为了满足Google迅速增长的数据处理需求，我们设计并实现了Google文件系统(Google File System – GFS)。GFS与传统的分布式文件系统有着很多相同的设计目标，比如，性能、可伸缩性、可靠性以及可用性。但是，我们的设计还基于我们对我们自己的应用 的负载情况和技术环境的观察的影响，不管现在还是将来，GFS和早期文件系统的假设都有明显的不同。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。

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。

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设计预期

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 （alex注：well-defined）实现多 客户端并行追加数据到同一个文件里的语意。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。

谷歌三大核心技术（二）Google MapReduce中文版

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。

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，当我们第一次把GFS用于批处理队列系统的时候，热点的问题还是产生了：一个可执行文件在GFS上保存为single-chunk文件， 之后这个可执行文件在数百台机器上同时启动。存放这个可执行文件的几个Chunk服务器被数百个客户端的并发请求访问导致系统局部过载。我们通过使用更大 的复制参数来保存可执行文件，以及错开批处理队列系统程序的启动时间的方法解决了这个问题。一个可能的长效解决方案是，在这种的情况下，允许客户端从其它 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，每个文件的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服务器并不保存持久化保存哪个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）。