HBase原理之数据写入流程

范欣欣 HBase技术社区 1周前

众所周知，HBase默认适用于写多读少的应用，正是依赖于它相当出色的写入性能：一个100台RS的集群可以轻松地支撑每天10T的写入量。当然，为了支持更高吞吐量的写入，HBase还在不断地进行优化和修正，这篇文章结合0.98版本的源码全面地分析HBase的写入流程，全文分为三个部分，第一部分介绍客户端的写入流程，第二部分介绍服务器端的写入流程，最后再重点分析WAL的工作原理。

客户端流程解析

（1）用户提交put请求后，HBase客户端会将put请求添加到本地buffer中，符合一定条件就会通过AsyncProcess异步批量提交。HBase默认设置autoflush=true，表示put请求直接会提交给服务器进行处理；用户可以设置autoflush=false，这样的话put请求会首先放到本地buffer，等到本地buffer大小超过一定阈值（默认为2M，可以通过配置文件配置）之后才会提交。很显然，后者采用group commit机制提交请求，可以极大地提升写入性能，但是因为没有保护机制，如果客户端崩溃的话会导致提交的请求丢失。

（2）在提交之前，HBase会在元数据表.meta.中根据rowkey找到它们归属的region server，这个定位的过程是通过HConnection的locateRegion方法获得的。如果是批量请求的话还会把这些rowkey按照HRegionLocation分组，每个分组可以对应一次RPC请求。

（3）HBase会为每个HRegionLocation构造一个远程RPC请求MultiServerCallable<Row>，然后通过rpcCallerFactory.<MultiResponse> newCaller()执行调用，忽略掉失败重新提交和错误处理，客户端的提交操作到此结束。

服务器端流程解析

服务器端RegionServer接收到客户端的写入请求后，首先会反序列化为Put对象，然后执行各种检查操作，比如检查region是否是只读、memstore大小是否超过blockingMemstoreSize等。检查完成之后，就会执行如下核心操作：

（1）获取行锁、Region更新共享锁： HBase中使用行锁保证对同一行数据的更新都是互斥操作，用以保证更新的原子性，要么更新成功，要么失败。

（2）开始写事务：获取write number，用于实现MVCC，实现数据的非锁定读，在保证读写一致性的前提下提高读取性能。

（3）写缓存memstore：HBase中每列族都会对应一个store，用来存储该列数据。每个store都会有个写缓存memstore，用于缓存写入数据。HBase并不会直接将数据落盘，而是先写入缓存，等缓存满足一定大小之后再一起落盘。

（4）Append HLog：HBase使用WAL机制保证数据可靠性，即首先写日志再写缓存，即使发生宕机，也可以通过恢复HLog还原出原始数据。该步骤就是将数据构造为WALEdit对象，然后顺序写入HLog中，此时不需要执行sync操作。0.98版本采用了新的写线程模式实现HLog日志的写入，可以使得整个数据更新性能得到极大提升，具体原理见下一个章节。

（5）释放行锁以及共享锁

（6）Sync HLog：HLog真正sync到HDFS，在释放行锁之后执行sync操作是为了尽量减少持锁时间，提升写性能。如果Sync失败，执行回滚操作将memstore中已经写入的数据移除。

（7）结束写事务：此时该线程的更新操作才会对其他读请求可见，更新才实际生效。具体分析见文章《数据库事务系列 - HBase行级事务模型》

（8）flush memstore：当写缓存满64M之后，会启动flush线程将数据刷新到硬盘。刷新操作涉及到HFile相关结构，后面会详细对此进行介绍。

WAL机制解析

WAL(Write-Ahead Logging)是一种高效的日志算法，几乎是所有非内存数据库提升写性能的不二法门，基本原理是在数据写入之前首先顺序写入日志，然后再写入缓存，等到缓存写满之后统一落盘。之所以能够提升写性能，是因为WAL将一次随机写转化为了一次顺序写加一次内存写。提升写性能的同时，WAL可以保证数据的可靠性，即在任何情

况下数据不丢失。假如一次写入完成之后发生了宕机，即使所有缓存中的数据丢失，也可以通过恢复日志还原出丢失的数据。

WAL持久化等级

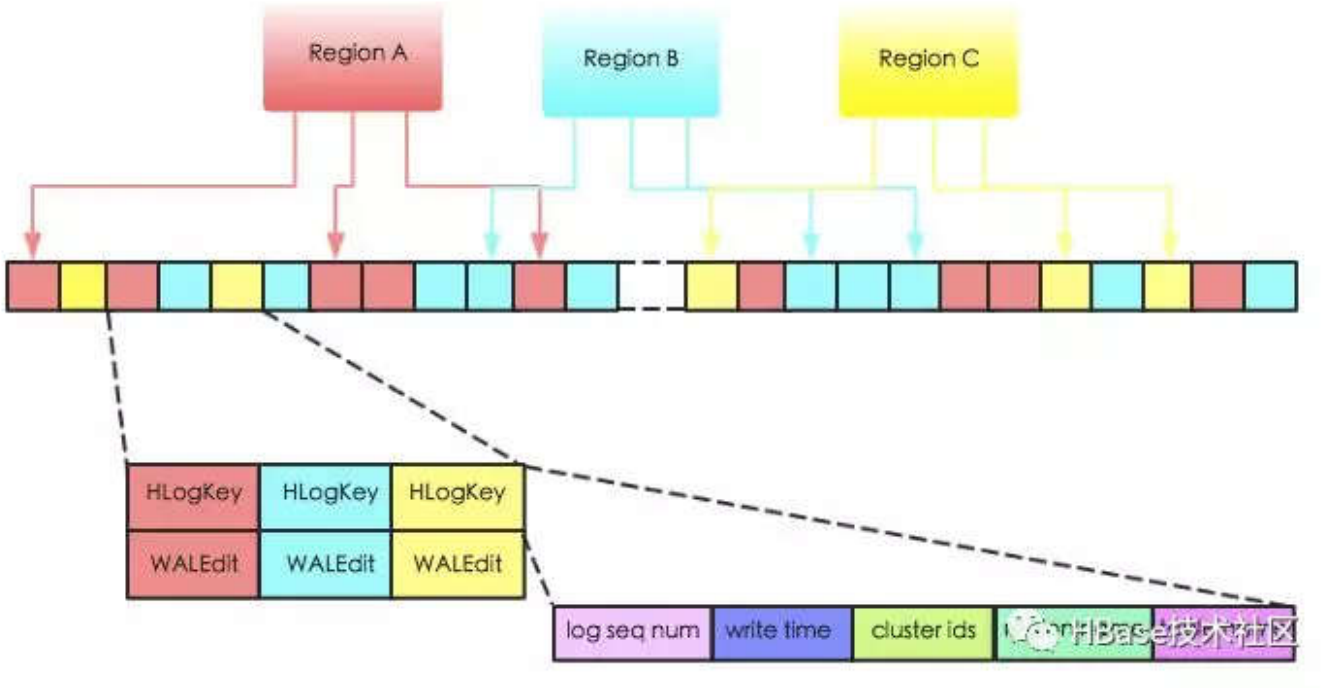
HBase中可以通过设置WAL的持久化等级决定是否开启WAL机制、以及HLog的落盘方式。WAL的持久化等级分为如下四个等级：

1. SKIP_WAL：只写缓存，不写HLog日志。这种方式因为只写内存，因此可以极大的提升写入性能，但是数据有丢失的风险。在实际应用过程中并不建议设置此等级，除非确认不要求数据的可靠性。
2. ASYNC_WAL：异步将数据写入HLog日志中。
3. SYNC_WAL：同步将数据写入日志文件中，需要注意的是数据只是被写入文件系统中，并没有真正落盘。
4. FSYNC_WAL：同步将数据写入日志文件并强制落盘。最严格的日志写入等级，可以保证数据不会丢失，但是性能相对比较差。
5. USER_DEFAULT：默认如果用户没有指定持久化等级，HBase使用SYNC_WAL等级持久化数据。

用户可以通过客户端设置WAL持久化等级，代码：put.setDurability(Durability. SYNC_WAL);

HLog数据结构

HBase中，WAL的实现类为HLog，每个Region Server拥有一个HLog日志，所有region的写入都是写到同一个HLog。下图表示同一个Region Server中的3个 region共享一个HLog。当数据写入时，是将数据对<HLogKey,WALEdit>按照顺序追加到HLog中，以获取最好的写入性能。



上图中HLogKey主要存储了log sequence number，更新时间 write time，region name，表名table name以及cluster ids。其中log sequncece number作为HFile中一个重要的元数据，和HLog的生命周期息息相关，后续章节会详细介绍；region name和table name分别表征该段日志属于哪个region以及哪张表；cluster ids用于将日志复制到集群中其他机器上。

WALEdit用来表示一个事务中的更新集合，在之前的版本，如果一个事务中对一行row R中三列c1，c2，c3分别做了修改，那么hlog中会有3个对应的日志片段如下所示：

<logseq1-for-edit1>:<keyvalue-for-edit-c1>

<logseq2-for-edit2>:<keyvalue-for-edit-c2>

<logseq3-for-edit3>:<keyvalue-for-edit-c3>

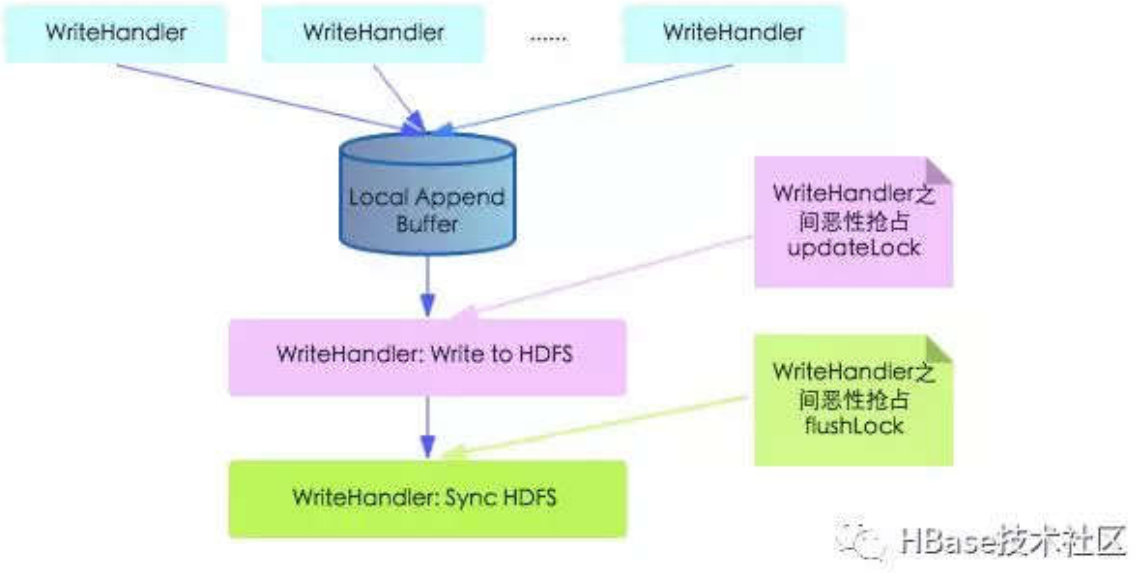
然而，这种日志结构无法保证行级事务的原子性，假如刚好更新到c2之后发生宕机，那么就会产生只有部分日志写入成功的现象。为此，hbase将所有对同一行的更新操作都表示为一个记录，如下：

<logseq#-for-entire-txn>:<WALEdit-for-entire-txn>

其中WALEdit会被序列化为格式<-1，# of edits，<KeyValue>，<KeyValue>，<KeyValue>>，比如<-1，3，<keyvalue-for-edit-c1>，<keyvalue-for-edit-c2>，<keyvalue-for-edit-c3>>，其中-1作为标示符表征这种新的日志结构。

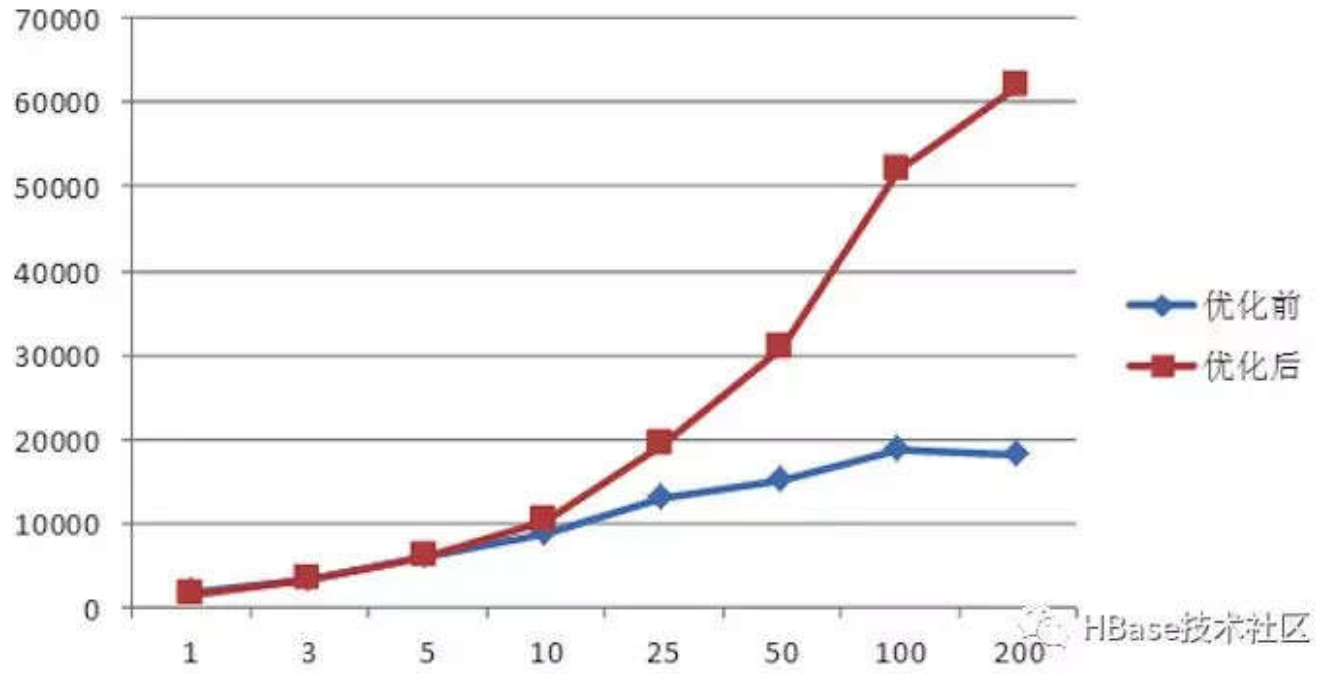
WAL写入模型

了解了HLog的结构之后，我们就开始研究HLog的写入模型。HLog的写入可以分为三个阶段，首先将数据对<HLogKey,WALEdit>写入本地缓存，然后再将本地缓存写入文件系统，最后执行sync操作同步到磁盘。在以前老的写入模型中，上述三步都由工作线程独自完成，如下图所示：

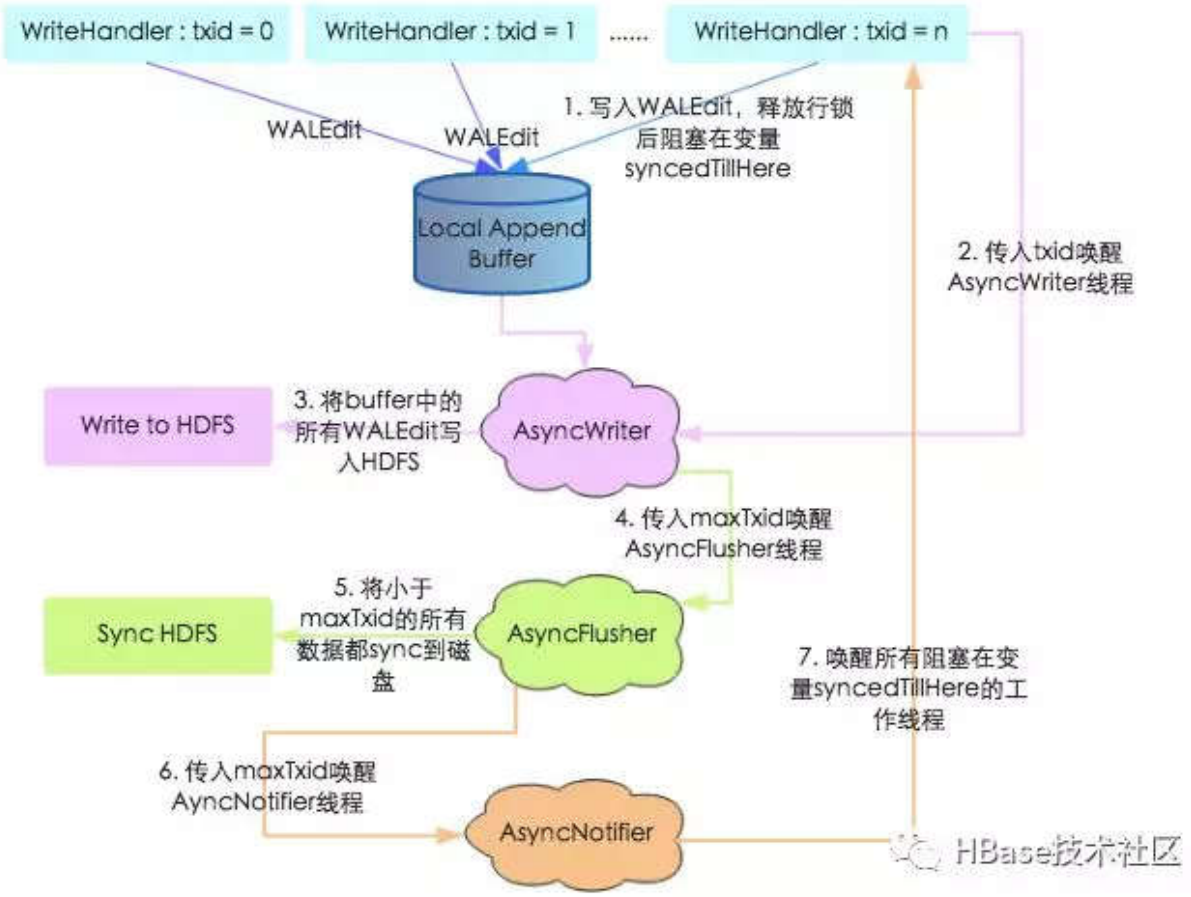


上图中，本地缓存写入文件系统那个步骤工作线程需要持有updateLock执行，不同工作线程之间必然会恶性竞争；不仅如此，在Sync HDFS这步中，工作线程之间需要抢占flushLock，因为Sync操作是一个耗时操作，抢占这个锁会导致写入性能大幅降低。

所幸的是，来自中国（准确的来说，是来自小米，鼓掌）的3位工程师意识到了这个问题，进而提出了一种新的写入模型并被官方采纳。根据官方测试，新写入模型的吞吐量比之前提升3倍多，单台RS写入吞吐量介于12150～31520，5台RS组成的集群写入吞吐量介于22000～70000（见HBASE-8755）。下图是小米官方给出来的对比测试结果：



在新写入模型中，本地缓存写入文件系统以及Sync HDFS都交给了新的独立线程完成，并引入一个Notify线程通知工作线程是否已经Sync成功，采用这种机制消除上述锁竞争，具体如下图所示：



1. 上文中提到工作线程在写入WALEdit之后并没有进行Sync，而是等到释放行锁阻塞在syncedTillHere变量上，等待AsyncNotifier线程唤醒。
2. 工作线程将WALEdit写入本地Buffer之后，会生成一个自增变量txid，携带此txid唤醒AsyncWriter线程
3. AsyncWriter线程会取出本地Buffer中的所有WALEdit，写入HDFS。注意该线程会比较传入的txid和已经写入的最大txid（writtenTxid），如果传入的txid小于writteTxid，表示该txid对应的WALEdit已经写入，直接跳过
4. AsyncWriter线程将所有WALEdit写入HDFS之后携带maxTxid唤醒AsyncFlusher线程
5. AsyncFlusher线程将所有写入文件系统的WALEdit统一Sync刷新到磁盘
6. 数据全部落盘之后调用setFlushedTxid方法唤醒AyncNotifier线程
7. AyncNotifier线程会唤醒所有阻塞在变量syncedTillHere的工作线程，工作线程被唤醒之后表示WAL写入完成，后面再执行MVCC结束写事务，推进全局读取点，本次更新才会对用户可见

通过上述过程的梳理可以知道，新写入模型采取了多线程模式独立完成写文件系统、sync磁盘操作，避免了之前多工作线程恶性抢占锁的问题。同时，工作线程在将WALEdit写入本地Buffer之后并没有马上阻塞，而是释放行锁之后阻塞等待WALEdit落盘，这样可以尽可能地避免行锁竞争，提高写入性能。

总结

本文首先介绍了HBase的写入流程，之后重点分析了WAL的写入模型以及相关优化。希望借此能够对HBase写入的高性能特性能够理解。后面一篇文章会接着介绍写入到memstore的数据如何真正的落盘，敬请期待！

大家工作学习遇到HBase技术问题，把问题发布到HBase技术社区论坛 **http://hbase.group**，欢迎大家论坛上面提问留言讨论。想了解更多HBase技术关注HBase技术社区公众号(微信号:**hbasegroup**)，非常欢迎大家积极投稿。

长按下面的二维码加入HBase技术社区微信群