

ImportNew

```
while(true) {
    I.Love(这些技术类微信公众号);
}
```

- [首页](#)
- [所有文章](#)
- [资讯](#)
- [Web](#)
- [架构](#)
- [基础技术](#)
- [书籍](#)
- [教程](#)
- [我要投稿](#)
- [更多频道 »](#)

- 导航条 - 

经典论文翻译导读之《Finding a needle in Haystack: Facebook's photo storage》

2013/03/07 | 分类: [技术架构](#) | 8 条评论 | 标签: [Facebook](#), [Haystack](#), [TFS](#), [分布式文件存储](#)

分享到: 86 本文作者: [ImportNew - 储晓颖](#) 未经许可, 禁止转载!

【译者预读】面对海量小文件的存储和检索, Google发表了GFS, 淘宝开源了TFS, 而Facebook又是如何应对千亿级别的图片存储、每秒百万级别的图片查询? Facebook与同样提供了海量图片服务的淘宝, 解决方案有何异同? 本篇文章, 为您揭晓。

本篇论文的原文可谓通俗易懂、行云流水、结构清晰、图文并茂.....正如作者所说的——“替换Facebook的图片存储系统就像高速公路上给汽车换轮子, 我们无法去追求完美的设计.....我们花费了很多的注意力来保持它的简单”, 本篇论文也是一样, 没有牵扯空洞的庞大架构、也没有晦涩零散的陈述, 有的是对痛点的反思, 对目标的分解, 条理清晰, 按部就班。既描述了宏观的整体流程, 又推导了细节难点的技术突破过程。以至于译者都不需要在文中插入过多备注和解读了^_^. 不过在文章末尾, 译者以淘宝的解决方案作为对比, 阐述了文章中的一些精髓的突破点, 以供读者参考。

摘要

本篇论文描述了Haystack, 一个为Facebook的照片应用而专门优化定制的对象存储系统。Facebook当前存储了超过260 billion的图片, 相当于20PB的数据。用户每个星期还会上传1 billion的新照片 (60TB), Facebook在峰值时需提供每秒查询超过1 million图片的能力。相比我们以前的方案(基于NAS和NFS), Haystack提供了一个成本更低的、性能更高的解决方案。我们观察到一个非常关键的问题: 传统的设计因为元数据查询而导致了过多的磁盘操作。我们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。

1 介绍

分享照片是Facebook最受欢迎的功能之一。迄今为止, 用户已经上传了超过65 billion的图片, 使得Facebook成为世界上最大的图片分享网站。对每个上传的照片, Facebook生成和存储4种不同大小的图片(比如在某些场景下只需展示缩略图), 这就产生了超过260 billion张图片、超过20PB的数据。用户每个星期还在上传1 billion的新照片 (60TB), Facebook峰值时需要提供每秒查询1 million张图片的能力。这些数字未来还会不断增长, 图片存储对Facebook的基础设施提出了一个巨大的挑战。

这篇论文介绍了Haystack的设计和实现, 它已作为Facebook的图片存储系统投入生产环境24个月了。Haystack是一个为Facebook上分享照片而设计的对象存储技术, 在这个应用场景中, 每个数据只会写入一次、读操作频繁、从不修改、很少删除。在Facebook遭遇的负荷下, 传统的文件系统性能很差, 优化定制出Haystack是大势所趋。

根据我们的经验，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。这是我们从之前（NAS+NFS）方案中总结的血的教训。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（CDN，比如Akamai）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。

了解传统途径的缺点后，我们设计了Haystack来达到4个主要目标：

- 高吞吐量和低延迟。我们的图片存储系统必须跟得上每天数百万用户查询请求。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。
- 容错。在大规模系统中，故障每天都会发生。尽管服务器崩溃和硬盘故障是不可避免的，也绝不可以给用户返回一个error，哪怕整个数据中心都停电，哪怕一个跨国网络断开。所以，Haystack复制每张图片到地理隔离的多个地点，一台机器倒下了，多台机器会替补上来。
- 高性价比。Haystack比我们之前（NAS+NFS）方案性能更好，而且更省钱。我们按两个维度来衡量：每TB可用存储的花费、每TB可用存储的读取速度。相对NAS设备，Haystack每个可用TB省了28%的成本，每秒支撑了超过4倍的读请求。
- 简单。替换Facebook的图片存储系统就像高速公路上给汽车换轮子，我们无法去追求完美的设计，这会导致实现和维护都非常耗时耗力。Haystack是一个新系统，缺乏多年的生产环境级别的测试。我们花费了很多的注意力来保持它的简单，所以构建和部署一个可工作的Haystack只花了几个月而不是好几年。

本篇文章3个主要的贡献是：

- Haystack，一个为高效存储和检索billion级别图片而优化定制的对象存储系统。
- 构建和扩展一个低成本、高可靠、高可用图片存储系统中的经验教训。
- 访问Facebook照片分享应用的请求的特征描述

文章剩余部分结构如下。章节2阐述了背景、突出了之前架构遇到的挑战。章节3描述了Haystack的设计和实现。章节4描述了各种图片读写场景下的系统负载特征，通过实验数据证明Haystack达到了设计目标。章节5是对比和相关工作，以及章节6的总结。

2 背景 & 我的前任

在本章节，我们将描述Haystack之前的架构，突出其主要的经验教训。由于文章大小限制，一些细节就不细述了。

2.1 背景

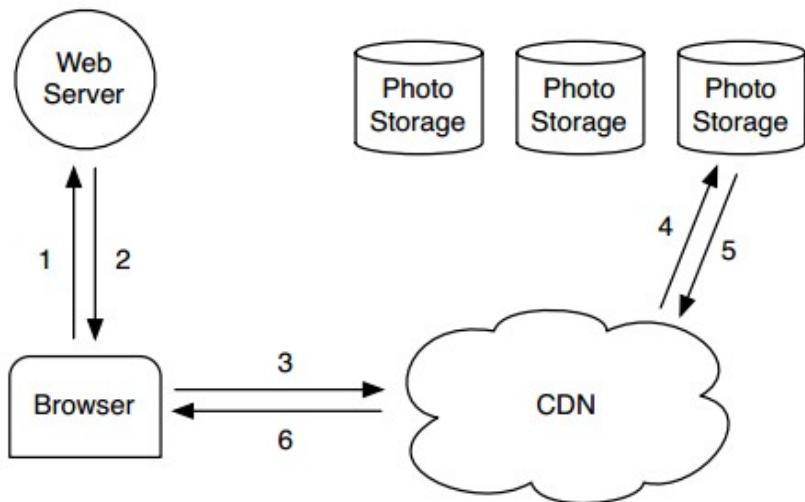


Figure 1: Typical Design

我们先来看一个概览图，它描述了通常的设计方案，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。图1描述了从用户访问包含某个图片的页面开始，直到她最终从磁盘的特定位置下载此图片结束的全过程。访问一个页面时，用户的浏览器首先发送HTTP请求到一个web服务器，它负责生成markup以供浏览器渲染。对每张图片，web服务器为其构造一个URL，引导浏览器在此位置下载图片数据。对于热门站点，这个URL通常指向一个CDN。如果CDN缓存了此图片，那么它会立刻将数据回复给浏览器。否则，CDN检查URL，URL中需要嵌入足够的信息以供CDN从本站点的存储系统中检索图片。拿到图片后，CDN更新它的缓存数据、将图片发送回用户的浏览器。

2.2 基于NFS的设计

在我们最初的设计中，我们使用了一个基于NFS的方案。我们吸取的主要教训是，对于一个热门的社交网络站点，只有CDN不足以提供一个实用的解决方案。对于热门图片，CDN确实很高效——比如个人信息图片和最近上传的照片——但是一个像Facebook的社交网络站点，会产生大量的对不热门（较老）内容的请求，我们称之为long tail（长尾理论中的名词）。long tail的请求也占据了很大流量，它们都需要访问更下游的图片存储主机，因为这些请求在CDN缓存里基本上都会命中失败。缓存所有的图片是可以解决此问题，但这么做代价太大，需要极大容量的缓存。

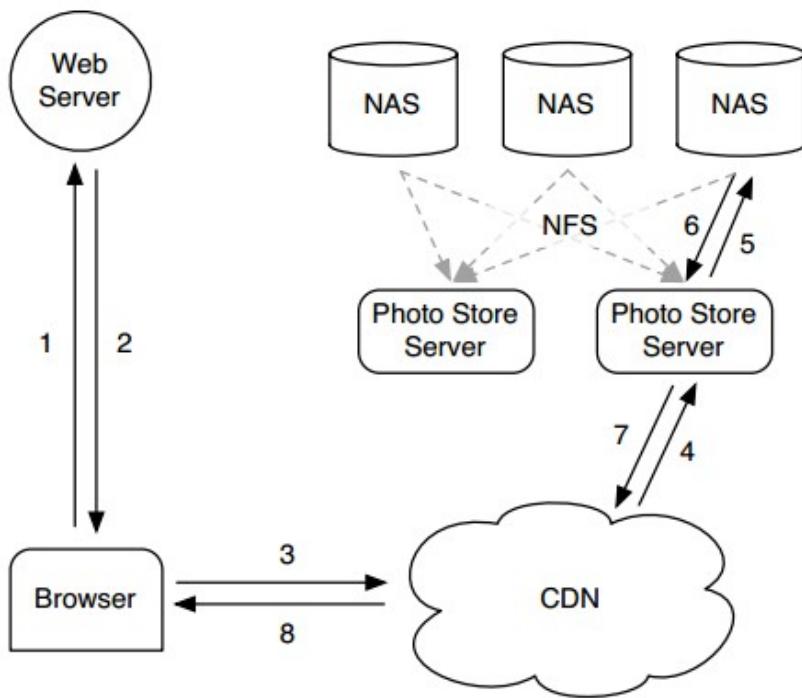


Figure 2: NFS-based Design

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。图2展示了这个架构。Photo Store Server解析URL得出卷和完整的文件路径，在NFS上读取数据，然后返回结果到CDN。

我们最初在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，哪怕只是读单个图片。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。

为了继续减少磁盘操作，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）。

2.3 讨论

我们很难提出一个指导方针关于何时应该构建一个自定义的存储系统。下面是我们在最终决定搭建Haystack之前的一些思考，希望能给大家提供参考。

面对基于NFS设计的瓶颈，我们探讨了是否可以构建一个类似GFS的系统。而我们大部分用户数据都存储在Mysql数据库，文件存储主要用于开发工作、日志数据以及图片。NAS设备其实为这些场景提供了性价比很好的方案。此外，我们补充了hadoop以供海量日志数据处理。面对图片服务的long tail问题，Mysql、NAS、Hadoop都不太合适。

我们面临的困境可简称为“已存在存储系统缺乏合适的RAM-to-disk比率”。然而，没有什么比率是绝对正确的。系统需要足够的内存才能缓存所有的文件系统元数据。在我们基于NAS的方案中，一个图片对应到一个文件，每个文件需要至少一个inode，这已经占了几百byte。提供足够的内存太昂贵。所以我们决定构建一个定制存储系统，减少每个图片的元数据总量，以便能有足够的内存。相对购买更多的NAS设备，这是更加可行的、性价比更好的方案。

3 设计和实现

Facebook使用CDN来支撑热门图片查询，结合Haystack则解决了它的long tail问题。如果web站点在查询静态内容时遇到I/O瓶颈，传统方案就是使用CDN，它为下游的存储系统挡住了绝大部分的查询请求。在Facebook，为了传统的、廉价的底层存储不受I/O摆布，CDN往往需要缓存难以置信的海量静态内容。

上面已经论述过，在不久的将来，CDN也不能完全的解决我们的问题，所以我们设计了Haystack来解决这个严重瓶颈：磁盘操作。我们接受long tail请求必然导致磁盘操作的现实，但是会尽量减少除了访问真实图片数据之外的其他操作。Haystack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。

每个图片存储为一个文件将会导致元数据太多，难以被全部缓存。Haystack的对策是：将多个图片存储在单个文件中，控制文件个数，维护大型文件，我们将论述此方案是非常有效的。另外，我们强调了它设计的简洁性，以促进快速的实现和部署。我们将以此核心技术展开，结合它周边的所有架构组件，描述Haystack是如何实现了一个高可靠、高可用的存储系统。在下面对Haystack的介绍中，需要区分两种元数据，不要混淆。一种是应用元数据，它是用来为浏览器构造检索图片所需的URL；另一种是文件系统元数据，用于在磁盘上检索文件。

3.1 概览

Haystack架构包含3个核心组件：Haystack Store、Haystack Directory和Haystack Cache（简单起见我们下面就不带Haystack前缀了）。Store是持久化存储系统，并负责管理图片的文件系统元数据。Store将数据存储在物理的卷上。比如，在一台机器上提供100个物理卷，每个提供100GB的存储容量，整台机器则可以支撑10TB的存储。更进一步，不同机器上的多个物理卷将对应一个逻辑卷。Haystack将一个图片存储到一个逻辑卷时，图片被写入到所有对应的物理卷，这个冗余可避免由于硬盘故障，磁盘控制器bug等导致的数据丢失。Directory维护了逻辑到物理卷的映射以及其他应用元数据，比如某个图片寄存在哪个逻辑卷、某个逻辑卷的空闲空间等。Cache的功能类似我们系统内部的CDN，它帮Store挡住热门图片的请求（可以缓存的就绝不交给下游的持久化存储）。在独立设计Haystack时，我们要设想它处于一个没有CDN的大环境中，即使有CDN也要预防其节点故障导致大量请求直接进入存储系统，所以Cache是十分必要的。

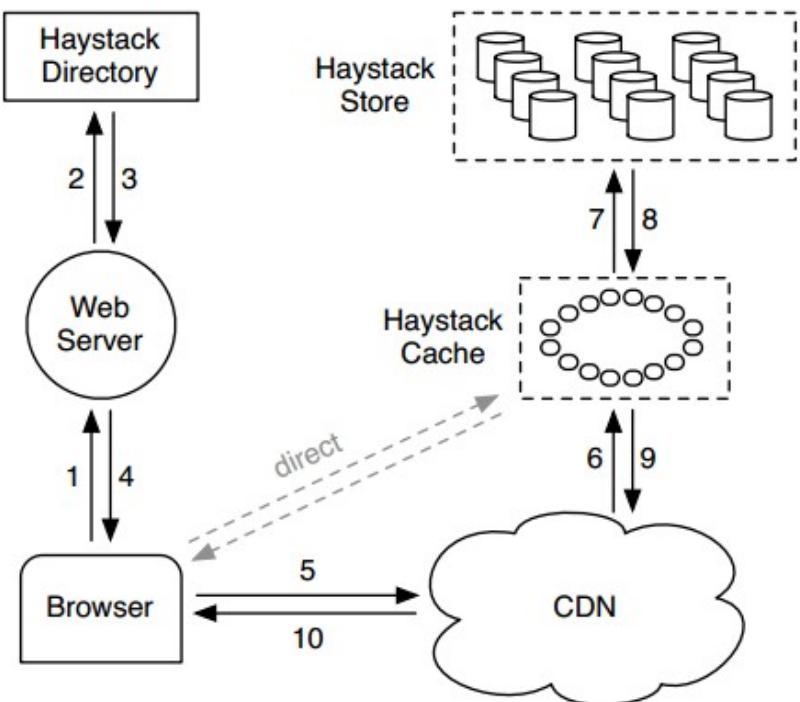


Figure 3: Serving a photo

图3说明了Store、Directory、Cache是如何协作的，以及如何与外部的浏览器、web服务器、CDN和存储系统交互。在Haystack架构中，浏览器会被引导至CDN或者Cache上。需要注意的是Cache本质上也是一个CDN，为了避免困惑，我们使用“CDN”表示外部的系统、使用“Cache”表示我们内部的系统。有一个内部的缓存设施能减少对外部CDN的依赖。

当用户访问一个页面，web服务器使用Directory为每个图片来构建一个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Cache)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://<CDN>/<Cache>/<Machine id>/<Logical volume, Photo>`

第一个部分<CDN>指明了从哪个CDN查询此图片。到CDN后它使用最后部分的URL（逻辑卷和图片ID）即可查找缓存的图片。如果CDN未命中缓存，它从URL中删除<CDN>相关信息，然后访问Cache。Cache的查找过程与之类似，如果还没命中，则去掉<Cache>相关信息，请求被发至指定的Store机器(<Machine id>)。如果请求不经过CDN直接发至Cache，其过程与上述类似，只是少了CDN这个环节。

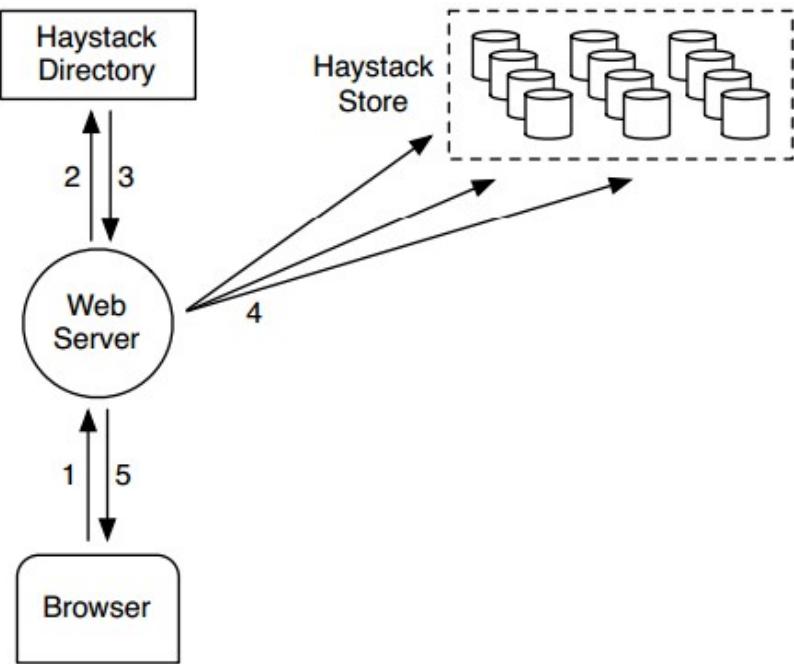


Figure 4: Uploading a photo

图4说明了在Haystack中的上传流程。用户上传一个图片时，她首先发送数据到web服务器。web服务器随后从Directory中请求一个可写逻辑卷。最后，web服务器为图片分配一个唯一的ID，然后将其上传至逻辑卷对应的每个物理卷。

3.2 Haystack Directory

Directory提供4个主要功能。首先，它提供一个从逻辑卷到物理卷的映射。web服务器上传图片和构建图片URL时都需要使用这个映射。第二，Directory在分配写请求到逻辑卷、分配读请求到物理卷时需保证负载均衡。第三，Directory决定一个图片请求应该被发至CDN还是Cache，这个功能可以让我们动态调整是否依赖CDN。第四，Directory指明那些逻辑卷是只读的（只读限制可能是源自运维原因、或者达到存储容量上限；为了运维方便，我们以机器粒度来标记卷的只读）。

当我们增加新机器以增大Store的容量时，那些新机器是可写的；仅仅可写的机器会收到upload请求。随时流逝这些机器的可用容量会不断减少。当一个机器达到容量上限，我们标记它为只读，在下一个子章节我们将讨论如何这个特性如何影响Cache和Store。

Directory将应用元数据存储在一个冗余复制的数据库，通过一个PHP接口访问，也可以换成memcache以减少延迟。当一个Store机器故障、数据丢失时，Directory在应用元数据中删除对应的项，新Store机器上线后则接替此项。

【译者YY】 3.2章节是整篇文章中唯一一处译者认为没有解释清楚的环节。结合3.1章节中的URL结构解析部分，读者可以发现Directory需要拿到图片的“原始URL”（页面html中link的URL），再结合应用元数据，就可以构造出“引导URL”以供下游使用。从3.2中我们知道Directory必然保存了逻辑卷到物理卷的映射，仅用此映射+原始URL足够发掘其他应用元数据吗？原始URL中到底包含了什么信息（论文中没看到介绍）？我们可以做个假设，假如原始URL中仅仅包含图片ID，那Directory如何得知它对应哪个逻辑卷（必须先完成这一步映射，才能继续挖掘更多应用元数据）？Directory是否在upload阶段将图片ID与逻辑卷的映射也保存了？如果是，那这个映射的数据量不能忽略不计，论文也不该一笔带过。

从原文一些细枝末节的描述中，译者认为Directory确实保存了很多与图片ID相关的元数据（存储在哪个逻辑卷、cookie等）。但整篇论文译者也没找到对策，总感觉这样性价比太低，不符合Haystack的作风。对于这个疑惑，文章末尾扩展阅读部分将尝试解答。读者先认为其具备此能力吧。

3.3 Haystack Cache

Cache会从CDN或者直接从用户浏览器接收到图片查询请求。Cache的实现可理解为一个分布式Hash Table，使用图片ID作为key来定位缓存的数据。如果Cache未命中，Cache则根据URL从指定Store机器上获取图片，视情况回复给CDN或者用户浏览器。

我们现在强调一下Cache的一个重要行为概念。只有当符合两种条件之一时它才会缓存图片：(a)请求直接来自用户浏览器而不是CDN；(b)图片获取自一个可写的Store机器。第一个条件的理由是一个请求如果在CDN中没命中（非热门图片），那在我们内部缓存也不太需要命中（即使此图片开始逐渐活跃，那也能在CDN中命中缓存，这里无需多此一举；直接的浏览器请求说明是不经过CDN的，那就需要Cache代为CDN，为其缓存）。第二个条件的理由是间接的，有点经验论，主要是为了保护可写Store机器；原因挺有意思，大部分图片在上传之后很快会被频繁访问（比如某个美女新上传了一张自拍），而且文件系统在只有读或者只有写的情况下执行的更好，不太喜欢同时并发读写（章节4.1）。如果没有Cache，可写Store机器往往会遭遇频繁的读请求。因此，我们甚至会主动的推送最近上传的图片到Cache。

3.4 Haystack Store

Store机器的接口设计的很简约。读操作只需提供一些很明确的元数据信息，包括图片ID、哪个逻辑卷、哪台物理Store机器等。机器如果找到图片则返回其真实数据，否则返回错误信息。

每个Store机器管理多个物理卷。每个物理卷存有百万张图片。读者可以将一个物理卷想象为一个非常大的文件（100GB），保存为’/hay/haystack<logical volume id>’。Store机器仅需要逻辑卷ID和文件offset就能非常快的访问一个图片。这是Haystack设计的主旨：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的文件“句柄”，卷的数量不多，数据量不大）缓存在内存中。同时，图片ID到文件系统元数据（文件、偏移量、大小等）的映射（后文简称为“[内存中映射](#)”）是检索图片的重要条件，也会全部缓存在内存中。

现在我们描述一下物理卷和内存中映射的结构。一个物理卷可以理解为一个大型文件，其中包含一系列的needle。每个needle就是一张图片。图5说明了卷文件和每个needle的格式。Table1描述了needle中的字段。

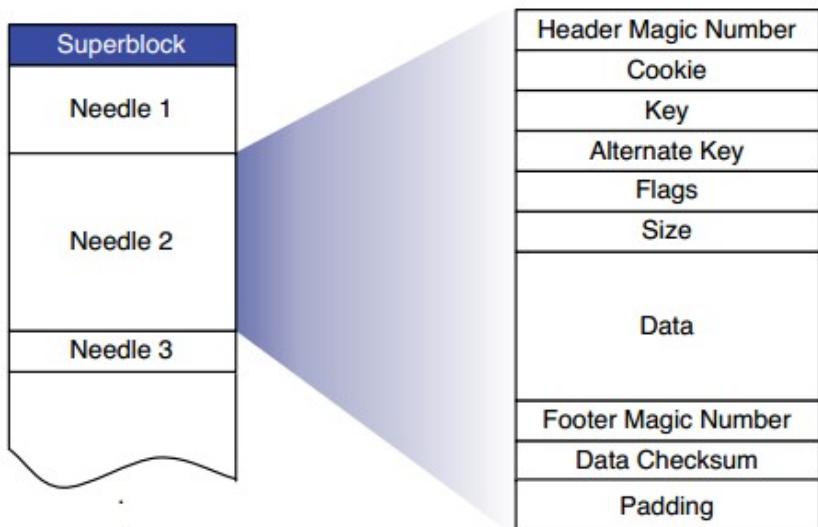


Figure 5: Layout of Haystack Store file

Field	Explanation
Header	Magic number used for recovery
Cookie	Random number to mitigate brute force lookups
Key	64-bit photo id
Alternate key	32-bit supplemental id
Flags	Signifies deleted status
Size	Data size
Data	The actual photo data
Footer	Magic number for recovery
Data Checksum	Used to check integrity
Padding	Total needle size is aligned to 8 bytes

Table 1: Explanation of fields in a needle

为了快速的检索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(needle.key+needle.alternate_key)的组合，映射的Value就是needle的flag、size、卷offset（都以byte为单位）。如果Store机器崩溃、重启，它可以直接分析卷文件来重新构建这个映射（构建完成之前不处理请求）。下面我们介绍Store机器如何响应读写和删除请求（Store仅支持这些操作）。

【译者注】从Table1我们看到needle.key就是图片ID，为何仅用图片ID做内存中映射的Key还不够，还需要一个alternate_key？这是因为一张照片会有4份副本，它们的图片ID相同，只是类型不同（比如大图、小图、缩略图等），于是将图片ID作为needle.key，将类型作为needle.alternate_key。根据译者的理解，内存中映射不是一个简单的HashMap结构，而是类似一个两层的嵌套HashMap，`Map<long/*needle.key*/, Map<int/*alternate_key*/, Object> >`。这样做可以让4个副本共用同一个needle.key，避免为重复的内容浪费内存空间。

3.4.1 读取图片

Cache机器向Store机器请求一个图片时，它需要提供逻辑卷id、key、alternate key，和cookie。cookie是个数字，嵌在URL中。当一张新图片被上传，Directory为其随机分配一个cookie值，并作为应用元数据之一存储在Directory。它就相当于一张图片的“私人密码”，此密码可以保证所有发往Cache或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机器发来的图片查询请求，它会利用内存中映射快速的查找相关的元数据。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（needle的size可以提前计算出来），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。

3.4.2 写入图片

上传一个图片到Haystack时，web服务器向Directory咨询得到一个可写逻辑卷及其对应的多台Store机器，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。

3.4.3 图片删除

在删除图片时，Store机器将内存中映射和卷文件中相应的flag同步的设置为已删除（软删除机制，此刻不会

删除needle的磁盘数据）。当接收到已删除图片的查询请求，Store会检查内存中flag并返回错误信息。值得注意的是，已删除needle依然占用的空间是个问题，我们稍后将讨论如何通过压缩技术来回收已删除needle的空间。

3.4.4 索引文件

Store机器使用一个重要的优化——索引文件——来帮助重启初始化。尽管理论上一个机器能通过读取所有的物理卷来重新构建它的内存中映射，但大量数据（TB级别）需要从磁盘读取，非常耗时。索引文件允许Store机器快速的构建内存中映射，减少重启时间。

Store机器为每个卷维护一个索引文件。索引文件可以想象为内存中映射的一个“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。图6描述了索引文件的布局，Table2解释了记录中的不同的字段。

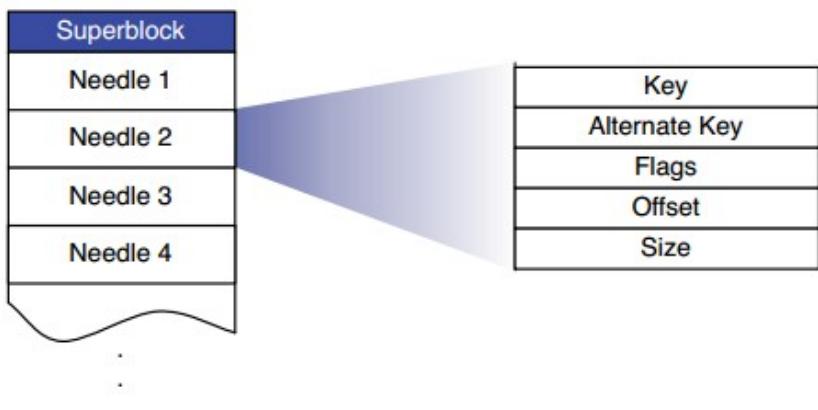


Figure 6: Layout of Haystack Index file

Field	Explanation
Key	64-bit key
Alternate key	32-bit alternate key
Flags	Currently unused
Offset	Needle offset in the Haystack Store
Size	Needle data size

Table 2: Explanation of fields in index file.

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。

3.4.5 文件系统

Haystack可以理解为基于通用的类Unix文件系统搭建的对象存储，但是某些特殊文件系统能更好的适应Haystack。比如，Store机器的文件系统应该不需要太多内存就能在一个大型文件上快速的执行随机seek。当前我们所有的Store机器都在使用的文件系统是XFS，一个基于“范围(extent)”的文件系统。XFS有两个优势：首先，XFS中邻近的大型文件的“blockmap”很小，可放心使用内存存储；第二，XFS提供高效文件预分配，减轻磁盘碎片等问题。

使用XFS，Haystack可以在读取一张图片时完全避免检索文件系统元数据导致的磁盘操作。但是这并不意味着Haystack能保证读取单张图片绝对只需要一个磁盘操作。在一些极端情况下会发生额外的磁盘操作，比如当图片数据跨越XFS的“范围(extent)”或者RAID边界时。不过Haystack会预分配1GB的“范围(extent)”、设置RAID stripe大小为256KB，所以实际上我们很少遭遇这些极端场景。

3.5 故障恢复

对于运行在普通硬件上的大规模系统，容忍各种类型的故障是必须的，包括硬盘驱动故障、RAID控制器错误、主板错误等，Haystack也不例外。我们的对策由两个部分组成——一个为侦测、一个为修复。

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。

3.6 优化

Haystack的成功还归功于几个非常重要的细节优化。

3.6.1 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。

3.6.2 节省更多内存

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。

3.6.3 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。

章节4、5、6是实验和总结等内容，这里不再赘述了。

【扩展阅读】

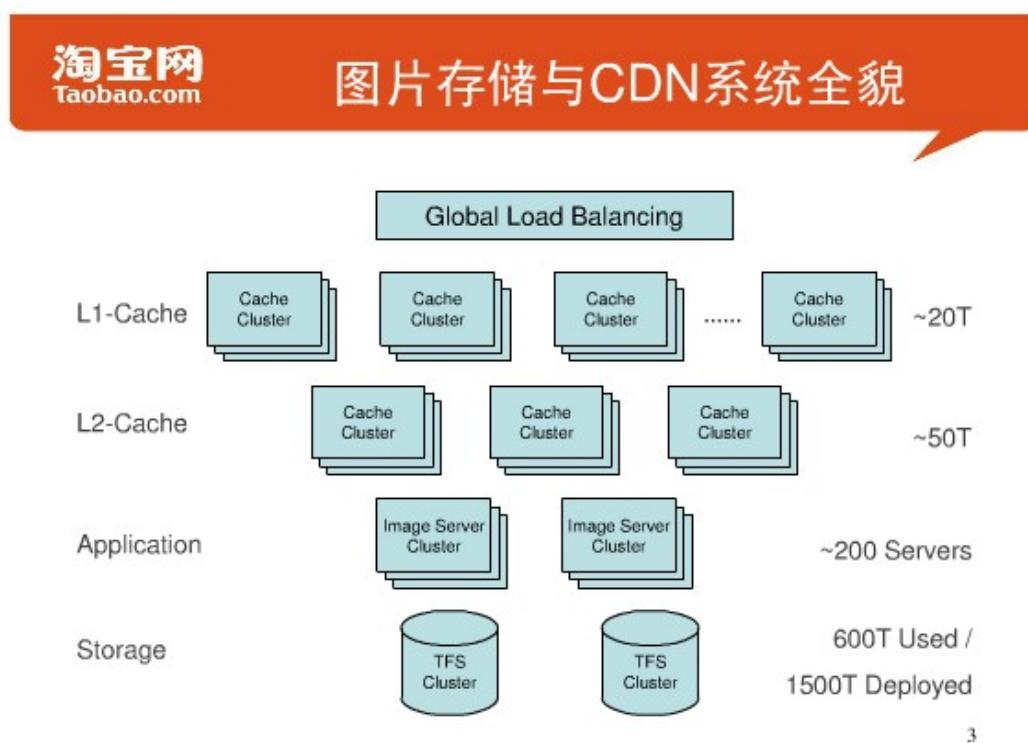
提到CDN和分布式文件存储就不得不提到淘宝，它的商品图片不会少于Facebook的个人照片。其著名的CDN+TFS的解决方案由于为公司节省了巨额的预算开支而获得创新大奖，团队成员也得到不菲的奖金（羡慕嫉妒恨）。淘宝的CDN技术做了非常多的技术创新和突破，不过并非本文范畴，接下来的讨论主要是针对Haystack与TFS在存储、检索环节的对比，并尝试提取出此类场景常见的技术难点。（译者对TFS的理解仅限于介绍文档，若有错误望读者矫正）

淘宝CDN、TFS的介绍请移步

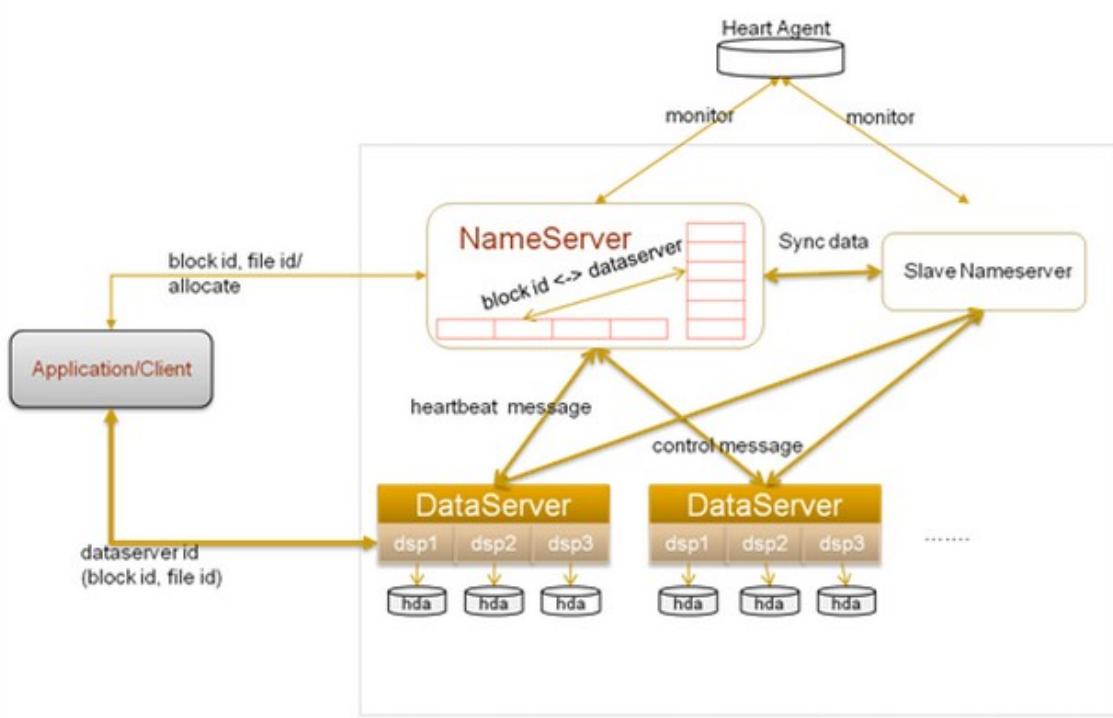
<http://www.infoq.com/cn/presentations/zws-taobao-image-store-cdn>

<http://tfs.taobao.org/index.html>

注意：下文中很多术语（比如应用元数据、Store、文件系统元数据等，都是基于本篇论文的上下文，请勿混淆）



上图是整个CDN+TFS解决方案的全貌，对应本文就是图3。CDN在前三层上实现了各种创新和技术突破，不过并非本文焦点，这里主要针对第四层Storage（淘宝的分布式文件系统TFS），对比Haystack，看其是否也解决了long tail问题。下面是TFS的架构概览：



从粗粒度的宏观视角来看，TFS与Haystack的最大区别就是：TFS只care存储层面，它没有Haystack 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。

到底是定制一整套优化的解决方案，还是使用通用分布式文件存储平台强强联手？Facebook的工程师也曾纠结过（章节2.3），这个没有标准答案，各有所长，视情况去选择最合适的方式吧。

下面我们以本文中关注的一些难点来对比一下双方的实现：

1 存储机器上的文件结构、文件系统元数据对策

Haystack的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”。

在TFS中（后文为清晰起见，引用TFS文献的内容都用淘宝最爱的橙色展示）：

“.....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.....”

看来面对可怜的操作系统，大家都不忍心把海量的小文件直接放到它的文件系统上，合并成super block，维护super block中各entry的元数据和索引信息（并全量缓存），才是王道。这里TFS的Block应该对应到Haystack中的一个物理卷。

2 分布式协调调度、应用元数据策略

Haystack在接收到读写请求时，依靠Directory分析应用元数据，再结合一定策略（如负载均衡、容量、运维、只读、可写等），决定请求被发送到哪台Store机器，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。

在TFS中：

“.....!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.....”

“.....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.....”

TFS中与Directory对应的就是NameServer了，职责大同小异，就是分布式协调调度和应用元数据分配管理，并基于此实现系统的平滑扩容、故障容忍。下面专门讨论一下这两个重要特性。

3 扩展性

Haystack和TFS都基于（分布式协调调度+元数据分配管理）实现了非常优雅的可扩展方案。我们先回顾一下传统扩展性方案中的那些简单粗暴的方法。

最简单最粗暴的场景：

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。

稍微改进的场景是：

现在有海量的数据，比如data [key : value]，我假想自己是高富帅，有一万台机器，同样按照上述的策略进行路由。但是我只有100台机器，这一万台是假想的，怎么办？先给它们一个称号，叫虚拟节点（简称vnode，vnode的序号简称为vnodeld），然后想办法将vnode与真实机器建立多对一映射关系（每个真实机器上100个vnode），这个办法可以是某种策略，比如故技重施对vnodeld%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vnodeld，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（5年差不多了，像我等码农估计一辈子也玩不了一万台机器的集群吧），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vnodeld，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vnodeld (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。

上面两个老方案还可以加上一致性Hash等策略来尽量避免数据命中失败和数据迁移。但是始终逃避不了这样一个公式：

```
int machine_id=function(data.key , x)
```

machine_id指最终路由到哪台机器，function代表我们的路由策略函数，data.key就是数据的key（数据ID之类的），x在第一个方案里就是机器数量100，在第二个方案里就是vnode数量+(vnode与机器的映射关系)。在这个公式里，永远存在了x这个未知数，一旦它风吹草动，function的执行结果就可能改变，所以它逃避不了命中失败。

只有当公式变成下面这个，才能绝对避免：

```
Map<data.key,final machine_id> map = xxx;
```

```
int machine_id=map.get(data.key);
```

注意map只是个理论上的结构，这里只是简单的伪代码，并不强制它是个简单的<key-value>结构，它的结构可能会更复杂，但是无论怎么复杂，此map都真实的、明确的存在，其效果都是——用data.key就能映射到machine_id，找到目标机器，不管是直接，还是间接，反正不是用一个function去动态计算得到。map里的final不符合语法，加在这里是想强调，此map一旦为某个data.key设置了machine_id，就永不改变（起码不会因为日常扩容而改变）。当增加机器时，此map的已有值也不会受到影响。这样一个没有未知数x的公式，才能保证新老数据来了都能根据key拿到一个永远不变的machine_id，永远命中成功。

因此我们得出这样一个结论，只要拥有这样一个map，系统就能拥有非常优雅平滑的可扩展潜力。当系统扩容时，老的数据不会命中失败，在分布式协调调度的保证下，新的增量数据会更倾向于写入新机器，整个集群的负载会逐渐均衡。

很显然Haystack和TFS都做到了，下面忽略其他细节问题，着重讨论一下它们是如何装备上这个map的。

读者回顾一下3.2章节留下的那个疑惑——原始URL中到底包含什么信息，是不是只有图片ID？Directory到底需不需要维护图片ID到逻辑卷的映射？

这个“图片ID到逻辑卷的映射”，就是我们需要的map，用图片ID (data.key) 能get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。

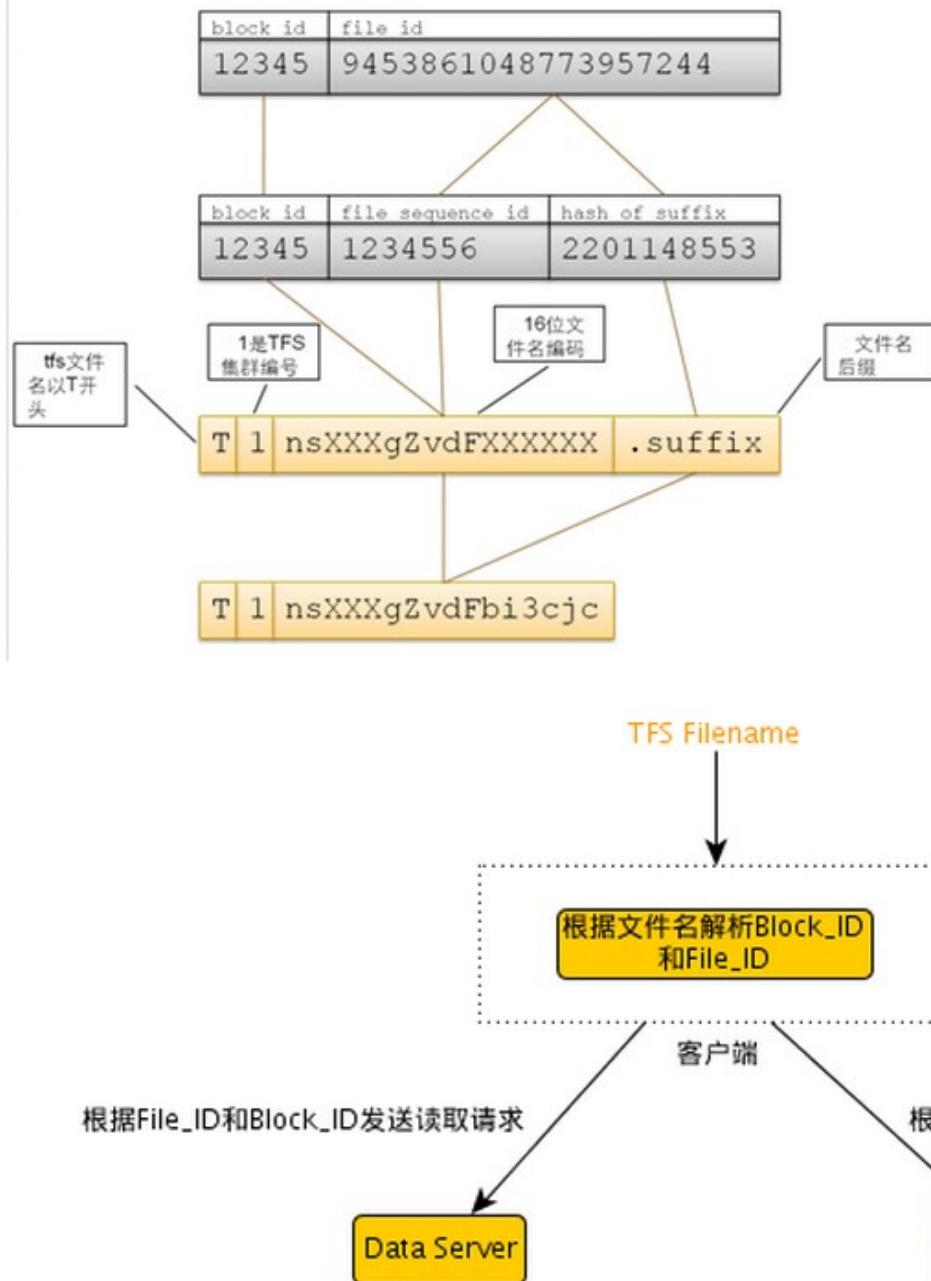
Haystack也因此，具备了十分优雅平滑的可扩展能力。但是译者提到的疑惑并没有解答——“这个映射（图片ID到逻辑卷的映射）的数据量不能忽略不计，论文也不该一笔带过”

作者提到过memcache，也许这就是相关的解决方案，此数据虽然不小，但是也没大到望而生畏的地步。不过我们依然可以发散一下，假如Haystack没保存这个映射呢？

这就意味着原始URL不只包含图片ID，还包含逻辑卷ID等必要信息。这样也是遵循map的行为定义的，即使map的信息没有集中存储在系统内，但是却分散在各个原始URL中，依然存在。不可避免的，这些信息就要在upload阶段返回给业务系统（比如Facebook的照片分享应用系统），业务系统需要理解、存储和处理它们（随后再利用它们组装为原始URL去查询图片）。这样相当于把map的维护工作分担给了各个用户系统，这也是让人十分痛苦的，导致了不可接受的耦合。

我们可以看看TFS的解决方案：

“.....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.....”



“.....根据TFS文件名解析出Block ID和block中的File ID.....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.....”

一切，迎刃而解了..... 这个方案可以称之为“结构化ID”、“聚合ID”，或者是“命名规则大于配置”。当我们纠结于仅有图片ID不够时，可以给ID简单的动动手脚，比如ID是long类型，8个byte，左边给点byte用于存储逻辑卷ID，剩下的用于存储真实的图片ID（某些场景下还可以多截几段给更多的元数据），于是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，鱼和熊掌兼得。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：

“.....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。

值得商榷的是, 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)。如果Haystack的作者能和TFS的同学们做个交流, 说不定大家都能少走点弯路吧(这都是后话了~)

小结一下, 针对第三个可扩展性痛点, 译者描述了传统方案的缺陷, 以及Haystack和TFS是如何弥补了这些缺陷, 实现了平滑优雅的可扩展能力。此小节的最后再补充一个TFS的特性:

“.....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.....”

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。

4 容错性

Haystack的容错是依靠: 一个逻辑卷对应多个物理卷(不同机器上); “客户端”向一个逻辑卷的写操作会翻译为对多个物理卷的写, 达到冗余备份; 机器故障时Directory优雅的修改应用元数据(在牵涉到的逻辑卷映射中删除此机器的物理卷项)、或者标记只读, 继而指导路由过程(分布式协调调度)将请求发送到后备的节点, 避免请求错误; 通过bulk复制重置来安全的恢复数据。等等。

在TFS中:

“.....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下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。.....”

可以看出冗余备份+协调调度是解决这类问题的惯用范式, 在大概思路上两者差不多, 但是有几个技术方案却差别很大:

第一, 冗余写机制。Haystack Store是将冗余写的责任交给“客户端”(发起写操作的客户端, 就是图3中的web server), “客户端”需要发起多次写操作到不同的Store机器上; 而TFS是依靠自身的master-slave机制, 由master向slave复制。

第二, 机房容错机制。TFS依然是遵循master-slave机制, 集群也分主辅, 主辅集群分布在不同机房, 主集群负责重放数据操作到辅集群。而Haystack在这方面没有详细介绍, 只是略微提到“.....Haystack复制每张图片到地理隔离的多个地点.....”

针对上面两点, 按译者的理解, Haystack可能更偏向于对等结构的设计, 也就是说没有master、slave之分, 各个Store是对等的节点, 没有谁负责给谁复制数据, “客户端”向各个Store写入数据, 一视同仁。

不考虑webserver、Directory等角色, 只考虑Store, 来分析一下它的容错机制: 如果单台Store挂了, 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; 一整个机房挂了, 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。

主从结构和对等结构各有所长, 视情况选择。对等结构看似简洁美好, 也有很多细节上的妥协; 主从结构增加了复杂度, 比如严格角色分配、约定角色行为等等(TFS的辅集群为何只读? 在主集群挂掉时是否依然只读? 这些比较棘手也是因为此复杂度吧)

第三, 修复机制。Haystack的修复机制依靠周期性后台任务pitchfork和离线bulk重置等。在TFS中:

“.....Dataserver后台线程介绍.....”

“.....心跳线程.....这里的心跳是指Ds向Ns发的周期性统计信息.....负责keepalive.....汇报block的工作.....”

“.....检查线程.....修复check_file_queue_中的逻辑块.....每次对文件进行读写删操作失败的时候，会try_add_repair_task(blockid, ret)来将ret错误的block加入check_file_queue_中.....若出错则会请求Ns进行update_block_info.....”

除了类似的远程心跳机制，TFS还多了在DataServer上对自身的错误统计和自行恢复，必要时还会请求上级（NameServer）帮助恢复。

5 文件系统

Haystack提到了预分配、磁盘碎片、XFS等方案，TFS中也有所涉及：

“.....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.....”

各有各的考究吧，比较了解底层的读者可以深入研究下。

6 删除和压缩

Haystack使用软删除（设置flag）、压缩回收来支持delete操作，在TFS中：

“.....压缩线程（compact_block.cpp）.....真正的压缩线程也从压缩队列中取出并进行执行（按文件进行，小文件合成一起发送）。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.....”

可见两者大同小异，这也是此类场景中常用的解决机制。

总结

本篇论文以long tail无法避免出发，探究了文件元数据导致的I/O瓶颈，推导了海量小文件的存储和检索方案，以及如何与CDN等外部系统配合搭建出整套海量图片服务。其在各个痛点的解决方案以及简约而不简单的设计值得我们学习。文章末尾将这些痛点列出并与淘宝的解决方案逐一对比，以供读者发散。

英文原文：[Facebook Haystack](#)，编译：[ImportNew - 储晓颖](#) 新浪微博：[@疯狂编码中的xiaoY](#)

译文链接：<http://www.importnew.com/3292.html>

【如需转载，请在正文中标注并保留原文链接、译文链接和译者等信息，谢谢合作！】

关于作者：储晓颖



现任支付宝架构师，负责监控分析域的架构和产品设计。架构时严谨，编码时疯狂。新浪微博：[@疯狂编码中的xiaoY](#)

[查看储晓颖的更多文章 >>](#)



相关文章

- [Infer: Facebook Java静态分析工具初探](#)
- [Facebook推出Android构建工具——Buck](#)
- [推荐给初级Java程序员的3本进阶书](#)
- [Android性能优化案例研究\(上\)](#)
- [Java8学习: Lambda表达式、Stream API和功能性接口 — 教程、资源、书籍和实例](#)
- [得墨忒耳定律](#)
- [Java语言基础特性—第一部分 \(下\)](#)
- [跟我一起学Spring 3\(4\) – 深入理解IoC\(控制反转\)和DI\(依赖注入\)](#)
- [Apache Lucene和Solr 5.0发布](#)
- [Java 8动态代理的新技巧 \(1\) : 为什么使用动态代理?](#)

发表评论

Comment form

Name*

姓名

邮箱*

请填写邮箱

网站 (请以 http://开头)

请填写网站地址

评论内容*

请填写评论内容

(*) 表示必填项

[提交评论](#)

8 条评论

1. futureage 说道:

[2013/04/12 下午 5:09](#)

你好，在“3.4.4 索引文件”，有一句话“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。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。”不是很理解，知道非孤儿文件怎么对孤儿文件的识别有影响呢，非孤儿文件是需要遍历整个卷才能发现吧，

0 0

[回复](#)

2. 储晓颖说道:

[2013/04/13 下午 9:57](#)

@futureage

我YY它的“顺序的”检查过程是从索引文件的末尾向开头扫描。

假设needle文件的数据存的是

A B C D E

其中A、B、C、D、E都是一个needle

假设索引文件存的数据是

a b c d

每一项对应的是needle的索引

从索引文件的结尾向开头扫描时，会发现d是倒数第一个，d对应的位置是D，而D不是倒数第一个，那排在倒数第一的E就有问题，就可能是孤儿

可以将这种思路发散到更多场景：

如 needle文件: A B C D E

索引文件: a b c e

扫描到e时没有问题，到c时，c是倒数第二个，c对应C却是倒数第三个，那needle文件里skip掉的倒数第二个D就可能是孤儿

如此等等。

当然了这只是我YY的，但是大致思路应该差不了多少吧。希望能给你参考。

0 0

[回复](#)

3. guo 说道:

[2013/07/01 下午 6:28](#)

hi，你这里说的文件id与物理机的唯一的永恒的map是怎么实现的好像还是没说明白呀。比如tfs，其文件名中包含了block id以及文件id，这个block id是如何得来的呢？

0 0

[回复](#)

4. 储晓颖说道:

[2013/07/07 下午 9:39](#)

@guo

hi，map的实现可以很简单，比如最简单暴力的方式，弄一个DB，一个表，表结构包含 dataId 和 machineId，出现一条新数据时都要在这个DB里“申请”一个映射（插入一条映射记录），控制一下并发。然后再通过分区缓存、分库分表来优化一下性能。手段多种多样，看你元数据有多大吧。

第二个blockId的问题，那肯定是靠元数据管理的组件分配得来的，它会根据负载均衡的原则给每个新数据分配它的归属

0 0

[回复](#)

5. pierce1024 说道:

[2013/07/09 下午 3:51](#)

Hi，孤儿问题产生的原因是store添加图片之后在更新index file之前，机器发生了重启，导致index file中没有新图片的needle。重启后机器查找孤儿应该是先定位到index file的末尾，找出末尾的图片在volume中的位置，这个位置之后的都是孤儿吧，图片是顺序存放的，新的图片只能放在后面，你给@futureage解释的needle为ABCDE，index file为abce看不懂，D怎么会放在CE中间呢？两边都是顺序存放的，顺序应该是一致的呀。

0 0

[回复](#)

6. 储晓颖说道：
[2013/07/11 上午 11:10](#)

@pierce

比如在ABC正常写入后，来了个D，写到了needle却没写到index，重启了，恢复之后，孤儿还没开始做检查，来了个E，正常写入。这样就会造成ABCDE和abce的场景了。
这种孤儿扫描的算法，个人认为就跟银行查账一样，不能给自己做任何前提假设，必须换一种全新的、能自圆其说的思路去查

0 0

[回复](#)

7. Baul说道：
[2014/02/26 上午 9:57](#)

就像高速公路上给汽车换轮子 这个是译者自己添加上去的。锦上添花！

1 0

[回复](#)

8. Baul说道：
[2014/02/26 下午 11:48](#)

【译者YY】 3.2章节是整篇文章中唯一一处译者认为没有解释清楚的环节。结合3.1章节中的URL结构解析部分，读者可以发现Directory需要拿到图片的“原始URL”（页面html中link的URL），再结合应用元数据，就可以构造出“引导URL”以供下游使用。从3.2中我们知道Directory必然保存了逻辑卷到物理卷的映射，仅用此映射+原始URL足够发掘其他应用元数据吗？原始URL中到底包含了什么信息（论文中没看到介绍）？ // Finally, the web server assigns a unique id to the photo and uploads it to each of the physical volumes mapped to the assigned logical volume. 看这句话web server 分配了图片id。具体可否判断原始url本身就用的是图片id？ // 我们可以做个假设，假如原始URL中仅仅包含图片ID，那Directory如何得知它对应哪个逻辑卷（必须先完成这一步映射，才能继续挖掘更多应用元数据）？ // The Directory maintains the logical to physical mapping along with other application metadata, such as the logical volume where each photo resides (这个就是对应关系) and the logical volumes with free spaceDirectory // 是否在upload阶段将图片ID与逻辑卷的映射也保存了？ // 确实是 // 如果是，那这个映射的数据量不能忽略不计，论文也不该一笔带过。

0 0

[回复](#)

来自微博的评论

[登录](#) | [注册](#)还可以输入 **140** 字

顺便说点什么吧.....

表情 同步到微博

0条评论

还没有人评论过，赶快抢沙发吧！

获得微博评论箱

[«为Hbase建立高可用性多主节点](#)[经典论文翻译导读之《Google File System》»](#)

Search for:



- [本月热门文章](#)
- [年度热门文章](#)
- [热门标签](#)

0 [Java实现单例的难点](#)

- 1 [使用FreeMarker替换JSP的10个理由](#)
- 2 [泛型中? super T和? extends T的区别](#)
- 3 [Stackoverflow上人气最旺的10个Java问题](#)
- 4 [快速高效学习Java编程在线资源Top 20](#)
- 5 [理解Spring MVC Model Attribute 和 Session Attribute](#)
- 6 [完全理解Gson \(3\) : Gson反序列化](#)
- 7 [用Java 8 Lambda表达式实现设计模式：命令模式](#)
- 8 [知名公司的Java面试题](#)
- 9 [Java编程入门 \(1.7\) : 互联网](#)



最新评论

-  Re: [使用FreeMarker替换JSP的10个理由](#)
用JSP的好处就是能够自动补全了。Freemarker的插件不好用啊，还要自己检查拼写。[zonghua](#)
-  Re: [泛型中? super T和? extends T的区别](#)
里面的? extends T打不上去 ken
-  Re: [泛型中? super T和? extends T的区别](#)
List foo3 = new ArrayList(); 这块编译错误： Cannot instant... ken
-  Re: [Java线程面试题 Top 50](#)
很多答案是错的 凉粉
-  Re: [给Eclipse提速的7个技巧](#)
我去、niubility啊~。~大神、您还收弟子么、我想拜你为师。 箱子

-  Re: [Java实现单例的难点](#)
public class Single {public final static Single ge... weiguangyue
-  Re: [Java实现单例的难点](#)
总结得非常好，能转载吗？我的个人博客：<http://qifuguang.me> [winwill2012](#)
-  Re: [使用FreeMarker替换JSP的10个理由](#)
好吧，我更喜欢JSP。小松鼠



关于ImportNew

ImportNew 专注于 Java 技术分享。于2012年11月11日 11:11正式上线。是的，这是一个很特别的时刻 :)

ImportNew 由两个 Java 关键字 import 和 new 组成，意指：Java 开发者学习新知识的网站。import 可认为是学习和吸收，new 则可认为是新知识、新技术圈子和新朋友.....

- 
- 

推荐关注

- [小组](#) - 好的话题、有启发的回复、值得信赖的圈子
- [头条](#) - 写了文章？看干货？去头条！
- [相亲](#) - 为IT单身男女服务的征婚传播平台
- [资源](#) - 优秀的工具资源导航
- [翻译](#) - 活跃 & 专业的翻译小组
- [博客](#) - 国内外的精选博客文章
- [前端](#) - JavaScript, HTML5, CSS
- [安卓](#) - 专注Android技术分享
- [iOS](#) - 专注iOS技术分享
- [Java](#) - 专注Java技术分享
- [Python](#) - 专注Python技术分享

联系我们

Email: ImportNew.com@gmail.com

新浪微博: [@ImportNew](#)

微信号: importnew

反馈建议: ImportNew.com@gmail.com

广告与商务合作QQ: 2302462408

© 2015 ImportNew