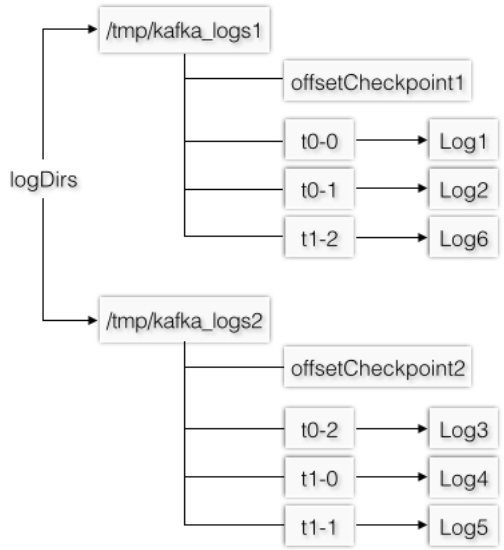
Kafka技术内幕样章：Kafka的日志压缩（LogCompaction）

3.3 日志管理类的后台线程

分布式存储系统除了要保证客户端写请求流程的正确性，节点可能会非正常宕机或者需要重启，在启动的时候必须要能够正常地加载/恢复已有的数据，日志管理类在创建的时候要加载已有的所有日志文件，这和创建Log时要加载所有的Segment是类似的。 LogManager 的 logDirs 参数对应了 log.dirs 配置项，每个TopicPartition文件夹都对应一个Log实例，所有的Partition文件夹都在日志目录下，当成功加载完所有的Log实例后logs才可以被日志管理类真正地用在战场上。

假设logDirs= /tmp/kafka_logs1,/tmp/kafka_logs2 ，logs1下有[t0-0,t0-1,t1-2]，logs2下有[t0-2,t1-0,t1-1]，图3-26的logDir指的是Log对象的dir，和log.dirs是不同的概念，可以认为所有Log的dir都是在每个log.dirs下，如果把Log.dir叫做Partition级别的文件夹，则checkpoint文件和Partition文件夹是同一层级。



logs

dir	Partition	Log	logDir
kafka_logs1	to-0	Log1	t0-0
kafka_logs1	to-1	Log2	t0-1
kafka_logs2	to-2	Log3	t0-2
kafka_logs2	t1-0	Log4	t1-0
kafka_logs2	t1-1	Log5	t1-1
kafka_logs1	t1-2	Log6	t1-2

log.dirs每个目录下的每个Partition对应一个Log

logsByDir

dir	logs
/tmp/kafka_logs1	[Log1,Log2,Log6]
/tmp/kafka_logs2	[Log3,Log4,Log5]

recoveryPointCheckpoints=[kafka_logs1,kafka_logs2]

图3-26 日志的组织方式和对应的数据结构

```
class LogManager(val logDirs: Array[File]){
  val logs = new Pool[TopicAndPartition, Log]()
  val recoveryPointCheckpoints=logDirs.map((_,new OffsetCheckpoint(new File(_,"checkpoint"))))

  loadLogs() //启动LogManager实例时,如果已经存在日志文件，要把它们加载到内存中

  private def loadLogs(): Unit = {
    val threadPools = mutable.ArrayBuffer.empty[ExecutorService]
    for (dir <- this.logDirs) { //按照log.dirs创建线程池，如果只配置一个目录就只有一个线程池
      val pool = Executors.newFixedThreadPool(ioThreads)
      threadPools.append(pool)
      //checkpoint文件一个日志目录只有一个，并不是每个Partition级别!
      //既然所有Partition公共一个checkpoint文件，那么文件内容当然要有Partition信息
      var recoveryPoints:Map[TopicAndPartition,Long]=recoveryPointCheckpoints(dir).read
      val jobsForDir = for {
        dirContent <- Option(dir.listFiles).toList //日志目录下的所有文件/文件夹
        logDir <- dirContent if logDir.isDirectory //Partition文件夹,忽略日志目录下的文件
      } yield {
        CoreUtils.runnable { //每个Partition文件夹创建一个线程，由线程池执行
          val topicPartition = Log.parseTopicPartitionName(logDir)
          val config = topicConfigs.getOrElse(topicPartition.topic, defaultConfig)
          val logRecoveryPoint = recoveryPoints.getOrElse(topicPartition, 0L) //分区的恢复点
          val current = new Log(logDir, config, logRecoveryPoint, scheduler, time) //恢复Log
          this.logs.put(topicPartition, current) //这里放入logs集合中，所有分区的Log满血复活
        }
      }
      jobsForDir.map(pool.submit).toSeq //提交任务
    }
  }

  //只有调用loadLogs后，logs才有值，后面的操作都依赖于logs
  def allLogs(): Iterable[Log] = logs.values
  def logsByDir = logs.groupBy{case (_,log)=>log.dir.getParent}
  val cleaner: LogCleaner = new LogCleaner(cleanerConfig,logDirs,logs)

  def startup() {
    scheduler.schedule("log-retention", cleanupLogs)
    scheduler.schedule("log-flusher", flushDirtyLogs)
    scheduler.schedule("recovery-point-checkpoint",checkpointRecoveryPointOffsets)
    if(cleanerConfig.enableCleaner) cleaner.startup()
  }
}
```

LogManager.startup()启动后会在后台运行多个定时任务和线程，表3-7列举了这些线程的方法和用途，这些线程最后都会操作Log实例（幸好我们已经成功地加载了logs），毕竟LogManager从名字来看就是要对Log进行管理（把checkpoint也看做是日志文件的一部分，因为它伴随着日志而生，所以也在LogManager的管理范畴内）。

线程/任务	方法	作用
日志保留任务（log retention）	cleanupLogs	删除失效的Segment或者为了控制日志文件大小要删除一些文件
日志刷写任务（log flusher）	flushDirtyLogs	根据时间策略，将还在操作系统缓存层的文件刷写到磁盘上
检查点刷写任务（checkpoint）	checkpointRecoveryPointOffsets	定时地将checkpoint恢复点状态写到文件中
日志清理线程（cleaner）	cleaner.startup()	日志压缩，针对带有key的消息的清理策略

表3-7 日志管理类的后台线程

日志文件和checkpoint的刷写 flush 都只是将当前最新的数据写到磁盘上。checkpoint检查点也叫做恢复点（顾名思义是从指定的点开始恢复数据），log.dirs的每个目录下只有一个所有Partition共享的全局checkpoint文件。

```
//日志文件刷写任务
private def flushDirtyLogs() = {
  for ((topicAndPartition, log) <- logs) {
    val timeSinceLastFlush = time.milliseconds - log.lastFlushTime
    if(timeSinceLastFlush >= log.config.flushMs) log.flush
  }
}
//checkpoint文件刷写任务
def checkpointRecoveryPointOffsets() {
  this.logDirs.foreach(checkpointLogsInDir)
}
private def checkpointLogsInDir(dir: File): Unit = {
  val recoveryPoints = this.logsByDir.get(dir.toString) //checkpoint是log.dirs目录级别
  //logsByDir对于每个dir都有多个Partition对应的Log，所以mapValues对每个Log获取recoveryPoint
  recoveryPointCheckpoints(dir).write(recoveryPoints.get.mapValues(_.recoveryPoint))
}

//只有flush的时候才会更新恢复点，不过flush并不是每次写都会发生的
def flush(offset: Long) : Unit = {
  if (offset <= this.recoveryPoint) return
  for(segment<-logSegments(this.recoveryPoint,offset)) //选择恢复点和当前之间的Segment
    segment.flush() //会分别刷写Log数据文件和index索引文件（调用底层的fsync）
  if(offset > this.recoveryPoint) {
    this.recoveryPoint = offset //recoveryPoint实际上是offset
    lastflushedTime.set(time.milliseconds)
  }
}
```

为什么所有Partition共用一个checkpoint文件，而不是每个Partition都有自己的checkpoint文件，因为checkpoint数据量不是很大，那么为什么前面分析的索引文件则是以Partition级别，甚至每个Segment都有对应的数据文件和索引文件，索引本身也是offset，它和checkpoint数据量也都是不大的啊，那么是不是也可以每个Partition只有一个索引文件，而不是每个Segment一个索引文件，实际上索引文件的用途是为了更快地查询，该省的地方还是要节约资源（所有Partition只有一个checkpoint文件），不该节省的还是要大方点（每个Segment一个索引文件），做人何尝不是这个道理。

3.3.1 日志清理

清理日志实际上是清理过期的Segment，或者日志文件太大了需要删除最旧的数据，使得整体的日志文件大小不超过指定的值。举例用队列来缓存所有的请求任务，每个任务都有一定的存活时间，超过时间后任务就应该自动被删除掉，同时队列也有一个上限，不能无限制地添加任务，如果超过指定大小时，就要把最旧的任务删除掉，以维持队列的固定大小，这样可以保证队列不至于无限大导致系统资源被耗尽。

```
//日志清理任务
def cleanupLogs() {
  for(log <- allLogs; if !log.config.compact)
    cleanupExpiredSegments(log) + cleanupSegmentsToMaintainSize(log)
}
private def cleanupExpiredSegments(log: Log): Int = {
  log.deleteOldSegments(time.milliseconds-_.lastModified>log.config.retentionMs)
}
private def cleanupSegmentsToMaintainSize(log: Log): Int = {
  var diff = log.size - log.config.retentionSize
  def shouldDelete(segment: LogSegment) = {
    if(diff - segment.size >= 0) {
      diff -= segment.size
      true
    } else false
  }
  log.deleteOldSegments(shouldDelete)
}
```

Log的deleteOldSegments方法接收一个高阶函数，参数是Segment，返回布尔类型表示这个Segment是否需要被删除，在LogManager中调用的地方并没有传递Segment，而是在Log中获取每个Segment。这是因为LogManager无法跨过Log直接和Segment通信，LogManager无法直接管理Segment，Segment只属于Log，只能由Log管理。

```
def deleteOldSegments(predicate: LogSegment => Boolean): Int = {
  //logSegments是Log的所有Segment，s是每个Segment
  val deletable = logSegments.takeWhile(s => predicate(s))
  if(segments.size == numToDelete) roll()
  deletable.foreach(deleteSegment(_))
}
private def deleteSegment(segment: LogSegment) {
  segments.remove(segment.baseOffset) //删除数据结构
  asyncDeleteSegment(segment) //异步删除Segment
}
private def asyncDeleteSegment(segment: LogSegment) {
  segment.changeFileSuffixes("", Log.DeletedFileSuffix)
  def deleteSeg() = segment.delete() //和flush一样最后调用log和index.delete
  scheduler.schedule("delete-file", deleteSeg)
}
```

清理日志有两种策略，一种是上面的cleanupLogs根据时间或大小策略（粗粒度），还有一种是针对每个key的日志删除策略（细粒度）即LogCleaner方式，如果消息没有key，那只能采用第一种清理策略了。删除策略是以topic为级别的，所以不同的topic可以设置不同的删除策略，所以一个集群中可能存在有些topic按照粗粒度模式删除，有些则按照细粒度模式删除，完全取决于你的业务需求（当然要不要给消息设置key是一个关键决策）。

3.3.2 日志压缩

不管是传统的RDBMS还是分布式的NoSQL存储在数据库中的数据总是会更新的，相同key的新记录更新数据的方式简单来说有两种：直接更新（找到数据库中的已有位置以最新的值替换旧的值），或者以追加的方式（保留旧的值，查询时再合并，或者也有一个后台线程会定期合并）。采用追加记录的做法在节点崩溃时可以用于恢复数据，还有一个好处是写性能很高，因为这样在写的时候就不需要查询操作，这也是表3-8中很多和存储相关的分布式系统都采用这种方式的原因，它的代价就是需要有Compaction操作来保证相同key的多条记录需要合并。

分布式系统	更新数据追加到哪里	数据文件	是否需要Compaction
ZooKeeper	log	snapshot	不需要，因为数据量不大
Redis	aof	rdb	不需要，因为是内存数据库
Cassandra	commit log	data.db	需要，数据存在本地文件
HBase	commit log	HFile	需要，数据存在HDFS
Kafka	commit log	commit log	需要，数据存在Partition的多个Segment里

表3-8 分布式系统的更新操作用commit log保存

Kafka中如果消息有key，相同key的消息在不同时刻有不同的值，则只允许存在最新的一条消息，这就好比传统数据库的update操作，查询结果一定是最近update的那一条，而不应该查询出多条或者查询出旧的记录，当然对于HBase/Cassandra这种支持多版本的数据库而言，update操作可能导致添加新的列，查询时是合并的结果而不一定就是最新的记录。图3-27中示例了多条消息，一旦key已经存在，相同key的旧的消息会被删除，新的被保留。

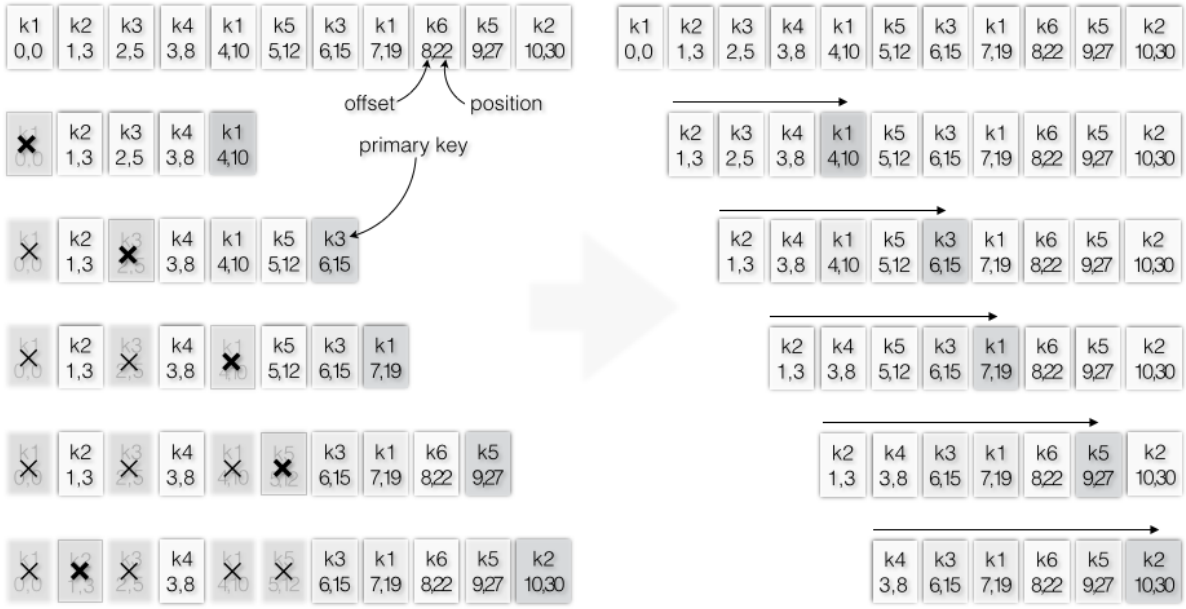


图3-27 更新操作要删除旧的消息

Kafka的更新操作也采用追加（commit log就是追加）也需要有Compaction操作，当然它并不是像上面那样一条消息一条消息地比较，通常Compaction是对多个文件做一次整体的压缩，图3-28是Log的压缩操作前后示例，压缩确保了相同key只存在一个最新的value，旧的value在压缩过程会被删除掉。

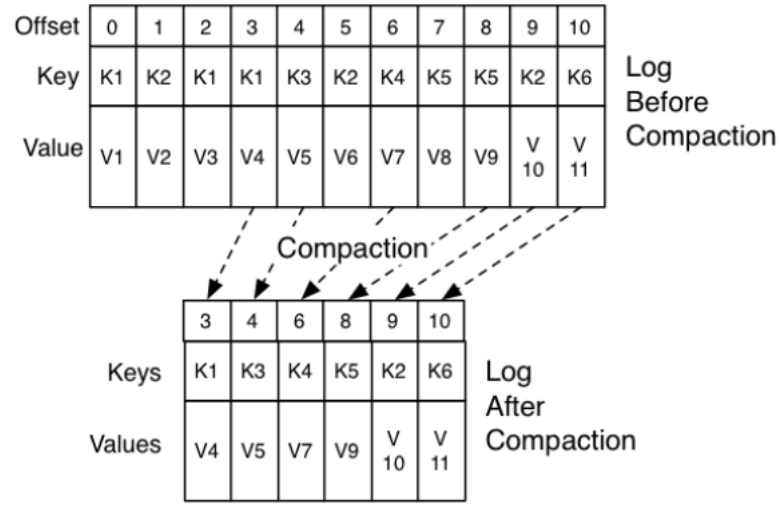


图3-28 LogCompaction的过程

每个Partition的（Leader Replica的）Log有多个Segment文件，为了不影响正在写的最近的那个activeSegment，日志压缩不应该清理activeSegment，而是清理剩下的所有Segment。清理Segment时也不是一个个Segment慢吞吞地清理，也不是一次性所有Segment想要全部清理，而是几个Segment分成一组，分批清理。清理线程会占用一定的CPU，因为要读取已有的Segment并压缩成新的Segment，为了不影响其他组件（主要是读，因为读操作会读取旧的Segment，而写不会被影响因为写操作只往activeSegment写，而activeSegment不会被清理），可以设置清理线程的线程个数，同时Kakfa还支持Throttler限速（读取旧的Segment时和写入新的Segment都可以限速）。当然也并不是每个Partition在同一时间都进行清理，而是选择其中最需要被清理的Partition。

清理/压缩指的是删除旧的更新操作，只保留最近的一个更新操作，清理方式有多种，比如JVM中的垃圾回收算法将存活的对象拷贝/整理到指定的区域，HBase/Cassandra的Compaction会将多个数据文件合并/整理成新的数据文件。Kafka的LogCleaner清理Log时会将所有的Segment在CleanerPoint清理点位置分成Tail和Head两部分，图3-29中每条消息所在的Segment并没有画出来（这些消息可能在不同的Segment里），因为清理是以Partition为级别，就淡化了Segment的边界问题，不过具体的清理动作还是要面向Segment，因为复制消息时不得不面对Segment文件。

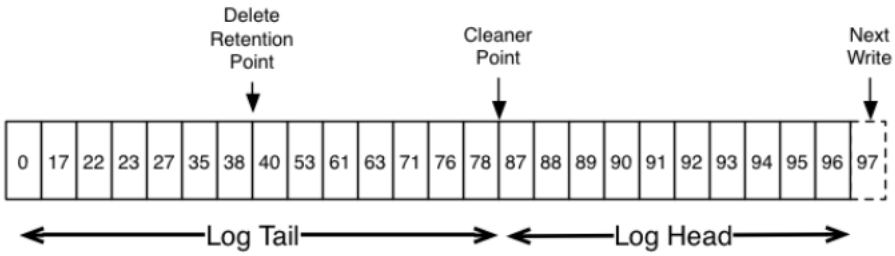


图3-29 Log包括Tail和Head两部分

清理后Log Head部分每条消息的offset都是逐渐递增的，而Tail部分消息的offset是断断续续的。 LogToClean 表示需要被清理的日志，其中firstDirtyOffset会作为Tail和Head的分界点，图3-20中举例了在一个Log的分界点发生Compaction的步骤。

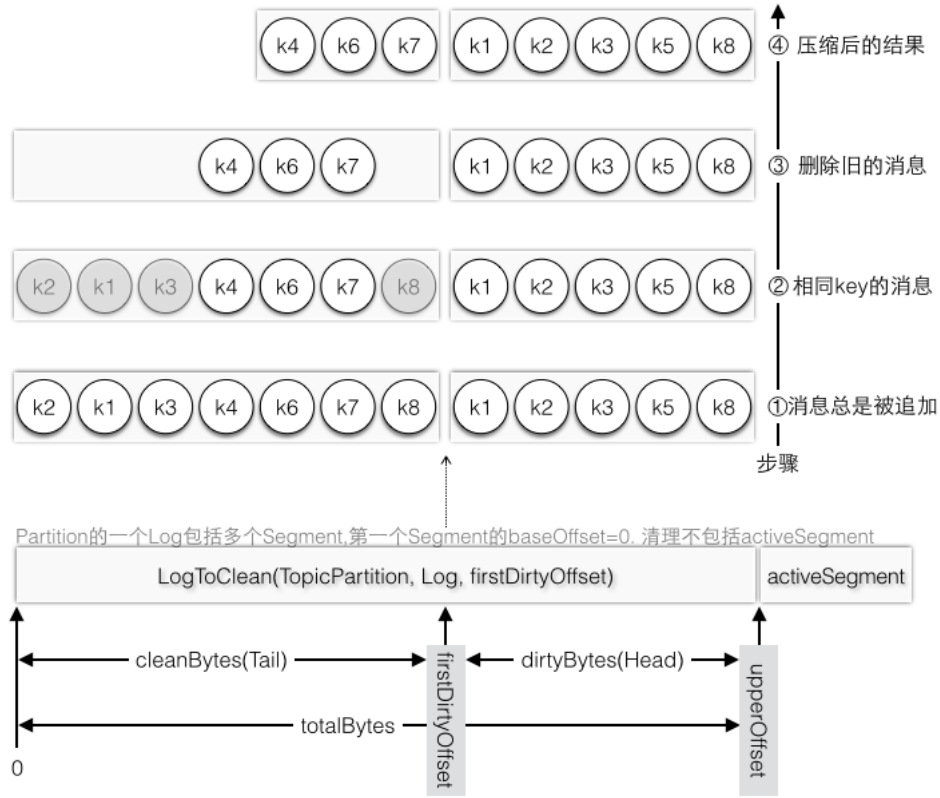


图3-30 日志分成Tail和Head的消息压缩步骤

每个Partition的Log都对应一个LogToClean对象，在选择哪个Partition需要优先做Compaction操作时是依据cleanableRatio的比率即Head部分大小（dirtyBytes）除于总大小中最大的，假设日志文件一样大，firstDirtyOffset越小，dirtyBytes就越大。而firstDirtyOffset每次Compaction后都会增加，所以实际上选择算法是优先选择还没有发生或者发生次数比较少的Partition，因为这样的Partition的firstDirtyOffset没有机会增加太多。

```
case class LogToClean(topicPartition: TopicAndPartition, log: Log,
    firstDirtyOffset: Long) extends Ordered[LogToClean] {
  val cleanBytes = log.logSegments(-1, firstDirtyOffset).map(_.size).sum
  val dirtyBytes = log.logSegments(firstDirtyOffset,
    math.max(firstDirtyOffset, log.activeSegment.baseOffset)).map(_.size).sum
  val cleanableRatio = dirtyBytes / totalBytes.toDouble
  def totalBytes = cleanBytes + dirtyBytes
  override def compare(th:LogToClean)=math.signum(this.cleanableRatio-th.cleanableRatio)
}
```

不仅仅是更新需要清理旧的数据，删除操作也需要清理，生产者客户端如果发送的消息key的value是空的，表示要删除这条消息，发生在删除标记之前的记录都需要删除掉，而发生在删除标记之后的记录则不会被删除。日志压缩保证了：

1. 任何消费者如果能够赶上Log的Head部分，它就会看到写入的每条消息，这些消息都是顺序递增（中间不会间断）的offset
2. 总是维持消息的有序性，压缩并不会对消息进行重新排序，而是移除一些消息
3. 每条消息的offset永远不会被改变，它是日志文件标识位置的永久编号
4. 读取/消费时如果从最开始的offset=0开始，那么至少可以看到所有记录按照它们写入的顺序得到的最终状态（状态指的是value，相同key不同value，最终的状态以最新的value为准）：因为这种场景下写入顺序和读取顺序是一致的，写入时和读取时offset都是不断递增。举例写入key1的value在offset=1和offst=5的值分别是v1和v2，那么读取到offset=1时，最终的状态（value值）是v1，读取到offset=5时，最终状态是v2（不能指望说读取到offset=1时就要求状态是v2）

来自： zqhxuyuan.github.io/2016/05/13/2016-5-13-Kafka-Book-Sample-LogCompaction/