

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

2020-06-18 胡夕

Kafka核心源码解读

[进入课程 >](#)



讲述：胡夕

时长 16:50 大小 15.43M



你好，我是胡夕。

今天，我们要学习的是 Kafka 中的副本管理器 ReplicaManager。它负责管理和操作集群中 Broker 的副本，还承担了一部分的分区管理工作，比如变更整个分区的副本日志路径等。

你一定还记得，前面讲到状态机的时候，我说过，Kafka 同时实现了副本状态机和分区状态机。但对于管理器而言，Kafka 源码却没有专门针对分区，定义一个类似于“分区管理器”这样的类，而是只定义了 ReplicaManager 类。该类不只实现了对副本的管理，还包含了很多操作分区对象的方法。

ReplicaManager 类的源码非常重要，它是构建 Kafka 副本同步机制的重要组件之一。副本同步过程中出现的大多数问题都是很难定位和解决的，因此，熟练掌握这部分源码，将有助于我们深入探索线上生产环境问题的根本原因，防止以后踩坑。下面，我给你分享一个真实的案例。

我们团队曾碰到过一件古怪事：在生产环境系统中执行删除消息的操作之后，该操作引发了 Follower 端副本与 Leader 端副本的不一致问题。刚碰到这个问题时，我们一头雾水，在正常情况下，Leader 端副本执行了消息删除后，日志起始位移值被更新了，Follower 端副本也应该更新日志起始位移值，但是，这里的 Follower 端的更新失败了。我们查遍了所有日志，依然找不到原因，最后还是通过分析 ReplicaManager 类源码，才找到了答案。

我们先看一下这个错误的详细报错信息：

 复制代码

```
1 Caused by: org.apache.kafka.common.errors.OffsetOutOfRangeException: Cannot in
```

这是 Follower 副本抛出来的异常，对应的 Leader 副本日志则一切如常。下面的日志显示出 Leader 副本的 Log Start Offset 已经被成功调整了。

 复制代码

```
1 INFO Incrementing log start offset of partition XXX-12 to 22786435
```

碰到这个问题时，我相信你的第一反应和我一样：这像是一个 Bug，但又不确定到底是什么原因导致的。后来，我们顺着 KafkaApis 一路找下去，直到找到了 ReplicaManager 的 deleteRecords 方法，才看出端倪。

Follower 副本从 Leader 副本拉取到消息后，会做两个操作：

1. 写入到自己的本地日志；
2. 更新 Follower 副本的高水位值和 Log Start Offset。

如果删除消息的操作 deleteRecords 发生在这两步之间，因为 deleteRecords 会变更 Log Start Offset，所以，Follower 副本在进行第 2 步操作时，它使用的可能是已经过期的值

了，因而会出现上面的错误。由此可见，这的确是一个 Bug。在确认了这一点之后，后面的解决方案也就呼之欲出了：虽然 deleteRecords 功能实用方便，但鉴于这个 Bug，我们还是应该尽力避免在线上环境直接使用该功能。

说到这儿，我想说一句，碰到实际的线上问题不可怕，可怕的是我们无法定位到问题的根本原因。写过 Java 项目的你一定有这种体会，很多时候，单纯依靠栈异常信息是不足以定位问题的。特别是涉及到 Kafka 副本同步这块，如果只看输出日志的话，你是很难搞清楚这其中的原理的，因此，我们必须借助源码，这也是我们今天学习 ReplicaManager 类的主要目的。

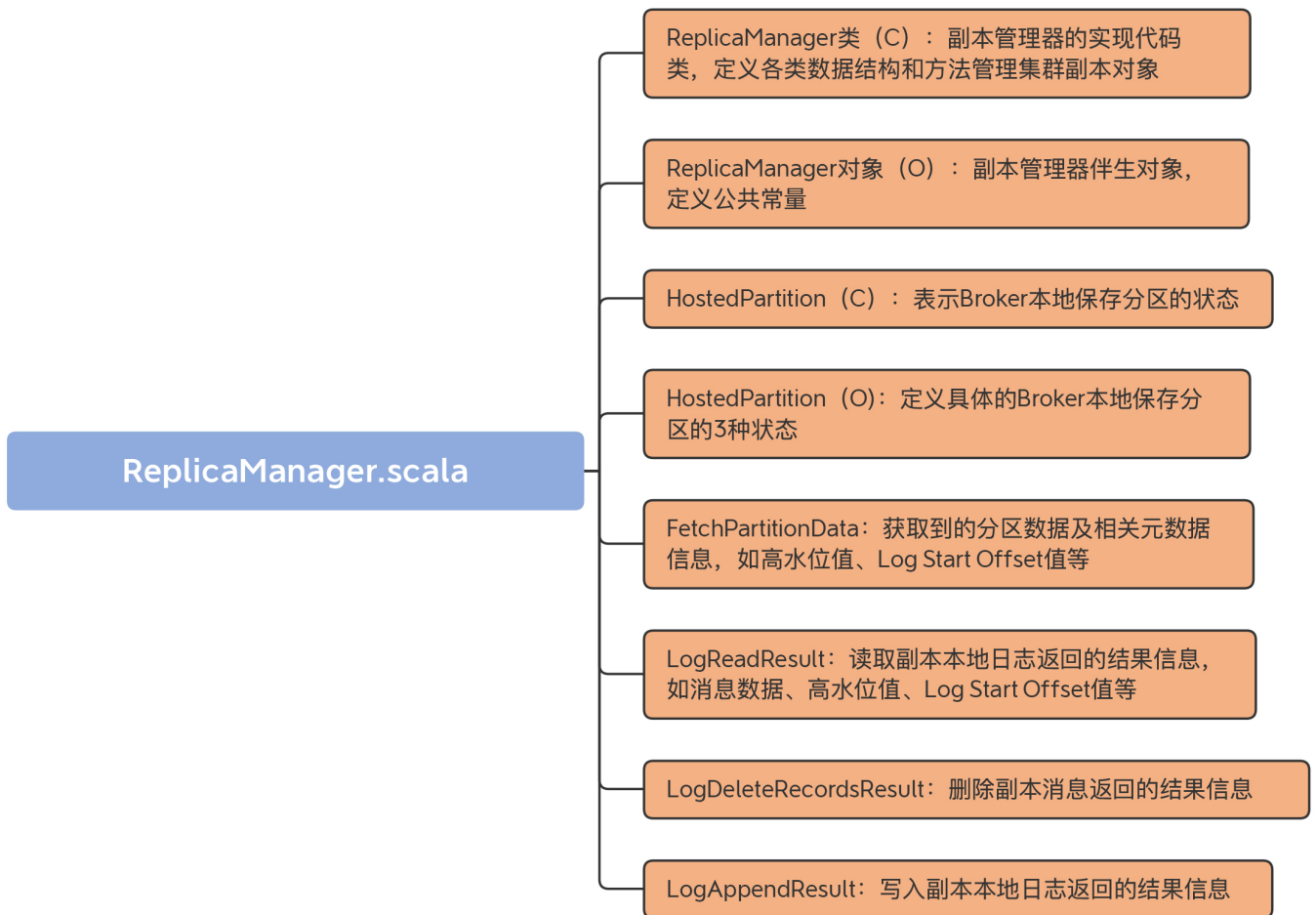
接下来，我们就重点学习一下这个类。它位于 server 包下的同名 scala 文件中。这是一个有着将近 1900 行的大文件，里面的代码结构很琐碎。

因为副本的读写操作和管理操作都是重磅功能，所以，在深入细节之前，我们必须理清 ReplicaManager 类的结构之间的关系，并且搞懂类定义及核心字段，这就是我们这节课的重要目标。

在接下来的两节课里，我会给你详细地解释副本读写操作和副本管理操作。学完这些之后，你就能清晰而深入地掌握 ReplicaManager 类的主要源码了，最重要的是，你会搞懂副本成为 Leader 或者是 Follower 时需要执行的逻辑，这就足以帮助你应对实际遇到的副本操作问题了。

代码结构

我们首先看下这个 scala 文件的代码结构。我用一张思维导图向你展示下：



虽然从代码结构上看，该文件下有 8 个部分，不过 HostedPartition 接口以及实现对象放在一起更好理解，所以，我把 ReplicaManager.scala 分为 7 大部分。

ReplicaManager 类：它是副本管理器的具体实现代码，里面定义了读写副本、删除副本消息的方法以及其他管理方法。

ReplicaManager 对象：ReplicaManager 类的伴生对象，仅仅定义了 3 个常量。

HostedPartition 及其实现对象：表示 Broker 本地保存的分区对象的状态。可能的状态包括：不存在状态 (None)、在线状态 (Online) 和离线状态 (Offline)。

FetchPartitionData：定义获取到的分区数据以及重要元数据信息，如高水位值、Log Start Offset 值等。

LogReadResult：表示副本管理器从副本本地日志中**读取**到的消息数据以及相关元数据信息，如高水位值、Log Start Offset 值等。

LogDeleteRecordsResult：表示副本管理器执行副本日志**删除**操作后返回的结果信息。

LogAppendResult: 表示副本管理器执行副本日志**写入**操作后返回的结果信息。


从含义来看，FetchPartitionData 和 LogReadResult 很类似，它们的区别在哪里呢？

其实，它们之间的差别非常小。如果翻开源码的话，你会发现，FetchPartitionData 类总共有 8 个字段，而构建 FetchPartitionData 实例的前 7 个字段都是用 LogReadResult 的字段来赋值的。你大致可以认为两者的作用是类似的。只是，FetchPartitionData 还有个字段标识该分区是不是处于重分配中。如果是的话，需要更新特定的 JXM 监控指标。这是这两个类的主要区别。

在这 7 个部分中，ReplicaManager 类是我们学习的重点。其他类要么仅定义常量，要么就是保存数据的 POJO 类，作用一目了然，我就不展开讲了。

ReplicaManager 类定义

接下来，我们就从 Replica 类的定义和重要字段这两个维度入手，进行学习。首先看 ReplicaManager 类的定义。

 复制代码

```
1 class ReplicaManager(  
2     val config: KafkaConfig, // 配置管理类  
3     metrics: Metrics, // 监控指标类  
4     time: Time, // 定时器类  
5     val zkClient: KafkaZkClient, // ZooKeeper客户端  
6     scheduler: Scheduler, // Kafka调度器  
7     val isShuttingDown: AtomicBoolean, // 是否已经关闭  
8     quotaManagers: QuotaManagers, // 配额管理器  
9     val brokerTopicStats: BrokerTopicStats, // Broker主题监控指标类  
10    val metadataCache: MetadataCache, // Broker元数据缓存  
11    logDirFailureChannel: LogDirFailureChannel,  
12    // 处理延时PRODUCE请求的Purgatory  
13    val delayedProducePurgatory: DelayedOperationPurgatory[DelayedProduce],  
14    // 处理延时FETCH请求的Purgatory  
15    val delayedFetchPurgatory: DelayedOperationPurgatory[DelayedFetch],  
16    // 处理延时DELETE_RECORDS请求的Purgatory  
17    val delayedDeleteRecordsPurgatory: DelayedOperationPurgatory[DelayedDeleteRe  
18    // 处理延时ELECT_LEADERS请求的Purgatory  
19    val delayedElectLeaderPurgatory: DelayedOperationPurgatory[DelayedElectLeade  
20    threadNamePrefix: Option[String]) extends Logging with KafkaMetricsGroup {  
21        .....  
22    }
```

ReplicaManager 类构造函数的字段非常多。有的字段含义很简单，像 time 和 metrics 这类字段，你一看就明白了，我就不多说了，我详细解释几个比较关键的字段。这些字段是我们理解副本管理器的重要基础。

1.logManager

这是日志管理器。它负责创建和管理分区的日志对象，里面定义了很多操作日志对象的方法，如 getOrCreateLog 等。

2.metadataCache

这是 Broker 端的元数据缓存，保存集群上分区的 Leader、ISR 等信息。注意，它和我们之前说的 Controller 端元数据缓存是有联系的。每台 Broker 上的元数据缓存，是从 Controller 端的元数据缓存异步同步过来的。

3.logDirFailureChannel

这是失效日志路径的处理器类。Kafka 1.1 版本新增了对于 JBOD 的支持。这也就是说，Broker 如果配置了多个日志路径，当某个日志路径不可用之后（比如该路径所在的磁盘已满），Broker 能够继续工作。那么，这就需要一整套机制来保证，在出现磁盘 I/O 故障时，Broker 的正常磁盘下的副本能够正常提供服务。

其中，logDirFailureChannel 是暂存失效日志路径的管理器类。我们不用具体学习这个特性的源码，但你最起码要知道，该功能算是 Kafka 提升服务器端高可用性的一个改进。有了它之后，即使 Broker 上的单块磁盘坏掉了，整个 Broker 的服务也不会中断。

4. 四个 Purgatory 相关的字段

这 4 个字段是 delayedProducePurgatory、delayedFetchPurgatory、delayedDeleteRecordsPurgatory 和 delayedElectLeaderPurgatory，它们分别管理 4 类延时请求的。其中，前两类我们应该不陌生，就是处理延时生产者请求和延时消费者请求；后面两类是处理延时消息删除请求和延时 Leader 选举请求，属于比较高阶的用法（可以暂时不用理会）。

在副本管理过程中，状态的变更大多都会引发对延时请求的处理，这时候，这些 Purgatory 字段就派上用场了。

只要掌握了刚刚的这些字段，就可以应对接下来的副本管理操作了。其中，最重要的就是 logManager。它是协助副本管理器操作集群副本对象的关键组件。

重要的自定义字段

学完了类定义，我们看下 ReplicaManager 类中那些重要的自定义字段。这样的字段大约有 20 个，我们不用花时间逐一学习它们。像 isrExpandRate、isrShrinkRate 这样的字段，我们只看名字，就能知道，它们是衡量 ISR 变化的监控指标。下面，我详细介绍几个对理解副本管理器至关重要的字段。我会结合代码，具体讲解它们的含义，同时还会说明它们的重要用途。

controllerEpoch

我们首先来看 controllerEpoch 字段。

这个字段的作用是**隔离过期 Controller 发送的请求**。这就是说，老的 Controller 发送的请求不能再被继续处理了。至于如何区分是老 Controller 发送的请求，还是新 Controller 发送的请求，就是**看请求携带的 controllerEpoch 值，是否等于这个字段的值**。以下是它的定义代码：


```
1 @volatile var controllerEpoch: Int =  
2     KafkaController.InitialControllerEpoch
```

 复制代码

该字段表示最新一次变更分区 Leader 的 Controller 的 Epoch 值，其默认值为 0。Controller 每发生一次变更，该字段值都会 +1。

在 ReplicaManager 的代码中，很多地方都会用到它来判断 Controller 发送过来的控制类请求是否合法。如果请求中携带的 controllerEpoch 值小于该字段值，就说明这个请求是由一个老的 Controller 发出的，因此，ReplicaManager 直接拒绝该请求的处理。

值得注意的是，它是一个 var 类型，这就说明它的值是能够动态修改的。当 ReplicaManager 在处理控制类请求时，会更新该字段。可以看下下面的代码：

 复制代码

```
1 // becomeLeaderOrFollower方法
2 // 处理LeaderAndIsrRequest请求时
3 controllerEpoch = leaderAndIsrRequest.controllerEpoch
4 // stopReplicas方法
5 // 处理StopReplicaRequest请求时
6 this.controllerEpoch = controllerEpoch
7 // maybeUpdateMetadataCache方法
8 // 处理UpdateMetadataRequest请求时
9 controllerEpoch = updateMetadataRequest.controllerEpoch
```

Broker 上接收的所有请求都是由 Kafka I/O 线程处理的，而 I/O 线程可能有多个，因此，这里的 controllerEpoch 字段被声明为 volatile 型，以保证其内存可见性。

allPartitions

下一个重要的字段是 allPartitions。这节课刚开始时我说过，Kafka 没有所谓的分区管理器，ReplicaManager 类承担了一部分分区管理的工作。这里的 allPartitions，就承载了 Broker 上保存的所有分区对象数据。其定义代码如下：

 复制代码

```
1 private val allPartitions = new Pool[TopicPartition, HostedPartition](
2     valueFactory = Some(tp => HostedPartition.Online(Partition(tp, time, this)))
3 )
```

从代码可见，allPartitions 是分区 Partition 对象实例的容器。这里的 HostedPartition 是代表分区状态的类。allPartitions 会将所有分区对象初始化成 Online 状态。

值得注意的是，这里的分区状态和我们之前讲到的分区状态机里面的状态完全隶属于两套“领导班子”。也许未来它们会有合并的可能。毕竟，它们二者的功能是有重叠的地方的，表示的含义也有相似之处。比如它们都定义了 Online 状态，其实都是表示正常工作状态下的分区状态。当然，这只是我根据源码功能做的一个大胆推测，至于是否会合并，我们拭目以待吧。

再多说一句，Partition 类是表征分区的对象。一个 Partition 实例定义和管理单个分区，它主要是利用 logManager 帮助它完成对分区底层日志的操作。ReplicaManager 类对于分区的管理，都是通过 Partition 对象完成的。

replicaFetcherManager

第三个比较关键的字段是 replicaFetcherManager。它的主要任务是**创建 ReplicaFetcherThread 类实例**。上节课，我们学习了 ReplicaFetcherThread 类的源码，它的主要职责是**帮助 Follower 副本向 Leader 副本拉取消息，并写入到本地日志中**。

下面展示了 ReplicaFetcherManager 类的主要方法 createFetcherThread 源码：

 复制代码

```
1 override def createFetcherThread(fetcherId: Int, sourceBroker: BrokerEndPoint)
2   val prefix = threadNamePrefix.map(tp => s"$tp:").getOrElse("")
3   val threadName = s"${prefix}ReplicaFetcherThread-$fetcherId-${sourceBroker.id}
4   // 创建ReplicaFetcherThread线程实例并返回
5   new ReplicaFetcherThread(threadName, fetcherId, sourceBroker, brokerConfig,
6     metrics, time, quotaManager)
7 }
```

该方法的主要目的是创建 ReplicaFetcherThread 实例，供 Follower 副本使用。线程的名字是根据 fetcherId 和 Broker ID 来确定的。ReplicaManager 类利用 replicaFetcherManager 字段，对所有 Fetcher 线程进行管理，包括线程的创建、启动、添加、停止和移除。

总结

这节课，我主要介绍了 ReplicaManager 类的定义以及重要字段。它们是理解后面 ReplicaManager 类管理功能的基础。

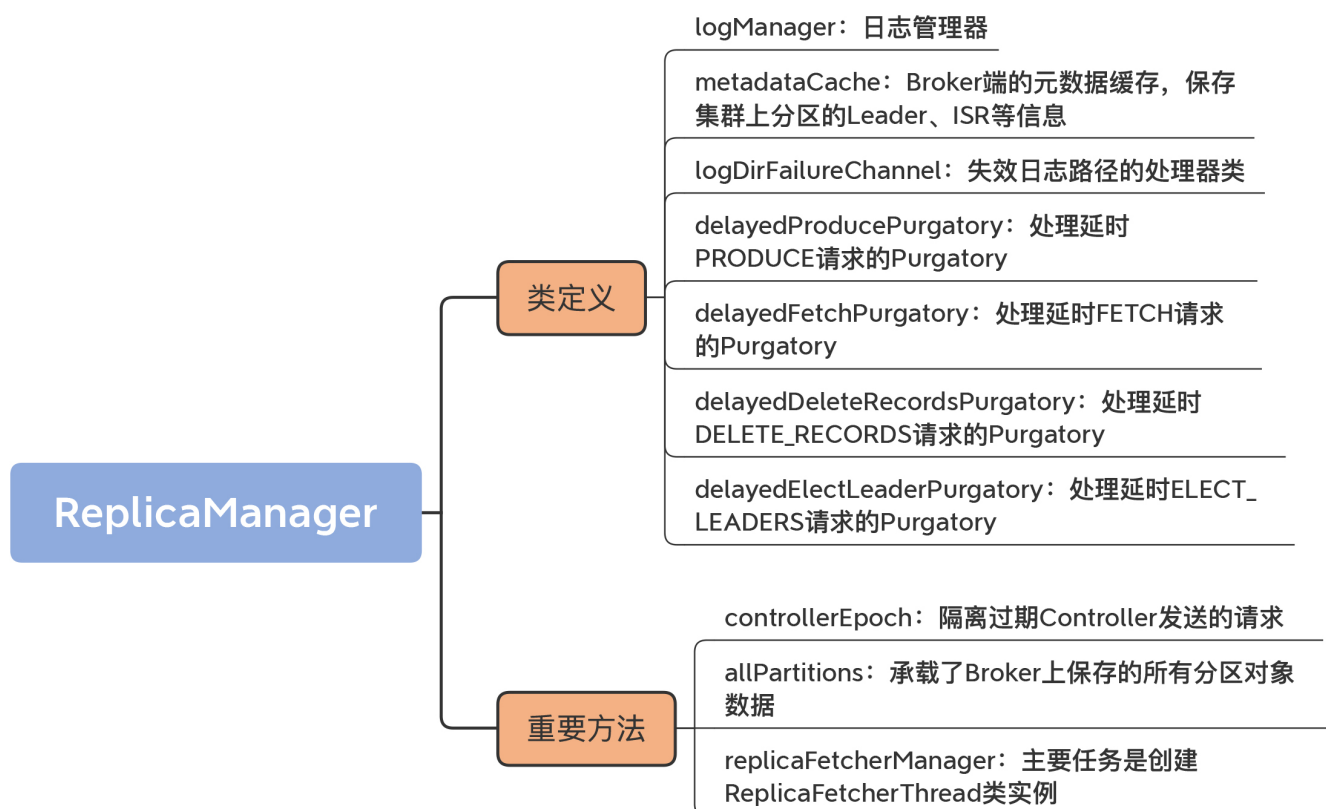
总的来说，ReplicaManager 类是 Kafka Broker 端管理分区和副本对象的重要组件。每个 Broker 在启动的时候，都会创建 ReplicaManager 实例。该实例一旦被创建，就会开始行使副本管理器的职责，对其下辖的 Leader 副本或 Follower 副本进行管理。

我们再简单回顾一下这节课的重点。

ReplicaManager 类：副本管理器的具体实现代码，里面定义了读写副本、删除副本消息的方法，以及其他的一些管理方法。

allPartitions 字段：承载了 Broker 上保存的所有分区对象数据。ReplicaManager 类通过它实现对分区下副本的管理。

replicaFetcherManager 字段：创建 ReplicaFetcherThread 类实例，该线程类实现 Follower 副本向 Leader 副本实时拉取消息的逻辑。



今天，我多次提到 ReplicaManager 是副本管理器这件事。实际上，副本管理中的两个重要功能就是读取副本对象和写入副本对象。对于 Leader 副本而言，Follower 副本需要读取它的消息数据；对于 Follower 副本而言，它拿到 Leader 副本的消息后，需要将消息写入到自己的底层日志上。那么，读写副本的机制是怎么样的呢？下节课，我们深入地探究一下 ReplicaManager 类重要的副本读写方法。

课后讨论

在 ReplicaManager 类中有一个 offlinePartitionCount 方法，它的作用是统计 Offline 状态的分区数，你能写一个方法统计 Online 状态的分区数吗？

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 22 | ReplicaFetcherThread：Follower拉取Leader消息是如何实现的？

下一篇 24 | ReplicaManager（中）：副本管理器是如何读写副本的？

精选留言 (2)

💬 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ReplicaFetcherThread类的源码。课后我请你思考updateFetchOffsetAndMaybeMarkTruncationComplete方法是做什么用的。其实这个方法的主要目的是将给定的一组分区去刷新Fetcher线程读取它们的位移值以及设置截断完成与否的状...

展开 ▾



1



伯安知心

2020-06-18

```
private def onlinePartitionCount:Int={  
  allPartitions.values.iterator.count(_ == HostedPartition.Online)  
}
```

作者回复: 题目出的好像有点简单了, 哈哈:)

