文档的增删改（下）（part 3）

本文承接文档的增删改（上）、文档的增删改（中）、文档的增删改（下）（part 1）、文档的增删改（下）（part 2）继续介绍文档的增删改，是文档的增删改系列的最后一篇文章，另外下文中如果出现未展开介绍的变量名，说明在先前的文章中已经给出介绍，不赘述。

预备知识

三个重要的容器对象

以下介绍的几个容器对理解 添加/更新 跟 flush是异步操作、索引占用内存大小控制起关键的作用。

fullFlushBuffer

定义如下：

```
private final List<DocumentsWriterPerThread> fullFlushBuffer = new ArrayList<>
();
```

当全局flush触发时，意味着该flush之前所有添加的文档都在该flush的作用范围内，那么需要将DWPTP中所有持有DWPT对象引用，并且DWPT的私有变量numDocsInRAM 需要大于0（numDocsInRAM 的概念在文档的增删改（下）（part 2）已介绍），即该DWPT处理过文档的所有ThreadState置为flushPending，他们所持有的DWPT随后将其收集到的索引信息各自生成一个段，在此之前DWPT先存放到fullFlushBuffer链表中。

blockedFlushes

定义如下：

```
private final Queue<BlockedFlush> blockedFlushes = new LinkedList<>();
```

在介绍blockedFlushes前，先补充一个ThreadState的知识点，ThreadState类实际是继承了可重入锁ReentrantLock类的子类：

```
final static class ThreadState extends ReentrantLock {
    ... ...
}
```

在文档的增删改（中）的文章中我们知道，一个线程总是从DWPTP中获得一个ThreadState，然后执行添加/更新文档的操作，当成功获得ThreadState，该线程就获得了ThreadState的锁，直到处理完文档后才会释放锁。如果在处理文档期间（未释放锁），有其他线程触发了全局的flush，并且ThreadState中持有的DWPT对象达到了flush的条件（见文档的增删改（中）中介绍ThreadState失去DWPT引用的章节），那么该DWPT会被添加到blockedFlushes中，并且在blockedFlushes中的DWPT

需要等待fullFlushBuffer中的所有DWPT 执行完doFlush( )(每个DWPT中收集的索引信息生成索引文件，或者说生成一个段)才能执行该操作。

为什么blockedFlushes中的DWPT最后才执行doFlush( )，具体原因会在介绍flush时候展开：

```
源码中给出的解释：only for safety reasons if a DWPT is close to the RAM limit
```

flushingWriters

定义如下：

```
private final IdentityHashMap<DocumentsWriterPerThread, Long> flushingWriters =
new IdentityHashMap<>();
```

flushingWriters容器的key为DWPT的引用，value为DWPT收集的索引信息占用的内存大小，flushingWriters中元素的个数描述了当前待flush的DWPT的个数，所有元素的value的和值描述了当前内存中的flushBytes（flushBytes的概念在文档的增删改（下）（part 1）已经介绍）。

flushQueue

flushQueue的定义如下：

```
private final Queue<DocumentsWriterPerThread> flushQueue = new LinkedList<>();
```

flushQueue中存放了待执行doFlush( )的DWPT集合。

fullFlushBuffer、blockedFlushes、flushingWriters、flushQueue之间的关联

当全局flush触发，fullFlushBuffer跟blockedFlushes中DWPT都会被添加进flushQueue，触发全局flush的线程总是只从flushQueue中依次取出每一个DWPT，当执行完doFlush( )的操作后，将该DWPT占用的内存大小从flushingWriters中移除，这里只是简单的概述下，详细的过程在介绍flush时展开。

flushDeletes

flushDeletes的定义如下：

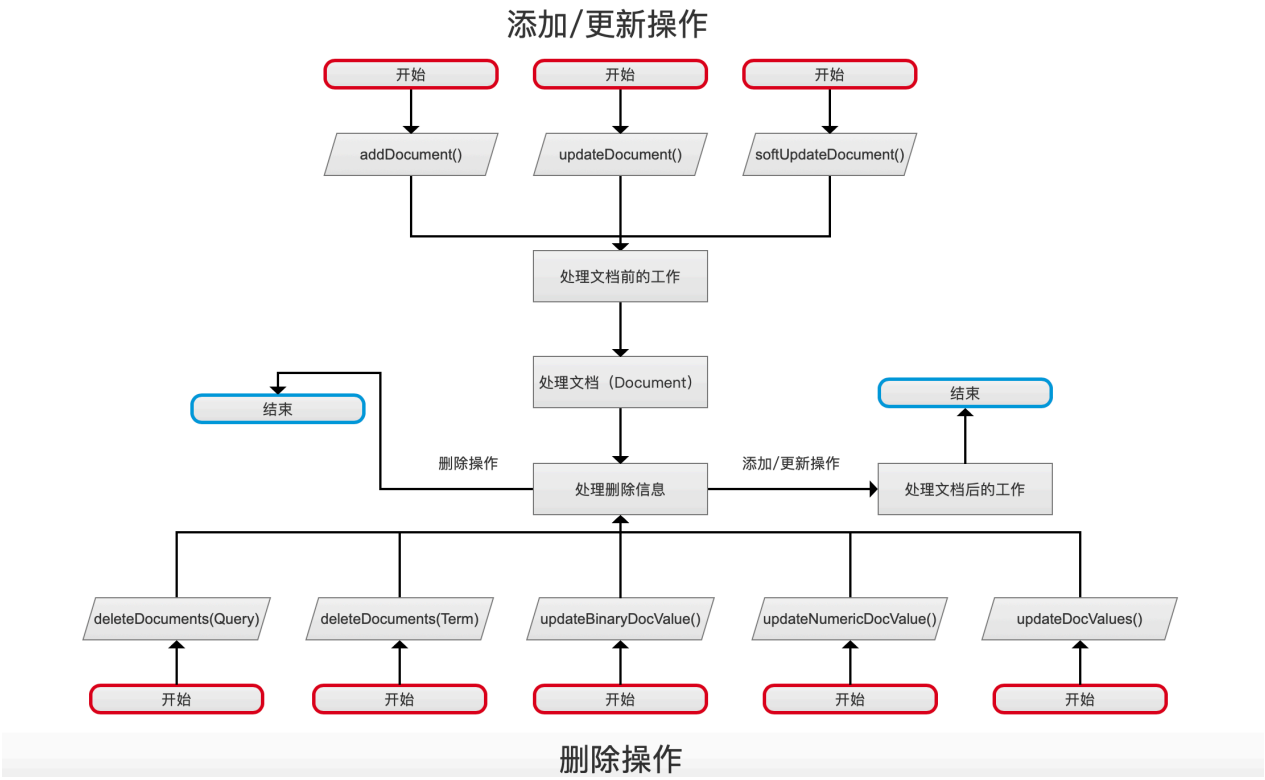
```
final AtomicBoolean flushDeletes = new AtomicBoolean(false);
```

每当将删除信息添加到DeleteQueue后，如果DeleteQueue中的删除信息使用的内存量超过ramBufferSizeMB，flushDeletes会被置为true。

文档的增删改流程图

我们继续介绍最后的 删除文档的`处理删除信息`与添加/更新文档的`处理文档后的工作`的流程点，添加/更新文档的`处理删除信息`的内容已经在文档的增删改（下）（part 2）介绍了。

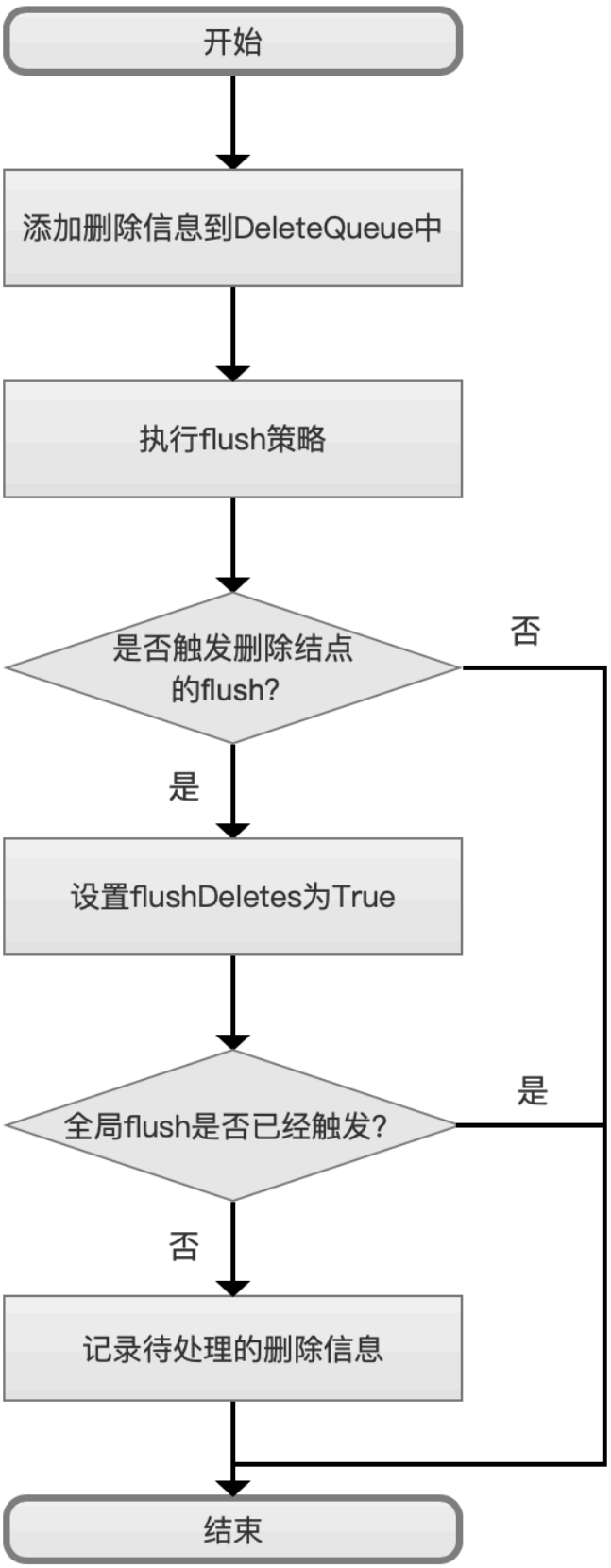
图1：



删除文档之`处理删除信息`

删除文档即deleteDocuments(Querys)、deleteDocuments(Terms)，不同于添加/更新文档，删除文档没有DWPT的概念，即没有私有DeleteSlice的概念。

图2：



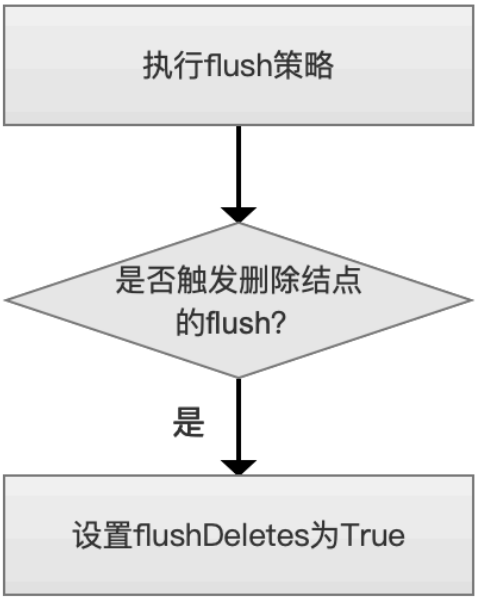
添加删除信息到DeleteQueue中

图3:

在文档的增删改（下）（part 2）中我们已经介绍了如何将一个删除信息添加到DeleteQueue中，故这里不赘述。

执行flush策略，设置flushDeletes

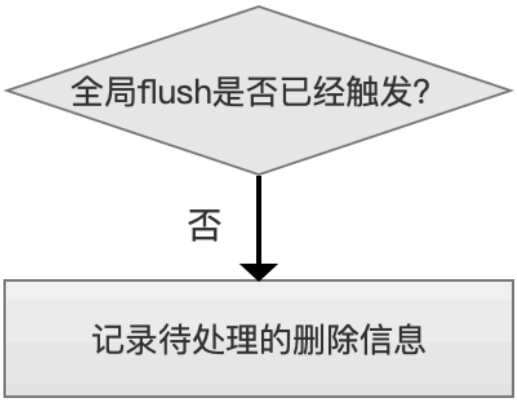
图4：



如果更新文档、删除文档的操作较多，堆积的删除信息占用的内存也不能被忽视，所以在添加到DeleteQueue后，需要判断DeleteQueue中的删除信息大小是否超过ramBufferSizeMB（ramBufferSizeMB描述了索引信息被写入到磁盘前暂时缓存在内存中允许的最大使用内存值，可以通过LiveIndexWriterConfig自定义配置），如果超过那么设置flushDeletes为True。

记录待处理的删除信息

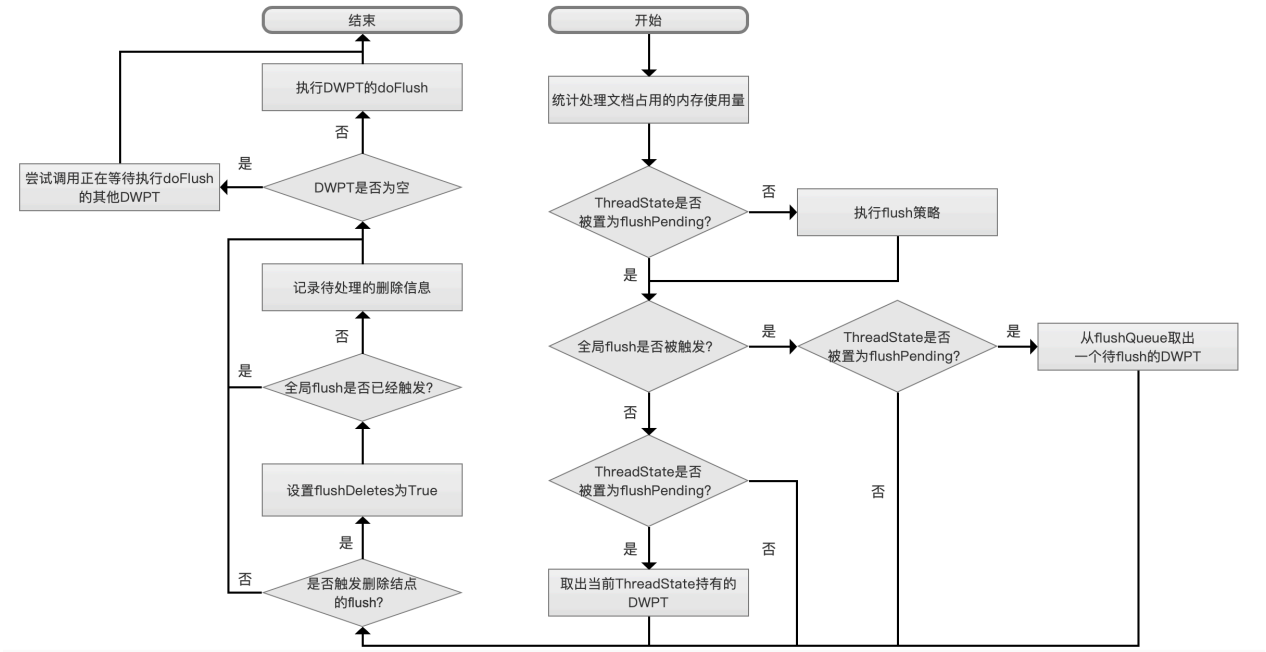
图5：



如果此时其他线程已经触发了全局flush，那么该线程会处理删除信息，故当前线程就不用重复的处理删除信息了，否则将删除信息添加到`待处理队列`中，生成一个`事件`，之后IndexWriter会处理该事件，`待处理队列`跟`事件`的概念在介绍flush时候会展开。

添加/更新之`处理文档后的工作`

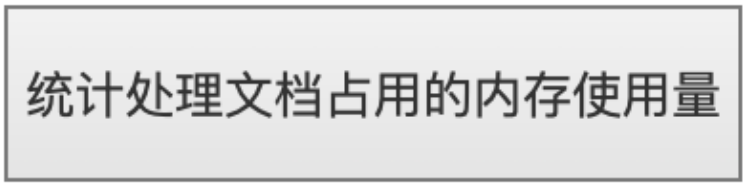
图6：



点击查看大图

统计处理文档占用的内存使用量

图7：



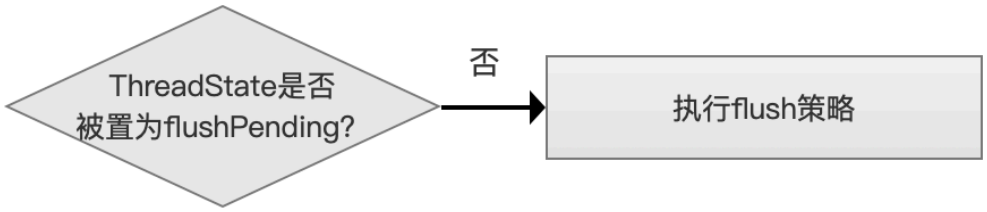
DWPT处理完一篇文档后，需要将这篇文档对应的索引信息占用的内存添加到全局的flushBytes或activeBytes，如果ThreadState被置为flushPending了，那么添加到flushBytes，否则添加到activeBytes。

为什么在这个流程点，ThreadState可能是flushPending的状态：

> 这种情况只发生在索引占用的内存(包括删除信息)达到ramBufferSizeMB的情况，如果线程1中的ThreadState处理完一篇文档后，索引占用的内存超过阈值，那么会从DWPTP中找到一个持有DWPT，并且该DWPT收集的索引信息量最大的ThreadState，将其置为flushPending，如果此时正好线程2开始执行添加/更新的操作，那么可能会从DWPTP中取出该ThreadState。

执行flush策略

图8：



如果当前的ThreadState没有被置为flushPending(内存索引未达到ramBufferSizeMB前ThreadState都是flushPending为false的状态)，那么需要执行flush策略，目前Lucene7.5.0中有且仅有一种flush策略：FlushByRamOrCountsPolicy。用户可以实现自己的flush策略并通过IndexWriterConfig自定义设置。

flush策略用来判断执行完增删改的操作后，是否要执行flush，这里的介绍是对文档的增删改（中）的补充：

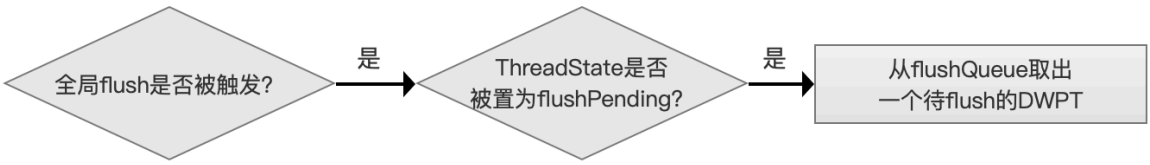
- 添加操作：如果设置了flushOnDocCount，那么判断DWPT处理的文档个数是否达到maxBufferedDocs，满足条件则将ThreadState置为flushPending，否则判断activeBytes与deleteRamByteUsed的和值是否达到ramBufferSizeMB，如果达到阈值，那么从DWPTP中找到一个持有DWPT，并且该DWPT收集的索引信息量最大的ThreadState，将其置为flushPending。
- 删除操作：判断deleteRamByteUsed是否达到ramBufferSizeMB，满足条件则另flushDeletes为true

- 更新操作：更新操作实际是执行删除跟添加的操作

另外每一个DWPT允许收集的索引信息量最大值为perThreadHardLimitMB （默认值为1945MB），当达到该阈值时，ThreadState会被置为flushPending，同样的可以通过IndexWriterConfig自定义设置，所以即使ramBufferSizeMB设置的很大时，也有可能因为达到了perThreadHardLimitMB而自动触发flush。

将当前ThreadState持有的DWPT添加到blockedFlushes中

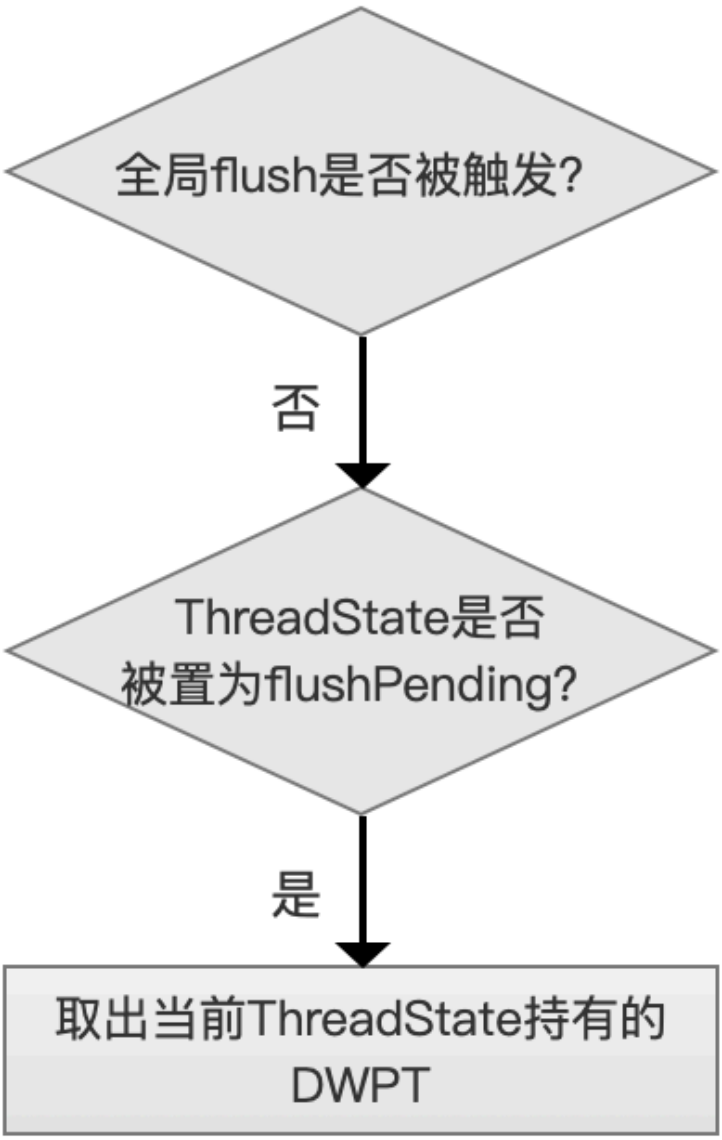
图9：



此时其他线程触发了全局flush，那么当前ThreadState如果没有被置为flushPending，说明其持有的DWPT没有达到flush的条件，ThreadState在随后释放锁后，其DWPT会被其他线程添加到fullFlushBuffer中，等待执行doFlush( )，否则该DWPT被添加到blockedFlushes中，然后从flushQueue中取出一个DWPT，目的就是尽可能先让flushQueue中的DWPT执行doFlush( )，只有这样，blockedFlushes中的DWPT才能随着尽快执行doFlush( )。

取出当前ThreadState持有的DWPT

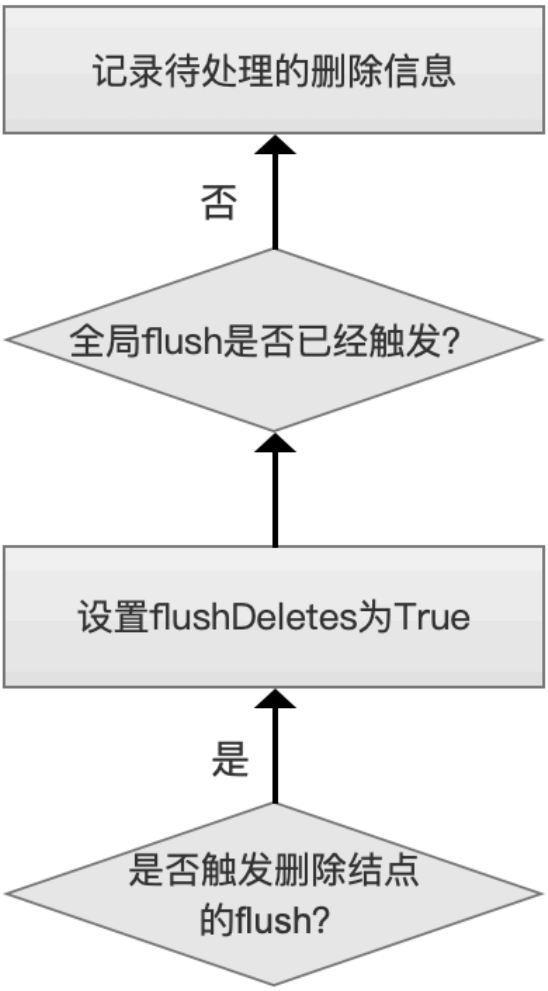
图10：



全局flush未被触发，如果当前ThreadState被置为flushPending，那么取出其持有的DWPT，在随后执行doFlush( )操作。

处理删除信息

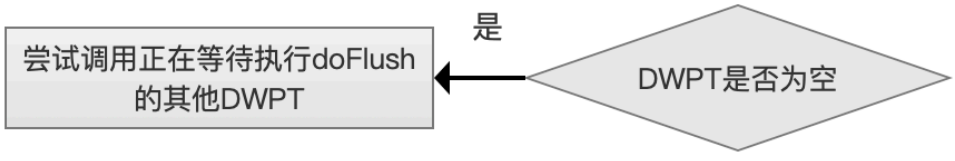
图11：



添加/更新操在处理完文档后，需要跟删除操作之`处理删除信息`一样，因为更新操作、其他线程的删除操作都会改变DeleteQueue。这一段逻辑在上文中已介绍，不赘述。

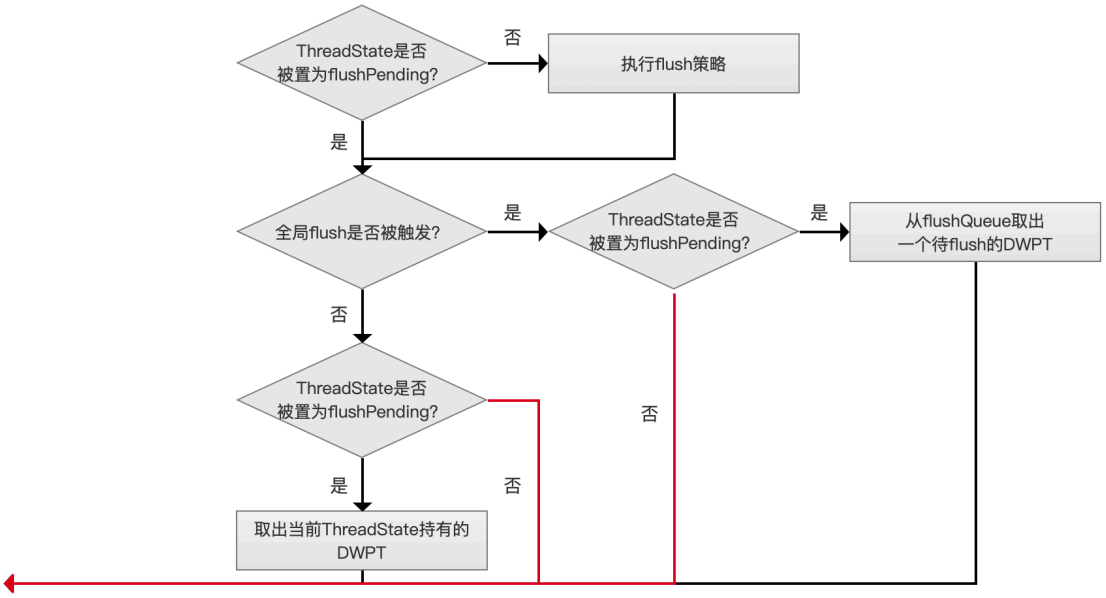
DWPT为空的情况

图12：



这里的描述不太好，在上面的流程中DWPT收集的索引信息未达到flush要求，故不需要处理该DWPT，故描述为DWPT为空，下图中的红线是DWPT为空的流程线。

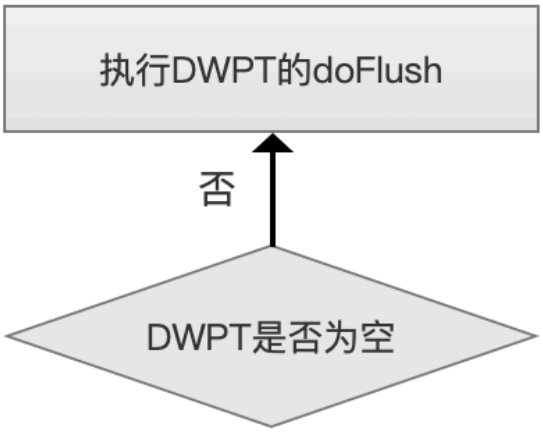
图13：



如果DWPT为空，那么我们这个线程需要"帮助"(help out)那些已经可以生成一个段的DWPT执行doFlush( )。帮助的方式是先尝试从flushQueue中找到一个DWPT，如果没有，那么从DWPTP中找，前提是当前没有触发全局flush，因为在全局flush的情况下，执行全局flush的线程会去DWPTP中找到所有已经收集过文档的DWPT，无论DWPT对应的ThreadState是否被置为flushPending。

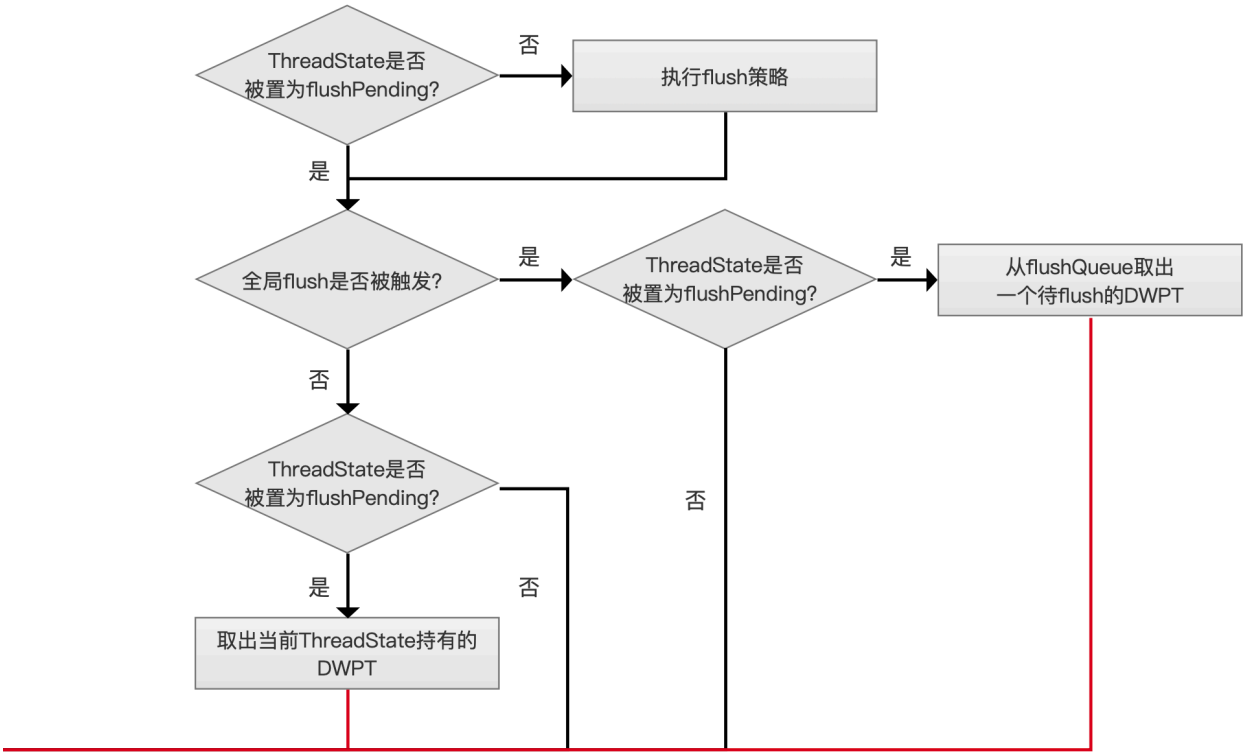
DWPT不为空的情况

图14：



DWPT不为空也就是下图中的两种情况，那么随后执行DWPT的doFlush( )：

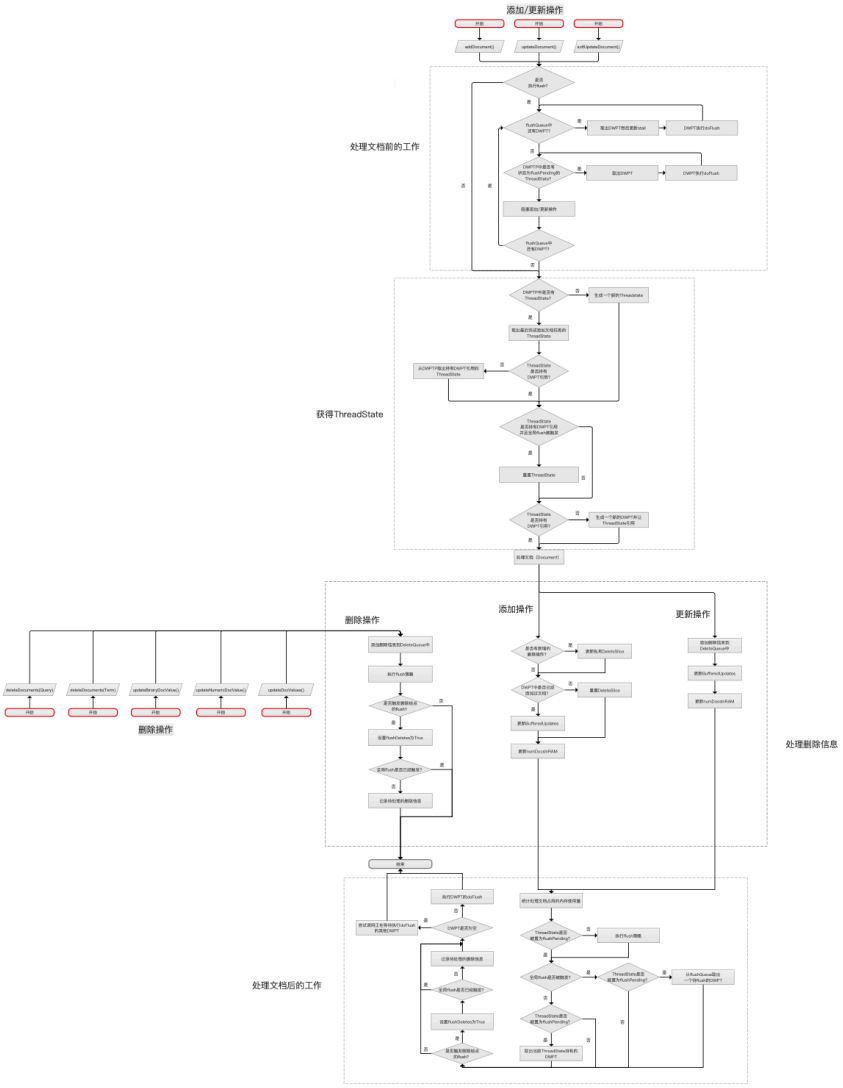
图15：



完整的单文档操作流程图

结合前几篇文章，单文档操作的流程图如下：

图16：



点击查看大图

结语

文档的增删改系列正式介绍完毕，如果你准备深入源码理解，那么必须结合全局flush的源码一起看，否则有些流程点很难理解，比如说上文中提高的三个容器，还有为什么ThreadState设计成继承可重入锁的类等等。

在下一篇介绍flush的文章中，当前系列的内容还会被反复提及，所以关心flush流程的朋友，需要先了解文档的增删改的流程。

点击下载附件