Google 文件系统

Sanjay Ghemawat、Howard Gobioff 和梁舜德

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。它能够在廉价的商用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。

虽然我们的设计与之前的分布式文件系统有许多相同的目标，但我们的设计是基于对当前和预期的应用工作负载和技术环境的观察，这与之前的一些文件系统假设截然不同。这促使我们重新审视传统的选择，并探索截然不同的设计点。

该文件系统成功满足了我们的存储需求。它在 Google 内部被广泛部署，作为我们服务以及需要海量数据集的研发工作所需的数据生成和处理的存储平台。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，可供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，并报告了微基准和实际使用情况的测量结果。

类别和主题描述

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，美国纽约州博尔顿码头。版权所有 2003 ACM 1-58113-757-5/03/0010 ...$5.00。

1. 引言

我们设计并实现了 Google 文件系统 (GFS)，以满足 Google 快速增长的数据处理需求。GFS 与之前的分布式文件系统有着许多相同的目标，例如性能、可扩展性、可靠性和可用性。然而，GFS 的设计是基于我们对当前和预期的应用工作负载和技术环境的关键观察，这与之前的一些文件系统设计假设截然不同。我们重新审视了传统的选择，并在设计领域探索了截然不同的点。

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采用廉价商用部件构建的存储设备组成，并由数量相当的客户端设备访问。组件的数量和质量几乎决定了某些组件在任何给定时间都无法正常工作，而某些组件则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成为系统不可或缺的一部分。

其次，按照传统标准，文件体积巨大。数 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 TB 级快速增长的数据集时，即使文件系统能够支持，管理数十亿个 KB 左右大小的文件也十分困难。因此，必须重新审视 I/O 操作和块大小等设计假设和参数。

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修改的。文件中几乎不存在随机写入。一旦写入，文件就只能读取，而且通常只能顺序读取。各种数据都具有这些特征。有些数据可能构成供数据分析程序扫描的大型存储库。有些数据可能是正在运行的应用程序持续生成的数据流。有些数据可能是存档数据。有些数据可能是在一台机器上生成并在另一台机器上处理的中间结果，无论是同时处理还是稍后处理。鉴于这种对大型文件的访问模式，附加操作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则失去了吸引力。

第四，共同设计应用程序和文件系统 API 可以提高我们的灵活性，从而使整个系统受益。

例如，我们放宽了 GFS 的一致性模型，从而大大简化了文件系统，同时又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，使多个客户端可以同时向一个文件追加数据，而无需在它们之间进行额外的同步。本文稍后将更详细地讨论这些内容。

目前已部署多个 GFS 集群用于不同用途。最大的集群拥有超过 1000 个存储节点、超过 300 TB 的磁盘存储空间，并且数百个客户端在不同机器上持续频繁访问。

2. 设计概述

2.1 假设

在设计满足我们需求的文件系统时，我们遵循了一些假设，这些假设既带来了挑战，也带来了机遇。我们之前提到了一些关键的观察结果，现在我们将更详细地阐述这些假设。

- 该系统由许多经常发生故障的廉价商用组件构成。它必须持续监控自身，并定期检测、容忍并迅速恢复组件故障。
- 系统存储少量大型文件。我们预计会有数百万个文件，每个文件通常大小为 100 MB 或更大。多 GB 大小的文件很常见，应该能够高效管理。小文件必须支持，但我们无需针对它们进行优化。
- 工作负载主要包含两种读取：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 的数据，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以任意偏移量读取几 KB 的数据。注重性能的应用程序通常会对小型读取进行批处理和排序，以便稳步地遍历文件，而不是来回移动。
- 工作负载还包含许多大型、连续的写入操作，用于将数据追加到文件。典型的操作大小与读取操作类似。文件一旦写入，很少会再次被修改。支持在文件中任意位置进行小规模写入，但不必非常高效。
- 系统必须高效地实现定义明确的语义，以便多个客户端同时向同一文件追加数据。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）将并发地向一个文件追加数据。最小同步开销的原子性至关重要。该文件可能稍后被读取，也可能消费者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注重高速批量处理数据，而很少有应用程序对单次读取或写入的响应时间有严格的要求。

2.2 接口

GFS 提供了一个熟悉的文件系统接口，尽管它没有实现像 POSIX 这样的标准 API。文件按层次结构组织在目录中，并通过路径名标识。我们支持以下常用操作：*创造*，*删除*，*打开*，*关闭*，*读*，和 *写*菲莱斯。

此外，GFS 还*快照*和 *记录追加*操作。快照操作以低成本创建文件或目录树的副本。记录追加操作允许多个客户端同时向同一文件追加数据，同时保证每个客户端追加操作的原子性。它对于实现多路合并结果和生产者-消费者队列非常有用，多个客户端可以同时向其追加数据而无需额外的锁定。我们发现这些类型的文件在构建大型分布式应用程序中非常有价值。快照操作和记录追加操作将分别在 3.4 节和 3.3 节中进一步讨论。

2.3 架构

GFS 集群由单个 *掌握*以及多个 *块服务器*并由多个访问 *客户*如图 1 所示。这些通常都是运行用户级服务器进程的商用 Linux 机器。在同一台机器上同时运行块服务器和客户端很容易，只要机器资源允许，并且运行可能不稳定的应用程序代码所导致的可靠性降低是可以接受的。

文件被分成固定大小 *块*. 每个块由一个不可变且全局唯一的 64 位 *块句柄*由主服务器在创建块时分配。块服务器将块作为 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。为了提高可靠性，每个块都会在多个块服务器上进行复制。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，包括命名空间、访问控制信息、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，例如块租约管理、孤立块的垃圾回收以及块服务器之间的块迁移。Master 会定期与每个块服务器进行通信。*心跳* 消息来向它发出指令并收集它的状态。

链接到每个应用程序的 GFS 客户端代码实现了文件系统 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。客户端与主服务器交互以进行元数据操作，但所有数据通信都直接发送到块服务器。我们不提供 POSIX API，因此无需连接到 Linux vnode 层。

客户端和块服务器都不会缓存文件数据。客户端缓存几乎没有什么用处，因为大多数应用程序会处理大型文件，或者工作集太大而无法缓存。取消客户端缓存可以简化客户端和整个系统，因为它消除了缓存一致性问题。（不过，客户端会缓存元数据。）块服务器不需要缓存文件数据，因为块存储为本地文件，而 Linux 的缓冲区缓存已经将频繁访问的数据保存在内存中。

2.4 单主

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

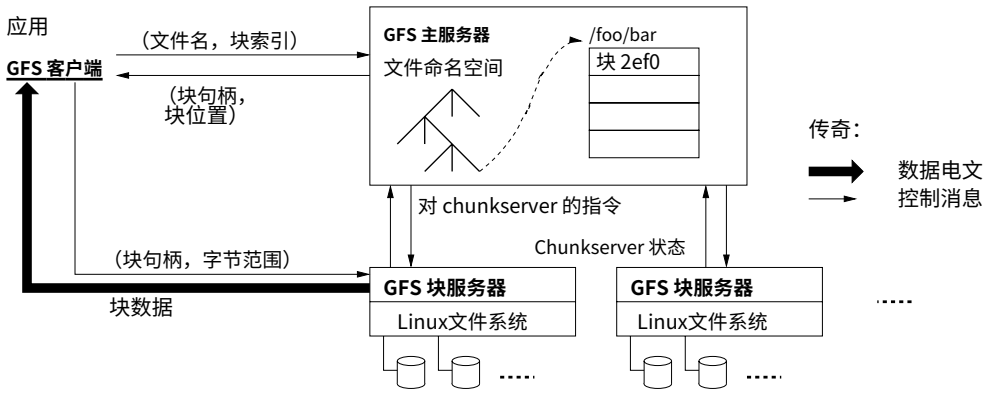


图 1：GFS 架构

并使用全局知识进行复制决策。但是，我们必须尽量减少其参与读写操作，以免成为瓶颈。客户端永远不会通过主服务器读写文件数据。相反，客户端会向主服务器询问应该联系哪些块服务器。它会在有限的时间内缓存这些信息，并在后续的许多操作中直接与块服务器交互。

让我们参考图 1 解释一下简单读取的交互。首先，客户端使用固定的块大小，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，它向主服务器发送包含文件名和块索引的请求。主服务器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这些信息。

然后，客户端会向其中一个副本（很可能是最近的副本）发送请求。该请求指定块句柄及其内的字节范围。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无需客户端与主服务器进行更多交互。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随请求块之后的块的信息。这些额外的信息可以避免未来多次客户端与主服务器的交互，且几乎不会产生任何额外开销。

2.5 块大小

块大小是关键设计参数之一。我们选择了 64 MB，这比典型的文件系统块大小要大得多。每个块副本都以普通 Linux 文件的形式存储在块服务器上，并且仅在需要时进行扩展。惰性空间分配可以避免因内部碎片而造成的空间浪费，这或许是反对如此大块大小的最大原因。

较大的块大小有几个重要的优点。首先，它减少了客户端与主服务器交互的需要，因为对同一块的读写只需要向主服务器发送一次初始请求以获取块位置信息。对于我们的工作负载来说，这种减少尤其重要，因为应用程序大多是顺序读写大文件。即使是小规模的随机读取，客户端也可以轻松地缓存多 TB 工作集的所有块位置信息。其次，由于在大块上，客户端更有可能对给定块执行许多操作，因此可以通过保持持久性来减少网络开销。

长时间保持与 ChunkServer 的 Temp TCP 连接。第三，它减少了存储在 Master 上的元数据的大小。这使我们能够将元数据保存在内存中，从而带来其他优势，我们将在 2.6.1 节中讨论。

另一方面，即使采用惰性空间分配，较大的块大小也有其缺点。小文件由少量块组成，可能只有一个。如果许多客户端访问同一个文件，存储这些块的块服务器可能会成为热点。实际上，热点并不是一个主要问题，因为我们的应用程序大多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，确实出现了热点问题：一个可执行文件以单块文件的形式写入 GFS，然后在数百台机器上同时启动。存储该可执行文件的少数块服务器因数百个并发请求而超负荷。我们通过以更高的复制因子存储此类可执行文件，并让批处理队列系统错开应用程序的启动时间来解决了这个问题。一个潜在的长期解决方案是允许客户端在这种情况下读取其他客户端的数据。

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、文件到块的映射以及每个块副本的位置。所有元数据都保存在 Master 的内存中。前两种类型（命名空间和文件到块的映射）也通过将变更记录到*操作日志*存储在主服务器的本地磁盘上，并在远程机器上进行复制。使用日志可以让我们简单、可靠地更新主服务器状态，并且不会在主服务器崩溃时出现不一致的风险。主服务器不会持久存储块的位置信息。相反，它会在主服务器启动时以及每当有块服务器加入集群时，向每个块服务器询问其块的位置。

2.6.1 内存数据结构

由于元数据存储在内存中，主服务器操作速度很快。此外，主服务器可以轻松高效地在后台定期扫描其整个状态。这种定期扫描用于实现块垃圾收集、在块服务器故障时重新复制以及块迁移以平衡负载和磁盘空间。

跨块服务器的使用情况。第 4.3 节和 4.4 节将进一步讨论这些活动。

这种仅使用内存的方法的一个潜在问题是，块的数量以及整个系统的容量受限于主服务器的内存大小。实际上，这并不是一个严重的限制。主服务器为每个 64 MB 块维护的元数据少于 64 字节。大多数块都是满的，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每个文件需要少于 64 字节，因为它使用前缀压缩来紧凑地存储文件名。

如果需要支持更大的文件系统，那么为主文件系统添加额外的内存只是很小的代价，因为我们通过将元数据存储在内存中而获得了简单性、可靠性、性能和灵活性。

2.6.2 块位置

Master 不会持久记录哪些 ChunkServer 拥有给定 Chunk 的副本。它只是在启动时轮询 ChunkServer 以获取该信息。Master 可以在此之后保持最新状态，因为它控制所有 Chunk 的放置，并定期监控 ChunkServer 的状态。*心跳*消息。

我们最初尝试将块位置信息持久保存在主服务器上，但后来发现，在启动时向块服务器请求数据，并在之后定期请求数据会更简单。这样就解决了在块服务器加入和离开集群、更改名称、故障、重启等情况下保持主服务器和块服务器同步的问题。在一个拥有数百台服务器的集群中，这些事件发生得太频繁了。

理解这一设计决策的另一种方式是，Chunkserver 对其自身磁盘上哪些数据块有最终决定权。尝试在主服务器上维护这些信息的一致性视图毫无意义，因为 Chunkserver 上的错误可能会导致数据块自动消失（例如，磁盘损坏并被禁用），或者操作员可能会重命名 Chunkserver。

2.6.3 操作日志

操作日志包含关键元数据变更的历史记录。它是 GFS 的核心。它不仅是元数据的唯一持久记录，而且还充当定义并发操作顺序的逻辑时间线。文件和块及其版本（参见第 4.5 节）都由其创建的逻辑时间唯一且永久地标识。

由于操作日志至关重要，我们必须可靠地存储它，并且在元数据更改持久化之前，客户端无法看到任何更改。否则，即使块本身仍然存在，我们实际上也会丢失整个文件系统或最近的客户端操作。因此，我们将日志复制到多台远程机器上，并且只有在将相应的日志记录（本地和远程）刷新到磁盘后才响应客户端操作。主服务器在刷新之前会将多个日志记录批量处理，从而减少刷新和复制对整体系统吞吐量的影响。

主服务器通过重放操作日志来恢复其文件系统状态。为了最大限度地缩短启动时间，我们必须保持日志较小。每当日志超过一定大小时，主服务器都会检查其状态，以便能够通过从本地磁盘加载最新的检查点并仅重放

	写	记录追加
串行 成功	*定义*	*定义* 穿插着 *不一致*
并发 成功	*持续的* 但*不明确的*	
失败	*不一致*	

表 1：突变后的文件区域状态

此后的日志记录数量有限。检查点采用类似紧凑 B 树的形式，可以直接映射到内存中并用于命名空间查找，无需额外解析。这进一步加快了恢复速度并提高了可用性。

由于构建检查点可能需要一段时间，因此主服务器的内部状态结构设计为能够在不延迟传入变更的情况下创建新的检查点。主服务器切换到新的日志文件，并在单独的线程中创建新的检查点。新的检查点包含切换之前的所有变更。对于包含数百万个文件的集群，创建时间大约为一分钟。完成后，它将被写入本地和远程磁盘。

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，但我们会保留一些以防万一。检查点操作期间的故障不会影响正确性，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 拥有一个宽松的一致性模型，能够很好地支持我们高度分布式的应用程序，同时又相对简单高效地实现。现在我们将讨论 GFS 的一致性保证及其对应用程序的意义。我们还会重点介绍 GFS 如何维护这些一致性保证，但具体细节留待本文其他部分讨论。

2.7.1 GFS 的担保

文件命名空间的变更（例如，文件创建）是原子性的。它们由主服务器独占处理：命名空间锁定保证了原子性和正确性（第 4.1 节）；主服务器的操作日志定义了这些操作的全局全序（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、突变是否成功或失败以及是否存在并发突变。表 1 总结了结果。文件区域*持续的*如果所有客户端始终看到相同的数据，无论它们从哪个副本读取。区域*定义*如果文件数据发生变异，且变异一致，客户端将看到变异写入的完整内容。如果变异成功且不受并发写入干扰，则受影响的区域将被定义（并因此保持一致）：所有客户端将始终看到变异写入的内容。并发成功的变异会使区域保持未定义但一致的状态：所有客户端都看到相同的数据，但可能无法反映任何一个变异写入的内容。通常，它由来自多个变异的混合片段组成。变异失败会使区域不一致（因此也未定义）：不同的客户端可能在不同时间看到不同的数据。下文我们将介绍我们的应用程序如何区分已定义区域和未定义区域。

区域。应用程序不需要进一步区分不同类型的未定义区域。

数据突变可能*写道*或者*记录附加*写入操作会将数据写入应用程序指定的文件偏移量。记录追加操作会将数据（“记录”）追加到文件末尾。*原子地至少一次*即使存在并发修改，GFS 也会执行，但会以 GFS 选择的偏移量执行（见 3.3 节）。（相比之下，“常规”追加操作仅仅是在客户端认为是文件当前末尾的偏移量处进行写入。）该偏移量会返回给客户端，并标记包含该记录的定义区域的起始位置。此外，GFS 可能会在中间插入填充或记录重复项。它们会占用被认为不一致的区域，并且通常与用户数据量相比显得微不足道。

在一系列成功的变更之后，变更后的文件区域将保证被定义并包含最后一次变更写入的数据。GFS 通过以下方式实现这一点：(a) 在所有副本上以相同的顺序对块应用变更（第 3.1 节）；(b) 使用块版本号来检测任何由于其块服务器宕机而错过变更而变得陈旧的副本（第 4.5 节）。陈旧的副本永远不会参与变更，也不会提供给向主服务器请求块位置的客户端。它们会尽快被垃圾回收。

由于客户端会缓存区块位置，因此它们可能会在信息刷新之前从过时的副本读取数据。此窗口受缓存条目超时和文件下次打开的限制，下次打开文件时会从缓存中清除该文件的所有区块信息。此外，由于我们的大多数文件都是仅追加的，过时的副本通常会返回区块的提前结束信息，而不是过期数据。当读取器重试并联系主服务器时，它将立即获取当前的区块位置。

即使突变成功很久之后，组件故障仍然可能损坏或摧毁数据。GFS 通过主服务器与所有块服务器之间的定期握手来识别故障的块服务器，并通过校验和检测数据损坏（见 5.2 节）。一旦出现问题，数据会尽快从有效副本中恢复（见 4.3 节）。只有当块的所有副本在 GFS 做出反应之前丢失（通常在几分钟内）时，块才会被不可逆地丢失。即使在这种情况下，块也只是变得不可用，而不是损坏：应用程序收到的是明确的错误信息，而不是损坏的数据。

2.7.2 对应用的影响

GFS 应用程序可以通过一些已经用于其他目的的简单技术来适应宽松的一致性模型：依靠附加而不是覆盖、检查点以及编写自我验证、自我识别的记录。

实际上，我们所有的应用程序都通过追加而不是覆盖来修改文件。在一种典型的用法中，写入器会从头到尾生成一个文件。它会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的数据量。检查点也可能包含应用程序级校验和。读取器仅验证和处理到最后一个检查点的文件区域，该检查点已知处于定义状态。无论一致性和并发性问题如何，这种方法都运行良好。与随机写入相比，追加操作效率更高，并且对应用程序故障的恢复能力更强。检查点允许写入器以增量方式重启，并防止读取器处理已成功写入的数据。

从应用程序的角度来看，文件数据仍然不完整。

在另一个典型用法中，许多写入器并发地将数据追加到文件以合并结果或作为生产者-消费者队列。记录追加的“至少追加一次”语义会保留每个写入器的输出。读取器按如下方式处理偶尔出现的填充和重复。写入器准备的每条记录都包含校验和等额外信息，以便验证其有效性。读取器可以使用校验和识别并丢弃多余的填充和记录片段。如果它无法容忍偶尔出现的重复（例如，如果它们会触发非幂等操作），则可以使用记录中的唯一标识符将其过滤掉，这些标识符通常无论如何都需要用于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）包含在我们应用程序共享的库代码中，并且适用于 Google 的其他文件接口实现。这样，相同的记录序列以及罕见的重复项始终会传递给记录读取器。

3. 系统交互

我们设计该系统的目的是尽量减少主服务器在所有操作中的参与。基于此背景，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变更、原子记录追加和快照。

3.1 租赁和变更令

变更是指更改块内容或元数据的操作，例如写入或附加操作。每个变更都会在块的所有副本上执行。我们使用租约来维护跨副本的一致变更顺序。主服务器会将块租约授予其中一个副本，我们称之为*基本的*主节点会为所有针对该块的变更选择一个序列顺序。所有副本在应用变更时都遵循此顺序。因此，全局变更顺序首先由主节点选择的租约授予顺序定义，在租约内部则由主节点分配的序列号定义。

租约机制旨在最大限度地减少主节点的管理开销。租约的初始超时时间为 60 秒。但是，只要块正在发生突变，主节点就可以无限期地向主节点请求并通常接收延期。这些延期请求和授权由*心跳*主服务器和所有块服务器之间定期交换消息。主服务器有时可能会在租约到期前尝试撤销租约（例如，当主服务器想要禁用正在重命名的文件的变更时）。即使主服务器与主服务器失去通信，它也可以安全地在旧租约到期后将新租约授予另一个副本。

在图 2 中，我们通过遵循这些编号步骤的写入控制流程来说明此过程。

1. 客户端向主服务器询问哪个块服务器持有该块的当前租约，以及其他副本的位置。如果没有其他副本持有租约，主服务器会将租约授予它选择的副本（图中未显示）。

2. 主服务器回复主服务器的身份和其他服务器的位置（*次要的*）副本。客户端会缓存这些数据，以备将来更改。只有当主服务器

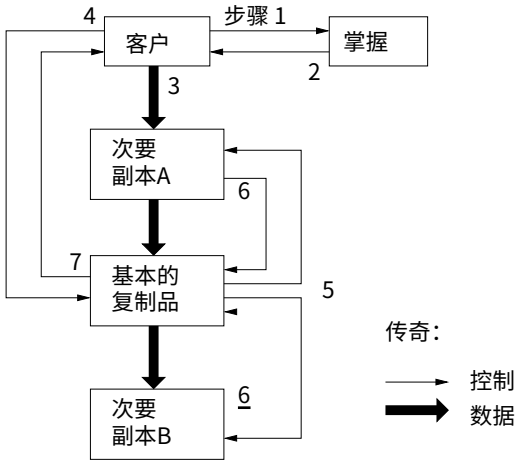


图 2：写入控制和数据流

变得无法联系或回复不再持有租约。

3. 客户端将数据推送到所有副本。客户端可以按任意顺序执行此操作。每个块服务器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解耦，无论哪个块服务器是主服务器，我们都可以根据网络拓扑结构调度昂贵的数据流，从而提高性能。3.2 节将对此进行进一步讨论。

4. 所有副本确认收到数据后，客户端会向主服务器发送写入请求。该请求标识先前推送给所有副本的数据。主服务器会为其收到的所有变更（可能来自多个客户端）分配连续的序列号，以提供必要的序列化。然后，主服务器会按序列号顺序将变更应用到其本地状态。

5. 主副本将写入请求转发给所有辅助副本。每个辅助副本都按照主副本指定的序列号顺序应用变更。

6. 所有从节点都回复主节点，表明它们已经完成操作。

7. 主服务器回复客户端。任何副本服务器遇到的错误都会报告给客户端。
如果发生错误，写入操作可能在主副本和任意子集的辅助副本上都成功了。（如果在主副本上失败，则不会分配序列号并进行转发。）客户端请求将被视为失败，修改后的区域将处于不一致状态。我们的客户端代码会通过重试失败的变更来处理此类错误。它会在步骤 (3) 到 (7) 进行几次尝试，然后回退到从写入操作开始重试。

如果应用程序的写入操作较大或跨越块边界，GFS 客户端代码会将其拆分为多个写入操作。这些操作均遵循上述控制流，但可能会与其他客户端的并发操作交错执行或被覆盖。因此，共享文件区域最终可能包含来自不同客户端的片段，尽管副本完全相同，因为各个操作在所有副本上都以相同的顺序成功完成。这使得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，以高效利用网络。控制流从客户端流向主服务器，再流向所有从服务器，而数据则以流水线的方式沿着精心挑选的块服务器链线性推送。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，数据会沿着块服务器链线性推送，而不是按照其他拓扑结构（例如树形）进行分发。这样，每台机器的全部出站带宽都会被用来尽可能快地传输数据，而不是分散在多个接收方之间。

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有），每台机器都会将数据转发到网络拓扑中尚未接收数据的“最近”的机器。假设客户端正在将数据推送到块服务器 S1 到 S4。它将数据发送到最近的块服务器，例如 S1。S1 将其转发到距离 S1 最近的块服务器 S2 到 S4，例如 S2。同样，S2 将其转发到 S3 或 S4，以距离 S2 较近的为准，依此类推。我们的网络拓扑足够简单，可以根据 IP 地址准确估算“距离”。

最后，我们通过 TCP 连接对数据传输进行流水线化，从而最大限度地降低延迟。一旦块服务器收到数据，它就会立即开始转发。由于我们使用全双工链路的交换网络，因此流水线化对我们尤其有用。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，理想的传输时间是 B字节数 R副本是 $交接+RL$在哪里 T是网络吞吐量，$左$是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),和$左$远低于1毫秒。因此，理想情况下，1MB可以在大约80毫秒内分发。

3.3 原子记录附加

GFS 提供了一个原子追加操作，称为 *记录追加*在传统的写入操作中，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。然而，在记录追加操作中，客户端仅指定数据。GFS 会以原子方式（即，作为一个连续的字节序列）将其追加到文件至少一次，并以 GFS 选择的偏移量将其追加到文件，并将该偏移量返回给客户端。这类似于写入在附加当多个写入者同时执行操作时，Unix 中的模式不会出现竞争条件。

-

我们的分布式应用程序大量使用记录追加功能，其中许多客户端位于不同的机器上，并发地向同一个文件追加记录。如果客户端使用传统的写入方式，则需要额外复杂且昂贵的同步，例如通过分布式锁管理器。在我们的工作负载中，这类文件通常

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。

记录追加是一种变异，它遵循第 3.1 节中的控制流，仅在主服务器上增加了一点逻辑。客户端将数据推送到文件最后一个块的所有副本，然后，它将请求发送到主服务器。主服务器检查将记录追加到当前块是否会导致块超过最大大小（64 MB）。如果是这样，它会将块填充到最大大小，告诉从服务器也这样做，并回复客户端，指示应在下一个块上重试该操作。（记录追加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。）如果记录在最大大小范围内（这是常见情况），则主服务器将数据追加到其副本，告诉从服务器将数据写入其所在的精确偏移量，最后回复客户端成功。

如果任何副本上的记录追加失败，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，其中可能包含同一条记录的全部或部分重复。GFS 不保证所有副本的字节顺序完全相同。它仅保证数据至少以原子单位写入一次。此属性很容易通过以下简单观察得出：要使操作报告成功，数据必须已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，所有副本的长度至少与记录末尾的长度相同，因此即使之后其他副本成为主副本，任何后续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已定义的（因此是一致的），而中间区域是不一致的（因此是未定义的）。我们的应用程序可以处理不一致的区域，正如我们在 2.7.2 节中讨论的那样。

3.4 快照

快照操作几乎可以即时复制文件或目录树（“源”），同时最大程度地减少正在进行的变更造成的中断。我们的用户使用它来快速创建海量数据集的分支副本（通常还会递归地复制这些副本），或者在尝试更改之前检查当前状态，以便稍后轻松提交或回滚。

与 AFS [5] 类似，我们使用标准的写时复制技术来实现快照。当主服务器收到快照请求时，它会首先撤销即将快照的文件中所有未完成的块租约。这确保了对这些块的任何后续写入都需要与主服务器交互以找到租约持有者。这将使主服务器有机会首先创建该块的新副本。

租约撤销或到期后，主服务器会将该操作记录到磁盘。然后，它会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。

快照操作后，客户端首次想要写入块 C 时，它会向主服务器发送请求，以查找当前的租约持有者。主服务器注意到块 C 的引用计数大于 1。它推迟回复客户端请求，而是选择一个新的块处理 C'。然后，它会请求每个拥有 C 当前副本的块服务器创建一个名为 C' 的新块。通过在与原始块服务器相同的块服务器上创建新块，我们确保数据可以在本地复制，而无需通过网络复制（我们的磁盘速度大约是 100 MB 以太网链路的三倍）。从这一点来看，请求处理与任何块的处理没有什么不同：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该块，而不知道该块刚刚从现有块创建。

4. 主操作

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，创建新的块并由此创建副本，并协调各种系统范围的活动以保持块的完整复制，平衡所有块服务器的负载，并回收未使用的存储空间。我们现在将逐一讨论这些主题。

4.1 命名空间管理和锁定

许多主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的块服务器租约。我们不希望其他主操作在运行时被延迟。因此，我们允许多个操作同时处于活动状态，并在命名空间区域上使用锁来确保正确的序列化。

与许多传统文件系统不同，GFS 没有一个按目录列出该目录中所有文件的数据结构。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 在逻辑上将其命名空间表示为一个将完整路径名映射到元数据的查找表。使用前缀压缩，该表可以在内存中高效地表示。命名空间树中的每个节点（无论是绝对文件名还是绝对目录名）都关联有一个读写锁。

每个主操作在运行前都会获取一组锁。通常，如果涉及 /d1/d2/.../dn/叶，它将获取目录名称/的读锁d1、/d1/d2、...、/d1/d2/.../dn、以及完整路径名上的读锁或写锁 /d1/d2/.../dn/叶。注意叶子根据操作，可能是文件或目录。

我们现在说明这种锁定机制如何防止文件/主页/用户/foo从创建时 /主页/用户 正在快照到 /保存/用户。快照操作获取 / 上的读锁家和 /节省，并在 / 上写锁主页/用户和 /保存/用户。文件创建获取 / 上的读锁家和 /家庭/用户，以及 / 上的写锁主页/用户/foo。这两个操作将被正确序列化，因为它们试图获取 / 上的冲突锁家庭/用户。文件创建不需要对父目录进行写锁定，因为没有“目录”，或者 *索引节点*类似，需要保护数据结构不被修改。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优点是它允许在同一目录中并发修改。例如，可以在同一目录中并发执行多个文件创建：每个文件都会获取目录名的读锁和文件名的写锁。目录名的读锁足以防止目录被删除、重命名或创建快照。

文件名序列化尝试两次创建具有相同名称的文件。

由于命名空间可以包含多个节点，因此读写锁对象会以惰性方式分配，并在不再使用时被删除。此外，为了避免死锁，锁的获取遵循一致的全序：它们首先按命名空间树中的层级排序，然后在同一层级内按字典顺序排序。

4.2 副本放置

GFS 集群高度分布于多个层级。它通常拥有数百个块服务器，分布在许多机架上。这些块服务器又可以被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。多级分布对数据分发的可扩展性、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。对于这两者而言，仅仅将副本分布在不同的机器上是不够的，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。我们还必须将块副本分布在不同的机架上。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故障），块的某些副本仍能存活并保持可用。这也意味着块的流量（尤其是读取流量）可以利用多个机架的总带宽。另一方面，写入流量必须流经多个机架，这是我们愿意做出的权衡。

4.3 创建、重新复制、重新平衡

创建块副本有三个原因：块创建、重新复制和重新平衡。

当主人*创建*一个块，它会选择将最初为空的副本放置在何处。它会考虑几个因素。（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，这将均衡各个块服务器的磁盘利用率。（2）我们希望限制每个块服务器上“最近”创建的数量。虽然创建本身很便宜，但它可以可靠地预测即将发生的大量写入流量，因为块是在写入需要时创建的，并且在我们的一次追加多次读取工作负载中，它们一旦完全写入，通常就变为只读。（3）如上所述，我们希望将块的副本分布在各个机架上。

大师*重新复制*一旦可用副本数低于用户指定的目标，就会立即重新复制一个块。发生这种情况的原因有很多：chunkserver 变得不可用、它报告其副本可能已损坏、它的某个磁盘由于错误而被禁用或者复制目标增加了。每个需要重新复制的块都根据几个因素确定优先级。其一是它与复制目标的距离。例如，我们给予丢失两个副本的块比只丢失一个副本的块更高的优先级。此外，我们倾向于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，为了最大限度地减少故障对正在运行的应用程序的影响，我们会提高任何阻碍客户端进度的块的优先级。

主服务器选择优先级最高的块，并通过指示某个块服务器直接从现有的有效副本复制块数据来“克隆”它。新副本的放置目标与创建副本的目标类似：均衡磁盘空间利用率，限制单个块服务器上的活跃克隆操作，并将副本分布到各个机架。为了防止克隆流量过大，主服务器会限制集群和每个块服务器的活跃克隆操作数量。此外，每个块服务器还会通过限制其对源块服务器的读取请求来限制其在每次克隆操作上消耗的带宽。

最后，主人*重新平衡*定期更新副本：它会检查当前副本分布情况，并移动副本以获得更好的磁盘空间和负载平衡。同样，通过此过程，主服务器会逐渐填充新的块服务器，而不是立即用新块及其带来的大量写入流量将其淹没。新副本的放置标准与上面讨论的类似。此外，主服务器还必须选择要删除哪些现有副本。通常，它会优先删除那些位于可用空间低于平均水平的块服务器上的副本，以均衡磁盘空间使用率。

4.4 垃圾收集

文件删除后，GFS 不会立即回收可用的物理存储空间。它只会在文件和块级别的常规垃圾回收过程中以惰性方式回收。我们发现这种方法使系统更加简单可靠。

4.4.1 机制

当应用程序删除文件时，主服务器会像其他更改一样立即记录删除操作。但是，文件不会立即回收资源，而是重命名为包含删除时间戳的隐藏名称。在主服务器定期扫描文件系统命名空间期间，如果这些隐藏文件存在超过三天（间隔可配置），它会将其删除。在此之前，该文件仍然可以使用新的特殊名称读取，并且可以通过将其重命名为正常名称来恢复删除。当隐藏文件从命名空间中删除时，其内存元数据将被擦除。这实际上切断了它与所有块的链接。

在对块命名空间进行类似的定期扫描时，主服务器会识别孤立块（即无法从任何文件访问的块）并擦除这些块的元数据。*心跳*通过定期与主服务器交换消息，每个块服务器报告其拥有的块的子集，主服务器则回复所有不再存在于主服务器元数据中的块的标识。块服务器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在编程语言的背景下是一个难题，需要复杂的解决方案，但在我们的例子中却非常简单。我们可以轻松识别所有对块的引用：它们位于主服务器独占维护的文件到块的映射中。我们还可以轻松识别所有块副本：它们是每个块服务器上指定目录下的 Linux 文件。任何主服务器未知的此类副本都是“垃圾”。

与急切删除相比，垃圾收集方法的存储回收具有诸多优势。首先，在组件故障频发的大规模分布式系统中，它简单可靠。块创建可能在某些块服务器上成功，但在其他块服务器上失败，从而留下主服务器不知道其存在的副本。副本删除消息可能会丢失，主服务器必须记住在自身和块服务器发生故障时重新发送这些消息。垃圾收集提供了一种统一可靠的方法来清理任何已知无用的副本。其次，它将存储回收合并到主服务器的常规后台活动中，例如定期扫描命名空间以及与块服务器握手。因此，回收是批量进行的，成本可以摊销。此外，它仅在主服务器相对空闲时才进行。主服务器可以更迅速地响应需要及时关注的客户端请求。第三，回收存储的延迟为防止意外的、不可逆的删除提供了安全保障。

根据我们的经验，主要的缺点是，当存储空间紧张时，延迟有时会妨碍用户调整使用情况。反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空间。我们通过加快已删除文件再次被明确删除时的存储回收速度来解决这些问题。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文件的所有数据块均以非复制方式存储，并且任何已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

如果某个 ChunkServer 发生故障，导致 Chunk 副本丢失，则副本可能会变得陈旧。对于每个 Chunk，Master 维护一个*块版本号*区分最新副本和陈旧副本。

每当主服务器授予某个块的新租约时，它都会增加块的版本号并通知最新的副本。主服务器和这些副本都会在其持久状态中记录新的版本号。这发生在任何客户端收到通知之前，因此也发生在客户端开始写入块之前。如果另一个副本当前不可用，则其块版本号不会增加。当该块服务器重启并报告其块集及其关联的版本号时，主服务器会检测到该块服务器拥有过时的副本。如果主服务器发现一个大于其记录中的版本的版本号，主服务器会假定它在授予租约时失败了，因此将更高的版本号作为最新版本。

Master 会在常规垃圾回收中移除过期副本。在此之前，它会在回复客户端对块信息的请求时，实际上认为过期副本根本不存在。作为另一项安全措施，Master 在通知客户端哪个块服务器持有某个块的租约，或在克隆操作中指示某个块服务器从另一个块服务器读取该块时，会包含块的版本号。客户端或块服务器在执行操作时会验证版本号，以确保始终访问最新数据。

5. 容错与诊断

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。质量和如此多的组件加在一起，使得这些问题成为常态而非例外：我们无法完全信任机器，也无法完全信任磁盘。组件故障可能导致系统不可用，甚至更糟的是数据损坏。我们将讨论如何应对这些挑战，以及我们内置于系统中的工具，以便在问题不可避免地发生时进行诊断。

5.1 高可用性

GFS 集群中的数百台服务器，总有一些服务器在特定时刻不可用。我们通过两种简单有效的策略来保证整个系统的高可用性：快速恢复和复制。

5.1.1 快速恢复

无论主服务器和块服务器如何终止，它们都被设计为能够在几秒钟内恢复状态并启动。实际上，我们并不区分正常终止和异常终止；服务器通常会通过终止进程来关闭。客户端和其他服务器在处理未完成的请求时会遇到轻微的卡顿，它们会重新连接到重新启动的服务器并重试。第 6.2.2 节报告了观察到的启动时间。

5.1.2 块复制

如前所述，每个块都会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。默认值为三级。主服务器会根据需要克隆现有副本，以便在块服务器离线或通过校验和验证检测到损坏的副本时（参见 5.2 节）保持每个块的完整复制。尽管复制方案已经为我们提供了良好的性能，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，以满足我们日益增长的只读存储需求。我们预计，在我们非常松散耦合的系统中实现这些更复杂的冗余方案将具有挑战性，但也是可以实现的，因为我们的流量主要由追加和读取操作组成，而不是小规模的随机写入操作。

5.1.3 主复制

主服务器状态会被复制以确保可靠性。其操作日志和检查点会在多台机器上复制。只有当日志记录被刷新到本地磁盘以及所有主服务器副本后，状态变更才被视为已提交。为简单起见，一个主服务器进程仍然负责所有变更以及后台活动，例如垃圾收集等会在系统内部发生变化的活动。当它发生故障时，它几乎可以立即重启。如果其所在机器或磁盘发生故障，GFS 外部的监控基础设施会在其他位置使用复制的操作日志启动一个新的主服务器进程。客户端仅使用主服务器的规范名称（例如 gfs-test），这是一个 DNS 别名，如果主服务器迁移到另一台机器，该别名可以更改。

此外，“影子”主服务器即使在主服务器宕机时也能提供对文件系统的只读访问。它们是影子，而不是镜像，因为它们可能会稍微滞后于主服务器，通常是几分之一秒。它们增强了那些未被主动修改的文件或不介意获得稍微过时结果的应用程序的读取可用性。事实上，由于文件内容是从块服务器读取的，因此应用程序不会观察到过时的文件内容。

在短时间内陈旧的是文件元数据，如目录内容或访问控制信息。

为了保持自身信息畅通，影子主服务器会读取不断增长的操作日志副本，并像主服务器一样，对其数据结构应用相同的更改顺序。与主服务器类似，它在启动时（之后偶尔轮询）会轮询块服务器以定位块副本，并与它们频繁交换握手消息以监视它们的状态。影子主服务器仅在主服务器创建和删除副本的决策导致的副本位置更新时才依赖主服务器。

5.2 数据完整性

每个块服务器都使用校验和来检测存储数据的损坏情况。鉴于GFS集群通常在数百台机器上拥有数千个磁盘，它经常会遇到磁盘故障，导致读写路径上的数据损坏或丢失。（其中一个原因参见第7节。）我们可以使用其他块副本来恢复损坏，但通过比较不同块服务器的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修改的语义，特别是前面讨论的原子记录追加，并不保证副本完全相同。因此，每个块服务器必须通过维护校验和来独立验证其自身副本的完整性。

一个块被拆分成 64 KB 大小的块。每个块都有一个对应的 32 位校验和。与其他元数据一样，校验和保存在内存中，并与日志记录一起持久存储，与用户数据分开存储。

对于读取操作，chunkserver 会在将任何数据返回给请求者（无论是客户端还是其他 chunkserver）之前，验证与读取范围重叠的数据块的校验和。因此，chunkserver 不会将损坏的数据传播到其他机器。如果某个块与记录的校验和不匹配，chunkserver 会向请求者返回错误，并将不匹配情况报告给 master。作为响应，请求者将从其他副本读取数据，而 master 则会从另一个副本克隆该块。在新的有效副本到位后，master 会指示报告不匹配情况的 chunkserver 删除其副本。

校验和对读取性能的影响很小，原因有几个。由于我们的大多数读取操作至少跨越几个块，因此我们只需要读取并校验相对少量的额外数据进行验证。GFS 客户端代码通过尝试将读取操作与校验和块边界对齐，进一步降低了这种开销。此外，chunkserver 上的校验和查找和比较无需任何 I/O，而校验和计算通常可以与 I/O 重叠。

校验和计算针对追加到块末尾的写入操作（而不是覆盖现有数据的写入操作）进行了深度优化，因为这类写入操作在我们的工作负载中占主导地位。我们只需增量更新最后一个部分校验和块的校验和，并为任何由追加操作填充的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，而我们现在无法检测到，新的校验和值也不会与存储的数据匹配，下次读取该块时，损坏情况仍会照常检测到。

相反，如果写入操作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块和最后一个块，然后执行写入操作，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没有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情况。一旦检测到损坏，主服务器可以创建一个新的未损坏的副本并删除损坏的副本。这可以防止非活动但损坏的块副本欺骗主服务器，使其误以为它拥有足够多的有效块副本。

5.3 诊断工具

广泛而详细的诊断日志在问题隔离、调试和性能分析方面提供了巨大的帮助，同时成本却极低。如果没有日志，就很难理解机器之间瞬时且不可重复的交互。GFS 服务器会生成诊断日志，记录许多重要事件（例如，chunkserver 的启动和关闭）以及所有 RPC 请求和回复。这些诊断日志可以随意删除，而不会影响系统的正确性。但是，我们会在空间允许的范围内尽量保留这些日志。

RPC 日志包含网络上发送的确切请求和响应，但不包括正在读取或写入的文件数据。通过将请求与响应进行匹配，并整理不同机器上的 RPC 记录，我们可以重建完整的交互历史记录来诊断问题。这些日志还可以作为负载测试和性能分析的跟踪记录。

日志记录对性能的影响极小（而 且远超其带来的好处），因为这些日志是顺序异步写入的。最新事件也会保存在内存中，可供持续在线监控。

6. 测量

在本节中，我们将提供一些微基准测试来说明 GFS 架构和实现中固有的瓶颈，以及来自 Google 正在使用的实际集群的一些数字。

6.1 微基准测试

我们在一个由一个主服务器、两个主副本、16 个块服务器和 16 个客户端组成的 GFS 集群上测试了性能。请注意，此配置是为了方便测试而设置的。典型的集群通常包含数百个块服务器和数百个客户端。

所有机器均配置双路 1.4 GHz PIII 处理器、2 GB 内存、两块 80 GB 5400 rpm 磁盘，并通过 100 Mbps 全双工以太网连接到一台 HP 2524 交换机。所有 19 台 GFS 服务器连接到一台交换机，所有 16 台客户端连接到另一台交换机。两台交换机之间通过 1 Gbps 链路连接。

6.1.1 读取

*否*客户端同时从文件系统读取数据。每个客户端从 320 GB 的文件集中随机读取 4 MB 区域。此过程重复 256 次，最终每个客户端读取 1 GB 数据。所有块服务器加起来只有 32 GB 内存，因此我们预计 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的结果。

图 3(a) 显示了*否*客户端及其理论极限。当两个交换机之间的 1 Gbps 链路饱和时，该限制峰值为 125 MB/s，当其 100 Mbps 网络接口饱和时，每个客户端的峰值为 12.5 MB/s，以适用者为准。当只有一个客户端读取时，观察到的读取速率为 10 MB/s，或每个客户端限制的 80%。对于 16 个读取器，总读取速率达到 94 MB/s，约为 125 MB/s 链路限制的 75%，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块服务器读取的概率也会增加。

6.1.2 写入

*否*客户端同时写入*否*不同的文件。每个客户端通过一系列 1 MB 的写入操作将 1 GB 数据写入新文件。总写入速率及其理论极限如图 3(b) 所示。极限稳定在 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，每个块服务器的输入连接速率为 12.5 MB/s。

单个客户端的写入速率为 6.3 MB/s，大约是上限的一半。造成这种情况的主要原因是我们的网络堆栈。它与我们用于将数据推送到块副本的流水线方案交互不佳。数据从一个副本传播到另一个副本的延迟降低了整体写入速率。

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，多个客户端并发写入同一块服务器的可能性会更大。此外，由于每次写入都涉及三个不同的副本，因此 16 个写入器比 16 个读取器更容易发生冲突。

写入速度比我们预期的要慢。实际上，这并不是一个大问题，因为尽管它增加了单个客户端的延迟，但并不显著影响系统向大量客户端提供的总体写入带宽。

6.1.3 记录附加

图 3(c) 显示了记录附加性能。*否*客户端同时向单个文件追加数据。性能受限于存储文件最后一个块的块服务器的网络带宽，与客户端数量无关。单个客户端的初始速度为 6.0 MB/s，16 个客户端的初始速度降至 4.8 MB/s，这主要是由于拥塞以及不同客户端的网络传输速率差异造成的。

我们的应用程序往往会同时生成多个这样的文件。换句话说，*否*客户端附加到*米*同时共享文件*否*和*米*有几十甚至几百个。因此，我们实验中的chunkserver网络拥塞在实践中不是一个严重的问题，因为当一个文件的chunkserver繁忙时，客户端可以继续写入一个文件。

6.2 真实世界集群

我们现在研究 Google 内部正在使用的两个集群，它们代表了其他几个类似的集群。集群 A 定期由一百多名工程师用于研发。一个典型的任务由人类用户发起，运行时间长达数小时。它读取几 MB 到几 TB 的数据，转换或分析数据，然后将结果写回集群。集群 B 主要用于生产数据处理。这些任务持续很长时间。

簇	一个	B
块服务器	342	227
可用磁盘空间	72 TB	180 TB
已用磁盘空间	55 TB	155 TB
文件数量	735 千	737 千
死文件数量 块数量	22 千	232 千
	992 千	1550 千
块服务器上的元数据 主服务器上的元数据	13 GB	21 GB
	48 MB	60 MB

表2：两个GFS集群的特征

更长时间地持续生成和处理多 TB 数据集，只需偶尔进行人工干预。在这两种情况下，单个“任务”都包含多台机器上的多个进程，同时读写多个文件。

6.2.1 存储

如表中前五项所示，两个集群都拥有数百个块服务器，支持数 TB 的磁盘空间，并且空间相当充裕，但并未完全耗尽。“已用空间”包含所有块副本。几乎所有文件都复制了三次。因此，这两个集群分别存储了 18 TB 和 52 TB 的文件数据。

两个集群的文件数量相似，但 B 集群的死文件（即已被删除或被新版本替换但尚未回收存储空间的文件）比例更高。由于 B 集群的文件通常较大，因此其数据块也更多。

6.2.2 元数据

块服务器总共存储了数十GB的元数据，其中大部分是64KB用户数据块的校验和。块服务器中唯一保存的其他元数据是4.5节中讨论的块版本号。

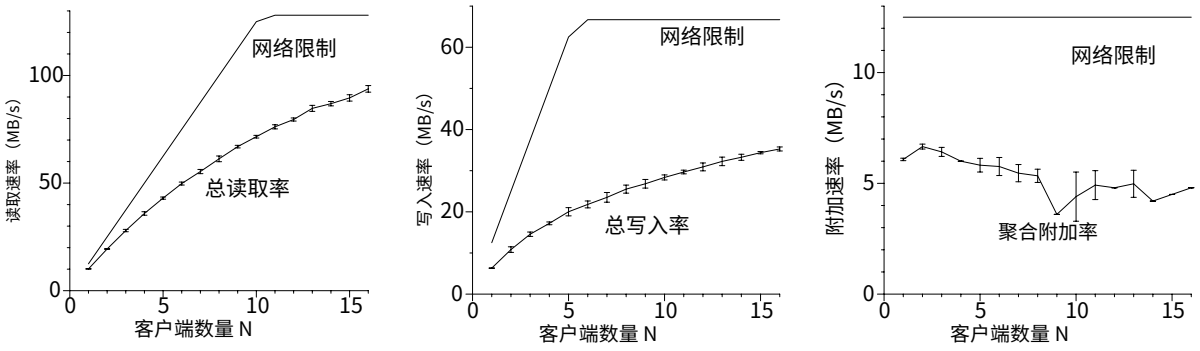
主服务器上保存的元数据要小得多，平均每个文件只有几十 MB，大约100字节。这符合我们的假设，即主服务器内存的大小实际上不会限制系统的容量。大多数文件元数据是以前缀压缩形式存储的文件名。其他元数据包括文件所有权和权限、文件到块的映射以及每个块的当前版本。此外，我们还为每个块存储了当前副本位置和引用计数，以实现写时复制。

每个单独的服务器（包括块服务器和主服务器）都只有 50 到 100 MB 的元数据。因此恢复速度很快：只需几秒钟即可从磁盘读取这些元数据，然后服务器就可以响应查询。然而，主服务器在一段时间内（通常为 30 到 60 秒）会有些卡顿，直到它从所有块服务器获取到块的位置信息。

6.2.3 读写速率

表 3 显示了不同时间段的读写速率。进行这些测量时，两个集群均已运行约一周。（集群最近因升级到新版本的 GFS 而重启。）

自重启以来，平均写入速率不到 30 MB/s。当我们进行这些测量时，B 正处于写入活动的爆发阶段，产生了大约 100 MB/s 的数据，由于写入操作传播到三个副本，因此产生了 300 MB/s 的网络负载。



（a）读取　　（b）写入　　（c）记录附加

图 3：总吞吐量。顶部曲线显示了我们网络拓扑所施加的理论极限。底部曲线显示了测量的吞吐量。它们带有误差线，表示 95% 的置信区间，由于测量方差较小，在某些情况下这些误差线难以辨认。

簇	一个	B
读取率（最后一分钟） 读取率（最后一小时） 读取率（自重启以来） 写入率（最后一分钟） 写入率（最后一小时） 写入率（自重启以来） 主操作（最后一分钟）	583 MB/秒 562 MB/秒 589 MB/秒 1 MB/秒 2 MB/秒 25 MB/秒	380MB/秒 384 MB/秒 49 MB/秒 101MB/秒 117 MB/秒 13 MB/秒
主操作（最后一小时） 主操作（自重启以来）	325 次操作/秒 381 次操作/秒 202 次操作/秒	533 次操作/秒 518 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，总工作负载中的读取次数多于写入次数。两个集群都处于高读取活动状态。具体来说，集群 A 在前一周一直维持着 580 MB/s 的读取速率。其网络配置可以支持 750 MB/s，因此资源利用效率很高。集群 B 可以支持 1300 MB/s 的峰值读取速率，但其应用程序的读取速率仅为 380 MB/s。

6.2.4 主负载

表 3 还显示，发送给主服务器的操作速率约为每秒 200 到 500 次。主服务器可以轻松跟上这个速率，因此不会成为这些工作负载的瓶颈。

在 GFS 的早期版本中，主服务器偶尔会成为某些工作负载的瓶颈。它大部分时间都花在顺序扫描大型目录（其中包含数十万个文件）以查找特定文件上。此后，我们更改了主服务器的数据结构，使其能够高效地在命名空间中进行二进制搜索。现在，它可以轻松支持每秒数千次文件访问。如有必要，我们可以通过在命名空间数据结构之前放置名称查找缓存来进一步加快速度。

6.2.5 恢复时间

当一个块服务器发生故障时，一些块的复制量会降低，必须进行克隆才能恢复其复制级别。恢复所有此类块所需的时间取决于资源量。在一个实验中，我们关闭了集群 B 中的一台块服务器。该块服务器大约有 15,000 个数据块，共包含 600 GB 数据。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（占数据块服务器数量的 40%），每个克隆操作的带宽限制为 6.25 MB/s（50 Mbps）。所有数据块在 23.2 分钟内恢复，有效复制速率为 440 MB/s。

在另一项实验中，我们关闭了两台 ChunkServer，每台服务器大约有 16,000 个 Chunk，数据量为 660 GB。这次双重故障导致 266 个 Chunk 只剩下一个副本。这 266 个 Chunk 以更高的优先级被克隆，并在 2 分钟内全部恢复到至少 2 个副本，从而使集群处于可以容忍另一台 ChunkServer 故障且不会丢失数据的状态。

6.3 工作量细分

在本节中，我们将详细分析两个 GFS 集群的工作负载，这两个集群与 6.2 节中的集群类似但不完全相同。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

这些结果仅包含客户端发起的请求，以便反映我们的应用程序为整个文件系统产生的工作负载。它们不包含执行客户端请求的服务器间请求或内部后台活动（例如转发写入或重新平衡）。

I/O 操作的统计信息基于 GFS 服务器记录的实际 RPC 请求启发式重构的信息。例如，GFS 客户端代码可能会将一次读取拆分为多个 RPC 以提高并行度，从而推断出原始读取。由于我们的访问模式高度程式化，我们预计任何错误都可能包含在噪声中。应用程序的显式日志记录可能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启数千个正在运行的客户端，而且从如此多的机器上收集结果也非常繁琐。

需要注意的是，不要过度泛化我们的工作负载。由于 Google 完全控制着 GFS 及其应用程序，因此这些应用程序往往会针对 GFS 进行调整，反之，GFS 也是为这些应用程序设计的。这种相互影响也可能存在于一般应用程序之间。

手术	读		写		记录追加	
簇	十	是	十	是	十	是
0千	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K到8K	65.2	38.5	0.4	1.0	18.9	15.2
8K到64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	<。1	4.3
128K到256K	0.2	0.3	31.6	0.4	<。1	10.6
256K到512K	0.1	0.1	4.2	7.7	<。1	31.2
512K..1M	3.9	6.9	35.5	28.7	0.1	2.2	25.5
1M..inf	1.8	1.5	12.3			0.7	2.2

表 4：按规模划分的业务细分（%）。 读取，大小是实际读取和了传输的数据量，而不是请求的数据量。

和文件系统，但在我们的案例中效果可能更为明显。

6.3.2 Chunkserver 工作负载

表 4 显示了操作按大小的分布情况。读取大小呈现双峰分布。小读取（小于 64 KB）来自寻道密集型客户端，这些客户端在大型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顺序读取。

在集群 Y 中，大量读取操作完全没有返回任何数据。我们的应用程序，尤其是生产系统中的应用程序，经常使用文件作为生产者-消费者队列。生产者并发地向文件追加数据，而消费者则读取文件末尾的数据。偶尔，当消费者的速度超过生产者时，不会返回任何数据。集群 X 出现这种情况的频率较低，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。

写入大小也呈现出双峰分布。较大的写入（超过 256 KB）通常是由于写入器内部存在大量缓冲造成的。较小的写入（小于 64 KB）则是由缓冲数据较少、检查点或同步频率较高，或者仅仅是生成数据较少的写入器造成的。

至于记录附加，集群 Y 看到的大型记录附加百分比比集群 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 显示了各种大小操作中传输的数据总量。对于所有类型的操作，较大的操作（超过 256 KB）通常占传输字节数的大部分。由于随机寻道工作负载，较小的读取操作（小于 64 KB）确实会传输一小部分但相当可观的读取数据。

6.3.3 附加与写入

记录追加在我们的生产系统中被广泛使用。对于集群 X，写入与记录追加的比例（按传输字节数计算）为 108:1，按操作数计算为 8:1。对于生产系统使用的集群 Y，该比例分别为 3.7:1 和 2.5:1。此外，这些比例表明，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应用程序特定缓冲区大小选择的影响。

正如预期的那样，我们的数据变更工作负载主要由追加操作而非覆盖操作组成。我们测量了主副本上被覆盖的数据量。此应用程序

手术	读		写		记录追加	
簇	十	是	十	是	十	是
1B..1K	<。1	<.1	<。1	<.1	<。1	<。1
1K到8K	13.8	3.9	<。1	<.1	<。1	0.1
8K到64K	11.4	9.3	2.4	5.9	2.3	0.3	0.3
64K..128K	0.7		0.3	0.3	22.7	1.2
128K到256K	0.8	0.6	16.5	0.2	<。1	5.8
256K到512K	1.4	0.3	3.4	7.7	<。1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

表 5：按操作大小划分的传输字节数（%）。对于读取操作，大小指的是实际读取和传输的数据量，而不是请求的数据量。如果读取操作尝试读取文件末尾以外的内容，两者可能会有所不同，这在我们的工作负载中并不罕见。

簇	十	是
打开	26.1	16.3
删除	0.7	1.5
查找位置	64.3	65.8
查找租赁持有人	7.8	13.4
查找匹配文件	0.6	2.2
所有其他合并	0.5	0.8

表 6：主请求按类型细分（%）

这近似于客户端故意覆盖先前写入的数据而不是附加新数据的情况。对于集群 X，覆盖占变异字节数的不到 0.0001%，占变异操作的不到 0.0003%。对于集群 Y，这两个比例均为 0.05%。虽然这个比例很低，但仍然高于我们的预期。事实证明，这些覆盖大部分来自客户端因错误或超时而重试。它们不属于工作负载的一部分。*本身*而是重试机制的结果。

6.3.4 主工作负载

表 6 显示了向 Master 发出的请求类型明细。大多数请求询问的是块位置（*查找位置*) 读取和租赁持有人信息 (*FindLease-Locker*) 用于数据突变。

X 组和 Y 组的数量明显不同 *删除*请求，因为集群 Y 存储了定期重新生成并被新版本替换的生产数据集。这种差异的一部分进一步隐藏在*打开*请求，因为文件的旧版本可能会通过从头开始写入（Unix 开放术语中的模式“w”）而被隐式删除。

*查找匹配文件*是一个支持“ls”及类似文件系统操作的模式匹配请求。与其他针对主服务器的请求不同，它可能会处理很大一部分命名空间，因此开销可能很大。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某些部分以了解全局应用程序状态。相比之下，集群 X 的应用程序受到更明确的用户控制，并且通常预先知道所有所需文件的名称。

7. 体验

在构建和部署 GFS 的过程中，我们遇到了各种各样的问题，有些是操作问题，有些是技术问题。

最初，GFS 被设想为生产系统的后端文件系统。随着时间的推移，其用途逐渐扩展至研发任务。它最初几乎不支持权限和配额等功能，但现在已包含了一些基本功能。虽然生产系统规范且可控，但用户有时却并非如此。需要更多基础设施来防止用户之间互相干扰。

我们遇到的一些最大问题与磁盘和 Linux 相关。许多磁盘向 Linux 驱动程序声称它们支持一系列 IDE 协议版本，但实际上它们只对较新的版本做出可靠的响应。由于协议版本非常相似，这些驱动器通常都能正常工作，但偶尔出现的不匹配会导致驱动器和内核对驱动器状态产生分歧。这会由于内核问题而悄无声息地损坏数据。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以处理这些协议不匹配的情况。

之前，由于 Linux 2.2 内核的成本，我们遇到了一些问题 fsync()。它的成本与文件大小成正比，而不是与修改部分的大小成正比。这对于我们的大型操作日志来说是一个问题，尤其是在我们实现检查点之前。我们曾一度通过使用同步写入来解决这个问题，最终迁移到了 Linux 2.4。

Linux 的另一个问题是单个读写锁，当地址空间中的任何线程从磁盘调入页面（读取锁）或修改地址空间时，都必须持有该锁。mmap()调用（写锁）。我们在轻负载下观察到系统出现短暂的超时，并努力寻找资源瓶颈或偶发的硬件故障。最终，我们发现，当磁盘线程正在分页之前映射的数据时，这把锁阻塞了主网络线程将新数据映射到内存中。由于我们主要受网络接口而非内存复制带宽的限制，我们通过替换mmap()和预读（）需额外支付一份副本的费用。

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。我们会在适当的时候改进内核，并与开源社区分享这些改进。

8.相关工作

与其他大型分布式文件系统（例如 AFS [5]）一样，GFS 提供了一个位置独立的命名空间，允许透明地移动数据，从而实现负载均衡或容错。与 AFS 不同的是，GFS 将文件数据分散到多个存储服务器，这种方式更类似于 xFS [1] 和 Swift [3]，旨在提供总体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，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] 等系统不同，GFS 在文件系统接口下不提供任何缓存。我们的目标工作负载在单个应用程序运行中几乎无法重用，因为它们要么在大型数据集中流式传输，要么在数据集内随机寻址并每次读取少量数据。

一些分布式文件系统，如 Frangipani、xFS、Minnesota 的 GFS[11] 和 GPFS [10] 都移除了集中式服务器并依靠分布式算法来保持一致性和管理。我们选择集中式方法是为了简化设计、提高可靠性并获得灵活性。具体而言，集中式主服务器可以更轻松地实现复杂的块放置和复制策略，因为主服务器已经拥有大部分相关信息并控制其更改方式。我们通过保持主服务器状态较小并在其他机器上完全复制来解决容错问题。我们的影子主服务器机制目前提供可扩展性和高可用性（对于读取而言）。通过附加到预写日志来持久化对主服务器状态的更新。因此，我们可以采用类似 Harp [7] 中的主副本方案，以提供高可用性以及比我们当前方案更强的一致性保证。

我们正在解决与 Lustre [8] 类似的问题，即为大量客户端提供总体性能。然而，我们通过专注于应用程序的需求而不是构建符合 POSIX 标准的文件系统，大大简化了这个问题。此外，GFS 假设存在大量不可靠的组件，因此容错能力是我们设计的核心。

GFS 与 NASD 架构最为相似 [4]。NASD 架构基于网络连接磁盘驱动器，而 GFS 使用商用机器作为块服务器，就像 NASD 原型一样。与 NASD 不同的是，我们的块服务器使用惰性分配的固定大小块，而不是可变长度的对象。此外，GFS 还实现了生产环境中所需的重新平衡、复制和恢复等功能。

与明尼苏达州的GFS和NASD不同，我们并不寻求改变存储设备的模型。我们专注于利用现有的商用组件来满足复杂分布式系统的日常数据处理需求。

由原子记录追加功能支持的生产者-消费者队列解决了与 River [2] 中的分布式队列类似的问题。River 使用基于内存的队列，分布在不同的机器上，并进行严格的数据流控制，而 GFS 使用一个持久文件，可供多个生产者并发追加数据。River 模型支持 m 对 n 的分布式队列，但缺乏持久存储所具备的容错能力，而 GFS 仅高效地支持 m 对 1 的队列。多个消费者可以读取同一个文件，但它们必须协调以对传入的负载进行分区。

9.结论

Google 文件系统展现了在商用硬件上支持大规模数据处理工作负载的必要特性。虽然有些设计决策是针对我们的独特场景而设计的，但许多设计决策可能适用于类似规模和成本意识的数据处理任务。

我们首先根据当前和预期的应用工作负载和技术环境，重新审视了传统的文件系统假设。这些观察最终在设计空间中引出了截然不同的观点。我们将组件故障视为常态而非例外，并针对主要以追加操作（可能并发）和读取操作（通常顺序执行）的大型文件进行优化，同时扩展和放宽了标准文件系统接口，以改进整个系统。

我们的系统通过持续监控、关键数据复制以及快速自动恢复来提供容错能力。块复制使我们能够容忍块服务器

故障。这些故障的频繁发生促使我们设计了一种新颖的在线修复机制，该机制定期透明地修复损坏，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，考虑到系统中磁盘的数量，这种情况非常常见。

我们的设计能够为执行各种任务的众多并发读写器提供高聚合吞吐量。我们通过将文件系统控制（通过主服务器）与数据传输（直接在块服务器和客户端之间传递）分离来实现这一点。通过较大的块大小和块租约（将数据变更的权限委托给主副本），最大限度地减少了主服务器对常见操作的参与。这使得简单、集中式的主服务器成为可能，并且不会成为瓶颈。我们相信，我们网络堆栈的改进将突破当前单个客户端写入吞吐量的限制。

GFS 成功满足了我们的存储需求，并在 Google 内部被广泛用作研发和生产数据处理的存储平台。它是我们能够持续创新、攻克整个网络规模难题的重要工具。

致谢

我们要感谢以下人员对系统或论文的贡献。Brain Bershad（我们的指导者）和匿名审阅者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 致力于跨块服务器副本的比较。Guy Edjlali 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 对论文的早期草稿发表了评论。我们在 Google 的许多同事勇敢地将他们的数据信任给新的文件系统，并给了我们有用的反馈。Yoshka 帮助了早期测试。

参考

[1] Thomas Anderson、Michael Dahlin、Jeanna Neefe、David Patterson、Drew Roselli 和 Randolph Wang。无服务器网络文件系统。*第十五届 ACM 操作系统原理研讨会论文集*，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 集群 I/O：让快速用例变得常见》。在*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] Luis-Felipe Cabrera 和 Darrell DE Long. Swift：使用分布式磁盘条带化来提供高 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。一种经济高效的高带宽存储建筑。在*第八届编程语言和操作系统的架构支持会议论文集*，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 Sidebotham 和 Michael West。分布式文件系统的规模和性能。*ACM 计算机系统学报*, 6(1):51–81, 1988年2月。

[6] 国际梅佐。 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*第十三届操作系统原理研讨会*，第 226-238 页，加利福尼亚州太平洋格罗夫，1991 年 10 月。

[8] Lustre. http://www.lustreorg，2003 年。

[9] David A. Patterson、Garth A. Gibson 和 Randy H. Katz。廉价磁盘冗余阵列 (RAID) 案例。在*1988 年 ACM SIGMOD 国际数据管理会议论文集*，第 109-116 页，伊利诺伊州芝加哥，1988 年 9 月。

[10] Frank Schmuck 和 Roger Haskin。GPFS：面向大型计算集群的共享磁盘文件系统。*第一届 USENIX 文件和存储技术会议论文集*，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. O'Keefe. 全球文件系统。*第五届美国宇航局戈达德太空飞行中心大容量存储系统和技术会议论文集*，马里兰州学院公园，1996 年 9 月。

[12] Chandramohan A. Thekkath、Timothy Mann 和 Edward K. Lee. Frangipani：一种可扩展的分布式文件系统。*第16届ACM操作系统原理研讨会论文集*，第 224-237 页，法国圣马洛，1997 年 10 月。