谷歌文件系统

Sanjay Ghemawat, Howard Gobioff，和 shuntak Leung

谷歌*

摘要

我们设计并实现了谷歌文件系统，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。它在廉价的商用硬件上运行时提供了容错能力，并为大量客户端提供了高聚合性能。

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我们的应用程序工作负载和技术环境的观察所驱动的，包括当前的和预期的，这反映了与一些早期文件系统假设的显著背离。这让我们重新审视了传统的选择，并探索了完全不同的设计点。

文件系统成功地满足了我们的存储需求。它被广泛部署在谷歌中，作为我们服务使用的数据生成和处理的存储平台，以及需要大型数据集的研究和开发工作。迄今为止最大的集群提供了超过 1000 台机器上的数千个磁盘的数百 tb 的存储，并由数百个客户端并发访问。

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现实世界使用的测量结果。

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

关键字

容错，可扩展性，数据存储，集群存储

* 作者可在以下地址联系到:

{桑杰,hgobioff, shuntak} @google.com。

2003 年 10 月 19 日至 22 日，美国纽约博尔顿机场。Copyright 2003 ACM 1-58113-757-5/03/0010…$5.00。

1.介绍

我们设计并实现了谷歌文件系统(GFS)，以满足谷歌快速增长的数据处理需求。GFS 与以前的分布式文件系统有许多相同的目标，如性能、可扩展性、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显著背离。我们重新审视了传统的选择，并在设计空间中探索了完全不同的点。

首先，组件故障是常态，而不是例外。文件系统由数百甚至数千台存储机器组成，这些存储机器由廉价的商品部件组装而成，并由数量相当的客户机访问。组件的数量和质量实际上保证了一些在任何给定的时间都不能正常工作，而一些则无法从当前的故障中恢复。我们已经看到了由应用程序 bug、操作系统 bug、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组成部分。

其次，按照传统的标准，文件是巨大的。多 gb 文件很常见。每个文件通常包含许多应用程序对象，如 web 文档。当我们定期处理由许多 TBs 组成的快速增长的数据集时，即使在文件系统可以支持它的情况下，管理数十亿个大约 kb 大小的文件也是笨拙的。因此，设计假设和参数，如 I/O 操作和块大小，必须重新访问。

第三，大多数文件是通过追加新数据而不是覆盖现有数据来改变的。文件内的随机写入实际上是不存在的。一旦写入，文件就只能被读取，而且通常只能按顺序读取。各种数据都具有这些特征。其中一些可能构成数据分析程序扫描的大型存储库。有些可能是运行中的应用程序不断生成的数据流。有些可能是存档数据。有些可能是在一台机器上产生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。考虑到巨大文件上的这种访问模式，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则失去了吸引力。

第四，联合设计应用程序和文件系统 API 可以增加我们的灵活性，从而使整个系统受益。

例如，我们放宽了 GFS 的一致性模型，以极大地简化文件系统，而不会给应用程序带来繁重的负担。我们还引入了原子追加操作，使得多个客户端可以并发地追加到一个文件，而不需要它们之间额外的同步。这些将在本文后面进行更详细的讨论。

目前部署了多个 GFS 集群，用于不同的目的。最大的集群有超过 1000 个存储节点，超过 300 TB 的磁盘存储，并由不同机器上的数百个客户连续访问。

2.设计概述 2.1 假设

在为我们的需求设计文件系统时，我们一直受到一些假设的指导，这些假设既提供了挑战，也提供了机遇。我们之前提到了一些关键的观察结果，现在更详细地列出我们的假设。

- 这个系统是由许多经常失效的廉价商品组成的。它必须不断地监控自己，并在日常基础上检测、容忍和迅速从组件故障中恢复。
- 系统存储适度数量的大文件。我们预计有几百万个文件，每个文件的大小通常为 100 MB 或更大。多 gb 文件是常见情况，应该有效管理。小文件必须支持，但我们不需要针对它们进行优化。

工作负载主要由两种读取组成:大的流读取和小的随机读取。在大型流读取中，单个操作通常读取数百 KBs，更常见的是 1 MB 或更多。来自同一客户端的连续操作通常通过文件的一个连续区域读取。一个小的随机读取通常以任意的偏移量读取几个 KBs。注重性能的应用程序通常会对小的读取进行批量处理和排序，以稳定地遍历文件，而不是来回读取。

- 工作负载也有许多向文件追加数据的大的、连续的写入。典型的操作大小与读操作类似。文件一旦写入，就很少再修改。支持文件中任意位置的小写入，但不要求效率高。
- 系统必须高效地为并发地追加到同一个文件的多个客户端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，将并发地追加到一个文件。原子性与最小的同步开销是必不可少的。文件可能稍后读取，或者消费者可能同时读取整个文件。

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重视以高速率批量处理数据，而很少有对单个读或写有严格的响应时间要求。

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。文件在目录中被分层组织，并由路径名标识。我们支持通常的操作，如创建、删除、打开、关闭、读取和写入文件。

此外，GFS 有快照和记录追加操作。Snapshot 以较低的成本创建文件或目录树的副本。记录追加允许多个客户端并发地向同一个文件追加数据，同时保证每个单独客户端的追加操作的原子性。它对于实现多路合并结果和生产者-消费者队列非常有用，许多客户端可以同时追加，而不需要额外的锁定。我们发现这些类型的文件在构建大型分布式应用程序时是非常宝贵的。快照和记录追加分别在 3.4 节和 3.3 节中进一步讨论。

2.3 体系结构

一个 GFS 集群由一个 master 和多个 chunkserver 组成，由多个客户端访问，如图 1 所示。这些服务器中的每一个都是运行用户级服务器进程的典型的商用 Linux 机器。只要机器资源允许，并且运行可能不可靠的应用程序代码所导致的低可靠性是可以接受的，在同一台机器上同时运行 chunkserver 和客户端是很容易的。

文件被划分为固定长度的块。每个分块由一个不可变且全局唯一的 64 位分块句柄标识，该句柄由 master 在创建分块时分配。Chunkservers 将块以 Linux 文件的形式存储在本地磁盘上，并通过块句柄和字节范围指定读写块数据。为了保证可靠性，每个分块都被复制到多个 chunkserver 上。默认情况下，我们存储三个副本，尽管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 维护所有的文件系统元数据。这包括命名空间，访问控制信息，从文件到块的映射，以及块的当前位置。它还控制系统范围内的活动，如块租赁管理，孤儿块的垃圾收集，以及 chunkservers 之间的块迁移。master 定期通过心跳消息与每个 chunkserver 通信，给它指令并收集它的状态。

链接到每个应用程序中的 GFS 客户端代码实现了文件系统 API，并与 master 和 chunkserver 通信，代表应用程序读取或写入数据。客户端与 master 交互进行元数据操作，但所有承载数据的通信都直接到 chunkservers。我们不提供 POSIX API，因此不需要 hook 到 Linux vnode 层。

客户端和 chunkserver 都不缓存文件数据。客户端缓存没有什么好处，因为大多数应用程序都要通过巨大的文件，或者工作集太大而无法缓存。没有客户端缓存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。)chunkserver 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频繁访问的数据保存在内存中。

2.4 单主服务器

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

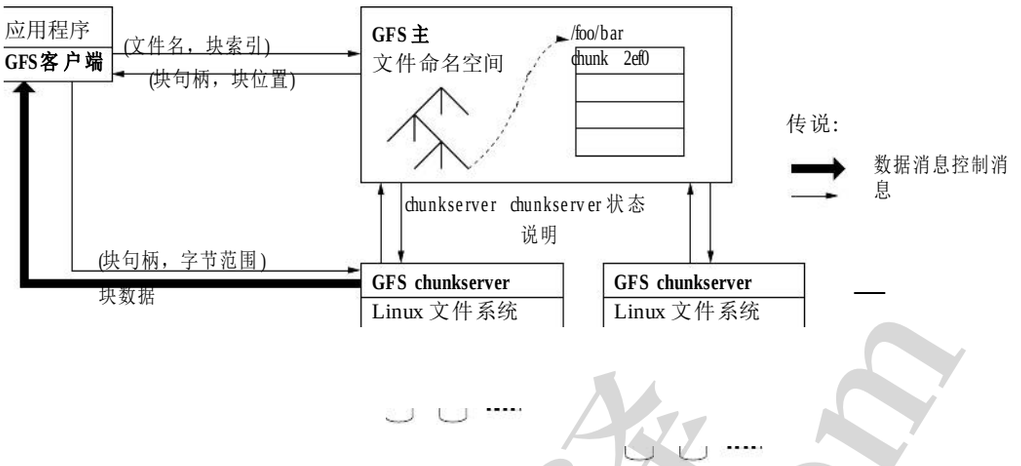

图 1:GFS 架构

以及利用全局知识复制决策。但是，我们必须尽量减少它在读写中的参与，这样它才不会成为瓶颈。客户端从不通过 master 读写文件数据。相反，客户端会询问主服务器它应该联系哪些 chunkserver。它在有限的时间内缓存这些信息，并直接与 chunkservers 交互以进行许多后续操作。

让我们参考图 1 简单地解释一下交互。首先，使用固定的块大小，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 返回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。

然后客户端向其中一个副本发送请求，最有可能是最近的副本。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-主交互。事实上，客户端通常会在同一个请求中请求多个块，并且主端也可以在这些请求之后立即包含块的信息。这些额外的信息可以避免几个未来的客户端-主端交互，而几乎不需要额外的成本。

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。每个块副本以普通 Linux 文件的形式存储在 chunkserver 上，仅在需要时进行扩展。惰性空间分配避免了由于内部碎片而造成的空间浪费，这可能是反对如此大的块大小的最大反对意见。

大的块大小提供了几个重要的优点。首先，它减少了客户端与主服务器交互的需要，因为对同一个块的读写只需要向主服务器提出一次初始请求，就可以获得块的位置信息。这种减少对于我们的工作负载尤其显著，因为应用程序大多是顺序读写大文件。即使是小的随机读取，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，由于在一个大的块上，客户端更有可能对一个给定的块执行许多操作，它可以通过保持一个 persis-来减少网络开销

帐篷 TCP 连接到 chunkserver 在一段较长的时间。第三，它减少了存储在 master 上的元数据的大小。这使得我们可以将元数据保存在内存中，进而带来其他优势，我们将在 2.6.1 节中讨论。

另一方面，大的块大小，即使使用惰性空间分配，也有其缺点。一个小文件由少量块组成，可能只有一个块。如果许多客户端访问同一个文件，存储这些块的 chunkserver 可能会成为热点。在实践中，热点并不是一个主要问题，因为我们的应用程序通常顺序读取大型多块文件。

然而，当 GFS 第一次被批处理队列系统使用时，热点确实出现了: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。存储这个可执行文件的少数 chunkserver 由于数百个同时请求而过载。我们通过存储具有更高复制因子的可执行文件和使批处理队列系统错开应用程序启动时间来解决这个问题。一个潜在的长期解决方案是在这种情况下允许客户端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

以及分块命名空间，文件到分块的映射，以及每个分块副本的位置。所有的元数据都保存在 master 的内存中。前两种类型(名称-空间和文件-块映射)也通过将变化记录到存储在 master 本地磁盘上的操作日志并复制到远程机器上来保持持久性。使用日志允许我们简单、可靠地更新主状态，并且在主崩溃的情况下不会有不一致的风险。master 不会持久化存储块位置信息。相反，它会在 master 启动时以及每当一个 chunkserver 加入集群时询问每个 chunkserver 关于它的块的信息。

2.6.1 内存中的数据结构

由于元数据存储在内存中，因此主操作速度很快。此外，master 在后台定期扫描它的整个状态也很容易和高效。这种周期性扫描用于实现块垃圾收集，在 chunkserver 出现故障时进行重新复制，以及块迁移以平衡负载和磁盘空间

跨 chunkserver 的使用。4.3 节和 4.4 节将进一步讨论这些活动。

这种只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到主服务器拥有的内存数量的限制。这在实践中并不是一个严重的限制。master 为每个 64 MB 的块维护少于 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满。类似地，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。

如果有必要支持更大的文件系统，那么为 master 增加额外内存的成本是为我们通过将元数据存储在内存中获得的简单性、可靠性、性能和灵活性所付出的小代价。

2.6.2 块位置

对于哪些 chunkserver 有给定块的副本，主服务器不会保存一个持久的记录。它只是在启动时轮询 chunkserver 以获取这些信息。此后，master 可以让自己保持最新状态，因为它控制所有块的放置，并通过定期的心跳消息监控 chunkserver 的状态。

我们最初试图在 master 上持久地保持块的位置信息，但我们决定在启动时从 chunkservers 请求数据更简单，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、重新启动等情况下保持 master 和 chunkservers 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。

另一种理解这种设计决策的方法是认识到，chunkserver 对它自己的磁盘上有或没有什么块有最终决定权。试图在主服务器上维护这一信息的一致视图是没有意义的，因为 chunkserver 上的错误可能会导致块自动消失(例如，磁盘可能会坏掉并被禁用)或操作员可能会重命名 chunkserver。

2.6.3 操作日志

操作日志记录了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，而且还充当了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(参见 4.5 节)，都是由它们被创建的逻辑时间唯一而永久地标识的。

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，不要让更改对客户端可见。否则，即使块本身存活下来，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，我们在多个远程机器上复制它，并仅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 在刷写之前将几个日志记录一起批处理，从而减少了刷写和复制对整体系统吞吐量的影响。

master 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，我们必须保持日志较小。当日志增长超过一定大小时，master 会检查它的状态，以便它可以通过从本地磁盘加载最新的检查点并仅重放日志来恢复

	写记录追加	
串行成功并发成功失败	定义定义	穿插着一致的不一
	致但未定义	
	不一致的	

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类似 b 树的形式，可以直接映射到内存中，用于命名空间查找，而不需要额外的解析。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，master 的内部状态是这样构造的，可以在不延迟到来的突变的情况下创建一个新的检查点。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。新的检查点包括了 switch 之前的所有突变。对于一个有几百万文件的集群来说，它可以在一分钟左右创建出来。完成后，它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的日志文件。旧的检查点和日志文件可以自由删除，尽管我们保留了一些以防止灾难发生。检查点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 有一个宽松的一致性模型，可以很好地支持我们的高度分布式应用程序，但仍然保持相对简单和高效的实现。我们现在讨论 GFS 的保证以及它们对应用程序的意义。我们还强调了 GFS 如何维护这些保证，但将细节留给论文的其他部分。

2.7.1 GFS 的保证

文件命名空间的变化(例如，文件创建)是原子的。它们由 master 独家处理:命名空间锁定保证了原子性和正确性(第 4.1 节);master 的操作日志定义了这些操作的全局总顺序(章节 2.6.3)。

一个文件区域在数据突变后的状态取决于突变的类型，是否成功或失败，以及是否有并发突变。表 1 对结果进行了总结。如果所有客户端总是看到相同的数据，无论从哪个副本读取数据，那么文件区域就是一致的。一个区域是在文件数据突变后定义的，如果它是一致的，客户端将看到突变所写的全部内容。当一个修改成功而没有并发写入者的干扰时，受影响的区域就被定义了(并且暗示是一致的):所有的客户端总是会看到修改所写的内容。并发的成功突变会留下未定义但一致的区域:所有客户端看到相同的数据，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杂片段组成。一个失败的突变使区域不一致(因此也未定义):不同的客户端可能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类型的未定义区域。

数据突变可以是写入或添加记录。写入导致数据被写入应用程序指定的文件偏移。记录追加导致数据(“记录”)被原子追加至少一次，即使是在并发突变存在的情况下，但在 GFS 选择的偏移量(第 3.3 节)。(相比之下，“常规”追加仅仅是在客户端认为是当前文件结束的偏移量处进行的一次写入。)偏移量返回给客户端，并标记包含该记录的已定义区域的开始。此外，GFS 可能在中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，通常与用户数据量相比显得微不足道。

在一系列成功修改之后，修改后的文件区域可以保证是定义好的，并且包含了上次修改后写入的数据。GFS 通过(a)在它的所有副本上以相同的顺序对一个块应用突变(3.1 节)，(b)使用块版本号来检测任何已经过时的副本，因为它在它的 chunkserver 关闭时错过了突变(4.5 节)。过期的副本将永远不会涉及到突变或提供给客户端请求主块位置。它们会在最早的时候被垃圾收集。

由于客户端缓存了块的位置，他们可能会在信息刷新之前从过时的副本中读取数据。这个窗口受到缓存条目的超时和文件的下一次打开的限制，该文件将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是只追加的，过期的副本通常返回的是块的提前结束，而不是过期的数据。当读取器重试并联系主服务器时，它将立即获得当前块的位置。

在一次成功的突变之后很长一段时间，组件故障当然仍然会损坏或破坏数据。GFS 通过主服务器和所有 chunkserver 之间的常规握手来识别失败的 chunkserver，并通过校验和来检测数据损坏(章节 5.2)。一旦问题浮出水面，数据就会尽快从有效副本中恢复(章节 4.3)。只有当所有副本在 GFS 能够做出反应之前(通常在几分钟内)丢失时，一个数据块才会不可逆转地丢失。即使在这种情况下，它也不可用，而不是损坏:应用程序收到明确的错误，而不是损坏的数据。

对应用程序的影响

GFS 应用程序可以通过一些其他目的已经需要的简单技术来适应宽松的一致性模型:依赖于附加而不是覆盖、检查点，以及编写自验证、自识别的记录。

实际上，我们所有的应用程序都是通过追加而不是覆盖来修改文件的。在一个典型的使用中，一个写入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一个永久名称，或者定期检查已成功写入的数据量。检查点也可能包括应用程序级别的校验和。读取器只验证和处理文件区域直到最后一个检查点，这是已知的在定义的状态。抛开一致性和并发性问题不谈，这种方法对我们很有帮助。与随机写入相比，追加操作的效率更高，对应用程序失败的适应性更强。检查点允许写入器增量地重新启动，并阻止读取器成功地处理写入从应用程序的角度来看，仍然不完整的文件数据。

在另一种典型的使用中，许多写入器并发地将结果追加到文件中，以合并结果或作为生产者-消费者队列。Record append 的“至少追加一次”语义保存了每个写入者的输出。读者对偶尔出现的填充和重复的处理如下。作者准备的每条记录都包含校验和之类的额外信息，以便验证其有效性。读取器可以使用校验和识别并丢弃额外的填充和记录片段。如果它不能容忍偶尔出现的重复(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过滤掉它们，这些标识符通常需要用于命名相应的应用程序实体，如 web 文档。这些用于记录 I/O 的功能(除了删除重复)在我们的应用程序共享的库代码中，并适用于谷歌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，总是被交付给记录阅读器。

3.系统的交互

我们设计这个系统是为了尽量减少 master 在所有操作中的参与。有了这个背景，我们现在描述客户端、master 和 chunkservers 如何交互来实现数据突变、原子记录追加和快照。

3.1 租期和突变顺序

mutation 是一种改变块的内容或元数据的操作，例如写操作或追加操作。每次变异都是在块的所有副本上执行的。我们使用租约来维护副本之间一致的突变顺序。master 授予其中一个 replica 一个 chunk lease，我们称之为 primary。主副本为块的所有突变选择一个串行顺序。所有副本在应用突变时都遵循这个顺序。因此，全局的突变顺序首先由 master 选择的租赁授予顺序来定义，在租赁内由 primary 分配的序列号来定义。

租约机制旨在最小化主服务器的管理开销。租约的初始超时时间为 60 秒。然而，只要块被修改，主服务器就可以无限期地请求并从主服务器获得扩展。这些扩展请求和授权被依附在主服务器和所有 chunkserver 之间定期交换的心跳消息上。master 有时可能会试图在租约到期前撤销它(例如，当 master 想要禁用正在被重命名的文件的更改时)。即使主节点失去了与主节点的通信，它也可以在旧租约到期后安全地将新的租约授予另一个副本。

在图 2 中，我们通过遵循写操作的控制流通过这些编号的步骤来说明这个过程。

1. 客户端询问主服务器哪个 chunkserver 持有当前块的租约以及其他副本的位置。如果没有人拥有租约，主服务器会将租约授予它选择的副本(图中未显示)。

2. 主副本回复主副本的身份和其他(次要)副本的位置。客户端缓存这些数据以备将来的突变。它只需要在 primary 时再次联系 master

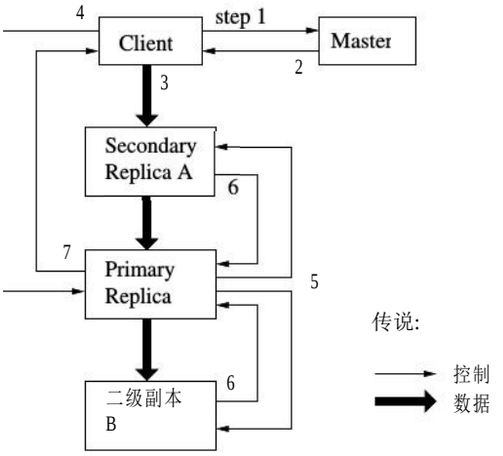

图 2:写入控制和数据流

变得不可达或回复它不再持有租约。

3. 客户端将数据推送到所有副本。客户端可以以任何顺序执行此操作。每个 chunkserver 都会将数据存储在内部的 LRU 缓冲缓存中，直到数据被使用或老化。通过将数据流与控制流解耦，我们可以根据网络拓扑来调度昂贵的数据流，从而提高性能，而不管哪个 chunkserver 是主服务器。3.2 节进一步讨论了这一点。

4. 一旦所有的副本都确认收到了数据，客户端就向主服务器发送写请求。该请求识别了之前推送给所有副本的数据。主副本为它接收到的所有突变分配连续的序列号，可能来自多个客户端，这提供了必要的序列化。它以序列号的顺序将突变应用到自己的本地状态。

5. 主副本将写请求转发给所有从副本。每个从副本按照主副本分配的相同序列号顺序应用突变。

6. 次用户都回复主用户，表示他们已经完成了操作。

7. 主服务器回复客户端。在任何副本中遇到的任何错误都会报告给客户端。在发生错误的情况下，写入可能在主副本和任意一个副副本子集上成功。(如果在主副本失败，它就不会被分配一个序列号并转发。)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 mutation 来处理这样的错误。它将在步骤(3)到(7)进行几次尝试，然后回落到从写入开始重试。

如果应用程序的写操作很大或者跨越了块的边界，GFS 客户端代码会将其分解为多个写操作。它们都遵循上面描述的控制流，但可能与来自其他客户端的并发操作交叉并被覆盖。因此，共享的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，因为所有副本上的单个操作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 2.7 节所述。

3.2 数据流

我们将数据流与控制流解耦，以高效地使用网络。当控制流从客户端流向主服务器，然后流向所有辅助服务器时，数据以流水线的方式沿着精心挑选的 chunkserver 链线性地推进。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，并最小化通过所有数据的延迟。

为了充分利用每台机器的网络带宽，数据是沿着 chunkservers 链线性推进的，而不是分布在其他一些拓扑结构中(例如，树)。因此，每台机器的全部出站带宽被用来尽可能快地传输数据，而不是分配给多个接收者。

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)，每个机器都将数据转发到网络拓扑中“最近”的机器，但数据尚未收到。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 chunkserver，比如 S1。S1 将它转发给最近的 chunkserver S2，通过 S4 最接近 S1，比如 S2。类似地，S2 把它转发给 S3 或 S4，哪个更接近 S2，以此类推。我们的网络拓扑结构足够简单，可以从 IP 地址准确地估算出“距离”。

最后，我们通过在 TCP 连接上对数据传输进行管道化来最小化延迟。一旦 chunkserver 接收到一些数据，它就立即开始转发。管道对我们特别有用，因为我们使用全双工链路的交换式网络。立即发送数据并不会降低接收速率。在没有网络拥塞的情况下，向 R 副本传输 B 字节的理想运行时间是 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是 100 Mbps (T)，而 L 远低于 1 ms。因此，1 MB 理想情况下可以在大约 80 ms 内分发。

3.3 原子记录追加

GFS 提供了一个名为 record append 的原子追加操作。在传统的写操作中，客户端指定要写入数据的 off- set。并发写同一个 region 是不可序列化的:region 可能最终包含来自多个客户端的数据片段。然而，在记录追加中，客户端只指定了数据。GFS 按照 GFS 选择的偏移量(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，当多个写入器并发地以 O 追加模式打开一个文件时，在没有竞争条件的情况下进行写入。

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并发地追加同一个文件。如果使用传统的写操作，客户端将需要额外复杂和昂贵的同步，例如通过分布式锁管理器进行同步。在我们的工作负载中，这样的文件经常

作为多生产者/单消费者队列，或者包含来自许多不同客户端的合并结果。

记录追加是一种突变，遵循 3.1 节中的控制流，只是在主节点有一点额外的逻辑。客户端将数据推送到文件最后一块的所有副本，然后将其请求发送给主服务器。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，它将块填充到最大大小，告诉次要服务器做同样的操作，并回复客户端，指示该操作应该在下一个块上重试。(记录追加被限制为最大块大小的四分之一，以使最差情况下的碎片保持在可接受的水平。)如果记录符合最大大小，这是常见的情况，主服务器将数据追加到它的副本中，告诉辅助服务器在它所拥有的精确偏移位置写入数据，最后向客户端响应成功。

如果任何一个副本的记录追加失败，客户端就会重试该操作。结果，同一块的副本可能包含不同的数据，其中可能包括相同记录的全部或部分副本。GFS 不保证所有的副本按字节顺序是相同的。它只保证数据至少以原子单位写入一次。这个属性很容易从一个简单的观察中得到，即为了使操作报告成功，数据必须在某个块的所有副本上的相同偏移量写入。此外，在这之后，所有的副本都至少和记录的末尾一样长，因此任何未来的记录都会被分配一个更高的偏移量或不同的块，即使不同的副本后来成为主副本。在我们的一致性保证方面，成功的记录追加操作已经写入数据的区域是定义的(因此是一致的)，而中间的区域是不一致的(因此是未定义的)。正如我们在 2.7.2 节中讨论的，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复制一个文件或目录树(“源”)，同时尽量减少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巨大数据集的分支副本(通常是这些副本的副本，递归)，或者在试验更改之前对当前状态进行检查点，这些更改稍后可以轻松提交或回滚。

像 AFS[5]一样，我们使用标准的写时复制技术来实现快照。当 master 收到快照请求时，它首先撤销即将快照的文件块上所有未到期的租约。这确保了任何对这些块的后续写入都需要与 master 进行交互，以找到租约的持有者。这将给 master 一个机会先创建一个新的数据块的副本。

在租约被撤销或过期之后，master 会将操作记录到磁盘上。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在内存中的状态。新创建的快照文件指向与源文件相同的块。

在快照操作之后，客户端第一次想要写入一个 chunk C 时，会向 master 发送一个请求，寻找当前的租约持有者。master 注意到 chunk C 的引用计数大于 1。它推迟回复客户端的请求，并选择一个新的块处理 C”。然后它要求每个当前拥有 C 副本的 chunkserver 创建一个名为 C`的新块。通过在与原始块相同的 chunkserver 上创建新块，我们可以确保数据可以在本地复制，而不是通过网络(我们的磁盘速度大约是我们 100 Mb 以太网链路的三倍)。从这一点开始，请求处理与任何块都没有什么不同:主授予其中一个副本一个新的块 C’的租约，并回复客户端，客户端可以正常写入块，而不知道它刚刚从现有块创建。

4.主操作

master 执行所有的命名空间操作。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，从而创建副本，并协调各种系统范围内的活动，以保持块的完全复制，在所有 chunkserver 之间平衡负载，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上的 chunkserver 租约。我们不希望在其他主操作运行时延迟它们。因此，我们允许多个操作被激活，并在命名空间的区域上使用锁以确保正确的序列化。

与许多传统的文件系统不同，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。通过前缀压缩，这个表可以在内存中有效地表示。命名空间树中的每个节点(无论是绝对的文件名还是绝对的目录名)都有一个相关的读写锁。

每个主操作在运行前都会获取一组锁。一般情况下，如果涉及/d1/d2/…/dn/leaf，它将获得目录名称/d1、/d1/d2、…, /d1、d2 /……/dn，以及全路径名/d1/d2/…/dn/叶子上的读锁或写锁。注意，根据操作的不同，leaf 可能是一个文件或目录。

我们现在举例说明，当/home/user 被快照到/save/user 时，这种锁定机制如何阻止文件/home/user/foo 被创建。快照操作获得/home 和/save 上的读锁，以及/home/user 和/save/user 上的写锁。文件创建获得/home 和/home/user 的读锁，以及/home/user/foo 的写锁。这两个操作将被正确序列化，因为它们试图在/home/user 上获得冲突的锁。文件创建不需要父目录上的写锁，因为没有“目录”或类似 inode 的数据结构来保护不被修改。名称上的读锁足以保护父目录不被删除。

这种加锁方案的一个很好的特性是，它允许在同一个目录中并发修改。例如，可以在同一个目录中并发地执行多个文件创建:每个创建都获得对目录名的读锁和对文件名的写锁。目录名称上的读锁足以防止目录被删除、重命名或快照。对目录的写锁定

文件名序列化尝试创建同名文件两次。

由于命名空间可以有许多节点，读写锁对象是惰性分配的，一旦它们不使用就删除。此外，锁以一致的总顺序获得，以防止死锁:它们首先在命名空间树中按级别排序，并按字典序在同一级别内。

4.2 副本放置

GFS 集群在多个级别上高度分布。它通常有数百个 chunkserver 分布在许多机架上。这些 chunkserver 反过来可能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以跨越一个或多个网络交换机。此外，进入或流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，最大化网络带宽利用率。对于两者来说，仅仅在机器之间散布副本是不够的，这只会防止磁盘或机器故障，并充分利用每台机器的网络带宽。我们还必须在机架之间散布块副本。这确保了即使整个机架损坏或脱机(例如，由于网络交换机或电源电路等共享资源的故障)，一个块的一些副本仍能存活并保持可用。这也意味着，一个 chunk 的流量，特别是读取，可以利用多个机架的聚合带宽。另一方面，写流量必须流经多个机架，这是我们要做的权衡

心甘情愿。

4.3 创造、再复制、再平衡

创建块副本有三个原因:块创建、再复制和再平衡。

当 master 创建一个 chunk 时，它会选择在哪里放置最初的空副本。它会考虑几个因素。(1)我们希望在低于平均磁盘空间利用率的 chunkserver 上放置新的副本。随着时间的推移，这将使 chunkservers 之间的磁盘利用率均衡。(2)我们希望限制每个 chunkserver 上“最近”创建的数量。虽然创建本身是廉价的，但它可以可靠地预测即将到来的大量写流量，因为块是在写请求时创建的，并且在我们的“添加一次读取多次”的工作负载中，一旦它们完全写入，它们通常实际上变成只读的。(3)如上所述，我们希望将块的副本分散到机架上。

一旦可用副本的数量低于用户指定的目标，主服务器就会重新复制一个块。发生这种情况的原因有很多:一个 chunkserver 变得不可用，它报告说它的副本可能被损坏，它的一个磁盘因为错误而被禁用，或者复制目标被增加。每个需要重新复制的块会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，我们给丢失了两个副本的块比只丢失一个副本的块更高的优先级。此外，我们更喜欢首先为活动文件重新复制块，而不是属于最近删除的文件的块(参见 4.4 节)。最后，为了将故障对运行中的应用程序的影响降到最低，我们提高了任何阻塞客户端进程的块的优先级。

master 选择优先级最高的块，并通过指示某些 chunkserver 直接从现有的有效副本中复制块数据来“克隆”它。新副本的放置目标与创建时的目标类似:均衡磁盘空间利用率，限制任何单个 chunkserver 上的主动克隆操作，以及跨机架传播副本。为了保持克隆流量不超过客户端流量，master 限制了集群和每个 chunkserver 的活动克隆操作数量。此外，每个 chunkserver 通过限制其对源 chunkserver 的读请求来限制其在每个克隆操作上花费的带宽。

最后，主服务器周期性地重新平衡副本:它检查当前的副本分布，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，主服务器逐渐填满一个新的 chunkserver，而不是立即用新的块和随之而来的沉重的写流量淹没它。新副本的放置标准与上面讨论的类似。此外，master 还必须选择移除哪些现有副本。一般来说，它更喜欢删除 chunkserver 上那些空闲空间低于平均水平的副本，以便均衡磁盘空间使用。

4.4 垃圾回收

文件删除后，GFS 不会立即回收可用的物理存储空间。它只在文件和块级别的常规垃圾收集期间懒回收。我们发现这种方法使系统更加简单和可靠。

4.1.1 机制

当应用删除文件时，master 会立即记录删除操作，就像其他更改一样。然而，并不是立即回收资源，而是将文件重命名为包含删除时间戳的隐藏名称。在 master 对文件系统命名空间的定期扫描期间，如果这些隐藏文件已经存在超过三天(间隔可配置)，则会删除任何此类隐藏文件。在此之前，该文件仍然可以以新的、特殊的名称读取，并可以通过重命名恢复正常来恢复删除。当隐藏的文件从命名空间中移除时，它在内存中的元数据就会被删除。这有效地切断了它与所有块的链接。

在对块命名空间进行类似的常规扫描时，master 会识别出孤立的块(即那些无法从任何文件中访问的块)，并擦除这些块的元数据。在与 master 定期交换的心跳消息中，每个 chunkserver 报告它所拥有的块的子集，master 以不再存在于 master 元数据中的所有块的身份作为回复。chunkserver 可以自由删除这些数据块的副本。

10/24/11 讨论

虽然分布式垃圾收集是一个很难的问题，在编程语言的背景下需要复杂的解决方案，但在我们的例子中它是相当简单的。我们可以很容易地识别出所有对块的引用:它们在由 master 专门维护的文件到块的映射中。我们也可以很容易地识别出所有块的副本:它们是每个 chunkserver 上指定目录下的 Linux 文件。任何 master 不知道的这样的副本都是“垃圾”。

与立即删除相比，用于存储回收的垃圾收集方法有几个优点。首先，它在组件故障常见的大规模分布式系统中简单可靠。分块创建可能在某些分块服务器上成功，但在其他分块服务器上失败，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，主服务器必须记住在自己和 chunkserver 失败时重新发送它们。垃圾收集提供了一种统一且可靠的方式来清理任何不知道有用的副本。其次，它将存储回收合并到主服务器的定期后台活动中，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，只有在师傅相对空闲的时候才会做。对于需要及时关注的客户请求，master 可以更及时地做出回应。第三，回收存储的延迟为防止意外、不可逆的删除提供了一个安全网。

在我们的经验中，主要的缺点是，这种延迟有时会阻碍用户在存储紧张时微调使用情况的努力。重复创建和删除临时文件的应用程序可能无法立即重用存储空间。我们通过加速存储回收来解决这些问题，如果删除的文件被明确地再次删除。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，用户可以指定某些目录树内文件中的所有块都将不进行复制存储，任何被删除的文件都将立即且不可撤销地从文件系统状态中删除。

4.5 过期副本检测

如果一个 chunkserver 失败并且在它停机时错过了对块的修改，块副本可能会变得陈旧。对于每个块，主维护一个块版本号，以区分最新和过时的副本。

每当 master 授予一个新的块租约时，它就会增加块版本号并通知最新的副本。master 和这些副本都会在它们的持久化状态下记录新版本号。这发生在任何客户端收到通知之前，因此也就发生在它开始写入数据块之前。如果另一个副本目前不可用，它的块版本号将不会提前。当 chunkserver 重启时，master 会检测到这个 chunkserver 有一个过时的副本，并报告它的分块集及其关联的版本号。如果 master 看到的版本号比它记录中的版本号大，那么 master 就认为它在授予租约时失败了，因此使用更高的版本号作为最新版本。

master 会在定期的垃圾回收中移除过期的副本。在此之前，当它响应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，当 master 通知客户端哪个 chunkserver 持有一个块的租约时，或者当它指示 chunkserver 在克隆操作中从另一个 chunkserver 读取块时，master 会包含块的版本号。客户端或 chunkserver 在执行操作时验证版本号，以便它总是访问最新的数据。

5.容错与诊断

在设计系统时，我们最大的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多地成为常态，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，更糟糕的是，数据被损坏。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，以便在不可避免地发生问题时诊断问题。

5.1 高可用性

在 GFS 集群的数百台服务器中，某些服务器在任何时候都不可用。我们通过两种简单而有效的策略来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。事实上，我们并不区分正常终止和异常终止;服务器通常只是通过杀死进程来关闭。客户端和其他服务器会遇到一个小问题，因为他们的未完成请求超时，重新连接到重新启动的服务器，然后重试。6.2.2 节报告观察到的启动时间。

5.1.2 块复制

如前所述，每个块被复制到不同机架上的多个 chunkserver 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。默认为 3 个。当 chunkserver 脱机或通过校验和验证(参见 5.2 节)检测损坏的副本时，主服务器根据需要克隆现有的副本，以保持每个块的完全复制。虽然复制已经为我们提供了很好的服务，但我们正在探索其他形式的跨服务器冗余，如校验或纠删码，以满足我们日益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现这些更复杂的冗余方案是具有挑战性但可管理的，因为我们的流量主要由附加和读取而不是小的随机写入控制。

5.1.3 主复制

主复制是为了保证可靠性。它的操作日志和检查点被复制到多台机器上。只有在其日志记录被刷新到本地磁盘和所有主副本上之后，状态的突变才被认为是提交。为简单起见，一个主进程仍然负责所有的突变以及在内部改变系统的后台活动，如垃圾收集。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，监控 GFS 之外的基础设施会用复制的操作日志在其他地方启动一个新的主进程。客户端只使用主服务器的规范名称(例如 gfs-test)，这是一个 DNS 别名，如果主服务器被迁移到另一台机器，可以更改它。

此外，“影子”master 提供对文件系统的只读访问，即使在主 master 宕机时也是如此。它们是影子，而不是镜像，因为它们可能会稍微滞后于主主机，通常只有几分之一秒。它们增强了没有被主动修改的文件或不介意得到稍微陈旧的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。可以是什么？

在短时间内可能过时的是文件元数据，如目录内容或访问控制信息。

为了让自己保持知情，影子主读取不断增长的操作日志的副本，并与主节点完全一样，对其数据结构应用相同的更改序列。像主服务器一样，它在启动时轮询chunkserver(此后也不频繁)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。它只在主服务器创建和删除副本的决定所导致的副本位置更新上依赖于主服务器。

5.2 数据完整性

每个 chunkserver 使用校验和来检测存储的数据是否损坏。考虑到一个 GFS 集群通常在数百台机器上有数千个磁盘，它经常会经历磁盘故障，导致读写路径上的数据损坏或丢失。(原因之一见第 7 节。)我们可以使用其他块副本从腐败中恢复，但通过在 chunkservers 之间比较副本来检测腐败是不切实际的。此外，分歧副本可能是合法的:GFS 突变的语义，特别是如前所述的原子记录追加，并不保证相同的副本。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。

一个块被分解成 64 KB 的块。每个块有一个对应的 32 位校验和。与其他元数据一样，校验和保存在内存中，并与日志一起持久存储，与用户数据分开。

对于读取，chunkserver 在将任何数据返回给请求者(无论是客户端还是另一个 chunkserver)之前，会验证与读取范围重叠的数据块的校验和。因此 chunkserver 不会将损坏传播到其他机器。如果一个块与记录的校验和不匹配，chunkserver 会向请求者返回一个错误，并向主服务器报告不匹配。作为回应，请求者将从其他副本读取数据，而主服务器将从另一个副本克隆该块。在一个有效的新副本到位后，主服务器通知报告不匹配的 chunkserver 删除它的副本。

校验和对读取性能几乎没有影响，原因有几个。因为我们的大多数读取都跨越了至少几个块，所以我们只需要读取和校验相对少量的额外数据来进行验证。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，并且校验和计算经常可以与 I/O 重叠。

校验和计算对于追加到块末尾的写操作(与覆盖现有数据的写操作相反)进行了大量优化，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，并为任何由追加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，我们现在没有检测到它，新的校验和值将与存储的数据不匹配，并且在下一次读取该块时，会像往常一样检测到损坏。

相比之下，如果一次写入重写了块的现有范围，我们必须读取并验证被重写的范围的第一个和最后一个块，然后执行写入，和最后计算并记录新的校验和。如果我们在部分覆盖前不验证第一个和最后一个块，新的校验和可能会隐藏未覆盖区域中存在的损坏。

在空闲期间，chunkservers 可以扫描和验证不活动块的内容。这使我们能够在很少被阅读的块中检测到损坏。一旦检测到损坏，主可以创建一个新的未损坏的副本，并删除损坏的副本。这可以防止一个不活跃但已损坏的块副本欺骗主服务器，使其认为它有足够的有效块副本。

5.3 诊断工具

广泛和详细的诊断日志记录在问题隔离、调试和性能分析方面有不可估量的帮助，同时只产生最小的成本。如果没有日志，就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，记录许多重大事件(如 chunkservers 的上升和下降)和所有 RPC 请求和应答。这些诊断日志可以自由删除，而不会影响系统的正确性。然而，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的请求和响应，除了正在读取或写入的文件数据。通过将请求与应答进行匹配，并整理不同机器上的 RPC 记录，我们可以重构整个交互历史来诊断问题。这些日志还可以作为负载测试和性能分析的踪迹。

日志记录的性能影响是最小的(而且远远超过了好处)，因为这些日志是按顺序和异步写入的。最近的事件也被保存在内存中，可用于持续的在线监控。

6.测量

在本节中，我们提出一些微观基准来说明 GFS 架构和实现中固有的瓶颈，以及谷歌中使用的一些真实集群的数字。

6.1 微型基准测试中

我们在一个由一个主服务器、两个主副本、16 个 chunkserver 和 16 个客户端组成的 GFS 集群上测量了性能。注意，设置这个配置是为了便于测试。典型的集群有数百个 chunkserver 和数百个客户端。

所有的机器都配置了双 1.4 GHz PIII 处理器，2 GB 内存，两个 80 GB 5400 rpm 磁盘，以及一个 100 Mbps 的全双工以太网连接到 HP 2524 交换机。所有 19 台 GFS 服务器连接到一台交换机上，所有 16 台客户端连接到另一台交换机上。这两个交换机连接的链路为 1gbps。

但是读

N 客户端同时从文件系统读取数据。每个客户端从一个 320 GB 的文件集中随机选择一个 4mb 的区域读取。这样重复 256 次，每个客户端最终读取 1 GB 的数据。这些 chunkserver 加起来只有 32 GB 内存，所以我们预计在 Linux 缓冲区缓存中最多有 10%的命中率。我们的结果应该接近冷缓存的结果。

图 3(a)显示了 N 个客户端的总读速率及其理论极限。当两个交换机之间的 1 Gbps 链路饱和时，极限峰值为 125 MB/s，或当其 100 Mbps 网络接口饱和时，每个客户端 12.5 MB/s，无论哪一个适用。当只有一个客户端读取时，观察到的读取速率是 10 MB/s，或每个客户端限制的 80%。聚合读取速率达到 94 MB/s，约为 125 MB/s 链路限制的 75%，对于 16 个阅读器，或每个客户端 6 MB/s。效率从 80%下降到 75%，因为随着阅读器数量的增加，多个阅读器同时从同一个 chunkserver 读取的概率也在增加。

6.1.2 写道

N 个客户端同时写 N 个不同的文件。每个客户端以一系列写 1mb 的方式向一个新文件写入 1gb 的数据。总的写入速率及其理论极限如图 3(b)所示。极限在 67 MB/s 时趋于稳定，因为我们需要将每个字节写入 16 个 chunkserver 中的 3 个，每个 chunkserver 都有 12.5 MB/s 的输入连接。

一个客户端的写入速率是 6.3 MB/s，大约是限制的一半。罪魁祸首是我们的网络堆栈。它不能很好地与我们用于推送数据到分块副本的管道方案交互。将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 个客户端总的写速率达到 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个 chunkserver 的可能性变得更大。此外，16 个写入者比 16 个读取者更容易发生碰撞，因为每次写入涉及 3 个不同的副本。

写入速度比我们期望的要慢。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，它也不会显著影响系统交付给大量客户端的总写带宽。

6.1.3 记录追加

图 3(c)显示了记录追加性能。N 个客户端同时追加到单个文件。性能受限于存储文件最后一块的 chunkserver 的网络带宽，与客户端数量无关。它从一个客户端 6.0 MB/s 开始，下降到 16 个客户端 4.8 MB/s，主要是由于拥塞和不同客户端看到的网络传输速率的差异。

我们的应用程序倾向于同时产生多个这样的文件。换句话说，N 个客户端同时对 M 个共享文件进行追加，而 N 和 M 都在几十个或几百个文件中。因此，我们实验中的 chunkserver 网络拥塞在实践中并不是一个重大问题，因为客户端可以在写一个文件时取得进展，而 chunkserver 为另一个文件忙碌。

6.2 真实世界的集群

我们现在检查谷歌中使用的两个集群，它们代表了其他几个类似的集群。Cluster A 经常被 100 多名工程师用于研究和开发。一个典型的任务是由人类用户发起的，耗时长达几个小时。它读取几个 MBs 到几个 TBs 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

集群	一个	B
Chunkservers	342 年	227
可用磁盘空间已用磁盘空	年	
间文件数失效文件数块数 chunkservers 元数据 master 元数据	72tb 180 TB 55 TB 155 TB 735k 737 k 22 k 232 k 992 k 1550k 13 GB 21 GB 48 MB 60 MB	

表 2:两个 GFS 集群的特点

更长的、连续的生成和处理多 tb 数据集，只需要偶尔的人工干预。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 5 项所示，两个集群都有数百个 chunkserver，支持许多 TBs 的磁盘空间，并且是公平但不是完全满的。“已用空间”包括所有块的副本。几乎所有的文件都被复制了三次。因此，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差不多，不过 B 集群的死文件比例更大，即被删除或替换为新版本但存储尚未被回收的文件。它也有更多的块，因为它的文件往往更大。

6.2.2 元数据

chunkserver 在 aggregate 中存储了数十 gb 的元数据，主要是 64 KB 的用户数据块的校验和。在 chunkservers 中保存的唯一的其他元数据是 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 MBs，或者说平均每个文件大约 100 字节。这与我们的假设是一致的，即主存的大小在实践中并不限制系统的容量。大部分的逐文件元数据都是以前缀压缩的形式存储的文件名。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，对于每个分块，我们存储当前副本位置和用于实现写时复制的引用计数。

每台单独的服务器，无论是 chunkservers 还是 master，只有 50 到 100 MB 的元数据。因此，恢复速度很快:在服务器能够应答查询之前，从磁盘读取这些元数据只需要几秒钟。然而，在从所有 chunkserver 获取块位置信息之前，主服务器会有一段时间有点困难——通常为 30 到 60 秒。

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，两个集群都已经上线了大约一周。(这些集群最近已经重新启动，以升级到 GFS 的新版本。)

重启后的平均写速率低于 30 MB/s。当我们进行这些测量时，B 正在爆发写活动，产生大约 100 MB/s 的数据，这产生了 300 MB/s 的网络负载，因为写传播到三个副本。

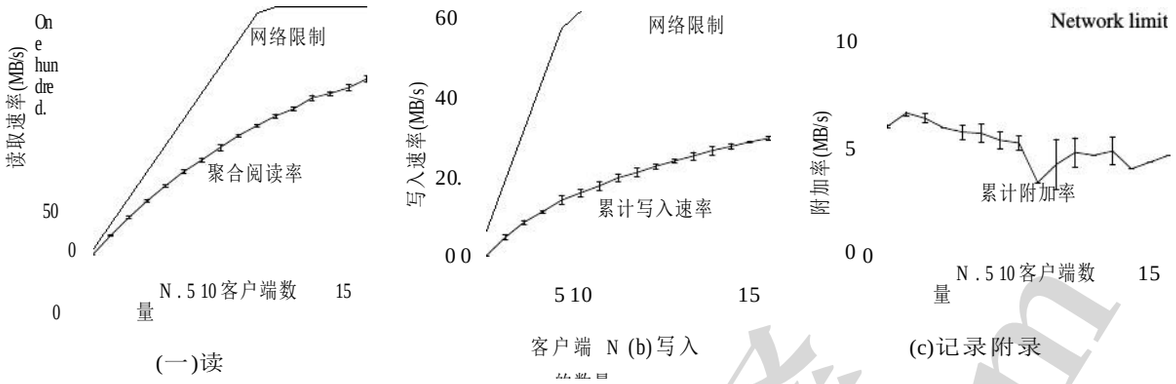

图 3:总吞吐量。上面的曲线显示了我们的网络拓扑施加的理论限制。底部曲线显示测量的吞吐量。它们有显示 95%置信区间的误差条，在某些情况下由于测量的低方差而难以辨认。

集群	一个	B
读速率(最后一分钟)读速率(最近一小时)读速率(重启后)写速率(最近一小时)	583 MB/s 562 MB/s	380 MB / 秒 384 MB / 秒
写速率(重启后)Master ops(最后一分钟)Master ops(最近一小时)Master ops(重启后)	589 MB/s 1 MB / 秒 2 MB/s 25 MB/s 325	49 MB / 秒 101 MB / 秒 117 MB / 秒 13

表 3:两个 GFS 集群的性能指标

读速率远高于写速率。正如我们所假设的那样，总的工作负载包含更多的读操作而不是写操作。两个集群都处于繁重的读取活动中。特别是，A 在前一周一直保持着 580 MB/s 的读取速率。它的网络配置可以支持 750 MB/s，所以它在高效地利用资源。集群 B 可以支持 1300 MB/s 的峰值读取速率，但它的应用程序仅使用 380 MB/s。

6.2.4 主负载

表 3 还显示，发送到 master 的操作速率大约是每秒 200 到 500 个操作。master 可以很容易地跟上这个速率，因此对于这些工作负载来说不是瓶颈。

在 GFS 的早期版本中，master 偶尔会成为某些工作负载的瓶颈。它的大部分时间都是顺序地扫描大型目录(其中包含数十万个文件)，寻找特定的文件。从那以后，我们改变了主数据结构，以允许在命名空间中进行高效的二进制搜索。它现在可以轻松地支持每秒数千次的文件访问。如果有必要，我们可以通过在命名空间数据结构前放置名称查找缓存来进一步加快速度。

6.2.5 恢复时间

在 chunkserver 失败后，一些块将变得复制不足，必须克隆以恢复它们的复制级别。恢复所有这样的块所需的时间取决于资源的数量。在一个实验中，我们在集群 b 中杀死了一个单独的 chunkserver

15000 个块包含 600 GB 的数据。为了限制对运行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(chunkservers 数量的 40%)，其中每个克隆操作最多允许消耗 6.25 MB/s (50 Mbps)。所有块在 23.2 分钟内恢复，有效复制速率为 440 MB/s。

在另一个实验中，我们用大约 16000 块和 660 GB 的数据杀死了两个 chunkserver。这种双重失败将 266 个块减少到只有一个副本。这 266 个块以更高的优先级克隆，并在 2 分钟内全部恢复到至少 2 倍的复制，从而使集群处于可以容忍另一个 chunkserver 故障而不丢失数据的状态。

6.3 工作负载崩溃

在本节中，我们将对两个 GFS 集群上的工作负载进行详细分析，这两个集群与 6.2 节中的工作负载可以比较，但并不完全相同。集群 X 用于研究和开发，集群 Y 用于生产数据处理。

6.3.1 方法和注意事项

这些结果只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产生的工作负载。它们不包括执行客户端请求的服务器间请求或内部后台活动，如转发写入或重新平衡。

I/O 操作的统计是基于 GFS 服务器记录的实际 RPC 请求的启发式重构信息。例如，GFS 客户端代码可能会将一次读取拆分成多个 rpc 来增加并行度，我们由此推断出原始的读取。由于我们的访问模式是高度风格化的，我们期望任何错误都在噪声中。应用程序的显式日志记录可能提供了稍微准确的数据，但从逻辑上来说，重新编译和重启数千个运行中的客户端来做到这一点是不可能的，从尽可能多的机器收集结果也很麻烦。

人们应该注意，不要从我们的工作负载中过度概括。由于谷歌完全控制 GFS 和它的应用程序，应用程序往往是为 GFS 调优的，反之，GFS 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

集群操作	读	写	记录追加 X Y
0 k	X Y	X Y	0 0 0.2 9.2 18.9 15.2
1 b . .1 k 1 k . .8 k 8 k . .64 k 64 k . .128 k 128 k . .256 k 256 k . .512 k 512 k . .1 米 1 m . . inf	0.4 - 2.6 0.1 - 4.1 65.2 - 38.5 29.9 -	0 0 6.6 - 4.9 0.4 - 1.0 17.8 - 43.0	78.0 2.8 < .1 4.3 < .1 10.6 < .1 31.2 2.2 25.5 0.7 2.2

表 4:按大小分解的操作(%)。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。

以及文件系统，但在我们的例子中，效果可能更明显。

6.3.2 Chunkserver 工作负载

表 4 显示了按大小排列的操作分布。读取大小呈双峰分布。小读取(64 KB 以下)来自于搜索密集型客户端，它们在巨大的文件中查找小块数据。大的读取(超过 512 KB)来自于对整个文件的长顺序读取。

在集群 y 中，有相当数量的读操作根本没有返回数据。我们的应用程序，特别是那些在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，而消费者读取文件末尾。偶尔，当消费者的速度超过生产者时，没有数据返回。集群 X 显示这种情况的频率较低，因为它通常用于短期的数据分析任务，而不是长期的分布式应用程序。

写大小也呈现双峰分布。大的写入(超过 256 KB)通常是由于写入器内部的显著缓冲造成的。缓冲较少的数据，更频繁地检查点或同步，或只是生成较少的数据的写入者占较小的写入(低于 64 KB)。

至于记录追加，cluster Y 比 cluster X 看到的大记录追加百分比要高得多，因为我们使用 cluster Y 的生产系统对 GFS 进行了更积极的调优。

表 5 显示了各种大小的操作中传输的总数据量。对于所有类型的操作，较大的操作(超过 256 KB)通常占传输的大部分字节。小读(64 KB 以下)确实传输了一小部分但很重要的读数据，因为随机寻道工作负载。

6.3.3 追加与写入

记录追加被大量使用，特别是在我们的生产系统中。对于集群 X，按传输的字节数计算，写入与记录追加的比率是 108:1，按操作次数计算是 8:1。对于生产系统使用的集群 Y，比率分别是 3.7:1 和 2.5:1。此外，这些比率表明，对于两个簇，记录的追加往往比写入大。然而，对于 cluster X 来说，在测量期间记录追加的总体使用是相当低的，因此结果可能会被一两个选择了特定缓冲区大小的应用程序所扭曲。

正如预期的那样，我们的数据突变工作负载主要是追加而不是覆盖。我们测量了主副本上重写的数据量。美联社-

1B集群行动	读	写	添加记录
..1 k 1 k . .8 k 8 k . .64 k 64 k . .128 k 128 k . .256 k 256 k . .512 k 512 k . .1 米 1 m . . inf	X y < .1 < .1 13.8 3.9 11.4 9.3 0.3 - 0.7 0.8 - 0.6	X y < .1 < .1 < .1 < .1 2.4 5.9 0.3 0.3 16.5 0.2 3.4 7.7 74.1 58.0	X y < .1 < .1 < .1 0.1 2.3 0.3 22.7 1.2 < .1 5.8 < .1 38.4 1 46.8 53.9 7.4

表 5:按操作大小划分的传输字节数(%)。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。如果读操作试图读取超出文件末尾的数据，则两者可能不同，这在设计上在我们的工作负载中并不罕见。

集群	X Y
开放	26.1 16.3
删除	0.7 1.5
FindLocation	64.3 65.8
FindLeaseHolder	7.8 13.4

表 6:主请求按类型细分(%)

近似地反映了客户端故意重写先前写入的数据而不是追加新数据的情况。对于集群 X，覆盖占变异字节数的 0.0001%以下，占变异操作数的 0.0003%以下。对于簇 Y，这两个比例都是 0.05%。虽然这是一分钟，但仍然高于我们的预期。事实证明，这些覆盖大部分来自于由于错误或超时而导致的客户端重试。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 显示了对主服务器的请求类型的分类。大多数请求为读取请求块位置(findlocation)，为数据突变请求租借持有者信息(FindLease- Locker)。

集群 X 和 Y 看到的删除请求数量明显不同，因为集群 Y 存储的生产数据集会定期重新生成，并用更新的版本替换。这其中的一些差异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被隐式删除，因为它被打开进行从头开始写(Unix 开放术语中的模式“w”)。

FindMatchingFiles 是一个模式匹配请求，支持“ls”和类似的文件系统操作。与对 master 的其他请求不同，它可能会处理很大一部分命名空间，因此可能会很昂贵。集群 Y 看到它的频率要高得多，因为自动数据处理任务倾向于检查文件系统的部分以了解全局应用程序状态。相比之下，cluster X 的应用程序则处于更明确的用户控制之下，通常会提前知道所有需要的文件的名称。

7.经历

在构建和部署 GFS 的过程中，我们经历了各种各样的问题，一些是操作问题，一些是技术问题。

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，使用发展到包括研究和开发任务。一开始，它几乎不支持权限和配额之类的东西，但现在包括了这些基本形式。虽然生产系统有良好的纪律和控制，但用户有时却不是这样。需要更多的基础设施来防止用户之间相互干扰。

我们最大的一些问题是磁盘和 Linux 相关的。我们的许多磁盘向 Linux 驱动程序声称它们支持一系列的 IDE 协议版本，但实际上只对最新的版本有可靠的响应。由于协议版本非常相似，这些驱动器大多工作，但偶尔不匹配会导致驱动器和内核对驱动器的状态产生分歧。这将由于内核中的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。它的开销与文件的大小成正比，而不是与修改部分的大小成正比。对于我们庞大的操作日志来说，这是一个问题，特别是在我们实现检查点之前。我们通过使用同步写入来解决这个问题，并最终迁移到 Linux 2.4。

Linux 的另一个问题是单一的读写锁，地址空间中的任何线程在从磁盘页入(读写锁)或在 mmap()调用中修改地址空间(写锁)时都必须持有该锁。我们在轻负载下看到系统中的瞬时超时，并努力寻找资源瓶颈或偶发的硬件故障。最终，我们发现这个单一的锁阻塞了主网络线程将新数据映射到内存中，而磁盘线程则在之前映射的数据中进行分页。由于我们主要受限于网络接口而不是内存拷贝带宽，我们通过用 pread()替换 mmap()来解决这个问题，代价是额外的拷贝。

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，并将变化分享给开源社区。

8.相关工作

像其他大型分布式文件系统，如 AFS[5]，GFS 提供了一个独立于位置的命名空间，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 以一种更类似于 xFS[1]和 Swift[3]的方式在存储服务器上传播文件的数据，以提供总体性能和增加的容错性。

由于磁盘相对便宜，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]、Intermezzo[6]等系统不同，GFS 在文件系统接口之下不提供任何缓存。我们的目标工作负载在单个应用程序运行中几乎没有重用性，因为它们要么通过一个大型数据集进行流处理，要么在其中随机查找，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]删除了集中式服务器并依赖分布式算法进行一致性和管理。我们选择集中式的方法是为了简化设计，增加其可靠性，并获得灵活性。特别是，中心化的主节点使得实现复杂的块放置和复制策略变得更加容易，因为主节点已经拥有大部分相关信息并控制其如何变化。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。通过追加到预写日志来持久化主状态的更新。因此，我们可以采用类似 Harp[7]中的主拷贝方案，以提供比我们当前方案更强的一致性保证的高可用性。

我们正在解决一个类似于 Lustre[8]的问题，在向大量客户端交付总体性能方面。然而，我们已经通过专注于我们应用程序的需求而不是构建一个兼容 posix 的文件系统来显著简化这个问题。此外，GFS 假定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 架构是基于网络连接的磁盘驱动器，但 GFS 使用商品机作为 chunkserver，正如在 NASD 原型中所做的那样。与 NASD 的工作不同，我们的 chunkserver 使用惰性分配的固定大小的块，而不是可变长度的对象。此外，GFS 实现了生产环境中所需的再平衡、复制和恢复等功能。

不像明尼苏达的 GFS 和 NASD，我们不寻求改变存储设备的模型。我们专注于用现有的商品组件解决复杂分布式系统的日常数据处理需求。

原子记录追加所启用的生产者-消费者队列解决了与 River[2]中的分布式队列类似的问题。River 使用基于内存的分布在不同机器上的队列和谨慎的数据流控制，而 GFS 使用一个可以被许多生产者并发追加的持久化文件。River 模型支持 m-to-n 的分布式队列，但缺乏持久存储所附带的容错性，而 GFS 只高效支持 m-to-1 的队列。多个消费者可以读取相同的文件，但他们必须协调来分区传入的负载。

9.结论

谷歌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质量。虽然一些设计决策是针对我们独特的设置的，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的文件系统假设。我们的观察在设计空间中导致了完全不同的观点。我们将组件故障视为常态而非异常，针对主要附加(可能是并发)然后读取(通常是顺序读取)的巨大文件进行优化，同时扩展和放松标准文件系统接口以改进整体系统。

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。块复制使我们能够容忍 chunkserver

失败。这些故障的频率激发了一种新颖的在线修复机制，它定期、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，考虑到系统中的磁盘数量，这变得太常见了。

我们的设计为许多执行各种任务的并发读取器和写入器提供了高总体吞吐量。我们通过分离文件系统控制(通过 master)和数据传输(直接在 chunkserver 和客户端之间传递)来实现这一点。Master 参与公共操作的程度通过一个大的块大小和块租约(chunk lease)最小化，这将权限委托给数据突变中的主副本。这使得一个简单的、集中式的 master 成为可能，而不会成为瓶颈。我们相信，我们的网络堆栈的改进将提升当前单个客户端所看到的写吞吐量限制。

GFS 已经成功地满足了我们的存储需求，并被广泛用于谷歌内，作为研发以及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不断创新和攻击问题。

致谢

我们希望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、JeffDean 和 David desJardins 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 从事存储配额的工作。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。

参考文献

[1] Thomas Anderson、Michael Dahlin、Jeanna Neefe、David Patterson、Drew Roselli、Randolph Wang。无服务器网络文件系统。第 15 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, Kathy Yelick。Cluster I/O with River: Making the fast case common。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，乔治亚州亚特兰大，1999 年 5 月。

Luis-Felipe Cabrera 和 Darrell D. E. Long。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, Jim Zelenka。一个高性价比、高带宽的存储体系结构。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，加州圣何塞，1998 年 10 月。

[5] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert

西德博瑟姆、迈克尔·韦斯特。规模和

分布式文件系统的性能。计算机系统学报，6(1):51-81,1988 年 2 月。

[6]插曲。http://www.inter-mezzo.org, 2003。

[7]芭芭拉·利斯科夫、桑杰·格玛沃特、罗伯特·格鲁伯、保罗·约翰逊、柳巴·什里拉、迈克尔·威廉姆斯。在 Harp 文件系统中复制。第 13 届操作系统原理研讨会，第 226-238 页，太平洋格罗夫，CA，1991 年 10 月。

[8]光泽。http://www.lustreorg, 2003。

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

[10] Frank Schmuck 和 Roger Haskin。GPFS:用于大型计算集群的共享磁盘文件系统。2002 年 1 月，加州蒙特雷，第一次 USENIX 文件和存储技术会议论文集，第 231-244 页。

[11] Steven R. Soltis, Thomas M. Ruwart 和 Matthew T. O 'Keefe。全球文件系统。1996 年 9 月，在马里兰州大学公园举行的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上。

[12] Chandramohan A. Thekkath, Timothy Mann, Edward K. Lee。Frangipani: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，页 224-237，法国圣马洛，1997 年 10 月。