

21 | Kafka Consumer源码分析：消息消费的实现过程

李玥 · 消息队列高手课



你好，我是李玥。

我们在上节课中提到过，用于解决消息队列一些常见问题的知识和原理，最终落地到代码上，都包含在收、发消息这两个流程中。对于消息队列的生产和消费这两个核心流程，在大部分消息队列中，它实现的主要流程都是一样的，所以，通过这两节课的学习之后，掌握了这两个流程的实现过程。无论你使用的是哪种消息队列，遇到收发消息的问题，你都可以用同样的思路去分析和解决问题。

上一节课我和你一起通过分析源代码学习了 RocketMQ 消息生产的实现过程，本节课我们来看一下 Kafka 消费者的源代码，理清 Kafka 消费的实现过程，并且能从中学习到一些 Kafka 的优秀设计思路和编码技巧。

在开始分析源码之前，我们一起来回顾一下 Kafka 消费模型的几个要点：

Kafka 的每个 Consumer（消费者）实例属于一个 ConsumerGroup（消费组）；

在消费时，ConsumerGroup 中的每个 Consumer 独占一个或多个 Partition（分区）；


对于每个 ConsumerGroup，在任意时刻，每个 Partition 至多有 1 个 Consumer 在消费；

每个 ConsumerGroup 都有一个 Coordinator(协调者) 负责分配 Consumer 和 Partition 的对应关系，当 Partition 或是 Consumer 发生变更时，会触发 rebalance（重新分配）过程，重新分配 Consumer 与 Partition 的对应关系；

Consumer 维护与 Coordinator 之间的心跳，这样 Coordinator 就能感知到 Consumer 的状态，在 Consumer 故障的时候及时触发 rebalance。

掌握并理解 Kafka 的消费模型，对于接下来理解其消费的实现过程是至关重要的，如果你对上面的这些要点还有不清楚的地方，建议回顾一下之前的课程或者看一下 Kafka 相关的文档，然后再继续接下来的内容。


我们使用当前最新的版本 2.2 进行分析，使用 Git 在 GitHub 上直接下载源码到本地：

 复制代码

```
1 git clone git@github.com:apache/kafka.git
2 cd kafka
3 git checkout 2.2
```

在《[09 | 学习开源代码该如何入手？](#)》这节课中，我讲过，分析国外源码最好的方式就是从文档入手，接下来我们就找一下 Kafka 的文档，看看从哪儿来入手开启我们的分析流程。

Kafka 的 Consumer 入口类 [KafkaConsumer](#) 的 [JavaDoc](#)，给出了关于如何使用 KafkaConsumer 非常详细的说明文档，并且给出了一个使用 Consumer 消费的最简代码示例：

 复制代码

```
1 // 设置必要的配置信息
2 Properties props = new Properties();
3 props.put("bootstrap.servers", "localhost:9092");
4 props.put("group.id", "test");
5 props.put("enable.auto.commit", "true");
6 props.put("auto.commit.interval.ms", "1000");
7 props.put("key.deserializer", "org.apache.kafka.common.serialization.StringD
```

```
8      props.put("value.deserializer", "org.apache.kafka.common.serialization.String");
9
10     // 创建Consumer实例
11     KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
12
13     // 订阅Topic
14     consumer.subscribe(Arrays.asList("foo", "bar"));
15
16     // 循环拉消息
17     while (true) {
18         ConsumerRecords<String, String> records = consumer.poll(100);
19         for (ConsumerRecord<String, String> record : records)
20             System.out.printf("offset = %d, key = %s, value = %s\n", record.offset, record.key, record.value);
21     }
```

这段代码主要的主要流程是：

1. 设置必要的配置信息，包括：起始连接的 Broker 地址，Consumer Group 的 ID，自动提交消费位置的配置和序列化配置；
2. 创建 Consumer 实例；
3. 订阅了 2 个 Topic：foo 和 bar；
4. 循环拉取消息并打印在控制台上。


通过上面的代码实例我们可以看到，消费这个大的流程，在 Kafka 中实际上是被分成了“订阅”和“拉取消息”这两个小的流程。另外，我在之前的课程中反复提到过，Kafka 在消费过程中，每个 Consumer 实例是绑定到一个分区上的，那 Consumer 是如何确定，绑定到哪一个分区上的呢？这个问题也是可以通过分析消费流程来找到答案的。所以，我们分析整个消费流程主要聚焦在三个问题上：

1. 订阅过程是如何实现的？
2. Consumer 是如何与 Coordinator 协商，确定消费哪些 Partition 的？
3. 拉取消息的过程是如何实现的？

了解前两个问题，有助于你充分理解 Kafka 的元数据模型，以及 Kafka 是如何在客户端和服务端之间来交换元数据的。最后一个问题，拉取消息的实现过程，实际上就是消费的主要流程，我们上节课讲过，这是消息队列最核心的两个流程之一，也是必须重点掌握的。我们就带着这三个问题，来分析 Kafka 的订阅和拉取消息的过程如何实现。

订阅过程如何实现？

我们先来看看订阅的实现流程。从上面的例子跟踪到订阅的主流程方法：

 复制代码

```
1  public void subscribe(Collection<String> topics, ConsumerRebalanceListener list
2      acquireAndEnsureOpen();
3      try {
4          // 省略部分代码
5
6          // 重置订阅状态
7          this.subscriptions.subscribe(new HashSet<>(topics), listener);
8
9          // 更新元数据
10         metadata.setTopics(subscriptions.groupSubscription());
11     } finally {
12         release();
13     }
14 }
```

在这个代码中，我们先忽略掉各种参数和状态检查的分支代码，订阅的主流程主要更新了两个属性：一个是订阅状态 `subscriptions`，另一个是更新元数据中的 topic 信息。订阅状态 `subscriptions` 主要维护了订阅的 topic 和 partition 的消费位置等状态信息。属性 `metadata` 中维护了 Kafka 集群元数据的一个子集，包括集群的 Broker 节点、Topic 和 Partition 在节点上分布，以及我们聚焦的第二个问题：Coordinator 给 Consumer 分配的 Partition 信息。

请注意一下，这个 `subscribe()` 方法的实现有一个非常值得大家学习的地方：就是开始的 `acquireAndEnsureOpen()` 和 `try-finally release()`，作用就是保护这个方法只能单线程调用。


Kafka 在文档中明确地注明了 Consumer 不是线程安全的，意味着 Consumer 被并发调用时会出现不可预期的结果。为了避免这种情况发生，Kafka 做了主动的检测并抛出异常，而不是

放任系统产生不可预期的情况。

Kafka“**主动检测不支持的情况并抛出异常，避免系统产生不可预期的行为**”这种模式，对于增强的系统的健壮性是一种非常有效的做法。如果你的系统不支持用户的某种操作，正确的做法是，检测不支持的操作，直接拒绝用户操作，并给出明确的错误提示，而不应该只是在文档中写上“不要这样做”，却放任用户错误的操作，产生一些不可预期的、奇怪的错误结果。

具体 Kafka 是如何实现的并发检测，大家可以看一下方法 `acquireAndEnsureOpen()` 的实现，很简单也很经典，我们就不再展开讲解了。

继续跟进到更新元数据的方法 `metadata.setTopics()` 里面，这个方法的实现除了更新元数据类 `Metadata` 中的 `topic` 相关的一些属性以外，还调用了 `Metadata.requestUpdate()` 方法请求更新元数据。

 复制代码

```
1     public synchronized int requestUpdate() {
2         this.needUpdate = true;
3         return this.updateVersion;
4     }
```

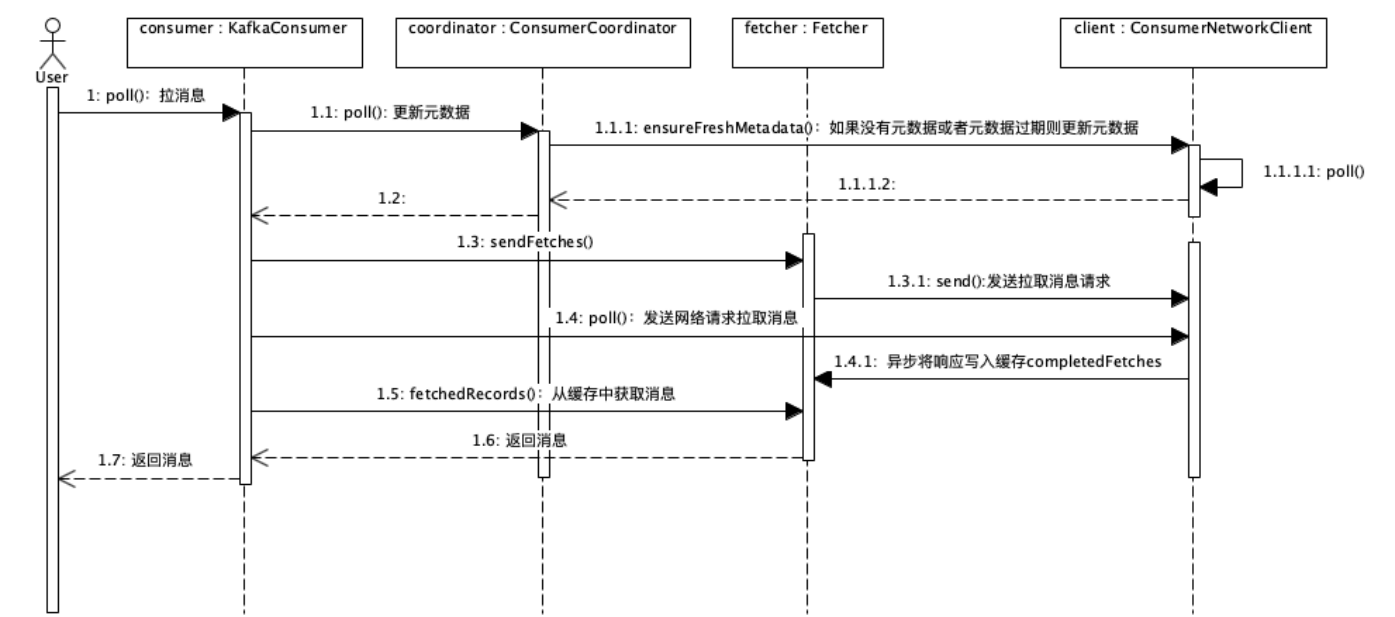
跟进到 `requestUpdate()` 的方法里面我们会发现，这里面并没有真正发送更新元数据的请求，只是将需要更新元数据的标志位 `needUpdate` 设置为 `true` 就结束了。Kafka 必须确保在第一次拉消息之前元数据是可用的，也就是说在第一次拉消息之前必须更新一次元数据，否则 Consumer 就不知道它应该去哪个 Broker 上去拉哪个 Partition 的消息。

分析完订阅相关的代码，我们来总结一下：在订阅的实现过程中，Kafka 更新了订阅状态 `subscriptions` 和元数据 `metadata` 中的相关 `topic` 的一些属性，将元数据状态置为“需要立即更新”，但是并没有真正发送更新元数据的请求，整个过程没有和集群有任何网络数据交换。

那这个元数据会在什么时候真正做一次更新呢？我们可以先带着这个问题接着看代码。

拉取消息的过程如何实现？

接下来，我们分析拉取消息的流程。这个流程的时序图如下（点击图片可放大查看）：



我们对着时序图来分析它的实现流程。在 `KafkaConsumer.poll()` 方法（对应源码 1179 行）的实现里面，可以看到主要是先后调用了 2 个私有方法：

- 1. `updateAssignmentMetadataIfNeeded()`: 更新元数据。
- 2. `pollForFetches()`: 拉取消息。


方法 `updateAssignmentMetadataIfNeeded()` 中，调用了 `coordinator.poll()` 方法，`poll()` 方法里面又调用了 `client.ensureFreshMetadata()` 方法，在 `client.ensureFreshMetadata()` 方法中又调用了 `client.poll()` 方法，实现了与 Cluster 通信，在 Coordinator 上注册 Consumer 并拉取和更新元数据。至此，“元数据会在什么时候真正做一次更新”这个问题也有了答案。

类 `ConsumerNetworkClient` 封装了 Consumer 和 Cluster 之间所有的网络通信的实现，这个类是一个非常彻底的异步实现。它没有维护任何的线程，所有待发送的 Request 都存放在属性 `unsent` 中，返回的 Response 存放在属性 `pendingCompletion` 中。每次调用 `poll()` 方法的时候，在当前线程中发送所有待发送的 Request，处理所有收到的 Response。

我们在之前的课程中讲到过，这种异步设计的优势就是用很少的线程实现高吞吐量，劣势也非常明显，极大增加了代码的复杂度。对比上节课我们分析的 RocketMQ 的代码，Producer 和

Consumer 在主要收发消息流程上功能的复杂度是差不多的，但是你可以很明显地感受到 Kafka 的代码实现要比 RocketMQ 的代码实现更加的复杂难于理解。

我们继续分析方法 pollForFetches() 的实现。

 复制代码

```
1      private Map<TopicPartition, List<ConsumerRecord<K, V>>> pollForFetches(Timer
2          // 省略部分代码
3          // 如果缓存里面有未读取的消息，直接返回这些消息
4          final Map<TopicPartition, List<ConsumerRecord<K, V>>> records = fetcher.f
5          if (!records.isEmpty()) {
6              return records;
7          }
8          // 构造拉取消息请求，并发送
9          fetcher.sendFetches();
10         // 省略部分代码
11         // 发送网络请求拉取消息，等待直到有消息返回或者超时
12         client.poll(pollTimer, () -> {
13             return !fetcher.hasCompletedFetches();
14         });
15         // 省略部分代码
16         // 返回拉到的消息
17         return fetcher.fetchedRecords();
18     }
```

这段代码的主要实现逻辑是：

1. 如果缓存里面有未读取的消息，直接返回这些消息；
2. 构造拉取消息请求，并发送；
3. 发送网络请求并拉取消息，等待直到有消息返回或者超时；
4. 返回拉到的消息。

在方法 fetcher.sendFetches() 的实现里面，Kafka 根据元数据的信息，构造到所有需要的 Broker 的拉消息的 Request，然后调用 client.Send() 方法将这些请求异步发送出去。并且，注册了一个回调类来处理返回的 Response，所有返回的 Response 被暂时存放在

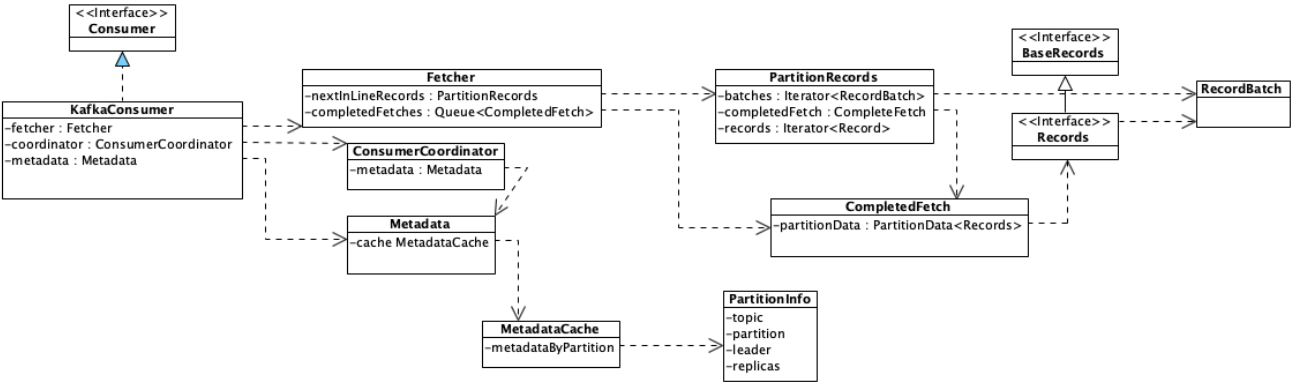
Fetcher.completedFetches 中。需要注意的是，这时的 Request 并没有被真正发给各个 Broker，而是被暂存在了 client.unsend 中等待被发送。

然后，在调用 client.poll() 方法时，会真正将之前构造的所有 Request 发送出去，并处理收到的 Response。

最后，fetcher.fetchedRecords() 方法中，将返回的 Response 反序列化后转换为消息列表，返回给调用者。

综合上面的实现分析，我在这里给出整个拉取消息的流程涉及到的相关类的类图，在这个类图中，为了便于你理解，我并没有把所有类都绘制上去，只是把本节课两个流程相关的主要类和这些类里的关键属性画在了图中。你可以配合这个类图和上面的时序图进行代码阅读。

类图（点击图片可放大查看）：



小结

本节课我们一起分析了 Kafka Consumer 消费消息的实现过程。大家来分析代码过程中，不仅仅是要掌握 Kafka 整个消费的流程是如何实现的，更重要的是理解它这种完全异步的设计思想。

发送请求时，构建 Request 对象，暂存入发送队列，但不立即发送，而是等待合适的时机批量发送。并且，用回调或者 RequestFuture 方式，预先定义好如何处理响应的逻辑。在收到

Broker 返回的响应之后，也不会立即处理，而是暂存在队列中，择机处理。那这个择机策略就比较复杂了，有可能是需要读取响应的时候，也有可能是缓冲区满了或是时间到了，都有可能触发一次真正的网络请求，也就是在 poll() 方法中发送所有待发送 Request 并处理所有 Response。

这种设计的好处是，不需要维护用于异步发送的和处理响应的线程，并且能充分发挥批量处理的优势，这也是 Kafka 的性能非常好的原因之一。这种设计的缺点也非常的明显，就是实现的复杂度太大了，如果没有深厚的代码功力，很难驾驭这么复杂的设计，并且后续维护的成本也很高。

总体来说，不推荐大家把代码设计得这么复杂。代码结构简单、清晰、易维护是我们在设计过程中需要考虑的一个非常重要的因素。很多时候，为了获得较好的代码结构，在可接受的范围内，去牺牲一些性能，也是划算的。

思考题

我们知道，Kafka Consumer 在消费过程中是需要维护消费位置的，Consumer 每次从当前消费位置拉取一批消息，这些消息都被正常消费后，Consumer 会给 Coordinator 发一个提交位置的请求，然后消费位置会向后移动，完成一批消费过程。那 kafka Consumer 是如何维护和提交这个消费位置的呢？请你带着这个问题再回顾一下 Consumer 的代码，尝试独立分析代码并找到答案。欢迎在留言区与我分享讨论。

感谢阅读，如果你觉得这篇文章对你有一些启发，也欢迎把它分享给你的朋友。

© 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。

精选留言 (29)



业余草

2019-09-12

我们需要一些速成的实战经验，比如消息队列突然延迟2个小时该如何解决？希望带着这些类似的问题，结合设计原理，帮助我们层层解析。。。

**Peter**

2019-11-01

课后作业，希望老师指正。

在基础篇03的时候讲过消费位置是消息队列服务器针对每个消费组和每个队列维护的一个位置变量。那么也就是说最终真正更新这个位置变量应该是交由服务器去执行的，而Consumer只是发送一个请求。那么顺着这个思路，我猜应该是在更新元数据的时候就应该发送这个请求，原因很简单：消费者需要知道“从哪发起”并且“发多少”，因此这时就已经知道了应该将消费位置更新为多少了，所以这时候就可以发送这个请求了。至于服务器最终会将消费位置更新为多少，还取决于客户端返回的结果。

在方法updateAssignmentMetadataIfNeeded中，最后一行return updateFetchPositions(timer);

从updateFetchPositions这个方法点进去，看到coordinator.refreshCommittedOffsetsIfNeeded(timer)

这个方法点进去之后会看到fetchCommittedOffsets方法，进这个方法，找到sendOffsetFetchRequest，点进去，最终会发现 client.send(coordinator, requestBuilder)

作者回复: 细致👍

**DFighting**

2019-09-17

offset的提交不知道是不是在kafkaConsumer.commitAsync中调用coordinator.commitOffsetsAsync(offsets,callback)

1、这里设计成异步方式一开始我是比较奇怪的，他是如何保证offset不丢失呢？看了代码才知道在异步返回前会等待ConcurrentLinkQueue<offsetCommitCompletion>中没有其他的待处理的其他的offset的commit后，才会返回，这里的非阻塞队列是线程安全的，可以避免当前提交冲掉其他的offset的提交

2、真正进行提交的时候也不是调用什么具体操作net的接口，而是向另一个ConcurrentLinkedQueue中注册了一个RequestFutureListener的监听者，当然注册之前使用了AtomicInteger来保证并发安全。

3、每个监听者应该都会由相应的Coordinator轮询处理队列中的待提交请求，将offset提交从具体的Consumer中解耦到每个组的Coordinator中。

当然以上只是个人理解，如有不当欢迎指正。

读完这个代码，我发现kafka在这里保证并发数据一致性时，使用了安全的数据结构+CAS的

数据访问，灵活且大大降低了锁机制的粗粒度带来的性能损耗，只是这个代码真不容易写好，真是大牛作品！！



👍 8



鲁班大师

2020-06-01

老师，kafak consumer 在reblance期间，如何实现不重复消费

作者回复: 实现“不重复消费”是非常困难的，你需要做的就是让你的consumer具备幂等性，这样即使发生重复消费，也不会对系统数据产生任何影响。

共 3 条评论 >

👍 6



山头

2019-09-30

老师，你好，broker和消费端都重启了，消费端还知道从哪个offset开始消费吗

作者回复: 从服务端的协调者获取。服务端的协调者会记录主题的每个消费组的每个分区当前的消费位置

共 4 条评论 >

👍 6



于成龙

2020-09-10

分析了下acquireAndEnsureOpen如何加锁，供大家参考

```
private void acquireAndEnsureOpen() {
    acquire();
    //KafkaConsumer成员变量，初始值为false，调用close(Duration)方法后才会置为true
    if (this.closed) {
        release();
        throw new IllegalStateException("This consumer has already been closed.");
    }
}
```

//变量声明

```
private static final long NO_CURRENT_THREAD = -1L;
```

```
private void acquire() {  
    //拿到当前线程的线程id  
    long threadId = Thread.currentThread().getId();  
    /*if threadId与当前正执行的线程的id不一致（并发，多线程访问） && threadId对应的线程没有争抢到锁
```

then 抛出异常

举例：

现在有两个KafkaConsumer线程，线程id分别是thread1, thread2，要执行acquire()方法。

thread1先启动，执行完上面这条语句、赋值threadId后， thread1栈帧中threadId=thread1，此时CPU线程调度、执行thread2，

thread2也走到if语句时，在thread2的栈帧中，threadId已经赋值为thread2，走到这里，currentThread作为成员变量，初始值为NO_CURRENT_THREAD（-1），因此必然不相等，继续走第二判断条件，即利用AtomicInteger的CAS操作，将当前线程id threadId(thread2)赋值给currentThread这个AtomicInteger，必然返回true，因此会继续执行，使得refcount加1；

接着，此时执行thread1，那么再继续执行if，threadId(thread1) != currentThread.get() (thread2)能满足，但是currentThread的CAS赋值将会失败，因此此时currentThread的值并不是NO_CURRENT_THREAD。

refcount用于记录重入锁的情况，参见release()方法，当refcount=0时，currentThread将重新赋值为NO_CURRENT_THREAD，保证彻底解锁。

```
*/  
    if (threadId != currentThread.get() && !currentThread.compareAndSet(NO_CURRENT_THREAD, threadId))  
        throw new ConcurrentModificationException("KafkaConsumer is not safe for multi-threaded access");  
    refcount.incrementAndGet();  
}
```

共 1 条评论 >

👍 3



凌空飞起的剪刀腿

2020-04-13

老师您好：

kafka consumer中没有分析到心跳线程是怎么处理的，我看源代码上写的是单独开了一个后台线程负责心跳，这样处理的优势是什么啊？

作者回复: Kafka Consumer与服务端的协调者维护心跳, 而协调者所在的Broker不一定和接收消息的Broker是同一个实例。

所以, 必须得分开。



👍 3



鲁班大师

2020-05-21

每个 ConsumerGroup 都有一个 Coordinator(协调者) 负责分配 Consumer 和 Partition 的对应关系, 当 Partition 或是 Consumer 发生变更时, 会触发 rebalance (重新分配) 过程, 重新分配 Consumer 与 Partition 的对应关系;在rebalance期间应该是不能消费的吧

作者回复: 是的, 消费会暂停, 直到Rebalance完成。



👍 2



SKang

2020-03-19

老师 我看完之后 可以理解为 消费组A 消费一个cc主题的消息, 然后过程中我将消费组A 的名字改成消费组B后, 不会出现重复消费, 只会接着A的 继续消费剩下的吧 我认为毕竟A已经成功消费了 偏移量已经成功被更新了吧

作者回复: 不是的, 不同消费组的偏移量是分开记录的。



👍 1



山头

2019-09-13

消费者如何从服务端拉取消息的, 用for循环效率太低吧, 能否说说实际的代码

作者回复: 同学, 我们这节课通篇就是讲得这个问题啊, 给你们讲解使用的就是实际的代码。

共 5 条评论 >

👍 1



leslie

2019-09-12

先打卡：代码慢慢研究；老师今天讲述了研究代码的目的：

1.消息队列实现的主要流程都一样，掌握流程的实现过程；遇到收发消息的问题，都可以用同样的思路去分析和解决问题。

2.看一下源代码，理清消费生产的实现过程，从中学习一些优秀的设计思路和编码技巧。

这个算是一个小的总结吧：明天中秋休息刚好可以啃代码，好好研究代码去体会。

明天是中秋佳节：愿老师节日快乐^_^



小杨

2022-10-20 来自北京

我有个疑问，1个topic，3个partition，增加consumer数量能提升消费速度么？或者说kafka应该如何提升消费能力。期待老师解答。



tianbingJ

2022-03-01

大部分内容都在讨论网络、CAS、异步啊等等知识对于实现一个系统多么重要；但是，各种框架实现的场景都会用到这些内容，总不能所有的课程都先罗列一遍这些内容吧。

即使介绍，花个两三节课差不多就行了，占比过大；结果就是跑题了，没有聚焦在MQ上。



Sam Fu

2021-11-24

老师 我最近看了rocketmq消费的源码，您看看我的理解对不对。

rocketmq consumer消费完消息后，其实不管成功或失败都会提交这批信息的最大位移。如果存在失败的消息，则会将整个这一批消息全部发到重试队列去。这样的话，之前消费过的消息就会重复消费了。

所以每批拉取的消息设置的不能太大，否则有一条失败整个都得重试，重试率会增高



对与错

2020-10-13

请问kafka消费者使用手动提交位移的方式，当前消费进度为10，然后消费几条失败之后，提交位移失败，后面消费新的消息成功之后，当前消费进度被更新为15，那中间消费失败的几条消息会随着重启消费者而重新消费吗？位移主题里面的消费进度会随着重启消费者而被删除吗？如果不被删除，那应该不会重新消费失败的那几条消息吧？

共 1 条评论 >



Geek_411c57

2020-07-21

老师您好，我想问下：客户端拉取消息的时候是同步拉取吗？如果是同步拉取的话应该会占用连接吧？为什么不是用异步监听io事件呢？



蛤蟆先生

2020-06-15

有个问题请教一下老师，目前我们公司某个应用在生产环境一共有两台机器，这时候有一台机器挂了，但是某个消息还是会经常消费到这台挂的机器上，导致消息没有消费成功，这是为什么呢？

作者回复：可以贴一下Topic的具体配置，大家一起帮你分析一下原因。



鲁班大师

2020-05-21

多个consumer消费同一个partition会有什么问题么

作者回复：如果是不同的group.id，互相之间没有影响。

共 3 条评论 >



lizhibo

2020-05-13

老师好，kafka消息要是在消费端消费出现异常了怎么办，他没有再次消费的机制，比如1分之后再去消费，这个怎么实现

作者回复：所以消费的时候，如果不能接受丢消息，一定不要设置成自动提交消费位置。这样下次拉取的时候，还会拉到这个位置的消息。

```
ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000L));  
// 执行消费业务逻辑，然后再提交消费位置。  
consumer.commitSync();
```


共 4 条评论 >



凡

2020-02-24

提交位置是在ConsumerCoordinator类提供了同步异步提交方法，具体提交位置可以查看到调用这几个方法的位置

