# Google File System

<https://juejin.im/post/582d9c91d203090067fba883>

中文版GFS论文

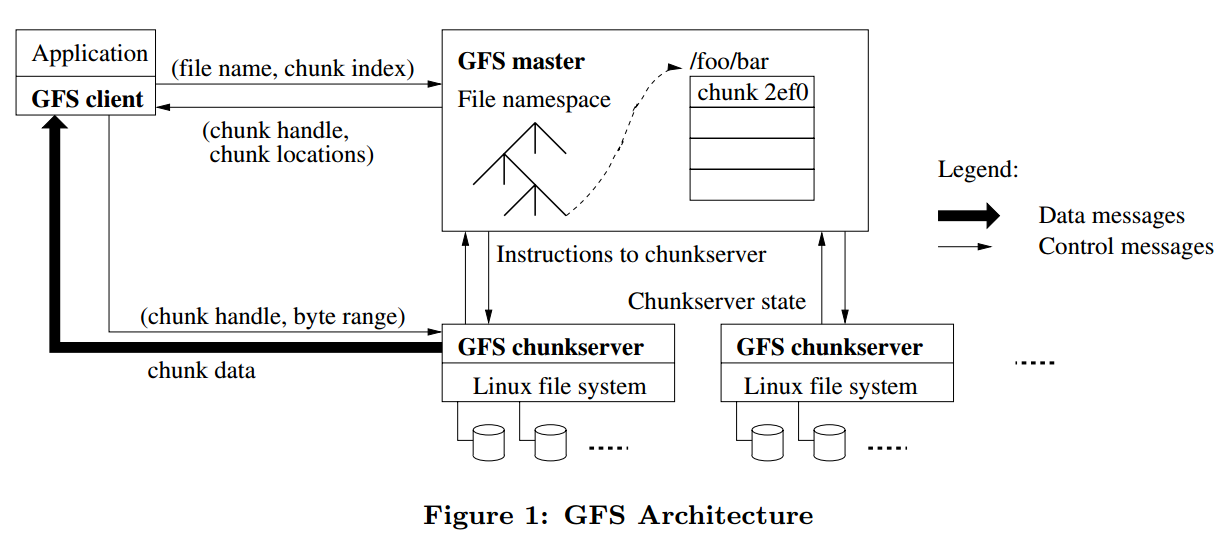
<http://www.open-open.com/lib/view/open1328763454608.html>

## Design Motivation

Google对现有的系统运行状态以及应用系统进行总结，抽象出对文件系统的需求，主要有以下几个方面：

1. 普通商用的机器硬件发生故障是常态；
2. 存储的文件普遍比较大，几个G的文件很常见；
3. 大部分的文件操作都是在追加数据，覆盖原来写入的数据的情况比较少见，随机写几乎不存在；
4. 读操作主要包括两种，large streaming read和small random read；
5. 为了应用的使用方便，多客户端并行地追加同一个文件需要非常高效；
6. 带宽的重要性大于时延，目标应用是高速读大块数据的应用，对响应时间没有过多的需求；

## 系统架构



GFS系统的节点可分为三种角色：master（主控服务器），chunkserver（CS，数据块服务器）和client。

### master

主控服务器中维护系统的元数据，包括文件及chunk命名空间、文件到chunk之间的映射、chunk位置信息。它也负责整个系统的全局控制，如chunk租约管理、垃圾回收无用chunk、chunk复制等。主控服务器会定期与chunkserver通过心跳的方式交换信息。

### chunkserver

GFS文件被划分为固定大小的数据块（chunk），在创建时由master分配一个64位全局唯一的chunk handle来标识。chunkserver以普通Linux文件的形式将chunk存储在磁盘中，读或写的时候需要**指定文件名和字节范围**，然后定位到对应的chunk。为了保证数据的可靠性，一个chunk一般会在多台chunkserver上存储，默认为3份，但用户也可以根据自己的需要修改这个值。

### client

client是GFS应用端使用的API接口，client与master交互来获取元数据信息，但是所有和数据相关的信息都是直接和chunkserver来交互的。

client不缓存文件数据，只缓存从master获取的元数据，这是由GFS的应用特点决定的。GFS最主要的应用有两个：MapReduce与Bigtable。对于MapReduce来说，GFS客户端使用方式为顺序读写，没有缓存文件数据的必要；Bigtable作为分布式表格系统，内部实现了一套缓存机制。

## Master设计

### Namespace Management and Locking

每个master操作都需要获得一系列的锁。如果一个操作涉及到/d1/d2/.../dn/leaf，那么需要获得/d1,/d1/d2,/d1/d2/.../dn的读锁，然后，根据操作类型，获得/d1/d2/.../dn/leaf的读锁或者写锁，其中leaf可能是文件或者路径。

一个例子，当/home/user被快照到/save/user的时候，/home/user/foo的创建是被禁止的。

对于快照，需要获得/home和/save的读锁，/home/user和/save/user的写锁。对于创建操作，会获得/home,/home/user的读锁，然后/home/user/foo的写锁。其中，/home/user的锁产生冲突，/home/user/foo创建会被禁止。

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。

### Chunk creation

GFS在创建chunk的时候，选择chunkserver时需要考虑以下因素：

* 磁盘空间使用率低于平均值的chunkserver；
* 限制每台chunkserver的最近的创建chunk的次数，因为创建chunk往往意味着后续需要写大量数据，所以，应该把写流量尽量均摊到每台chunkserver上；
* chunk的副本放在处于不同机架的chunkserver上。

### Chunk Re-replication

当一个chunk的副本数量少于预设定的数量时，需要做复制的操作，例如，chunkserver宕机，副本数据出错，磁盘损坏，或者设定的副本数量增加。

chunk的复制的优先级是按照下面的因素来确定的：

1. 丢失两个副本的chunk比丢失一个副本的chunk的复制认为优先级高；
2. 文件正在使用比文件已被删除的chunk的优先级高；
3. 阻塞了client进程的chunk的优先级高。

chunk复制的时候，选择新chunkserver要考虑的点：

1. 磁盘使用率；
2. 单个chunkserver的复制个数限制；
3. 多个副本需要在多个机架；
4. 集群的复制个数限制；
5. 限制每个chunkserver的复制网络带宽，通过限制读流量的速率来限制。、

### Rebalancing

周期性地检查副本分布情况，然后调整到更好的磁盘使用情况和负载均衡。master对于新加入的chunkserver会逐渐地迁移副本到上面，防止新chunkserver带宽占满。

### Master容错

Master容错与传统方法类似，通过操作日志加checkpoint的方式进行，并且有一台称为“shadow master”的实时热备。

Master的修改操作总是**先记录操作日志，然后修改内存**。为了减少Master故障重启后恢复时间，Master定期将内存中的数据以checkpoint文件的形式转储到磁盘中，从而减少回放的日志量。GFS还执行实时热备，所有的元数据修改操作必须保证发送到实时热备才算成功。远程的实时热备实时接收Master发送的操作日志并在内存中回放操作。Master出现故障可以秒级切换到实时热备提供服务。为保证同一时刻只有一个Master，GFS采用Chubby进行选主操作。

### 垃圾回收

GFS采用延迟删除的机制，当删除文件后，GFS并不要求立即归还可用的物理存储，而是在元数据中将文件名改为一个隐藏的名字，并且包含一个删除时间戳。master定期检查，如果发现文件删除时间超过一段时间（比如3天），会把文件从内存元数据删除，以后Chunkserver和master的心跳消息中，Chunkserver会报告自己的chunk集合，master会回复在master元数据中已经不存在的chunk信息，Chunkserver会释放这些chunk副本。

采用这种方式删除的好处：

* 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；
* 删除操作和其他的全局扫描metadata的操作可以放到一起做。

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。

### 过期副本的处理

当一台chunkserver挂掉的时候，有新的写入操作到chunk副本，会导致chunkserver的数据不是最新的。

假设chunk A有三个副本A1、A2、A3，假如A2下线了又重新上线，在A2下线过程中A1和A3有更新，A2需要被master当做垃圾回收：GFS对每个chunk维护一个版本号，每次primary chunkserver重新延长租约有效期时，master会将chunk的版本号加1。A2重新上线后，master发现A2的版本号低，将A2标记为可删除，master的垃圾回收任务会定时检查，通知chunkserver将A2回收。

master会把落后的chunk当做垃圾来清理，并且不会把落后的chunkserver的位置信息传给client。

### 快照

快照是对源文件或者目录进行一个“快照”，生成该时刻源文件或目录的瞬间状态。GFS对快照使用了写时复制策略，快照只是增加GFS中chunk的引用计数，在客户端修改这个chunk时，才在Chunkserver中拷贝chunk数据生成新的chunk，后续的操作落到新的chunk上。

对文件执行快照大致步骤：

* 通过租约机制收回对文件的每个chunk写权限，停止对文件的写服务；
* master拷贝文件名等元数据生成一个新的快照文件；
* 对执行快照的文件的所有chunk增加引用计数。

foo文件生成快照foo.bak，foo在GFS中有三个chunk：c1、c2、c3，Master首先回收c1、c2和c3的写租约，Master复制foo文件的元数据用于生成foo.bak，boo.bak同样指向c1、c2、c3。快照前，c1、c2和c3只被一个foo引用，引用计数为1；快照后，引用计数为2。客户端向c3追加数据时，Master发现c3的引用计数大于1，通知c3所在的chunkserver拷贝c3生成c3’，客户端的追加操作都作用在c3’上。

## Metadata

master上保存了三种元数据：

1. 命名空间，也就是整个文件系统的目录结构以及chunk基本信息；
2. 文件到chunk之间的映射；
3. chunk副本的位置信息。

master需要持久化**命名空间**和**文件到chunk之间的映射**。使用内存存储metadata的好处是读取metadata速度快，方便master做一些全局扫描metadata相关信息的操作，比如负载均衡。

chunk副本的位置信息不需要持久化，因为chunkserver维护这些信息，而且chunkserver容易宕机重启，在出现故障时，master都需要修改内存和已经持久化存储的位置信息。master故障重启时通过chunkserver汇报可以获取chunk副本的位置信息。

### Operation Log

操作日志的作用是：

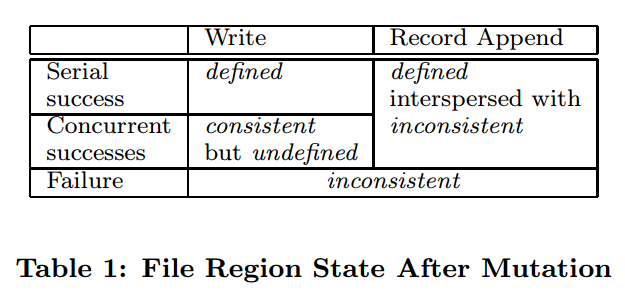
* 持久化存储metadata；
* 它的存储顺序定义了client并行操作最终的操作顺序。

操作日志会存储在master和多台远程机器上，只有操作日志在master和多台远程机器都写入成功后，master才会向client返回操作成功。为了减少操作日志在多台机器同步对系统吞吐量的影响，可以将一批操作日志形成一个请求，然后再写入到master和其他远程机器上。

当操作日志达到一定大小时，master会做checkpoint，相当于把内存中的b-tree给dump到磁盘中。master重启时，可以读取最近一次的checkpoint，然后重放在checkpoint之后的操作日志，这样就大大减少了系统恢复时间。

在做checkpoint的时候，master会先切换到新的操作日志上，然后运行新的线程做checkpoint，所以对于新来的请求是基本不会受到影响的。

## Consistency Model



GFS中consistent、defined的定义如下：

consistent：所有的客户端都能看到一样的数据，不管它们从哪个副本读取，但是可能这些文件内容不是最新的，这样就属于undefined；

defined：当一个文件区域发生操作后，client可以看到刚刚操作的所有数据，那么说这次操作是defined。意思是说client不仅看到的内容一样，而且文件也是经过修改后最新的。defined其实更深层的含义在于一个客户端写文件，之后读能否读出之前写的文件，也就是说和自己的写的是否一致。在追加模式下，defined是可以做到的，因为每次的记录追加都是原子操作。但是write中，并行写可能造成数据交叉，那么对于client来说，**不同client读取的数据是一致的，但是读出来的内容是一种未定义的状态**。追加模式由于保留了每个client的原子追加，存在追加失败之类的异常情况，同为副本的chunkserver上存在的数据可能不一致。

下面分析表格中出现的几种情况。

Write(Serial Success)，单个写操作，并且返回成功，那么所有副本都写入了这次操作的数据，因此所有客户端都能看到这次写入的数据，所以是defined。

Write(Concurrent Successes)，多个写操作，并且返回成功，由于多个客户端写请求发送给primary后，由primary来决定写的操作顺序，但是，有可能多个写操作可能是有区域重叠的，这样，最终写完成的数据可能是多个写操作数据叠加在一起，所以这种情况是consistent和undefined。

Write(Failure)，写操作失败，则可能有的副本写入了数据，有的没有，所以是inconsistent。

Record Append(Serial Success and Concurrent Success)，由于Record Append可能包含重复数据，因此，是inconsistent，由于整个写入的数据都能看到，所以是defined。

Record Append(Failure)，可能部分副本append成功，部分副本append失败，所以，结果是inconsistent。

GFS用version来标记一个chunkserver挂掉的期间，是否有client进行了write或者append操作。每进行一次write或者append，version会增加。

需要考虑的点是client会缓存chunk的位置信息，有可能其中某些chunkserver已经挂掉又起来了，这个时候chunkserver的数据可能是老的数据，读到的数据是会不一致的。读流程中，好像没有看到要带version信息来读的。这个论文中没看到避免的措施，目前还没有结果。

应用层需要采用的机制：用append而不是write，做checkpoint，writing self-validating和self-identifying records。具体地，如下：

1. 应用的使用流程是append一个文件，到最终写完后，重命名文件；
2. 对文件做checkpoint，这样应用只需要关注上次checkpoint时的文件区域到最新文件区域的数据是否是consistent的，如果这期间发生不一致，可以重新做这些操作；
3. 对于并行做append的操作，可能会出现重复的数据，GFS client提供去重的功能。

## 容错和诊断

### High Availability

为了实现高可用性，GFS在通过两方面来解决，一是fast recovery，二是replication。

**Fast Recovery：**

master和chunkserver都被设计成能够在秒级重启。

**Chunk Replications：**

每个chunk在多个机架上有副本，副本数量由用户来指定。当某一个chunkserver不可用时，master会自动的复制副本，保证副本数量和用户指定的一致。

**Master Replication：**

master的operation log和checkpoint都会复制到多台机器上，要保证这些机器的写都成功了，才认为是成功。只有一台master在来做garbage collection等后台操作。当master挂掉后，它能在很短时间内重启；当master所在的机器挂掉后，监控会在其他具有operation log的机器上重启启动master。

新启动的master只提供读服务，因为可能在挂掉的一瞬间，有些日志记录到primary master上，而没有记录到secondary master上（这里GFS没有具体说同步的流程）。

### Data Integrity

每个chunkserver都会通过checksum来验证数据是否损坏的。

每个chunk被分成多个64KB的block，每个block有32位的checksum，checksum在内存中和磁盘的log中都有记录。

对于读请求，chunkserver会检查读操作所涉及block的所有checksum值是否正确，如果有一个block的checksum不对，那么会报错给client和master。client这时会从其他副本读数据，而master会clone一个新副本，当新副本clone好后，master会删除掉这个checksum出错的副本。

## 关键问题

### Single Master和租约机制

GFS架构中只有单个master，这种架构的好处是设计和实现简单，例如，实现负载均衡时可以利用master上存储的全局的信息来做决策。但是，在这种架构下，要避免的一个问题是，应用读和写请求时，要弱化master的参与度，防止它成为整个系统架构中的瓶颈。

其实在GFS中，master的内存不会成为GFS瓶颈。chunk的元信息包括全局唯一ID、版本号、副本所在的chunkserver编号、引用计数等，不超过64B。对于1PB的数据，总共需要的元信息：1PB\*3/64MB\*64=3GB。

GFS数据追加以记录为单位，每个记录大小几十KB到几MB不等，如果每一次记录追加都需要请求master，那么master会成为系统的瓶颈。GFS系统中通过租约机制将chunk写操作授权给chunkserver。拥有租约授权的chunkserver为主chunkserver，其他为备chunkserver。租约授权针对单个chunk，在租约有效期内，对该chunk的写操作都由主chunkserver负责，从而减轻Master的负载。只要没有出现异常，主chunkserver可以不断的向Master请求延长租约有效期直到整个chunk写满。（**这里的意思是将某个chunk的追加操作授权给了一个chunkserver，之后client对这个chunk的操作不需要master参与，直接通过chunkserver就可以了**）。

从一个请求流程来讨论这个问题。首先，上层的应用把文件名和偏移量信息传递给client，client把得到的信息转成文件名和chunk index，并发送给master。master把chunk handle和chunk所在chunkserver位置信息返回给client，client会把这个信息缓存起来。在下次应用需要读这个chunk的时候就不需要从master获取chunk的位置信息了（**虽然下次client读这个chunk可以使用缓存的chunkserver的信息，但是如果chunkserver的租约过期了，还是需要再次从master请求的得到chunkserver信息，同时master选择一个chunkserver作为主chunkserver**），从而减小了master的压力。

此外，GFS支持在一个请求中同时读取多个chunk的位置信息，这样更进一步的减少了client和master之间的交互次数。

### chunk的大小

chunk的默认大小为64MB，比一般的文件系统要大。

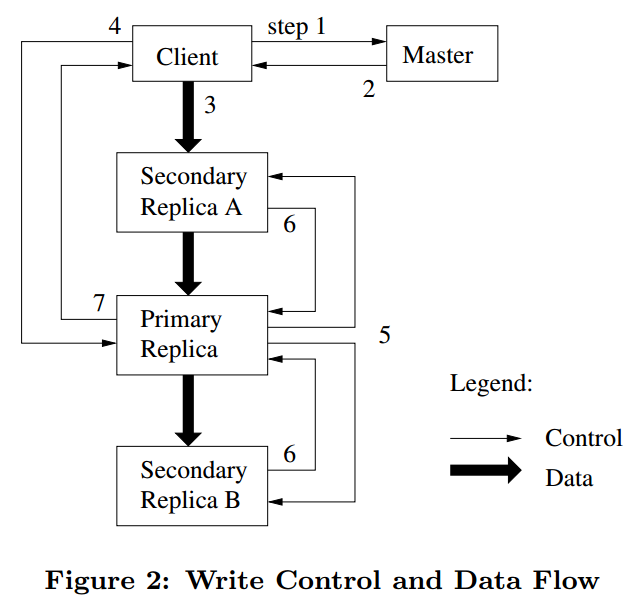
**优点：**

* 可以减少client和master的交互次数，chunk size比较大的时候，多次读的可能是一块chunk的数据，这样，可以减少client向 master请求chunk位置信息的次数。
* 对于同一个chunk，client可以和chunkserver之间保持持久连接，提升读的性能。
* chunk size越大，chunk的metadata的总大小就越小，使得chunk相关的metadata可以存储在GFS master的内存中。

**缺点：**

* chunk size越大时，可能对部分文件来讲只有1个chunk，那么这个时候对该文件的读写就会落到一个GFS chunkserver上（**比如现在有很多的小文件，由于chunk很大，那么这些小文件可能都存放在同一个chunk里，那么多于这些小文件的读写请求都会交给这一个chunk**），成为热点。

### 写流程



1. client向master请求chunk每个副本所在的chunkserver，其中主chunkserver持有修改租约。如果没有chunkserver持有租约说明这个chunk最近没有写操作，Master会发起一个任务，按照一定的策略将这个chunk的租约授权给其中一台chunkserver。
2. Master返回客户端各个副本所在chunkserver位置信息，客户端缓存这些信息以供以后使用。如果不出故障，客户端以后读写**该chunk**都不需要再次请求Master。
3. 客户端将要追加的记录发送到**每一个副本**，每一个chunkserver会在内部的LRU结构中缓存这些数据。（**需要多加思考的设计思路**）
4. 当所有副本确认收到数据后（应该是通知client了），客户端发起一个写请求控制命令给主副本。由于主副本可能会接收带多个客户端对同一个chunk的并发追加操作，主副本将确定这些操作的顺序并写入本地。
5. 主副本把写请求提交给所有的备副本。每一个备副本会根据主副本确定的顺序执行写操作。
6. 备副本成功完成后应答主副本。
7. 主副本应答客户端，如果有副本写入发生错误，将出现主副本写成功但是某些被副本不成功的情况，客户端将重试。

GFS写流程有两个特色：流水线以及分离数据流与控制流。

流水线操作用来减少延迟，为了充分利用每台机器的网络带宽，上面写流程的第3步采用了流水线。当一个chunkserver收到一些数据，它立即开始转发。假设一台机器总共有三个副本S1-S3，整个的流程为：

1. client选择离它最近的chunkserver S1，开始推送数据；
2. 当chunkserver S1收到数据后，它会立马转发到离它最近的chunkserver S2；
3. chunkserver S2收到数据后，会立马转发给离它最近的chunkserver S3。

分离数据流与控制流为了优化数据传输，是在第5步中应用的。

实际在写和追加的过程中可能出现主副本租约过期而失去chunk修改操作的权限，以及Chunkserver出现故障等。

### 追加流程

追加的操作流程和写差不多，主要有以下区别：

* client把数据推送到所有副本的最后一个chunk，然后发送写请求到primary chunkserver（**这里要这么理解，GFS中的文件通常是远大于64MB的，所以数据推送到副本的最后一个chunk**）；
* 主chunkserver首先检查最后一个chunk的剩余空间是否可以满足当前写请求，如果可以，那么执行写流程，否则，它会把当前的chunk的剩余空间pad起来，然后告诉其他的副本也这么干，最后告诉client这个chunk满了，写入下个chunk。

**如果追加操作部分失败如何处理？**

例如，写操作要追加到S1-S3，但是，仅仅是S1,S2成功了，S3失败了，GFS client会重试操作，假如第二次成功了，那么S1,S2写了两次，S3写了一次，目前的理解是GFS会先把失败的记录进行padding对齐到primary的记录，然后再继续append。

### GFS追加过程中副本出现故障如何处理

对于备副本故障，写入的时候会失败，然后primary会返回错误给client。按照一般的系统设计，client会重试一定次数，发现还是失败，这时候client会把情况告诉给master，master可以检测chunkserver的情况，然后把最新的chunkserver信息同步给client，client端再继续重试。

对于主副本故障，写入的时候会失败，client端应该是超时了。client端会继续重试一定次数，发现还是一直超时，那么把情况告诉给master，master发现primary挂掉，会重新授权租约到其他chunkserver，并把情况返回给client。

### master如何实现高可用

* namespace和文件到chunk信息会通过操作日志的方式持久化存储，操作日志会同步到包括master在内的多台机器上；
* 对metadata做checkpoint，保证重启后replay消耗时间比较短，checkpoint可以直接映射到内存使用，不用解析；
* 在primary master发生故障的时候，并且无法重启时，会有外部监控将secondary master，并提供读服务。secondary master也会监控chunkserver的状态，然后把primary master的日志replay到内存中。

### 新建chunk如何选择chunkserver？如何避免新机器上线其他机器同时迁移？

选择chunkserver根据以下几点：

* 磁盘空间使用率低于平均值的chunkserver；
* 限制每台chunkserver最近创建chunk的次数，因为创建chunk往往意味着后续需要写入大量数据，所以，应该把写流量均摊到每台chunkserver；
* chunk的副本放置于不同机架的chunkserver上。

在集群中加入新的机器之后，由于新机器的负载会较低，容易造成其他节点同时向新机器迁移数据的情况。通过限制单个chunkserver的clone操作的个数，以及clone使用的带宽来限制，即从源chunkserver对数据的频率做控制。

### chunkserver的数据结构如何设计

chunkserver主要是存储64KB block的checksum信息，需要由chunk+offset，能够快速定位到checksum，可以用hashmap。

### chunkserver如何应对磁盘可能发生位翻转

利用checksum机制，分读和写两种情况来讨论：

对于读，要检查所读的所有block的checksum值；

对于写，分为append和write。对于append，不检查checksum，延迟到读的时候检查，因为append的时候，对于最后一个不完整的block计算checksum时候采用的是增量的计算，即使前面存在错误，也能在后来的读发现。对于overwrite，因为不能采用增量计算，要覆盖checksum，所以，必须要先检查只写入部分数据的checksum是否不一致，否则，数据错误会被隐藏。