计算机组织与体系结构 Project
期末报告

大数据下基于 HDFS 的
吞吐频率与存储结构优化

学生姓名：　周喆、张闻涛、孙志玉

所在院系：　信息科学技术学院

日　　期：　二〇一六年一月

大数据下基于 HDFS 的吞吐频率与存储结构优化

信息科学技术学院 周喆(1300012752)、张闻涛(1300012758)、孙志玉(1300012840)

摘要：管理跨网络机器存储的文件系统叫做分布式文件系统。随着多节点的引入，相应的问题也就出现了，例如其中最重要的一个问题就是如何保证在某个节点失败的情况下数据不会丢失。Hadoop 中有一个核心子项目 HDFS 就是用来管理集群的存储问题的。由于 HDFS 是为高数据吞吐量应用而设计的，必然以高延迟为代价并且当存储大量小文件则会受到更为严重的影响，本文研究并实现了一个基于 HDFS 与 LFS 的缓冲存储系统，以进一步对其吞吐频率与存储结构进行优化。

关键词：分布式文件系统 HDFS 存储结构 吞吐频率 LFS

一、 绪论

由于服务器具有远超过普通用户的计算能力和存储能力，“云计算”的模式日趋普及。为了让服务器更好的提供存储服务和其他各种服务，越来越多的应用依赖于分布式存储系统。例如，并行计算的高性能应用，需要服务器之间的协同合作与交流；以及我们接下来所优化的文件存储系统 HDFS，需要额外的记录或备份以实现高容错率。

我们的 Log File System 基于 Hadoop 实现的一个分布式文件系统（Hadoop Distributed File System），简称 HDFS。Hadoop 框架最核心的设计就是：HDFS 和 MapReduce。HDFS 为海量的数据提供了存储，而 MapReduce 则为海量的数据提供了计算。HDFS 拥有高容错性，并且设计用来部署在低廉的硬件上；而且它能提供很高的吞吐量来访问应用程序的数据，适合那些有着超大数据集的应用程序。HDFS 放宽了 POSIX 的要求，可以以流的形式访问文件系统中的数据。当然，为了实现这些优美的特性，HDFS 不得不在延迟方面做出一些牺牲。

HDFS 产生高延迟，一是因为数据通过网络传输，本身就经过了各个层之间的多次封装、拆包，以及线路上的不小的延迟，二是因为对于上传的数据，HDFS 拥有特殊的存储策略：大文件的存储会被分割为多个 block 进行存储；每一个 block 会在多个 datanode 上存储多份副本；另外还需要 namenode 负责存储文件的元信息，这种设计使得数据传输可以并行进行，提高了带宽，同时也需要一些额外的处理时间。针对这个特点，我们的文件系统 LFS 对 HDFS 的传输方式进行了上层的包装。LFS 考虑用户平时使用 HDFS 的习惯，并且利用本地存储对于网络存储的巨大延迟优势，采用了类似缓存的做法。另外，使用 LOG 存储的方式，提供方便的接口，不仅能够实现更加细致的操作，而且能在发生意外时使用户从崩溃中立即恢复，提高了实用性和可靠性。

二、 系统设计与机制

1. Log 式存储系统：

在大多数文件系统中，写操作往往都是零碎的。而这是极其没有效率的，例如一个 50μs 的磁盘写操作前通常需要 10ms 的寻道时间和 4ms 的旋转延迟时间，而这在 HDFS 中则更为严重。于是，Log 式的磁盘存储系统实现了将磁盘结构化为一个日志，需要的时候将被缓冲

在内存中的所欲未决的写操作都放到一个单独的段中，作为日志末尾的一个邻接段写入磁盘。我们对 HDFS 修改的灵感正来自于此。

我们的 Log File System 将用户对 HDFS 的写操作结构化为日志，比如说，每次在文件尾添加一段内容，存储系统将这个行为记录为一次操作，这样一次一次的操作首先保存在本地空间中，在有需要的时候将文件内容做出真正的修改。被缓冲在内存中的所欲未决的写操作都放到一个单独的段中，作为日志末尾的一个邻接段写入磁盘。

为了保持本地不会被占用大量的空间以及正常的数据更新，要对 LOG 存储文件进行周期性地扫描，看有没有达到一定大小的文件，由于它们进行上传时延迟产生的代价占比并不大，所以可以进行上传，并且对相应的本地 LOG 文件进行操作。

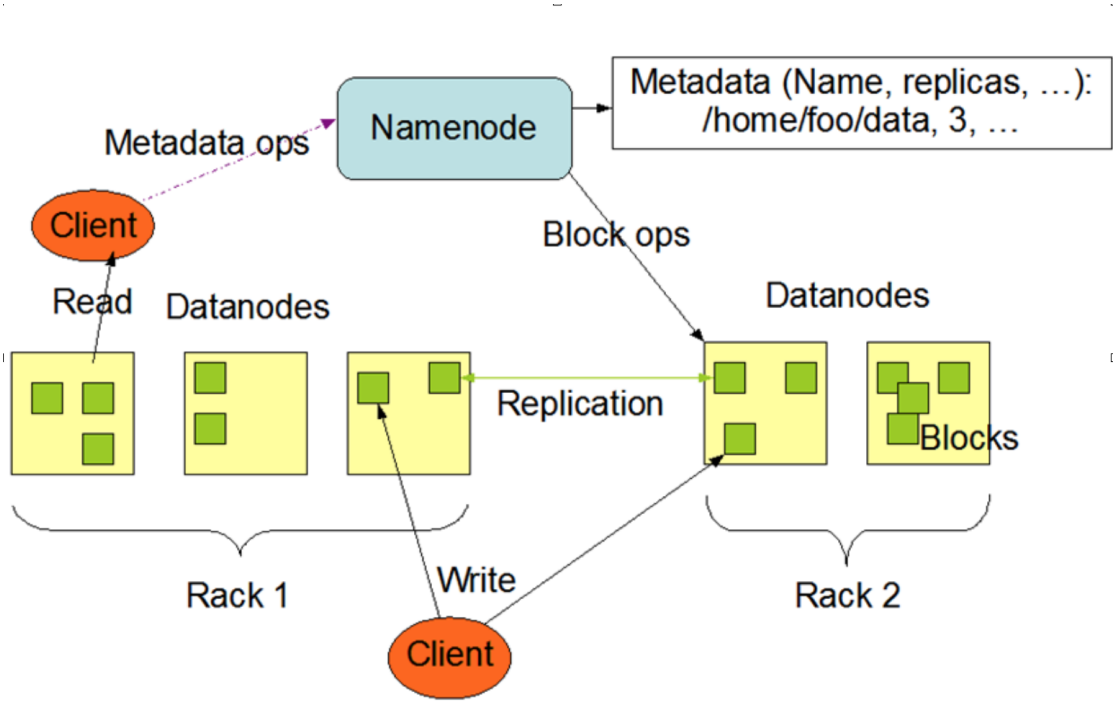
2. 崩溃恢复和错误修改：

由于使用了 LOG 式存储系统，借助其可以将因网络崩溃或者进程崩溃等情况带来的损失降到最低：上传大文件的行为失败时，本地的 LOG 条目仍然存在，用户可以查看条目来了解错误信息，也可以主动调用存储系统的接口，使之再次生成要上传的内容进行上传。另外，用户可以很方便地撤销某一次操作。

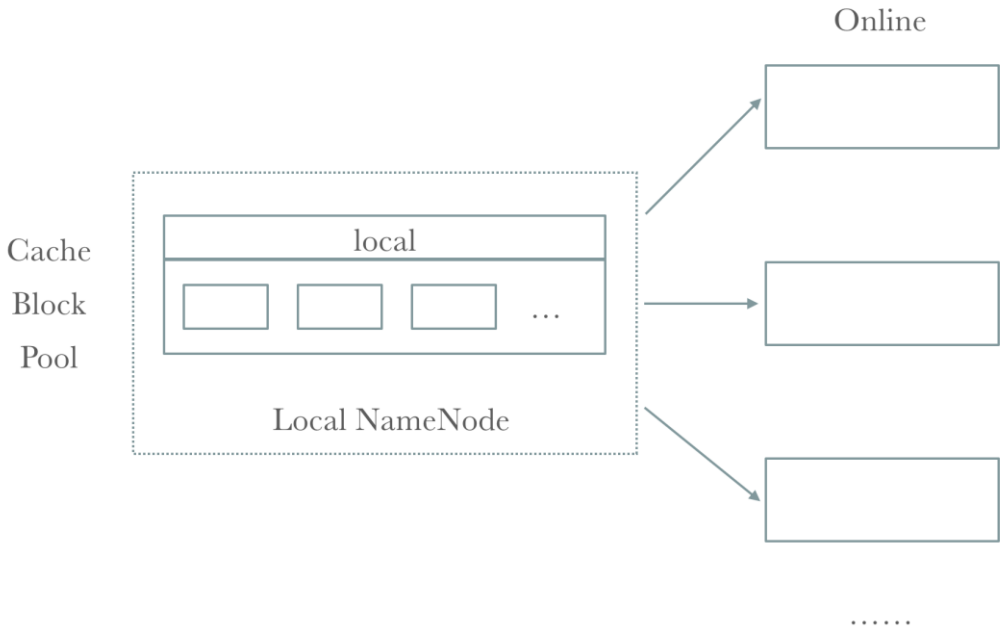
三、 模块实现

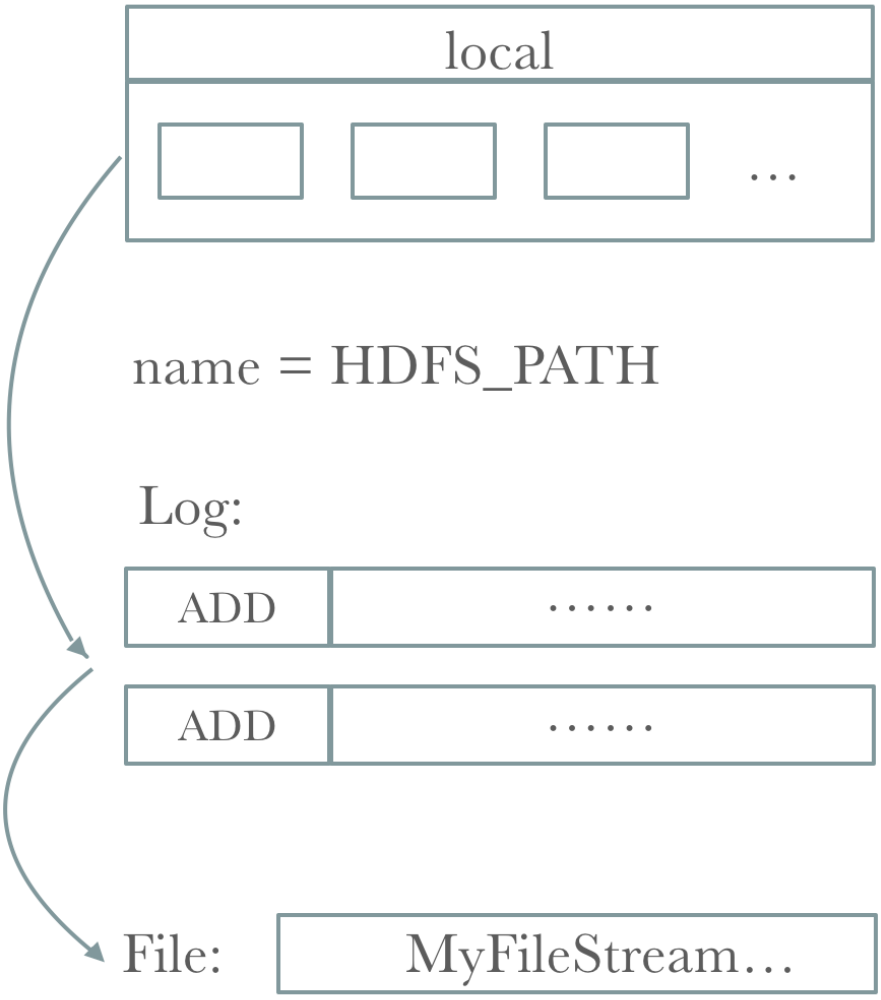
1. 总体结构（图）：

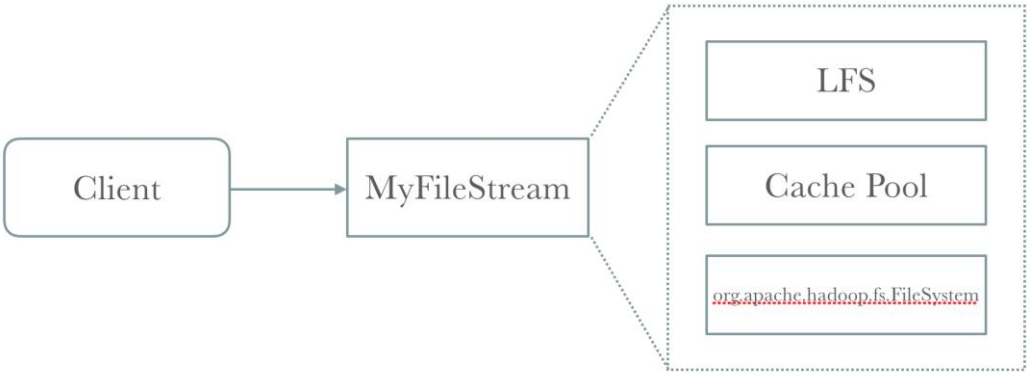
HDFS：



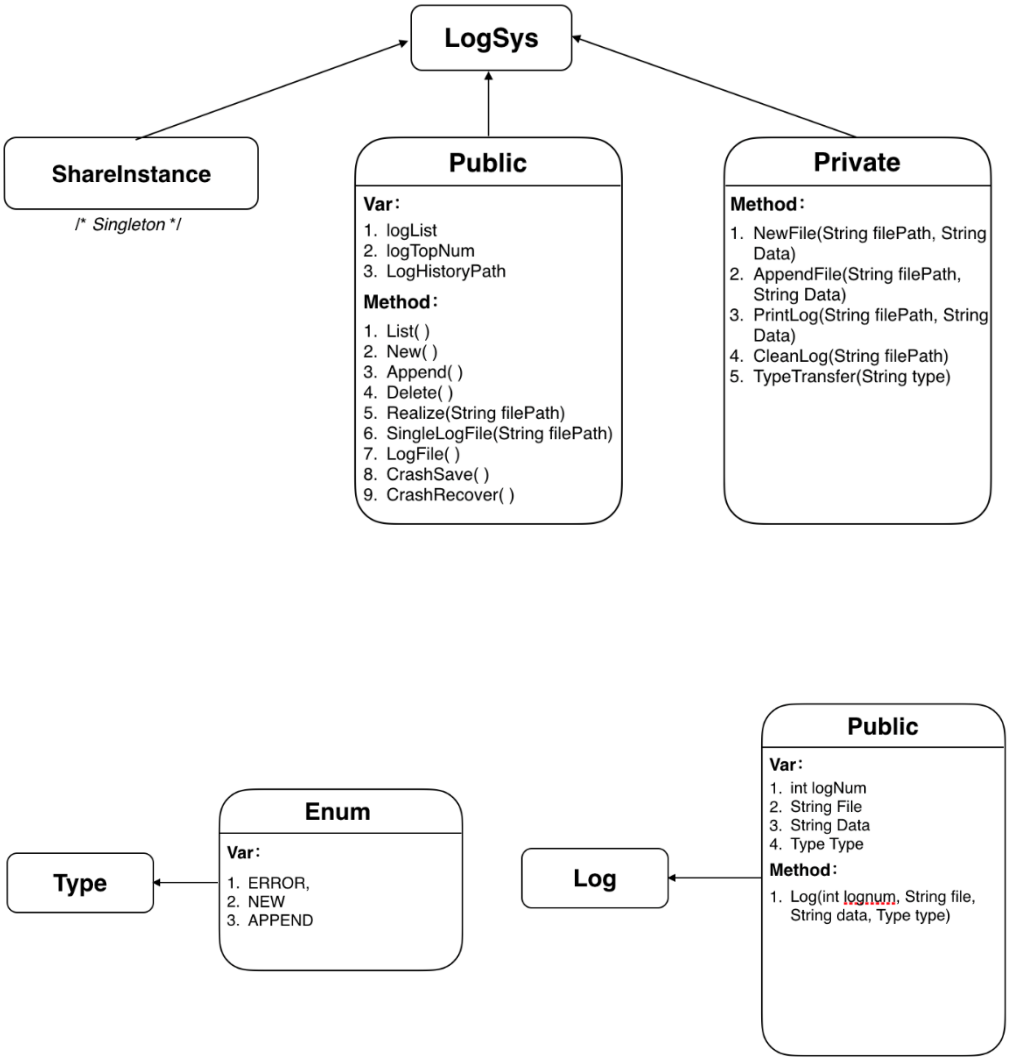
缓冲系统结构：


Online
Cache
Block
Pool
local
...
Local NameNode
.....

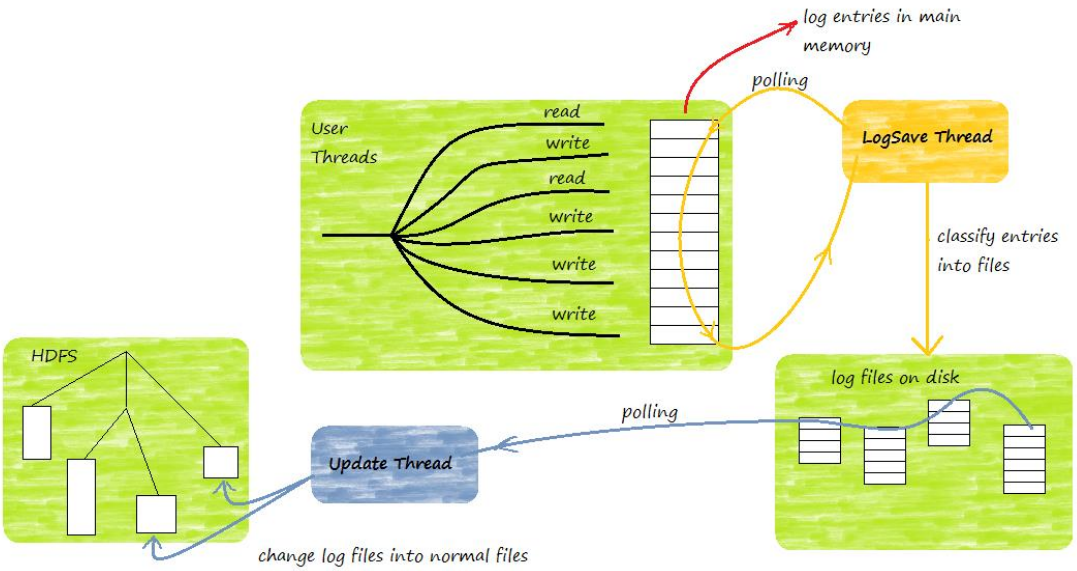

local
...
name = HDFS_PATH
Log:
ADD
.....
ADD
.....
File:
MyFileStream...


Client
MyFileStream
LFS
Cache Pool
org.apache.hadoop.fs.FileSystem

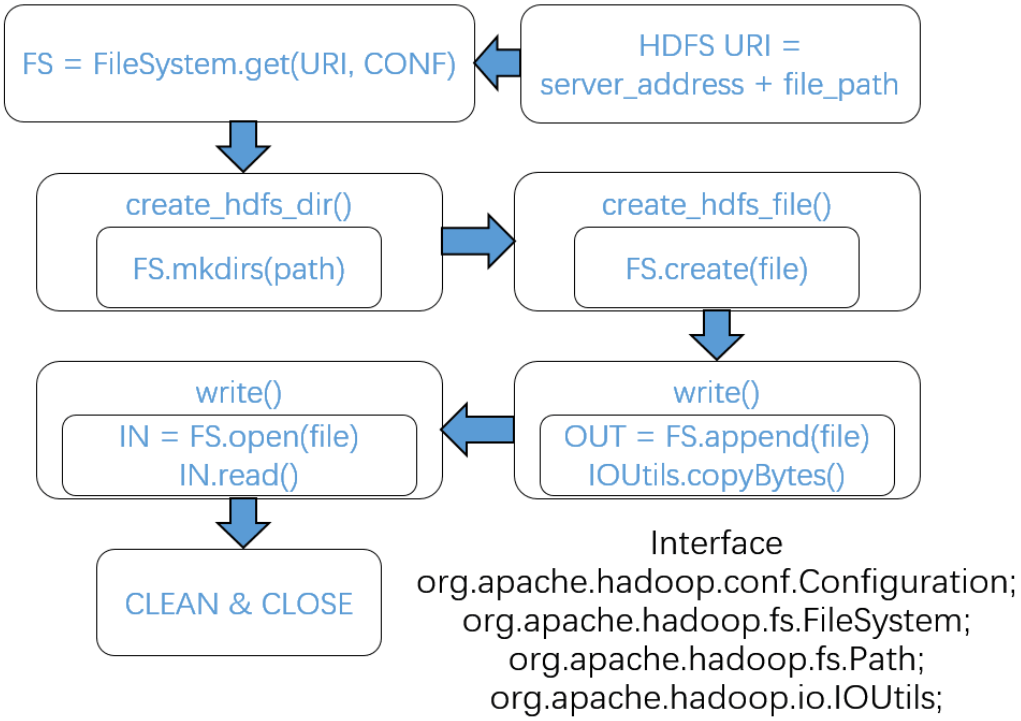
2. LOG 式的存储方式：


LogSys
ShareInstance
/* Singleton */
Public
Var:
1. logList
2. logTopNum
3. LogHistoryPath
Method:
1. List( )
2. New( )
3. Append( )
4. Delete( )
5. Realize(String filePath)
6. SingleLogFile(String filePath)
7. LogFile( )
8. CrashSave( )
9. CrashRecover( )
Private
Method:
1. NewFile(String filePath, String Data)
2. AppendFile(String filePath, String Data)
3. PrintLog(String filePath, String Data)
4. CleanLog(String filePath)
5. TypeTransfer(String type)
Public
Var:
1. int logNum
2. String File
3. String Data
4. Type Type
Method:
1. Log(int lognum, String file, String data, Type type)
Log
Enum
Var:
1. ERROR,
2. NEW
3. APPEND
Type

3. 异步式的读写策略 线程不断的轮询 过大上传



4. 和 hadoop 的对接工作接口



四、 重点与难点

1. 线程安全的问题

用户进程、守护进程都有可能对同一个文件进行写操作，所以我们将每个文件和一个可重入锁（reentrylock）关联起来，这样就可以完成对同一资源的互斥访问。获取不到锁的进程会进入休眠状态，等到合适的时机再次尝试获取锁。

另一方面，读操作的优先级高于写操作。这是因为每次读操作的时候需要让内存中的 log 条目，本地硬盘储存的 log 文件以及 HDFS 之中的远程文件满足一致性。于是每次读操作都会将本地的 log 文件全部上传到 HDFS 之中，在读取。这 read 的延迟虽然会比较高，但是方便了写操作。正好契合了多次低数据量写和少次高数据量读的现实需求。

2. 双守护线程

LFS 系统相当于是对用户写操作的一种 cache，必须要考虑 cache 一致性的问题。我们利用 memory hierarchy，在内存，硬盘，HDFS 上面都有用户的数据。这些数据是以 exclusive 的准则组织的。只要出现在内存之中的 log 条目，就不应该出现在硬盘之中的 log 文件里，在 log 文件里的数据，也不应该在 HDFS 之中。

为了达成这一目的，我们使用了两个守护线程。守护线程是一种“在后台提供通用性支持”的线程，它并不属于程序本体。守护线程用来服务用户线程的，如果没有其他用户线程在运行，那么就没有可服务对象，也就没有理由继续下去。其中一个守护线程“LogSave”自动将内存之中的 log 条目写成对应的 log 文件。这样在崩溃之后，就可以根据硬盘上的 log 文件快速回复。另外一个守护进程是“Update”。在用户需要读取，或者是 log 文件到达一定大小的时候，Update 就会将 log 文件转化为普通文件，并且上传到 HDFS 之中。通过这样定期的清空，可以保证 read 的延迟不会太大。

3. 方便可靠的修改模式

（1） 支持任意一条记录的删除

由于实现了将磁盘结构化为一个日志，需要的时候将被缓冲在内存中的所欲未决的写操作都放到一个单独的段中，作为日志末尾的一个邻接段写入磁盘。并且存在一个清理线程，周期性地扫描有没有因为被覆盖而未被使用的节点，将其清理。所以在真正生成实际文件之前，用户可以对自己以往的任意操作日志进行修改或者删除。方便地回退到某个版本。

（2） 崩溃时根据 LOG 日志恢复

使用了守护线程轮询检查 Log 文件，由于有了 Log 文件作为缓冲，所以即使在上传以及生成实际文件的过程中出现了崩溃，同样可以再次继续任务，极大地减少了丢失数据造成的影响。

五、 测试与数据分析

测试环境

VMware Workstation(Ubuntu 14.04, CPU @ 2.60GHz 2.59GHz, RAM @ 3568MB)
Eclipse Java EE Mars.1 Release (4.5.1)

Benchmark 设定

以下测试都是基于用户进行高频率写操作的情况下进行测试。

1. 与 Naive 式调用 HDFS 接口对比

单线程测试。和原生的 HDFS 接口函数的速度进行对比。进行 1000 次写，以相同的频率夹杂一定次数的写操作。每次写 10Bytes，每次读 1000Bytes。

1） 测试结果：

测试项/测试号	写次数	读次数	写时间(ms)(LFS)	读时间(ms)(LFS)	写时间(ms)(Naive)	读时间(ms)(Naive)	总时间(ms)(LFS)	总时间(ms)(Naive)	速度提升
测试1	1000	100	73	3521	28217	342	3594	28559	694.6%
测试2	1000	50	72	2481	27718	213	2553	27931	994%
测试3	1000	20	77	1042	25626	139	1119	25765	2202.5%
测试4	1000	10	70	751	24386	86	821	24472	2880.8%
测试5	1000	4	89	536	23644	56	625	23700	3692%
测试6	1000	1	383	327	23810	26	710	23836	3257.2%
测试7	1000	1000	113	40844	41076	2642	40957	43718	6.7%

2） 结果分析：

LFS 对于高频写操作总体速度相对于 Naïve 的实现提升明显，与我们的预期相符合。同时也可以看到，LFS 中读操作所占的时间与总时间的比例始终很大，而写操作的时间很小，尽管这不是优点，但这跟我们的 LOG 式本地存储所采用的机制是相符的。

另外，当读操作次数十分小时，测试结果产生了奇怪的现象，写操作的总时间增加了许多。经过分析，我觉得可能是由于如下原因：文件大小达到一定程度时 LFS 会将本地的 log 存储文件上传至服务器，而读操作数目小时，累积的写操作数目就多了，进行这样的上传将会消耗稍多的时间。此时若进行写操作，由于加入了文件互斥锁，可能线程就会被暂时挂起一段时间，这段时间就是写操作增加的时间。

2. 模拟实际应用

以一定的间隔使程序进行其他操作（测试中以每 200 次写睡眠 1 秒的行为来模拟）。进行 1000 次写，以相同的频率夹杂若干次写操作。每次写 10Bytes，每次读 1000Bytes。

1） 测试结果：

测试项/测试号	写次数	读次数	写时间(ms)(LFS)	读时间(ms)(LFS)	总时间(ms) LFS 加入间隔	总时间(ms) LFS 未加入间隔
测试1	1000	100	49	3612	3661	3594
测试2	1000	50	50	2351	2401	2553
测试3	1000	20	52	1099	1151	1119
测试4	1000	10	52	763	815	821
测试5	1000	4	45	569	614	625
测试6	1000	1	54	503	557	710
测试7	1000	1000	151	41073	41224	40957

2） 结果分析：

我们做这个测试，主要是为了了解实际应用中用户其他非 HDFS 操作的加入对 LFS 的效率是否有影响。测试结果表明夹杂其他指令，LFS 的效果基本不会变坏，轻微的浮动可能是由于机器状态的轻微变化。

当读操作很少时，我们发现基于 LFS 的操作在实际应用中能取得更加好的效果，这是因为 LFS 中会有两个守护进程，定期维护 log 日志的存储以及文件的上传，故对于这些本来可能需要一些时间的写操作，在用户进行其他行为时，守护线程已经自动更新了服务器

和本地的文件状态。

3. 多线程正确性测试

多个线程对不同文件同时读、写。我们的测试会同时开启 10 个线程，每个线程进行 1000 次每次 10Bytes 的写操作，若干次 1000Bytes 的读操作，是对应单线程测试的 10 倍操作量。

1） 测试结果：

测试运行后，服务器上的文件内容与对应的操作相一致。

由于多线程中，读操作时间和写操作时间不方便累计（若要加锁的话影响程序本身执行），我们采用运行总时间来替代。

每个线程 10 次读时，总时间 9679ms。

每个线程 100 次读时，总时间 24860ms。

2） 结果分析：

这里我们注意到，读操作数目较多时，多线程的时间比单线程时间简单地乘 10 要小不少，多线程并发在这里起到了作用。但是在双核四线程的 CPU 下并没有达到理想的状态，可能是因为，本地与 HDFS 的 File System 连接有一定的限制，不能完美地并行。

4. 高频率大数据量读写测试

单个线程，进行 10000 次 10KB 的写操作和 10 次读操作，测试耗费时间。

1） 测试结果：

写耗时 23.8s，读耗时 41.6s。

2） 结果分析：

经过计算，数据率约 1.54MB/s。在如此高的写入频率下仍然能保持不错的数据率，LFS 存储系统的表现不错。

六、 结论

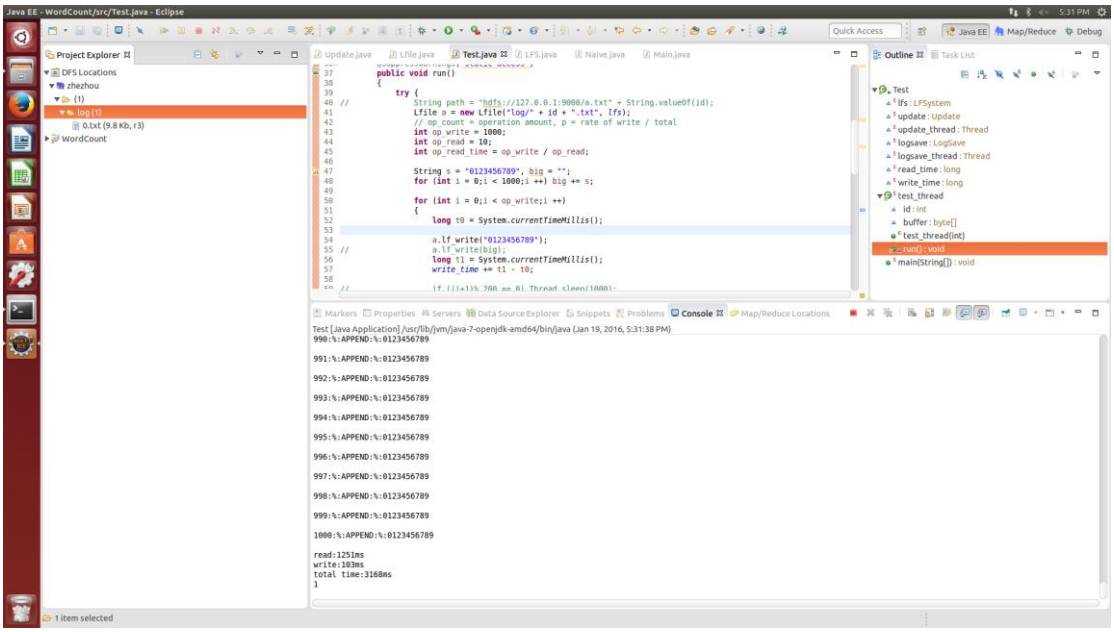
随着数据量的增长以及文件碎片化的提升，HDFS 分布式文件系统的吞吐量与传输速率问题越来越突出，本文提出了一种基于 LFS 日志文件系统与存储结构对大数据多碎片情况下的 HDFS 读写的优化系统。

经过测试，通过实验数据可以看到，在与直接调用，模拟实际应用，正确性以及在高频率大数据量下进行读写测试，在保证正确性的前提下，优化效果明显。

综上所述，随着全球数据呈爆炸式地增长，分布式文件系统将会越来越多地被研究与应用，在其对流式大数据存储的同时对碎片化以及存储模式的优化也同样是必不可少的。

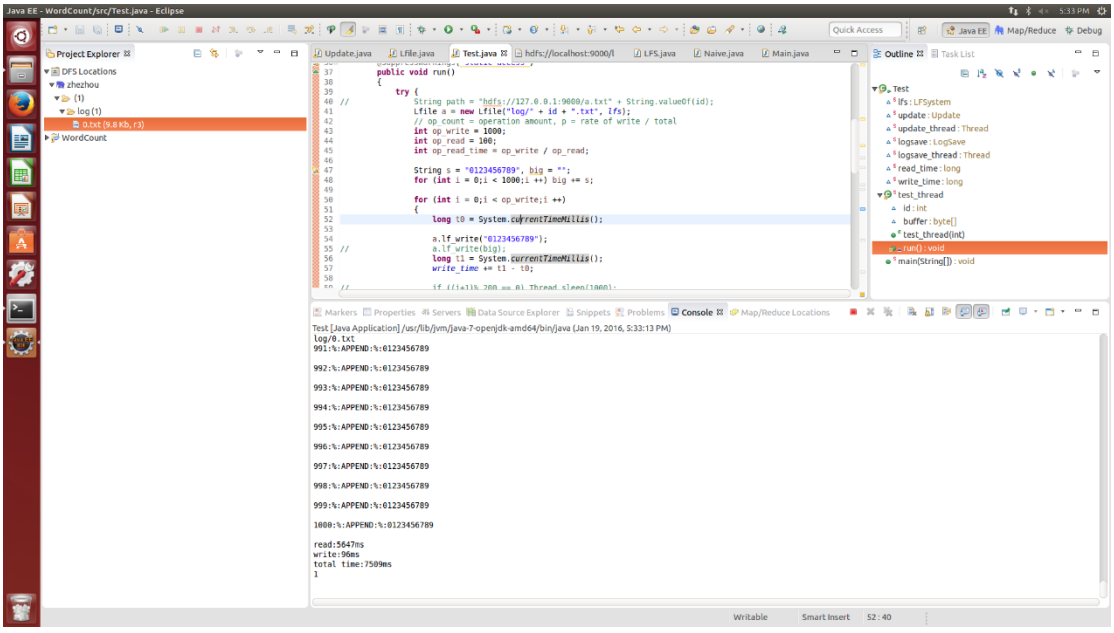
七、 程序截图

1. 10 次读：

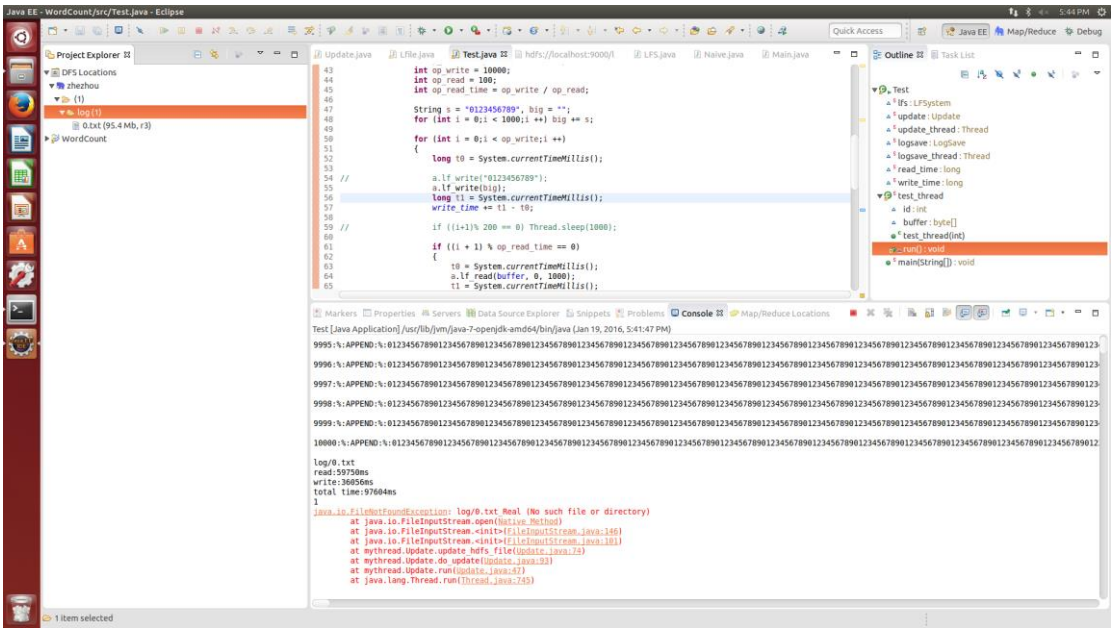


说明：上传文件大小正确（其中内容略过不截图了），total time 包括创建路径、文件等其他一些时间，是总共的运行时间。

2. 100 次读：



3. 100MB 写：



说明：最后那个 log/0.txt_Real 是将 LOG 式文件处理成真实文件的形式，在这里找不到是因为若所有 LOG 都已经上传，LOG 文件为空时将不会生成真实文件，出现这种异常是被程序 catch 的，不会影响结果正确性。

4. 多线程不同文件：