GFS论文导读

> GFS 是谷歌的分布式文件系统，对应的开源经典实现是HDFS，其已经成为当今大数据领域的标配组件，我们从原始的GFS论文来讨论分布式文件系统是如何被设计出来的？
>
> 从论文出发，去分析一个系统设计问题应该如何思考，如何写出一篇优秀的设计文档？

论文译者: 阎伟老师

摘要

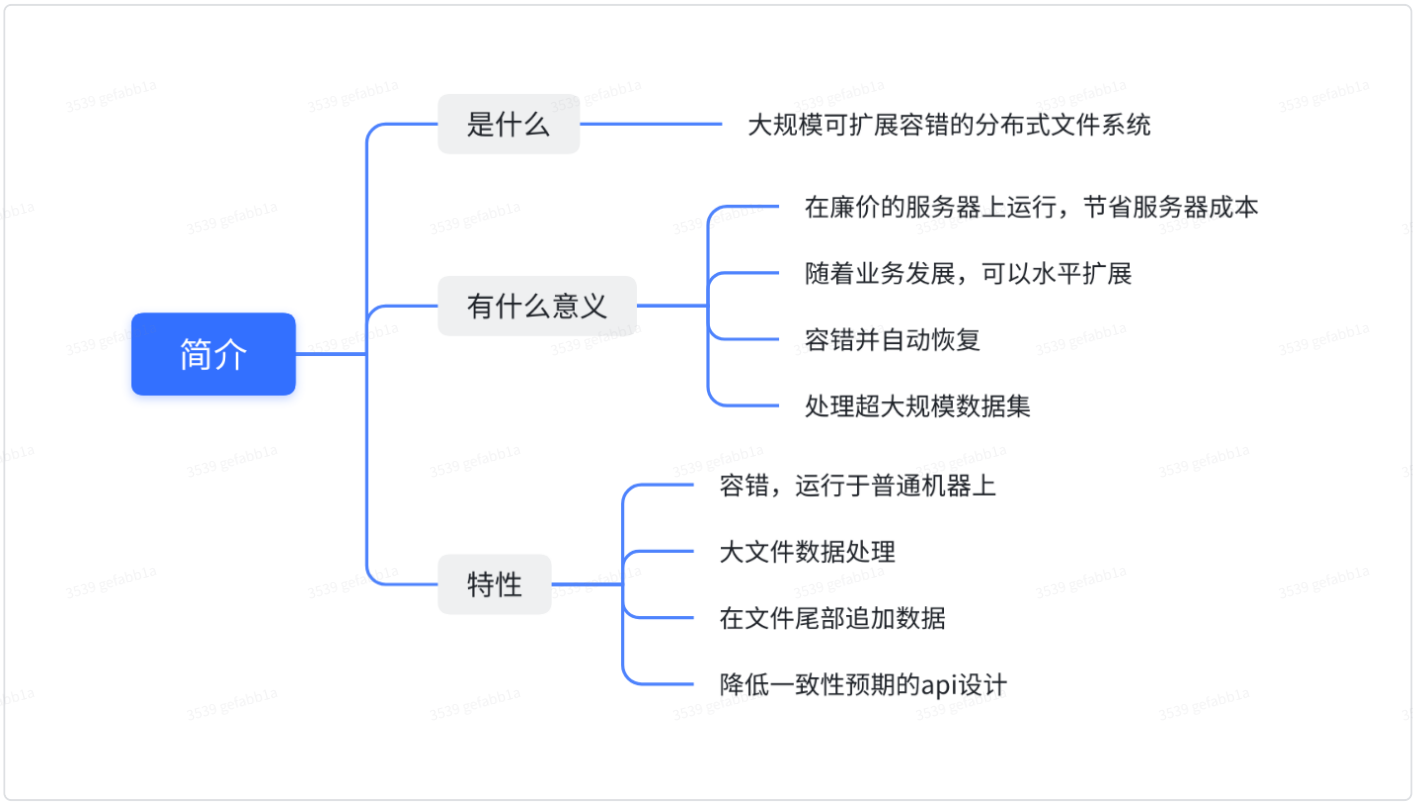
我们设计并实现了Google GFS文件系统，一个面向大规模数据密集型应用的、可伸缩的分布式文件系 统。GFS虽然运行在廉价的普遍硬件设备上，但是它依然了提供灾难冗余的能力，为大量客户机提供了高性能的服务。

虽然GFS的设计目标与许多传统的分布式文件系统有很多相同之处，但是，我们的设计还是以我们对自己的应用的负载情况和技术环境的分析为基础的，不管现在还是将来，GFS和早期的分布式文件系统的设想 都有明显的不同。所以我们重新审视了传统文件系统在设计上的折衷选择，衍生出了完全不同的设计思路。

GFS完全满足了我们对存储的需求。GFS作为存储平台已经被广泛的部署在Google内部，存储我们的服务 产生和处理的数据，同时还用于那些需要大规模数据集的研究和开发工作。目前为止，最大的一个集群利 用数千台机器的数千个硬盘，提供了数百TB的存储空间，同时为数百个客户机服务。

在本论文中，我们展示了能够支持分布式应用的文件系统接口的扩展，讨论我们设计的许多方面，最后列 出了小规模性能测试以及真实生产系统中性能相关数据。

1. 简介



首先，**组件失效被认为是常态事件**，而不是意外事件。GFS包括几百甚至几千台普通的廉价设备组装的存储机器，同时被相当数量的客户机访问。GFS组件的数量和质量导致在事实上，任何给定时间内都有可能发生某些组件无法工作，某些组件无法从它们目前的失效状态中恢复。我们遇到过各种各样的问题，比如 应用程序bug、操作系统的bug、人为失误，甚至还有硬盘、内存、连接器、网络以及电源失效等造成的 问题。所以，持续的监控、错误侦测、灾难冗余以及自动恢复的机制必须集成在GFS中。

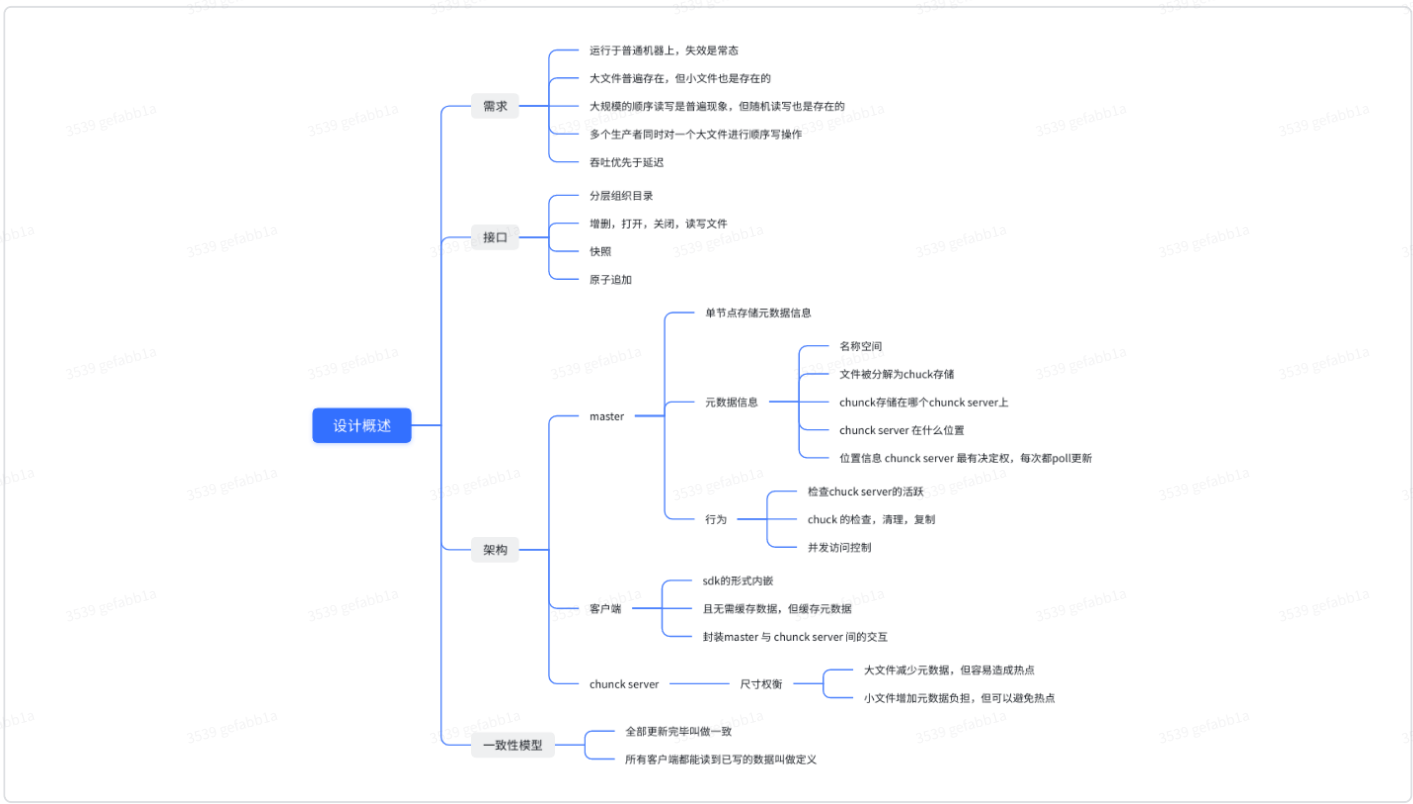
其次，**以通常的标准衡量，我们的文件非常巨大**。数GB的文件非常普遍。每个文件通常都包含许多应用程序对象，比如web文档。当我们经常需要处理快速增长的、并且由数亿个对象构成的、数以TB的数据集 时，采用管理数亿个KB大小的小文件的方式是非常不明智的，尽管有些文件系统支持这样的管理方式。因此，设计的假设条件和参数，比如I/O操作和Block的尺寸都需要重新考虑。

第三，**绝大部分文件的修改是采用在文件尾部追加数据**，而不是覆盖原有数据的方式。对文件的随机写入操作在实际中几乎不存在。一旦写完之后，对文件的操作就只有读，而且通常是按顺序读。大量的数据符合这些特性，比如:数据分析程序扫描的超大的数据集;正在运行的应用程序生成的连续的数据流;存档的数据;由一台机器生成、另外一台机器处理的中间数据，这些中间数据的处理可能是同时进行的、也可能是后续才处理的。对于这种针对海量文件的访问模式，客户端对数据块缓存是没有意义的，数据的追加操作是性能优化和原子性保证的主要考量因素。 (logic：顺序读取将不存在高热情况，因此没必要缓存)。

第四，**应用程序和文件系统API的协同设计提高了整个系统的灵活性**。比如，我们放松了对GFS一致性模型的要求，这样就减轻了文件系统对应用程序的苛刻要求，大大简化了GFS的设计。我们引入了原子性的记 录追加操作，从而保证多个客户端能够同时进行追加操作，不需要额外的同步操作来保证数据的一致性。 本文后面还有对这些问题的细节的详细讨论。

Google已经针对不同的应用部署了多套GFS集群。最大的一个集群拥有超过1000个存储节点，超过 300TB的硬盘空间，被不同机器上的数百个客户端连续不断的频繁访问。

2. 设计概述



2.1 假设

在设计满足我们需求的文件系统时候，我们的设计目标既有机会、又有挑战。之前我们已经提到了一些需要关注的关键点，这里我们将设计的预期目标的细节展开讨论。

- **系统由许多廉价的普通组件组成，组件失效是一种常态**。系统必须持续监控自身的状态，它必须将组 件失效作为一种常态，能够迅速地侦测、冗余并恢复失效的组件。
- **系统存储一定数量的大文件**。我们预期会有几百万文件，文件的大小通常在100MB或者以上。数个 GB大小的文件也是普遍存在，并且要能够被有效的管理。系统也必须支持小文件，但是不需要针对小文件做专门的优化。
- **系统的工作负载主要由两种读操作组成:大规模的流式读取和小规模的随机读取**。大规模的流式读取通常一次读取数百KB的数据，更常见的是一次读取1MB甚至更多的数据。来自同一个客户机的连续 操作通常是读取同一个文件中连续的一个区域。小规模的随机读取通常是在文件某个随机的位置读取 几个KB数据。如果应用程序对性能非常关注，通常的做法是把小规模的随机

读取操作合并并排序， 之后按顺序批量读取，这样就避免了在文件中前后来回的移动读取位置。

- **系统的工作负载还包括许多大规模的、顺序的、数据追加方式的写操作**。一般情况下，每次写入的数据的大小和大规模读类似。数据一旦被写入后，文件就很少会被修改了。系统支持小规模的随机位置写入操作，但是可能效率不彰。
- **系统必须高效的、行为定义明确的实现多客户端并行追加数据到同一个文件里的语义**。我们的文件通常被用于”生产者-消费者“队列，或者其它多路文件合并操作。通常会有数百个生产者，每个生产者进程运行在一台机器上，同时对一个文件进行追加操作。使用最小的同步开销来实现的原子的多路追加数据操作是必不可少的。文件可以在稍后读取，或者是消费者在追加的操作的同时读取文件。
- **高性能的稳定网络带宽远比低延迟重要**。我们的目标程序绝大部分要求能够高速率的、大批量的处理 数据，极少有程序对单一的读写操作有严格的响应时间要求。

2.2接口

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

另外，GFS提供了快照和记录追加操作。快照以很低的成本创建一个文件或者目录树的拷贝。记录追加操作允许多个客户端同时对一个文件进行数据追加操作，同时保证每个客户端的追加操作都是原子性的。这对于实现多路结果合并，以及”生产者-消费者”队列非常有用，多个客户端可以在不需要额外的同步锁定的情况下，同时对一个文件追加数据。我们发现这些类型的文件对于构建大型分布应用是非常重要的。快照 和记录追加操作将在3.4和3.3节分别讨论。

2.3架构

一个GFS集群包含一个单独的Master节点(alex注:这里的一个单独的Master节点的含义是GFS系统中 只存在一个逻辑上的Master组件。后面我们还会提到Master节点复制，因此，为了理解方便，我们把 Master节点视为一个逻辑上的概念，一个逻辑的Master节点包括两台物理主机，即两台Master服务 器)、多台Chunk服务器，并且同时被多个客户端访问，如图1所示。所有的这些机器通常都是普通的 Linux机器，运行着用户级别(user-level)的服务进程。我们可以很容易的把Chunk服务器和客户端都放在同一台机器上，前提是机器资源允许，并且我们能够接受不可靠的应用程序代码带来的稳定性降低的风险。

GFS存储的文件都被分割成固定大小的Chunk。在Chunk创建的时候，Master服务器会给每个Chunk分配一个不变的、全球唯一的64位的Chunk标识。Chunk服务器把Chunk以linux文件的形式保存在本地硬盘上，并且根据指定的Chunk标识和字节范围来读写块数据。出于可靠性的考虑，每个块都会复制到多个块服务器上。缺省情况下，我们使用3个存储复制节点，不过用户可以为不同的文件命名空间设定不同的复制级别。

Master节点管理所有的文件系统元数据。这些元数据包括名字空间、访问控制信息、文件和Chunk的映射信息、以及当前Chunk的位置信息。Master节点还管理着系统范围内的活动，比如，Chunk租

用管理 (alex注:BDB也有关于lease的描述，不知道是否相同)、孤儿Chunk(alex注:orphaned chunks)的回收、以及Chunk在Chunk服务器之间的迁移。Master节点使用心跳信息周期地和每个Chunk服务器通讯，发送指令到各个Chunk服务器并接收Chunk服务器的状态信息。

GFS客户端代码以库的形式被链接到客户程序里。客户端代码实现了GFS文件系统的API接口函数、应用程 序与Master节点和Chunk服务器通讯、以及对数据进行读写操作。客户端和Master节点的通信只获取元数据，所有的数据操作都是由客户端直接和Chunk服务器进行交互的。我们不提供POSIX标准的API的功能，因此，GFS API调用不需要深入到Linux vnode级别。

无论是客户端还是Chunk服务器都不需要缓存文件数据。客户端缓存数据几乎没有什么用处，因为大部分 程序要么以流的方式读取一个巨大文件，要么工作集太大根本无法被缓存。无需考虑缓存相关的问题也简化了客户端和整个系统的设计和实现。(不过，客户端会缓存元数据。)Chunk服务器不需要缓存文件数 据的原因是，Chunk以本地文件的方式保存，Linux操作系统的文件系统缓存会把经常访问的数据缓存在内存中。

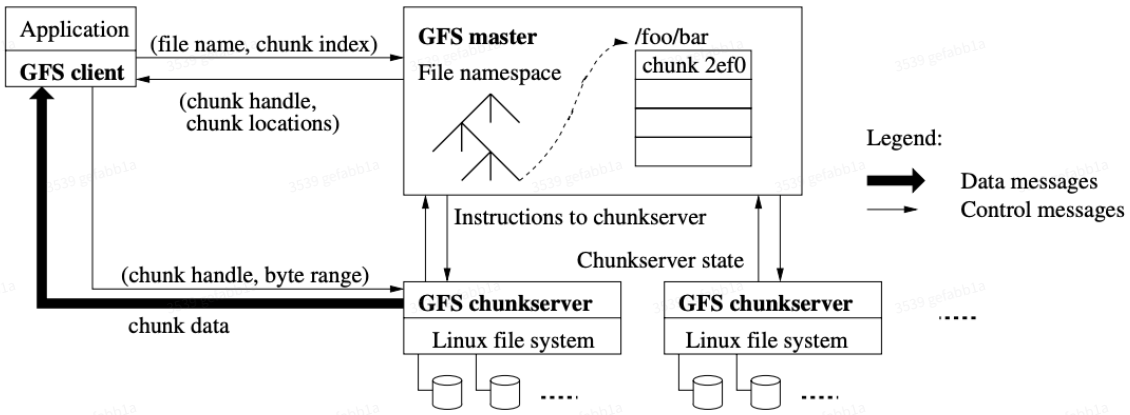


Figure 1: GFS Architecture

2.4 单一Master节点

单一的Master节点的策略大大简化了我们的设计。单一的Master节点可以通过全局的信息精确定位 Chunk的位置以及进行复制决策。另外，我们必须减少对Master节点的读写，避免Master节点成为系统的瓶颈。客户端并不通过Master节点读写文件数据。反之，客户端向Master节点询问它应该联系的 Chunk服务器。客户端将这些元数据信息缓存一段时间，后续的操作将直接和Chunk服务器进行数据读写操作。

我们利用图1解释一下一次简单读取的流程。首先，客户端把文件名和程序指定的字节偏移，根据固定的 Chunk大小，转换成文件的Chunk索引。然后，它把文件名和Chunk索引发送给Master节点。Master节点将相应的Chunk标识和副本的位置信息发还给客户端。客户端用文件名和Chunk索引作为key缓存这些信息。

之后客户端发送请求到其中的一个副本处，一般会选择最近的。请求信息包含了Chunk的标识和字节范 围。在对这个Chunk的后续读取操作中，客户端不必再和Master节点通讯了，除非缓存的元数据信息过期 或者文件被重新打开。实际上，客户端通常会在一次请求中查询多个Chunk信息，

Master节点的回应也可 能包含了紧跟着这些被请求的Chunk后面的Chunk的信息。在实际应用中，这些额外的信息在没有任何代 价的情况下，避免了客户端和Master节点未来可能会发生的几次通讯。

2.5 Chunk尺寸

Chunk的大小是关键的设计参数之一。我们选择了64MB，这个尺寸远远大于一般文件系统的Block size。每个Chunk的副本都以普通Linux文件的形式保存在Chunk服务器上，只有在需要的时候才扩大。 惰性空间分配策略避免了因内部碎片造成的空间浪费，内部碎片或许是对选择这么大的Chunk尺寸最具争议一点。

选择较大的Chunk尺寸有几个重要的优点。首先，它减少了客户端和Master节点通讯的需求，因为只需要 一次和Mater节点的通信就可以获取Chunk的位置信息，之后就可以对同一个Chunk进行多次的读写操作。这种方式对降低我们的工作负载来说效果显著，因为我们的应用程序通常是连续读写大文件。即使是 小规模的随机读取，采用较大的Chunk尺寸也带来明显的好处，客户端可以轻松的缓存一个数TB的工作数 据集所有的Chunk位置信息。其次，采用较大的Chunk尺寸，客户端能够对一个块进行多次操作，这样就 可以通过与Chunk服务器保持较长时间的TCP连接来减少网络负载。第三，选用较大的Chunk尺寸减少了 Master节点需要保存的元数据的数量。这就允许我们把元数据全部放在内存中，在2.6.1节我们会讨论元 数据全部放在内存中带来的额外的好处。

另一方面，即使配合惰性空间分配，采用较大的Chunk尺寸也有其缺陷。小文件包含较少的Chunk，甚至只有一个Chunk。当有许多的客户端对同一个小文件进行多次的访问时，存储这些Chunk的Chunk服务 器就会变成热点。在实际应用中，由于我们的程序通常是连续的读取包含多个Chunk的大文件，热点还不是主要的问题。

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：一个可执行文件被作为单个chunkfile写入GFS，然后同时在数百台机器上启动。存储这个可执行文件的少数chunkserver被数百个同时请求重载。我们通过以更高的复制因子存储这些可执行文件，并通过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允许客户端在这种情况下从其他客户端读取数据。

2.6元数据

Master服务器(alex注:注意逻辑的Master节点和物理的Master服务器的区别。后续我们谈的是每个 Master服务器的行为，如存储、内存等等，因此我们将全部使用物理名称)存储3种主要类型的元数据， 包括:文件和Chunk的命名空间、文件和Chunk的对应关系、每个Chunk副本的存放地点。所有的元数据都保存在Master服务器的内存中。前两种类型的元数据(命名空间、文件和Chunk的对应关系)同时也会以记录变更日志的方式记录在操作系统的系统日志文件中，日志文件存储在本地磁盘上，同时日志会被复制到其它的远程Master服务器上。采用保存变更日志的方式，我们能够简单可靠的更新Master服务器 的状态，并且不用担心Master服务器崩溃导致数据不一致的风险。Master服务器不会持久保存Chunk位 置信息。Master服务器在启动时，或者有新的Chunk服务器加入时，向各个Chunk服务器轮询它们所存储的Chunk的信息。

2.6.1 内存中的数据结构

因为元数据保存在内存中，所以Master服务器的操作速度非常快。并且，Master服务器可以在后台简单 而高效的周期性扫描自己保存的全部状态信息。这种周期性的状态扫描也用于实现Chunk垃圾收集、在 Chunk服务器失效的时重新复制数据、通过Chunk的迁移实现跨Chunk服务器的负载均衡以及磁盘使用 状况统计等功能。4.3和4.4章节将深入讨论这些行为。

将元数据全部保存在内存中的方法有潜在问题:Chunk的数量以及整个系统的承载能力都受限于Master服务器所拥有的内存大小。但是在实际应用中，这并不是一个严重的问题。Master服务器只需要不到64个字节的元数据就能够管理一个64MB的Chunk。由于大多数文件都包含多个Chunk，因此绝大多数Chunk 都是满的，除了文件的最后一个Chunk是部分填充的。同样的，每个文件在命名空间中的数据大小通常在64字节以下，因为保存的文件名是用前缀压缩算法压缩过的。

即便是需要支持更大的文件系统，为Master服务器增加额外内存的费用是很少的，而通过增加有限的费用，我们就能够把元数据全部保存在内存里，增强了系统的简洁性、可靠性、高性能和灵活性。

2.6.2 Chunk位置信息

Master服务器并不持久化保存哪个Chunk服务器存有指定Chunk的副本的信息。Master服务器只是 在启动的时候轮询Chunk服务器以获取这些信息。Master服务器能够保证它持有的信息始终是最新的，因为它控制了所有的Chunk位置的分配，而且通过周期性的心跳信息监控Chunk服务器的状态。

最初设计时，我们试图把Chunk的位置信息持久的保存在Master服务器上，但是后来我们发现在启动的时 候轮询Chunk服务器，之后定期轮询更新的方式更简单。这种设计简化了在有Chunk服务器加入集群、离 开集群、更名、失效、以及重启的时候，Master服务器和Chunk服务器数据同步的问题。在一个拥有数百 台服务器的集群中，这类事件会频繁的发生。

可以从另外一个角度去理解这个设计决策:只有Chunk服务器才能最终确定一个Chunk是否在它的硬盘 上。我们从没有考虑过在Master服务器上维护一个这些信息的全局视图，因为Chunk服务器的错误可能会 导致Chunk自动消失(比如，硬盘损坏了或者无法访问了)，亦或者操作人员可能会重命名一个Chunk服务 器。

2.6.3操作日志

操作日志包含了关键的元数据变更历史记录。这对GFS非常重要。这不仅仅是因为操作日志是元数据唯一的持久化存储记录，它也作为判断同步操作顺序的逻辑时间基线(alex注:也就是通过逻辑日志的序号作为操作发生的逻辑时间，类似于事务系统中的LSN)。文件和Chunk，连同它们的版本(参考4.5节)，都由它们创建的逻辑时间唯一的、永久的标识。

操作日志非常重要，我们必须确保日志文件的完整，确保只有在元数据的变化被持久化后，日志才对客户端是可见的。否则，即使Chunk本身没有出现任何问题，我们仍有可能丢失整个文件系统，或者丢失客户端最近的操作。所以，我们会把日志复制到多台远程机器，并且只有把相应的日志记录写入到本地以及远程机器的硬盘后，才会响应客户端的操作请求。Master服务器会收集多个日志记录后批量处理，以减少写入磁盘和复制对系统整体性能的影响。

Master服务器在灾难恢复时，通过重演操作日志把文件系统恢复到最近的状态。为了缩短Master启动的时间，我们必须使日志足够小(alex注:即重演系统操作的日志量尽量的少)。Master服务器在日志增长 到一定量时对系统状态做一次Checkpoint(alex注:Checkpoint是一种行为，一种对数据库状态作一次 快照的行为)，将所有的状态数据写入一个Checkpoint文件(alex注:并删除之前的日志文件)。 在灾难恢复的时候，Master服务器就通过从磁盘上读取这个Checkpoint文件，以及重演Checkpoint之后的有 限个日志文件就能够恢复系统。Checkpoint文件以压缩B-树形势的数据结构存储，可以直接映射到内存，在用于命名空间查询时无需额外的解析。这大大提高了恢复速度，增强了可用性。

由于创建一个Checkpoint文件需要一定的时间，所以Master服务器的内部状态被组织为一种格式，这种 格式要确保在Checkpoint过程中不会阻塞正在进行的修改操作。Master服务器使用独立的线程切换到新 的日志文件和创建新的Checkpoint文件。新的Checkpoint文件包括切换前所有的修改。对于一个包含数 百万个文件的集群，创建一个Checkpoint文件需要1分钟左右的时间。创建完成后，Checkpoint文件会被写入在本地和远程的硬盘里。

Master服务器恢复只需要最新的Checkpoint文件和后续的日志文件。旧的Checkpoint文件和日志文件 可以被删除，但是为了应对灾难性的故障(alex注:catastrophes，数据备份相关文档中经常会遇到这个 词，表示一种超出预期范围的灾难性事件)，我们通常会多保存一些历史文件。Checkpoint失败不会对 正确性产生任何影响，因为恢复功能的代码可以检测并跳过没有完成的Checkpoint文件。

2.7一致性模型

GFS支持一个宽松的一致性模型，这个模型能够很好的支撑我们的高度分布的应用，同时还保持了相对简单且容易实现的优点。本节我们讨论GFS的一致性的保障机制，以及对应用程序的意义。我们也着重描述了GFS如何管理这些一致性保障机制，但是实现的细节将在本论文的其它部分讨论。

2.7.1 GFS一致性保障机制

文件命名空间的修改(例如，文件创建)是原子性的。它们仅由Master节点的控制:命名空间锁提供了原子性和正确性(4.1章)的保障;Master节点的操作日志定义了这些操作在全局的顺序(2.6.3章)。

数据修改后文件region(alex注:region这个词用中文非常难以表达，我认为应该是修改操作所涉及的文件中的某个范围)的状态取决于操作的类型、成功与否、以及是否同步修改。表1总结了各种操作的结果。如果所有客户端，无论从哪个副本读取，读到的数据都一样，那么我们认为文件region是“一致的”; 如果对文件的数据修改之后，region是一致的，并且客户端能够看到写入操作全部的内容，那么这个 region是“已定义的”。当一个数据修改操作成功执行，并且没有受到同时执行的其它写入操作的干扰，那么影响的region就是已定义的(隐含了一致性):所有的客户端都可以看到写入的内容。并行修改操作成功完成之后，region处于一致的、未定义的状态:所有的客户端看到同样的数据，但是无法读到任何一次 写入操作写入的数据。通常情况下，文件region内包含了来自多个修改操作的、混杂的数据片段。失败的修改操作导致一个region处于不一致状态(同时也是未定义的):不同的客户在不同的时间会看到不同的 数据。后面我们将描述应用如何区分已定义和未定义的region。应用程序没有必要再去细分未定义region 的不同类型。

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Table 1: File Region State After Mutation

数据修改操作分为写入或者记录追加两种。写入操作把数据写在应用程序指定的文件偏移位置上。即使有多个修改操作并行执行时，记录追加操作至少可以把数据原子性的追加到文件中一次，但是偏移位置是由 GFS选择的(3.3章)(alex注:这句话有点费解，其含义是所有的追加写入都会成功，但是有可能被执行了多次，而且每次追加的文件偏移量由GFS自己计算)。(相比而言，通常说的追加操作写的偏移位置 是文件的尾部。)GFS返回给客户端一个偏移量，表示了包含了写入记录的、已定义的region的起点。另外，GFS可能会在文件中间插入填充数据或者重复记录。这些数据占据的文件region被认定是不一致的， 这些数据通常比用户数据小的多。

经过了一系列的成功的修改操作之后，GFS确保被修改的文件region是已定义的，并且包含最后一次修改操作写入的数据。GFS通过以下措施确保上述行为:(a) 对Chunk的所有副本的修改操作顺序一致 (3.1章)，(b)使用Chunk的版本号来检测副本是否因为它所在的Chunk服务器宕机(4.5章)而错过了修改操作而导致其失效。失效的副本不会再进行任何修改操作，Master服务器也不再返回这个Chunk副本的位置信息给客户端。它们会被垃圾收集系统尽快回收。

由于Chunk位置信息会被客户端缓存，所以在信息刷新前，客户端有可能从一个失效的副本读取了数据。 在缓存的超时时间和文件下一次被打开的时间之间存在一个时间窗，文件再次被打开后会清除缓存中与该文件有关的所有Chunk位置信息。而且，由于我们的文件大多数都是只进行追加操作的，所以，一个失效的副本通常返回一个提前结束的Chunk而不是过期的数据。当一个Reader(alex注:本文中将用到两个 专有名词，Reader和Writer，分别表示执行GFS读取和写入操作的程序)重新尝试并联络Master服务器时，它就会立刻得到最新的Chunk位置信息。

即使在修改操作成功执行很长时间之后，组件的失效也可能损坏或者删除数据。GFS通过Master服务器和 所有Chunk服务器的定期“握手”来找到失效的Chunk服务器，并且使用Checksum来校验数据是否损坏 (5.2章)。一旦发现问题，数据要尽快利用有效的副本进行恢复(4.3章)。只有当一个Chunk的所有副本在GFS检测到错误并采取应对措施之前全部丢失，这个Chunk才会不可逆转的丢失。在一般情况下GFS 的反应时间(alex注:指Master节点检测到错误并采取应对措施)是几分钟。即使在这种情况下，Chunk 也只是不可用了，而不是损坏了:应用程序会收到明确的错误信息而不是损坏的数据。

2.7.2 程序的实现

使用GFS的应用程序可以利用一些简单技术实现这个宽松的一致性模型，这些技术也用来实现一些其它的目标功能，包括:尽量采用追加写入而不是覆盖，Checkpoint，自验证的写入操作，自标识的记录。

在实际应用中，我们所有的应用程序对文件的写入操作都是尽量采用数据追加方式，而不是覆盖方式。一种典型的应用，应用程序从头到尾写入数据，生成了一个文件。写入所有数据之后，应用程序自动将文件 改名为一个永久保存的文件名，或者周期性的作Checkpoint，记录成功写入了多少数据。Checkpoint文 件可以包含程序级别的校验和。Readers仅校验并处理上个Checkpoint之后产生的文件region，这些文 件region的状态一定是已定义的。这个方法满足了我们一致性和并发处理的要求。追加写入比随机位置写 入更加有效率，对应用程序的失败处理更具有弹性。Checkpoint可以让Writer以渐进的方式重新开始， 并且可以防止Reader处理已经被成功写入，但是从应用程序的角度来看还并未完成的数据。

我们再来分析另一种典型的应用。许多应用程序并行的追加数据到同一个文件，比如进行结果的合并或者是一个生产者-消费者队列。记录追加方式的“至少一次追加”的特性保证了Writer的输出。Readers使用下面的方法来处理偶然性的填充数据和重复内容。Writers在每条写入的记录中都包含了额外的信息，例如 Checksum，用来验证它的有效性。Reader可以利用Checksum识别和抛弃额外的填充数据和记录片 段。如果应用不能容忍偶尔的重复内容(比如，如果这些重复数据触发了非幂等操作)，可以用记录的唯一 标识符来过滤它们，这些唯一标识符通常用于命名程序中处理的实体对象，例如web文档。这些记录I/O功 能(alex注:These functionalities for record I/O)(除了剔除重复数据)都包含在我们的程序共享的 库中，并且适用于Google内部的其它的文件接口实现。所以，相同序列的记录，加上一些偶尔出现的重复 数据，都被分发到Reader了。

3. 系统交互

我们在设计这个系统时，一个重要的原则是最小化所有操作和Master节点的交互。带着这样的设计理念， 我们现在描述一下客户机、Master服务器和Chunk服务器如何进行交互，以实现数据修改操作、原子的记录追加操作以及快照功能。

3.1 租约(lease)和变更顺序

变更是一个会改变Chunk内容或者元数据的操作，比如写入操作或者记录追加操作。变更操作会在Chunk 的所有副本上执行。我们使用租约(lease)机制来保持多个副本间变更顺序的一致性。Master节点为 Chunk的一个副本建立一个租约，我们把这个副本叫做主Chunk。主Chunk对Chunk的所有更改操作进行序列化。所有的副本都遵从这个序列进行修改操作。因此，修改操作全局的顺序首先由Master节点选择 的租约的顺序决定，然后由租约中主Chunk分配的序列号决定。

设计租约机制的目的是为了最小化Master节点的管理负担。租约的初始超时设置为60秒。不过，只要 Chunk被修改了，主Chunk就可以申请更长的租期，通常会得到Master节点的确认并收到租约延长的时间。这些租约延长请求和批准的信息通常都是附加在Master节点和Chunk服务器之间的心跳消息中来传递。有时Master节点会试图提前取消租约(例如，Master节点想取消在一个已经被改名的文件上的修改 操作)。即使Master节点和主Chunk失去联系，它仍然可以安全地在旧的租约到期后和另外一个Chunk 副本签订新的租约。 在图2中，我们依据步骤编号，展现写入操作的控制流程。

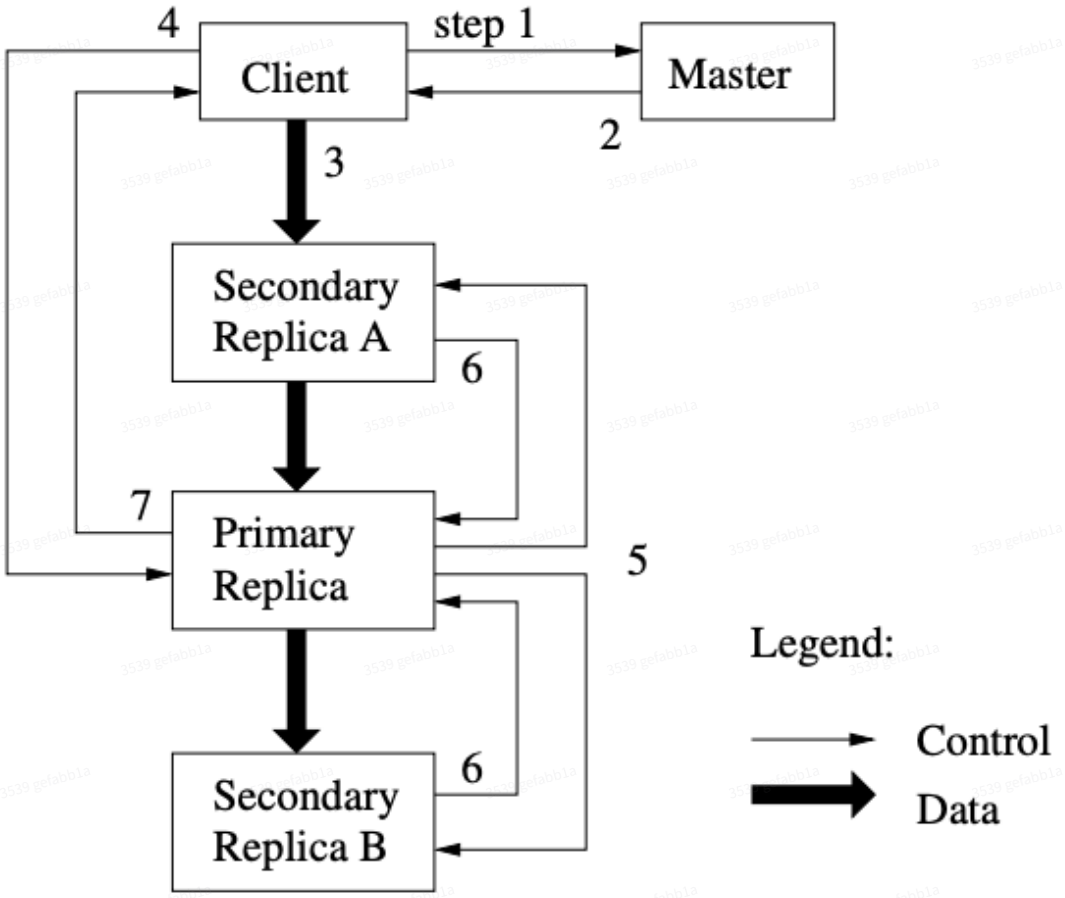


Figure 2: Write Control and Data Flow

1. 客户机向Master节点询问哪一个Chunk服务器持有当前的租约，以及其它副本的位置。如果没有一 个Chunk持有租约，Master节点就选择其中一个副本建立一个租约(这个步骤在图上没有显示)。

2. Master节点将主Chunk的标识符以及其它副本(又称为secondary副本、二级副本)的位置返回给 客户机。客户机缓存这些数据以便后续的操作。只有在主Chunk不可用，或者主Chunk回复信息表明它已不再持有租约的时候，客户机才需要重新跟Master节点联系。
3. 客户机把数据推送到所有的副本上。客户机可以以任意的顺序推送数据。Chunk服务器接收到数据并保存在它的内部LRU缓存中，一直到数据被使用或者过期交换出去。由于数据流的网络传输负载非常高，通过分离数据流和控制流，我们可以基于网络拓扑情况对数据流进行规划，提高系统性能，而不用去理会哪个Chunk服务器保存了主Chunk。3.2章节会进一步讨论这点。
4. 当所有的副本都确认接收到了数据，客户机发送写请求到主Chunk服务器。这个请求标识了早前推送到所有副本的数据。主Chunk为接收到的所有操作分配连续的序列号，这些操作可能来自不同的客户机，序列号保证了操作顺序执行。它以序列号的顺序把操作应用到它自己的本地状态中(alex注:也就是在本地执行这些操作，这句话按字面翻译有点费解，也许应该翻译为“它顺序执行这些操作，并更新自己的状态”)
5. 主Chunk把写请求传递到所有的二级副本。每个二级副本依照主Chunk分配的序列号以相同的顺序执行这些操作。
6. 所有的二级副本回复主Chunk，它们已经完成了操作
7. 主Chunk服务器(alex注:即主Chunk所在的Chunk服务器)回复客户机。任何副本产生的任何错误都会返回给客户机。在出现错误的情况下，写入操作可能在主Chunk和一些二级副本执行成 功。(如果操作在主Chunk上失败了，操作就不会被分配序列号，也不会被传递。)客户端的请求 被确认为失败，被修改的region处于不一致的状态。我们的客户机代码通过重复执行失败的操作来处理这样的错误。在从头开始重复执行之前，客户机会先从步骤(3)到步骤(7)做几次尝试。

如果应用程序一次写入的数据量很大，或者数据跨越了多个Chunk，GFS客户机代码会把它们分成多个写 操作。这些操作都遵循前面描述的控制流程，但是可能会被其它客户机上同时进行的操作打断或者覆盖。 因此，共享的文件region的尾部可能包含来自不同客户机的数据片段，尽管如此，由于这些分解后的写入 操作在所有的副本上都以相同的顺序执行完成，Chunk的所有副本都是一致的。这使文件region处于2.7 节描述的一致的、但是未定义的状态。

3.2数据流

为了提高网络效率，我们采取了把数据流和控制流分开的措施。在控制流从客户机到主Chunk、然后再到所有二级副本的同时，数据以管道的方式，顺序的沿着一个精心选择的Chunk服务器链推送。我们的目标是充分利用每台机器的带宽，避免网络瓶颈和高延时的连接，最小化推送所有数据的延时。

为了充分利用每台机器的带宽，数据沿着一个Chunk服务器链顺序的推送，而不是以其它拓扑形式分散推送(例如，树型拓扑结构)。线性推送模式下，每台机器所有的出口带宽都用于以最快的速度传输数据， 而不是在多个接受者之间分配带宽。

为了尽可能的避免出现网络瓶颈和高延迟的链接(eg，inter-switch最有可能出现类似问题)，每台机器都尽量的在网络拓扑中选择一台还没有接收到数据的、离自己最近的机器作为目标推送数据。假设客户机 把数据从Chunk服务器S1推送到S4。它把数据推送到最近的Chunk服务器S1。S1把数据推送到S2，因 为S2和S4中最接近的机器是S2。同样的，S2把数据传递给S3和S4之间更近的机器，依次类推推送下 去。我们的网络拓扑非常简单，通过IP地址就可以计算出节点的“距离”。

最后，我们利用基于TCP连接的、管道式数据推送方式来最小化延迟。Chunk服务器接收到数据后，马上开始向前推送。管道方式的数据推送对我们帮助很大，因为我们采用全双工的交换网络。接收到数据后立刻向前推送不会降低接收的速度。在没有网络拥塞的情况下，传送B字节的数据到R个副本的理想时间是 B/T+RL ，T是网络的吞吐量，L是在两台机器数据传输的延迟。通常情况下，我们的网络连接速度是 100Mbps(T)，L将远小于1ms。因此，1MB的数据在理想情况下80ms左右就能分发出去。

3.3原子记录附加

GFS提供了一种原子的数据追加操作–记录追加。传统方式的写入操作，客户程序会指定数据写入的偏移量。对同一个region的并行写入操作不是串行的:region尾部可能会包含多个不同客户机写入的数据片段。使用记录追加，客户机只需要指定要写入的数据。GFS保证至少有一次原子的写入操作成功执行(即 写入一个顺序的byte流)，写入的数据追加到GFS指定的偏移位置上，之后GFS返回这个偏移量给客户机。这类似于在Unix操作系统编程环境中，对以O_APPEND模式打开的文件，多个并发写操作在没有竞态条件时的行为。

记录追加在我们的分布应用中非常频繁的使用，在这些分布式应用中，通常有很多的客户机并行地对同一个文件追加写入数据。如果我们采用传统方式的文件写入操作，客户机需要额外的复杂、昂贵的同步机制，例如使用一个分布式的锁管理器。在我们的工作中，这样的文件通常用于多个生产者/单一消费者的队列系统，或者是合并了来自多个客户机的数据的结果文件。

记录追加是一种修改操作，它也遵循3.1节描述的控制流程，除了在主Chunk有些额外的控制逻辑。客户机把数据推送给文件最后一个Chunk的所有副本，之后发送请求给主Chunk。主Chunk会检查这次记录追加操作是否会使Chunk超过最大尺寸(64MB)。如果超过了最大尺寸，主Chunk首先将当前Chunk填充到最大尺寸，之后通知所有二级副本做同样的操作，然后回复客户机要求其对下一个Chunk重新进行记 录追加操作。(记录追加的数据大小严格控制在Chunk最大尺寸的1/4，这样即使在最坏情况下，数据碎片的数量仍然在可控的范围。)通常情况下追加的记录不超过Chunk的最大尺寸，主Chunk把数据追加到自己的副本内，然后通知二级副本把数据写在跟主Chunk一样的位置上，最后回复客户机操作成功。

如果记录追加操作在任何一个副本上失败了，客户端就需要重新进行操作。重新进行记录追加的结果是， 同一个Chunk的不同副本可能包含不同的数据–重复包含一个记录全部或者部分的数据。GFS并不保证 Chunk的所有副本在字节级别是完全一致的。它只保证数据作为一个整体原子的被至少写入一次。这个特性可以通过简单观察推导出来:如果操作成功执行，数据一定已经写入到Chunk的所有副本的相同偏移位 置上。这之后，所有的副本至少都到了记录尾部的长度，任何后续的记录都会追加到更大的偏移地址，或者是不同的Chunk上，即使其它的Chunk副本被Master节点选为了主Chunk。就我们的一致性保障模型而言，记录追加操作成功写入数据的region是已定义的(因此也是一

致的)，反之则是不一致的(因此也就是未定义的)。正如我们在2.7.2节讨论的，我们的程序可以处理不一致的区域。

3.4快照

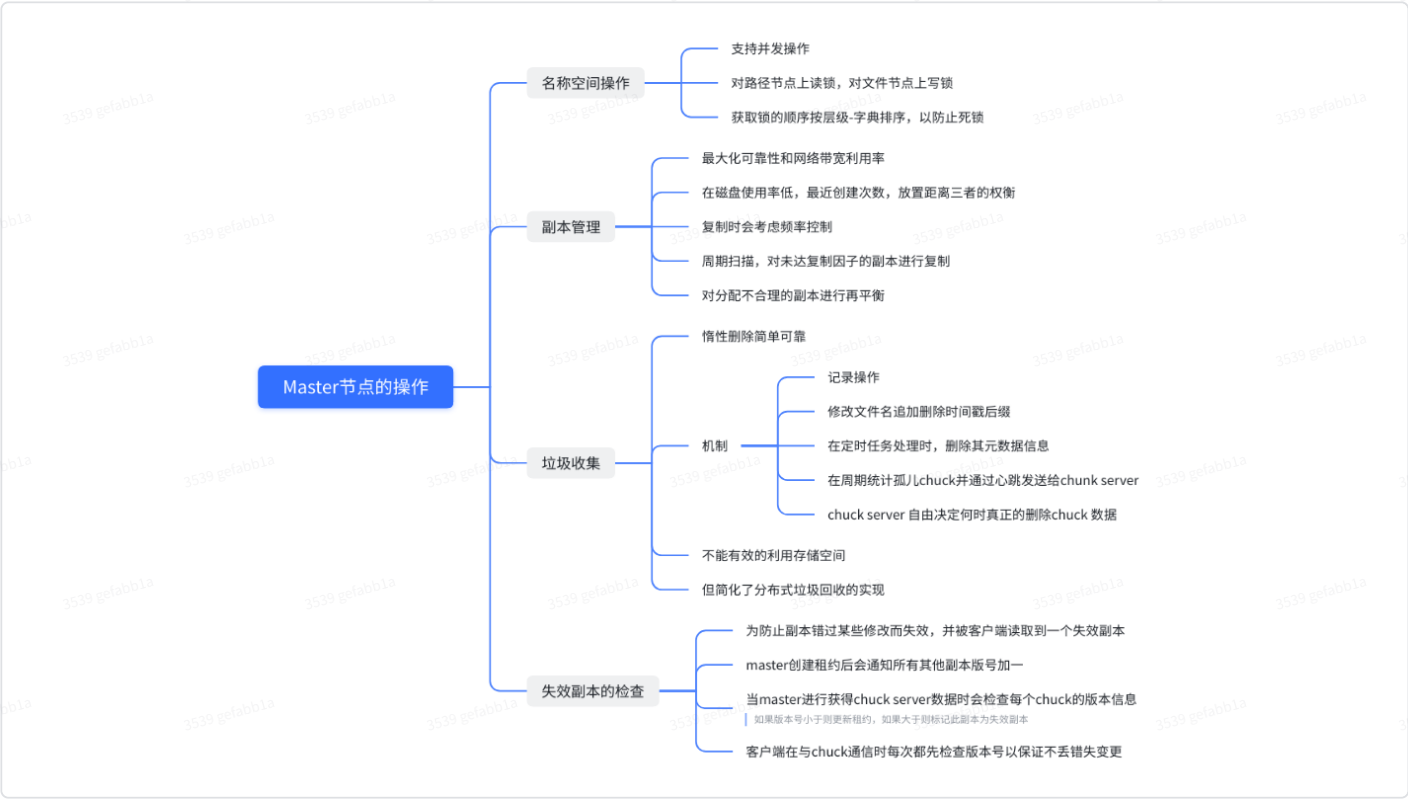
快照操作几乎可以瞬间完成对一个文件或者目录树(“源”)做一个拷贝，并且几乎不会对正在进行的其它操作造成任何干扰。我们的用户可以使用快照迅速的创建一个巨大的数据集的分支拷贝(而且经常是递归的拷贝)，或者是在做实验性的数据操作之前，使用快照操作备份当前状态，这样之后就可以轻松的提交或者回滚到备份时的状态。

就像AFS(alex注:AFS，即Andrew File System，一种分布式文件系统)，我们用标准的copy-on-write技术实现快照。当Master节点收到一个快照请求，它首先取消作快照的文件的所有Chunk的租约。 这个措施保证了后续对这些Chunk的写操作都必须与Master交互以找到租约持有者。这就给Master 节点一个率先创建Chunk的新拷贝的机会。

租约取消或者过期之后，Master节点把这个操作以日志的方式记录到硬盘上。然后，Master节点通过复制源文件或者目录的元数据的方式，把这条日志记录的变化反映到保存在内存的状态中。新创建的快照文件和源文件指向完全相同的Chunk地址。

在快照操作之后，当客户机第一次想写入数据到Chunk C，它首先会发送一个请求到Master节点查询当前的租约持有者。Master节点注意到Chunke C的引用计数超过了1(alex注:不太明白为什么会大于1.难道 是Snapshot没有释放引用计数?)。Master节点不会马上回复客户机的请求，而是选择一个新的Chunk 句柄C`。之后，Master节点要求每个拥有Chunk C当前副本的Chunk服务器创建一个叫做C`的新 Chunk。通过在源Chunk所在Chunk服务器上创建新的Chunk，我们确保数据在本地而不是通过网络复制(我们的硬盘比我们的100Mb以太网大约快3倍)。从这点来讲，请求的处理方式和任何其它Chunk没 什么不同:Master节点确保新Chunk C`的一个副本拥有租约，之后回复客户机，客户机得到回复后就可 以正常的写这个Chunk，而不必理会它是从一个已存在的Chunk克隆出来的。

4.Master 节点的操作



Master节点执行所有的名称空间操作。此外，它还管理着整个系统里所有Chunk的副本:它决定Chunk 的存储位置，创建新Chunk和它的副本，协调各种各样的系统活动以保证Chunk被完全复制，在所有的 Chunk服务器之间的进行负载均衡，回收不再使用的存储空间。本节我们讨论上述的主题。

4.1命名空间管理与锁定

Master节点的很多操作会花费很长的时间:比如，快照操作必须取消Chunk服务器上快照所涉及的所有的 Chunk的租约。我们不希望在这些操作的运行时，延缓了其它的Master节点的操作。因此，我们允许多个 操作同时进行，使用名称空间的region上的锁来保证执行的正确顺序。

不同于许多传统文件系统，GFS没有针对每个目录实现能够列出目录下所有文件的数据结构。GFS也不支持文件或者目录的链接(即Unix术语中的硬链接或者符号链接)。在逻辑上，GFS的名称空间就是一个全 路径和元数据映射关系的查找表。利用前缀压缩，这个表可以高效的存储在内存中。在存储名称空间的树 型结构上，每个节点(绝对路径的文件名或绝对路径的目录名)都有一个关联的读写锁。

每个Master节点的操作在开始之前都要获得一系列的锁。通常情况下，如果一个操作涉及/d1/d2/.../dn /leaf，那么操作首先要获得目录/d1，/d1/d2，...，/d1/d2/.../dn的读锁，以及/d1/d2/.../dn/leaf的读 写锁。注意，根据操作的不同，leaf可以是一个文件，也可以是一个目录。

现在，我们演示一下在/home/user被快照到/save/user的时候，锁机制如何防止创建文件/home/user /foo。快照操作获取/home和/save的读取锁，以及/home/user和/save/user的写入锁。文件创建操作获 得/home和/home/user的读取锁，以及/home/user/foo的写入锁。这两个操作要顺序执行，因为它们试 图获取的/home/user的锁是相互冲突。文件创建操作不需要获取父目录的

写入锁，因为这里没有”目 录”，或者类似inode等用来禁止修改的数据结构。文件名的读取锁足以防止父目录被删除。

采用这种锁方案的优点是支持对同一目录的并行操作。比如，可以再同一个目录下同时创建多个文件:每 一个操作都获取一个目录名的上的读取锁和文件名上的写入锁。目录名的读取锁足以的防止目录被删除、 改名以及被快照。文件名的写入锁序列化文件创建操作，确保不会多次创建同名的文件。

因为名称空间可能有很多节点，读写锁采用惰性分配策略，在不再使用的时候立刻被删除。同样，锁的获 取也要依据一个全局一致的顺序来避免死锁:首先按名称空间的层次排序，在同一个层次内按字典顺序排序。

4.2 副本的位置

GFS集群是高度分布的多层布局结构，而不是平面结构。典型的拓扑结构是有数百个Chunk服务器安装在许多机架上。Chunk服务器被来自同一或者不同机架上的数百个客户机轮流访问。不同机架上的两台机器 间的通讯可能跨越一个或多个网络交换机。另外，机架的出入带宽可能比机架内所有机器加和在一起的带 宽要小。多层分布架构对数据的灵活性、可靠性以及可用性方面提出特有的挑战。

Chunk副本位置选择的策略服务两大目标:最大化数据可靠性和可用性，最大化网络带宽利用率。为了实 现这两个目的，仅仅是在多台机器上分别存储这些副本是不够的，这只能预防硬盘损坏或者机器失效带来 的影响，以及最大化每台机器的网络带宽利用率。我们必须在多个机架间分布储存Chunk的副本。这保证 Chunk的一些副本在整个机架被破坏或掉线(比如，共享资源，如电源或者网络交换机造成的问题)的情 况下依然存在且保持可用状态。这还意味着在网络流量方面，尤其是针对Chunk的读操作，能够有效利用 多个机架的整合带宽。另一方面，写操作必须和多个机架上的设备进行网络通信，但是这个代价是我们愿 意付出的。

4.3 创建，重新复制，重新负载均衡

Chunk的副本有三个用途:Chunk创建，重新复制和重新负载均衡。

当Master节点创建一个Chunk时，它会选择在哪里放置初始的空的副本。Master节点会考虑几个因 素。(1)我们希望在低于平均硬盘使用率的Chunk服务器上存储新的副本。这样的做法最终能够平衡 Chunk服务器之间的硬盘使用率。(2)我们希望限制在每个Chunk服务器上”最近”的Chunk创建操作的次数。虽然创建操作本身是廉价的，但是创建操作也意味着随之会有大量的写入数据的操作，因为Chunk 在Writer真正写入数据的时候才被创建，而在我们的”追加一次，读取多次”的工作模式下，Chunk一旦写入成功之后就会变为只读的了。(3)如上所述，我们希望把Chunk的副本分布在多个机架之间。

当Chunk的有效副本数量少于用户指定的复制因数的时候，Master节点会重新复制它。这可能是由几个原 因引起的:一个Chunk服务器不可用了，Chunk服务器报告它所存储的一个副本损坏了，Chunk服务器 的一个磁盘因为错误不可用了，或者Chunk副本的复制因数提高了。每个需要被重新复制的Chunk都会根 据几个因素进行排序。一个因素是Chunk现有副本数量和复制因数相差多少。例如，丢失两个副本的 Chunk比丢失一个副本的Chunk有更高的优先级。另外，我们优先重新复制活跃(live)文件的Chunk而 不是最近刚被删除的文件的Chunk(查看4.4节)。最后，为了最小化失效的Chunk对正在运行的应用程序的影响，我们提高会阻塞客户机程序处理流程的Chunk的优先级。

Master节点选择优先级最高的Chunk，然后命令某个Chunk服务器直接从可用的副本”克隆”一个副本出 来。选择新副本的位置的策略和创建时类似:平衡硬盘使用率、限制同一台Chunk服务器上的正在进行的 克隆操作的数量、在机架间分布副本。为了防止克隆产生的网络流量大大超过客户机的流量，Master节点 对整个集群和每个Chunk服务器上的同时进行的克隆操作的数量都进行了限制。另外，Chunk服务器通过 调节它对源Chunk服务器读请求的频率来限制它用于克隆操作的带宽。

最后，Master服务器周期性地对副本进行重新负载均衡:它检查当前的副本分布情况，然后移动副本以便 更好的利用硬盘空间、更有效的进行负载均衡。而且在这个过程中，Master服务器逐渐的填满一个新的 Chunk服务器，而不是在短时间内用新的Chunk填满它，以至于过载。新副本的存储位置选择策略和上面 讨论的相同。另外，Master节点必须选择哪个副本要被移走。通常情况，Master节点移走那些剩余空间 低于平均值的Chunk服务器上的副本，从而平衡系统整体的硬盘使用率。

4.4垃圾收集

GFS在文件删除后不会立刻回收可用的物理空间。GFS空间回收采用惰性的策略，只在文件和Chunk级的常规垃圾收集时进行。我们发现这个方法使系统更简单、更可靠。

4.4.1机制

当一个文件被应用程序删除时，Master节点象对待其它修改操作一样，立刻把删除操作以日志的方式记录 下来。但是，Master节点并不马上回收资源，而是把文件名改为一个包含删除时间戳的、隐藏的名字。当 Master节点对文件系统命名空间做常规扫描的时候，它会删除所有三天前的隐藏文件(这个时间间隔是可 以设置的)。直到文件被真正删除，它们仍旧可以用新的特殊的名字读取，也可以通过把隐藏文件改名为 正常显示的文件名的方式“反删除”。当隐藏文件被从名称空间中删除，Master服务器内存中保存的这个文件的相关元数据才会被删除。这也有效的切断了文件和它包含的所有Chunk的连接(alex注:原文是This effectively severs its links to all its chunks)。

在对Chunk名字空间做类似的常规扫描时，Master节点找到孤儿Chunk(不被任何文件包含的Chunk) 并删除它们的元数据。Chunk服务器在和Master节点交互的心跳信息中，报告它拥有的Chunk子集的信 息，Master节点回复Chunk服务器哪些Chunk在Master节点保存的元数据中已经不存在了。Chunk服务 器可以任意删除这些Chunk的副本。

4.4.2讨论

虽然分布式垃圾回收在编程语言领域是一个需要复杂的方案才能解决的难题，但是在GFS系统中是非常简单的。我们可以轻易的得到Chunk的所有引用:它们都只存储在Master服务器上的文件到块的映射表中。 我们也可以很轻易的得到所有Chunk的副本:它们都以Linux文件的形式存储在Chunk服务器的指定目录下。所有Master节点不能识别的副本都是”垃圾”。

垃圾回收在空间回收方面相比直接删除有几个优势。首先，对于组件失效是常态的大规模分布式系统，垃圾回收方式简单可靠。Chunk可能在某些Chunk服务器创建成功，某些Chunk服务器上创建失败，失败的副本处于无法被Master节点识别的状态。副本删除消息可能丢失，Master节点必须重新发送失败的删除消息，包括自身的和Chunk服务器的(alex注:自身的指删除metadata的消息)。垃圾回收提供了一致的、可靠的清除无用副本的方法。第二，垃圾回收把存储空间的回收操作合并到Master节点规律性的后台活动中，比如，例行扫描和与Chunk服务器握手等。因此，操作被批量的执

行，开销会被分散。另外， 垃圾回收在Master节点相对空闲的时候完成。这样Master节点就可以给那些需要快速反应的客户机请求 提供更快捷的响应。第三，延缓存储空间回收为意外的、不可逆转的删除操作提供了安全保障。

根据我们的使用经验，延迟回收空间的主要问题是，延迟回收会阻碍用户调优存储空间的使用，特别是当存储空间比较紧缺的时候。当应用程序重复创建和删除临时文件时，释放的存储空间不能马上重用。我们通过显式的再次删除一个已经被删除的文件的方式加速空间回收的速度。我们允许用户为命名空间的不同部分设定不同的复制和回收策略。例如，用户可以指定某些目录树下面的文件不做复制，删除的文件被即时的、不可恢复的从文件系统移除。

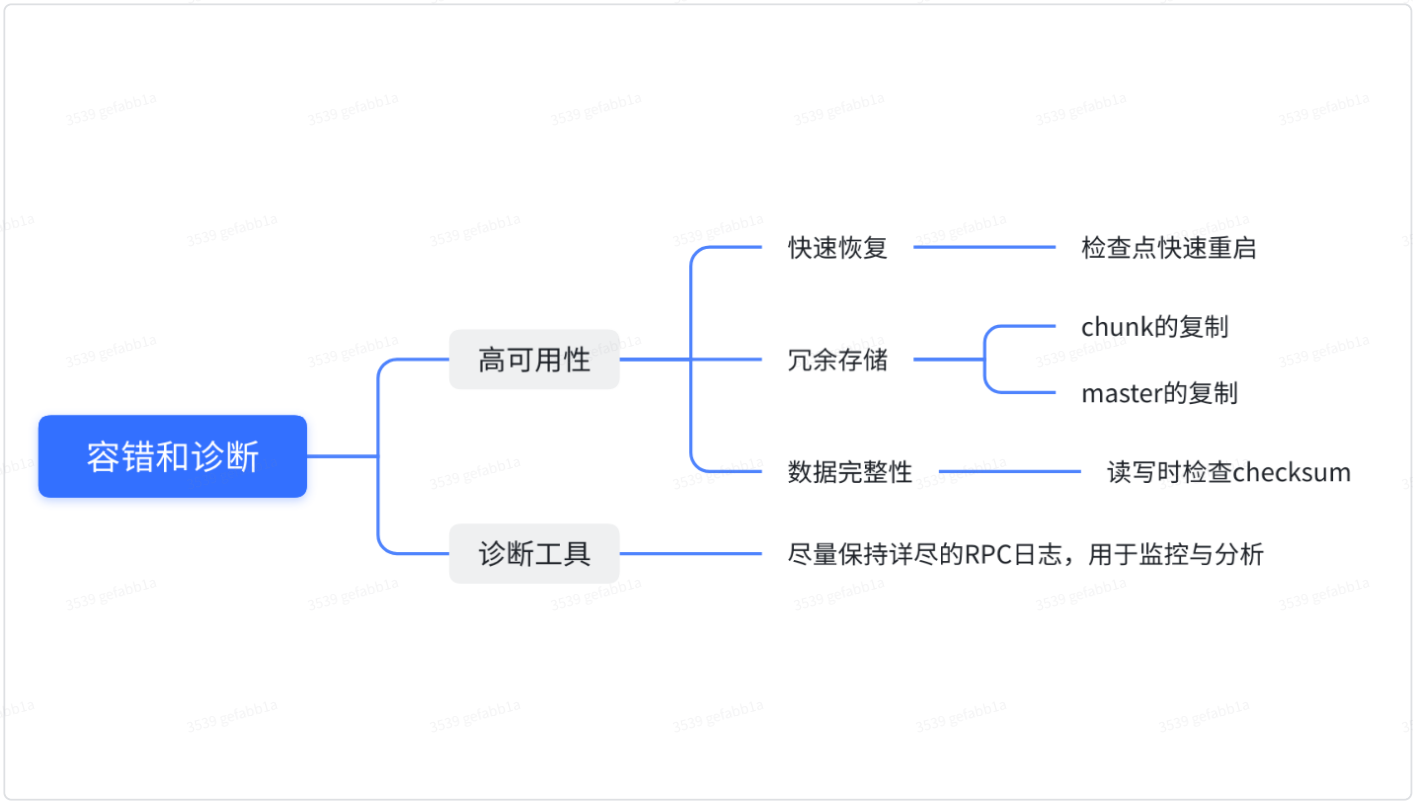
4.5 过期失效的副本检测

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

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

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

5.容错和诊断



我们在设计GFS时遇到的最大挑战之一是如何处理频繁发生的组件失效。组件的数量和质量让这些问题出现的频率远远超过一般系统意外发生的频率:我们不能完全依赖机器的稳定性，也不能完全相信硬盘的可靠性。组件的失效可能造成系统不可用，更糟糕的是，还可能产生不完整的数据。我们讨论我们如何面对这些挑战，以及当组件失效不可避免的发生时，用GFS自带工具诊断系统故障。

5.1高可用性

在GFS集群的数百个服务器之中，在任何给定的时间必定会有些服务器是不可用的。我们使用两条简单但是有效的策略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

不管Master服务器和Chunk服务器是如何关闭的，它们都被设计为可以在数秒钟内恢复它们的状态并重新启动。事实上，我们并不区分正常关闭和异常关闭;通常，我们通过直接kill掉进程来关闭服务器。客户机 和其它的服务器会感觉到系统有点颠簸(alex注:a minor hiccup)，正在发出的请求会超时，需要重新连 接到重启后的服务器，然后重试这个请求。6.6.2章节记录了实测的启动时间。

5.1.2 chunk复制

正如之前讨论的，每个Chunk都被复制到不同机架上的不同的Chunk服务器上。用户可以为文件命名空间的不同部分设定不同的复制级别。缺省是3。当有Chunk服务器离线了，或者通过Chksum校验(参考 5.2节)发现了已经损坏的数据，Master节点通过克隆已有的副本保证每个Chunk都被完整复制(alex 注:即每个Chunk都有复制因子制定的个数个副本，缺省是3)。虽然Chunk复制策略对我们非常有效， 但是我们也在寻找其它形式的跨服务器的冗余解决方案，比如使用奇偶校验、或者Erasure codes(alex 注:Erasure codes用来解决链接层中不相关的错误，以及网络拥塞和buffer限制造成的丢包错误)来解决我们日益增长的只读存储需求。我们的系统主要的工作负载是追加方式的写入和读取操

作，很少有随机 的写入操作，因此，我们认为在我们这个高度解耦合的系统架构下实现这些复杂的冗余方案很有挑战性， 但并非不可实现。

5.1.3 Master服务器的复制

为了保证Master服务器的可靠性，Master服务器的状态也要复制。Master服务器所有的操作日志和 checkpoint文件都被复制到多台机器上。对Master服务器状态的修改操作能够提交成功的前提是，操作日志写入到Master服务器的备节点和本机的磁盘。简单说来，一个Master服务进程负责所有的修改操作，包括后台的服务，比如垃圾回收等改变系统内部状态活动。当它失效的时，几乎可以立刻重新启动。 如果Master进程所在的机器或者磁盘失效了，处于GFS系统外部的监控进程会在其它的存有完整操作日志的机器上启动一个新的Master进程。客户端使用规范的名字访问Master(比如gfs-test)节点，这个名字 类似DNS别名，因此也就可以在Master进程转到别的机器上执行时，通过更改别名的实际指向访问新的 Master节点。

此外，GFS中还有些“影子”Master服务器，这些“影子”服务器在“主”Master服务器宕机的时候提供文件 系统的只读访问。它们是影子，而不是镜像，所以它们的数据可能比“主”Master服务器更新要慢，通常是不到1秒。对于那些不经常改变的文件、或者那些允许获取的数据有少量过期的应用程序，“影子”Master 服务器能够提高读取的效率。事实上，因为文件内容是从Chunk服务器上读取的，因此，应用程序不会发 现过期的文件内容。在这个短暂的时间窗内，过期的可能是文件的元数据，比如目录的内容或者访问控制 信息。

“影子”Master服务器为了保持自身状态是最新的，它会读取一份当前正在进行的操作的日志副本，并且依照和主Master服务器完全相同的顺序来更改内部的数据结构。和主Master服务器一样，“影子”Master服务器在启动的时候也会从Chunk服务器轮询数据(之后定期拉数据)，数据中包括了Chunk副本的位置信息;“影子”Master服务器也会定期和Chunk服务器“握手”来确定它们的状态。在主Master服务器因创建 和删除副本导致副本位置信息更新时，“影子”Master服务器才和主Master服务器通信来更新自身状态。

5.2数据完整性

每个Chunk服务器都使用checksum来检查保存的数据是否损坏。考虑到一个GFS集群通常都有好几百台 机器、几千块硬盘，磁盘损坏导致数据在读写过程中损坏或者丢失是非常常见的(第7节讲了一个原 因)。我们可以通过别的Chunk副本来解决数据损坏问题，但是跨越Chunk服务器比较副本来检查数据是 否损坏很不实际。另外，GFS允许有歧义的副本存在:GFS修改操作的语义，特别是早先讨论过的原子纪录追加的操作，并不保证副本完全相同(alex注:副本不是byte-wise完全一致的)。因此，每个Chunk服务器必须独立维护Checksum来校验自己的副本的完整性。

我们把每个Chunk都分成64KB大小的块。每个块都对应一个32位的Checksum。和其它元数据一 样，Checksum与其它的用户数据是分开的，并且保存在内存和硬盘上，同时也记录操作日志。

对于读操作来说，在把数据返回给客户端或者其它的Chunk服务器之前，Chunk服务器会校验读取操作涉及的范围内的块的Checksum。因此Chunk服务器不会把错误数据传递到其它的机器上。如果发生某个块 的Checksum不正确，Chunk服务器返回给请求者一个错误信息，并且通知Master服务器这个错误。作为回应，请求者应当从其它副本读取数据，Master服务器也会从其它副本克隆数据进行恢复。当一个新的 副本就绪后，Master服务器通知副本错误的Chunk服务器删掉错误的副本。

Checksum对读操作的性能影响很小，可以基于几个原因来分析一下。因为大部分的读操作都至少要读取 几个块，而我们只需要读取一小部分额外的相关数据进行校验。GFS客户端代码通过每次把读取操作都对 齐在Checksum block的边界上，进一步减少了这些额外的读取操作的负面影响。另外，在Chunk服务器 上，Chunksum的查找和比较不需要I/O操作，Checksum的计算可以和I/O操作同时进行。

Checksum的计算针对在Chunk尾部的追加写入操作作了高度优化(与之对应的是覆盖现有数据的写入操作)，因为这类操作在我们的工作中占了很大比例。我们只增量更新最后一个不完整的块的Checksum， 并且用所有的追加来的新Checksum块来计算新的Checksum。即使是最后一个不完整的Checksum块 已经损坏了，而且我们不能够马上检查出来，由于新的Checksum和已有数据不吻合，在下次对这个块进 行读取操作的时候，会检查出数据已经损坏了。

相比之下，如果写操作覆盖已经存在的一个范围内的Chunk，我们必须读取和校验被覆盖的第一个和最后 一个块，然后再执行写操作;操作完成之后再重新计算和写入新的Checksum。如果我们不校验第一个和 最后一个被写的块，那么新的Checksum可能会隐藏没有被覆盖区域内的数据错误。

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

5.3诊断工具

详尽的、深入细节的诊断日志，在问题隔离、调试、以及性能分析等方面给我们带来无法估量的帮助，同 时也只需要很小的开销。没有日志的帮助，我们很难理解短暂的、不重复的机器之间的消息交互。GFS的 服务器会产生大量的日志，记录了大量关键的事件(比如，Chunk服务器启动和关闭)以及所有的RPC的 请求和回复。这些诊断日志可以随意删除，对系统的正确运行不造成任何影响。然而，我们在存储空间允 许的情况下会尽量的保存这些日志。

RPC日志包含了网络上发生的所有请求和响应的详细记录，但是不包括读写的文件数据。通过匹配请求与 回应，以及收集不同机器上的RPC日志记录，我们可以重演所有的消息交互来诊断问题。日志还用来跟踪 负载测试和性能分析。

日志对性能的影响很小(远小于它带来的好处)，因为这些日志的写入方式是顺序的、异步的。最近发生 的事件日志保存在内存中，可用于持续不断的在线监控。

6.数据分析

本节中，我们将使用一些小规模基准测试来展现GFS系统架构和实现上的一些固有瓶颈，还有些来自Google内部使用的真实的GFS集群的基准数据。

6.1 小规模基准测试

我们在一个包含1台Master服务器，2台Master服务器复制节点，16台Chunk服务器和16个客户机组成的GFS集群上测量性能。注意，采用这样的集群配置方案只是为了易于测试。典型的GFS集群

有数百个 Chunk服务器和数百个客户机。

所有机器的配置都一样:两个PIII 1.4GHz处理器，2GB内存，两个80G/5400rpm的硬盘，以及100Mbps全双工以太网连接到一个HP2524交换机。GFS集群中所有的19台服务器都连接在一个交换机，所有16台客户机连接到另一个交换机上。两个交换机之间使用1Gbps的线路连接。

6.1.1 读取

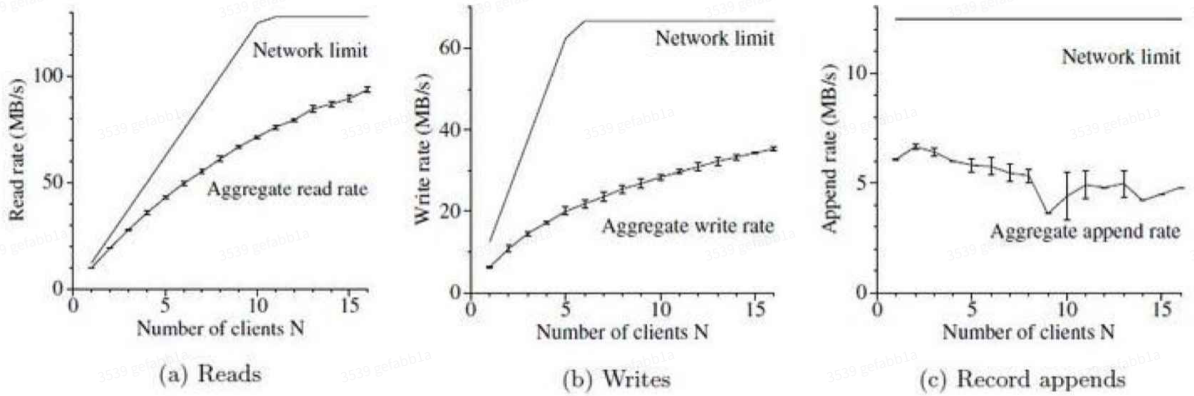


Figure 3: Aggregate Throughputs.

N个客户机从GFS文件系统同步读取数据。每个客户机从320GB的文件集合中随机读取4MB region的内容。读取操作重复执行256次，因此，每个客户机最终都读取1GB的数据。所有的Chunk服务器加起来总 共只有32GB的内存，因此，我们预期只有最多10%的读取请求命中Linux的文件系统缓冲。我们的测试 结果应该和一个在没有文件系统缓存的情况下读取测试的结果接近。

图三:合计吞吐量:上边的曲线显示了我们网络拓扑下的合计理论吞吐量上限。下边的曲线显示了观测到 的吞吐量。这个曲线有着95%的可靠性，因为有时候测量会不够精确。 图3(a)显示了N个客户机整体的读取速度以及这个速度的理论极限。当连接两个交换机的1Gbps的链路 饱和时，整体读取速度达到理论的极限值是125MB/S，或者说每个客户机配置的100Mbps网卡达到饱和 时，每个客户机读取速度的理论极限值是12.5MB/s。实测结果是，当一个客户机读取的时候，读取的速 度是10MB/s，也就是说达到客户机理论读取速度极限值的80%。对于16个客户机，整体的读取速度达到了94MB/s，大约是理论整体读取速度极限值的75%，也就是说每个客户机的读取速度是6MB/s。读取效 率从80%降低到了75%，主要的原因是当读取的客户机增加时，多个客户机同时读取一个Chunk服务器 的几率也增加了，导致整体的读取效率下降。

6.1.2 写入

N个客户机同时向N个不同的文件中写入数据。每个客户机以每次1MB的速度连续写入1GB的数据。图 3(b)显示了整体的写入速度和它们理论上的极限值。理论上的极限值是67MB/s，因为我们需要把每一 byte写入到16个Chunk服务器中的3个上，而每个Chunk服务器的输入连接速度是12.5MB/s。

一个客户机的写入速度是6.3MB，大概是理论极限值的一半。导致这个结果的主要原因是我们的网络协议栈。它与我们推送数据到Chunk服务器时采用的管道模式不相适应。从一个副本到另一个副本的数据传输 延迟降低了整个的写入速度。

16个客户机整体的写入速度达到了35MB/s(即每个客户机2.2MB/s)，大约只是理论极限值的一半。和多个客户机读取的情形很类型，随着客户机数量的增加，多个客户机同时写入同一个Chunk服

务器的几率也增加了。而且，16个客户机并行写入可能引起的冲突比16个客户机并行读取要大得多，因为每个写入 都会涉及三个不同的副本。

写入的速度比我们想象的要慢。在实际应用中，这没有成为我们的主要问题，因为即使在单个客户机上能够感受到延时，它也不会在有大量客户机的时候对整体的写入带宽造成显著的影响。

6.1.3 记录追加

图3(c)显示了记录追加操作的性能。N个客户机同时追加数据到一个文件。记录追加操作的性能受限于 保存文件最后一个Chunk的Chunk服务器的带宽，而与客户机的数量无关。记录追加的速度由一个客户机 的6.0MB/s开始，下降到16个客户机的4.8MB/s为止，速度的下降主要是由于不同客户端的网络拥塞以及 网络传输速度的不同而导致的。

我们的程序倾向于同时处理多个这样的文件。换句话说，即N个客户机同时追加数据到M个共享文件中，这 里N和M都是数十或者数百以上。所以，在我们的实际应用中，Chunk服务器的网络拥塞并没有成为一个 严重问题，如果Chunk服务器的某个文件正在写入，客户机会去写另外一个文件。

6.2 实际应用中的集群

我们现在来仔细评估一下Google内部正在使用的两个集群，它们具有一定的代表性。集群A通常被上百个工程师用于研究和开发。典型的任务是被人工初始化后连续运行数个小时。它通常读取数MB到数TB的数 据，之后进行转化或者分析，最后把结果写回到集群中。集群B主要用于处理当前的生产数据。集群B的任 务持续的时间更长，在很少人工干预的情况下，持续的生成和处理数TB的数据集。在这两个案例中，一个 单独的”任务”都是指运行在多个机器上的多个进程，它们同时读取和写入多个文件。

Cluster	A	B
Chunkservers	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

如上表前五行所描述的，两个集群都由上百台Chunk服务器组成，支持数TB的硬盘空间;两个集群虽然都 存储了大量的数据，但是还有剩余的空间。“已用空间”包含了所有的Chunk副本。实际上所有的文件都复 制了三份。因此，集群实际上各存储了18TB和52TB的文件数据。

两个集群存储的文件数量都差不多，但是集群B上有大量的死文件。所谓“死文件”是指文件被删除了或者 是被新版本的文件替换了，但是存储空间还没有来得及被回收。由于集群B存储的文件较大，因此它的 Chunk数量也比较多。

6.2.2元数据

Chunk服务器总共保存了十几GB的元数据，大多数是来自用户数据的、64KB大小的块的Checksum。保存在Chunk服务器上其它的元数据是Chunk的版本号信息，我们在4.5节描述过。

在Master服务器上保存的元数据就小的多了，大约只有数十MB，或者说平均每个文件100字节的元数 据。这和我们设想的是一样的，Master服务器的内存大小在实际应用中并不会成为GFS系统容量的瓶颈。 大多数文件的元数据都是以前缀压缩模式存放的文件名。Master服务器上存放的其它元数据包括了文件的 所有者和权限、文件到Chunk的映射关系，以及每一个Chunk的当前版本号。此外，针对每一个 Chunk，我们都保存了当前的副本位置以及对它的引用计数，这个引用计数用于实现写时拷贝(alex注: 即COW，copy-on-write)。

对于每一个单独的服务器，无论是Chunk服务器还是Master服务器，都只保存了50MB到100MB的元数 据。因此，恢复服务器是非常快速的:在服务器响应客户请求之前，只需要花几秒钟时间从磁盘上读取这 些数据就可以了。不过，Master服务器会持续颠簸一段时间–通常是30到60秒–直到它完成轮询所有的 Chunk服务器，并获取到所有Chunk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。在进行这些测量时，这两个集群已经上升了大约一周。（集群最近重新启动，以升级到新版本的GFS。）

重启后的平均写入速率小于30 MB/s。当我们进行这些测量时，B正处于写入活动爆发的中间，产生了大约100 MB/s的数据，这产生了300 MB/s的净工作负载，因为写入会传播到三个副本。

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

表三显示了不同时段的读写速率。在测试的时候，这两个集群都运行了一周左右的时间。(这两个集群最 近都因为升级新版本的GFS重新启动过了)。

集群重新启动后，平均写入速率小于30MB/s。当我们提取性能数据的时候，集群B正进行大量的写入操 作，写入速度达到了100MB/s，并且因为每个Chunk都有三个副本的原因，网络负载达到了300MB/s。 读取速率要比写入速率高的多。正如我们设想的那样，总的工作负载中，读取的比例远远高于写入的比 例。两个集群都进行着繁重的读取操作。特别是，集群A在一周时间内都维持了580MB/s的读取速度。集 群A的网络配置可以支持750MB/s的速度，显然，它有效的利用了资源。集群B支持的峰值读取速度是 1300MB/s，但是它的应用只用到了380MB/s。

6.2.4 Master服务器的负载

表3的数据显示了发送到Master服务器的操作请求大概是每秒钟200到500个。Master服务器可以轻松的应付这个请求速度，所以Master服务器的处理能力不是系统的瓶颈。

在早期版本的GFS中，Master服务器偶尔会成为瓶颈。它大多数时间里都在顺序扫描某个很大的目录(包 含数万个文件)去查找某个特定的文件。因此我们修改了Master服务器的数据结构，通过对名字空间进行 二分查找来提高效率。现在Master服务器可以轻松的每秒钟进行数千次文件访问。如果有需要的话，我们 可以通过在名称空间数据结构之前设置名称查询缓冲的方式进一步提高速度。

6.2.5 恢复时间

当某个Chunk服务器失效了，一些Chunk副本的数量可能会低于复制因子指定的数量，我们必须通过克隆 副本使Chunk副本数量达到复制因子指定的数量。恢复所有Chunk副本所花费的时间取决于资源的数量。 在我们的试验中，我们把集群B上的一个Chunk服务器Kill掉。这个Chunk服务器上大约有15000个 Chunk，共计600GB的数据。为了减小克隆操作对正在运行的应用程序的影响，以及为GFS调度决策提 供修正空间，我们缺省的把集群中并发克隆操作的数量设置为91个(Chunk服务器的数量的40%)，每 个克隆操作最多允许使用的带宽是6.25MB/s(50mbps)。所有的Chunk在23.2分钟内恢复了，复制的 速度高达440MB/s。

在另外一个测试中，我们Kill掉了两个Chunk服务器，每个Chunk服务器大约有16000个Chunk，共计 660GB的数据。这两个故障导致了266个Chunk只有单个副本。这266个Chunk被GFS优先调度进行复 制，在2分钟内恢复到至少有两个副本;现在集群被带入到另外一个状态，在这个状态下，系统可以容忍 另外一个Chunk服务器失效而不丢失数据。

6.3 工作负荷分析(Workload Breakdown)

本节中，我们展示了对两个GFS集群工作负载情况的详细分析，这两个集群和6.2节中的类似，但是不完全相同。集群X用于研究和开发，集群Y用于生产数据处理。

6.3.1 方法论和注意事项

这些结果只包括客户端发起的请求，以便它们反映我们的应用程序为整个文件系统产生的工作量。它们不包括执行客户端请求或内部后台活动的服务器间请求，如转发写入或重新平衡

我们从GFS服务器记录的真实的RPC请求日志中推导重建出关于IO操作的统计信息。例如，GFS客户程序 可能会把一个读操作分成几个RPC请求来提高并行度，我们可以通过这些RPC请求推导出原始的读操作。 因为我们的访问模式是高度程式化，所以我们认为任何不符合的数据都是误差(alex注:Since our access patterns are highly stylized, we expect any error to be in the noise)。应用程

序如果能够 记录更详尽的日志，就有可能提供更准确的诊断数据;但是为了这个目的去重新编译和重新启动数千个正 在运行的客户机是不现实的，而且从那么多客户机上收集结果也是个繁重的工作。

应该避免从我们的工作负荷数据中过度的归纳出普遍的结论(alex注:即不要把本节的数据作为基础的指导 性数据)。因为Google完全控制着GFS和使用GFS的应用程序，所以，应用程序都针对GFS做了优化，同 时，GFS也是为了这些应用程序而设计的。这样的相互作用也可能存在于一般程序和文件系统中，但是在 我们的案例中这样的作用影响可能更显著。

Operation	Read		Write		Record Append	
Cluster	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

> 表4：操作按大小（%）分列。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。

6.3.2 Chunk服务器工作负荷

表4显示了操作按涉及的数据量大小的分布情况。读取操作按操作涉及的数据量大小呈现了双峰分布。小 的读取操作(小于64KB)一般是由查找操作的客户端发起的，目的在于从巨大的文件中查找小块的数 据。大的读取操作(大于512KB)一般是从头到尾顺序的读取整个文件

在集群Y上，有相当数量的读操作没有返回任何的数据。在我们的应用中，尤其是在生产系统中，经常使用 文件作为生产者-消费者队列。生产者并行的向文件中追加数据，同时，消费者从文件的尾部读取数据。某些情况下，消费者读取的速度超过了生产者写入的速度，这就会导致没有读到任何数据的情况。集群X通 常用于短暂的数据分析任务，而不是长时间运行的分布式应用，因此，集群X很少出现这种情况。

写操作按数据量大小也同样呈现为双峰分布。大的写操作(超过256KB)通常是由于Writer使用了缓存机 制导致的。Writer缓存较小的数据，通过频繁的Checkpoint或者同步操作，或者只是简单的统计小的写 入(小于64KB)的数据量(alex注:即汇集多次小的写入操作，当数据量达到一个阈值，一次写入)，之后 批量写入。

再来观察一下记录追加操作。我们可以看到集群Y中大的记录追加操作所占比例比集群X多的多，这是因为 集群Y用于我们的生产系统，针对GFS做了更全面的调优。

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y

Cluster	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

> 表5：按操作大小（%）分列的传输字节数。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。如果读取试图读取文件末尾，两者可能会有所不同，这在我们的工作负载中并不少见。

写入大小也呈现双峰分布。大写入（超过256 KB）通常是由于写入器内的大量缓冲造成的。缓冲较少数据的写入器、检查点或同步更频繁，或者只是生成较少数据的写入器（低于64 KB）占较小写入的比例。

至于记录追加，群集Y看到的大记录追加的百分比比群集X高得多，因为我们使用群集Y的生产系统更积极地针对GFS进行调整。

表5显示了各种大小操作中传输的数据总量。对于各种操作，较大的操作（超过256 KB）通常占传输的大部分字节。由于随机查找工作负载，小读取（低于64 KB）确实传输了一小部分但重要的读取数据。

6.3.3追加与写入

记录追加操作在我们的生产系统中大量使用。对于集群X，记录追加操作和普通写操作的比例按照字节比 是108:1，按照操作次数比是8:1。对于作为我们的生产系统的集群Y来说，这两个比例分别是3.7:1和 2.5:1。更进一步，这一组数据说明在我们的两个集群上，记录追加操作所占比例都要比写操作要大。对 于集群X，在整个测量过程中，记录追加操作所占比率都比较低，因此结果会受到一两个使用某些特定大 小的buffer的应用程序的影响。

如同我们所预期的，我们的数据修改操作主要是记录追加操作而不是覆盖方式的写操作。我们测量了第一 个副本的数据覆盖写的情况。这近似于一个客户机故意覆盖刚刚写入的数据，而不是增加新的数据。对于 集群X，覆盖写操作在写操作所占据字节上的比例小于0.0001%，在所占据操作数量上的比例小于 0.0003%。对于集群Y，这两个比率都是0.05%。虽然这只是某一片断的情况，但是仍然高于我们的预 期。这是由于这些覆盖写的操作，大部分是由于客户端在发生错误或者超时以后重试的情况。这在本质上 应该不算作工作负荷的一部分，而是重试机制产生的结果。

6.3.4 master工作量

Cluster	X	Y
Open	26.1	16.3

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 (%)

表6显示了按请求类型分列的主请求。大多数请求要求读取块位置（FindPlace）和数据突变的租赁持有人信息（FindLeaseLocker）。

集群X和Y在删除请求的数量上有着明显的不同，因为集群Y存储了生产数据，一般会重新生成数据以及用新版本的数据替换旧有的数据。数量上的差异也被隐藏在了Open请求中，因为旧版本的文件可能在以重 新写入的模式打开时，隐式的被删除了(类似UNIX的open函数中的“w”模式)。

FindMatchingFiles是一个模式匹配请求，支持“ls”以及其它类似的文件系统操作。不同于Master服务器 的其它请求，它可能会检索namespace的大部分内容，因此是非常昂贵的操作。集群Y的这类请求要多一 些，因为自动化数据处理的任务进程需要检查文件系统的各个部分，以便从全局上了解应用程序的状态。 与之不同的是，集群X的应用程序更加倾向于由单独的用户控制，通常预先知道自己所需要使用的全部文 件的名称。

7.经验

在建造和部署GFS的过程中，我们经历了各种各样的问题，有些是操作上的，有些是技术上的。

起初，GFS被设想为我们的生产系统的后端文件系统。随着时间推移，在GFS的使用中逐步的增加了对研究和开发任务的支持。我们开始增加一些小的功能，比如权限和配额，到了现在，GFS已经初步支持了这 些功能。虽然我们生产系统是严格受控的，但是用户层却不总是这样的。需要更多的基础架构来防止用户 间的相互干扰。

我们最大的问题是磁盘以及和Linux相关的问题。很多磁盘都声称它们支持某个范围内的Linux IDE硬盘驱 动程序，但是实际应用中反映出来的情况却不是这样，它们只支持最新的驱动。因为协议版本很接近，所 以大部分磁盘都可以用，但是偶尔也会有由于协议不匹配，导致驱动和内核对于驱动器的状态判断失误。 这会导致数据因为内核中的问题意外的被破坏了。这个问题促使我们使用Checksum来校验数据，同时我 们也修改内核来处理这些因为协议不匹配带来的问题。

较早的时候，我们在使用Linux 2.2内核时遇到了些问题，主要是fsync()的效率问题。它的效率与文件的 大小而不是文件修改部分的大小有关。这在我们的操作日志文件过大时给出了难题，尤其是在我们尚未实 现Checkpoint的时候。我们费了很大的力气用同步写来解决这个问题，但是最后还是移植到了Linux2.4 内核上。

另一个和Linux相关的问题是单个读写锁的问题，也就是说，在某一个地址空间的任意一个线程都必须在 从磁盘page in(读锁)的时候先hold住，或者在mmap()调用(写锁)的时候改写地址空间。我

们发现 即使我们的系统负载很轻的情况下也会有偶尔的超时，我们花费了很多的精力去查找资源的瓶颈或者硬件 的问题。最后我们终于发现这个单个锁在磁盘线程交换以前映射的数据到磁盘的时候，锁住了当前的网络 线程，阻止它把新数据映射到内存。由于我们的性能主要受限于网络接口，而不是内存copy的带宽，因 此，我们用pread()替代mmap()，用了一个额外的copy动作来解决这个问题。

尽管偶尔还是有其它的问题，Linux的开放源代码还是使我们能够快速探究和理解系统的行为。在适当的时候，我们会改进内核并且和公开源码组织共享这些改动。

8.相关工作

像其他大型分布式文件系统（如AFS[5]）一样，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以更类似于xFS[1]和Swift[3]的方式在存储服务器上传播文件数据，以提供总体性能和更高的容错能力

由于磁盘相对便宜，复制比更复杂的RAID[9]方法更简单，GFS目前只使用复制来冗余，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在文件系统界面下不提供任何缓存。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，因为它们要么流过一个大数据集，要么随机在其中查找，每次读取少量数据。

一些分布式文件系统，如Frangipani、xFS、明尼苏达州的GFS[11]和GPFS[10]，删除了集中式服务器，并依赖分布式算法来实现一致性和管理。我们选择集中式方法，以简化设计，提高其可靠性，并获得灵活性。特别是，集中式主服务器使实现复杂的组块放置和复制策略变得更加容易，因为主服务器已经拥有大多数相关信息，并控制其更改方式。我们通过保持主状态较小并在其他机器上完全复制来解决容错问题。可伸缩性和高可用性（用于读取）目前由我们的影子主机制提供。主状态的更新通过附加到预写日志来持久化。因此，我们可以适应像竖琴[7]中的主副本方案，以提供比当前方案更强的一致性保证的高可用性。

我们正在解决一个与Lustre[8]类似的问题，即向大量客户端提供总体性能。然而，我们通过关注应用程序的需求而不是构建符合POSIX的文件系统，大大简化了这个问题。此外，GFS假设大量不可靠的组件，因此容错是我们设计的核心。

GFS与NASD体系结构最为相似[4]。NASD体系结构基于网络连接的磁盘驱动器，而GFS使用商品机作为组块服务器，就像NASD原型中所做的那样。与NASD的工作不同，我们的组块服务器使用懒散分配的固定大小组块，而不是可变长度的对象。此外，GFS实现了生产环境中需要的重新平衡、复制和恢复等功能。

与明尼苏达州的GFS和NASD不同，我们不寻求改变存储设备的模型。我们专注于解决现有商品组件复杂分布式系统的日常数据处理需求。

原子记录追加启用的生产者-消费者队列解决了与River[2]中的分布式队列类似的问题。River使用基于内存的队列分布在机器上，并进行仔细的数据流控制，而GFS使用一个持久文件，可以由许多生产者同时追加。River模型支持m-to-n分布式队列，但缺乏持久存储带来的容错能力，而GFS只有效地支持m-to-1队列。多个消费者可以读取同一个文件，但他们必须协调来分区传入的负载。

9.结论

谷歌文件系统展示了在商品硬件上支持大规模数据处理工作负载所必需的品质。虽然一些设计决策是针对我们独特的设置的，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。

我们首先根据我们当前和预期的应用程序工作负载和技术环境重新审视传统的文件系统假设。我们的观察导致了设计领域中截然不同的点。我们将组件故障视为常态而不是例外，针对大部分附加到（可能同时）然后读取（通常是顺序读取）的巨大文件进行优化，并扩展和放松标准文件系统接口以改善整个系统。

我们的系统通过持续监控、复制关键数据以及快速自动恢复来提供容错。chunk复制允许我们容忍chunkserver故障。这些故障的频率激发了一种新的在线修复机制，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IDE子系统级别的数据损坏，鉴于系统中的磁盘数量，这种情况变得非常普遍。

我们的设计为许多执行各种任务的并发阅读器和编写器提供了高聚合吞吐量。我们通过将通过主服务器的文件系统控制与直接在组块服务器和客户端之间传递的数据传输分开来实现这一点。主服务器对常见操作的参与通过大组块和组块租赁最小化，组块租赁将权限委托给数据突变中的主副本。这使得一个简单、集中的主服务器成为可能，而不会成为瓶颈。我们相信，网络堆栈的改进将解除当前对单个客户端写入吞吐量的限制。

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

致谢

我们要感谢以下人员对系统或论文的贡献。Brain Bershad（我们的牧羊人）和匿名评论者给了我们宝贵的意见和建议。Anurag Acharya，Jeff Dean和David desJardins为早期设计做出了贡献。Fay Chang致力于跨chunkserver的副本比较。Guy Edjlali致力于存储配额。Markus Gutschke致力于测试框架和安全性增强。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。我们在谷歌的许多同事勇敢地将他们的数据信任到一个新的文件系统中，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和Randolph Wang。无服务器网络文件系统。载于第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，和Kathy Yelick。Cluster I/O with River：使快速情况变得普遍。载于《并行和分布式系统中的输入/输出第六次研讨会记录》（IOPADS'99），第10-22页，1999年5月，佐治亚州亚特兰大。

[3]Luis-Felipe Cabrera和Darrell D. E. Long. Swift：使用分布式磁盘条带提供高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， ErikRiedel， David Rochberg， and Jim Zelenka.一种具有成本效益的高带宽存储架构，载于《第八届编程语言和操作系统架构支持会议录》，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·梅尼斯、大卫·尼科尔斯、马哈德夫·萨蒂亚纳拉亚南、罗伯特·西德博坦和迈克尔·韦斯特。分布式文件系统中的规模和性能。《计算机系统ACM交易》，6（1）： 51-81,1988年2月。

http://www.inter-mezzo.org，2003。

[7]芭芭拉·利斯科夫，桑杰·格玛瓦特，罗伯特·格鲁伯，保罗·约翰逊，柳巴·什里拉和迈克尔·威廉姆斯。竖琴文件系统中的复制。在第13次操作系统原理研讨会上，第226-238页，加利福尼亚州太平洋格罗夫，1991年10月。

[8]光泽.http://www.lustreorg，2003.

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

[10]FrankSchmuckand Roger Haskin. GPFS：一种用于大型计算集群的共享磁盘文件系统，载于《第一届USENIX文件和存储技术会议记录》，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，《戈贝尔文件系统》，载于《美国航天局戈达德空间飞行中心第五次大容量存储系统和技术会议记录》，马里兰州大学公园，1996年9月。

[12]Chandramohan A. Thekkath， Timothy Mann，和Edward K. Lee. Frangipani：一个可扩展的分布式文件系统，载于ACM第十六届操作系统原则研讨会论文集，第224-237页，法国圣马洛，1997年10月

扩展资料

1. 字节10万节点的HDFS架构实践