# 分布式文件系统

分布式文件系统的主要功能有两个：

1. 存储文档、图像、视频等Blob类型文件；
2. 作为分布式表格系统的持久化层。

## Google File System

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

中文版GFS论文

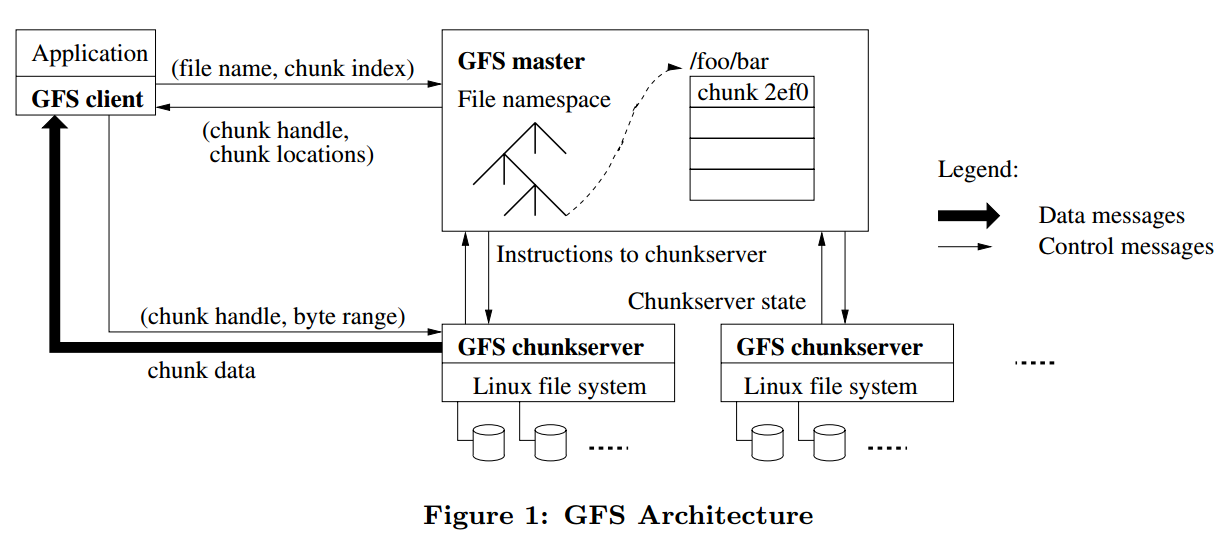
<http://www.open-open.com/lib/view/open1328763454608.html>

### 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：

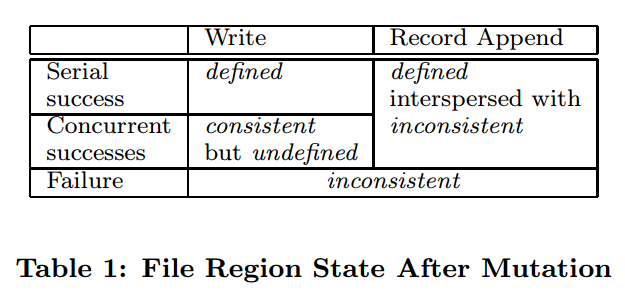
* 持久化存储metadata；
* 它的存储顺序定义了client并行操作最终的操作顺序。

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。

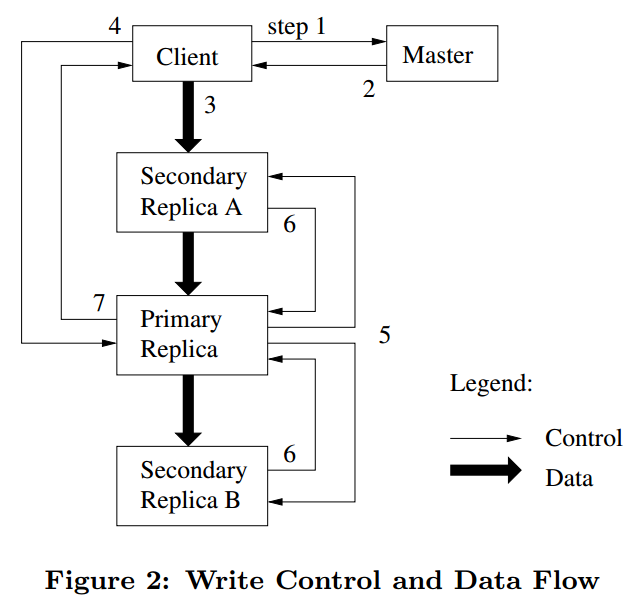
**优点：**

* 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

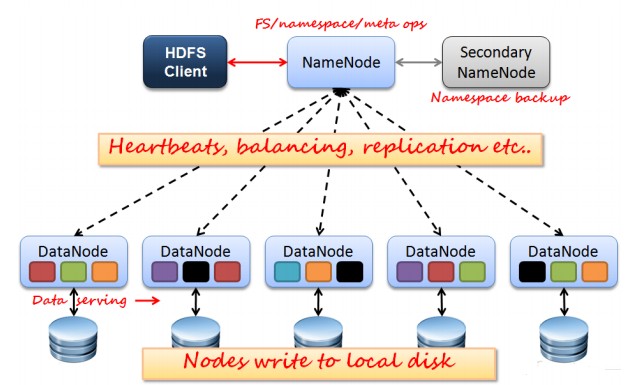
利用checksum机制，分读和写两种情况来讨论：

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。

## HDFS

<https://www.cnblogs.com/laov/p/3434917.html>



HDFS也是采用Master和Slave结构，在系统中分为NameNode、SecondaryNameNode和DataNode这三个角色。

**NameNode**

也称为Master节点。在集群中的主要任务有：管理数据块映射；处理客户端的读写请求；配置副本策略；管理HDFS的名称空间。

**SecondaryNameNode**

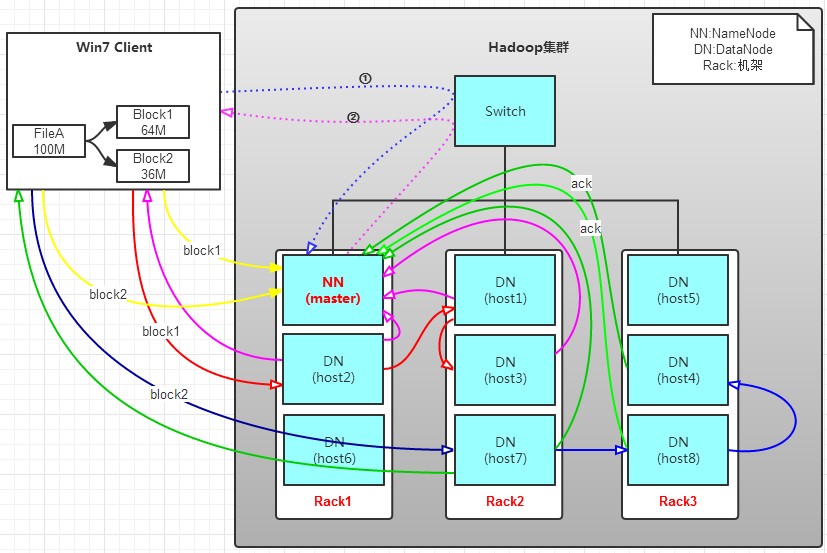
从命名就可以看出在系统中的作用，主要重要是分担NameNode的工作量，是作为NameNode的冷备份，合并fsimage（元数据镜像文件，是文件系统的目录树）和fsedits（元数据的操作日志）然后再发给namenode。

**DataNode**

称为Slave节点，负责存储clinet发来的数据块block；执行数据块的读写操作。

NameNode内存中存储的是fsimage+fsedits，SecondaryNameNode负责定时（默认1小时）从NameNode上获取fsimage和fsedits来进行合并，然后再发送给namenode，减少namenode的工作量。

### 写操作



假设客户端有一个100M的文件FileA，client要把FileA写入到HDFS中。HDFS按照上图的配置，block大小为64MB，HDFS分布在Rock1、Rock2和Rock3上。

1. Client将FileA按64M分块。分成两块，block1和Block2；
2. Client向nameNode发送写数据请求，如图蓝色虚线；
3. NameNode节点，记录block信息。并返回可用的DataNode，如粉色虚线；

Block1: host2,host1,host3

Block2: host7,host8,host4

master选择DataNode的原理是：

NameNode具有RackAware机架感知功能，这个可以配置。若client为DataNode节点，那存储block时，规则为：副本1，同client的节点上；副本2，不同机架节点上；副本3，同第二个副本机架的另一个节点上；其他副本随机挑选。若client不为DataNode节点，那存储block时，规则为：副本1，随机选择一个节点上；副本2，不同副本1，机架上；副本3，同副本2相同的另一个节点上；其他副本随机挑选。

1. client向DataNode发送block1；发送过程是以流式写入。

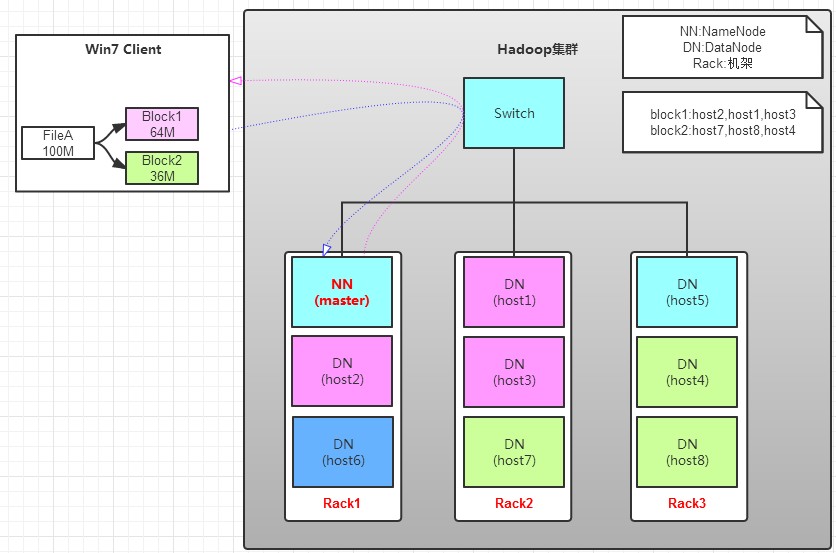
具体的写入流程为：

* 将64M的block1按64k的package划分;
* 然后将第一个package发送给host2;
* host2接收完后，将第一个package发送给host1，同时client向host2发送第二个package；
* host1接收完第一个package后，发送给host3，同时接收host2发来的第二个package。
* 以此类推，如图红线实线所示，直到将block1发送完毕。
* host2,host1,host3向NameNode，host2向Client发送通知，说“消息发送完了”。如图粉红颜色实线所示。
* client收到host2发来的消息后，向namenode发送消息，说我写完了。这样就真完成了。如图黄色粗实线
* 发送完block1后，再向host7，host8，host4发送block2，如图蓝色实线所示。
* 发送完block2后，host7,host8,host4向NameNode，host7向Client发送通知，如图浅绿色实线所示。
* client向NameNode发送消息，说我写完了，如图黄色粗实线。。。这样就完毕了。

通过分析写操作的过程，可以发现：

1. 写1T文件，我们需要3T的存储，3T的网络流量贷款。
2. 在执行读或写的过程中，NameNode和DataNode通过HeartBeat进行保存通信，确定DataNode活着。如果发现DataNode死掉了，就将死掉的DataNode上的数据，放到其他节点去。读取时，要读其他节点去。
3. 挂掉一个节点，没关系，还有其他节点可以备份；甚至，挂掉某一个机架，也没关系；其他机架上，也有备份。

### 读操作



读操作就简单一些了，如图所示，client要从datanode上，读取FileA。而FileA由block1和block2组成。读取的流程为：

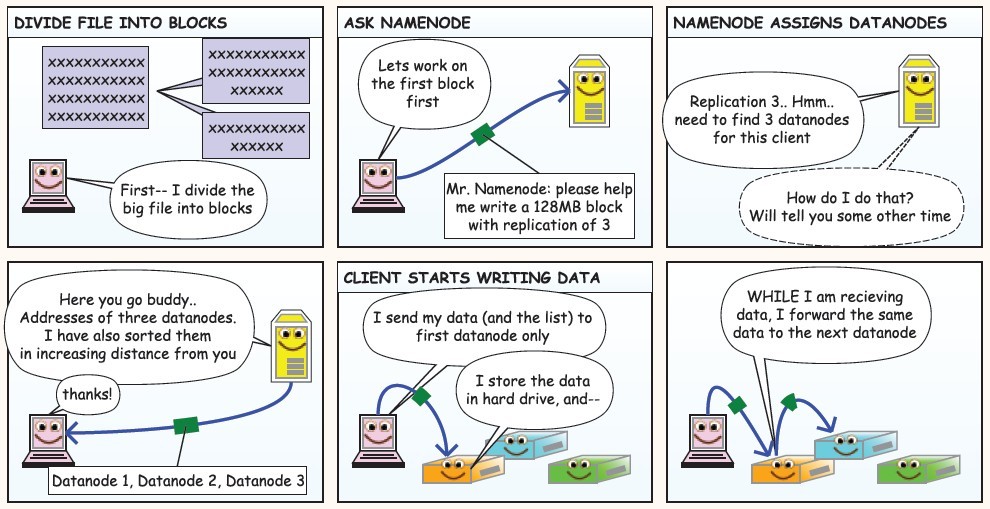
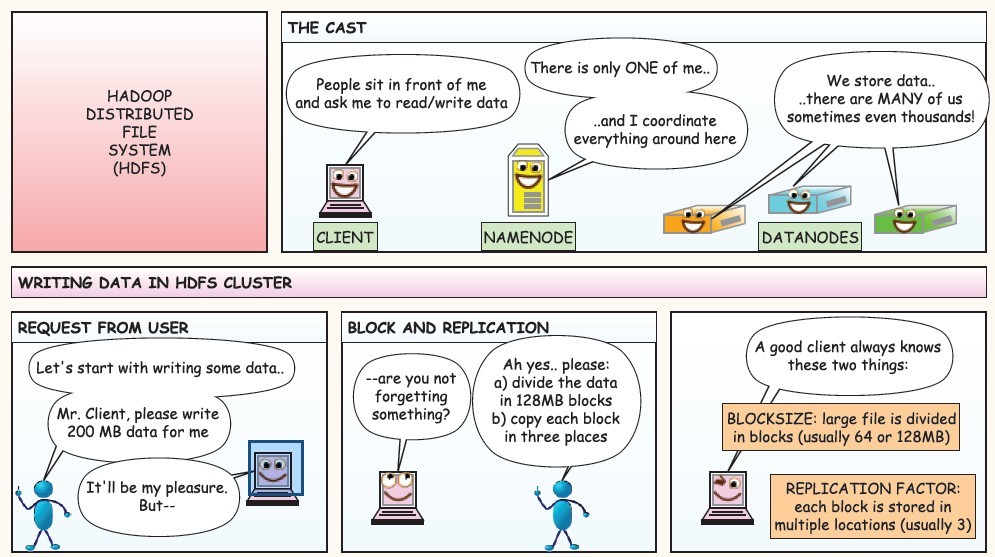
1. client向namenode发送读请求。
2. namenode查看Metadata信息，返回fileA的block的位置。block1:host2,host1,host3

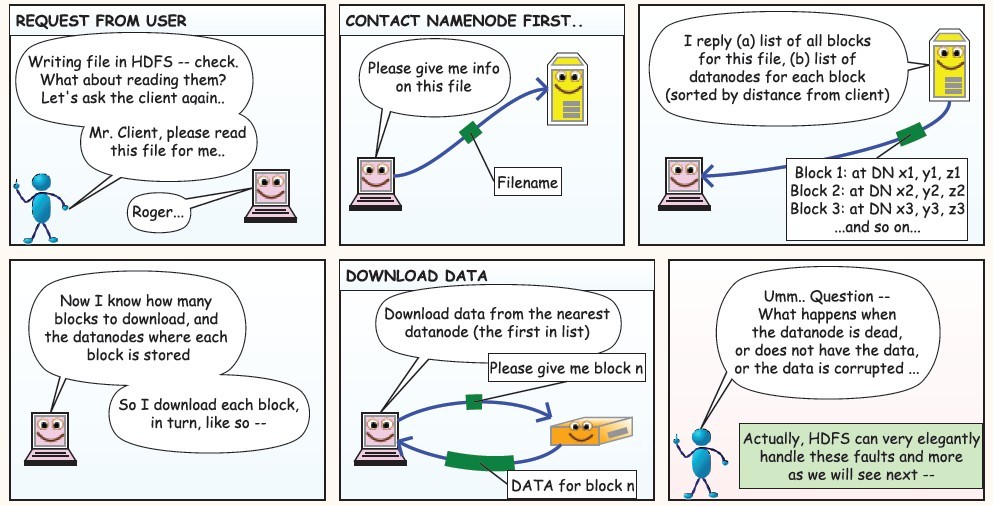
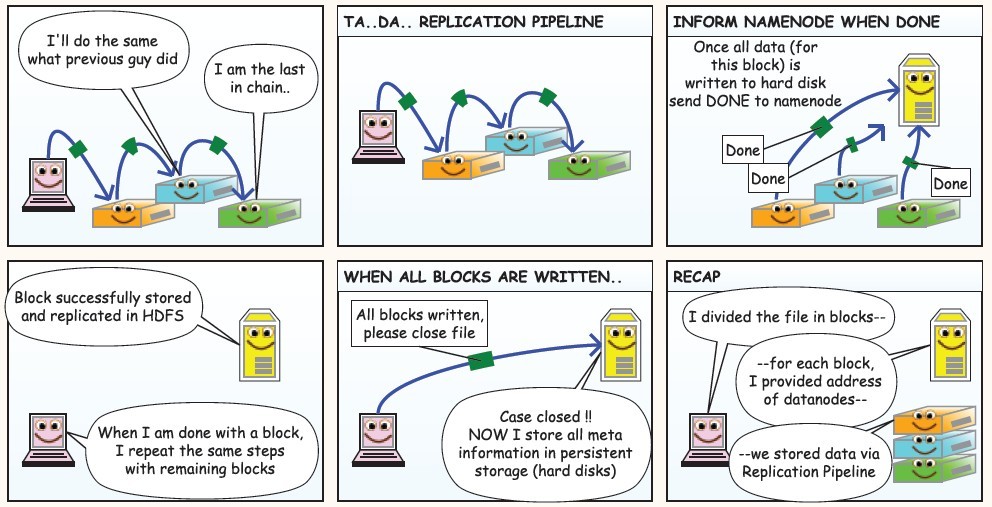
block2:host7,host8,host4

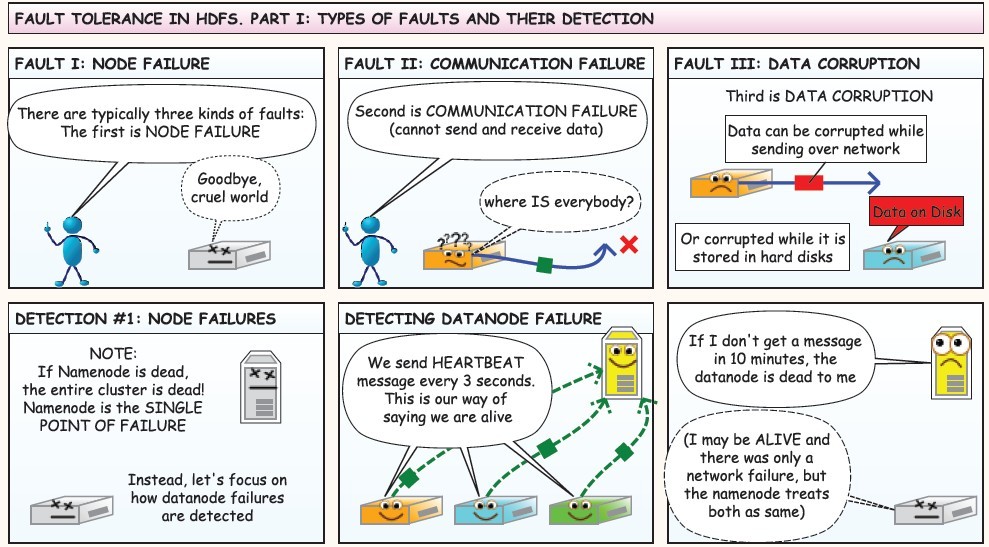
1. block的位置是有先后顺序的，先读block1，再读block2。而且block1去host2上读取；然后block2，去host7上读取。

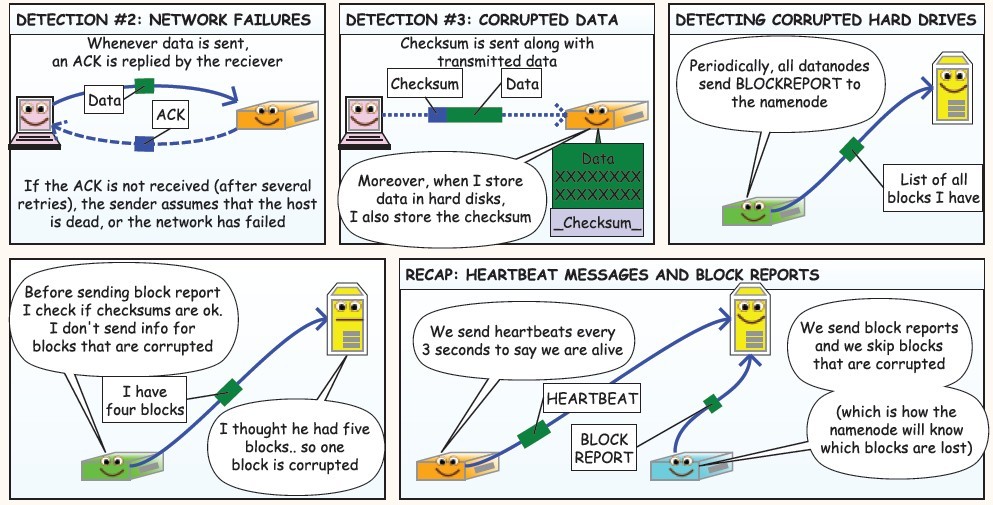
上面例子中，client位于机架外，那么如果client位于机架内某个DataNode上，例如,client是host6。那么读取的时候，遵循的规律是：优选读取本机架上的数据。

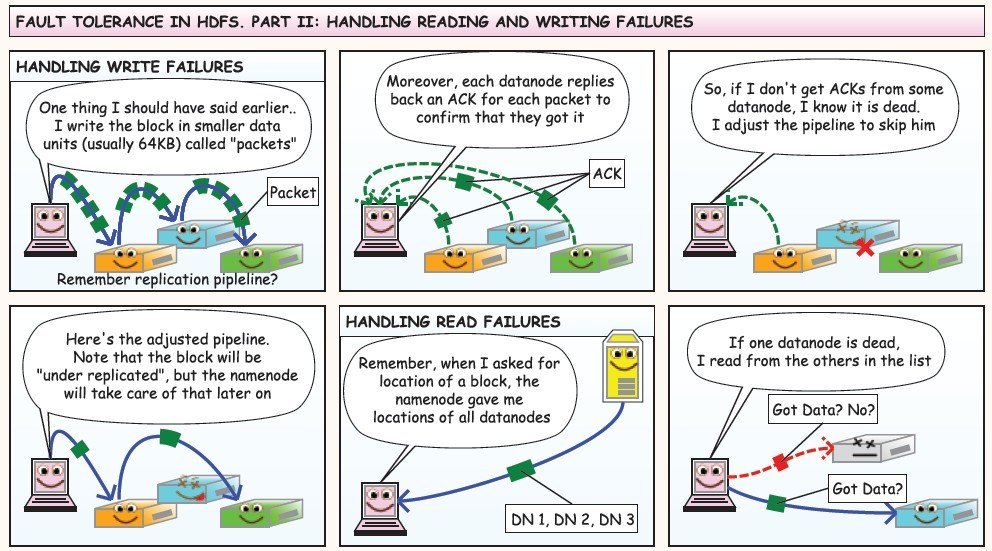
### 漫画图解HDFS

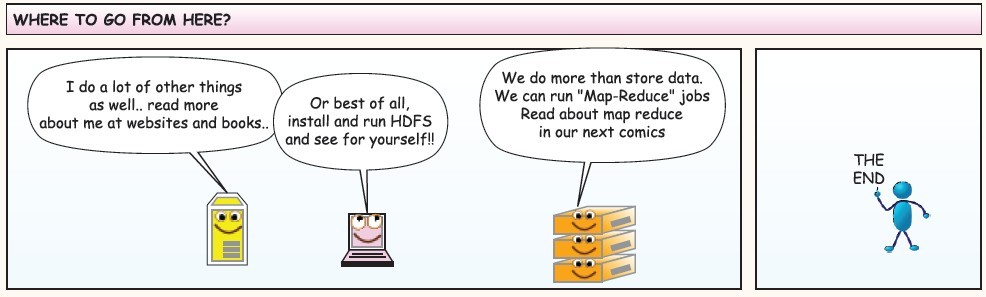
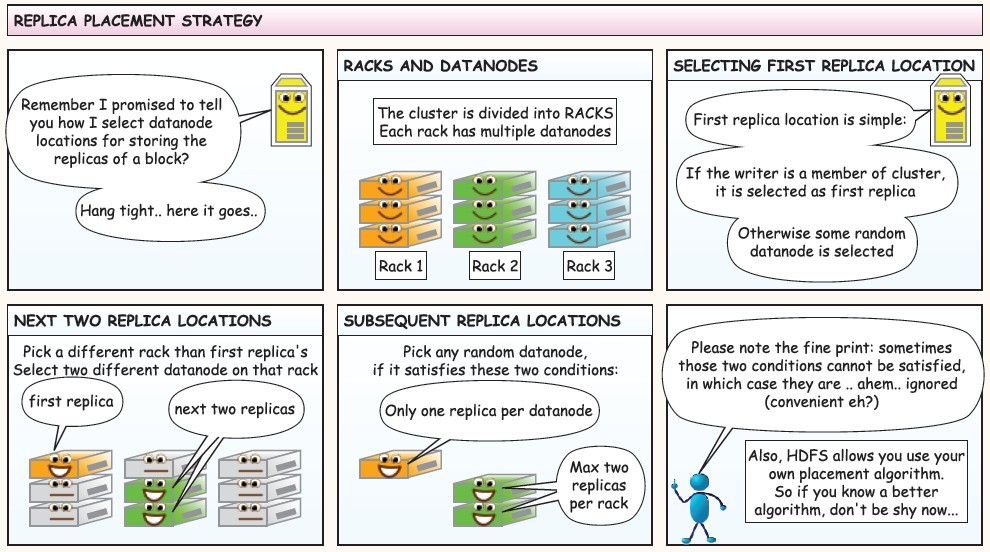
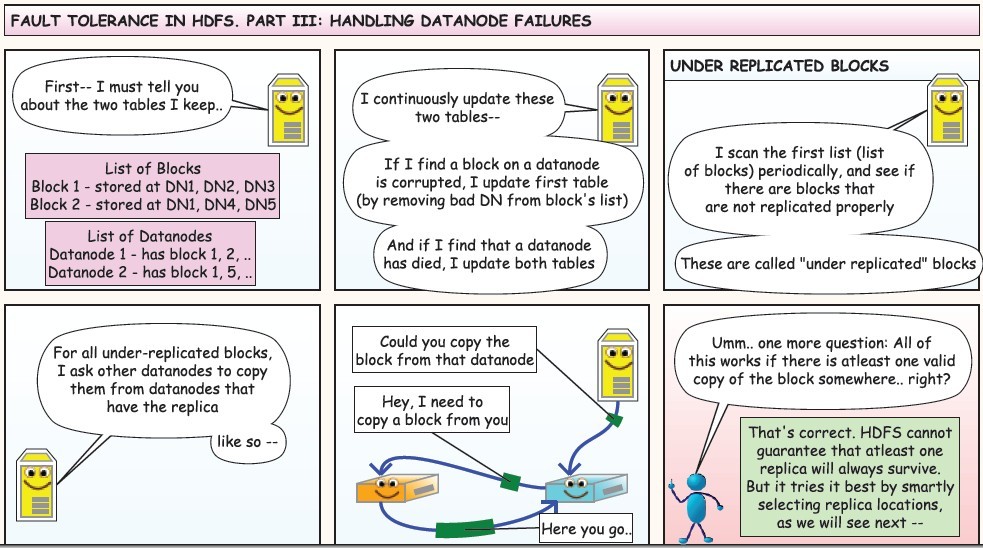












## Taobao File System

TFS为淘宝提供海量小文件存储，通常文件大小不超过1M，满足了淘宝对小文件存储的需求，被广泛地应用在淘宝各项应用中。它采用了HA（High Available，高可用性集群，双机集群，一般有主备节点）架构和平滑扩容，保证了整个文件系统的可用性和扩展性。同时扁平化的数据组织结构，可将文件名映射到文件的物理地址，简化了文件的访问流程，一定程度上为TFS提供了良好的读写性能。

TFS架构设计时需要考虑如下两个问题：

Metadata信息存储。假设每个文件的元数据100B，100亿图片的元数据需要1TB，单台机器无法提供元数据服务；

减少图片的IO次数：普通Linux文件系统中，读取一个文件需要三次磁盘IO，先读取目录元数据到内存，再把文件的inode节点读入内存，最后读取文件数据到内存。**由于小文件数量太多，无法把所有的目录及文件的inode信息缓存到内存中，因此磁盘IO次数很难达到每个图片读取只需要一次IO理想状态。**

TFS设计时采用的思路：多个逻辑图片文件共享一个物理文件。

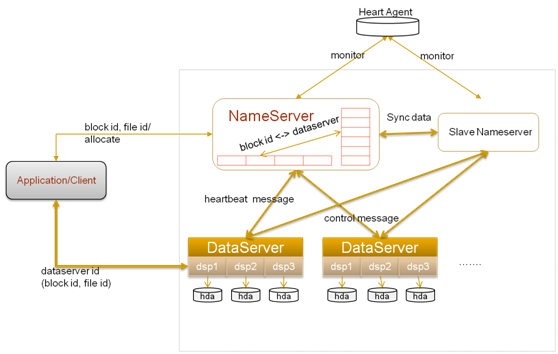
### TFS总体结构

一个TFS集群由两个NameServer节点（一主一备）和多个DataServer节点组成。这些服务程序都是作为一个用户级的程序运行在普通Linux机器上的。在TFS中，将大量的小文件(实际数据文件)合并成为一个大文件，这个大文件称为块(Block), 每个Block拥有在集群内唯一的编号(Block Id), Block Id在NameServer创建Block的时候分配, NameServer维护Block与DataServer的关系。Block中的实际数据都存储在DataServer上。而一台DataServer服务器一般会有多个独立DataServer进程存在，每个进程负责管理一个挂载点，这个挂载点一般是一个独立磁盘上的文件目录，以降低单个磁盘损坏带来的影响。

NameServer主要功能是: 管理维护Block和DataServer相关信息,包括DataServer加入、退出，心跳信息，Block和DataServer的对应关系建立、解除。正常情况下，一个块会在DataServer上存在，主NameServer负责Block的创建、删除、复制、均衡、整理。NameServer不负责实际数据的读写，实际数据的读写由DataServer完成。

DataServer主要功能是: 负责实际数据的存储和读写。

为了考虑容灾，NameServer采用了HA结构，即两台机器互为热备，同时运行，一台为主，一台为备，主机绑定到对外vip（Virtual IP），提供服务；当主机器宕机后，迅速将vip绑定至备份NameServer，将其切换为主机，对外提供服务。通过心跳机制实现故障检测。



TFS内部**不维护文件目录树**，每个**小文件用64位的编号表示**。TFS将大量的小文件合并成一个大文件，大文件称为Block，每个Block拥有在集群内唯一的编号（块ID），通过<块ID，文件编号>可以唯一确定一个文件。Block大小一般为64MB，在DataServer存储三份。DataServer进程会给Block中的每个文件分配一个ID(File ID，该ID在每个Block中唯一)，并将**每个文件在Block中的信息存放在和Block对应的Index文件中**。这个Index文件一般都会全部load在内存，除非出现DataServer服务器内存和集群中所存放文件平均大小不匹配的情况。

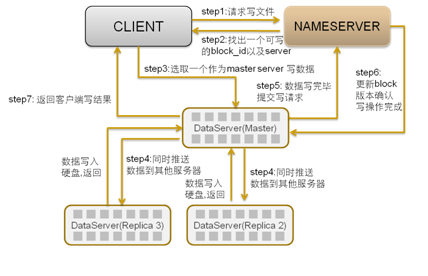
另外，还可以部署一个对等的TFS集群，作为当前集群的辅集群。辅集群不提供来自应用的写入，只接受来自主集群的写入。当前主集群的每个数据变更操作都会重放至辅集群。辅集群也可以提供对外的读，并且在主集群出现故障的时候，可以接管主集群的工作。

客户端不缓存文件数据，只缓存NameServer的元数据。

### 写流程

TFS的追加流程比GFS简单有效（原因是根据所要服务业务的特点，一个分布式系统的设计是根据自己业务需求不断改进得到的），GFS为了减少Master的压力，引入了租约机制，将修改权限下放到主chunkserver，很多的追加操作不需要Master参与。TFS是写少读多的，即使每次写操作要经过NameNode也没关系，这样才大大简化了系统的设计。TFS不支持类似GFS多客户端并发追加操作，同一时刻每个block只能有一个写操作，多个客户端写操作会被串行化。

在TFS系统中，NameServer会保证一个文件有多个副本存储于不同的DataServer上以保证冗余。当由于DataServer服务器宕机或者由于其他原因退出系统导致某些文件副本数量下降时，NameServer将会调度新的DataServer节点存储文件备份。同样为了保证数据一致性，当写入一个文件时，只有所有参与的dataserver均写入成功时，该操作才算成功。TFS的写操作数据流图如下所示：



客户端首先向nameserver发起写请求，nameserver需要根据dataserver上的可写块，容量和负载加权平均来选择一个可写的Block。并且在该Block所在的多个dataserver中选择一个作为写入的master，这个选择过程也需要根据dataserver的负载以及当前作为master的次数来计算，使得每个dataserver作为master的机会均等。**master一旦选定，除非master宕机，不会更换**，一旦master宕机，需要在剩余的dataserver中选择新的master。返回一个dataserver列表。

客户端向master dataserver开始数据写入操作。master server将数据传输为其他的dataserver节点，只有当所有dataserver节点写入均成功时，master server才会向nameserver和客户端返回操作成功的信息。

相比GFS，TFS的写流程不够优化。

1. 每个写请求都需要多次访问NameServer；
2. DataServer之间数据的推送没有采用流水线方式减少延迟。

写操作返回后，会返回客户端两个信息，小文件在TFS中的block编号以及文件编号。系统将这些信息保存在数据库中。读文件时，首先根据block编号从NameServer查找block所在的DataServer，然后根据文件编号读取图片数据。

### 读流程

1. **获得Block ID和File ID**

根据TFS文件名解析出Block ID和Block中的File ID.

1. **获取dataserver地址**

向nameserver发送查询请求得到Block ID所在的dataserver地址。

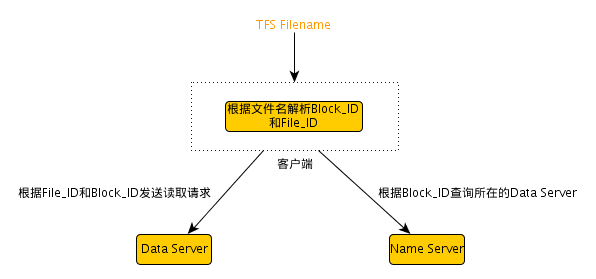
由于nameserver中维护了Block和dataserver的对应关系，所以nameserver能够提供相应的信息。

Note: 由于TFS是把大量小文件放在一个Block里面，所以TFS的文件复制是基于Block的，而且复制出来的Block的block id应该是一致的。

1. **请求文件**

通过发送Block\_ID、File\_ID和offset为参数的读请求到对应的dataserver，得到文件内容。

dataserver会根据本地记录的信息来得到File ID所在block的偏移量，从而读取到正确的文件内容.



### 平滑扩容

原有TFS集群运行一定时间后，集群容量不足，此时需要对TFS集群扩容。由于DataServer与NameServer之间使用心跳机制通信，如果系统扩容，只需要将相应数量的新DataServer服务器部署好应用程序后启动即可。这些DataServer服务器会向NameServer进行心跳汇报。NameServer会根据DataServer容量的比率和DataServer的负载决定新数据写往哪台DataServer的服务器。根据写入策略，容量较小，负载较轻的服务器新数据写入的概率会比较高。同时，在集群负载比较轻的时候，NameServer会对DataServer上的Block进行均衡，使所有DataServer的容量尽早达到均衡。

进行均衡计划时，首先计算每台机器应拥有的blocks平均数量，然后将机器划分为两堆，一堆是超过平均数量的，作为移动源；一类是低于平均数量的，作为移动目的。

移动目的的选择：首先一个block的移动的源和目的，应该保持在同一网段内，也就是要与另外的block不同网段；另外，在作为目的的一定机器内，优先选择同机器的源到目的之间移动，也就是同台DataServer服务器中的不同DataServer进程。

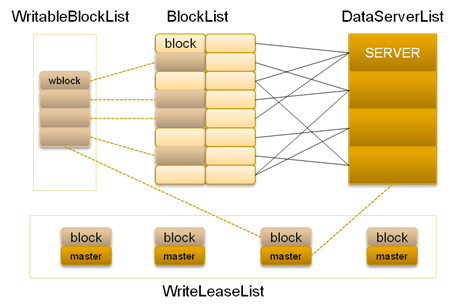
当有服务器故障或者下线退出时（单个集群内的不同网段机器不能同时退出），不影响TFS的服务。此时NameServer会检测到备份数减少的block，对这些block重新进行数据复制。

在创建复制计划时，一次要复制多个block, 每个block的复制源和目的都要尽可能的不同，并且保证每个block在不同的子网段内。因此采用轮换选择(roundrobin)算法，并结合加权平均。

由于DataServer之间的通信是主要发生在数据写入转发的时候和数据复制的时候，集群扩容基本没有影响。假设一个Block为64M，数量级为1PB。那么NameServer上会有 1 \* 1024 \* 1024 \* 1024 / 64 = 16.7M个block。假设每个Block的元数据大小为0.1K，则占用内存不到2G。

### 存储机制

在TFS中，将大量的小文件(实际用户文件)合并成为一个大文件，这个大文件称为块(Block)。TFS以Block的方式组织文件的存储。每一个Block在整个集群内拥有唯一的编号，这个编号是由NameServer进行分配的，而DataServer上实际存储了该Block。在NameServer节点中存储了所有的Block的信息，一个Block存储于多个DataServer中以保证数据的冗余。对于数据读写请求，均先由NameServer选择合适的DataServer节点返回给客户端，再在对应的DataServer节点上进行数据操作。NameServer需要维护Block信息列表，以及Block与DataServer之间的映射关系，其存储的元数据结构如下：



在DataServer节点上，在挂载目录上会有很多物理块，物理块以文件的形式存在磁盘上，并在DataServer部署前预先分配，以保证后续的访问速度和减少碎片产生。为了满足这个特性，DataServer现一般在EXT4文件系统上运行。物理块分为主块和扩展块，一般主块的大小会远大于扩展块，使用扩展块是为了满足文件更新操作时文件大小的变化。每个Block在文件系统上以“主块+扩展块”的方式存储。每一个Block可能对应于多个物理块，其中包括一个主块，多个扩展块。

在DataServer端，每个Block可能会有多个实际的物理文件组成：一个主Physical Block文件，N个扩展Physical Block文件和一个与该Block对应的索引文件。Block中的每个小文件会用一个Block内唯一的fileid来标识。DataServer会在启动的时候把自身所拥有的Block和对应的Index加载进来。

### 容错

**集群容错**

TFS可以配置主辅集群，一般主辅集群会存放在两个不同的机房。主集群提供所有功能，辅集群只提供读。主集群会把所有操作重放到辅集群。这样既提供了负载均衡，又可以在主集群机房出现异常的情况不会中断服务或者丢失数据。

**NameServer容错**

Namserver主要管理了DataServer和Block之间的关系。如每个DataServer拥有哪些Block，每个Block存放在哪些DataServer上等。同时，NameServer采用了HA结构，一主一备，主NameServer上的操作会重放至备NameServer。如果主NameServer出现问题，可以实时切换到备NameServer。另外NameServer和DataServer之间也会有定时的heartbeat，DataServer会把自己拥有的Block发送给NameServer。NameServer会根据这些信息重建DataServer和Block的关系。

**DataServer容错**

TFS采用Block存储多份的方式来实现DataServer的容错。每一个Block会在TFS中存在多份，一般为3份，并且分布在不同网段的不同DataServer上。对于每一个写入请求，必须在所有的Block写入成功才算成功。当出现磁盘损坏DataServer宕机的时候，TFS启动复制流程，把备份数未达到最小备份数的Block尽快复制到其他DataServer上去。TFS对每一个文件会记录校验crc，当客户端发现crc和文件内容不匹配时，会自动切换到一个好的block上读取。此后客户端将会实现自动修复单个文件损坏的情况。

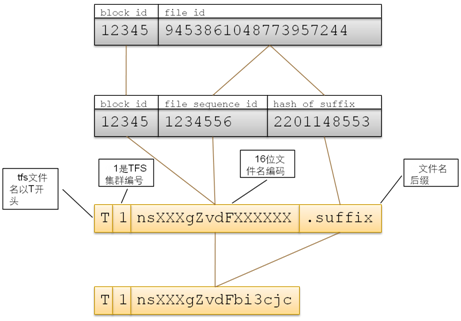
### 并发

对于同一个文件来说，多个用户可以并发读。

现有TFS并不支持并发写一个文件。一个文件只会有一个用户在写。这在TFS的设计里面对应着是一个block同时只能有一个写或者更新操作。

### TFS文件名结构

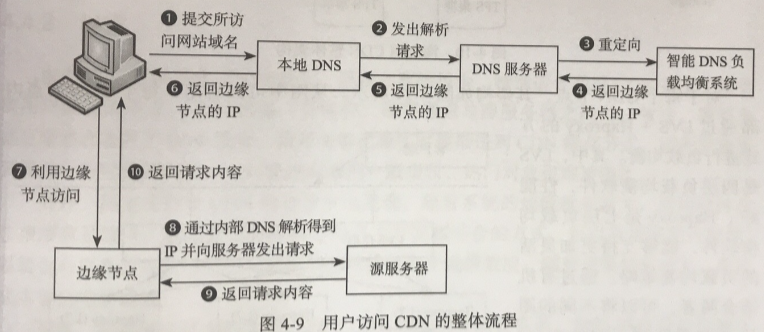
TFS的文件名由块号和文件号通过某种对应关系组成，最大长度为18字节。文件名固定以T开始，第二字节为该集群的编号(可以在配置项中指定，取值范围 1-9)。余下的字节由Block ID和File ID通过一定的编码方式得到。文件名由客户端程序进行编码和解码，它映射方式如下图：



TFS客户程序在读文件的时候通过将文件名转换为BlockID和FileID信息，然后可以在NameServer取得该块所在DataServer信息（如果客户端有该block与DataServere的缓存，则直接从缓存中取），然后与DataServer进行读取操作。

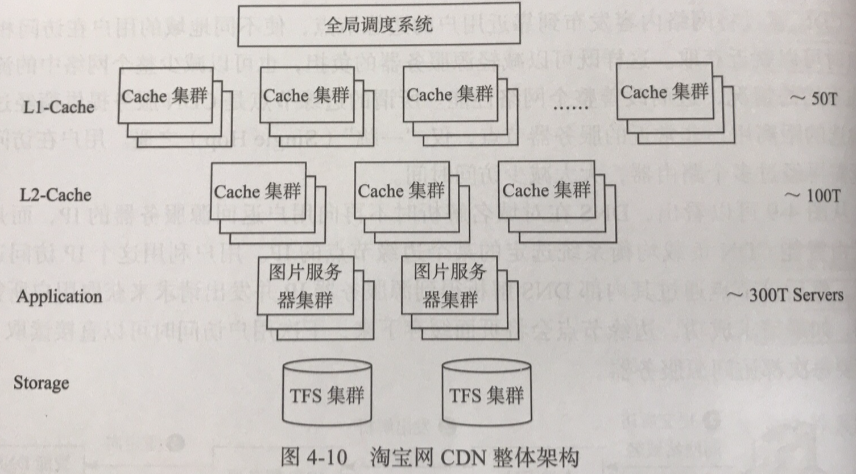
## 内容分发网络

CDN（Content Delivery Network）通过将网络内容发布到靠近用户的边缘节点，使不同区域的用户在访问相同网页时可以就近获取。边缘节点是CDN服务提供商经过挑选出来的距离用户非常近的服务器节点，用户访问时无需经过多个路由，大大减少访问时间。



客户端通过智能DNS负载均衡系统得到边缘节点的地址后访问边缘节点，边缘节点向服务器请求数据，如果请求成功，边缘节点会将页面缓存下来，下次用户访问时可以直接从边缘节点读取页面内容，不需要每次都访问服务器。

### 淘宝CND架构



淘宝CDN采用两级Cache，用户访问淘宝网的图片时，通过全局调度系统调度到某个L1-cache节点，如果L1命中，直接将图片数据返回；否则请求L2-cache节点，并将返回的图片缓存到L1-cache，如果L2-cache命中，直接将图片数据返回给L1-cache，否则请求源服务器的图片服务器集群。每台图片服务器是一个运行Nginx的web服务器，它会本地缓存图片，只有当在图片服务器本地缓存不命中的情况下才会请求TFS集群。图片服务器集群和TFS集群部署在同一个数据中心内。

## Facebook Haystack

<https://www.cnblogs.com/yuki-lau/archive/2013/03/29/2988594.html>

<http://www.importnew.com/3292.html> 差不多就是Haystack的论文以及与TFS的比较

Facebook每张照片平均80KB，用户每周新增照片10亿，总大小60TB，平均每秒新增照片数为3500次，也就是每秒约3500次写操作，读操作峰值可达每秒百万次。用户上传超过65 billion的图片，对每个上传的照片，Facebook生成和存储4种不同大小的图片，这就产生了超过260 billion张图片，超过20 PB的数据（考虑副本情况，而且一张照片被存储为4种不同大小的）。

Facebook最初采用的是NAS(Network Attached Storage)和NFS(Network File System)存储图片，这种设计因为元数据查询而导致了过多的磁盘操作。所以要竭尽全力的减少每个图片的元数据，让Haystack能**在内存中执行所有的元数据查询**。这个突破让系统腾出了更多的性能来读取真实的数据，增加了整体的吞吐量。

传统基于POSIX的文件系统的缺点主要是目录和每个文件的元数据。对于图片应用，很多元数据（比如文件权限），是无用的而且浪费了很多存储容量。而且更大的性能消耗在于文件的元数据必须从磁盘读到内存来定位文件。文件规模较小时这些花费无关紧要，然而面对几百billion的图片和PB级别的数据，访问元数据就是吞吐量瓶颈所在。

通常情况下，我们读取单个照片就需要好几个磁盘操作：一个（有时候更多）转换文件名为inode number，另一个从磁盘上读取inode，最后一个读取文件本身。简单来说，为了查询元数据使用磁盘I/O是限制吞吐量的重要因素。在实际生产环境中，我们必须依赖内容分发网络来支撑主要的读取流量，即使如此，文件元数据的大小和I/O同样对整体系统有很大影响。

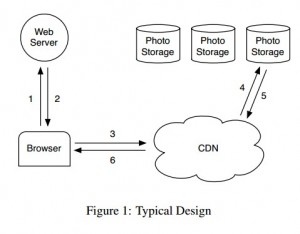
Haystack要实现高吞吐量和低延迟，满足海量用户的查询请求。通常超过处理容量上限的请求，要么被忽略（对用户体验是不可接受的），要么被CDN处理（成本昂贵而且可能遭遇一个性价比转折点）。想要用户体验好，图片查询必须快速。Haystack希望每个读操作至多需要一个磁盘操作，基于此才能达到高吞吐量和低延迟。为了实现这个目标，我们竭尽全力的减少每个图片的必需元数据，然后将所有的元数据保存在内存中。

### 存储场景

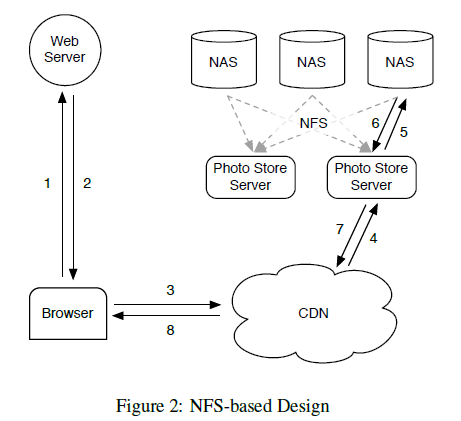
Haystack的使用场景是：

1. write once, read often, never modified and rearly deleted;
2. 海量小文件。Facebook存储了海量的图片文件；用户上传一张图片，Facebook会将其生产4种尺寸的图片，每种存储3份，因此写入量是用户上传量的12倍（准确的说，由于尺寸的缩小，写入量不应该是12倍，二是写入次数是12倍）。

### 原有设计



这是一个概览图，描述了web服务器、CDN和存储系统如何交互协作来实现一个热门站点的图片服务。访问一个页面时，浏览器发送请求到web服务器，对于请求中包含的图片，web服务器会为其构造URL，引导浏览器到对应的位置下载图片。如果是热门的图片，这个URL通常会指向一个CDN，如果CDN缓存了此图片，那么它立刻就会把数据返回给浏览器；如果CDN没有缓存图片，CDN检查URL，URL中包含了足够的信息以供CDN从本站点的存储系统中检索图片。拿到图片后CDN更新它的缓存并将图片发送给浏览器。



上图是基于NFS的图片存储系统设计。对于一个热门的社交网络站点，只有CDN不足以为图片服务提供一个实用的解决方案。对于热门图片，CDN确实可以很高效，但是像Facebook这样的社交网站，会产生大量对不热门图片的请求，称为long tail，long tail的请求也会占据很大流量，并且这些请求还需要访问更下游的图片存储主机，因为这些图片的请求在CDN基本都会命中失败。缓存所有的图片到CDN倒是可以解决问题，但是代价太大了。

基于NFS的设计中，图片文件存储在一组商用NAS设备上，NAS设备的卷被mount到Photo Store Server的NFS上。Photo Store Server解析URL得出卷和完整的文件路径，在NFS上读取数据，然后返回结果给CDN。

最初NFS卷的每个目录下存储几千个文件，导致读取文件时产生了过多的磁盘操作，即使只是读取单个图片，由于NAS设备管理目录元数据的机制，放置几千个文件在一个目录是极其低效的，因为目录的blockmap太大不能被设备有效的缓存。因此检索单个图片都可能需要超过10个磁盘操作。在减少到每个目录下几百个图片后，系统仍然大概需要3个磁盘操作来获取一个图片：一个读取目录元数据到内存、第二个装载inode到内存、最后读取文件内容。

为了减少磁盘操作，我们让图片存储服务器明确的缓存NAS设备返回的文件“句柄”。第一次读取一个文件时，图片存储服务器正常打开一个文件，将文件名与文件“句柄”的映射缓存到memcache中。同时，我们在os内核中添加了一个通过句柄打开文件的接口，当查询被缓存的文件时，图片存储服务器直接用此接口和“句柄”参数打开文件。遗憾的是，文件“句柄”缓存改进不大，因为越冷门的图片越难被缓存到（没有解决long tail问题）。值得讨论的是可以将所有文件“句柄”缓存到memcache，不过这也需要NAS设备能缓存所有的inode信息，这么做是非常昂贵的。总结一下，我们从NAS方案吸取的主要教训是，仅针对缓存——不管是NAS设备缓存还是额外的像memcache缓存——对减少磁盘操作的改进是有限的。存储系统终究是要处理long tail请求（不热门图片）。

**一个文件对应至少一个inode（如果考虑不同尺寸和副本），一个inode已经占了几百byte了，所以在海量图片的情况下，真的需要非常多的内存才可以。（所以要把多个图片存储在一个文件中，这样可以减少很多的内存空间）**

**原有设计的问题**

在POSIX标准的文件系统中：

1. 每个目录、文件都含有元数据，并包含权限等在facebook图片存储场景中无用的信息，占用了不必要的存储空间；
2. 查找一个文件时，需要先将磁盘上的元数据load到内存中，才能找到目标文件，磁盘访问多，效率低；

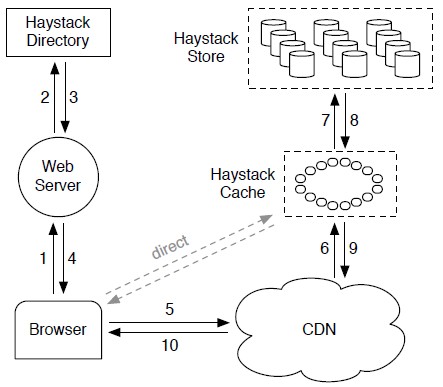
元数据访问成为瓶颈。

Facebook想要使用CDN来支撑热门图片的查询，结合Haystack解决long tail问题。Web站点在查询静态内容遇到I/O瓶颈时，传统的解决方案是使用CDN，它可为下游的存储系统挡住绝大部分的查询请求。但是在Facebook中，为了使底层存储系统不受I/O的摆布，CDN需要缓存难以想象的海量静态内容。

即使有CDN，磁盘操作依然是系统的瓶颈，long tail的请求必然要导致磁盘操作的实现，Haystack会尽量减少除了访问真实图片数据之外其他的操作，并且有效的减少了文件系统元数据的空间，并在内存中保存所有元数据。

### Haystack结构

Haystack的目标：Avoid disk operations when accessing metadata.为了实现这一目标，Haystack系统主要包括三部分：目录、存储以及缓存。



**Haystack Directory**

存放逻辑卷轴和物理卷轴的对应关系，以及照片id到逻辑卷轴之间的映射关系。Haystack缓存主要用于解决对CDN提供商过于依赖的问题，提供最近增加的照片的缓存服务。

1. 逻辑卷和物理卷的对应关系，web服务器上传图片和构建图片URL是都需要这个映射关系；
2. Directory在分配写请求到逻辑卷、分配读请求到物理卷时需要保证负载均衡；
3. Directory要决定一个图片请求应该被发至CDN还是cache，这个功能可以动态调整是否要依赖CDN；
4. Directory指明哪些逻辑卷是只读的，只读意味着存储容量达到上限或者处于运维的原因等，为了运维方便，以机器的粒度来标记卷的只读。

**我们考虑一个网页请求带有图片的情况。网页中包含图片的原始URL，在Directory中利用应用元数据和原始URL构造出“引导URL”以供下游使用。目前只知道Directory中包含了逻辑卷和物理卷之间的映射关系，只有这些关系能够得到最终的URL吗？通常来说原始URL应该只会包含图片的ID，那么Directory如何得知Logical volumn的信息？Directory是不是在upload阶段将photo id与逻辑卷的映射也保存了？如果是，那么这个映射的数量将是非常大的，论文中却没有提起。（Directory中应当保存了很多与photo id相关的元数据，比如所在的逻辑卷、cookie等）**

**Haystack Cache**

cache类似一个内部的CND，它为store提供热门图片的cache，并在上游的CDN节点出现故障时提供庇护，接受客户端的直接请求以重新读取内容。

cache会从CDN或者直接从用户浏览器收到图片查询请求。cache的实现可以理解为一个分布式的Hash table，使用photo id作为key来定位缓存的数据。如果cache未命中，cache根据URL从指定的机器上获取图片，视情况回复给CDN或者用户浏览器。

关于cache中缓存数据的问题值得注意，如果CDN发生了read miss，在cache又发生的read miss，则需要访问store获取数据。按照一般的理解，这个数据应该被缓存在cache中，但是Haystack中，只有符合下面两个条件中的一个时才会缓存图片：

1. The request comes directly from a user and not the CND（一个请求在CDN没有命中一定是非热门图片，那么cache也无需命中，即使图片接下来可能变得活跃，那么也只需要在CDN中可以命中就可以了）;
2. The photo is fetched from a write-enabled store machine.（表明这个图片是最近写入的，因为较早写入的图片所在的物理机很可能已经写满，进入read-only状态）

这两点判断是Facebook对图片分享应用的用户分析得到的：照片在上传后不久就会被访问得最多，对于我们的工作负载，文件系统在读取或写入时通常性能更好，而不是同时读和写。

**Haystack Store**

store机器仅需要逻辑卷ID和offset就能非常快的访问一个图片。**Haystack设计的目的：不需要磁盘操作就可以检索文件名、偏移量、文件大小等元数据。** store机器会将其下所有物理卷的文件描述符（open打开的文件句柄）缓存在内存中，同时photo id到文件系统元数据（文件、偏移量、大小）的映射是检索图片的重要条件，也会缓存在内存中。

store是物理存储节点，以物理卷轴（physical volume）的形式组织存储空间，每个物理卷轴一般都很大，比如100G，这样10TB的数据也只需要100个物理卷轴。每个物理卷轴对应一个物理文件，因此，每个存储节点上的物理文件元数据都很小。多个物理存储节点上的物理卷轴组成一个逻辑卷轴（logical volume），用于备份，Haystack将一个图片存储到逻辑卷轴时，图片被写入到逻辑卷轴对应的所有物理卷，这样实现备份。

Haystack每个逻辑卷轴大小100GB，减少了元信息，但是增加了迁移的时间。

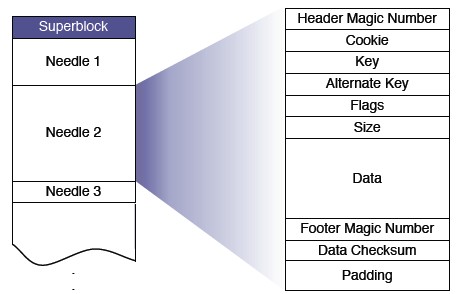
当用户访问一个页面，web服务器使用Directory为每一个图片来构建URL（Directory中有足够的应用元数据来构造URL）。URL包含几块信息，每一块内容可以对应到从浏览器访问CDN（或者cahce）直至最终在一台Store机器上检索到图片的各个步骤。一个典型的URL如下：

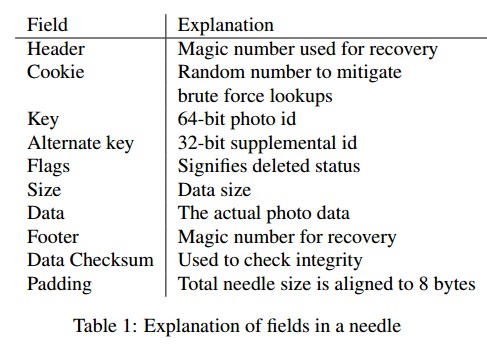
http://CND/cache/Machine id/<Logical volume, Photo>/

CDN指明了从哪个CDN查询图片，到CDN之后再使用<Logical volume, Photo>即可查找缓存的图片。如果CDN没有命中，它从URL中删除CDN相关的信息，然后访问cache。cache其实也是CDN的功能，不过是Haystack系统内部的CDN，所以在cache中查找的过程和CDN类似。如果cache中也没有命中，则从URL中删除cache相关的信息，请求接着被发送给指定Machine id的机器。

### 图片元数据压缩

将多个图片作为一个文件进行存储，这样就可以避免每个图片存储一个文件带来的大量文件元数据的问题。并将该文件称为一个superblock，superblock中的每个图片就称为Needle，一个Needle包含该图片的元数据，和其实际的数据内容，如下图所示：



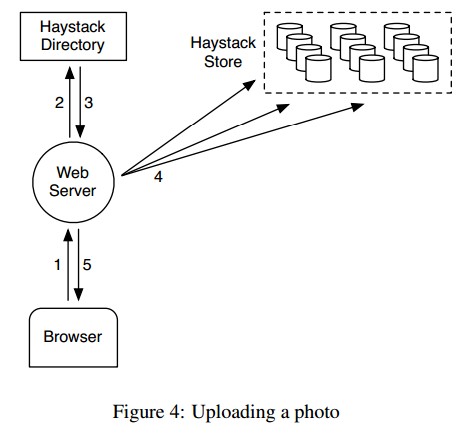


为了快速检索needle，store机器需要为每个卷维护一个内存中的key-value映射。映射的key就是key（photo id）和alternate key的组合，映射的value就是flag、size和卷offset。如果store机器崩溃、重启，它可以分析卷文件来重建这映射关系。

新图片写入的方式与log类似，采用追加的方式。如果一个图片被删除了，则只需对Flags标志位进行置位，并在compaction阶段进行真正的删除。

haystack将每个小文件（needle）在大文件（superblock）中的分布映射保存在内存中，这就减少了查找文件所需的磁盘访问次数。而且这些元信息是根据应用需求而定制的，每个图片需要10字节的元信息（4个尺寸的图片共40字节），而 XFS 需要536字节。

### 写入图片



用户上传一个图片到Haystack时，首先通过浏览器发送数据到web server，web server随后向Directory咨询得到一个可写逻辑卷及其对应的多台Store机器，并且为图片分配一个唯一的ID，随后直接访问这些Store机器，向其提供逻辑卷id、key、alternate key、cookie和真实数据。

每个Store机器为图片创建一个新needle，append到相应的物理卷文件，更新内存中映射。过程很简单，但是append-only策略不能很好的支持修改性的操作，比如旋转（图片顺时针旋转90度之类的）。Haystack并不允许覆盖needle，所以图片的修改只能通过添加一个新needle，其拥有相同的key和alternate key。如果新needle被写入到与老needle不同的逻辑卷，则只需要Directory更新它的应用元数据，未来的请求都路由到新逻辑卷，不会获取老版本的数据。如果新needle写入到相同的逻辑卷，Store机器也只是将其append到相同的物理卷中。Haystack利用一个十分简单的手段来区分重复的needle，那就是判断它们的offset（新版本的needle肯定是offset最高的那个），在构造或更新内存中映射时如果遇到相同的needle，则用offset高的覆盖低的。

### 读取图片

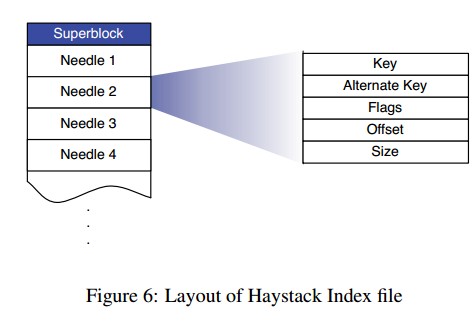
cache向store请求一个图片时，它提供逻辑卷id、key、alternate key和cookie。cookie是一个数字，存在于URL中。新图片上传时Directory为其随机分配一个cookie值，cookie值会作为应用元数据之一存储在Directory。cookie相当于是图片的“私人密码”，可保证所有发往cache或者CDN的请求都是经过Directory批准的。（Cache和Store都持有图片的cookie，若用户自己在浏览器中伪造、猜测URL或发起攻击，则会因为cookie不匹配而失败，从而保证Cache、Store能放心处理合法的图片请求）

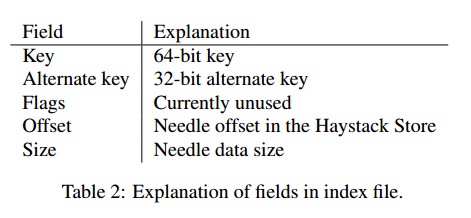
当store机器接收到cache机器发来的图片请求查询时，会利用内存中的映射快速查找相关的元数据。如果图片没有被删除，store则在卷文件中seek到相应的offset，从磁盘上读取整个needle，然后检查cookie和数据完整性，若全部合法则将图片数据返回到cache机器。

### 索引文件

在前面的图片元数据压缩部分提到，store机器故障重启时，它可以分析卷文件来重建卷中元数据的映射关系。起始Haystack使用了索引文件来帮助重启初始化操作。尽管理论上机器可以通过读取所有的物理卷来重新构建它在内存中的映射，但是考虑磁盘大量的数据时，读磁盘还是非常耗时的。索引文件允许store机器快速构建内存中的映射，尽量减少重启时间。

store机器为每个卷维护一个索引文件，索引文件可以想象成内存中映射的一个“存档”，索引文件的布局和卷文件类似，一个超级块包含了一系列索引记录，每个记录对应到各个needle。**索引文件中的记录与卷文件中对应的needle必须保证相同的存储顺序**。





使用索引帮助重启稍微增加了系统复杂度，因为索引文件都是异步更新的，这意味着当前索引文件中的“存档”可能不是最新的。当我们写入一个新图片时，store机器同步append一个needle到卷文件末尾，并异步append一个记录到索引文件。当我们删除图片时，store机器在对应needle上同步设置flag，而不会更新索引文件。这些设计决策是为了让写和删除操作更快返回，避免附加的同步磁盘写。但是也导致了两方面的影响：一个needle可能没有对应的索引记录、索引记录中无法得知图片已删除。

我们将对应不到任何索引记录的needle称为“孤儿”。在重启时，store机器顺序的检查每个孤儿，重新创建匹配的索引记录，append到索引文件。我们能快速的识别孤儿是因为索引文件中最后的记录能对应到卷文件中最后的非孤儿needle。处理完孤儿问题，store机器则开始使用索引文件初始化它的内存中映射。

由于索引记录中无法得知图片已删除，store机器可能去检索一个实际上已经被删除的图片。为了解决这个问题，可以在store机器读取整个needle后检查其flag，若标记为已删除，则更新内存中映射的flag，并回复Cache此对象未找到。

### 故障恢复

为了主动找到有问题的Store机器，我们维护了一个后台任务，称之为pitchfork，它周期性的检查每个Store机器的健康度。pitchfork远程的测试到每台Store机器的连接，检查其每个卷文件的可用性，并尝试读取数据。如果pitchfork确定某台Store机器没通过这些健康检查，它会自动标记此台机器涉及的所有逻辑卷为只读。我们的工程师将在线下人工的检查根本故障原因。

一旦确诊，我们就能快速的解决问题。不过在少数情况下，需要执行一个更加严厉的bulk同步操作，此操作需要使用复制品中的卷文件重置某个Store机器的所有数据。Bulk同步发生的几率很小（每个月几次），而且过程比较简单，只是执行很慢。主要的瓶颈在于bulk同步的数据量经常会远远超过单台Store机器NIC速度，导致好几个小时才能恢复。我们正积极解决这个问题。

### 优化

1. 压缩

压缩操作是直接在线执行的，它能回收已删除的、重复的needle所占据的空间。Store机器压缩卷文件的方式是，逐个复制needle到一个新的卷文件，并跳过任何重复项、已删除项。在压缩时如果接收到删除操作，两个卷文件都需处理。一旦复制过程执行到卷文件末尾，所有对此卷的修改操作将被阻塞，新卷文件和新内存中映射将对前任执行原子替换，随后恢复正常工作。

1. 节省内存

上面描述过，Store机器会在内存中映射中维护一个flag，但是目前它只会用来标记一个needle是否已删除，有点浪费。所以我们通过设置偏移量为0来表示图片已删除，物理上消除了这个flag。另外，映射Value中不包含cookie，当needle从磁盘读出之后Store才会进行cookie检查。通过这两个技术减少了20%的内存占用。

当前，Haystack平均为每个图片使用10byte的内存。每个上传的图片对应4张副本，它们共用同一个key（占64bits），alternate keys不同（占32bits），size不同（占16bits），目前占用(64+(32+16)\*4)/8=32个bytes。另外，对于每个副本，Haystack在用hash table等结构时需要消耗额外的2个bytes，最终总量为一张图片的4份副本共占用40bytes。作为对比，一个xfs\_inode\_t结构在Linux中需占用536bytes。

1. 批量上传

磁盘在执行大型的、连续的写时性能要优于大量小型的随机写，所以我们尽量将相关写操作捆绑批量执行。幸运的是，很多用户都会上传整个相册到Facebook，而不是频繁上传单个图片。因此只需做一些巧妙的安排就可以捆绑批量upload，实现大型、连续的写操作。

## TFS和Haystack

**主要是对两者的存储和检索实现做对比。**

从宏观来看，TFS与Haystack的最大区别就是：TFS只关心存储层面，它没有Haystack的cache组件。Haystack期望提供的是从浏览器、到CDN、到最终存储的一整套解决方案，架构定位稍有不同，Haystack也是专门为这种场景下的图片服务所定制的，做了很多精细的优化；TFS的目标是通用分布式文件存储，除了CDN还会支持其他各种场景。

是使用定制的？还是使用通用的？这个并没有标准答案。

### 存储机器上的文件结构、文件系统元数据对策

Haystack：机器上维护了少量的大型物理卷文件，其中包含一系列needle来存储小文件，同时needle的文件系统元数据被全量缓存、持久化“存档”。

TFS：在TFS中，将大量的小文件(实际用户文件)合并成为一个大文件，这个大文件称为块(Block)。TFS以Block的方式组织文件的存储。DataServer进程会给Block中的每个文件分配一个ID(File ID，该ID在每个Block中唯一)，并将每个文件在Block中的信息存放在和Block对应的Index文件中。这个Index文件一般都会全部load在内存。

海量的小文件都不会直接放在文件系统上，而是合并成super block，维护super block中各entry的元数据和索引信息，并全部载入内存中。

### 分布式协调调度、应用元数据策略

Haystack：系统接收到读写请求时，依靠Directory分析应用元数据，再结合一定策略（负载均衡、容量、运维、只读、可写等），决定请求被发送到哪台store机器上，并向store提供足够的存储或检索信息。Directory负责了整体分布式环境的协调调度、应用元数据管理职能，并基于此帮助实现了系统的可扩展性、容错性。

TFS：NameServer主要功能是: 管理维护Block和!DataServer相关信息,包括!DataServer加入，退出, 心跳信息, block和!DataServer的对应关系建立，解除。正常情况下，一个块会在!DataServer上存在， 主!NameServer负责Block的创建，删除，复制，均衡，整理。

Haystack中的Directory和TFS的NameServer职责类似，负责分布式协调调度和应用元数据分配管理。

### 扩展性

Haystack和TFS都是基于分布式协调调度和元数据分配管理实现了扩展方案。

**传统的方案**

现在有海量的数据，比如data[key:value]，有100台机器，通过一种策略让这些数据能负载均衡的发给各台机器。策略可以是这样，int index=Math.abs(key.hashCode)%100，这就得到了一个唯一的、确定的、[0,99]的序号，按此序号发给对应的某台机器，最终能达到负载均衡的效果。此方案的粗暴显而易见，当我们新增机器后（比如100变成130），大部分老数据的key执行此策略后得到的index会发生变化，这也就意味着对它们的检索都会发往错误的机器，找不到数据。

**稍微改进的方案**

现在有海量的数据，比如data[key:value]，虽然只有100台机器，但是我们假想自己有10000台机器。对于这些假想的机器，每个机器给一个编号，称为虚拟节点（vnode），然后想办法让vnode与真实机器建立多对一的映射，这个办法可以是某种策略，比如故技重施对vnodeId%100得到[0,99]的机器序号，或者在数据库中建几张表维护一下这个多对一的映射关系。

在路由时，先按老办法得到vnodeId，再执行一次映射，找到真实机器。这个方案还需要一个架构假设：我的系统规模在5年内都不需要上涨到一万台机器，因此10000这个数字“永远”不会变，这就保证了一个key永远对应某个vnodeId，不会发生改变。然后在扩容时，我们改变的是vnode与真实机器的映射关系，但是此映射关系一改，也会不可避免的导致数据命中失败，因为必然会产生这样的现象：某个vnodeId（v1）原先是对应机器A的，现在变成了机器B。但是相比之前的方案，现在已经好很多了，我们可以通过运维手段先阻塞住对v1的读写请求，然后执行数据迁移（以已知的vnode为粒度，而不是千千万万个未知的data，这种迁移操作还是可以接受的），迁移完毕后新机器开始接收请求。做的更好一点，可以不阻塞请求，想办法做点容错处理和写同步之类的，可以在线无痛的完成迁移。

上面两个方案都可以用下面的这个公式表示：

int machine\_id = hash\_function(data.key, node\_number)

machine\_id是指最终路由到的机器，hash\_funciton是我们的粗略函数，data.key就是数据的ID，node\_number是指系统中机器的数量，在第一个方案里面就是100，第二个方案里就是10000。因为node\_number是一个变量，所以只要这个变量存在，hash\_function的结果就可能会改变。

如果公式可以变成下面这个，就能够避免问题：

Map<data.key, machine\_id> map = new HashMap()

int machine\_id = map.get(data.key)

这里只是用代码描述，map应该是一个更加复杂的结构，无论这个map的结构是什么样的，其效果都是，通过data.key能够获得machine\_id，这个machine\_id不是动态计算的，所以data.key与machine\_id之间的映射不会改变。当增加机器的时候，此map中已有的值不会受到影响，不会发生老数据命中失败的情况。在分布式协调调读的保证下，新增的数据会倾向于写入新的机器，整个集群的负载逐渐均衡。

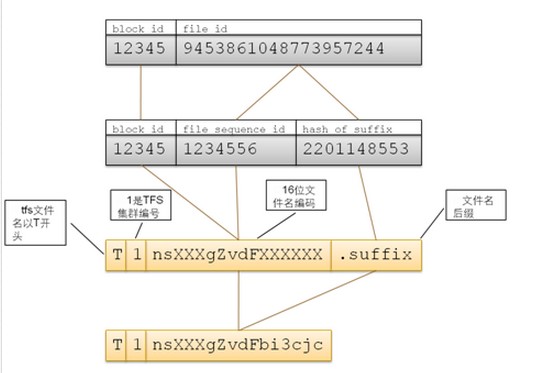
在Haystack和TFS中都做到了上面描述的map，但是两者实现的方式却不同。

Haystack：浏览器向web server提交图片原始URL的时候，假设原始URL中只有photo id, 那么Directory中维护了photo id到逻辑卷的映射关系？（之所以有疑问是论文中没有说）

photo id到逻辑卷的映射就是map，photo id（data.key）能的到逻辑卷ID（此值是upload时就明确分配的，不会改变），再间接从“逻辑卷到物理卷映射”中就能get到目标store机器；无论是新增逻辑卷还是新增物理卷，“图片ID到逻辑卷的映射”中的已有值都可以不受影响。这些都符合map的行为定义。

如果在是在Directory中维护了photo id到逻辑卷的映射关系，那么这个数据量就不能忽略不计。稳重提到memcache，也许这就是解决方案。

TFS：TFS的文件名由块号和文件号通过某种对应关系组成，最大长度为18字节。文件名固定以T开始，第二字节为该集群的编号(可以在配置项中指定，取值范围 1-9)。余下的字节由Block ID和File ID通过一定的编码方式得到。文件名由客户端程序进行编码和解码，它映射方式如下图：



在DataServer，根据TFS文件名解析出Block ID和block中的File ID，DataServer会根据本地的记录信息得到File ID在block中的偏移量，从而读取到正确的文件内容。

TFS的这个方案称为“结构化ID”、“聚合ID”，或者“命名规则大于配置”，既避免了保存大量的映射数据，又避免了增加系统间的耦合。不过这个方案对图片ID有所约束，也不支持自定义的图片名称，针对这个问题，TFS在新版本中解决的自定义文件名问题，“metaserver是我们在2.0版本引进的一个服务。用来存储一些元数据信息，这样原本不支持自定义文件名的 TFS 就可以在 metaserver 的帮助下，支持自定义文件名了”。此metaserver的作用无疑就和Haystack Directory中部分应用元数据相关的职责类似了。个人认为可以两者结合双管齐下，毕竟自定义文件名这种需求应该不是主流。

其实在Haystack中全量保存这部分应用元数据其实还是有很多好处的，最典型的就是顺带保存的cookie，有效的帮助Haystack不受伪造URL攻击的困扰，这个问题不知道TFS是如何解决的（大量的文件检索异常势必会影响系统性能）。

TFS还有一个特性：在集群负载比较轻的时候，NameServer会对DataServer上的Block进行均衡，使所有DataServer的容量尽早达到均衡。进行均衡计划时，首先计算每台机器应拥有的blocks平均数量，然后将机器划分为两堆，一堆是超过平均数量的，作为移动源；一类是低于平均数量的，作为移动目的。

均衡计划的职责是在负载较低的时候（深夜），按计划执行Block数据的迁移，促进整体负载更加均衡。根据译者的理解，此计划会改变公式中的map，因为根据文件名拿到的BlockId对应的机器可能发生变化，这也是它为何要在深夜负载较低时按计划缜密执行的原因。其效果是避免了因为运维操作等原因导致的数据分布不均。

### 容错性

Haystack：它的容错依靠的是将一个逻辑卷对应多个物理卷，向一个逻辑卷的写操作会被翻译成对多个物理卷的写，实现了冗余备份。store机器故障时，Directory修改应用元数据（涉及到逻辑卷映射中删除物理卷），指导路由过程将请求发送到后备节点。

TFS：TFS可以配置主辅集群，一般主辅集群会存放在两个不同的机房。主集群提供所有功能，辅集群只提供读。主集群会把所有操作重放到辅集群。这样既提供了负载均衡，又可以在主集群机房出现异常的情况不会中断服务或者丢失数据。每一个Block会在TFS中存在多份，一般为3份，并且分布在不同网段的不同DataServer上。客户端向master DataServer开始数据写入操作。master server将数据传输为其他的DataServer节点，只有当所有DataServer节点写入均成功时，master server才会向nameserver和客户端返回操作成功的信息。

**冗余机制**

Haystack将冗余写的交给“客户端”（这个客户端指的是Haystack中的web server），web server发起多次写请求把数据写入不同的store机器上；而TFS是依靠master-slave机制，由master向slave复制。

**机房容错**

在机房容错方面，TFS依然采取的是master-slave机制，集群也分主辅，两个集群在不同的机房，主集群负责在辅集群上重放操作。Haystack论文中没有详细说，只是提到“Haystack复制每张图片到地理隔离的多个地点”。

Haystack中可能没有像TFS那样严格的master slave之分，偏向于对等结构，各个store机器是对等的节点，没有谁负责给谁复制数据。单独考虑Haystack中store机器容错机制：**如果单台store故障**，Directory在应用元数据的相关逻辑卷映射中删除此台机器的物理卷（此过程简称为“调整逻辑物理映射”），其他“对等”的物理卷能继续服务，没有问题；**一整个机房故障**，Directory处理过程和单台故障相同，只是会对此机房中每台机器都执行一遍“调整逻辑物理映射”，由于逻辑卷到物理卷的映射是在Directory中明确维护的，所以只要在维护和管理过程中确保一个逻辑卷下不同的物理卷分布在不同的机房，哪怕在映射中删除一整个机房所有机器对应的物理卷，各个逻辑卷下依然持有到其他机房可用物理卷的映射，依然有对等Store的物理卷做后备，没有问题。

### 删除和压缩

Haystack：使用软删除，对删除的图片设置flag，压缩回收来支持delete操作。

TFS：压缩的过程其实和复制有点像，只是说不需要将删除的文件数据以及index数据复制到新创建的压缩块中。要判断某个文件是否被删除，还需要拿index文件的offset去fileinfo里面取删除标记，如果标记不是删除的，那么就可以进行write\_raw\_data的操作，否则则滤过