

【译文】The Google File System 经典的分布式文件存储系统

作为现代分布式软件系统入门经典论文，*The Google File System (2003)* 有着里程碑式的意义，并由此诞生了Hadoop生态中HDFS的开源实现。笔者出于学习的目的，将此文章翻译成中文，其中部分地方会加上自己的理解和注释。前五章参考 *Praying* 的翻译。

1. 介绍 (INTRODUCTION)

我们已经设计和实现了Google File System (GFS) 来满足Google快速增长的数据处理需求。GFS和以前的分布式文件系统有着许多相同的目标，比如**性能 (performance)**，**可扩展性 (scalability)**，**可靠性 (reliability)** 和**可用性 (availability)**。但是，它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主要观察结果来驱动的，这些观察明显不同于早期的文件系统设计的假设。我们已经重新评估传统的选择，并探索在设计空间上完全不同的点。

1. 首先，组件失效是正常现象而非异常现象。文件系统由数百或数千台存储机器组成，这些机器由廉价的商业部件组合而来，并且会被大量的客户机访问。无论是这些部件的质量还是数量都几乎可以保证，其中的一些组件在给定某个时刻会出现功能失效并且一些组件无法从当前的失效中恢复。
我们已经见到由应用程序bug、操作系统bug、人为失误和磁盘、内存、连接器、网络的失效以及电力供应引起的问题。因此，**持续监测 (constant monitoring)**、**错误检测 (error detection)**、**容错 (fault tolerance)** 以及**自动恢复 (automatic recovery)** 必须集成到这个系统当中。
2. 第二，以传统标准来看，文件是巨大的。大小为数G的文件是很普遍的。每个文件通常包含很多应用对象，比如web document。当我们经常处理由数十亿个对象组成的快速增长的许多TB大小的数据集时，即使文件系统能够支持，管理数十亿个约KB大小的文件依然很笨拙的。因此，设计假设和参数，如I/O操作和块大小，不得不被重新审视。
3. 第三，大多数文件被修改的方式是**追加新数据而不是重写已存在数据**。文件内的随机写操作在实际上是不存在的。一旦写入，这些文件就仅会被读取，而且通常只按顺序读取。许多数据都具备这些特点。有些可能构成数据分析程序扫描的**大型存储仓库**。有些可能是运行的应用程序不断生成的数据流。有些可能是归档数据。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，无论是同时还是稍后 (*Praying*注:指处理中间结果)。由于这种对大型文件的访问模式，追加成为性能优化和原子性保证的重点，而在客户机中缓存数据块则失去了吸引力。
4. 第四，应用程序和文件系统API的协同设计通过增加我们的灵活性而使整个系统受益。例如，我们已经减弱GFS的一致性模型 (*Hades*注: 弱一致性模型，是分布式系统中一直强调的设计理念) 来大大简化这个文件系统而没有给应用程序带来繁重的负担。我们还引入了一个原子追加操作能够使多个客户端之间**并发地对同一个文件追加而无需额外的同步机制**。这些内容会在本文后面更详细地讨论。

多个GFS集群可以针对不同的目的同时进行部署。最大的一个集群拥有超过1000个存储节点，超过300TB的磁盘存储，并且被不同机器上的数百个客户端连续地大量访问。

2. 设计概述 (DESIGN OVERVIEW)

2.1 假设 (Assumptions)

在为我们的需求设计一个文件系统时，我们一直遵循着挑战与机遇并存的假设。我们之前提到了一些关键的观察结果，现在我们更详细地来表述我们的假设。

- 系统构建于许多廉价的商业组件，这些组件**经常失效**。系统必须持续监测自身并且检测 (detect)，容错 (tolerate)，以及从日常的组件失效中立即恢复 (recover)。
- 系统存储数量适中的**大文件**。我们预计大概有几百万个文件，每个文件通常大小为100M或者更大。大小为数G的文件很常见并且应该被高效地管理。小文件必须被支持，但是我们**不需要对它们进行优化**。
- 工作负载主要由两种类型的读取（操作）组成：**大规模流读取和小规模随机读取**。在大规模流读取操作中，单次操作通常读取数百KB大小，更常见的是1M或者更多。来自同一客户端的连续操作经常读取某一文件的一个连续区域。小规模的随机读取通常在任意偏移位置读取若干KB大小。性能敏感型的应用程序通常对小规模读取进行批处理 (batch) 和排序 (sort)，从而能够稳定地遍历文件而不是来回切换。
- 工作负载还包含许多对文件进行追加数据的大规模、顺序的写操作。通常操作的大小和读操作类似。一旦写入，文件就**很少会被再修改**。在文件的任意位置进行小规模地写操作是被支持的，但是不必是高效的。
- 系统必须针对多个客户端并发追加到相同文件高效地实现具有良好定义的语义。我们的文件通常用作生产者-消费者队列或用于多路合并 (many-way merging)。每台机器运行一个消费者（程序），数百个消费者程序将会并发地追加到同一个文件。具有**最小同步开销的原子性**是至关重要的。文件可能是稍后读取，或者一个消费者可能同时读取该文件 (*Praying*^{注：同时指在文件被写入的同时进行读取})。
- 高持续带宽比低延迟重要得多。我们的大多数目标应用程序都重视以高速率处理大量数据，而很少有应用程序对单个的读或写有严格的响应时间要求。

(*Hades*^{注：综上所示，GFS更关注大文件管理，顺序读写，适当的并发处理，以及高吞吐量})

2.2 接口 (Interface)

尽管GFS没有实现一套像POSIX这样的标准API，但是它提供了一个熟悉的文件系统接口。文件在目录中按层次结构组织，并通过路径名标识。我们支持常见的操作来创建 (create)、删除 (delete)、打开 (open)、关闭 (close)、读取 (read) 和写入 (write) 文件。

不仅如此，GFS还有快照 (snapshot) 和记录追加 (record append) 操作。快照以**低开销**创建一个文件或者目录树的拷贝。记录追加允许多个客户端并发地对同一个文件进行追加数据，同时保证每个独立客户端的追加操作的原子性。这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对该队列进行追加（append）操作而无需额外的锁操作（locking）。我们发现这些类型的文件在构建大型分布式应用程序时非常宝贵。快照和记录追加分别在Section 3.4 和 3.3 会进行更深入的讨论。

2.3 架构 (Architecture)

一个GFS集群由一个 *master* 和多个 *chunkserver* 组成并且被多个客户端 (*client*) 访问，如下图Figure 1所示。这些中的每一个通常都是一个运行着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 *chunkserver* 和客户端 (*client*) 是很容易的，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更低的可靠性是可以接受的。

文件被分成固定大小的块 (*chunk*)。每个块由一个不可变的全局唯一的64 bit的块句柄 (*chunk handle*) 来标识，这个块句柄是由 *master* 在块创建时赋予的。*chunkserver* 把块作为Linux文件存储在本地磁盘并且通过一个块句柄和字节区间 (*byte range*) 来读写指定的块数据。出于可靠性 (reliability)，每个块在多个 *chunkserver* 上复制。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，默认情况下，我们存储三份副本。

master 维护所有的文件系统元数据。这些元数据包括命名空间 (namespace)、访问控制信息 (access control information)、文件到块的映射，以及块的当前位置。它还控制整个系统的活动，比如块租约管理 (chunk lease management)、孤儿块的垃圾回收，以及 *chunkserver* 之间的块迁移。*master* 以心跳 (HeartBeat) 消息的方式周期性地和每个 *chunkserver* 进行通信，下达指令并收集其状态信息。

被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 *master* 和 *chunkserver* 通信来代表应用程序进行读写数据操作。客户端和 *master* 交互来进行元数据操作，但是所有承载数据的通信都是直接和 *chunkserver* 进行。我们不提供POSIX API，因此不需要对Linux的vnode层进行挂钩连接 (hook into)。

客户端和 *chunkserver* 都不缓存文件数据。客户端的缓存几乎提供不了多少益处，因为大多数应用程序都要处理巨大的文件或者工作集太大而无法缓存。没有它们 (*Praying* 注：指缓存文件数据) 可以消除缓存一致性问题，从而简化客户端和整个系统 (尽管如此，客户端会缓存元数据)。*chunkserver* 不需要缓存文件数据是因为块作为本地文件被存储，所以Linux的buffer cache 已经将经常访问的数据保存在内存中。

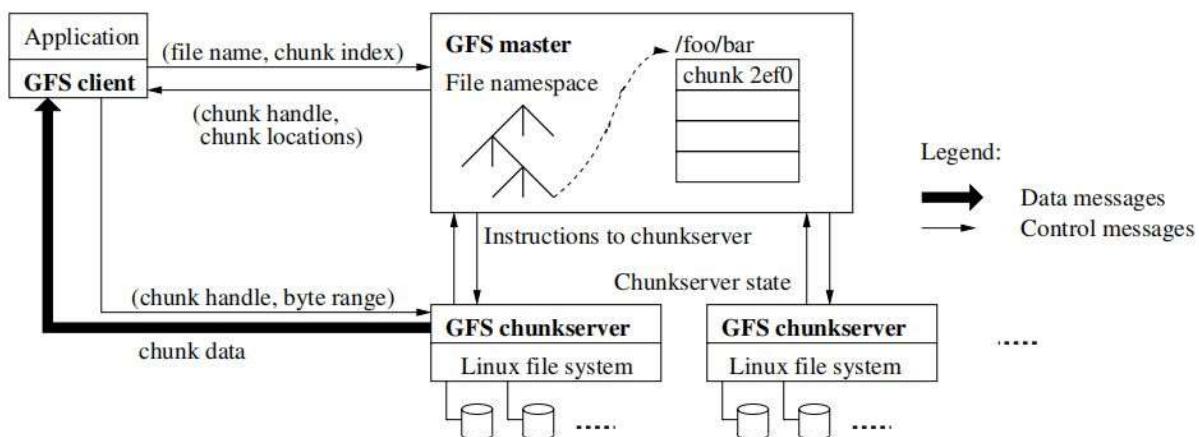


Figure 1: GFS Architecture

2.4 单个主服务器 (Single Master)

使用单个master极大地简化了我们的设计，并使master能够使用全局知识作出复杂的块放置和复制决策。尽管如此，我们必须最小化它对读写的参与，从而不至于成为瓶颈。客户端从不通过master读写文件数据。取而代之的是，客户端会问master它应该联系哪个chunkserver。客户端把这个信息缓存一段有限的时间，然后直接和chunkserver交互以进行后续的操作。

让我们通过Figure 1来解释一次简单的读取操作。首先，使用固定的块大小，客户端把文件名和应用程序指定的字节偏移量转换为一个文件内的块索引。然后，它向master发送一个包含有文件名和块索引的请求。master回复对应的**块句柄和副本的位置**。客户端使用文件名和块索引作为key缓存这条信息。

客户端接着发送一个请求到其中一个副本，**最可能是最近的那个**。这个请求指定了块句柄和一个块内的字节区间。之后对相同块的读取就不再需要客户端和master的交互了，直到缓存信息过期或者文件被重新打开。事实上，客户端通常会在同一个请求中请求多个块，并且master也可以立即包含紧随其后的块的信息。这些额外的信息几乎不需要额外开销就可以避开未来的几次客户端到master的交互。

2.5 块大小 (Chunk Size)

块大小是关键设计参数之一。我们选择了64MB，这比典型的文件系统块大小大得多。每个块副本作为普通Linux文件存储在一个chunkserver上，且只在需要的时候进行扩展。惰性空间分配 (lazy space allocation) 避免了因内部碎片而浪费空间，这可能是对如此大的块大小的最大争议。

一个较大的块大小提供了几个重要的优势。首先，**它减少了客户端和master交互的需要**，因为读写相同的块只需要对master发起一个询问块位置信息的初始请求。这个减少对于我们工作负载尤其重要，因为应用程序通常都是顺序地读写大文件。即使对于小规模的随机读取，客户端也可以轻松地将数TB的工作集的所有块位置信息缓存。其次，由于在大数据块上，客户端更有可能在给定数据块上执行许多操作，因此可以通过和chunkserver保持**更长时间持续的TCP连接**来减少网络开销。第三，**它减少了存储在master的元数据的大小**。这使得我们能够把元数据保存在内存中，从而带来我们将会在Section 2.6.1中将会讨论的其他的优势。

另一方面，一个大的块大小，即使是惰性空间分配，也有它的缺点。一个小文件由少量块组成，可能就一个块。如果很多客户端正在访问同一个文件，存储这些块的chunkserver可能会成为热点 (hot spot)。在实践中，热点 (hot spot) 没有成为一个主要问题，因为我们的应用程序主要是顺序读取较大的具有多个块的文件。

尽管如此，当GFS最初用于一个批处理队列系统的时候，热点确实出现了：一个可执行程序作为一个单块文件被写到GFS里，然后同时在数百台机器上发起请求。存储这个可执行文件的几台chunkserver因为数百个同时的请求而过载。通过以更高的复制因子来存储这样的可执行文件，并使批处理队列系统错开应用启动时间，我们修复了这个问题。一个潜在的长期解决方案是允许客户端在这种情况下从其他客户端读取数据。

2.6 元数据 (Metadata)

master存储了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, 以及每个块的副本的位置**。所有的元数据保存在master的内存中。前两种类型(命名空间和文件到块的映射)也会通过将变更记录到存储在master本地磁盘的**操作日志 (operation log)**而持久化保存, 并且被复制到远端机器。使用日志能使我们简单、可靠地更新master的状态, 并且如果发生master崩溃(crash)也不会出现不一致的风险。master不会对块位置信息进行持久化存储。取而代之, 它会在master启动和chunkserver加入集群时询问每个chunkserver关于该chunkserver上的块(信息)。

2.6.1 内存中的数据结构 (In-Memory Data Structures)

由于元数据存储在内存中, master上的操作是很快的。此外, master可以简单高效地在后台周期性扫描它的记录状态。这个周期性扫描用于实现**块的垃圾回收**, 在chunkserver失效时进行**重新复制 (re-replication)**, 以及**块迁移**, 块迁移是为了在chunkserver之间均衡负载和磁盘空间使用。Section 4.3 和 4.4 将会更深入地讨论这些内容。

这种仅使用内存的方式的一个潜在问题是, 块的数量和整个系统的容量受限于master的内存大小。这在实践中不是一个很严重的限制。master为每个64MB大小的块维护了不到64字节的元数据。大多数块是填满的, 因为大多数文件包含许多块, 只有最后的块可能是部分填充的。类似的, 文件命名空间数据通常每个文件需要不到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, 相对于在内存中存储元数据所获得的简单性、可靠性、性能和灵活性来说, 向master添加额外内存的成本是很小的开销。

2.6.2 块位置 (Chunk Locations)

关于哪个chunkserver拥有一个给定块的副本, master对此**没有保存持久化的记录**。它只是简单地在启动时对chunkserver轮询那些信息。master可以在之后保持最新状态, 因为它通过周期性的心跳(HeartBeat)消息控制所有的块放置并监控chunkserver的状态。

我们最初尝试把块位置信息在master上持久化保存, 但是我们认为在启动时以及之后周期性地从chunkserver请求数据会更简单。这样消除了在当chunkserver加入和离开集群、更改名称、失效(fail)、重启诸如此类的情况下, 保持master和chunkserver同步的问题。在一个拥有数百台服务器的集群中, 这些事件会经常发生。

理解这个设计决策的另一种方式是认识到chunkserver对它自己的磁盘上有什么块或没有什么块有最终决定权。试图在**master上维护这个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, 磁盘会坏掉并且无法启用), 或者操作员可能会对chunkserver重命名。

2.6.3 操作日志 (Operation Log)

操作日志包含关键元数据变化的历史记录。它是GFS的核心。**它不仅是元数据唯一的持久化记录, 而且它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(见Section 4.5), 都是唯一的, 且永久性地被它们创建时的逻辑时间所标识。

因为操作日志非常关键, 所以我们必须对其进行可靠的存储, 并且在元数据的更改被持久化之前, 不能使更改对客户端可见。否则, 即使文件块自身被保存下来, 我们也会丢失整个文件系统或者最近的客户端操作。因此, 我们在多个远端机器上对其进行复制, 并且仅在对应的日志记录在本地和远端的磁盘都被刷入(flush)之后**才会对客户端操作进行响应**。在刷入之前, master对若干条日志记录一起进行批处理(batch), 从而减少刷入

(flush) 和复制对整个系统吞吐量的影响。 (Hades注：对Operation Log进行冗余备份，提高其可用性。同时用户的操作可能产生多条log，此时master会以一次操作会话为单位将多条log一起刷入磁盘。)

master通过重复执行操作日志来恢复它的文件系统状态。为了最小化启动时间，我们必须使日志较小。每当日志增长超过一个特定大小时，master就对它的状态生成一个核对点（checkpoint），以便于它可以通过从磁盘载入最新的核对点（checkpoint）并且仅重新执行核对点之后的有限数量的日志记录来进行恢复。核对点是一种紧凑的类似b树的形式，可以直接映射到内存中并用于命名空间查找，而无需进行额外的解析。这进一步地加快了恢复和改善了可用性。

因为生成一个核对点会花费一些时间，所以master的内部状态以这样一种方式构造，即在不推迟即将到来的变更（mutation）的情况下创建一个新的核对点。master切换到一个新的日志文件并且在一个单独的线程中创建新的核对点。新的核对点包括切换（日志）之前的所有变更（mutation）。对于有几百万个文件的集群，可以在一分钟左右创建核对点。创建完成后，核对点会被写入到本地和远端的磁盘。

恢复只需要最新的完整核对点和后续的日志文件。较老的核对点和日志文件可以被自由删除，但是我们保留了一些以预防灾难。核对点生成期间的失效（failure）不会影响正确性，因为用于恢复的代码会检测并跳过不完整的核对点。

2.7 一致性模型 (Consistency Model)

GFS有一个宽松的一致性模型，它可以很好地支持高度分布式的应用程序，但是实现起来相对简单和高效。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如何维持这些保证，但是将细节放在论文的其他部分。

2.7.1 GFS的保证 (Guarantees by GFS)

文件命名空间变更（比如，文件创建）是原子性（atomic）的。它们仅由master处理：命名空间锁定保证了原子性和正确性（Section 4.1）；master的操作日志定义了这些操作的全局总顺序（Section 2.6.3）。

数据变更后的文件区域（file region）的状态取决于变更的类型、变更成功还是失败以及是否存在并发变更。Table 1对此进行了总结。如果所有的客户端不论从文件的哪个副本读取，都能读到相同的数据，那么这个文件区域就是一致（consistent）的。如果一个区域（region）在文件变更后是一致的，并且客户端将会看到变更写入的全部内容，那么这个区域就被**定义**（defined）了。当变更成功并且没有受到并发写者（writer）的干扰时，受影响的区域就被定义了（也意味着是一致的）：所有客户端将总能看到变更写入的内容。并发的成功变更产生了未定义（undefined）但一致的区域：所有客户端看到相同的数据，但是它可能不会体现（reflect）任意一个变更所写入的内容。典型地，它由多个变更的混合片段组成。一个失败的变更使得区域不一致（因此也是未定义的）：不同的客户端可能在不同的时间点看到不同的数据。下面我们将介绍我们的应用程序是如何区分**定义区域**（defined region）和**未定义区域**（undefined region）。应用程序不需要更详细地区分不同类型的未定义区域。

	Write	Record Append
Serial success	<i>defined</i>	<i>defined interspersed with inconsistent</i>
Concurrent successes	<i>consistent but undefined</i>	
Failure		<i>inconsistent</i>

Table 1: File Region State After Mutation

数据变更可能是**写操作 (writes)** 或者是**记录追加 (record appends)**。写操作使得数据在一个由应用程序指定的文件偏移位置被写入。记录追加使得数据（即“记录”）在即使存在并发变更的情况下，在GFS选择的偏移位置（Section 3.3），至少被原子性地追加一次。（相对的，一个“常规 (regular)”追加操作仅仅是在客户端认为的文件的当前结尾处进行写入）。偏移量返回给客户端，并标志着包含记录的定义区域的开始。此外，GFS可能会在其间填充空白或者复制记录。它们占据的区域被认为是不一致的，而且通常与用户数据量相比微不足道。

在一系列成功的变更后，变更后的文件区域被保证是定义（defined）的且包含最后一次变更写入的数据。GFS通过以下方式实现这个保证：(a) 在所有副本上以相同的顺序对块进行变更（Section 3.1），(b) 使用**块版本号**来检测任何因为在chunkserver宕机时错过变更而过期的副本（Section 4.5）。过期的副本永远不会参与变更，也不会作为结果返回给正在向master询问块位置的客户端。它们会被当作垃圾被尽早地回收。

由于客户端缓存了块信息，所以它们可能会在信息刷新之前从一个过期的块进行读取。这个窗口受限于缓存条目的超时和文件的下一次打开，文件重新打开会清除缓存中关于这个文件的所有块信息。此外，因为大多数文件是仅追加（append-only）的，一个过期的副本通常返回一个过早结束的块而不是过期的数据。当一个读者（reader）重新连接（retry）并联系（contact）master时，它会立即得到当前的块位置。

在一次成功变更后很长一段时间内，组件失效仍会理所当然地损坏（corrupt）或销毁（destroy）数据。GFS通过master和所有chunkservers之间的定期握手来识别失效的chunkserver，并通过校验和（checksum）检测数据损坏（Section 5.2）。一旦出现问题，数据会从有效的副本中被尽可能快地恢复（Section 4.3）。只有在当GFS未作出反应之前，通常是几分钟内，所有的副本都丢失了，这个块才会被不可逆转地丢弃。即使是这种情况，它只是变成了不可用（unavailable），而不是损坏：应用程序收到清晰的错误而不是损坏的数据。

2.7.2 对应用程序的影响 (Implications for Applications)

GFS应用可以用一些简单的技术来适应宽松的一致性模型，这些技术也是其他目标所需要的：依靠追加（append）而不是覆盖（overwrite），核对点（checkpoint），以及写入自验证（self-validating）、自识别（self-identifying）的记录。

实际上，我们所有的应用程序都通过追加而非重写的方式变更文件。在一个典型用法中，一个写者（writer）从头到尾生成一个文件。它在写入全部数据后原子性地将文件重命名为一个永久的名字，或者周期性对写入的数据生成核对点。核对点还可能包含应用程序级别的校验和。读者（reader）只验证和处理直到最后一个核对点的文件区域，该区域已知是处于定义状态。无论一致性和并发性问题如何，这种方法都对我们很有用。与随机写入相比，追加远比随机写入更高效，能更好地适应应用故障。生成核对点允许写者（writer）增量重启并使读者无法成功处理已经写入的文件数据，这些文件数据从应用程序的视角来看仍是不完整的。

在另一个典型用法中，许多写者（writer）并发地对一个文件追加（数据），该文件作为一个合并后的结果或者是生产者-消费者队列。记录追加的至少追加一次（append-at-least-once）语义保留每个写者的输出。读者对偶尔的填充和重复的处理如下。写者准备的每条记录包含额外的信息，比如校验和，因此它的有效性可以得到验证。读者可以**使用校验和来识别和丢弃额外的填充和记录片段**。如果它无法容忍偶尔的重复（比如，如果它们（指重复）会触发非幂等性（non-idempotent）操作），读者可以使用记录中的**唯一标识符**将它们（重复记录）过滤出去，这在对相应的应用实体，如web document，进行命名时，经常会需要用到。这些针对记录I/O的功能（不包括消除重复）都在我们通过应用程序共享的库代码里并且可以应用于Google内部的其他文件接口的实现。接着，相同顺序的记录，加上罕见的重复，总能将记录交付给记录读者（record reader）。

3. 系统交互 (SYSTEM INTERACTIONS)

我们设计这个系统以求在所有的操作中最小化 master 的涉入（involvement）。在此背景下，现在我们来描述客户端、master 以及 chunkserver 是如何交互从而实现数据变更、原子性记录追加和快照（snapshot）。

3.1 租约和变更顺序 (Lease and Mutation Order)

变更是一种改变块的内容或者元数据的操作，比如写操作和追加操作。每个变更会在块的所有副本上被执行。我们使用租约在副本间维护一致的变更顺序。master 把一个块租约赋予其中一个副本，我们称该副本为 *primary*，*primary* 为块上的所有变更选择一个顺序。所有的副本在应用变更的时候都遵循这个顺序。因此，全局的变更顺序首先由 master 选择的租约准予顺序（lease grant order）定义，在租约内通过一个 *primary* 赋予的序列号（serial number）定义。

租约机制被设计用于最小化 master 上的管理开销。租约有一个初始的 60 秒超时。但是，只要块被变更，*primary* 就可以请求且通常会受到来自 master 的无限期扩延。这些扩延请求和准予（grant）依附于 master 和 chunkserver 之间周期性交换的**心跳**

(HeartBeat) 消息。master 有时可能尝试在租约过期之前将租约撤销（比如，当 master 想禁用对一个正在进行重命名的文件的变更）。即使 master 和 *primary* 失去通信，它也可以在旧的租约到期之后安全地把新租约赋予另一个副本。

在 Figure 2 中，我们按照写的控制流程，通过这些数字步骤来说明这个过程。

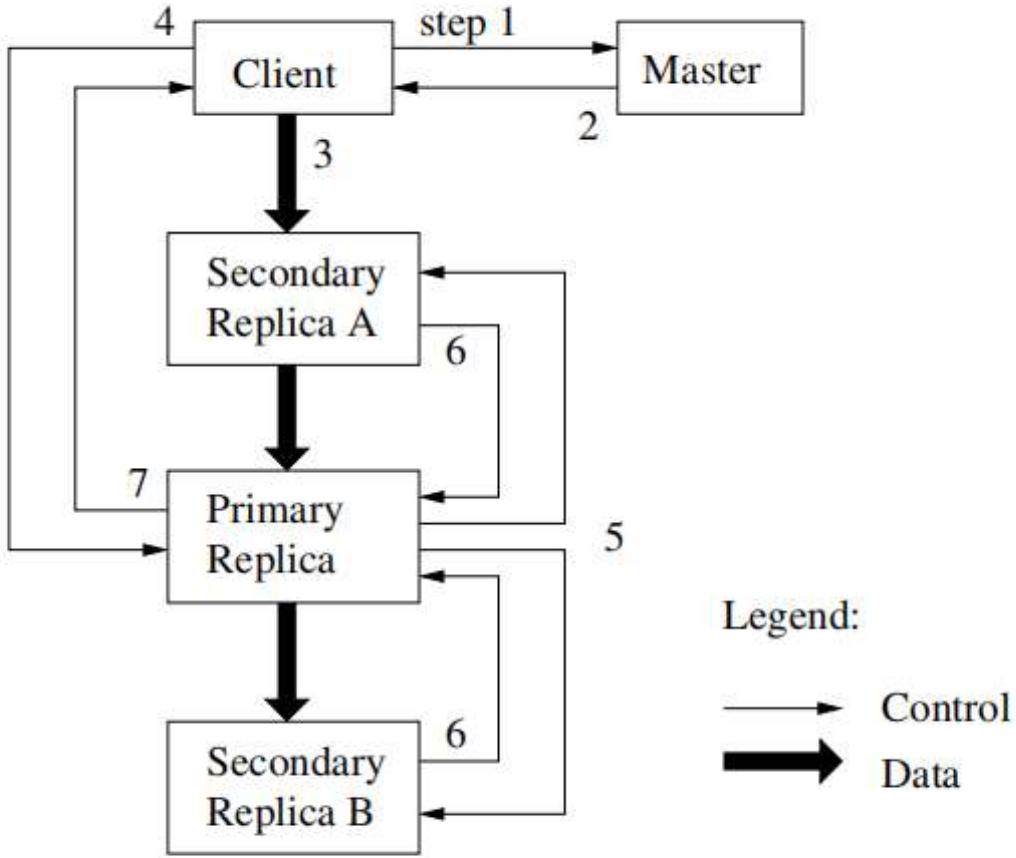


Figure 2: Write Control and Data Flow

1. 客户端询问 master 哪个 chunkserver 持有块的当前租约以及其它块的副本位置。如果没有 chunkserver 持有租约，master 会对它选择的一个副本赋予租约（图中没有展示）。
2. master 回复 primary 的标识以及其它副本 (*secondary*) 的位置。客户端会为将来的变更缓存这份数据。仅当 primary 不可到达 (unreachable) 或者 primary 回复其不再持有租约时，客户端才需要再次联系 master。
3. 客户端把数据推送到所有的副本。客户端可以以任意顺序推送数据。每个 chunkserver 会把数据存储在内部的 LRU buffer 缓存，直到数据被使用或超期 (age out)。通过从控制流中将数据流解耦，我们可以根据网络拓扑来调度开销较大的数据流，而不必关心哪个 chunkserver 是 primary，从而改善性能。Section 3.2 对此进行了更深入的讨论。
4. 一旦所有的副本都确认收到了数据，客户端就会给 primary 发送一个写请求。这个请求标识了之前被推送到所有副本的数据。primary 为其收到的所有变更分配连续的序列号，这些变更可能来自于多个客户端，序列号提供了必要的编序。primary 依据序列号顺序将变更应用到自己的本地状态。
5. primary 转发写请求到所有的从副本 (*secondary replicas*)，每个从副本以 primary 分配的相同的序列号顺序应用变更。
6. 所有的从副本对 primary 进行回复以表明它们完成了操作。
7. primary 对客户端进行回复。在任何副本上出现的任何错误都会报告给客户端。在发生错误的情况下，写操作可能已经在 primary 和从副本的某个子集上执行成功（如果它在 primary 上失败，就不会被分配一个序列号和被转发）。这个客户

端请求被当作是失败的，且被修改的区域停留在不一致状态。我们的客户端代码通过重试失败的变更来处理这样的错误。它在从写操作开头进行重试之前会在步骤 (3) 和 (7) 之间进行几次尝试。

如果应用程序的一个写操作比较大或跨越了块的边界，GFS 客户端代码会把它拆解为多个写操作。它们全都遵循上面描述的控制流程，但是可能会和其他客户端的并发操作有交叉或者被其重写。因此，共享的文件区域最终会包含来自不同客户端的片段，尽管副本都是相同的，因为所有副本上的各个操作都以相同的顺序在副本上完成。这使得文件区域处于 Section 2.7 所描述的一致但未定义的状态。

3.2 数据流 (Data Flow)

我们将数据流从控制流中分解出来从而高效地使用网络。控制流从客户端到 primary 然后再到所有的从副本，而数据流以流水线的方式在精心挑选的 chunkserver 链之间被线性地推送。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，并且最小化所有数据的推送时延。

为了充分利用每台机器的网络带宽，数据在 chunkserver 链之间被推送，而没有以其他拓扑形式分发（比如，树）。因此，每台机器的全部出站（outbound）带宽都用于尽可能快地传输数据而不是分给多个接收者（recipient）。

为了尽可能地避免网络瓶颈和高延迟链路（比如，交换机之间的链路通常都是），每台机器把数据转发到网络拓扑上尚未收到数据的“最近的（closest）”机器。假定客户端正在把数据推送到 chunkserver S1 到 S4。它会把数据推送到最近的 chunkserver，比如说是 S1。S1 把数据转发到 S2 到 S4 中最近的一个 chunkserver，比如说是 S2。类似的，S2 转发数据到 S3 或者 S4，取决于哪个离 S2 更近，以此类推。我们的网络拓扑是足够简单的，距离（distance）可以从 IP 地址中准确地估算出来。

最后，我们通过对 TCP 连接上的数据传输进行管道化（pipeline）来最小化时延。一旦一个 chunkserver 收到某些数据，它就立即开始转发。管道化对我们非常有帮助，因为我们使用的是全双工链路（full-duplex link）的交换网络。立即发送数据不会降低接收速率。在没有网络拥堵的情况下，传输 B 个字节到 R 个副本的理想耗时是 $B/T + RL$ ，其中 T 是网络吞吐量， L 是在两台机器间传输字节的时延。我们的网络链路通常是 100 Mbps (T)， L 远低于 1ms。因此，1MB 在最好情况下可以在 80ms 左右的时间内被分发完成。

3.3 原子性记录追加 (Atomic Record Appends)

GFS 提供了一项原子追加操作，称为记录追加（record append）。在传统的写操作中，客户端指定写入数据的偏移量。对相同区域的并发写入不是串行的（serializable）：该区域可能最终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 会原子性至少一次在 GFS 选择的偏移量将数据写到文件中（比如，作为一个连续的字节序列）并将偏移位置返回给客户端。 这类似于在 Unix 中对以 O_APPEND 模式打开的文件的写入，当多个写者并发地进行写入时，没有竞争条件（race condition）。（Hades 注：Unix 文件系统中以 O_APPEND 模式调用 write 函数时，不会发生并发问题，文件指针在每一次调用时都会指向文件尾。在此处 paper 并未说明 GFS 具体是如何实现原子追加操作的。）

记录追加被我们的分布式应用程序频繁使用，在这些应用中，不同机器的多个客户端并发地对同一文件进行追加。如果客户端使用传统的写入操作，它们可能需要额外的复杂且成本较高的同步机制，比如，通过一个分布式锁管理者（distributed lock manager）。在我们的工作负载中，这样（需要被并发写入）的文件经常用作多生产者/单消费者队列（multiple-producer/single-consumer queues）或者包含来自不同客户端的合并结果。（*Hades*注：*GFS只保证了追加写这个单一操作的原子性，至于客户端应用要达到怎样的并发控制效果，应该由客户端去控制，GFS做不到、无义务也不应该去干涉。*）

记录追加是一种变更，遵循 Section 3.1 的控制流程，只在 primary 上有一些额外的逻辑。客户端把数据推送到文件的最后一个块的所有副本上，然后，它把它的请求发送到 primary。primary 会检查追加记录到当前块是否会引起块超出最大大小（64 MB）。如果会超出，它就会把这个块填充至最大大小，接着告诉从副本进行相同的操作，然后回复客户端，表明这个操作应该在下一个块上进行重试。（记录追加被限制在最多为最大块大小的四分之一以保持最坏情况下的碎片在一个可接受的范围）。如果记录在最大块大小内可以符合，这也是最常见的情况，primary 追加数据到它的副本，告诉从副本（secondaries）在其准确的偏移处将数据写入，最后向客户端回复成功。

如果一个记录追加在任意副本上失败，客户端会重试操作。因此，**相同块的副本可能包含不同的数据，这些数据包含相同记录的完整重复或部分重复**。GFS 不保证所有的副本每个字节都相同。它只保证数据作为一个原子性单位，至少被写入一次。这个属性很容易从简单的观察中得出，为了使操作报告成功，**数据必须在某个块的所有副本上相同的偏移量写入**。此外，在这之后，所有的副本都至少与记录结尾一样长，因此，即使一个不同的副本成为 primary，那么未来的任何记录都会被分配更高的偏移量或一个不同的块。在我们的一致性保证方面，成功的记录追加将数据写入的区域是定义的（defined）（因此也是一致的），而中间的区域是不一致的（因此也是未定义的）。我们的应用程序可以处理不一致的区域，正如我们在 Section 2.7.2 所讨论的那样。

3.4 快照 (Snapshot)

快照操作几乎是在瞬间生成了一个文件或目录树（"源（source）"）的拷贝，同时最小化对正在进行的变更的中断。我们的用户使用它来**快速创建大数据集的分支拷贝**（以及递归地创建这些拷贝的拷贝），或者在带有改动的试验进行之前为当前的状态生成核对点，这些改动后面可以会被提交或轻松地回滚。

类似 AFS，我们使用标准的写时复制（copy-on-write）技术来实现快照。当 master 收到一个快照请求时，它首先**撤销**了要对其进行快照的文件的块上的尚未到期的租约（*Hades*注：这里翻译的有一点绕，但意思就是撤销该文件对应的Primary的租期）。这就确保了接下来对这些块的任何写操作都需要和 master 进行交互以找到租约的持有者。这就给了 master 一个先对块创建副本的机会。

在租约被撤销或者过期之后，master 把操作记录到磁盘上。接着他通过复制源文件或目录树的元数据，把这个日志记录应用到它的内存状态，新创建的快照文件和源文件都指向相同的块。

客户端在快照操作后第一次要向块 C 写入时，就会向 master 发送请求，寻找当前的租约持有者。master 注意到块 C 的引用计数超过一个。它推迟对客户端的回复，而是选择了一个新的块句柄 C'。接着它要求每个拥有块 C 副本的 chunkserver 去创建一个名为 C' 的新的块。通过在与原来的块相同的 chunkserver 上创建新的块，我们确保了数据可以在本地拷贝（copy locally），而没有经过网络（我们的磁盘大约比我们的 100 Mb 的以太网链路

快三倍）。从这一点来看，请求处理对任意的块都是相同的：master 将新块 C' 的租约赋予其中一个副本然后对客户端回复，客户端可以在不知道这个块是刚刚从一个已存在的块重新创建的情况下，正常地对块进行写入。（*Hades*注：即懒加载）

4. Master 操作 (MASTER OPERATION)

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：它作出块放置决策、创建新的块、复制以及协调系统范围内的各种活动来保持块被完整地复制、平衡 chunkserver 之间的负载，回收未使用的存储。我们现在来讨论这些主题。

4.1 命名空间管理和块锁定 (Namespace Management and Locking)

许多 master 操作耗时较长：例如，快照操作不得不撤销其涉及的所有块的 chunkserver 租约。我们不想在这些操作进行的时候推迟其他的 master 操作。因此，我们允许多个操作处于活动状态并且在命名空间区域使用锁来保证合适的串行 (serialization)。

不同于许多传统的文件系统，GFS 没有能够列出目录下所有文件的针对每个目录的数据结构。它既不支持对文件的别名，也不支持对目录的别名（例如，Unix 术语中的硬链接或软链接）。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被高效地表示。命名空间树里的每个节点（绝对文件名或绝对目录名）有一个相关联的读写锁。

每个 master 上的操作在其运行之前要先获取一组锁。典型地，如果它涉及到 **/d1/d2/.../dn/leaf**，它将会获取目录名为 **/d1**，**/d1/d2**，...，**/d1/d2/.../dn** 的读锁，并且获取完整路径名 **/d1/d2/.../dn** 上的一个读锁或写锁。注意，**leaf** 可能是一个文件或者目录，这取决于具体的操作。

我们现在来阐述，当 **/home/user** 正在进行快照操作保存到 **/save/user** 时，这个锁机制是如何阻止文件 **/home/user/foo** 被创建的。快照操作获取 **/home** 和 **/save** 上的读锁，以及 **/home/user** 和 **/save/user** 的写锁。文件创建获取 **/home** 和 **/home/user** 的读锁，以及 **/home/user/foo** 上的写锁。因为它们尝试获取 **/home/user** 上有冲突的锁，这两个操作将会合适地串行。文件创建不会获取父级目录的写锁，因为没有“目录 (directory)”，或者类似 inode 的数据结构需要被保护以免于被修改。名称上的写锁足以保护父级目录不被删除。

这种锁机制一个比较好的属性是，它允许在相同目录上进行并发的变更。例如，可以在相同目录下并发地执行多个文件创建：每个文件创建获取目录名的读锁和文件名的写锁。**目录名的读锁能够防止该目录被删除、重命名或者进行快照操作。文件名上的写锁会将两次创建同名文件的尝试串行化。**

因为命名空间可以有很多节点，读写锁对象被惰性分配且一旦他们没有被使用就会被删除。而且，**锁会以一致性顺序被获取以避免死锁：他们首先通过命名空间树的层级来排序，相同层级的通过字典序来排序。**

4.2 副本放置 (Replica Placement)

GFS 集群高度分布于多个层次。它通常拥有散布于许多机架上的数百台 chunkserver。这些 chunkserver 同样会被来自相同或不同机架上的数百个客户端访问。不同机架上的两台

机器间的通信可能要跨越一个或多个网络交换机。此外，一个机架上的出站带宽或入站带宽可能会小于机架内所有机器的总带宽。多级分布对分布数据的可扩展性 (scalability)、可靠性 (reliability) 和可用性 (availability) 提出了独特的挑战。

块副本放置策略有两个目标：**最大化数据可靠性 (reliability) 与可用性 (availability)** 和**最大化网络带宽利用率**。对于这两种情况，仅仅将副本分布于不同的机器上是不够的，这仅能应对磁盘或机器故障和充分利用每台机器的网络带宽。我们必须还要把副本分布于不同的机架上。这样可以确保在整个机架被损坏或者离线的情况下（例如，由于像网络交换机或电源电路等共享资源的故障），一个块的某些副本仍然存活并保持可用。这也意味着一个块的流量，尤其是读，**可以利用多个机架的总带宽**。另一方面，**写流量必须流经多个机架**，这是我们心甘情愿做出的取舍。

4.3 创建，再复制，再平衡 (Creation, Re-replication, Rebalancing)

创建块副本有三个原因：块创建、再复制、以及再平衡。

当 master **创建 (create)** 一个块时，它选择在什么位置来初始化空的副本。它考虑几个因素：（1）我们想把新的副本放在磁盘空间利用率低于平均水平的 chunkservers 上。随着时间推移，这会使 chunkserver 之间的磁盘利用率趋于均衡。（2）我们希望限制每个 chunkserver 上 "最近" 创建的数量。虽然创建本身开销很低，但它**可靠地预测了即将到来的大量写入流量**，因为块是在被要求写入时创建的，而在我们的追加一次读取多次 (append-once-read-many) 工作负载中，一旦它们被完全写入，它们通常在实际中会变成只读的。（3）正如上面讨论的，我们想要把一个块的副本分布在不同的机架上。

一旦副本的数量低于用户指定的目标数量，master 就会对一个块进行**再复制 (re-replicate)**。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已经损坏，它的一个磁盘因为错误而被无法使用，或者复制目标增加了。每一个需要再复制的块都会根据以下几个因素进行优先级排序。一是它离复制目标有多远。例如，相较于只丢失了一个副本的块，我们会给丢失了两个副本的块更高的优先级。此外，相较于最近被删除的文件的块，我们更倾向于对存在的文件的块进行再复制（见 Section 4.4）。最后，为了最小化故障对正在运行的程序的影响，**我们对任意阻塞客户端操作的块提高其优先级**。

master 选择优先级最高的块，然后通过命令某个 chunkserver 直接从一个已存在的有效副本拷贝块数据来对其"克隆 (clone)"。新副本以和被克隆块相似的目标来进行放置：均衡磁盘空间利用率，限制任意单个 chunkserver 上的活动克隆操作，以及在机架间分布副本。为了避免克隆流量淹没客户端流量，master 在集群和每个 chunkserver 上都限制了活动的克隆操作的数量。此外，每个 chunkserver 通过限流其对源 chunkserver 的读取请求从而限制其在每次克隆操作上花费的带宽。

最后，master 周期性地对副本进行**再平衡 (rebalance)**：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。同时通过这个过程，master 逐渐填满一个新的 chunkserver，而不是瞬间用新的块和随之而来的大量写流量使其应接不暇。对于新副本的放置标准类似于上面讨论的那些。此外，master 还必须选择移除哪个已存在的副本。通常来讲，它更倾向于移除那些位于磁盘空闲空间低于平均水平 chunkserver 上的块从而均衡磁盘空间使用。

4.4 垃圾回收 (Garbage Collection)

当一个文件被删除后，GFS 不会立即回收可用的物理存储空间。它只是在文件和块层级的周期性垃圾收集过程中进行惰性回收。我们发现，这种方法使系统更加简单和可靠。

4.4.1 机制 (Mechanism)

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操作。然而，文件并没有立即回收资源，而是被重新命名为一个包含删除时间戳的隐藏名称。在 master 周期性扫描文件系统命名空间的过程中，如果这些隐藏文件存在超过三天（时间间隔可配置），它就会将其删除。在这之前，该文件仍然可以通过新的特殊名称来读取，并且可以通过将其重新命名为正常名称来取消删除。当隐藏的文件从命名空间中删除时，**它在内存中的元数据会被删除**。这就有效地切断了它和它所有的块之间的链接

在一个类似的对块命名空间进行的周期性扫描中，master 识别孤儿块（即，那些从任何文件都无法访问的块）并且擦除这些块的元数据。在和 master 周期性交换的心跳 (HeartBeat) 消息中，每个 chunkserver 报告它拥有的块的子集，master 则回复在其元数据中不再存在的所有块的标识。**chunkserver 可以自由删除这类块的副本**。

4.4.2 讨论 (Discussion)

尽管分布式垃圾回收是个难题，它在编程语言的上下文环境中需要复杂的解决方案，但是在我们的情况中，它是相当简单的。我们可以轻易识别出块的所有引用：它们位于仅由 master 维护的文件到块的映射中。我们还可以很容易地识别出所有的块副本：它们是每个 chunkserver 上指定目录下的 Linux 文件。**任何不被 master 所知的副本都是“垃圾 (garbage) ”。**

与急于回收相比，垃圾回收的存储回收方式提供了几个优势：首先，它在组件失效十分常见的大型分布式系统中是简单可靠的。块创建可能在某些 chunkserver 上成功但是在另一些 chunkserver 上失败，留下了 master 不知晓其存在的副本。副本删除消息可能会（因为网络故障）丢失，master 必须记得在自己和 chunkserver 的失效中将其重发。垃圾回收提供了一个统一且可靠的方式来清理任何不知道是否有用的副本。其次，它将存储回收**合并进 master 周期性的后台活动中**，比如命名空间的周期性扫描和与 chunkserver 之间的周期性握手 (handshake)。因此，它可以被批量完成，成本是摊销的。此外，它仅在 master 相对空闲的情况下才被完成。master 可以更迅速地响应客户端的请求。第三，回收存储的延迟**提供了一个防止意外的、不可逆转的删除的保护网**。

根据我们的经验，它的主要的缺点是，当存储空间紧张时，延迟有时会阻碍用户对使用情况进行微调的努力。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空间**。如果被删除的文件再次被明确删除，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内的文件中的所有块都进行无复制存储，任何被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 过期副本检测 (Stale Replica Detection)

如果 chunkserver 失效以及在其宕机期间错过块的变更，块副本就变得过期 (stable)。对于每个块，master 维护了一个**块版本号 (chunk version number)** 来区分最新和过期的副本。

每当 master 赋予一个块新的租约时，它会增加块版本号并通知最新的副本。 master 和这些副本全都在其持久化状态值记录最新的版本号。这在客户端被通知之前发生，因此也在客

客户端能够开始对块进行写入之前。如果另一个副本当前不可用，它的块版本号将不会被提升。当chunkserver重启以及报告它的块及其所关联的块版本号的集合时，master将会检测这个chunkserver是否有过期的副本。如果master看到一个版本号大于其记录中的版本号，master就认为它在赋予租约时失效了并会把较高的版本号作为最新的版本号。

master会在其周期性的垃圾回收中移除过期的副本。在那之前，当master回复客户端关于块信息的请求时，它会把过期副本当做是完全不存在的。作为另一项保障措施，master在通知客户端哪个chunkserver持有一个块的租约以及当它在一个克隆操作中命令一个chunkserver从另一个chunkserver读取块时，master会把块版本号包含在内。客户端或者会在其执行操作之前验证版本号从而保证总是访问最新的数据。

5. 容错和诊断 (FAULT TOLERANCE AND DIAGNOSIS)

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失效。组件的质量和数量共同使这些问题成为常态而非例外：**我们不能完全信任机器，也不能完全信任磁盘**。组件失效可能导致系统不可用，或者更糟糕的是，数据被破坏。我们将讨论如何应对这些挑战，以及我们在系统中构建的工具，以便在发生不可避免的问题时进行诊断。

5.1 高可用性 (High Availability)

在一个GFS集群中的数百台服务器中，任意给定时间都必然有些服务器是不可用的。我们通过两种简单有效的策略来保持整个系统的高可用：快速恢复和复制。

5.1.1 快速恢复 (Fast Recovery)

无论master和chunkserver如何终止，它们都被设计成能在几秒钟内恢复状态并启动。事实上，我们没有区分正常和非正常的终止。服务器日常也是通过杀掉进程来关闭。客户端和其他服务器在当未完成的请求超时时，会遇到一个小插曲，它们重新连接到重新启动的服务器，然后重新尝试请求。Section 6.2.2报告了观察到的启动次数。

5.1.2 块复制 (Chunk Replication)

如前面所述，每个块在不同机架上的多个chunkserver上进行复制。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。默认是三份。当chunkserver离线或者通过验证校验和检测到损坏的副本时，master会根据需要克隆已存在的副本，以保持每个块被充分复制。

(见 Section 5.2)。尽管复制已经对我们很有帮助，但是我们仍在探索其他形式的跨服务器冗余，如**奇偶校验 (parity)** 或**纠删码 (erasure code)**，以满足我们日益增长的只读存储需求。我们认为，在我们这个非常松耦合的系统中实现这些更复杂的冗余方案是具有挑战性的，但也是可以解决的，因为我们的流量是由追加和读取而不是小规模的随机写入所主导的。

5.1.3 Master复制 (Master Replication)

master状态出于可靠性被复制。**它的操作日志和核对点在多台机器上被复制**。对状态的一个变更仅在它的日志记录被刷入本地和所有master副本上的磁盘之后，才被认为是已提交的 (committed)。简单起见，一个master进程仍然负责所有的变更，以及在内部改变系统的后台活动，如垃圾回收。当它发生故障时，它几乎可以立即重新启动。如果它的机器或磁盘故障，**GFS外部的监控基础设施会在其他拥有复制操作日志的地方启动一个新的**

master进程。客户端仅使用master的标准名称（例如，gfs-test），这是一个DNS别名，且如果master被分配到另一台机器上，它可以被修改。*(Hades注：热备的方式提高可用性)*

而且，“影子(shadow)”master即使是在主master宕机的时候也**只提供对文件系统的只读访问**。它们是影子，而不是镜子，因为它们**可能略微滞后于主体(primary)**，**通常是几分之一秒**。对于那些没有被频繁变更的文件或应用程序不介意得到稍微过期的结果的文件，这些文件的读可用性被提高了。事实上，因为文件内容是从chunkserver读取的，所以应用程序不会观察到过期的文件内容。短时间内可能会过期的是文件元数据，像目录内容或者访问控制信息。

为了让自己及时被通知，影子master会读取不断增长的操作日志的副本，并对其数据结构应用与master完全相同的变更顺序。和master一样，它在启动时（之后就不经常）轮询chunkservers以定位块副本，并与它们频繁地交换握手(handshake)消息以监控它们的状态。它仅在master决定创建和删除副本所导致的副本位置更新方面依赖于master。

5.2 数据完整性 (Data Integrity)

每个chunkserver使用校验和来检测存储数据的损坏。考虑到一个GFS集群通常在数百台机器上有数千个磁盘，它会经常发生磁盘故障，导致读写路径上的数据损坏或丢失。（一个原因见Section 7。）我们使用其他的块副本从损坏中恢复数据，但是通过比较chunkserver之间的副本来自检测数据损坏是不切实际的。**而且，不一致的副本可能是合理的：GFS变更的语义，尤其是前面讨论的原子性记录追加，不保证完全一致的副本。**因此，每个chunkserver必须通过维护校验和独立地验证它拥有的拷贝的完整性。

一个块(chunk)被分成64KB大小的区块(block)。每个区块有一个对应的32位的校验和。像其他的元数据一样，校验和与用户数据分开，被保存在内存中且通过日志持久化存储。*(Hades注：在应用层再做一层校验，但其实物理磁盘本身也会有汉明码校验)*

对于读取操作，chunkserver在向请求者（无论是客户端还是其他chunkserver）返回任何数据之前，会验证对应读取范围的数据块的校验和。因此，chunkserver不会将损坏的数据传播到其他的机器。如果一个区块和记录的校验和不匹配，chunkserver会给请求者返回一个错误并把这个不匹配的情况上报给master。对应的，请求者将会从其他副本读取，而master将会从另一个副本克隆这个块。在一个有效的新副本到位后，master命令上报不匹配情况的chunkserver删除它的副本。

校验和出于几个原因对读取性能有些许影响。由于我们大部分的读取至少跨越了几个区块，所以为了验证，我们只需要读取并对相对较少的额外数据进行校验和检验。GFS客户端代码通过尝试在校验和区块边界对齐读取，进一步降低了这种开销。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，而且校验和计算经常可以和I/O同时进行。

校验和计算针对追加到块尾部的写入操作进行了大量优化（相对于重写已存在数据的写入），因为这种写入操作在工作负载中占主导地位。我们只是递增地更新最后一个部分校验和区块的校验和，并为任何通过追加填充的全新的校验和区块计算新的校验和。即使最后的部分校验和区块已经损坏且我们当前无法检测出来，新的校验和的值和已存储的数据将无法匹配，并且当这个块下次被读取时，数据损坏会像平常一样被检测到。

与此相反，如果一个写操作覆盖块的一个已存在（数据的）区间，我们必须读取和验证被覆盖区间内的第一个区块和最后一个区块，然后执行写操作，并且最后计算和记录新的校

验和。如果我们没有在部分覆盖它们之前验证第一个和最后一个区块，新的校验和可能会掩盖存在于未被覆盖的区域内的损坏情况。

在空闲期间，chunkserver可以扫描和验证非活动块的内容。这使得我们可以检测很少被读取的块中的损坏。一旦检测到损坏，master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master，使master认为它有数量足够的有效的块副本。

5.3 诊断工具 (Diagnostic Tools)

广泛而详细的诊断日志在**问题隔离 (problem isolation)**、**调试 (debugging)** 和**性能分析 (performance analysis)** 方面提供了不可估量的帮助，而产生的成本却微乎其微。如果没有日志，就很难理解机器之间的转瞬即逝、不可重现的交互。GFS服务器生成记录了许多重要事件的诊断日志（比如，chunkserver的启动和停机）以及所有的RPC请求和回复。这些诊断日志可以被随意删除而不会影响系统的正确性。尽管如此，在空间允许的情况下，我们还是试图尽可能地保留这些日志。

RPC日志包括了线上发送的准确的请求和响应，除了正在读取或写入的文件数据。通过匹配请求和回复以及整理不同机器上的RPC记录，我们可以复现整个交互历史来诊断问题。这些日志也作为我们负载测试和性能分析的追溯。

日志对性能的影响是最小的（而且远远低于所带来的好处），因为这些日志是按顺序和异步写入的。最近发生的事件也保存在内存中，可用于持续在线监测。

6. 测量 (MEASUREMENTS)

在本节中我们会展示一些微基准测试，以展示GFS架构与实现中的固有瓶颈，以及一些来自谷歌的正在使用的真实集群的数字。

6.1 微基准测试 (Micro-benchmarks)

我们在一个由一个master、两个master备机、16个chunkserver以及16个client的GFS集群上测试性能。请注意，此配置是为了便于测试而设置的。典型的集群有数百个chunkserver和数百个client。

所有的机器都配置了双1.4GHz PIII CPU、2GB内存、两个80GB 5400rpm磁盘，以及一个连接着HP 2524交换机的100Mbps全双工以太网。所有19台GFS服务器机器都连接到一个交换机，以及所有16台客户端机器都连接到另一个交换机。这两个交换机通过一个1Gbps的链路连接。

6.1.1 读 (Reads)

N个客户端同时从文件系统中读取。每个客户端从一个320GB的文件集中读取一个随机选择的4MB区域。这样会重复256次，以便每个客户端最终读取1GB的数据。这些chunkserver加在一起只有32GB的内存，所以我们预计在Linux缓冲区缓存中最多会有10%的命中率。我们的结果应该接近于冷高速缓存 (cache) 的结果。

图3 (a) 显示了N个客户端的总体读取率及其理论极限。当两个交换机之间的1Gbps链路饱和时，极限峰值为125MB/s，当100Mbps网络接口饱和时，极限上限为12.5MB/s，以适用者为准。当只有一个客户机正在读取时，观察到的读取速率为10MB/s，或每个客户机限制的80%。对于16个读者 (reader) 或每个客户端，总读取率达到94MB/s，约为125MB/s链接限制的75%。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块服务器读取的概率也会增加。

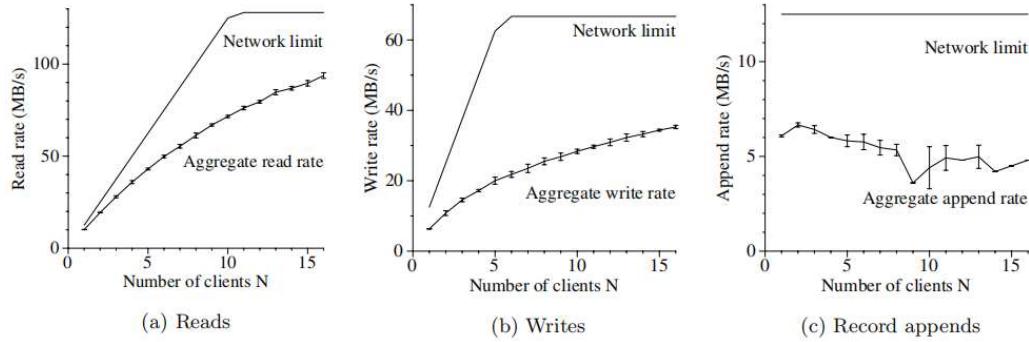


Figure 3: Aggregate Throughputs. Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% confidence intervals, which are illegible in some cases because of low variance in measurements.

6.1.2 写 (Writes)

客户端同时写入N个不同的文件。每个客户端以一系列1MB的写入方式将1GB的数据写入一个新文件。总体写速率及其理论极限（如图3 (b) 所示）限制稳定在67MB/s，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12.5MB/s的输入连接。

一个客户端的写入速率是6.3MB/s，大约是上限的一半。最主要的罪魁祸首是我们的网络栈 (network stack)。它不能很好地与我们用于将数据推送副本块的管道方案交互。将数据从一个副本传播到另一个副本的延迟会降低总体写入速率。

16个客户端的总体写速率达到35MB/s（或每个客户端2.2MB/s），大约是理论上限的一半。与读取的情况一样，随着客户机数量的增加，多个客户机更有可能向同一chunkserver并发地写入。此外，16位writer比16位reader更有可能发生冲突，因为每一篇文章都涉及三个不同的复制品。

写操作的速度比我们想要的要慢。在实践中，这并不是一个主要问题，因为尽管它增加了单个客户机所看到的延迟，但它也不会显著影响系统交付给大量客户机的总体写操作带宽。

6.1.3 记录追加 (Record Appends)

图3 (c) 显示了记录追加操作的性能。N个客户端同时对单个文件进行追加操作。性能受到存储文件最后一块的chunkserver的网络带宽的限制，这与客户端的数量无关。它从一个客户端的6.0MB/s开始，从16个客户端降至4.8MB/s，**这主要是由于不同客户端的拥塞和网络传输速率的差异。**

我们的应用程序倾向于同时生成多个这样的文件。换句话说，N个客户端同时附加到M个共享文件中，其中N个和M个都在几十个或几百个中。因此，在我们的实验中，块服务器网络拥塞在实践中并不是一个重要的问题，因为当另一个文件的块服务器繁忙时，客户端可以在编写一个文件方面取得进展。

6.2 实践中的集群 (Real World Clusters)

我们现在研究了谷歌中使用的两个集群，它们代表了其他几个类似的集群。集群A经常被100多名工程师用于研发。一个典型的任务由用户发起，运行时间长达几个小时。它将读取几个MB到几个TB的数据，转换或分析数据，并将结果写回集群。集群B主要用于生产数据处理。任务会持续很长时间并且会持续地产生和处理数个TB的数据集，同时只有偶尔的人为介入。在这两种情况下，一个“任务”由许多机器上同时读写多个文件的多个进程组成。

Cluster	A	B
Chuckservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Table 2: Characteristics of two GFS clusters

6.2.1 存储 (Storage)

如表2中的前五个条目所示，两个集群都有数百个chunkserver，支持很多TB的磁盘空间，并且磁盘利用率适当地满。“已使用的空间”包括所有的块副本。几乎所有的文件都被复制了三次。因此，集群分别存储了18TB和52TB的文件数据。

这两个集群有相似数量的文件，尽管B有更大比例的死文件，即这些文件将被删除或替换为一个新版本，但其存储空间尚未被回收。它也有更多的数据块，因为它的文件往往更大。

6.2.2 元数据 (Metadata)

chunkservers总共存储了数十个GB的元数据，主要是64KB大小的用户数据块的校验和 (checksum)。保存在chunkserver上的唯一其他元数据是第4.5节中讨论的数据块版本号。

保存在主服务器上的元数据要小得多，只有几十个MB，或者平均每个文件大约100个字节。这与我们的假设相一致，即在实际应用中，master的内存大小并不会限制系统的容量 (*Hades*注：master的内存使用其实很小，硬件完全足以支撑)。每个文件的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(访问)权限、从文件到chunk的映射，以及每个chunk的当前版本号。此外，对于每个块，我们存储当前的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，包括分块服务器和主服务器，都只有50到100MB的元数据。因此，恢复是快速的：在服务器能够回答查询之前，从磁盘读取此元数据只需要几秒钟。然而，主

服务器在一段时间内有些困难——通常是30到60秒——直到它从所有的块服务器中获取了块的位置信息。

6.2.3 读写速率 (Read and Write Rates)

表3显示了不同时间段的读取速率和写速率。当进行这些测量时，这两个星团已经上升了大约一周。（集群最近重新启动，升级到GFS的新版本）

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Table 3: Performance Metrics for Two GFS Clusters

因为服务器重启的缘故，平均写速率小于30MB/s。当我们进行这些测量时，B处于瞬时写入爆发当中，产生了大约100MB/s的数据，这产生了300MB/s的网络负载，因为写入被传播到三个副本。

读速率远远高于写速率。正如我们所假设的一样，总工作负载包含的读操作多于写操作。这两个集群都有着大量的读操作。特别是，A在前一周的读取率一直保持在580MB/s。它的网络配置可以支持750个MB/s，因此它可以有效地利用其资源。集群B可以支持1300MB/s的峰值读取速率，但它的应用程序仅使用了380MB/s。

6.2.4 Master负载 (Master Load)

表3还显示，发送到主节点的操作速率约为每秒200到500次操作。主服务器可以很容易地跟上这个速率，因此它不是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。它花了大部分时间按顺序扫描大型目录（其中包含数十万个文件）来寻找特定的文件。此后，我们已经更改了主数据结构，以允许通过名称空间进行有效的二分搜索。它现在可以轻松地支持每秒进行成千上万次的文件访问。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 (Recovery Time)

chunkserver失效后，一些块会副本数不足，必须要克隆才能恢复其副本数量。恢复所有这些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中杀死了两个chunkserver。该分块服务器有大约15,000个包含600GB数据的分块。为了限制对运行应用程序的影响，并为调度决策提供回旋余地，我们的默认参数将该集群限制为91个并发克隆（占chunkserver数量的40%），其中每个克隆操作最多允许消耗6.25MB/s（50Mbps）。所有的chunk在23.2分钟内恢复，有效复制速率为440MB/s。

在另一个实验中，我们杀死了两个块服务器，每个服务器都有大约16000个块和660GB的数据。这个双重失败将266个块减少为只有一个副本。这266个块以更高的优先级被克隆，并且都在2分钟内恢复到至少2倍的副本数，从而使集群处于可以容忍另一个块服务器失败而不丢失数据的状态。

6.3 工作负载分解 (Workload Breakdown)

在本节中，我们将详细介绍两个GFS集群的工作负载，与6.2节类似但不相同。集群X用于研究和开发，而集群Y用于生产数据处理。

6.3.1 方法和注意事项 (Methodology and Caveats)

这些结果只包括客户机发起的请求，因此它们反映了应用程序为整个文件系统生成的工作负载。它们不包括为了响应客户端而导致server之间发起的请求，或内部的后台活动，例如转发的写入或重新平衡。

关于I/O操作的统计数据，是基于由GFS服务器记录的、实际RPC请求启发式地重建的信息。例如，GFS客户端代码可能会将一个读取分解成多个RPC请求，以增加并行性，因此我们可以从中推断出原始读操作是什么。由于我们的访问模式是高度格式化的，所以我们期望在噪声中出现任何错误。应用程序的显式日志记录可能会提供稍微准确一点的数据，但在逻辑上不可能重新编译和重新启动数千个正在运行的客户端，而且从尽可能多的机器上收集结果也很麻烦。

我们应该小心，不要过分概括我们的工作量。由于谷歌完全控制GFS和它的应用程序，这些应用程序往往会对GFS进行调整，反过来GFS也是为这些应用程序设计的。这样的相互影响可能也存在于常规的应用程序和文件系统之间，但是在我们这个例子当中，二者的影响是更加显著的。

6.3.2 Chunkserver负载 (Chunkserver Workload)

表4显示了按大小显示的操作分布。读取大小呈双峰分布。小的读操作（低于64kb）来自搜索密集型客户端，它们在大文件中查找小块数据。大的读操作（超过512KB）来自于对整个文件的长且顺序性的读取。

Operation Cluster	Read		Write		Record Append	
	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..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..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

在集群Y中，大量的读取根本不返回任何数据。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。当消费者读取文件的末尾时，生产者会并发地附加到一个文件中。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较少显示这种情况，因为它通常用于短寿命的数据分析任务，而不是长寿命的分布式应用程序。

写操作的大小也显示出一个双峰分布。大的写操作（超过256KB）通常是由于写入者内部关键的缓冲。缓冲更少的数据、检查点或同步，或者只是为更少的写入（低于64KB的写入）生成更少的数据。

至于追加记录操作，集群Y看到的大型记录附加的比例比集群X要高得多，因为我们使用集群Y的生产系统更积极地针对GFS进行了调优。

表5显示了在不同大小的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256KB）通常转移了大部分字节。由于随机查找工作负载，小的读操作（小于64kb）确实传输了一小部分但重要的读取数据。

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..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

Table 5: Bytes Transferred Breakdown by Operation Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

6.3.3 追加与写 (Appends versus Writes)

记录追加操作被大量使用，特别是在我们的生产系统中。对于集群X，写操作与记录追加操作的比例，按传输字节来计算为108: 1，按操作次数来计算为8: 1。对于生产系统使用的集群Y，比例分别为3.7: 1和2.5: 1。此外，这些比例表明，对于两个集群，记录附加操作往往大于写操作。然而，对于集群X，在测量期间，附加记录的总体使用量相当低，因此结果可能会被一到两个具有特定缓冲区大小选择的应用程序所影响准确性。*(Hades注：这里有点没看懂，不应该是写操作的数据量和操作数更大吗？)*

正如预期的那样，我们的关于数据改变的工作负载是由追加写操作而不是覆盖写操作来主导的。我们测量了在Primary副本上覆盖写的数据量。这估计了一种情形，即客户端故意覆盖以前写入的数据，而不是附加新的数据。对于集群X，覆盖写操作占修改字节的0.0001%以下，占修改操作的0.0003%以下。对于集群Y，这些比率都是0.05%。虽然这很微小，但仍然比我们预期的要高。结果是，由于错误或超时，这些覆盖大多来自客户端重试。它们本身并不是工作负载的一部分，而是重试机制的结果。

6.3.4 Master负载 (Master Workload)

表6显示了对Master的请求类型的分类。大多数请求是为了读操作而询问块位置(FindLocation)，以及租赁持有人信息(FindLeaseHolder)的数据改变。*(Hades注：即Primary的周期性变更或强制变更)*

Cluster	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
All other combined	0.5	0.8

Table 6: Master Requests Breakdown by Type (%)

集群X和Y看到的删除请求数量显著不同，因为集群Y存储的生产数据集会定期重新生成并替换为新版本。这些差异在打开请求中的差异中进一步隐藏，因为旧版本的文件可能通过被打开从头写入而隐式删除（在Unix打开术语中，模式为“w”）。

查找匹配文件是一种模式匹配请求，它支持“ls”和类似的文件系统操作。与对主服务器的其他请求不同，它可能会处理名称空间的很大一部分，因此（操作代价）可能会很昂贵。集群Y更经常看到它，因为自动数据处理任务倾向于检查文件系统的一部分，**以理解全局应用程序状态**。相比之下，集群X的应用程序受到更显式的用户控制，并且通常会提前知道全部所需文件的名称。

7. 经历 (EXPERIENCES)

在构建和部署GFS的过程中，我们遇到了各种各样的问题，包括一些操作问题和一些技术问题。

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，其用途逐渐发展到包括研究和开发任务。在设计之初，它几乎不支持权限控制（permissions）和限额（quotas）。但现在，它包括了这些内容的初步形态。虽然生产系统有很好的纪律和控制，但用户有时却不一定。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是与磁盘和Linux相关的。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，但实际上只对最近的版本作出了可靠的响应。由于协议版本非常相似，这些驱动器大多可以工作，但偶尔不匹配会导致驱动器对自身的状态和内核对驱动器的状态之间存在分歧。这将由于内核中的问题而悄无声息地破坏数据。**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。**

(Hades注：GFS会提出数据校验的初衷，便是用来解决OS和磁盘之间状态不一致的问题的，否则磁盘本身是有错误校验的，理应无需OS自行判断)

之前，由于fsync()的成本，我们在Linux2.2内核中遇到了一些问题。它的成本与文件的大小成正比，而不是修改部分的大小。这对于我们的大型操作日志来说是一个问题，特别是在我们实现检查点之前。我们通过使用同步写操作解决这个问题了一段时间，并最终迁移到了Linux2.4。

另一个Linux问题是一个读写锁，当一个地址空间中的任何线程，从磁盘换入内存页（读锁）或在调用mmap()修改地址空间（写锁）时，都必须持有它。我们看到系统在轻负载下出现了瞬态超时，并努力寻找资源瓶颈或零星的硬件故障。最终，我们发现，当磁盘线程

在以前映射的数据中分页时，这个锁阻止了主网络线程将新数据映射到内存中。由于我们主要受到网络接口的限制，而不是内存复制带宽，我们通过用pread()替换mmap()，代价是额外的副本。

尽管偶尔会出现问题，但Linux代码的可用性已经帮助我们不断地探索和理解系统行为。在适当的情况下，我们会改进内核，并与开源社区共享这些更改。

8. 相关工作 (RELATED WORK)

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名称空间，**允许数据透明地移动，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]的方式跨存储服务器传播文件的数据，以提供总体性能和更强的容错能力。

由于磁盘相对便宜，而且复制比更复杂的RAID[9]方法更简单，因此GFS目前只使用复制来进行冗余，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面提供任何缓存。我们的目标工作负载在单个应用程序运行中很少被重用，因为它们要么在大数据集上进行流式操作，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、Minnesota的GFS[11]和GPFS[10]，删除了集中式服务器，并依赖于分布式算法来实现一致性和管理。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，集中式master使实现复杂的块放置和复制策略变得更加容易，因为master已经拥有了大部分相关信息并控制它如何更改。**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**。可伸缩性和高可用性（对读操作而言）目前是由我们的shadow master机制提供的（*Hades*注：即主备架构，热备）。通过附加到预写日志（write-ahead log, WAL），使对主状态的更新持久化。因此，我们可以采用像Harp[7]中那样的master拷贝方案，以提供比我们当前的方案更强的高可用性和一致性保证。

我们正在解决一个类似于Lustre[8]的问题，即向大量客户提供总体性能。然而，我们通过**关注应用程序的需求**，而不是构建一个兼容POSIX的文件系统，从而大大简化了这个问题。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最接近于NASD架构[4]。虽然NASD架构是基于网络连接的磁盘驱动器，但GFS使用商品机器作为chunkserver，就像在NASD原型中所做的那样。与NASD工作不同，我们的chunkserver使用延迟分配的固定大小的块，而不是可变长度的对象。此外，GFS还实现了生产环境中需要的再平衡、复制和恢复等特性。

与Minnesota的GFS和NASD不同，我们并不改变存储设备的模型。我们专注于解决包含既有商品组件的复杂分布式系统的日常数据处理需求。

由原子追加写操作（Atomic Record Appends, Section 3.3）实现的生产者-消费者队列，解决了与River[2]中的分布式队列类似的问题。**River使用分布在机器上的基于内存的队列和细致的数据流控制，而GFS使用一个持久的文件，许多生产者可以并发地附加到该文件中**。River模型支持m到n个分布式队列，但缺乏持久存储所附带的容错能力，而GFS只有效地支持m到1个队列。多个使用者可以读取同一个文件，但它们必须相互协调，以划分传入的负载。

9. 结论 (CONCLUSIONS)

Google File System证明了在商用硬件上支持大规模数据处理工作负载的基本质量。虽然一些设计决策是特定于我们独特的设置，但许多可能适用于类似规模和成本意识的数据处理任务。

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，我们重新检查了传统的文件系统假设。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规范而不是异常，对大部分附加到（可能并发）然后读取（通常按顺序）的大型文件进行优化，并扩展和放松标准文件系统接口，以改进整个系统。

我们的系统通过不断监控、复制关键数据和快速自动重构提供容错。块复制允许我们容忍chunkserver失效。这些故障发生的频率激发了一种新的在线修复机制，该机制可以定期和透明地修复损坏，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者和写入者 (readers and writers) 提供了高总体吞吐量。我们通过将通过主服务器传递的文件系统控制，与直接在chunkserver和客户端之间传递的数据传输分离来实现这一点。大型的块尺寸和块租赁 (机制)，使得常见操作中master的参与度最小化，并将权限委托给变更数据中的主副本 (primary replicas)。这使得一个简单的、集中化的主机成为可能，且它不会成为瓶颈。我们相信，我们的网络栈的改进，将解除当前对单个客户端所看到的写吞吐量的限制。

GFS已经成功地满足了我们的存储需求，并在谷歌中广泛应用于我们作为研发和生产数据处理的存储平台。它是一个重要的工具，使我们能够继续创新和打击整个网络规模上的问题。