quarrier工程，为二进制建索引，用于利用指纹快速检索文档

创建原因：

为听歌识曲所写的指纹检索服务，由于为了顾及几百万的音频，以及在大量音频的情况下性能问题，对内存的访问次数几利用率到了变态的地步，笨重又不容易扩展。此时如果有一些其他的服务需要用音频指纹来识别音频，而这些服务的音频数量比较小，如果将这些音频加到虾米的指纹库中，或多或少会相互影响；如果用当前的指纹服务为这些服务搭建一个新的服务，又太笨重，虾米听歌识趣为了速度，对音频指纹的键值用完美hash进行映射，这样仅仅倒排索引表就要占用4G多的内存，更何况为了更高效的使用内存，虾米的音频指纹服务用c++编写，服务的搭建和扩容都不方便。综上种种情况，希望可以用java写一个针对小量指纹库的索引服务，方便以后集团中类似的需求。

模块设计：

1. 建库模块

功能：为二进制文档创建1. 倒排索引文件；2. 倒排文件；3. 正排索引文件；4. 正排文件。

* 倒排索引文件内存中结构：TreeMap结构，key保存子指纹的值；value保存倒排项在倒排文件中首地址以及倒排项的个数。这里用TreeMap结构方便对key进行排序。
* 倒排索引文件磁盘中结构：按照子指纹的大小依次存入文件，每个子指纹存放三个数据：docID，addr，length。
* 倒排文件内存中结构：一个倒排项的数组，每个倒排项包含两个数据：docID，offset；其中offset是该倒排项所属子指纹在docID所指文档中的偏移量。
* 倒排文件磁盘中结构：倒排项目按照子指纹的顺序依次写入文件。每个倒排项包含docID和offset两个数据。
* 正排索引文件内存中结构：HashMap结构，key保存docID，value保存文档在正排文件中的偏移量，文档的长度以及原始的id（docId是本检索系统给定的id，并不一定是原始文档id）。
* 正排索引文件磁盘中结构：每个文档依次写入docID，length，offset，orgID到正排文件中。
* 正排文件内存中结构：一个byte的数组，存放原始文件二进制内容，原始文件存放的地址和正排索引表中的offset对应。
* 正排文件磁盘中结构：将内存中的byte数组存放到文件中。



类图如上所示：

* CreatorBunch:提供添加docment接口，创建CreatorTask对象；
* CreatorTask:为批量docment创建segment对象，将segment添加到MergerBunch中；
* MergerBunch:提供添加segment接口，创建MergerTask对象；
* MergerTask:用多路合并算法将批量segment对象合并，将合并后的segment对象添加到MergerBunch或者WriterBunch中；
* WriterBunch:提供添加segment接口，创建WriterTask对象；
* WriterTask:写磁盘操作，将segment对象数据依次写入磁盘。



* 建库主线程或者实时添加文档的用户线程创建doc对象添加到creatorBunch中，creatorBunch维护一个线程安全的doc list，当文档数达到阈值时，创建一个createTask并清空列表，添加到调度中心（任务调度中心就是我之前讲的线程池）。
* 工作线程获取createTask并执行，为这批doc创建倒排索引和正排索引，并创建segment对象持有倒排索引和正排索引。创建完后将segment添加到MergerBunch对象中。
* MergerBunch对象接收segment对象，并保存在一个多线程安全队列中，当segment数量达到一个阈值时，创建一个mergeTask并清空列表，然后将mergeTask添加到任务调度中心。
* 工作线程获取mergeTask，将这批segment合并成一个更大的segment，通过判断设置的合并深度或者segment大小，或者将segment添加到merge对象中，或者将segment添加到writerBunch对象中
* WriterBunch对象接受segment对象，也维护一个多线程安全的segment列表，如果segment数量大于阈值，创建writerTask，清空列表，并将writerTask添加到任务调度中心。
* 工作线程获取writertask，对这一批segment执行写磁盘的操作。

每个creatorTask执行的任务：

为doc创建倒排索引：

1. 迭代每一个doc，取出doc的content，content是一个byte数组。
2. 以四个byte为单位构造一个key，迭代每一个key。
3. 计算该key值再doc中的偏移量，结合docID，一并保存在一个以key为键值的TreeMap中。
4. 所有doc都迭代完毕后，将treeMap中的数据写入segment的index和indexData结构中，index和indexData分别代表倒排索引文件和倒排文件，其index是一个treeMap键值为key值，value为key值倒排项个数和key值倒排项在倒排表中的起始地址。indexData为倒排项数组，依次存放各倒排项，倒排项包含docID和offset。

为doc创建正排文件：

1. 迭代每一个doc，取出该doc的content，content是一个byte数组。
2. 将doc的content写入segment的positData结构中，positData代表正排文件
3. 记录下该doc再positData中的offset，结合doc的length一并存入segment的posit文件中。Posit是一个map，key值为docID，value包含doc的长度和在positData中的偏移量。Posit代表正排索引表。

每个MergerTask执行的任务：

合并多个segment的index，posit，indexData，positData数据到一个大的segment对象中。合并正排文件比较简单，将每个segment中positData的数据存入新的segment中，然后将posit中的docId对应的positData中的offset对应调整以下就可以。

合并倒排文件比较复杂，需要用到多路合并算法。Segment的index是treeMap结构，所有的key值都已经按照大小顺序排列好。

1. 建立一个优先权队列，每次从队列中取到是最小的。
2. 从每个segment中取出最小的key值，放入优先权队列。
3. 不断从队列中取出key值，直到队列为空。
4. 判断该key值是否已经存在于新的segment的index中，如果存在，更新length，将该key值对应的倒排项copy到新segment的indexData的尾部。如果不存在，插入到新segment的index，记录下该key值的偏移量，和临时长度。
5. 处理完当前key值后，从该key值所属的segment中取出下一个key放入优先权队列。

每个WriterBunch执行的任务：

将segment对象中的index，indexData，posit，positData数据分别写入.in，.iv，.po，.pv文件中，最后将segment的编号写入segment文件中。

.in文件：按照key值从小到大的顺序依次写入，每个key值写入三个数据：key，length，offset。

.iv文件：将indexData中的倒排项依次写入，每个倒排项包含两个数据：docID，offset。

.po文件：每个文档的正排索引写入该文件，每个文档写入四个数据：docID，length，offset，orgid。

.pv文件：原始文件数据依次写入该文件。

Flush操作：

按照我们的多线程建库流程描述，只有creatorBunch持有的doc数目大于阈值时，才会创建creatorTask；同理只有MergerBunch持有的segment数目大于阈值时，才会创建mergerTask；只有writerBunch持有的segment数目大于阈值时，才会创建writerTask。这样就会出现一个问题，加入最后建库完成后，不会再有文档加入了，而此时的creatorBunch，MergerBunch和WriterBunch持有的对象有不满阈值的情况，这样内存中就始终存在一些数据无法建库以及写入磁盘文件。对于这种情况，设计了flush操作。步骤如下：

* 1. 当creatorBunch被设置了flush操作后，将持有的所有doc对象无论其数目是否达到阈值，都创建一个creatorTask对象。
  2. 同时creatorBunch对象设置creatorTask的静态标志flush，所有的creatorTask对象都能访问这个标志。CreatorTask有一个静态的对象计数器count。创建一个creatorTask对象时count+1；一个creatorTask对象执行完操作时count-1。当creatorTask对象中判断当前计数器为0且flush标志为true时，调用MergerBunch的flush操作，同时将CreatorTask的flush标志设置为false。
  3. MergerBunch被设置了flush操作后，将持有的所有segment对象无论其数目是否达到阈值，都创建一个MergerTask对象。
  4. 同时MergerBunch对象设置MergerTask的静态标志flush，所有的MergerTask对象都能访问这个标志。MergerTask有一个静态的对象计数器count。创建一个MergerTask对象时count+1；一个MergerTask对象执行完操作时count-1。当MergerTask对象中判断当前计数器为0且flush标志为true时，调用对应Bunch的flush操作，同时将MergerTask的flush标志设置为false。
  5. WriterBunch被调用了flush操作后，将持有的所有segment对象无论其数目是否达到阈值，都创建一个WriterTask对象。等待工作线程写入磁盘，过程结束。

flush方案当前缺点：

由于CreatorTask和MergerTask的操作的flush标志是类层次的静态标志，所以要求创建这些任务的CreatorBunch和MergerBunch只能有一个，这个不是问题，我们本来要求建库的对象唯一。

另一个比较头疼的问题，我们只能设置一层mergeBunch。因为如果设置多层的mergerBunch时，不同层创建的MergerTask也是不同层次的，但是都共享MergerTask的flush标志。这样会相互影响。

可以考虑将计数器设置在creatorBunch和MergerBunch对象中，这样可以保证同一个对象创建出来的任务，共享一个计数器，就可以避免不同层次的MergerBunch创建的MergerTask相互影响了。

1. 加载模块

功能：

将磁盘中的倒排索引文件（.in），倒排文件（.iv），正排索引文件（.po），正排文件（.pv）读入内存，创建相应的数据结构保存。因为计算量巨大，所以指纹搜索所有的数据都要加载到内存中。

本工程支持单线程加载和多线程加载，单线程加载只是简单的依次加载每个segment，并为每个segment创建一个内存对象维护。多线程加载为每个segment文件创建一个任务，由多线程并行去执行加载任务。



类图如上：

* Load类：到指定路径下加载build过程创建的文件，到内存结构中
* SegmentLoad类：Load类的子类，将加载的数据文件以segment的内存形式组织
* SegmentLoadTask类：单线程加载模式下此类无用；在多线程加载模式下，由SegmentLoad类创建，执行具体加载一个segment的操作。
* OtherLoad和OtherLoadTask类：待扩展的类。

多线程加载流程图如上：

* Load对象创建多个LoadTask任务，添加到任务调度中心的任务队列中；
* 工作线程取LoadTask任务并执行load代码；
* 每个工作线程会加载指定的文件并在内存中创建一个相应的对象保存这些数据。
* 工作线程加载完所有数据到memeryData后，将memeryData添加到对应的searcher对象中

每个LoadTask执行的任务：

将.in .iv .po .pv文件中的内容分别load进内存中的相应结构。其过程和writerTask中的过程相逆。不过内存中持有这些数据的结构未必时前面定义的segment结构，可以定义其他各种快速高效的结构保存这些数据。本工程中目前只实现了segment结构。

1. 查询模块

功能：

接收一段二进指纹，输出这段二进制指纹对应的文档ID



查询模块类图如上：

* Search类：搜索类的基类，定义了多线程搜索时创建searchTask的流程，单线程搜索在各个子类中执行。
* SegmentSearch类：实现在segment结构中检索指纹的操作
* SegmentSearchTask类：在单线程检索中该类无用；在多线程检索中负责执行在一个segment中的检索。



多线程查询的流程图如上：

* Searcher对象为每个子索引创建一个SearcherTask，SearcherTask负责具体的查询操作
* 每个SearcherTask都是ITask的派生类，可以放入调度中心中执行。
* 工作获取SearcherTask任务，执行查询代码，将查询结构放入结果队列中。
* 结果队列是一个多线程队列，由Searcher对象创建，传入每个SearcherTask中。查询到传入文档id，未查询到传入-1。
* Search对象从结果队列中取结果，如果取得一个正确结果，则直接返回并终止其他线程的搜索。如果-1个数等于创建任务的个数，则返回-1。

每个searchTask执行的工作步骤：

* 对于传递进来的二进制数组，以4个字节为单位，每四个字节构造一个key值。
* 迭代每一个key值，用key值在倒排索引表中查询该key值对应倒排项的首地址和倒排个数。
* 迭代每个倒排项，从倒排项中解析出docid和offset。迭代完毕跳至第二步
* 利用docid查询正排索引表，找到文档在正排文件中的地址。
* 利用文档地址和offset找到该key值在正排文件中的地址，并读取出相对应的raw文件
* 比较raw文件和传入的二进制数组的汉明距离，如果汉明距离小于某个阈值，则匹配成功并返回，不成功继续跳至第3步。