**配置优化zookeeper.session.timeout**  
**默认值**：3分钟（180000ms）  
**说明**：RegionServer与Zookeeper间的连接超时时间。当超时时间到后，ReigonServer会被Zookeeper从RS集群清单中移除，HMaster收到移除通知后，会对这台server负责的regions重新balance，让其他存活的RegionServer接管.  
**调优**：  
这个timeout决定了RegionServer是否能够及时的failover。设置成1分钟或更低，可以减少因等待超时而被延长的failover时间。  
不过需要注意的是，对于一些Online应用，RegionServer从宕机到恢复时间本身就很短的（网络闪断，crash等故障，运维可快速介入），如果调低timeout时间，反而会得不偿失。因为当ReigonServer被正式从RS集群中移除时，HMaster就开始做balance了（让其他RS根据故障机器记录的WAL日志进行恢复）。当故障的RS在人工介入恢复后，这个balance动作是毫无意义的，反而会使负载不均匀，给RS带来更多负担。特别是那些固定分配regions的场景。  
  
**hbase.regionserver.handler.count**  
**默认值**：10  
**说明**：RegionServer的请求处理IO线程数。  
**调优**：  
这个参数的调优与内存息息相关。  
较少的IO线程，适用于处理单次请求内存消耗较高的Big PUT场景（大容量单次PUT或设置了较大cache的scan，均属于Big PUT）或ReigonServer的内存比较紧张的场景。  
较多的IO线程，适用于单次请求内存消耗低，TPS要求非常高的场景。设置该值的时候，以监控内存为主要参考。  
这里需要注意的是如果server的region数量很少，大量的请求都落在一个region上，因快速充满memstore触发flush导致的读写锁会影响全局TPS，不是IO线程数越高越好。  
压测时，开启[Enabling RPC-level logging](http://hbase.apache.org/book.html#rpc.logging)，可以同时监控每次请求的内存消耗和GC的状况，最后通过多次压测结果来合理调节IO线程数。  
这里是一个案例 [Hadoop and HBase Optimization for Read Intensive Search Applications](http://software.intel.com/en-us/articles/hadoop-and-hbase-optimization-for-read-intensive-search-applications/)，作者在SSD的机器上设置IO线程数为100，仅供参考。  
  
**hbase.hregion.max.filesize**  
**默认值**：256M  
**说明**：在当前ReigonServer上单个Reigon的最大存储空间，单个Region超过该值时，这个Region会被自动split成更小的region。  
**调优**：  
小region对split和compaction友好，因为拆分region或compact小region里的storefile速度很快，内存占用低。缺点是split和compaction会很频繁。  
特别是数量较多的小region不停地split, compaction，会导致集群响应时间波动很大，region数量太多不仅给管理上带来麻烦，甚至会引发一些Hbase的bug。  
一般512以下的都算小region。  
大region，则不太适合经常split和compaction，因为做一次compact和split会产生较长时间的停顿，对应用的读写性能冲击非常大。此外，大region意味着较大的storefile，compaction时对内存也是一个挑战。  
当然，大region也有其用武之地。如果你的应用场景中，某个时间点的访问量较低，那么在此时做compact和split，既能顺利完成split和compaction，又能保证绝大多数时间平稳的读写性能。  
既然split和compaction如此影响性能，有没有办法去掉？  
compaction是无法避免的，split倒是可以从自动调整为手动。  
只要通过将这个参数值调大到某个很难达到的值，比如100G，就可以间接禁用自动split（RegionServer不会对未到达100G的region做split）。  
再配合[RegionSplitter](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/util/RegionSplitter.html)这个工具，在需要split时，手动split。  
手动split在灵活性和稳定性上比起自动split要高很多，相反，管理成本增加不多，比较推荐online实时系统使用。  
内存方面，小region在设置memstore的大小值上比较灵活，大region则过大过小都不行，过大会导致flush时app的IO wait增高，过小则因store file过多影响读性能。  
  
**hbase.regionserver.global.memstore.upperLimit/lowerLimit**  
**默认值：**0.4/0.35  
  
**upperlimit说明**：hbase.hregion.memstore.flush.size 这个参数的作用是当单个Region内所有的memstore大小总和超过指定值时，flush该region的所有memstore。RegionServer的flush是通过将请求添加一个队列，模拟生产消费模式来异步处理的。那这里就有一个问题，当队列来不及消费，产生大量积压请求时，可能会导致内存陡增，最坏的情况是触发OOM。  
这个参数的作用是防止内存占用过大，当ReigonServer内所有region的memstores所占用内存总和达到heap的40%时，HBase会强制block所有的更新并flush这些region以释放所有memstore占用的内存。  
  
**lowerLimit说明**： 同upperLimit，只不过lowerLimit在所有region的memstores所占用内存达到Heap的35%时，不flush所有的memstore。它会找一个memstore内存占用最大的region，做个别flush，此时写更新还是会被block。lowerLimit算是一个在所有region强制flush导致性能降低前的补救措施。在日志中，表现为 “\*\* Flush thread woke up with memory above low water.”  
**调优**：这是一个Heap内存保护参数，默认值已经能适用大多数场景。  
参数调整会影响读写，如果写的压力大导致经常超过这个阀值，则调小读缓存hfile.block.cache.size增大该阀值，或者Heap余量较多时，不修改读缓存大小。  
如果在高压情况下，也没超过这个阀值，那么建议你适当调小这个阀值再做压测，确保触发次数不要太多，然后还有较多Heap余量的时候，调大hfile.block.cache.size提高读性能。  
还有一种可能性是 hbase.hregion.memstore.flush.size保持不变，但RS维护了过多的region，要知道 region数量直接影响占用内存的大小。  
  
**hfile.block.cache.size**  
**默认值**：0.2  
**说明**：storefile的读缓存占用Heap的大小百分比，0.2表示20%。该值直接影响数据读的性能。  
**调优**：当然是越大越好，如果写比读少很多，开到0.4-0.5也没问题。如果读写较均衡，0.3左右。如果写比读多，果断默认吧。设置这个值的时候，你同时要参考 hbase.regionserver.global.memstore.upperLimit ，该值是memstore占heap的最大百分比，两个参数一个影响读，一个影响写。如果两值加起来超过80-90%，会有OOM的风险，谨慎设置。  
  
**hbase.hstore.blockingStoreFiles**  
**默认值：**7  
**说明**：在flush时，当一个region中的Store（Coulmn Family）内有超过7个storefile时，则block所有的写请求进行compaction，以减少storefile数量。  
**调优**：block写请求会严重影响当前regionServer的响应时间，但过多的storefile也会影响读性能。从实际应用来看，为了获取较平滑的响应时间，可将值设为无限大。如果能容忍响应时间出现较大的波峰波谷，那么默认或根据自身场景调整即可。  
  
**hbase.hregion.memstore.block.multiplier**  
**默认值：**2  
**说明**：当一个region里的memstore占用内存大小超过hbase.hregion.memstore.flush.size两倍的大小时，block该region的所有请求，进行flush，释放内存。  
虽然我们设置了region所占用的memstores总内存大小，比如64M，但想象一下，在最后63.9M的时候，我Put了一个200M的数据，此时memstore的大小会瞬间暴涨到超过预期的hbase.hregion.memstore.flush.size的几倍。这个参数的作用是当memstore的大小增至超过hbase.hregion.memstore.flush.size 2倍时，block所有请求，遏制风险进一步扩大。  
**调优**： 这个参数的默认值还是比较靠谱的。如果你预估你的正常应用场景（不包括异常）不会出现突发写或写的量可控，那么保持默认值即可。如果正常情况下，你的写请求量就会经常暴长到正常的几倍，那么你应该调大这个倍数并调整其他参数值，比如hfile.block.cache.size和hbase.regionserver.global.memstore.upperLimit/lowerLimit，以预留更多内存，防止HBase server OOM。  
  
**hbase.hregion.memstore.mslab.enabled**  
**默认值：**true  
**说明**：减少因内存碎片导致的Full GC，提高整体性能。  
**调优**：详见 <http://kenwublog.com/avoid-full-gc-in-hbase-using-arena-allocation>  
**其他**  
**启用LZO压缩**  
LZO对比Hbase默认的GZip，前者性能较高，后者压缩比较高，具体参见 [**Using LZO Compression**](http://wiki.apache.org/hadoop/UsingLzoCompression) **。**对于想提高HBase读写性能的开发者，采用LZO是比较好的选择。对于非常在乎存储空间的开发者，则建议保持默认。  
  
**不要在一张表里定义太多的Column Family**  
Hbase目前不能良好的处理超过包含2-3个CF的表。因为某个CF在flush发生时，它邻近的CF也会因关联效应被触发flush，最终导致系统产生更多IO。  
  
**批量导入**  
在批量导入数据到Hbase前，你可以通过预先创建regions，来平衡数据的负载。详见 [Table Creation: Pre-Creating Regions](http://hbase.apache.org/book.html#precreate.regions)  
  
**避免CMS concurrent mode failure**  
HBase使用CMS GC。默认触发GC的时机是当年老代内存达到90%的时候，这个百分比由 -XX:CMSInitiatingOccupancyFraction=N 这个参数来设置。concurrent mode failed发生在这样一个场景：  
当年老代内存达到90%的时候，CMS开始进行并发垃圾收集，于此同时，新生代还在迅速不断地晋升对象到年老代。当年老代CMS还未完成并发标记时，年老代满了，悲剧就发生了。CMS因为没内存可用不得不暂停mark，并触发一次stop the world（挂起所有jvm线程），然后采用单线程拷贝方式清理所有垃圾对象。这个过程会非常漫长。为了避免出现concurrent mode failed，建议让GC在未到90%时，就触发。  
通过设置 -XX:CMSInitiatingOccupancyFraction=N  
这个百分比， 可以简单的这么计算。如果你的 hfile.block.cache.size 和 hbase.regionserver.global.memstore.upperLimit 加起来有60%（默认），那么你可以设置 70-80，一般高10%左右差不多。  
**Hbase客户端优化AutoFlush**  
将[HTable](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html)的setAutoFlush设为false，可以支持客户端批量更新。即当Put填满客户端flush缓存时，才发送到服务端。  
默认是true。  
  
**Scan Caching**  
scanner一次缓存多少数据来scan（从服务端一次抓多少数据回来scan）。  
默认值是 1，一次只取一条。  
  
**Scan Attribute Selection**  
scan时建议指定需要的Column Family，减少通信量，否则scan操作默认会返回整个row的所有数据（所有Coulmn Family）。  
  
**Close ResultScanners**  
通过scan取完数据后，记得要关闭ResultScanner，否则RegionServer可能会出现问题（对应的Server资源无法释放）。  
  
**Optimal Loading of Row Keys**  
当你scan一张表的时候，返回结果只需要row key（不需要CF, qualifier,values,timestaps）时，你可以在scan实例中添加一个filterList，并设置 MUST\_PASS\_ALL操作，filterList中add [FirstKeyOnlyFilter](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/FirstKeyOnlyFilter.html)或[KeyOnlyFilter](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/KeyOnlyFilter.html)。这样可以减少网络通信量。  
  
**Turn off WAL on Puts**  
当Put某些非重要数据时，你可以设置writeToWAL(false)，来进一步提高写性能。writeToWAL(false)会在Put时放弃写WAL log。风险是，当RegionServer宕机时，可能你刚才Put的那些数据会丢失，且无法恢复。  
  
**启用Bloom Filter**  
[Bloom Filter](http://hbase.apache.org/book.html#blooms)通过空间换时间，提高读操作性能。  
  
转载请注明原文链接：<http://kenwublog.com/hbase-performance-tuning>  
  
感谢[嬴北望](http://lidejiasw.wordpress.com/)同学对”hbase.hregion.memstore.flush.size”和“hbase.hstore.blockingStoreFiles”错误观点的修正。