1. HBase LSMT模型理解
2. HBase适合顺序读写，不适合随机读写。
3. 协处理器学习

# 1.hbase日常操作及维护

一，日常维护的命令

1，major\_compact 'testtable'，通常生产环境会关闭自动major\_compact(配置文件中hbase.hregion.majorcompaction设为0)，选择一个晚上用户少的时间窗口手工major\_compact，如果hbase更新不是太频繁，可以一个星期对所有表做一次major\_compact，这个可以在做完一次major\_compact后，观看所有的storefile数量，如果storefile数量增加到major\_compact后的storefile的近二倍时，可以对所有表做一次major\_compact，时间比较长，操作尽量避免高锋期。

2，flush 'testtable'，将所有memstore刷新到hdfs，通常如果发现regionserver的内存使用过大，造成该机的regionserver很多线程block，可以执行一下flush操作，这个操作会造成hbase的storefile数量剧增，应尽量避免这个操作，还有一种情况，在hbase进行迁移的时候，如果选择拷贝文件方式，可以先停写入，然后flush所有表，拷贝文件。

3，balance\_switch true或者balance\_switch flase，配置master是否执行平衡各个regionserver的region数量，当我们需要维护或者重启一个regionserver时，会关闭balancer，这样就使得region在regionserver上的分布不均，这个时候需要手工的开启balance。

二，重启一个regionserver

bin/graceful\_stop.sh --restart --reload --debug nodename

这个操作是平滑的重启regionserver进程，对服务不会有影响，他会先将需要重启的regionserver上面的所有region迁移到其它的服务器，然后重启，最后又会将之前的region迁移回来，但我们修改一个配置时，可以用这种方式重启每一台机子，这个命令会关闭balancer，所以最后我们要在hbase shell里面执行一下balance\_switch true，对于hbase regionserver重启，不要直接kill进程，这样会造成在zookeeper.session.timeout这个时间长的中断，也不要通过bin/hbase-daemon.sh stop regionserver去重启，如果运气不太好，-ROOT-或者.META.表在上面的话，所有的请求会全部失败。

三，关闭下线一台regionserver

bin/graceful\_stop.sh –stop nodename

和上面一样，系统会在关闭之前迁移所有region，然后stop进程，同样最后我们要手工balance\_switch true，开启master的region均衡。

四，检查region是否正常以及修复

bin/hbase hbck (检查)

bin/hbase hbck -fix （修复）

会返回所有的region是否正常挂载，如没有正常挂载可以使用下一条命令修复，如果还是不能修复，那需要看日志为什么失败，手工处理。

五，hbase的迁移

1，copytable方式

bin/hbase org.apache.hadoop.hbase.mapreduce.CopyTable --peer.adr=zookeeper1,zookeeper2,zookeeper3:/hbase 'testtable'

目前0.92之前的版本的不支持多版本的复制，0.94已经支持多个版本的复制。当然这个操作需要添加hbase目录里的conf/mapred-site.xml，可以复制hadoop的过来。

2，Export/Import

bin/hbase org.apache.hadoop.hbase.mapreduce.Export testtable /user/testtable [versions] [starttime] [stoptime]

bin/hbase org.apache.hadoop.hbase.mapreduce.Import testtable /user/testtable

跨版本的迁移，我觉得是一个不错的选择，而且copytable不支持多版本，而export支持多版本，比copytable更实用一些。

3，直接拷贝hdfs对应的文件

首先拷贝hdfs文件，如bin/hadoop distcp hdfs://srcnamenode:9000/hbase/testtable/ hdfs://distnamenode:9000/hbase/testtable/

然后在目的hbase上执行bin/hbase org.jruby.Main bin/add\_table.rb /hbase/testtable生成meta信息后，重启hbase

这个操作是简单的方式，操作之前可以关闭hbase的写入，执行flush所有表（上面有介绍）,再distcp拷贝，如果hadoop版本不一致，可以用hftp接口的方式，我推荐使用这种方式，成本低。

**HBase backup master**

[cdh4@ecmserver conf]$ vi backup-masters

server86

server87

# 2.HBase理解

## hbase中多对多的关系

假如现在有三张表如下：

1.商品表：商品ID,名称，描述

2.超市表：超市ID,名称，地址

3.关系表：商品ID，超市ID

**问：**

1.如何在Hbase中设计此关系表？

2.假如我修改了商品，或者超市信息，在对应Hbase中如何进行操作，对应关系中数据保持一致？

**答：**

hbase是nosql数据库，不擅长关系型数据处理。2，从表名来看，似乎数据量也大不到哪里去，用hbase的意义在哪，mysql，postgres扛不住？3，如果你真要这么玩的话，前两张表id是唯一的，直接是rowkey，第三张表，两者加起来做rowkey，正着来一下，反着来一下。或者两个id单独做row，列簇里面记录关联的另一个id。

## What is meant by “HDFS lacks random read and write access”?

The default HDFS block size is 64 MB. So you cannot read one line here, one line there. You always read and write 64 MB blocks. This is fine when you want to process the whole file. But it makes HDFS unsuitable for some applications, like where you want to use an index to look up small records.

HBase on the other hand is great for this. If you want to read a small record, you will only read that small record.

HBase uses HDFS as its backing store. So how does it provide efficient record-based access?

HBase loads the tables from HDFS to memory or local disk, so most reads do not go to HDFS. Mutations are stored first in an append-only journal. When the journal gets large, it is built into an "addendum" table. When there are too many addendum tables, they all get compacted into a brand new primary table. For reads, the journal is consulted first, then the addendum tables, and at last the primary table. This system means that we only write a full HDFS block when we have a full HDFS block's worth of changes.

## 缓存查询结果

对于频繁查询HBase的应用场景，可以考虑在应用程序中做缓存，当有新的查询请求时，首先在缓存中查找，如果存在则直接返回，不再查询HBase；否则对HBase发起读请求查询，然后在应用程序中将查询结果缓存起来。至于缓存的替换策略，可以考虑LRU等常用的策略。

## Blockcache

HBase上Regionserver的内存分为两个部分，一部分作为Memstore，主要用来写；另外一部分作为BlockCache，主要用于读。

写请求会先写入Memstore，Regionserver会给每个region提供一个Memstore，当Memstore满64MB以后，会启动 flush刷新到磁盘。当Memstore的总大小超过限制时（heapsize \* hbase.regionserver.global.memstore.upperLimit \* 0.9），会强行启动flush进程，从最大的Memstore开始flush直到低于限制。

读请求先到Memstore中查数据，查不到就到BlockCache中查，再查不到就会到磁盘上读，并把读的结果放入BlockCache。由于BlockCache采用的是LRU策略，因此BlockCache达到上限(heapsize \* hfile.block.cache.size \* 0.85)后，会启动淘汰机制，淘汰掉最老的一批数据。

一个Regionserver上有一个BlockCache和N个Memstore，它们的大小之和不能大于等于heapsize \* 0.8，否则HBase不能启动。默认BlockCache为0.2，而Memstore为0.4。对于注重读响应时间的系统，可以将 BlockCache设大些，比如设置BlockCache=0.4，Memstore=0.39，以加大缓存的命中率。

有关BlockCache机制，请参考这里：HBase的Block cache，HBase的blockcache机制，hbase中的缓存的计算与使用。

# 3.代码分析

批处理动作有Put,Delete,Get。同一个RegionServer的所有动作只形成一次RPC调用。

客户端会缓存Region信息，也可以清除指定或所有的缓存信息。

ServerManager--- 负责管理 Region server 信息，比如每个 Region server 的 HServerInfo( 这个对象包含 HServerAddress 和 startCode), 已 load Region 个数，死亡的 Region server 列表。

MetaScanner--- 定期扫描 meta region, 以发现没有分配的 user region

# 4.HBase使用

## 数据量统计

### Use RowCounter in HBase

RowCounter is a mapreduce job to count all the rows of a table. This is a good utility to use as a sanity check to ensure that HBase can read all the blocks of a table if there are any concerns of metadata inconsistency. It will run the mapreduce all in a single process but it will run faster if you have a MapReduce cluster in place for it to exploit.

$ hbase org.apache.hadoop.hbase.mapreduce.RowCounter <tablename>

Usage: RowCounter [options]

<tablename> [

--starttime=[start]

--endtime=[end]

[--range=[startKey],[endKey]]

[<column1> <column2>...]

]

### Shell

count 't1', CACHE => 10000

## Get output from scans in hbase shell

echo "scan 'foo'" | ./hbase shell > myText

# 5.优化

## hbase性能调优

### 服务端调优

**参数配置**

hbase.regionserver.handler.count：该设置决定了处理RPC的线程数量，默认值是10，通常可以调大，比如：150，当请求内容很大（上MB，比如大的put、使用缓存的scans）的时候，如果该值设置过大则会占用过多的内存，导致频繁的GC，或者出现OutOfMemory，因此该值不是越大越好。

hbase.hregion.max.filesize ：配置region大小，0.94.12版本默认是10G，region的大小与集群支持的总数据量有关系，如果总数据量小，则单个region太大，不利于并行的数据处理，如果集群需支持的总数据量比较大，region太小，则会导致region的个数过多，导致region的管理等成本过高，如果一个RS配置的磁盘总量为3T\*12=36T数据量，数据复制3份，则一台RS服务器可以存储10T的数据，如果每个region最大为10G，则最多1000个region，如此看，94.12的这个默认配置还是比较合适的，不过如果要自己管理split，则应该调大该值，并且在建表时规划好region数量和rowkey设计，进行region预建，做到一定时间内，每个region的数据大小在一定的数据量之下，当发现有大的region，或者需要对整个表进行region扩充时再进行split操作，一般提供在线服务的hbase集群均会弃用hbase的自动split，转而自己管理split。

hbase.hregion.majorcompaction：配置major合并的间隔时间，默认为1天，可设置为0，禁止自动的major合并，可手动或者通过脚本定期进行major合并，有两种compact：minor和major，minor通常会把数个小的相邻的storeFile合并成一个大的storeFile，minor不会删除标示为删除的数据和过期的数据，major会删除需删除的数据，major合并之后，一个store只有一个storeFile文件，会对store的所有数据进行重写，有较大的性能消耗。

hbase.hstore.compactionThreshold：HStore的storeFile数量>= compactionThreshold配置的值，则可能会进行compact，默认值为3，可以调大，比如设置为6，在定期的major compact中进行剩下文件的合并。

hbase.hstore.blockingStoreFiles：HStore的storeFile的文件数大于配置值，则在flush memstore前先进行split或者compact，除非超过hbase.hstore.blockingWaitTime配置的时间，默认为7，可调大，比如：100，避免memstore不及时flush，当写入量大时，触发memstore的block，从而阻塞写操作。

hbase.regionserver.global.memstore.upperLimit：默认值0.4，RS所有memstore占用内存在总内存中的upper比例，当达到该值，则会从整个RS中找出最需要flush的region进行flush，直到总内存比例降至该数限制以下，并且在降至限制比例以下前将阻塞所有的写memstore的操作，在以写为主的集群中，可以调大该配置项，不建议太大，因为block cache和memstore cache的总大小不会超过0.8，而且不建议这两个cache的大小总和达到或者接近0.8，避免OOM，在偏向写的业务时，可配置为0.45，memstore.lowerLimit保持0.35不变，在偏向读的业务中，可调低为0.35，同时memstore.lowerLimit调低为0.3，或者再向下0.05个点，不能太低，除非只有很小的写入操作，如果是兼顾读写，则采用默认值即可。

hbase.regionserver.global.memstore.lowerLimit：默认值0.35，RS的所有memstore占用内存在总内存中的lower比例，当达到该值，则会从整个RS中找出最需要flush的region进行flush，配置时需结合memstore.upperLimit和block cache的配置。

file.block.cache.size：RS的block cache的内存大小限制，默认值0.25，在偏向读的业务中，可以适当调大该值，具体配置时需试hbase集群服务的业务特征，结合memstore的内存占比进行综合考虑。

hbase.hregion.memstore.flush.size：默认值128M，单位字节，超过将被flush到hdfs，该值比较适中，一般不需要调整。

hbase.hregion.memstore.block.multiplier：默认值2，如果memstore的内存大小已经超过了hbase.hregion.memstore.flush.size的2倍，则会阻塞memstore的写操作，直到降至该值以下，为避免发生阻塞，最好调大该值，比如：4，不可太大，如果太大，则会增大导致整个RS的memstore内存超过memstore.upperLimit限制的可能性，进而增大阻塞整个RS的写的几率。如果region发生了阻塞会导致大量的线程被阻塞在到该region上，从而其它region的线程数会下降，影响整体的RS服务能力。

hfile.block.index.cacheonwrite：在index写入的时候允许put无根（non-root）的多级索引块到block cache里，默认是false，设置为true，或许读性能更好，但是是否有副作用还需调查。

hbase.regionserver.regionSplitLimit：控制最大的region数量，超过则不可以进行split操作，默认是Integer.MAX，可设置为1，禁止自动的split，通过人工，或者写脚本在集群空闲时执行。如果不禁止自动的split，则当region大小超过hbase.hregion.max.filesize时会触发split操作（具体的split有一定的策略，不仅仅通过该参数控制，前期的split会考虑region数据量和memstore大小），每次flush或者compact之后，regionserver都会检查是否需要Split，split会先下线老region再上线split后的region，该过程会很快，但是会存在两个问题：1、老region下线后，新region上线前client访问会失败，在重试过程中会成功但是如果是提供实时服务的系统则响应时长会增加；2、split后的compact是一个比较耗资源的动。

**Jvm调整**

内存大小：master默认为1G，可增加到2G，regionserver默认1G，可调大到10G，或者更大，zk并不耗资源，可以不用调整。

**其它调优**

列族、rowkey要尽量短，每个cell值均会存储一次列族名称和rowkey，甚至列名称也要尽量短。短的列族名称、rowkey、列名称对最终的文件内容大小影响很大。

RS的region数量：一般每个RegionServer不要过1000，过多的region会导致产生较多的小文件，从而导致更多的compact，当有大量的超过5G的region并且RS总region数达到1000时，应该考虑扩容。

**建表时**

如果不需要多版本，则应设置version=1；

开启lzo或者snappy压缩，压缩会消耗一定的CPU，但是，磁盘IO和网络IO将获得极大的改善，大致可以压缩4~5倍；

合理的设计rowkey，在设计rowkey时需充分的理解现有业务并合理预见未来业务，不合理的rowkey设计将导致极差的hbase操作性能；

合理的规划数据量，进行预分区，避免在表使用过程中的不断split，并把数据的读写分散到不同的RS，充分的发挥集群的作用；

列族名称尽量短，比如：“f”，并且尽量只有一个列族；

视场景开启bloomfilter，优化读性能。

### Client端调优

hbase.client.write.buffer：写缓存大小，默认为2M，推荐设置为6M，单位是字节，当然不是越大越好，如果太大，则占用的内存太多；

hbase.client.scanner.caching：scan缓存，默认为1，太小，可根据具体的业务特征进行配置，原则上不可太大，避免占用过多的client和rs的内存，一般最大几百，如果一条数据太大，则应该设置一个较小的值，通常是设置业务需求的一次查询的数据条数，比如：业务特点决定了一次最多100条，则可以设置为100。

设置合理的超时时间和重试次数，具体的内容会在后续的blog中详细讲解。

**client应用读写分离**

读和写分离，位于不同的tomcat实例，数据先写入redis队列，再异步写入hbase，如果写失败再回存redis队列，先读redis缓存的数据（如果有缓存，需要注意这里的redis缓存不是redis队列），如果没有读到再读hbase。

当hbase集群不可用，或者某个RS不可用时，因为HBase的重试次数和超时时间均比较大（为保证正常的业务访问，不可能调整到比较小的值，如果一个RS挂了，一次读或者写，经过若干重试和超时可能会持续几十秒，或者几分钟），所以一次操作可能会持续很长时间，导致tomcat线程被一个请求长时间占用，tomcat的线程数有限，会被快速占完，导致没有空余线程做其它操作，读写分离后，写由于采用先写redis队列，再异步写hbase，因此不会出现tomcat线程被占满的问题， 应用还可以提供写服务，如果是充值等业务，则不会损失收入，并且读服务出现tomcat线程被占满的时间也会变长一些，如果运维介入及时，则读服务影响也比较有限。

如果把org.apache.hadoop.hbase.client.HBaseAdmin配置为spring的bean，则需配置为懒加载，避免在启动时链接hbase的Master失败导致启动失败，从而无法进行一些降级操作。

**Scan查询编程优化**

调整caching；

如果是类似全表扫描这种查询，或者定期的任务，则可以设置scan的setCacheBlocks为false，避免无用缓存；

关闭scanner，避免浪费客户端和服务器的内存；

限定扫描范围：指定列簇或者指定要查询的列；

如果只查询rowkey时，则使用KeyOnlyFilter可大量减少网络消耗；

**作为hbase依赖的状态协调者ZK和数据的存储则HDFS，也需要调优：**

zookeeper.session.timeout：默认值3分钟，不可配置太短，避免session超时，hbase停止服务，也不可配置太长，如果太长，当rs挂掉，zk不能快速知道，从而导致master不能及时对region进行迁移。

zookeeper数量：至少5个节点。给每个zookeeper 1G左右的内存，最好有独立的磁盘。 (独立磁盘可以确保zookeeper不受影响).如果集群负载很重，不要把Zookeeper和RegionServer运行在同一台机器上面。就像DataNodes 和 TaskTrackers一样，只有超过半数的zk存在才会提供服务，比如：共5台，则最多只运行挂2台，配置4台与3台一样，最多只运行挂1台。

hbase.zookeeper.property.maxClientCnxns：zk的最大连接数，默认为300，可配置上千。

**hdfs调优**

dfs.name.dir： namenode的数据存放地址，可以配置多个，位于不同的磁盘并配置一个NFS远程文件系统，这样nn的数据可以有多个备份。

dfs.data.dir：dn数据存放地址，每个磁盘配置一个路径，这样可以大大提高并行读写的能力。

dfs.namenode.handler.count：nn节点RPC的处理线程数，默认为10，需提高，比如：60。

dfs.datanode.handler.count：dn节点RPC的处理线程数，默认为3，需提高，比如：20。

dfs.datanode.max.xcievers：dn同时处理文件的上限，默认为256，需提高，比如：8192。

dfs.block.size：dn数据块的大小，默认为64M，如果存储的文件均是比较大的文件则可以考虑调大，比如，在使用hbase时，可以设置为128M，注意单位是字节。

dfs.balance.bandwidthPerSec：在通过start-balancer.sh做负载均衡时控制传输文件的速度，默认为1M/s，可配置为几十M/s，比如：20M/s。

dfs.datanode.du.reserved：每块磁盘保留的空余空间，应预留一些给非hdfs文件使用，默认值为0。

dfs.datanode.failed.volumes.tolerated：在启动时会导致dn挂掉的坏磁盘数量，默认为0，即有一个磁盘坏了，就挂掉dn，可以不调整。

（参考：<http://itindex.net/detail/49632-hbase->性能调优）

## HBase 负载均衡

HBase 可以根据当前集群的负载以region为单位进行rebalance。在HMaster中，后台会起一个线程定期检查是否需要进行rebalance，线 程叫做BalancerChore。线程每隔 hbase.balancer.period会定期执行 master.balance()函数，配置项默认300000毫秒，5分钟。每次balance最多执行。

hbase.balancer.max.balancing，如果没有配置，则使用 hbase.balancer.period配置项的值。master.balance()首先通过loadBalancerTracker去zk上看是 否load balance开启，如果开启，则从AssignmentManager中检查当前是否有region处于in transition状态，如果有，则直接返回。否则将集群的状态给balancer以便后续做决策，HMaster的 assignmentManager成员内部维护着一个表在哪些机器上，这些机器上分别有哪些region。对于每张表，都会执行 balancer.balanceCluster()方法。HBase中load balance的策略是可插拔的，开发者可以根据自己业务的需求来开发自己的load balance策略。在HBase中，是通过接口LoadBalancer类实现的。具体使用哪个load balance策略由配置项hbase.master.loadbalancer.class决定，默认使用 StochasticLoadBalancer。所有的逻辑都在StochasticLoadBalancer这个负载均衡器的 balanceCluster()方法中。

StochasticLoadBalancer负载均衡器首先会根据每个region server上的region个数作决定要不要进行rebalance，具体方法是算出所有server的平均region个数，然后根据配置项 hbase.regions.slop产生一个区间[floor(average \* (1-slop)), ceil(average \* (1+slop))]，配置项默认0.2，如果region 个数最多的region server不比右区间大，并且region个数最少的region server不比左区间小，则说明region个数比较平均，就不进行rebalance，直接退出，等待下次调度。否则，计算当前集群状态的cost 值，这个cost值的计算会考虑到移动region的成本，region 本地化策略，region count分布，每个server上table的分布等做一个加权平均。然后一共迭代computedMaxSteps次，次数由配置项 hbase.master.balancer.stochastic.maxSteps和 hbase.master.balancer.stochastic.stepsPerRegion，还有当前集群的region个数，server个数 共同决定。每次迭代，都会随机选择一种pick region的策略，一共有三种，分别为RandomRegionPicker，LoadPicker和LocalityBasedPicker。随机选 定一个picker策略后，这个picker就会从集群中选出两个用于的region或者选出一个用于迁移到其他server的region，然后更新集 群状态的数据结构，重新计算当前集群状态的cost值，如果发现新的cost比原来的小，则说明，这种region的交换或者迁移是有效的。每次迭代都是 基于上次的成果，总共做computedMaxSteps。最后产生出一系列的plan，每个plan就是交换region或者迁移region。对于所 有的表都做一次，把所有的plan都放入AssignmentManager的regionsPlans中。然后对于每个plan，都调用assignmentManager.balance(plan)，这个函数会 调用unassign()方法，首先在zk上为这个region创建/hbase/region-in-transition /region\_encoded\_name节点，节点内容为这个原来在某个server上的region处于closing状态了，然后给这个 region原来所在的server发送close region命令对region进行卸载，随后再调用public void assign(HRegionInfo region, boolean setOfflineInZK)给region的目标region server发送open region的命令，目标region server是从regionPlans中查到的。最后删除zk上的节点。其中，每次做完一个plan后都会检查是否时间到了。

**Region负载均衡**

打开开关：balance\_switch true

执行：balancer

There is a parameter in hbase-site.xml:

<property>

<name>hbase.master.loadbalance.bytable</name>

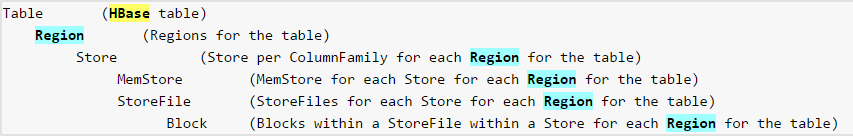
<value>true</value>

</property>

By default, it's false. So the balance between region servers is good, but load balance at the table level is not taken into account. Set this value to true, HBase will balance the region over region servers at the table level.

## Region 大小

Region是表获取和分布的基本元素，由每个列族的一个Store组成。对象层级图如下：



Region的大小是一个棘手的问题，需要考量如下几个因素。

region数目太多就会造成性能下降，现在比以前好多了。但是对于同样大小的数据，700个region比3000个要好。

region数目太少就会妨碍可扩展性，降低并行能力。有的时候导致压力不够分散。这就是为什么，你向一个10节点的HBase集群导入200MB的数据，大部分的节点是idle的。

RegionServer中1个region和10个region索引需要的内存量没有太多的差别。

最好是使用默认的配置，可以把热的表配小一点(或者受到split热点的region把压力分散到集群中)。如果你的cell的大小比较大(100KB或更大)，就可以把region的大小调到1GB。region的最大大小在hbase配置文件中定义：

<property>

<name>hbase.hregion.max.filesize</name>

<value>10 \* 1024 \* 1024 \* 1024</value>

</property>

说明：

当region的大小超过了上面配置的值的时候，该region就会被拆分，具体的拆分策略见下文。

上面的值也可以针对每个表单独设置，例如在hbase shell中设置：

create 't','f'

disable 't'

alter 't', METHOD => 'table\_att', MAX\_FILESIZE => '134217728'

enable 't'

## Region 拆分策略

Region的分割操作是不可见的，因为Master不会参与其中。RegionServer拆分region的步骤是，先将该region下线，然后拆分，将其子region加入到META元信息中，再将他们加入到原本的RegionServer中，最后汇报Master。执行split的线程是CompactSplitThread。

自定义拆分策略

可以通过设置RegionSplitPolicy的实现类来指定拆分策略，RegionSplitPolicy类的实现类有：

ConstantSizeRegionSplitPolicy

IncreasingToUpperBoundRegionSplitPolicy

DelimitedKeyPrefixRegionSplitPolicy

KeyPrefixRegionSplitPolicy

对于split，并不是设置了hbase.hregion.max.filesize（默认10G）为很大就保证不split了，需要有以下的算法：

IncreasingToUpperBoundRegionSplitPolicy，0.94.0默认region split策略。根据公式min(r2\*flushSize，maxFileSize)确定split的maxFileSize，其中r为在线region个数，maxFileSize由hbase.hregion.max.filesize指定。

ConstantSizeRegionSplitPolicy，仅仅当region大小超过常量值（hbase.hregion.max.filesize大小）时，才进行拆分。

DelimitedKeyPrefixRegionSplitPolicy，保证以分隔符前面的前缀为splitPoint，保证相同RowKey前缀的数据在一个Region中

KeyPrefixRegionSplitPolicy，保证具有相同前缀的row在一个region中（要求设计中前缀具有同样长度）。指定rowkey前缀位数划分region，通过读取table的prefix\_split\_key\_policy.prefix\_length属性，该属性为数字类型，表示前缀长度，在进行split时，按此长度对splitPoint进行截取。此种策略比较适合固定前缀的rowkey。当table中没有设置该属性，或其属性不为Integer类型时，指定此策略效果等同与使用IncreasingToUpperBoundRegionSplitPolicy。

IncreasingToUpperBoundRegionSplitPolicy

这是0.94.0默认region split策略。根据根据公式min(r2\*flushSize，maxFileSize)确定split的maxFileSize

**配置拆分策略**

你可以在hbase配置文件中定义全局的拆分策略，设置hbase.regionserver.region.split.policy的值即可，也可以在创建和修改表时候指定：

// 更新现有表的split策略

HBaseAdmin admin = new HBaseAdmin( conf);

HTable hTable = new HTable( conf, "test" );

HTableDescriptor htd = hTable.getTableDescriptor();

HTableDescriptor newHtd = new HTableDescriptor(htd);

newHtd.setValue(HTableDescriptor. SPLIT\_POLICY, KeyPrefixRegionSplitPolicy.class .getName());// 指定策略

newHtd.setValue("prefix\_split\_key\_policy.prefix\_length", "2");

newHtd.setValue("MEMSTORE\_FLUSHSIZE", "5242880"); // 5M

admin.disableTable( "test");

admin.modifyTable(Bytes. toBytes("test"), newHtd);

admin.enableTable( "test");

说明：

上面的不同策略可以在不同的业务场景下使用，特别是第三种和第四种一般关注和使用的比较少。

如果想关闭自动拆分改为手动拆分，建议同时修改hbase.hregion.max.filesize和hbase.regionserver.region.split.policy值。

base.regionserver.regionSplitLimit：控制最大的region数量，超过则不可以进行split操作，默认是Integer.MAX，可设置为1，禁止自动的split，通过人工，或者写脚本在集群空闲时执行。如果不禁止自动的split，则当region大小超过hbase.hregion.max.filesize时会触发split操作（具体的split有一定的策略，不仅仅通过该参数控制，前期的split会考虑region数据量和memstore大小），每次flush或者compact之后，regionserver都会检查是否需要Split，split会先下线老region再上线split后的region，该过程会很快，但是会存在两个问题：1、老region下线后，新region上线前client访问会失败，在重试过程中会成功但是如果是提供实时服务的系统则响应时长会增加；2、split后的compact是一个比较耗资源的动作。

**HBase 0.94中的Split策略**

HBase 0.94之前版本中，split使用的是ConstantSizeRegionSplitPolicy。当region中文件大小超过配置中所指定大小时，会进行切分。

而在0.94版本之后，默认split策略修改为了IncreasingToUpperBoundRegionSplitPolicy。该策略使用了另一种方法来计算是否应当切割，导致原先的参数失效。

该方法中的分配策略，是根据table中region的个数平方，乘以memstore的大小。得出应当切分的大小。

假设memstore size配置为128M，则在memstore第一次刷入HFile数据时，进行第一次split，1 \* 1 \* 128M = 128M。

当region数达到2个时，2 \* 2 \* 128M = 512M。

当region数达到3个时，3 \* 3 \* 128M = 1152M。

依此类推。

当region个数到达30个时，30 \* 30 \* 128 = 107648M = 105.1G。即在此时，region的切分大小已经超过了我们原先在ConstantSizeRegionSplitPolicy策略中设置的100G大小。

简单分析

对这种策略进行简单的分析，可以看到，在数据写入初期，这种策略可以快速的对现有region进行split，使得在一开始就可以将热点region切分到多个server上。同时由于region size较小，也可以避免split操作对写入的阻塞。

而在后期，当region数量逐渐增多，单个region size逐渐增大时，split频率会急速减少，避免在region过大时频繁split的情况。

这种策略一方面在数据量增大的情况下减少了region的切分次数，达到了我们期望的尽量减少split的需求，避免对写入造成影响。同时在初期的快速切分，在基本不影响写入的同时，也减少了我们原先需要手动操作split的问题。可以认为，这种策略是符合我们需求的。当然，还需要进一步的测试来进行验证。

## store file合并（compaction）

hbase为了防止小文件（被刷到磁盘的menstore）过多，以保证保证查询效率，hbase需要在必要的时候将这些小的store file合并成相对较大的store file，这个过程就称之为compaction。

在hbase中，主要存在两种类型的compaction：minor compaction和major compaction。

major compaction 的功能是将所有的store file合并成一个，触发major compaction的可能条件有：major\_compact 命令、majorCompact() API、region server自动运行（相关参数：hbase.hregion.majoucompaction 默认为24 小时、hbase.hregion.majorcompaction.jetter 默认值为0.2 防止region server 在同一时间进行major compaction）。hbase.hregion.majorcompaction.jetter参数的作用是：对参数hbase.hregion.majoucompaction 规定的值起到浮动的作用，假如两个参数都为默认值24和0,2，那么major compact最终使用的数值为：19.2~28.8 这个范围。

minor compaction的运行机制要复杂一些，它由一下几个参数共同决定：

hbase.hstore.compaction.min :默认值为 3，表示至少需要三个满足条件的store file时，minor compaction才会启动

hbase.hstore.compaction.max 默认值为10，表示一次minor compaction中最多选取10个store file

hbase.hstore.compaction.min.size 表示文件大小小于该值的store file 一定会加入到minor compaction的store file中

hbase.hstore.compaction.max.size 表示文件大小大于该值的store file 一定会被minor compaction排除

hbase.hstore.compaction.ratio 将store file 按照文件年龄排序（older to younger），minor compaction总是从older store file开始选择，如果该文件的size 小于它后面hbase.hstore.compaction.max 个store file size 之和乘以 该ratio，则该store file 也将加入到minor compaction 中。

如果对minor compaction过程还是不了解，可以去看hbase中关于minor compaction 的源码，或者：http://blog.csdn.net/macyang/article/details/6423078

六、region split

描述： 这个是最最根蒂根基的题目了，当然，hbase大多半的bug就是呈如今这里。当region中的storefile的总大小大于hbase.hregion.max.filesize设置的大小时，这个region就要split，在这个region一分为二的过程中，进行的很是快，因为hbase只须要为两个新的region（在hbase中也成为daughters）建两个reference file罢了.当两个daughters筹办上可以上线的时辰，后台线程会把父region的store files写到两个子region中，同时也会调换掉reference file。最后master会看看到底需不须要做load balance。

**Compactions**

描述：当region内的storefile的数量大于hbase.hstore.compaction.min（hbase.hstore.compactionThreshold）时，就会触发minor compaction，而每次minor compaction所包含的最大文件数由hbase.hstore.compaction.max指定。并且，hbase.hstore.compaction.min.size（凡是设置成和memstore flush的值一样）与hbase.hstore.compaction.max.size（凡是设置为Long.MAX\_VALUE）会进一步限制参与compact的文件。与minor compaction不合， major compact会把所有的store里面的文件合成一个大文件，compact的类型在搜检阶段就已经断定好了。以下几种景象会触发搜检compaction，1、shell中的compact或者major\_compact恳求 2、memstore flush后 3、恳求响应的api中的majorCompact（ ） 4、后台线程轮询，时候由hbase.server.thread.wakefrequency和hbase.server.thread.wakefrequency，multiplier相乘所得。假如你在shell中恳求了major\_compact，或者恳求了api中的majorCompact，都邑强迫履行majorCompact，在其它景象下，rs会根据hbase.hregion.majorcompaction来断定是否应当履行major。

## 守护进程及内存调优

**HRegionServer**

这个进程是HBase中的核心守护进程，原则上是每个slave启动一个HRegionServer,但多种情况可能导致HRegionServer 意外退出，下面举几个简单的方面:

网络不好，导致RegionServer 和 HMaster通信超时，RegionServer被认为已经挂掉，从而退出集群 --网络问题，无法从软件方面解决，关于通信超时的设置下面做个简单介绍

Java full GC ，这个过程会block所有的线程，如果此事件过长，导致Session expired 会话过期，导致退出集群--下文会阐述

各节点时间不一致，导致RegionServer 退出。-- hbase.master.maxclockskew 增大容忍度，默认是30s，但不要太大，毕竟时间不一致是不正常现象，可将所有节点和同一服务器时间做同步，也可以和时间服务器同步。

第一种情况 和其它原因导致的RegionServer 超时挂掉的问题，我们要首先要调高Session的容忍度，默认180000其实这个回话有效期已经够长的了，但是有的集群是可以

降低了这个值，可能会造成Session 超时，这个参数是 zookeeper.session.timeout 默认18000。

第二种情况是要讨论的，导致产生这个问题的主要原因是很多，产生的情景很多，比如在做 major compact的过程中，时间过长，导致Full GC等，那就尽量去减少这种情况的发生。二个方面

适度增大守护进程的HeapSize

调整内存回收参数

第一个方面：Hbase　默认各守护进程为1G 在hbase-env.sh中有配置 export HBASE\_HEAPSIZE=1000，当我们启动hbase各守护进程的时候，那所有的hbase守护进程都

将是1000的heapsize，对于有的进程，够用，但有些进程取远远不够，我们可以考虑增大此参数，比如export HBASE-HEAPSIZE=6000 那就把守护进程的heap 内存调大到

6G,但是这样会有问题，有些进程不需要这么多，虽然设置的比较大不影响内存的实际占用，但却混淆了对各进程内存占用的认识。所以上述参数不做改变，在下面的参数中

修改守护进程Heap 内存。

export HBASE\_MASTER\_OPTS="$HBASE\_MASTER\_OPTS $HBASE\_JMX\_BASE -Xmx2000m -Xms2000m -Xmn750m -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:CMSInitatingOccupancyFraction=70"

export HBASE\_REGIONSERVER\_OPTS="$HBASE\_REGIONSERVER\_OPTS $HBASE\_JMX\_BASE -Xmx6000m -Xms6000m -Xmn2250m -XX:+UseParNewGC

-XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=70"

export HBASE\_THRIFT\_OPTS="$HBASE\_THRIFT\_OPTS $HBASE\_JMX\_BASE -Xms100m -Xmx2000m"

export HBASE\_ZOOKEEPER\_OPTS="$HBASE\_ZOOKEEPER\_OPTS $HBASE\_JMX\_BASE -Xms100m -Xmx2000m"

我们分别对各守护进程设置堆内存 其中-Xmx 表示最大可用内存，-Xms表示出事分配内存 -Xmn 表示 年轻代堆内存分配,这个值网上有的建议按照3/3 总heapsize来设

置，因为是经验值，暂无法考证合理性，更多详细的堆内存分配参数，本地不做过多阐述，后面有机会可做一个单元来解释。那其它参数是什么意思呢?

-XX：+UseParNewGC 等，这就到了我们说的第二个方面：

第二个方面：调整内存回收参数，比如-XX:+UseParNewGC 表示年轻带内存回收策略采用并发收集，此参数在JDK5.0已经自动配置，不需再手动配置；

-XX:+UseConcMarkSweepGC 表示年老代并发收集；

-XX:+CMSInitatingOccupancyFraction 表示年老代内存占用超过此比例即开始做CMS，这个参数很重要在JDK 5.0以后此值默认是90 也就是当年老代对内存占用90%以上时，

才开始做内存收集，而此时剩余的10%依然接受从年轻代迁移过来的对象，迁移过快，导致年老代heap 100%时，Full GC 即开始，才是会暂停所有的任务，直至Full GC 完

成，此时是造成RegionServer 意外退出的元凶，那为了安全起见，在调大堆内存的情况下 蒋此值降低到一个较低的阀值，减少Full GC的产生，那我建议此值设70%。

## Scan慢优化

* TurnOff blockcache at time of scan (it is churning ur heap memory)
* Figure out whats the size of ur record , if its > 1 MB , please increase hbase.scanner.timeout period scan.setCacheBlocks(false);
* scan.setCaching(x) x \* record size what is getting fetched in one short , make sure it is close to 1 MB .
* some necessary check : make sure regions for the Tabled being scanned are equally distributed across regions .

(If you have done bulk load run a Major compaction Once )

* the key is to separate "hot" content from "cold" content into separate column families. Column families are used to store columns in separate HFiles, so we can use one column family for indexed (or frequently read) columns, and one other column family (thus file) for all other columns.

# 6.HBase问题总结

## RegionServer挂掉

日志如下：



**原因：**

GC超时，导致RegionServer与ZK断开，造成块丢失，RegionServer自动关闭

**解决：**

1. 调小JVM可用内存，防止GC时间过长
2. 写个定时任务，定期检查regionServer运行状态，挂掉时自动启动



1. 不要在HBase同一个集群跑重CPU任务
2. 配置zookeeper.session.timeout，增大超时时间间隔

## HBase snappy压缩无法使用

日志如下：



1.确保LD\_LIBRARY\_PATH和JAVA\_LIBRARY\_PATH包括的native目录下有libsnappy.so\*\*文件

2.编辑hadoop-env.sh，新增以下内容并重启：

export LD\_LIBRARY\_PATH=${LD\_LIBRARY\_PATH}:/usr/lib/HDH/hadoop-2.5.0-hdh3.0.0/lib/native

export JAVA\_LIBRARY\_PATH=${JAVA\_LIBRARY\_PATH}:/usr/lib/HDH/hadoop-2.5.0-hdh3.0.0/lib/native

## Zookeeper超时

工作中运行hbase报的zookeeper异常信息：

2013-06-28 18:26:59,946 WARN org.apache.hadoop.hbase.zookeeper.ZKUtil: master:60000-0x33f7eb9d0650002-0x33f7eb9d0650002-0x33f7eb9d0650002 Unable to get data of znode /hbase/master

org.apache.zookeeper.KeeperException$SessionExpiredException: KeeperErrorCode = Session expired for /hbase/master

at org.apache.zookeeper.KeeperException.create(KeeperException.java:118)

at org.apache.zookeeper.KeeperException.create(KeeperException.java:42)

at org.apache.zookeeper.ZooKeeper.getData(ZooKeeper.java:927)

at org.apache.hadoop.hbase.zookeeper.ZKUtil.getDataInternal(ZKUtil.java:577)

at org.apache.hadoop.hbase.zookeeper.ZKUtil.getDataAndWatch(ZKUtil.java:554)

at org.apache.hadoop.hbase.zookeeper.ZKUtil.getDataAsAddress(ZKUtil.java:648)

at org.apache.hadoop.hbase.master.ActiveMasterManager.stop(ActiveMasterManager.java:202)

at org.apache.hadoop.hbase.master.HMaster.run(HMaster.java:318)

2013-06-28 18:26:59,946 ERROR org.apache.hadoop.hbase.zookeeper.ZooKeeperWatcher: master:60000-0x33f7eb9d0650002-0x33f7eb9d0650002-0x33f7eb9d0650002 Received unexpected KeeperException, re-throwing exception

org.apache.zookeeper.KeeperException$SessionExpiredException: KeeperErrorCode = Session expired for /hbase/master

at org.apache.zookeeper.KeeperException.create(KeeperException.java:118)

at org.apache.zookeeper.KeeperException.create(KeeperException.java:42)

at org.apache.zookeeper.ZooKeeper.getData(ZooKeeper.java:927)

at org.apache.hadoop.hbase.zookeeper.ZKUtil.getDataInternal(ZKUtil.java:577)

at org.apache.hadoop.hbase.zookeeper.ZKUtil.getDataAndWatch(ZKUtil.java:554)

at org.apache.hadoop.hbase.zookeeper.ZKUtil.getDataAsAddress(ZKUtil.java:648)

at org.apache.hadoop.hbase.master.ActiveMasterManager.stop(ActiveMasterManager.java:202)

at org.apache.hadoop.hbase.master.HMaster.run(HMaster.java:318)

2013-06-28 18:26:59,947 ERROR org.apache.hadoop.hbase.master.ActiveMasterManager: master:60000-0x33f7eb9d0650002-0x33f7eb9d0650002-0x33f7eb9d0650002 Error deleting our own master address node

org.apache.zookeeper.KeeperException$SessionExpiredException: KeeperErrorCode = Session expired for /hbase/master

at org.apache.zookeeper.KeeperException.create(KeeperException.java:118)

at org.apache.zookeeper.KeeperException.create(KeeperException.java:42)

at org.apache.zookeeper.ZooKeeper.getData(ZooKeeper.java:927)

at org.apache.hadoop.hbase.zookeeper.ZKUtil.getDataInternal(ZKUtil.java:577)

at org.apache.hadoop.hbase.zookeeper.ZKUtil.getDataAndWatch(ZKUtil.java:554)

at org.apache.hadoop.hbase.zookeeper.ZKUtil.getDataAsAddress(ZKUtil.java:648)

at org.apache.hadoop.hbase.master.ActiveMasterManager.stop(ActiveMasterManager.java:202)

at org.apache.hadoop.hbase.master.HMaster.run(HMaster.java:318)

以下异常原因以及解决方法为摘抄，待验证。

异常原因：

hbase中和GC相关的参数：

修改前（默认）：

export HBASE\_OPTS="$HBASE\_OPTS -ea -verbose:gc -Xloggc:$HBASE\_LOG\_DIR/hbase.gc.log -XX:ErrorFile=$HBASE\_LOG\_DIR/hs\_err\_pid.log -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -XX:+HeapDumpOnOutOfMemoryError -XX:+UseConcMarkSweepGC -XX:+CMSIncrementalMode"

咨询开发修改后：

export HBASE\_OPTS="$HBASE\_OPTS -verbose:gc -Xloggc:$HBASE\_LOG\_DIR/hbase.gc.log -XX:ErrorFile=$HBASE\_LOG\_DIR/hs\_err\_pid.log -XX:+PrintGCDateStamps -XX:+PrintGCDetails -XX:+HeapDumpOnOutOfMemoryError -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:CMSInitiatingOccupancyFraction=70"

-XX UseConcMarkSweepGC ：设置年老代为并发收集。（新老都有）

老：-XX:+CMSIncrementalMode :设置为增量模式。适用于单CPU情况。

新：-XX:+UseParNewGC：设置年轻代为并行收集。可与 CMS 收集同时使用。

-XX:CMSInitiatingOccupancyFraction=70:这个参数是我觉得产生最大作用的。因为最终的目的是减少FULL GC，因为full gc是会block其他线程的。

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，并触发一次全jvm的stop the world（挂起所有线程），然后采用单线程拷贝方式清理所有垃圾对象，也就是full gc。而我们的bulk的最开始的操作就是各种删表,建表频繁的操作，就会使用掉大量master的年轻代的内存，就会发生上面发生的场景，发生full gc。

解决办法：CMSInitiatingOccupancyFraction=70表示年老代占到约70%时就开始执行CMS，这样就不会出现（或很少出现）Full GC了。

hbase日志出现Session expired异常排查方法：

<http://www.aboutyun.com/blog-3979-195.html>

## hbase日志出现Session expired异常排查方法

在http://ip：60010页面发现有个regionserver服务挂机了，查看了日志发现时超时造成的，具体日志如下：

WARN org.apache.hadoop.hbase.zookeeper.RecoverableZooKeeper: Possibly transient ZooKeeper exception: org.apache.zookeeper.KeeperException$SessionExpiredException: KeeperErrorCode = Session expired for /hbase/rs/test1,60020,1400236557454

INFO org.apache.hadoop.hbase.util.RetryCounter: Sleeping 2000ms before retry #1...

WARN org.apache.hadoop.hbase.zookeeper.RecoverableZooKeeper: Possibly transient ZooKeeper exception: org.apache.zookeeper.KeeperException$SessionExpiredException: KeeperErrorCode = Session expired for /hbase/rs/test1,60020,1400236557454

INFO org.apache.hadoop.hbase.util.RetryCounter: Sleeping 4000ms before retry #2...

WARN org.apache.hadoop.hbase.zookeeper.RecoverableZooKeeper: Possibly transient ZooKeeper exception: org.apache.zookeeper.KeeperException$SessionExpiredException: KeeperErrorCode = Session expired for /hbase/rs/test1,60020,1400236557454

INFO org.apache.hadoop.hbase.util.RetryCounter: Sleeping 8000ms before retry #3...

WARN org.apache.hadoop.hbase.zookeeper.RecoverableZooKeeper: Possibly transient ZooKeeper exception: org.apache.zookeeper.KeeperException$SessionExpiredException: KeeperErrorCode = Session expired for /hbase/rs/test1,60020,1400236557454

ERROR org.apache.hadoop.hbase.zookeeper.RecoverableZooKeeper: ZooKeeper delete failed after 3 retries

WARN org.apache.hadoop.hbase.regionserver.HRegionServer: Failed deleting my ephemeral node

org.apache.zookeeper.KeeperException$SessionExpiredException: KeeperErrorCode = Session expired for /hbase/rs/test1,60020,1400236557454

at org.apache.zookeeper.KeeperException.create(KeeperException.java:127)

at org.apache.zookeeper.KeeperException.create(KeeperException.java:51)

at org.apache.zookeeper.ZooKeeper.delete(ZooKeeper.java:873)

at org.apache.hadoop.hbase.zookeeper.RecoverableZooKeeper.delete(RecoverableZooKeeper.java:133)

at org.apache.hadoop.hbase.zookeeper.ZKUtil.deleteNode(ZKUtil.java:1195)

at org.apache.hadoop.hbase.zookeeper.ZKUtil.deleteNode(ZKUtil.java:1184)

at org.apache.hadoop.hbase.regionserver.HRegionServer.deleteMyEphemeralNode(HRegionServer.java:1133)

at org.apache.hadoop.hbase.regionserver.HRegionServer.run(HRegionServer.java:900)

at java.lang.Thread.run(Thread.java:662)

下面介绍一下排查方法：

1、使用vmstat 1 命令查看si so两个swap列，确认没有发生交换，1代表每秒打印一次

2、使用jstat -gcutil pid 1000 查看fgct列，确认regionserver没有发生长时间gc暂停

3、使用top命令查看regionserver是否有充足的cpu资源，mapreduce会占用很多cpu，可以减少mapreduce任务数

4、加大zookeeper会话超时时间，编辑hbase-site.xml文件，添加下面的属性

<property>

<name>zookeeper.session.timeout</name>

<value>120000</value>

</property>

5、加大zookeeper会话最大超时时间编辑zoo.cfg 提高MaxSessionTimeout=120000，修改后重启zookeeper。

zookeeper的超时时间不要设置太大，在服务挂掉的情况下，会反映很慢。

## 文件块坏掉修复

## hadoop and hbase rebalancing after node additions

1. The Hadoop (HDFS) balancer moves blocks around from one node to another to try to make it so each datanode has the same amount of data (within a configurable threshold). This messes up HBases's data locality, meaning that a particular region may be serving a file that is no longer on it's local host.
2. HBase's balance\_switch balances the cluster so that each regionserver hosts the same number of regions (or close to). This is separate from Hadoop's (HDFS) balancer.
3. If you are running only HBase, I recommend not running Hadoop's (HDFS) balancer as it will cause certain regions to lose their data locality. This causes any request to that region to have to go over the network to one of the datanodes that is serving it's HFile.

# 7.安全

Kerberos 5

# 8.参考

## HBase 默认配置

该文档是用hbase默认配置文件生成的，文件源是 hbase-default.xml **hbase.rootdir**

这 个目录是region server的共享目录，用来持久化Hbase。URL需要是'完全正确'的，还要包含文件系统的scheme。例如，要表示hdfs中的 '/hbase'目录，namenode 运行在namenode.example.org的9090端口。则需要设置为hdfs://namenode.example.org:9000 /hbase。默认情况下Hbase是写到/tmp的。不改这个配置，数据会在重启的时候丢失。

默认: file:///tmp/hbase-${user.name}/hbase

**hbase.master.port**

Hbase的Master的端口.

默认: 60000

**hbase.cluster.distributed**

Hbase的运行模式。false是单机模式，true是分布式模式。若为false,Hbase和Zookeeper会运行在同一个JVM里面。

默认: false

**hbase.tmp.dir**

本地文件系统的临时文件夹。可以修改到一个更为持久的目录上。(/tmp会在重启时清楚)

默认: /tmp/hbase-${user.name}

**hbase.master.info.port**

HBase Master web 界面端口. 设置为-1 意味着你不想让他运行。

默认: 60010

**hbase.master.info.bindAddress**

HBase Master web 界面绑定的端口

默认: 0.0.0.0

**hbase.client.write.buffer**

HTable 客户端的写缓冲的默认大小。这个值越大，需要消耗的内存越大。因为缓冲在客户端和服务端都有实例，所以需要消耗客户端和服务端两个地方的内存。得到的好处 是，可以减少RPC的次数。可以这样估算服务器端被占用的内存： hbase.client.write.buffer \* hbase.regionserver.handler.count

默认: 2097152

**hbase.regionserver.port**

HBase RegionServer绑定的端口

默认: 60020

**hbase.regionserver.info.port**

HBase RegionServer web 界面绑定的端口 设置为 -1 意味这你不想与运行 RegionServer 界面.

默认: 60030

**hbase.regionserver.info.port.auto**

Master或RegionServer是否要动态搜一个可以用的端口来绑定界面。当hbase.regionserver.info.port已经被占用的时候，可以搜一个空闲的端口绑定。这个功能在测试的时候很有用。默认关闭。

默认: false

**hbase.regionserver.info.bindAddress**

HBase RegionServer web 界面的IP地址

默认: 0.0.0.0

**hbase.regionserver.class**

RegionServer 使用的接口。客户端打开代理来连接region server的时候会使用到。

默认: org.apache.hadoop.hbase.ipc.HRegionInterface

**hbase.client.pause**

通常的客户端暂停时间。最多的用法是客户端在重试前的等待时间。比如失败的get操作和region查询操作等都很可能用到。

默认: 1000

**hbase.client.retries.number**

最大重试次数。例如 region查询，Get操作，Update操作等等都可能发生错误，需要重试。这是最大重试错误的值。

默认: 10

**hbase.client.scanner.caching**

当 调用Scanner的next方法，而值又不在缓存里的时候，从服务端一次获取的行数。越大的值意味着Scanner会快一些，但是会占用更多的内存。当 缓冲被占满的时候，next方法调用会越来越慢。慢到一定程度，可能会导致超时。例如超过了 hbase.regionserver.lease.period。

默认: 1

**hbase.client.keyvalue.maxsize**

一 个KeyValue实例的最大size.这个是用来设置存储文件中的单个entry的大小上界。因为一个KeyValue是不能分割的，所以可以避免因为 数据过大导致region不可分割。明智的做法是把它设为可以被最大region size整除的数。如果设置为0或者更小，就会禁用这个检查。默认10MB。

默认: 10485760

**hbase.regionserver.lease.period**

客户端租用HRegion server 期限，即超时阀值。单位是毫秒。默认情况下，客户端必须在这个时间内发一条信息，否则视为死掉。

默认: 60000

**hbase.regionserver.handler.count**

RegionServers受理的RPC Server实例数量。对于Master来说，这个属性是Master受理的handler数量

默认: 10

**hbase.regionserver.msginterval**

RegionServer 发消息给 Master 时间间隔，单位是毫秒

默认: 3000

**hbase.regionserver.optionallogflushinterval**

将Hlog同步到HDFS的间隔。如果Hlog没有积累到一定的数量，到了时间，也会触发同步。默认是1秒，单位毫秒。

默认: 1000

**hbase.regionserver.regionSplitLimit**

region的数量到了这个值后就不会在分裂了。这不是一个region数量的硬性限制。但是起到了一定指导性的作用，到了这个值就该停止分裂了。默认是MAX\_INT.就是说不阻止分裂。

默认: 2147483647

hbase.regionserver.logroll.period

提交commit log的间隔，不管有没有写足够的值。

默认: 3600000

hbase.regionserver.hlog.reader.impl

HLog file reader 的实现.

默认: org.apache.hadoop.hbase.regionserver.wal.SequenceFileLogReader

hbase.regionserver.hlog.writer.impl

HLog file writer 的实现.

默认: org.apache.hadoop.hbase.regionserver.wal.SequenceFileLogWriter

hbase.regionserver.thread.splitcompactcheckfrequency

region server 多久执行一次split/compaction 检查.

默认: 20000

hbase.regionserver.nbreservationblocks

储备的内存block的数量(译者注:就像石油储备一样)。当发生out of memory 异常的时候，我们可以用这些内存在RegionServer停止之前做清理操作。

默认: 4

hbase.zookeeper.dns.interface

当使用DNS的时候，Zookeeper用来上报的IP地址的网络接口名字。

默认: default

hbase.zookeeper.dns.nameserver

当使用DNS的时候，Zookeepr使用的DNS的域名或者IP 地址，Zookeeper用它来确定和master用来进行通讯的域名.

默认: default

hbase.regionserver.dns.interface

当使用DNS的时候，RegionServer用来上报的IP地址的网络接口名字。

默认: default

hbase.regionserver.dns.nameserver

当使用DNS的时候，RegionServer使用的DNS的域名或者IP 地址，RegionServer用它来确定和master用来进行通讯的域名.

默认: default

hbase.master.dns.interface

当使用DNS的时候，Master用来上报的IP地址的网络接口名字。

默认: default

hbase.master.dns.nameserver

当使用DNS的时候，RegionServer使用的DNS的域名或者IP 地址，Master用它来确定用来进行通讯的域名.

默认: default

hbase.balancer.period

Master执行region balancer的间隔。

默认: 300000

hbase.regions.slop

当任一regionserver有average + (average \* slop)个region是会执行Rebalance

默认: 0

hbase.master.logcleaner.ttl

Hlog存在于.oldlogdir 文件夹的最长时间, 超过了就会被 Master 的线程清理掉.

默认: 600000

hbase.master.logcleaner.plugins

LogsCleaner 服务会执行的一组LogCleanerDelegat。值用逗号间隔的文本表示。这些WAL/HLog cleaners会按顺序调用。可以把先调用的放在前面。你可以实现自己的LogCleanerDelegat，加到Classpath下，然后在这里写 下类的全称。一般都是加在默认值的前面。

默认: org.apache.hadoop.hbase.master.TimeToLiveLogCleaner

hbase.regionserver.global.memstore.upperLimit

单个region server的全部memtores的最大值。超过这个值，一个新的update操作会被挂起，强制执行flush操作。

默认: 0.4

hbase.regionserver.global.memstore.lowerLimit

当强制执行flush操作的时候，当低于这个值的时候，flush会停止。默认是堆大小的 35% . 如果这个值和 hbase.regionserver.global.memstore.upperLimit 相同就意味着当update操作因为内存限制被挂起时，会尽量少的执行flush(译者注:一旦执行flush，值就会比下限要低，不再执行)

默认: 0.35

hbase.server.thread.wakefrequency

service工作的sleep间隔，单位毫秒。 可以作为service线程的sleep间隔，比如log roller.

默认: 10000

hbase.hregion.memstore.flush.size

当memstore的大小超过这个值的时候，会flush到磁盘。这个值被一个线程每隔hbase.server.thread.wakefrequency检查一下。

默认: 67108864

hbase.hregion.preclose.flush.size

当一个region中的memstore的大小大于这个值的时候，我们又触发了close.会先运行“pre-flush”操作，清理这个需要关闭的 memstore，然后将这个region下线。当一个region下线了，我们无法再进行任何写操作。如果一个memstore很大的时候，flush 操作会消耗很多时间。"pre-flush"操作意味着在region下线之前，会先把memstore清空。这样在最终执行close操作的时 候，flush操作会很快。

默认: 5242880

hbase.hregion.memstore.block.multiplier

如果memstore有hbase.hregion.memstore.block.multiplier倍数的 hbase.hregion.flush.size的大小，就会阻塞update操作。这是为了预防在update高峰期会导致的失控。如果不设上 界，flush的时候会花很长的时间来合并或者分割，最坏的情况就是引发out of memory异常。(译者注:内存操作的速度和磁盘不匹配，需要等一等。原文似乎有误)

默认: 2

hbase.hregion.memstore.mslab.enabled

体验特性：启用memStore分配本地缓冲区。这个特性是为了防止在大量写负载的时候堆的碎片过多。这可以减少GC操作的频率。(GC有可能会Stop the world)(译者注：实现的原理相当于预分配内存，而不是每一个值都要从堆里分配)

默认: false

hbase.hregion.max.filesize

最大HStoreFile大小。若某个Column families的HStoreFile增长达到这个值，这个Hegion会被切割成两个。 Default: 256M.

默认: 268435456

hbase.hstore.compactionThreshold

当一个HStore含有多于这个值的HStoreFiles(每一个memstore flush产生一个HStoreFile)的时候，会执行一个合并操作，把这HStoreFiles写成一个。这个值越大，需要合并的时间就越长。

默认: 3

hbase.hstore.blockingStoreFiles

当一个HStore含有多于这个值的HStoreFiles(每一个memstore flush产生一个HStoreFile)的时候，会执行一个合并操作，update会阻塞直到合并完成，直到超过了hbase.hstore.blockingWaitTime的值

默认: 7

hbase.hstore.blockingWaitTime

hbase.hstore.blockingStoreFiles所限制的StoreFile数量会导致update阻塞，这个时间是来限制阻塞时间的。当超过了这个时间，HRegion会停止阻塞update操作，不过合并还有没有完成。默认为90s.

默认: 90000

hbase.hstore.compaction.max

每个“小”合并的HStoreFiles最大数量。

默认: 10

hbase.hregion.majorcompaction

一个Region中的所有HStoreFile的major compactions的时间间隔。默认是1天。 设置为0就是禁用这个功能。

默认: 86400000

hbase.mapreduce.hfileoutputformat.blocksize

MapReduce 中HFileOutputFormat可以写 storefiles/hfiles. 这个值是hfile的blocksize的最小值。通常在Hbase写Hfile的时候，bloocksize是由table schema(HColumnDescriptor)决定的，但是在mapreduce写的时候，我们无法获取schema中blocksize。这个值 越小，你的索引就越大，你随机访问需要获取的数据就越小。如果你的cell都很小，而且你需要更快的随机访问，可以把这个值调低。

默认: 65536

hfile.block.cache.size

分配给HFile/StoreFile的block cache占最大堆(-Xmx setting)的比例。默认是20%，设置为0就是不分配。

默认: 0.2

hbase.hash.type

哈希函数使用的哈希算法。可以选择两个值:: murmur (MurmurHash) 和 jenkins (JenkinsHash). 这个哈希是给 bloom filters用的.

默认: murmur

hbase.master.keytab.file

HMaster server验证登录使用的kerberos keytab 文件路径。(译者注：Hbase使用Kerberos实现安全)

默认:

hbase.master.kerberos.principal

例 如. "hbase/\_HOST@EXAMPLE.COM". HMaster运行需要使用 kerberos principal name. principal name 可以在: user/hostname@DOMAIN 中获取. 如果 "\_HOST" 被用做hostname portion，需要使用实际运行的hostname来替代它。

默认:

hbase.regionserver.keytab.file

HRegionServer验证登录使用的kerberos keytab 文件路径。

默认:

hbase.regionserver.kerberos.principal

例如. "hbase/\_HOST@EXAMPLE.COM". HRegionServer运行需要使用 kerberos principal name. principal name 可以在: user/hostname@DOMAIN 中获取. 如果 "\_HOST" 被用做hostname portion，需要使用实际运行的hostname来替代它。在这个文件中必须要有一个entry来描述 hbase.regionserver.keytab.file

默认:

zookeeper.session.timeout

ZooKeeper 会话超时.Hbase把这个值传递改zk集群，向他推荐一个会话的最大超时时间。详见http://hadoop.apache.org /zookeeper/docs/current/zookeeperProgrammers.html#ch\_zkSessions "The client sends a requested timeout, the server responds with the timeout that it can give the client. "。 单位是毫秒

默认: 180000

zookeeper.znode.parent

ZooKeeper中的Hbase的根ZNode。所有的Hbase的ZooKeeper会用这个目录配置相对路径。默认情况下，所有的Hbase的ZooKeeper文件路径是用相对路径，所以他们会都去这个目录下面。

默认: /hbase

zookeeper.znode.rootserver

ZNode 保存的 根region的路径. 这个值是由Master来写，client和regionserver 来读的。如果设为一个相对地址，父目录就是 ${zookeeper.znode.parent}.默认情形下，意味着根region的路径存储在/hbase/root-region- server.

默认: root-region-server

hbase.zookeeper.quorum

Zookeeper 集群的地址列表，用逗号分割。例 如："host1.mydomain.com,host2.mydomain.com,host3.mydomain.com".默认是 localhost,是给伪分布式用的。要修改才能在完全分布式的情况下使用。如果在hbase-env.sh设置了HBASE\_MANAGES\_ZK， 这些ZooKeeper节点就会和Hbase一起启动。

默认: localhost

hbase.zookeeper.peerport

ZooKeeper节点使用的端口。详细参见：http://hadoop.apache.org/zookeeper/docs/r3.1.1/zookeeperStarted.html#sc\_RunningReplicatedZooKeeper

默认: 2888

hbase.zookeeper.leaderport

ZooKeeper用来选择Leader的端口，详细参见：http://hadoop.apache.org/zookeeper/docs/r3.1.1/zookeeperStarted.html#sc\_RunningReplicatedZooKeeper

默认: 3888

hbase.zookeeper.property.initLimit

ZooKeeper的zoo.conf中的配置。 初始化synchronization阶段的ticks数量限制

默认: 10

hbase.zookeeper.property.syncLimit

ZooKeeper的zoo.conf中的配置。 发送一个请求到获得承认之间的ticks的数量限制

默认: 5

hbase.zookeeper.property.dataDir

ZooKeeper的zoo.conf中的配置。 快照的存储位置

默认: ${hbase.tmp.dir}/zookeeper

hbase.zookeeper.property.clientPort

ZooKeeper的zoo.conf中的配置。 客户端连接的端口

默认: 2181

hbase.zookeeper.property.maxClientCnxns

ZooKeeper的zoo.conf中的配置。 ZooKeeper集群中的单个节点接受的单个Client(以IP区分)的请求的并发数。这个值可以调高一点，防止在单机和伪分布式模式中出问题。

默认: 2000

hbase.rest.port

HBase REST server的端口

默认: 8080

hbase.rest.readonly

定义REST server的运行模式。可以设置成如下的值： false: 所有的HTTP请求都是被允许的 - GET/PUT/POST/DELETE. true:只有GET请求是被允许的

默认: false

## HBase 在HDFS 上的目录树

一、0.94-cdh4.2.1版本

系统级别的一级目录如下，用户自定义的均在这个/hbase 下的一级子目录下

/hbase/-ROOT-

/hbase/.META.

/hbase/.archive

/hbase/.corrupt

/hbase/.hbck

/hbase/.logs

/hbase/.oldlogs

/hbase/.snapshot

/hbase/.tmp

/hbase/hbase.id

/hbase/hbase.version

1、/hbase/-ROOT-

hbase读写数据的时候采用三级寻址方式，首先找到从 zk 中找到ROOT 表所在位置，通过 ROOT 表找到 META 表所在位置，然后再从 META 表定位到你要读写数据Region 所在的Regionserver。所以-ROOT-目录对应 HBase 中的系统表ROOT，就不多做解释了。

2、/hbase/.META.

就是存储1中介绍的 META 表的存储路径。

3、/hbase/.archive

HBase 在做 Split或者 compact 操作完成之后，会将 HFile 移到.archive 目录中，然后将之前的 hfile 删除掉，该目录由 HMaster 上的一个定时任务定期去清理。

4、/hbase/.corrupt

存储HBase做损坏的日志文件，一般都是为空的。

5、/hbase/.hbck

HBase 运维过程中偶尔会遇到元数据不一致的情况，这时候会用到提供的 hbck 工具去修复，修复过程中会使用该目录作为临时过度缓冲。

6、/hbase/.logs

大家都知道 HBase 是支持 WAL（Write Ahead Log） 的，HBase 会在第一次启动之初会给每一台 RegionServer 在.log 下创建一个目录，若客户端如果开启WAL 模式，会先将数据写入一份到.log 下，当 RegionServer crash 或者目录达到一定大小，会开启 replay 模式，类似 MySQL 的 binlog。

7、/hbase/.oldlogs

当.logs 文件夹中的 HLog 没用之后会 move 到.oldlogs 中，HMaster 会定期去清理。

8、/hbase/.snapshot

hbase若开启了 snapshot 功能之后，对某一个用户表建立一个 snapshot 之后，snapshot 都存储在该目录下，如对表test 做了一个 名为sp\_test 的snapshot，就会在/hbase/.snapshot/目录下创建一个sp\_test 文件夹，snapshot 之后的所有写入都是记录在这个 snapshot 之上。

9、/hbase/.tmp

当对表做创建或者删除操作的时候，会将表move 到该 tmp 目录下，然后再去做处理操作。

10、/hbase/hbase.id

它是一个文件，存储集群唯一的 cluster id 号，是一个 uuid。

11、/hbase/hbase.version

同样也是一个文件，存储集群的版本号，貌似是加密的，看不到，只能通过web-ui 才能正确显示出来。

二、0.98.8版本

自0.96版本之后，hbase 源码结构上做了很大的优化，目录结构也发生了变化，做了精简和优化，这里以0.98.8为例介绍，目录如下：

/hbase/.tmp

/hbase/WALs

/hbase/archive

/hbase/corrupt

/hbase/data

/hbase/hbase.id

/hbase/hbase.version

/hbase/oldWALs

1、/hbase/.tmp

这个目录不变还是原来的tmp目录，作用是一样的。

2、/hbase/WALs

这里对应0.94的.logs 目录，取名为 WALs 更加见名知意了，点个赞！

3、/hbase/archive

和0.94一样，只是去掉了.而已，估计是作者不想把它作为一个隐藏文件夹了吧

4、/hbase/corrupt

和0.94一样，去了.

5、/hbase/data

这个才是 hbase 的核心目录，0.98版本里支持 namespace 的概念模型，系统会预置两个 namespace 即：hbase和default

5.1 /hbase/data/default

这个默认的namespace即没有指定namespace 的表都将会flush 到该目录下面。

5.2 /hbase/data/hbase

这个namespace 下面存储了 HBase 的 namespace、meta 和acl 三个表，这里的 meta 表跟0.94版本的.META.是一样的，自0.96之后就已经将 ROOT 表去掉了，直接从Zookeeper 中找到meta 表的位置，然后通过 meta 表定位到 region。 namespace 中存储了 HBase 中的所有 namespace 信息，包括预置的hbase 和 default。acl 则是表的用户权限控制。

如果自定义一些 namespace 的话，就会再/hbase/data 目录下新建一个 namespace 文件夹，该 namespace 下的表都将 flush 到该目录下。

6、/hbase/hbase.id

它是一个文件，存储集群唯一的 cluster id 号，是一个 uuid。

7、/hbase/hbase.version

同样也是一个文件，存储集群的版本号，貌似是加密的，看不到，只能通过web-ui 才能正确显示出来。

8、/hbase/oldWALs

# 9.遗留

1.hadoop jar [hbase-server-jar] completebulkload /tmp/mytable mytable(?)