

02 | 日志（上）：日志究竟是如何加载日志段的？

2020-04-16 胡夕

Kafka核心源码解读

[进入课程 >](#)



讲述：胡夕

时长 14:27 大小 13.25M



你好，我是胡夕。今天我来讲讲 Kafka 源码的日志（Log）对象。

上节课，我们学习了日志段部分的源码，你可以认为，**日志是日志段的容器，里面定义了很多管理日志段的操作**。坦率地说，如果看 Kafka 源码却不看 Log，就跟你买了这门课却不知道作者是谁一样。在我看来，Log 对象是 Kafka 源码（特别是 Broker 端）最核心的部分，没有之一。

它到底有多重要呢？我和你分享一个例子，你先感受下。我最近正在修复一个 Kafka Bug（[☞ KAFKA-9157](#)）：在某些情况下，Kafka 的 Compaction 操作会产生很多空的日志段文件。如果要避免这些空日志段文件被创建出来，就必须搞懂创建日志段文件的原理，而这些代码恰恰就在 Log 源码中。

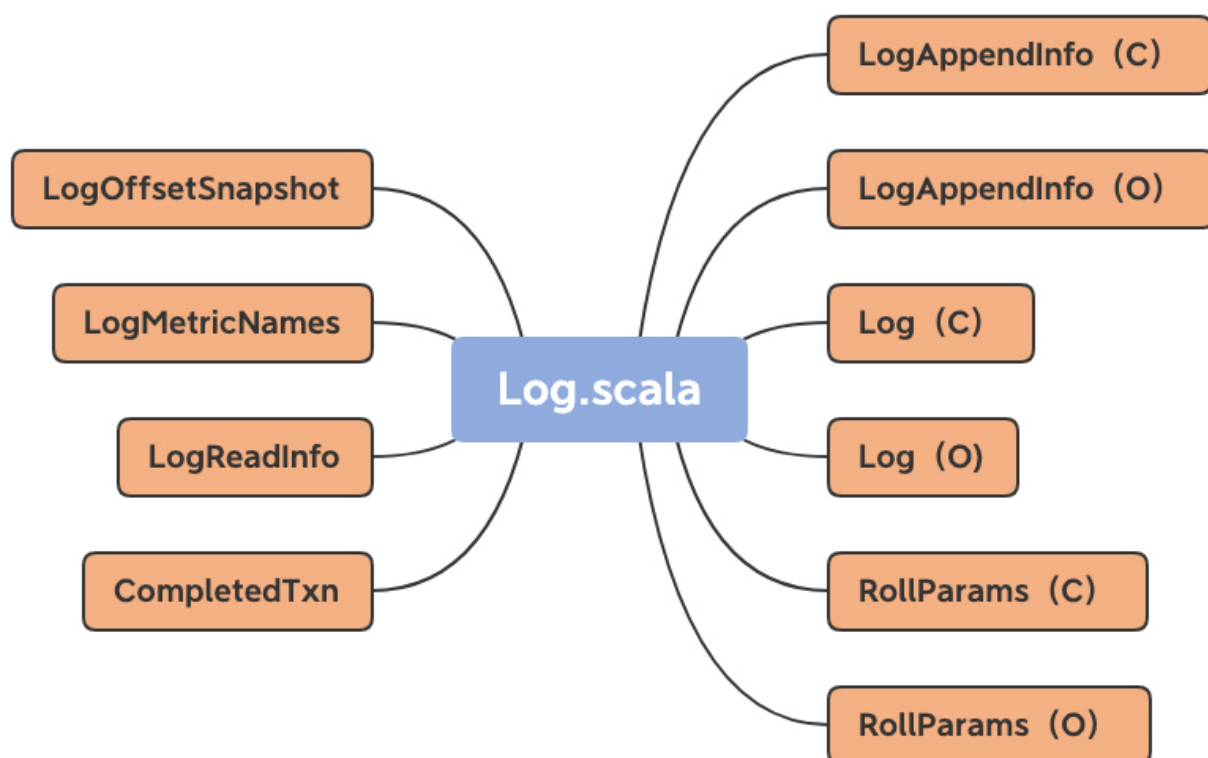


既然 Log 源码要管理日志段对象，那么它就必须先把所有日志段对象加载到内存里面。这个过程是怎么实现的呢？今天，我就带你学习下日志加载日志段的过程。

首先，我们来看下 Log 对象的源码结构。

Log 源码结构

Log 源码位于 Kafka core 工程的 log 源码包下，文件名是 Log.scala。总体上，该文件定义了 10 个类和对象，如下图所示：



那么，这 10 个类和对象都是做什么的呢？我先给你简单介绍一下，你可以对它们有个大致的了解。

不过，在介绍之前，我先提一句，图中括号里的 C 表示 Class，O 表示 Object。还记得我在上节课提到过的伴生对象吗？没错，同时定义同名的 Class 和 Object，就属于 Scala 中的伴生对象用法。

我们先来看伴生对象，也就是 LogAppendInfo、Log 和 RollParams。

1.LogAppendInfo

LogAppendInfo (C) : 保存了一组待写入消息的各种元数据信息。比如, 这组消息中第一条消息的位移值是多少、最后一条消息的位移值是多少; 再比如, 这组消息中最大的消息时间戳又是多少。总之, 这里面的数据非常丰富(下节课我再具体说说)。

LogAppendInfo (O) : 可以理解为其对应伴生类的工厂方法类, 里面定义了一些工厂方法, 用于创建特定的 LogAppendInfo 实例。

2.Log

Log (C) : Log 源码中最核心的代码。这里我先卖个关子, 一会儿细聊。

Log (O) : 同理, Log 伴生类的工厂方法, 定义了很多常量以及一些辅助方法。

3.RollParams

RollParams (C) : 定义用于控制日志段是否切分 (Roll) 的数据结构。

RollParams (O) : 同理, RollParams 伴生类的工厂方法。

除了这 3 组伴生对象之外, 还有 4 类源码。

LogMetricNames: 定义了 Log 对象的监控指标。

LogOffsetSnapshot: 封装分区所有位移元数据的容器类。

LogReadInfo: 封装读取日志返回的数据及其元数据。

CompletedTxn: 记录已完成事务的元数据, 主要用于构建事务索引。

Log Class & Object

下面, 我会按照这些类和对象的重要程度, 对它们一一进行拆解。首先, 咱们先说说 Log 类及其伴生对象。

考虑到伴生对象多用于保存静态变量和静态方法(比如静态工厂方法等), 因此我们先看伴生对象(即 Log Object)的实现。毕竟, 柿子先找软的捏!

```
1 object Log {  
2   val LogFileSuffix = ".log"  
3   val IndexFileSuffix = ".index"  
4   val TimeIndexFileSuffix = ".timeindex"  
5   val ProducerSnapshotFileSuffix = ".snapshot"  
6   val TxnIndexFileSuffix = ".txnindex"  
7   val DeletedFileSuffix = ".deleted"  
8   val CleanedFileSuffix = ".cleaned"  
9   val SwapFileSuffix = ".swap"  
10  val CleanShutdownFile = ".kafka_cleanshutdown"  
11  val DeleteDirSuffix = "-delete"  
12  val FutureDirSuffix = "-future"  
13  .....  
14 }
```

这是 Log Object 定义的所有常量。如果有面试官问你 Kafka 中定义了多少种文件类型，你可以自豪地把这些说出来。耳熟能详的.log、.index、.timeindex 和.txnindex 我就不解释了，我们来了解下其他几种文件类型。

.snapshot 是 Kafka 为幂等型或事务型 Producer 所做的快照文件。鉴于我们现在还处于阅读源码的初级阶段，事务或幂等部分的源码我就不详细展开讲了。

.deleted 是删除日志段操作创建的文件。目前删除日志段文件是异步操作，Broker 端把日志段文件从.log 后缀修改为.deleted 后缀。如果你看到一大堆.deleted 后缀的文件名，别慌，这是 Kafka 在执行日志段文件删除。

.cleaned 和.swap 都是 Compaction 操作的产物，等我们讲到 Cleaner 的时候再说。

-delete 则是应用于文件夹的。当你删除一个主题的时候，主题的分区文件夹会被加上这个后缀。

-future 是用于变更主题分区文件夹地址的，属于比较高阶的用法。

总之，记住这些常量吧。记住它们的主要作用是，以后不要被面试官唬住！开玩笑，其实这些常量最重要的地方就在于，它们能够让你了解 Kafka 定义的各种文件类型。

Log Object 还定义了超多的工具类方法。由于它们都很简单，这里我只给出一个方法的源码，我们一起读一下。

```
1 def filenamePrefixFromOffset(offset: Long): String = {  
2     val nf = NumberFormat.getInstance()  
3     nf.setMinimumIntegerDigits(20)  
4     nf.setMaximumFractionDigits(0)  
5     nf.setGroupingUsed(false)  
6     nf.format(offset)  
7 }
```

这个方法的作用是通过给定的位移值计算出对应的日志段文件名。Kafka 日志文件固定是 20 位的长度，filenamePrefixFromOffset 方法就是用前面补 0 的方式，把给定位移值扩充成一个固定 20 位长度的字符串。

举个例子，我们给定一个位移值是 12345，那么 Broker 端磁盘上对应的日志段文件名就应该是 000000000000000012345.log。怎么样，很简单吧？其他的工具类方法也很简单，我就不一一展开说了。

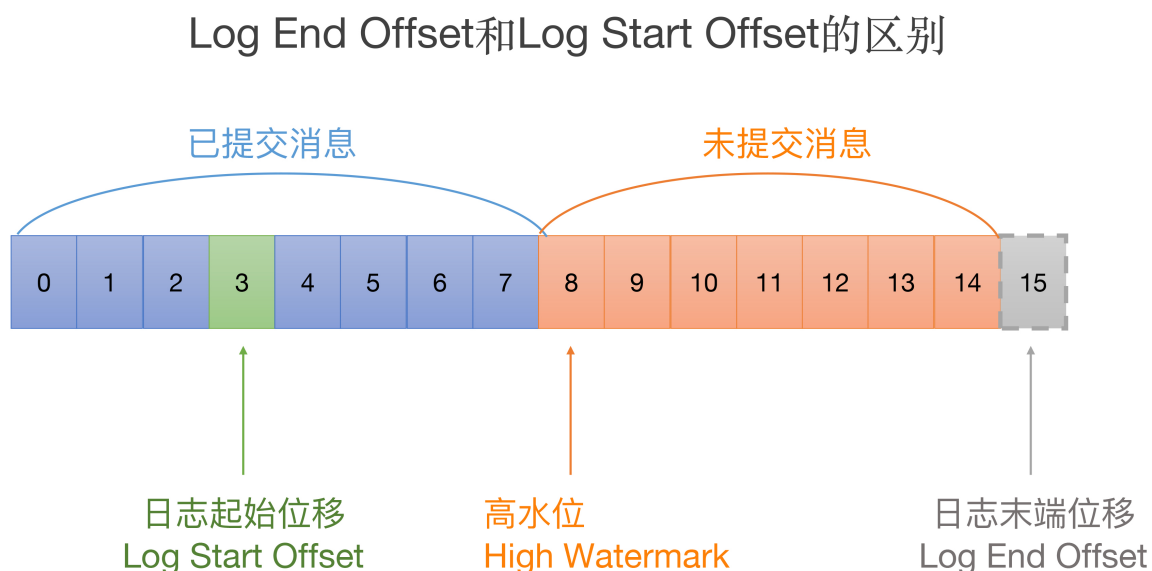
下面我们来看 Log 源码部分的重头戏：**Log 类**。这是一个 2000 多行的大类。放眼整个 Kafka 源码，像 Log 这么大的类也不多见，足见它的重要程度。我们先来看这个类的定义：

```
1 class Log(@volatile var dir: File,  
2           @volatile var config: LogConfig,  
3           @volatile var logStartOffset: Long,  
4           @volatile var recoveryPoint: Long,  
5           scheduler: Scheduler,  
6           brokerTopicStats: BrokerTopicStats,  
7           val time: Time,  
8           val maxProducerIdExpirationMs: Int,  
9           val producerIdExpirationCheckIntervalMs: Int,  
10          val topicPartition: TopicPartition,  
11          val producerStateManager: ProducerStateManager,  
12          logDirFailureChannel: LogDirFailureChannel) extends Logging with Kafk  
13 .....  
14 }
```

看着好像有很多属性，但其实，你只需要记住两个属性的作用就够了：**dir** 和 **logStartOffset**。dir 就是这个日志所在的文件夹路径，也就是**主题分区的路径**。而

logStartOffset, 表示**日志的当前最早位移**。dir 和 logStartOffset 都是 volatile var 类型, 表示它们的值是变动的, 而且可能被多个线程更新。

你可能听过日志的当前末端位移, 也就是 Log End Offset (LEO), 它是表示日志下一条待插入消息的位移值, 而这个 Log Start Offset 是跟它相反的, 它表示日志当前对外可见的最早一条消息的位移值。我用一张图来标识它们的区别:



图中绿色的位移值 3 是日志的 Log Start Offset, 而位移值 15 表示 LEO。另外, 位移值 8 是高水位值, 它是区分已提交消息和未提交消息的分水岭。

有意思的是, Log End Offset 可以简称为 LEO, 但 Log Start Offset 却不能简称为 LSO。因为在 Kafka 中, LSO 特指 Log Stable Offset, 属于 Kafka 事务的概念。这个课程中不会涉及 LSO, 你只需要知道 Log Start Offset 不等于 LSO 即可。

Log 类的其他属性你暂时不用理会, 因为它们要么是很明显的工具类属性, 比如 timer 和 scheduler, 要么是高阶用法才会用到的高级属性, 比如 producerStateManager 和 logDirFailureChannel。工具类的代码大多是做辅助用的, 跳过它们也不妨碍我们理解 Kafka 的核心功能; 而高阶功能代码设计复杂, 学习成本高, 性价比不高。

其实, 除了 Log 类签名定义的这些属性之外, Log 类还定义了一些很重要的属性, 比如下面这段代码:


```
1 @volatile private var nextOffsetMetadata: LogOffsetMetadata = _
2 @volatile private var highWatermarkMetadata: LogOffsetMetadata = LogOffsetM
3 private val segments: ConcurrentNavigableMap[java.lang.Long, LogSegment] =
4 @volatile var leaderEpochCache: Option[LeaderEpochFileCache] = None
```

第一个属性 `nextOffsetMetadata`，它封装了下一条待插入消息的位移值，你基本上可以把这个属性和 LEO 等同起来。

第二个属性 `highWatermarkMetadata`，是分区日志高水位值。关于高水位的概念，我们在 [《Kafka 核心技术与实战》](#) 这个课程中做过详细解释，你可以看一下 [这篇文章](#)（下节课我还会再具体给你介绍下）。

第三个属性 `segments`，我认为这是 `Log` 类中最重要的属性。它保存了分区日志下所有的日志段信息，只不过是用 `Map` 的数据结构来保存的。`Map` 的 `Key` 值是日志段的起始位移值，`Value` 则是日志段对象本身。Kafka 源码使用 `ConcurrentNavigableMap` 数据结构来保存日志段对象，就可以很轻松地利用该类提供的线程安全和各种支持排序的方法，来管理所有日志段对象。

第四个属性是 `Leader Epoch Cache` 对象。`Leader Epoch` 是社区于 0.11.0.0 版本引入源码中的，主要是用来判断出现 `Failure` 时是否执行日志截断操作（`Truncation`）。之前靠高水位来判断的机制，可能会造成副本间数据不一致的情形。这里的 `Leader Epoch Cache` 是一个缓存类数据，里面保存了分区 `Leader` 的 `Epoch` 值与对应位移值的映射关系，我建议你查看下 `LeaderEpochFileCache` 类，深入地了解下它的实现原理。

掌握了这些基本属性之后，我们看下 `Log` 类的初始化逻辑：

```
1 locally {
2     val startMs = time.milliseconds
3
4
5     // create the log directory if it doesn't exist
6     Files.createDirectories(dir.toPath)
7
8
9     initializeLeaderEpochCache()
10 }
```

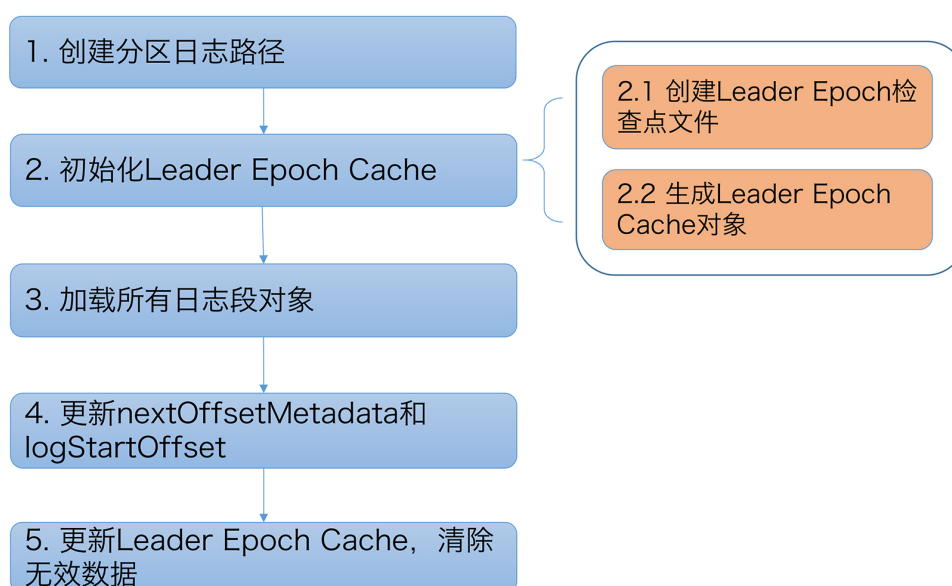
```

11
12     val nextOffset = loadSegments()
13
14
15     /* Calculate the offset of the next message */
16     nextOffsetMetadata = LogOffsetMetadata(nextOffset, activeSegment.baseOffset)
17
18
19     leaderEpochCache.foreach(_._truncateFromEnd(nextOffsetMetadata.messageOffset))
20
21
22     logStartOffset = math.max(logStartOffset, segments.firstEntry.getValue().logStartOffset)
23
24
25     // The earliest leader epoch may not be flushed during a hard failure.
26     leaderEpochCache.foreach(_._truncateFromStart(logStartOffset))
27
28
29     // Any segment loading or recovery code must not use producerStateManager
30     // from scratch.
31     if (!producerStateManager.isEmpty)
32         throw new IllegalStateException("Producer state must be empty during
33         loadProducerState(logEndOffset, reloadFromCleanShutdown = hasCleanShutdown)
34
35
36     info(s"Completed load of log with ${segments.size} segments, log start
37         s"log end offset $logEndOffset in ${time.milliseconds() - startMs}")


```

在详细解释这段初始化代码之前，我使用一张图来说明它到底做了什么：

Log类的初始化逻辑



这里我们重点说说第三步，即加载日志段的实现逻辑，以下是 loadSegments 的实现代码：

 复制代码

```
1  private def loadSegments(): Long = {
2      // first do a pass through the files in the log directory and remove all
3      // and find any interrupted swap operations
4      val swapFiles = removeTempFilesAndCollectSwapFiles()
5
6
7      // Now do a second pass and load all the log and index files.
8      // We might encounter legacy log segments with offset overflow (KAFKA-111)
9      // this happens, restart loading segment files from scratch.
10     retryOnOffsetOverflow {
11         // In case we encounter a segment with offset overflow, the retry logic
12         // loading of segments. In that case, we also need to close all segments
13         // call to loadSegmentFiles().
14         logSegments.foreach(_.close())
15         segments.clear()
16         loadSegmentFiles()
17     }
18
19
20     // Finally, complete any interrupted swap operations. To be crash-safe
21     // log files that are replaced by the swap segment should be renamed to
22     // before the swap file is restored as the new segment file.
23     completeSwapOperations(swapFiles)
24
25
26     if (!dir.getAbsolutePath.endsWith(Log.DeleteDirSuffix)) {
27         val nextOffset = retryOnOffsetOverflow {
28             recoverLog()
29         }
30
31
32         // reset the index size of the currently active log segment to allow
33         // activeSegment.resizeIndexes(config.maxIndexSize)
34         nextOffset
35     } else {
36         if (logSegments.isEmpty) {
37             addSegment(LogSegment.open(dir = dir,
38                 baseOffset = 0,
39                 config,
40                 time = time,
41                 fileAlreadyExists = false,
42                 initFileSize = this.initFileSize,
43                 preallocate = false))
44         }
45         0
46     }
```

这段代码会对分区日志路径遍历两次。

首先，它会移除上次 Failure 遗留下来的各种临时文件（包括.cleaned、.swap、.deleted 文件等），removeTempFilesAndCollectSwapFiles 方法实现了这个逻辑。

之后，它会清空所有日志段对象，并且再次遍历分区路径，重建日志段 segments Map 以及索引文件。

待执行完这两次遍历之后，它会完成未完成的 swap 操作，即调用 completeSwapOperations 方法。等这些都做完之后，再调用 recoverLog 方法恢复日志段对象，然后返回恢复之后的分区日志 LEO 值。

如果你现在觉得有点蒙，也没关系，我把这段代码再进一步拆解下，以更小的粒度跟你讲下它们做了什么。理解了这段代码之后，你大致就能搞清楚大部分的分区日志操作了。所以，这部分代码绝对值得我们多花一点时间去学习。

我们首先来看第一步，removeTempFilesAndCollectSwapFiles 方法的实现。我用注释的方式详细解释了每行代码的作用：

 复制代码

```
1  private def removeTempFilesAndCollectSwapFiles(): Set[File] = {
2
3      // 在方法内部定义一个名为deleteIndicesIfExist的方法，用于删除日志文件对应的索引文件
4
5      def deleteIndicesIfExist(baseFile: File, suffix: String = ""): Unit = {
6
7          info(s"Deleting index files with suffix $suffix for baseFile $baseFile")
8
9          val offset = offsetFromFile(baseFile)
10
11         Files.deleteIfExists(Log.offsetIndexFile(dir, offset, suffix).toPath)
12
13         Files.deleteIfExists(Log.timeIndexFile(dir, offset, suffix).toPath)
14
15         Files.deleteIfExists(Log.transactionIndexFile(dir, offset, suffix).toPath)
16
17     }
18 }
```

```
19     var swapFiles = Set[File]()
20
21     var cleanFiles = Set[File]()
22
23     var minCleanedFileOffset = Long.MaxValue
24
25     // 遍历分区日志路径下的所有文件
26
27     for (file <- dir.listFiles if file.isFile) {
28
29         if (!file.canRead) // 如果不可读, 直接抛出IOException
30
31             throw new IOException(s"Could not read file $file")
32
33         val filename = file.getName
34
35         if (filename.endsWith(DeletedFileSuffix)) { // 如果以.deleted结尾
36
37             debug(s"Deleting stray temporary file ${file.getAbsolutePath}")
38
39             Files.deleteIfExists(file.toPath) // 说明是上次Failure遗留下来的文件, 直接删除
40
41         } else if (filename.endsWith(CleanedFileSuffix)) { // 如果以.cleaned结尾
42
43             minCleanedFileOffset = Math.min(offsetFromFileName(filename), minCleanedFileOffset)
44
45             cleanFiles += file
46
47         } else if (filename.endsWith(SwapFileSuffix)) { // 如果以.swap结尾
48
49             val baseFile = new File(CoreUtils.replaceSuffix(file.getPath, SwapFileSuffix, ""))
50
51             info(s"Found file ${file.getAbsolutePath} from interrupted swap operation.")
52
53             if (isIndexFile(baseFile)) { // 如果该.swap文件原来是索引文件
54
55                 deleteIndicesIfExist(baseFile) // 删除原来的索引文件
56
57             } else if (isLogFile(baseFile)) { // 如果该.swap文件原来是日志文件
58
59                 deleteIndicesIfExist(baseFile) // 删除掉原来的索引文件
60
61                 swapFiles += file // 加入待恢复的.swap文件集合中
62
63             }
64
65         }
66
67     }
68
69     // 从待恢复swap集合中找出那些起始位移值大于minCleanedFileOffset值的文件, 直接删掉这些文件
```

```

71     val (invalidSwapFiles, validSwapFiles) = swapFiles.partition(file => offset < 0)
72
73     invalidSwapFiles.foreach { file =>
74
75         debug(s"Deleting invalid swap file ${file.getAbsolutePath} minCleanedFileOffset: $minCleanedFileOffset")
76
77         val baseFile = new File(CoreUtils.replaceSuffix(file.getPath, SwapFileSuffix, ""))
78
79         deleteIndicesIfExists(baseFile, SwapFileSuffix)
80
81         Files.deleteIfExists(file.toPath)
82
83     }
84
85     // Now that we have deleted all .swap files that constitute an incomplete segment, we can clean up stray files
86
87     // 清除所有待删除文件集合中的文件
88
89     cleanFiles.foreach { file =>
90
91         debug(s"Deleting stray .clean file ${file.getAbsolutePath}")
92
93         Files.deleteIfExists(file.toPath)
94
95     }
96
97     // 最后返回当前有效的.swap文件集合
98
99     validSwapFiles
100
101 }

```

执行完了 `removeTempFilesAndCollectSwapFiles` 逻辑之后，源码开始清空已有日志段集合，并重新加载日志段文件。这就是第二步。这里调用的主要方法是 `loadSegmentFiles`。

 复制代码

```


1     private def loadSegmentFiles(): Unit = {
2
3         // 按照日志段文件名中的位移值正序排列，然后遍历每个文件
4
5         for (file <- dir.listFiles.sortBy(_.getName) if file.isFile) {
6
7             if (isIndexFile(file)) { // 如果是索引文件
8
9                 val offset = offsetFromFile(file)
10
11                 val logFile = Log.logFile(dir, offset)
12

```

```
13     if (!logFile.exists) { // 确保存在对应的日志文件，否则记录一个警告，并删除该索引文件
14
15     warn(s"Found an orphaned index file ${file.getAbsolutePath}, with no corre:
16
17     Files.deleteIfExists(file.toPath)
18
19     }
20
21     } else if (isLogFile(file)) { // 如果是日志文件
22
23     val baseOffset = offsetFromFile(file)
24
25     val timeIndexFileNewlyCreated = !Log.timeIndexFile(dir, baseOffset).exists
26
27     // 创建对应的LogSegment对象实例，并加入segments中
28
29     val segment = LogSegment.open(dir = dir,
30
31     baseOffset = baseOffset,
32
33     config,
34
35     time = time,
36
37     fileAlreadyExists = true)
38
39     try segment.sanityCheck(timeIndexFileNewlyCreated)
40
41     catch {
42
43     case _: NoSuchFileException =>
44
45     error(s"Could not find offset index file corresponding to log file ${segmei
46
47     "recovering segment and rebuilding index files...")
48
49     recoverSegment(segment)
50
51     case e: CorruptIndexException =>
52
53     warn(s"Found a corrupted index file corresponding to log file ${segment.lo
54
55     s"to ${e.getMessage}}, recovering segment and rebuilding index files...")
56
57     recoverSegment(segment)
58
59     }
60
61     addSegment(segment)
62
63     }
64
```

```
65     }
66
67     }
68
```

第三步是处理第一步返回的有效.swap 文件集合。completeSwapOperations 方法就是做这件事的：

 复制代码

```
1  private def completeSwapOperations(swapFiles: Set[File]): Unit = {
2
3      // 遍历所有有效.swap文件
4
5      for (swapFile <- swapFiles) {
6
7          val logFile = new File(CoreUtils.replaceSuffix(swapFile.getPath, SwapFileSi
8
9          val baseOffset = offsetFromFile(logFile) // 拿到日志文件的起始位移值
10
11         // 创建对应的LogSegment实例
12
13         val swapSegment = LogSegment.open(swapFile.getParentFile,
14
15         baseOffset = baseOffset,
16
17         config,
18
19         time = time,
20
21         fileSuffix = SwapFileSuffix)
22
23         info(s"Found log file ${swapFile.getPath} from interrupted swap operation,
24
25         // 执行日志段恢复操作
26
27         recoverSegment(swapSegment)
28
29         // We create swap files for two cases:
30
31         // (1) Log cleaning where multiple segments are merged into one, and
32
33         // (2) Log splitting where one segment is split into multiple.
34
35         //
36
37         // Both of these mean that the resultant swap segments be composed of the
38
39         // must fall within the range of existing segment(s). If we cannot find su
```



```

24         }
25         if (truncatedBytes > 0) { // 如果有无效的消息导致被截断的字节数不为0, 直接
26             warn(s"Corruption found in segment ${segment.baseOffset}, truncat
27             removeAndDeleteSegments(unflushed.toList, asyncDelete = true)
28             truncated = true
29         }
30     }
31 }
32
33
34 // 这些都做完之后, 如果日志段集合不为空
35 if (logSegments.nonEmpty) {
36     val logEndOffset = activeSegment.readNextOffset
37     if (logEndOffset < logStartOffset) { // 验证分区日志的LEO值不能小于Log S
38         warn(s"Deleting all segments because logEndOffset ($logEndOffset) >
39             "This could happen if segment files were deleted from the file s
40             removeAndDeleteSegments(logSegments, asyncDelete = true)
41     }
42 }
43
44
45 // 这些都做完之后, 如果日志段集合为空了
46 if (logSegments.isEmpty) {
47     // 至少创建一个新的日志段, 以logStartOffset为日志段的起始位移, 并加入日志段集合中
48     addSegment(LogSegment.open(dir = dir,
49         baseOffset = logStartOffset,
50         config,
51         time = time,
52         fileAlreadyExists = false,
53         initFileSize = this.initFileSize,
54         preallocate = config.preallocate))
55 }
56
57
58 // 更新上次恢复点属性, 并返回
59 recoveryPoint = activeSegment.readNextOffset
60 recoveryPoint

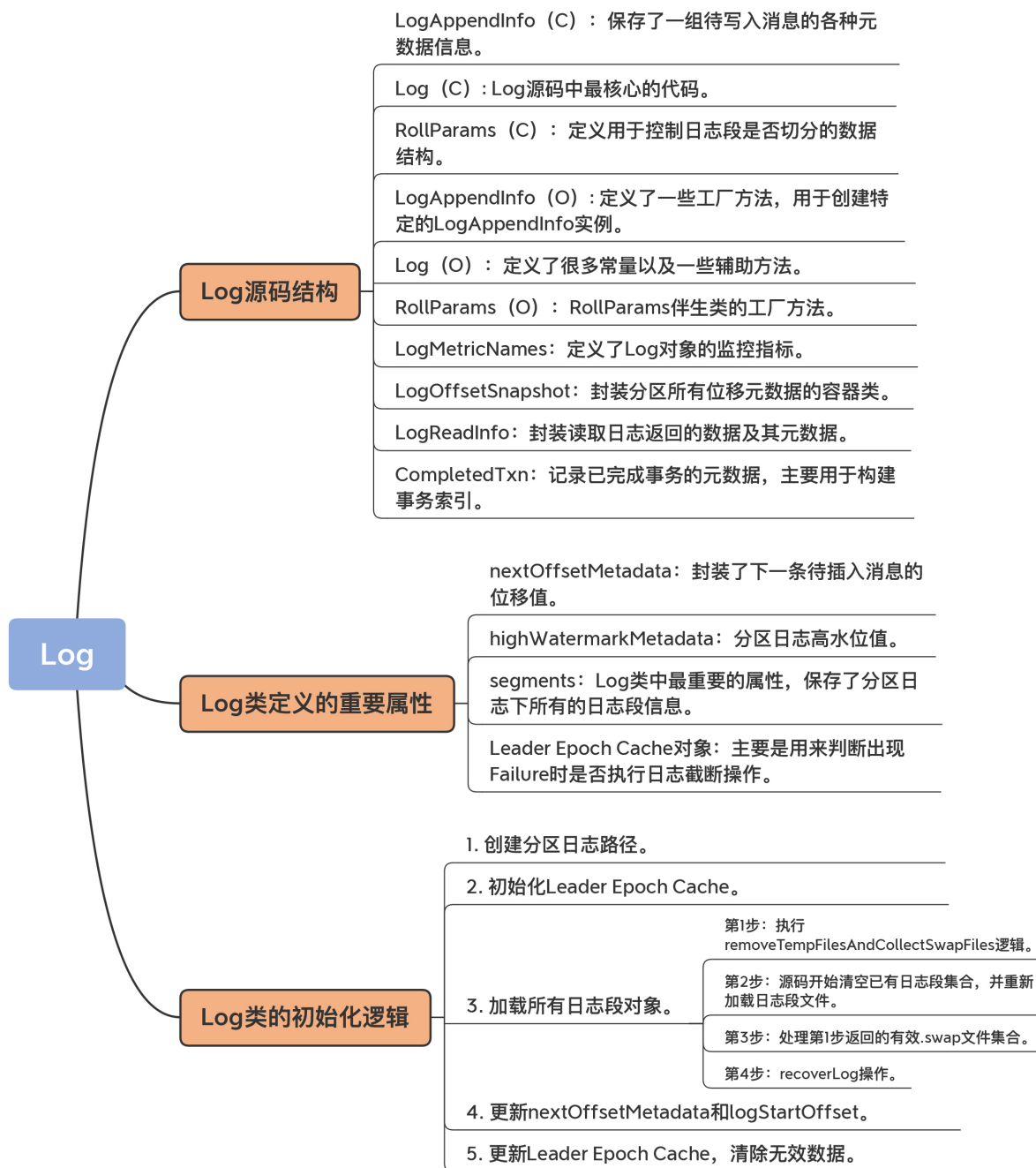
```

总结

今天, 我重点向你介绍了 Kafka 的 Log 源码, 主要包括:

1. **Log 文件的源码结构**: 你可以看下下面的导图, 它展示了 Log 类文件的架构组成, 你要重点掌握 Log 类及其相关方法。
2. **加载日志段机制**: 我结合源码重点分析了日志在初始化时是如何加载日志段的。前面说过了, 日志是日志段的容器, 弄明白如何加载日志段是后续学习日志段管理的前提条

件。



总的来说, 虽然洋洋洒洒几千字, 但我也只讲了最重要的部分。我建议你多看几遍 Log.scala 中加载日志段的代码, 这对后面我们理解 Kafka Broker 端日志段管理原理大有裨益。在下节课, 我会继续讨论日志部分的源码, 带你学习常见的 Kafka 日志操作。

课后讨论

Log 源码中有个 maybeIncrementHighWatermark 方法，你能说说它的实现原理吗？

欢迎你在留言区畅所欲言，跟我交流讨论，也欢迎你把今天的内容分享给你的朋友。

Kafka 核心源码解读

从底层到实战，深度解析源码

胡夕

友信金服商业智能部总监

Apache Kafka Contributor



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 01 | 日志段：保存消息文件的对象是怎么实现的？

下一篇 03 | 日志（下）：彻底搞懂Log对象的常见操作

精选留言 (8)

写留言



曾轶麟 置顶

2020-04-18

先回答老师的问题maybeIncrementHighWatermark的实现：

【首先需要注意以下几个内容】：

1、这个方法是通过leaderLog这个实例去调用的，当HW更新的时候follower就会更新自身的HW。

...

展开 v

作者回复: 🍵🍵🍵

1

1



胡夕 置顶

2020-04-23

你好，我是胡夕。我来公布上节课的“课后讨论”题答案啦~

上节课，咱们重点了解了日志段对象，课后我让你思考下如果给定位移值过大truncateTo方法的实现。关于这个问题，我的看法很简单。如果truncateTo的输入offset过大以至于超过了该日志段当前最大的消息位移值，那么这个方法不会执行任何截断操作，因为不...

展开

1

1



曾轶麟

2020-04-18

老师下面我想问一下我的一些问题：

1、为什么要遍历两次文件路径呢？我看了一下，如果在删除的时候顺便去加载segment会有什么问题吗？这样是否可以提高加载效率呢？

2、我看了一下在removeTempFilesAndCollectSwapFiles方法中minCleanedFileOffset是从文件名filename上面读取的，如果我修改了文件名的offset大小会出现什么意想不到...

展开

作者回复: 1. 就我个人而言，我觉得也没有什么问题。我觉得作者更多是为了把不同逻辑进行了分组导致遍历多次

2. 可能造成Broker的崩溃，无法启动。因为我们公司有小伙伴这么干过：（

3. 对的，这样可以快速根据给定offset找到对应的一上一下日志段对象

1

1



吃饭饭

2020-04-21

文中【图中绿色的位移值 3 是日志的 Log Start Offset】，这里不明白，为什么 logStart Offset 不等于 baseOffset = 0？

展开

作者回复: log start offset可以不等于baseOffset。事实上，这两个位移值没有直接的关联。图中仅仅是一个示例

1

1



一步

2020-04-19

这个 一个Log 对象 就相当与 一个日志分区文件夹吗？就是分区文件夹下的所有日志段对象的集合

作者回复: 是的:



我是小队长

2020-04-17

// 这些都做完之后，如果日志段集合不为空

// 验证分区日志的LEO值不能小于Log Start Offset值，否则删除这些日志段对象

想问下什么时候才会出现这种情况呢？

展开 ∨

作者回复: 当底层日志文件被删除或损坏的话就可能出现这种情况，因为无法读取文件去获取LEO了。你可以用2.0版本做个试验：

1. 发消息到分区日志
2. 使用Admin的DeleteRecords命令驱动Log start offset前进
3. 关闭Broker
4. 删除日志路径
5. 重启Broker



我是小队长

2020-04-17

老师，找出需要恢复的swap，直接恢复完成不就行了么，为什么还有下面的操作呢？

// We create swap files for two cases:

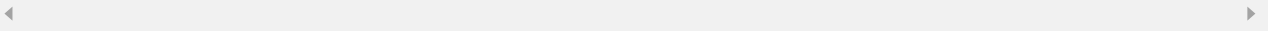
// (1) Log cleaning where multiple segments are merged into one, and...

展开 ∨

作者回复: 首先，不管是否要过滤出符合条件的oldSegments，回复之后都要进行替换，这个你实际上这是因为升级Broker版本而做的防御性编程。在老的版本中，代码写入消息后并不会对位移值进行校验。因此log cleaner老代码可能写入一个非常大的位移值（大于Int.MAX_VALUE）。当

broker升级后，这些日志段就不能正常被compact了，因为位移值越界了（新版本加入了位移校验）

代码需要去搜寻在swap start offset和swap LEO之间的所有日志段对象，但这还不够，还要保证这些日志段的LEO比swap的base offset大才行是同意的吧？否则



北纬8°C

北纬8°C

2020-04-16

太难了😓

展开 ▾

作者回复: 哈哈，坚持坚持！

