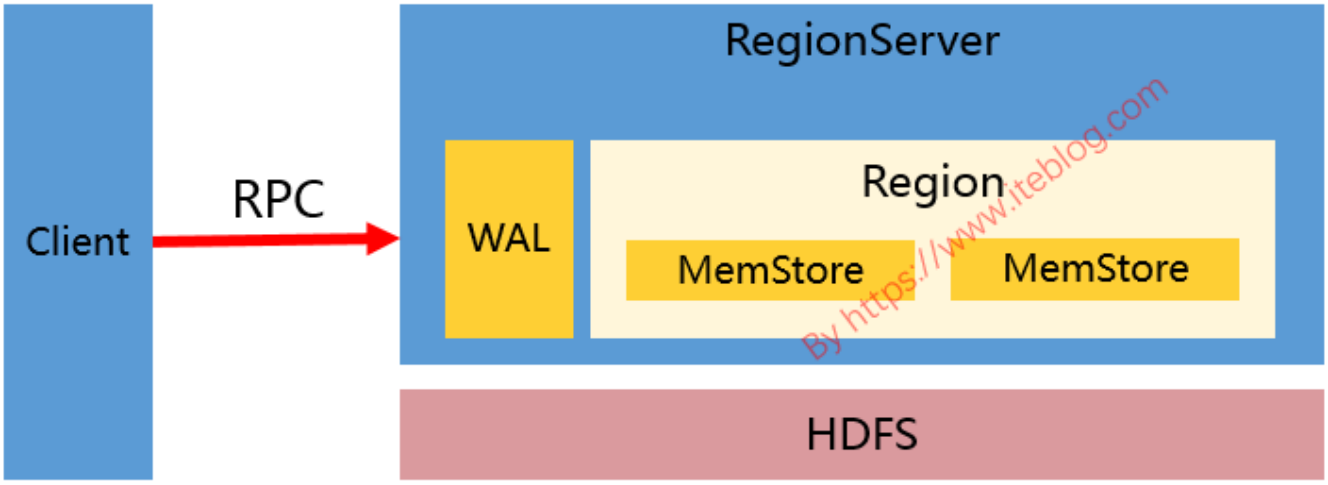
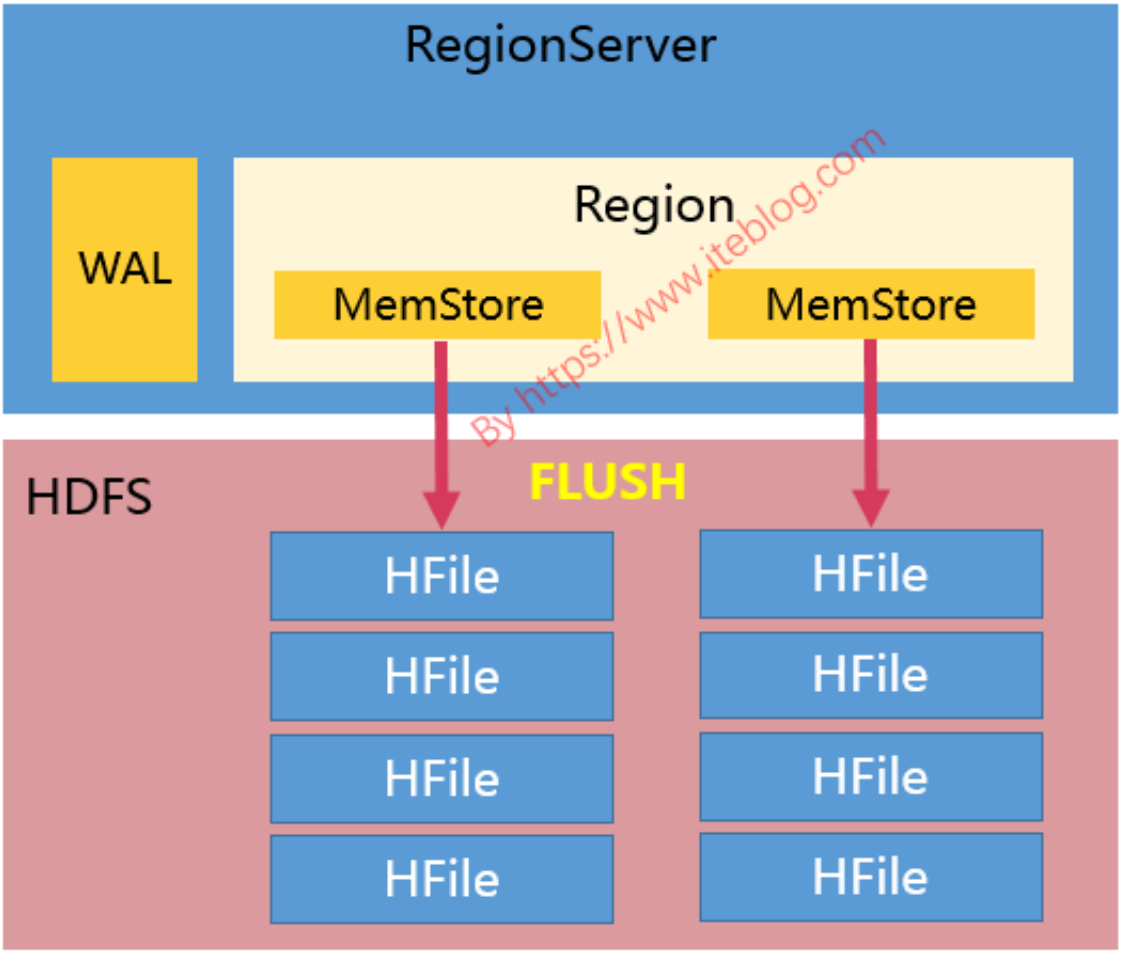
HBase 入门之数据刷写(Memstore Flush)详细说明

接触过 HBase 的同学应该对 HBase 写数据的过程比较熟悉（不熟悉也没关系）。HBase 写数据（比如 put、delete）的时候，都是写 WAL（假设 WAL 没有被关闭）
，然后将数据写到一个称为 MemStore 的内存结构里面的，如下图：



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

但是，MemStore
毕竟是内存里面的数据结构，写到这里面的数据最终还是需要持久化到磁盘的，生成
HFile。如下图：



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

理解 MemStore 的刷写对优化 MemStore 有很重要的意义，大部分人遇到的性能问题都是写操作被阻塞（Block）了，无法写入HBase。本文基于 HBase 2.0.2，并对 MemStore 的 Flush 进行说明，包括哪几种条件会触发 Memstore Flush 及目前常见的刷写策略（FlushPolicy）。

什么时候触发 MemStore Flush

有很多情况会触发 MemStore 的 Flush 操作，所以我们最好需要了解每种情况在什么时候触发 Memstore Flush。总的来说，主要有以下几种情况会触发 Memstore Flush：

- Region 中所有 MemStore 占用的内存超过相关阈值
- 整个 RegionServer 的 MemStore 占用内存总和大于相关阈值
- WAL数量大于相关阈值
- 定期自动刷写
- 数据更新超过一定阈值
- 手动触发刷写

下面对这几种刷写进行简要说明。

Region 中所有 MemStore 占用的内存超过相关阈值

当一个 Region 中所有 MemStore 占用的内存（包括 OnHeap + OffHeap）大小超过刷写阈值的时候会触发一次刷写，这个阈值由 hbase.hregion.memstore.flush.size 参数控制，默认为128MB。我们每次调用 put、delete 等操作都会检查的这个条件的。

但是如果我们的数据增加得很快，达到了 hbase.hregion.memstore.flush.size * hbase.hregion.memstore.block.multiplier 的大小，hbase.hregion.memstore.block.multiplier 默认值为4，也就是128*4=512MB的时候，那么除了触发 MemStore 刷写之外，HBase 还会在刷写的时候同时阻塞所有写入该 Store 的写请求！这时候如果你往对应的 Store 写数据，会出现 RegionTooBusyException 异常。

整个 RegionServer 的 MemStore 占用内存总和大于相关阈值

HBase 为 RegionServer 的 MemStore 分配了一定的写缓存，大小等于 hbase_heapsize（RegionServer 占用的堆内存大小）* hbase.regionserver.global.memstore.size。hbase.regionserver.global.memstore.size 的默认值是 0.4，也就是说写缓存大概占用 RegionServer 整个 JVM 内存使用量的 40%。

如果整个 RegionServer 的 MemStore 占用内存总和大于 hbase.regionserver.global.memstore.size.lower.limit * hbase.regionserver.global.memstore.size * hbase_heapsize 的时候，将会触发 MemStore 的刷写。其中 hbase.regionserver.global.memstore.size.lower.limit 的默认值为 0.95。

举个例子，如果我们 HBase 堆内存总共是 32G，按照默认的比例，那么触发 RegionServer 级别的 Flush 是 RS 中所有的 MemStore 占用内存为：32 * 0.4 * 0.95 = 12.16G。

注意：0.99.0 之前 hbase.regionserver.global.memstore.size 是 hbase.regionserver.global.memstore.upperLimit 参数；hbase.regionserver.global.memstore.size.lower.limit 是 hbase.regionserver.global.memstore.lowerLimit，参见 HBASE-5349

RegionServer 级别的 Flush 策略是每次找到 RS 中占用内存最大的 Region 对他进行刷写，这个操作是循环进行的，直到总体内存的占用低于全局 MemStore 刷写下限（hbase.regionserver.global.memstore.size.lower.limit * hbase.regionserver.global.memstore.size * hbase_heapsize）才会停止。

需要注意的是，如果达到了 RegionServer 级别的 Flush，那么当前 RegionServer 的所有写操作将会被阻塞，而且这个阻塞可能会持续到分钟级别。

WAL数量大于相关阈值

WAL（Write-ahead log，预写日志）用来解决宕机之后的操作恢复问题的。数据到达 Region 的时候是先写入 WAL，然后再被写到 Memstore 的。如果 WAL 的数量越来越大，这就意味着

MemStore 中未持久化到磁盘的数据越来越多。当 RS
挂掉的时候，恢复时间将会变成，所以有必要在 WAL
到达一定的数量时进行一次刷写操作。这个阈值（maxLogs）的计算公式如下：

```
this.blocksize = WALUtil.getWALBlockSize(this.conf, this.fs, this.walDir);
float multiplier = conf.getFloat("hbase.regionserver.logroll.multiplier", 0.5f);
this.logrollsize = (long)(this.blocksize * multiplier);
this.maxLogs = conf.getInt("hbase.regionserver.maxlogs",
    Math.max(32, calculateMaxLogFiles(conf, logrollsize)));

public static long getWALBlockSize(Configuration conf, FileSystem fs, Path dir)
    throws IOException {
  return conf.getLong("hbase.regionserver.hlog.blocksize",
      CommonFSUtils.getDefaultBlockSize(fs, dir) * 2);
}

private int calculateMaxLogFiles(Configuration conf, long logRollSize) {
  Pair<Long, MemoryType> globalMemstoreSize = MemorySizeUtil.getGlobalMemStoreSize(co
nf);
  return (int) ((globalMemstoreSize.getFirst() * 2) / logRollSize);
}
```

也就是说，如果设置了 hbase.regionserver.maxlogs，那就是这个参数的值；否则是 max(32, hbase_heapsize * hbase.regionserver.global.memstore.size * 2 / logRollSize)。如果某个 RegionServer 的 WAL 数量大于 maxLogs 就会触发 MemStore 的刷写。

WAL 数量触发的刷写策略是，找到最旧的 un-archived WAL 文件，并找到这个 WAL 文件对应的 Regions， 然后对这些 Regions 进行刷写。

定期自动刷写

如果我们很久没有对 HBase 的数据进行更新，这时候就可以依赖定期刷写策略了。RegionServer 在启动的时候会启动一个线程 PeriodicMemStoreFlusher 每隔
hbase.server.thread.wakefrequency 时间去检查属于这个 RegionServer 的 Region
有没有超过一定时间都没有刷写，这个时间是由 hbase.regionserver.optionalcacheflushinterval 参数控制的，默认是
3600000，也就是1小时会进行一次刷写。如果设定为0，则意味着关闭定时自动刷写。

为了防止一次性有过多的 MemStore 刷写，定期自动刷写会有 0 ~ 5 分钟的延迟，具体参见 PeriodicMemStoreFlusher 类的实现。

数据更新超过一定阈值

如果 HBase 的某个 Region 更新的很频繁，而且既没有达到自动刷写阀值，也没有达到内存的使用限制，但是内存中的更新数量已经足够多，比如超过 hbase.regionserver.flush.per.changes 参数配置，默认为30000000，那么也是会触发刷写的。

手动触发刷写

除了 HBase 内部一些条件触发的刷写之外，我们还可以通过执行相关命令或 API 来触发 MemStore 的刷写操作。比如调用可以调用 Admin 接口提供的方法：

```
void flush(TableName tableName) throws IOException;
void flushRegion(byte[] regionName) throws IOException;
void flushRegionServer(ServerName serverName) throws IOException;
```

分别对某张表、某个 Region 或者某个 RegionServer 进行刷写操作。也可以在 Shell 中通过执行 flush 命令：

```
hbase> flush 'TABLENAME'
hbase> flush 'REGIONNAME'
hbase> flush 'ENCODED_REGIONNAME'
hbase> flush 'REGION_SERVER_NAME'
```

需要注意的是，以上所有条件触发的刷写操作最后都会检查对应的 HStore 包含的 StoreFiles 文件超过 hbase.hstore.blockingStoreFiles
参数配置的个数，默认值是16。如果满足这个条件，那么当前刷写会被推迟到 hbase.hstore.blockingWaitTime 参数设置的时间后再刷写。在阻塞刷写的同时，HBase 还会请求 Split 或 Compaction 操作。

什么操作会触发 MemStore 刷写

我们常见的 put、delete、append、increment、调用 flush 命令、Region 分裂、Region Merge、bulkLoad HFiles
以及给表做快照操作都会对上面的相关条件做检查，以便判断要不要做刷写操作。

MemStore 刷写策略（FlushPolicy）

在 HBase 1.1 之前，MemStore 刷写是 Region 级别的。就是说，如果要刷写某个 MemStore ，MemStore 所在的 Region 中其他 MemStore
也是会被一起刷写的！这会造成一定的问题，比如小文件问题，具体参见 《为什么不建议在 HBase 中使用过多的列族》。针对这个问题，HBASE-10201/HBASE-3149

引入列族级别的刷写。我们可以通过 hbase.regionserver.flush.policy 参数选择不同的刷写策略。

目前 HBase 2.0.2 的刷写策略全部都是实现 FlushPolicy 抽象类的。并且自带三种刷写策略：FlushAllLargeStoresPolicy、FlushNonSloppyStoresFirstPolicy 以及 FlushAllStoresPolicy。

FlushAllStoresPolicy

这种刷写策略实现最简单，直接返回当前 Region 对应的所有 MemStore。也就是每次刷写都是对 Region 里面所有的 MemStore 进行的，这个行为和 HBase 1.1 之前是一样的。

FlushAllLargeStoresPolicy

在 HBase 2.0 之前版本是 FlushLargeStoresPolicy，后面被拆分成分 FlushAllLargeStoresPolicy 和FlushNonSloppyStoresFirstPolicy，参见 HBASE-14920。

这种策略会先判断 Region 中每个 MemStore 的使用内存（OnHeap + OffHeap）是否大于某个阀值，大于这个阀值的 MemStore 将会被刷写。阀值的计算是由 hbase.hregion.percolumnfamilyflush.size.lower.bound 、hbase.hregion.percolumnfamilyflush.size.lower.bound.min 以及 hbase.hregion.memstore.flush.size 参数决定的。计算逻辑如下：

```
//region.getMemStoreFlushSize() / familyNumber
//就是 hbase.hregion.memstore.flush.size 参数的值除以相关表列族的个数
flushSizeLowerBound = max(region.getMemStoreFlushSize() / familyNumber, hbase.hregion.percolumnfamilyflush.size.lower.bound.min)

//如果设置了 hbase.hregion.percolumnfamilyflush.size.lower.bound
flushSizeLowerBound = hbase.hregion.percolumnfamilyflush.size.lower.bound
```

计算逻辑上面已经很清晰的描述了。hbase.hregion.percolumnfamilyflush.size.lower.bound.min 默认值为 16MB，而 hbase.hregion.percolumnfamilyflush.size.lower.bound 没有设置。

比如当前表有3个列族，其他用默认的值，那么 flushSizeLowerBound = max((long)128 / 3, 16) = 42。

如果当前 Region 中没有 MemStore 的使用内存大于上面的阀值，FlushAllLargeStoresPolicy 策略就退化成 FlushAllStoresPolicy 策略了，也就是会对 Region 里面所有的 MemStore 进行 Flush。

FlushNonSloppyStoresFirstPolicy

HBase 2.0 引入了 in-memory compaction，参见 HBASE-13408。如果我们对相关列族 hbase.hregion.compacting.memstore.type 参数的值不是 NONE，那么这个 MemStore 的 isSloppyMemStore 值就是 true，否则就是 false。

FlushNonSloppyStoresFirstPolicy 策略将 Region 中的 MemStore 按照 isSloppyMemStore 分到两个 HashSet 里面（sloppyStores 和 regularStores）。然后

- 判断 regularStores 里面是否有 MemStore 内存占用大于相关阀值的 MemStore，有的话就会对这些 MemStore 进行刷写，其他的不做处理，这个阀值计算和 FlushAllLargeStoresPolicy 的阀值计算逻辑一致。
- 如果 regularStores 里面没有 MemStore 内存占用大于相关阀值的 MemStore，这时候就开始在 sloppyStores 里面寻找是否有 MemStore 内存占用大于相关阀值的 MemStore，有的话就会对这些 MemStore 进行刷写，其他的不做处理。
- 如果上面 sloppyStores 和 regularStores 都没有满足条件的 MemStore 需要刷写，这时候就 FlushNonSloppyStoresFirstPolicy 策略久退化成 FlushAllStoresPolicy 策略了。

刷写的过程

MemStore 的刷写过程很复杂，很多操作都可能触发，但是这些条件触发的刷写最终都是调用 HRegion 类中的 internalFlushcache 方法。

```
protected FlushResultImpl internalFlushcache(WAL wal, long myseqid,
     Collection<HStore> storesToFlush, MonitoredTask status, boolean writeFlushWalMarker,
     FlushLifeCycleTracker tracker) throws IOException {
   PrepareFlushResult result =
      internalPrepareFlushCache(wal, myseqid, storesToFlush, status, writeFlushWalMarker, tra
cker);
   if (result.result == null) {
     return internalFlushCacheAndCommit(wal, status, result, storesToFlush);
   } else {
     return result.result; // early exit due to failure from prepare stage
   }
}
```

从上面的实现可以看出，Flush 操作主要分以下几步做的

- prepareFlush 阶段：刷写的第一步是对 MemStore 做 snapshot，为了防止刷写过程中更新的数据同时在 snapshot 和 MemStore 中而造成后续处理的困难，所以在刷写期间需要持有 updateLock 。持有了 updateLock 之后，这将阻塞客户端的写操作。所以只在创建 snapshot 期间持有 updateLock，而且

snapshot 的创建非常快，所以此锁期间对客户的影响一般非常小。对 MemStore 做 snapshot 是 internalPrepareFlushCache 里面进行的。

- flushCache 阶段：如果创建快照没问题，那么返回的 result.result 将为 null。这时候我们就可以进行下一步 internalFlushCacheAndCommit。其实 internalFlushCacheAndCommit 里面包含两个步骤：flushCache 和 commit 阶段。flushCache 阶段其实就是将 prepareFlush 阶段创建好的快照写到临时文件里面，临时文件是存放在对应 Region 文件夹下面的 .tmp 目录里面。
- commit 阶段：将 flushCache 阶段生产的临时文件移到（rename）对应的列族目录下面，并做一些清理工作，比如删除第一步生成的 snapshot。