知乎 首发于 Elasticsearch技术研讨

Lucene解析 - IndexWriter

木洛
招贤纳士，欢迎自荐！

已关注

前言

在上一篇文章我们介绍了Lucene的基本概念，在本篇文章我们将深入Lucene中最核心的类之一IndexWriter，来探索Lucene中数据写入和索引构建的整个过程。

IndexWriter

```
// initialization
Directory index = new NIOFSDirectory(Paths.get("/index"));
IndexWriterConfig config = new IndexWriterConfig();
IndexWriter writer = new IndexWriter(index, config);
// create a document
Document doc = new Document();
doc.add(new TextField("title", "Lucene - IndexWriter", Field.Store.YES));
doc.add(new StringField("content", "招人，求私信", Field.Store.YES));
// index the document
writer.addDocument(doc);
writer.commit();
```

1. 初始化：初始化IndexWriter必要的两个元素是Directory和IndexWriterConfig，Directory是Lucene中数据持久层的抽象接口，通过这层接口可以实现很多不同类型的数据持久层，例如本地文件系统、网络文件系统、数据库或者是分布式文件系统。IndexWriterConfig内提供了很多可配置的高级参数，提供给高级玩家进行性能调优和功能定制，它提供的几个关键参数后面会细说。
2. 构造文档：Lucene中文档由Document表示，Document由Field构成。Lucene提供多种不同类型的Field，其FiledType决定了它所支持的索引模式，当然也支持自定义Field，具体方式可参考上一篇文章。
3. 写入文档：通过IndexWriter的addDocument函数写入文档，写入时同时根据FieldType创建不同的索引。文档写入完成后，还不可被搜索，最后需要调用IndexWriter的commit，在commit完后Lucene才保证文档被持久化并且是searchable的。

以上就是Lucene的一个简明的数据写入流程，核心是IndexWriter，整个过程被抽象的非常简洁明了。一个设计优良的库的最大特点，就是可以让普通玩家以非常小的代价学习和使用，同时又照顾高级玩家能够提供可调节的性能参数和功能定制能力。

IndexWriterConfig

IndexWriterConfig内提供了一些供高级玩家做性能调优和功能定制的核心参数，我们列几个主要的看下：

- **IndexDeletionPolicy**：Lucene开放对commit point的管理，通过对commit point的管理可以实现例如snapshot等功能。Lucene默认配置的DeletionPolicy，只会保留最新的一个commit point。
- **Similarity**：搜索的核心是相关性，Similarity是相关性算法的抽象接口，Lucene默认实现了TF-IDF和BM25算法。相关性计算在数据写入和搜索时都会发生，数据写入时的相关性计算称为Index-time boosting，计算Normalizaiton并写入索引，搜索时的相关性计算称为query-time boosting。
- **MergePolicy**：Lucene内部数据写入会产生很多Segment，查询时会对多个Segment查询并合并结果。所以Segment的数量一定程度上会影响查询的效率，所以需要对Segment进行合并，合并的过程就称为Merge，而何时触发Merge由MergePolicy决定。
- **MergeScheduler**：当MergePolicy触发Merge后，执行Merge会由MergeScheduler来管理。Merge通常是比较耗CPU和IO的过程，MergeScheduler提供了对Merge过程定制管理的能力。
- **Codec**：Codec可以说是Lucene中最核心的部分，定义了Lucene内部所有类型索引的Encoder和Decoder。Lucene在Config这一层将Codec配置化，主要目的是提供对不同版本数据的处理能力。对于Lucene用户来说，这一层的定制需求通常较少，能玩Codec的通常都是顶级玩家了。

- **FlushPolicy**：FlushPolicy决定了In-memory buffer何时被flush，默认的实现会根据RAM大小和文档个数来判断Flush的时机，FlushPolicy会在每次文档add/update/delete时调用判定。
- **MaxBufferedDoc**：Lucene提供的默认FlushPolicy的实现FlushByRamOrCountsPolicy中允许DocumentsWriterPerThread使用的最大文档数上限，超过则触发Flush。
- **RAMBufferSizeMB**：Lucene提供的默认FlushPolicy的实现FlushByRamOrCountsPolicy中允许DocumentsWriterPerThread使用的最大内存上限，超过则触发flush。
- **RAMPerThreadHardLimitMB**：除了FlushPolicy能决定Flush外，Lucene还会有一个指标强制限制DocumentsWriterPerThread占用的内存大小，当超过阈值则强制flush。
- **Analyzer**：即分词器，这个通常是定制化最多的，特别是针对不同的语言。

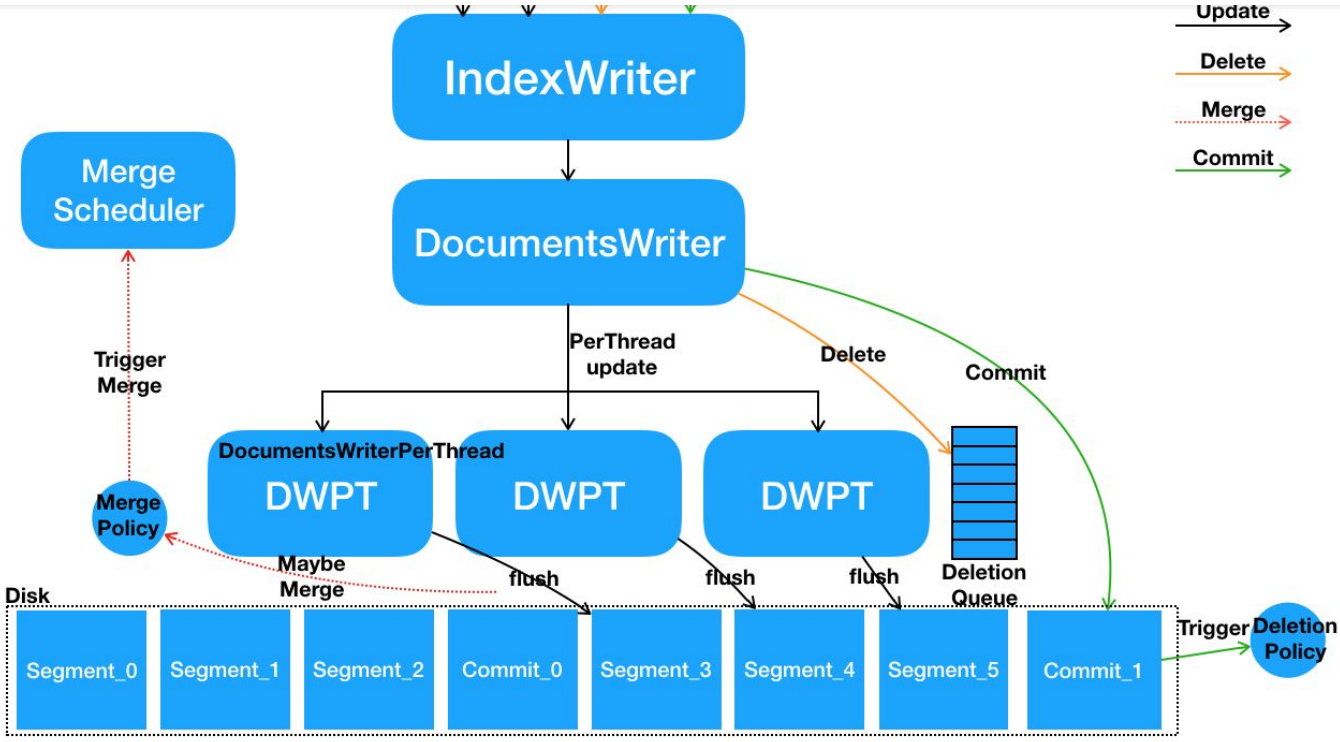
核心操作

IndexWriter提供很简单的几种操作接口，这一章节会做一个简单的功能和用途解释，下一个章节会对其内部实现做一个详细的剖析。IndexWrite的提供的核心API如下：

- addDocument：比较纯粹的一个API，就是向Lucene内新增一个文档。Lucene内部没有主键索引，所有新增文档都会被认为一个新的文档，分配一个独立的docId。
- updateDocuments：更新文档，但是和数据库的更新不太一样。数据库的更新是查询后更新，Lucene的更新是查询后删除再新增。流程是先delete by term，后add document。但是这个流程又和直接先调用delete后调用add效果不一样，只有update能够保证在Thread内部删除和新增保证原子性，详细流程在下一章节会细说。
- deleteDocument：删除文档，支持两种类型删除，by term和by query。在IndexWriter内部这两种删除的流程不太一样，在下一章节再细说。
- flush：触发强制flush，将所有Thread的In-memory buffer flush成segment文件，这个动作可以清理内存，强制对数据做持久化。
- prepareCommit/commit/rollback：commit后数据才可被搜索，commit是一个二阶段操作，prepareCommit是二阶段操作的第一个阶段，也可以通过调用commit一步完成，rollback提供了回滚到last commit的操作。
- maybeMerge/forceMerge：maybeMerge触发一次MergePolicy的判定，而forceMerge则触发一次强制merge。

数据路径

上面几个章节介绍了IndexWriter的基本流程、配置和核心接口，非常简单和易理解。这一章节，我们将深入IndexWriter内部，来探索其内核实现。



如上是IndexWriter内部核心流程架构图，接下来我们将以add/update/delete/commit这些主要操作来讲解IndexWriter内部的数据路径。

并发模型

IndexWriter提供的核心接口都是线程安全的，并且内部做了特殊的并发优化来优化多线程写入的性能。IndexWriter内部为每个线程都会单独开辟一个空间来写入，这块空间由DocumentsWriterPerThread来控制。整个多线程数据处理流程为：

1. 多线程并发调用IndexWriter的写接口，在IndexWriter内部具体请求会由DocumentsWriter来执行。DocumentsWriter内部在处理请求之前，会先根据当前执行操作的Thread来分配DocumentsWriterPerThread。
2. 每个线程在其独立的DocumentsWriterPerThread空间内部进行数据处理，包括分词、相关性计算、索引构建等。
3. 数据处理完毕后，在DocumentsWriter层面执行一些后续动作，例如触发FlushPolicy的判定等。

引入了DocumentsWriterPerThread（后续简称为DWPT）后，Lucene内部在处理数据时，整个处理步骤只需要对以上第一步和第三步进行加锁，第二步完全不用加锁，每个线程都在自己独立的空间内处理数据。而通常来说，第一步和第三步都是非常轻量级的，而第二步是对计算和内存资源消耗最大的。所以这样做之后，能够将加锁的时间大大缩短，提高并发的效率。每个DWPT内单独包含一个In-memory buffer，这个buffer最终会flush成不同的独立的segment文件。

动作可能会涉及删除不同线程空间内的数据，这里Lucene也采取了一种特殊的交互方式来降低锁的开销，在剖析delete操作时会细说。

在搜索场景中，全量构建索引的阶段，基本是纯新增文档式的写入，而在后续增量索引阶段（特别是数据源是数据库时），会涉及大量的update和delete操作。从原理上来分析，一个最佳实践是包含相同唯一主键Term的文档分配相同的线程来处理，使数据更新发生在一个独立线程空间内，避免跨线程。

add & update

add接口用于新增文档，update接口用于更新文档。但Lucene的update和数据库的update不太一样。数据库的更新是查询后更新，Lucene的更新是查询后删除再新增，不支持更新文档内部分列。流程是先delete by term，后add document。

IndexWriter提供的add和update接口，都会映射到DocumentsWriter的udpate接口，看下接口定义：

```
long updateDocument(final Iterable<? extends IndexableField> doc, final Analyzer analy
    final Term delTerm) throws IOException, AbortingException
```

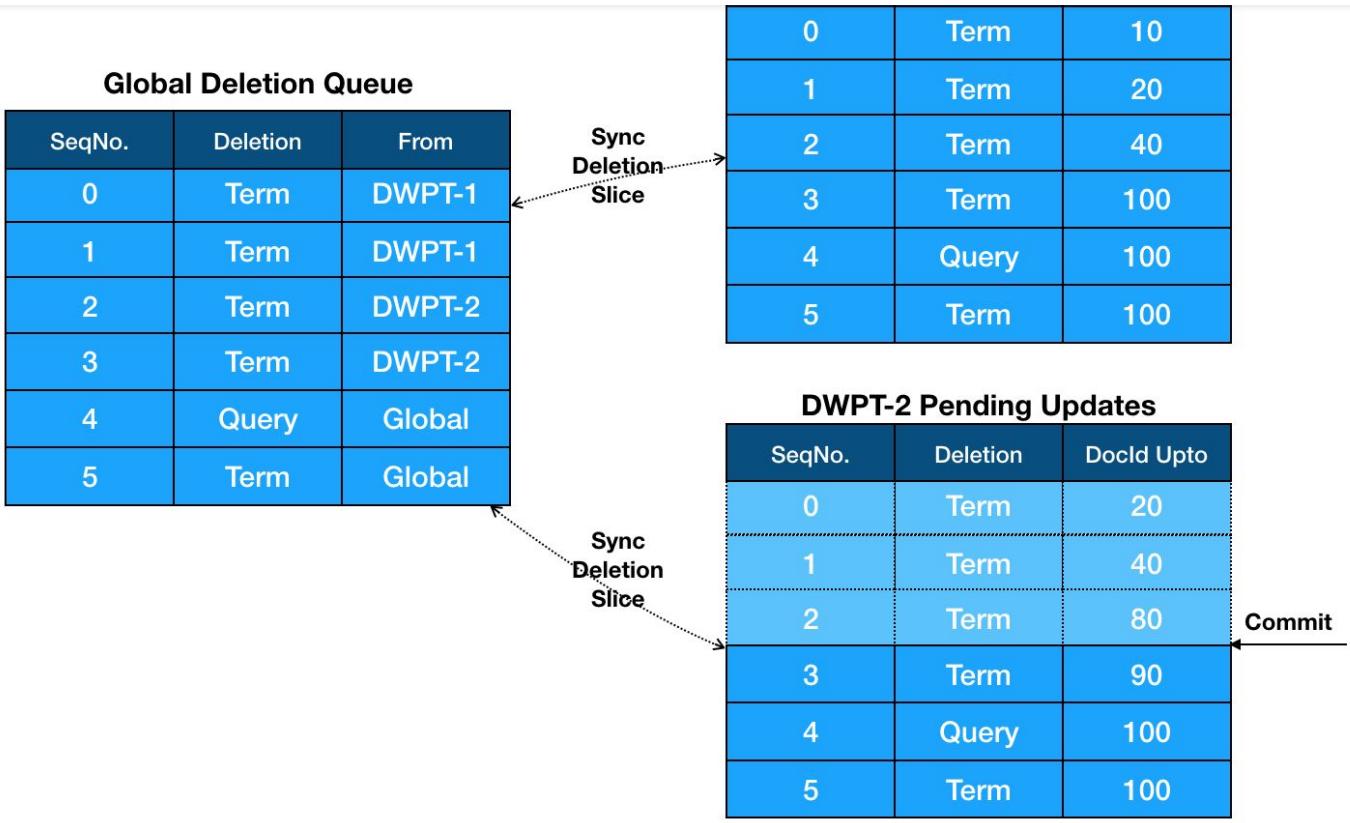
这个函数内的处理流程是：

1. 根据Thread分配DWPT
2. 在DWPT内执行delete
3. 在DWPT内执行add

关于delete操作的细节在下一小结详细说，add操作会直接将文档写入DWPT内的In-memory buffer。

delete

delete相对add和update来说，是完全不同的一个数据路径。而且update和delete虽然内部都会执行数据删除，但这两者又是不同的数据路径。文档删除不会直接影响In-memory buffer内的数据，而是会有另外的方式来达到删除的目的。



在Delete路径上关键的数据结构就是Deletion queue，在IndexWriter内部会有一个全局的Deletion Queue，称为Global Deletion Queue，而在每个DWPT内部，还会有一个独立的Deletion Queue，称为Pending Updates。DWPT Pending Updates会与Global Deletion Queue进行双向同步，因为文档删除是全局范围的，不应该只发生在DWPT范围内。

Pending Updates内部会按发生顺序记录每个删除动作，并且标记该删除影响的文档范围，文档影响范围通过记录当前已写入的最大DocId（DocId Upto）来标记，即代表这个删除动作只删除小于等于该DocId的文档。

update接口和delete接口都可以进行文档删除，但是有一些差异：

- update只能进行by term的文档删除，而delete除了by term，还支持by query。
- update的删除会先作用于DWPT内部，后作用于Global，再由Global同步到其他DWPT。
- delete的删除会作用在Global级别，后异步同步到DWPT级别。

update和delete流程上的差异也决定了他们行为上的一些差异，update的删除操作会先发生在DWPT内部，并且是和add同时发生，所以能够保证该DWPT内部的delete和add的原子性，即保证在add之前的所有符合条件的文档一定被删除。

DWPT Pending Updates里的删除操作什么时候会真正作用于数据呢？在Lucene Segment内部，数据实际上并不会被真正删除。Segment中有一个特殊的文件叫live docs，内部是一个位图的数据结构，记录了这个Segment内部哪些DocId是存活的，哪些DocId是被删除的。所以删

它关联的所有doc，所以很明显这个会发生在倒排索引构建时。而Query删除要求执行一次完整的查询后才能拿到其对应的docId，所以会发生在segment被flush完成后，基于flush后的索引文件构建IndexReader后执行搜索才能完成。

还有一点要注意的是，live docs只影响倒排，所以在live docs里被标记删除的文档没有办法通过倒排索引检索出，但是还能够通过doc id查询到store fields。当然文档数据最终是会被真正物理删除，这个过程会发生在merge时。

flush

flush是将DWPT内In-memory buffer里的数据持久化到文件的过程，flush会在每次新增文档后由FlushPolicy判定自动触发，也可以通过IndexWriter的flush接口手动触发。

每个DWPT会flush成一个segment文件，flush完成后这个segment文件是不可被搜索的，只有在commit之后，所有commit之前flush的文件才可被搜索。

commit

commit时会触发数据的一次强制flush，commit完成后再此之前flush的数据才可被搜索。commit动作会触发生成一个commit point，commit point是一个文件。Commit point会由IndexDeletionPolicy管理，lucene默认配置的策略只会保留last commit point，当然lucene提供其他多种不同的策略供选择。

merge

merge是对segment文件合并的动作，合并的好处是能够提高查询的效率以及回收一些被删除的文档。Merge会在segment文件flush时触发MergePolicy来判定自动触发，也可通过IndexWriter进行一次force merge。

IndexingChain

前面几个章节主要介绍了IndexWriter内部各个关键操作的流程，本小节会介绍最核心的DWPT内部对文档进行索引构建的流程。Lucene内部索引构建最关键的概念是IndexingChain，顾名思义，链式的索引构建。为啥是链式的？这个和Lucene的整个索引体系结构有关系，Lucene提供了各种不同类型的索引类型，例如倒排、正排（列存）、StoreField、DocValues等。每个不同的索引类型对应不同的索引算法、数据结构以及文件存储，有些是列级别的，有些是文档级别的。

去尝试。

在IndexWriter内部，indexing chain上索引构建顺序是invert index、store fields、doc values和point values。有些索引类型处理文档后会将索引内容直接写入文件（主要是store field和term vector），而有些索引类型会先将文档内容写入memory buffer，最后在flush的时候再写入文件。能直接写入文件的索引，通常是文档级的索引，索引构建可以文档级的增量构建。而不能写入文件的索引，例如倒排，则必须等Segment内所有文档全部写入完毕后，会先对Term进行一个全排序，之后才能构建索引，所以必须要有一个memory-buffer先缓存所有文档。

前面提到，IndexWriterConfig支持配置Codec，Codec就是对应每种类型索引的Encoder和Decoder。在上图可以看到，在Lucene 7.2.1版本中，主要有这么几种Codec：

- BlockTreeTermsWriter：倒排索引对应的Codec，其中倒排表部分使用Lucene50PostingsWriter(Block方式写入倒排链)和Lucene50SkipWriter(对Block的SkipList索引)，词典部分则是使用FST（针对倒排表Block级的词典索引）。
- CompressingTermVectorsWriter：对应Term vector索引的Writer，底层是压缩Block格式。
- CompressingStoredFieldsWriter：对应Store fields索引的Writer，底层是压缩Block格式。
- Lucene70DocValuesConsumer：对应Doc values索引的Writer。
- Lucene60PointsWriter：对应Point values索引的Writer。

这一章节主要了解IndexingChain内部的文档索引处理流程，核心是链式分阶段的索引，并且不同类型索引支持Codec可配置。

知乎 首发于 Elasticsearch技术研讨

这篇文章主要从一个全局视角来讲解IndexWriter的配置、接口、并发模型、核心操作的数据路径以及索引链，在之后的文章中，会再深入索引链中每种不同类型索引的构建流程，探索其memory-buffer的实现、索引算法以及数据存储格式。

编辑于 2018-04-17

编程 Lucene Elasticsearch

文章被以下专栏收录

Elasticsearch技术研讨
招 Elasticsearch 内核研发：base 杭州。

关注专栏

推荐阅读

Lucene解析 - 基本概念

木洛

elasticsearch

从 10 秒到 2 秒！ElasticSearch 性能调优

创宇前端

剖析Ela IndexS

前言前两 查询原理 能》，在 一篇会介 询性能的 技术——

阿里云云

12 条评论

切换为时间排序

写下你的评论...

john wu 2018-04-

john wu 2018-04-18

该评论已删除

木洛 (作者) 回复 john wu 2018-04-18

嗯，这里写的有问题，列存和DocValues是重复的。从结构看，DocValues和StoreFiled都算正排吧，一个是行存一个是列存。

1

john wu 2018-04-19

引用"有些索引类型处理文档后会将索引内容直接写入文件"，直接写入肯定不现实的吧，最起码得来个block压缩下吧，😄

1

木洛 (作者) 回复 john wu 2018-04-19

嗯是的，先攒够一个Block，压缩后直接写入。相比倒排，是攒够一个segment，之后再写入。

2

john wu 2018-04-19

比如storefield每16k写一把

1

bigquestion 2018-07-04

请问一个问题：lucene7.2.1中DocumentsWriterPerThread.java中190行

```
// this should be the last call in the ctor
// it really sucks that we need to pull this within the ctor and pass this ref to the chain!
consumer = indexWriterConfig.getIndexingChain().getChain(this);
this.enableTestPoints = enableTestPoints;
```

既然 他觉得这样写 很sucks 为啥还要这样写。为啥不改为工厂方法来防止this引用的逸出。

1

青鱼 2019-08-

请教一下，Lucene 的 commit，flush 与 ElasticSearch 里的 refresh 和 flush 是什么对应

白奕新 回复 青鱼 2019-09-03

es的refresh=lucene的flush，es的flush=lucene的commit

1

白奕新 2019-09-03

大佬，看完文章有一点疑惑。在es中，我的理解是每台node都会有一块index buffer内存，数据都被写入这块index buffer以后再被refresh下去，一次refresh产生一个segment file，不关乎thread数。但是根据您上面的描述，每个thread在写入的时候都会有自己相应的buffer，refresh时候各自的buffer产生各自的segment file？跟我的理解不大一样，是否是我理解错了？希望得到回复，感谢！email:byx313@yahoo.com

赞

xzy 回复 白奕新 2019-12-25

该评论已删除

白奕新 回复 xzy 2019-12-28

不是的，index buffer是es node级别的

赞

xzy 2019-12-25

你好，请问 ES refresh 以后便可进行搜索，refresh 是否对应 luence 的 flush，如果是，那么为什么 lucene flush 以后却不能搜索呢？望解答，谢谢

赞

Chris Lu 回复 xzy 04-25

lucene flush之后 需要更新reader 可以通过openIfChange方法获得最新的Reader

1