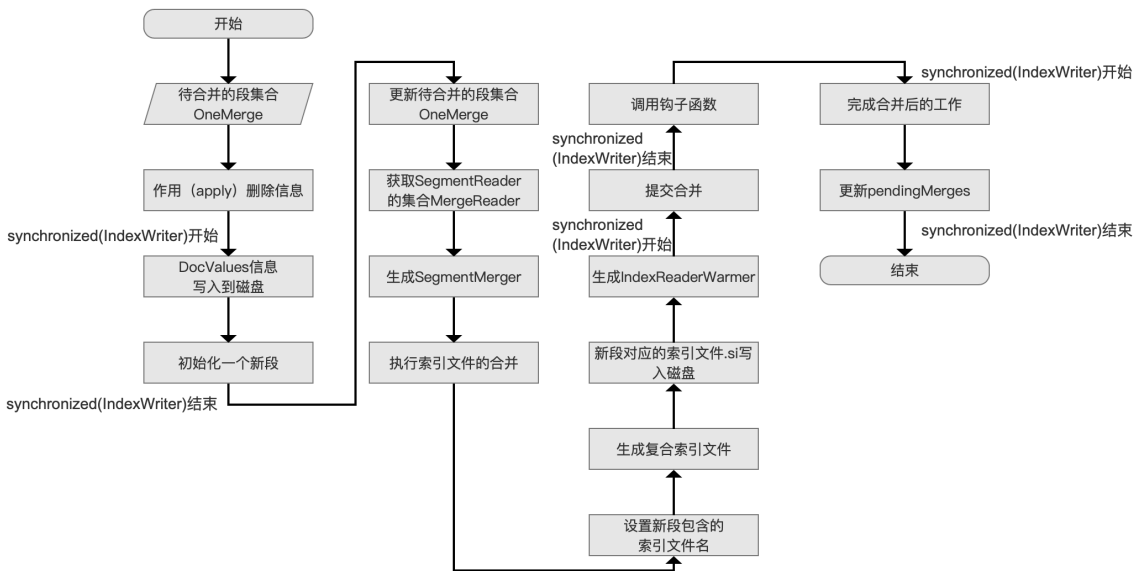


执行段的合并 (五)

本文承接[执行段的合并\(四\)](#)，继续介绍执行段的合并的剩余的流程，下面先给出执行段的合并的流程图：

图1:

[点击查看大图](#)

提交合并

MergeState

在文章[执行段的合并（四）](#)中我们说到，生成MergeState的过程中完成了几个任务，根据IndexWriter是否设置了IndexSort（见文章[索引文件之si](#)中关于IndexSort的介绍）可以将任务划分为如下两类：

- 设置了IndexSort
 - 任务一：对每一个待合并的段进行段内排序
 - 任务二：对合并后的新段进行段内排序
 - 任务三：获得所有待合并的段的被删除的文档号与段内真正的文档号的映射DocMap[]
- 未设置IndexSort
 - 任务三：获得所有待合并的段的被删除的文档号与段内真正的文档号的映射DocMap[]

任务一：对每一个待合并的段进行段内排序

执行任务一的目的其实是为了任务二准备的，因为只有每一个待合并的段是段内有序的，才能实现将这些段合并为一个段内有序的新段。

我们先通过一个例子来介绍段内排序对搜索结果的影响：

图2:

```

41 IndexWriterConfig indexWriterConfig = new IndexWriterConfig(analyzer);
42 indexWriter = new IndexWriter(directory, indexWriterConfig);
43 SortField sortField1 = new SortField( field: "age", SortField.Type.LONG);
44 SortField sortField2 = new SortField( field: "label", SortField.Type.STRING);
45 SortField[] allSortFields = new SortField[] { sortField1, sortField2};
46 Sort sort = new Sort(allSortFields);
47 indexWriterConfig.setIndexSort(sort);

```

图3:

```

49 FieldType type = new FieldType();
50 type.setStored(true);
51 type.setTokenized(true);
52 type.setIndexOptions(IndexOptions.DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS);
53 // 文档0
54 Document doc = new Document();
55 doc.add(new Field( name: "author", value: "author0", type));
56 doc.add(new NumericDocValuesField( name: "age", value: 10));
57 doc.add(new SortedDocValuesField( name: "label", new BytesRef( text: "f")));
58 indexWriter.addDocument(doc);
59 // 文档1
60 doc = new Document();
61 doc.add(new Field( name: "author", value: "author1", type));
62 doc.add(new NumericDocValuesField( name: "age", value: 20));
63 doc.add(new SortedDocValuesField( name: "label", new BytesRef( text: "c")));
64 indexWriter.addDocument(doc);
65 // 文档2
66 doc = new Document();
67 doc.add(new Field( name: "author", value: "author2", type));
68 doc.add(new NumericDocValuesField( name: "age", value: 20));
69 doc.add(new SortedDocValuesField( name: "label", new BytesRef( text: "b")));
70 indexWriter.addDocument(doc);
71 // 文档3
72 doc = new Document();
73 doc.add(new Field( name: "author", value: "author3", type));
74 doc.add(new NumericDocValuesField( name: "age", value: 60));
75 doc.add(new SortedDocValuesField( name: "label", new BytesRef( text: "a")));
76 indexWriter.addDocument(doc);
77 // 文档4
78 doc = new Document();
79 doc.add(new Field( name: "author", value: "author4", type));
80 indexWriter.addDocument(doc);
81 indexWriter.commit();
82 // 文档5
83 doc = new Document();
84 doc.add(new Field( name: "author", value: "author5", type));
85 doc.add(new NumericDocValuesField( name: "age", value: 60));
86 doc.add(new SortedDocValuesField( name: "label", new BytesRef( text: "a")));
87 indexWriter.addDocument(doc);

```

图2中描述了IndexWriter的段内排序规则，定义了两个排序规则：

- sortField1：根据域名为"age"的域值进行从小到大排序
- sortField2：根据域名为"label"的域值按照字典序排序

图2中的第45行代码，定义SortField[]数组时，sortField1相比sortField2有更小的数组下标，故总的排序规则为先按照sortField1进行排序，如果无法通过sortField1比较出两篇文档的先后关系，那么再使用sortField2来区分，如果两个排序规则都无法区分文档的先后关系，那么根据文档被添加的先后顺序来判断，即图3中文档0、文档1的编号。

图4:

```

87     DirectoryReader reader = DirectoryReader.open(indexWriter);
88     int docId = 0;
89     while (docId < reader.maxDoc()){
90         System.out.println("docId: "+docId+", fieldName: author, fieldValue: "+reader.document(docId).get("author")+""");
91         docId++;
92     }

```

图5:

```

docId: 0, 域名: author, 域值: author4
docId: 1, 域名: author, 域值: author0
docId: 2, 域名: author, 域值: author2
docId: 3, 域名: author, 域值: author1
docId: 4, 域名: author, 域值: author3
docId: 5, 域名: author, 域值: author5

```

图3中的文档都被添加到索引之后会生成一个段，我们通过图4的代码来打印该段的文档信息，即图5的内容，可以看出当使用了IndexSort后，段内的文档按照图2中的排序规则正确的排序了，通过图5我们可以知道下面的内容：

- docId为0的文档，它对应图3中的文档4，由于这篇文档没有"age"、"lable"的DocValues域，故它被认为是"最小的"，即排在最前面
- docId为3、docId为4的文档，它们分别对应图3中的文档1跟文档3，可以看出它们是先按照sortField1而不是sortField2进行排序，否则文档3根据域名"lable"的域值"a"，它应该比文档1"较小"
- docId为2，docId为3的文档，它们分别对应图3中的文档2跟文档1，可以看出这两篇文档根据sortField1无法区分大小关系，再根据sortField2能比较出文档2"较小"
- docId为4，docId为5的文档，它们分别对应图3中的文档3，文档5，可以看出这两篇文档根据sortField1跟sortField2都无法区分大小关系，所以只能根据文档被添加的先后顺序来判断，文档3先被添加，所以它"较小"

如果我们不设置IndexSort，图4的打印结果如下所示：

图6:

```

docId: 0, 域名: author, 域值: author0
docId: 1, 域名: author, 域值: author1
docId: 2, 域名: author, 域值: author2
docId: 3, 域名: author, 域值: author3
docId: 4, 域名: author, 域值: author4
docId: 5, 域名: author, 域值: author5

```

从图6中可以看出，如果不设置IndexSort，那么段内的文档顺序就是文档被添加的顺序。

另外在设置了IndexSort后，[Collector](#)的collect(int doc)方法依次收到的文档号也是排序的，故如果业务中对查询性能有较高要求，并且返回的结果有固定的排序规则的要求，那么我们可以将这个排序规则通过IndexSort来实现，将排序的开销扔到索引阶段。

上文的例子demo可以点这里查看：<https://github.com/LuXugang/Lucene-7.5.0/blob/master/LuceneDemo/src/main/java/lucene/docValues/SegmentInnerSort.java>。

在索引阶段通过IndexSort进行段内排序，即对段内的文档进行排序，实际上不是真正的改变文档在段中的位置，因为在性能上来讲是不可能的，段内排序的实质是生成一个映射，下文中将详细的介绍段内排序的过程。

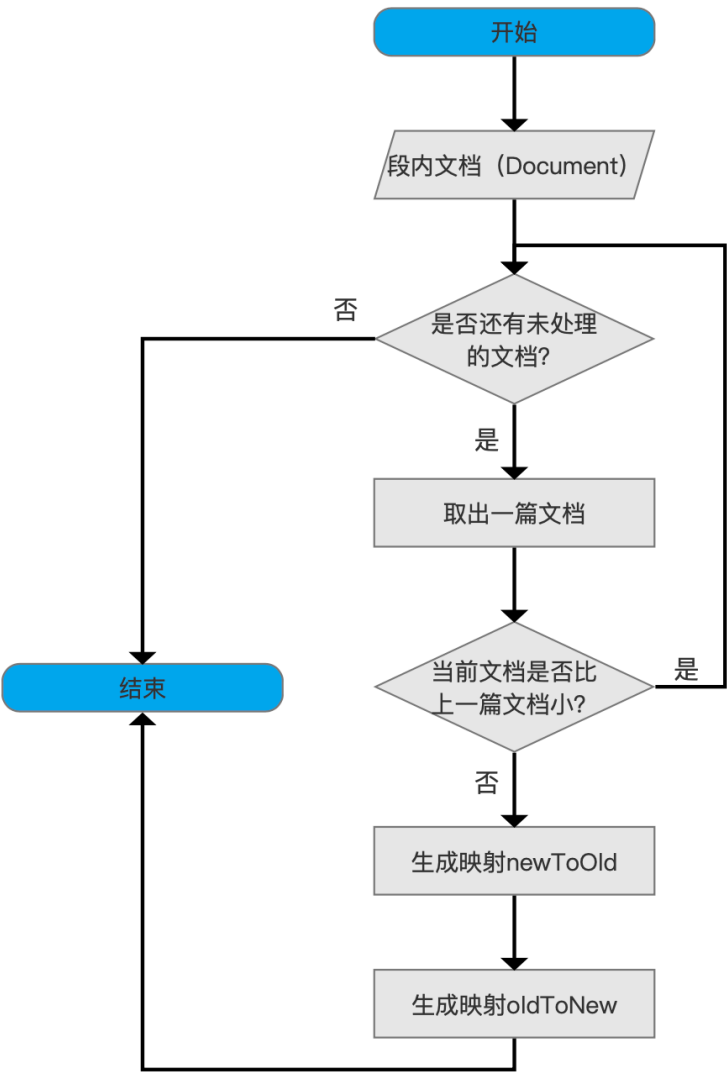
在索引阶段，什么时候进行段内排序：

- flush生成一个段的过程中进行段内排序，具体的见文章[文档提交之flush \(三\)](#)中流程点 将DWPT中收集的索引信息生成一个段newSegment的流程图 的介绍。

段内排序流程图

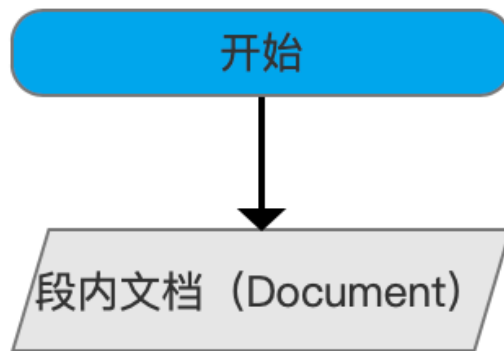
为了便于描述，图7中的流程图是按照从小到大的顺序进行排序。

图7：



段内文档

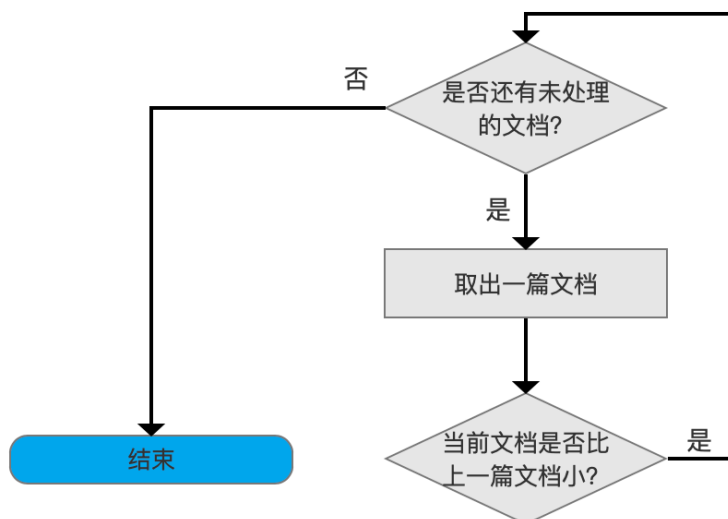
图8：



待排序的对象是段内所有的文档。

判断段内文档是否已经有序

图9:



从段内的第二篇文档（因为要跟上一篇文档进行比较）开始，根据IndexWriter提供的段内排序规则比较当前文档是否小于上一篇文档，如果所有的文档都满足这个关系，说明段内已经有序，那么直接退出即可，否则只要有某任意两篇文档不满足上述关系，说明段内不是按照IndexWriter提供的段内排序规则有序的，故需要进行排序。

生成映射newToOld

图10:

生成映射newToOld

如果待排序的文档没有按照IndexWriter提供的段内排序规则有序，那么需要进行排序，并且这里使用TimSort进行排序，本篇文章中不会对介绍TimSort的逻辑，我们只关心排序结束后，会生成一个映射newToOld，当然在源码中它是一个经过[压缩处理](#)的对象，为了便于介绍，我们可以简单的理解为newToOld是一个数组，对于图3中的例子，它对应的newToOld数组如下所示：

图11：

newToOld数组						
数组元素：文档编号	文档4	文档0	文档2	文档1	文档3	文档5
数组下标：docId	0	1	2	3	4	5

new指的是数组下标值，old指的是数组元素，newToOld数组实现了new到old的映射，所以段内排序并没有真正的去"移动"文档。如果图3中的文档都满足搜索条件，那么[Collector](#)的collect(int doc)方法依次收到的文档号即图11中的docId。

另外，如果阅读过[文档的增删改](#)的系列文章，图11中的数组元素，即文档编号，它是根据文档被添加到DWPT（见[文档的增删改（中）](#)）中的顺序赋值的，即文章[文档的增删改（下）（part 2）](#)中的numDocsInRAM。

生成映射oldToNew

图12：

生成映射oldToNew

根据映射newToOld生成一个相反的映射，如下所示：

图13：

oldToNew数组						
数组元素：docId	1	3	2	4	0	5
数组下标：文档编号	0（文档0）	1（文档1）	2（文档2）	3（文档3）	4（文档4）	5（文档5）

任务二：对合并后的新段进行段内排序

该任务的逻辑相对复杂，基于篇幅，在下一篇文档中展开介绍。

结语

无

[点击](#)下载附件