谷歌文件系统

**摘要**

我们已经设计并实现了谷歌文件系统，这是一个应用于数据密集型服务的可伸缩的分布式存储系统。这个系统不仅具有容错能力而且还可以运行在廉价的商用硬件上面，当客户端访问数量巨大的时候它提供总量非常可观的高性能。

在追求与先前类似分布式文件系统相似目标的同时，我们的设计是通过观察我们的服务的负载以及既是现在也是未来预期的技术环境，这个设计标志了一种和一些早期的文件系统的预期的设计的分离。这种情况已经使我们重新审视传统的选择并且探索本质的不同的设计点。

这个文件系统已经很成功地满足了我们的需求。被我们的服务作为产生和运行数据以及开发所需要大的数据集的存储平台，它在谷歌已经广泛的部署了。最大的集群能够提供上千TB的存储，这些存储分布在数千个机器上面的数千个磁盘，并且目前数千个客户端在访问它。

在这篇文章中，我们给出了文件系统的借口扩展设计来支持分布式的应用，讨论了我们的设计的很多方面，时事报告了微测试情况以及实际应用情况。

**目录和栏目描述**

分布式文件系统

**一般条款**

设计，可靠性，性能，测试

**关键词**

容错，可伸缩性，数据存储，集群存储

介绍

我们已经设计和实现了谷歌文件系统(GFS)来满足谷歌日益增长的数据处理需求。GFS和许多传统的分布式存储系统追求相似的目标，比如性能，可伸缩性，可靠性，以及可用性。但是，它的设计主要被我们的应用的负载情况以及技术环境（传统的或者预期的，反映了一个从早期的文件系统设计中的一次分离）所驱动。 我们重新审视了传统的设计选择并且探索完全不同的设计思路。

首先，在我们的系统中，组件失效是常态而不是异常。文件系统是由成百上千个由普通商业组件构成的机器所组成，并且被相当数量的客户端所访问。这些集群的组件中的一些被认为在任何时候都可能无法正常运转甚至有些组件从此以后再也不能恢复。我们实际应用中遇到过的问题包括服务的缺陷，操作系统的缺陷，认为错误，磁盘失效，内存，连接，网络，供电问题等等。所以，经常地监控，错误检测，容错处理以及自动恢复都必须考虑进系统。

其次，按照传统的标准，文件非常巨大。好几个Gb的文件是非常常见的。每个文件通常包括很多应用程序的对象，比如网络文档。当我们工作中经常遇到快速增长的数据集（数TB级的包含数亿个）的对象，我们的设计假设以及参数比如I/O的运转以及文件块的大小都需要重新设计。

第三，大多数文件都是以追加的形式添加而不是重写数据。在文件中随机的写实际上不再存在。一旦文件被写进去，文件只会被读，而且只能是有顺序的。很多数据都有这样的特性。一些也许能构成大的数据存储库用于数据分析程序扫描分析。另一些会是被在线应用不断的产生的数据流。有一些可能是档案数据，另一些可能是一些机器产生的中间结果，用于另一些机器的程序的输入数据，在产生的同时或者之后使用。在上面的大文件模型前提下，追加文件的关键变成了当缓存数据块在客户端失去吸引力的时候，如何提供优化的性能以及如何保证操作的原子性。

第四，提前设计应用程序和文件系统API可以通过增加灵活性来获得好处。例如，我们放低了GFS的一致性模型要求不用对应用程序添加繁重的负担来巨大的简化文件系统。我们也介绍了一种原子添加操作来保证多个客户端请求能够同时追加文件而不用在他们之间增加额外的同步操作。这些东西将会在接下来的章节中详细描述。

多个GFS集群已经用作不同目的而被部署。最大的集群有超过1000台存储节点，超过了300TB的磁盘空间，被数百个分布式的客户端连续的访问。

**2 设计概述**

**2．1 假设**

在设计满足我们需求的文件系统中，我们一直被这些既带来了挑战也带来了机会的假设所引导。先前我们提到了一些关键的观察结果，现在我们要给出这些假设的详细细节。

这个系统是基于一些经常失效的许多廉价的商业组件。在常规运行时，它必须时刻监控自己，并且检测，负荷，以及迅速的从组件失效中恢复。

系统存储适量的大文件。我们预期的是几百万个文件，每个文件典型的有100MB或者更大。几GB的文件是常见的情况，并且需要高效的管理。小文件也必须支持，但是我们的系统对它们来说不是最优的。

系统运行的主要负载来自于两种读的方式：大的流式的读和小的随机的读。在大的流式的读中，单个操作典型的读取几百KB或者更常见的1MB甚至更多。连续的来自于同一个客户端经常读取一个连续的一个块的文件。小文件的随机读取典型的是读取任意偏移量的几个KB。性能敏感的应用程序经常批处理或者对小文件读取进行排序来稳定向前读取文件而不是前后不断地移动读取。

负载也包括许多大的，连续的追加写数据到文件中。典型的运行大小是和读取的大小相似。我们的文件经常使用生产者消费者队列或者许多合并的方法。数百个生产者（每个机器运行一个），要同时追加文件。最少的同步构成的原子操作非常重要。文件在之后会被读取，或者消费者也许正在读取。

高并且持续的带宽比低的延迟更加重要。我们的大多数目标程序通过高的批量处理速率获得了一个溢价，因为很少有单个的读或者写需要快速的响应。

**2.2 接口**

虽然GFS没有提供和标准的POSIX完全相同的API，但是它提供了相似于文件系统的接口。文件被分等级的组织在通过路径辨识的文件夹中。我们支持创建，删除，打开，关闭，读和写文件等的常见操作。

更进一步的，GFS有快照和记录文件的添加操作。快照操作创建了一个低开销的文件或者目录树。追加记录可以允许多个客户端同时追加数据并且能够保证每个单个客户端追加操作的原子性。实现多种方式的合并结果以及生产者消费者队列那么许多客户端可以连续的追加数据而不需要额外的加锁操作是非常有用的。我们发现这几种文件对于我们建立大规模的分布式程序非常有价值。快照和追加将在后面的3.4和3.3分别讨论。

**2.3 架构**

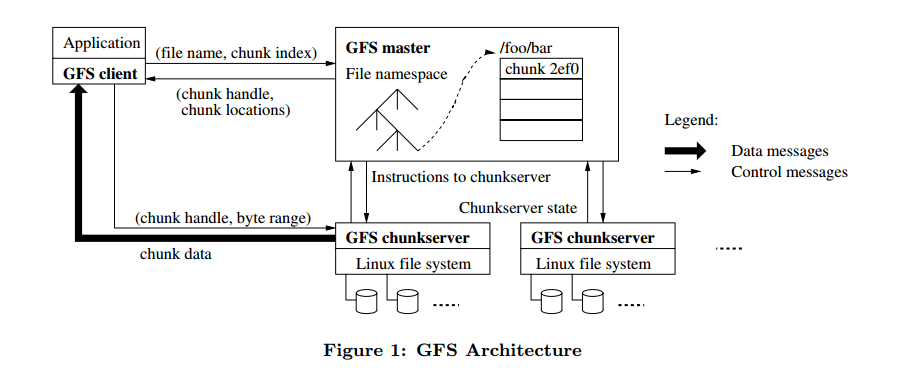
一个GFS集群包括了一个主和多个从服务器并且被多个客户端所访问，如图1. 0 集群中每个节点是一个Linux系统的商用机，每个机器上面运行着用户级的服务器程序。在一个服务器上面运块从服务器程序和客户端程序是非常容易的，只要机器资源允许并且可能运行的程序的代码引起的低可靠性是可以接受的。

文件被切分成固定大小的块。每个块被一个不可改变的并且全局唯一的64bit的块号所标识，这个块号是在这个块被创建的时候被主分配的。块服务器以Linux文件的方式存储数据块到本地磁盘通过块号和数据偏移（文件大小）来读或者写文件。为了可靠性，每个块被复制到多个数据块服务器。默认情况下，我们保存三备份，尽管用户可以为不同命名区域的文件指定不同备份等级。

主服务器维护所有文件系统的元数据。包括了命名空间，访问权限控制信息，从文件到块的映射，以及目前文件块的位置。它也控制着系统层级的活动比如块的分配管理，孤立块的资源回收，以及块服务器之间的数据迁移。主服务器利用心跳机制定期的与每个块服务器通信来引导块服务器以及收集它们的状态。

GFS连接到每个应用程序的客户端实现了文件系统的API并且与主服务器和块服务器通信来从应用程序这边读写数据。客户端从主服务器得到系统的元信息，但是所有的数据处理通信流程直接和块服务器交互。我们不提供 POSIX标准的API所以不需要关注到Linux系统的vnode层面。

客户端和块服务器都不会缓存文件数据。客户端缓存数据没什么好处因为大部分程序数据流通过巨大的文件或者数据集太大了以至于不能缓存。通过缓存减少相关事物并没有简化客户端和整个系统（但是客户端会缓存数据的元信息）。块服务器不必要缓存文件数据是因为块数据被存为本地文件所以Linux缓冲区已经缓冲了经常访问的数据进内存。



**2.4 单个主服务器**

有了一个主服务器大大简化了我们的设计以及使得主能够让复杂的块放置以及备份确定使用全局的信息。但是，我们必须使得主的读取和写的操作尽量最少以便不让它成为整个系统地瓶颈。客户端绝不会从主服务器中读取或者向其写入数据。取而代之的是，客户端程序会询问主服务器自己该链接哪个块服务器。它还会缓存这些信息一段有限的时间，然后直接和块服务器交互以及很多后记得操作。

现在来讲讲图1 中提到过的简单的读过程。首先，利用固定大小的块，客户端将文件名字和程序划分文件的文件偏移量转换成为块的索引。然后，块服务器发送一个包含文件名和快索引的请求。主服务器将相应的块处理和副本的位置返回给客户端。客户端把文件名和块索引作为键值来保存这些信息。

客户端然后向其中一个拷贝发送请求，这个拷贝最可能是离客户端最近的。请求给出块的ID以及在这个块里面的偏移量。想要读取本块更多的数据不再需要和主服务器交互了除非信息过期或者文件被重新打开。事实上，在客户端典型的请求是一次请求多个块，然后主服务器也能迅速的返回有关这些块的信息。这些额外的信息避免了一些可能的客户端向主服务器发出的请求。

**2.5 块大小**

块大小是设计参数的关键之一。我们选择了64MB，这个大小比通常的文件系统块大小大得多。每个块的备份以普通Linux文件的形式被存储在块服务器上面并且只在需要的时候进行扩展。被动的空间分配方式避免了由于内部碎片而造成的空间浪费，也许是如此大的块大小的最大的反对声音来源。

大文件块特性带来了一些重要的优势。首先，它减少了客户端和主服务器的交互必要因为在相同文件块上面的读写只需要向主服务器提出一次初始化请求来获得块的位置信息。这个请求的减少非常重要，因为应用程序多数时候是连续的读写文件。即使有少部分的随机读取，客户端也可以方便的缓存所有的多达几个TB大小数据集的块的位置信息。其次，在一个大的文件块上面，客户端更倾向于在每个数据块上面进行多次操作，这样可以通过在一段时间内减少到数据块服务器的TCP连接的次数来减少网络开销。第三，这样也减少了主服务器缓存的元信息量。这样也使得我们可以将数据的元信息保存在内存中，可以带来一些其他好处，我们将在2.6.1讨论。

另一方面，大的块即使消极空间分配方式，也有一些不好的地方。小文件只有很少的几个块，甚至只有一个块。如果许多客户端访问相同文件块服务器存储的文件可能成为热点数据。在实际中，热点数据没有成为主要的问题因为我们的应用程序多数时候是连续的读取大的多块数据。

然而，当GFS首次被用在批处理队列系统（一种可执行文件单次写入然后可以同时运行在数百台机器上面的系统）中的时候热点还没有开发出来。少数的几个块服务器保存着可执行文件将会因为数百个连续的请求而过载。我们通过提高这种可执行文件的备份系数和错开消息队列系统的请求开始时间修复了这个问题。一种可能的长久的解决方案是在这种场景下允许客户端读取其他客户端的数据。

**2.6 元数据**

主服务器主要存储三种元数据：文件和块的命名空间，文件到块的映射，以及每个块备份的位置。所有的元信息都保存在主服务器的内存中。前两个类型的元信息（文件盒块的命名空间，文件到块的映射）也被以动态的写入日志的方式永久保存在主服务器的本地磁盘和在远端服务器上面备份。使用日志的方式使得我们更新主的状态简介、可靠并且在主服务器宕机的时候不用间断。主服务器不会持久保存块服务器的位置信息。取而代之的是，它会在自己启动的时候或者有一个块服务器加入集群的时候询问每个块服务器。

**2.6.1 内存数据结构**

由于元信息被保存在内存中，主服务器的请求响应会非常快。另外，对于主服务器来说在后台周期性地检查整个系统的状态是简单和高效的。周期性检查的目的是完成块垃圾回收，在块服务器出现故障的时候再备份数据，以及为了实现块服务器之间的磁盘空间和负载均衡的块数据迁移。4.3和4.4将会更深入的讨论这些过程。

一个可能的担忧是，对于这种仅靠内存的模式块服务器的数量会被限制，所以整个系统的容量也会被限制，整个系统的容量会被主服务器的内存所限制。主服务器为每64MB的块维持少于64byte的元信息。绝大多数块是满的因为绝大所数文件包含许多块，只有最后的块不被填满。相似的是，每个文件的名字空间典型的需要低于64byte的，因为存文件的名字的时候只需要前缀压缩。

如果必要支持更大的文件系统，给主服务器添加额外的内存的花销相对于为了获得简化，可靠性，性能以及伸缩性来说花销小多了。

**2.6.2 块位置**

主服务器不会永久保存有关哪个块服务器保存了一个块的拷贝的信息。它只是在启动的时候轮询块服务器。主服务器可以保持自己的信息是最新的，因为它通过心跳机制掌握着块的放置以及监视块服务器状态。

刚开始的时候我们尝试着把块的位置信息永久的保存在主服务器，但是我们发现在服务器启动的时候向块服务器请求会简单很多，然后再周期性的重复上面操作（请求）。这样做的话就没有了当块服务器加入或者离开集群，更改名字，失效，重启等的时候要保证主服务器要与之同步的麻烦。在数百个服务器组成的集群里面这样的事情太常见了。

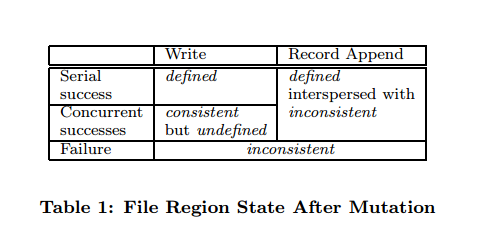
另一个理解这种设计方案的方式是明白就是块服务器有最后的发言权说自己的磁盘有哪个以及没有哪个数据块。试图在主服务器上面保存一个持久的块快照信息是没有实际意义的因为块服务器宕机肯定会引起自发的消失（例如，一块磁盘可能失效或者被损坏）或者控制器或对一个块服务器重命名。

**2.6.3 运行日志**

程序运行的日志包括了重要的元信息改变的历史记录。它是GFS的中央。他不仅是元信息唯一的持久化保存方式，也提供了一个定义并行运行顺序逻辑的时间行。文件和块，正如它们的版本一样（看4.5），都唯一并且永久的被他们创建的逻辑时间所标识。

由于运行日志至关重要，我们必须可靠地存储它们并且不能让客户端具有访问其变化的权限直到元信息的变化被持久化（保存）。另外，我们能够高效的丢弃整个文件系统或者最近的客户端访问行为的记录即使块服务器自己还运转着。所以，我们将其备份到多个远端的机器上面并且只在将相应的日志写入近端和远端磁盘后再对客户端做出响应。在写入磁盘之前主服务器搜集多个日志记录将其集中在一起以减少写入磁盘和备份对整个系统带来的影响。

主服务器通过重放日志来恢复整个系统。为了减少启动时间，我们必须让日志尽量少。当日志的大小超过某个值的时候主服务器对状态设置监测点因此它可以依靠来自于本地磁盘最近的监测点以及只对在监测点之后的有限的日志量进行响应来恢复系统。监测点被存在一个类似于B-tree的数据结构中这样就可以直接映射进内存利用名字查找，而不必额外的解析。这个大大的加快了恢复速度以及改善了可用性。



因为建立监测点会花费一段时间，在主服务器内部，其状态的建立方式是创建一个新的监测点不用延迟新近的改变。主服务器切换到一个新的日志文件然后建立在一个独立的线程里面新的监测点。这个新的监测点在切换之前包含了所有的变化。他可以在一分钟左右为一个包含几十亿个文件的系统创建监测点。当创建完成的时候，它会被写入远端和本地磁盘。

恢复仅仅需要最近的完成的监测点以及之后连续的日志文件。更加久远的监测点可以被自由删除，尽管我们会为了预防巨大的灾难而保存一些近期的监测点。检测失效不会影响正确性因为恢复代码会检测并且跳过不完整的监测点。

**2.7 一致性模型**

GFS具有一个低的一致性的模型，所以可以很好的支持我们的高度分布式的应用的同时还能保持相对简单高效的实现。我们现在来讨论GFS的设想以及这些设想对于应用程序意味着什么。我们也强调GFS怎样维持这些假设，细节将会被留到本篇文章的其他章节讨论。

**2.7.1 GFS的设想**

文件命名空间的变更（比如文件创建）是自动的。他们只被主服务器创建：名字空间锁保证原子性和正确性（章节4.1）；主服务器的日志定义了这些过程的全局的总的顺序（章节2.6.3）。

在文件改变之后文件域的状态和文件改变的类型，是否成功，以及是否有并发的改变有关系。表格1 总结了这个结果。文件域是一致的如果所有客户端程序不管读到了哪个备份要总是读取相同的数据。如果域是一致的并且客户端程序能够看到在文件写的整个过程中的改变那么域在改变之后定义。当文件改变没有来自于并发写者的干扰，有效地域就被定义了（表明满足了一致性）：所有的客户端程序都会看到写入的改变。并发的成功改变使得域没有被定义但是一致：所有的客户端看到相同的数据，但是这个不能反映任何变化。典型的，多个文件的改变包含着混合的片段。一次失败的修改使得域不一致（所以没有被定义）：不同的客户端会在不同的时间看到不同的数据。下面我们将展示我们的应用怎样区分定义了的和没有定义的域。我们的应用不必要更深入的分辨是哪种不同的未定义类型。

数据改变可能是以追加的形式。写会导致数据被写入一个特定应用的文件偏移量。记录追加导致数据（记录）至少一次原子性的追加即使在并法改变的场景，而不是一个GFS系统偏移量的选择（章节3.3）。（对比而言，一个“经常地”追加很少，而会是在客户端认为是目前文件的尾端的一定偏移量）。偏移量将会返回给客户端同时标记一个已经定义的包含记录的域的开头。另外，GFS可能插入填充或者拷贝记录在之间。被填充的数据被认为不一致并且通常典型的和用户数据量相比是小巫见大巫。

经过了一系列的成功修改，修改过的文件被认为是定义了的或者包含了最近的一次修改。GFS通过用相同的顺序在所有拷贝块上面完成修改（3.1），并且使用块版本号来检测所有拷贝的老旧数据因为老旧数据在块服务器宕机的时候丢失了修改。（4.5）。老旧拷贝数据永远不会涉入一个改变或者被客户端通过询问主服务器得到位置。他们是将会被最早回收的垃圾。

由于客户端缓存了块服务器的位置信息，在客户端的缓存信息被更新之前，客户端可能去访问一个老旧备份块。这个时间窗是被缓存数据时间以及下一次打开文件所限制的，这时候缓存会清洗涉及到这个文件的所有的块服务器信息。另外，由于我们大多数文件是追加的形式，老旧备份通常会返回不完全的块结尾而不是过时的数据。当读者重试连接主服务器，他很快会获得块服务器的位置。

成功改变之后很久，组件失效当然依然会腐蚀或者破坏掉数据。GFS通过在主服务器和所有的块服务器之间经常地握手来鉴别失效的块服务器以及通过校验和来检测被破坏的数据（5.2）。一旦有问题出现，数据会尽快的从永久的备份中恢复（4.3）。仅在块服务器来得及反应之前块的所有备份被丢失才会出现不可恢复的数据丢失情况，典型的情况是几分钟。即使在这种情况，数据不可访问，不是破坏：应用程序会收到明确的错误类型而不是被破坏的数据。

**2.7.2 对应用程序的影响**

在一些在其他目的中应用的简单技术的协助下GFS的应用能够适应低的一致性，这些技术包括：依靠追加而不是重写，检测点，自己写，自检测记录。

所有的应用程序文件改变都以追加的形式而不是重写。在典型的应用中，一个写者从头至尾的产生数据。它写完所有数据后原子性的修改文件的名字作为一个永久的文件名，或者定期的检测有多少数据已经被成功写入。检测点可能也会包含应用级的校验和。读者仅仅验证和运行到最新的检测点，这个检测点被认为是定义了的状态。不考虑一致性和并发性的问题，这个方法已经很好地被我们应用了。相对于随机写，追加的方式对应用的失效是更加的高效以及有弹性的。检测点使得写者可以增量的重启并且阻止读者读取成功写入但是从应用的角度看不完整的数据。

在其他的典型的使用场景，许多写者并发的追加数据到一个文件来合并数据或者作为一个生产者消费者队列。