

Client：

整个Hbase集群的访问入口；

使用HBase RPC机制与HMaster和HRegionServer进行通讯；

与HMaster进行通讯进行管理类操作；

与HRegionServer进行数据读写类操作(HTable类)；

包含访问HBase的接口，并维护cache来加快对Hbase的访问。

Zookeeper

保证任何时候，急群众只有一个HMaster；

存储所有HRegion的寻址入口；

实时监控HRegionServer的上线和下线信息，并实时通知给HMaster；

存储Hbase的schema和table元数据；

Zookeeper Quorum存储-ROOt-表地址、HMaster地址。

HMaster

HMater没有单点问题，Hbase中可以启动多个HMaster，通过Zookeeper的Master Election机制保证总有一个Master在运行主要负责Table和Region的管理工作；

管理用户对table的增删查改操作；

管理HRegionServer的负载均衡，调整Region分布；

Region Split后，负责新Region的分布；

在HRegionServer停机后，负责失效HRegionServer上Region迁移工作。

HRegionServer

维护HRegion，处理对这些HRegion的IO请求，想HDFS文件系统中读写数据；

负责切分在运行过程中变得过大的HRegion；

Client访问hbase上的数据的过程并不需要master参与(寻址访问Zookeeper和HRegionServer，数据读写访问HRegionServer)，HMaster仅仅维护者table和Region的元数据信息，负载很低。

HregionServer

Hregionserver内部管理由一系列HRegin对象，每个Hregion对应了table中的一个region，HRegion中由多个HStore组成。每个HStore对应了Table中的一个column family的存储，可以看出每个columnfamlily其实局势一个集中的存储单元，因此最好将具备共同IO特性的column放在一个column family中，这样最高效。

HStore存储似乎HBase存储的核心，由两部分组成，一部分是MemStore，一部分是StoreFile。MemStore是Sorted Memory Buffer，用户写入的数据首先会放入MemStore，当MemStore满了以后会Flush成一个StoreFile(底层实现是HFile)。

Hlog

WAL意为Write ahead log，用来做灾难恢复。HLog记录数据的所有变更，一旦数据修改，就可以从log中进行恢复。每个HRegionServer维护了一个HLog，而不是每个HRegion一个。这样不同region(来自不同table)的日志会混在一起，这样做的目的是不断追加单个文件相对于同时写多个文件而言，可以减少磁盘寻址次数，因此可以提高对table的写性能。带来的麻烦是，如果一个HRegionServer下线，为了恢复其上的region，需要将HRegionServer上的log进行拆分，然后分发到其他HRegionServer上进行恢复。

每次更新数据时,都会首先将数据记录在提交日志(commit log)中，在HBase中这叫做预写日志(write-ahead log,WAL)，然后才会将这些数据写入内存的memstore中。一旦内存保存的写入数据的累计大小超过了一个给定的最大值，系统就会将这些数据移出内存作为HFile文件写入到磁盘中。写入磁盘之后，系统会丢弃对应的提交日志。在系统将数据移出memstore写入磁盘的过程中，可以不必阻塞系统的读写，通过滚动内存转给你的memstore就能达到这个目的，即用空的新memstore获取更新数据，将满的旧memstore转换成一个文件，请注意，memstore中的数据已经按照行键排序，持久化到磁盘中的HFile也是按照这个顺序排列的，所以不必执行排序或其他特殊处理。

因为存储文件是不可改变的，所以无法通过移除某个键/值对来简单地删除值。可行的解决办法是，做一个标记，表明给定行已被删除的试试。在检索过程中，这些删除标记掩盖了实际值，客户端读不到实际值。

随着memstore中的数据不断刷写到磁盘中，会产生越来越多的HFile文件，然而，太多数据文件会导致数据查询IO次数增多。HBase内部有一个解决这个问题的管家机制，即用合并(compaction)将多个文件合并成一个较大的文件。合并有两种类型：minor合并(minor compaction)和major合并(majar compaction)。

Minor Compaction是指选取一些小的、相邻的StoreFile将他们合并成一个更大的StoreFile，在这个过程中不会处理已经Deleted或Expired的Cell。一次Minor Compaction的结果是更少并且更大的StoreFile。

Major Compaction是指将所有的StoreFile合并成一个StoreFile，这个过程还会清理三类无意义数据：被删除的数据、TTL过期数据、版本号超过设定版本号的数据。另外，一般情况下，Major Compaction时间会持续比较长，整个过程会消耗大量系统资源，对上层业务有比较大的影响。因此线上业务都会将关闭自动触发Major Compaction功能，改为手动在业务低峰期触发。

当写请求非常多，导致不断生成HFile，但compact的速度远远跟不上HFile生成的速度，这样就会使HFile的数量会越来越多，导致读性能急剧下降。为了避免这种情况，在HFile的数量过多的时候会限制写请求的速度：在每次执行MemStore flush的操作前，如果HStore的HFile数超过hbase.hstore.blockingStoreFiles (默认7)，则会阻塞flush操作hbase.hstore.blockingWaitTime时间，在这段时间内，如果compact操作使得HStore文件数下降到回这个值，则停止阻塞。另外阻塞超过时间后，也会恢复执行flush操作。这样做就可以有效地控制大量写请求的速度，但同时这也是影响写请求速度的主要原因之一。

参考文献：<http://www.thebigdata.cn/HBase/30633.html>

当单个StoreFile大小超过一定阈值后，触发split操作，把当前Region Split成2个Region，Region会下线，新Splite出的2个孩子Region会被HMaster分配到相应的HRegionServer上，使得原来1个Region的压力得以分流到2个Region上。

所以在增加数据或者更新数据方面，所有的更新和删除操作都是在compaction阶段做的，具体来说是在major compaction阶段做的。这个样子设计的好处是：用户写操作只需要进入到内存即可立即返回，从而保证I/O高性能。

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。

通过读写操作，可以知道：

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，主要用来读。在注重读响应的应用场景下，可以将BlockCache设置大些，Memstore设置小些，以加大缓存的命中率。

Java API

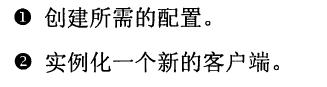
HTable：通过这个类，用户可以完成向HBase存储和检索数据，以及删除无效数据之类的操作。创建HTable实例是有代价的。每个实例都需要扫面.META.表，以检查该表是否存在、是否可用等，这些检查和操作导致实例调用非常耗时，因此，推荐只创建一次HTable实例，而且是每个线程创建一个，然后在客户端反复使用这个对象。

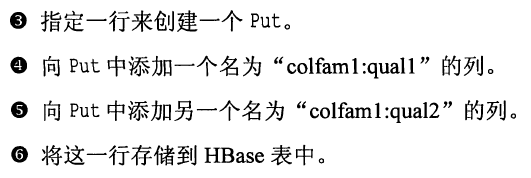
如果需要读个HTable实例，应使用HTablePool类。

所有的修改操作只保证行级别的原子性。

Put方法。







每一个put操作实际上都是一个RPC操作，他将客户端数据传送到服务器然后返回。这只适合小数据量的操作，如果有个应用程序需要每秒存储上千行数据到HBase表中，这样的处理就不太适合了。

Hbase的API配置了一个客户端的写缓冲区，缓冲区负责收集put操作，然后调用RPC操作一次性将put送往服务器。

Void setAutoFlush(Boolean autoFlush)

Boolean isAutoFlush()

默认情况下是禁用的，激活：

Table.setAutoFlush(false)

API会跟踪统计每个用户添加的实例的堆大小，从而计算出缓冲区的数据量。出了追踪所有的数据开销，还会追踪必要的内部数据结构。

一旦超出缓冲区指定的大小限制，客户端就会隐式地调用刷写命令。

当然用户也可以通过以下配置来调节缓冲区的大小：

Long getWriteBufferSize()

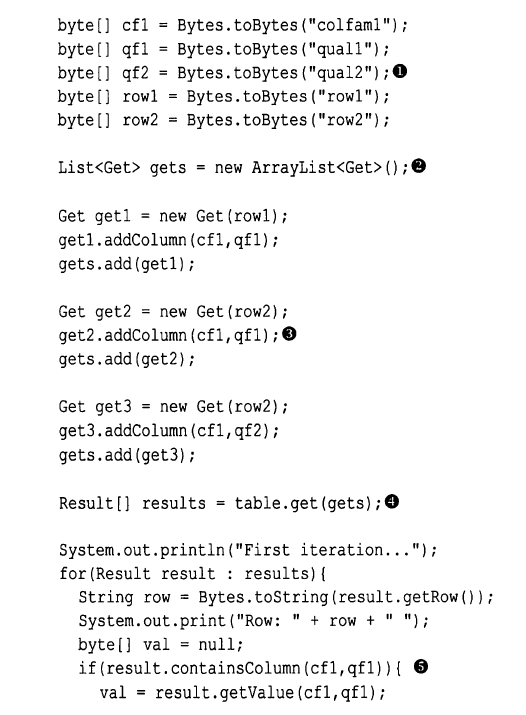
Void setWriteBufferSeze(long writeBUffersize) throws Exception

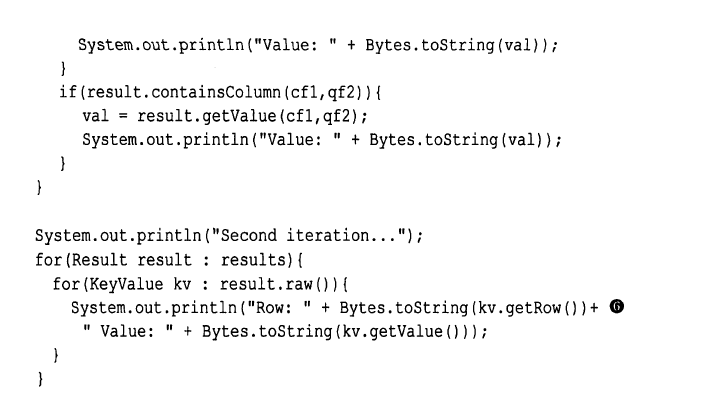
当需要把数据强制写到服务器时，可以调用另外一个API函数：

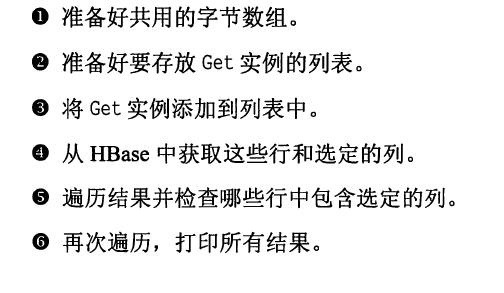
Void flushCommits() throws IOException

Get







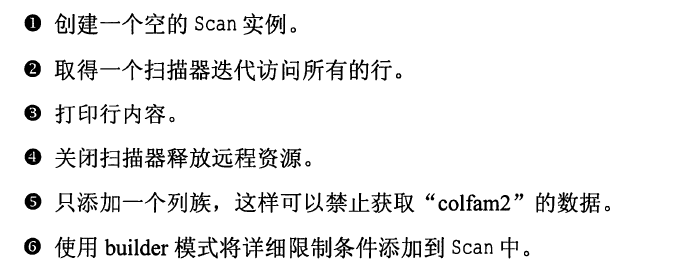


删除：Delete

略

扫描：





扫描操作不会通过一次RPC请求返回所有匹配的行，而是以行为单位进行返回。

这样做的性能不是很好。一次，如果一次RPC请求可以获取多行数据，这样会更有意义。这样的方法可以由扫描缓存实现。默认情况下，这个缓存是关闭的。

可以在两个层面上打开它：在表的层面，这个表所有扫描实例的缓存都会生效；也可以在扫描层面，这样便只会影响当前的扫描实例。

设置表级的扫描器缓存：

Void setScannerCaching(int scannerCaching)

Int getScannerCaching()

设置扫描级的缓存：

Void setCaching(int caching)

Int getCaching()

上面介绍了客户端使用扫描器缓存来从远程region服务器向客户端整批传输数据。不过需要注意：如果数据量非常大的行，这些行有可能超过客户端进程的内存容量。Hbase和它的客户度API对这个问题有一个解决方法：批量。用户可以使用一下方法控制批量获取操作：

Void setBatch(int batch)

Int getBatch()

缓存是面向行一级的操作，而批量则是面向列一级的操作。批量可以让用户选择每一次ResultScanner实例的next()操作要取回多少列。例如，在扫描中设置setBatch（5）,则一次next()返回的Result实例会包含5列。