**基于Google File System的分布式文件系统设计和实现**

摘要

本论文基于《The Google File System》论文，结合现有的相关实例，进行一个相对完整的分布式文件系统研究，最终完成系统的设计和实现。

Abstract

**目录**

[基于Google File System的分布式文件系统设计和实现 1](#_Toc450923287)

[摘要 1](#_Toc450923288)

[Abstract 1](#_Toc450923289)

[第1章 前言 2](#_Toc450923290)

[1.1 课题背景 3](#_Toc450923291)

[1.2 分布式文件系统现状 4](#_Toc450923292)

[1.2.1 GFS 4](#_Toc450923293)

[1.2.2 HDFS 5](#_Toc450923294)

[1.2.3 XFS 5](#_Toc450923295)

[1.2.4 TFS 6](#_Toc450923296)

[1.2.5 Ceph 7](#_Toc450923297)

[1.2.6 BWFS 8](#_Toc450923298)

[1.2.7 GFS2 8](#_Toc450923299)

[1.3 本文结构安排 9](#_Toc450923300)

[第2章 分布式文件系统的设计细节 9](#_Toc450923301)

[2.1 分布式文件系统的设计需求 9](#_Toc450923303)

[2.1.1 功能需求 9](#_Toc450923304)

[2.1.2 非功能需求 12](#_Toc450923305)

[2.2 分布式文件系统的主体架构 13](#_Toc450923306)

[2.3 元数据管理方案 13](#_Toc450923307)

[2.3.1 集中式元数据管理 14](#_Toc450923308)

[2.3.2 分布式元数据管理 16](#_Toc450923309)

[2.3.3 无中心元数据管理 17](#_Toc450923310)

[2.3.4 混合式元数据管理 18](#_Toc450923311)

[2.3.5 元数据管理方案的设计 19](#_Toc450923312)

[2.4 元数据服务 19](#_Toc450923313)

[2.4.1 名字空间管理方案 20](#_Toc450923314)

[2.4.2 元数据一致性保障 20](#_Toc450923315)

[2.4.3 影子服务器 20](#_Toc450923316)

[2.4.4 故障恢复 21](#_Toc450923317)

[2.4.5 负载均衡 22](#_Toc450923318)

[2.5 副本管理 22](#_Toc450923319)

[2.5.1 Chunk块管理 22](#_Toc450923320)

[2.5.2 数据完整性 22](#_Toc450923321)

[2.5.3 副本一致性 22](#_Toc450923322)

[2.5.4 数据压缩策略 23](#_Toc450923323)

[2.5.5 小文件聚集块 23](#_Toc450923324)

[2.5.6 RADOS存储引擎 24](#_Toc450923325)

[2.6 客户端接口设计 24](#_Toc450923326)

[第3章 分布式文件系统的实现概述 24](#_Toc450923327)

[3.1 元数据管理服务实现 24](#_Toc450923329)

[3.2 数据服务实现 25](#_Toc450923330)

[3.3 客户端接口实现 26](#_Toc450923331)

[第4章 结束语 26](#_Toc450923332)

[致谢 26](#_Toc450923333)

[参考文献 26](#_Toc450923334)

第1章 前言

## 课题背景

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

分布式一直都是一个计算机领域很前沿的课题，分布式的出现，使得一些只能在传统大型主机上做的事能用很廉价的集群实现。并且分布式系统最大的优势是可扩展性，它能够适应需求变化而扩展。企业级应用需求经常随时间而不断变化，这也对企业级应用平台提出了很高的要求。比如移动互联网2C应用，随着互联网企业的业务规模不断增大，业务变得越来越复杂，并发用户请求越来越多，要处理的数据也越来越多，这个时候企业级应用平台必须能够适应这些变化，支持高并发访问和海量数据处理。分布式系统有良好的可扩展性，可以通过增加服务器数量来增强分布式系统整体的处理能力，以应对企业的业务增长带来的计算需求。

Google File System正是Google公司针对自己的业务需求研究出来的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，并提供容错功能。它可以给大量的用户提供总体性能较高的服务。诸如Google File System的分布式文件系统的应用非常广泛，它属于一个产业的基础，云计算、云存储、CDN，甚至未来人工智能的发展也离不开分布式文件系统的支撑。

Google公司发表的论文《The Google File System》算是业界著名的分布式文件系统理论支撑之一，但Google并没有开源这个项目，不过业界有很多基于其论文的实践，比如著名的HDFS。分布式文件系统不仅改变了数据存储和管理的方式，也具有本地文件系统所无法具备的数据备份、数据安全等优点。通过此论文和各种相关开源项目，很多互联网公司也相继开发出自己的分布式文件系统，各类型的分布式文件系统由不同的应用需求催生而来，能够按照应用需求提供不同的服务，它们在数据存储、数据管理、数据共享、安全性、稳定性、可用性和扩展性方面也各有所长、各具特色。所以根据实际需求定制设计分布式文件系统是非常重要的。

## 分布式文件系统现状

随着整个互联网行业的发展，实现大规模并行计算成为了各项互联网服务的门槛，传统的利用硬件实现文件的并行存储，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。它们就是分布式文件系统。

分布式文件系统由来已久，最早可追溯到上个世纪八十年代的NFS 和 AFS，经过了这么多年的发展，技术已有天翻地覆的革新，实用性也被各公司的实践证明。Google File System虽然是一类很经典的分布式文件系统理论，但距今已有十多年的历史，并且目前谷歌公司早已过渡到了Colossus，又被称作GFS2的新一代分布式文件系统。但是，纵观国内的互联网公司，大多还停留在GFS的针对性改进阶段。其中原因不能只说研究能力跟不上，更多的考虑可能还是在研发成本和效益的平衡上。其实仅对应用而言，进行过针对性改进的GFS的功能足以满足大多数业务需求。下面对一些国内外具有实践基础的分布式文件系统进行简单的介绍。

### GFS

《The Google File System》论文发表于2003年，对当时的分布式文件系统的设计思路是一个比较大的启发。正如Google工程师所言，他们重新审视了传统文件系统在设计上的折衷选择，基于对自己的应用的负载情况和技术环境的观察，衍生出了完全不同的设计思路。GFS的设计思路与以下实际问题相辅相成：组件失效被认为是常态事件，而不是意外事件；文件非常巨大；绝大部分文件的修改是采用在文件尾部追加数据，而不是覆盖原有数据的方式；应用程序和文件系统 API 需要协同设计。

一个GFS集群由一个master（主服务器）和大量的chunkserver（块服务器）构成，并被许多client（客户端）访问。文件被划分为固定的块，由主服务器安排存放到块服务器的本地硬盘上。主服务器会记录存放位置等数据，并负责维护和管理文件系统，包括块的租用、垃圾块的回收以及块在不同块服务器之间的迁移。此外，主服务器还周期性地与每个块服务器通过消息交互，以监视运行状态或下达命令。应用程序通过与主服务器和块服务器的交互来实现对应用数据的读写，应用与主服务器之间的交互仅限于元数据，也就是一些控制数据，其他的数据操作都是直接与块服务器交互的。这种控制与业务相分离的架构，在其他互联网产品方案上较为广泛，也较为成功。具体细节将在之后的设计细节中详述。

### HDFS

HDFS(Hadoop Distribute File System) 作为GFS的一个最重要的实现，HDFS设计目标和GFS是高度一致的。在架构、块大小、元数据等的实现上，HDFS与GFS大致一致。但是，在某些地方，HDFS与GFS又有些不同。如：

快照：GFS中的快照功能是非常强大的，可以非常快的对文件或者目录进行拷贝，并且不影响当前操作（读/写/复制）。GFS中生成快照的方式叫copy-on-write。也就是说，文件的备份在某些时候只是将快照文件指向原chunk，增加对chunk的引用计数而已，等到chunk上进行了写操作时，Chunk Server才会拷贝chunk块，后续的修改操作落到新生成的chunk上。而HDFS暂时并不支持快照功能，而是运用最基础的复制来完成。当然，在HDFS2.x版本开始提供了快照功能。

记录追加操作：HDFS对于写操作的数据流和GFS的功能一样。但是，HDFS并不支持记录追加和并行写操作。

垃圾回收：GFS垃圾回收采用惰性回收策略，即master并不会立即回收程序所删除的文件资源。 GFS选择以一种特定的形式标记删除文件（通常是将文件名改为一个包含时间信息的隐藏名字），这样的文件不再被普通用户所访问。Master会定期对文件的命名空间进行检查，并删除一段时间前的隐藏文件（默认3天）。而最初版本的HDFS并没有采用这样的垃圾回收机制，而是采取了一种更加简单但是更容易实现的直接删除方式。之后的版本加入了这个功能，但是实现方式与GFS有一定的出入，HDFS会将删除的文件放入/trash目录，类似windows的回收站，而不是像GFS一样采用标记的方式。

### XFS

XFS是腾讯台风(Typhoon)云计算平台中的分布式文件系统。基于GFS的理论，并根据实际需求，设计了一套主备机制，维持Master的多个副本，以便出错时切换到备用Master，持续提供服务，改进了GFS的可靠性和高可用性。

GFS的单Master架构大大简化了分布式文件系统的设计与实现，Master存储整个机群的目录树等元信息，并提供全局调度等功能。但这使得Master成为了单点：一旦Master宕机，整个GFS机群将不可用。Hadoop中HDFS沿袭了GFS的设计，其NameNode也设计为单点。

XFS的Master主备机制通过三套子协议完成设计目标：通过基于两阶段提交的Replication协议提高了数据可靠性；通过failover协议处理各种错误，完成主备切换，提升Master的可用性；通过learning协议完成滞后或者新加Master的数据自动对齐，支持错误节点的自动恢复和运行时增加Master节点，简化了运维。

其中Replication协议对所有的写操作（Mutation）执行两阶段提交协议。Replication的写操作由Master发起，确保数据持久写入主和足够的备才报告用户成功，以提高Master可靠性。读操作由主Master完成，以确保强一致性语义：已报告用户成功的数据一定可以读取。Failover协议在主Master出错时，选举新的主Master，完成服务的平滑迁移，增强Master容错能力。事实上，Failover协议需要完成Master的主、备、Newbie、Init等角色之间的全部转换流程。Learning协议负责将新加入的或者滞后的Newbie Master对齐到与主Master同步的状态，以不停服务下的自动完成新加Master或者修复出错的Master。

XFS Master主备机制达到了5.58万的峰值写qps，几百毫秒级的failover和learning，支持了Master的Rolling Update。这套协议自2011年4月起陆续在全部的超过10个XFS机群部署运行。

### TFS

TFS(Taobao File System) 是淘宝自主研发的一套高可扩展、高可用、高性能、面向互联网服务的分布式文件系统，主要针对海量的非结构化数据，它构筑在普通的Linux机器集群上，可为外部提供高可靠和高并发的存储访问。TFS为淘宝提供通常文件大小不超过1M但数量巨大的小文件存储，满足了淘宝对小文件存储的需求，被广泛地应用在淘宝各项应用中。它采用了HA架构和平滑扩容，保证了整个文件系统的可用性和扩展性。同时扁平化的数据组织结构，可将文件名映射到文件的物理地址，简化了文件的访问流程，一定程度上为TFS提供了良好的读写性能。

在TFS中大量的小文件合并成为一个大文件，类似GFS中Chunk的概念，而Chunk的定位信息我们称之为一级索引，而chunk内部具体的文件定位信息我们称之为二级索引，同时在TFS文件名称中包含这些索引信息，在用户写入一个文件之前，他必须向TFS系统申请一个文件名称。这种方式虽然在某些情况下显得不像传统文件系统那样灵活，但也给了我们系统更大的可扩展性。我们保证可以中心控制节点的内存可以支撑PB级别的一级索引，而二级索引仅需要针对单台数据量。这样，我们就避免了数据量膨胀带来的扩容难度。

当TFS存储容量出现不足，需要进行系统扩容的时候，可以根据数据增长情况进行规划，任意数量的加入提供相应存储的服务器。而这些新的存储服务器会向中心控制节点进行报告。而中心控制节点在有数据写入时，将根据已存储容量的百分比、系统当前负载等参数动态地分配写入的服务器。同时，在系统空闲时间段，中心控制节点也会根据当前的数据分布情况制定数据迁移计划，并逐步完成数据平衡。与此类似，当发生服务器崩溃时，中心控制节点也会进行数据迁移以保证足够的备份，同时也会进行数据均衡操作。这些操作都是自动进行的，不需要人工干预。

TFS与目前一些主流的开源分布式文件系统设计思想是相似的，如HDFS、MFS、KFS、Sector。TFS的高可扩展、高可用性是很好的，然而也存在一定不足，如通用性、用户接口、性能等方面。

在通用性方面，TFS目前只支持小文件的应用，大文件应用是不支持的。对小图片、网页等几十KB内的数据存储非常适用，但对视频点播VOD、文件下载等应用暂时无法适用。

在性能方面，Client写文件是同步处理的，需要等所有dataserver写成功后才能返回，这很是影响性能。

在用户接口上，TFS没有提供POSIX接口，提供的API也与标准接口不一致。另外，TFS有自己的文件命名规则，如果用户使用自定义的文件名，则需要自已维护文件名与TFS文件名之间的映射关系。

### Ceph

Ceph是一个典型的非集中式的分布式文件系统，它的master节点采用集群的方式有效的避免了单点问题，并且传统的类GFS的分布式文件系统通常是采用两层命名空间，即文件系统和文件到chunk、chunk到存储节点的映射关系，这两层命名空间尤其是第二层映射关系的信息占据了大量元数据内容，在Ceph中的元数据只有第一层命名空间的文件信息，通过文件信息和CRUSH算法，Ceph可以在任何地方包括客户端计算出全部的映射关系，这样节省了大量元数据。

Ceph完成了三个关键挑战，性能、可扩展性和可靠性。通过剥离现有文件系统的分布表设计，Ceph最大限度地分离数据与元数据管理，允许他们独立使用。这种分离依赖于CRUSH，允许客户来计算对象的位置，而不是在分布表中寻找。CRUSH算法可在设备故障、扩张和集群重组为常态的大型存储集群中跨故障域执行数据复制。

RADOS是一个支持海量存储对象的分布式对象存储系统。RADOS最大的特色是使用CRUSH算法维护存储对象与存储服务器的对应关系，并采用了无集中管理服务设计，对象复制，集群扩容，数据迁移，故障检测和处理能复杂功能由OSD提供，既避免了单点失败，又提升了集群扩展性。尽管对象可以被认为是文件，并存储在一个通用的文件系统中，EBOFS提供更合适的语义和高性能的满足Ceph中特定的工作负载和接口需求。

Ceph的元数据管理体系结构解决了大型存储系统中最棘手的问题之一，即如何实现有效提供一个统一的目录层次结构并服从POSIX语义，并能随着元数据服务器数量增加性能保持良好的机制。Ceph中基于动态子树分区的元数据集群是一个独特的可伸缩的，高效的方法，可在不同工作负载中有保持良好的适应性。

### BWFS

蓝鲸分布式文件系统（BWFS）是中科院计算所国家高性能计算机工程中心在大规模网络存储领域多年专注研究的重要成果，是蓝鲸存储系列之一。BWFS是一个跨平台共享、可扩展的大型海量分布式文件系统。

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

### GFS2

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

## 本文结构安排

本文由四章组成：

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

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

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

第四章总结了基于GFS的分布式文件系统的设计成果，并提出今后的展望。

第2章 分布式文件系统的设计细节



## 分布式文件系统的设计需求

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

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

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

### 功能需求

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

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

（1）数据节点的无缝加载与卸载

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

（2）加载路径的统一管理

集群中需要统一每个数据节点的加载路径，确保每个节点加载路径的一致性。

（3）内部资源的管理

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

（4）全局文件的统一管理

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

（5）基本的文件系统操作

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

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

（1）支持海量文件存储

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

（2）大文件管理

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

（3）支持小文件

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

（4）流式读取和随机读取

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

（5）节点负载均衡

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

（6）文件备份

集群由普通的设备，文件的安全性必然是首位的，通过文件备份可以有效地规避因自然灾害或者系统故障造成的数据损坏或丢失。

（7）故障恢复

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

（8）冷热数据的区分对待

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

（9）提供客户端编程接口

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

（10）系统资源统计

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

（11）日志管理

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

### 非功能需求

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

（1）并行处理性能

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

（2）最大响应延迟

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

（3）系统高可用性

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

（4）数据容错性

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

（5）客户端安全性

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

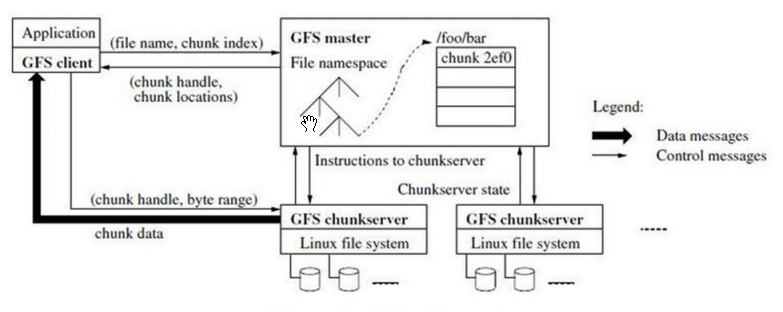
（6）集群可扩展性

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

（7）可靠性

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

## 分布式文件系统的主体架构

由于本论文所述的分布式系统设计是基于Google File System，整体架构如图1所示： 

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

## 元数据管理方案

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

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

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

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

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

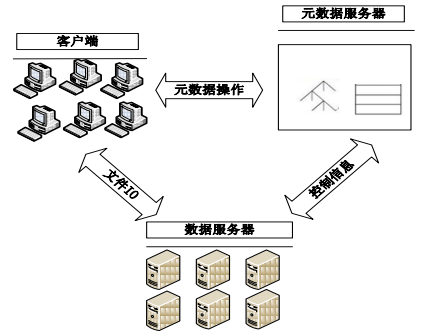
无中心元数据管理方案主要基于p2p技术，无中心管理节点和当然也无单点问题。支持无限扩展。就目前的技术而言，这种方案的商用实践几乎不现实，因为数据一致性的问题非常复杂，并且数据安全也得不到很好的保障。

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

下面具体分析这几种方案。

### 集中式元数据管理

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



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

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

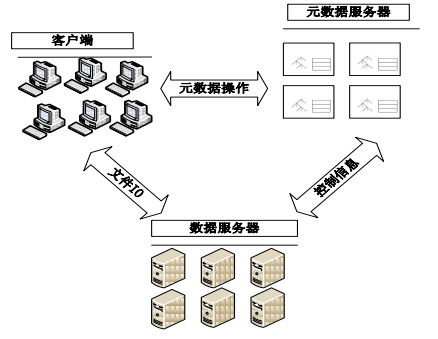
性能瓶颈：虽然说单点故障是毁灭性的，但是它并没有那么严格，并且目前来看有许多针对性方案比如说双机热备去缓解这个问题，所以性能瓶颈才是最主要的问题，也是集中式元数据管理方案中无法回避的问题。由于集中式元数据管理方案为了效率考虑，一般都会将全部元数据缓存在内存，所以单机内存的容量上限直接决定了元数据的上限，也决定了整个系统的存储上限。此外，单机的性能上限也直接影响了整个系统的并发处理计算能力。在越来越庞大的数据增长趋势下，性能瓶颈所带来的制约不下于单点故障。

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

待加

### 分布式元数据管理

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



在分布式元数据管理方案中，元数据服务器之间的关系有两种：1、元数据服务器之间角色对等，互相的元数据信息可能会有重叠；2、元数据服务器之间相互协作，各自负责一部分元数据，元数据之间没有重叠。

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

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

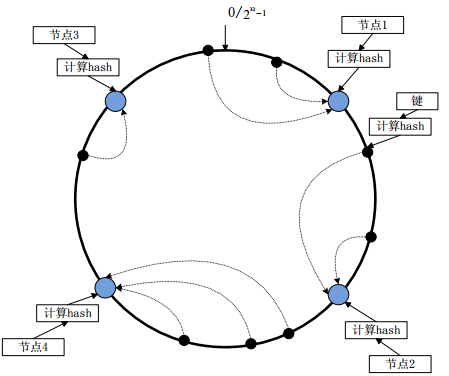
一致性问题：

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

待加

### 无中心元数据管理

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



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

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

一致性哈希：一致性哈希算法在1997年由麻省理工学院提出的一种分布式哈希（DHT）实现算法，设计目标是为了解决因特网中的热点(Hot spot)问题，初衷和CARP十分类似。一致性哈希修正了CARP(共用地址冗余协议)使用的简单哈希算法带来的问题，使得分布式哈希（DHT）可以在P2P环境中真正得到应用。

一致性哈希算法解决了普通哈希算法在动态变化的环境中的这四个问题：

1、平衡性(Balance)：平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。

2、单调性(Monotonicity)：单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到原有的或者新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。

3、分散性(Spread)：在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。

4、负载(Load)：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同 的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

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

### 混合式元数据管理

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

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

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

### 元数据管理方案的设计

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

## 元数据服务

元数据服务器中一般会保存三种类型的元数据：文件的命名空间，文件到块的映射，每个块副本的位置。所有的元数据都缓存在元数据服务器的内存里，以保证系统的高效。

### 名字空间管理方案

元数据选择全部保存在内存中。有两个好处：一是元数据管理服务器的操作速度非常快。二是元数据管理服务器可以在后台简单而高效的周期性扫描自己保存的全部状态信息。这种周期性的状态扫描也用于实现Chunk垃圾收集、在Chunk服务器失效的时重新复制数据、通过Chunk的迁移实现跨Chunk服务器的负载均衡以及磁盘使用状况统计等功能。

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

### 元数据一致性保障

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

另外，系统还利用版本号来区分过副本，无论何时，只要Master节点和Chunk签订一个新的租约，它就增加Chunk的版本号，然后通知最新的副本。

### 影子服务器

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

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

### 故障恢复

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

但是，元数据管理服务器并不保存持久化chunk块的副本位置映射信息。元数据管理服务器只是在启动的时候轮询数据服务器以获取这些信息。

#### Log server

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

#### Checkpoint

元数据管理服务器在故障恢复时，通过重演操作日志把文件系统恢复到最近的状态。为了缩短元数据管理服务器启动的时间，必须使日志足够小。元数据管理服务器会在日志增长到一定量时对系统状态（主要是名字空间的元数据状态）做一次 Checkpoint，将所有的状态数据写入一个 Checkpoint 文件。在灾难恢复的时候，元数据管理服务器就通过从磁盘上读取这个Checkpoint文件，以及重演Checkpoint之后的少量日志文件就能够恢复系统。Checkpoint文件以压缩B树形式的数据结构存储，可以直接映射到内存，在用于命名空间查询时无需额外的解析。这大大提高了恢复速度，增强了可用性。

元数据管理服务器恢复只需要最新的Checkpoint文件和后续的日志文件。旧的Checkpoint文件和日志文件可以被删除，但是为了应对灾难性的故障，通常会多保存一些历史文件。Checkpoint失败不会对正确性产生任何影响，因为恢复功能的代码可以检测并跳过没有完成的Checkpoint文件。

### 负载均衡

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

## 副本管理

### Chunk块管理

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

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

### 数据完整性

每个数据服务器都会使用Checksum来检查保存的数据块的完整性。Checksum是数据校验中很常用的手段，每个64M的Chunk会分成1024个64K的小块，每个小块对应一个32位的Checksum，Checksum数据一般存在数据服务器本地内存和硬盘。

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

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

另外，在数据服务器空闲的时候，它会扫描和校验每个不活动的Chunk块的内容。这使得我们能够发现很少被读取的Chunk是否完整。一旦发现有Chunk的数据损坏，元数据管理服务可以创建一个新的、正确的副本，然后把损坏的副本删除掉。这个机制也避免了非活动的、已损坏的Chunk欺骗元数据管理节点，使元数据管理节点认为它们已经有了足够多的副本了。

### 副本一致性

当数据服务器失效时，数据服务器的副本有可能因错失了一些修改操作而过期失效。元数据节点保存了每个Chunk块的版本号，用来区分当前的副本和过期副本。

无论何时，只要元数据管理节点和Chunk签订一个新的租约，它就增加Chunk的版本号，然后通知最新的副本。元数据管理节点和这些副本都把新的版本号记录在它们持久化存储的状态信息中。这个动作发生在任何客户机得到通知以前，因此也是对这个Chunk开始写之前。如果某个副本所在的Chunk服务器正好处于失效状态，那么副本的版本号就不会被增加。元数据管理节点在这个Chunk服务器重新启动，并且向元数据管理节点报告它拥有的Chunk的集合以及相应的版本号的时候，就会检测出它包含过期的Chunk。如果元数据管理节点看到一个比它记录的版本号更高的版本号，元数据管理节点会认为它和Chunk服务器签订租约的操作失败了，因此会选择更高的版本号作为当前的版本号。

元数据管理节点在例行的垃圾回收过程中移除所有的过期失效副本。在此之前，元数据管理节点在回复客户机的Chunk信息请求的时候，简单的认为那些过期的块根本就不存在。另外一重保障措施是，元数据管理节点在通知客户机哪个Chunk服务器持有租约、或者指示Chunk服务器从哪个Chunk服务器进行克隆时，消息中都附带了Chunk的版本号。客户端或者Chunk服务器在执行操作时都会验证版本号以确保总是访问当前版本的数据。

#### 租约

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

### 数据压缩策略

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

通过分析，发现纠删码技术能有效应对这个需求。下表是三副本数据存储方案和纠删码方案对比。

表格 2.5‑1

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

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

### 小文件聚集块

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

### RADOS存储引擎

RADOS是专门为Ceph设计的一个支持海量存储对象的分布式对象存储系统。RADOS最大的特色是使用CRUSH算法维护存储对象与存储服务器的对应关系，并采用了无Master设计，对象复制，集群扩容，数据迁移，故障检测和处理能复杂功能由OSD提供，既避免了单点失败，又提升了集群扩展性。而GFS的存储引擎并没有做特殊优化，所有的数据块都直接存在普通Linux文件系统上。本系统的数据块存储引擎采用双引擎，提供自由选择，默认将数据块存储在Linux文件系统上，但根据需要优化的程度存储大文件，为集群提供更好的扩展性，还可以缓解单点问题，也可以将小文件存储到RADOS上，这样就不需要采用小文件聚集策略，也不需要元数据块地址映射信息，因为这个地址可以根据CRUSH算法计算而来。由于Ceph是开源项目，并且直接提供了RADOS库供开发者直接调用，所以实现起来比较简单。

## 客户端接口设计

一般来说，一个分布式文件系统可以有很多个进行着不同业务的客户端共同访问。作为分布式文件系统本身而言，不提供直接的客户端，客户端代码以库的形式被提供。客户端代码实现了GFS文件系统的API接口函数、应用程序与元数据管理节点和Chunk服务器通讯、以及对数据进行读写操作。客户端和元数据管理节点的通信只获取元数据，所有的数据操作都是由客户端直接和Chunk服务器进行交互的。客户端缓存数据几乎没有什么用处，因为大部分程序要么以流的方式读取一个巨大文件，要么工作集太大根本无法被缓存。无需考虑缓存相关的问题也简化了客户端和整个系统的设计和实现。不过，通常而言，客户端会缓存元数据，在从元数据管理节点获取元数据的时候，客户端一般会再获取目标元数据接下来的部分元数据放入缓存，因为根据统计，客户端执行的业务很大可能会进行连续读取。

第3章 分布式文件系统的实现概述



## 元数据管理服务实现

本节简单介绍元数据管理服务的框架和实现的功能模块。

元数据管理服务器主要负责数据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。具体功能见下表所示。

表格 3.1‑1

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



图 3.1‑1元数据管理服务器模块划分

## 数据服务实现

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

表格 3.2‑1

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

## 客户端接口实现

提供了一套类似传统文件系统的 API 接口函数，虽然并不是严格按照 POSIX 等标准 API 的形式实现的，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。文件以分层目录的形式组织，用路径名来标识。支持常用的操作，如创建新文件、删除文件、打开文件、关闭文件、读和写文件。

另外，系统还提供了快照和记录追加操作。快照以很低的成本创建一个文件或者目录树的拷贝。记录追加操作允许多个客户端同时对一个文件进行数据追加操作，同时保证每个客户端的追加操作都是原子性的。这对于实现多路结果合并，以及“生产者-消费者”队列非常有用，多个客户端可以在不需要额外的同步锁定的情况下，同时对一个文件追加数据。

第4章 结束语

致谢

参考文献