文档的增删改（下）（part 2）

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）继续介绍文档的增删改，为了能深入理解，还是得先介绍下几个预备知识。

预备知识

Node类

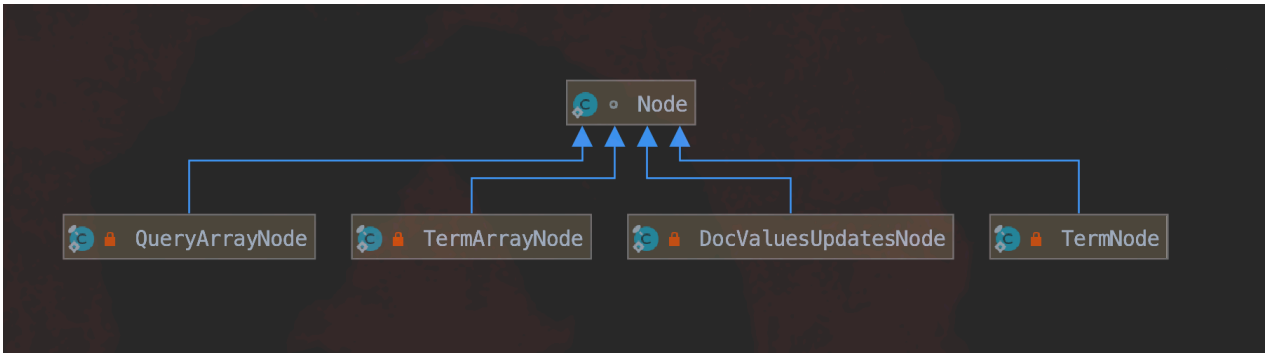
下面是Node类中仅有的两个成员变量：

```
static class Node<T> {
    volatile Node<?> next;
    final T item;
}
```

多个Node对象通过next实现了队列结构，其中item为队列中某个结点(Node)的删除信息，每当DWPT处理一个删除信息，就会将该删除信息作为一个item加入到队列中，即deleteQueue。

在文档的增删改（上）我们已经介绍了删除的几种方式，其删除信息会生成不同的Node子类：

图1：

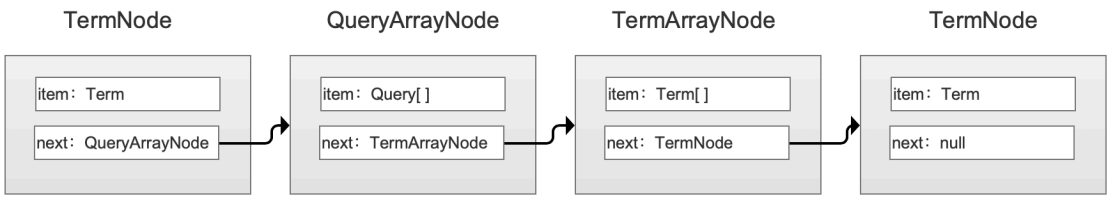


- QueryArrayNode：deleteDocuments(Querys)
- TermArrayNode：deleteDocuments(Terms)
- DocValuesUpdatesNode：updateBinaryDocValue( )、updateNumericDocValue( )、updateDocValues( )、softUpdateDocument( )、softUpdateDocuments( )
- TermNode：updateDocument(Term, Document)、updateDocuments(Term, Documents)

deleteQueue

在多线程（持有相同IndexWriter对象的引用）执行删除操作时，多个DWPT将自己携带的删除信息生成Node对象，并添加到deleteQueue中，deleteQueue是一个全局的变量：

图2：



为什么上图的队列有 "多线程"的前提：因为单线程的话每当添加一个Node到队列中，该Node的信息会马上被添加到BufferedUpdates（下文中会介绍）中，故在单线程下，不会出现结点数量大于1个的队列。

tail

该变量的定义如下：

```
private volatile Node<?> tail;
```

tail用来指向deleteQueue中最后一个Node结点，也就是最新的一个删除操作(latest delete operation)。

DeleteSlice类

下面是DeleteSlice仅有的两个成员变量：

```
static class DeleteSlice {
    Node<?> sliceHead;
    Node<?> sliceTail;
}
```

DeleteSlice中的sliceHead、sliceTail指向deleteQueue中的Node结点。

在文档的增删改（中）中我们知道，每一个执行添加/更新操作的线程会先从DWPTP获得一个ThreadState，如果ThreadState中没有持有DWPT的对象引用，那么需要生成一个新的DWPT对象让其持有，并且每一个DWPT对象中都拥有一个私有的DeleteSlice对象，并且在初始化DeleteSlice对象，会让DeleteSlice对象的sliceHead、sliceTail同时指向tail指向的deleteQueue对象中的Node结点，即最新的一个删除操作，DeleteSlice类的构造函数如下，其中参数currentTail为tail：

```
DeleteSlice(Node<?> currentTail) {
    // 另sliceHead、sliceTail与currentTail有相同的对象引用
    sliceHead = sliceTail = currentTail;
}
```

为什么sliceHead、sliceTail指向最后一个删除操作：因为最后一个删除操作只能对该删除操作执行前的添加的所有文档（满足删除要求的）进行删除，并且每一个DWPT中的索引信息最终会flush为一个段(在后面介绍flush时会详细介绍)，而DWPT的私有DeleteSlice对象记录了DWPT添加的第一篇文档（最后一个删除操作不能作用于该文档及后面的所有文档）到生成一个段这段时间内所有的删除操作，该操作包括自己跟其他线程的删除操作，这些删除信息在flush时需要作用于(apply)该DWPT对应的段中的文档，即删除段中满足删除要求的所有文档。

上面的解释有点绕。。。自己看着都有些变扭。

所以简单的表述就是sliceHead、sliceTail必须指向最后一个删除操作的目的就是不让该操作及之前的删除操作添加到DWPT私有的DeleteSlice对象中，因为这些操作不能作用于后生成的新的文档。

例子1

有两个线程ThreadA、ThreadB，他们分别调用了updateDocument(Term, Document)和deleteDocuments(Querys)，即分别会生成TermNode和QueryArrayNode。我们这里假设两个线程从DWPTP中获得的ThreadState对象持有的DWPT引用都是新生成的，并且当前DeleteQueue为空，并且假设（实际添加到deleteQueue顺序是不可确定的，下文会介绍）ThreadA先往deleteQueue中添加。

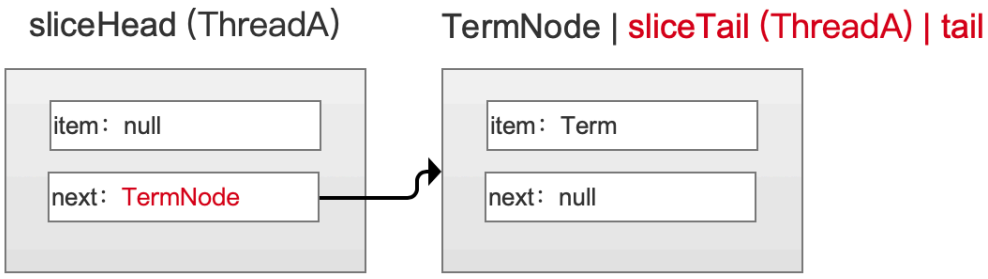
ThreadA添加结点

- 步骤一：获得一个ThreadState，新生成一个DWPT对象，初始化私有的DeleteSlice，由于deleteQueue未添加任何的删除结点，所以此时的deleteQueue的状态如下图, **其中红圈 tail、sliceHead (ThreadA)、sliceTail (ThreadA) 三个具有相同的对象引用：**

图3：

- 步骤二：添加TermNode结点到deleteQueue中，sliceTail (ThreadA) 跟tail 的对象引用更新为TermNode对象

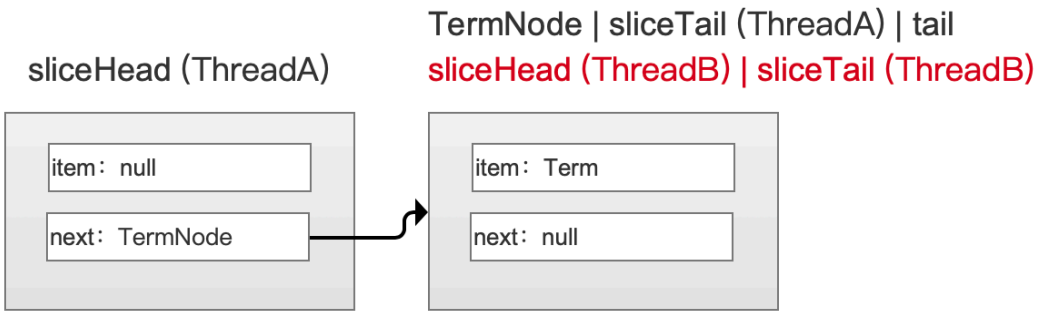
图4：



ThreadB添加结点

- 获得一个ThreadState，新生成一个DWPT对象，初始化私有的DeleteSlice，即让sliceHead(ThreadB)、sliceTail(ThreadB)跟tail指向相同的对象。

图5：



- 添加QueryArrayNode结点到deleteQueue中,sliceTail (ThreadB) 跟tail 的对象引用更新为QueryArrayNode的对象

图6：

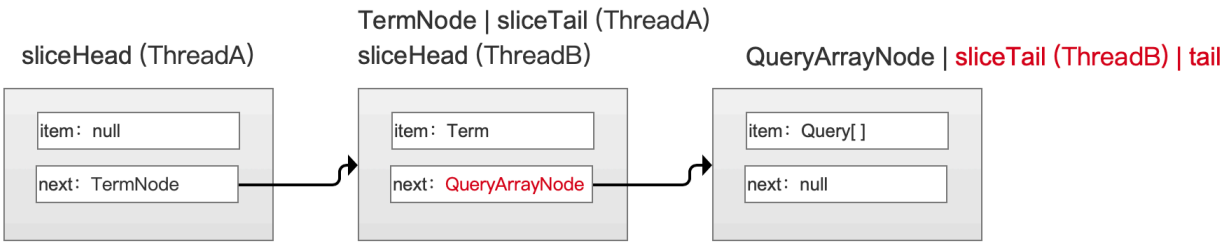


图6中，deleteQueue反映了两个情况：

- 情况1：ThreadA的删除操作，即TermNode不会作用于ThreadB添加的文档
- 情况2：ThreadB的删除操作，即QueryArrayNode会作用于ThreadA添加的文档（在介绍flush时会展开介绍作用（apply）的过程）

情况1是因为删除操作不能作用于后新增的文档，情况2是因为删除操作要作用于该删除操作前已添加的所有文档。

例子2：

由于DWPT初始化私有的DeleteSlice跟添加到删除结点到deleteQueue 这两个操作不是一个临界区内的两个操作，只有添加到deleteQueue是线程同步操作（synchronized），所以根据CPU调度的不同，同样是例子1中两个线程的删除操作，生成的deleteQueue不总是一样，下面例子的一个假设前提是deleteQueue为空。

例如图7跟图8是ThreadA跟ThreadB初始化各自私有的DeleteSlice，并且DeleteQueue还没有删除结点，ThreadA和ThreadB`分别先`将删除结点添加到DeleteQueue中：

图7：

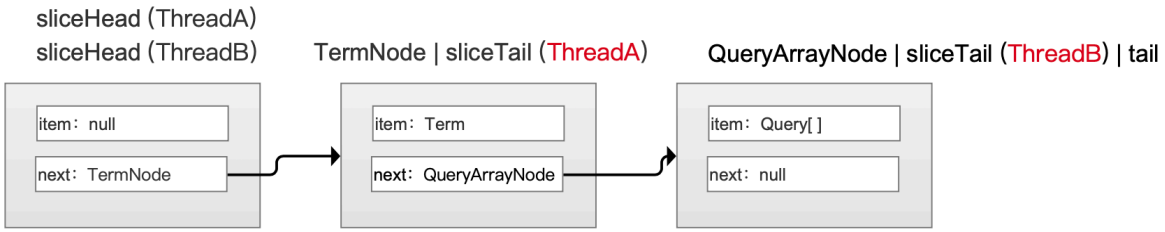
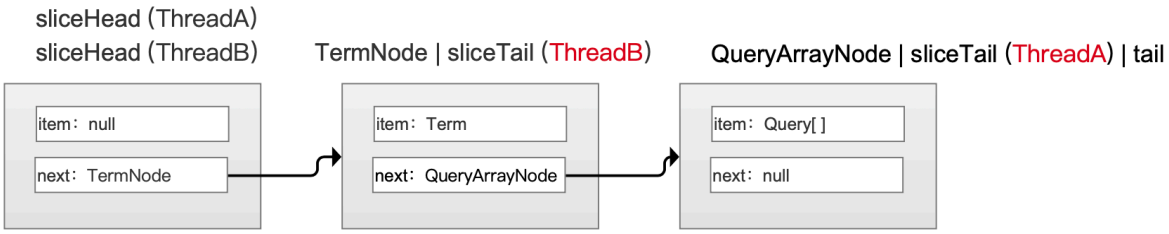


图8：



BufferedUpdates类

上文中提到一个DWPT从生成到flush到一个段，在这段期间其私有的DeleteSlice会记录所有的删除操作，这些删除操作会作用于(apply)DWPT添加的文档，即删除那些满足删除要求的文档，然而对于某一个删除操作来说，它只能作用于该删除操作前DWPT已添加的文档，其随后新添加的文档不能被apply，BufferedUpdates类就是通过以下几个Map对象来描述 某一个删除操作的作用范围(apply scope)：

- Map<Term,Integer> deleteTerms
- Map<Query,Integer> deleteQueries
- Map<String,LinkedHashMap<Term,NumericDocValuesUpdate>> numericUpdates：暂不作介绍
- Map<String,LinkedHashMap<Term,BinaryDocValuesUpdate>> binaryUpdate：暂不作介绍

numericUpdates、binaryUpdate会在介绍`软删除`的文章中展开，这里暂不作介绍（如果这里展开介绍，软删除的文章就没东西可写了😝）。

在deleteTerms中，该Map的key为Term，表示包含该Term的文档都会被删除，value为一个哨兵值，描述了该删除操作的作用范围，即只能作用于文档号小于哨兵值的文档，这里需要补充一个概念：

- numDocsInRAM：该值是每一个DWPT的私有变量。DWPT每添加一个文档，会为该文档赋予一个文档号，文档号是从0开始递增的值，numDocsInRAM描述了当前添加的文档数量

在deleteQueries中，该Map的key为Query，表示满足该查询要求的文档都会被删除，value的概念同deleteTerms。

哨兵值怎么用：

- 在deleteTerms中：在DWPT将索引信息生成索引文件期间，利用倒排表中的信息找到包含该Term的所有文档的文档号，文档号小于哨兵值的文档都会删除，当然所谓的""删除""其实是将被删除的文档号从文档号集合（docId Set，0 ~ (numDocsInRAM - 1)的文档号集合）中剔除，在处理完所有的删除(比如下文中的deleteQueries)后，该集合会生成索引文件.liv
- 在deleteQueries中：在DWPT将索引信息生成索引文件之后，通过查询的方式找出满足删除要求的文档号，然后从文档号集合中剔除这些文档号

处理被删除的文档号的详细过程在介绍flush时会详细展开。

例子

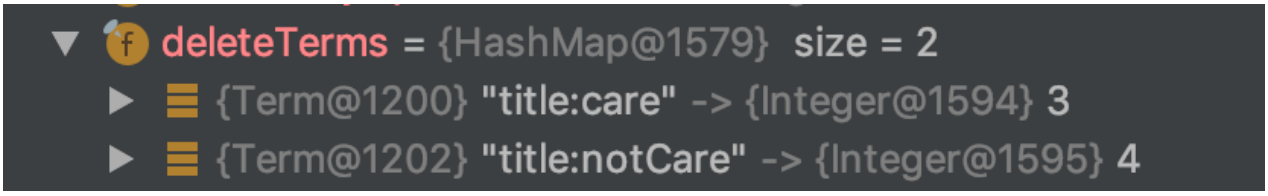
下面的例子介绍了当出现多个更新操作，并且存在相同删除操作的情况下，deleteTerms如何处理：

图9：

```
78      // 文档0
79      doc = new Document();
80      doc.add(new StringField( name: "author",  value: "Lucy", Field.Store.YES));
81      doc.add(new StringField( name: "title",  value: "care", Field.Store.YES));
82      indexWriter.addDocument(doc);
83      // 文档1
84      doc = new Document();
85      doc.add(new StringField( name: "author",  value: "Wang", Field.Store.YES));
86      Term term1 = new Term( fld: "title",  text: "care");
87      indexWriter.updateDocument(term1, doc);
88      // 文档2
89      doc = new Document();
90      doc.add(new StringField( name: "author",  value: "Lily", Field.Store.YES));
91      doc.add(new StringField( name: "title",  value: "care", Field.Store.YES));
92      indexWriter.addDocument(doc);
93      // 文档3
94      doc = new Document();
95      doc.add(new StringField( name: "content",  value: "nothing", Field.Store.YES));
96      Term term2 = new Term( fld: "title",  text: "care");
97      indexWriter.updateDocument(term2, doc);
98      // 文档4
99      doc = new Document();
100     doc.add(new StringField( name: "content",  value: "everything", Field.Store.YES));
101     Term term3 = new Term( fld: "title",  text: "notCare");
102     indexWriter.updateDocument(term3, doc);
```

- 第一个更新操作：第87行，需要删除包含term1的文档，此时deleteTerms中key为term1，value为1,因为当前添加了两篇文档（注意：更新操作是 先添加 后删除 的操作，故此时文档1已经被添加），value的值为1，表示该删除作用范围是文档号在[0, 1)左闭右开的区间内的文档
- 第二个更新操作：第97行，需要删除包含term2的文档，此时deleteTerms中key为还是term1（term1跟term2的hash值是一样的），value更新为3，表示该删除作用范围是文档号在[0, 3)左闭右开的区间内的文档
- 第三个更新操作：第102行，需要删除包含term3的文档，此时deleteTerms相比较前两次更新操作，新增了一个key为term3，value为4的元素，表示该删除作用范围是文档号在[0, 4)左闭右开的区间内的文档，当然了我们用肉眼看一下，这个操作不会删除任何文档，因为[0, 4)的文档没有一篇文档满足删除要求。

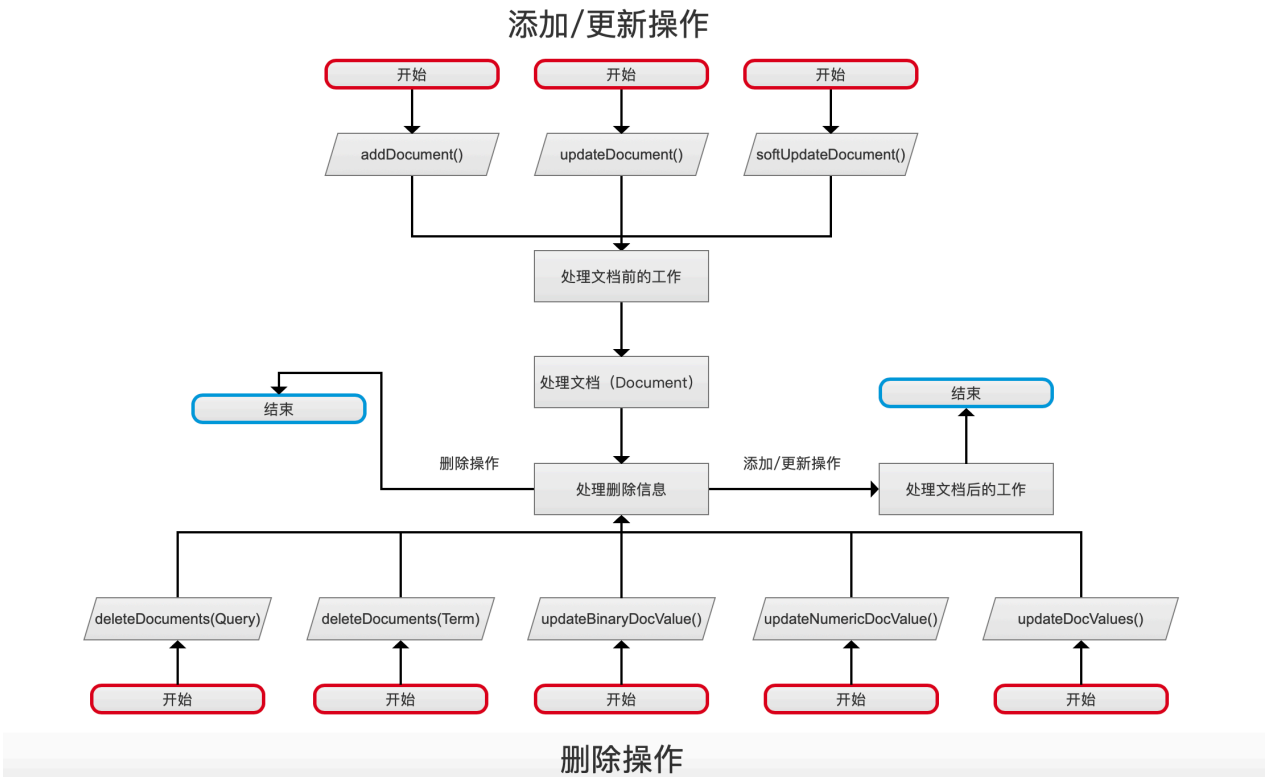
故最终deleteTerms中包含了两条删除信息： 图10：



文档的增删改流程图

在文档的增删改（上）的文章中，我们给出了文档的增删改流程图，我们紧接文档的增删改（下）（part 1），继续介绍剩余的两个流程点：处理删除信息、处理文档后的工作。

图11：



处理删除信息

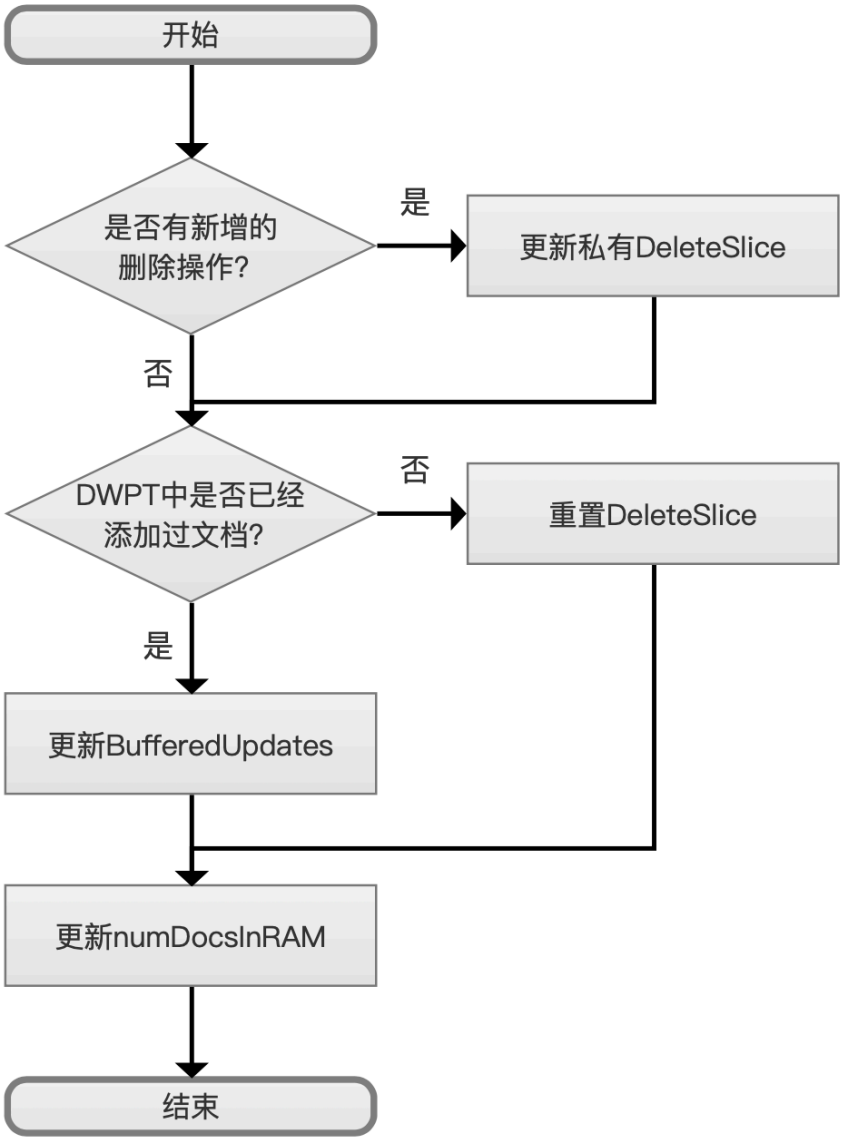
图12：

文档的增删改都需要处理删除信息。

添加文档

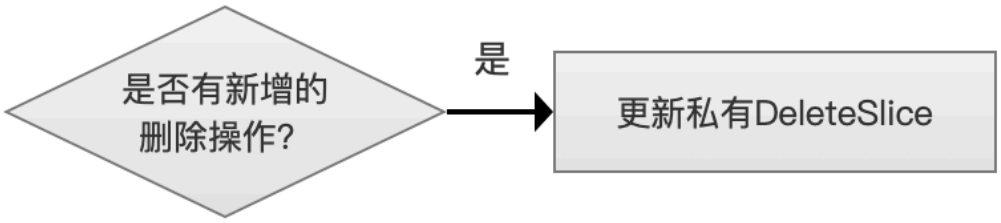
添加文档处理删除信息的流程图

图13：



更新DeleteSlice

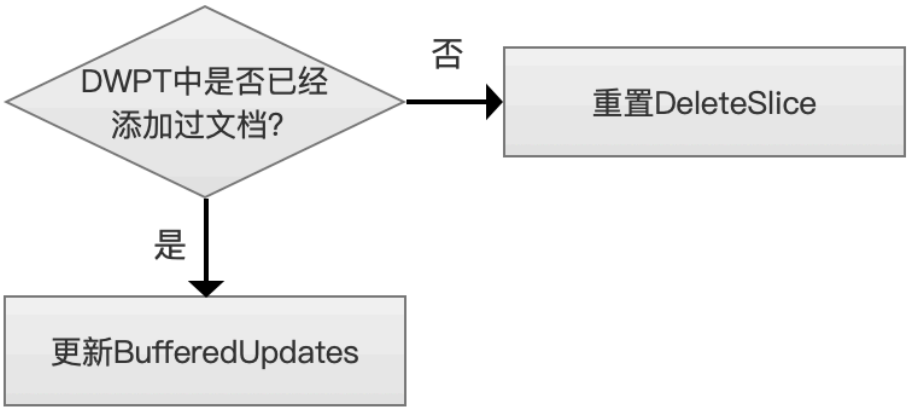
图14：



检查其他线程是否新增了删除操作，这些删除操作要作用于DWPT已经添加的文档，判断方式通过判断DWPT的私有DeleteSlice对象中的sliceTail是否跟tail有相同的对象引用，如果不是，那么更新私有DeleteSlice，即另sliceTail引用跟tail相同的对象。

重置DeleteSlice

图15：



通过DWPT的numDocsInRAM是否大于0来判断是否已经添加过文档：

- 没有添加过文档：只有DWPT第一次执行添加/更新的操作，该值才会为0，由于上一个流程点，该DWPT的私有DeleteSlice可能被更新了，但由于DWPT中没有添加过任何的文档，所以所以私有DeleteSlice中不需要删除信息，故需要重置DeleteSlice，重置的方法即将另私有DeleteSlice的sliceHead引用跟sliceTail相同的对象，当sliceHead跟sliceTail引用相同的对象时，表示DWPT的私有DeleteSlice没有删除信息
- 添加过文档：如果上一个流程点没有更新私有DeleteSlice，那么就不要更新BufferedUpdates，否则需要将新增的删除信息添加到BufferedUpdates中，更新BufferedUpdates的过程已在上文BufferedUpdate类中介绍，不赘述

更新numDocsInRAM

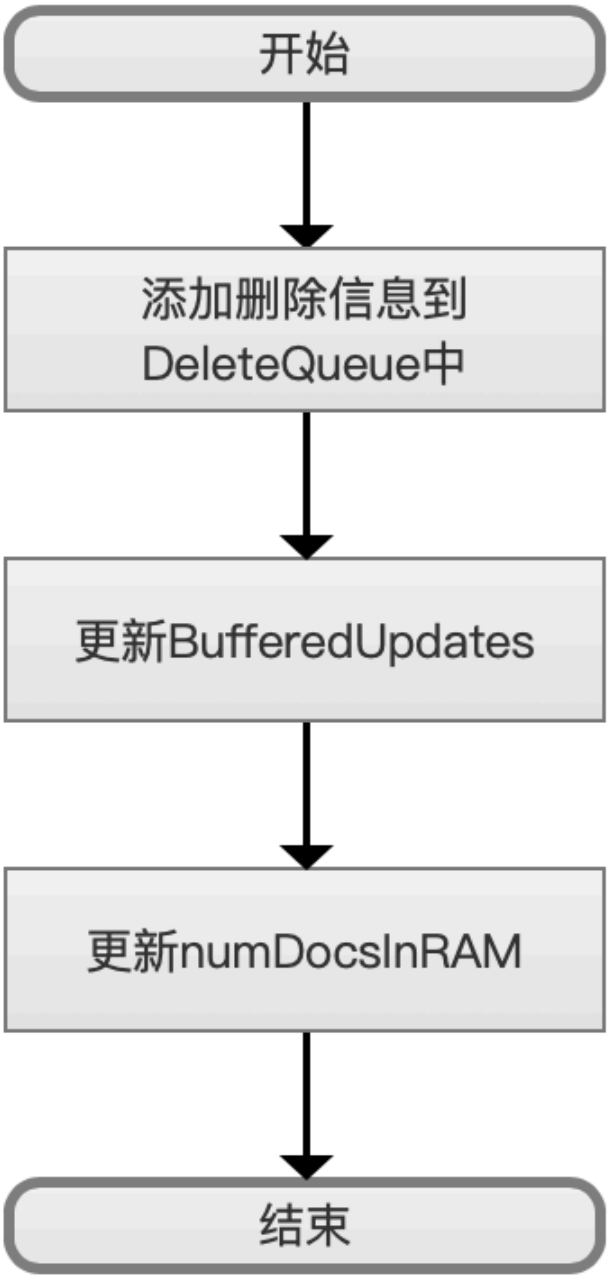
图16：

到达此流程点说明我们添加了一篇文档，故更新numDocsInRAM的值，即执行numDocsInRAM++的操作。

更新文档

添加文档处理删除信息的流程图

图17：



将DWPT自己的删除信息添加到DeleteQueue中，然后更新BufferedUpdates，注意的是，在上文中我们知道，当前DWPT添加到DeleteQueue时可能有其他的线程先添加了新的删除信息，故可能会更新自己以及其他线程的删除信息到BufferedUpdates。

删除文档

由于删除文档的`处理删除信息`的流程与添加/更新文档的`处理文档后的工作`的流程有很多的相似点，又因为剩余的内容会导致本篇文章的篇幅过长，故在下一篇文章中展开。

结语

无

点击下载附件