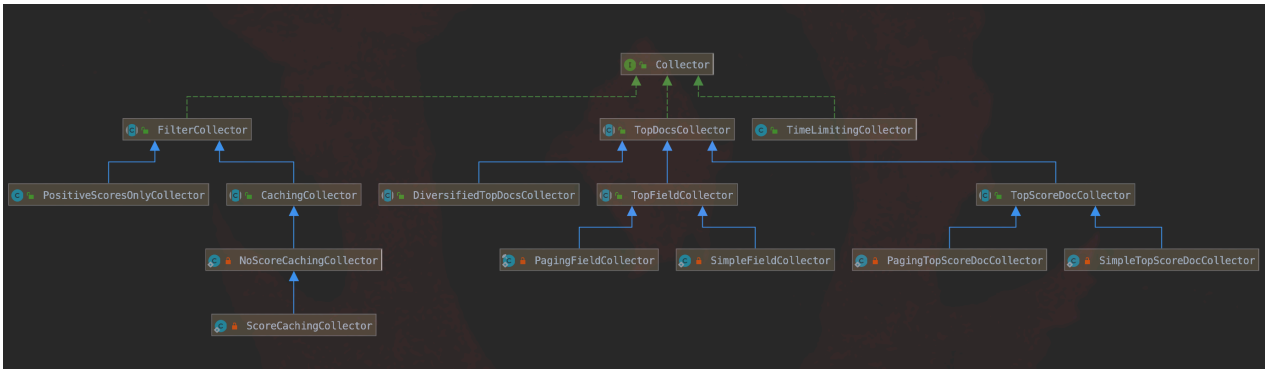
Collector（一）

在搜索阶段，每当Lucene找到一个满足查询条件的文档（Document），便会将该文档的文档号（docId）交给Collector，并在Collector中对收集的文档号集合进行排序（sorting）、过滤（filtering）或者用户自定义的操作。

本篇文章将根据图1中的类图（Class diagram），介绍Lucene常用的几个收集器（Collector）：

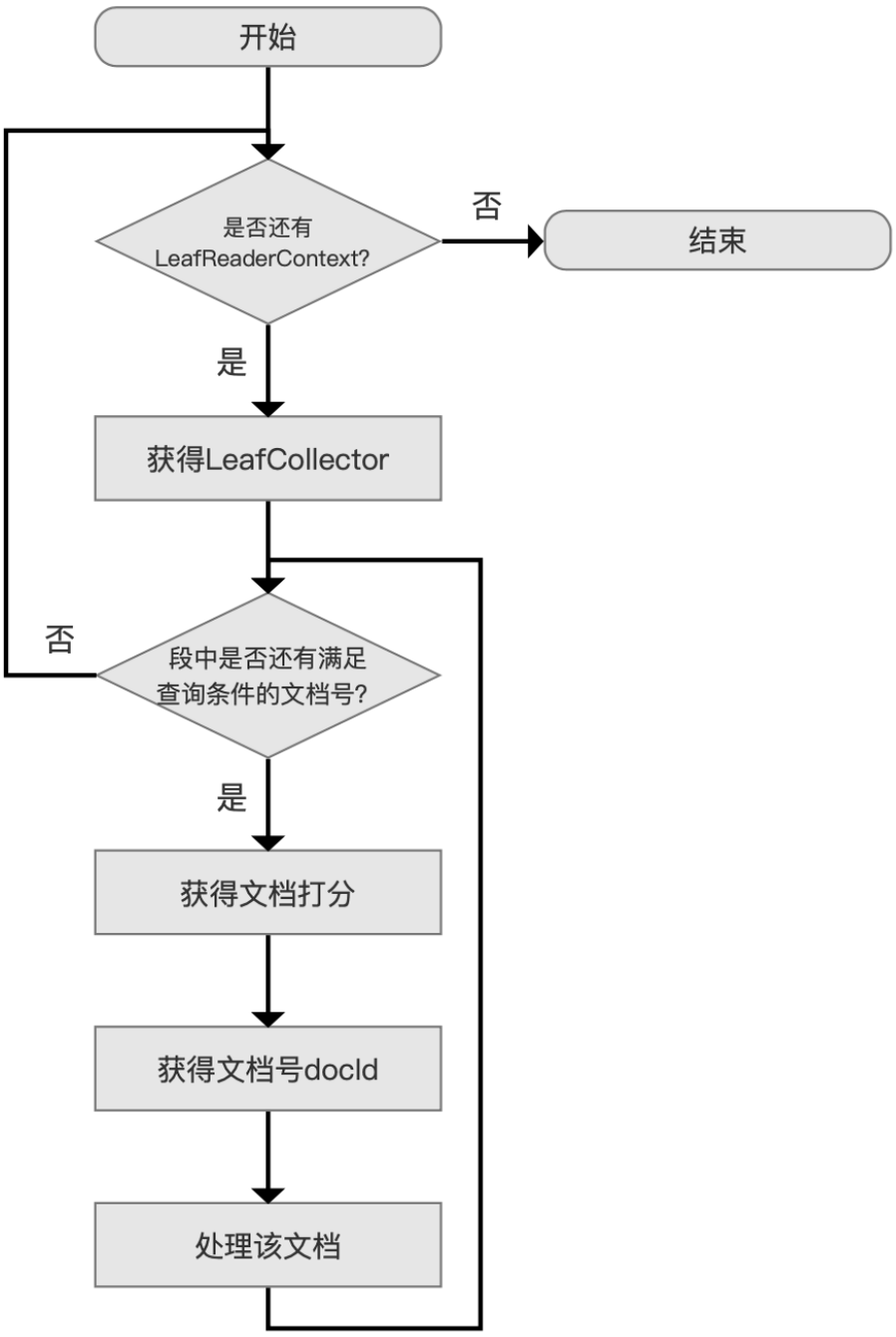
图1：



Collector处理文档

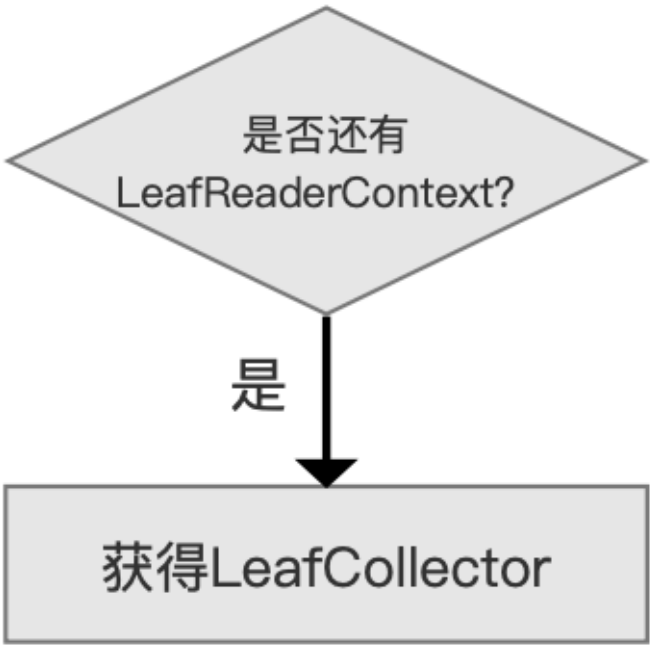
下图中描述的是Collector处理文档的流程：

图2：



获得LeafCollector

图3:



当索引目录中存在多个段时，我们需要从每个段中分别找出满足查询条件的文档，LeafReaderContext即用来描述某一个段的信息，并且通过它能获得一个LeafCollector对象，在本篇文章中我们只要知道LeafReaderContext有这个功能（functionality）即可，在后面介绍IndexReader的文章中会展开。

在搜索阶段，通过Collector类的方法来获得LeafCollector对象，下面是Collector类的代码，由于Collector类是一个接口类，并且只有两个接口方法，故列出并介绍：

```
public interface Collector {

    LeafCollector getLeafCollector(LeafReaderContext context) throws
IOException;

    boolean needsScores();

}
```

接口方法 getLeafCollector

通过该方法获得一个LeafCollector对象，Lucene每处理完一个段，就会调用该方法获得下一个段对应的LeafCollector对象。

LeafCollector对象有什么作用：

- 首先看下LeafCollector类的结构：

```
public interface LeafCollector {

    void setScorer(Scorer scorer) throws IOException;

    /** 参数doc即文档号docId*/
    void collect(int doc) throws IOException;
}
```

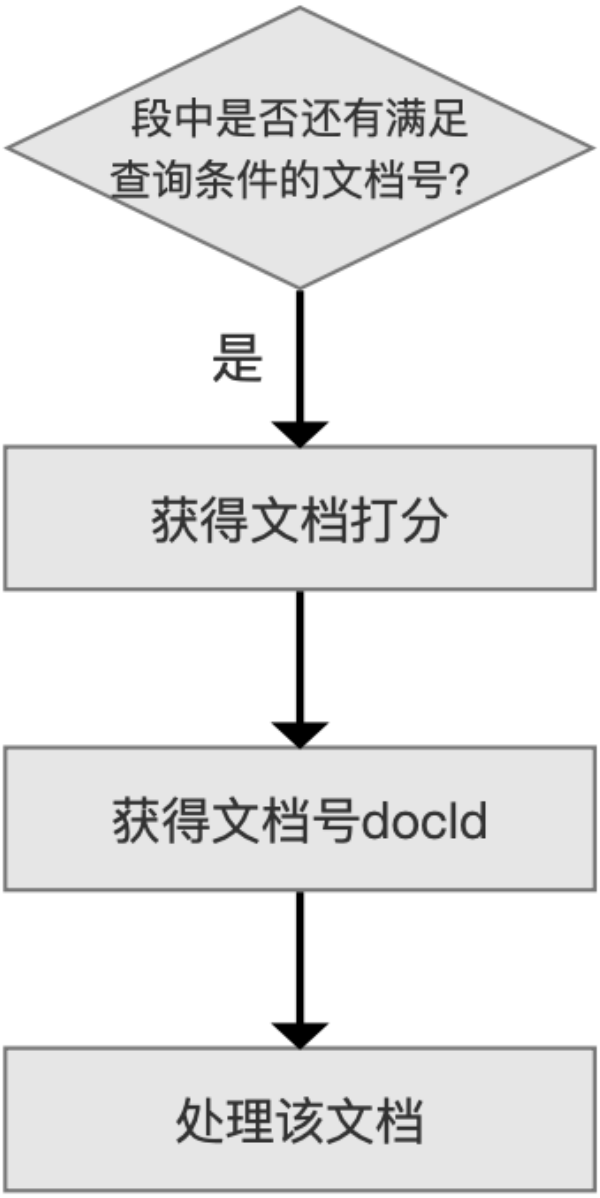
- setScorer方法：调用此方法通过Scorer对象获得一篇文档的打分，对文档集合进行排序时，可以作为排序条件之一，当然Scorer对象包含不仅仅是文档的打分值，在后面介绍查询的文章中会展开
- collect方法：在这个方法中实现了对所有满足查询条件的文档进行排序（sorting）、过滤（filtering）或者用户自定义的操作的具体逻辑。在下文中，根据图1中不同的收集器（Collector）会详细介绍collect方法的不同实现

接口方法 needsScores

设置该方法用来告知Lucene在搜索阶段，当找到一篇文档时，是否对其进行打分。如果用户期望的查询结果不依赖打分，那么可以设置为false来提高查询性能。

处理一篇文档

图4：



当Lucene找到一篇满足查询条件的文档，会调用LeafCollector的setScorer(Scorer score)方法来执行`获得文档打分`的流程，随后在`获得文档号docId`流程后获得一个docId，最后调用LeafCollector的collect(int doc)方法（参数doc即文档号docId）来实执行`处理该文档`的流程，在该流程中，实现对文档进行排序（sorting）、过滤（filtering）或者用户自定义的操作。

TimeLimitingCollector

在介绍完Collector处理文档的流程后，我们依次介绍图1中的收集器。

TimeLimitingCollector封装了其他的Collector，用来限制Collector处理文档的时间，即设定了一次查询允许的最长时间timeLimit。如果查询的时间超过timeLimit，那么会抛出超时异常TimeExceededException。

在哪些流程点会判断查询超时：

- 调用Collector.getLeafCollector(LeafReaderContext context)方法时会执行超时判断，即图3中的`是否还有 LeafReaderContext`的流程点
- 调用LeafCollector.collect(int doc)方法时会执行超时判断，即图4中的`处理该文档`的流程点

如何实现超时机制：

- 通过后台线程、解析值resolution、计数器counter实现、timeLimit
  - 计数器counter：AtomicLong类型，用来描述查询已花费的时间
  - 解析值resolution：long类型的数值，触发查询超时的精度值
  - 后台线程：Thread.setDaemon(true)的线程
  - timeLimit：上文已经介绍
- 后台线程先执行counter的累加操作，即调用counter.addAndGet(resolution)的方法，随后调用Thread.sleep(resolution)的方法，如此反复。收集文档号的线程在 判断查询超时的流程点处通过counter.get()的值判断是否大于timeLimit

使用这种超时机制有什么注意点：

- 由于后台线程先执行counter的累加操作，随后睡眠，故收集文档号的线程超时的时间范围为timeLimit - resolution 至 timeLimit + resolution 的区间，单位是milliseconds，故查询超时的触发时间不是精确的
- 由上一条可以知道，resolution的值设置的越小，查询超时的触发时间精度越高，但是性能越差（例如线程更频繁的睡眠唤醒等切换上下文行为）
- 由于使用了睡眠机制，在运行过程中实时将resolution的值被调整为比当前resolution较小的值时（比如由20milliseconds调整为5milliseconds），可能会存在调整延迟的问题（线程正好开始睡眠20milliseconds）
- resolution的值至少为5milliseconds，因为要保证能正确的调用执行Object.wait(long)方法

贪婪（greedy）模式：

在开启贪婪模式的情况下（默认不开启），如果在LeafCollector.collect( )中判断出查询超时，那么还是会收集当前的文档号并随后抛出超时异常，注意的是如果在Collector.getLeafCollector( )中判断出查询超时，那么直接抛出超时异常。

结语

剩余的Collector展开介绍会造成本篇文章篇幅过长，将在下一篇文章中展开。

点击下载附件