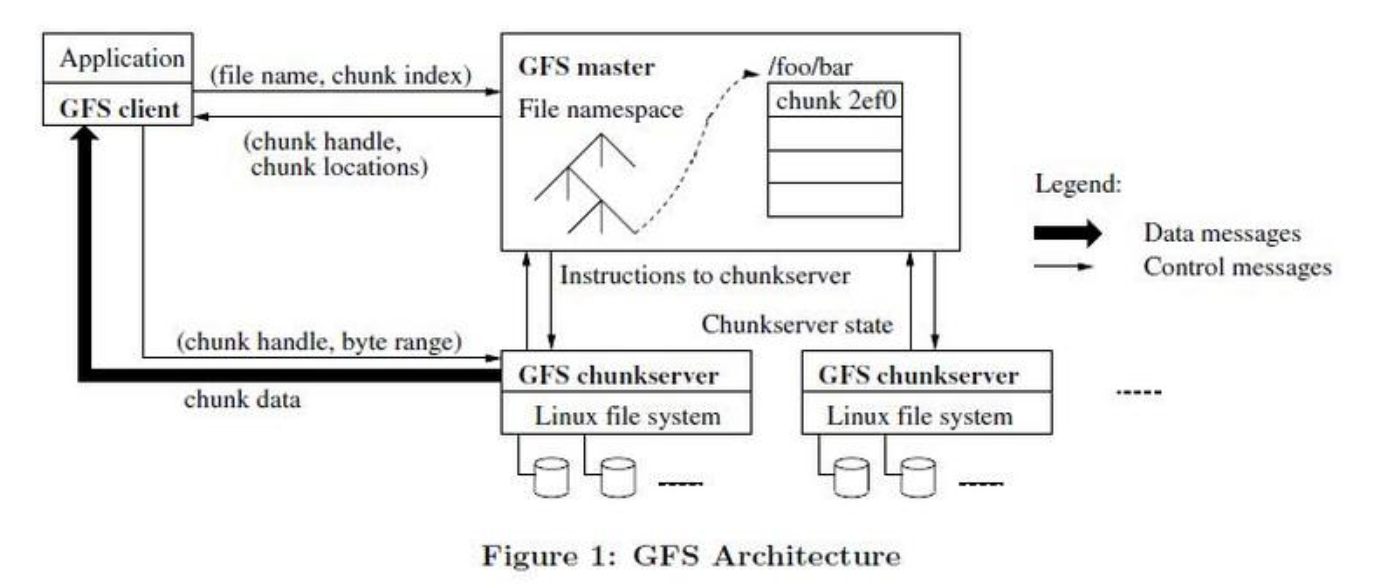
# GFS

## 1.1设计思路

1. 组件失效被认为是常态事件，而不是意外事件
2. 文件十分巨大
3. 绝大多数文件的修改是采用在文件尾部追加数据
4. 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

# 2设计原理

## 2.1架构



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、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节点存储的信息：

1，所有文件系统元数据：

1. [文件和chunk]名字空间（记录本地，以记录变更日志的方式记录在系统日志文件中）
2. 访问控制信息（同上）
3. 文件和chunk的映射信息（同上）
4. 每个chunk块副本的位置信息（不持久化，通过向chunk服务器轮询）

ps: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

不持久化chunk副本位置信息的理由：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

2，系统范围内的活动：

1. Chunk租用管理
2. 孤儿Chunk的回收
3. chunk在chunk服务器之间的迁移

ps：客户端和chunk服务器都没有（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。

### 2.1.1 GFS API接口

GFS虽是一个文件系统，但并没有遵循标准的POSIX接口，而仅仅提供了一下接口：  
1. Create 创建文件（仅元数据操作）  
2. Delete 删除文件（延迟删除与垃圾回收）  
3. Open 打开文件  
4. Close 关闭文件  
5. Read 读文件  
6. Write 写文件  
7. Record Append 原子性追加写操作（GFS重点优化的操作）  
8. Snapshot 对文件或目录做快照

不支持每个目录实现能够列出目录下所有文件（ls）,也不支持文件或者目录的链接（硬链接和软链接）

### 2.1.2 Master节点的操作

2.1.2.1 名称空间管理和锁

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。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。

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。（为了能支持并发的在同一目录下创建多个文件？）

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。

2.1.2.2 副本的管理

***创建副本***

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. 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（会删除之前过期的chunk，然后可能在另外的server上创建）、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。

***删除副本***

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。

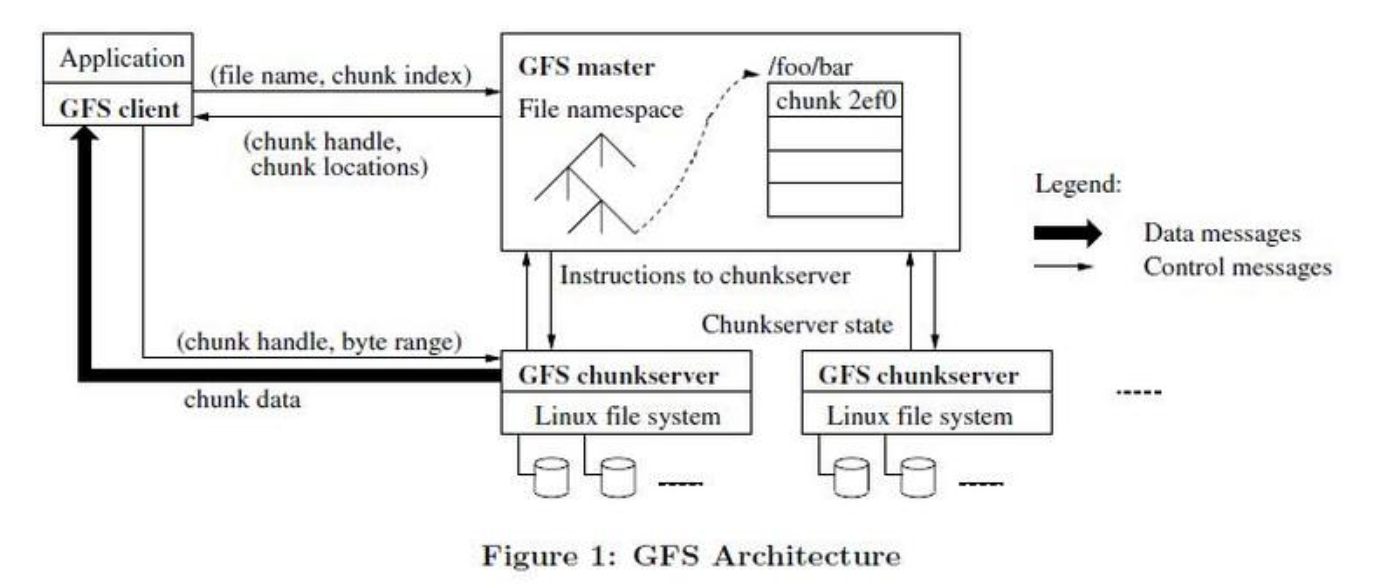
采用这种删除机制主要有如下三点好处：

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（空间得不到有效的利用）。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。

## 2.2 读写流程

### 2.2.1读过程



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。、

一处缓存有一致性问题：

客户端会缓存住文件中某个chunk的所对应的chunk标识，GFS的处理方式：缓存的元数据信息过期或者文件被重新打开，需要重新请求Master节点

这样能够成功的理由：跟文件的语义有关，正常在读取或写入文件的时候不会使文件大小变小，只会变大，所以chunk的标识就不会改变；有两种情况会导致文件大小变小，删除和打开的时候以trunck方式打开，所有在元数据信息过期或者文件被重新打开，重新请求即可

### 2.2.2 写过程

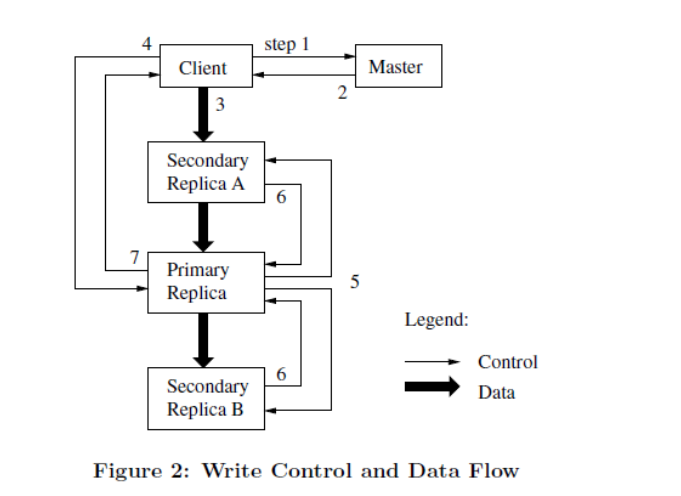
2.2.2.1租约

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。

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。

租约的主要目的在于，选择一个主的chunk副本出来，来控制写入的顺序；同时租约还有个作用，就是感知某些副本数据是否过期，没分配一次租约，chunk的版本号就会提高，如果在写入过程中

2.2.2.2 写入流程



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）。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。

对比HDFS:

数据流方式都是以流水线的方式推送到三台服务器上，但是控制流的方式不一样，GFS是由主chunk整理出落盘顺序，然后把这个顺序由主福本发给其他两个副本，两个副本按照这个顺序落盘；

而HDFS只有单写的情况，所以就不需要主datanode来调节顺序了，写入的顺序已经由下层的tcp保证了，写入的数据量packet的滑动窗口来控制就可以了

2.2.2.3 写入一致性分析

对于数据修改后文件的某个范围的数据（region）有两种状态：

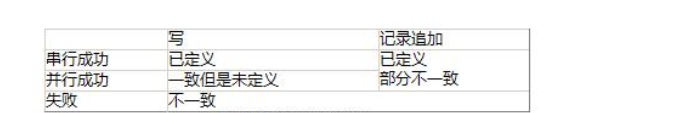
一致性：所有客户端，无论从哪个副本读取，读到的数据都一样

已定义：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（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）

一致的、未定义：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（比如对同一个偏移量两次分别写入abc,de,那么最后只能看到dec）

不一致：不同的客户在不同的时间会看到不同的数据

两种写入方式：



串行成功和并行成功：中间可能是由于chunkserver写入数据错误原因（接受到数据错误，写入磁盘出错）导致出现问题，这些问题都是有客户端重新发送数据，重写操作来解决

失败：可能就是因为某个chunkserver宕机，然后会导致不一致，这个交由版本号来处理

2.2.2.4 原子的记录追加

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，

例如：两次写入分别为abc,def,

第一次写到ab时出错，那么数据为ab,会通知客户端重写

第二次写def，成功，数据为abdef

第三次写abc,成功，数据为abdefabc

总结一下，正常情况没啥问题，在异常情况下，副本写入失败，然后需要重新追加（要保证一个记录完整性的添加一次），那么就需要接着上面不一致的数据继续append（所以副本都按照同一个偏移量往后写），所以对于之前失败的数据是不一致的，后面成功添加的话，这个数据是一致的。

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。

2.2.2.5 写入过程错误处理

随机写入和追加写入，若写入过程出错，即写入磁盘未成功都是交由客户端处理（重新进行写入流程）。

如果是由于某些副本的chunkserver宕机出错了，会继续写入过程，这个出错处理交给chunk副本的版本号机制来解决，过期的chunk会在“4.1中的过期失效的副本检测”的副本回收的时候删除，然后“2.1.2.2 副本的管理”的创建流程发现副本数低于目标值，就会创建新的副本出来。

## 2.3 chunk块尺寸

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（对小文件更友好些），内部碎片或许是对选择这么大的Chunk尺寸最具争议一点。

选择大chunk块的理由：

1.减少了客户端和Master节点通讯的需求

2.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

3.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

## 2.4 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（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，都由它们创建的逻辑时间唯一的、永久的标识。

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（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，将所有的状态数据写入一个Checkpoint文件（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 形式的数据结构存储，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。

不同于HDFS的点：

对于GFS来说，它会将日志复制到多台远程机器，等日志记录写入到本地以及远程机器的磁盘后才响应客户端操作请求。然后GFS的master通过老的checkpoint文件和日志生成新的checkpoint文件，这个是在本机完成的，

而对于HDFS1.x版本来说，它是将editlog和fsimage发送到secondnamenode上，在secondnamenode重构好新的fsimage之后，在发给namenode，然后namenode将新的fsimage替换旧的

# 3.快照

流程：

1. 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。）
2. Master节点复制源文件或者目录的元数据，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。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。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

# 异常处理

## 4.1 过期失效的副本检测

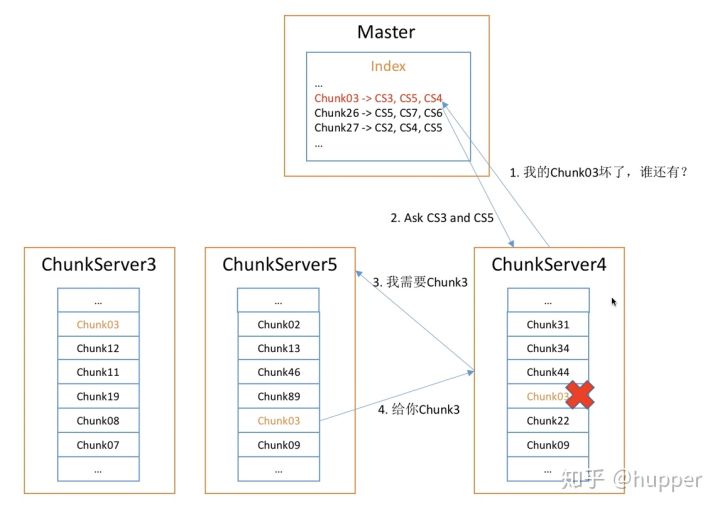
当Chunk服务器失效时，Chunk的副本有可能因错失了一些修改操作而过期失效。Master节点保存了每个Chunk的版本号，用来区分当前的副本和过期副本。

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。

## 4.2 恢复损坏的chunk



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（HDFS客户端来校验）读取操作涉及的范围内的块的Checksum。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。到底是在原chunkserver上重构副本，还是选择在新的机器上，这个就看选择的策略了

# 数据完整性

每个Chunk都分成64KB大小的块。每个块都对应一个32位的Checksum（HDFS分为block,packet,chunk）。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（checksum记录在日志里）。

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（HDFS客户端来校验）读取操作涉及的范围内的块的Checksum。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。

在写入过程中，校验和更新方式：

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只需进行增量式的更新（所以追加写要优于随机写），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。

数据监测：对于比较活跃的数据，通过在读取的过程中已经监测到数据是否损坏了；而对于不活跃的chunk副本，需要主动监测。

所有的chunk的数据损坏或者不一致，都是会将整个chunk恢复，而不是恢复chunk内部的对应部分。

# FAQ

**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。可以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。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。

**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。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。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。可以使用 Raft 协议实现可容错的 Master。

**Q：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**

1. 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
2. 由于 Shadow Master 读取到的过时文件元信息。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。

**ps:**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。

# 总结

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**。