

文档提交之flush (三)

本文承接[文档提交之flush \(二\)](#)，继续依次介绍每一个流程点，下面先给出在前面的文章中我们列出的流程图：

图1：

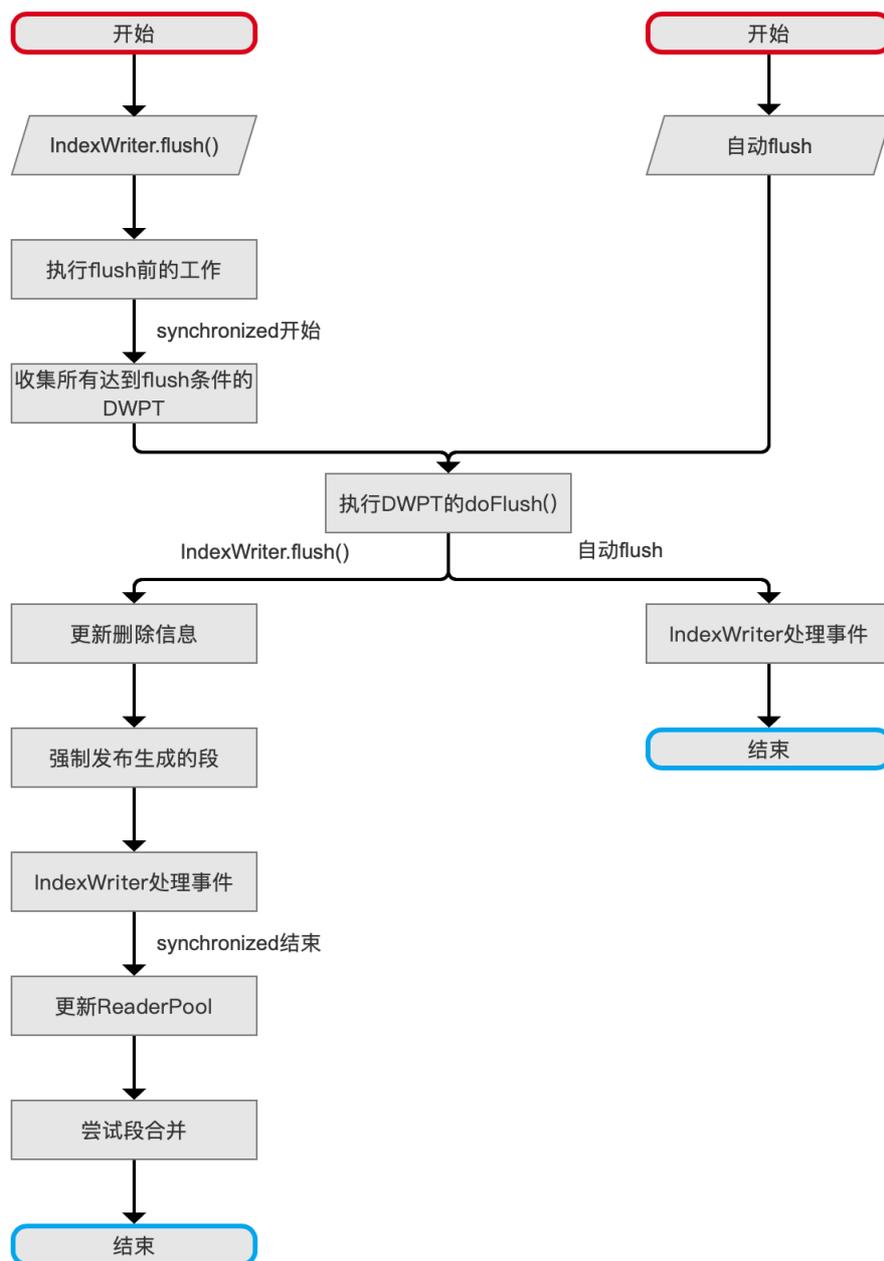


图1是文档提交之flush的流程图。

图2：

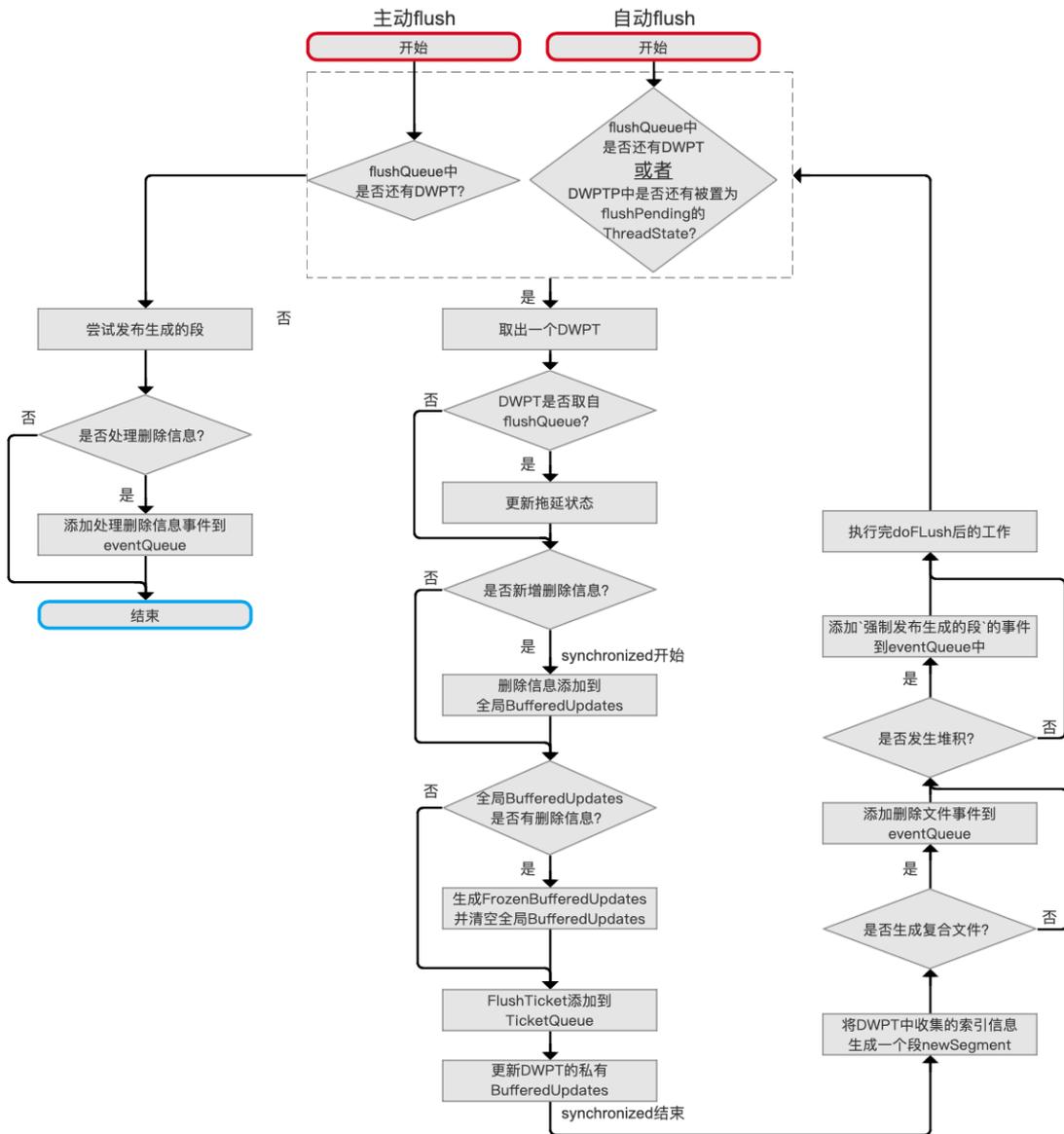
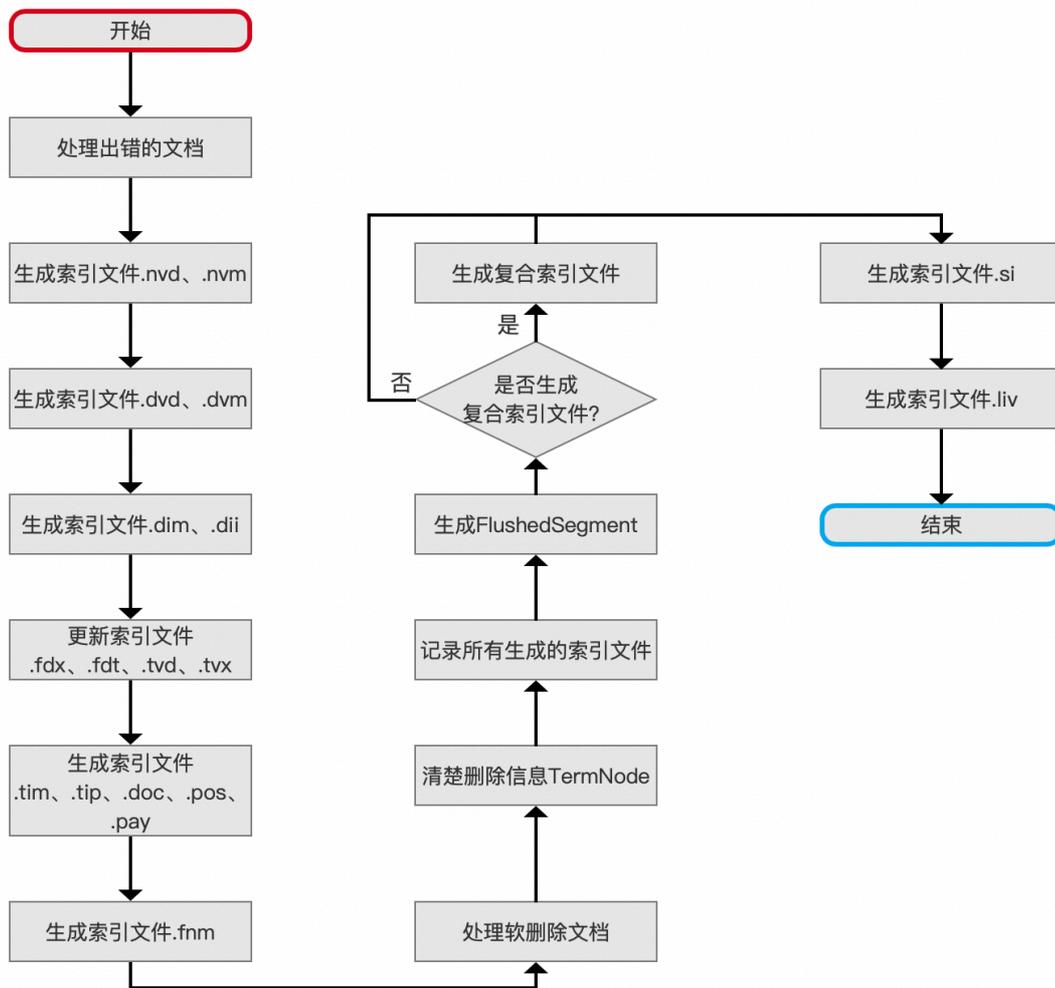


图2是图1中执行DWPT的doFlush()的流程图 流程点的流程图，在[文档提交之flush \(二\)](#)中我们已经介绍了 将DWPT中收集的索引信息生成一个段newSegment 之前的流程点。

[点击查看大图](#)

将DWPT中收集的索引信息生成一个段 newSegment的流程图

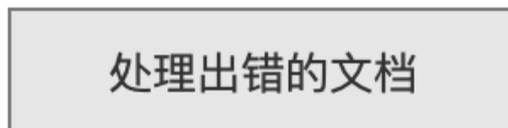
图3:



在上面流程图中，可以看出在流程点 将DWPT中收集的索引信息生成一个段newSegment 实现了所有索引文件的生成，DWPT中包含了生成这些索引文件需要的信息。

处理出错的文档

图4:



在[文档的增删改（下）（part 2）](#)中我们了解到，删除信息根据不同的删除方式会被记录到BufferedUpdates（全局BufferedUpdates或者DWPT的私有BufferedUpdates）不同的容器中：

- Map<Term,Integer> deleteTerms
- Map<Query,Integer> deleteQueries

- `Map<String,LinkedHashMap<Term,NumericDocValuesUpdate>>` numericUpdates
- `Map<String,LinkedHashMap<Term,BinaryDocValuesUpdate>>` binaryUpdate

而在[两阶段生成索引文件之第一阶段](#)的文章中我们介绍了添加一篇文档的过程，即DWPT收集该文档的索引信息的过程，如果在这个过程中发生任何错误，那么该文档的文档号会记录到DWPT的私有BufferedUpdates（见[文档提交之flush \(二\)](#)）中，即存放到下面的容器中：

- `List<Integer> deleteDocIDs = new ArrayList<>();`

执行到该流程点，将deleteDocIDs中所有的文档号写入到FixedBitSet对象中，该对象描述了那些被删除的文档号，在后面的流程中，FixedBitSet中的文档信息会被写入到索引文件.liv中。

生成索引文件

图5：

生成索引文件.nvd、.nvm



生成索引文件.dvd、.dvm



生成索引文件.dim、.dii



更新索引文件
.fdx、.fdt、.tvd、.tvx



生成索引文件
.tim、.tip、.doc、.pos、
.pay



生成索引文件.fnm

这里只简单介绍下生成各个索引文件的先后顺序，其生成过程没啥好写的，只要熟悉每个[索引文件](#)的数据结构就行啦。

不过生成索引文件的过程中，有两个知识点还是要说明下的，一个是 `Sorter.DocMap` 对象，另一个是 `找出部分删除文档的文档号`。

Sorter.DocMap sortMap

`DocMap`是类`Sorter`的内部类，而`sortMap`则是在源码中`Sorter.DocMap`类的一个对象名。

当我们在生成`IndexWriter`对象时，可以通过`IndexWriterConfig.setIndexSort(Sort)`的方法来定义一个排序规则，在生成索引文件的过程中，使得一个段内的所有索引文件中的文档根据该规则进行排序，当然并不是真正的排序，而是生成一个映射关系`sortMap`（见[Collector \(三\)](#)中的预备知识），`sortMap`描述了文档之间的顺序。至于为什么要对文档排序，`sortMap`如何实现映射，并不是本篇文章关心的，在后面的文章中会介绍。

找出部分删除文档的文档号

在上面的内容中我们知道，`BufferedUpdates`的多个容器中存放各种删除信息，其中 `Map<Term,Integer> deleteTerms`中存放了根据`Term`进行删除的删除信息，根据该删除信息，在生成索引文件 `.tim`、`.tip`、`.doc`、`.pos`、`.pay`的过程中会找到那些满足删除要求的文档号，随后将这些文档号添加到 `FixedBitSet`（上文介绍了该对象的用途）对象中。

至于查找过程，这块的内容会跟后面介绍文档查询的文章重复，故这里先不做介绍。

为什么只找出部分删除文档的文档号，而不是根据`BufferedUpdates`中所有容器的删除信息找到所有满足删除要求的文档：

- 原因一：先说为什么不根据 `Map<Query,Integer> deleteQueries` 容器找出满足删除要求的文档号，由于这是通过一个查询删除（跟在查询阶段，生成一个`Query`进行查询是一个操作），即查询一个段中文档的操作，而此时该段还没有完全生成结束，故无法实现该操作。至于为什么不根据 `numericUpdates`、`binaryUpdate` 容器找出满足删除要求的文档号，这块的话在后面介绍软删除的文章中会介绍。
- 原因二：在自动`flush`的操作中，允许并发的执行多个`DWPT`生成段，在当前阶段可以并发的找出被删除的文档号，如果不在此时执行 `找出部分删除文档的文档号` 的操作，尽管在后面的流程中也会执行，但是在那个过程中，是串行执行，所以提高了生成段的性能。至于为什么要串行执行，在下面的流程中会说明。

处理软删除文档

图6：

处理软删除文档

在上一个流程中，我们知道，有些文档被标记为删除了，而这些文档有可能是软删除的文档，那么软删除文档的个数需要被更新。

清楚删除信息TermNode

图7:

清楚删除信息TermNode

在前面的流程中，根据Term进行删除的删除信息已经作用（apply）到了当前段，所以需要清除TermNode的信息，以免在后面的流程中重复执行，TermNode描述的删除信息即容器Map<Term,Integer> deleteTerms中的删除信息。

记录所有生成的索引文件

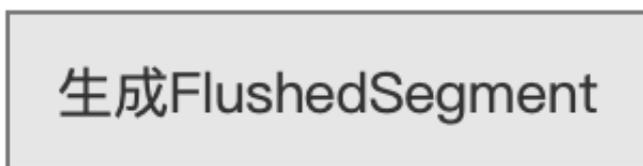
图8:

记录所有生成的索引文件

记录当前已经生成的索引文件的文件名，因为如果通过 [IndexWriterConfig.setUseCompoundFile\(boolean\)](#) 设置了使用复合索引文件 `cfs&&cfe` 存储文档的索引信息，那么在后面的流程中，在生成完复合索引文件后，需要删除这些索引文件，所以在 [索引文件之 `cfs&&cfe`](#) 的文章的开头部分，我们提到了当使用了复合索引文件后，索引目录中最终保留 `.cfs`、`.cfe`、`.liv`、`.si` 的索引文件（执行 `commit()` 操作之前）。

生成FlushedSegment

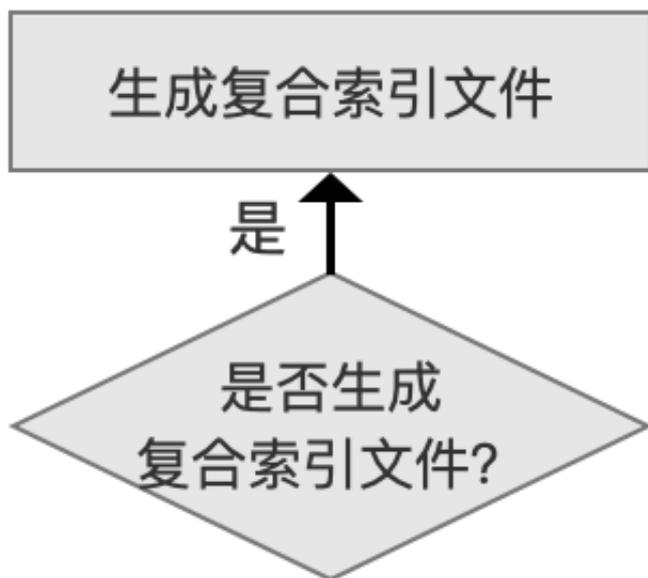
图9:



这里只是为了介绍生成 `FlushedSegment` 对象的时机，至于 `FlushedSegment` 是干嘛的，在本篇文章中并不重要，在后面的文章中会详细介绍。

生成复合索引文件

图10:



如果通过[IndexWriterConfig.setUseCompoundFile\(boolean\)](#)设置了使用复合索引文件`cfs`与`cfe`存储文档的索引信息，那么在当前流程点会正式的生成复合索引文件`.cfs`、`.cfe`，注意的是此时非复合索引文件还没有被删除，在后面的流程中才会被删除，在后面的文章中会介绍为什么不在这个流程点删除。

生成索引文件.si

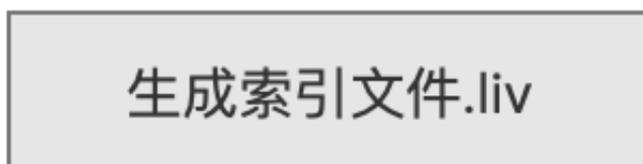
图11:



生成索引文件.si的过程不介绍了，没什么好讲的，了解[.si](#)文件的数据结构就行啦。

生成索引文件.liv

图12:



生成索引文件.liv的过程不介绍了，没什么好讲的，了解[.liv](#)文件的数据结构就行啦。

结语

本篇文章主要介绍了每一种[索引文件](#)的生成顺序，强调的是，想要理解Lucene如何实现查询的原理，那么必须了解所有索引文件的数据结构，当然在以后的文章中会介绍其查询原理。

[点击](#)下载附件