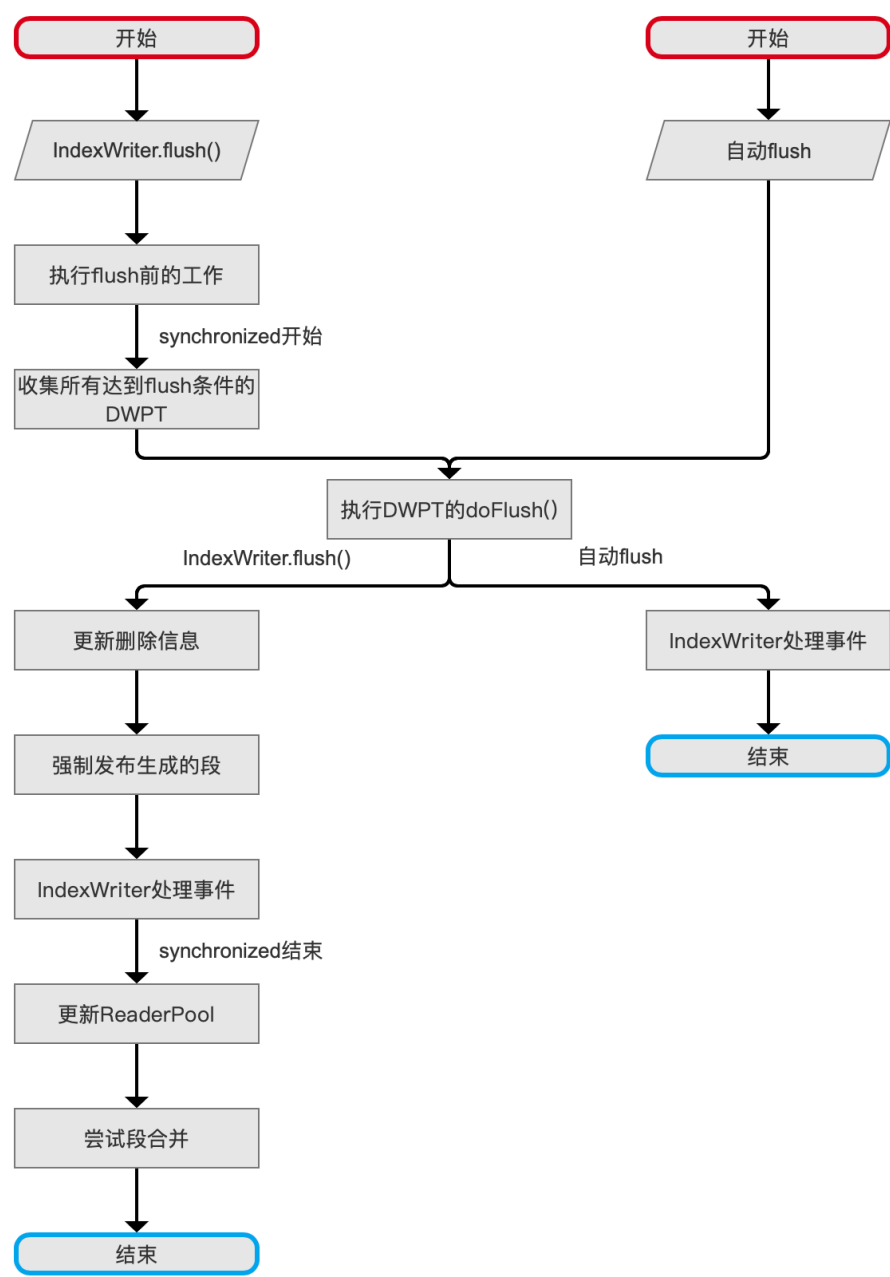


# 文档提交之flush (六)

本文承接[文档提交之flush \(五\)](#)，继续依次介绍每一个流程点。

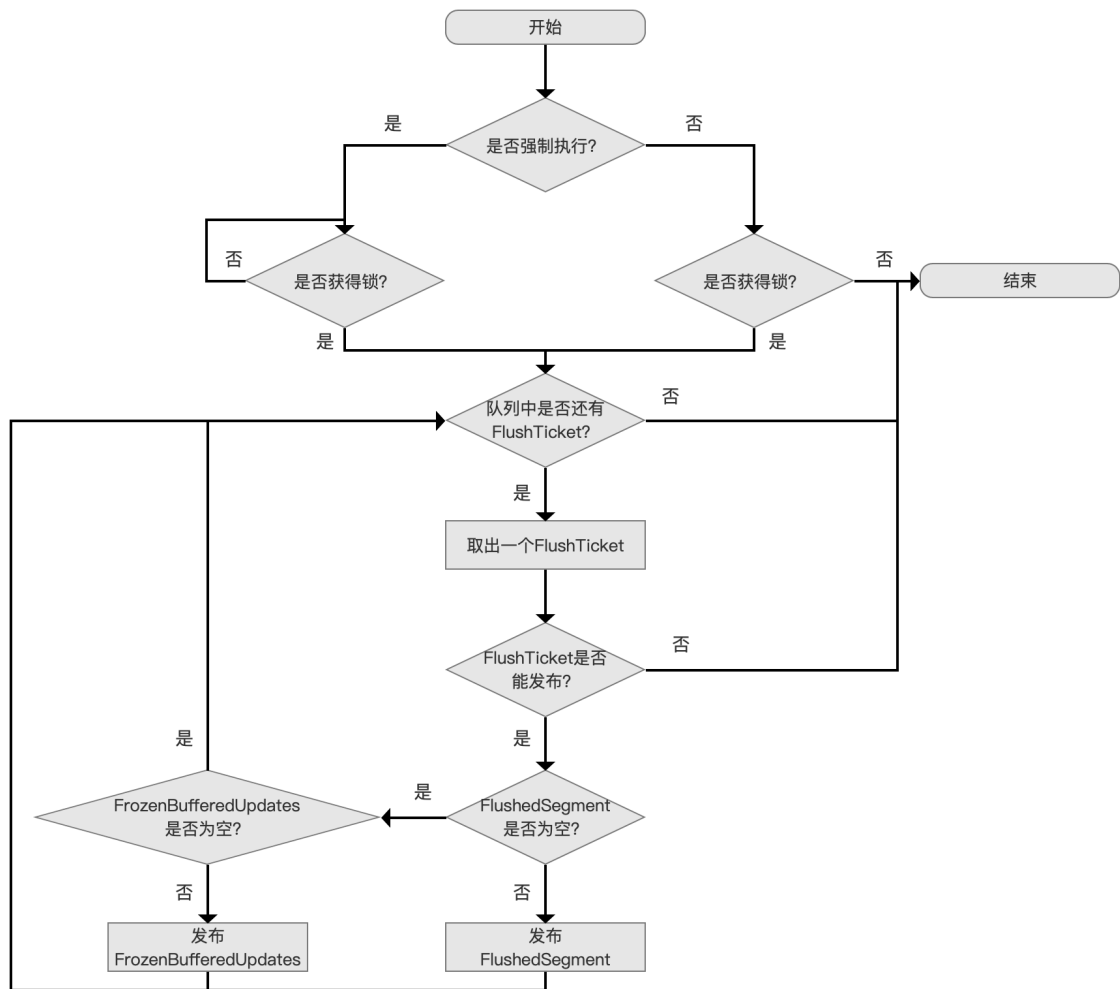
## 文档提交之flush的整体流程图

图1：



### 强制发布生成的段

图2：

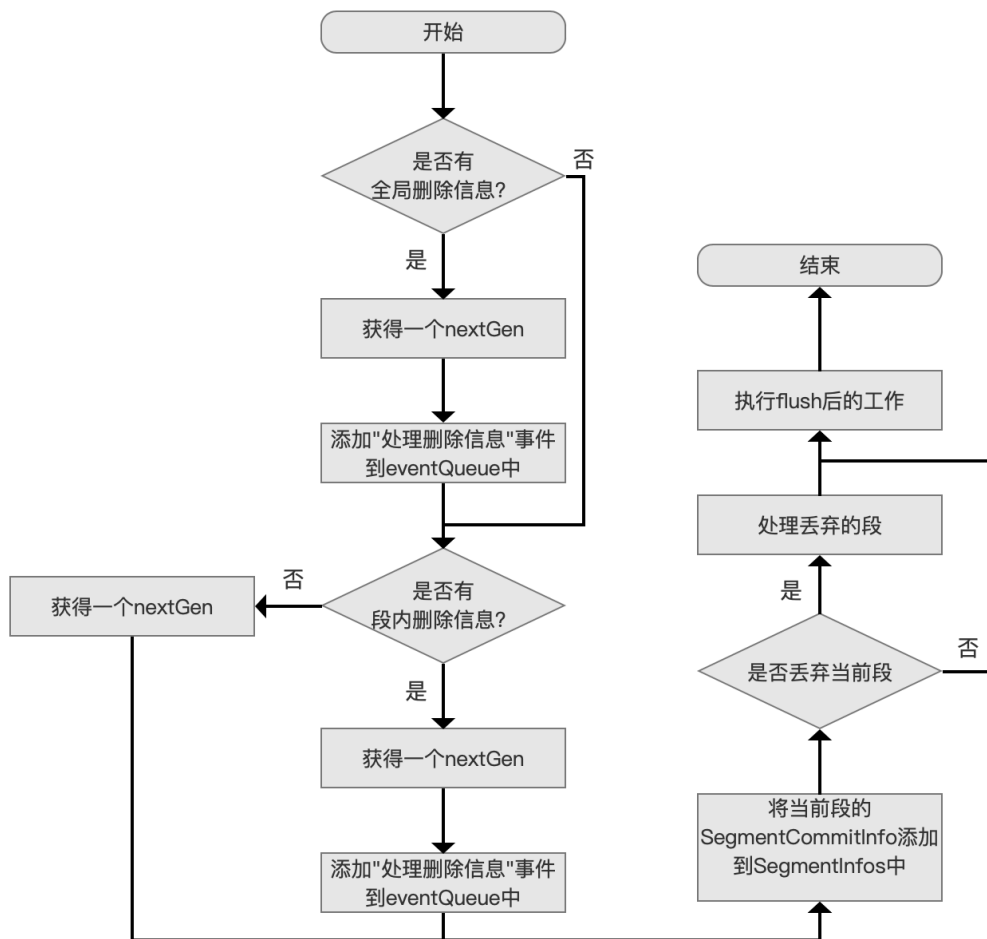


在[文档提交之flush（五）](#)中我们还剩余 发布FlushedSegment 跟 发布FlushedSegment 两个流程点未介绍。

## 发布FlushedSegment 的流程图

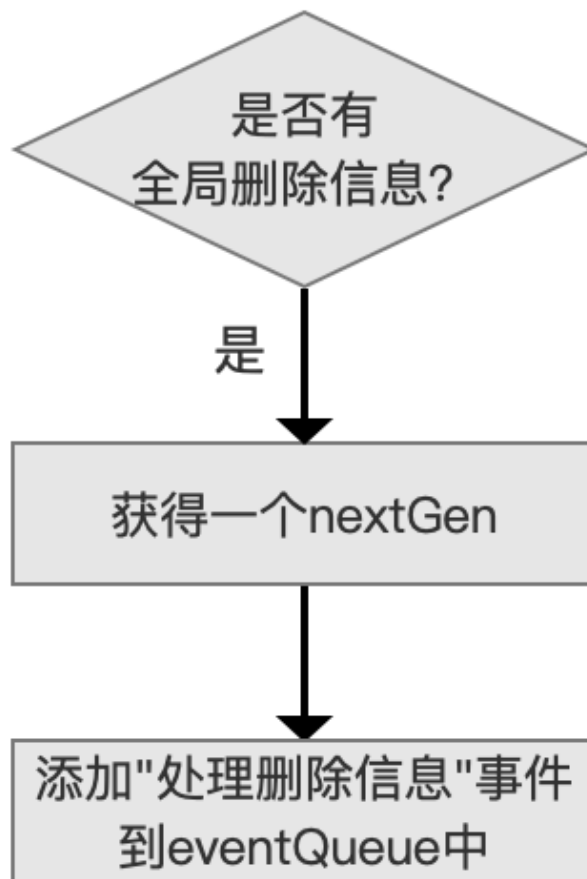
图3的流程图描述的是一个FlushTicket执行 发布FlushedSegment 流程所涉及的处理过程。

图3:



## 处理全局删除信息

图4:



首先给出FlushTicket类，类中包含的主要变量如下：

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

从前面的文章（见[文档提交之flush \(二\)](#)）中我们了解到，不是每一个FlushTicket都有全局删除信息，即FrozenBufferedUpdates可能为null。如果不为空，我们首先要获得一个nextGen。

**nextGen有什么作用：**

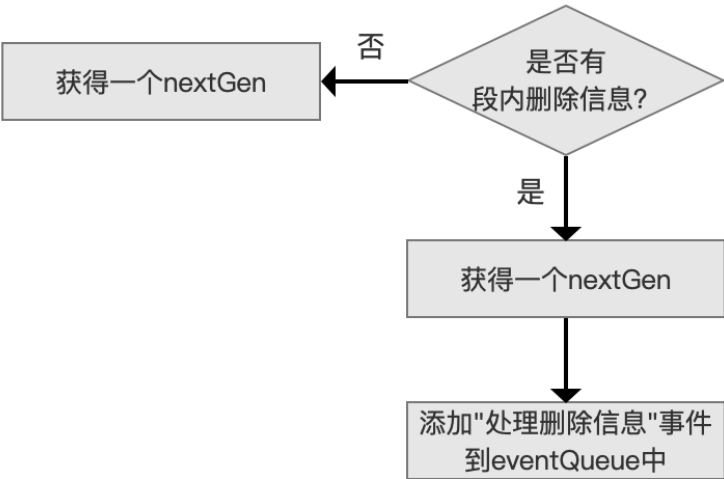
- nextGen是一个从1开始递增的值，每一个FlushTicket中的FrozenBufferedUpdates都会同步获得一个唯一的nextGen，某个FrozenBufferedUpdates中的删除信息会作用（apply）到所有比它持有的nextGen小的段
- 在[文档提交之flush \(二\)](#)中我们提到，我们生成FlushTicket后，将FlushTicket添加到

Queue<FlushTicket> queue中是一个同步的过程，这是一个使得多个删除信息能正确的获得唯一的nextGen的方法之一

由于根据nextGen能保证正确的处理删除信息，所以真正的从别的段中处理删除信息的操作，即处理删除信息，就可以作为一个事件添加到eventQueue（见[文档提交之flush（四）](#)），使得可以多线程并发执行，在下篇文章中我们将会了解到，处理删除信息的过程是开销较大的工作。

### 处理段内的删除信息

图5：



在[文档提交之flush（四）](#)中我们简单介绍了FlushTicket 中的FlushedSegment中包含的几个信息，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。在当前流程点，FlushedSegment中的FrozenBufferedUpdates（非FlushTicket中的FrozenBufferedUpdates）中包含了段内的删除信息。

另外，在[文档提交之flush（三）](#)中我们介绍了，在生成索引文件的过程中，我们只处理了部分满足删除信息（见下文 处理丢弃的段的介绍）的文档，到此流程点，我们需要处理剩余的删除信息。

无论当前FlushTicket中是否还有段内删除信息，当前段都需要获得一个nextGen，之后任何大于nextGen值的删除信息都需要作用到当前段。

### 将当前段的SegmentCommitInfo添加到SegmentInfos中

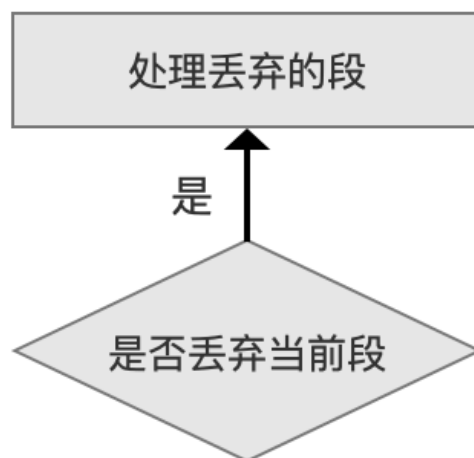
图6：



将FlushedSegment中的SegmentCommitInfo，即索引信息添加到SegmentInfos中。SegmentInfos即所有段的索引信息的集合，在commit()阶段，SegmentInfos中的信息对应生成[Segment\\_N](#)的索引文件。

## 处理丢弃的段

图7:



如果当前段中的文档的总个数maxDoc与被标记为删除的（deleted）文档的个数相同，那么该段需要被丢弃，判断条件如下：

```
delCount + softDelCount == maxDoc
```

- delCount：该值描述的是满足删除信息TermArrayNode、TermNode（见[文档的增删改（下）（part 2）](#)）的文档的个数，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，随后FixedBitSet中的文档信息在写入到索引文件.liv过程中，将被删除的文档的个数统计到SegmentCommitInfo的delCount中
- softDelCount：该值描述的是包含软删除信息的文档，软删除的具体介绍会单独开一篇文章介绍，

这里只要简单的知道，包含软删除信息的文档也是被标记被删除的（deleted）。在[文档提交之flush（三）](#)中介绍了在 处理软删除文档 的流程中，计算出了softDelCount的值，不赘述

- maxDoc：该值描述了当前段中的文档总数，即DWPT收集的文档（见[文档的增删改（中）](#)）的文档个数

从上文介绍中我们知道，当前段的SegmentCommitInfo已经被添加到SegmentInfos中，由于段的合并跟flush是异步操作，故运行到此流程点时，当前段可能正在执行段的合并。如果正在合并当前段，那么就不处理，在合并结束后，会自动丢弃该段，否则将此段对应SegmentCommitInfo从SegmentInfos中移除。

## 执行flush后的工作

图8：

### 执行flush后的工作

同图1中的 执行flush前的工作 的流程点, Lucene在此流程点预留了一个钩子函数（hook function），使用者可以实现在此实现自己的方法。

## 结束

至此，强制发布生成的段 中的 发布FlushedSegment 流程介绍结束，另外 发布FrozenBufferedUpdates 的流程逻辑即上文中的 处理全局删除信息。

## IndexWriter处理事件

在执行该流程之前，需要先执行下面的几个收尾工作，即执行源码中[DocumentsWriterFlushControl.finishFullFlush\(\)](#)的方法：

- 从blockedFlushes（见[文档的增删改（下）（part 3）](#)）中将newQueue（见[文档提交之flush（一）](#)）对应的DWPT添加到flushQueue（见[文档的增删改（下）（part 3）](#)）中，如果在主动flush期间，其他线程的添加/更新文档操作满足自动flush的要求，那么对应的DWPT会暂时被存放在blockedFlushes中，至于原因已在前面的文章中介绍，不赘述
- fullFlush（见[文档提交之flush（二）](#)）置为false：该值置为false，表示此次主动flush已经执行结束，自动flush的DWPT（跟主动flush中的DWPT具有不同的全局删除队列deleteSlice，见[文档提交之flush（二）](#)）可以开始执行图1中 执行DWPT的doFlush() 的流程，注意的是当前线程还未释放用来同步主动flush的fullFlushLock对象（见[文档提交之flush（一）](#)）
- 调用updateStallState：更新拖延状态，即调整当前索引写入的健康度，见[文档提交之flush（一）](#)中的详细介绍

上面的收尾工作结束后，接着还需要尝试处理newQueue中的删除信息。

为什么此时需要尝试处理newQueue中的删除信息：

- 在[文档提交之flush（四）](#)中其实我们在图1的 执行DWPT的doFlush() 流程中已经处理过一次newQueue中的删除信息，条件是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，目的是为了防止产生过多的小段。
- 而在此流程点，判断的条件是内存中的删除信息是否超过阈值，如果超过阈值并且此时不处理删除信息，那么其他线程的添加/更新文档的操作会被阻塞（见[文档的增删改（下）（part 1）](#)）

为什么此时能尝试处理newQueue中的删除信息：

- 主动flush对应的FrozenBufferedUpdates已经获得了nextGen，即能保证正确的作用（apply）删除信息的顺序，故处理自动flush的删除信息是没有问题的

执行完收尾工作后，当前线程从eventQueue队列中逐个取出所有的事件，即执行事件对应的函数调用，关于事件的概念见[文档提交之flush（四）](#)。

总结前几篇文章中的内容，eventQueue中包含以下几种事件类型：

- 删除文件事件：当使用[复合索引文件](#)时，我们需要删除非复合索引文件，见[文档提交之flush（四）](#)
- flush失败事件：未能正确生成FlushedSegment产生的事件，在后面的文章介绍IndexWriter的异常处理时会展开介绍
- 处理删除信息事件：当段中的删除信息超过阈值时，需要生成该事件，见[文档提交之flush（四）](#)
- 堆积（backlog）处理事件：flush（主动或者自动flush）的速度慢于添加/更新文档的操作时会发生堆积问题，那么生成一个事件来缓解堆积情况，见[文档提交之flush（四）](#)
- 发布生成的段中的处理删除信息事件：即上文中的内容

## 结语

---

在下篇文章中，将会介绍图1中剩下的几个流程点以及 发布生成的段中 生成的 处理删除信息 事件的逻辑。

[点击](#)下载附件