分布式系统学习——GFS谷歌文件系统Paper翻译

2020-4-25 23:32:28

# 读书笔记——The Google File System

[参考翻译版本：Google File System 翻译（转）](https://www.cnblogs.com/finalboss1987/p/5169559.html)

加粗和笔者注是重点关注的部分。

# Abstract

我们已经设计和实现了 Google File System，一个适用于大规模分布式数据处理相关应用的，可扩展的分布式文件系统。它基于普通的不算昂贵的硬件设备，**实现了容错的设计，并且为大量客户端提供极高的整体处理性能**。

# Introduction

我们已经为 Google 迅速增长的数据处理需求而设计和实现了 Google File System(GFS)。GFS 和上一个分布式文件系统有着很多相同的设计目的，比如性能，扩展性，可靠性，以及可用性。不过，包括现在和未来预期下，它的设计是基于我们应用的工作量和技术环境驱动的，因而 GFS 都有一些和上一个版本明显不同的地方。这就要求我们重新审视传统的设计选择，并探索完全不同的设计要点。

* 首先，**节点失效将被看成是正常情况，而不再视为异常情况**。整个文件系统包含了几百个或者几千个由廉价的普通机器组成的存储机器，而且这些机器是被与之匹配数量的客户端机器访问。这些节点的质量和数量都实际上都导致总会出现一些失效情况，并且某一部分机器并不会从当前失效中恢复回来。原因是多样的，比如程序的 bug，操作系统的 bug，人工操作 error，以及硬盘坏掉，内存，网络，插板的损坏，电源的坏掉等等。因此，**持续监视，错误检测，容错处理，自动恢复**必须集成到这个文件系统的设计中来。
* 第二，**大型数据文件的规模出现导致文件观念的改变**。当我们通常操作迅速增长的，由很多 TB 组成的，包含数十亿对象的数据集，我们可不希望管理数十亿个 KB 大小的文件（无论文件系统是否有如此能理）。所以，设计约定和设计参数比如 I/O 操作以及 blocksize（块大小），都需要重新审查。
* 第三，**文件的改变常为追加（append）而非覆盖（overwrite），对一个文件的随机写操作（random write）实际上是几乎不存在的**。当一旦写完，文件就是只读的，并且一般都是顺序读取。许多 data 都具备这样的特性：仅供应用扫描数据的超大数据库、应用不断产生的数据流、归档的数据、机器交互的中间数据生成和处理。对于这些巨型文件的访问模式来说，“追加（append）”是最重要的，所以我们首要在性能上优化并保证原子性操作的就是它，而缓存客户端中的数据块则失去吸引力（不是 GFS 重点关注的对象）。
* 第四，结合应用与文件系统 API 一同设计对于增加整个系统的弹性有很大的好处。例如我们放宽 GFS 一致性模型后可以在避免给应用增加负担的同时简化文件系统。我们也引入了原子的追加（append）操作，这样可以让多个客户端并发对同一文件进行追加操作，而不需要额外的同步。这些在本论文的后边章节有描述。

# Design Overview

## Assumptions

在针对我们的需求设计文件系统时，由一些既是机会又是挑战的假设指导我们的操作。前述给出了一些关键信息，现在将更细致的阐述这些假设。

* 系统由许多廉价而易出错的商品组件构成，他必须持续监管自身并在此基础上及时监测、容纳组件故障。
* 系统存储了大量的超大文件。预期好几百万个文件，每一个超过 100MB。数 GB 的文件会经常出现并且应当受到有效的管理。系统必须同时支持小型文件，但不必为小型文件进行特别的优化。
* 一般的工作都是由两类读取操作组成：大规模流式读取和小规模随机读取。
  + 在大规模流式读取中，每个独立操作通常要读取几百 KB 的数据，更常见的是每次读取 1MB 或者以上的数据。对于同一个客户端来说，往往会发起连续的读取操作顺序读取一个文件。
  + 小规模的随机读取通常在文件的不同位置，读取几 KB 数据。对于性能有过特别考虑的应用通常会作批处理并且对他们读取的内容进行排序，这样可以使得他们的读取始终是单向顺序读取，而不需要往回读取数据。
* 工作同时包含大规模顺序的写操作，将数据不断追加之文件中，其规模与读取规模相当。一旦写入完成，文件几乎不会再做该懂。小的随机写入操作应被支持但无需在意其效率。
* 系统必须为同时追加到同一文件的多个客户端有效地实现明确定义的语义含义。我们的文件通常用于生产者 - 消费者队列情形或用于多路合并（many-way merging）。 数百个独立运作在机器上的生产者将并发地向某个文件追加数据。 具有最小同步开销的原子性保证是必不可少的。
* 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在处理大量数据时以很高的速度预先完成任务，很少有对单个读或写有严格的响应时间要求的应用程序。

## Interface

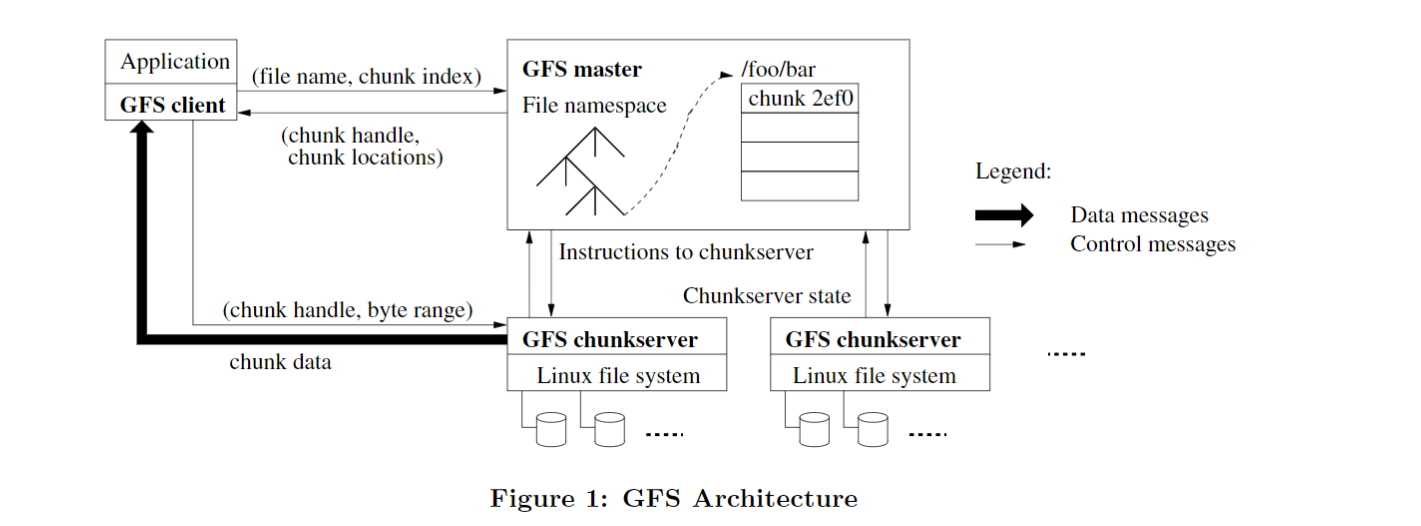
虽然 GFS 没有实现像 POSIX 这样一些标准地 API，但是它提供了一套常见的文件系统接口。文件在目录中按层次结构组织，并由路径名标识。我们提供了对文件的一些常规的操作：*create*，*delete*，*open*，*close*，*read*，*write*。

此外 GFS 还提供了*snapshot*，*record append*操作。Snapshot 以较低耗费建立一个文件或者是目录树的拷贝。Record append 则允许多个客户端对相同一个文件并发进行追加（append）数据时保证每个追加操作的原子性。这个对于多路合并操作和多个客户端同时操作的生产者/消费者队列的实现非常有用，它不用额外的加锁处理。这种文件对于构造大型分布式应用来说，是不可或缺的。两者将在后面 3.4 和 3.3 节单独讲述。

## Architecture

一个 GFS 集群由一个 master 和多个 chunkserver 组成，可以被多个 client 访问，如图 1 所示。

它们都是一个运行着用户级服务进程（user-level process）的商品化 linux 机器。只要机器资源允许以及由于运行可能的片状应用程序代码带来的低可靠性是可以接受的，那么可以很容易的在同一台机器上运行一个 chunkserver 和 client。



OS\_4\_1

文件被划分成固定大小的 chunk。每个 chunk 是由 chunk 创建时由 master 分配的一个不可变的全局唯一的 64bit 句柄来标识。Chunkserver 将 chunk 作为 linux 文件存储在本地，对于 chunk 数据的读写通过 chunk 的 handle 和字节边界来表示。为了可靠性，每个 chunk 存储在多个 chunkserver 上。尽管用户可以为不同文件名字空间区域指定不同的备份级别，默认地我们存储三个备份（与后续的 MapReduce 对于 GFS 输入文件常为三份备份习惯上是一回事）。

Master 维护所有的文件系统元数据。包括名字空间（namespace），访问控制信息（access control information），文件与 chunk 的映射信息，chunk 的当前位置（current locations 注意是 chunck 多个备份的位置）。它也控制系统范围内的一些活动，比如 chunk 租赁管理，僵死 chunk 的垃圾回收，chunkserver 间的 chunk 迁移。Master 与 chunkserver 通过心跳信息（*Heart Beat* messages）进行周期性的通信，以发送指令和收集 chunkserver 的状态。

应用程序链接的 GFS 客户端代码，实现了文件系统 API 以及代表应用程序与 master 和 chunkserver 进行通信以读写数据。客户端如果需要操作元数据则需要与 master 通信，但是所有的纯数据通信直接与 chunksever 通信。我们没有提供 POSIX API，因此也就不需要与 linux vnode layer 关联。

客户端或者 chunkserver 都不会对文件数据进行缓存（cache）。客户端缓存只能得到很少的好处，因为大部分的应用需要直接读取整个大文件或者工作集合太大根本无法缓存。没有 cache 简化了客户端和整个系统，因为不需要考虑缓存一致性问题(实际上客户端会缓存元数据)。Chunkserver 不需要进行文件数据缓存，是因为 chunk 是作为本地文件存储，这样 Linux 自身会将那些经常访问的数据进行缓存（间接利用了操作系统？）。

## Single Master

只有一个 master 大大简化了我们的设计，而且使得 master 可以利用全局信息对 chunk 的放置和备份进行更好（sophisticated 老练的）的判断。然而，我们必须最小化其在读写过程中的参与，以防止其成为运行的瓶颈。Clients 永远不会通过 master 读取数据，它只向 master 询问它该与哪个 chunckserver 联系，并且 client 将这些信息在有限的时间段内进行缓存，直接与 chunksever 交互进行很多后续的操作。（**注意，不缓存文件数据，但是会缓存和那个 chunkserver 交互的信息**）。

我们简单解释一些一个读操作的交互过程：首先，通过固定大小的 chunk，客户端将应用程序中标识的文件名和 offset 转换为 chunk 的 index。然后给 master 发送一个包含文件名和 chunk index 的请求，master 返回相应的 chunk 的 handle 和所有备份的位置。客户端以文件名和 chunk index 为 key 将这条信息进行缓存。

然后客户端给其中一个备份发送一个请求，通常是最近的那个。请求标识了 chunk 的 handle 以及在那个 chunk 内的字节边界。直到缓存信息过期或者重新打开文件之前，对于相同 chunk 的后续读操作就不需要 client-master 的通信了。事实上，客户端通常在一个请求中查询多个 chunk 的信息，master 也可以将这些被请求的多个 chunk 的信息包裹在一块进行返回。这种特别的信息，并没有额外的花费就避免了未来的 client-master 的多次通信。

## Chunk Size

Chunk 大小是一个关键的设计参数。我们选择了 64mb，远远大于现有的文件系统块。每个 chunk 的副本作为普通的 linux 文件存储在 chunkserver 上，如果需要才会进行扩展。Lazy 空间分配避免了内部碎片造成的空间浪费，很可能最大的碎片有向一个 chunk 那么大。

大的 chunk size 提供了几个重要的优势。首先，降低了 client 与 sever 的交互需求，因为在相同 chunk 上的读写只需要一个初始化请求就可以从 master 得到 chunk 的位置信息。这个减少对于我们的应用负载是非常明显的，因为我们应用大部分需要顺序的读写整个大文件。即使对于小的随机读取，客户端也可以很容易的缓存一个几 TB 工作集的所有 chunk 的位置信息。其次，由于 chunk 很大，那么客户端就很有可能在一个给定的 chunk 上执行更多的操作，这样可以将一个与 chunkserver 的 TCP 连接保持更长的时间，这就减少了网络开销。再者，降低了存储在 master 上的元数据大小。这样就允许我们将元数据存放在内存中，反过来就带来了我们将在 2.6.1 中讨论的其他优势。

另一方面，大的 chunk size，即使采用了 lazy 空间分配，也有它的缺点。小的文件可能只有少数几个 chunk，或许只有一个。如果很多的 client 都需要访问这个文件，这样那些存储了这些 chunk 的 chunkserver 就会变成热点。实际中，热点还没有成为一个主要的考虑点因为我们的应用绝大部分都是在顺序读大的多 chunk 文件。

然而，当 GFS 第一次使用在一个批处理队列（Batch-queue）系统时，热点确实出现了：一个可执行文件作为一个 chunk 的文件写到 GFS，然后同时在数百台机器上开始执行。存储了该可执行文件的那些 chunkserver 被数百个并发请求瞬间变成超载。我们通过更高的备份级别存储这样的可执行文件以及减慢队列系统的应用程序启动时间解决了这个问题。一个潜在的长远的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。（单一申请频次高的文件容易导致热点问题。）

## Metadata

Master 存储了三个主要类型的元数据：**文件和 chunk 名字空间，文件到 chunk 的映射信息，每个 chunk 的备份的位置**。所有的元数据都保存在 master 的内存中。前两种类型还通过将更新操作的日志保存在本地硬盘和备份在远程机器来保持持久化。使用 log 允许我们简单可靠地更新 master 的状态，不用担心当 master crash 的时候的不一致性。Master 并没有永久保存 chunk 的位置信息，而是在 master 启动或者某个 chunkserver 加入集群时，它会向每个 chunkserver 询问它的 chunks 信息（动态询问保存）。

### In-Memory Data Structures

由于元数据存储在内存里，master 的操作是很快的。因此对于 master 来说，可以简单有效地对在后台整个状态进行周期性扫描。这个周期性的扫描是用来实现 chunk 垃圾回收，chunkserver 出现失败时进行的重备份（re-replication），以及为了平衡负载和磁盘空间在 chunkserver 间的 chunk 迁移。4.3 4.4 将进一步讨论这些活动。

这样全内存策略存在一个潜在的限制就是 chunk 的数目，因此整个系统的容量取决于 master 有多少可用内存。实际中这不是一个很严重的限制。Master 为每个 64MB 的 chunk 维护少于 64byte 的数据。大部分的 chunk 是满的，因为大部分的文件包含多个 chunk，只有最后一个 chunk 可能是未慢的。类似的，每个文件名字空间数据通常需要少于 64byte 因为文件名称存储时会使用前缀压缩算法进行压缩。

如果需要支持更大的文件系统，只需要往 master 里添加内存。这点开销与通过将元数据存储到内存所得到简单性，可靠性，性能和灵活性，将是很小的一笔花费。

### Chunk Loactions

Master 并没有提供一个永久性的存储保存对于一个给定的 chunk 都是那些 chunkserver 保存了它的副本。它只是在启动时，简单地从 chunkserver 那里把这些信息拉过来（见本小节开头）。Master 能够保证它自己是更新过的，因为是由它来控制 chunk 的放置，以及通过周期性的心跳信息来监控 chunkserver。

起初，我们尝试将 chunk 位置信息永久保存在 master，但是我们发现在启动时去 chunkserver 请求这些数据更简单。这样避免了当 chunkserver 在加入或者离开集群，改名，失败，重启等待时需要的 master 与 chunkserver 间的同步。在一个数百台机器的集群中，这样的事件太经常了。

理解这个设计决定的另一个方式是 chunkserver 对于自己有还是没有某个 chunk 具有最终的发言权。在 master 上维护一个这些信息一致性视图是没有意义的，因为发生在 chunkserver 上的错误可能使得一些 chunk 突然间不见了(比如硬盘可能会坏掉或者不可用)，一个操作可能将 chunkserver 重命名。

（读者注：server 具备自己有没有某个 chunck 的最终发言权，要联络，不要记忆。）

### Operation Log

操作日志包含了关键元数据改变的历史记录。它是 GFS 的核心。它不仅是元数据的唯一一致性记录，而且它也定义了那些并发操作的逻辑上的时间表。**文件和 chunk 的版本都是唯一且永远地由它们创建时的逻辑时间来标识的**。

因此操作日志是很关键的，我们必须可靠地保存它，在任何元数据变更在被持久化之前不应当被客户端看到。否则，即便 chunckserver 自己保存了它们，我们将丢失整个文件系统或者最近的客户端操作。因此我们将它备份在多个远程机器上，对于一个客户端操作只有当该操作对应的日志记录被刷新到本地和远程的磁盘上时才会发出响应。Master 将几个操作日志捆在一块刷新，从而降低刷新和复制对于整个系统吞吐率的影响。

Master 通过重新执行操作日志来恢复它的文件系统。为了最小化启动时间，我们必须将日志保持在很小的规模。当日志增长超过一定的大小后，Master 给它的状态设置检查点，它可以通过从本地磁盘加载最新的检查点进行恢复，然后重新执行那些在该检查点之后的日志记录。检查点保存了一个压缩的类 B 树的结构，不需要额外的解析就可以直接映射到内存用于名字空间查找。这大大提高了恢复的速度和可用性。

因为建立一个检查点会花费一些时间，master 的内部状态的结构设计使得一个新的检查点可以不需要延时那些接受到的变化就可以被创建。Master 会启动一个新的线程切换到一个新的日志文件然后创建新的检查点。这个新的检查点包含在切换之前的所有变更。对于一个包含几百万文件的集群大概需要几分钟就可以完成。结束后，它将会被写回本地和远程的磁盘。

恢复只需要最新完全的检查点和后来的日志文件。更老的检查点和日志文件可以自由的删除，当然我们会保存了一些来应对某些突发情况。在创建检查点的时候发生的失败不会影响系统的正确性，因为恢复代码会检测和跳过不完全的检查点。

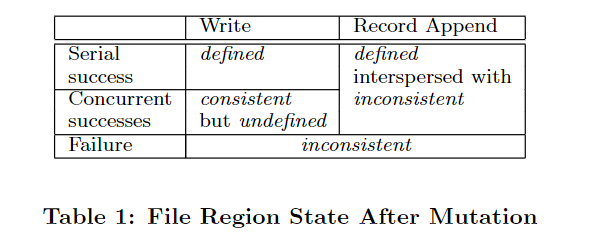
## Consistency Model

GFS 使用一个放宽的一致性模型不但很好的支持了我们的高度分布式的应用，而且实现起来也相对简单和有效率。我们现在讨论 GFS 所提供的保证以及它们对应用程序的意味着什么。我们也会讲述 GFS 如何维护这些保证，但是会将具体的细节留到其他论文里讲述。

### Guarantees by GFS

文件名字空间的改变(比如文件创建)是原子性的。它们只由 master 进行处理：名字空间锁来保证原子性和正确性(4.1 节)。Master 的操作日志定义了这些操作的全局性的排序。

当数据变更后，文件区域的状态取决于变更的类型，变更是否成功以及是否是并发进行的。表 1 是对结果的一个概述。



OS\_4\_2

**一致的（Consistent）：**：所有的客户端无论从哪个副本读取数据总是看到相同的数据。

**定义良好的（defined）**：一致的文件可以被客户端读取全部的变更。

操作影响：

1. 变更成功，且没有受到其他并发写者的影响，那么被影响的区域就是定义良好的（肯定是一致性的）。
2. 并发的成功变更，影响区域是一致的，但不是定义良好的，客户端可能无法看到所有的更改。(**如果变更是针对相同的数据写这样有的变更就会被新的变更所覆盖，这样用户就无法看到最先的变更了，同时发生在跨 chunk 的操作会被拆分成两个操作，这样这个操作的一部分可能会被其他操作覆盖，而另一部分则保留下来，如 3.1 节末尾所述**) ，通常其看到的是多个变更组合以后的结果。
3. 一个失败的变更，会使区域进入非一致的状态（因此也是未定义的状态）：不同的客户端在不同的访问中可能看到不同的数据。我们下面描述下我们的应用程序如何区分定义良好的区域和未定义的区域。应用程序不需要进一步区分未定义区域的各种不同的类型。

数据变更可能是写或者记录 append。写操作会使数据在应用程序指定的偏移位置写入。记录 append 操作会使数据原子性的 append，**如果是并发性的话则至少会被 append 一次**，但是偏移位置是由 GFS 决定的(见 3.3 节，通常的理解可能是在客户端想写入的那个文件的尾部)。偏移位置会被返回给客户端，同时标记包含这条记录的那个定义良好的文件区域的起始位置。另外 GFS 可能会在它们之间插入一些 padding 或者记录的副本。它们会占据那些被认为是不一致的区域，通常它们比用户数据小的多。

在一系列成功的变更之后，变更的文件区域被保证是**定义良好的**的，同时包含了最后一次变更的数据写入。GFS 通过两种方式来实现：

* 将这些变更以相同的操作顺序应用在该 chunk 的所有的副本上，
* 使用 chunk 的版本号来检测那些可能是由于它的 chunkserver 挂掉了而丢失了一些变更的陈旧副本。陈旧的副本永远都不会再参与变更或者返回给那些向 master 询问 chunk 位置的 client，它们会优先参与垃圾回收。

因为客户端会缓存 chunk 的位置，在信息更新之前它们可能会读到陈旧的副本。时间窗口由缓存值的超时时间以及文件的下一次打开（这种文件的打开会清除该文件相关的所有 chunk information）而限制。由于我们的大多数文件是 append-only 的，因此一个陈旧副本通常会返回一个过早结束信号（premature）的 chunk 而不是过时（outdated）的数据。当读取者（由 client 担当）重试并与 master 联系时，它会立即得到当前的 chunk 位置（更新相关的信息）。

成功的变更很久之后，组件失败仍有可能破坏或者污染数据。GFS 通过进行周期性的 master 和所有 chunkserver 的握手找到那些失败的 chunkserver，同时通过校验和(5.2 节)来检测数据的污染。一旦发现问题，会尽快的利用正确的副本恢复(4.3 节)。只有一个块的所有副本在 GFS 做出反应之前，全部丢失，这个块才会不可逆转的丢失，而通常 GFS 的反应是在几分钟内的。即使在这种情况下，块不可用，而不是被污染：应用程序会收到清晰的错误信息而不是被污染的数据。

### Implications for Applications

GFS 应用程序可以通过使用简单的技术来适应这种放宽的一致性模型，这些技术已经为其他目的所需要：依赖于 append 操作而不是 overwrite，检查点保存，写入时自我验证，自我标识记录。

典型应用一：一个写操作者会从头至尾生成一个文件。当写完所有数据后它自动的将文件重命名为一个永久性的名称，或者通过周期性的检查点检查已经有多少数据被成功写入了。检查点可以通过校验和实现。

典型应用二：多个写者同时向一个文件 append，为了归并/生产者消费者队列，记录的 append 的 append-at-least-once 语义保证了每个写者的输出。读取者这样处理偶然的 padding 和重复数据。写者为每条记录准备一些额外信息比如校验和，这样它的合法性就可以验证。如果不能容忍重复的数据（比如他们可以触发非幂等的操作，幂等操作是指无论执行多少次结果都会改变、而非幂等是指结果会改变），可以通过在记录中使用唯一标识符来过滤它们，很多时候都需要这些标识符命名相应的应用程序实体，比如网页文档。这些用于 record 输入输出的功能函数是以库的形式被我们的应用程序共享的，同时应用于 gongle 其他的文件接口实现。所以，相同系列的记录，加上一些罕见的重复，总是直接被分发给记录读取者。

# System Interactions

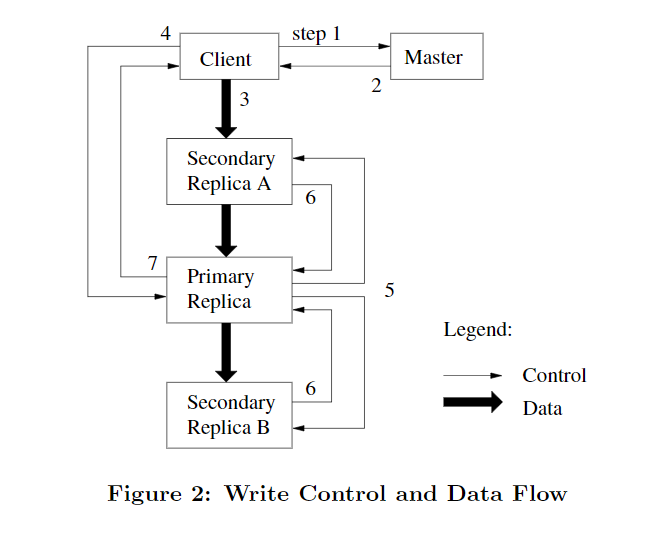
我们是以尽量最小化 master 在所有操作中的参与度来设计系统的。在这个背景下，我们现在描述下 client，master 以及 chunkserver 如何交互来实现数据变更，原子性 append 记录以及快照的。

## Leases and Mutation Order

每个变更在所有的副本上执行，我们使用租约（leases）来保持多个副本间变更顺序的一致性。Master 授权给其中的一个副本一个该 chunk 的租约，我们把它叫做主副本（primary）。这个主副本为针对该 chunk 的所有变更的选择一个执行顺序，然后所有的副本根据这个顺序执行变更。因此，全局的变更顺序首先是由 master 选择的租约授权顺序来确定的（可能有多个 chunk 需要进行修改），而同一个租约内的变更顺序则是由那个主副本（primary）来定义的。

租约机制是为了最小化 master 的管理开销而设计的。一个租约有一个初始化为 60s 的超时时间设置。然而只要这个 chunk 正在变更，那个主副本就可以向 master 请求延长租约。这些请求和授权通常是与 master 和 chunkserver 间的心跳信息（帮助 master 确定 chunkserver 的状态）一起发送的。有时候 master 可能想在租约过期前撤销它(比如，master 可能想使对一个正在重命名的文件的变更无效)。即使 master 无法与主副本进行通信，它也可以在旧的租约过期后安全的将租约授权给另一个新的副本。

如图 2，我们将用如下的数字标识的步骤来表示一个写操作的控制流程。



OS\_4\_3.png

1. client 向 master 询问那个 chunkserver 获取了当前 chunk 的租约以及其他副本所在的位置。如果没有人得到租约，master 将租约授权给它选择的一个副本。
2. master 返回该主副本的标识符以及其他副本的位置。Client 为未来的变更缓存这个数据。只有当主副本没有响应或者租约到期（60s 或者 master revoke）时它才需要与 master 联系。
3. client 将数据推送给所有的副本，client 可以以任意的顺序（针对副本的顺序）进行推送。每个 chunkserver 会将数据存放在内部的 LRU buffer cache 里，直到数据被使用或者过期（缓冲流）。通过将控制流与数据流分离，我们可以通过将昂贵的数据流基于网络拓扑进行调度来提高性能，而不用考虑哪个 chunkserver 是主副本。3.2 节更深入地讨论了这点。
4. 一旦所有的副本接受到了数据，client 发送一个写请求给主副本，这个请求标识了先前推送给所有副本的数据。主副本会给它收到的所有变更(可能来自多个 client)安排一个连续的序列号来进行必需的串行化。它将这些变更根据序列号应用在本地副本上。
5. 主副本将写请求发送给所有的次副本，每个次副本以与主副本相同的串行化顺序应用这些变更。
6. 所有的次副本完成操作后向主副本返回应答
7. 主副本向 client 返回应答。任何副本碰到的错误都会返回给 client。出现错误时，该写操作可能已经在主副本以及一部分次副本上执行成功。(如果主副本失败，那么它不会安排一个序列号并且发送给其他人)。客户端请求将会被认为是失败的，被修改的区域将会处在非一致状态下。我们的客户端代码会通过重试变更来处理这样的错误。它会首先在 3 到 7 步骤间进行一些尝试后在重新从头重试这个写操作。

如果应用程序的一个写操作很大或者跨越了 chunk 的边界，GFS client 代码会将它转化为多个写操作。它们都会遵循上面的控制流程，但是可能会被来自其他 client 的操作插入或者覆盖。因此共享的文件区域可能会包含来自不同 client 的片段，虽然这些副本是一致的，因为所有的操作都按照相同的顺序在所有副本上执行成功了。但是文件区域会处在一种一致但是未定义的状态，正如 2.7 节描述的那样（一致，但是由于并发 client 获取不到全部的操作信息，所以是未定义的）。

## Data Flow

为了更高效的使用网络，我们将数据流从控制流中分离出来。控制流从 client 到达主副本，然后到达其他的所有次副本，而数据则是线性地通过一个仔细选择的 chunkserver 链像流水线那样推送过去的。我们的目标是充分利用每个机器的网络带宽，避免网络瓶颈和高延时链路，最小化数据推送的延时。

为了充分利用每个机器的网络带宽，数据通过 chunkserver 链线性的推送过去而不是以其他的拓扑结构进行分布比如树型。因此每个机器的带宽可以全部用来发送数据而不是为多个接受者进行切分。

为了尽可能的避免网络瓶颈和高延时链路，每个机器向网络中还没有收到该数据的最近的那个机器推送数据。我们网络拓扑足够简单，以至于距离可以通过 IP 地址估计出来。

最后为了最小化延时，我们通过将 TCP 数据传输进行流水化。一旦一个 chunkserver 收到数据，它就开始立即往下发送数据。流水线对我们来说尤其有用，因为我们使用了一个全双工链路的交换网络。立即发送数据并不会降低数据接受速率。如果没有网络拥塞，向 个副本传输 字节的数据理想的时间耗费是 , 代表网络吞吐率， 是机器间的网络延时。我们的网络连接是 100Mbps(), 远远低于 1ms，因此 1MB 的数据理想情况下需要 80ms 就可以完成。

## Atomic Record Appends

GFS 提供一个原子性的 append 操作叫做 record append(注意这与传统的 append 操作也是不同的)。在传统的写操作中，用户指定数据需要写的便宜位置。对于相同区域的并行写操作是不可串行的：该区域的末尾可能包含来自多个 client 的数据片段。但在一个 record append 操作中，**client 唯一需要说明的只有数据**。GFS 会将它至少原子性地 append 到文件中一次，**append 的位置是由 GFS 选定的**，**同时会将这个位置返回给 client**。这很类似于 unix 文件打开模式中的 O\_APPEND，当多个写者并发操作时不会产生竞争条件。

Record append 在我们的分布式应用中被大量的使用。很多在不同机器的 client 并发地向同一个文件 append。如果使用传统的写操作，client 将需要进行复杂而又昂贵的同步化操作，比如通过一个分布式锁管理器。在我们的工作负载中，这样的文件通常作为一个多生产者/单消费者队列或者用来保存来自多个不同 client 的归并结果。

Record append 是一种类型的变更操作，除了一点点在主副本上的额外逻辑，其余依然遵循 3.1 节的控制流。Client 将所有的数据推送给所有副本后（3.1 第 4 步），它向主副本发送请求。主副本检查将该记录 append 到该 chunk 是否会导致该 chunk 超过它的最大值(64MB)。如果超过了，它就将该 chunk 填充到最大值，告诉次副本（scondaries）也填充到最大值，然后告诉客户端该操作应该在下一个 chunk 上重试。(append 的 Record 大小需要控制在最大 trunk 大小的四分之一以内，这样可以保证最坏情况下的碎片可以保持在一个可以接受的水平上 )。如果记录可以没有超过最大尺寸，就按照普通情况处理，主副本将数据 append 到它的副本上，告诉次副本将数据写在相同的偏移位置上，最后向 client 返回成功应答。

如果 record append 在任何一个副本上失败，client 就会重试这个操作。这样，相同 chunk 的多个副本就可能包含不同的数据，这些数据可能包含了相同记录的整个或者部分的重复值。GFS 并不保证所有的副本在位级别上的一致性，它只保证数据作为一个原子单元最少写入一次。这个属性是由如下的简单观察推导出来的，当操作报告成功时，数据肯定被写入到某个 trunk 的所有副本的相同偏移位置上。此后，所有的副本至少达到了记录尾部的大小，因此未来的记录将会被放置在更高的便宜位置，或者是另一个不同的 chunk，即使另一个副本变成了主副本。在我们的一致性保证里，record append 操作成功后写下的数据区域是已定义的(肯定是一致的)，然而介于其间的数据则是不一致的(因此也是未定义的)。我们的应用程序可以处理这样的不一致区域，正如我们在 2.7.2 里讨论的那样（对于少见的重复/应用应为记录添加一些额外的信息来保证其合法性，不能容忍的幂等操作，可以通过使用唯一标识符过滤）。

## Snapshot

快照操作可以非常快速的保存文件或者目录树的一个拷贝，同时可以最小化对于正在执行的变更操作的中断。用户经常用它来创建大数据集的分支拷贝，以及拷贝的拷贝。或者用来创建检查点，以实验接下来的提交或者回滚。

像 AFS，我们使用标准的 copy-on-write 技术来实现快照。当 master 收到一个快照请求时，它首先撤销将要进行快照的那些文件对应的 chunk 的所有已发出的租约。这就使得对于这些 chunk 的后续写操作需要与 master 交互来得到租约持有者。这就首先给 master 一个机会创建该 chunk 的新的拷贝（强迫与 master 交互以使新的租约生成）。

当这些租约被撤销或者过期后，master 将这些操作以日志形式写入磁盘。然后复制该文件或者目录树的元数据，然后将这些日志记录应用到内存中的复制后的状态上，新创建的快照文件与源文件一样指向相同的 chunk。

当 client 在快照生效后第一次对一个 chunk C 进行写入时，它会发送请求给 master 找到当前租约拥有者。Master 注意到对于 chunk C 的引用计数大于 1。它延迟回复客户端的请求，选择一个新的 chunk handle C’然后让每个拥有 C 的那些 chunkserver 创建一个新的叫做 C’的 chunk。通过在相同的 chunkserver 上根据原始的 chunk 创建新 chunk，就保证了数据拷贝是本地地，而不是通过网络(我们的硬盘比 100Mbps 网络快大概三倍)。这样，对于任何 chunk 的请求处理都没有什么不同：master 为新才 chunk C’的副本中的一个授权租约，然后返回给 client，这样它就可以正常的写这个 chunk 了，client 不需要知道该 chunk 实际上是从一个现有的 chunk 创建出来的。

# Master Operation

Master 执行所有的名字空间操作。此外，它还管理整个系统的 chunk replicas：决定如何放置，创建新的 chunk 和相应的副本，协调整个系统范围内的活动保证 chunk 都是完整备份的，在 chunkserver 间进行负载平衡，回收没有使用的存储空间。我们现在讨论这些主题。

## Namespace Management and Locking

很多 master 操作都需要花费很长时间：比如，一个快照操作要撤销该快照所包含的 chunk 的所有租约。我们并不想耽误其他运行中的 master 操作，因此我们允许多个操作同时是活动的，通过在名字空间区域使用锁来保证正确的串行化。

不像传统的文件系统，GFS 的目录并没有一种数据结构用来列出该目录下所有文件，而且也不支持文件或者目录别名（像 unix 的硬链接或者软连接那样）。GFS 在逻辑上将命名空间作为一个通过路径全称到元数据映射的查找表。通过采用前缀压缩，这个表可以有效地在内存中表示。名字空间树中的每个节点（要么是文件的绝对路径名称要么是目录的）具有一个相关联的读写锁。

每个 master 操作在它运行前，需要获得一个锁的集合。比如如果它想操作/d1/d2…/dn/leaf，那么它需要获得/d1,/d1/d2……/d1/d2…/dn 这些目录的读锁，然后才能得到路径/d1/d2…/dn/leaf 的读锁或者写锁。Leaf 可能是个文件或者目录，这取决于具体的操作。

我们现在解释一下，当为/home/user 创建快照/save/user 时，锁机制如何防止文件/home/user/foo 被创建。快照操作需要获得在/home /save 上的读锁，以及/home/user 和/save/user 上的写锁。文件创建需要获得在/home 和/home/user 上的读锁，以及在/home/user/foo 上的写锁。这两个操作将会被正确的串行化，因为它们试图获取在/home/user 上的相冲突的锁。文件创建并不需要父目录的写锁，因为实际上这里并没有”目录”或者说是类似于 inode 的数据结构，需要防止被修改。读锁已经足够用来防止父目录被删除。

（这种加锁机制的好处是对于同一个目录下，可以并行的操作文件，例如，同一个目录下并行的创建文件。快照会对新旧本层进行写加锁因而防止了对原始文件目录增加文件的操作产生的冲突。）

因为名字空间有很多节点，所以读写锁对象只有在需要时才会被分配，一旦不再使用用就删除。为了避免死锁，锁是按照一个一致的全序关系进行获取的：首先根据所处的名字空间树的级别，相同级别的则根据字典序。

## Replica Placement

如何安置 replicas 的目标是：

* 最大化数据可靠性和可用性
* 最大化网络带宽的利用

**这里的最大化不仅仅是机器间的问题**，还要考虑机架间的问题

在以下 3 种情况下，Master 会进行创建 replicas 的操作：

* 创建了新的 chunk
* 需要重新备份
* 负载均衡

如何选择将 replicas 放置到哪台机器上呢？

* 优先选择磁盘利用率低的 chunkserver，使得较长时间将会平均化 chunkserver 的利用。
* GFS 会限制每个 chunkserver『最近』创建的次数。换句话说，如果一个 chunkserver 近期创建 replicas 的操作比较频繁，就不会优先选择它（因为创建就意味着以后会进行读取，为了防止突然间大量的读取出现在同一台机器上）
* 保证可用性，尽可能跨机架（racks）进行创建操作

当可用的备份低于要求时（GFS 要求为 3 份），master 会对 chunk 进行重新备份，在以下情况有可能需要重新备份：

* chunkserver 不可用了
* 备份损坏了
* 硬盘挂掉了
* 所要求的最低备份数量提高了

当有多个 chunk 需要备份时，GFS 如何决定先备份哪个呢？策略如下：

* 优先选择可用备份少的（仅有一份备份的更加迫切）
* 优先备份最近没有 delete 文件的（for live files）
* 优先备份阻塞了 client 操作的（这样可以最小化对运行中应用的影响）

当 master 决定了备份哪个之后，会把当前可用的 chunk 直接克隆到目标位置（遵循 replicas 放置的类似规则：平均磁盘利用率、防止单一机器高 IO、跨机架）。

周期性的检查副本分布以求重平衡，副本的移动过程中是新建一个新的，放置规则和前述讨论一致。放置了新的必须删除旧的（通常情况下倾向删除自身空白空间低于平均水平的）。

## Garbage Collection

文件删除后，GFS 并不立即释放可用的物理存储（进行物理删除，而是等对应的垃圾清理进行到文件和 chunk 级别时再真正删除（Lazy）。

### Mechanism

具体地，对于一个文件的删除操作，GFS 仅仅是写一条日志记录，然后把文件命名成一个对外部不可见的名称，这个名称会包含删除的时间戳。GFS master 会定期的扫描，当这些文件存在超过 3 天（可设定）后，这些文件会从 namespace 中删掉，并且内存的中 metadata 会被删除，切断了和 chunk 的联系。

在对 chunk namespace 的定期扫描时，master 找到那些孤儿块(无法从任何文件到达)，擦除这些块的元数据。在与 chunkserver 的 heartbeat 的交互过程中，GFS master 会把不在 metadata 中的 chunk 告诉 chunkserver，然后 chunkserver 对照后就可以删除这些 chunk 了。

## Discussion

master 在内存中储存那个查找表（前缀压缩过的），保存了所有文件-chunck 映射，因而很容易找到 chunck 的引用。chunck 只不过是放在特定目录下的 linux 文件，master 不知道的副本就是 garbage。

采用这种方式删除的好处：

1. 利用心跳方式交互，在一次删除失败后，还可以通过下次心跳继续重试操作
2. 删除操作和其他的全局扫描 metadata 的操作可以放到一起做 为删除这种不可逆转操作提供了一定弹性恢复的保护网。 坏处：

有可能有的应用需要频繁的创建和删除文件，这种延期删除方式会导致磁盘使用率偏高，GFS 提供的解决方案是，对一个文件调用删除操作两次，GFS 会马上做物理删除操作，释放空间。同时允许用户在不同名字空间内使用不同的重备份和回收策略（不执行副本和直接物理删除）