图文 092、案例实战：每天10亿数据的日志分析系统的OOM问题排查实践！

详情 评论

案例实战：
每天10亿数据的日志清洗系统的OOM问题排查实践！

狸猫技术窝专栏上新，基于**真实订单系统**的消息中间件（mq）实战，重磅推荐：

未来3个月，我的好朋友原子弹大侠将带你一起，全程实战，360度死磕MQ

（点击下方蓝字进行试听）

从 0 开始带你成为消息中间件实战高手

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情在评论区留言提问，我都会逐一答疑

（**ps**：评论区还精选了一些小伙伴对**专栏每日思考题的作答**，有的答案真的非常好！大家可以通过看别人的思路，启发一下自己，从而加深理解）

如何加群：购买了狸猫技术窝专栏的小伙伴都可以加入**狸猫技术交流群**。

（群里有不少**一二线互联网大厂的助教**，大家可以一起讨论交流各种技术）

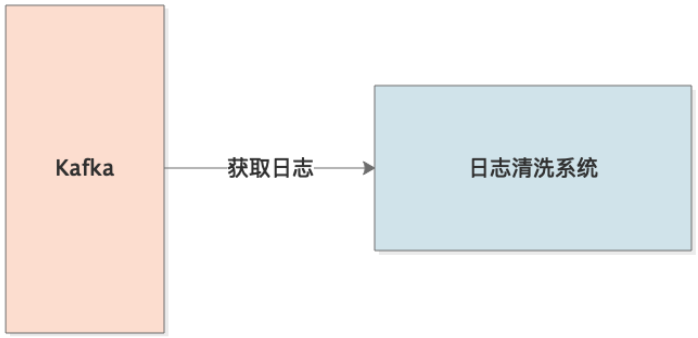
具体**加群方式**请参见文末。

（**注**：以前通过其他专栏加过群的同学就不要重复加了）

1、案例背景引入

今天的案例背景是一个每天10亿数据量的日志清洗系统，这个系统做的事情其实非常的简单，他主要就是从Kafka中不停的消费各种日志数据，然后对日志的格式进行很多清洗，比如对一些涉及到用户敏感信息的字段（姓名、手机号、身份证号）进行脱敏处理，然后把清洗后的数据交付给其他的系统去使用。

比如推荐系统、广告系统、分析系统都会去使用这些清洗好的数据，我们先看下面的图，大致就知道这个系统的运行情况了。



2、事故发生现场

某天我们也是突然收到线上的报警，发现日志清洗系统发生了OOM的异常！

我们登陆到线上机器查看日志之后，发现还是那么经典的java.lang.OutOfMemoryError: java heap space的问题，又是堆内存溢出。

此时我们当然就会来分析一下问题到底出在哪里了，大家应该还记得我们分析OOM问题的套路

首先先看看异常日志，去定位一下到底是谁导致的这个问题，当时我们在日志里大致看到了类似如下的一些信息：

```
java.lang.OutOfMemoryError: java heap space
xx.xx.xx.log.clean.XXClass.process()
xx.xx.xx.log.clean.XXClass.xx()
xx.xx.xx.log.clean.XXClass.xx()
xx.xx.xx.log.clean.XXClass.process()
xx.xx.xx.log.clean.XXClass.xx()
xx.xx.xx.log.clean.XXClass.xx()
xx.xx.xx.log.clean.XXClass.process()
xx.xx.xx.log.clean.XXClass.xx()
xx.xx.xx.log.clean.XXClass.xx()
```

当然大量无关紧要的日志信息可以直接忽略掉了，毕竟当时也没有截图，直接看上面最关键的一些信息

大家可以很明显的发现，似乎同样的一个方法（XXClass.process()）反复出现了多次，最终导致了堆内存溢出的问题。

这个时候通过日志，有经验的朋友可能已经可以发现一个问题了，那就是在某一处代码出现了大量的递归操作。正是大量的递归操作之后，也就是反复调用一个方法之后，导致了堆内存溢出的问题。

初步是大致定位出来问题所在了，接着当然我们就得去用MAT分析一下内存快照了。

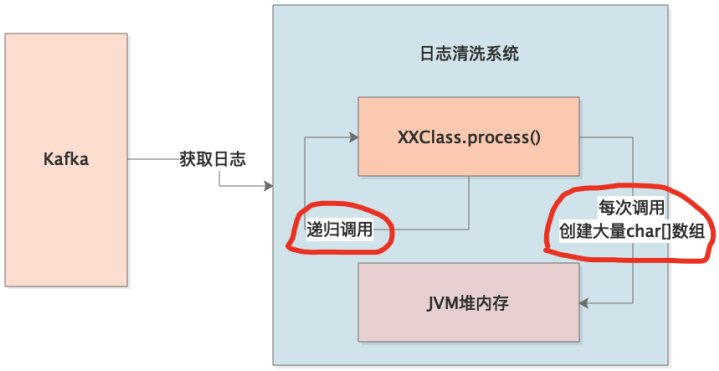
3、初步分析内存快照

接着我们开始分析生产现场的内存快照，之前我们已经详细讲解了如何通过MAT去分析内存快照，快速定位创建大量对象的代码和方法，其实在日志中我们是可以看到是哪个方法导致的内存溢出，但是我们通过日志不知道到底是哪个方法调用创建了大量的对象。

因此最终无论如何，还是得通过MAT去分析一下，在分析的时候，我们就发现了一个问题，因为有大量的XXClass.process()方法的递归执行，每个XXClass.process()中都创建了大量的char数组！

最后因为XXClass.process()方法又多次递归调用，也就导致了大量的char[]数组耗尽了内存。

先看看下图，在图里我们表示出来了方法递归调用，每次调用都创建大量char[]数组导致的内存溢出问题。



4、功夫在诗外：问题在JVM参数上

基本定位出了问题所在了，但是先别着急直接去代码中检查问题所在，因为我们当时发现了一个比较大的问题。

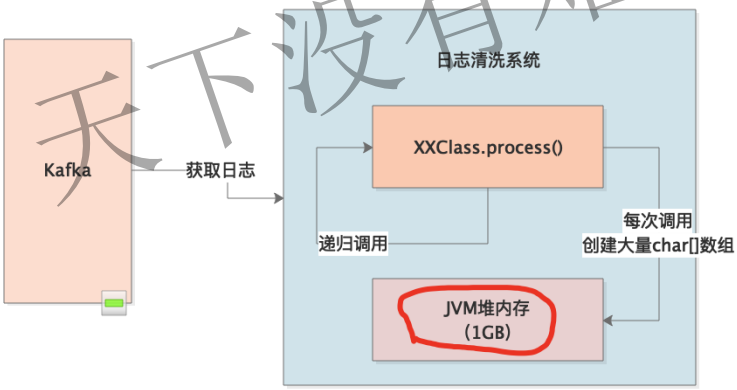
虽然XXClass.process()方法递归调用了多次，但是实际上我们在MAT中发现递归调用的次数也并不是很多，大概也就是十几次递归调用到最多几十次递归调用而已

而且我们观察了一下，所有递归调用加起来创建的char[]数组对象总和其实也就最多1G而已。

如果是这样的话，其实我们应该先注意一个问题，那就是可能这次OOM的发生不一定是代码就写的有多么的烂，可能就是我们的JVM的内存参数设置的不对，给堆内存分配的空间太小了！

如果要是给JVM堆内存分配更大的空间呢？一切都要尝试一下，所以先别着急，慢慢来。

先看下面的图，在里面我们表示出来了这个内存过小的问题。



5、分析一下JVM的GC日志

如果你要知道这个堆内存到底是不是设置太小了，就得先分析一下JVM运行时的内存使用模型。

现在系统已经宕机了，我们唯一可以看到的，就是当时在JVM启动参数中加入的自动记录的GC日志了。

从GC日志中，我们是可以看到JVM启动时的完整参数设置的，核心的内容如下所示：

-Xmx1024m -Xms1024m -XX:+PrintGCDetails -XX:+PrintGC() -XX:+HeapDumpOnOutOfMemoryError -Xloggc:/opt/logs/gc.log -XX:HeapDumpPath=/opt/logs/dump

大家可以看到，这里主要是把gc日志详细记录在了一个日志文件里，另外指定了内存溢出的时候要导出内存快照，另外就是堆内存给的是1GB大小，但是要知道这台机器可是4核8G的！

接着我们看一下当时记录下来的gc.log日志。

[Full GC (Allocation Failure) 866M->654M(1024M)]
[Full GC (Allocation Failure) 843M->633M(1024M)]
[Full GC (Allocation Failure) 855M->621M(1024M)]
[Full GC (Allocation Failure) 878M->612M(1024M)]

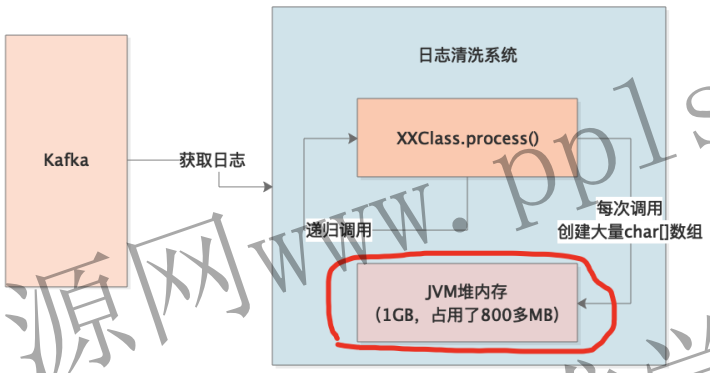
我把GC日志中大量的无关紧要的信息省略掉了，因为跟我们分析关系不大

但是大家可以发现一点，因为Allocation Failure触发的Full GC很多，也就是堆内存无法分配新的对象了，然后触发GC，结果触发的时候肯定是先触发Full GC了，这个关于Full GC触发的原因和时机之前大量的分析过，这里不多说了。

而且你会发现每次Full GC都只能回收掉一点点对象，发现堆内存几乎都是占满了。

另外我们这里没有显示时间，当时日志里显示的是每秒钟都会执行一次Full GC，这个就很可怕了。基本上我们可以明确一点，应该是在系统运行的时候，因为XXClass.process()方法不停递归创建了大量的char[]数组，导致堆内存几乎是塞满的。

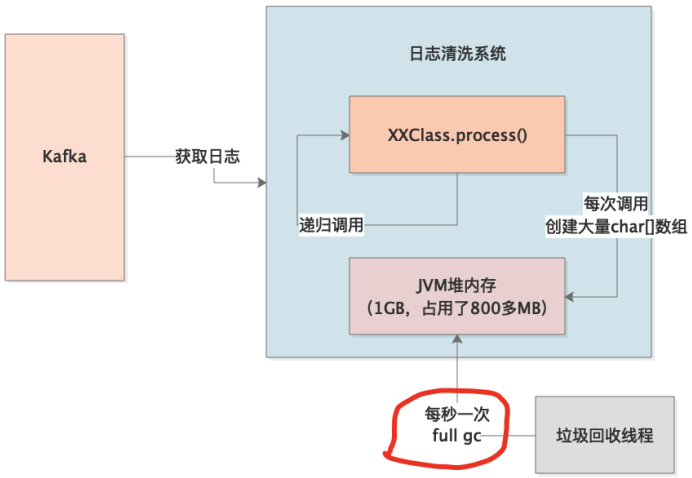
我们先看下面的图，表示出了这一点。



然后这就导致了连续一段时间，每秒触发一次Full GC，因为内存都满了，特别是老年代可能几乎都满了，所以可能是每秒钟执行young gc之前，发现老年代可用空间不足，就会提前触发full gc

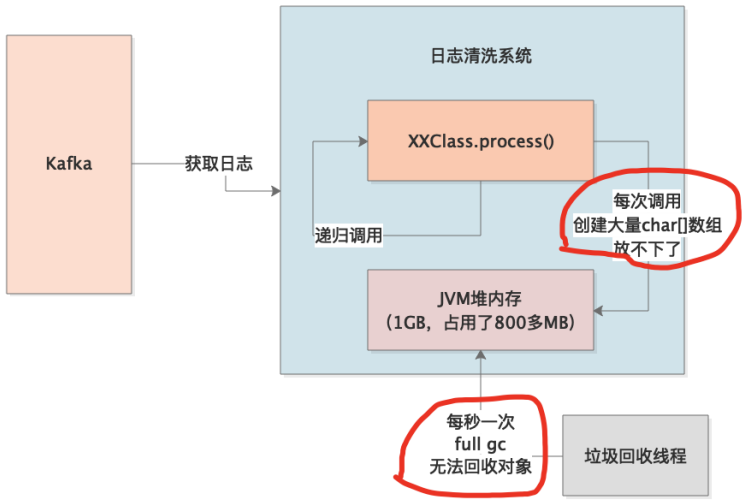
也可能是young gc过后存活对象太多无法放入Survivor中，都要进入老年代，放不下了，只能进行full gc。

我们看下图，表示出来了每秒钟执行一次full gc的场景。



但是每次full gc只能回收少量对象，直到最后可能某一次full gc回收不掉任何对象了，然后新的对象无法放入堆内存了，此时就会触发OOM内存溢出的异常。

我们看下面的图，表示出来了这个过程。



分析到这里不知道大家有什么感觉？其实很明显一点，就是堆内存肯定是偏小了，这个就导致频繁的full gc。

6、分析一下JVM运行时内存使用模型

接着我们再用jstat分析一下当时JVM运行时的内存模型，当时我们重启了系统，每秒钟打印一次jstat的统计信息，就看到了下面的情况：

S0 S1 E O YGC FGC

0 100 57 69 36 0
0 100 57 69 36 0
0 100 65 69 37 0
0 100 0 99 37 0
0 100 0 87 37 1

我就给出部分信息大家就可以看出来问题所在了，刚开始都是年轻代的Eden区在涨，接着YGC从36到37，就是发生了一次YGC，接着Old区直接从占比69%到99%

说明什么？

说明YGC后存活对象太多，Survivor放不下，直接进老年代了！

接着老年代都占了99%了，直接就触发了一次Full GC，但是也仅仅让老年代从占比99%到87%而已，回收了少量的对象。

上面的那个过程反复循环几次，大家思考一下，年轻代的对象反复进入老年代，不停的触发Full GC，但是还回收不了多少对象，几次循环过后，老年代满了，可能Full GC没回收多少对象，新的对象一大批放不下了，就触发OOM了。

7、优化第一步：增加堆内存大小

所以这个OOM的问题，说白了不能直接说是代码问题，从JVM运行情况以及内存大小来看，就是内存分配不足的问题。

因此这里第一步，直接在4核8G的机器上，给堆内存加大空间，直接给了堆内存5G的内存。

接着运行系统，通过jstat观察，就可以发现，每次Young GC过后存活对象都落入Survivor区域了，不会随便进入老年代，而且因为堆内存很大，基本上运行一段时间不会发生OOM问题了。

8、优化第二步：改写代码

另外就是改写代码，让他不要占用过多的内存。当时代码之所以递归，就是因为在一条日志中，可能会出现很多用户的信息，一条日志也许会合并包含了十几个到几十个用户的信息。

这个时候代码中就是会递归十几次到几十次去处理这个日志，每次递归都会产生大量的char[]数组，是切割了日志用来处理的。

其实这个代码写的完全没有必要，因为对每一条日志，如果发现包含了多个用户的信息，其实就对这一条日志切割出来进行处理就可以了，完全没有必要递归调用，每次调用都切割一次日志，生成大量的char[]数组。

所以把这一步代码优化了之后，一下子发现线上系统的内存使用情况降低了10倍以上。

9、案例总结

今天这个案例，大家会发现，我们先是通过OOM的排查方法去分析，发现主要是内存太小导致的问题

然后用gc日志和jstat分析，明显发现是内存不够用了，最后加大系统内存，并且优化代码就可以了。

End

狸猫技术窝专栏上新，基于**真实订单系统**的消息中间件（mq）实战，重磅推荐：

未来3个月，我的好朋友原子弹大侠将带你一起，全程实战，360度死磕MQ

（点击下方蓝字进行试听）

从 0 开始带你成为消息中间件实战高手

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情在评论区留言提问，我都会逐一答疑

（ps：评论区还精选了一些小伙伴对**专栏每日思考题的作答**，有的答案真的非常好！大家可以通过看别人的思路，启发一下自己，从而加深理解**）**

如何加群：购买了狸猫技术窝专栏的小伙伴都可以加入**狸猫技术交流群**。

（群里有不少**一二线互联网大厂的助教**，大家可以一起讨论交流各种技术）

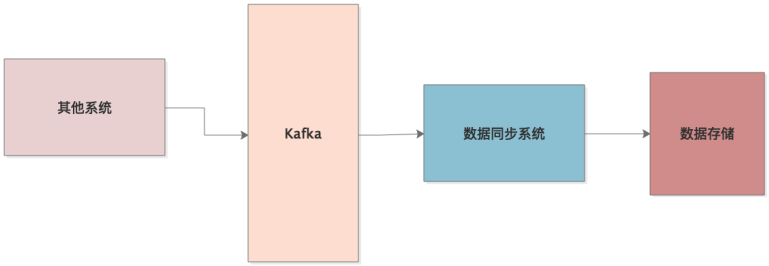
具体**加群方式**请参见文末。

（**注**：以前通过其他专栏加过群的同学就不要重复加了）

1、案例背景

线上有一个数据同步系统，是专门负责从另外一个系统去同步数据的，简单来说，另外一个系统会不停的发布自己的数据到Kafka中去，然后我们有一个数据同步系统就专门从Kafka里消费数据，接着保存到自己的数据库中去，大概就是这样的一个流程。

我们看下图，就是这个系统运行的一个流程。



结果就这么一个非常简单的系统，居然时不时就报一个内存溢出的错误，然后就得重启系统，过了一段时间又会再次内存溢出一下。而且这个系统处理的数据量是越来越大，因此我们发现他内存溢出的频率越来越高，到这个情况，就必须要处理一下了。

2、经验丰富的工程师：从现象看到本质

其实一般遇到这种现象，只要是经验丰富的工程师，应该已经可以具备从现象看到本质的能力了。我们可以来分析和思考一下，他既然是每次重启过后都会在一段时间以后出现内存溢出的问题，说明肯定是每次重启过后，内存都会不断的上涨。

而且一般要搞到JVM出现内存溢出，通常的就是两种情况，要不然是并发太高，瞬间大量并发创建过多的对象，导致系统直接崩溃了。要不就是有内存泄漏之类的问题，就是很多对象都赖在内存里，无论你如何GC就是回收不掉。

那么这个场景可能是怎么回事呢？我们当时分析了一下，这个系统的负载并不是很高，随着数据量不少，但是并不是那种瞬时高并发的场景。那么很可能就是随着时间推移，有某种对象越来越多，赖在内存里了。

然后不断的触发gc，结果每次gc都回收不掉这些对象。

一直到最后，内存实在不足了，就会内存溢出，我们看看下面的图，在下图里就画出了这个问题。

3、通过jstat来确认我们的推断

接着直接在一次重启系统之后，用jstat观察了一下JVM运行的情况：

发现老年代的对象一直在增长，不停的在增长，每次Young GC过后，老年代的对象就会增长不少，而且当老年代的使用率达到100%之后，我们发现会正常触发Full GC，但是Full GC根本回收不掉任何对象。

导致老年代使用率还是100%！

然后老年代使用率维持100%一段时间过后，就会爆出内存溢出的问题，因为再有新的对象进入老年代，实在是没有空间放他了！

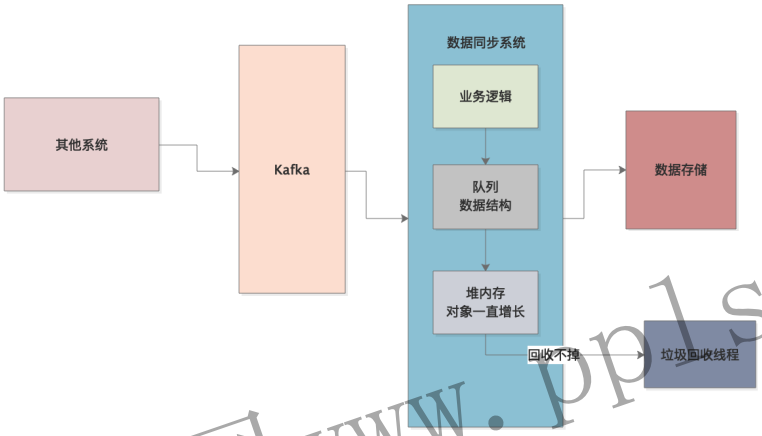
所以基本就确认了我们的判断，每次系统启动，不知道什么对象会一直进入堆内存，而且随着Young GC执行，对象会一直进入老年代，最后触发Full GC都无法回收老年代的对象，最终就是内存溢出。

4、通过MAT找到占用内存最大的对象！

关于MAT分析内存快照的方法，之前已经讲解的很详细了，其实在这些案例中就不用重复一些截图了，直接说出过程和结论就好！在内存快照中，我们发现了一个问题，那就是有一个队列数据结构，直接引用了大量的数据，就是这个队列数据结构占满了内存！

那么这个队列是干什么用的？简单来说，从Kafka消费出来的数据会先写入这个队列，接着从这个队列再慢慢写入数据库中，这个主要是要额外做一些中间的数据处理和转换，所以自己在中间又加了一个队列。

我们看下面的图。



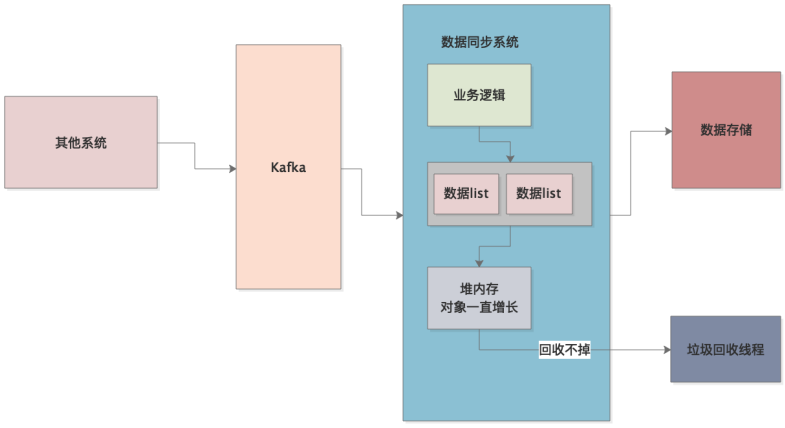
那么这个队列是怎么用的？问题就出在这里了！

大家都知道，从Kafka消费数据，是可以一下子消费一批出来的，比如消费几百条数据粗来。因此当时这个写代码的同学，直接就是每次消费几百条数据出来给做成一个List，然后把这个List放入到队列里去！

最后就搞成了，一个队列比如有1000个元素，每个元素都是一个List，每个List里都有几百条数据！这种做法怎么行？会导致内存中的队列里积压几十万条，甚至百万条数据！最终一定会导致内存溢出！

而且只要你数据还停留在队列中，就是没有办法被回收的。

我们看下面的图。



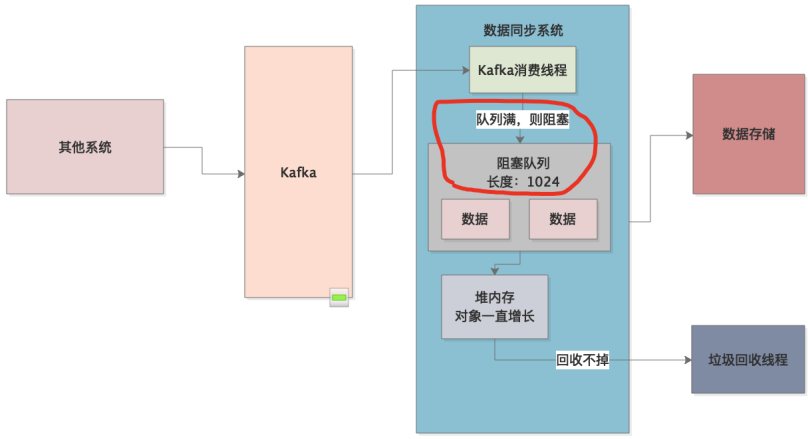
其实上图就是一个典型的对生产和消费的速率没控制好的例子。从Kafka里消费出来数据放入队列的速度很快，但是从队列里消费数据进行处理然后写入存储的速度较慢，最终会导致内存队列快速积压数据，导致内存溢出。

而且这种队列每个元素都是一个List的做法，会导致内存队列能容纳的数据量大幅度膨胀。

最终解决这个问题也很简单，把上述内存队列的使用修改了一下，做成了定长的阻塞队列，比如最多1024个元素，然后每次从Kafka消费出来数据，一条一条数据写入队列，而不是做成一个List放入队列作为一个元素。

因此这样内存中最多就是1024个数据，一旦内存队列满了，此时Kafka消费线程就会停止工作，因为被队列给阻塞住了。不会说让内存队列中的数据过多。

我们看下面的图。



5、本文小结

本文是我们整个专栏的最后一个案例，其实相信大家认真学习完这个专栏之后，就会感受到我们设计这个专栏的思路。专栏的核心是通过一步一图和大白话的方式，让大家学会JVM的核心运行原理，接着学习了JVM GC优化的核心原理和OOM问题的核心原理。

接着我们给大家讲解了JVM的GC问题以及OOM的常见发生场景和解决方法。

同时我们带给了大家数十个来源于我们真实生产环境的JVM优化案例，包括GC优化案例和OOM优化案例，大量的优化案例让大家可以对各种不同场景的问题有一个了解，同时积累起来了对不同问题进行分析、排查和解决的一个思路。

在这个过程中，如果大家反复去把这些案例看几遍，吸收透彻了，本质上就会积累起来较为丰富的JVM优化实践的经验积累，当你日后真的在工作中需要解决JVM问题的时候，就会发现这些知识全部都可以派上用场了。

End
