

## 24 | ReplicaManager (中)：副本管理器是如何读写副本的？

2020-06-20 胡夕

Kafka核心源码解读

[进入课程 >](#)



讲述：胡夕

时长 17:51 大小 16.35M



你好，我是胡夕。上节课，我们学习了 ReplicaManager 类的定义和重要字段，今天我们接着学习这个类中的读写副本对象部分的源码。无论是读取副本还是写入副本，都是通过底层的 Partition 对象完成的，而这些分区对象全部保存在上节课所学的 allPartitions 字段中。可以说，理解这些字段的用途，是后续我们探索副本管理器类功能的重要前提。

现在，我们就来学习下副本读写功能。整个 Kafka 的同步机制，本质上就是副本读取 + 副本写入，搞懂了这两个功能，你就知道了 Follower 副本是如何同步 Leader 副本数据的。

### 副本写入：appendRecords



所谓的副本写入，是指向副本底层日志写入消息。在 ReplicaManager 类中，实现副本写入的方法叫 appendRecords。

放眼整个 Kafka 源码世界，需要副本写入的场景有 4 个。

场景一：生产者向 Leader 副本写入消息；

场景二：Follower 副本拉取消息后写入副本；


场景三：消费者组写入组信息；

场景四：事务管理器写入事务信息（包括事务标记、事务元数据等）。

除了第二个场景是直接调用 Partition 对象的方法实现之外，其他 3 个都是调用 `appendRecords` 来完成的。

该方法将给定一组分区的信息写入到对应的 Leader 副本中，并且根据 PRODUCE 请求中 `acks` 设置的不同，有选择地等待其他副本写入完成。然后，调用指定的回调逻辑。

我们先来看下它的方法签名：

 复制代码

```
1 def appendRecords(  
2   timeout: Long, // 请求处理超时时间  
3   requiredAcks: Short, // 请求acks设置  
4   internalTopicsAllowed: Boolean, // 是否允许写入内部主题  
5   origin: AppendOrigin, // 写入方来源  
6   entriesPerPartition: Map[TopicPartition, MemoryRecords], // 待写入消息  
7   // 回调逻辑  
8   responseCallback: Map[TopicPartition, PartitionResponse] => Unit,  
9   delayedProduceLock: Option[Lock] = None,  
10  recordConversionStatsCallback:  
11    Map[TopicPartition, RecordConversionStats] => Unit = _ => ()  
12  : Unit = {  
13    .....  
14  }
```

输入参数有很多，而且都很重要，我一个一个地说。

**timeout**：请求处理超时时间。对于生产者来说，它就是 `request.timeout.ms` 参数值。

**requiredAcks**：是否需要等待其他副本写入。对于生产者而言，它就是 `acks` 参数的值。而在其他场景中，Kafka 默认使用 `-1`，表示等待其他副本全部写入成功再返回。

**internalTopicsAllowed**：是否允许向内部主题写入消息。对于普通的生产者而言，该字段是 False，即不允许写入内部主题。对于 Coordinator 组件，特别是消费者组 GroupCoordinator 组件来说，它的职责之一就是向内部位移主题写入消息，因此，此时，该字段值是 True。

**origin**：AppendOrigin 是一个接口，表示写入方来源。当前，它定义了 3 类写入方，分别是 Replication、Coordinator 和 Client。Replication 表示写入请求是由 Follower 副本发出的，它要将从 Leader 副本获取到的消息写入到底层的消息日志中。Coordinator 表示这些写入由 Coordinator 发起，它既可以是管理消费者组的 GroupCoordinator，也可以是管理事务的 TransactionCoordinator。Client 表示本次写入由客户端发起。前面我们说过了，Follower 副本同步过程不调用 appendRecords 方法，因此，这里的 origin 值只可能是 Replication 或 Coordinator。


**entriesPerPartition**：按分区分组的、实际要写入的消息集合。

**responseCallback**：写入成功之后，要调用的回调逻辑函数。

**delayedProduceLock**：专门用来保护消费者组操作线程安全的锁对象，在其他场景中用不到。

**recordConversionStatsCallback**：消息格式转换操作的回调统计逻辑，主要用于统计消息格式转换操作过程中的一些数据指标，比如总共转换了多少条消息，花费了多长时间。

接下来，我们就看看，appendRecords 如何利用这些输入参数向副本日志写入消息。我把它的完整代码贴出来。对于重要的步骤，我标注了注释：

 复制代码

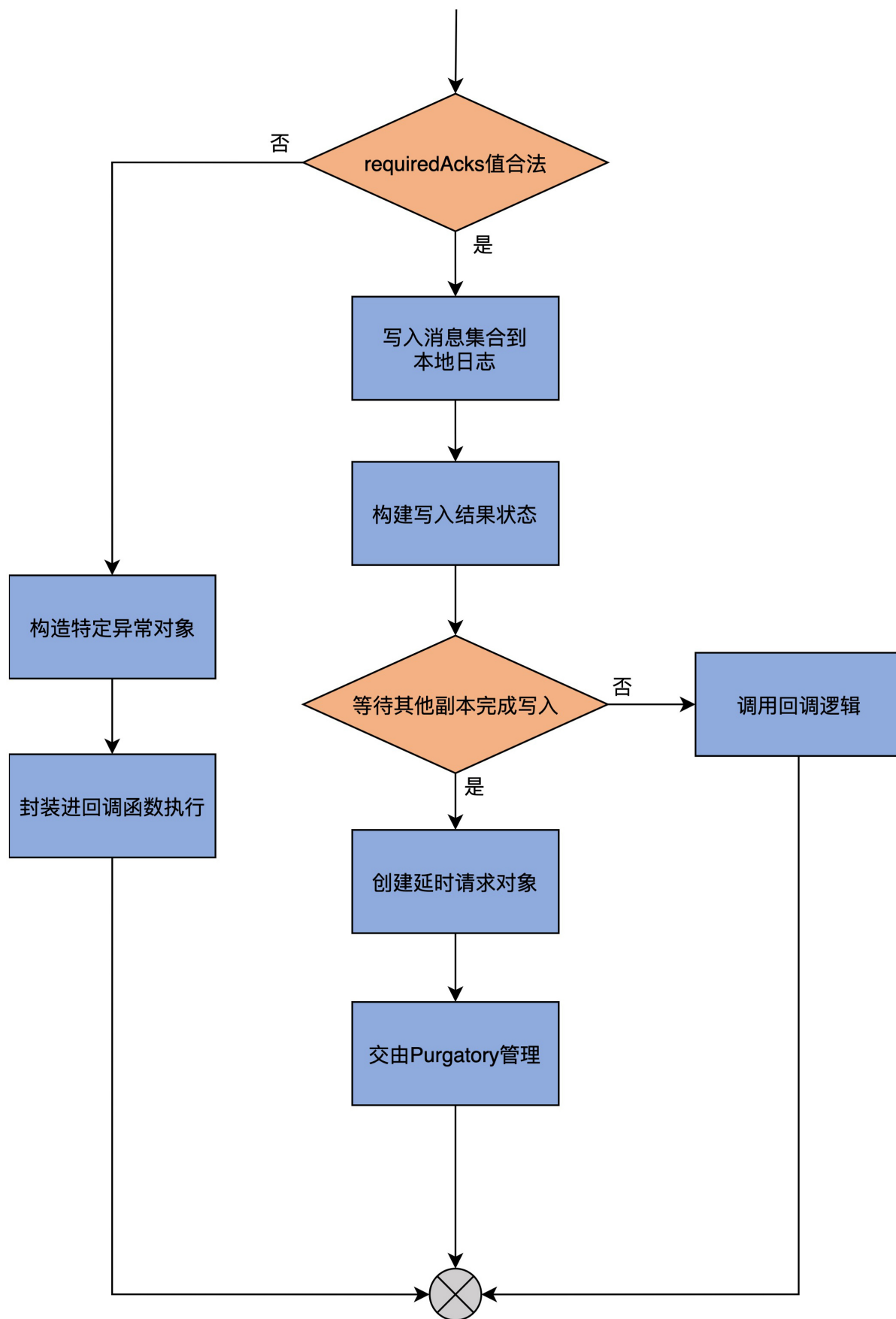
```
1 // requiredAcks合法取值是-1, 0, 1, 否则视为非法
2 if (isValidRequiredAcks(requiredAcks)) {
3     val sTime = time.milliseconds
4     // 调用appendToLocalLog方法写入消息集合到本地日志
5     val localProduceResults = appendToLocalLog(
6         internalTopicsAllowed = internalTopicsAllowed,
7         origin, entriesPerPartition, requiredAcks)
8     debug("Produce to local log in %d ms".format(time.milliseconds - sTime))
9     val produceStatus = localProduceResults.map { case (topicPartition, result) :
10         topicPartition ->
11             ProducePartitionStatus(
12                 result.info.lastOffset + 1, // 设置下一条待写入消息的位移值
13                 // 构建PartitionResponse封装写入结果
14                 new PartitionResponse(result.error, result.info.firstOffset.get0
15                     result.info.logStartOffset, result.info.recordErrors.asJava, r
```

```

16     }
17     // 尝试更新消息格式转换的指标数据
18     recordConversionStatsCallback(localProduceResults.map { case (k, v) => k -> v })
19     // 需要等待其他副本完成写入
20     if (delayedProduceRequestRequired(
21         requiredAcks, entriesPerPartition, localProduceResults)) {
22         val produceMetadata = ProduceMetadata(requiredAcks, produceStatus)
23         // 创建DelayedProduce延时请求对象
24         val delayedProduce = new DelayedProduce(timeout, produceMetadata, this, recordConversionStatsCallback)
25         val producerRequestKeys = entriesPerPartition.keys.map(TopicPartitionOperationKey)
26         // 再一次尝试完成该延时请求
27         // 如果暂时无法完成，则将对象放入到相应的Purgatory中等待后续处理
28         delayedProducePurgatory.tryCompleteElseWatch(delayedProduce, producerRequestKeys)
29     } else { // 无需等待其他副本写入完成，可以立即发送Response
30         val produceResponseStatus = produceStatus.map { case (k, status) => k -> status }
31         // 调用回调逻辑然后返回即可
32         responseCallback(produceResponseStatus)
33     }
34 } else { // 如果requiredAcks值不合法
35     val responseStatus = entriesPerPartition.map { case (topicPartition, _) =>
36         topicPartition -> new PartitionResponse(Errors.INVALID_REQUIRED_ACKS,
37             LogAppendInfo.UnknownLogAppendInfo.firstOffset.getOrElse(-1), RecordBatchMetadataSize)
38     }
39     // 构造INVALID_REQUIRED_ACKS异常并封装进回调函数调用中
40     responseCallback(responseStatus)
41 }

```

为了帮助你更好地理解，我再用一张图说明一下 `appendRecords` 方法的完整流程。





首先，它会判断 `requiredAcks` 的取值是否在合理范围内，也就是“是否是 -1、0、1 这三个数值中的一个”。如果不是合理取值，代码就进入到外层的 `else` 分支，构造名为 `INVALID_REQUIRED_ACKS` 的异常，并将其封装进回调函数中执行，然后返回结果。否则的话，代码进入到外层的 `if` 分支下。

进入到 `if` 分支后，代码调用 **`appendToLocalLog`** 方法，将要写入的消息集合保存到副本的本地日志上。然后构造 `PartitionResponse` 对象实例，来封装写入结果以及一些重要的元数据信息，比如本次写入有没有错误 (`errorMessage`)、下一条待写入消息的位移值、本次写入消息集合首条消息的位移值，等等。待这些做完了之后，代码会尝试更新消息格式转换的指标数据。此时，源码需要调用 `delayedProduceRequestRequired` 方法，来判断本次写入是否算是成功了。


如果还需要等待其他副本同步完成消息写入，那么就不能立即返回，代码要创建 `DelayedProduce` 延时请求对象，并把该对象交由 `Purgatory` 来管理。`DelayedProduce` 是生产者端的延时发送请求，对应的 `Purgatory` 就是 `ReplicaManager` 类构造函数中的 `delayedProducePurgatory`。所谓的 `Purgatory` 管理，主要是调用 `tryCompleteElseWatch` 方法尝试完成延时发送请求。如果暂时无法完成，就将对象放入到相应的 `Purgatory` 中，等待后续处理。

如果无需等待其他副本同步完成消息写入，那么，`appendRecords` 方法会构造响应的 `Response`，并调用回调逻辑函数，至此，方法结束。

从刚刚的分析中，我们可以知道，`appendRecords` 实现消息写入的方法是 **`appendToLocalLog`**，用于判断是否需要等待其他副本写入的方法是 **`delayedProduceRequestRequired`**。下面我们就深入地学习下这两个方法的代码。

首先来看 `appendToLocalLog`。从它的名字来看，就是写入副本本地日志。我们来看一下该方法的主要代码片段。

```
1 private def appendToLocalLog(  
2     internalTopicsAllowed: Boolean,  
3     origin: AppendOrigin,  
4     entriesPerPartition: Map[TopicPartition, MemoryRecords],  
5     requiredAcks: Short): Map[TopicPartition, LogAppendResult] = {  
6     .....  
7     entriesPerPartition.map { case (topicPartition, records) =>
```

 复制代码

```

8     brokerTopicStats.topicStats(topicPartition.topic)
9     .totalProduceRequestRate.mark()
10    brokerTopicStats.allTopicsStats.totalProduceRequestRate.mark()
11    // 如果要写入的主题是内部主题, 而internalTopicsAllowed=false, 则返回错误
12    if (Topic.isInternal(topicPartition.topic)
13        && !internalTopicsAllowed) {
14        (topicPartition, LogAppendResult(
15            LogAppendInfo.UnknownLogAppendInfo,
16            Some(new InvalidTopicException(s"Cannot append to internal topic ${top
17    } else {
18        try {
19            // 获取分区对象
20            val partition = getPartitionOrException(topicPartition, expectLeader =
21            // 向该分区对象写入消息集合
22            val info = partition.appendRecordsToLeader(records, origin, requiredAcl
23            .....
24            // 返回写入结果
25            (topicPartition, LogAppendResult(info))
26        } catch {
27            .....
28        }
29    }
30 }
31 }

```

我忽略了很多打日志以及错误处理的代码。你可以看到, 该方法主要就是利用 Partition 的 appendRecordsToLeader 方法写入消息集合, 而后者就是利用我们在 [第 3 节课](#)学到的 appendAsLeader 方法写入本地日志的。总体来说, appendToLocalLog 的逻辑不复杂, 你应该很容易理解。

下面我们看下 delayedProduceRequestRequired 方法的源码。它用于判断消息集合被写入到日志之后, 是否需要等待其他副本也写入成功。我们看下它的代码:

 复制代码

```

1 private def delayedProduceRequestRequired(
2     requiredAcks: Short,
3     entriesPerPartition: Map[TopicPartition, MemoryRecords],
4     localProduceResults: Map[TopicPartition, LogAppendResult]): Boolean = {
5     requiredAcks == -1 && entriesPerPartition.nonEmpty &&
6     localProduceResults.values.count(_.exception.isDefined) < entriesPerPartit
7 }

```

该方法返回一个布尔值，True 表示需要等待其他副本完成；False 表示无需等待。上面的代码表明，如果需要等待其他副本的写入，就必须同时满足 3 个条件：

1. requiredAcks 必须等于 -1；
2. 依然有数据尚未写完；
3. 至少有一个分区的信息已经成功地被写入到本地日志。

其实，你可以把条件 2 和 3 联合在一起来看。如果所有分区的数据写入都不成功，就表明可能出现了很严重的错误，此时，比较明智的做法是不再等待，而是直接返回错误给发送方。相反地，如果有部分分区成功写入，而部分分区写入失败了，就表明可能是由偶发的瞬时错误导致的。此时，不妨将本次写入请求放入 Purgatory，再给它一个重试的机会。

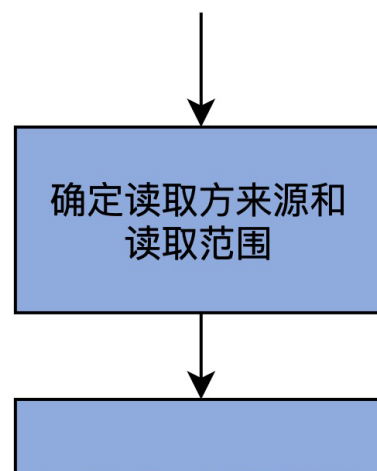
## 副本读取：fetchMessages

好了，说完了副本的写入，下面我们进入到副本读取的源码学习。

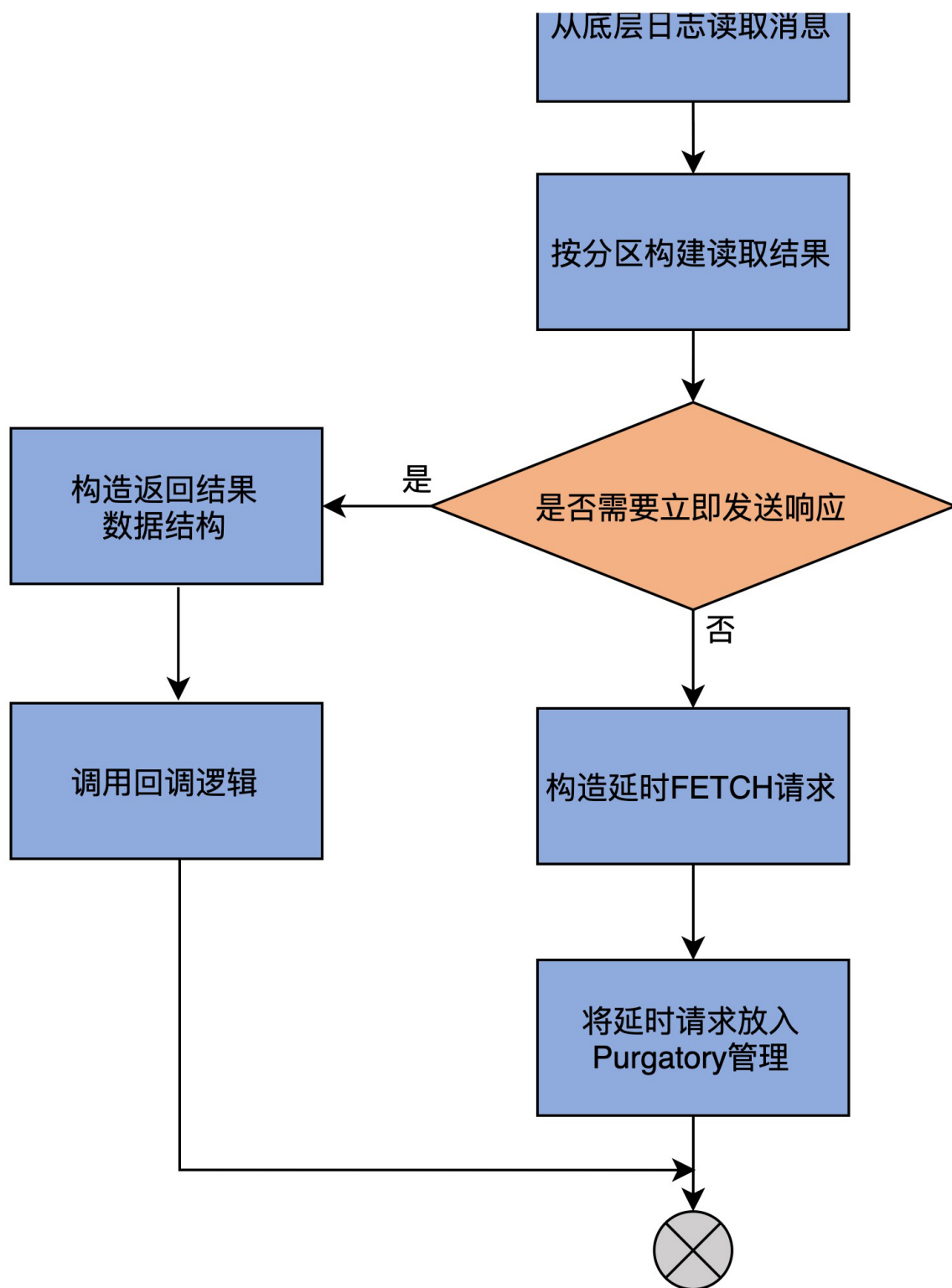
在 ReplicaManager 类中，负责读取副本数据的方法是 fetchMessages。不论是 Java 消费者 API，还是 Follower 副本，它们拉取消息的主要途径都是向 Broker 发送 FETCH 请求，Broker 端接收到该请求后，调用 fetchMessages 方法从底层的 Leader 副本取出消息。

和 appendRecords 方法类似，fetchMessages 方法也可能会延时处理 FETCH 请求，因为 Broker 端必须要累积足够多的数据之后，才会返回 Response 给请求发送方。

可以看一下下面的这张流程图，它展示了 fetchMessages 方法的主要逻辑。







我们来看下该方法的签名：

```
1 def fetchMessages(timeout: Long,  
2     replicaId: Int,  
3     fetchMinBytes: Int,  
4     fetchMaxBytes: Int,  
5     hardMaxBytesLimit: Boolean,
```

复制代码

```
6         fetchInfos: Seq[(TopicPartition, PartitionData)],
7         quota: ReplicaQuota,
8         responseCallback: Seq[(TopicPartition, FetchPartitionData)] :
9         isolationLevel: IsolationLevel,
10        clientMetadata: Option[ClientMetadata]): Unit = {
11        .....
12    }
```

这些输入参数都是我们理解下面的重要方法的基础，所以，我们来逐个分析一下。

**timeout**：请求处理超时时间。对于消费者而言，该值就是 `request.timeout.ms` 参数值；对于 Follower 副本而言，该值是 Broker 端参数 `replica.fetch.wait.max.ms` 的值。

**replicaId**：副本 ID。对于消费者而言，该参数值是 -1；对于 Follower 副本而言，该值就是 Follower 副本所在的 Broker ID。

**fetchMinBytes & fetchMaxBytes**：能够获取的最小字节数和最大字节数。对于消费者而言，它们分别对应于 Consumer 端参数 `fetch.min.bytes` 和 `fetch.max.bytes` 值；对于 Follower 副本而言，它们分别对应于 Broker 端参数 `replica.fetch.min.bytes` 和 `replica.fetch.max.bytes` 值。

**hardMaxBytesLimit**：对能否超过最大字节数做硬限制。如果 `hardMaxBytesLimit=True`，就表示，读取请求返回的数据字节数绝不允许超过最大字节数。

**fetchInfos**：规定了读取分区的信息，比如要读取哪些分区、从这些分区的哪个位移值开始读、最多可以读多少字节，等等。

**quota**：这是一个配额控制类，主要是为了判断是否需要在读取的过程中做限速控制。

**responseCallback**：Response 回调逻辑函数。当请求被处理完成后，调用该方法执行收尾逻辑。

有了这些铺垫之后，我们进入到方法代码的学习。为了便于学习，我将整个方法的代码分成两部分：第一部分是读取本地日志；第二部分是根据读取结果确定 Response。

我们先看第一部分的源码：

```

1 // 判断该读取请求是否来自于Follower副本或Consumer
2 val isFromFollower = Request.isValidBrokerId(replicaId)
3 val isFromConsumer = !(isFromFollower || replicaId == Request.FutureLocalRepli
4 // 根据请求发送方判断可读取范围
5 // 如果请求来自于普通消费者，那么可以读到LEO值
6 // 如果请求来自于配置了READ_COMMITTED的消费者，那么可以读到Log Stable Offset值
7 // 如果请求来自于Follower副本，那么可以读到高水位值
8 val fetchIsolation = if (!isFromConsumer)
9     FetchLogEnd
10 else if (isolationLevel == IsolationLevel.READ_COMMITTED)
11     FetchTxnCommitted
12 else
13     FetchHighWatermark
14 val fetchOnlyFromLeader = isFromFollower || (isFromConsumer && clientMetadata.
15 // 定义readFromLog方法读取底层日志中的消息
16 def readFromLog(): Seq[(TopicPartition, LogReadResult)] = {
17     val result = readFromLocalLog(
18         replicaId = replicaId,
19         fetchOnlyFromLeader = fetchOnlyFromLeader,
20         fetchIsolation = fetchIsolation,
21         fetchMaxBytes = fetchMaxBytes,
22         hardMaxBytesLimit = hardMaxBytesLimit,
23         readPartitionInfo = fetchInfos,
24         quota = quota,
25         clientMetadata = clientMetadata)
26     if (isFromFollower) updateFollowerFetchState(replicaId, result)
27     else result
28 }
29 // 读取消息并返回日志读取结果
30


```

这部分代码首先会判断，读取消息的请求方到底是 Follower 副本，还是普通的 Consumer。判断的依据就是看 **replicaId 字段是否大于 0**。Consumer 的 replicaId 是 -1，而 Follower 副本的则是大于 0 的数。一旦确定了请求方，代码就能确定可读取范围。

这里的 fetchIsolation 是读取隔离级别的意思。对于 Follower 副本而言，它能读取到 Leader 副本 LEO 值以下的所有消息；对于普通 Consumer 而言，它只能“看到”Leader 副本高水位值以下的消息。

待确定了可读取范围后，fetchMessages 方法会调用它的内部方法 **readFromLog**，读取本地日志上的消息数据，并将结果赋值给 logReadResults 变量。readFromLog 方法的主要实现是调用 readFromLocalLog 方法，而后者就是在待读取分区上依次调用其日志对象的 read 方法执行实际的消息读取。

fetchMessages 方法的第二部分，是根据上一步的读取结果创建对应的 Response。我们看下具体实现：

 复制代码

```
1 var bytesReadable: Long = 0
2 var errorReadingData = false
3 val logReadResultMap = new mutable.HashMap[TopicPartition, LogReadResult]
4 // 统计总共可读取的字节数
5 logReadResults.foreach { case (topicPartition, logReadResult) =>
6   brokerTopicStats.topicStats(topicPartition.topic).totalFetchRequestRate.mark(
7     brokerTopicStats.allTopicsStats.totalFetchRequestRate.mark()
8     if (logReadResult.error != Errors.NONE)
9       errorReadingData = true
10   bytesReadable = bytesReadable + logReadResult.info.records.sizeInBytes
11   logReadResultMap.put(topicPartition, logReadResult)
12 }
13 // 判断是否能够立即返回Response，满足以下4个条件中的任意一个即可：
14 // 1. 请求没有设置超时时间，说明请求方想让请求被处理后立即返回
15 // 2. 未获取到任何数据
16 // 3. 已累积到足够多的数据
17 // 4. 读取过程中出错
18 if (timeout <= 0 || fetchInfos.isEmpty || bytesReadable >= fetchMinBytes || er
19   // 构建返回结果
20   val fetchPartitionData = logReadResults.map { case (tp, result) =>
21     tp -> FetchPartitionData(result.error, result.highWatermark, result.leader
22       result.lastStableOffset, result.info.abortedTransactions, result.preferri
23   }
24   // 调用回调函数
25   responseCallback(fetchPartitionData)
26 } else { // 如果无法立即完成请求
27   val fetchPartitionStatus = new mutable.ArrayBuffer[(TopicPartition, FetchPar
28   fetchInfos.foreach { case (topicPartition, partitionData) =>
29     logReadResultMap.get(topicPartition).foreach(logReadResult => {
30       val logOffsetMetadata = logReadResult.info.fetchOffsetMetadata
31       fetchPartitionStatus += (topicPartition -> FetchPartitionStatus(logOffse
32     })
33   }
34   val fetchMetadata: SFetchMetadata = SFetchMetadata(fetchMinBytes, fetchMaxBy
35     fetchOnlyFromLeader, fetchIsolation, isFromFollower, replicaId, fetchParti
36   // 构建DelayedFetch延时请求对象
37   val delayedFetch = new DelayedFetch(timeout, fetchMetadata, this, quota, cli
38     responseCallback)
39   val delayedFetchKeys = fetchPartitionStatus.map { case (tp, _) => TopicParti
40   // 再一次尝试完成请求，如果依然不能完成，则交由Purgatory等待后续处理
41   delayedFetchPurgatory.tryCompleteElseWatch(delayedFetch, delayedFetchKeys)
42 }
```

这部分代码首先会根据上一步得到的读取结果，统计可读取的总字节数，之后，判断此时是否能够立即返回 `Reponse`。那么，怎么判断是否能够立即返回 `Response` 呢？实际上，只要满足以下 4 个条件中的任意一个即可：

1. 请求没有设置超时时间，说明请求方想让请求被处理后立即返回；
2. 未获取到任何数据；
3. 已累积到足够多数据；
4. 读取过程中出错。

如果这 4 个条件一个都不满足，就需要进行延时处理了。具体来说，就是构建 `DelayedFetch` 对象，然后把该延时对象交由 `delayedFetchPurgatory` 后续自动处理。

至此，关于副本管理器读写副本的两个方法 `appendRecords` 和 `fetchMessages`，我们就学完了。本质上，它们在底层分别调用 `Log` 的 `append` 和 `read` 方法，以实现本地日志的读写操作。当完成读写操作之后，这两个方法还定义了延时处理的条件。一旦发现满足了延时处理的条件，就交给对应的 `Purgatory` 进行处理。

从这两个方法中，我们已经看到了之前课程中单个组件融合在一起的趋势。就像我在开篇词里面说的，虽然我们学习单个源码文件的顺序是自上而下，但串联 `Kafka` 主要组件功能的路径却是自下而上。

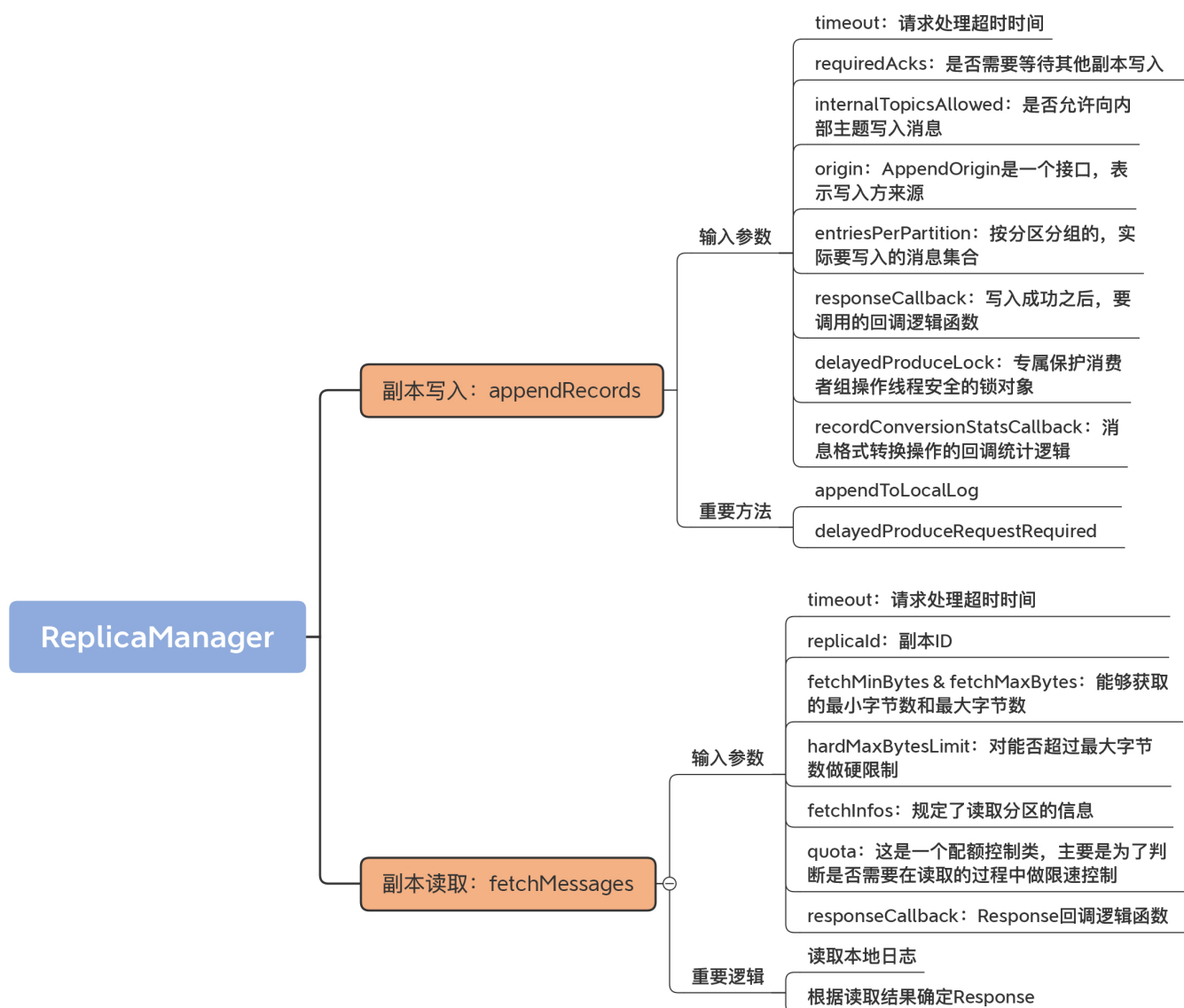
就拿这节课的副本写入操作来说，日志对象的 `append` 方法被上一层 `Partition` 对象中的方法调用，而后者又进一步被副本管理器中的方法调用。我们是按照自上而下的方式阅读副本管理器、日志对象等单个组件的代码，了解它们各自的独立功能的，现在，我们开始慢慢地把它们融合在一起，勾勒出了 `Kafka` 操作分区副本日志对象的完整调用路径。咱们同时采用这两种方式来阅读源码，就可以更快、更深入地搞懂 `Kafka` 源码的原理了。

## 总结

今天，我们学习了 `Kafka` 副本状态机类 `ReplicaManager` 是如何读写副本的，重点学习了它的两个重要方法 `appendRecords` 和 `fetchMessages`。我们再简单回顾一下。

`appendRecords`：向副本写入消息的方法，主要利用 `Log` 的 `append` 方法和 `Purgatory` 机制，共同实现 `Follower` 副本向 `Leader` 副本获取消息后的数据同步工作。

fetchMessages：从副本读取消息的方法，为普通 Consumer 和 Follower 副本所使用。当它们向 Broker 发送 FETCH 请求时，Broker 上的副本管理器调用该方法从本地日志中获取指定消息。



下节课中，我们要把重心转移到副本管理器对副本和分区对象的管理上。这是除了读写副本之外，副本管理器另一大核心功能，你一定不要错过！

## 课后讨论

appendRecords 参数列表中有个 origin。我想请你思考一下，在写入本地日志的过程中，这个参数的作用是什么？你能找出最终使用 origin 参数的具体源码位置吗？



欢迎在留言区写下你的思考和答案，跟我交流讨论，也欢迎你把今天的内容分享给你的朋友。

## 更多课程推荐

# 从0开始学架构

—— 前阿里P9技术专家的  
实战架构心法 ——

李运华 前阿里P9技术专家



涨价倒计时 🕒

今日秒杀 **¥79**，7月1日涨价至 **¥129**

© 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 23 | ReplicaManager（上）：必须要掌握的副本管理类定义和核心字段

下一篇 25 | ReplicaManager（下）：副本管理器是如何管理副本的？

## 精选留言 (1)

💬 写留言



胡夕 置顶

2020-06-23

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，我们重点学习了ReplicaManager类的类定义和核心字段。课后我请你自行写一个统计Online状态分区数的方法。我的代码如下：

```
private def onlinePartitionCount: Int = {...
```

展开 ∨



👍 1

