- 在用户访问层中，普通用户可以通过云存储系统向外提供的一系列数据访问接口或是 Web 界面等来操作云存储系统中的数据。同时支持多种设备的访问，用户可以使用手机、桌面电脑、便携式计算机等来进行访问。
- 在数据服务层中，主要是用于直接与用户交互，并且具有较强的扩展能力。通过这一层次，用户可以进行数据存储，数据共享或是对本地数据进行备份的操作。
- 在数据访问层中，主要是对用户的访问进行管理，基于 REST 软件风格的设计原则，对外提供一个 REST 风格的数据访问接口，对用户所需要的各种不同服务进行映射。同时，还需要实现大规模并发数据访问请求的调度。这一层，与上层的数据服务层之间有着无缝的连接，同时，也衔接着下层的存储设备。它同样也促进了下层存储设备之间的协作，使得下层设备对外提供更多样和更优的服务。
- 在数据存储层中，存在着多种多样的存储设备，存储在设备中的数据规模庞大。为了对这些数据进行高效的管理，必须对其进行合理的组织。传统的方式使用

少量的本地服务器，但这并不能满足当前云存储平台对于吞吐量和存储容量的要求。经过调研发现，基于 P2P（Point to Point）即点对点的体系结构的组织方法对节点数量的要求很高，并且需要很复杂的算法来保证数据的可靠性[43]。与之相比，使用多台存储服务器能够满足大规模数据的存储的需求，同时，也更加容易实现和进行管理。这一层中采用基于 HDFS 的分布式存储方式，能够提供更好的 QoS（Quality of Service）即服务质量，尤其是在大规模用户的使用场景中。使用优化过的 HDFS 挂载组件，通过 FUSE 来访问 HDFS，将整个存储集群中不同的网络存储设备全部组织起来进行集中管理和状态监测，以及动态地进行容量调整。

本章后文将对本方案中的各个层次和模块进行具体实现，重点是数据访问层和数据存储层，以及日志管理组件和监测组件。

5.2 数据存储层的实现

5.2.1 HDFS 集群高可用设计

在早期的 Hadoop 版本中，由于高可用的设计还不完善，因而 NameNode 中的数据可能会存在着不一致的问题，即 SPOF（Single Point of Failure）问题，也就是单点故障导致的集群中节点数据不同步的问题[44]。前文已经介绍过，由于 NameNode 是整个 Hadoop 集群中保存着元数据信息的节点，如果 NameNode 无法正常工作，那么 HDFS 上的数据都将无法正常访问。并且，早期的 HDFS 恢复过程也比较繁琐，效率较低，这在生产环境中是不可接受的。对于本方案的设计来讲，更加如此。因而，本方案为了提升 HDFS 集群的高可用性，进行了针对性的设计。这里主要介绍的是在 HDFS 服务端的设计。

通过设计主备 NameNode，在集群中同时运行着多个 NameNode，其数据保持一致。一般情况下，只有一个处于激活状态的 NameNode 向外提供服务，而剩下来的 NameNode 处于 Standby 状态。当主 NameNode 节点出现问题时，就会切换为激活状态，代替之前的节点，用于处理数据访问请求。图 5-2 描述了一个基本的高可用 NameNode 节点设计。

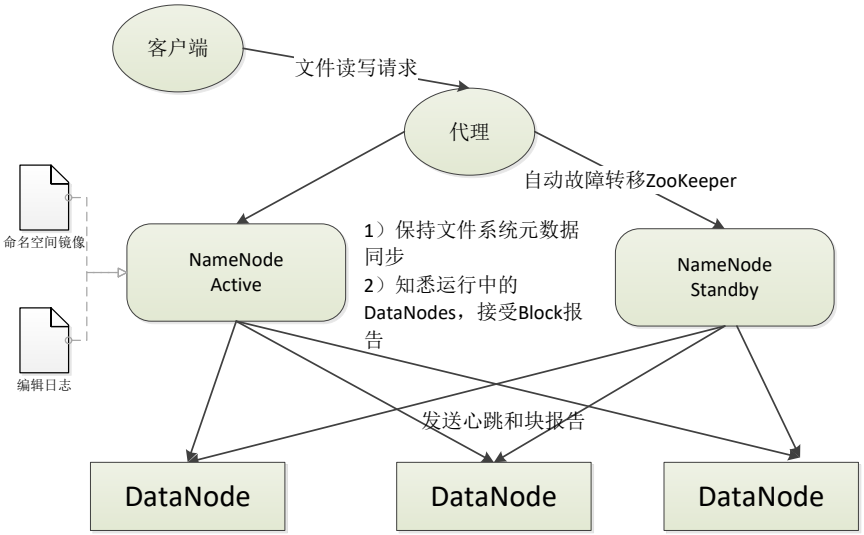


图 5-2 NameNode 的服务端高可用设计

如图 5-2 所示，客户端的的访问请求先到达代理服务模块，然后转发到当前向外提供服务的节点。主备 NameNode 之间通过代理来保持通信，实现保存 NameNode 中所有信息的两个文件之间的完全同步。这里，必须保证多个 NameNode 上的数据完全一致，否则会导致 NameNode 和 DataNode 之间的交互出现问题，DataNode 上的块数据信息可能会丢失。

此外，除了在 HDFS 服务端中使用到了高可用的设计之外，本文同样对 HDFS 进行挂载的组件进行了高可用的设计，这个过程在 5.2.3 节会进行介绍。

5.2.2 HDFS 集群实现

上一节对 HDFS 高可用集群的设计原理进行了描述，这里进行具体的实现。

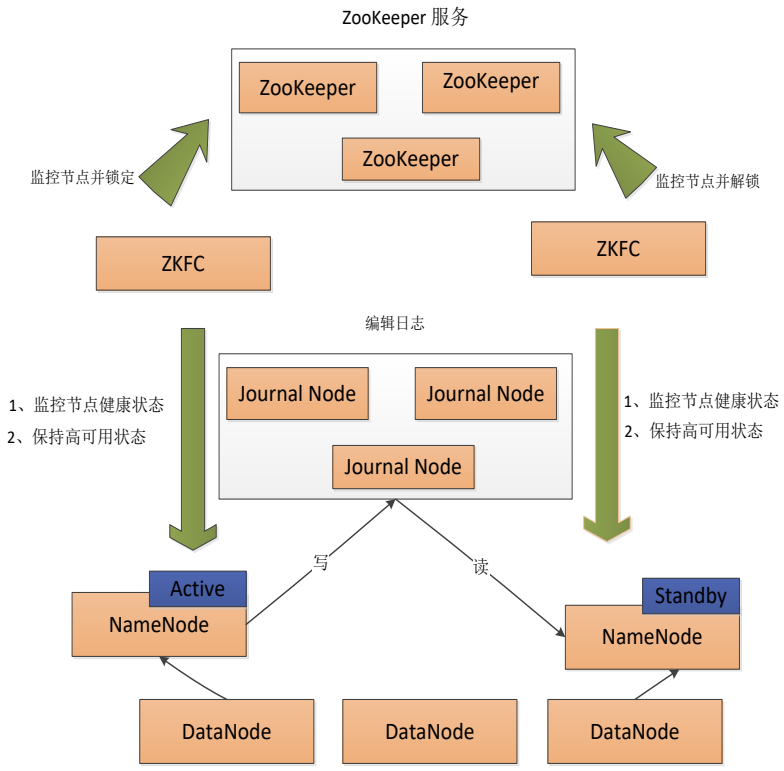


图 5-3 高可用的 HDFS 集群

如图 5-3 所示，集群中有两个 NameNode。但是，只有一个是处于激活状态。激活状态的节点用户处理来自客户端访问请求，而备用节点则是用于保持服务器中始终有充足的资源以便在主节点出现问题时，迅速进行切换。主备节点需要保持数据一致，以保证在发生切换时不会有数据丢失。主 NameNode 节点发生错误以后，Hadoop 中的防护进程将负责断开该节点对共享存储目录的访问。这样一来，就可以保证它无法对命名空间进行任何进一步的编辑，从而使得新的 NameNode 节点可以接管它的工作，开始向外提供服务。

在 Hadoop 的 HA 集群中，主节点和备用节点之间通过 ZooKeeper 来进行协调工作。ZooKeeper 具有高可用性和可靠性，其优势在于反应快速、简单有效、适用场景丰富。在分布式系统中，ZooKeeper 的作用是用来存储和更新密钥信息，并进行领导者选举。需要注意的是，这里的领导者就是具体用来提供服务的节点。通过分析业务方对于存储环境在可用性和吞吐量方面的需求，在 ZooKeeper 中，主节点的选择算法如下所示。

算法 领导者选举算法

输入： ZooKeeper 中的多个节点

输出： 提供服务的节点

```
1:  发起投票，并选举自己为 leader
2:  while 状态为 LOOKING do
3:      从接受到的选票队列中选出一张
4:      if 状态为 LOOKING then
5:          if 逻辑时钟大于目前的逻辑时钟 then
6:              goto tag1
7:          endif
8:          if 处于上轮投票 then
9:              continue
10:         elseif 处于本轮：
11:             如果更优则更新投票，
12:         endif
13:         tag1:
14:             将选票放入投票箱
15:     elseif 状态为 FOLLOWING 或 LEADING
16:         if 处于同一轮 then
17:             保存到投票箱
18:             if 当前投票者得到大多数选票 then
19:                 break
20:             else
21:                 设置 outofelection
22:             endif
23:     elseif 状态为 OBSERVING:
24:         直接退出
25:     endif
26:     return 领导者节点
```

在选举过程中发送的消息包含四个类别：服务器 ID，特点是越大其权重越大；数据 ID，标识其新旧程度；逻辑时钟，可以理解为投票的次数，但是并不是以每次加一的方式来表达。选举状态分为 LOOKING（竞选状态），FOLLOWING（随从状态，参与投

票），OBSERVING（观察状态，不参与投票），LEADING（领导者状态）。

为了更清楚地说明该算法的运行过程，假设集群中的启动节点分别为 A、B、C，图 5-4 描述了具体的主节点选择过程。

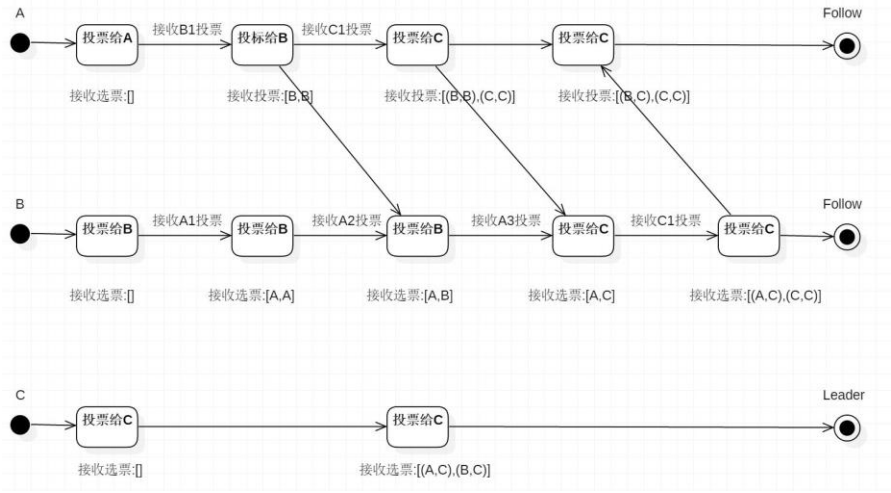


图 5-4 领导者选举过程状态迁移

ZooKeeper 中，还有一个新的组件，名为 ZKFC（ZKFailoverController），它同样对 NameNode 的状态进行监控和管理。集群中每一个运行着 NameNode 的节点同样也运行着 ZKFC，ZKFC 主要有以下几个职责：1）NameNode 节点健康状态监控；2）ZooKeeper 中的会话管理；3）基于 ZooKeeper 来管理 Active 节点。

在本文中，使用了四台服务器来进行实验，其中，两台 NameNode 互为主备，另外两台服务器作为 DataNode 来存储数据。集群中的节点规划如表 5-1 所示：

表 5-1 集群规划

IP 地址	主机名称	角色
192.168.1.200	master1	主节点 NameNode，job-history-server
192.168.1.202	master2	主节点 NameNode，job-history-server
192.168.1.201	slave1	数据节点 DataNode
192.168.1.203	slave2	数据节点 DataNode

另外，所有的节点都同时是 ZooKeeper，ZKFC，JournalNode 节点。

1）首先，为了方便集群中的节点进行通信，以及相互之间能够较为方便地进行文件传输，需要配置 ssh 免密码登录。一般来讲，Linux 上使用 rsa 算法比较多，这里进行如下操作。

首先生成 key：

```
ssh-keygen －t rsa －C testmail@test.com
```

接下来再拷贝到其它节点：

```
ssh-copy-id master1@192.168.1.200
ssh-copy-id master2@192.168.1.201
ssh-copy-id slave1@192.168.1.202
ssh-copy-id slave2@192.168.1.203
```

2）对环境变量进行配置，方便 Hadoop 命令的执行。

Hadoop 所在路径：

```
export HADOOP_HOME=/usr/local/Hadoop
```

ZooKeeper 路径：

```
export ZOOKEEPER_HOME=/usr/local/zookeeper
export
PATH=$ZOOKEEPER_HOME/bin:$HADOOP_HOME/sbin:$HADOOP_HOME/bin:
$JAVA_HOME/bin:$JRE_HOME/bin:$MAVEN_HOME/bin:$FINDBUGS_HOME/bi
n:$ANT_HOME/bin:$PATH
```

3）配置 Hadoop 的参数，重点是配置$HADOOP_HOME/etc/hadoop 下的四个文件。

表 5-2 hdfs-site.xml 文件配置

name	value	备注
fs.defaultFS	hdfs://ns	NameNode 的 URI
hadoop.tmp.dir	file:/usr/local/hadoop/tmp	Hadoop 数据存储目录
ha.zookeeper.quorum	master1:2181,master2:2181,slave1:2181,slave2:218	ZooKeeper 的地址

表 5-3 core-site.xml 文件配置

name	value	备注
dfs.nameservices	ns	命名空间的逻辑名称
dfs.ha.namenodes.ns	nn1,nn2	标识 NameNode
dfs.namenode.rpc-address.ns.nn1	master1:9000	nn 的地址
dfs.namenode.shared.edits.dir	/usr/local/hadoop/journal	存放 editlog 和其他状态信息
dfs.ha.fencing.methods	sshfence	一种隔离机制

表 5-4 yarn-site.xml 文件配置

name	value	备注
yarn.resourcemanager.hostname	master1	yarn 所在的主机
yarn.nodemanager.aux-services	mapreduce_shuffle	——

表 5-5 mapreduce-site.xml 文件配置

name	value	备注
mapreduce.framework.nameger.hostname	yarn	——
mapreduce.jobhistory.address	master1:10020	jobhistory 地址
mapreduce.jobhistory.webapp.address	master1:19888	网络访问地址

5.2.3 轮询挂载实现

在上文的描述中，可以看到，在建立 HDFS 连接过程中，Fuse-DFS 仅考虑的是一个 NameNode 节点的情况，其在调用 libhdfs 库中的接口时，并没有针对多个 NameNode 节点的情况做考虑，因而，如果刚好连接的是一个在 HA 模式中处于 Standby 状态的节点时，就会导致连接失败。事实上，这种情况是因为设计之初没有特意针对高可用集群进行优化而带来的的问题，因而，如何针对高可用集群进行优化，是一个亟待解决的问题。

由于在所提出方案中的数据存储层，使用的是 Hadoop 的开源组件 Fuse-DFS。Fuse-DFS 本身并没有任何高可用性，而本方案的设计是基于高可用的 HDFS 集群来实现的，也就是说，在 Hadoop 中的 NameNode 出现了 Active 到 Standby 状态的切换时，Fuse-DFS 本身并无法检测到，从而当其继续保持原有的 HDFS 连接时，会出现挂载失败的问题。为了解决这一问题，本文提出了一种方案：将所有可以用于挂载的 Hadoop 的 NameNode 加入一个队列中，定期的去检测当前 HDFS 的挂载状态，当检测到挂载失败的时候，以轮询的方式来遍历队列中的所有 NameNode，重新执行挂载，如果挂载成

功，则停止遍历，如果挂载失败，则继续尝试连接其他节点。具体的实现方式如图 5-5 所示。

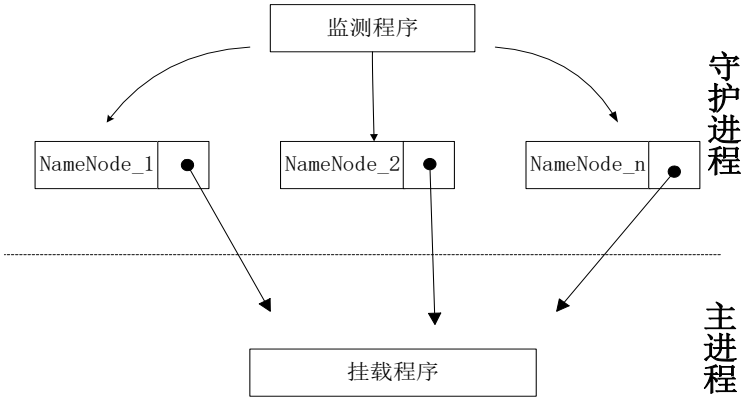


图 5-5 轮询挂载实现

这里实现了一个守护进程来运行监测程序，之所以要使用守护进程，是因为监测程序需要定期执行。在 Linux 上，如果使用其本身自带的 Crontab 机制来实现定时任务，存在较多限制，其可靠性难以保证。因而，这里使用 Linux 上的守护进程，实现了监测程序可以定期执行，并且运行在后台，不占用 Linux 终端。

5.3 数据访问层的实现

5.3.1 负载均衡集群设计

在云存储技术的基本需求中，具有对复杂网络环境下的高可用性和高可伸缩性的要求。负载均衡技术一般工作在 OSI（Open System Interconnection）[45]中的网络四层，它实现了对数据访问请求进行合理的分发。本文在进行数据访问请求调度时，使用了自行设计的动态调整的负载均衡算法，在性能上比常规的算法表现更好。

如图 5-6 所示，集群的工作原理是在前端由一个虚拟的负载均衡调度器来接收用户来自因特网的请求，然后将请求发送至通过 LAN（本地网络）或是 WAN（广域网络）连接起来的多台用于提供服务的服务器，调度器才完成了真正处理请求的工作。

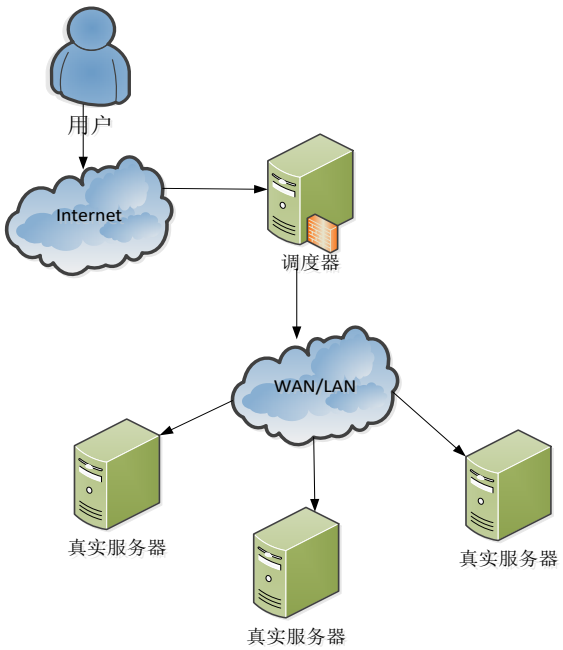


图 5-6 负载均衡集群设计

5.3.2 数据访问 API 的设计

在设计数据访问 API 的时候，基于 RESTful 的设计风格，将存储集群上的文件以资源的形式来对外提供服务。在实现过程中，每一个资源都通过 URI（Universal Resource Identifier）来进行表示。在发起数据访问请求的客户端和提供服务的存储集群中间，通过 HTTP 中的 GET，PUT，POST 和 DELETE 方法来传递客户端的具体请求。具体实现中，通过 Flask 框架来设计后台处理模块监听在 80 端口向外提供服务。

这里规定用户的的访问请求在发送到服务端是必须是标准的 REST 模式，以方便进行管理。下表描述了具体的 API 对应到文件操作的映射关系。

表 5-6 RESTful API 与文件操作的映射

来自用户的请求	使用的 HTTP 方法	文件 URI
浏览文件列表	GET	/tmp/fuse/files/
上传文件	POST	/tmp/fuse/files/filename
下载文件	GET	/tmp/fuse/files/filename
删除文件	DELETE	/tmp/fuse/files/filename
重命名文件	PUT	/tmp/fuse/files/filename

5.4 监测组件的设计与实现

在具体的应用过程中发现，大规模集群中机器数量非常大，因而出现故障的可能性也非常大，想要人工来进行监测几乎不可能。并且，某个节点出现问题时，并不一定是硬件的故障，也可能是软件运行错误。为了解决这一问题，本文设计了监测组件，来对集群中节点的软硬件进行监测。本节后续内容对这个组件进行详细说明。

5.4.1 整体架构

大规模分布式集群在运行过程中面临的一个重大挑战就是如何实时的获取集群中节点的运行状态。在一个大型云存储系统中，I/O 和网络负载都会变得尤其庞大。对于单个存储节点来讲故障或许是小概率事件，但是放大到整个存储集群，往往就会变得司空见惯。因而，监测组件必须能够准确地记录和识别系统故障，以便节点中的硬件以及应用程序可以使用自带或者外部的方法来进行恢复。并且，在存储集群容量不断增长的过程之中，监测组件可以对集群的整体规划带来极大的帮助。

为了解决分布式存储集群信息采集中存在的一系列问题，本文设计了更加契合实际场景的监测组件。本文所设计的监测组件主要分为三个模块，client 端，server 端，view 端。在这里，client 端具体负责从存储集群中的各个节点来获取各种运行指标，不限于硬件信息，也包括节点上应用程序的运行信息。此外，client 端还负责将采集到的信息发送到统一的数据处理接口 server 端。server 端负责的是接收来自不同节点的数据并进行实时分析后进行持久化存储。view 端在这里主要给监测组件提供展示以及控制功能。整体的框架如图 5-7 所示。

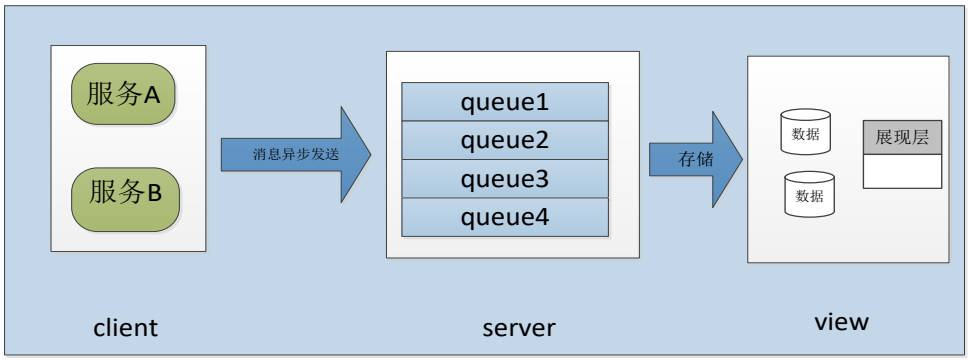


图 5-7 监测组件整体架构

监测组件的部署以提高系统稳定性和减少资源消耗为基本原则。上图中可以看到 client 端部署在每一个注册到存储集群的节点中，通过在本地进行自定义的配置来采集

不同的信息，而 server 端和 view 端部署到统一的服务集群中。这里采用了集群化的设计思路，主要是监测服务高可用性方面的考虑。后文具体描述了较为核心的 client 端和 server 端的设计，对于展现层面的实现，则不再赘述。

5.4.2 client 端的设计

在监测组件 client 端的实现中，使用到的是 ThreadLocal（线程本地）变量的方式。由于不同的节点中需要监测的业务信息往往很多，并且可能随时会存在加入新的监测需求的可能。因而，使用 ThreadLocal 将所有的监测请求都放到一个上下文环境变量中，这样，线程池中的每一个监测线程都拥有该变量的一个副本，并且可以随意修改。client 端基本的架构设计如图 5-8 所示。

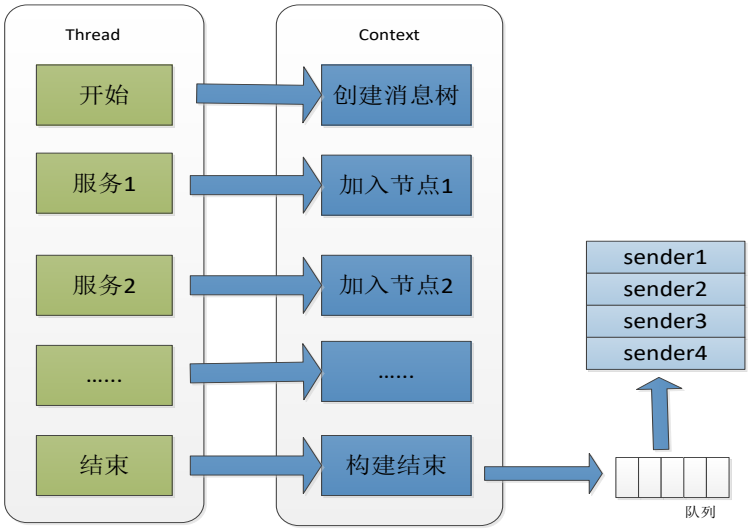


图 5-8 client 端架构

client 端采集到数据之后，需要将信息发送到 server 端进行处理。由于要进行网络传输，为了让接收端能够正确的理解并解析接收到的数据，这里对发送的数据进行了序列化的处理。

client 端数据的发送还考虑到了监测节点本身可能存在的 I/O 问题。client 端的数据发送时，对数据的序列化操作是异步进行的，以免由于序列化本身耗时太多影响节点本身的正常运转。在数据序列化完成以后，考虑到可能出现的网络抖动，数据的发送过程以异步线程的方式来实现。

5.4.3 server 端的设计

server 端使用到了 RabbitMQ 这个开源的中间件，来接收来自各节点的数据。使用

RabbitMQ 实现了完全的异步处理，将出现故障的可能性降到了最低。

server 端中以 producer 和 consumer 来实现数据的接收和具体处理。其大致流程如下：

1）接收来自 client 端上的 producer 产生的数据，放入内存队列中；

2）针对每一个接收到的消息，consumer 都会以并发的方式启动一个线程来进行处理，具体是以回调函数的方式来实现，并且，每一个线程之间互不干扰，以达到消息之间的隔离；

3）consumer 在对数据进行分析以后，会以异步的方式来将数据持久化存储到数据库中，这里使用的是 MongoDB。

server 端的工作过程如图 5-9 所示：

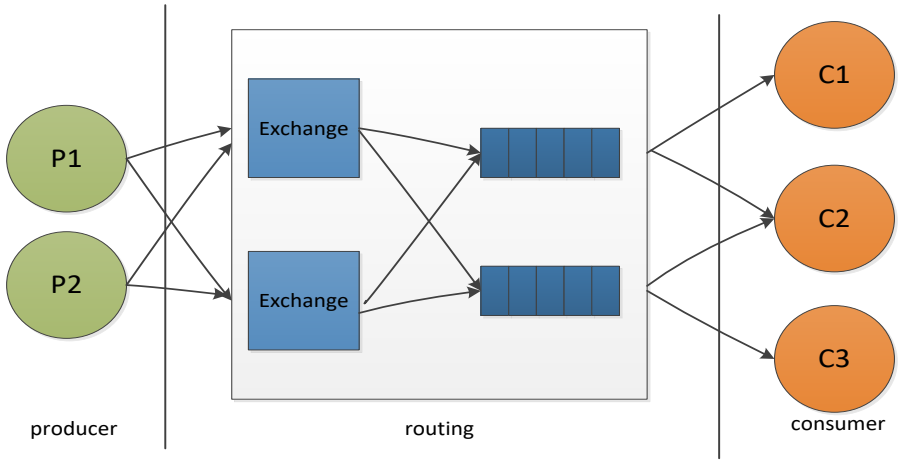


图 5-9 server 端架构

在 server 端中，数据需要通过 exchange 这个类似于交换机的组件来进行具体的队列匹配。其使用到的消息路由算法如下：

算法 consumer 中的消息路由算法

输入：routing keys 数组 routingArray，n 为数组长度，匹配模式 pattern

输出：routing key 与其所绑定队列的 map

```
1:  function SearchTrie(trie, key, m)
2:      res ← false
3:      i ← 0
4:      j ← 0
5:      while j < m and i < len(key) do
6:          if key[i] = trie[j]:
```

```
7:                  i++
8:                  j++
9:            elseif key[i]匹配* or key[i]匹配#
10:                   i++
11:             end if
12:       end while
13:       if i = len(key) then
14:          res ← true
15:       return res
16:   end function
17:   function BindQueue(routingArray, pattern, n)
18:         i ← 0
19:         resArray ← {}
20:         if pattern  只包含#,*以及字母  then
21:              生成 trie 树 node_trie
22:         end if
23:         while i < n do
24:              if SearchTrie(node_trie, routingArray[i]) then
25:              resArray[key]=pattern
26:       end while
27:       return resArray
28   end function
```

在 consumer 收到一个消息的队列绑定请求之后，首先会检测用于匹配的 pattern 是否符合只包含“#”或“*”或是字母，如果是那么会根据这个 pattern 生成一个字典树 trie_node，在具体的实现中以链表替代以简化处理过程。这里以“*.abc.xyz.#.end”来说明具体过程。首先会生成如图 5-10 所示的一个链表：

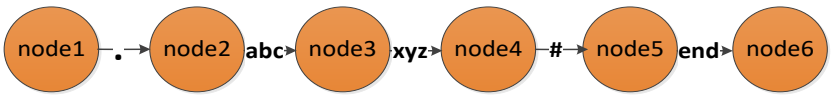


图 5-10 生成的链表

这里尝试以“test.abc.xyz.123.456.end”进行匹配，首先程序会搜寻 node1 到 node2 有没有直达的路径，发现没有，于是以“*”进行匹配，成功。而从 node2 到 node3 有直达路径。node3 到 node4 同样有“xyz”这一条直达路径。node4 到 node5 通过“#”匹配成功。node5 和 node6 之间会跳过“456”，直接匹配最后的“end”。总体过程如图 5-11 所示。

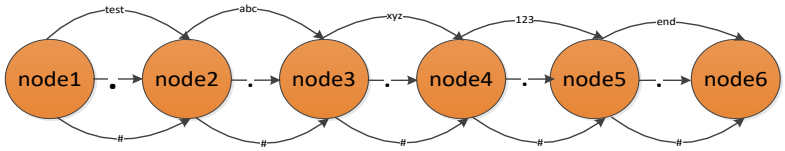


图 5-11 具体实例

5.5 日志管理组件的设计与实现

由于存储集群中运行着大量服务和软件，因而每天都会产生大量的日志。这些日志位置不一，当出现故障的时候，如果想要排查原因，必须要从服务器不同的目录下进行查找，然后来进行分析。除此之外，每个应用程序的日志格式不一，寻找故障原因非常麻烦，于是，本文设计了统一的日志服务器，通过定义统一的日志模板来对日志进行管理。

日志管理组件的作用是对平台在运行过程中出现的一些问题进行记录，并持久化到文件中，同时对这些文件定期的进行备份和清理。在本系统中，所有的日志记录都是通过 Linux 的 syslog 内核模块来实现的。所涉及到的各种软件或是工具，都设置了将日志保存到本地的相应目录下。日志管理的过程如图 5-12 所示：

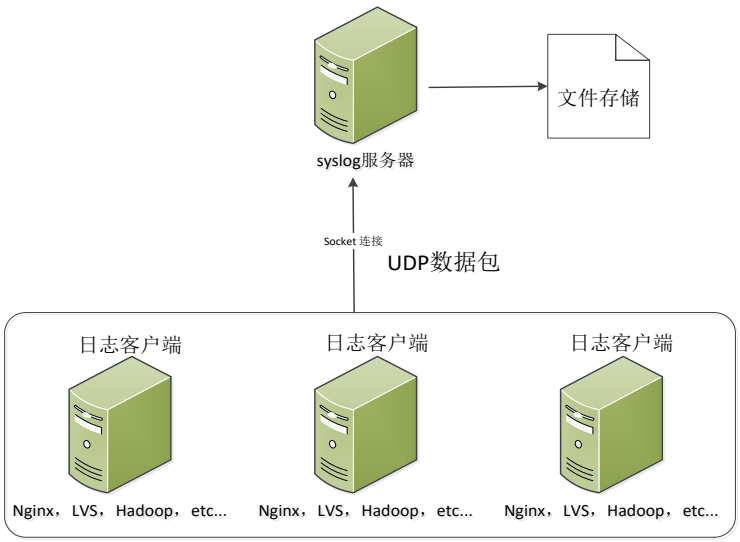


图 5-12 日志管理过程

在图中可以看到，服务器 192.168.1.206 作为一个日志服务器，使用 syslog 来接收来自集群中其它软件运行过程中产生的日志，主要包括 Nginx、LVS、Hadoop 等。在这里，通过使用 UDP 协议，基于 socket 通信方式实现了一个用于发送日志的 agent，部署在整个集群中，同时，针对这个 agent 提供了一键安装和部署脚本，以方便 agent 在整个集群中的迁移和部署。

5.5.1 日志管理组件的设计

在集群中，所有软件的日志都设置了通过 syslog 发送到本地目录中的文件，并且，依照 syslog 已经有的一些定义，对日志的格式进行了统一，具体的实现方式如下所示。

1）首先，定义应用程序的日志模板：

```
$template my_format,"%TIMESTAMP% host=%host%,
                                    relayHost=%fromhost%,
                                    tag=%syslogtag%
                                    programName=%programname%,
                                    facility=%syslogfacility-text%,
                                    sev=%syslogseverity-text%,
                                    appName=%app-name%,
                                    msg=%msg%\n"
$ActionFileDefaultTemplate my_format
```

在上述的定义中，各个配置项所代表的含义如表 5-7 所示：

表 5-7 自定义日志模板

配置项	含义
hostname	消息中带有的主机名
syslogtag	消息标记
programname	程序名称
syslogfacility-text	文本形式的设备名
syslogseverity-text	日志的严重程度
app-name	syslog 协议中的 app-name 字段
msg	日志的具体信息

2）重新启动 syslog 服务

```
service rsyslog restart
```

为了方便对整个系统中的日志进行统一管理，参照 syslog 中的定义，这里重新定义了日志级别，主要如表 5-8 所示：

表 5-8 自定义日志级别

日志级别	说明	ID	日志文件来源
DEBUG	调试信息	1	系统运行时日志
WARN	告警信息	2	系统运行时日志
INFO	通知信息	3	系统运行时日志
ERROR	错误信息	4	系统运行时日志

每一条日志记录在日志发送线程都会被重新处理，并在发送报文中加上相应的日志标记。经过处理以后的日志报文如图 5-13 所示。

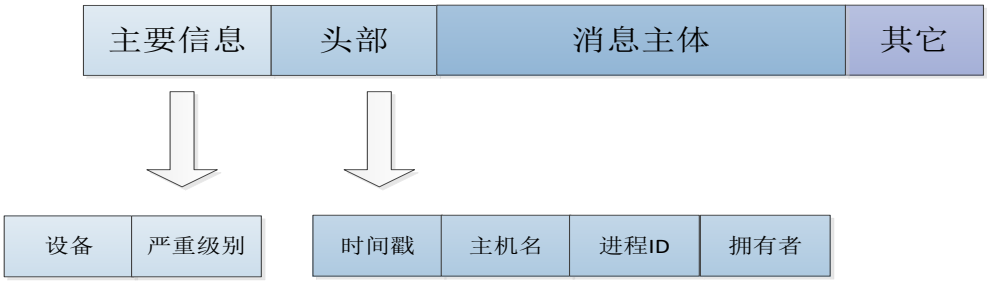


图 5-13 日志报文结构

如图 5-13 所示，通过日志发送 agent 发送到 syslog 服务器的消息主要由四个部分组成：日志记录的主要信息，即 Linux 中的运行设备和日志的严重级别；头部信息，包括日志的时间戳，主机名称，产生记录的进程 ID，进程的拥有者；消息主体，即由应用程序产生的信息；其它信息，根据不同的应用程序的运行场景，可能需要发送的附加信息。

5.5.2 日志管理组件的运行过程

如图 5-14 所示，整个日志管理组件从启动到结束的运行过程主要包括如下几个步骤：

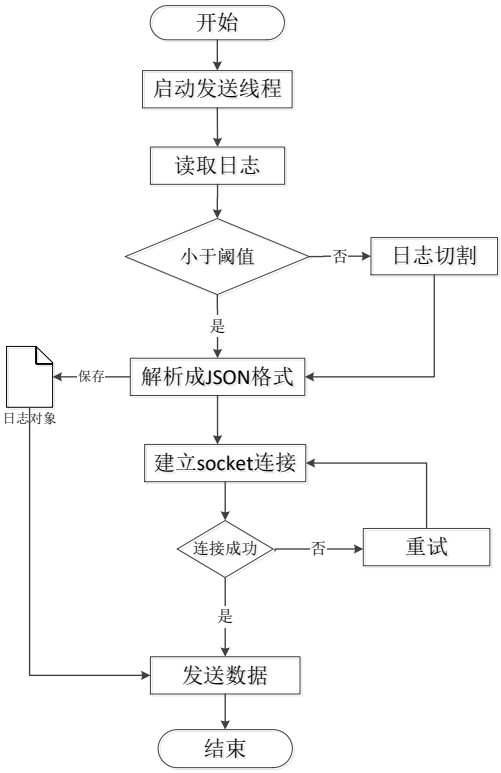


图 5-14 日志管理组件的运行流程

1）首先，在日志发送 agent 运行以后，会启动一个发送线程。在本文中的实现方式是，针对每个发送请求来启动一个线程。发送线程首先要做的是读取各应用程序产生的日志文件。前文已经提到过，这些日志通过 syslog 中统一定义的模板，以一种相同的格式保存在日志文件中。读取到日志文件以后，会将文件大小与之前定义的文件大小阈值进行比较。如果大于已经定义的阈值，则会进行文件切割，分为多个文件来进行处理。这里，之所以要进行分割是因为如果发送的数据包过大，使用 UDP 协议时可能会由于网络抖动或是服务器本身的性能问题而出现发送失败的问题。日志读取完成以后会将其解析成 JSON 格式，使用自定义类型保存成一个文件对象。

2）在日志文件处理完成以后，日志发送 agent 将与服务器建立一个 socket 连接，如果连接失败，将会进行重试。连接成功以后，发送线程会将之前已经保存在内存中的日志文件对象以 Key-Value 的格式拼接成一个发送的数据报文，然后使用 socket 的发送 API 进行数据发送。

5.6 本章小结

本章内容主要是对基于 FUSE 的云存储技术在实际场景中的应用过程进行了详细的

描述。基于所设计的云存储技术方案以及本文从负载均衡算法和读写性能方面所做的优化，对数据存储层，数据访问层的实现进行了细致的描述。在此基础上，为了进一步地完善所实现的云存储系统，提升其实用性和稳定性，对日志组件和监测组件的实现进行了说明。

总结与展望

本文基于用户空间文件系统 FUSE，以分布式文件系统 HDFS 作为底层存储方式，研究了一种新的云数据访问与存储方案。在此基础上，对原始方案本身所表现出来的不足从多个方面进行了优化。最后，将该方案应用在腾讯 TEG 大数据分析平台中，满足业务方每天数百 TB 级别的数据存储和访问需求。具体来讲，本文主要是从三个层面对已有的方案进行进一步的深入研究和优化：

- 考虑到在大规模存储集群中，往往会有大规模的并发访问请求。本文设计的负载均衡算法，解决了对这些请求进行分发，从服务器池中选择合适的节点来进行处理的问题。算法的设计充分考虑了 FUSE 访问 HDFS 的运行特点，通过服务器自身的运行状态，来评价其当前的负载程度，并进行实时反馈。调度器再根据各节点的反馈重新计算权值，以对后端的服务器资源进行充分利用。
- 针对 FUSE 访问 HDFS 中所存在的性能瓶颈，通过对 FUSE 内核源代码以及用户空间的接口进行分析，从缩短 CPU 上下文切换时间，使用直接 I/O 替换缓存 I/O，缓存文件 inode 和 dentry 信息三个角度对性能进行了优化。实验证明，本文的优化达到了约 20%的性能提升。同时，为了提升云存储方案的安全性，本文设计了细粒度的访问控制机制。
- 针对高可用 HDFS 集群中存在的挂载切换问题，在实际的应用场景中设计了实时监测和切换机制，提升了云存储系统中后端集群的高可用性，达到了较好的效果。

此外，在云存储方案的实际应用中，本文还自行设计了监测和管理组件，以进一步提升系统的实用性和可维护性。在具体实现时，摒弃了开源工具，充分结合云存储方案的特点来进行设计与开发工作，以使其更加契合实际应用场景。

下一步工作展望：

- 当前的方案是基于 Hadoop 本身提供的挂载组件来进行优化，但是受限于其事实上是通过 JNI 使用 Java 的接口来访问 HDFS，因此性能方面与本地文件系统始终有一定的差异，对一些 I/O 要求很高的应用程序如数据库软件，不太适用，下一步考虑对其进行优化。
- 由于 Fuse-DFS 读文件的修改操作不是原子的，因而并发修改同一文件时，可能出现数据不一致的情况，这也是下一步要进行优化的问题。

参考文献

[1] Mell P, Grance T. The NIST Definition of Cloud Computing[J]. Communications of the ACM, 2011, 53(6):50-50.

[2] Wu J, Ping L, Ge X, et al. Cloud Storage as the Infrastructure of Cloud Computing[C]. International Conference on Intelligent Computing and Cognitive Informatics. IEEE Computer Society, 2010:380-383.

[3] White T. Hadoop 权威指南.第 3 版[M]. 清华大学出版社, 2015.

[4] Karun A K, Chitharanjan K. A Review on Hadoop — HDFS Infrastructure Extensions[C]. Information & Communication Technologies. IEEE, 2013:132-137.

[5] Wu Y, Jiang Z L, Wang X, et al. Dynamic Data Operations with Deduplication in Privacy-Preserving Public Auditing for Secure Cloud Storage[C]. IEEE International Conference on Computational Science and Engineering. IEEE, 2017:562-567.

[6] Nguyen T T, Vu T K, Nguyen M H. BFC: High-performance Distributed Big-file Cloud Storage based on Key-Value Store[C]. IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, NETWORKING and Parallel/distributed Computing. IEEE, 2015:1-6.

[7] Balasaraswathi V R, Manikandan S. Enhanced Security for Multi-Cloud Storage Using Cryptographic Data Splitting with Dynamic Approach[C]. International Conference on Advanced Communication Control and Computing Technologies. IEEE, 2014:1190-1194.

[8] Hong T, Wu Y, Cao B, et al. A Dynamic Data Allocation Method with Improved Load-balancing for Cloud Storage System[C]. IET International Conference on Smart and Sustainable City. IET, 2013:183-188.

[9] Soares T S, Dantas M A R, Macedo D D J D, et al. A Data Management in a Private Cloud Storage Environment Utilizing High Performance Distributed File Systems[C]. IEEE, International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises. IEEE, 2013:158-163.

[10] Sivasubramanian S. Amazon DynamoDB: A Seamlessly Scalable Non-Relational Database Service[J]. 2012:729-730.

[11] Calder B, Wang J, Ogus A, et al. Windows Azure Storage: A Highly Available Cloud Storage Service With Strong Consistency[C]. ACM Symposium on Operating Systems Principles. ACM, 2011:143-157.

[12] Ghemawat S, Gobioff H, Leung S T. The Google File System[J]. ACM Sigops Operating Systems Review, 2003, 37(5):29-43.

[13] Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters[M].

ACM, 2008.

[14] Chang F, Dean J, Ghemawat S, et al. Bigtable:A Distributed Storage System for Structured Data[J]. ACM Transactions on Computer Systems, 2008, 26(2):1-26.

[15] Kolodner E K, Tal S, Harnik D, et al. A Cloud Environment for Data-Intensive Storage Services[C]. IEEE Third International Conference on Cloud Computing Technology and Science. IEEE Computer Society, 2011:357-366.

[16] Samundiswary S, Dongre N M. Object Storage Architecture in Cloud for Unstructured Data[C]. International Conference on Inventive Systems and Control. 2017:1-6..

[17] Frank S, Webman E, Hallak R, et al. Scalable Block Data Storage Using Content Addressing: US, US9104326[P]. 2015.

[18] Aliyun. 盘古：阿里云飞天分布式存储系统设计深度解析[EB/OL]. https://yq.aliyun.com/articles/64374, 2016.

[19] Rathod D M, Dahiya M S, Parikh S M. Towards Composition of RESTful Web Services[C]. International Conference on Computing, Communication and NETWORKING Technologies. IEEE, 2016:1-6.

[20] Jin X, Krishnan R, Sandhu R. A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC[C]. IFIP WG 11.3 Conference on Data and Applications Security and Privacy. Springer-Verlag, 2017:41-55.

[21] Morenovozmediano R, Montero R S, Llorente I M. IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures[J]. Computer, 2012, 45(12):65-72.

[22] Pahl C. Containerization and the PaaS Cloud[J]. IEEE Cloud Computing, 2015, 2(3):24-31.

[23] Palos-Sanchez P R, Arenas-Marquez F J, Aguayo-Camacho M. Cloud Computing (SaaS) Adoption as a Strategic Technology: Results of an Empirical Study[J]. Mobile Information Systems,2017,(2017-6-19), 2017, 2017(1):20-pages.

[24] 孙大为, 张广艳, 郑纬民. 大数据流式计算:关键技术及系统实例[J]. 软件学报, 2014, 25(4):839-862.

[25] Atlidakis V, Andrus J, Geambasu R, et al. POSIX Abstractions in Modern Operating Systems: the Old, the New, and the Missing[C]. Eleventh European Conference on Computer Systems. ACM, 2016:19..

[26] Vangoor B K R, Tarasov V, Zadok E. To FUSE or not to FUSE: Performance of User Space File Systems[C]. Usenix Conference on File and Storage Technologies. USENIX Association, 2017:59-72.

[27] 洛夫. Linux 内核设计与实现[M]. 机械工业出版社, 2004.

[28] 张栗粽, 崔园, 罗光春,等. 面向大数据分布式存储的动态负载均衡算法[J]. 计算机科

学, 2017, 44(5):178-183.
[29] 罗拥军, 李晓乐, 孙如祥. 负载均衡算法综述[J]. 图书情报导刊, 2008, 18(23):134-136.
[30] 买京京, 龚红艳, 宋纯贺. 集群系统中的动态反馈负载均衡策略[J]. 计算机工程, 2008, 34(16):114-115.
[31] Yang M, Wang H, Zhao J. Research on Load Balancing Algorithm Based on the Unused Rate of the CPU and Memory[C]. Fifth International Conference on Instrumentation and Measurement, Computer, Communication and Control. IEEE, 2016:542-545.
[32] 杨薇, 赵亮. Web 服务器性能优化研究[J]. 电子技术与软件工程, 2016(13):20-20..
[33] Edmundas Kazimieras Zavadskas, Abbas Mardani, Zenonas Turskis, et al. Development of TOPSIS Method to Solve Complicated Decision-Making Problems: An Overview on Developments From 2000 to 2015[J]. International Journal of Information Technology & Decision Making, 2016, 15(03):645-682.
[34] 张书奎. 高可用性集群中的动态负载平衡应用研究[J]. 计算机工程, 2007, 33(23):40-42.
[35] 郭红. 概率论与数理统计[M]. 高等教育出版社, 2010.
[36] 欧阳资生, 龚曙明. 广义帕累托分布模型:风险管理的工具[J]. 财经理论与实践, 2005, 26(5):88-92.
[37] 付戈, 张欣华, 李超. 面向多应用多租户的消息数据订阅关键技术研究[J]. 信息网络安全, 2017(11):44-49.
[38] Sourceforge. Strace[CP/OL]. Available:https://sourceforge.net/projects/strace/.
[39] 董晓明, 李小勇, 程煜. 分布式文件系统的写性能优化[J]. 微型电脑应用, 2012, 28(12):8-11.
[40] 汤小兵. 计算机操作系统[M]. 西安电子科技大学出版社, 2008.
[41] 张冬. 大话存储[M]. 清华大学出版社, 2015.
[42] 杨传辉. 大规模分布式存储系统:原理解析与架构实战[J]. 中国科技信息, 2013(19):140-140.
[43] 刘衍珩, 李松江, 王爱民. P2P 流媒体中动态分级传输模型及传输算法[J]. 吉林大学学报(工学版), 2016, 46(1):259-264.
[44] Chaeo Y, Kim D, Shin H, et al. Study on Improvement of Hadoop's Name Node SPOF Using NAS[C]. Korea Information Science Society Korea Computer Conference. 2015.
[45] Zimmermann H. OSI Reference Model--The ISO Model of Architecture for Open Systems Interconnection[J]. IEEE Transactions on Communications, 1980, 28(4):425-432.

攻读硕士学位期间取得的研究成果

一、已发表（包括已接受待发表）的论文，以及已投稿、或已成文打算投稿、或拟成文投稿的论文情况**（只填写与学位论文内容相关的部分）**：

序号	作者（全体作者，按顺序排列）	题　　目	发表或投稿刊物名称、级别	发表的卷期、年月、页码	相当于学位论文的哪一部分（章、节）	被索引收录情况

注：在“发表的卷期、年月、页码”栏：
1 如果论文已发表，请填写发表的卷期、年月、页码；
2 如果论文已被接受，填写将要发表的卷期、年月；
3 以上都不是，请据实填写“已投稿”，“拟投稿”。
不够请另加页。

二、与学位内容相关的其它成果（包括专利、著作、获奖项目等）

序号	作者（全体作者，按顺序排列）	题　　目	成果类型	编号
1	李拥军，**邱双旭**	一种云环境下的动态负载均衡方法	发明专利	受理号：201810348131.7
2	李拥军，**邱双旭**，林浩	一种购买预测建模方法	发明专利	公开号：CN107392644A
3	**邱双旭**，李拥军，韩国强	物联网智能家居云平台V1.0	计算机软件著作权	登记号：2016SR220365

致谢

研究生生涯即将结束，在论文即将完成的时候，向在过程之中给我帮助的老师、同学、朋友致以衷心的感谢。

感谢我的导师韩国强教授，在本篇论文的撰写过程中，多次审阅论文并给我提出修改意见，完善论文的内容和框架结构。在生活中，韩教授平易近人，严于律己，宽以待人，是一位有着崇高师德的老师。韩教授无论是从学术上，还是为人品质上，都激励着我不断提升自己，成为对社会有用的人。在这里，祝愿韩教授身体健康，工作顺利，万事如意！

感谢李拥军老师，在本篇论文的撰写过程中给予我很多意见。在我三年的研究生生涯中，李老师教会我学习的方法和为人处世的道理。在学习中，李老师不仅注重我们理论知识的积累，还非常注重提升大家的实际动手能力。在生活上，李老师对学生关怀备至，经常鼓励大家多锻炼身体，以轻松的心态应对生活中的挑战。感谢李老师为我们提供优良的科研环境，让我们可以潜心学术，发挥创造力。

感谢 336 实验室的小伙伴们，你们的活泼开朗，让我的三年求学生涯充满乐趣。感谢谢博，在科研中给予我很大的帮助，教会我思考问题的方式，让我在实际的工作和学习中受用无穷。

感谢我的父母，二十多年来，他们始终如一的支持我，让我在求学阶段心无旁骛，没有后顾之忧。今后，我将用实际行动，努力工作，不辜负你们的期望。

最后，感谢百忙之中抽空来评阅我论文的老师们，祝你们身体健康，万事如意！

Ⅳ - 2 答辩委员会对论文的评定意见

邱双旭同学的硕士论文“基于 FUSE 的云数据访问与存储优化研究”有较好的理论意义和应用价值。

论文参照了较多的相关文献资料，基于用户空间文件系统 FUSE，以分布式文件系统 HDFS 作为底层存储方式，研究了一种新的云数据访问与存储方案。在此基础上，对原始方案本身所表现出来的不足从多个方面进行了优化。并对系统进行了实现。通过实验测试，取得了较好的实验效果。

作者对文献的调研、数据的收集较充分，论文资料丰富，论点正确，结论合理，实验数据真实可信。反映作者具有一定的理论基础、较好的科研和工程能力。

论文结构清晰，图文表达规范，达到硕士学位论文水平。答辩过程讲述清楚，回答问题正确，经答辩委员会无记名投票，同意该同学通过硕士论文答辩，同意授予硕士学位。

论文答辩日期：2018 年 5 月 31 日

答辩委员会委员共 5 人，到会委员 5 人

表决票数：优秀（0）票；良好（5）票；及格（0）票；不及格（0）票

表决结果（打“√”）：优秀（ ）；良好（√）；及格（ ）；不及格（ ）

决议：同意授予硕士学位（√）　不同意授予硕士学位（ ）

答辩委员会成员签名			
	（主席）		