## HBase性能优化方法总结（一）：表的设计

# 1. 表的设计

## 1.1 Pre-Creating Regions

默认情况下，在创建HBase表的时候会自动创建一个region分区，当导入数据的时候，所有的HBase客户端都向这一个region写数据，直到这个region足够大了才进行切分。一种可以加快批量写入速度的方法是通过预先创建一些空的regions，这样当数据写入HBase时，会按照region分区情况，在集群内做数据的负载均衡。

有关预分区，详情参见：[Table Creation: Pre-Creating Regions](http://hbase.apache.org/book.html#precreate.regions)，下面是一个例子：

[复制代码](javascript:void(0);)

public static boolean createTable(HBaseAdmin admin, HTableDescriptor table, byte[][] splits)  
throws IOException {  
 try {  
 admin.createTable(table, splits);  
 return true;  
 } catch (TableExistsException e) {  
 logger.info("table " + table.getNameAsString() + " already exists");  
 // the table already exists...  
 return false;   
 }  
}  
  
public static byte[][] getHexSplits(String startKey, String endKey, int numRegions) { //start:001,endkey:100,10region [001,010]

[011,020]  
 byte[][] splits = new byte[numRegions-1][];  
 BigInteger lowestKey = new BigInteger(startKey, 16);  
 BigInteger highestKey = new BigInteger(endKey, 16);  
 BigInteger range = highestKey.subtract(lowestKey);  
 BigInteger regionIncrement = range.divide(BigInteger.valueOf(numRegions));  
 lowestKey = lowestKey.add(regionIncrement);  
 for(int i=0; i < numRegions-1;i++) {  
 BigInteger key = lowestKey.add(regionIncrement.multiply(BigInteger.valueOf(i)));  
 byte[] b = String.format("%016x", key).getBytes();  
 splits[i] = b;  
 }  
 return splits;  
}

[复制代码](javascript:void(0);)

## 1.2 Row Key

HBase中row key用来检索表中的记录，支持以下三种方式：

* 通过单个row key访问：即按照某个row key键值进行get操作；
* 通过row key的range进行scan：即通过设置startRowKey和endRowKey，在这个范围内进行扫描；
* 全表扫描：即直接扫描整张表中所有行记录。

在HBase中，row key可以是任意字符串，最大长度64KB，实际应用中一般为10~100bytes，存为byte[]字节数组，**一般设计成定长的**。

row key是按照**字典序**存储，因此，设计row key时，要充分利用这个排序特点，将经常一起读取的数据存储到一块，将最近可能会被访问的数据放在一块。

举个例子：如果最近写入HBase表中的数据是最可能被访问的，可以考虑将时间戳作为row key的一部分，由于是字典序排序，所以可以使用Long.MAX\_VALUE - timestamp作为row key，这样能保证新写入的数据在读取时可以被快速命中。

Rowkey规则：

1. 越小越好
2. Rowkey的设计是要根据实际业务来
3. 散列性
   1. 取反 001 002 100 200
   2. Hash

## 1.3 Column Family

**不要在一张表里定义太多的column family**。目前Hbase并不能很好的处理超过2~3个column family的表。因为某个column family在flush的时候，它邻近的column family也会因关联效应被触发flush，最终导致系统产生更多的I/O。感兴趣的同学可以对自己的HBase集群进行实际测试，从得到的测试结果数据验证一下。

## 1.4 In Memory

创建表的时候，可以通过HColumnDescriptor.setInMemory(true)将表放到RegionServer的缓存中，保证在读取的时候被cache命中。

## 1.5 Max Version

创建表的时候，可以通过HColumnDescriptor.setMaxVersions(int maxVersions)设置表中数据的最大版本，如果只需要保存最新版本的数据，那么可以设置setMaxVersions(1)。

## 1.6 Time To Live

创建表的时候，可以通过HColumnDescriptor.setTimeToLive(int timeToLive)设置表中数据的存储生命期，过期数据将自动被删除，例如如果只需要存储最近两天的数据，那么可以设置setTimeToLive(2 \* 24 \* 60 \* 60)。

## 1.7 Compact & Split

在HBase中，数据在更新时首先写入WAL 日志(HLog)和内存(MemStore)中，MemStore中的数据是排序的，当MemStore累计到一定阈值时，就会创建一个新的MemStore，并且将老的MemStore添加到flush队列，由单独的线程flush到磁盘上，成为一个StoreFile。于此同时， 系统会在zookeeper中记录一个redo point，表示这个时刻之前的变更已经持久化了**(minor compact)**。

StoreFile是只读的，一旦创建后就不可以再修改。因此Hbase的更新其实是不断追加的操作。当一个Store中的StoreFile达到一定的阈值后，就会进行一次合并**(major compact)**，将对同一个key的修改合并到一起，形成一个大的StoreFile，当StoreFile的大小达到一定阈值后，又会对 StoreFile进行分割**(split)**，等分为两个StoreFile。

由于对表的更新是不断追加的，处理读请求时，需要访问Store中全部的StoreFile和MemStore，将它们按照row key进行合并，由于StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，通常合并过程还是比较快的。

实际应用中，可以考虑必要时手动进行major compact，将同一个row key的修改进行合并形成一个大的StoreFile。同时，可以将StoreFile设置大些，减少split的发生。

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。

minor compaction:的是较小、很少文件的合并。

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。

1. 关闭自动major compaction
2. 手动编程major compaction

Timer类，contab

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

## [HBase性能优化方法总结（二）：写表操作](http://www.cnblogs.com/panfeng412/archive/2012/03/08/hbase-performance-tuning-section2.html)

下面是本文总结的第二部分内容：写表操作相关的优化方法。

# 2. 写表操作

## 2.1 多HTable并发写

创建多个HTable客户端用于写操作，提高写数据的吞吐量，一个例子：

[复制代码](javascript:void(0);)

static final Configuration conf = HBaseConfiguration.create();  
static final String table\_log\_name = “user\_log”;  
wTableLog = new HTable[tableN];  
for (int i = 0; i < tableN; i++) {  
 wTableLog[i] = new HTable(conf, table\_log\_name);  
 wTableLog[i].setWriteBufferSize(5 \* 1024 \* 1024); //5MB  
 wTableLog[i].setAutoFlush(false);  
}

[复制代码](javascript:void(0);)

## 2.2 HTable参数设置

### 2.2.1 Auto Flush

通过调用HTable.setAutoFlush(false)方法可以将HTable写客户端的自动flush关闭，这样可以批量写入数据到HBase，而不是有一条put就执行一次更新，只有当put填满客户端写缓存时，才实际向HBase服务端发起写请求。默认情况下auto flush是开启的。

### 2.2.2 Write Buffer

通过调用HTable.setWriteBufferSize(writeBufferSize)方法可以设置HTable客户端的写buffer大小，如果新设置的buffer小于当前写buffer中的数据时，buffer将会被flush到服务端。其中，writeBufferSize的单位是byte字节数，可以根据实际写入数据量的多少来设置该值。

### 2.2.3 WAL Flag

在HBae中，客户端向集群中的RegionServer提交数据时（Put/Delete操作），首先会先写WAL（Write Ahead Log）日志（即HLog，一个RegionServer上的所有Region共享一个HLog），只有当WAL日志写成功后，再接着写MemStore，然后客户端被通知提交数据成功；如果写WAL日志失败，客户端则被通知提交失败。这样做的好处是可以做到RegionServer宕机后的数据恢复。

因此，对于相对不太重要的数据，可以在Put/Delete操作时，通过调用Put.setWriteToWAL(false)或Delete.setWriteToWAL(false)函数，放弃写WAL日志，从而提高数据写入的性能。

**值得注意的是：谨慎选择关闭WAL日志，因为这样的话，一旦RegionServer宕机，Put/Delete的数据将会无法根据WAL日志进行恢复。**

## 2.3 批量写

通过调用HTable.put(Put)方法可以将一个指定的row key记录写入HBase，同样HBase提供了另一个方法：通过调用HTable.put(List<Put>)方法可以将指定的row key列表，批量写入多行记录，这样做的好处是批量执行，只需要一次网络I/O开销，这对于对数据实时性要求高，网络传输RTT高的情景下可能带来明显的性能提升。

## 2.4 多线程并发写

在客户端开启多个HTable写线程，每个写线程负责一个HTable对象的flush操作，这样结合定时flush和写buffer（writeBufferSize），可以既保证在数据量小的时候，数据可以在较短时间内被flush（如1秒内），同时又保证在数据量大的时候，写buffer一满就及时进行flush。下面给个具体的例子：

[复制代码](javascript:void(0);)

for (int i = 0; i < threadN; i++) {  
 Thread th = new Thread() {  
 public void run() {  
 while (true) {  
 try {  
 sleep(1000); //1 second  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
synchronized (wTableLog[i]) {  
 try {  
 wTableLog[i].flushCommits();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}  
 };  
 th.setDaemon(true);  
 th.start();  
}

## [HBase性能优化方法总结（三）：读表操作](http://www.cnblogs.com/panfeng412/archive/2012/03/08/hbase-performance-tuning-section3.html)

本文主要是从HBase应用程序设计与开发的角度，总结几种常用的性能优化方法。有关HBase系统配置级别的优化，可参考：[淘宝Ken Wu同学的博客](http://kenwublog.com/hbase-performance-tuning)。

下面是本文总结的第三部分内容：读表操作相关的优化方法。

# 3. 读表操作

## 3.1 多HTable并发读

创建多个HTable客户端用于读操作，提高读数据的吞吐量，一个例子：

[复制代码](javascript:void(0);)

static final Configuration conf = HBaseConfiguration.create();  
static final String table\_log\_name = “user\_log”;  
rTableLog = new HTable[tableN];  
for (int i = 0; i < tableN; i++) {  
 rTableLog[i] = new HTable(conf, table\_log\_name);  
 rTableLog[i].setScannerCaching(50);  
}

[复制代码](javascript:void(0);)

## 3.2 HTable参数设置

### 3.2.1 Scanner Caching

hbase.client.scanner.caching配置项可以设置HBase scanner一次从服务端抓取的数据条数，默认情况下一次一条。通过将其设置成一个合理的值，可以减少scan过程中next()的时间开销，代价是scanner需要通过客户端的内存来维持这些被cache的行记录。

有三个地方可以进行配置：1）在HBase的conf配置文件中进行配置；2）通过调用HTable.setScannerCaching(int scannerCaching)进行配置；3）通过调用Scan.setCaching(int caching)进行配置。**三者的优先级越来越高。**

### 3.2.2 Scan Attribute Selection

scan时指定需要的Column Family，可以减少网络传输数据量，否则默认scan操作会返回整行所有Column Family的数据。

### 3.2.3 Close ResultScanner

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）。

## 3.3 批量读

通过调用HTable.get(Get)方法可以根据一个指定的row key获取一行记录，同样HBase提供了另一个方法：通过调用HTable.get(List<Get>)方法可以根据一个指定的row key列表，批量获取多行记录，这样做的好处是批量执行，只需要一次网络I/O开销，这对于对数据实时性要求高而且网络传输RTT高的情景下可能带来明显的性能提升。

## 3.4 多线程并发读

在客户端开启多个HTable读线程，每个读线程负责通过HTable对象进行get操作。下面是一个多线程并发读取HBase，获取店铺一天内各分钟PV值的例子：

[复制代码](javascript:void(0);)

public class DataReaderServer {  
 //获取店铺一天内各分钟PV值的入口函数  
 public static ConcurrentHashMap<String, String> getUnitMinutePV(long uid, long startStamp, long endStamp){  
 long min = startStamp;  
 int count = (int)((endStamp - startStamp) / (60\*1000));  
 List<String> lst = new ArrayList<String>();  
 for (int i = 0; i <= count; i++) {  
 min = startStamp + i \* 60 \* 1000;  
 lst.add(uid + "\_" + min);  
 }  
 return parallelBatchMinutePV(lst);  
 }  
 //多线程并发查询，获取分钟PV值  
private static ConcurrentHashMap<String, String> parallelBatchMinutePV(List<String> lstKeys){  
 ConcurrentHashMap<String, String> hashRet = new ConcurrentHashMap<String, String>();  
 int parallel = 3;  
 List<List<String>> lstBatchKeys = null;  
 if (lstKeys.size() < parallel ){  
 lstBatchKeys = new ArrayList<List<String>>(1);  
 lstBatchKeys.add(lstKeys);  
 }  
 else{  
 lstBatchKeys = new ArrayList<List<String>>(parallel);  
 for(int i = 0; i < parallel; i++ ){  
 List<String> lst = new ArrayList<String>();  
 lstBatchKeys.add(lst);  
 }  
  
 for(int i = 0 ; i < lstKeys.size() ; i ++ ){  
 lstBatchKeys.get(i%parallel).add(lstKeys.get(i));  
 }  
 }  
   
 List<Future< ConcurrentHashMap<String, String> >> futures = new ArrayList<Future< ConcurrentHashMap<String, String> >>(5);  
   
 ThreadFactoryBuilder builder = new ThreadFactoryBuilder();  
 builder.setNameFormat("ParallelBatchQuery");  
 ThreadFactory factory = builder.build();  
 ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(lstBatchKeys.size(), factory);  
   
 for(List<String> keys : lstBatchKeys){  
 Callable< ConcurrentHashMap<String, String> > callable = new BatchMinutePVCallable(keys);  
 FutureTask< ConcurrentHashMap<String, String> > future = (FutureTask< ConcurrentHashMap<String, String> >) executor.submit(callable);  
 futures.add(future);  
 }  
 executor.shutdown();  
   
 // Wait for all the tasks to finish  
 try {  
 boolean stillRunning = !executor.awaitTermination(  
 5000000, TimeUnit.MILLISECONDS);  
 if (stillRunning) {  
 try {  
 executor.shutdownNow();  
 } catch (Exception e) {  
 // TODO Auto-generated catch block  
 e.printStackTrace();  
 }  
 }  
 } catch (InterruptedException e) {  
 try {  
 Thread.currentThread().interrupt();  
 } catch (Exception e1) {  
 // TODO Auto-generated catch block  
 e1.printStackTrace();  
 }  
 }  
   
 // Look for any exception  
 for (Future f : futures) {  
 try {  
 if(f.get() != null)  
 {  
 hashRet.putAll((ConcurrentHashMap<String, String>)f.get());  
 }  
 } catch (InterruptedException e) {  
 try {  
 Thread.currentThread().interrupt();  
 } catch (Exception e1) {  
 // TODO Auto-generated catch block  
 e1.printStackTrace();  
 }  
 } catch (ExecutionException e) {  
 e.printStackTrace();  
 }  
 }  
   
 return hashRet;  
 }  
 //一个线程批量查询，获取分钟PV值  
 protected static ConcurrentHashMap<String, String> getBatchMinutePV(List<String> lstKeys){  
 ConcurrentHashMap<String, String> hashRet = null;  
 List<Get> lstGet = new ArrayList<Get>();  
 String[] splitValue = null;  
 for (String s : lstKeys) {  
 splitValue = s.split("\_");  
 long uid = Long.parseLong(splitValue[0]);  
 long min = Long.parseLong(splitValue[1]);  
 byte[] key = new byte[16];  
 Bytes.putLong(key, 0, uid);  
 Bytes.putLong(key, 8, min);  
 Get g = new Get(key);  
 g.addFamily(fp);  
 lstGet.add(g);  
 }  
 Result[] res = null;  
 try {  
 res = tableMinutePV[rand.nextInt(tableN)].get(lstGet);  
 } catch (IOException e1) {  
 logger.error("tableMinutePV exception, e=" + e1.getStackTrace());  
 }  
  
 if (res != null && res.length > 0) {  
 hashRet = new ConcurrentHashMap<String, String>(res.length);  
 for (Result re : res) {  
 if (re != null && !re.isEmpty()) {  
 try {  
 byte[] key = re.getRow();  
 byte[] value = re.getValue(fp, cp);  
 if (key != null && value != null) {  
 hashRet.put(String.valueOf(Bytes.toLong(key,  
 Bytes.SIZEOF\_LONG)), String.valueOf(Bytes  
 .toLong(value)));  
 }  
 } catch (Exception e2) {  
 logger.error(e2.getStackTrace());  
 }  
 }  
 }  
 }  
  
 return hashRet;  
 }  
}  
//调用接口类，实现Callable接口  
class BatchMinutePVCallable implements Callable<ConcurrentHashMap<String, String>>{  
 private List<String> keys;  
  
 public BatchMinutePVCallable(List<String> lstKeys ) {  
 this.keys = lstKeys;  
 }  
  
 public ConcurrentHashMap<String, String> call() throws Exception {  
 return DataReadServer.getBatchMinutePV(keys);  
 }  
}

## 3.5 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。

## 3.6 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](http://punishzhou.iteye.com/blog/1277141)，[HBase的blockcache机制](http://www.cnblogs.com/raymondshiquan/archive/2011/07/24/2115397.html)，[hbase中的缓存的计算与使用](http://koven2049.iteye.com/blog/1002135)。

## [HTable和HTablePool使用注意事项](http://www.cnblogs.com/panfeng412/archive/2012/07/11/htable-and-htablepool-apply-notes.html)

HTable和HTablePool都是HBase客户端API的一部分，可以使用它们对HBase表进行CRUD操作。下面结合在项目中的应用情况，对二者使用过程中的注意事项做一下概括总结。

Configuration conf = HBaseConfiguration.create();

try (Connection connection = ConnectionFactory.createConnection(conf)) {

try (Table table = connection.getTable(TableName.valueOf(tablename)) {

// use table as needed, the table returned is lightweight

}

}

# HTable

[HTable](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html)是HBase客户端与HBase服务端通讯的Java API对象，客户端可以通过HTable对象与服务端进行CRUD操作（增删改查）。它的创建很简单：

Configuration conf = HBaseConfiguration.create();

HTable table = new HTable(conf, "tablename");

//TODO CRUD Operation……

HTable使用时的一些注意事项：

**1.   规避HTable对象的创建开销**

因为客户端创建HTable对象后，需要进行一系列的操作：检查.META.表确认指定名称的HBase表是否存在，表是否有效等等，整个时间开销比较重，可能会耗时几秒钟之长，因此最好在程序启动时一次性创建完成需要的HTable对象，如果使用Java API，一般来说是在构造函数中进行创建，程序启动后直接重用。

**2.   HTable对象不是线程安全的**

HTable对象对于客户端读写数据来说不是线程安全的，因此多线程时，要为每个线程单独创建复用一个HTable对象，不同对象间不要共享HTable对象使用，特别是在客户端auto flash被置为false时，由于存在本地write buffer，可能导致数据不一致。

**3.   HTable对象之间共享Configuration**

HTable对象共享Configuration对象，这样的好处在于：

* 共享ZooKeeper的连接：每个客户端需要与ZooKeeper建立连接，查询用户的table regions位置，这些信息可以在连接建立后缓存起来共享使用；
* 共享公共的资源：客户端需要通过ZooKeeper查找-ROOT-和.META.表，这个需要网络传输开销，客户端缓存这些公共资源后能够减少后续的网络传输开销，加快查找过程速度。

因此，与以下这种方式相比：

HTable table1 = new HTable("table1");

HTable table2 = new HTable("table2");

下面的方式更有效些：

Configuration conf = HBaseConfiguration.create();

HTable table1 = new HTable(conf, "table1");

HTable table2 = new HTable(conf, "table2");

备注：即使是高负载的多线程程序，也并没有发现因为共享Configuration而导致的性能问题；如果你的实际情况中不是如此，那么可以尝试不共享Configuration。

# HTablePool

[HTablePool](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTablePool.html)可以解决HTable存在的线程不安全问题，同时通过维护固定数量的HTable对象，能够在程序运行期间复用这些HTable资源对象。

Configuration conf = HBaseConfiguration.create();

HTablePool pool = new HTablePool(conf, 10);

1.   HTablePool可以自动创建HTable对象，而且对客户端来说使用上是完全透明的，可以避免多线程间数据并发修改问题。

2.   HTablePool中的HTable对象之间是公用Configuration连接的，能够可以减少网络开销。

HTablePool的使用很简单：每次进行操作前，通过HTablePool的getTable方法取得一个HTable对象，然后进行put/get/scan/delete等操作，最后通过HTablePool的putTable方法将HTable对象放回到HTablePool中。

下面是个使用HTablePool的简单例子：

[复制代码](javascript:void(0);)

public void createUser(String username, String firstName, String lastName, String email, String password, String roles) throws IOException {

　　HTable table = rm.getTable(UserTable.NAME);

　　Put put = new Put(Bytes.toBytes(username));

　　put.add(UserTable.DATA\_FAMILY, UserTable.FIRSTNAME,

　　Bytes.toBytes(firstName));

　　put.add(UserTable.DATA\_FAMILY, UserTable.LASTNAME,

　　　　Bytes.toBytes(lastName));

　　put.add(UserTable.DATA\_FAMILY, UserTable.EMAIL, Bytes.toBytes(email));

　　put.add(UserTable.DATA\_FAMILY, UserTable.CREDENTIALS,

　　　　Bytes.toBytes(password));

　　put.add(UserTable.DATA\_FAMILY, UserTable.ROLES, Bytes.toBytes(roles));

　　table.put(put);

　　table.flushCommits();

　　rm.putTable(table);

}

Hbase和DBMS比较：

查询数据不灵活：

1. 不能使用column之间过滤查询
2. 不支持全文索引。使用solr和hbase整合完成全文搜索。
   1. 使用MR批量读取hbase中的数据，在solr里面建立索引（no store）之保存rowkey的值。
   2. 根据关键词从索引中搜索到rowkey（分页）
   3. 根据rowkey从hbase查询所有数据