# Google File System

## 中文翻译

<https://blog.csdn.net/hguisu/article/details/7244798>

## 前提假设

### 组件失效被认为是常态事件，而不是意外事件。

系统必须持续监控自身的状态，它必须将组件失效作为一种常态，能够迅速地侦测、冗余并恢复失效的组件。

### 读写的假设条件

#### 读

##### 大规模的流式读取（Large streaming reads ）

##### 小规模的随机读取（small random reads）

#### 写

##### 大规模顺序的数据追加方式的写（Large, sequential writes that append data to files）

##### 小规模随机位置写（Small writes at arbitrary positions）

## 接口（API）

### 创建文件（Create）

### 删除文件（Delete）

### 打开文件（Open）

### 关闭文件（Close）

### 读文件（Read）

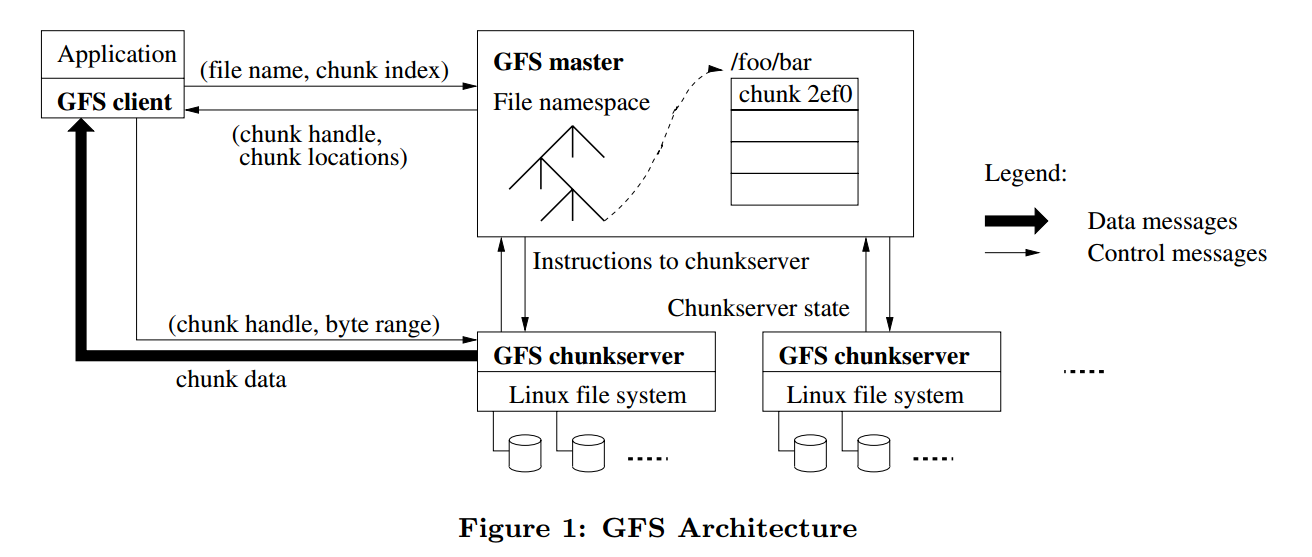
### 写文件（Write）

### 快照（Snapshot）

### 记录追加（Record append）

## 架构

### 图



### 一个master

#### 作用

维护所有文件系统的元信息。

名字空间。

访问控制信息。

文件和Chunk的映射信息。

当前Chunk的位置信息。

Chunk租约管理。

孤儿Chunk的回收。

Chunk在Chunk服务器之间的迁移。

使用心跳信息周期地和每个Chunk服务器通讯，发送指令到各个Chunk服务器并接收Chunk服务器的状态信息。

#### 元数据分类

##### 文件和Chunk的命名空间

##### 文件和Chunk的对应关系

##### 每个Chunk副本的存放地点

#### 元数据数据结构与算法

##### 文件名是用前缀压缩算法压缩过的

#### 元数据组织

所有的元数据都保存在Master服务器的内存中。

前两种类型的元数据（命名空间、文件和Chunk的对应关系）同时也会以记录变更日志的方式记录在操作系统的系统日志文件中，日志文件存储在本地磁盘上，同时日志会被复制到其它的远程Master服务器上。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。

Master服务器不会持久保存Chunk位置信息。Master服务器在启动时，或者有新的Chunk服务器加入时，向各个Chunk服务器轮询它们所存储的Chunk的信息。

#### 元数据操作日志

操作日志包含了关键的元数据变更历史记录。

把日志复制到多台远程机器，并且只有把相应的日志记录写入到本地以及远程机器的硬盘后，才会响应客户端的操作请求。

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。

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。

#### 部署

### 多个chunk server

#### 一个chunk的大小

##### 选择64MB

##### 选择较大的chunk尺寸的优点

1. 减少了客户端和Master节点通讯的需求，因为只需要一次和Mater节点的通信就可以获取Chunk的位置信息，之后就可以对同一个Chunk进行多次的读写操作。这种方式对降低我们的工作负载来说效果显著，因为我们的应用程序通常是连续读写大文件。即使是小规模的随机读取，采用较大的Chunk尺寸也带来明显的好处，客户端可以轻松的缓存一个数TB的工作数据集所有的Chunk位置信息。
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。

##### 热点问题

当有许多的客户端对同一个小文件进行多次的访问时，存储这些Chunk的Chunk服务器就会变成热点。

一个可执行文件在GFS上保存为single-chunk文件，之后这个可执行文件在数百台机器上同时启动。存放这个可执行文件的几个Chunk服务器被数百个客户端的并发请求访问导致系统局部过载。

我们通过使用更大的复制参数来保存可执行文件，以及错开批处理队列系统程序的启动时间的方法解决了这个问题。

### 文件的组织

文件被分成固定大小的chunk。

每一个chunk由一个全局唯一的64 bit chunk handle标识。

根据指定的Chunk handle和字节范围来读写块数据。

chunk一般是3份拷贝。

### 客户端API

客户端和Master节点的通信只获取元数据。

所有的数据操作都是由客户端直接和Chunk服务器进行交互的。

## 一致性模型

GFS支持一个宽松的一致性模型。

### 一致的

如果所有客户端，无论从哪个副本读取，读到的数据都一样，那么我们认为文件region是“一致的”。

### 已定义的

如果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，那么这个region是“已定义的”。

### 保障措施

对Chunk的所有副本的修改操作顺序一致。

使用Chunk的版本号来检测副本是否因为它所在的Chunk服务器宕机而错过了修改操作而导致其失效。

### 客户端读取到失效副本

由于Chunk位置信息会被客户端缓存，所以在信息刷新前，客户端有可能从一个失效的副本读取了数据。

在缓存的超时时间和文件下一次被打开的时间之间存在一个时间窗，文件再次被打开后会清除缓存中与该文件有关的所有Chunk位置信息。

## 系统交互

### 客户读取数据流程

#### 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

#### 客户端把文件名和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

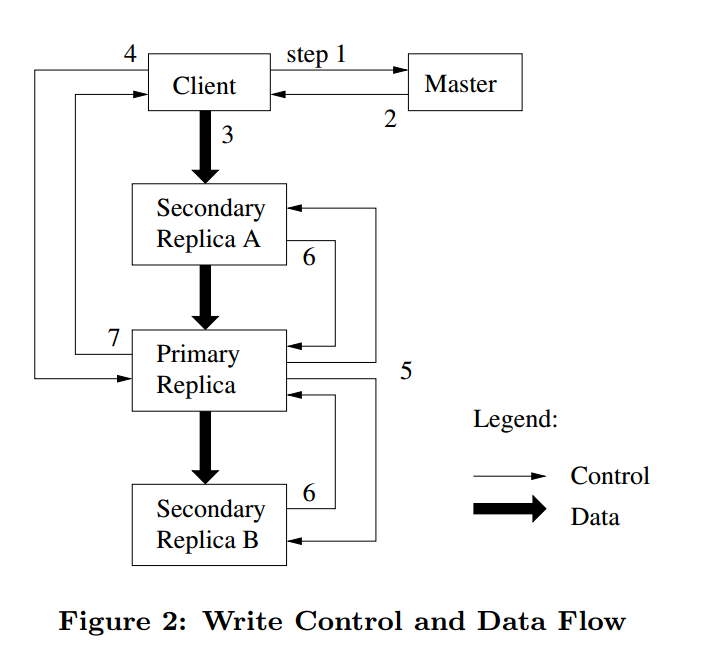
#### 客户端发送请求到其中的一个副本处，一般会选择最近的。请求信息包含了Chunk的标识和字节范围。

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。

### 客户端写入流程

#### 图



#### 步骤

##### 客户机向Master节点询问哪一个Chunk服务器持有当前的租约，以及其它副本的位置。

如果没有一个Chunk持有租约，Master节点就选择其中一个副本建立一个租约（这个步骤在图上没有显示）。

##### Master节点将主Chunk的标识符以及其它副本（又称为secondary副本、二级副本）的位置返回给客户机。

客户机缓存这些数据以便后续的操作。只有在主Chunk不可用，或者主Chunk回复信息表明它已不再持有租约的时候，客户机才需要重新跟Master节点联系。

##### 客户机把数据推送到所有的副本上。

客户机可以以任意的顺序推送数据。Chunk服务器接收到数据并保存在它的内部LRU缓存中，一直到数据被使用或者过期交换出去。由于数据流的网络传输负载非常高，通过分离数据流和控制流，我们可以基于网络拓扑情况对数据流进行规划，提高系统性能，而不用去理会哪个Chunk服务器保存了主Chunk。

##### 当所有的副本都确认接收到了数据，客户机发送写请求到主Chunk服务器。

这个请求标识了早前推送到所有副本的数据。主Chunk为接收到的所有操作分配连续的序列号，这些操作可能来自不同的客户机，序列号保证了操作顺序执行。它以序列号的顺序把操作应用到它自己的本地状态中。

##### 主Chunk把写请求传递到所有的二级副本。

每个二级副本依照主Chunk分配的序列号以相同的顺序执行这些操作。

##### 所有的二级副本回复主Chunk，它们已经完成了操作。

##### 主Chunk服务器回复客户机。

任何副本产生的任何错误都会返回给客户机。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。

### 原子的记录追加

记录追加是一种修改操作，它也遵循[客户端写入流程](#_客户端写入流程)描述的控制流程，除了在主Chunk有些额外的控制逻辑。

客户机把数据推送给文件最后一个Chunk的所有副本，之后发送请求给主Chunk。主Chunk会检查这次记录追加操作是否会使Chunk超过最大尺寸（64MB）。如果超过了最大尺寸，主Chunk首先将当前Chunk填充到最大尺寸，之后通知所有二级副本做同样的操作，然后回复客户机要求其对下一个Chunk重新进行记录追加操作。

（记录追加的数据大小严格控制在Chunk最大尺寸的1/4，这样即使在最坏情况下，数据碎片的数量仍然在可控的范围。）

通常情况下追加的记录不超过Chunk的最大尺寸，主Chunk把数据追加到自己的副本内，然后通知二级副本把数据写在跟主Chunk一样的位置上，最后回复客户机操作成功。

### 快照

#### 文件快照

##### 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。

##### 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。通过在源Chunk所在Chunk服务器上创建新的Chunk，我们确保数据在本地而不是通过网络复制。

从这点来讲，请求的处理方式和任何其它Chunk没什么不同。

##### 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

不必理会它是从一个已存在的Chunk克隆出来的。

#### 目录快照

## 垃圾回收

### 机制

当一个文件被应用程序删除时，Master节点象对待其它修改操作一样，立刻把删除操作以日志的方式记录下来。

但是，Master节点并不马上回收资源，而是把文件名改为一个包含删除时间戳的、隐藏的名字。

当Master节点对文件系统命名空间做常规扫描的时候，它会删除所有三天前的隐藏文件（这个时间间隔是可以设置的）。

直到文件被真正删除，它们仍旧可以用新的特殊的名字读取，也可以通过把隐藏文件改名为正常显示的文件名的方式“反删除”。

当隐藏文件被从名称空间中删除，Master服务器内存中保存的这个文件的相关元数据才会被删除。这也有效的切断了文件和它包含的所有Chunk的连接

### 过期失效的副本检测

##### 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。

## 容错

### 快速恢复

### Chunk复制

### Master服务器的复制

#### 操作日志写入到Master服务器的备节点和本机的磁盘。

#### “影子”Master

“影子”Master服务器为了保持自身状态是最新的，它会读取一份当前正在进行的操作的日志副本，并且依照和主Master服务器完全相同的顺序来更改内部的数据结构。

和主Master服务器一样，“影子”Master服务器在启动的时候也会从Chunk服务器轮询数据（之后定期拉数据），数据中包括了Chunk副本的位置信息；

“影子”Master服务器也会定期和Chunk服务器“握手”来确定它们的状态。

在主Master服务器因创建和删除副本导致副本位置信息更新时，“影子”Master服务器才和主Master服务器通信来更新自身状态。