# 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。

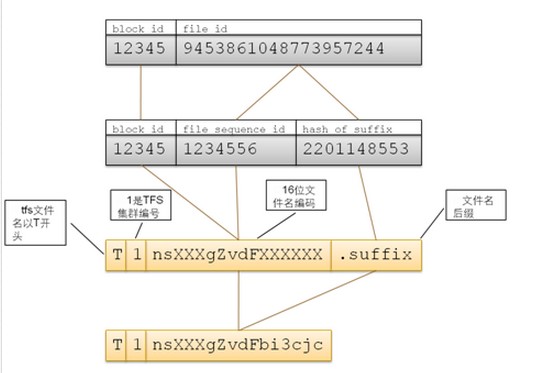
在Haystack和TFS中都做到了上面描述的map，但是两者实现的方式却不同。

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