

ImportNew

- [首页](#)
- [所有文章](#)
- [资讯](#)
- [Web](#)
- [架构](#)
- [基础技术](#)
- [书籍](#)
- [教程](#)
- [Java/小组](#)
- [工具资源](#)

- 导航条 - ▾

经典论文翻译导读之《Google File System》

2013/03/26 | 分类：[未分类](#) | [3 条评论](#)

分享到：

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

【译者预读】



GFS这三个字母无需过多修饰，《Google File System》的论文也早有译版。但是这不妨碍我们加点批注、重温经典，并结合上篇[Haystack](#)的文章，将GFS、TFS、Haystack进行一次全方位的对比，一窥各巨头的架构师们是如何权衡利弊、各取所需。

1. 介绍

我们设计和实现了GFS来满足Google与日俱增的数据处理需求。与传统的分布式文件系统一样，GFS着眼在几个重要的目标，比如性能、可伸缩性、可靠性和可用性。不过它也会优先考虑我们自身应用场景的特征和技术环境，所以与早先一些文件系统的设计思想还是有诸多不同。我们取传统方案之精华、根据自身需求做了大胆的设计创新。在我们的场景中：

首先，组件故障是常态而不是异常。文件系统包含成百上千的存储机器，而且是廉价的普通机器，被大量的客户端机器访问。这样的机器质量和数量导致任何时间点都可能有一些机器不可用，甚至无法从当前故障中恢复。导致故障的原因很多，比如应用bug、操作系统bug、人为错误，以及磁盘、内存、连接器、网络等硬件故障，甚至是电力供应。因此，持续监控、错误侦测、故障容忍和自动恢复必须全面覆盖整个系统。

其次，用传统视角来看，我们要处理的文件很多都是巨型的，好几GB的文件也很常见。通常情况下每个文件中包含了多个应用对象，比如web文档。面对快速增长、TB级别、包含数十亿对象的数据集合，如果按数十亿个KB级别的小文件来管理，即使文件系统能支持，也是非常不明智的。因此，一些设计上的假设和参数，比如I/O操作和块大小，需要被重新审视。

第三，大部分文件发生变化是通过append新数据，而不是覆盖、重写已有的数据，随机写几乎不存在。被写入时，文件变成只读，而且通常只能是顺序读。很多数据场景都符合这些特征。比如文件组成大型的库，使用数据分析程序对其扫描。比如由运行中的程序持续生成的数据流。比如归档数据。还可能是分布式计算的中间结果，在一台机器上产生、然后在另一台处理。这些数据场景都是由制造者持续增量的产生新数据，再由消费者读取处理。在这种模式下append是性能优化和保证原子性的焦点。然而在客户端缓存数据块没有太大意义。

第四，向应用提供类似文件系统API，增加了我们的灵活性。松弛的一致性模型设计也极大的简化了API，不会给应用程序强加繁重负担。我们将介绍一个原子的append操作，多客户端能并发的对一个文件执行append，不需考虑任何同步。

当前我们部署了多个GFS集群，服务不同的应用。最大的拥有超过1000个存储节点，提供超过300TB的磁盘存储，被成百上千个客户端机器大量访问。

2 设计概览

2.1 假设

设计GFS过程中我们做了很多的设计假设，它们既意味着挑战，也带来了机遇。现在我们详细描述下这些假设。

- 系统是构建在很多廉价的、普通的组件上，组件会经常发生故障。它必须不间断监控自己、侦测错误，能够容错和快速恢复。
- 系统存储了适当数量的大型文件，我们预期几百万个，每个通常是100MB或者更大，即使是GB级别的文件也需要高效管理。也支持小文件，但是不需要着重优化。
- 系统主要面对两种读操作：大型流式读和小型随机读。在大型流式读中，单个操作会读取几百KB，也可以达到1MB或更多。相同客户端发起的连续操作通常是在一个文件读取一个连续的范围。小型随机读通常在特定的偏移位置上读取几KB。重视性能的应用程序通常会将它们的小型读批量打包、组织排序，能显著的提升性能。
- 也会面对大型的、连续的写，将数据append到文件。append数据的大小与一次读操作差不多。一旦写入，几乎不会被修改。不过在文件特定位置的小型写也是支持的，但没有着重优化。
- 系统必须保证多客户端对相同文件并发append的高效和原子性。我们的文件通常用于制造者消费者队列或者多路合并。几百个机器运行的制造者，将并发的append到一个文件。用最小的同步代价实现原子性是关键所在。文件被append时也可能出现并发的读。
- 持久稳定的带宽比低延迟更重要。我们更注重能够持续的、大批量的、高速度的处理海量数据，对某一次读写操作的回复时间要求没那么严格。



2.2 接口

GFS提供了一个非常亲切的文件系统接口，尽管它没有全量实现标准的POSIX API。像在本地磁盘中一样，GFS按层级目录来组织文件，一个文件路径（path）能作为一个文件的唯一ID。我们支持常规文件操作，比如create、delete、open、close、read和write。

除了常规操作，GFS还提供快照和record append操作。快照可以用很低的花费为一个文件或者整个目录树创建一个副本。record append允许多个客户端并发的append数据到同一个文件，而且保证它们的原子性。这对于实现多路合并、制造消费者队列非常有用，大量的客户端能同时的append，也不用要考虑锁等同步问题。这些特性对于构建大型分布式应用是无价之宝。快照和record append将在章节3.4、3.3讨论。

2.3 架构

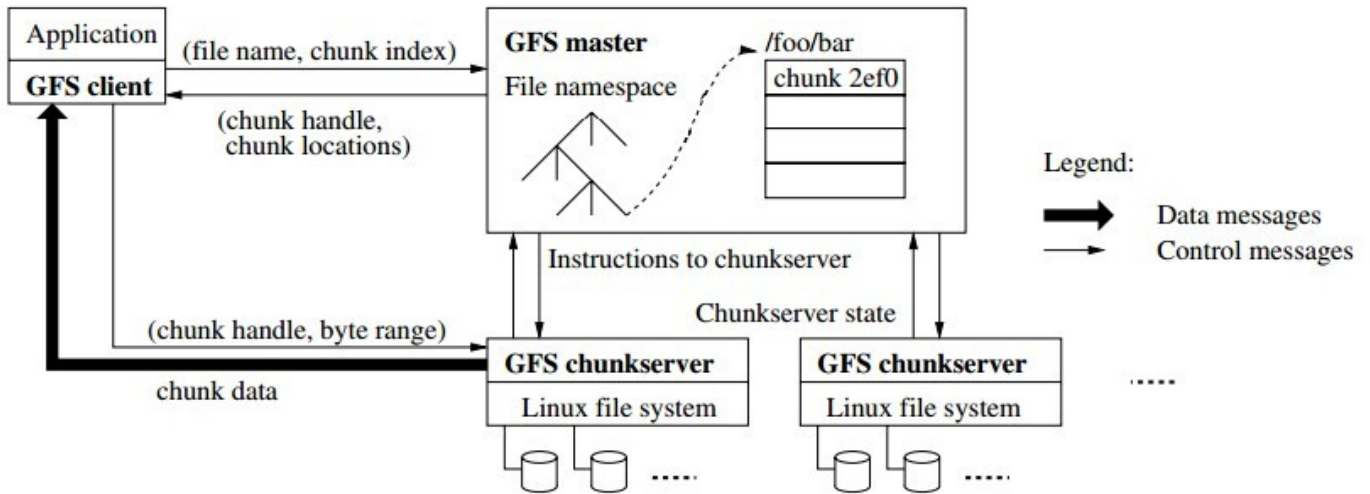


Figure 1: GFS Architecture

一个GFS集群包含单个master和多个chunkserver，被多个客户端访问，如图1所示。图1中各组件都是某台普通Linux机器上运行在用户级别的一个进程。在同一台机器上一起运行chunkserver和客户端也很容易，只要机器资源允许。

文件被划分为固定大小的chunk。每个chunk在创建时会被分配一个chunk句柄，chunk句柄是一个不变的、全局唯一的64位的ID。chunkserver在本地磁盘上将chunk存储为Linux文件，按照chunk句柄和字节范围来读写chunk数据。为了可靠性，每个chunk被复制到多个chunkserver上，默认是3份，用户能为不同命名空间的文件配置不同的复制级别。

master维护所有的文件系统元数据。包括命名空间，访问控制信息，从文件到chunk的映射，和chunk位置。它也负责主导一些影响整个系统的活动，比如chunk租赁管理、孤儿chunk的垃圾回收，以及chunkserver之间的chunk迁移。master会周期性的与每台chunkserver通讯，使用心跳消息，以发号施令或者收集chunkserver状态。

每个应用程序会引用GFS的客户端API，此API与正规文件系统API相似，并且负责与master和chunkserver通讯，基于应用的行为来读写数据。客户端只在获取元数据时与master交互，真实的数据操作会直接发至chunkserver。我们不需提供严格完整的POSIX API，因此不需要hook到Linux的vnode层面。

客户端和chunkserver都不会缓存文件数据。客户端缓存文件数据收益很小，因为大部分应用通常会顺序扫描大型文件，缓存重用率不高，要么就是工作集合太大缓存很困难。没有缓存简化了客户端和整个系统，排除缓存一致性问题。（但是客户端会缓存元数据。）chunkserver不需要缓存文件数据因为chunk被存储为本地文件，Linux提供的OS层面的buffer缓存已经保存了频繁访问的文件。

2.4 单一Master

单一master大大的简化了我们的设计，单一master能够放心使用全局策略执行复杂的chunk布置、制定复制决策等。然而，我们必须在读写过程中尽量减少对它的依赖，它才不会成为一个瓶颈。客户端从不通过master读写文件，它只会询问master自己应该访问哪个chunkserver。客户端会缓存这个信息一段时间，随后的很多操作即可以复用此缓存，与chunkserver直接交互。

我们利用图1来展示一个简单读操作的交互过程。首先，使用固定的chunk size，客户端将应用程序指定的文件名和字节偏移量翻译为一个GFS文件及内部chunk序号，随后将它们作为参数，发送请求到master。master找到对应的chunk句柄和副本位置，回复给客户端。客户端缓存这些信息，使用GFS文件名+chunk序号作为key。

客户端然后发送一个读请求到其中一个副本，很可能是最近的那个。请求中指定了chunk句柄以及在此chunk中读取的字节范围。后面对相同chunk的读不再与master交互，直到客户端缓存信息过期或者文件被重新打开。事实上，客户端通常会在一个与master的请求中顺带多索要一些其他chunk的信息，而且master也可能将客户端索要chunk后面紧跟的其他chunk信息主动回复回去。这些额外的信息避免了未来可能发生的一些client-master交互，几乎不会导致额外的花费。

2.5 chunk size

chunk size是其中一个关键的设计参数。我们选择了64MB，这是比典型的文件系统的块大多了。每个chunk副本在chunkserver上被存储为一个普通的Linux文件，只在必要的时候才去扩展。懒惰的空间分配避免了内部碎片导致的空

间浪费，chunk size越大，碎片的威胁就越大。

chunk size较大时可以提供几种重要的优势。首先，它减少了客户端与master的交互，因为对同一个chunk的读写仅需要对master执行一次初始请求以获取chunk位置信息。在我们的应用场景中大部分应用会顺序的读写大型文件，chunk size较大（chunk数量就较少）能有效的降低与master的交互次数。对于小型的随机读，即使整个数据集合达到TB级别，客户端也能舒服的缓存所有的chunk位置信息（因为chunk size大，chunk数量小）。其次，既然用户面对的是较大的chunk，它更可能愿意在同一个大chunk上执行很多的操作（而不是操作非常多的小chunk），这样就可以对同一个chunkserver保持长期的TCP连接以降低网络负载。第三，它减少了master上元数据的大小，这允许我们放心的在内存缓存元数据，章节2.6.1会讨论继而带来的各种好处。

不过chunk size如果很大，即使使用懒惰的空间分配，也有它的缺点。一个小文件包含chunk数量较少，可能只有一个。在chunkserver上这些chunk可能变成热点，因为很多客户端会访问相同的文件（如果chunk size较小，那小文件也会包含很多chunk，资源竞争可以分担到各个小chunk上，就可以缓解热点）。不过实际上热点没有导致太多问题，因为我们的应用大部分都是连续的读取很大的文件，包含很多chunk（即使chunk size较大）。

然而，热点确实曾经导致过问题，当GFS最初被用在批量队列系统时：用户将一个可执行程序写入GFS，它只占一个chunk，然后几百台机器同时启动，请求此可执行程序。存储此可执行文件的chunkserver在过多的并发请求下负载较重。我们通过提高它的复制级别解决了这个问题（更多冗余，分担压力），并且建议该系统交错安排启动时间。一个潜在的长期解决方案是允许客户端从其他客户端读取数据（P2P模式~）。

2.6 元数据

master主要存储三种类型的元数据：文件和chunk的命名空间，从文件到chunk的映射，每个chunk副本的位置。所有的元数据被保存在master的内存中。前两种也会持久化保存，通过记录操作日志，存储在master的本地磁盘并且复制到远程机器。使用操作日志允许我们更简单可靠的更新master状态，不会因为master的当机导致数据不一致。master不会持久化存储chunk位置，相反，master会在启动时询问每个chunkserver以获取它们各自的chunk位置信息，新chunkserver加入集群时也是如此。



2.6.1 内存中数据结构

因为元数据存储在内存中，master可以很快执行元数据操作。而且可以简单高效的在后台周期性扫描整个元数据状态。周期性的扫描作用很多，有些用于实现chunk垃圾回收，有些用于chunkserver故障导致的重新复制，以及为了均衡各机器负载与磁盘使用率而执行的chunk迁移。章节4.3和4.4将讨论其细节。

这么依赖内存不免让人有些顾虑，随着chunk的数量和今后整体容量的增长，整个系统将受限于master有多少内存。不过实际上这不是一个很严重的限制。每个64MB的chunk，master为其维护少于64byte的元数据。大部分chunk是填满数据的，因为大部分文件包含很多chunk，只有少数可能只填充了部分。同样的，对于文件命名空间数据，每个文件只能占用少于64byte，文件名称会使用前缀压缩紧密的存储。

如果整个文件系统真的扩展到非常大的规模，给master添点内存条、换台好机器scale up一下也是值得的。为了单一master+内存中数据结构所带来的简化、可靠性、性能和弹性，咱豁出去了。

2.6.2 Chunk位置

master不会持久化的保存哪个chunkserver有哪些chunk副本。它只是在自己启动时拉取chunkserver上的信息（随后也会周期性的执行拉取）。master能保证它自己的信息时刻都是最新的，因为它控制了所有的chunk布置操作，并用常规心跳消息监控chunkserver状态。

我们最初尝试在master持久化保存chunk位置信息，但是后来发现这样太麻烦，每当chunkserver加入或者离开集群、改变名称、故障、重启等等时候就要保持master信息的同步。一般集群都会有几百台服务器，这些事件经常发生。

话说回来，只有chunkserver自己才对它磁盘上存了哪些chunk有最终话语权。没理由在master上费尽心机的维护一个一致性视图，chunkserver上发生的一个错误就可能导致chunk莫名消失（比如一个磁盘可能失效）或者运维人员可能重命名一个chunkserver等等。

2.6.3 操作日志

操作日志是对重要元数据变更的历史记录。它是GFS的核心之一。不仅因为它是元数据唯一的持久化记录，而且它还要承担一个逻辑上的时间标准，为并发的操作定义顺序。各文件、chunk、以及它们的版本（见章节4.5），都会根据它们创建时的逻辑时间被唯一的、永恒的标识。

既然操作日志这么重要，我们必须可靠的存储它，而且直至元数据更新被持久化完成（记录操作日志）之后，才能让变化对客户端可见。否则，我们有可能失去整个文件系统或者最近的客户端操作，即使chunkserver没有任何问题（元数据丢了或错了，chunkserver没问题也变得有问题了）。因此，我们将它复制到多个远程机器，直到日志记录被flush到本地磁盘以及远程机器之后才会回复客户端。master会捆绑多个日志记录，一起flush，以减少flush和复制对整个系统吞吐量的冲击。

master可以通过重放操作日志来恢复它的元数据状态。为了最小化master的启动时间，日志不能太多（多了重放就需要很久）。所以master会在适当的时候执行“存档”，每当日志增长超过一个特定的大小就会执行存档。所以它不需要从零开始回放日志，仅需要从本地磁盘装载最近的存档，并回放存档之后发生的有限数量的日志。存档是一个紧密的类B树结构，它能直接映射到内存，不用额外的解析。通过这些手段可以加速恢复和改进可用性。

因为构建一个存档会消耗点时间，master的内部状态做了比较精细的结构化设计，创建一个新的存档不会延缓持续到来的请求。master可以快速切换到一个新的日志文件，在另一个后台线程中创建存档。这个新存档能体现切换之前所有的变异结果。即使一个有几百万文件的集群，创建存档也可以在短时间完成。结束时，它也会写入本地和远程的磁盘。

恢复元数据时，仅仅需要最后完成的存档和其后产生的日志。老的存档和日志文件能被自由删除，不过我们保险起见不会随意删除。在存档期间如果发生故障（存档文件烂尾了）也不会影响正确性，因为恢复代码能侦测和跳过未完成的存档。

2.7 一致性模型

GFS松弛的一致性模型能很好的支持我们高度分布式的应用，而且实现起来非常简单高效。我们现在讨论GFS的一致性保证。

2.7.1 GFS的一致性保证

文件命名空间变化（比如文件创建）是原子的，只有master处理此种操作：master中提供了命名空间的锁机制，保证了原子性的和正确性（章节4.1）；master的操作日志为这些操作定义了一个全局统一的顺序（章节2.6.3）

	Write	Record Append
Serial success	<i>defined</i>	<i>defined interspersed with inconsistent</i>
Concurrent successes	<i>consistent but undefined</i>	
Failure	<i>inconsistent</i>	

Table 1: File Region State After Mutation

各种数据变异在不断发生，被它们改变的文件区域处于什么状态？这取决于变异是否成功了、有没有并发变异等各种因素。表1列出了所有可能的结果。对于文件区域A，如果所有客户端从任何副本上读到的数据都是相同的，那A就是一致的。如果A是一致的，并且客户端可以看到变异写入的完整数据，那A就是defined。当一个变异成功了、没有受到并发写的干扰，它写入的区域将会是defined（也是一致的）：所有客户端都能看到这个变异写入的完整数据。对同个区域的多个并发变异成功写入，此区域是一致的，但可能是undefined：所有客户端看到相同的数据，但是它可能不会反应任何一个变异写的东西，可能是多个变异混杂的碎片。一个失败的变异导致区域不一致（也是undefined）：不同客户端可能看到不同的数据在不同的时间点。下面描述我们的应用程序如何区分defined区域和undefined区域。

数据变异可能是写操作或者record append。写操作导致数据被写入一个用户指定的文件偏移。而record append导致数据（record）被原子的写入GFS选择的某个偏移（正常情况下是文件末尾，见章节3.3），GFS选择的偏移被返回给客户端，其代表了此record所在的defined区域的起始偏移量。另外，某些异常情况可能会导致GFS在区域之间插入了padding或者重复的记录。他们占据的区域可认为是不一致的，不过数据量不大。

如果一系列变异都成功写入了，GFS保证发生变异的文件区域是defined的，并完整的包含最后一个变异。GFS通过两点来实现：(a)chunk的所有副本按相同的顺序来实施变异（章节3.1）；(b)使用chunk版本数来侦测任何旧副本，副本变旧可能是因为它发生过故障、错过了变异（章节4.5）。执行变异过程时将跳过旧的副本，客户端调用master获取chunk位置时也不会返回旧副本。GFS会尽早的通过垃圾回收处理掉旧的副本。

因为客户端缓存了chunk位置，所以它们可能向旧副本发起读请求。不过缓存项有超时机制，文件重新打开时也会更新。而且，我们大部分的文件是append-only的，这种情况下旧副本最坏只是无法返回数据（append-only意味着只增不减也不改，版本旧只意味着会丢数据、少数据），而不会返回过期的、错误的数据。一旦客户端与master联系，它将立刻得到最新的chunk位置（不包含旧副本）。

在一个变异成功写入很久之后，组件的故障仍然可能腐化、破坏数据。GFS中，master和所有chunkserver之间会持续handshake通讯并交换信息，借此master可以识别故障的chunkserver并且通过检查checksum侦测数据腐化（章节5.2）。一旦发现此问题，会尽快执行一个restore，从合法的副本复制合法数据替代腐化副本（章节4.3）。一个chunk也可能发生不可逆的丢失，那就是在GFS反应过来采取措施之前，所有副本都被丢失。通常GFS在分钟内就能反应。即使出现这种天灾，chunk也只是变得不可用，而不会腐化：应用收到清晰的错误而不是错误的数据。

【译者注】一致性的问题介绍起来难免晦涩枯燥，下面译者用一些比较浅显的例子来解释GFS中的一致、不一致、defined、undefined四种状态。

读者可以想象这样一个场景，某人和他老婆共用同一个Facebook账号，同时登陆，同时看到某张照片，他希望将其顺时针旋转90度，他老婆希望将其逆时针旋转90度。两人同时点了修改按钮，Facebook应该听谁的？俗话说意见相同听老公的，意见不同听老婆的。但是Facebook不懂这个算法，当他们重新打开页面时可能会：1 都看到图片顺时针旋转了90度；2 都看到图片逆时针旋转了90度；3 其他情况。对于1、2两种情况，都是可以接受的，小夫妻若来投诉那只能如实相告让他们自己回去猜拳，不关Facebook的事儿。因为1、2既满足一致性（两人在并发修改发生后都一直看到一致相同的内容），又满足defined（内容是其中一人写入的完整数据）。对于3会有哪些其他情况呢？如果这事儿发生在单台电脑的本地硬盘（相当于两人同时打开一个图片软件、编辑同一个图片、然后并发提交保存），若不加锁让其串行，则可能导致数据碎片，以简单的代码为例：

```
File file = new File("D:/temp.txt");
FileOutputStream fos1 = new FileOutputStream(file);
FileOutputStream fos2 = new FileOutputStream(file);
fos1.write('1');
fos1.write('2');
fos1.write('3');
fos2.write('a');
fos2.write('b');
fos2.write('c');
fos1.close();
fos2.close();
```



这样一段代码可以保证temp.txt的内容是“abc”（fos2写入的字节流完全覆盖了fos1），fos2写入是完全的，也就是defined。而写入字节流是一个持续过程，不是原子的，如果在多线程环境下则可能因为线程调度、I/O中断等因素导致代码的执行顺序交错，形成这样的效果：

```
File file = new File("D:/temp.txt");
FileOutputStream fos1 = new FileOutputStream(file);
FileOutputStream fos2 = new FileOutputStream(file);
fos1.write('1');
fos2.write('a');
fos2.write('b');
fos1.write('2');
fos1.write('3');
fos2.write('c');
fos1.close();
fos2.close();
```

这段代码导致temp.txt的内容变成了“a2c”，它不是fos1的写入也不是fos2的写入，它是碎片的组合，这就是undefined状态。还有更糟的情况，这种情况在单台电脑本地硬盘不会出现，而会在分布式文件系统上出现：分布式文件系统都有冗余备份，fos1和fos2的写入需要在每个副本上都执行，而在每个副本上会因为各自的线程调度、I/O中断导致交错的情况不一、顺序不一，于是出现了副本数据不一致的情况（不仅有a2c，还可能是12c、1b3等等），在查询时由于会随机选择副本，于是导致多个查询可能看到各种不一致的数据。这就是既不一致又undefined的情况。在分布式文件系统上还有另一种情况，在各副本上fos1和fos2都没有交错产生碎片，但是它们整体顺序不一致，一个副本产出了123，另一个产出了abc，这种也是不一致的异常情况。

如何解决上述问题呢？比较可行的方案就是串行化，按顺序执行，fos1写完了才轮到fos2。不过即使如此也不能完全避免一些令人不悦的现象：比如fos1要写入的是“12345”，fos2要写入的是“abc”，即使串行，最后也会产出“abc45”。不过对于这种现象，只能认为是外界需求使然，不是文件系统能解决的，GFS也不会把它当做碎片，而认为它是defined。在分布式环境下，不仅要保证每个副本串行执行变异，还要保证串行的顺序是一致的，GFS的对策就是后文中的租赁机制。这样还不够，还要谨防某个副本因为机器故障而执行异常，GFS的对策是版本侦测机制，利用版本侦测剔除异常的副本。

2.7.2 对应用的启示

在使用GFS时，应用如果希望达到良好的一致性效果，需要稍作考虑以配合GFS的松弛一致性模型。但GFS的要求并不高，而且它要求的事情一般你都会去做（为了某些其他的目的）：比如GFS希望应用使用append写而不是覆盖重写，以及一些自我检查、鉴定和验证的能力。

无论你面对GFS还是普通的本地文件API（比如FileInputStream、FileOutputStream），有些一致性问题你都要去考虑。当一个文件正在被写时，它依然可以被另一个线程读，写入磁盘不是一瞬间的事情，当然有可能读到没有写入完全的数据（可以理解为上述的undefined情况，你只看到了碎片没有看到完整写入的内容），这种情况GFS不会帮你解决（它是按照标准文件API来要求自己的，标准文件API也没有帮你解决这种问题）。比较严谨的程序会使用各种方法来避免此问题，比如先写入临时文件，写入结束时才原子的重命名文件，此时才对读线程可见。或者在写入过程中不断记录写入完成量，称之为checkpoint，读线程不会随意读文件的任何位置，它会判断checkpoint（哪个偏移之前的数据是写入完全的，也就是defined），checkpoint甚至也可以像GFS一样做应用级别的checksum。这些措施都可以帮助reader只读取到defined区域。

还有这种情况：你正在写入一个文件，将新数据append到文件末尾，还没结束时程序异常或者机器故障了，于是你必须重试，但是之前那次append可能已经写入了部分数据，这部分数据也是undefined，也不希望让reader读到。无论在本地磁盘还是在GFS上都要面临这种问题。不过这一点上GFS提供了一些有效的帮助。在GFS里，刚才那种情况可能会导致两种异常，一是没有写入完全的padding，二是重复数据（GFS有冗余副本，写入数据时任一副本故障会导致所有副本都重试，这就可能导致正常的副本上不止写入一次）。对于padding，GFS提供了checksum机制，读取时通过简单的核查即可跳过不合法的padding。不过对于重复，应用如果不能容忍的话最好能加强自身的幂等性检查，比如当你将大量应用实体写入文件时，实体可以包含ID，读取实体进行业务处理时能通过ID的幂等性检查避免重复处理。

GFS虽然没有直接在系统层面解决上述难以避免的一致性问题，但是上面提到的解决方案都会作为共享代码库供大家使用。

3 系统交互

在GFS的架构设计中，我们会竭尽所能的减少所有操作对master的依赖（因为架构上的牺牲权衡，master是个理论上的单点）。在这个背景下，下面将描述客户端、master、chunkserver之间是如何交互，最终实现了各种数据变异、原子的record append、快照等特性。

3.1 租赁和变异顺序

变异可以理解作为一种操作，此操作会改变chunk的数据内容或者元数据，比如一个写操作或者一个append操作。对chunk的任何变异都需要实施到此chunk的各个副本上。我们提出了一种“租赁”机制，来维护一个跨副本的一致性变异顺序。master会在chunk各副本中选择一个，授予其租赁权，此副本称之为首要副本，其他的称之为次级副本。首要副本负责为chunk的所有变异排出一个严格的顺序。所有副本在实施变异时都必须遵循此顺序。因此，全局统一的变异过程可以理解为：首先由master选出首要和次级副本；首要副本为这些变异制定实施序号；首要和次级副本内严格按首要副本制定的序号实施变异。

租赁机制需要尽量减少对master产生的负载。一个租赁初始的超时时间为60秒。然而只要chunk正在实施变异，首要副本能向master申请连任，一般都会成功。master和所有chunkserver之间会持续的交换心跳消息，租赁的授予、请求连任等请求都是在这个过程中完成。master有时候会尝试撤回一个还没过期的租赁（比如要重命名一个文件，master希望暂停所有对其实施的变异）。即使master与首要副本失去通讯，它也能保证在老租赁过期后安全的选出一个新的首要副本。

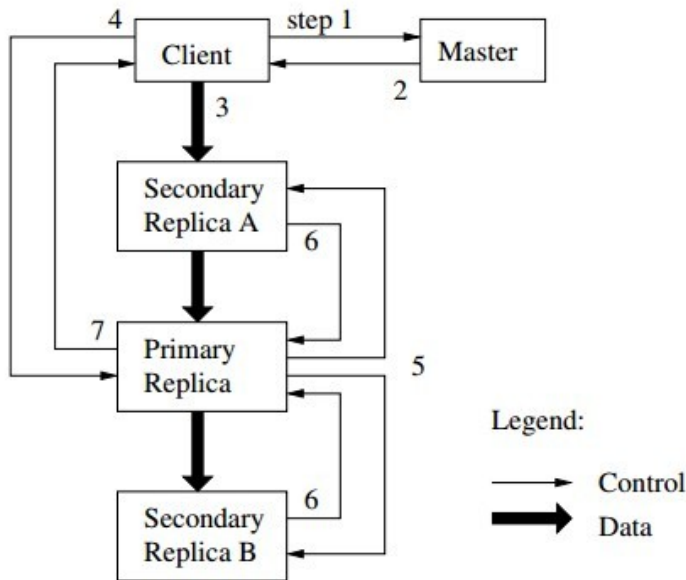


Figure 2: Write Control and Data Flow

图2描述了具体的控制流程，其中步骤的解释如下：

1. 客户端要对某chunk执行操作，它询问master哪个chunkserver持有其租赁以及各副本的位置信息。如果没有任何人拿到租赁，master选择一个副本授予其租赁（此时不会去通知这个副本）。
2. master将首要者、副本位置信息回复到客户端。客户端缓存这些数据以便未来重用，这样它仅需要在当前首要副本无法访问或者卸任时去再次联系master。
3. 客户端推送数据到所有的副本。只是推送，不会实施，只是在各chunkserver上将数据准备好，推送的顺序也与控制流无关。每个副本所在的chunkserver将数据储存在一个内部的LRU的缓冲中，直到数据被使用或者过期。通过将数据流和控制流解耦，我们能有效的改进性能，实现基于网络拓扑的算法来调度“昂贵”的数据流，而不需要关心控制流中哪个chunkserver是首要的还是次要的。章节3.2将讨论此算法的细节。
4. 一旦所有副本都确认收到了数据，客户端正式发送一个写请求到首要副本。写请求无真实数据，只有一个身份标识，对应第三步中发给各个副本的数据包。在首要副本中会持续的收到来自各个客户端的各种变异请求，本次写请求只是其中一个而已。在持续接收请求的过程中，首要副本会为每个请求分配唯一的递增序号，它也会严格按照此顺序来在本地状态中实施变异。
5. 首要将写请求推送到所有次级副本（请求中已带有分配的序号），每个次级副本都会严格按顺序依次实施变异。
6. 次级副本回复给首要的，确认他们已完成操作
7. 首要副本回复客户端。在任何副本遭遇的任何错误，都被汇报给客户端。在错误发生时，此写操作可能已经在首要和某些次级副本中实施成功。（如果它首要就失败，就不会分配序号也不会往后推进。）客户端则认为此次请求失败，请求所修改的区域变成了不一致状态。对于失败变异，客户端会重试，它首先会做一些尝试在步骤3到步骤7，实在不行就重试整个流程。

一个写请求（非append）可能很大，跨越了chunk边界，GFS客户端代码会将其拆分为对多个chunk的多个写操作。各个写操作都遵从上述控制流，但是也可能因为来自其他客户端的并发写导致某几个子操作的文件区域产生数据碎片。不过即使如此，各副本的数据是相同的，因为此控制流保证了所有副本执行的变异顺序是完全一致的。所以即使某些区域产生了碎片，还是满足一致性的，但是会处于undefined状态（章节2.7描述的）。

【译者YY】上述流程中多次提到要按顺序、依次、串行等词汇，来避免并发导致的一致性问题。这些会不会导致性能问题？毕竟这是一个I/O密集型系统，请求串行化不是一个值得骄傲的解决方案。文章末尾对此疑问会尝试解答。

3.2 数据流

我们将数据流和控制流解耦来更高效的利用网络。从上述控制流的分析中可以看出，从客户端到首要副本然后到所有次级副本，请求是沿着一个小心谨慎的链路、像管道一样，在各个chunkserver之间推送。我们不能容忍真实数据的流程被此严谨的控制流绑架，我们的目标是最大化利用每个机器的网络带宽，避免网络瓶颈和高延迟连接，最小化推送延迟。

为了最大化利用每台机器的网络带宽，我们让数据沿着一个线性链路推送（chunkserver就是链路中的一个节点），而不是零乱地分布于其他拓扑结构中（比如树状）。我们希望每台机器都会使用全量带宽尽快传输一整批数据，而不是频繁收发零乱的小批数据。

为了尽可能的避免网络瓶颈和高延迟连接（内联交换机经常遇到此问题），每个机器都会尝试推送数据到网络拓扑中最近的其他目标机器。假设客户端希望推送数据到chunkserver S1、S2、S3、S4。不管网络拓扑结构如何，我们假设S1离客户端最近，S2离S1最近。首先客户端会发送数据到最近的S1；S1收到数据，传输目标减少为[S2、S3、S4]，继而推送到离S1最近的S2，传输目标减少为[S3、S4]。相似的，S2继续推送到S3或者S4（看谁离S2更近），如此继续。我们的网络拓扑并不复杂，可以用IP地址准确的预估出“距离”。

最后，我们使用TCP流式传输数据，以最小化延迟。一旦chunkserver收到数据，它立刻开始推送。TCP管道流式传输的效果显著，因为我们使用的是switched network with full-duplex links。立刻发送数据并不会影响接收速度。没有网络拥挤的情况下，传输B个字节到R个副本的理想耗时是 $B/T + RL$ ，T是网络吞吐量，L是在机器间传输字节的延迟。我们网络连接是典型的100Mbps（T），L小于1ms，因此1MB的数据流大约耗时80ms。

3.3 原子append

GFS提供了原子append能力，称之为record append。在传统的写操作中，客户端指明偏移量，写入时seek到此偏移，然后顺序的写入新数据。而record append操作中，客户端仅需要指明数据。GFS可以选择一个偏移量（一般是文件末尾），原子的将数据append到此偏移量，至少一次（没有数据碎片，是一个连续序列的字节）偏移量被返回到客户端。类似的，UNIX中多个writer并发写入O_APPEND模式打开的文件时也没有竞争条件。

record append在我们分布式应用中被大量的使用，其中很多机器上的大量客户端会并发的append到相同的文件。如果用传统的写模式，将严重增加客户端的复杂度，实施昂贵的同步，比如通过一个分布式锁管理器。我们的实际应用场景中，record append经常用于多个制造者、单个消费者队列情景，或者用于存储多客户端的合并结果。

record append也是一种变异，遵从控制流（章节3.1），但是会需要首要副本执行一点点额外的逻辑。客户端将数据推送到文件末尾对应的chunk的所有副本上。然后发送写请求到首要副本。首要副本需要检查append到此chunk是否会导致chunk超过最大的size（64MB）。如果超过，它将此chunk填补到最大size，并告诉次级副本也这么做，随后回复客户端这个操作需要重试，并使用下一个chunk（上一个chunk刚刚已经被填满，文件末尾会对应到一个新chunk）。record append的数据大小被限制为小于等于chunk maxsize的四分之一，这样可以避免填补导致的过多碎片。如果不需要填补（通常都不需要），首要副本append数据到它的副本，得出其偏移量，并告诉次级副本将数据准确的写入此偏移，最终回复客户端操作已成功。

如果一个record append在任何副本失败了，客户端需要重试。因此，同一个chunk的各个副本可能包含不同的数据，各自都可能包含重复的记录。GFS不保证所有副本是字节上相同的。它仅仅保证record append能原子执行，写入至少一次。不过有一点可以保证，record append最终成功后，所有副本写入此有效record的偏移量是相同的。另外，所有副本至少和此record的结尾是一样长的，因此任何未来的record将被分配到更高的偏移或者不同的chunk，即使首要副本换人。依据我们的一致性保证，成功的record append操作写入的区域是defined（因此也是一致的），若操作最终失败，则此区域是不一致的（因此undefined的）。我们的应用能处理这种不一致区域（2.7.2讨论过）。

3.4 快照

快照操作能非常快的对一个文件或者一个目录树（称之为源）执行一次拷贝，期间收到的新变异请求也只会受到很小的影响。我们的用户经常使用快照功能快速的为大型的数据集合创建分支拷贝（经常拷贝再拷贝，递归的），或者存档当前状态，以便安全的实验一些变异，随后可以非常简单提交或回滚。

与AFS类似，我们使用标准的copy-on-write技术来实现快照。当master收到一个快照请求，它找出此快照涉及的文件对应的所有chunk，撤回这些chunk上任何未偿还的租赁。这样即可保证随后对这些chunk的写请求将需要一个与master的交互来找到租赁拥有者。master利用此机会暗地里对此chunk创建一个新拷贝。

在撤回租赁完成后，master将此快照操作日志记录到磁盘。实施快照操作时，它会在内存状态中快速复制一份源文件、源目录树的元数据，复制出来的元数据映射到相同的chunk（和JVM中对象的引用计数相似，此chunk的引用计数为2，源元数据和快照元数据两份引用）。

假设快照操作涉及的某个文件包含一个chunk（称之为C），在快照操作后，某个客户端需要写入chunk C，它发送一个请求到master来找到当前租赁持有者。master注意到C的引用计数大于1（源元数据和快照元数据，2个引用）。它不着急给客户端回复，而是选择一个新的chunk句柄（称之为C'），然后要求包含C的副本的chunkserver都为C'创建一个新副本。新老副本在同一个chunkserver，数据都是本地复制，不需要网络传输（磁盘比100Mb的以太网快三倍）。master确认C'的副本都创建完毕后会回复客户端，客户端只是略微感到了一点延迟，随后它会对C及其副本执行正常的写入操作。

4. master操作

所有的命名空间操作都由master执行。而且，它还负责管理所有chunk副本，贯穿整个系统始终：它需要做出布置决策、创建新chunk及其副本，协调控制各种系统级别的活动，比如保持chunk的复制级别、均衡所有chunkserver的负载，以及回收无用存储。下面我们就各个主题展开讨论。

4.1 命名空间管理和锁

很多master操作会花费较长时间：比如一个快照操作需要撤回很多chunkserver的租赁。因此master操作必须能够同时并发的执行以提高效率，但是又要避免它们产生的冲突。为此我们提供了命名空间的区域锁机制，来保证在某些点的串行，避免冲突。

不像传统的文件系统，GFS没有目录的listFiles功能。也不支持文件或者目录的别名（也就是软链接、硬链接、快捷方式）。master中的命名空间逻辑上可以理解为一个lookup table，其中包含完整的路径名到元数据的映射。并且利用前缀压缩提高其效率。命名空间树的每个节点（无论一个绝对文件名或者一个绝对目录名）都有一个对应的读写锁。

每个master操作都会为其牵涉的节点申请读锁或写锁。如果它涉及/d1/d2/./dn/leaf，它将为目录名称为/d1、/d1/d2/、...、/d1/d2/.../dn申请读锁，以及完整路径/d1/d2/.../dn/leaf的读锁。注意leaf可能是文件也可能是目录。

下面举例说明其细节。比如当/home/user/目录正在被快照到/save/user时，我们能利用锁机制防止用户创建一个/home/user/foo的新文件。首先快照操作会为/home和/save申请读锁，以及在/home/user和/save/user申请写锁。创建新文件的请求会申请/home和/home/user的读锁，和/home/user/foo上的写锁。由于在/home/user上的锁冲突，快照和创建新文件操作会串行执行。GFS中的目录比标准文件API要弱化（不支持listFiles等），没有类似的inode信息需要维护，所以在创建、删除文件时不会修改此文件上级目录的结构数据，创建/home/user/foo时也不需要申请父目录/home/user的写锁。上述例子中申请/home/user的读锁可以保护此目录不被删除。

通过命名空间锁可以允许在相同目录发生并发的变化。比如多个文件在同一个目录被并发创建：每个创建会申请此目录的读锁和各自文件的写锁，不会导致冲突。目录的读锁可以保护在创建时此目录不会被删除、重命名或者执行快照。对相同文件的创建请求，由于写锁的保护，也只会导致此文件串行的创建两次。

因为命名空间的节点不少，全量分配读写锁有点浪费资源，所以它们都是lazy分配、用完即删。而且锁申请不是随意的，为了防止死锁，一个操作必须按特定的顺序来申请锁：首先按命名空间树的层级排序，在相同层级再按字典序。

4.2 副本布置

GFS集群是高度分布式的，而且有多个层级（层级是指：机房/机架/服务器这样的层级结构）。通常会在多个机架上部署几百个chunkserver。这些chunkserver可能被各机架的几百个客户端访问。不同机架之间的机器通讯可能跨一个或多个网络交换机。进出一个机架的带宽可能会低于机架内所有机器的总带宽。多级分布式要求我们更加合理的分布数据，以提高可扩展性、可靠性和可用性。

chunk副本的布置策略主要遵循两个目标：最大化数据可靠性和可用性，最大化网络带宽利用。仅仅跨机器的冗余副本是不够的，这仅仅能防御磁盘或者机器故障，也只考虑到单台机器的网络带宽。我们必须跨机架的冗余chunk副本。这能保证系统仍然可用即使整个机架损坏下线（比如网络交换机或者电力故障）。而且能按机架的带宽来分摊读操作的流量。不过这会导致写流量被发往多个机架，这一点牺牲我们可以接受。

4.3 创建、重复制、重负载均衡

chunk副本会在三个情况下被创建：chunk创建、restore、重负载均衡

当master创建一个chunk，它需要选择在哪些chunkserver上布置此chunk的初始化空副本。选择过程主要会考虑几个因素。1 尽量选择那些磁盘空间利用率低于平均值的chunkserver。这样长此以往可以均衡各chunkserver的磁盘使用率。2 我们不希望让某台chunkserver在短时间内创建过多副本。尽管创建本身是廉价的，但它预示着即将来临的大量写流量（客户端请求创建chunk就是为了向其写入），而且据我们观察它还预示着紧随其后的大量读操作。3 上面论述过，我们想要跨机架的为chunk保存副本。

master需要关注chunk的复制级别是否达标（每个chunk是否有足够的有效副本），一旦不达标就要执行restore操作为其补充新副本。很多原因会导致不达标现象：比如某个chunkserver不可用了，某个副本可能腐化了，某个磁盘可能不可用了，或者是用户提高了复制级别。restore时也要按优先级考虑几个因素。第一个因素是chunk低于复制标准的程度，比如有两个chunk，一个缺两份副本、另一个只缺一份，那必须先restore缺两份的。第二，我们会降低已被删除和曾被删除文件对应chunk的优先级。最后，我们会提高可能阻塞客户端进程的chunk的优先级。

master选择高优先级的chunk执行restore时，只需指示某些chunkserver直接从一个已存在的合法副本上拷贝数据并创建新副本。选择哪些chunkserver也是要考虑布置策略的，其和创建时的布置策略类似：尽量均衡的利用磁盘空间、避免在单台chunkserver上创建过多活跃的chunk副本、以及跨机架。restore会导致整个chunk数据在网络上传输多次，为了尽量避免影响，master会限制整个集群以及每台chunkserver上同时执行的restore数量，不会在短时间执行大量的restore。而且每个chunkserver在拷贝源chunkserver的副本时也会采用限流等措施来避免占用过多网络带宽。

重负载均衡是指：master会检查当前的副本分布情况，为了更加均衡的磁盘空间利用率和负载，对必要的副本执行迁移（从负担较重的chunkserver迁移到较轻的）。当新的chunkserver加入集群时也是依靠这个活动来慢慢的填充它，而不是立刻让它接收大量的写流量。master重新布置时不仅会考虑上述的3个标准，还要注意哪些chunkserver的空闲空间较低，优先为其迁移和删除。

4.4 垃圾回收

在一个文件被删除后，GFS不会立刻回收物理存储。它会在懒惰的、延迟的垃圾回收时才执行物理存储的回收。我们发现这个方案让系统更加简单和可靠。

4.4.1 机制

当一个文件被应用删除时，master立刻打印删除操作的日志，然而不会立刻回收资源，仅仅将文件重命名为一个隐藏的名字，包含删除时间戳。在master对文件系统命名空间执行常规扫描时，它会删除任何超过3天的隐藏文件（周期可配）。在那之前此隐藏文件仍然能够被读，而且只需将它重命名回去就能恢复。当隐藏文件被删除时，它才在内存中元数据中被清除，高效的切断它到自己所有chunk的引用。

在另一个针对chunk命名空间的常规扫描中，master会识别出孤儿chunk（也就是那些任何文件都不会引用的chunk），并删除它们的元数据。在与master的心跳消息交换中，每个chunkserver都会报告它的一个chunk子集，master会回复哪些chunk已经不在其元数据中了，chunkserver于是删除这些chunk的副本。

4.4.2 讨论



尽管分布式垃圾回收是一个困难的问题，它需要复杂的解决方案，但是我们的做法却很简单。master的“文件到chunk映射”中记录了对各chunk引用信息。我们也能轻易的识别所有chunk副本：他们是在某台chunkserver上、某个指定的目录下的一个Linux文件。任何master没有登记在册的副本都可以认为是垃圾。

我们的垃圾回收方案主要有三点优势。首先，它保证了可靠性的同时也简化了系统。chunk创建操作可能在一些chunkserver成功了、在另一些失败了，失败的也有可能是创建完副本之后才失败，如果对其重试，就会留下垃圾。副本删除消息也可能丢失，master是否需要严谨的关注每个消息并保证重试？垃圾回收提供了一个统一的可依靠的方式来清理没有任何引用的副本，可以让上述场景少一些顾虑，达到简化系统的目的。其次，垃圾回收的逻辑被合并到master上各种例行的后台活动中，比如命名空间扫描，与chunkserver的握手等。所以它一般都是批处理的，花费也被大家分摊。而且它只在master相对空闲时执行，不影响高峰期master的快速响应。第三，延迟的回收有时可挽救偶然的不可逆的删除（比如误操作）。

在我们的实验中也遇到了延迟回收机制的弊端。当应用重复的创建和删除临时文件时，会产生大量不能被及时回收的垃圾。针对这种情况我们在删除操作时会主动判断此文件是否是首次删除，若不是则主动触发一些回收动作。与复制级别类似，不同的命名空间区域可配置各自的回收策略。

4.5 旧副本侦测

当chunkserver故障，错过对chunk的变异时，它的版本就会变旧。master会为每个chunk维护一个版本号来区分最新的和旧的副本。

每当master授予一个新的租赁给某个chunk，都会增长chunk版本号并通知各副本。master和这些副本都持久化记录新版本号。这些都是在写操作被处理之前就完成了。如果某个副本当前不可用，它的chunk版本号不会被更新。master可以侦测到此chunkserver有旧的副本，因为chunkserver重启时会汇报它的chunk及其版本号信息。如果master看到一个比自己记录的还要高的版本号，它会认为自己在授予租赁时发生了故障，继而认为更高的版本才是最新的。

master会在常规垃圾回收活动时删除旧副本。在那之前，它只需保证回复给客户端的信息中不包含旧副本。不仅如此，master会在各种与客户端、与chunkserver的其他交互中都附带带上版本号信息，尽可能避免任何操作、活动访问到旧的副本。

5 故障容忍和诊断

我们最大挑战之一是频繁的组件故障。GFS集群中组件的质量（机器质量较低）和数量（机器数量很多）使得这些问题更加普遍：我们不能完全的信赖机器，也不能完全信赖磁盘。组件故障能导致系统不可用甚至是腐化的数据。下面讨论我们如何应对这些挑战，以及我们构建的帮助诊断问题的工具。

5.1 高可用性

GFS集群中有几百台机器，任何机器任何时间都可能不可用。我们保持整体系统高度可用，只用两个简单但是高效的策略：快速恢复和复制。

5.1.1 快速恢复

master和chunkserver都可以在几秒内重启并恢复它们的状态。恢复的时间非常短，甚至只会影响到那些正在执行中的未能回复的请求，客户端很快就能重连到已恢复的服务器。

5.1.2 chunk复制

早先讨论过，每个chunk会复制到多个机架的chunkserver上。用户能为不同的命名空间区域指定不同的复制级别。默认是3份。master需要保持每个chunk是按复制级别完全复制的，当chunkserver下线、侦测到腐化副本时master都要补充新副本。尽管复制机制运行的挺好，我们仍然在开发其他创新的跨服务器冗余方案。

5.1.3 master复制

master保存的元数据状态尤其重要，它必须被冗余复制。其操作日志和存档会被复制到多台机器。只有当元数据操作的日志已经成功flush到本地磁盘和所有master副本上才会认为其成功。所有的元数据变化都必须由master负责执行，包括垃圾回收之类的后台活动。master故障时，它几乎能在一瞬间完成重启。如果它的机器或磁盘故障，GFS之外的监控设施会在另一台冗余机器上启用一个新master进程（此机器保存了全量的操作日志和存档）。客户端是通过canonical域名（比如gfs-test）来访问master的，这是一个DNS别名，对其做些手脚就能将客户端引导到新master。

此外我们还提供了阴影master，它能在master宕机时提供只读访问。他们是阴影，而不是完全镜像，阴影会比主master状态落后一秒左右。如果文件不是正在发生改变，或者应用不介意拿到有点旧的结果，阴影确实增强了系统的可用性。而且应用不会读取到旧的文件内容，因为文件内容是从chunkserver上读取的，最多只会从阴影读到旧的文件元数据，比如目录内容或者访问控制信息。

阴影master会持续的读取某个master副本的操作日志，并重放到自己的内存中数据结构。和主master一样，它也是在启动时拉取chunkserver上的chunk位置等信息（不频繁），也会频繁与chunkserver交换握手消息以监控它们的状态。仅仅在master决定创建或删除某个master副本时才需要和阴影交互（阴影需要从它的副本里抓日志重放）。

5.2 数据完整性

每个chunkserver使用checksum来侦测腐化的存储数据。一个GFS集群经常包含几百台服务器、几千个磁盘，磁盘故障导致数据腐化或丢失是常有的事儿。我们能利用其他正常的chunk副本恢复腐化的数据，但是通过跨chunkserver对比副本之间的数据来侦测腐化是不切实际的。另外，各副本内的字节数据出现差异也是正常的、合法的（原子的record append就可能造成这种情况，不会保证完全一致的副本，但是不影响客户端使用）。因此，每个chunkserver必须靠自己来核实数据完整性，其对策就是维护checksum。

一个chunk被分解为多个64KB的块。每个块有对应32位的checksum。像其他元数据一样，checksum被保存在内存中，并用利用日志持久化保存，与用户数据是隔离的。

在读操作中，chunkserver会先核查读取区域涉及的数据块的checksum。因此chunkserver不会传播腐化数据到客户端（无论是用户客户端还是其他chunkserver）。如果一个块不匹配checksum，chunkserver向请求者明确返回错误。请求者收到此错误后，将向其他副本重试读请求，而master则会尽快从其他正常副本克隆数据创建新的chunk。当新克隆的副本准备就绪，master命令发生错误的chunkserver删除异常副本。

checksum对读性能影响不大。因为大部分读只会跨几个块，checksum的数据量不大。GFS客户端代码在读操作中可以尽量避免跨越块的边界，进一步降低checksum的花费。而且chunkserver查找和对比checksum是不需要任何I/O的，checksum的计算通常也在I/O等待时被完成，不抢占CPU资源。

checksum的计算是为append操作高度优化的，因为append是我们的主要应用场景。append时可能会修改最后的块、也可能新增块。对于修改的块只需增量更新其checksum，对于新增块不管它有没有被填满都可以计算其当前的

checksum。对于最后修改的块，即使它已经腐化了而且append时没有检测到，还对其checksum执行了增量更新，此块的checksum匹配依然会失败，在下次被读取时即能侦测到。

普通的写操作则比append复杂，它会覆盖重写文件的某个区域，需要在写之前检查区域首尾块的checksum。它不会创建新的块，只会修改老的块，而且不是增量更新。对于首尾之间的块没有关系，反正是被全量的覆盖。而首尾块可能只被覆盖了一部分，又不能增量更新，只能重新计算整个块的checksum，覆盖老checksum，此时如果首尾块已经腐化，就无法被识别了。所以必须先检测后写。

在系统较空闲时，chunkserver会去扫描和检查不太活跃的chunk。这样那些很少被读的chunk也能被侦测到。一旦腐化被侦测到，master会为其创建一个新副本，并删除腐化副本。GFS必须保证每个chunk都有足够的有效副本以防不可逆的丢失，chunk不活跃可能会导致GFS无法察觉它的副本异常，此机制可以有效的避免这个风险。

5.3 诊断工具

大量详细的诊断日志对于问题隔离、调试、和性能分析都能提供无法估量的价值，打印日志却只需要非常小的花费。如果没有日志，我们永远捉摸不透那些短暂的、不可重现的机器间交互。GFS服务器生成的诊断日志存储了很多重要的事件（比如chunkserver的启动和关闭）以及所有RPC请求和回复。这些诊断日志能被自由的删除而不影响系统正确性。然而我们会尽一切可能尽量保存这些有价值的日志。

RPC日志包含了在线上每时每刻发生的请求和回复，除了读写的真实文件数据。通过在不同机器之间匹配请求和回复、整理RPC记录，我们能重现整个交互历史，以便诊断问题。日志也能服务于负载测试和性能分析的追踪。

日志造成的性能影响很小（与收益相比微不足道），可以用异步缓冲等各种手段优化。有些场景会将大部分最近的事件日志保存在机器内存中以供更严格的在线监控。

【扩展阅读】

“GFS.....也支持小文件，但是不需要着重优化”，这是论文中的一句原话，初读此文时还很纳闷，GFS不是据说解决了海量小文件存储的难题吗，为何前后矛盾呢？逐渐深读才发现这只是个小误会。下面译者尝试在各个视角将GFS、TFS、Haystack进行对比分析，读者可结合[前文](#)基础，了解个中究竟。

1. 愿景和目标

GFS的目标可以一言以蔽之：给用户一个无限容量、放心使用的硬盘，快速的存取文件。它并没有把自己定位成某种特定场景的文件存储解决方案，比如小文件存储或图片存储，而是提供了标准的文件系统API，让用户像使用本地文件系统一样去使用它。这与Haystack、TFS有所不同，GFS的目标更加通用、更加针对底层，它的编程界面也更加标准化。

比如用户在使用Haystack存取图片时，可想而知，编程界面中肯定会有类似create(photo)、read(photo_id)这样的接口供用户使用。而GFS给用户提供的接口则更类似File、FileInputStream、FileOutputStream（以Java语言举例）这样的标准文件系统接口。对比可见，Haystack的接口更加高层、更加抽象，更加贴近于应用，但可能只适合某些定制化的应用场景；GFS的接口则更加底层、更加通用，标准文件系统能支持的它都能支持。举个例子，有一万张图片，每张100KB左右，用Haystack、GFS存储都可以，用Haystack更方便，直接有create(photo)这样的接口可以用，调用即可；用GFS就比较麻烦，你需要自己考虑是存成一万张小文件还是组装为一些大文件、按什么格式组装、要不要压缩.....GFS不去管这些，你给它什么它就存什么。假如把一万张图片换成一部1GB的高清视频AVI文件，总大小差不多，一样可以放心使用GFS来存储，但是Haystack可能就望而却步了（难道把一部电影拆散放入它的一个个needle？）。

这也回答了刚刚提到的那个小误会，GFS并不是缺乏对小文件的优化和支持，而是它压根就没有把自己定位成小文件存储系统，它是通用的标准文件系统，它解决的是可靠性、可扩展性、存取性能等后顾之忧，至于你是用它来处理大文件、存储增量数据、打造一个NoSQL、还是解决海量小文件，那不是它担心的问题。只是说它这个文件系统和标准文件系统一样，也不喜欢数量太多的小文件，它也建议用户能够将数据合理的组织安排，放入有结构有格式的大文件中，而不要将粒度很细的一条条小数据保存为海量的小文件。相反，Haystack和TFS则更注重实用性，更贴近应用场景，并各自做了很多精细化的定制优化。在通用性和定制化之间如何抉择，前文2.3中Haystack的架构师也纠结过，但是可以肯定的是，基于GFS，一样可以设计needle结构、打造出Haystack。

2 存储数据结构

这里的数据结构仅针对真实文件内容所涉及的存储数据结构。三者在这种数据结构上有些明显的相同点：


首先，都有一个明确的逻辑存储单元。在GFS中就是一个Chunk，在Haystack、TFS中分别是逻辑卷和Block。三者都是靠各自的大量逻辑存储单元组成了一个庞大的文件系统。设计逻辑存储单元的理由很简单——保护真正的物理存储结构，不被用户左右。用户给的数据太小，那就将多个用户数据组装进一个逻辑存储单元（Haystack将多个图片作为needle组装到一个逻辑卷中）；用户给的数据太大，那就拆分成多个逻辑存储单元（GFS将大型文件拆分为多个

chunk)。总之就是不管来者是大还是小，都要转换为适应本系统的物理存储格式，而不按照用户给的格式。一个分布式文件系统想要保证自身的性能，它首先要保证自己能基于真实的物理文件系统打造出合格的性能指标（比如GFS对chunk size=64MB的深入考究），在普通Linux文件系统上，固定大小的文件+预分配空间+合理的文件总数量+合理的目录结构等等，往往是保证I/O性能的常用方案。所以必须有个明确的逻辑存储单元。

另一个很明显的相同点：逻辑存储单元和物理机器的多对多关系。在GFS中一个逻辑存储单元Chunk对应多个Chunk副本，副本分布在多个物理存储chunkserver上，一个chunkserver为多个chunk保存副本（所以chunk和chunkserver是多对多关系）。Haystack中的逻辑卷与Store机器、TFS中的Block与DataServer都有这样的关系。这种多对多的关系很好理解，一个物理存储机器当然要保存多个逻辑存储单元，而一个逻辑存储单元对应多个物理机器是为了冗余备份。

三者在数据结构上也有明显的区别，主要是其编程界面的差异导致的。比如在Haystack、TFS的存储结构中明确有needle这种概念，但是GFS却不见其踪影。这是因为Haystack、TFS是为小文件定制的，小文件是它们的存储粒度，是用户视角下的存储单元。比如在Haystack中，需要考虑needle在逻辑卷中如何组织检索等问题，逻辑卷和needle是一对多的关系，一个逻辑卷下有多个needle，某个needle属于一个逻辑卷。这些关系GFS都不会去考虑，它留给用户自行解决（上面愿景和目标里讨论过了）。

3 架构组件角色

[Haystack一文](#)中的对比已经看到分布式文件系统常用的架构范式就是“元数据总控+分布式协调调度+分区存储”。在Haystack中Directory掌管所有应用元数据、为客户端指引目标（指引到合适的目标Store机器做合适的读写操作，指引过程就是协调调度过程），单个Store机器则负责处理好自身的物理存储、本地数据读写；TFS中NameServer控制所有应用元数据、指引客户端，DataServer负责物理存储结构、本地数据读写。可以看出这个范式里的两个角色——协调组件、存储组件。协调组件负责了元数据总控+分布式协调调度，各存储组件作为一个分区，负责实际的存储结构和本地数据读写。架构上的相同点显而易见——GFS也满足此范式，作为协调组件的master、和作为存储组件的chunkserver。在组件角色的一些关键设计上，三者也曾面临了一些相似的技术难题，比如它们都竭力减少对协调组件的依赖、减少其交互次数，不希望给它造成过大压力。协调组件为啥显得这么脆弱、这么缺乏安全感呢？从两篇论文都可以看出，GFS和Haystack的作者非常希望协调组件能够简，其原因很简单——人如果有两个大脑那很多事会很麻烦。协调组件可认为是整个系统的大脑，它不停的派发命令、指点迷津；如果大脑有两个，那客户端听谁的、两个大脑信息是否需要同步、两个大脑在指定策略时是否有资源竞争.....就跟人一样，这种核心总控的大脑有多个会带来很多复杂度、一致性问题，难以承受。即使各平台都有备用协调组件，比如GFS的阴影master，但都只是作为容灾备用，不会在线参与协调。

4 元数据

虽然Haystack的Directory、TFS的NameServer、GFS的master干的是同样的活，但是其在元数据问题上也有值得讨论的差异。上篇文章中曾重点介绍了译者对于全量缓存应用元数据的疑惑（Haystack全量缓存所有图片的应用元数据，而TFS用了巧妙的命名规则），难道GFS不会遇到这个难题吗？而且此篇文章还反复强调了GFS的master是单例的、纯内存的，难道它真的不会遭遇单点瓶颈吗？答案还是同一个：编程界面。GFS的编程界面决定了即使它整个集群存了几百TB的数据，也不会有太多的文件。对于Haystack和TFS，它们面对的是billions的图片文件，对于GFS，它可能面对的仅仅是一个超巨型文件，此文件里有billions的图片数据。也就是说Haystack和TFS要保存billions的应用元数据，而GFS只需保存一个。那GFS把工作量丢给谁了？对，丢给用户了。所以GFS虽然很伟大、很通用、愿景很酷，但是对特定应用场景的支持不一定友好，这也是Haystack最终自行定制的原因之一吧。

同时，这也是GFS敢于设计单点、内存型master的原因，它认为一个集群内的文件数量不会超过单点master的能力上限。同样的，GFS也没有着重介绍文件系统元数据的方案（Haystack文章中反复强调了文件系统元数据牵扯的I/O问题），原因也是一样：文件系统元数据的I/O问题对于Haystack来说很难缠是因为Haystack要面对海量小文件，而GFS并不需要。有了这样的前提，GFS的master确实可以解脱出来，做更多针对底层、面向伟大愿景的努力。比如可以轻松地在单机内存中全量扫描所有元数据、执行各种策略（如果master受元数据所累、有性能压力、需要多个master分布式协作，那执行这种全局策略就会很麻烦了，各种一致性问题会扑面而来）。

不过GFS中会多出一种Haystack和TFS都没有的元数据——文件命名空间。当你把GFS当做本地磁盘一样使用时，你需要考虑文件的目录结构，一个新文件产生时是放到一个已有目录还是创建新目录、创建新文件时会不会有另一个请求正在删除父目录.....这些内容是GFS特有的（Haystack和TFS的用户不需要面对这些内容），所以它考虑了master的命名空间锁、操作日志顺序等机制，来保护命名空间的更新。

5 控制流、数据流（以及一致性模型）

GFS、Haystack、TFS的控制流大致相同，其思路不外乎：1 客户端需要发起一次新请求；2 客户端询问协调组件，自己应该访问哪个存储组件；3 协调组件分析元数据，给出答复；4 客户端拿到答复直接访问指定的存储组件，提交请求；5 各存储组件执行请求，返回结果。

但是细节上各自有差异，上篇文章提到Haystack是对等结构，客户端直接面对各对等存储组件（比如写入时看到所有物理卷、分别向其写入），而TFS是主备结构，客户端只面对主DataServer，主向备同步数据.....GFS在这方面最大的差异在于：

5.1 租赁机制。回想Haystack，客户端直接将写请求提给各个Store机器即可解决问题，GFS也是对等设计，为何要捣鼓出令人费解的租赁机制？不能直接由客户端分别写入各个对等chunkserver吗？这个问题已经在一致性模型章节讨论过，当时提到了租赁机制是为了保证各个副本按相同顺序串行变异，那我们也可以反过来问一句：Haystack、TFS里为啥没有遇到这等问题？是它们对一致性考虑太少了吗？这个答案如果要追本溯源，还是要牵扯到编程界面的问题：Haystack、TFS中用户面对的是一个一个小图片，用户将图片存进去、拿到一个ID，将来只要保证他拿着此ID依然能找到对应图片就行，即使各个副本执行存储时顺序有差别，也丝毫不影响用户使用（比如Haystack收到3个并发图片A、B、C的存储请求，提交到3台Store机器，分别存成了ABC、ACB、BAC三种不同的顺序，导致副本内容不一致，但是当用户无论拿着A、B、C哪个图片ID来查询、无论查的是哪个Store机器，都能检索到正确的图片，没有问题；Haystack和TFS的编程界面封装的更高级，GFS在底层遭遇的难题影响不到它们）。而在GFS比较底层的编程界面中，用户面对的是众多图片组装而成的一整个文件，如果GFS在不同副本里存储的文件内容不一致，那就会影响用户的检索逻辑，那摊上大事儿了。另外，Haystack、TFS只支持小文件的append和remove，而GFS怀抱着伟大的愿景，它需要支持对文件的随机写，只有保证顺序才能避免同个文件区域并发随机写导致的undefined碎片问题。所以GFS花心思设计租赁机制也就合情合理了。

谈到串行就不得不考虑一下性能问题，串行绝不是大规模并发系统的目标，而只是一种妥协——对于多核操作系统下的I/O密集型应用，当然是越并行越有益于性能。比如有10个并发的append请求，是逐个依次执行，每一个都等待上一个I/O处理完成吗？这样每个请求的I/O Wait时间cpu不就空闲浪费吗？根据译者的推测，GFS所谓的串行仅仅应该是理论上的串行（即执行效果符合串行效果），但真正执行时并不会采用加锁、同步互斥、按顺序单线程依次执行等性能低劣的方案。以10个并发append为例，理论上只需要一个AtomicLong对象，保存了文件长度，10个append线程可并发调用其addAndGet()，得到各自不冲突的合法偏移位置，继而并发执行I/O写入操作，互不干扰。再比如对同个文件的随机偏移写，也不是要全部串行，只有在影响了同一块文件区域时才需要串行，因此可以减小串行粒度，影响不同区域的请求可并行写入。通过这样的技巧，再结合在首要chunkserver上分配的唯一序号，GFS应该可以实现高性能的串行效果，尤其是append操作。



TFS和Haystack的文章之所以没有过多提及一致性问题，并不是因为它们不保证，而是在它们的编程界面下（面向小文件的append-only写），一致性问题不大，没有什么难缠的陷阱。

5.2 数据流分离。Haystack没法在数据流上做什么文章，它的客户端需要分别写入各个Store机器；TFS可以利用其主备机制，合理安排master-slave的拓扑位置以优化网络负载。而GFS则是完全将真实文件的数据流和控制流解耦分离，在网络拓扑、最短路径、最小化链式传输上做足了文章。系统规模到一定程度时，机房、机架网络拓扑结构对于整体性能的影响是不容忽略的，GFS在这种底层机制上考虑的确实更加到位。

6 可扩展性和容错性

在上篇文章中已经详尽的讨论了Haystack、TFS是如何实现了优雅的高可扩展性。对于元数据不成难题的GFS，当然也具备同样的实力，其原理就不重复叙述了。值得一提的是，GFS从元数据的负担中解放出来后，它充分利用了自身的优势，实现了各种全局策略、系统活动，极大的提高了系统的可扩展性及附属能力，比如restore、重负载均衡等等。而且其并不满足于简单的机器层面，还非常深入的考虑了网络拓扑、机架布置.....GFS这些方面确实要领先于同类产品。

容错性的对比同样在上篇文章中详细描述过，在结构方面GFS的chunkserver也是对等结构（控制流的首次之分与容错无关），其效果与Haystack的容错机制类似，无论是机器故障还是机房故障。相对来说，GFS在故障的恢复、侦测、版本问题、心跳机制、协调组件主备等环节介绍的更加细致，这些细节Haystack没有怎么提及。值得一提的差异是GFS在数据完整性上的深入考究，其checksum机制可有效的避免腐化的数据，这一点Haystack、TFS没有怎么提及。

7 删除和修改

很多时候我们会重点关注一个系统是否能删除和修改数据，这是因为当今越来越多架构设计为了追求更高的主流可用性，而牺牲了其他次要的特性。比如你发表一篇微博时一瞬间就提交成功，速度快、用户体验非常好；结果你发现不小心带了一句“大概八点二十分”，傻眼了，到处找不着微博修改的按钮；于是你只能删除老微博，再发一条新的，老微博的评论、转发都丢失了。当你不承认自己发过八点二十分的微博时，网友拿出了截图，你说是PS的，新浪在旁冷笑，你那条八点二十分的微博一直在原处，从未被删除，估计要过好几个小时之后才会真正从磁盘删掉。

在学习Haystack、TFS、GFS的过程中我们可以看出大家都有这种倾向，原因可能有很多，其中有两个是比较明显的。首先，对于存储的数据结构，增加新数据造成的破坏较小，而删除、修改造成的破坏较大。新增只是往末尾附加一些新data，而不会影响已有的data；删除则导致已有data中空出了一块区域，这块区域不能就这么空放着，最低劣的做法是直接将有data进行大规模位移来填补狭缝，比较婉转的做法是先不着急等不忙的时候做一次碎片整理；修改会导致某条数据size发生变化，size变小会导致狭缝碎片，size变大则空间不足，要挤占后面数据的位置。其次，新增数据可

以做到无竞争（刚才5.1讨论了GFS的原子append，并发的新增操作影响的是不同的文件区域，互不干扰），而删除和修改则很难（并发的删除和修改操作可能影响相同的文件区域，这就必须有条件竞争）。有这两个原因存在，大家都偏爱append、不直接实现修改和删除，就不难理解了。

不过GFS依然支持直接的修改功能——随机偏移写。这并不是因为GFS的架构设计更强大，而是因为它不需要承担某些责任。比如在Haystack和TFS中，它们需要承担图片在真实文件中的存储格式、检索等责任，当将一个图片从100KB修改为101KB时，对存储格式的破坏是难以承受的，所以它们不支持这种直接的修改，只能采用删除+新增来模拟修改。而GFS并没有维护一个文件内部格式的责任，还是那句话，你交给它什么它就存什么。所以用户会保护自己的文件内部格式，他说可以写那就可以写，GFS并没有什么难题，只需解决好一致性、defined、统一变异顺序等问题即可。

同样，GFS的删除与Haystack、TFS的删除，其意义也不同。GFS的删除指的是整个大文件的删除。Haystack和TFS删除的是用户视角下的一条数据，比如一张图片，它是逻辑存储单元中的一个entry而已。而GFS的删除则是用户视角下的一个文件，一个文件就对应了多个逻辑存储单元（chunk），里面也包含了海量的应用实体（比如图片）。在这种差异的背景下，他们面临的难题也完全不同。Haystack和TFS面临的的就是刚才提到的“删除会破坏存储文件已有数据的格式、造成狭缝碎片”问题，它们的对策就是懒惰软删除、闲了再整理。而GFS则不会遭遇此难题，用户在文件内部删除几条数据对于GFS来说和随机偏移写没啥区别，狭缝碎片也是用户自己解决。GFS需要解决的是整个文件被删除，遗留下了大量的chunk，如何回收的问题。相比Haystack和TFS，GFS的垃圾回收其实更加轻松，因为它要回收的是一个逻辑存储单元（chunk），一个chunk（副本）其实就是一个真实的Linux文件，调用file.delete()删了就等于回收了，而不需要担心文件内部那些精细的组织格式、空间碎片。

8 其他细节和总结

GFS还是有很多与众不同的技巧，比如合理有效的利用操作日志+存档来实现元数据持久化存储；比如细粒度、有条不紊的命名空间锁机制；便捷的快照功能等等。从整体风格来说，GFS偏爱底层上的深入考究，追求标准化通用化的理想，它希望别人把它当成一个普通的文件系统，无需华丽的产品包装，而只是务实的搭建好底层高可靠、高可用、高可扩展的基础。但是美好的不一定是合适的，通用的不一定是最佳的，Haystack、TFS在追求各自目标的道路上一样风雨无阻披荆斩棘。这两篇论文的翻译和对比，只希望能将巨头的架构师们纠结权衡的分岔路口摆到读者面前，一起感同身受，知其痛，理解其抉择背后的意义。



英文原文：[GFS](#)，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/3491.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 >>](#)



可能感兴趣的文章

- [为集群配置Impala和Mapreduce](#)
- [Java 10大优点—Part1—Java编译器](#)
- [追求代码质量（9）：用JUnitPerf进行性能测试](#)
- [设计模式六大原则（6）：开闭原则](#)
- [23种设计模式（6）：模版方法模式](#)
- [Apache HBase 入门教程](#)
- [JVM内存管理—GC算法精解（五分钟让你彻底明白标记/清除算法）](#)
- [通向架构师的道路（第二十天）万能框架spring\(二\)maven结合spring与ibatis](#)
- [numfmt：让数字变得更容易理解](#)
- [kafka 源码分析 4：broker 处理生产请求](#)

3 条评论

1. [刘明森](#) 说道 :
[2013/03/29 下午 4:32](#)

好文，必须得顶一下

 0  0

2. [fans](#) 说道 :
[2013/05/24 下午 5:05](#)

博主辛苦了。

 0  0

3. [chuan](#) 说道 :
[2014/02/06 下午 8:54](#)

批的精彩

 0  0

[« HBase数据迁移 \(1 \)](#)

[Hadoop入门教程\(四\):MR作业的提交监控、输入输出控制及特性使用 »](#)

Search for:



- [本周热门文章](#)
- [本月热门](#)
- [热门标签](#)

0 [Spring Boot 自动配置的 "魔法..."](#)

1 [JDK 源码阅读 : FileDescript...](#)

2 [linux 如何更改网卡 MAC 地址](#)



最新评论

-  Re: [Service Mesh 及其主流开源...](#)
赞同最后的观点。 [Alan](#)
-  Re: [Spring Boot 自动配置的 "...](#)

简单的说自动配置是根据约定来的，也就是说springboot要求必须按照他给的规则开发starter...

www.wuliaokankan.cn



Re: [HashMap 和 Hashtable 到...](#)

简单易懂 王新引



Re: [40个Java多线程问题总结](#)

兄弟，查看线程栈是jps 查看idtop -H -p pid 查看线程jstack pid 查看里面... 茅山道士



Re: [把Java数组转换为List时的注意...](#)

感谢，一直被int[]无法转换成list困扰。 [wingsico](#)



Re: [JVM \(5 \) : Tomcat 性能调优和性...](#)

下面没图了 mmma



Re: [深入理解Java之线程池](#)

非常感谢，写得很详细，很容易理解 叶彪



Re: [程序员你为什么这么累【续】：编码习惯之...](#)

阔然开朗！很赞同一句：先有思想再有技术。程序员可以忙可以累，但是此时此刻的忙跟累，一定要为了日后的轻... tony.chenjy



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.....



联系我们

Email : ImportNew.com@gmail.com

新浪微博 : [@ImportNew](#)

推荐微信号



ImportNew 数据分析与开发 算法爱好者

反馈建议 : ImportNew.com@gmail.com

广告与商务合作QQ : 2302462408

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

[设计](#) – UI,网页，交互和用户体验

[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

© 2018 ImportNew

