

# 文档提交之flush (四)

本文承接[文档提交之flush \(三\)](#)，继续依次介绍每一个流程点。

先给出文档提交之flush的整体流程图：

图1：

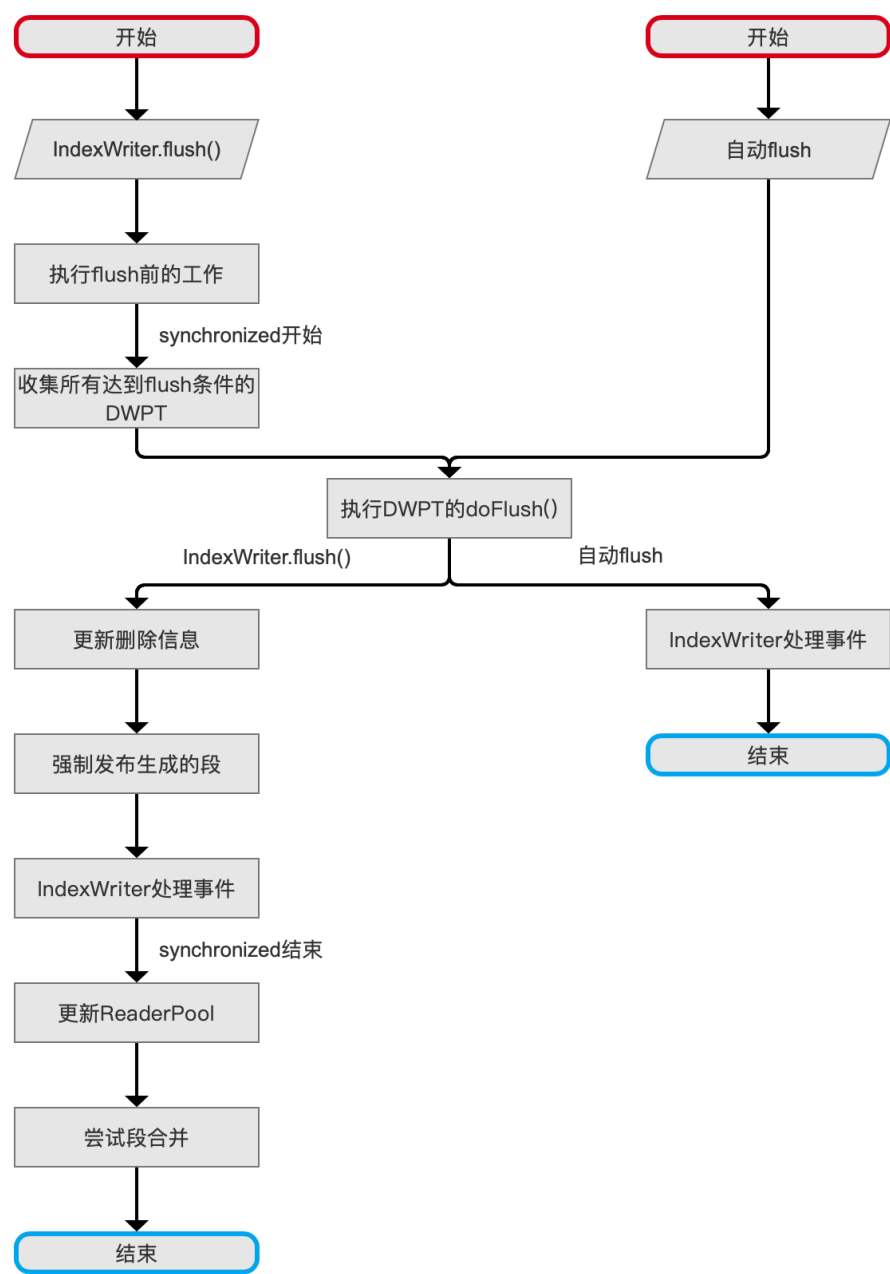
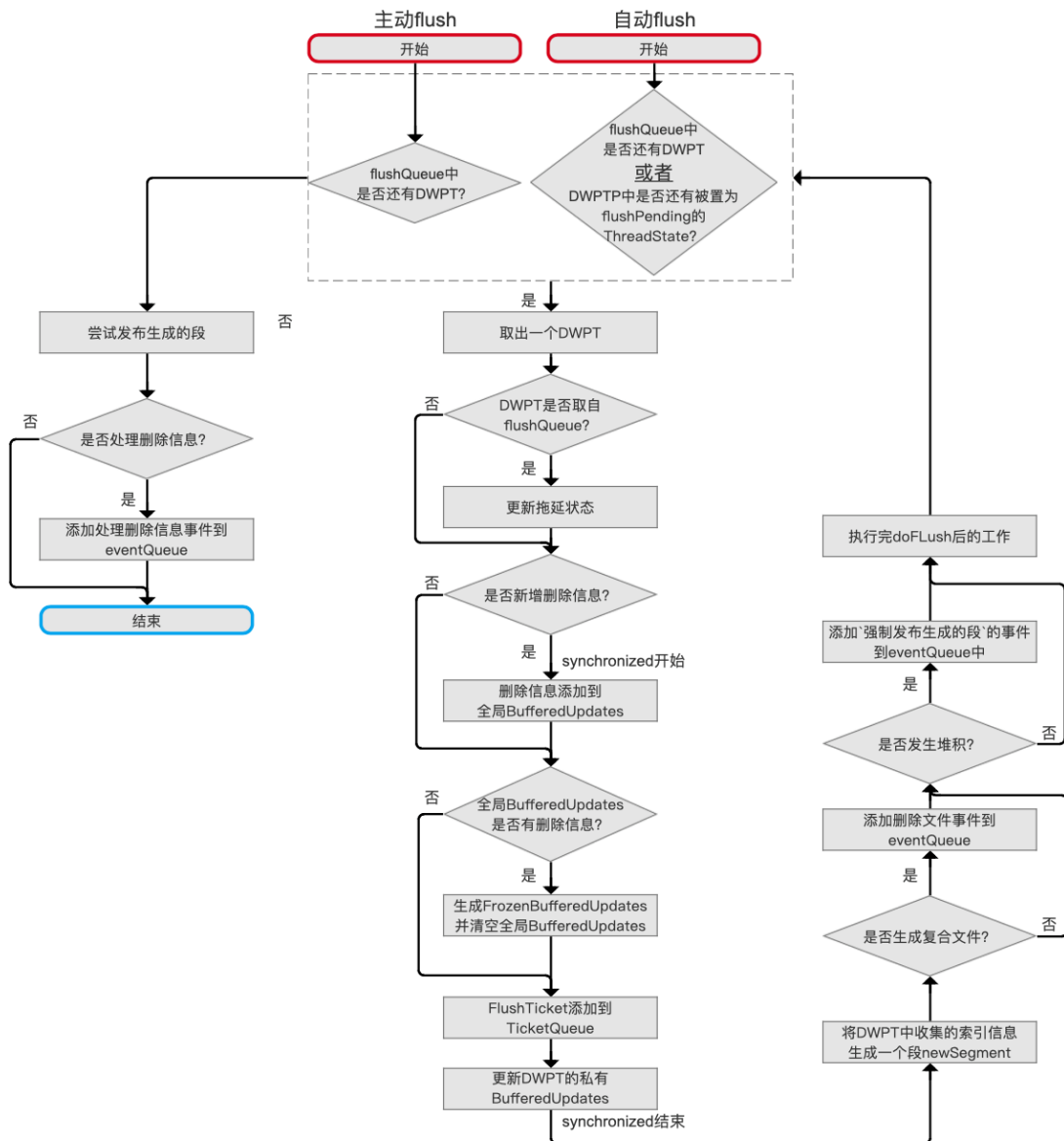


图2是文档提交之flush中的 执行DWPT的doFlush() 的流程图，在前面的文章中，我们介绍到了此流程图中 将DWPT中收集的索引信息生成一个段newSegment 的流程点，在本篇文章中会将 执行DWPT的 doFlush() 中剩余的流程点介绍完毕：

图2：



## 预备知识

### FlushTicket

在[文档提交之flush \(二\)](#)中，我们简单的介绍了FlushTicket类，类中包含的主要变量如下：

```
static final class FlushTicket {
    private final FrozenBufferedUpdates frozenUpdates;
    private FlushedSegment segment;
    ... ..
}
```

frozenUpdates是一个包含删除信息且作用于其他段中的文档的全局FrozenBufferedUpdate对象（见[文档提交之flush \(二\)](#)），而segment则是图2中的流程点 将DWPT中收集的索引信息生成一个段newSegment 执行结束后生成的FlushedSegment对象，它至少包含了一个DWPT处理的文档对应的索引信息（SegmentCommitInfo）、段中被删除的文档信息（FixedBitSet对象）、未处理的删除信息

FrozenBufferedUpdates（见[文档提交之flush（三）](#)）、Sorter.DocMap对象，以上内容在[文档提交之flush（三）](#)的文章中已介绍。

为什么FlushTicket中生成FrozenBufferedUpdates跟FlushedSegment是两个有先后关系的流程

- 如果FrozenBufferedUpdates未能正确生成，那么FlushedSegment也不会生成，即全局的删除信息跟DWPT收集的文档就无法生成索引文件，但是如果FrozenBufferedUpdates正确生成（只有第一个DWPT会生成不为null的FrozenBufferedUpdates，见[文档提交之flush（二）](#)），而FlushedSegment没有生成，那删除信息还能正确的作用（apply）到索引目录中的所有段，即只丢失添加的文档的信息，不会丢失删除信息

如何处理DWPT未能正确的生成一个FlushedSegment对象的情况：

- 在[文档提交之flush（三）](#)中我们了解到，如果DWPT在收集文档索引信息阶段（见[两阶段生成索引文件之第一阶段](#)），那么出错的文档的文档号会被标记在索引文件`.liv`中，而如果DWPT因某种原因（在介绍IndexWriter的异常处理时会展开）导致没有完成图2中的将DWPT中收集的索引信息生成一个段`newSegment`流程点，那么需要删除该DWPT中的对应的所有索引数据（如果已经生成的索引文件的话）

## Queue<FlushTicket> queue

该队列用来存放FlushTicket对象，每一个DWPT执行doFlush后，都会生成一个FlushTicket对象，并同步的添加到Queue<FlushTicket> queue中。

## Queue<Event> eventQueue

eventQueue队列用来存放事件（Event），Event类是一个添加了@FunctionalInterface注解的类，每一个Event对象用来描述一个函数调用，通过函数调用实现一个事件的执行。

在多线程下，每个线程同步的从eventQueue队列中取出一个事件，即执行该事件对应的函数调用。

在eventQueue队列中，事件间（Between Event）执行结束的先后顺序是无法保证的，不过可以根据事件内（Inner Event）的同步机制实现某些事件间的同步。

什么时候会添加事件到eventQueue：

- 在[文档提交之flush（三）](#)中我们提到，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，那么在生成完复合索引文件后，需要删除那些[非复合索引文件](#)，而删除操作就会作为一个事件会添加到eventQueue中
- 上文中提到DWPT可能无法正确的生成一个FlushedSegment对象，那么需要删除该DWPT中的对应的所有索引数据（如果已经生成的索引文件的话），而删除操作就会作为一个事件会添加到eventQueue中
- 从Queue<FlushTicket> queue中取出每一个FlushTicket去执行某个操作，该操作也会作为一个事件添加到eventQueue中，该操作会在后面的流程中介绍
- 还有一些其他的需要添加到eventQueue会在后面的流程中提及

为什么要将事件添加到eventQueue中处理：

- 在后面的介绍中我们将会了解到，文档提交之flush的整个流程中部分逻辑（包括异常）是可以并行执行的，如果将逻辑拆分成多个事件，那么可以充分利用多线程来并行的执行这些事件，**其中一个作用可以使得主动flush的操作能优先、尽快完成**，因为在[文档提交之flush（一）](#)中我们知道，如果flush的速度慢于添加/更新文档的操作，那么会阻塞添加/更新文档的操作。比如说在主动flush期间，如果此时其他线程触发了自动flush时，那么该线程执行完自动flush后，会去执行eventQueue中的事件（自动flush的有些操作也会作为事件添加到eventQueue中），基于队列FIFO的性质，如果此时队列中还有主动flush时添加的事件，那么就可以"帮助"主动flush先尽快完成
- 如果事件之间（Between Event）需要同步，可以通过事件内部（Inner Event）的同步机制来实现，比如说当一个类中的两个方法作为两个事件时，可以通过对象锁synchronized实现同步，不同类中的两个方法可以使用ReentrantLock可重入锁实现同步。在上文中**删除非复合索引文件 跟 从Queue<FlushTicket> queue中取出每一个FlushTicket去执行某个操作**就属于可以并行的两个事件（其实两个方法中的部分代码块还是需要同步的）

什么时候会执行eventQueue中的事件：

- 执行了添加事件到eventQueue中的操作的线程最终都会执行eventQueue中的事件，并且直到eventQueue中不存在事件才会退出

## 发布（publish）生成的段

发布生成的段的过程描述的是依次从Queue<FlushTicket> queue中取出FlushTicket，将其包含全局删除信息的FrozenBufferedUpdates对象作用到当前索引目录中已有的段的过程，同时还是对FlushedSegment对象进行最终处理的过程，比如找出未处理的删除信息（在[文档提交之flush（三）](#)中我们只找出了部分删除的文档）等一些操作，这里先简单的提一下，因为发布生成的段的逻辑篇幅较长，会在下一篇文章中展开介绍。

发布生成的段 还可以划分成两种类型，即 **强制发布生成的段** 和 **尝试发布生成的段**，图1跟图2中均有该流程点：

- 在多线程下，多个线程可能同时执行 **发布生成的段** 的逻辑，如果线程调用的是 **尝试发布生成的段**，那么当发现有其他线程正在执行 **发布生成的段** 的操作，当前线程就不等待，继续执行后面的流程，否则等待其他线程执行结束，即等待Queue<FlushTicket> queue中的所有FlushTicket都被处理结束

源码中是如何实现的：

- 通过tryLock()跟lock()分别实现 **尝试发布生成的段** 跟 **强制发布生成的段**

为什么要划分 **强制发布生成的段** 和 **尝试发布生成的段**：

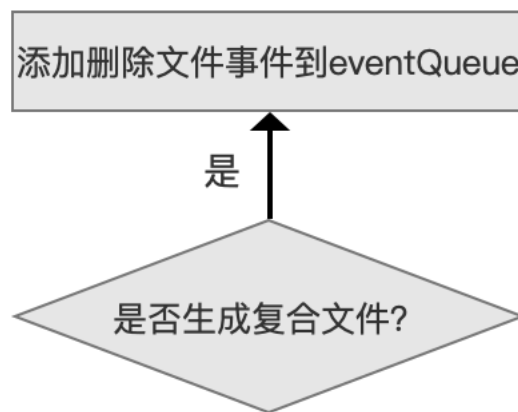
- 为了缓解堆积（backlog）问题。下文中会具体介绍

## 执行DWPT的doFlush()

继续介绍执行DWPT的doFlush()中的剩余流程点。

## 添加删除文件事件到eventQueue

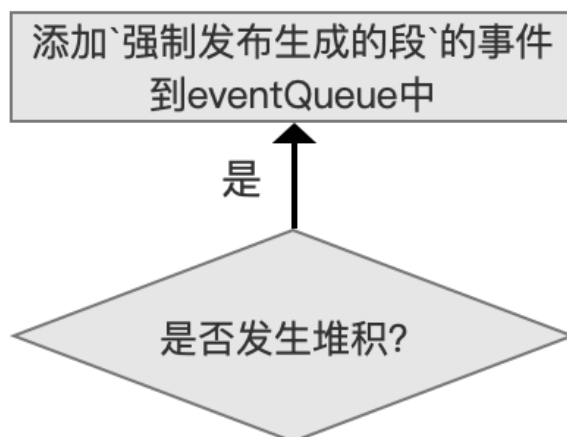
图3：



该流程即将删除非复合索引文件的操作作为一个事件添到eventQueue中，如果此时有其他线程正在处理eventQueue中的事件，那么删除非复合索引文件的操作可能会被马上执行。

## 堆积（backlog）处理

图4:



发生堆积情况即flush（主动或者自动flush）的速度慢于添加/更新文档的操作，判断是否堆积的条件如下：

```
ticketQueue.getTicketCount() >= perThreadPool.getActiveThreadStateCount()
```

其中`ticketQueue.getTicketCount()`描述的是`Queue<FlushTicket>` queue中`FlushTicket`的个数，该个数即当前等待flush为一个段的段的个数（可能包含出错的`FlushTicket`，见下文介绍），而`perThreadPool.getActiveThreadStateCount()`描述的是线程池DWPTP中`ThreadState`的个数（见[文档的增删改（中）](#)）

为什么通过上面的方式能判断是否发生堆积：

- 上文中我们知道了`FlushTicket`的个数即DWPT的个数，又因为在[文档的增删改（中）](#)中，我们了解

到，一个ThreadState中持有一个DWPT的引用之后，才去执行文档的添加/更新操作，当DWPT中收集的索引量满足自动flush的条件（见[文档的增删改（下）（part 3）](#)）后，DWPT进入图1中自动flush的流程点，开始生成一个段，最后释放DWPT对象，而ThreadState对象则是回到DWPTP中，故在单线程下，DWPT的个数总是小于等于DWPTP中ThreadState的个数。在多线程下，ThreadState对象回到DWPTP之后，又有新的线程执行添加/更新的操作，那么ThreadState会再次持有新的DWPT对象去执行任务，如果再次出发自动flush，当flush的速度（即DWPT生成一个段）较慢时，就会满足ticketQueue.getTicketCount() >= perThreadPool.getActiveThreadStateCount()的条件，即发生了堆积

- 上一条中，DWPT满足自动flush后进入生成一个段与ThreadState回到DWPTP体现了flush跟添加/更新文件是并行操作，另外主动flush的情况也是一样的，见[文档提交之flush（一）](#)

上文中为什么说Queue<FlushTicket> queue中可能包含出错的FlushTicket：


- 在[文档提交之flush（二）](#)中我们知道，FlushTicket在成功生成了FrozenBufferedUpdates对象frozenUpdates之后，FlushTicket就添加到了Queue<FlushTicket> queue中，此时的FlushTicket中还没有生成FlushedSegment对象segment，如果在后续流程中未能正确生成一个FlushedSegment对象，那么FlushTicket被认为是未能正确生成的，即出错的FlushTicket，当然了如果未能生成FrozenBufferedUpdates对象，FlushTicket就不会被添加到Queue<FlushTicket> queue中

## 添加强制发布生成的段的事件到eventQueue中

发生堆积后，通过该流程点使得所有执行flush的线程必须等待Queue<FlushTicket> queue中所有的FlushTicket处理结束后才能去执行新的添加/更新文档的任务来处理堆积问题。

## 执行完doFlush后的工作

图5：



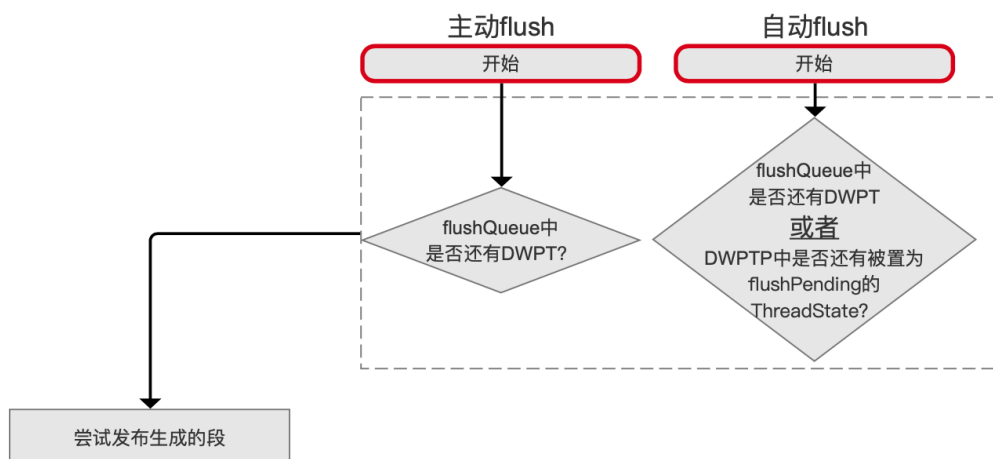
执行完doFlush后的工作

到此流程点，我们需要更新几个全局的信息，以下的内容在前面的文章中已经介绍，不详细展开：

- flushingWriters：见[文档的增删改（下）（part 3）](#)
- flushBytes：见[文档的增删改（下）（part 1）](#)
- 执行updateStallState()方法：见[文档提交之flush（一）](#)

## 尝试发布生成的段

图6：

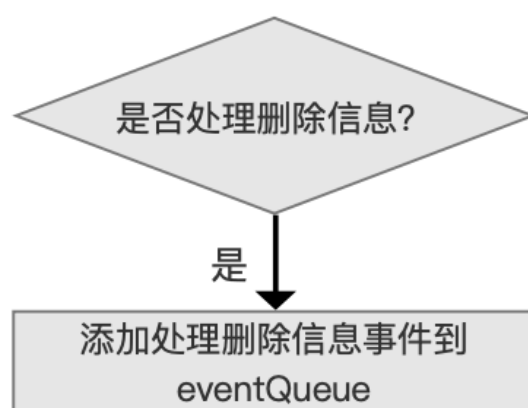


尝试发布生成的段的概念在上文关于发布（publish）生成的段的内容中已作介绍，不赘述，其详细的发布过程在下一篇文章会展开。

- 主动flush：当flushQueue中的DWPT都处理结束，就可以执行下一个流程点 尝试发布生成的段
- 自动flush：除了满足图6中的条件，另外如果出现堆积问题，那么该线程就不再取新的DWPT执行，目的很明确，即缓解堆积问题

## 处理删除信息

图7：



当前内存中的删除信息如果超过阈值的一半，那么需要处理删除信息，阈值即通过 [IndexWriterConfig setRAMBufferSizeMB](#) 设置允许缓存在内存的索引量（包括删除信息）的最大值，当超过该阈值，会触发自动flush（见[文档提交之flush（一）](#)）

为什么内存中的删除信息如果超过阈值的一半，需要处理删除信息：

- 触发自动flush的其中一个条件如下所示：

```
activeBytes + deleteBytesUsed >= ramBufferSizeMB
```

- 其中activeBytes描述的是当前内存中所有DWPT收集的索引总量，deleteBytesUsed描述的是当前内存中删除信息的总量，ramBufferSizeMB描述的是即允许缓存在内存的索引量的最大值。
- 根据公式可以看出如果不处理删除信息，那么使得触发自动flush的频率更高，这样可能会产生很多的小段（Tiny Segment），即处理删除信息的目的

如何处理删除信息：

- 如图7中所示，执行 添加处理删除信息事件到eventQueue 即可，处理删除信息事件实际是将 强制发布生成的段 作为一个事件添加到eventQueue中。

为什么通过 强制发布生成的段 能用来处理删除信息

- 使得所有执行flush的线程必须等待Queue<FlushTicket> queue中所有的FlushTicket处理结束后才能去执行新的添加/更新文档、删除的任务，等待删除信息从内存写到磁盘（.liv索引文件），同时放慢触发自动flush的速度

## 结语

---

至此我们介绍完了图1中逻辑相对最复杂的 执行DWPT的doFlush() 的流程。

[点击](#)下载附件