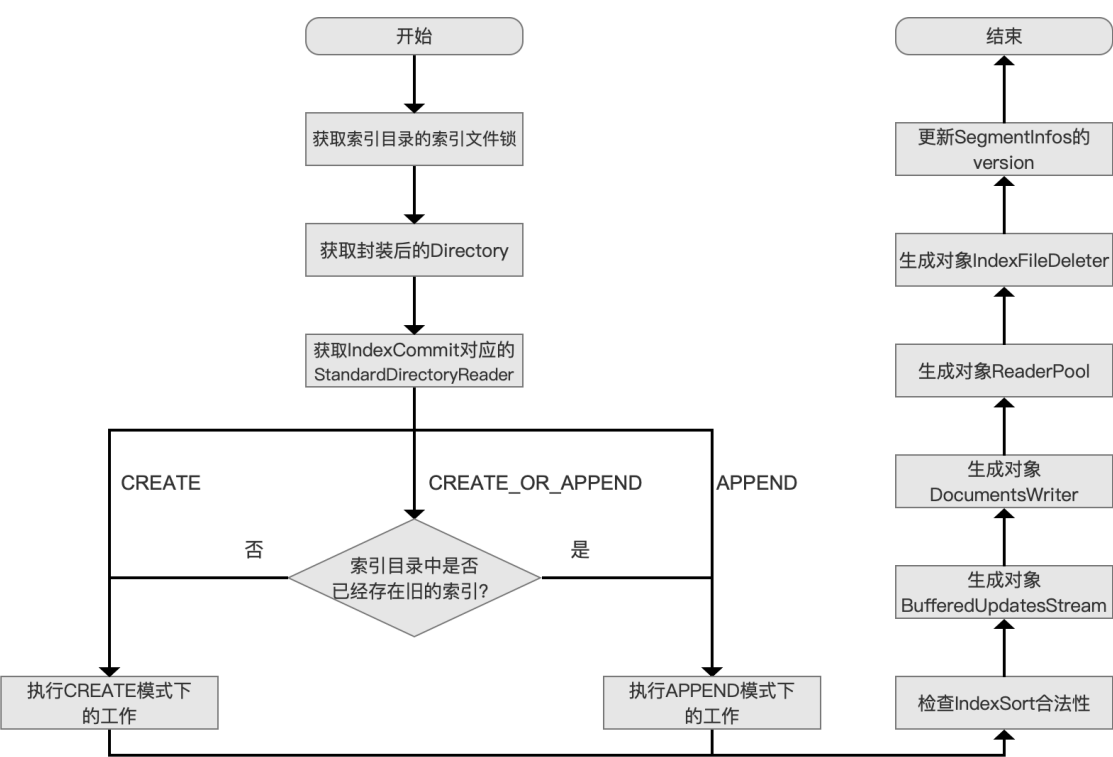


构造IndexWriter对象（七）

本文承接[构造IndexWriter对象（六）](#)，继续介绍调用IndexWriter的构造函数的流程。

调用IndexWriter的构造函数的流程图

图1：

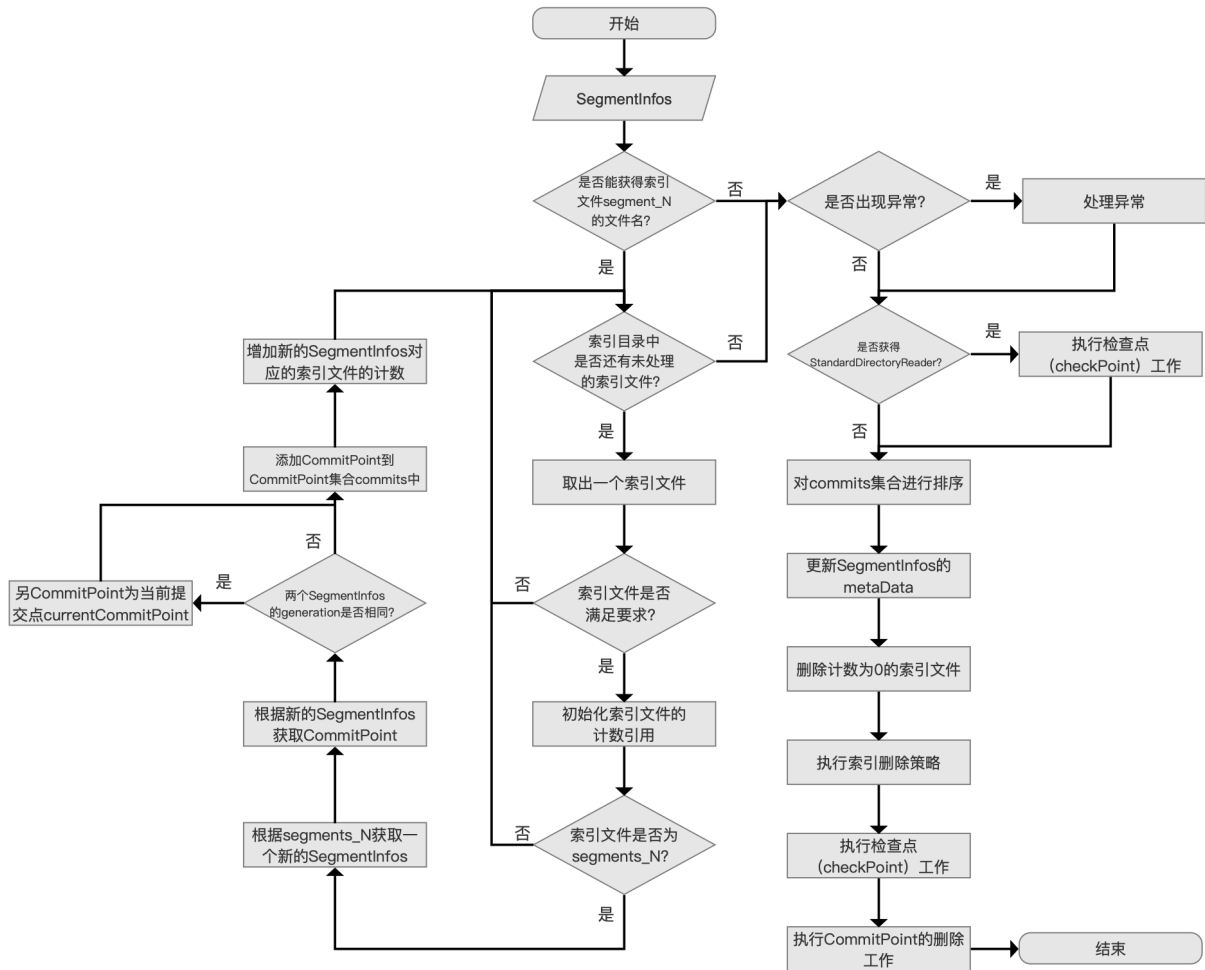


生成对象IndexFileDeleter

在文章[构造IndexWriter对象（六）](#)中，我们简单介绍了IndexFileDeleter作用，即用来删除索引目录中的索引文件，本文根据IndexFileDeleter的构造函数的实现来介绍关于计数引用、删除无效（Invalid）索引文件的内容。

IndexFileDeleter的构造函数流程图

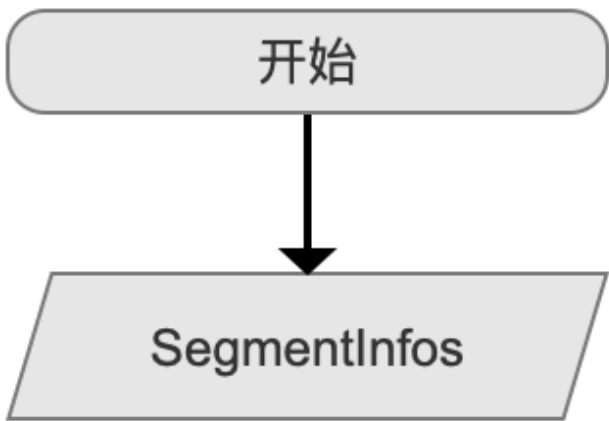
图2：



[点击查看大图](#)

SegmentInfos

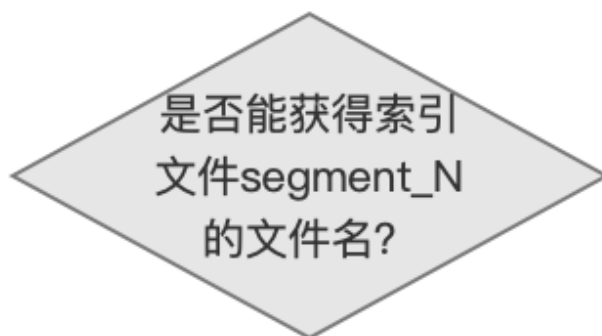
图3:



构造 IndexFileDeleter最重要的一个对象就是SegmentInfos，它是在图1的 生成对象 IndexFileDeleter 流程点之前通过用户配置的IndexCommit或者索引目录中的旧索引生成的 SegmentInfos对象（见[构造IndexWriter对象（三）](#)、[构造IndexWriter对象（四）](#)、[构造IndexWriter对象（五）](#) 中关于生成SegmentInfos的介绍），并作为 IndexFileDeleter的构造函数的参数之一。

是否能获得索引文件segments_N的文件名？

图4：

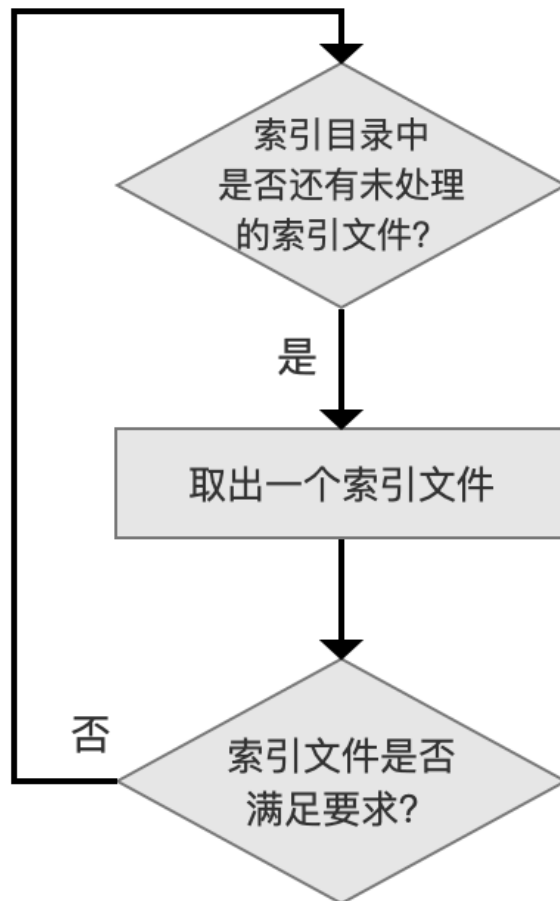


构造IndexWriter对象里面调用 IndexFileDeleter的构造函数时，总是能通过SegmentInfos获得一个segments_N的文件名，并且通过下面的方法来获得segments_N的文件名，那么当前流程点的判断结果总是为true，由于该方法可能会返回null，所以在这里会有一个判断。

```
SegmentInfos.getSegmentsFileName()
```

判断索引文件是否满足要求

图5：



接着就是从索引目录依次取出索引文件，然后判断是否满足某个要求。

需要满足什么要求：

先给出该要求对应的代码：

```
if (!fileName.endsWith("write.lock") && (m.matches() ||  
fileName.startsWith(IndexFileNames.SEGMENTS) ||  
fileName.startsWith(IndexFileNames.PENDING_SEGMENTS)){  
    ...  
}
```

其中fileName就是待处理的索引文件的文件名、m.matches()描述的是fileName是否满足下面的正则表达式

```
_[a-z0-9]+(_.*)?\\.*
```

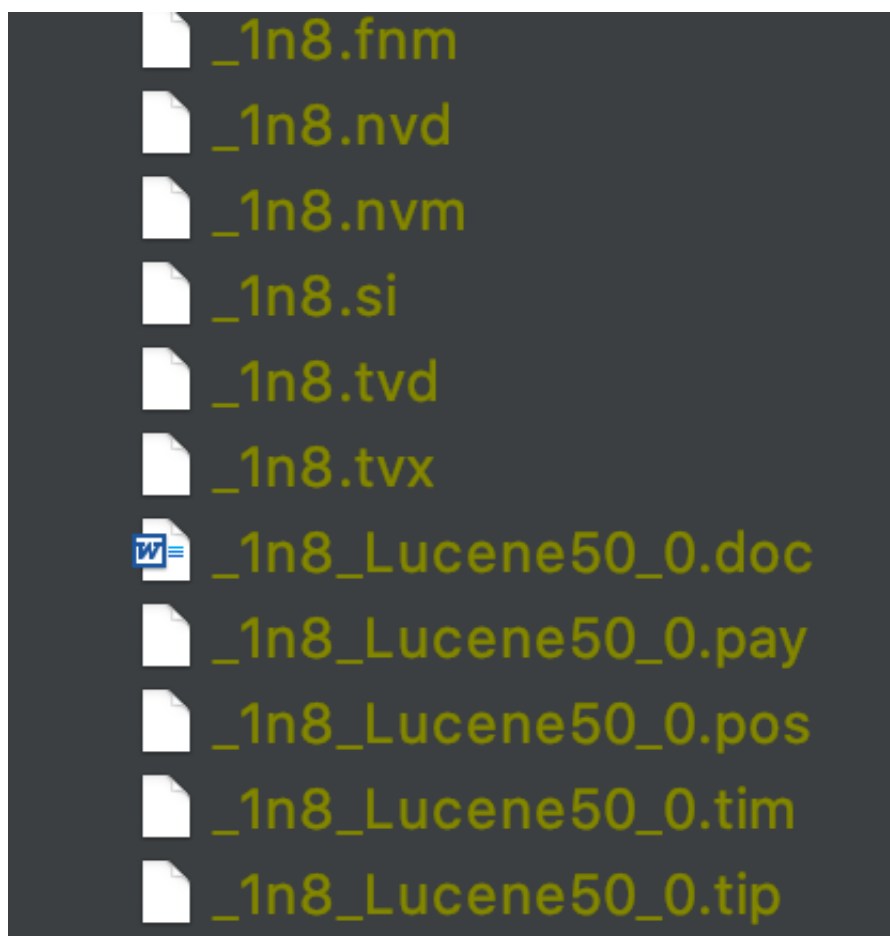
上述的代码进行拆分后即需要满足下面三个字要求之一：

- 子要求一：!fileName.endsWith("write.lock") && (m.matches()
 - writer.lock即索引目录的[索引文件锁](#)，用来同步不同的IndexWriter对象，只允许一个IndexWriter可以操作同一个索引目录，占用了索引文件锁的IndexWriter可以通过调用

Index.close()方法来释放该锁，writer.lock不满足要求

- m.matches()则是根据正则表达式来匹配命名方式为下图中的文件名，满足正则表达式即满足要求：

图6：



- 子要求二：fileName.startsWith(IndexFileNames.SEGMENTS)
 - 满足以"segment"开头的文件名即满足子要求二，比如segments_N文件
- 子要求三：fileName.startsWith(IndexFileNames.PENDING_SEGMENTS)
 - 满足以"pending_segments"开头的文件名即满足子要求三，比如执行commit()操作时，在两段提交之第一阶段会生成该命名方式的文件，详见文件[文档提交之commit \(二\)](#)

初始化索引文件的计数引用

图7：

初始化索引文件的
计数引用

如果索引文件满足上文中的要求，那么我们初始化这些索引文件的计数引用。

Lucene中如何实现对索引文件的计数引用：

通过一个HashMap对象refCounts以及一个RefCount类实现，他们的定义如下所示：

```
final private static class RefCount {
    final String fileName; // 索引文件名
    int count; // 计数值

    public int IncRef() {
        // 增加计数
    }

    public int DecRef() {
        // 减少计数
    }
}

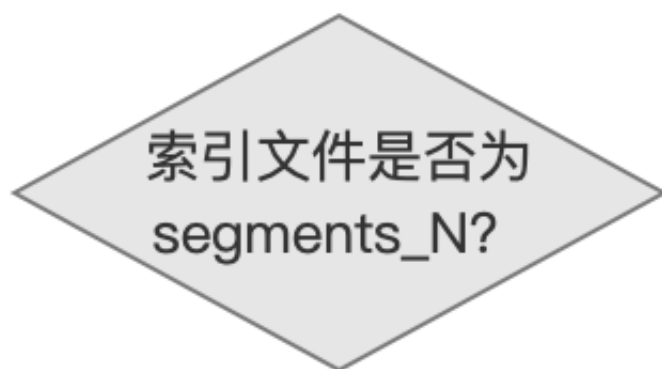
// key为索引文件名, value为RefCount对象
private Map<String, RefCount> refCounts = new HashMap<>();
```

如果我们要增加某个索引文件的计数引用，那么根据refCounts找到该文件对应的RefCount对象，接着通过对象的IncRef()方法来增加计数。

上述的RefCount类的具体内容可以查看这个链接：<https://github.com/LuXugang/Lucene-7.5.0/blob/master/solr-7.5.0/lucene/core/src/java/org/apache/lucene/index/IndexFileDeleter.java>。

索引文件是否为segments_N?

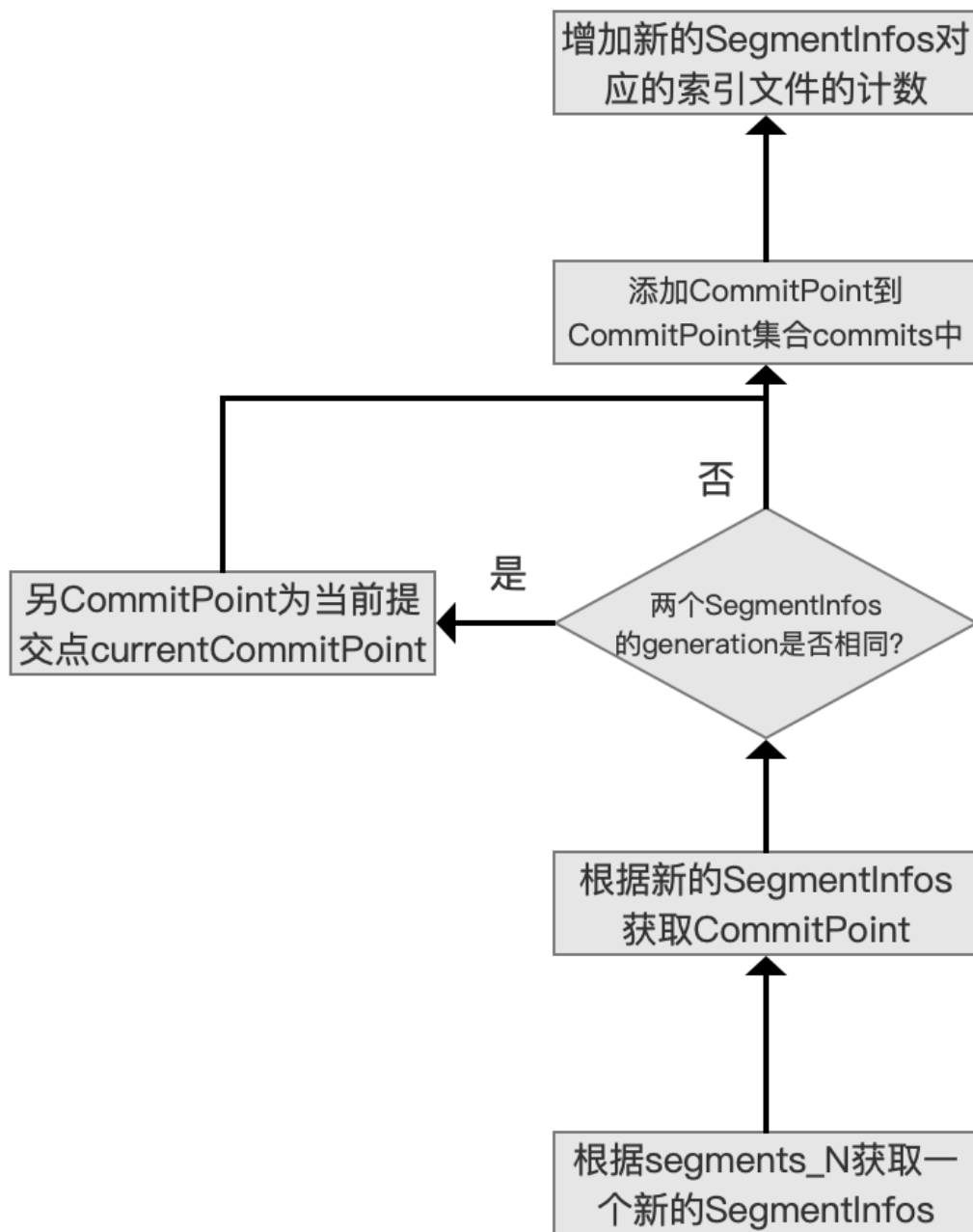
图8：



如果是segments_N文件，那么需要对该文件进行额外的处理，判断方法同上文中的子要求二。

处理索引目录中所有的segments_N

图9：



索引目录中可能存在多个segments_N文件，那么这些文件都需要被处理，其中只有一个segments_N对应的SegmentInfos为构造IndexFileDeleter对象的参数，即图2中 `SegmentInfos` 流程点的segmentInfos。

为什么索引目录中会存在多个segments_N文件：

原因主要取决于上一个IndexWriter对象使用了哪种索引删除策略IndexDeletionPolicy（见[文档提交之commit（二）](#)关于IndexDeletionPolicy的介绍），比如使用了索引删除策略NoDeletionPolicy，那么每次提交都会保留，又比如使用了默认的索引删除策略KeepOnlyLastCommitDeletionPolicy，那么只会保留最后一次提交。

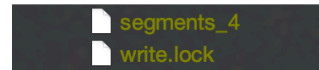
图10：

索引删除策略IndexDeletionPolicy

NoDeletionPolicy



KeepOnlyLastCommitDeletionPolicy



在图10中，使用不同的索引删除策略对相同的数据进行索引，在执行了4次commit提交后，对于NoDeletionPolicy来说，它会保留所有的提交，而对于KeepOnlyLastCommitDeletionPolicy，当生成segments_2时，会删除segments_1，生成segments_3时，会删除segments_2，以此类推，即只保留最后一次提交。

为什么要根据每一个segments_N对应的SegmentInfos生成CommitPoint，并且添加到CommitPoint集合commits中：

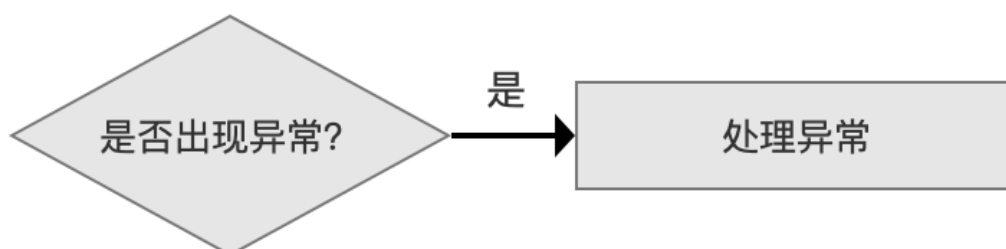
- 原因是我们需要根据正在构造的IndexWriter对象中的索引删除策略来处理这些提交，而CommitPoint对象为索引删除策略作用的对象。
- 这里将所有的CommitPoint添加到commits中是为了在下面的流程中作为索引删除策略的输入数据进行统一处理。

接着如果某个segments_N对应的SegmentInfos为构造IndexFileDeleter的参数，即图2中SegmentInfos流程点的SegmentInfos，那么它对应生成的CommitPoint被设置为当前提交点currentCommitPoint，该对象在后面的流程点是否出现异常会作为条件进行异常判断。

最后我们需要增加每一个segments_N对应的SegmentInfos中对应的索引文件的计数引用，其原因是在后面的流程中能判断能否删除索引文件。

是否处理异常

图11：



这里考虑的异常是使用NFS（Network File System）网络文件系统的场景，使用该文件系统可能会导致下面的情况：我们能通过构造IndexFileDeleter的参数获得SegmentInfos对象，并且通过SegmentInfos获得segments_N的文件名（注意只是文件名），那么segments_N文件必定是在索引目录中（见[构造IndexWriter对象（五）](#)/[构造IndexWriter对象（四）](#)/[构造IndexWriter对象（三）](#)不同的打开模式OpenMode的内容），但是我们在图9的流程中，如果没有读取到segments_N文件（通过是

否获得currentCommitPoint对象来判断），那么有可能就是NFS导致的，例如目录缓存机制的影响，那么可以根据segment_N文件名，通过重试机制来获得它对应的SegmentInfos对象，并且生成currentCommitPoint对象，如果还是读取不到，那么就会抛出下面的异常：

```
throw new CorruptIndexException("unable to read current segments_N file",
currentSegmentsFile, e);
```

结语

基于篇幅，剩余的内容将在下一篇文章中展开。

[点击](#)下载附件