Google File System 总结 | Robert Peng's Blog

Created	@Jun 29, 2020 5:36 PM
Tags	
URL	https://mr-dai.github.io/gfs/

这篇文章是本人按照 MIT 6.824 的课程安排阅读 Google File System 的论文以及相关课程资料并总结而来。

GFS 的主要需求

在学习 GFS 的原理前，首先我们应当了解 GFS 在设计时所面对的需求场景。简单概括，GFS 的设计主要基于以下几个需求：

- 节点失效是常态。系统会构建在大量的普通机器上，这使得节点失效的可能性很高。因此，GFS 必须能有较高的容错性、能够持续地监控自身的状态，同时还要能够顺畅地从节点失效中快速恢复
- **存储内容以大文件为主**。系统需要存储的内容在通常情况下由数量不多的大文件构成，每个文件通常有几百 MB 甚至是几 GB 的大小；系统应当支持小文件，但不需要为其做出优化
- 主要负载为**大容量连续读**、**小容量随机读**以及**追加式的连续写**
- 系统应当支持**高效且原子的文件追加操作**，源于在 Google 的情境中，这些文件多用于生产者-消费者模式或是多路归并
- 当需要做出取舍时，系统应**选择高数据吞吐量而不是低延时**

GFS 集群组成

本章会先给大家介绍一下一个 GFS 集群的基本组成以及各个组件的基本职责。

简单来讲，除了客户端以外，一个 GFS 集群还包括一个 **Master** 节点和若干个 **Chunk Server**。它们会作为用户级进程运行在普通的 Linux 机器上。

在存储文件时，GFS 会把文件切分成若干个拥有固定长度的 Chunk（块）并存储。Master 在创建 Chunk 时会为它们赋予一个唯一的 64 位 Handle（句柄），并把它们移交给 Chunk Server，而 Chunk Server 则以普通文件的形式将每个 Chunk 存储在自己的本地磁盘上。为了确保 Chunk 的可用性，GFS 会把每个 Chunk 备份成若干个 Replica 分配到其他 Chunk Server 上。

GFS 的 Master 负责维护整个集群的元数据，包括集群的 Namespace（命名空间，即文件元数据）以及 Chunk Lease 管理、无用 Chunk 回收等系统级操作。Chunk Server 除了保存 Chunk 以外也会周期地和 Master 通过心跳信号进行通信，Master 也借此得以收集每个 Chunk Server 当前的状态，并向其发送指令。

鉴于整个集群只有一个 Master，客户端在和 GFS 集群通信时，首先会从 Master 处获取 GFS 的元数据，而实际文件的数据传输则会与 Chunk Server 直接进行，以避免 Master 成为整个系统的数据传输瓶颈；除此以外，客户端也会在一定时间内缓存 Master 返回的集群元数据。

GFS 的元数据

GFS 集群的所有元数据都会保存在 Master 的内存中。鉴于整个集群只会有一个 Master，这也使得元数据的管理变得更为简单。GFS 集群的元数据主要包括以下三类信息：

- 文件与 Chunk 的 Namespace
- 文件与 Chunk 之间的映射关系
- 每个 Chunk Replica 所在的位置

元数据保存在 Master 的内存中使得 Master 要对元数据做出变更变得极为容易；同时，这也使得 Master 能够更加高效地扫描集群的元数据，以唤起 Chunk 回收、Chunk 均衡等系统级管理操作。唯一的不足在于这使得整个集群所能拥有的 Chunk 数量受限于 Master 的内存大小，不过从论文的内容来看，这样的瓶颈在 Google 中从来没有被触及过，源于对于一个 64MB 大小的 Chunk，Master 只需要维持不到 64 字节的元数据。况且，相比于增加代码的复杂度，提高 Master 内存容量的成本要小得多。

为了保证元数据的可用性，Master 在对元数据做任何操作前对会用**先写日志**的形式将操作进行记录，日志写入完成后再进行实际操作，而这些日志也会被备份到多个机器上进行保存。不过，Chunk Replica 的位置不会被持久化到日志中，而是由 Master 在启动时询问各个 Chunk Server 其当前所有的 Replica。这可以省去 Master 与 Chunk Server 同步数据的成本，同时进一步简化 Master 日志持久化的工作。这样的设计也是合情合理

的，毕竟 Chunk Server 当前实际持有哪些 Replica 也应由 Chunk Server 自己说了算。

数据一致性

用户在使用 GFS 这类数据存储系统时，首先应当了解其所能提供的数据一致性，而作为学习者我们也应先理解 GFS 对外呈现的数据一致性功能。

首先，命名空间完全由单节点 Master 管理在其内存中，这部分数据的修改可以通过让 Master 为其添加互斥锁来解决并发修改的问题，因此命名空间的数据修改是可以确保完全原子的。

文件的数据修改则相对复杂。在讲述接下来的内容前，首先我们先明确，在文件的某一部分被修改后，它可能进入以下三种状态的其中之一：

- 客户端读取不同的 Replica 时可能会读取到不同的内容，那这部分文件是**不一致**的（Inconsistent）
- 所有客户端无论读取哪个 Replica 都会读取到相同的内容，那这部分文件就是**一致**的（Consistent）
- 所有客户端都能看到上一次修改的所有完整内容，且这部分文件是一致的，那么我们说这部分文件是**确定**的（Defined）

在修改后，一个文件的当前状态将取决于此次修改的类型以及修改是否成功。具体来说：

- 如果一次写入操作成功且没有与其他并发的写入操作发生重叠，那这部分的文件是**确定**的（同时也是一致的）
- 如果有若干个写入操作并发地执行成功，那么这部分文件会是**一致**的但会是**不确定**的：在这种情况下，客户端所能看到的数据通常不能直接体现出其中的任何一次修改
- 失败的写入操作会让文件进入**不一致**的状态

这之间的关系也被整理为了论文中的表格 1：

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

Table 1: File Region State After Mutation

GFS 支持的文件数据修改数据包括两种：指定偏移值的数据写入（Write）以及数据追加（Record Append）。当写入时，指定的数据会被直接写入到客户端指定的偏移位置中，覆盖原有的数据。GFS 并未为该操作提供太多的一致性保证：如果不同的客户端并发地写入同一块文件区域，操作完成后这块区域的数据可能由各次写入的数据碎片所组成，即进入**不确定**的状态。

与写入操作不同，GFS 确保即便是在并发时，数据追加操作也是原子且 at least once（至少一次）的。操作完成后，GFS 会把实际写入的偏移值返回给客户端，该偏移值即代表包含所写入数据的**确定**的文件区域的起始位置。由于数据追加操作是 at least once 的，GFS 有可能会在文件中写入填充（padding）或是重复数据，但出现的概率不高。

在读取数据时，为了避免读入填充数据或是损坏的数据，数据在写入前往往会放入一些如校验和等元信息以用于验证其可用性，如此一来 GFS 的客户端 library 便可以在读取时自动跳过填充和损坏的数据。不过，鉴于数据追加操作的 at lease once 特性，客户端仍有可能读入重复的数据，此时只能由上层应用通过鉴别记录的唯一 ID 等信息来过滤重复数据了。

对应用的影响

GFS 的一致性模型是相对松散的，这就要求上层应用在使用 GFS 时能够适应 GFS 所提供的一致性语义。简单来讲，上层应用可以通过两种方式来做到这一点：更多使用追加操作而不是覆写操作；写入包含校验信息的数据。

青睐追加操作而不是覆写操作的原因是明显的：GFS 针对追加操作做出了显著的优化，这使得这种数据写入方式的性能更高，而且也能提供更强的一致性语义。尽管如此，追加操作 at least once 的特性仍使得客户端可能读取到填充或是重复数据，这要求客户端能够容忍这部分无效数据。一种可行的做法是在写入的同时为所有记录写入各自的校验和，

并在读取时进行校验，以剔除无效的数据；如果客户端无法容忍重复数据，客户端也可以在写入时为每条记录写入唯一的标识符，以便在读取时通过标识符去除重复的数据。

GFS 集群常见操作流程

Master Namespace 管理

在前面我们已经了解到，Namespace 作为 GFS 元信息的一部分会被维持在 Master 的内存中，由 Master 负责管理。在逻辑上，GFS Master 并不会根据文件与目录的关系以分层的结构来管理这部分数据，而是单纯地将其表示为从完整路径名到对应文件元数据的映射表，并在路径名上应用前缀压缩以减少内存占用。

为了管理来自不同客户端的并发请求对 Namespace 的修改，Master 会为 Namespace 中的每个文件和目录都分配一个读写锁（Read-Write Lock）。由此，对不同 Namespace 区域的并发请求便可以同时进行。

所有 Master 操作在执行前都会需要先获取一系列的锁：通常，当操作涉及某个路径 `/d1/d2/.../dn/leaf` 时，Master 会需要先获取从 `/d1`、`/d1/d2` 到 `/d1/d2/.../dn` 的读锁，然后再根据操作的类型获取 `/d1/d2/.../dn/lead` 的读锁或写锁 —— 获取父目录的读锁是为了避免父目录在此次操作执行的过程中被重命名或删除。

由于大量的读写锁可能会造成较高的内存占用，这些锁会在实际需要时才进行创建，并在不再需要时被销毁。除外，所有的锁获取操作也会按照一个相同的顺序进行，以避免发生死锁：锁首先按 Namespace 树的层级排列，同一层级内则以路径名字典序排列。

读取文件

客户端从 GFS 集群中读取文件内容的过程大致如下：

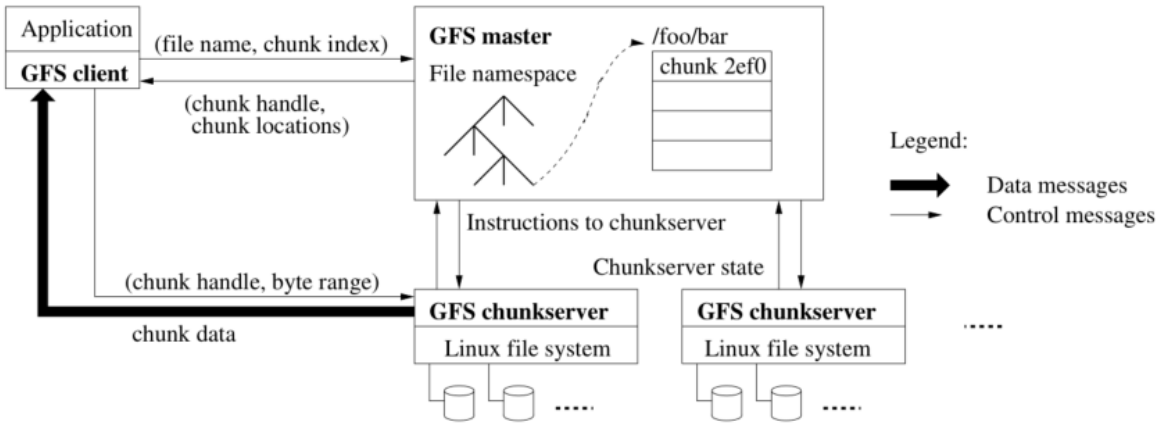

Figure 1: GFS Architecture

1. 对于指定的文件名和读取位置偏移值，客户端可以根据固定的 Chunk 大小来计算出该位置在该文件的哪一个 Chunk 中
2. 客户端向 Master 发出请求，其中包含要读取的文件名以及 Chunk 索引值
3. Master 向客户端响应该 Chunk 的 Handle 以及其所有 Replica 当前所在的位置。客户端会以文件名和 Chunk 索引值为键缓存该数据
4. 之后，客户端便可以选取其中一个 Replica 所在的 Chunk Server 并向其发起请求，请求中会指定需要读取的 Chunk 的 Handle 以及要读取的范围

Chunk Lease

在客户端对某个 Chunk 做出修改时，GFS 为了能够处理不同的并发修改，会把该 Chunk 的 Lease 交给某个 Replica，使其成为 Primary：Primary 会负责为这些修改安排一个执行顺序，然后其他 Replica 便按照相同的顺序执行这些修改。

Chunk Lease 在初始时会有 60 秒的超时时间。在未超时前，Primary 可以向 Master 申请延长 Chunk Lease 的时间；必要时 Master 也可以直接撤回已分配的 Chunk Lease。

文件写入

客户端尝试将数据写入到某个 Chunk 的指定位置的过程大致如下：

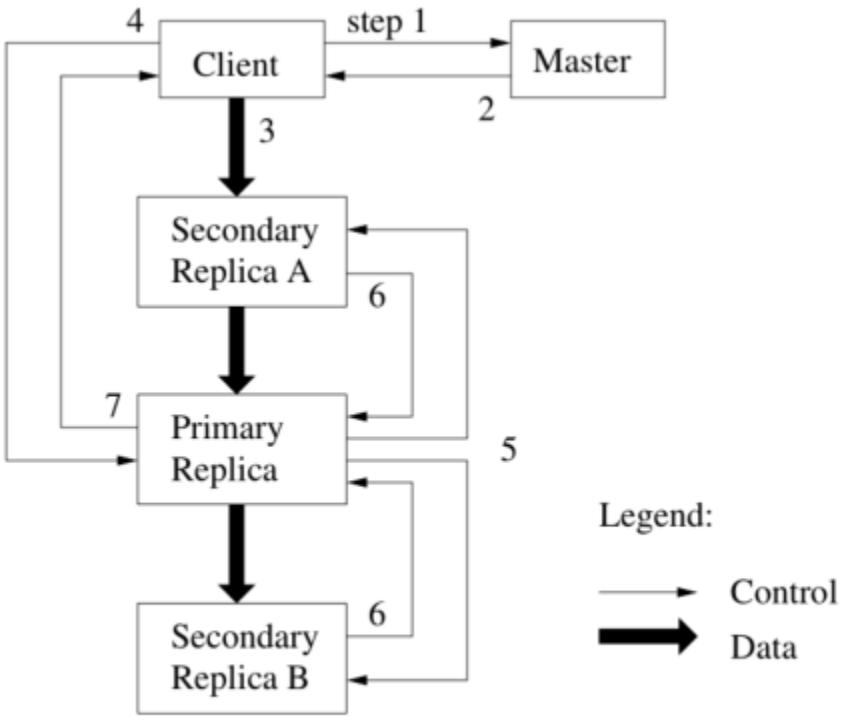

Figure 2: Write Control and Data Flow

1. 客户端向 Master 询问目前哪个 Chunk Server 持有该 Chunk 的 Lease
2. Master 向客户端返回 Primary 和其他 Replica 的位置
3. 客户端将数据推送到所有的 Replica 上。Chunk Server 会把这些数据保存在缓冲区中，等待使用
4. 待所有 Replica 都接收到数据后，客户端发送写请求给 Primary。Primary 为来自各个客户端的修改操作安排连续的执行序列号，并按顺序地应用于其本地存储的数据
5. Primary 将写请求转发给其他 Secondary Replica，Replica 们按照相同的顺序应用这些修改
6. Secondary Replica 响应 Primary，示意自己已经完成操作
7. Primary 响应客户端，并返回该过程中发生的错误（若有）

如果该过程有发生错误，可以认为修改已在 Primary 和部分 Secondary 上成功执行（如果在 Primary 上就出错了，那么写请求不会被转发出去）。此时可以认为此次修改操作没有成功，因为数据会处于**不一致**的状态。实际上，GFS 所使用的客户端 lib 在此时会重新尝试执行此次操作。

值得注意的是，这个流程特意将数据流与控制流分开：客户端先向 Chunk Server 提交数据，再将写请求发往 Primary。这么做的好处在于 GFS 能够更好地利用网络带宽资源。

从上述步骤可见，控制流借由写请求从客户端流向 Primary，再流向其他 Secondary Replica。实际上，数据流以一条线性数据管道进行传递的：客户端会把数据上传到离自己最近的 Replica，该 Replica 在接收到数据后再转发给离自己最近的另一个 Replica，如此递归直到所有 Replica 都能接收到数据，如此一来便能够利用上每台机器的所有出口带宽。

文件追加

文件追加操作的过程和写入的过程有几分相似：

1. 客户端将数据推送到每个 Replica，然后将请求发往 Primary
2. Primary 首先判断将数据追加到该块后是否会令块的大小超过上限：如果是，那么 Primary 会为该块写入填充至其大小达到上限，并通知其他 Replica 执行相同的操作，再响应客户端，通知其应在下一个块上重试该操作
3. 如果数据能够被放入到当前块中，那么 Primary 会把数据追加到自己的 Replica 中，拿到追加成功返回的偏移值，然后通知其他 Replica 将数据**写入**到该偏移位置中
4. 最后 Primary 再响应客户端

当追加操作在部分 Replica 上执行失败时，Primary 会响应客户端，通知它此次操作已失败，客户端便会重试该操作。和写入操作的情形相同，此时已有部分 Replica 顺利写入这些数据，重新进行数据追加便会导致这一部分的 Replica 上出现重复数据，不过 GFS 的一致性模型也确实并未保证每个 Replica 都会是完全一致的。

GFS 只确保数据会以一个原子的整体被追加到文件中至少一次。由此我们可以得出，当追加操作成功时，数据必然已被写入到所有 Replica 的相同偏移位置上，且每个 Replica 的长度都至少超出此次追加的记录的尾部，下一次的追加操作必然会被分配一个比该值更大的偏移值，或是被分配到另一个新的块上。

文件快照

GFS 还提供了文件快照操作，可为指定的文件或目录创建一个副本。

快照操作的实现采用了写时复制（Copy on Write）的思想：

1. 在 Master 接收到快照请求后，它首先会撤回这些 Chunk 的 Lease，以让接下来其他客户端对这些 Chunk 进行写入时都会需要请求 Master 获知 Primary 的位置，Master 便可利用这个机会创建新的 Chunk

2. 当 Chunk Lease 撤回或失效后，Master 会先写入日志，然后对自己管理的命名空间进行复制操作，复制产生的新记录指向原本的 Chunk
3. 当有客户端尝试对这些 Chunk 进行写入时，Master 会注意到这个 Chunk 的引用计数大于 1。此时，Master 会为即将产生的新 Chunk 生成一个 Handle，然后通知所有持有这些 Chunk 的 Chunk Server 在本地复制出一个新的 Chunk，应用上新的 Handle，然后再返回给客户端

Replica 管理

为了进一步优化 GFS 集群的效率，Master 在 Replica 的位置选取上会采取一定的策略。

Master 的 Replica 编排策略主要为了实现两个目标：最大化数据的可用性，以及最大化网络带宽的利用率。为此，Replica 不仅需要被保存在不同的机器上，还会需要被保存在不同的机架上，这样如果整个机架不可用了，数据仍然得以存活。如此一来，不同客户端对同一个 Chunk 进行读取时便可以利用上不同机架的出口带宽，但坏处就是进行写入时数据也会需要在不同机架间流转，不过在 GFS 的设计者看来这是个合理的 trade-off。

Replica 的生命周期转换操作实际只有两个：创建和删除。首先，Replica 的创建可能源于以下三种事件：创建 Chunk、为 Chunk 重备份、以及 Replica 均衡。

在 Master 创建一个新的 Chunk 时，首先它会需要考虑在哪放置新的 Replica。Master 会考虑如下几个因素：

1. Master 会倾向于把新的 Replica 放在磁盘使用率较低的 Chunk Server 上
2. Master 会倾向于确保每个 Chunk Server 上“较新”的 Replica 不会太多，因为新 Chunk 的创建意味着接下来会有大量的写入，如果某些 Chunk Server 上有太多的新 Chunk Replica，那么写操作压力就会集中在这些 Chunk Server 上
3. 如上文所述，Master 会倾向于把 Replica 放在不同的机架上

当某个 Chunk 的 Replica 数量低于用户指定的阈值时，Master 就会对该 Chunk 进行重备份。这可能是由 Chunk Server 失效、Chunk Server 回报 Replica 数据损坏或是用户提高了 Replica 数量阈值所触发。

首先，Master 会按照以下因素为每个需要重备份的 Chunk 安排优先级：

1. 该 Chunk 的 Replica 数距离用户指定的 Replica 数量阈值的差距有多大
2. 优先为未删除的文件（见下文）的 Chunk 进行重备份
3. 除外，Master 还会提高任何正在阻塞用户操作的 Chunk 的优先级

有了 Chunk 的优先级后，Master 会选取当前拥有最高优先级的 Chunk，并指定若干 Chunk Server 直接从现在已有的 Replica 上复制数据。Master 具体会指定哪些 Chunk Server 进行复制操作同样会考虑上面提到的几个因素。除外，为了减少重备份对用户使用的影响，Master 会限制当前整个集群正在进行的复制操作的数量，同时 Chunk Server 也会限制复制操作所使用的带宽。

最后，Master 会周期地检查每个 Chunk 当前在集群内的分布情况，并在必要时迁移部分 Replica 以更好地均衡各节点的磁盘利用率和负载。新 Replica 的位置选取策略和上面提到的大体相同，除此以外 Master 还会需要选择要移除哪个已有的 Replica：简单概括的话，Master 会倾向于移除磁盘占用较高的 Chunk Server 上的 Replica，以均衡磁盘使用率。

删除文件

当用户对某个文件进行删除时，GFS 不会立刻删除数据，而是在文件和 Chunk 两个层面上都 lazy 地对数据进行移除。

首先，当用户删除某个文件时，GFS 不会从 Namespace 中直接移除该文件的记录，而是将该文件重命名为另一个隐藏的名称，并带上删除时的时间戳。在 Master 周期扫描 Namespace 时，它会发现那些已被“删除”较长时间，如三天，的文件，这时候 Master 才会真正地将其从 Namespace 中移除。在文件被彻底从 Namespace 删除前，客户端仍然可以利用这个重命名后的隐藏名称读取该文件，甚至再次将其重命名以撤销删除操作。

Master 在元数据中有维持文件与 Chunk 之间的映射关系：当 Namespace 中的文件被移除后，对应 Chunk 的引用计数便自动减 1。同样是在 Master 周期扫描元数据的过程中，Master 会发现引用计数已为 0 的 Chunk，此时 Master 便会从自己的内存中移除与这些 Chunk 有关的元数据。在 Chunk Server 和 Master 进行的周期心跳通信中，Chunk Server 会汇报自己所持有的 Chunk Replica，此时 Master 便会告知 Chunk Server 哪些 Chunk 已不存在于元数据中，Chunk Server 则可自行移除对应的 Replica。

采用这种删除机制主要有如下三点好处：

1. 对于大规模的分布式系统来说，这样的机制更为**可靠**：在 Chunk 创建时，创建操作可能在某些 Chunk Server 上成功了，在其他 Chunk Server 上失败了，这导致某些 Chunk Server 上可能存在 Master 不知道的 Replica。除此以外，删除 Replica 的请求可能会发送失败，Master 会需要记得尝试重发。相比之下，由 Chunk Server 主动地删除 Replica 能够以一种更为统一的方式解决以上的问题

2. 这样的删除机制将存储回收过程与 Master 日常的周期扫描过程合并在了一起，这就使得这些操作可以以批的形式进行处理，以减少资源损耗；除外，这样也得以让 Master 选择在相对空闲的时候进行这些操作
3. 用户发送删除请求和数据被实际删除之间的延迟也有效避免了用户误操作的问题

不过，如果在存储资源较为稀缺的情况下，用户对存储空间使用的调优可能就会受到该机制的阻碍。为此，GFS 允许客户端再次指定删除该文件，以确实地从 Namespace 层移除该文件。除外，GFS 还可以让用户为 Namespace 中不同的区域指定不同的备份和删除策略，如限制 GFS 不对某个目录下的文件进行 Chunk 备份等。

高可用机制

前面我们提到，Master 会以先写日志（Operation Log）的形式对集群元数据进行持久化：日志在被确实写出前，Master 不会对客户端的请求进行响应，后续的变更便不会继续执行；除外，日志还会被备份到其他的多个机器上，日志只有在写入到本地以及远端备份的持久化存储中才被视为完成写出。

在重新启动时，Master 会通过重放已保存的操作记录来恢复自身的状态。为了保证 Master 能够快速地完成恢复，Master 会在日志达到一定大小后为自身的当前状态创建 Checkpoint（检查点），并删除 Checkpoing 创建以前的日志，重启时便从最近一次创建的 Checkpoint 开始恢复。Checkpoint 文件的内容会以 B 树的形式进行组织，且在被映射到内存后便能够在不做其他额外的解析操作的情况下检索其所存储的 Namespace，这便进一步减少了 Master 恢复所需的时间。

为了简化设计，同一时间只会有一个 Master 起作用。当 Master 失效时，外部的监控系统会侦测到这一事件，并在其他地方重新启动新的 Master 进程。

除外，集群中还会有其他提供只读功能的 **Shadow Master**：它们会同步 Master 的状态变更，但有可能延迟若干秒，其主要用于为 Master 分担读操作的压力。Shadow Master 会通过读取 Master 操作日志的某个备份来让自己的状态与 Master 同步；它也会像 Master 那样，在启动时轮询各个 Chunk Server，获知它们所持有的 Chunk Replica 信息，并持续监控它们的状态。实际上，在 Master 失效后，Shadow Master 仍能为整个 GFS 集群提供只读功能，而 Shadow Master 对 Master 的依赖只限于 Replica 位置的更新事件。

作为集群中的 Slave 角色，Chunk Server 失效的几率比 Master 要大得多。在前面我们已经提到，Chunk Server 失效时，其所持有的 Replica 对应的 Chunk 的 Replica 数量便会降低，Master 也会发现 Replica 数量低于用户指定阈值的 Chunk 并安排重备份。

除外，当 Chunk Server 失效时，用户的写入操作还会不断地进行，那么当 Chunk Server 重启后，Chunk Server 上的 Replica 数据便有可能是已经过期的。为此，Master 会为每个 Chunk 维持一个版本号，以区分正常的和过期的 Replica。每当 Master 将 Chunk Lease 分配给一个 Chunk Server 时，Master 便会提高 Chunk 的版本号，并通知其他最新的 Replica 更新自己的版本号。如果此时有 Chunk Server 失效了，那么它上面的 Replica 的版本号就不会变化。

在 Chunk Server 重启时，Chunk Server 会向 Master 汇报自己所持有的 Chunk Replica 及对应的版本号。如果 Master 发现某个 Replica 版本号过低，便会认为这个 Replica 不存在，如此一来这个过期的 Replica 便会在下一次的 Replica 回收过程中被移除。除外，Master 向客户端返回 Replica 位置信息时也会返回 Chunk 当前的版本号，如此一来客户端便不会读取到旧的数据。

数据完整性

如前面所述，每个 Chunk 都会以 Replica 的形式被备份在不同的 Chunk Server 中，而且用户可以为 Namespace 的不同部分赋予不同的备份策略。

为了保证数据完整，每个 Chunk Server 都会以校验和的形式来检测自己保存的数据是否有损坏；在侦测到损坏数据后，Chunk Server 也可以利用其它 Replica 来恢复数据。

首先，Chunk Server 会把每个 Chunk Replica 切分为若干个 64KB 大小的块，并为每个块计算 32 位校验和。和 Master 的元数据一样，这些校验和会被保存在 Chunk Server 的内存中，每次修改前都会用先写日志的形式来保证可用。当 Chunk Server 接收到读请求时，Chunk Server 首先会利用校验和检查所需读取的数据是否有发生损坏，如此一来 Chunk Server 便不会把损坏的数据传递给其他请求发送者，无论它是客户端还是另一个 Chunk Server。发现损坏后，Chunk Server 会为请求发送者发送一个错误，并向 Master 告知数据损坏事件。接收到错误后，请求发送者会选择另一个 Chunk Server 重新发起请求，而 Master 则会利用另一个 Replica 为该 Chunk 进行重备份。当新的 Replica 创建完成后，Master 便会通知该 Chunk Server 删除这个损坏的 Replica。

当进行数据追加操作时，Chunk Server 可以为位于 Chunk 尾部的校验和块的校验和进行增量式的更新，或是在产生了新的校验和块时为其计算新的校验和。即使是被追加的校验和块在之前已经发生了数据损坏，增量更新后的校验和依然会无法与实际的数据相匹配，在下一次读取时依然能够检测到数据的损坏。在进行数据写入操作时，Chunk Server 必须读取并校验包含写入范围起始点和结束点的校验和块，然后进行写入，最后再重新计算校验和。

除外，在空闲的时候，Chunk Server 也会周期地扫描并校验不活跃的 Chunk Replica 的数据，以确保某些 Chunk Replica 即使在不怎么被读取的情况下，其数据的损坏依然能被检测到，同时也确保了这些已损坏的 Chunk Replica 不至于让 Master 认为该 Chunk 已有足够数量的 Replica。

附录

节点缓存

在 GFS 中，客户端和 Chunk Server 都不会对文件数据进行缓存。对于客户端而言，考虑到大多数应用都会选择顺序读取某些大文件，缓存的作用微乎其微，不过客户端确实会缓存 GFS 的元数据以减少和 Master 的通信；对于 Chunk Server 来说，缓存文件数据也是不必要的，因为这些内容本身就保存在它的本地磁盘上，Linux 内核的缓存机制也会把经常访问的磁盘内容放置在内存中。

对于 GFS 而言，Chunk 的大小是一个比较重要的参数，而 GFS 选择了使用 64MB 作为 Chunk 的大小。

较大的 Chunk 主要带来了如下几个好处：

1. 降低客户端与 Master 通信的频率
2. 增大客户端进行操作时这些操作落到同一个 Chunk 上的概率
3. 减少 Master 所要保存的元数据的体积

不过，较大的 Chunk 会使得小文件占据额外的存储空间；一般的小文件通常只会占据一个 Chunk，这些 Chunk 也容易成为系统的负载热点。但正如之前所设想的需求那样，这样的文件在 Google 的场景下不是普遍存在的，这样的问题并未在 Google 中真正出现过。即便真的出现了，也可以通过提升这类文件的 Replica 数量来将负载进行均衡。

组件的快速恢复

GFS 的组件在设计时着重提高了状态恢复的速度，通常能够在几秒钟内完成启动。在这样的保证下，GFS 的组件实际上并不对正常关闭和异常退出做区分：要关闭某个组件时直接 `kill -9` 即可。

FAQ

MIT 6.824 的课程材料中给出了和 GFS 有关的 FAQ，在此我简单地翻译一下其中比较重要的一些内容。

> Q：为什么原子记录追加操作是至少一次（At Least Once），而不是确定一次（Exactly Once）？

要让追加操作做到确定一次是不容易的，因为如此一来 Primary 会需要保存一些状态信息以检测重复的数据，而这些信息也需要复制到其他服务器上，以确保 Primary 失效时这些信息不会丢失。在 Lab 3 中你会实现确定一次的行为，但用的是比 GFS 更复杂的协议（Raft）。

> Q：应用怎么知道 Chunk 中哪些是填充数据或者重复数据?

要想检测填充数据，应用可以在每个有效记录之前加上一个魔数（Magic Number）进行标记，或者用校验和保证数据的有效性。应用可通过在记录中添加唯一 ID 来检测重复数据，这样应用在读入数据时就可以利用已经读入的 ID 来排除重复的数据了。GFS 本身提供了 library 来支撑这些典型的用例。

> Q：考虑到原子记录追加操作会把数据写入到文件的一个不可预知的偏移值中，客户端该怎么找到它们的数据?

追加操作（以及 GFS 本身）主要是面向那些会完整读取文件的应用的。这些应用会读取所有的记录，所以它们并不需要提前知道记录的位置。例如，一个文件中可能包含若干个并行的网络爬虫获取的所有链接 URL。这些 URL 在文件中的偏移值是不重要的，应用只会想要完整读取所有 URL。

> Q：如果一个应用使用了标准的 POSIX 文件 API，为了使用 GFS 它会需要做出修改吗?

答案是需要的，不过 GFS 并不是设计给已有的应用的，它主要面向的是新开发的应用，如 MapReduce 程序。

> Q：GFS 是怎么确定最近的 Replica 的位置的?

论文中有提到 GFS 是基于保存 Replica 的服务器的 IP 地址来判断距离的。在 2003 年的时候，Google 分配 IP 地址的方式应该确保了如果两个服务器的 IP 地址在 IP 地址空间中较为接近，那么它们在机房中的位置也会较为接近。

Q：Google 现在还在使用 GFS 吗？

Google 仍然在使用 GFS，而且是作为其他如 BigTable 等存储系统的后端。由于工作负载的扩大以及技术的革新，GFS 的设计在这些年里无疑已经经过大量调整了，但我并不了解其细节。HDFS 是公众可用的对 GFS 的设计的一种效仿，很多公司都在使用它。

Q：Master 不会成为性能瓶颈吗？

确实有这个可能，GFS 的设计者也花了很多心思来避免这个问题。例如，Master 会把它的状态保存在内存中以快速地进行响应。从实验数据来看，对于大文件读取（GFS 主要针对的负载类型），Master 不是瓶颈所在；对于小文件操作以及目录操作，Master 的性能也还跟得上（见 6.2.4 节）。

Q：GFS 为了性能和简洁而牺牲了正确性，这样的选择有多合理呢？

这是分布式系统领域的老问题了。保证强一致性通常需要更加复杂且需要机器间进行更多通信的协议（正如我们会在接下来几门课中看到的那样）。通过利用某些类型的应用可以容忍较为松懈的一致性的事实，人们就能够设计出拥有良好性能以及足够的一致性的系统。例如，GFS 对 MapReduce 应用做出了特殊优化，这些应用需要的是对大文件的高读取效率，还能够容忍文件中存在数据空洞、重复记录或是不一致的读取结果；另一方面，GFS 则不适用于存储银行账号的存款信息。

Q：如果 Master 失效了会怎样？

GFS 集群中会有持有 Master 状态完整备份的 Replica Master；通过论文中没有提到的某个机制，GFS 会在 Master 失效时切换到其中一个 Replica（见 5.1.3 节）。有可能这会需要一个人类管理者的介入来指定一个新的 Master。无论如何，我们都可以确定集群中潜伏着一个故障单点，理论上能够让集群无法从 Master 失效中进行自动恢复。我们会在后面的课程中学习如何使用 Raft 协议实现可容错的 Master。

除了 FAQ，课程还要求学生在阅读 GFS 的论文后回答一个问题，问题如下：

> Describe a sequence of events that result in a client reading stale data from the Google File System 描述一个事件序列，使得客户端会从 Google File System 中读取到过时的数据

通过查阅论文，不难找到两处答案：由失效后重启的 Chunk Server + 客户端缓存的 Chunk 位置数据导致客户端读取到过时的文件内容（见 4.5 和 2.7.1 节），和由于 Shadow Master 读取到的过时文件元信息（见 5.1.3 节）。以上是保证所有写入操作都成功时客户端可能读取到过时数据的两种情况 —— 如果有写入操作失败，数据会进入**不确定**的状态，自然客户端也有可能读取到过时或是无效的数据。

结语

本文没有总结论文中第六、七章的内容：第六章是 GFS 各项指标的测试结果，受限于篇幅故没能在此放出，若读者对 Google 测试 GFS 性能指标的方法有所兴趣也可参考这一章的内容；第七章则提到了 Google 在开发 GFS 时踩过的一些坑，主要和 Linux 本身的 bug 有关，此处没能放出这部分的内容主要是考虑到这些 bug 主要涉及 Linux 2.2 和 2.4 版本，相较于今日已失去其时效性，况且这些 bug 也很有可能已经由 GFS 的开发者修复并提交到新版的 Linux 中了。

从内容上看，阅读 GFS 的论文是对高性能和强一致性之间的矛盾的一次很好的 Case Study：在强一致性面前，GFS 选择了更高的吞吐性能以及自身架构的简洁。高性能与强一致性之间的矛盾是分布式系统领域经久不衰的话题，源于它们通常是不可兼得的。除外，为了实现理想的一致性，系统也可能面临来自并发操作、机器失效、网络隔离等问题所带来的挑战。

从概念上来讲，一致指的是某个正确的状态，而一个系统往往会有很多种不同的正确的状态，它们又常被统称为系统的**一致性模型**。在后面要阅读的论文中，我们还会不断地看到这个概念。

Google File System 论文的发表催生了后来的 HDFS，后者直到今天依然是开源分布式文件系统解决方案的首选。Google MapReduce 加上 Google File System 这两篇论文可谓是大数据时代的开山之作，与 Google BigTable 并称 Google 的三架马车，由此看来这几篇经典论文还是很值得我们去学习一番的。