# 概述

分布式文件系统的主要功能有两个：一个是存储文档、图像、视频之类的**Blob类型**数据；另一个是作为**分布式表格系统的持久化层**。

分布式文件系统中最为著名的莫过于Google File System（GFS），它构建在廉价的PC服务器上，支持自动容错。GFS内部将大文件划分为大小约为64MB的数据块（chunk），并通过主控服务器（Master）实现元数据管理、副本管理、自动负载均衡等操作。其他的文件系统，例如Taobao File System（TFS）、Facebook Haystack或多或少借鉴了GFS的思路，架构上都比较接近。

# 分类

## GFS

Google文件系统（GFS）是构建在廉价服务器之上的大型分布式系统。它将服务器故障视为正常现象，通过软件的方式自动容错，在保证系统可靠性和可用性的同时，大大降低系统的成本。

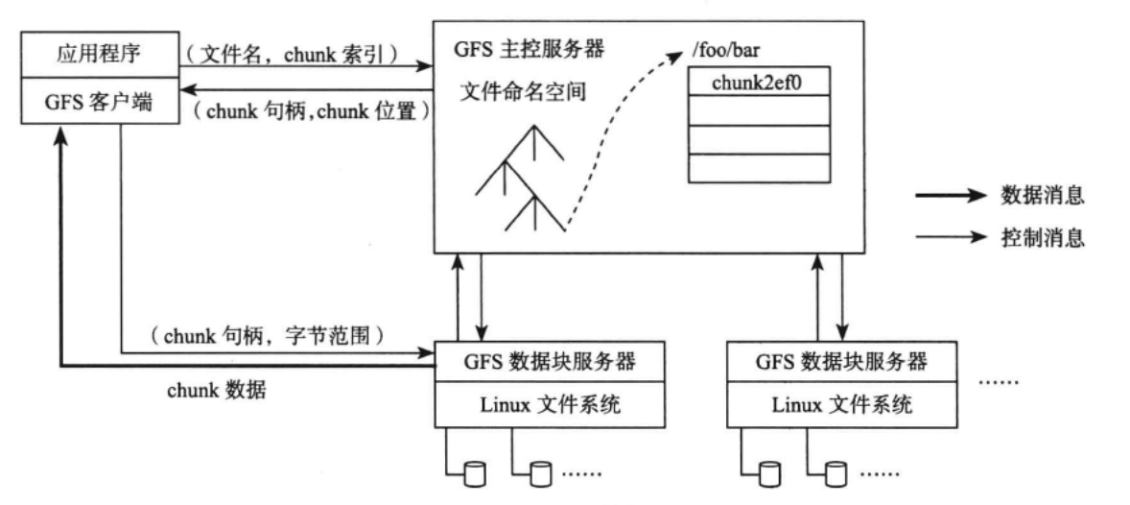
GFS是Google分布式存储的基石，其他存储系统，比如Google Bigtable、Google Megastore、Google Percolator均直接或间接地构建在GFS之上。另外，Google大规模批处理系统MapReduce也需要利用GFS作为海量数据的输入输出。

### 系统架构

GFS系统的节点可分为三种角色：GFS Master（主控服务器）、GFS ChunkServer（CS，数据块服务器）以及GFS客户端。

GFS文件被划分为固定大小的数据块（chunk），由主服务器在创建时分配一个64位全局唯一的chunk句柄。CS以普通的Linux文件的形式将chunk存储在磁盘中。为了保证可靠性，chunk在不同的机器中备份多份，默认为三份。

主控服务器中维护了系统的元数据，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CS通过心跳的方式交换信息。



客户端是GFS提供给应用程序的访问接口，它是一组专用接口，不遵循POSIX规范，以库文件的形式提供。客户端访问GFS时，首先访问主控服务器节点，获取与之进行交互的CS信息，然后直接访问这些CS，完成数据存取工作。

需要注意的是，GFS中的客户端不缓存文件数据，只缓存主控服务器中获取的元数据，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，GFS客户端使用方式为顺序读写，没有缓存文件数据的必要；而Bigtable作为分布式表格系统，内部实现了一套缓存机制。另外，如何维护客户端缓存与实际数据之间的一致性是一个极其复杂的问题。

注：MapReduce这种读写特性与Kafka内部的顺序读写原理一致，这样不需要缓存而且高效。

### 关键问题

#### 租约机制

GFS数据追加以记录为单位，每个记录的大小为几十KB到几MB不等，如果每次追加都需要请求Master，那么Master显然会成为系统的性能瓶颈，因此，GFS系统中通过租约（lease）机制将chunk写操作授权给ChunkServer。拥有租约授权的ChunkServer称为主ChunkServer，其他副本所在的ChunkServer称为备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。一般来说，租约的有效期比较长，比如60s，只要没有出现异常，主ChunkServer可以不断向Master请求延长租约的有效期直到整个chunk写满。

假设chunk A在GFS中保存了三个副本A1、A2、A3，其中，A1是主副本。如果副本A2所在的ChunkServer下线后又重新上线，并且在A2下线的过程中，副本A1和A3有更新，那么A2需要被Master当成垃圾回收掉。GFS通过对每个chunk维护一个版本号来解决，每次给chunk进行租约授权或者主ChunkServer重新延长租约有效期时，Master会将chunk的版本号加1。A2下线的过程中，副本A1和A3有更新，说明主ChunkServer向Master重新申请租约并增加了A1和A3的版本号，等到A2重新上线后，Master能够发现A2的版本号太低，从而将A2标记为可删除的chunk，Master的垃圾回收任务会定时检查，并通知ChunkServer将A2回收掉。

注：将A2回收掉之后，那么副本少了一个，怎么处理？

#### 一致性模型

GFS主要是为了追加（append）而不是改写（overwrite）而设计的。一方面是因为改写的需求比较少，或者可以通过追加来实现，比如可以只使用GFS的追加功能构建分布式表格系统Bigtable；另一方面是因为追加的一致性模型相比改写要更加简单有效。考虑chunk A的三个副本A1、A2、A3，有一个改写操作修改了A1、A2但没有修改A3，这样，落到副本A3的读操作可能读到不正确的数据；相应地，如果有一个追加操作往A1、A2上追加了一个记录，但是追加A3失败，那么即使读操作落到副本A3也只是读到过期而不是错误的数据。

注：这个其实就是以牺牲存储空间（追加要比直接修改多占用空间）来换取高效操作和简单。

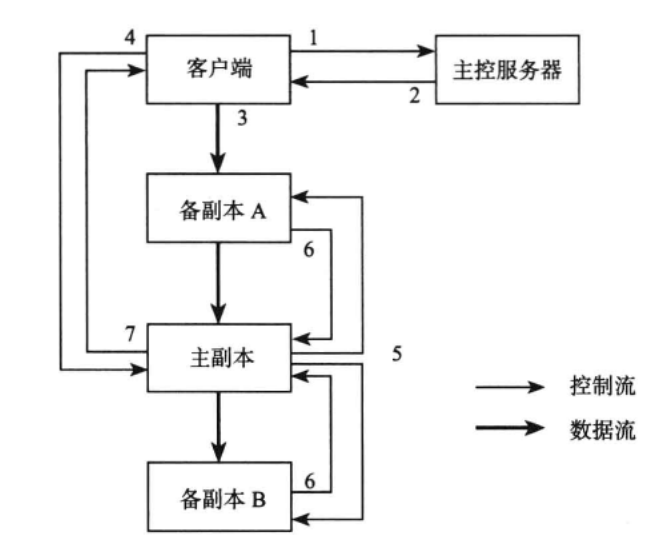
我们只讨论追加的一致性。如果不发生异常，追加成功的记录在GFS的各个副本中是确定并且严格一致的；但是如果出现了异常，可能出现某些副本追加成功而某些副本没有成功的情况，失败的副本可能会产生一些可识别的填充（padding）记录。**GFS客户端追加失败将重试**，只要返回用户追加成功，说明在所有副本中都至少追加成功了一次。当然，可能出现记录在某些副本中被追加了多次，即重复记录；也可能出现一些可识别的填充记录，应用层需要能够处理这些问题。

另外，由于GFS支持多个客户端并发追加，多个客户端之间的顺序是无法保证的，同一个客户端连续追加成功的多个记录也可能被打断，比如客户端先后追加成功记录R1和R2，由于追加R1和R2这两条记录的过程不是原子的，中途可能被其他客户端打断，那么GFS的chunk中记录的R1和R2可能不连续，中间夹杂着其他客户端追加的数据。

GFS的这种一致性模型是追求性能导致的，这增加了应用开发的难度。对于MapReduce应用，由于其批处理特性，可以先将数据追加到一个临时文件，在临时文件中维护索引记录每个追加成功的记录的偏移，等到文件关闭时一次性将临时文件改名为最终文件。对于上层的Bigtable，有两种处理方式。

#### 追加流程

追加流程是GFS系统中最为复杂的地方，而且，高效支持记录追加对于基于GFS实现的分布式表格系统Bigtable是至关重要的。其大致流程如下：



#### 容错机制

### Master设计

### ChunkServer设计

注：ChunkServer副本管理是怎么设计的？采用什么算法打散副本到不同服务器？

### 讨论

## TFS

TFS架构设计时需要考虑如下两个问题：

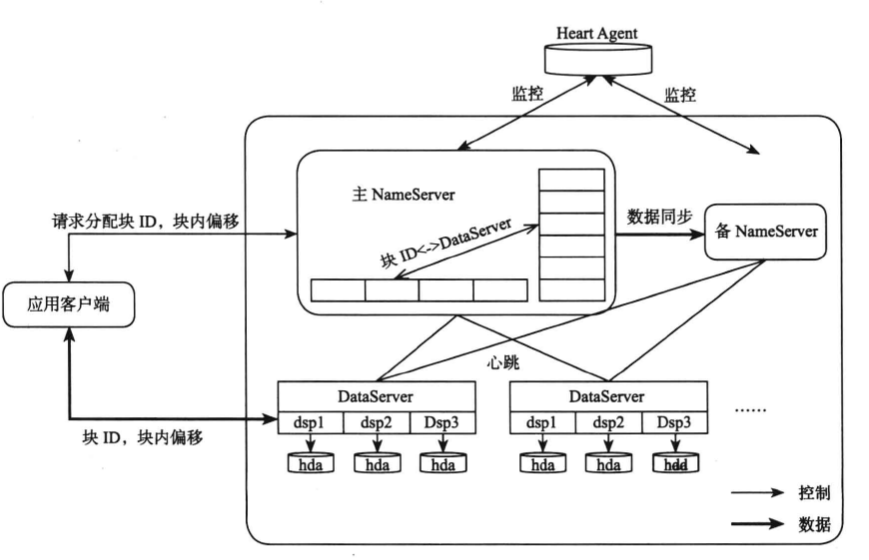
1. Metadata信息存储。由于图片数量巨大，单机存放不了所有的元数据信息。
2. 减少图片读取的IO次数。在普通的Linux文件系统中，读取一个文件包括三次磁盘IO：首先读取目录元数据到内存，其次把文件的inode节点装载到内存，最后读取实际的文件内容。由于小文件个数太多，无法将所有目录及文件的inode信息缓存到内存，因此磁盘IO次数很难达到每个图片读取只需要一次磁盘IO的理想状态。

因此，TFS设计时采用的思路是：多个逻辑图片文件共享一个物理文件。

### 系统架构

TFS架构上借鉴了GFS，但是与GFS又有很大的不同。首先，TFS内部不维护文件目录树，每个小文件使用一个64位的编号表示；其次，TFS是一个读多写少的应用，相比GFS，TFS的写流程可以做得更加简单有效。

如图所示，一个TFS集群由两个NameServer节点（一主一备）和多个DataServer节点组成，NameServer通过心跳对DataServer的状态进行监测。NameServer相当于GFS中的Master，DataServer相当于GFS中的ChunkServer。NameServer区分为主NameServer和备NameServer，只有主NameServer提供服务，当主NameServer出现故障时，能够被心跳守护进程检测到，并将服务切换到备NameServer。每个DataServer上会运行多个dsp进程，一个dsp进程对应一个挂载点，这个挂载点一般对应一个独立磁盘，从而管理多块磁盘。



### 讨论

## Haystack

# 内容分发网络