谷歌文件系统

Sanjay Ghemawat、Howard Gobioff 和 Shun-Tak Leung

谷歌*

抽象的

我们设计并实现了 Google 文件系统,这是一个可扩展的分布式文件系统,适用于大型分布式数据密集型应用程序。它在廉价的商品硬件上运行时提供容错能力,并为大量客户端提供高聚合性能。

虽然与以前的分布式文件系统有许多相同的目标,但我们的设计是由对当前和预期的应用程序工作负载和技术环境的观察驱动的,这反映了与早期文件系统假设的明显背离。这促使我们重新审视传统选择并探索截然不同的设计点。

文件系统已经成功满足了我们的存储需求。
它在 Google 内部广泛部署,作为我们服务所用数据的生成和处理以及需要大数据集的研发工作的存储平台。迄今为止最大的集群在一千多台机器上的数千个磁盘上提供了数百 TB 的存储空间,并由数百个客户端同时访问。

在本文中,我们介绍了旨在支持分布式应用程序的文件系统接口扩展,讨论了我们设计的许多方面,并报告了来自微基准测试和实际使用的测量结果。

类别和主题描述符

D [4]: 3分布式文件系统

一般条款

设计、可靠性、性能、测量

关键词

容错、可扩展性、数据存储、集群存储

*可以通过以下地址联系作者:{sanjay,hgobioff,shuntak}@google.com。

SOSP 03,2003年 10 月 19 日至 22 日,美国纽约博尔顿兰丁。

一、简介

我们设计并实施了 Google 文件系统 (GFS),以满足 Google 数据处理需求快速增长的需求。 GFS 与以前的分布式文件系统有许多相同的目标,例如性能、可伸缩性、可靠性和可用性。然而,它的设计是由我们对当前和预期的应用程序工作负载和技术环境的关键观察所驱动的,这反映了与一些早期文件系统设计假设的明显背离。我们重新审视了传统选择,并探索了设计空间中截然不同的点。

首先,组件故障是常态而非例外。文件系统由数百甚至数千台由廉价商品部件构建的存储机器组成,并由相当数量的客户端机器访问。组件的数量和质量实际上保证了某些组件在任何给定时间都无法正常工作,而某些组件将无法从当前故障中恢复。我们已经看到由应用程序错误、操作系统错误、人为错误以及磁盘、内存、连接器、网络和电源故障引起的问题。因此,持续监控、错误检测、容错和自动恢复必须是系统不可或缺的一部分。

其次,按照传统标准,文件很大。多 GB 的文件很常见。每个文件通常包含许多应用程序对象,例如 Web 文档。当我们经常处理包含数十亿个对象的许多 TB 的快速增长的数据集时,即使文件系统可以支持,管理数十亿个大约 KB 大小的文件也很笨拙。因此,必须重新审视设计假设和参数,例如 I/O 操作和块大小。

第三,大多数文件是通过附加新数据而不是覆盖现有数据来改变的。文件中的随机写入实际上是不存在的。一旦写入,文件只能被读取,而且通常只能按顺序读取。各种数据都具有这些特征。有些可能构成数据分析程序扫描的大型存储库。有些可能是运行应用程序不断产生的数据流。有些可能是档案数据。有些可能是在一台机器上产生并在另一台机器上同时或稍后处理的中间结果。鉴于这种对大文件的访问模式,追加成为性能优化和原子性保证的重点,而在客户端缓存数据块就失去了吸引力。

第四,共同设计应用程序和文件系统 API 通过增加我们的灵活性使整个系统受益。

例如,我们放宽了 GFS 的一致性模型,以极大地简化文件系统,而不会给应用程序带来沉重的负担。我们还引入了原子追加操作,以便多个客户端可以同时追加到一个文件,而无需在它们之间进行额外的同步。这些将在后面更详细地讨论

纸。

目前部署了多个 GFS 集群用于不同的目的。最大的一个有超过 1000 个存储节点,超过 300 TB 的磁盘存储,并被不同机器上的数百个客户端连续大量访问。

2. 设计概述

2.1 假设在为我们的

需要设计文件系统时,我们一直以既提供挑战又提供机会的假设为指导。我们之前提到了一些关键观察结果,现在更详细地阐述了我们的假设。

- 该系统由许多经常发生故障的廉价商品组件构建而成。它必须不断地自我监控,定期检测、容忍组件故障并从中迅速恢复。

- 系统存储少量大文件。我们预计会有几百万个文件,每个文件的大小通常为 100 MB 或更大。多 GB 的文件是常见的情况,应该有效地管理。必须支持小文件,但我们不需要为它们优化。

- 工作负载主要包括两种读取:大型流式读取和小型随机读取。在大型流式读取中,单个操作通常读取数百 KB,更常见的是 1 MB 或更多。

  来自同一客户端的连续操作通常会读取文件的连续区域。小型随机读取通常会在某个任意偏移处读取几 KB。注重性能的应用程序通常对它们的小读取进行批处理和排序,以在文件中稳步前进,而不是来回移动。

- 工作负载还有许多将数据附加到文件的大型顺序写入。典型的操作大小与读取的操作大小相似。一旦写入,文件很少再次修改。支持文件中任意位置的小写操作,但不一定要高效。

- 系统必须为同时附加到同一个文件的多个客户端有效地实现定义良好的语义。我们的文件通常用作生产者消费者队列或用于多路合并。数百个生产者,每台机器运行一个,将同时追加到一个文件。具有最小同步开销的原子性是必不可少的。该文件可能稍后被读取,或者消费者可能正在同时读取该文件。

- 高持续带宽比低延迟更重要。我们的大多数目标应用程序都非常重视以高速率批量处理数据,而很少有应用程序对单个读取或写入的响应时间有严格的要求。

2.2 界面

GFS 提供了一个熟悉的文件系统接口,尽管它没有实现标准的 API,例如 POSIX。文件在目录中分层组织,并由路径名标识。我们支持创建、删除、打开、关闭、读取和写入文件的常规操作。

此外,GFS 具有快照和记录追加操作。快照以低成本创建文件或目录树的副本。Record append 允许多个客户端同时将数据追加到同一个文件,同时保证每个客户端追加的原子性。它对于实现多路合并结果和生产者消费者队列很有用,许多客户端可以同时附加到这些队列而无需额外锁定。我们发现这些类型的文件在构建大型分布式应用程序时具有无可估量的价值。快照和记录追加分别在 3.4 和 3.3 节中进一步讨论。

2.3 体系结构GFS 集群由

一个主服务器和多个块服务器组成,并由多个客户端访问,如图 1 所示。每个客户端通常都是运行用户级服务器进程的商用 Linux 机器。在同一台机器上运行 chunkserver 和客户端是很容易的,只要机器资源允许,并且可以接受运行可能不稳定的应用程序代码所导致的较低可靠性。

文件被分成固定大小的块。每个块都由主在创建块时分配的不可变且全局唯一的 64 位块句柄标识。

Chunkservers 将块作为 Linux 文件存储在本地磁盘上,并读取或写入由块句柄和字节范围指定的块数据。为了可靠性,每个块都被复制到多个块服务器上。默认情况下,我们存储三个副本,但用户可以为文件命名空间的不同区域指定不同的复制级别。

master 维护所有文件系统元数据。这包括名称空间、访问控制信息、从文件到块的映射以及块的当前位置。

它还控制系统范围的活动,例如块租用管理、孤立块的垃圾收集以及块服务器之间的块迁移。 master 周期性地与 HeartBeat 消息中的每个 chunkserver 通信,给它指令并收集它的状态。

链接到每个应用程序的 GFS 客户端代码实现文件系统 API 并与主服务器和块服务器通信以代表应用程序读取或写入数据。客户端与 master 进行元数据操作交互,但所有数据承载通信直接进入 chunkservers。我们不提供 POSIX API,因此不需要连接到 Linux vnode 层。

客户端和 chunkserver 都不缓存文件数据。
客户端缓存几乎没有什么好处,因为大多数应用程序流式传输大量文件或工作集太大而无法缓存。没有它们可以通过消除缓存一致性问题来简化客户端和整个系统。

(然而,客户端会缓存元数据。)Chunkservers 不需要缓存文件数据,因为块存储为本地文件,因此 Linux 的缓冲区缓存已经将经常访问的数据保存在内存中。

2.4 单主

拥有一个单一的 master 大大简化了我们的设计,并使 master 能够进行复杂的块放置

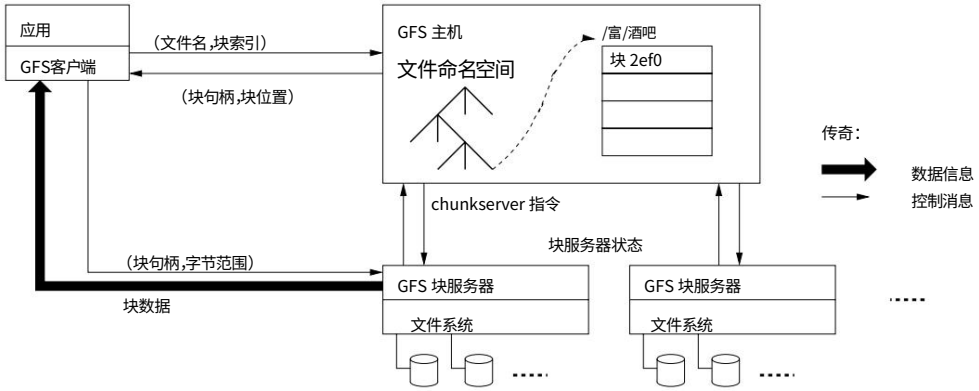


图 1:GFS 架构

和使用全球知识的复制决策。但是,我们必须尽量减少它对读写的参与,以免它成为瓶颈。客户端永远不会通过master读写文件数据。相反,客户端询问主服务器它应该联系哪些块服务器。它在有限的时间内缓存此信息并与

chunkservers 直接用于许多后续操作。

让我们参考图 1 解释简单读取的交互。首先,使用固定块大小,客户端将应用程序指定的文件名和字节偏移量转换为文件内的块索引。然后,它向 master 发送一个包含文件名和块索引的请求。 master 回复相应的块句柄和副本的位置。客户端使用文件名和块索引作为键来缓存此信息。

然后客户端向其中一个副本发送请求,最有可能是最近的副本。该请求指定块句柄和该块内的字节范围。在缓存信息过期或文件重新打开之前,对同一块的进一步读取不需要更多的客户端与主机交互。

事实上,客户端通常会在同一个请求中请求多个块,而主服务器也可以包含紧跟在请求之后的块的信息。这些额外的信息几乎无需额外费用就可以避免未来与客户之间的多次互动。

2.5 块大小

块大小是关键的设计参数之一。我们选择了 64 MB,这比典型的文件系统块大小大得多。每个块副本都作为普通 Linux 文件存储在块服务器上,并且仅在需要时进行扩展。

惰性空间分配避免了由于内部碎片造成的空间浪费,这可能是对如此大的块大小的最大反对意见。

大块大小提供了几个重要的优势。
首先,它减少了客户端与 master 交互的需要,因为在同一块上读取和写入只需要向 master 发出一次初始请求以获取块位置信息。这种减少对于我们的工作负载来说尤其重要,因为应用程序主要是按顺序读取和写入大文件。即使对于小的随机读取,客户端也可以轻松缓存多 TB 工作集的所有块位置信息。其次,由于在大块上,客户端更有可能在给定块上执行许多操作,它可以通过保持持久性来减少网络开销

在一段较长的时间内建立到 chunkserver 的 TCP 连接。第三,它减少了存储在母版上的元数据的大小。这允许我们将元数据保存在内存中,这反过来又带来了我们将在第 2.6.1 节中讨论的其他优势。

另一方面,大块大小,即使有惰性空间分配,也有其缺点。一个小文件由少量块组成,也许只有一个。如果许多客户端访问同一个文件,存储这些块的 chunkservers 可能会成为热点。实际上,热点并不是主要问题,因为我们的应用程序大多按顺序读取大型多块文件。

然而,当 GFS 首次被批处理队列系统使用时,热点确实出现了:一个可执行文件作为单块文件写入 GFS,然后同时在数百台机器上启动。存储此可执行文件的少数 chunkservers 因数百个同时请求而超载。我们通过以更高的复制因子存储此类可执行文件并使批处理队列系统错开应用程序启动时间来解决此问题。一个潜在的长期解决方案是允许客户端在这种情况下从其他客户端读取数据。

2.6 元数据

master 存储三种主要类型的元数据:文件和块命名空间、从文件到块的映射以及每个块的副本的位置。所有元数据都保存在主人的记忆中。前两种类型（名称空间和文件到块的映射)也通过将突变记录到存储在主服务器本地磁盘上并复制到远程机器上的操作日志中来保持持久性。使用日志可以让我们简单、可靠地更新主节点状态,并且不会在主节点崩溃时冒不一致的风险。 master 不会持久存储块位置信息。相反,它会在 master 启动时以及每当一个 chunkserver 加入集群时向每个 chunkserver 询问它的 chunks。

2.6.1 内存数据结构

由于元数据存储在内存中,主操作速度很快。此外,主节点在后台定期扫描其整个状态既简单又高效。

这种定期扫描用于实现块垃圾收集、在块服务器出现故障时重新复制以及块迁移以平衡负载和磁盘空间

跨块服务器的使用。 4.3 和 4.4 节将进一步讨论这些活动。

这种仅内存方法的一个潜在问题是,块的数量以及整个系统的容量受主服务器拥有的内存量的限制。这在实践中并不是一个严重的限制。 master 为每个 64 MB 块维护少于 64 字节的元数据。大多数块是满的,因为大多数文件包含许多块,只有最后一个块可能被部分填充。同样,文件命名空间数据通常需要每个文件少于 64 个字节,因为它使用前缀压缩紧凑地存储文件名。

如果有必要支持更大的文件系统,为 master 添加额外内存的成本对于我们通过将元数据存储在内存中获得的简单性、可靠性、性能和灵活性来说是一个很小的代价。

2.6.2 块位置

master 不会永久记录哪些 chunkservers 具有给定块的副本。它只是在启动时轮询 chunkservers 以获取该信息。 master 之后可以使自己保持最新状态,因为它控制所有块的放置并使用常规的 HeartBeat 消息监视块服务器的状态。

我们最初尝试将块位置信息持久保存在 master 中,但我们认为在启动时从块服务器请求数据要简单得多,此后定期请求数据。这消除了在 chunkservers 加入和离开集群、更改名称、失败、重新启动等时保持 master 和 chunkservers 同步的问题。在具有数百台服务器的集群中,这些事件经常发生。

理解这个设计决策的另一种方法是认识到块服务器对它在自己的磁盘上有或没有什么块有最终决定权。试图在 master 上维护此信息的一致视图是没有意义的,因为 chunkserver 上的错误可能导致 chunk 自发消失（例如,磁盘可能损坏并被禁用)或者操作员可能重命名 chunkserver。

2.6.3 操作日志

操作日志包含关键元数据更改的历史记录。它是 GFS 的核心。它不仅是元数据的唯一持久记录,而且还是定义并发操作顺序的逻辑时间线。文件和块,以及它们的版本（见第 4.5 节）,都由它们创建的逻辑时间唯一且永久地标识。

由于操作日志很关键,我们必须可靠地存储它,并且在元数据更改持久化之前,不要让更改对客户端可见。否则,我们实际上会丢失整个文件系统或最近的客户端操作,即使这些块本身仍然存在。因此,我们将它复制到多台远程机器上,只有在本地和远程将相应的日志记录刷新到磁盘后才响应客户端操作。 master 在刷新之前将多个日志记录一起批处理,从而减少刷新和复制对整个系统吞吐量的影响。

主服务器通过重放操作日志来恢复其文件系统状态。为了最小化启动时间,我们必须保持日志小。每当日志增长超过一定大小时,master 就会检查其状态,以便它可以通过从本地磁盘加载最新的检查点并仅重放

	写	Record Append定义
连续剧 成功	定义的	穿插不一致
并发一致但未定义 成功	不一致	
失败		

表 1:突变后的文件区域状态

之后的日志记录数量有限。检查点是一种紧凑的 B-tree 形式,可以直接映射到内存中,用于命名空间查找,无需额外解析。这进一步加快了恢复速度并提高了可用性。

因为建立检查点可能需要一段时间,所以 master 的内部状态的结构使得可以在不延迟传入突变的情况下创建新的检查点。 master 切换到一个新的日志文件,并在一个单独的线程中创建新的检查点。新检查点包括切换前的所有突变。对于拥有几百万个文件的集群,它可以在一分钟左右创建。完成后,它会在本地和远程写入磁盘。

恢复只需要最新的完整检查点和后续日志文件。较旧的检查点和日志文件可以自由删除,但我们保留了一些以防止灾难。检查点期间的失败不会影响正确性,因为恢复代码会检测并跳过不完整的检查点。

2.7 一致性模型GFS 有一个宽松的

一致性模型,可以很好地支持我们高度分布式的应用程序,但实现起来仍然相对简单和高效。我们现在讨论 GFS 的保证及其对应用程序的意义。我们还强调了 GFS 如何维护这些保证,但将细节留给本文的其他部分。

2.7.1 GFS 的保证文件命名空间突

变（例如,文件创建)是原子的。
它们由 master 专门处理:命名空间锁定保证原子性和正确性（第 4.1 节）； master 的操作日志定义了这些操作的全局总顺序（第 2.6.3 节）。

数据突变后文件区域的状态取决于突变的类型、成功或失败以及是否存在并发突变。表 1 总结了结果。如果所有客户端将始终看到相同的数据,无论它们从哪个副本读取,则文件区域是一致的。如果区域是一致的,则在文件数据突变之后定义一个区域,并且客户端将看到突变写入的全部内容。当突变在没有并发写入者干扰的情况下成功时,受影响的区域被定义（并且暗示一致）:所有客户端将始终看到突变写入的内容。并发成功的突变使该区域未定义但保持一致:所有客户端都看到相同的数据,但它可能不会反映任何一个突变所写的内容。通常,它由来自多个突变的混合片段组成。失败的突变使区域保持一致（因此也是未定义的）:不同的客户端可能会在不同的时间看到不同的数据。我们在下面描述我们的应用程序如何区分已定义区域和未定义区域

地区。应用程序不需要进一步区分不同种类的未定义区域。

数据突变可能是写入或记录追加。写入导致数据写入应用程序指定的文件偏移量。记录追加导致数据（“记录”）被原子地追加至少一次,即使存在并发突变,但在 GFS 选择的偏移量处（第 3.3 节）。 （相比之下,“常规”附加只是在客户认为是当前的偏移量处写入

文件结尾。）偏移量返回给客户端并标记包含记录的已定义区域的开始。

此外,GFS 可能会在两者之间插入填充或重复记录。它们占据的区域被认为是不一致的,并且通常与用户数据量相比相形见绌。

在一系列成功的变异之后,变异的文件区域保证被定义并包含最后一次变异写入的数据 10。 GFS 通过 (a) 在其所有副本上以相同的顺序对一个块应用突变（第 3.1 节）,以及 (b) 使用块版本号来检测任何变得陈旧的副本,因为它在其块服务器上错过了突变已关闭（第 4.5 节）。过时的副本永远不会参与突变或提供给向主服务器询问块位置的客户端。他们会尽早收集垃圾。

由于客户端缓存块位置,因此它们可能会在刷新该信息之前从陈旧的副本中读取。此窗口受缓存条目的超时和文件的下一次打开限制,这会从缓存中清除该文件的所有块信息。此外,由于我们的大多数文件都是仅附加的,陈旧的副本通常会返回块的过早结束而不是过时的数据。当读者重试并联系主人时,它将立即获得当前块位置。

在成功突变很久之后,组件故障当然仍然会损坏或破坏数据。 GFS 通过 master 和所有 chunkservers 之间的定期握手来识别失败的 chunkservers,并通过校验和检测数据损坏（第 5.2 节）。一旦出现问题,就会尽快从有效副本中恢复数据（第 4.3 节）。只有在 GFS 可以做出反应之前（通常在几分钟内）丢失所有副本时,块才会不可逆转地丢失。即使在这种情况下,它也是不可用的,而不是损坏的:应用程序收到明显的错误而不是损坏的数据。

2.7.2 对应用程序的影响GFS 应用程序可以适应松散

的一致性模型和其他目的已经需要的一些简单技术:依靠附加而不是覆盖、检查点和编写自我验证、自我识别的记录。

实际上,我们所有的应用程序都通过附加而不是覆盖来改变文件。在一个典型的用途中,编写器从头到尾生成一个文件。它在写入所有数据后自动将文件重命名为永久名称,或者定期检查已成功写入 10 次的数量。检查点还可以包括应用程序级校验和。读取器仅验证和处理直到最后一个检查点的文件区域,该检查点已知处于已定义状态。无论一致性和并发性问题如何,这种方法都对我们很有帮助。与随机写入相比,追加对应用程序故障的效率和弹性要高得多。检查点允许写入者重新启动增量计数并阻止读取器处理成功写入

从应用程序的角度来看仍然不完整的文件数据。

在另一个典型的用途中,许多作者并发地附加到一个文件以获得合并结果或作为生产者 - 消费者队列。 Record append 的 append-at-least-once 语义保留了每个作者的输出。读者按如下方式处理偶尔出现的填充和重复。作者准备的每条记录都包含额外的信息,如校验和,以便验证其有效性。读者可以使用校验和识别和丢弃额外的填充和记录片段。如果它不能容忍偶尔的重复（例如,如果它们会触发非幂等操作）,它可以使用记录中的唯一标识符将它们过滤掉,无论如何通常需要这些标识符来命名相应的应用程序实体,例如 Web 文档。这些用于记录 I/O 的功能（重复删除除外）位于我们的应用程序共享的库代码中,适用于 Google 的其他文件接口实现。这样,相同的记录序列,加上罕见的重复,总是被传送到记录阅读器。

3. 系统交互

我们设计的系统是为了尽量减少船长对所有操作的参与。在此背景下,我们现在描述客户端、主服务器和块服务器如何交互以实现数据突变、原子记录追加和快照。

3.1 租约和变更顺序变更是一种改变块的内容或元数据

的操作,例如写入或追加操作。每个突变都在块的所有副本上执行。

我们使用租约来维护副本之间一致的突变顺序。 master 将块租约授予其中一个副本,我们称之为 primary。 primary 为块的所有突变选择一个序列顺序。应用突变时,所有副本都遵循此顺序。因此,全局变异顺序首先由 master 选择的租约授予顺序定义,并在租约内由 primary 分配的序列号定义。

租用机制旨在最小化主服务器的管理开销。租约的初始超时为 60 秒。然而,只要块在变化,主块就可以无限期地请求并通常从主块接收扩展。这些扩展请求和授权是搭载在 master 和所有 chunkservers 之间定期交换的 HeartBeat 消息上的。

master 有时可能会在租期到期之前尝试撤销租约（例如,当 master 想要禁用正在重命名的文件上的突变时）。即使 master 失去了与 primary 的通信,它也可以在旧租约到期后安全地向另一个副本授予新租约。

在图 2 中,我们通过遵循
通过这些编号的步骤控制写入流程。

1. client 询问 master 哪个 chunkserver 持有该 chunk 的当前租约以及其他副本的位置。如果没有人有租约,则主人将租约授予它选择的副本（未显示）。

2. master 回复主副本的身份和其他（次要）副本的位置。客户端缓存此数据以备将来更改。它需要

   仅当主要时再次联系主人

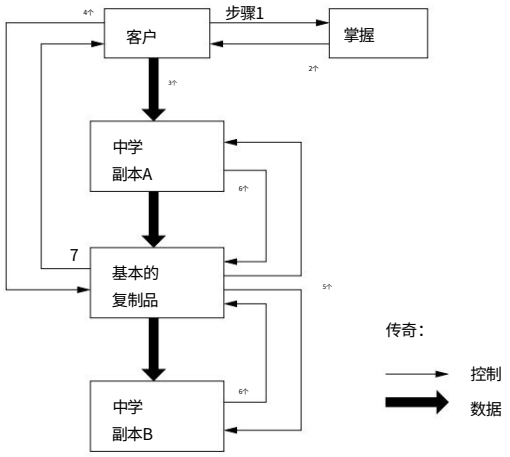


图 2:写控制和数据流

变得无法访问或回复它不再持有租约。

3. 客户端将数据推送到所有副本。客户端可以按任何顺序执行此操作。每个 chunkserver 都会将数据存储在内部 LRU 缓冲区缓存中,直到数据被使用或老化。通过将数据流与控制流分离,我们可以通过基于网络拓扑调度昂贵的数据流来提高性能,而不管哪个 chunkserver 是主要的。第 3.2 节进一步讨论了这一点。

4. 一旦所有副本确认接收到数据,客户端向主副本发送写请求。

该请求标识了之前推送到所有副本的数据。 primary 为它收到的所有 mutations 分配连续的序列号,可能来自多个 clients,这提供了必要的序列化。它按序列号顺序将突变应用于其自己的本地状态。

5. primary将写请求转发给所有的secondary replicas。每个次要副本都按照主要副本分配的相同序列号顺序应用突变。

6. secondary都回复primary表示已经完成操作。

7. 主要回复客户。在任何副本中遇到的任何错误都会报告给客户端。

如果出现错误,写入可能已在主要副本和次要副本的任意子集上成功。（如果它在主服务器上失败,它就不会被分配序列号并被转发。）

客户端请求被认为失败,修改后的区域处于不一致状态。我们的客户端代码通过重试失败的突变来处理此类错误。它将在步骤 (3) 到 (7) 中进行几次尝试,然后从写入开始回退到重试。

如果应用程序的写入很大或跨越块边界,GFS 客户端代码会将其分解为多个写入操作。它们都遵循上述控制流程,但可能会与其他客户端的并发操作交错和覆盖。因此,共享文件区域可能最终包含来自不同客户端的片段,尽管副本将是相同的,因为在所有副本上以相同的顺序成功完成了各个操作。这使文件区域处于一致但未定义的状态,如第 2.7 节所述。

3.2 数据流

我们将数据流与控制流分离,以有效地使用网络。当控制从客户端流向主服务器,然后流向所有辅助服务器时,数据以流水线方式沿着精心挑选的 chunkservers 链线性推送。我们的目标是充分利用每台机器的网络带宽,避免网络瓶颈和高延迟链路,并最大限度地减少推送所有数据的延迟。

为了充分利用每台机器的网络带宽,数据沿着 Chunkservers 链线性推送,而不是分布在某些其他拓扑结构（例如,树)中。因此,每台机器的全部出站带宽都用于传输

尽可能快地传送数据,而不是将数据分配给多个接收者。

为了尽可能避免网络瓶颈和高延迟链路（例如,交换机间链路通常两者兼而有之）,每台机器将数据转发到网络拓扑中“最近”的尚未接收到数据的机器。假设客户端正在将数据推送到 chunkservers S1 到 S4。它将数据发送到最近的块服务器,比如 S1。 S1 将它转发到最近的 chunkserver S2 到最接近 S1 的 S4,比如 S2。类似地,S2 将它转发给 S3 或 S4,以更接近 S2 的那个为准,依此类推。我们的网络拓扑非常简单,可以根据 IP 地址准确估计“距离”。

最后,我们通过流水线化 TCP 连接上的数据传输来最小化延迟。一旦一个 chunkserver 收到一些
数据,它立即开始转发。流水线对我们特别有帮助,因为我们使用具有全双工链路的交换网络。立即发送数据不会降低接收速率。在没有网络拥塞的情况下,将 B 个字节传输到 R 个副本的理想耗用时间是 B/T + RL,其中 T 是网络吞吐量,L 是在两台机器之间传输字节的延迟。我们的网络链路通常为 100 Mbps (T),而 L 远低于 1 毫秒。

因此,理想情况下,可以在 80 毫秒左右分发 1 MB。

3.3 原子记录追加

GFS 提供了一个称为记录追加的原子追加操作。在传统写入中,客户端指定要写入数据的偏移量。对同一区域的并发写入不可序列化:该区域最终可能包含来自多个客户端的数据片段。但是,在记录追加中,客户端仅指定数据。 GFS 在 GFS 选择的偏移量处以原子方式（即,作为一个连续的字节序列)将其附加到文件中至少一次,并将该偏移量返回给客户端。这类似于在 Unix 中写入以 O APPEND 模式打开的文件,当多个写入者并发时没有竞争条件。

-

记录追加被我们的分布式应用程序大量使用,在这些应用程序中,不同机器上的许多客户端同时追加到同一个文件。客户端将需要额外的复杂和昂贵的同步,例如通过分布式锁管理器,如果他们使用传统的写操作。在我们的工作负载中,此类文件经常

充当多生产者/单消费者队列或包含来自许多不同客户端的合并结果。

Record append 是一种 mutation,它遵循 3.1 节中的控制流程,仅在 primary 处增加了一点额外的逻辑。客户端将数据推送到文件最后一个块的所有副本,然后将其请求发送到主节点。主要检查将记录附加到当前块是否会导致块超过最大大小 (64 MB)。如果是这样,它将块填充到最大大小,告诉辅助节点也这样做,并回复客户端指示应该在下一个块上重试该操作。 （记录附加被限制为最多最大块大小的四分之一,以将最坏情况下的碎片保持在可接受的水平。)如果记录适合最大大小,这是常见的情况,主将数据附加到它的replica,告诉secondaries把数据写到它所在的准确offset处,最后向client回复success。

如果记录追加在任何副本上失败,客户端将重试该操作。因此,同一块的副本可能包含不同的数据,可能包括同一记录的全部或部分副本。 GFS 不保证所有的副本都是字节相同的。它只保证数据作为一个原子单元至少被写入一次。这个属性很容易从简单的观察中得出,即对于报告成功的操作,数据必须在某个块的所有副本上以相同的偏移量写入。此外,在此之后,所有副本至少与记录末尾一样长,因此即使不同的副本后来成为主副本,任何未来的记录都将被分配更高的偏移量或不同的块。就我们的一致性保证而言,成功的记录追加操作写入数据的区域是定义的（因此是一致的）,而中间区域是不一致的（因此是未定义的）。正如我们在第 2.7.2 节中讨论的那样,我们的应用程序可以处理不一致的区域。

3.4 快照

快照操作几乎在瞬间就制作了一个文件或目录树（“源”）的副本,同时最大限度地减少了正在进行的突变的任何中断。我们的用户使用它来快速创建庞大数据集的分支副本（通常是这些副本的副本,递归地）,或者在试验更改之前检查当前状态,这些更改可以在以后轻松提交或回滚。

与 AFS [5] 一样,我们使用标准的写时复制技术来实现快照。当 master 收到一个快照请求时,它首先撤销对它要快照的文件中的块的任何未完成的租约。这确保了对这些块的任何后续写入都需要与 master 交互以找到租约持有者。这将使 master 有机会首先创建块的新副本。

租约被撤销或过期后,主服务器将操作记录到磁盘。然后,它通过复制源文件或目录树的元数据,将此日志记录应用于其内存状态。新创建的快照文件指向与源文件相同的块。

客户端在快照操作后第一次想要写入块 C 时,它会向 master 发送请求以查找当前的租约持有者。 master 注意到块 C 的引用计数大于 1。它推迟回复客户端请求,而是选择一个新块

处理 C 。然后它询问每个具有当前

C 的副本以创建一个名为 C 的新块。通过在与原始块相同的块服务器上创建新块,我们确保可以在本地复制数据,而不是通过网络（我们的磁盘大约是 100 Mb 以太网链路速度的三倍）。从这一点来看,请求处理与任何块都没有区别:主服务器授予其中一个副本对新块 C 的租约,并回复客户端,客户端可以正常写入块,不知道它刚刚从现有块创建。

4. 掌握操作

master 执行所有命名空间操作。此外,它还管理整个系统中的块副本:它做出放置决策,创建新块并因此创建副本,并协调各种系统范围的活动以保持块完全复制,平衡所有块服务器之间的负载,并回收未使用的存储.我们现在讨论这些主题中的每一个。

4.1 命名空间管理和锁定

许多 master 操作可能需要很长时间:例如,快照操作必须撤销快照覆盖的所有块上的块服务器租约。我们不想在其他主操作运行时延迟它们。因此,我们允许多个操作处于活动状态,并对命名空间的区域使用锁以确保正确的序列化。

与许多传统文件系统不同,GFS 没有列出该目录中所有文件的目录数据结构。它也不支持同一文件或目录的别名（即 Unix 术语中的硬链接或符号链接）。 GFS 在逻辑上将其名称空间表示为将完整路径名映射到元数据的查找表。通过前缀压缩,可以在内存中高效地表示该表。命名空间树中的每个节点（绝对文件名或绝对目录名)都有一个关联的读写锁。

每个主操作在运行前都会获取一组锁。通常,如果它涉及 /d1/d2/.../dn/leaf,它将获取目录名称 /d1、/d1/d2、...、/d1/d2/.../dn 的读锁,以及完整路径名 /d1/d2/.../dn/leaf 上的读锁或写锁。请注意,叶可能是一个文件或目录,具体取决于操作。

我们现在说明这种锁定机制如何防止在 /home/user 被快照到 /save/user 时创建文件 /home/user/foo。快照操作在/home 和/save 上获取读锁,在/home/user 和/save/user 上获取写锁。文件创建获取 /home 和 /home/user 上的读锁,以及 /home/user/foo 上的写锁。这两个操作将被正确序列化,因为它们试图获得对 /home/user 的冲突锁。文件创建不需要对父目录进行写锁定,因为没有“目录”或类似 inode 的数据结构可以防止修改。

名称上的读锁足以保护父目录不被删除。

这种锁定方案的一个很好的特性是它允许在同一目录中进行并发更改。例如,可以在同一个目录中同时执行多个文件创建:每个文件都获取目录名的读锁和文件名的写锁。目录名称上的读锁足以防止目录被删除、重命名或快照。写入锁定

file names serialize 尝试创建同名文件两次。

由于命名空间可以有很多节点,读写锁对象被延迟分配,一旦不被使用就被删除。此外,以一致的总顺序获取锁以防止死锁:它们首先按名称空间树中的级别排序,并在同一级别内按字典顺序排列。

4.2 副本放置GFS 集群高度分布在

多个层次上。它通常有数百个 chunkservers 分布在许多机器机架上。这些 chunkservers 又可以从来自相同或不同机架的数百个客户端访问。不同机架上的两台机器之间的通信可能会跨越一个或多个网络交换机。此外,进出机架的带宽可能小于机架内所有机器的总带宽。

多级分布对分布数据的可伸缩性、可靠性和可用性提出了独特的挑战。

块副本放置策略有两个目的:最大化数据可靠性和可用性,以及最大化网络带宽利用率。对于两者来说,将副本分布在机器之间是不够的,这只能防止磁盘或机器故障并充分利用每台机器的网络带宽。我们还必须跨机架分布块副本。这确保即使整个机架损坏或离线(例如,由于网络交换机或电源电路等共享资源出现故障),块的某些副本也能存活并保持可用。这也意味着块的流量,尤其是读取,可以利用多个机架的聚合带宽。另一方面,写入流量必须流经多个机架,这是我们自愿做出的权衡。

4.3 创建、重新复制、重新平衡

chunk 副本的创建有以下三个原因: chunk cre
化、再复制和再平衡。

当 master 创建一个块时,它会选择放置最初为空的副本的位置。它考虑了几个因素。 (1) 我们希望将新副本放置在磁盘空间利用率低于平均水平的块服务器上。随着时间的推移,这将均衡跨块服务器的磁盘利用率。 (2) 我们想限制每个 chunkserver 上“最近”创建的数量。

尽管创建本身很便宜,但它可靠地预测了即将到来的大量写入流量,因为块是在写入需要时创建的,并且在我们的一次附加读取多次工作负载中,它们通常在完全写入后实际上变成只读的。 (3) 正如上面所讨论的,我们希望跨机架传播块的副本。

一旦可用副本的数量低于用户指定的目标,主节点就会重新复制一个块。发生这种情况的原因有多种:chunkserver 变得不可用,它报告其副本可能已损坏,其中一个磁盘因错误而被禁用,或者复制目标增加。每个需要重新复制的块都根据几个因素确定优先级。一个是它离复制目标有多远。例如,我们对丢失两个副本的块给予比只丢失一个副本的块更高的优先级。此外,我们更喜欢首先重新复制活动文件的块,而不是属于最近删除的文件的块(参见第 4.4 节)。最后,为了尽量减少故障对正在运行的应用程序的影响,我们提高了任何阻止客户端进程的块的优先级。

master 选择最高优先级的块并通过指示某些块服务器直接从现有的有效副本复制块数据来“克隆”它。放置新副本的目标与创建副本的目标相似:均衡磁盘空间利用率,限制任何单个 chunkserver 上的活动克隆操作,以及跨机架分布副本。

为了防止克隆流量压倒客户端流量,master 限制了集群和每个 chunkserver 的活动克隆操作的数量。此外,每个 chunkserver 通过限制其对源 chunkserver 的读取请求来限制它在每个克隆操作上花费的带宽量。

最后,主节点定期重新平衡副本:它检查当前副本分布并移动副本以获得更好的磁盘空间和负载平衡。同样通过这个过程,master 逐渐填满一个新的 chunkserver,而不是立即用新的块和随之而来的大量写入流量淹没它。新副本的放置标准与上面讨论的类似。此外,master 还必须选择要删除的现有副本。一般来说,它更喜欢删除那些空闲空间低于平均水平的 chunkservers,以平衡磁盘空间的使用。

4.4 垃圾收集文件被删除后,GFS 不

会立即回收可用的物理存储空间。它只是在文件和块级别的常规垃圾收集期间懒惰地这样做。

我们发现这种方法使系统更加简单和可靠。

4.4.1 机制当一个文件被

应用程序删除时,master 会像其他更改一样立即记录删除。然而,该文件并没有立即回收资源,而是被重命名为包含删除时间戳的隐藏名称。在 master 定期扫描文件系统命名空间期间,如果隐藏文件已经存在超过三天(时间间隔是可配置的),它将删除任何此类隐藏文件。

在此之前,该文件仍然可以在新的特殊名称下读取,并且可以通过将其重命名为正常名称来取消删除。
当隐藏文件从命名空间中删除时,它在内存中的元数据将被删除。这有效地切断了它与所有块的链接。

在对块名称空间进行类似的定期扫描时,master 识别孤立块(即无法从任何文件访问的块)并擦除这些块的元数据。在定期与 master 交换的 HeartBeat 消息中,每个 chunkserver 报告它拥有的块的一个子集,master 回复所有不再存在于 master 的元数据中的块的标识。chunkserver 可以自由删除这些块的副本。

4.4.2 讨论

虽然分布式垃圾收集是一个难题,需要在编程语言的上下文中提供复杂的解决方案,但在我们的例子中它非常简单。我们可以很容易地识别对块的所有引用:它们位于由主文件专门维护的文件到块的映射中。

我们还可以很容易地识别所有的块副本:它们是每个块服务器上指定目录下的 Linux 文件。
主人不知道的任何此类副本都是“垃圾”。

存储回收的垃圾收集方法与急切删除相比有几个优点。首先,它在组件故障常见的大规模分布式系统中简单可靠。块创建可能在某些块服务器上成功,但在其他服务器上可能不会成功,从而留下主服务器不知道存在的副本。副本删除消息可能会丢失,Master 必须记住在失败时重新发送它们,包括它自己的和 ChunkServer 的。

垃圾收集提供了一种统一且可靠的方式来清理任何未知有用的副本。其次,它将存储回收合并到 master 的常规后台活动中,例如定期扫描名称空间和与 chunkservers 握手。因此,它是分批完成的,成本是摊销的。而且,只有在主人比较空闲的时候才做。主人可以更迅速地响应需要及时关注的客户请求。第三,回收存储的延迟提供了防止意外、不可逆删除的安全网。

根据我们的经验,主要缺点是延迟有时会阻碍用户在存储空间紧张时微调使用的努力。重复创建和删除临时文件的应用程序可能无法立即重新使用存储。如果已删除的文件被再次明确删除,我们通过加快存储回收来解决这些问题。我们还允许用户对命名空间的不同部分应用不同的复制和回收策略。例如,用户可以指定某个目录树中的文件中的所有块都将在不复制的情况下存储,并且任何删除的文件都会立即且不可撤销地从文件系统状态中删除。

4.5 陈旧副本检测

如果 chunkserver 发生故障并且在它关闭时错过了对 chunk 的更改,则 chunk 副本可能会变得陈旧。为了
每个块,主服务器维护一个块版本号以区分最新和陈旧的副本。

每当 master 在块上授予新租约时,它都会增加块版本号并通知最新的副本。主服务器和这些副本都在其持久状态中记录新版本号。这发生在任何客户端被通知之前,因此在它可以开始写入块之前。如果另一个副本当前不可用,则其块版本号将不会被提升。当 chunkserver 重启并报告它的 chunk 集合和它们相关的版本号时,master 会检测到这个 chunkserver 有一个过时的副本。如果 master 看到版本号大于其记录中的版本号,则 master 认为它在授予租约时失败了,因此将更高版本更新为最新版本。

master 在其常规垃圾收集中删除陈旧的副本。在此之前,当它回复客户端对块信息的请求时,它实际上认为一个过时的副本根本不存在。作为另一种保护措施,master 在通知客户端哪个 chunkserver 持有一个块的租约时,或者当它指示一个 chunkserver 在克隆操作中从另一个 chunkserver 读取该块时,它包括块版本号。客户端或 chunkserver 在执行操作时会验证版本号,以便始终访问最新的数据。

5. 容错和诊断

我们在设计系统时面临的最大挑战之一是处理频繁的组件故障。质量和组件的数量加在一起使这些问题更常见而不是例外:我们不能完全信任机器,也不能完全信任磁盘。组件故障可能导致系统不可用,或者更糟的是,数据损坏。我们讨论了我们如何应对这些挑战,以及我们在系统中内置的工具,以便在问题不可避免地发生时对其进行诊断。

5.1 高可用性

在 GFS 集群中的数百台服务器中,有些服务器在任何给定时间都注定不可用。我们通过两个简单而有效的策略保持整个系统的高可用性:快速恢复和复制。

5.1.1 快速恢复

master 和 chunkserver 都被设计为恢复它们的状态并在几秒钟内启动,无论它们如何终止。实际上,我们不区分正常终止和异常终止;服务器通常只是通过终止进程来关闭。客户端和其他服务器在未完成的请求超时、重新连接到重新启动的服务器并重试时会遇到轻微的问题。第 6.2.2 节报告观察到的启动时间。

5.1.2 块复制

如前所述,每个块都被复制到不同机架上的多个块服务器上。用户可以为文件命名空间的不同部分指定不同的复制级别。

默认值为三个。 master 根据需要克隆现有副本,以在 chunkservers 离线或通过校验和验证检测损坏的副本时保持每个块完全复制(参见第 5.2 节)。尽管复制对我们很有帮助,但我们正在探索其他形式的跨服务器冗余,例如奇偶校验或纠删码,以满足我们不断增加的只读存储需求。我们预计在我们非常松散耦合的系统中实施这些更复杂的冗余方案具有挑战性但易于管理,因为我们的流量主要是追加和读取而不是小的随机写入。

5.1.3 Master Replication Master 状

态被复制是为了可靠性。它的操作日志和检查点被复制到多台机器上。只有在其日志记录已在本地和所有主副本上刷新到磁盘后,才认为对状态的更改已提交。为简单起见,一个主进程仍然负责所有变更以及后台活动,例如在内部更改系统的垃圾收集。

当它失败时,它几乎可以立即重新启动。如果它的机器或磁盘出现故障,GFS 外部的监控基础设施会在别处启动一个新的主进程,并使用复制的操作日志。客户端仅使用主机的规范名称(例如 gfs-test),这是一个 DNS 别名,如果主机被重新定位到另一台机器,则可以更改该别名。

此外,“影子”master 提供对文件系统的只读访问,即使在 primary master 关闭时也是如此。它们是影子,而不是镜子,因为它们可能会稍微滞后于主节点,通常是几分之一秒。它们增强了未被主动改变的文件或不介意获得略微过时结果的应用程序的读取可用性。

事实上,由于文件内容是从 chunkservers 读取的,应用程序不会观察到陈旧的文件内容。可能是什么

短窗口内的陈旧是文件元数据,如目录内容或访问控制信息。

为了让自己了解情况,影子主控读取不断增长的操作日志的副本,并将与主控完全相同的更改序列应用于其数据结构。

与主服务器一样,它在启动时（之后很少)轮询块服务器以定位块副本并与它们交换频繁的握手消息以监视它们的状态。它仅依赖于 primary master 来获取由 primary 创建和删除副本的决定所导致的副本位置更新。

5.2 数据完整性

每个 chunkserver 使用校验和来检测存储数据的损坏。鉴于 GFS 集群通常在数百台机器上有数千个磁盘,它经常会遇到磁盘故障,导致读取和写入路径上的数据损坏或丢失。 （一个原因参见第 7 节。)我们可以使用其他块副本从损坏中恢复,但是通过跨块服务器比较副本来检测损坏是不切实际的。此外,不同的副本可能是合法的:GFS 突变的语义,特别是前面讨论的原子记录追加,不保证相同的副本。因此,每个 chunkserver 必须通过维护校验和来独立验证自己副本的完整性。

块被分成 64 KB 的块。每个都有相应的 32 位校验和。与其他元数据一样,校验和保存在内存中并与日志记录一起永久存储,与用户数据分开。

对于读取,chunkserver 在将任何数据返回给请求者之前验证与读取范围重叠的数据块的校验和,无论是客户端还是另一个 chunkserver。

因此 chunkservers 不会将损坏传播到其他机器。如果块与记录的校验和不匹配,chunkserver 会向请求者返回一个错误,并向 master 报告不匹配。作为响应,请求者将从其他副本读取,而主服务器将从另一个副本克隆块。在一个有效的新副本就位后,master 指示报告不匹配的 chunkserver 删除其副本。

由于多种原因,校验和对读取性能几乎没有影响。由于我们的大部分读取至少跨越几个块,因此我们只需要读取和校验和校验相对少量的额外数据以进行验证。 GFS 客户端代码通过尝试在校验和块边界对齐读取进一步减少了这种开销。此外,chunkserver 上的校验和查找和比较是在没有任何 I/O 的情况下完成的,并且校验和计算通常可以与 I/O 重叠。

校验和计算针对附加到块末尾的写入（与覆盖现有数据的写入相反)进行了高度优化,因为它们在我们的工作负载中占主导地位。我们只是增量地更新最后一个部分校验和块的校验和,并为追加填充的任何全新校验和块计算新的校验和。即使最后一个部分校验和块已经损坏并且我们现在无法检测到它,新的校验和值也不会与存储的数据匹配,并且在下一次读取块时会像往常一样检测到损坏。

相反,如果写入覆盖了块的现有范围,我们必须读取并验证被覆盖范围的第一个和最后一个块,然后执行写入,并且最后计算并记录新的校验和。如果我们在部分覆盖之前不验证第一个和最后一个块,新的校验和可能会隐藏未被覆盖区域中存在的损坏。

在空闲期间,块服务器可以扫描并验证非活动块的内容。这使我们能够检测很少读取的块中的损坏。一旦检测到损坏,master 就可以创建一个新的未损坏的副本并删除损坏的副本。这可以防止不活动但已损坏的块副本欺骗主节点,使其认为它具有足够的块有效副本。

5.3 诊断工具

广泛而详细的诊断日志记录在问题隔离、调试和性能分析方面提供了不可估量的帮助,同时仅产生最低限度的成本。没有日志,很难理解机器之间短暂的、不可重复的交互。 GFS 服务器生成诊断日志,记录许多重要事件（例如 chunkservers 启动和关闭)和所有 RPC 请求和回复。这些诊断日志可以随意删除而不影响系统的正确性。但是,我们尽量在空间允许的范围内保留这些日志。

RPC 日志包括在线路上发送的确切请求和响应,但正在读取或写入的文件数据除外 10。通过匹配请求与回复并整理不同机器上的 RPC 记录,我们可以重建整个交互历史来诊断问题。日志还用作负载测试和性能分析的跟踪。

日志记录对性能的影响很小（并且远远超过其好处） ,因为这些日志是按顺序和异步写入的。最近的事件也保存在内存中,可用于连续在线监控。

6. 测量在本节中,我们提供了一些微基

准来说明 GFS 体系结构和实现中固有的瓶颈,以及来自谷歌使用的真实集群的一些数据。

6.1 微基准

我们测量了一个 GFS 集群的性能,该集群由一个主节点、两个主节点副本、16 个块服务器和 16 个客户端组成。请注意,设置此配置是为了便于测试。典型的集群有数百个 chunkservers 和数百个客户端。

所有机器都配置有双 1.4 GHz PIII 处理器、2 GB 内存、两个 80 GB 5400 rpm 磁盘和一个 100 Mbps 全双工以太网连接到 HP 2524 交换机。所有 19 台 GFS 服务器机器都连接到一台交换机,所有 16 台客户端机器都连接到另一台交换机。两台交换机通过 1 Gbps 链路连接。

6.1.1 读取

N 个客户端同时从文件系统读取。每个客户端从 320 GB 的文件集中读取随机选择的 4 MB 区域。这将重复 256 次,以便每个客户端最终读取 1 GB 的数据。 chunkservers 加在一起只有 32 GB 的内存,所以我们预计 Linux 缓冲区缓存的命中率最多为 10%。我们的结果应该接近冷缓存结果。

图 3(a) 显示了 N 个客户端的总读取率及其理论限制。当两个交换机之间的 1 Gbps 链路饱和时,限制峰值达到 125 MB/s,或者当其 100 Mbps 网络接口饱和时,每个客户端达到 12.5 MB/s,以适用者为准。当只有一个客户端在读取时,观察到的读取速率为 10 MB/s,或每个客户端限制的 80%。总读取速率达到 94 MB/s,大约是 125 MB/s 链接限制的 75%,对于 16 个读取器,或每个客户端 6 MB/s。效率从 80% 下降到 75%,因为随着读取器数量的增加,多个读取器同时从同一个 chunkserver 读取的概率也会增加。

簇	A	乙
块服务器	342	227
可用磁盘空间	72 TB 180 TB	
已用磁盘空间	55TB 155TB	735k 737k
文件数	22k 232k	992k 1550k
死文件数		
块数		
块服务器上的元数据	13GB 21GB	
master 的元数据	48MB 60MB	

表 2:两个 GFS 集群的特征

6.1.2 写入

N 个客户端同时写入 N 个不同的文件。每个客户端在一系列 1 MB 的写入中将 1 GB 的数据写入一个新文件。聚合写入速率及其理论极限如图 3(b) 所示。限制稳定在 67 MB/s,因为我们需要将每个字节写入 16 个块服务器中的 3 个,每个都有 12.5 MB/s 的输入连接。

一个客户端的写入速率为 6.3 MB/s,大约是限制的一半。造成这种情况的罪魁祸首是我们的网络堆栈。它与我们用于将数据推送到块副本的流水线方案不能很好地交互。将数据从一个副本传播到另一个副本的延迟会降低整体写入速率。

16 个客户端的总写入速率达到 35 MB/s（或每个客户端 2.2 MB/s），大约是理论极限的一半。与读取的情况一样,随着客户端数量的增加,多个客户端更有可能同时写入同一个 chunkserver。此外,16 个写入者比 16 个读取者更容易发生冲突,因为每次写入都涉及三个不同的副本。

写入比我们想要的要慢。在实践中,这并不是一个主要问题,因为即使它增加了单个客户端所看到的延迟,它也不会显着影响系统向大量客户端提供的聚合写入带宽。

6.1.3 记录追加

图 3(c) 显示了记录附加性能。 N 个客户端同时追加到一个文件。性能受限于存储文件最后一块的块服务器的网络带宽,与客户端数量无关。一个客户端的速度从 6.0 MB/s 开始,16 个客户端的速度下降到 4.8 MB/s,这主要是由于拥塞和不同客户端看到的网络传输速率差异所致。

我们的应用程序目前倾向于生成多个此类文件。换句话说,N个客户端同时附加到M个共享文件,其中N和M都是几十或几百。因此,我们实验中的 chunkserver 网络拥塞在实践中并不是一个重要问题,因为客户端可以在写入一个文件时取得进展,而另一个文件的 chunkservers 却很忙。

6.2 真实世界的集群我们现在检查谷歌内部使

用的两个集群,它们代表了其他几个类似的集群。集群 A 被 100 多名工程师定期用于研究和开发。一个典型的任务由人类用户启动并运行长达几个小时。它读取几 MB 到几 TB 的数据,转换或分析数据,并将结果写回集群。 Cluster B主要用于生产数据处理。任务持续时间长

只需偶尔的人为干预,就可以更长时间地连续生成和处理多 TB 数据集。在这两种情况下,单个“任务”由许多机器上的许多进程同时读取和写入许多文件组成。

6.2.1 存储如表中的前

五个条目所示,两个集群都有数百个 chunkserver,支持数 TB 的磁盘空间,而且还算满,但不是完全满。 “已用空间”包括所有块副本。几乎所有文件都被复制了三次。因此,集群分别存储了18TB和52TB的文件数据。

这两个集群具有相似数量的文件,但 B 具有更大比例的死文件,即已删除或被新版本替换但其存储尚未回收的文件。它也有更多的块,因为它的文件往往更大。

6.2.2 元数据

chunkservers 总共存储了数十 GB 的元数据,主要是 64 KB 用户数据块的校验和。
保存在块服务器中的唯一其他元数据是第 4.5 节中讨论的块版本号。

保存在 master 中的元数据要小得多,只有几十 MB,或者平均每个文件大约 100 字节。这与我们的假设一致,即主存储器的大小在实践中不会限制系统的容量。

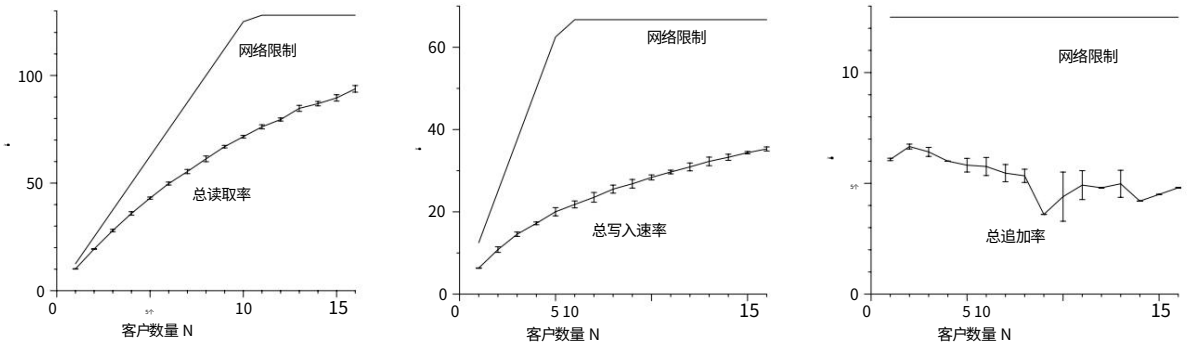
每个文件的大部分元数据是以前缀压缩形式存储的文件名。其他元数据包括文件所有权和权限、从文件到块的映射以及每个块的当前版本。此外,对于每个块,我们存储当前副本位置和用于实现写时复制的引用计数。

每个单独的服务器,包括块服务器和主服务器,只有 50 到 100 MB 的元数据。因此恢复速度很快:在服务器能够回答查询之前从磁盘读取元数据只需要几秒钟。然而,master 在一段时间内有点受阻　通常是 30 到 60 秒　直到它从所有 chunkservers 获取了 chunk 位置信息。

6.2.3 读写速率

表 3 显示了不同时间段的读写速率。在进行这些测量时,这两个集群都已经运行了大约一周。 （集群最近已重新启动以升级到新版本的 GFS。）

自重新启动以来,平均写入速率低于 30 MB/s。当我们进行这些测量时,B 正处于突发写入活动的中间,生成大约 100 MB/s 的数据,这产生了 300 MB/s 的网络负载,因为写入被传播到三个副本。



(一)读 (b) 写入 (c) 记录追加

图 3:聚合吞吐量。顶部曲线显示了我们的网络拓扑施加的理论限制。底部曲线显示测量的吞吐量。他们有显示 95% 置信区间的误差条,在某些情况下由于测量方差小而难以辨认。

簇	A	乙
阅读率（最后一分钟）	583 兆字节/秒	380 兆字节/秒
阅读率（过去一小时）	562 兆字节/秒	384 兆字节/秒
读取速率（重启后）	589 兆字节/秒	49 兆字节/秒
写入速率（最后一分钟）	1 兆字节/秒	101 兆字节/秒
写入率（最后一小时）	2 兆字节/秒	117 兆字节/秒
写入速率（重启后）	25 兆字节/秒	13 兆字节/秒
主操作（最后一分钟）	325 次操作/秒	533 次操作/秒
主操作（最后一小时）	381 次操作/秒	518 次操作/秒
主操作（重启后）	202 次操作/秒	347 次操作/秒

表 3:两个 GFS 集群的性能指标

读取速率远高于写入速率。
正如我们假设的那样,总工作负载包含的读取次数多于写入次数。两个集群都处于大量读取活动的中间。特别是,A 在前一周一直保持 580 MB/s 的读取速率。它的网络配置可以支持 750 MB/s,因此它可以有效地利用其资源。集群 B 可以支持 1300 MB/s 的峰值读取速率,但其应用程序仅使用 380 MB/s。

6.2.4 主负载

表 3 还显示发送到主服务器的操作速率约为每秒 200 到 500 次操作。 master 可以很容易地跟上这个速度,因此不是这些工作负载的瓶颈。

在早期版本的 GFS 中,master 偶尔会成为某些工作负载的瓶颈。它花费大部分时间顺序扫描大型目录（其中包含数十万个文件)以查找特定文件。从那以后,我们更改了主数据结构,以允许通过命名空间进行高效的二进制搜索。它现在可以轻松支持每秒数千次文件访问。如果有必要,我们可以通过在名称空间数据结构前面放置名称查找缓存来进一步加快速度。

6.2.5 恢复时间

在一个 chunkserver 发生故障后,一些 chunk 将变得复制不足并且必须被克隆以恢复它们的复制级别。恢复所有此类块所需的时间取决于资源量。在一个实验中,我们杀死了集群 B 中的一个 chunkserver。该 chunkserver 大约有
包含 600 GB 数据的 15,000 个块。为了限制对正在运行的应用程序的影响并为调度决策提供余地,我们的默认参数将此集群限制为 91 个并发克隆（块服务器数量的 40%）,其中每个克隆操作最多允许消耗 6.25 MB/s（ 50 兆比特/秒）。所有块都在 23.2 分钟内恢复,有效复制速率为 440 MB/s。

在另一个实验中,我们杀死了两个块服务器,每个块服务器大约有 16,000 个块和 660 GB 的数据。这种双重故障将 266 个块减少为具有单个副本。这 266 个 chunk 以更高的优先级克隆,并在 2 分钟内全部恢复到至少 2x 复制,从而使集群处于可以容忍另一个 chunkserver 故障而不会丢失数据的状态。

6.3 工作负载分解

在本节中,我们详细介绍了两个 GFS 集群上的工作负载,与第 6.2 节中的工作负载相当但不相同。集群 X 用于研发,而集群 Y 用于生产数据处理。

6.3.1 方法论和注意事项

这些结果仅包括客户端发起的请求,因此它们反映了我们的应用程序为整个文件系统生成的工作负载。它们不包括用于执行客户端请求或内部后台活动（例如转发写入或重新平衡)的服务器间请求。

I/O 操作的统计信息基于从 GFS 服务器记录的实际 RPC 请求中启发式重建的信息。例如,GFS 客户端代码可能会将一次读取分解为多个 RPC 以增加并行性,我们可以从中推断出原始读取。由于我们的访问模式是高度程式化的,我们希望任何错误都在噪音中。应用程序的显式日志记录可能会提供稍微更准确的数据,但从逻辑上讲不可能重新编译并重新启动数千个正在运行的客户端来这样做,而且从这么多机器收集结果也很麻烦。

人们应该注意不要过度概括我们的工作量。由于谷歌完全控制了 GFS 及其应用程序,因此应用程序倾向于针对 GFS 进行调优,而反过来 GFS 就是为这些应用程序设计的。一般应用程序之间也可能存在这种相互影响

手术	读	写	记录追加	
簇	XYXYX			是
0K	0.4 2.6 0 0 0 0 0 0 0 0.1 4.1 6.6			
1B..1K	4.9 0.2 65.2 2 39.9 0 45.1 0 7.8			0
1K..8K	43.0 78.0 78.0 2 31.1 0 1.1 0 0.2			9.2
8K..64K	0.2 0.2 0.2 0.3 31.6 0.4 <.1 0.1			
64K..128K	0.1 0.1 0.1 0.1 7.0 0.4 7.2 7.2			
128K..256K				15.2
256K..512K				2.8
512K..1M				4.3
1M..inf				10.6 31.2 25.5 2.2

表 4:按规模 (%) 划分的运营细分。对于读取,大小是实际读取和传输的数据量,而不是请求的数据量。

手术	读	写	记录追加	
簇	XYXYX			是
1B..1K	< .1 < .1 < .1 < .1 < .1 13.8 3.9 <			< .1
1K..8K	.1 < .1 < .1 11.4 9.3 2.4 5.9 2.3			0.1
8K..64K	0.3 0.7 0.3 0.3 2.2 7 0 1.8 1.6 0.5			0.3
64K..128K	7.7 < .1 65.9 55.1 74.1 58.0 .1			1.2
128K..256K				5.8
256K..512K				38.4
512K..1M				46.8
1M..inf	6.4 30.1	3.3 28.0 53.9		7.4

表 5:按操作大小 (%) 的传输字节数细分。对于读取,大小是实际读取和传输的数据量,而不是请求的数据量。如果读取尝试读取文件末尾以外的内容,这两者可能会有所不同,这在我们的工作负载中并不少见。

和文件系统,但效果可能更明显
我们的案例。

6.3.2 块服务器工作负载

表 4 显示了按规模划分的操作分布。读取大小呈现双峰分布。小读取（小于 64 KB)来自搜索密集型客户端,这些客户端在大文件中查找小块数据。大读取（超过 512 KB)来自对整个文件的长时间顺序读取。

在集群 Y 中,大量读取根本不返回任何数据。我们的应用程序,尤其是生产系统中的应用程序,通常使用文件作为生产者-消费者队列。生产者并发追加到文件,而消费者读取文件末尾。有时,当消费者超过生产者时,不会返回任何数据。Cluster X 较少显示这一点,因为它通常用于短期数据分析任务,而不是长期分布式应用程序。

写入大小也呈现双峰分布。大型写入（超过 256 KB)通常是由写入器中的大量缓冲区引起的。缓冲较少数据、更频繁地检查点或同步,或者只是生成较少数据的写入器占较小的写入（小于 64 KB）。

至于记录追加,集群 Y 看到的大记录追加百分比比集群 X 高得多,因为我们使用集群 Y 的生产系统针对 GFS 进行了更积极的调整。

表 5 显示了各种规模的操作中传输的数据总量。对于各种操作,较大的操作（超过 256 KB)通常占传输字节的大部分。由于随机查找工作负载,小型读取（小于 64 KB)确实会传输一小部分但很重要的读取数据。

6.3.3 附加与写入记录附加被大量使用,尤其

是在我们的生产系统中。对于集群 X,写入与记录追加的比率按传输的字节数为 108:1,按操作计数为 8:1。对于生产系统使用的集群 Y,比率分别为 3.7:1 和 2.5:1。此外,这些比率表明,对于这两个集群,记录追加往往大于写入。然而,对于集群 X,在测量期间记录追加的总体使用率相当低,因此结果可能会受到一两个具有特定缓冲区大小选择的应用程序的影响。

正如预期的那样,我们的数据突变工作负载主要是附加而不是覆盖。我们测量了主副本上覆盖的数据量。这个应用程序

簇	XY
打开	26.1 16.3
删除	0.7 1.5
查找位置	64.3 65.8
寻找租约持有人	7.8 13.4
查找匹配文件	0.6 2.2
所有其他组合	0.5 0.8

表 6:按类型划分的主请求细分 (%)

近似于客户端故意覆盖以前写入的数据而不是附加新数据的情况。对于集群 X,覆盖占不到 0.0001% 的变异字节和不到 0.0003% 的变异操作。对于集群 Y,比率均为 0.05%。虽然这很小,但仍然高于我们的预期。事实证明,这些覆盖中的大部分都是由于错误或超时导致的客户端重试。它们本身不是工作负载的一部分,而是重试机制的结果。

6.3.4 主工作负载

表 6 显示了按向主站发出的请求类型的细分。大多数请求要求块位置 (FindLocation) 用于读取和租赁持有人信息 (FindLease Locker) 用于数据突变。

集群 X 和 Y 看到明显不同的删除请求数量,因为集群 Y 存储定期重新生成并替换为更新版本的生产数据集。这种差异的一部分进一步隐藏在打开请求的差异中,因为旧版本的文件可能会通过从头开始写入而被隐式删除（Unix 打开术语中的模式“w”）。

FindMatchingFiles 是一种模式匹配请求,支持“ls”和类似的文件系统操作。与对 master 的其他请求不同,它可能处理命名空间的大部分,因此可能很昂贵。集群 Y 更频繁地看到它,因为自动化数据处理任务倾向于检查文件系统的各个部分以了解全局应用程序状态。相比之下,集群 X 的应用程序处于更明确的用户控制之下,并且通常提前知道所有需要的文件的名称。

7. 经验

在构建和部署 GFS 的过程中,我们遇到了各种各样的问题,有操作上的,也有技术上的。

最初,GFS 被设想为我们生产系统的后端文件系统。随着时间的推移,用法演变为包括研究和开发任务。它开始时对权限和配额之类的东西支持很少,但现在包括这些的基本形式。虽然生产系统受到良好的纪律和控制,但用户有时却并非如此。需要更多的基础设施来防止用户相互干扰。

我们最大的一些问题与磁盘和 Linux 相关。
我们的许多磁盘都向 Linux 驱动程序声称它们支持一系列 IDE 协议版本,但实际上只对较新的版本做出可靠响应。由于协议版本非常相似,这些驱动器大部分都可以工作,但偶尔不匹配会导致驱动器和内核对驱动器的状态不一致。由于内核中的问题,这会无声地破坏数据。这个问题促使我们使用校验和来检测数据损坏,同时我们修改了内核来处理这些协议不匹配。

早些时候,由于 fsync() 的成本,我们在 Linux 2.2 内核中遇到了一些问题。它的成本与文件的大小成正比,而不是与修改部分的大小成正比。这对我们的大型操作日志来说是一个问题,尤其是在我们实施检查点之前。我们通过使用同步写入解决了这个问题,并最终迁移到 Linux 2.4。

另一个 Linux 问题是单个读写器锁,地址空间中的任何线程在从磁盘调入页面（读取器锁)或在 mmap() 调用中修改地址空间（写入器锁)时都必须持有该锁。我们在轻负载下看到了系统中的短暂超时,并努力寻找资源瓶颈或零星的硬件故障。甚至最终,我们发现当磁盘线程在先前映射的数据中进行分页时,这个单一的锁会阻止主网络线程将新数据映射到内存中。

由于我们主要受限于网络接口而不是内存复制带宽,因此我们通过将 mmap() 替换为 pread() 来解决这个问题,代价是额外复制一次。

尽管偶尔会出现问题,但 Linux 代码的可用性一次又一次地帮助我们探索和理解系统行为。在适当的时候,我们会改进内核并与开源社区分享更改。

8. 相关工作

与 AFS [5] 等其他大型分布式文件系统一样,GFS 提供了一个位置独立的命名空间,它使数据能够透明地移动以实现负载平衡或容错。与 AFS 不同,GFS 以更类似于 xFS [1] 和 Swift [3] 的方式跨存储服务器传播文件数据,以提供聚合性能和更高的容错能力。

由于磁盘相对便宜并且复制比更复杂的 RAID [9] 方法更简单,因此 GFS 目前仅使用复制来实现冗余,因此比 xFS 或 Swift 消耗更多的原始存储。

与 AFS、xFS、Frangipani [12] 和 Intermezzo [6] 等系统相比,GFS 不在文件系统接口下提供任何缓存。我们的目标工作负载在单个应用程序运行中几乎没有重用,因为它们要么流经大型数据集,要么在其中随机查找并每次读取少量数据。

一些分布式文件系统如 Frangipani、xFS、Min nesota 的 GFS[11] 和 GPFS [10] 去除了中心化服务器并依靠分布式算法来实现一致性和管理。我们选择集中式方法以简化设计、提高可靠性并获得灵活性。特别是,集中式主机使得实施复杂的块放置和复制策略变得更加容易,因为主机已经拥有大部分相关信息并控制其更改方式。我们通过保持主状态较小并在其他机器上完全复制来解决容错问题。可扩展性和高可用性（用于读取)目前由我们的影子主机制提供。通过附加到预写日志,使对主状态的更新持久化。因此,我们可以采用像 Harp [7] 中的那样的主副本方案来提供比我们当前方案更强的一致性保证的高可用性。

在为大量客户端提供聚合性能方面,我们正在解决类似于 Lustre [8] 的问题。然而,我们通过专注于我们应用程序的需求而不是构建一个 POSIX 兼容的文件系统来显着简化问题。此外,GFS 假设有大量不可靠的组件,因此容错是我们设计的核心。

GFS 最类似于 NASD 架构 [4]。
虽然 NASD 架构基于网络连接的磁盘驱动器,但 GFS 使用商品机器作为块服务器,就像在 NASD 原型中所做的那样。与 NASD 的工作不同,我们的块服务器使用延迟分配的固定大小块而不是可变长度对象。此外,GFS 还实现了生产环境所需的重新平衡、复制和恢复等功能。

与明尼苏达州的 GFS 和 NASD 不同,我们不寻求更改存储设备的型号。我们专注于解决具有现有商品组件的复杂分布式系统的日常数据处理需求。

由原子记录追加启用的生产者-消费者队列解决了与 River [2] 中的分布式队列类似的问题。 River 使用跨机器分布的基于内存的队列和仔细的数据流控制,而 GFS 使用可以由许多生产者同时附加的持久文件。 River 模型支持 m-to-n 分布式队列,但缺乏持久存储的容错能力,而 GFS 仅高效支持 m-to-1 队列。多个消费者可以读取同一个文件,但他们必须协调以划分传入的负载。

9.结论

Google 文件系统展示了在商用硬件上支持大规模数据处理工作负载所必需的品质。虽然一些设计决策特定于我们独特的环境,但许多可能适用于类似规模和成本意识的数据处理任务。

我们首先根据我们当前和预期的应用程序工作负载和技术环境重新检查传统文件系统假设。我们的观察导致了设计空间中截然不同的点。

我们将组件故障视为常态而不是例外,针对大部分附加（可能同时)然后读取（通常顺序)的大文件进行优化,同时扩展和放宽标准文件系统接口以改进整个系统。

我们的系统通过持续监控、复制关键数据以及快速自动恢复来提供容错能力。 chunk 复制允许我们容忍 chunkserver

失败。这些故障的频率激发了一种新颖的在线修复机制，该机制定期且透明地修复损坏并尽快补偿丢失的副本。此外，我们使用校验和来检测磁盘或 IDE 子系统级别的数据损坏，考虑到系统中的磁盘数量，这种情况变得太常见了。

我们的设计为执行各种任务的许多并发读取器和写入器提供了高聚合吞吐量。
我们通过将通过主服务器的文件系统控制与直接在块服务器和客户端之间传递的数据传输分开来实现这一点。大块大小和块租约最大限度地减少了主节点对常见操作的参与，这将权限委托给数据突变中的主要副本。这使得不会成为瓶颈的简单、集中的主控成为可能。

我们相信，我们网络堆栈的改进将解除当前对单个客户端看到的写入吞吐量的限制。

GFS成功满足了我们的存储需求，在谷歌内部被广泛用作研发和生产数据处理的存储平台。它是一个重要的工具，使我们能够在整个网络的规模上继续创新和解决问题。

致谢

我们要感谢以下人员对系统或论文的贡献。 Brain Bershad（我们的牧羊人）和匿名审稿人给了我们宝贵的意见和建议。 Anurag Acharya、Jeff Dean 和 David des Jardins 为早期设计做出了贡献。 Fay Chang 致力于跨 chunkservers 的副本比较。 Guy Ed jlali 致力于存储配额。 Markus Gutschke 致力于测试框架和安全增强。 David Kramer 致力于性能增强。 Fay Chang、Urs Hoelzle、Max Ibel、Sharon Perl、Rob Pike 和 Debby Wallach 对本文的早期草稿进行了评论。我们在谷歌的许多同事勇敢地将他们的数据托付给了一个新的文件系统，并给了我们有用的反馈。 Yoshka 帮助进行了早期测试。

参考

[1] 托马斯·安德森、迈克尔·达林、珍娜·尼夫，
大卫·帕特森、德鲁·罗塞利和伦道夫·王。
无服务器网络文件系统。在第 15 届 ACM 操作系统原理研讨会论文集中，第 109-126 页，科罗拉多州铜山度假村，1995 年 12 月。

[2] Remzi H. Arpaci-Dusseau、Eric Anderson、Noah Treuhaft、David E. Culler、Joseph M. Hellerstein、David Patterson 和 Kathy Yelick。 River 的集群 I/O：使快速案例变得普遍。第六届并行和分布式系统输入/输出研讨会论文集 (IOPADS 99)，第 10-22 页，佐治亚州亚特兰大，1999 年 5 月。

[3] 路易斯·费利佩·卡布雷拉和达雷尔·德·隆。 Swift：使用分布式磁盘条带化提供高 I/O 数据速率。计算机系统，4(4):405–436, 1991。

[4] Garth A. Gibson、David F. Nagle、Khalil Amiri、Jeff Butler、Fay W. Chang、Howard Gobioff、Charles Hardin、Erik Riedel、David Rochberg 和 Jim Zelenka。具有成本效益的高带宽存储建筑学。在第 8 届编程语言和操作系统架构支持会议记录中，第 92-103 页，加利福尼亚州圣何塞，1998 年 10 月。

[5] 约翰·霍华德、迈克尔·卡扎尔、雪莉·梅内斯、大卫·尼科尔斯、马哈德夫·萨蒂亚纳拉亚南、罗伯特·西德博瑟姆和迈克尔·韦斯特。分布式文件系统中的规模和性能。 ACM 计算机系统交易，6(1):51–81，1988 年 2 月。

[6] 间奏曲。 http://www.inter-mezzo.org，2003 年。

[7] Barbara Liskov、Sanjay Ghemawat、Robert Gruber、Paul Johnson、Liuba Shrira 和 Michael Williams。
Harp 文件系统中的复制。在第 13 届操作系统原理研讨会上，第 226-238 页，加利福尼亚州太平洋丛林市，1991 年 10 月。

[8] 光泽。 http://www.lustreorg，2003 年。

[9] David A. Patterson、Garth A. Gibson 和 Randy H. 卡茨。廉价磁盘冗余阵列 (RAID) 的案例。在 1988 年 ACM SIGMOD 数据管理国际会议记录中，第 109-116 页，芝加哥，伊利诺伊州，1988 年 9 月。

[10] 弗兰克·施穆克和罗杰·哈斯金。通用文件系统：A
用于大型计算集群的共享磁盘文件系统。在第一届 USENIX 文件和存储技术会议记录中，第 231–244 页，加利福尼亚州蒙特雷，2002 年 1 月。

[11] Steven R. Soltis、Thomas M. Ruwart 和 Matthew T. 奥基夫。全球文件系统。在第五届美国宇航局戈达德太空飞行中心海量存储系统和技术会议记录中，马里兰州大学公园市，1996 年 9 月。

[12] Chandramohan A. Thekkath、Timothy Mann 和 爱德华·K·李。 Frangipani：一个可扩展的分布式文件系统。在第 16 届 ACM 操作系统原理研讨会论文集中，第 224-237 页，法国圣马洛，1997 年 10 月。