

21 | AbstractFetcherThread：拉取消息分几步？

2020-06-13 胡夕

Kafka核心源码解读

[进入课程 >](#)



讲述：胡夕

时长 17:58 大小 16.46M



你好，我是胡夕。从今天开始，我们正式进入到第 5 大模块“副本管理模块”源码的学习。

在 Kafka 中，副本是最重要的概念之一。为什么这么说呢？在前面的课程中，我曾反复提到过副本机制是 Kafka 实现数据高可靠性的基础。具体的实现方式就是，同一个分区下的多个副本分散在不同的 Broker 机器上，它们保存相同的消息数据以实现高可靠性。对于分布式系统而言，一个必须要解决的问题，就是如何确保所有副本上的数据是一致的。

针对这个问题，最常见的方案当属 Leader/Follower 备份机制（Leader/Follower Replication）。在 Kafka 中，分区的某个副本会被指定为 Leader，负责响应客户端的读写请求。其他副本自动成为 Follower，被动地同步 Leader 副本中的数据。



这里所说的被动同步，是指 Follower 副本不断地向 Leader 副本发送读取请求，以获取 Leader 处写入的最新消息数据。


那么在接下来的两讲，我们就一起学习下 Follower 副本是如何通过拉取线程做到这一点的。另外，Follower 副本在副本同步的过程中，还可能发生名为截断（Truncation）的操作。我们一并来看下它的实现原理。

课前案例

坦率地说，这部分源码非常贴近底层设计架构原理。你可能在想：阅读它对我实际有什么帮助吗？我举一个实际的例子来说明下。

我们曾经在生产环境中发现，一旦 Broker 上的副本数过多，Broker 节点的内存占用就会非常高。查过 HeapDump 之后，我们发现根源在于 ReplicaFetcherThread 文件中的 buildFetch 方法。这个方法里有这样一句：

```
1 val builder = fetchSessionHandler.newBuilder()
```


 复制代码

这条语句底层会实例化一个 LinkedHashMap。如果分区数很多的话，这个 Map 会被扩容很多次，因此带来了很多不必要的数据拷贝。这样既增加了内存的 Footprint，也浪费了 CPU 资源。

你看，通过查询源码，我们定位到了这个问题的根本原因。后来，我们通过将负载转移到其他 Broker 的方法解决了这个问题。

其实，Kafka 社区也发现了这个 Bug，所以当你现在再看这部分源码的时候，就会发现这行语句已经被修正了。它现在长这个样子，你可以体会下和之前有什么不同：

```
1 val builder = fetchSessionHandler.newBuilder(partitionMap.size, false)
```

 复制代码

你可能也看出来了，修改前后最大的不同，其实在于修改后的这条语句直接传入了 FETCH 请求中总的分区数，并直接将其传给 LinkedHashMap，免得再执行扩容操作了。

你看，有的时候改进一行源码就能解决实际问题。而且，你千万不要以为修改源码是一件多么神秘的事情，搞懂了原理之后，就可以有针对性地调整代码了，这其实是一件非常愉悦的事情。

好了，我们说回 Follower 副本从 Leader 副本拉取数据这件事儿。不知道你有没有注意到，我在前面的例子提到了一个名字：ReplicaFetcherThread，也就是副本获取线程。没错，Kafka 源码就是通过这个线程实现的消息拉取及处理。

今天这节课，我们先从抽象基类 AbstractFetcherThread 学起，看看它的类定义和三个重要方法。下节课，我们再继续学习 AbstractFetcherThread 类的一个重要方法，以及子类 ReplicaFetcherThread 的源码。这样，我们就能彻底搞明白 Follower 端同步 Leader 端消息的原理。

抽象基类：AbstractFetcherThread

等等，我们不是要学 ReplicaFetcherThread 吗？为什么要先从它的父类 AbstractFetcherThread 开始学习呢？

其实，这里的原因也很简单，那就是因为 AbstractFetcherThread 类是 ReplicaFetcherThread 的抽象基类。它里面定义和实现了很多重要的字段和方法，是我们学习 ReplicaFetcherThread 源码的基础。同时，AbstractFetcherThread 类的源码给出了很多子类需要实现的方法。

因此，我们需要事先了解这个抽象基类，否则便无法顺畅过渡到其子类源码的学习。

好了，我们来正式认识下 AbstractFetcherThread 吧。它的源码位于 server 包下的 AbstractFetcherThread.scala 文件中。从名字来看，它是一个抽象类，实现的功能是从 Broker 获取多个分区的消息数据，至于获取之后如何对这些数据进行处理，则交由子类来实现。

类定义及字段

我们看下 AbstractFetcherThread 类的定义和一些重要的字段：

```
1 abstract class AbstractFetcherThread(
```

复制代码

```

2  name: String, // 线程名称
3  clientId: String, // Client Id, 用于日志输出
4  val sourceBroker: BrokerEndPoint, // 数据源Broker地址
5  failedPartitions: FailedPartitions, // 处理过程中出现失败的分区
6  fetchBackOffMs: Int = 0, // 获取操作重试间隔
7  isInterruptible: Boolean = true, // 线程是否允许被中断
8  val brokerTopicStats: BrokerTopicStats) // Broker端主题监控指标
9  extends ShutdownableThread(name, isInterruptible) {
10 // 定义FetchData类型表示获取的消息数据
11 type FetchData = FetchResponse.PartitionData[Records]
12 // 定义EpochData类型表示Leader Epoch数据
13 type EpochData = OffsetsForLeaderEpochRequest.PartitionData
14 private val partitionStates = new PartitionStates[PartitionFetchState]
15 .....
16 }

```

我们来看一下 AbstractFetcherThread 的构造函数接收的几个重要参数的含义。

name: 线程名字。

sourceBroker: 源 Broker 节点信息。源 Broker 是指此线程要从哪个 Broker 上读取数据。

failedPartitions: 线程处理过程报错的分区集合。

fetchBackOffMs: 当获取分区数据出错后的等待重试间隔，默认是 Broker 端参数 replica.fetch.backoff.ms 值。

brokerTopicStats: Broker 端主题的各类监控指标，常见的有 MessagesInPerSec、BytesInPerSec 等。

这些字段中比较重要的是 **sourceBroker**，因为它决定 Follower 副本从哪个 Broker 拉取数据，也就是 Leader 副本所在的 Broker 是哪台。

除了构造函数的这几个字段外，AbstractFetcherThread 类还定义了两个 type 类型。用关键字 type 定义一个类型，属于 Scala 比较高阶的语法特性。从某种程度上，你可以把它当成一个快捷方式，比如 FetchData 这句：


```
1 type FetchData = FetchResponse.PartitionData[Records]
```

 复制代码

这行语句类似于一个快捷方式：以后凡是源码中需要用到 `FetchResponse.PartitionData[Records]` 的地方，都可以简单地使用 `FetchData` 替换掉，非常简洁方便。自定义类型 `EpochData`，也是同样的用法。

`FetchData` 定义里的 `PartitionData` 类型，是客户端 `clients` 工程中 `FetchResponse` 类定义的嵌套类。`FetchResponse` 类封装的是 `FETCH` 请求的 `Response` 对象，而里面的 `PartitionData` 类是一个 `POJO` 类，保存的是 `Response` 中单个分区数据拉取的各项数据，包括从该分区的 `Leader` 副本拉取回来的消息、该分区的高水位值和日志起始位移值等。

我们看下它的代码：

 复制代码

```
1 public static final class PartitionData<T extends BaseRecords> {
2     public final Errors error;           // 错误码
3     public final long highWatermark;     // 高水位值
4     public final long lastStableOffset;  // 最新LSO值
5     public final long logStartOffset;    // 最新Log Start Offset值
6     // 期望的Read Replica
7     // KAFKA 2.4之后支持部分Follower副本可以对外提供读服务
8     public final Optional<Integer> preferredReadReplica;
9     // 该分区对应的已终止事务列表
10    public final List<AbortedTransaction> abortedTransactions;
11    // 消息集合，最重要的字段！
12    public final T records;
13    // 构造函数.....
14 }
```

`PartitionData` 这个类定义的字段中，除了我们已经非常熟悉的 `highWatermark` 和 `logStartOffset` 等字段外，还有一些属于比较高阶的用法：

`preferredReadReplica`，用于指定可对外提供读服务的 `Follower` 副本；

`abortedTransactions`，用于保存该分区当前已终止事务列表；

`lastStableOffset` 是最新的 `LSO` 值，属于 `Kafka` 事务的概念。

关于这几个字段，你只要了解它们的基本作用就可以了。实际上，在 `PartitionData` 这个类中，最需要你重点关注的是 **`records` 字段**。因为它保存实际的消息集合，而这是我们最关

心的数据。

说到这里，如果你去查看 EpochData 的定义，能发现它也是 PartitionData 类型。但，你一定要注意的是，EpochData 的 PartitionData 是 OffsetsForLeaderEpochRequest 的 PartitionData 类型。

事实上，在 Kafka 源码中，有很多名为 PartitionData 的嵌套类。很多请求类型中的数据都是按分区层级进行分组的，因此源码很自然地在这些请求类中创建了同名的嵌套类。我们在查看源码时，一定要注意区分 PartitionData 嵌套类是定义在哪类请求中的，不同类型请求中的 PartitionData 类字段是完全不同的。


分区读取状态类

好了，我们把视线拉回到 AbstractFetcherThread 类。在这个类的构造函数中，我们看到它还封装了一个名为 PartitionStates[PartitionFetchState]类型的字段。

是不是看上去有些复杂？不过没关系，我们分开来看，先看它泛型的参数类型 PartitionFetchState 类。直观上理解，它是表征分区读取状态的，保存的是分区的已读取位移值和对应的副本状态。

注意这里的状态有两个，一个是分区读取状态，一个是副本读取状态。副本读取状态由 ReplicaState 接口表示，如下所示：

```
1 sealed trait ReplicaState
2 // 截断中
3 case object Truncating extends ReplicaState
4 // 获取中
5 case object Fetching extends ReplicaState
```

 复制代码

可见，副本读取状态有截断中和获取中两个：当副本执行截断操作时，副本状态被设置成 Truncating；当副本被读取时，副本状态被设置成 Fetching。

而分区读取状态有 3 个，分别是：

可获取，表明副本获取线程当前能够读取数据。

截断中，表明分区副本正在执行截断操作（比如该副本刚刚成为 Follower 副本）。

被推迟，表明副本获取线程获取数据时出现错误，需要等待一段时间后重试。

值得注意的是，分区读取状态中的可获取、截断中与副本读取状态的获取中、截断中两个状态并非严格对应的。换句话说，副本读取状态处于获取中，并不一定表示分区读取状态就是可获取状态。对于分区而言，它是否能够被获取的条件要比副本严格一些。

接下来，我们就来看看这 3 类分区获取状态的源码定义：

 复制代码

```
1 case class PartitionFetchState(fetchOffset: Long,
2   lag: Option[Long],
3   currentLeaderEpoch: Int,
4   delay: Option[DelayedItem],
5   state: ReplicaState) {
6   // 分区可获取的条件是副本处于Fetching且未被推迟执行
7   def isReadyForFetch: Boolean = state == Fetching && !isDelayed
8   // 副本处于ISR的条件：没有lag
9   def isReplicaInSync: Boolean = lag.isDefined && lag.get <= 0
10  // 分区处于截断中状态的条件：副本处于Truncating状态且未被推迟执行
11  def isTruncating: Boolean = state == Truncating && !isDelayed
12  // 分区被推迟获取数据的条件：存在未过期的延迟任务
13  def isDelayed: Boolean =
14    delay.exists(_.getDelay(TimeUnit.MILLISECONDS) > 0)
15    .....
16 }
```

这段源码中有 4 个方法，你只要重点了解 `isReadyForFetch` 和 `isTruncating` 这两个方法即可。因为副本获取线程做的事情就是这两件：日志截断和消息获取。

至于 `isReplicaInSync`，它被用于副本限流，出镜率不高。而 `isDelayed`，是用于判断是否需要推迟获取对应分区的消息。源码会不断地调整那些不需要推迟的分区的读取顺序，以保证读取的公平性。

这个公平性，其实就是在 `partitionStates` 字段的类型 `PartitionStates` 类中实现的。这个类是在 `clients` 工程中定义的。它本质上会接收一组要读取的主题分区，然后以轮询的方式依次读取这些分区以确保公平性。

鉴于咱们这门儿课聚焦于 Broker 端源码，因此，这里我只是简单和你说下这个类的实现原理。如果你想要深入理解这部分内容，可以翻开 clients 端工程的源码，自行去探索下这部分的源码。

 复制代码

```
1 public class PartitionStates<S> {
2     private final LinkedHashMap<TopicPartition, S> map = new LinkedHashMap<>()
3     .....
4     public void updateAndMoveToEnd(TopicPartition topicPartition, S state) {
5         map.remove(topicPartition);
6         map.put(topicPartition, state);
7         updateSize();
8     }
9     .....
10 }
```

前面说过了，PartitionStates 类用轮询的方式来处理要读取的多个分区。那具体是怎么实现的呢？简单来说，就是依靠 LinkedHashMap 数据结构来保存所有主题分区。

LinkedHashMap 中的元素有明确的迭代顺序，通常就是元素被插入的顺序。

假设 Kafka 要读取 5 个分区上的消息：A、B、C、D 和 E。如果插入顺序就是 ABCDE，那么自然首先读取分区 A。一旦 A 被读取之后，为了确保各个分区都有同等机会被读取到，代码需要将 A 插入到分区列表的最后一位，这就是 updateAndMoveToEnd 方法要做的事情。

具体来说，就是把 A 从 map 中移除掉，然后再插回去，这样 A 自然就处于列表的最后一位了。大体上，PartitionStates 类就是做这个用的。


重要方法

说完了 AbstractFetcherThread 类的定义，我们再看下它提供一些的重要方法。

这个类总共封装了近 40 个方法，那接下来我就按照这些方法对于你使用 Kafka、解决 Kafka 问题的重要程度，精选出 4 个方法做重点讲解，分别是 processPartitionData、truncate、buildFetch 和 doWork。这 4 个方法涵盖了拉取线程所做的最重要的 3 件事儿：构建 FETCH 请求、执行截断操作、处理拉取后的结果。而 doWork 方法，其实是串联起了前面的这 3 个方法。

好了，我们一个一个来看看吧。

首先是它最重要的方法 `processPartitionData`，用于处理读取回来的消息集合。它是一个抽象方法，因此需要子类实现它的逻辑。具体到 Follower 副本而言，是由 `ReplicaFetcherThread` 类实现的。以下是它的方法签名：

 复制代码

```
1 protected def processPartitionData(  
2     topicPartition: TopicPartition, // 读取哪个分区的数据  
3     fetchOffset: Long,              // 读取到的最新位移值  
4     partitionData: FetchData        // 读取到的分区消息数据  
5 ): Option[LogAppendInfo]           // 写入已读取消息数据前的元数据
```

我们需要重点关注的字段是，该方法的返回值 `Option[LogAppendInfo]`：

对于 Follower 副本读消息写入日志而言，你可以忽略这里的 `Option`，因为它肯定会返回具体的 `LogAppendInfo` 实例，而不会是 `None`。

至于 `LogAppendInfo` 类，我们在“日志模块”中已经介绍过了。它封装了很多消息数据被写入到日志前的重要元数据信息，比如首条消息的位移值、最后一条消息位移值、最大时间戳等。

除了 `processPartitionData` 方法，**另一个重要的方法是 `truncate` 方法**，其签名代码如下：

 复制代码

```
1 protected def truncate(  
2     topicPartition: TopicPartition, // 要对哪个分区下副本执行截断操作  
3     truncationState: OffsetTruncationState // Offset + 截断状态  
4 ): Unit
```

这里的 `OffsetTruncationState` 类封装了一个位移值和一个截断完成与否的布尔值状态。它的主要作用是，告诉 Kafka 要把指定分区下副本截断到哪个位移值。

第 3 个重要的方法是 `buildFetch` 方法。代码如下：

```
1 protected def buildFetch(  
2     // 一组要读取的分区列表  
3     // 分区是否可读取取决于PartitionFetchState中的状态  
4     partitionMap: Map[TopicPartition, PartitionFetchState]):  
5     // 封装FetchRequest.Builder对象  
6     ResultWithPartitions[Option[ReplicaFetch]]
```

[复制代码](#)

buildFetch 方法的返回值看似很复杂，但其实如果你阅读源码的话，就会发现 buildFetch 的本质就是，为指定分区构建对应的 FetchRequest.Builder 对象，而该对象是构建 FetchRequest 的核心组件。Kafka 中任何类型的消息读取，都是通过给指定 Broker 发送 FetchRequest 请求来完成的。

第 4 个重要的方法是 doWork。虽然它的代码行数不多，但却是**串联前面 3 个方法的主要入口方法，也是 AbstractFetcherThread 类的核心方法。**因此，我们要多花点时间，弄明白这些方法是怎么组合在一起共同工作的。我会在下节课和你详细拆解这里的代码原理。

总结

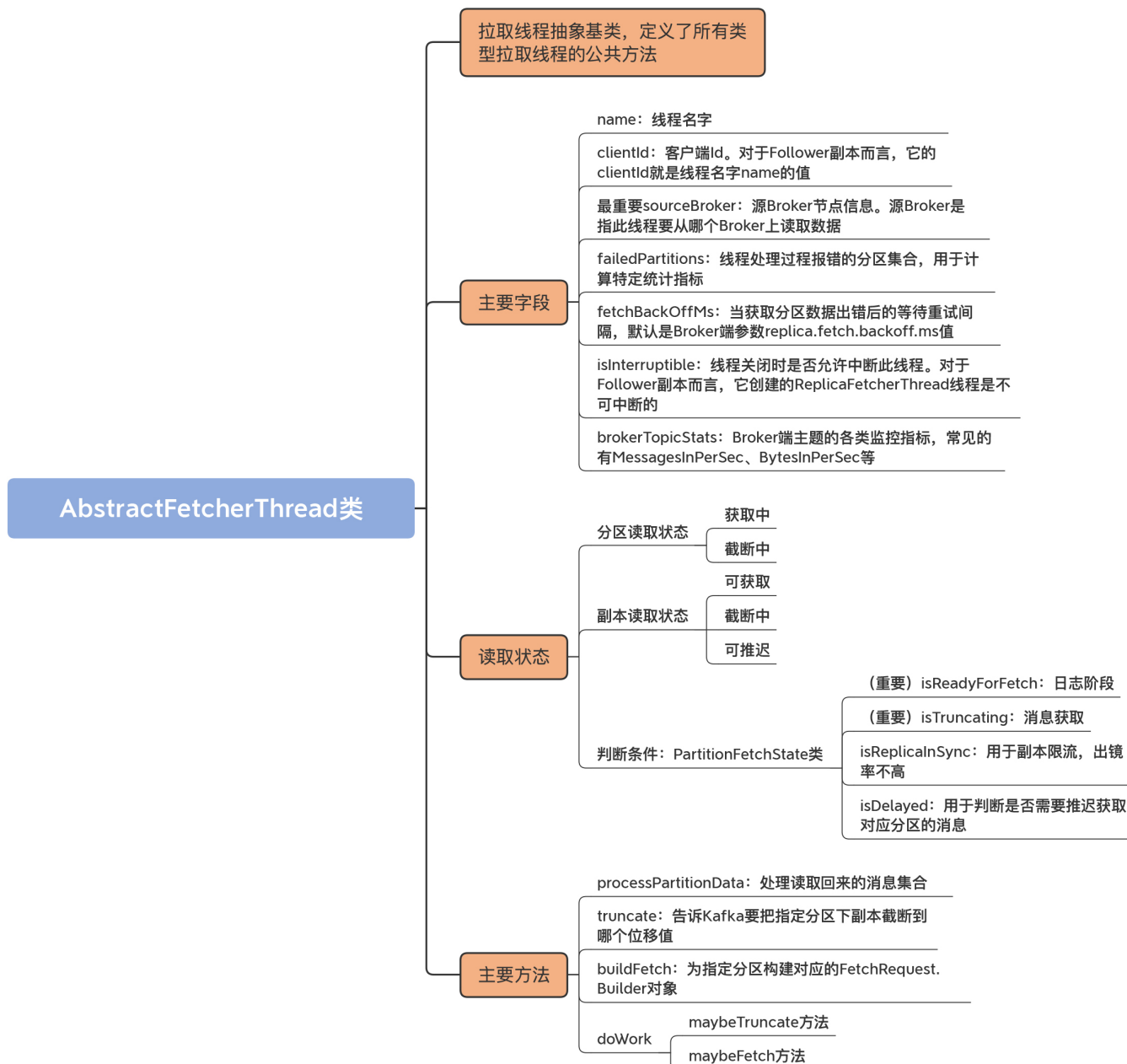
今天，我们主要学习了 Kafka 的副本同步机制和副本管理器组件。目前，Kafka 副本之间的消息同步是依靠 ReplicaFetcherThread 线程完成的。我们重点阅读了它的抽象基类 AbstractFetcherThread 线程类的代码。作为拉取线程的公共基类，AbstractFetcherThread 类定义了很多重要方法。

我们来回顾一下这节课的重点。

AbstractFetcherThread 类：拉取线程的抽象基类。它定义了公共方法来处理所有拉取线程都要实现的逻辑，如执行截断操作，获取消息等。

拉取线程逻辑：循环执行截断操作和获取数据操作。

分区读取状态：当前，源码定义了 3 类分区读取状态。拉取线程只能拉取处于可读取状态的分区的数据。



下节课，我会带你一起对照着 doWork 方法的代码，把拉取线程的完整执行逻辑串联一遍，这样的话，我们就能彻底掌握 Follower 副本拉取线程的工作原理了。在这个过程中，我们还会陆续接触到 ReplicaFetcherThread 类源码的 3 个重要方法的代码。你需要理解它们的实现机制以及 doWork 是怎么把它们组织在一起的。

课后讨论

请简单描述一下 handlePartitionsWithErrors 方法的实现原理。

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

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



涨价倒计时 🕒

今日秒杀 **¥79**，6月13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 20 | DelayedOperation: Broker是怎么延时处理请求的？

下一篇 22 | ReplicaFetcherThread: Follower拉取Leader消息是如何实现的？

精选留言 (2)

写留言



胡夕 置顶

2020-06-15

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，咱们结合源码重点了解了Broker是如何处理延时请求的。课后我挑了advanceClock方法中的一个语句让你去分析它的目的。这行语句计算你的事是watcherList中总的已完成的请求数。另外在遍历watcherList的同时，代码还会将这些已完成的任务从链表中移...
展开 ∨



伯安知心

2020-06-14

handlePartitionsWithErrors方法简单来说就是，如果可能出现leader切换，延迟处理带错误的partitions，并且延迟处理过程加了中断优先处理的锁，
展开 ▾

作者回复: 👍

