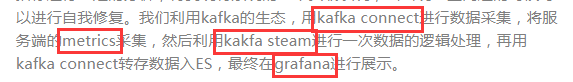
# 简介：

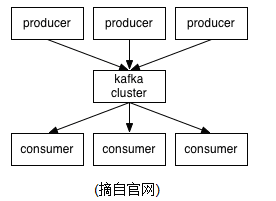


[kafka实战](https://www.cnblogs.com/hei12138/p/7805475.html)

[kafka入门：简介、使用场景、设计原理、主要配置及集群搭建（转）](https://www.cnblogs.com/likehua/p/3999538.html)

https://mp.weixin.qq.com/s/eyXr9Df6GcfvdYHpy\_qyZg

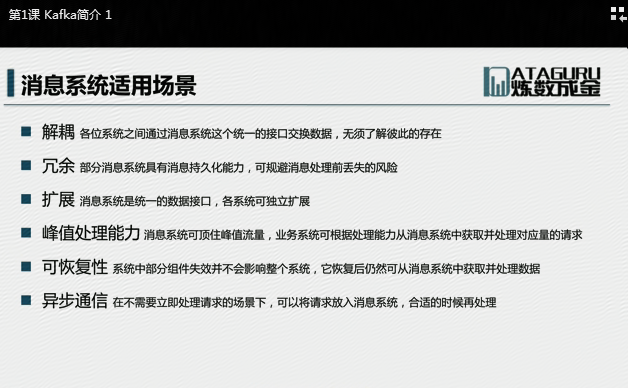
1. kafka对消息保存时根据Topic进行归类，发送消息者成为Producer,消息接受者成为Consumer,此外kafka集群有多个kafka实例组成，每个实例([server](http://cpro.baidu.com/cpro/ui/uijs.php?rs=1&u=http%3A%2F%2Fwww%2Eaboutyun%2Ecom%2Fthread%2D9341%2D1%2D1%2Ehtml&p=baidu&c=news&n=10&t=tpclicked3_hc&q=92051019_cpr&k=server&k0=java&kdi0=8&k1=%B1%E0%B3%CC&kdi1=8&k2=%BF%CD%BB%A7%B6%CB&kdi2=8&k3=%C9%E8%BC%C6&kdi3=8&k4=server&kdi4=1&sid=4ebca4a25f27e407&ch=0&tu=u1692056&jk=fb2f0911808fa875&cf=29&fv=14&stid=9&urlid=0&luki=5&seller_id=1&di=128))成为broker。无论是kafka集群，还是producer和consumer都依赖于zookeeper来保证系统可用性集群保存一些meta信息。



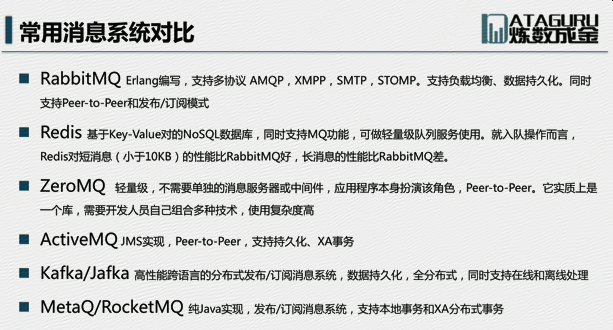
2． Topics/logs

一个Topic可以认为是一类消息，每个topic将被分成多个partition(区),每个partition在存储层面是append log文件。任何发布到此partition的消息都会被直接追加到log文件的尾部，每条消息在文件中的位置称为offset（偏移量），offset为一个long型数字，它是唯一标记一条消息。它唯一的标记一条消息。kafka并没有提供其他额外的索引机制来存储offset，因为在kafka中几乎不允许对消息进行“随机读写”。、

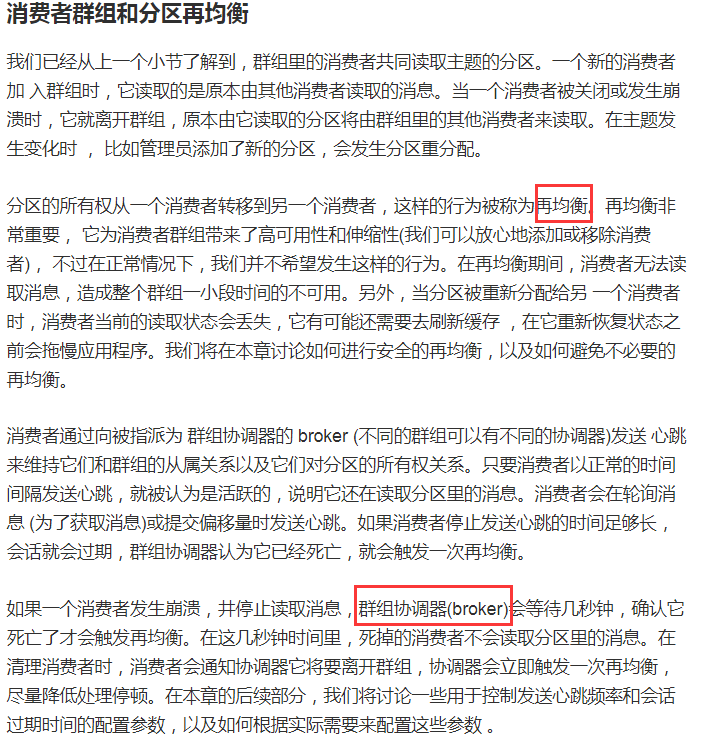
消息队列存在的好处：

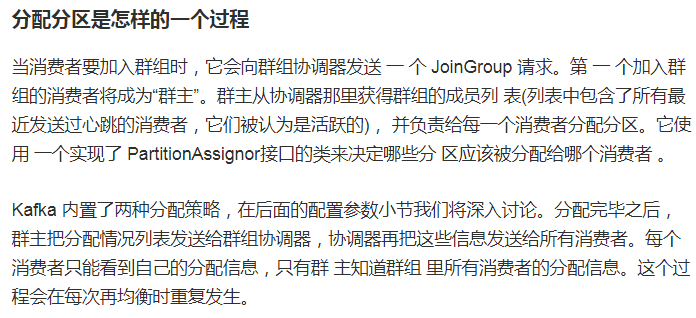


常用的消息系统：



# kafka配置





| **NAME** | **DESCRIPTION** | **TYPE** | **DEFAULT** | **VALID VALUES** | **IMPORTANCE** |
| --- | --- | --- | --- | --- | --- |
| bootstrap.servers | 用于建立到Kafka集群的初始连接的主机/端口对列表。客户端将使用所有服务器，而不管在这里指定哪些服务器用于引导 - 该列表仅影响用于发现全套服务器的初始主机。这个清单应该在表格中host1:port1,host2:port2,...。由于这些服务器仅用于初始连接以发现完整的群集成员资格（可能会动态更改），因此此列表不必包含整套服务器（但可能需要多个服务器，以防服务器关闭） 。 | list |  |  | high |
| key.deserializer | 实现org.apache.kafka.common.serialization.Deserializer接口的密钥的反序列化器类。 | class |  |  | high |
| value.deserializer | 用于实现org.apache.kafka.common.serialization.Deserializer接口的值的反序列化器类。 | class |  |  | high |
| fetch.min.bytes | 服务器为获取请求返回的最小数据量。如果没有足够的数据可用，请求将等待那么多的数据在应答请求之前积累。1字节的默认设置意味着只要有一个字节的数据可用，或者提取请求超时等待数据到达，就会立即应答提取请求。将其设置为大于1的值将导致服务器等待大量的数据累积，这可以稍稍提高服务器吞吐量，但需要花费一些额外的延迟时间。 当consumer向一个broker发起fetch请求时，broker返回的records的大小最小值。如果broker中数据量不够的话会wait，直到数据大小满足这个条件。取值范围是：[0, Integer.Max]，默认值是1。默认值设置为1的目的是：使得consumer的请求能够尽快的返回。 | int | 1 | [0,...] | high |
| group.id | 标识此消费者所属的消费者群组的唯一字符串。如果消费者通过使用subscribe(topic)基于卡夫卡的偏移量管理策略来使用组管理功能，则此属性是必需的 | string | "" |  | high |
| heartbeat.interval.ms | 心跳间隔。心跳是在consumer与coordinator之间进行的。心跳是确定consumer存活，加入或者退出group的有效手段。这个值必须设置的小于session.timeout.ms，因为：当Consumer由于某种原因不能发Heartbeat到coordinator时，并且时间超过session.timeout.ms时，就会认为该consumer已退出，它所订阅的partition会分配到同一group 内的其它的consumer上。通常设置的值要低于session.timeout.ms的1/3。 | int | 3000 |  | high |
| max.partition.fetch.bytes | 一次fetch请求，从一个partition中取得的records最大大小。如果在从topic中第一个非空的partition取消息时，如果取到的第一个record的大小就超过这个配置时，仍然会读取这个record，也就是说在这片情况下，只会返回这一条record。 broker、topic都会对producer发给它的message size做限制。所以在配置这值时，可以参考broker的message.max.bytes 和 topic的max.message.bytes的配置。 | int | 1048576 | [0,...] | high |
| session.timeout.ms | 使用Kafka的组管理设施时，用于检测消费者失败的超时。消费者定期发送心跳来向经纪人表明其活跃度。如果代理在该会话超时到期之前没有收到心跳，那么代理将从该组中删除该消费者并启动重新平衡。请注意，该值必须在允许的范围内 Consumer session 过期时间。这个值必须设置在broker configuration中的group.min.session.timeout.ms 与 group.max.session.timeout.ms之间。 | int | 10000 |  | high |
| ssl.key.password | 密钥存储文件中的私钥密码。这对于客户端是可选的。 | password | null |  | high |
| ssl.keystore.location | 密钥存储文件的位置。这对客户端是可选的，可以用于客户端的双向认证。 | string | null |  | high |
| ssl.keystore.password | 密钥存储文件的商店密码。这对客户端是可选的，只有在配置了ssl.keystore.location时才需要。 | password | null |  | high |
| ssl.truststore.location | 信任存储文件的位置。 | string | null |  | high |
| ssl.truststore.password | 信任存储文件的密码。如果密码未设置，信任库的访问仍然可用，但完整性检查被禁用。 | password | null |  | high |
| auto.offset.reset | 这个配置项，是告诉Kafka Broker在发现kafka在没有初始offset，或者当前的offset是一个不存在的值（如果一个record被删除，就肯定不存在了）时，该如何处理。它有4种处理方式：  ● earliest: 自动将偏移量重置为最早的偏移量 ● latest:自动将偏移量重置为最新的偏移量 ● none: 如果边更早的offset也没有的话，就抛出异常给consumer，告诉consumer在整个consumer group中都没有发现有这样的offset。 ● anything else: 如果不是上述3种，只抛出异常给consumer。 | string | latest | [latest, earliest, none] | medium |
| connections.max.idle.ms | 连接空闲超时时间。因为consumer只与broker有连接（coordinator也是一个broker），所以这个配置的是consumer到broker之间的。 | long | 540000 |  | medium |
| enable.auto.commit | Consumer 在commit offset时有两种模式：自动提交，手动提交。手动提交在前面已经说过。自动提交：是Kafka Consumer会在后台周期性的去commit。默认值是true。 | boolean | true |  | medium |
| exclude.internal.topics | 内部主题（如偏移）的记录是否应该暴露给消费者。如果设置为true从内部主题接收记录的唯一方法是订阅它 | boolean | true |  | medium |
| fetch.max.bytes | 一次fetch请求，从一个broker中取得的records最大大小。如果在从topic中第一个非空的partition取消息时，如果取到的第一个record的大小就超过这个配置时，仍然会读取这个record，也就是说在这片情况下，只会返回这一条record。 broker、topic都会对producer发给它的message size做限制。所以在配置这值时，可以参考broker的message.max.bytes 和 topic的max.message.bytes的配置。 | int | 52428800 | [0,...] | medium |
| isolation.level | 控制如何阅读事务处理的消息。如果设置为read\_committed，consumer.poll（）将仅返回已提交的事务消息。如果设置为read\_uncommitted'（默认），consumer.poll（）将返回所有的消息，甚至是已经中止的事务消息。非交易消息将在任一模式下无条件返回。 消息将始终以偏移顺序返回。因此，在 read\_committed模式下，consumer.poll（）将只返回到最后一个稳定偏移量（LSO）的消息，这比第一个打开事务的偏移量小。特别是在属于正在进行的交易的消息之后出现的任何消息将被扣留，直到相关的交易完成。因此，read\_committed消费者在飞行交易中将无法读取高水印。 而且，当进入时 read\_committed the seekToEnd method will return the LSO | string | read\_uncommitted | [read\_committed, read\_uncommitted] | medium |
| max.poll.interval.ms | 在使用消费者组管理时，调用poll（）之间的最大延迟。这提出了消费者在获取更多记录之前可以闲置的时间量的上界。如果在此超时到期之前未调用poll（），则认为使用者失败，并且组将重新平衡以将分区重新分配给其他成员。 | int | 300000 | [1,...] | medium |
| max.poll.records | 在一次调用poll（）中返回的最大记录数。 | int | 500 | [1,...] | medium |
| partition.assignment.strategy | 当使用组管理时，客户端将用于在客户实例之间分配分区所有权的分区分配策略的类名 | list | class org.apache.kafka.clients.consumer.RangeAssignor |  | medium |
| receive.buffer.bytes | 读取数据时使用的TCP接收缓冲区（SO\_RCVBUF）的大小。如果值为-1，则将使用操作系统默认值。 | int | 65536 | [-1,...] | medium |
| request.timeout.ms | 配置控制客户端等待请求响应的最长时间。如果在超时过去之前未收到响应，则客户端将在必要时重新发送请求，或者如果重试耗尽，则请求失败。 | int | 305000 | [0,...] | medium |
| sasl.jaas.config | 用于JAAS配置文件使用的格式的SASL连接的JAAS登录上下文参数。 这里描述JAAS配置文件格式。 值的格式是：'（=）\*;' | password | null |  | medium |
| sasl.kerberos.service.name | Kafka运行的Kerberos主体名称。这可以在Kafka的JAAS配置或Kafka的配置中定义。 | string | null |  | medium |
| sasl.mechanism | 用于客户端连接的SASL机制。这可能是安全提供者可用的任何机制。GSSAPI是默认的机制。 | string | GSSAPI |  | medium |
| security.protocol | 用于与经纪人沟通的协议。有效值为：PLAINTEXT，SSL，SASL\_PLAINTEXT，SASL\_SSL。 | string | PLAINTEXT |  | medium |
| send.buffer.bytes | 发送数据时要使用的TCP发送缓冲区（SO\_SNDBUF）的大小。如果值为-1，则将使用操作系统默认值。 | int | 131072 | [-1,...] | medium |
| ssl.enabled.protocols | 启用了SSL连接的协议列表。 | list | TLSv1.2,TLSv1.1,TLSv1 |  | medium |
| ssl.keystore.type | 密钥存储文件的文件格式。这对于客户端是可选的。 | string | JKS |  | medium |
| ssl.protocol | 用于生成SSLContext的SSL协议。默认设置是TLS，在大多数情况下这是很好的。最近的JVM中允许的值是TLS，TLSv1.1和TLSv1.2。较旧的JVM中可能支持SSL，SSLv2和SSLv3，但由于已知的安全漏洞，不鼓励使用SSL。 | string | TLS |  | medium |
| ssl.provider | 用于SSL连接的安全提供程序的名称。默认值是JVM的默认安全提供程序。 | string | null |  | medium |
| ssl.truststore.type | 信任存储文件的文件格式。 | string | JKS |  | medium |
| auto.commit.interval.ms | 自动提交间隔。范围：[0,Integer.MAX]，默认值是 5000 （5 s） 消费者偏移的频率以毫秒为单位自动提交给Kafka，如果enable.auto.commit设置为true。 | int | 5000 | [0,...] | low |
| check.crcs | 自动检查消耗的记录的CRC32。这可以确保没有在线或磁盘损坏的消息发生。这个检查会增加一些开销，所以在寻求极高性能的情况下可能会被禁用。 | boolean | true |  | low |
| client.id | 发出请求时传递给服务器的id字符串。这样做的目的是通过允许在服务器端请求日志中包含一个逻辑应用程序的名字来跟踪请求的来源，而不仅仅是ip / port。 | string | "" |  | low |
| fetch.max.wait.ms | Fetch请求发给broker后，在broker中可能会被阻塞的（当topic中records的总size小于fetch.min.bytes时），此时这个fetch请求耗时就会比较长。这个配置就是来配置consumer最多等待response多久。 | int | 500 | [0,...] | low |
| interceptor.classes | 用作拦截器的类的列表。实现org.apache.kafka.clients.consumer.ConsumerInterceptor接口允许你拦截（也可能是变异）消费者收到的记录。默认情况下，没有拦截器。 | list | null |  | low |
| metadata.max.age.ms | 以毫秒为单位的时间段之后，即使我们没有看到任何分区领导变化，以主动发现任何新的代理或分区，我们强制更新元数据。 | long | 300000 | [0,...] | low |
| metric.reporters | 用作度量记录的类的列表。实现org.apache.kafka.common.metrics.MetricsReporter接口允许插入将被通知新度量创建的类。JmxReporter始终包含在注册JMX统计信息中。 | list | "" |  | low |
| metrics.num.samples | 维持用于计算度量的样本数量。 | int | 2 | [1,...] | low |
| metrics.recording.level | 指标的最高记录级别。 | string | INFO | [INFO, DEBUG] | low |
| metrics.sample.window.ms | 计算指标样本的时间窗口。 | long | 30000 | [0,...] | low |
| reconnect.backoff.max.ms | 重新连接到重复连接失败的代理程序时要等待的最长时间（以毫秒为单位）。如果提供的话，每个主机的退避将以指数方式增加，对于每个连续的连接失败，达到这个最大值。计算后退增加后，增加20％随机抖动以避免连接风暴。 | long | 1000 | [0,...] | low |
| reconnect.backoff.ms | 尝试重新连接到给定主机之前等待的基本时间。这避免了在一个紧密的循环中重复连接到主机。该退避适用于客户端向经纪人的所有连接尝试。 | long | 50 | [0,...] | low |
| retry.backoff.ms | 尝试重试对给定主题分区的失败请求之前等待的时间量。这样可以避免在某些故障情况下重复发送请求。 | long | 100 | [0,...] | low |
| sasl.kerberos.kinit.cmd | Kerberos kinit命令路径。 | string | /usr/bin/kinit |  | low |
| sasl.kerberos.min.time.before.relogin | 登录线程在刷新尝试之间的休眠时间 | long | 60000 |  | low |
| sasl.kerberos.ticket.renew.jitter | 随机抖动增加到更新时间的百分比。 | double | 0.05 |  | low |
| sasl.kerberos.ticket.renew.window.factor | 登录线程将休眠，直到已经到达从上次刷新到票证到期的指定窗口时间因子，届时它将尝试更新票证。 | double | 0.8 |  | low |
| ssl.cipher.suites | 密码套件列表。这是用于使用TLS或SSL网络协议来协商网络连接的安全设置的认证，加密，MAC和密钥交换算法的命名组合。默认情况下，所有可用的密码套件都受支持。 | list | null |  | low |
| ssl.endpoint.identification.algorithm | 使用服务器证书验证服务器主机名的端点识别算法。 | string | null |  | low |
| ssl.keymanager.algorithm | 密钥管理器工厂用于SSL连接的算法。默认值是为Java虚拟机配置的密钥管理器工厂算法。 | string | SunX509 |  | low |
| ssl.secure.random.implementation | 用于SSL加密操作的SecureRandom PRNG实现。 | string | null |  | low |
| ssl.trustmanager.algorithm | 信任管理器工厂用于SSL连接的算法。默认值是为Java虚拟机配置的信任管理器工厂算法。 | string | PKIX |  | low |

# [Kafka生产者——向 Kafka写入数据](https://www.jianshu.com/p/26532247d4cc)

# Kafka消费者

[Kafka消费者——从 Kafka读取数据](https://www.jianshu.com/p/619ceb9ae489)

http://kafka.apache.org/0101/documentation.html#newconsumerconfigs

<https://www.cnblogs.com/rilley/p/5391268.html>

KEY\_DESERIALIZER\_CLASS\_CONFIG与VALUE\_DESERIALIZER\_CLASS\_CONFIG指定key和value的编码、解码策略。kafka用key值确定value存放在哪个分区中。

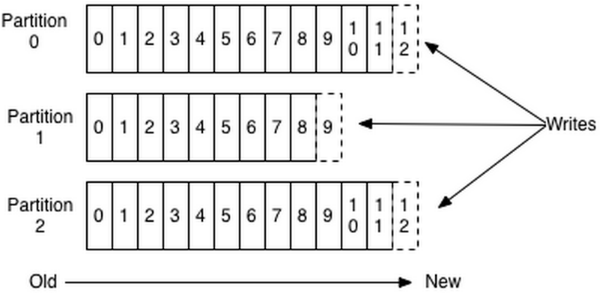
# Kafka 高吞吐量的秘诀

消息中间件从功能上看就是写入数据、读取数据两大类，优化也可以从这两方面来看。

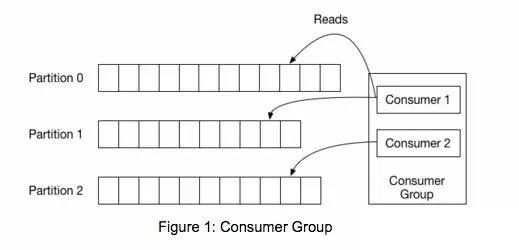
为了优化写入速度 Kafak 采用以下技术：

1. 顺序写入

磁盘大多数都还是机械结构（SSD不在讨论的范围内），如果将消息以随机写的方式存入磁盘，就需要按柱面、磁头、扇区的方式寻址，缓慢的机械运动（相对内存）会消耗大量时间，导致磁盘的写入速度与内存写入速度差好几个数量级。为了规避随机写带来的时间消耗，Kafka 采取了顺序写的方式存储数据，如下图所示：



顺序写



消费消息

2. 内存映射文件

即便是顺序写入硬盘，硬盘的访问速度还是不可能追上内存。所以 Kafka 的数据并不是实时的写入硬盘，它充分利用了现代操作系统分页存储来利用内存提高I/O效率。Memory Mapped Files （后面简称mmap）也被翻译成内存映射文件，在64位操作系统中一般可以表示 20G 的数据文件，它的工作原理是直接利用操作系统的 Page 来实现文件到物理内存的直接映射。完成映射之后对物理内存的操作会被同步到硬盘上（由操作系统在适当的时候）。通过 mmap 进程像读写硬盘一样读写内存，也不必关心内存的大小，有虚拟内存为我们兜底。使用这种方式可以获取很大的 I/O 提升，因为它省去了用户空间到内核空间复制的开销（调用文件的 read 函数会把数据先放到内核空间的内存中，然后再复制到用户空间的内存中）但这样也有一个很明显的缺陷——不可靠，写到 mmap 中的数据并没有被真正的写到硬盘，操作系统会在程序主动调用 flush 的时候才把数据真正的写到硬盘。所以 Kafka 提供了一个参数—— producer.type 来控制是不是主动 flush，如果Kafka 写入到 mmap 之后就立即 flush 然后再返回 Producer 叫同步(sync)；如果写入 mmap 之后立即返回，Producer 不调用 flush ，就叫异步(async)。

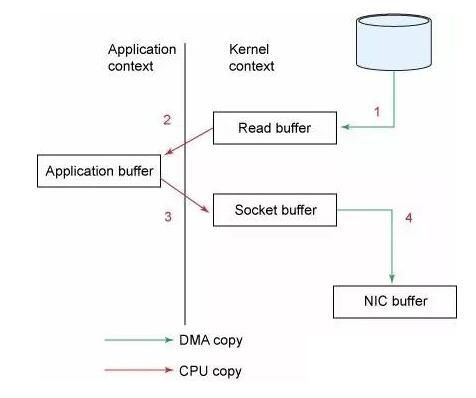
3. 标准化二进制消息格式

为了避免无效率的字节复制，尤其是在负载比较高的情况下影响是显著的。为了避免这种情况，Kafka 采用由 Producer，Broker 和 Consumer 共享的标准化二进制消息格式，这样数据块就可以在它们之间自由传输，无需转换，降低了字节复制的成本开销。

而在读取速度的优化上 Kafak 采取的主要是零拷贝

零拷贝（Zero Copy）的技术：

传统模式下我们从硬盘读取一个文件是这样的

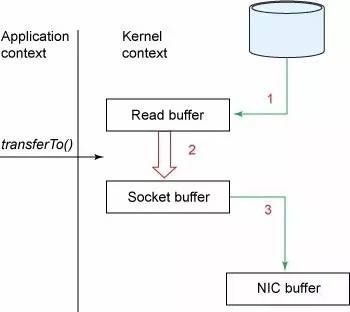


(2) 应用将数据从内核空间读到用户空间的缓存中

(3) 应用将数据写会内核空间的套接字缓存中

(4)操作系统将数据从套接字缓存写到网卡缓存中，以便将数据经网络发出

这样做明显是低效的，这里有四次拷贝，两次系统调用。针对这种情况 Unix 操作系统提供了一个优化的路径，用于将数据从页缓存区传输到 socket。在 Linux 中，是通过 sendfile 系统调用来完成的。Java提供了访问这个系统调用的方法：FileChannel.transferTo API。这种方式只需要一次拷贝：操作系统将数据直接从页缓存发送到网络上，在这个优化的路径中，只有最后一步将数据拷贝到网卡缓存中是需要的。



零拷贝方式传输到 Socket

Kafka 速度的秘诀在于它把所有的消息都变成一个的文件。通过 mmap 提高 I/O的速度，写入数据的时候是末尾添加所以速度最优；读取数据的时候配合sendfile 直接暴力输出。所以单纯的去测试 MQ 的速度没有任何意义，Kafka 的这种暴力的做法已经脱了 MQ 的底裤，更像是一个暴力的数据传送器。

# Kafka的缓冲池机制

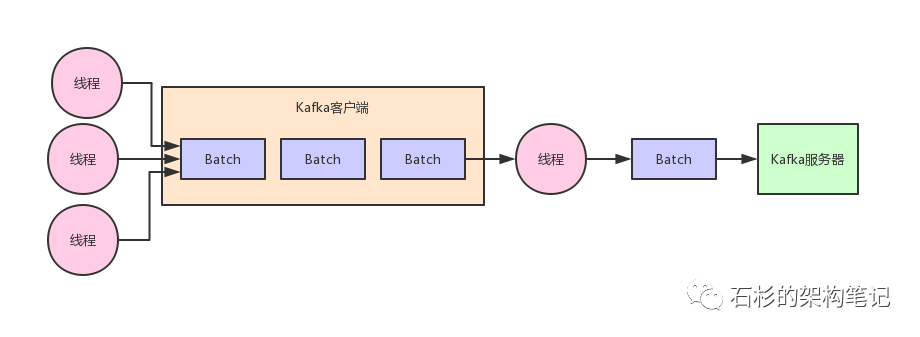
[【架构设计的艺术】Kafka如何通过精妙的架构设计优化JVM GC问题？【石杉的架构笔记】](https://mp.weixin.qq.com/s/XheJNFr5iEVCptZNw6c2oQ)

## 1、Kafka的客户端缓冲机制

首先，先得给大家明确一个事情，那就是在客户端发送消息给kafka服务器的时候，一定是有一个内存缓冲机制的。

也就是说，消息会先写入一个内存缓冲中，然后直到多条消息组成了一个Batch，才会一次网络通信把Batch发送过去。

整个过程如下图所示：



## 2、内存缓冲造成的频繁GC问题

那么这种内存缓冲机制的本意，其实就是把多条消息组成一个Batch，一次网络请求就是一个Batch或者多个Batch。

这样每次网络请求都可以发送很多数据过去，避免了一条消息一次网络请求。从而提升了吞吐量，即单位时间内发送的数据量。

但是问题来了，大家可以思考一下，一个Batch中的数据，会取出来然后封装在底层的网络包里，通过网络发送出去到达Kafka服务器。

那么然后呢？这个Batch里的数据都发送过去了，现在Batch里的数据应该怎么处理？

你要知道，这些Batch里的数据此时可还在客户端的JVM的内存里啊！那么此时从代码实现层面，一定会尝试避免任何变量去引用这些Batch对应的数据，然后尝试触发JVM自动回收掉这些内存垃圾。

这样不断的让JVM回收垃圾，就可以不断的清理掉已经发送成功的Batch了，然后就可以不断的腾出来新的内存空间让后面新的数据来使用。

这种想法很好，但是实际线上运行的时候一定会有问题，最大的问题，就是JVM GC问题。

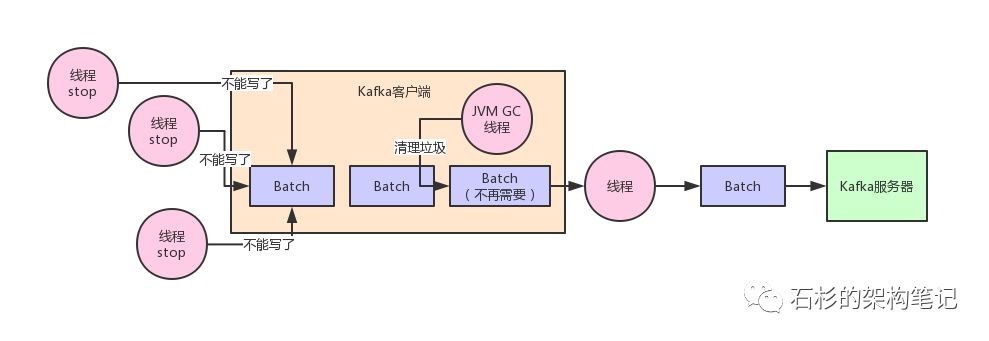
大家都知道一点，JVM GC在回收内存垃圾的时候，他会有一个“Stop the World”的过程，也就是垃圾回收线程运行的时候，会导致其他工作线程短暂的停顿，这样可以便于他自己安安静静的回收内存垃圾。

这个也很容易想明白，毕竟你要是在回收内存垃圾的时候，你的工作线程还在不断的往内存里写数据，制造更多的内存垃圾，那你让人家JVM怎么回收垃圾？

这就好比在大马路上，如果地上有很多垃圾，现在要把垃圾都扫干净，最好的办法是什么？大家都让开，把马路空出来，然后清洁工就是把垃圾清理干净。

但是如果清洁工在清扫垃圾的时候，结果一帮人在旁边不停的嗑瓜子扔瓜子壳，吃西瓜扔西瓜皮，不停的制造垃圾，你觉得清洁工内心啥感受？当然是很愤慨了，照这么搞，地上的垃圾永远的都搞不干净了！

通过了上面的语言描述，我们再来一张图，大家看看就更加清楚了



现在JVM GC是越来越先进，从CMS垃圾回收器到G1垃圾回收器，核心的目标之一就是不断的缩减垃圾回收的时候，导致其他工作线程停顿的时间。

所以现在越是新款的垃圾回收器导致工作线程停顿的时间越短，但是再怎么短，他也还是存在啊！

所以说，如何尽可能在自己的设计上避免JVM频繁的GC就是一个非常考验水平的事儿了。

## 3、Kafka设计者实现的缓冲池机制

在Kafka客户端内部，对这个问题实现了一个非常优秀的机制，就是缓冲池的机制

简单来说，就是每个Batch底层都对应一块内存空间，这个内存空间就是专门用来存放写入进去的消息的。

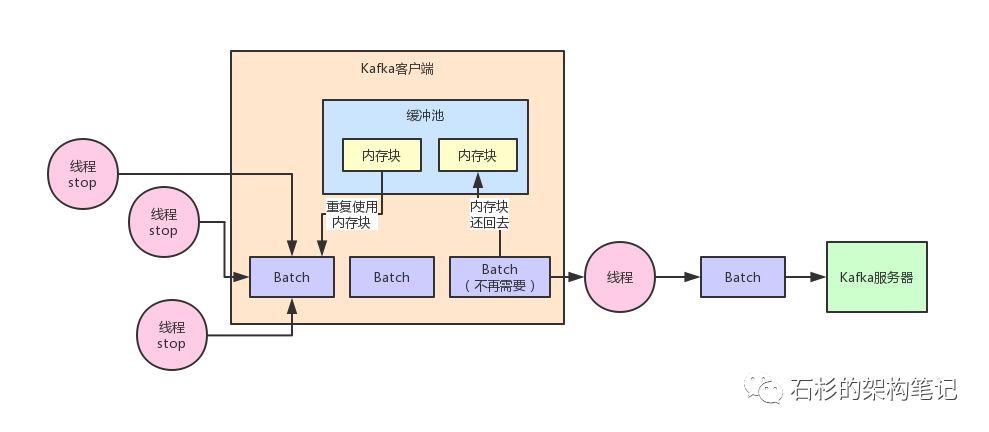
然后呢，当一个Batch被发送到了kafka服务器，这个Batch的数据不再需要了，就意味着这个Batch的内存空间不再使用了。

此时这个Batch底层的内存空间不要交给JVM去垃圾回收，而是把这块内存空间给放入一个缓冲池里。

这个缓冲池里放了很多块内存空间，下次如果你又有一个新的Batch了，那么不就可以直接从这个缓冲池里获取一块内存空间就ok了？

然后如果一个Batch发送出去了之后，再把内存空间给人家还回来不就好了？以此类推，循环往复。

同样，听完了上面的文字描述，再来一张图，看完这张图相信大伙儿就明白了：



一旦使用了这个缓冲池机制之后，就不涉及到频繁的大量内存的GC问题了。

为什么呢？因为他可以上来就占用固定的内存，比如32MB。然后把32MB划分为N多个内存块，比如说一个内存块是16KB，这样的话这个缓冲池里就会有很多的内存块。

然后你需要创建一个新的Batch，就从缓冲池里取一个16KB的内存块就可以了，然后这个Batch就不断的写入消息，但是最多就是写16KB，因为Batch底层的内存块就16KB。

接着如果Batch被发送到Kafka服务器了，此时Batch底层的内存块就直接还回缓冲池就可以了。

下次别人再要构建一个Batch的时候，再次使用缓冲池里的内存块就好了。这样就可以利用有限的内存，对他不停的反复重复的利用。因为如果你的Batch使用完了以后是把内存块还回到缓冲池中去，那么就不涉及到垃圾回收了。

如果没有频繁的垃圾回收，自然就避免了频繁导致的工作线程的停顿了，JVM GC问题是不是就得到了大幅度的优化？

没错，正是这个设计思想让Kafka客户端的性能和吞吐量都非常的高，这里蕴含了大量的优秀的机制。

那么此时有人说了，如果我现在把一个缓冲池里的内存资源都占满了，现在缓冲池里暂时没有内存块了，怎么办呢？

很简单，阻塞你的写入操作，不让你继续写入消息了。把你给阻塞住，不停的等待，直到有内存块释放出来，然后再继续让你写入消息。

# JDQ

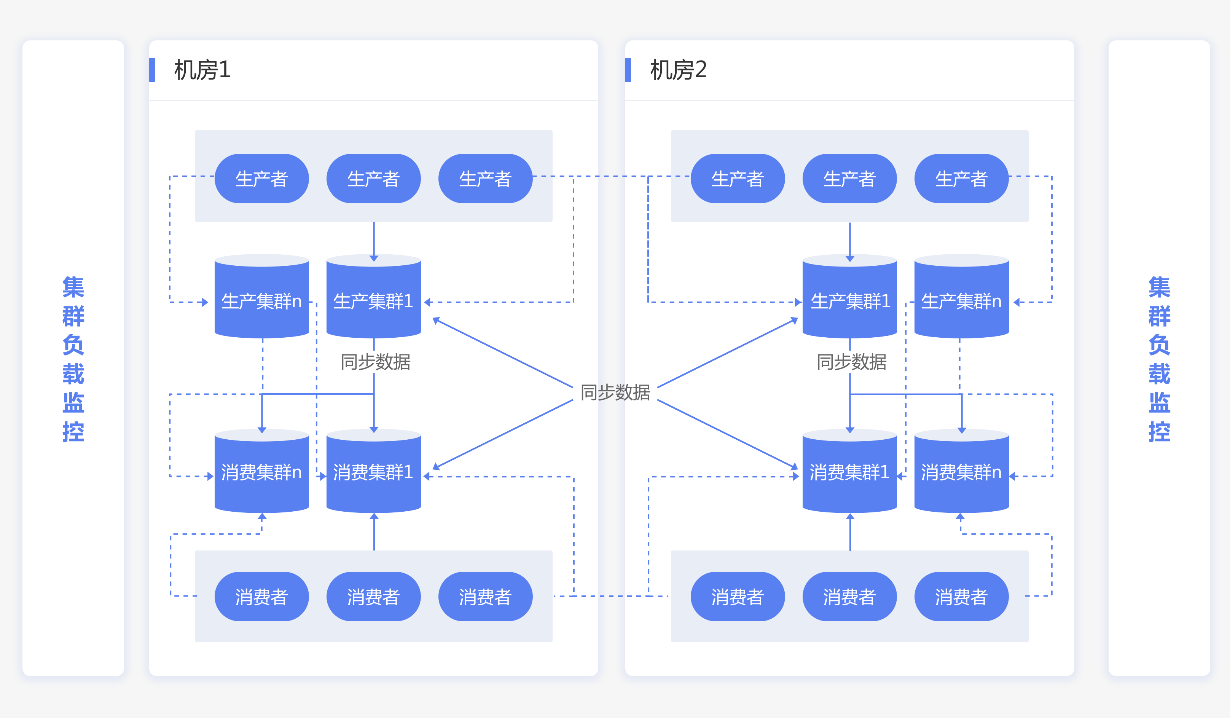
## 介绍：

分布式消息队列JDQ是京东数据中台基于Kafka实现的消息管理平台，是结合京东对线上业务数据及系统日志的管理需要，在对主流消息管理系统进行选型对比后，在商城大促中经过实践验证的平台产品。

Kafka是由LinkedIn开发的一个分布式的消息系统，它以强大的水平扩展和高吞吐能力而被广泛使用，很多著名的互联网企业将其做为数据管道和消息系统。

在业务极速发展的京东，增长的数据量已经超过想象，为了满足业务方对数据获取时效及质量的需求，实时数据平台也以超过业界水平的速度发展。以2018年618大促为例，实时数据平台生产消息13882.86亿条/天，大小约为 545.63TB，而业务方的消费量更是达到2000TB，我们的分布式消息队列（JDQ）产品在其中扮演了不可或缺的角色。

JDQ的演进稳定之路，也就是我们在kafka上的实践和踩坑的过程。经过了数年的积累和沉淀，kafka日趋稳定，支撑了整个实时计算平台，下面介绍下JDQ的在平台中应用：



**JDQ—读写分离**

在线上生产环境中，同时读写一个kafka集群的时候会相互影响，大部分情况下由于消费客户端的数量比较多，导致流出的数据量过大，可能会到达机器的硬件瓶颈，这个时候kafka的集群稳定性会受到一定的影响，可能会影响到生产，导致生产数据不能实时写入，造成生产的延迟。这种状况下，我们采用读写分离，生产集群中只会接收生产数据，而消费端只会有同步程序来进行消费，然后数据同步到消费集群的时候，大家再从消费集群消费数据，同时消费集群也可以为多个，消费者数据很多的时候可以消费多个集群的数据。

**JDQ—跨机房灾备**

减少跨机房传输：每个机房都会部署1个或者多个生产集群和消费集群，在生产的时候客户端会选择本机房的kafka集群进行生产以减少跨机房的带宽。

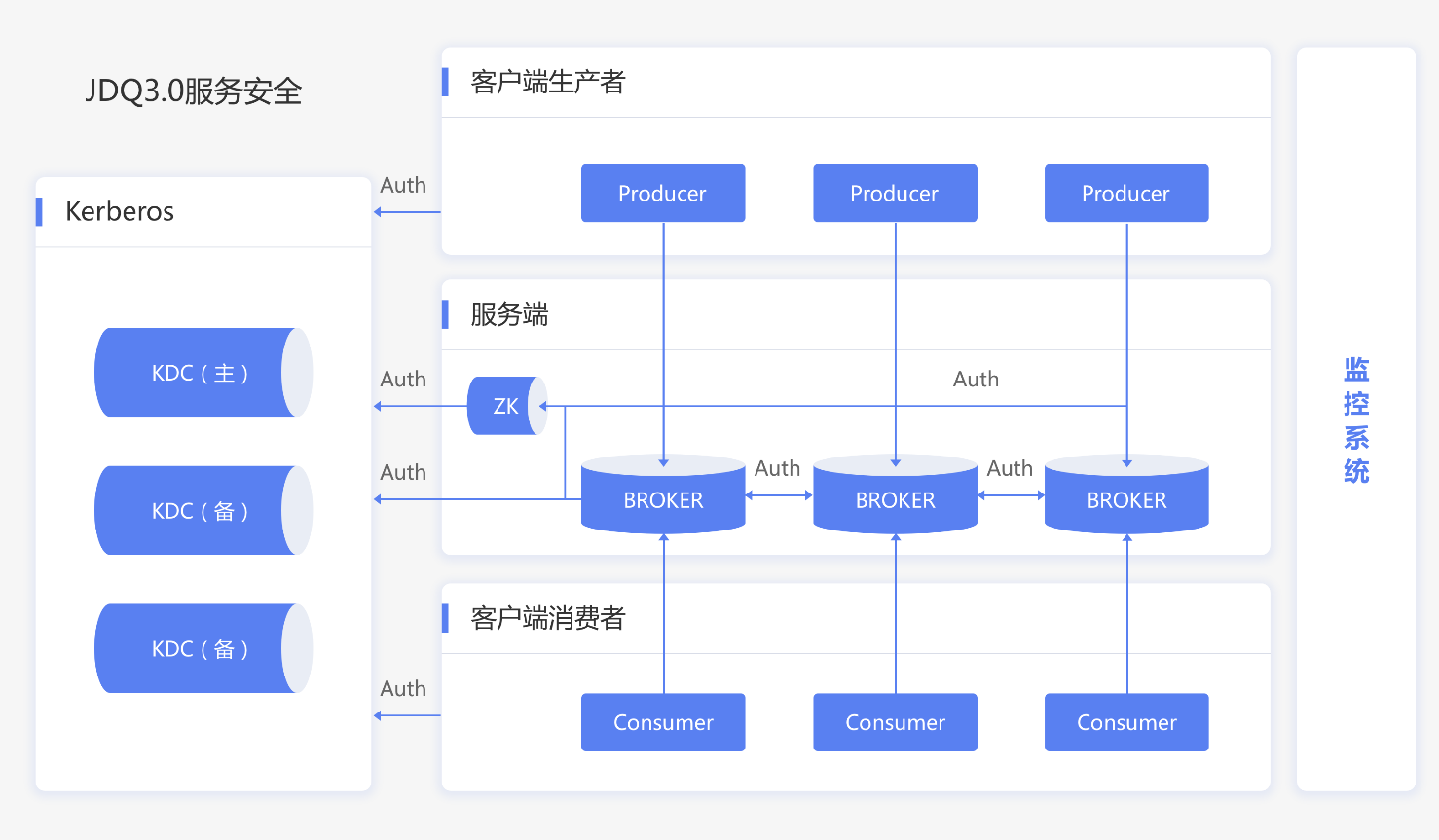
生产灾备：当生产集群出现故障的时候，生产者可切换本机房的生产集群，如果本机房的集群没有可用的情况下可转发到其他机房的生产集群。

消费灾备：生产集群的是数据会被同步到各个机房的消费集群，如果某个消费客户端出现故障无法消费，则可切换到其他可用的消费集群，优先选择本机房的集群，如果本机房无可用集群切换至其他机房的消费集群。

**JDQ—消费切换衔接**

集群数据同步是指将生产集群的数据同步到消费集群，而且对于单个topic来讲，所有消费集群的数据都是一样的。当消费中断需要切换消费的时候则需要将切换后的消费位点进行衔接，这里我们会在数据同步的时候将读写集群的位点关系维护起来，这样每个生产集群的位点都会对应到消费集群位点，而且消费位点对应生产位点的关系我们也会维护起来，这样每次再需要切换衔接位点的时候，我们就会从消费位点找到生产位点，然后再由生产位点找到其他的消费位点，这样保证消费衔接的问题。

**JDQ—安全认证**

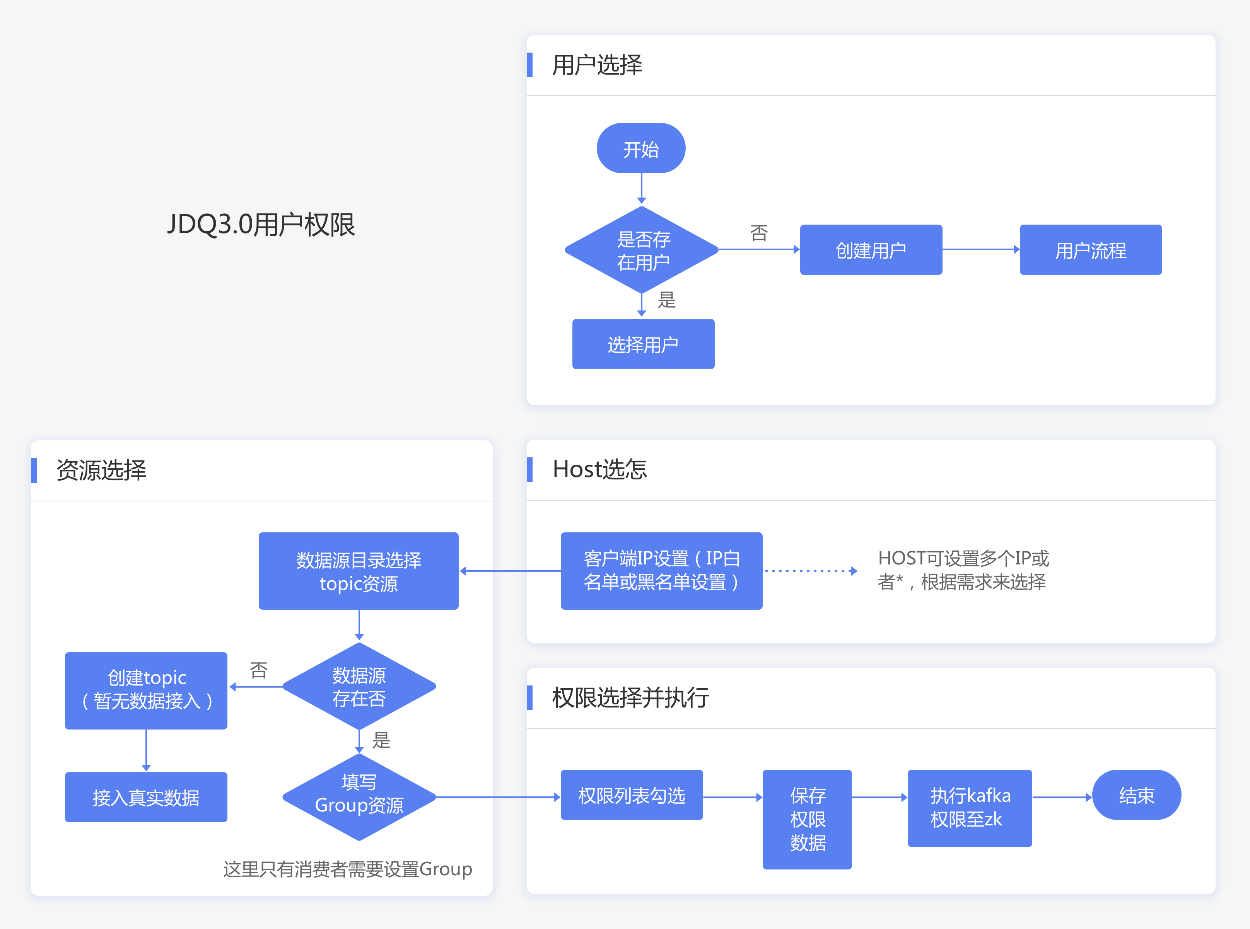


kafka在0.9之后才加入安全认证，在此之前集群的安全问题还是很严重，JDQ需要保证平台的安全稳定运行，需要防止不明身份的broker对集群的访问，不明身份的broker不应该成为集群的一员，不明身份的broker不能从集群中的broker拉取数据，防止集群在ZK上的metadata被篡改，保证数据存储的安全，保证集群与外部客户端的安全通讯，执行客户端的用户身份认证。

我们引入了kerberos帮助kafka做安全认证，我们要对客户端和各个组件都进行认证，保证每个环境的安全。

**JDQ—权限管理**

在安全认证之后，权限管理就是下一个需要注重的环境，我们需要控制客户端对资源的访问，只能对授权的资源进行访问，这样我们的安全程度就会更加提高一个层次。如图，我们的权限可以根据各自的需要进行设置。



**JDQ—监控**

平台能够健康稳定的一个重要因素就是一个精准的监控系统，各个环节以及硬件等各种指标的采集分析都能够帮助平台进行自我检查和恢复。JDQ监控系统会采集服务端，客户端等各项指标进行梳理展示，帮助大家分析问题，也会根据这些指标进行一定的分析，得到现有系统的一个负载状况，当出现一些问题的时候可以进行自我修复。我们利用kafka的生态，用kafka connect进行数据采集，将服务端的metrics采集，然后利用kakfa steam进行一次数据的逻辑处理，再用kafka connect转存数据入ES，最终在grafana进行展示。



**JDQ—产品化**

目前平台通过简单的web页面，支持数据权限、进程、速度、监控的管理及查看。



**总结**

JDQ已经在实时数据平台内部进行了2个版本的更新，包括集群负载，灾备，读写分离，安全和监控等方面都做了一定的改造，同时经历了多次618和双十一的洗礼之后已经越来越稳定，面对实时数据平台中不断暴涨的业务和数据还是有很大的挑战，我们依然会对现有的框架进行不断的升级，更好的服务各种类型的用户。

## 生产者代码

public class JDQProducer {

    public static void main(String[] args) throws Exception {

        String username = args[0];// username 对应 应用域名

        String password = args[1];// password 对应 accesskey

        String clientId = args[2];// 线上环境会验证clientId是否属于当前用户,所有这里的clientId一定要用用户申请下来的clientId,另外这里的clientId也会授权提速(默认5MB/s),非授权的clientId速度为1KB/s

        String topic = args[3];//topic

        Properties authProps = JDQClientUtils.JDQClientProperties(clientId, username, password);//获取关于认证的配置以及brokerlist

        KafkaProducer<String, String> producer = new KafkaProducer<>(getProperties(authProps));

        try {

            while (!Thread.currentThread().isInterrupted()) {

                while (true) {

                    ProducerRecord<String, String> record = new ProducerRecord<>(topic, "key" + System.currentTimeMillis(), "value");//1504195200000

                    /\*\*

                     \* 默认都是异步发送，如果想同步发送producer.send(xxx).get()单条发送，效率比较低

                     \* 每条消息都会有callback，可以根据需要来执行发送成功或者失败后的操作

                     \*/

                    producer.send(record, new Callback() {

                        @Override

                        public void onCompletion(RecordMetadata metadata, Exception exception) {

                            if (exception != null) {

                                /\*\*

                                 \* 当存在异常的时候,发送失败,大家可以根据自己需要设置失败逻辑

                                 \*/

                                System.out.println("hhhh - " + exception.getMessage());

                            } else {

                                /\*\*

                                 \* 没有异常,认为发送成功,也可设置成功之后的逻辑设置

                                 \* 注意:发送成功之后这里的RecordMetadata内包含此条消息写入到JDQ集群的分区和对应的位点信息等

                                 \*/

                                System.out.println("OFFSET[" + metadata.offset() + "] - PARTITION[" + metadata.partition() + "] - TIME[" + "] is aready send ok");

                            }

                        }

                    });

                    /\*\*

                     \*  flush :  执行flush代表内存中的消息数据都会send到服务端，可根据callback来判断成功与否

                     \*  自己控制flush

                     \*  注意：这里可发送一批数据之后再调flush，达到批量的效果

                     \*/

                    producer.flush();

                }

            }

        } finally {

            /\*\*

             \* 注意在不用的时候释放资源

             \*/

            producer.close();

        }

    }

    /\*\*

     \* 添加kafka客户端的源生的一些配置

     \* @param authProps

     \* @return

     \*/

    private static Properties getProperties(Properties authProps) {

        /\*\*

         \* 需要查看DEBUG日志的可以设置为true，默认为false

         \*/

        //authProps.setProperty(ProducerConfig.JDQ\_ENABLE\_DEBUG\_CONFIG, "true");

        /\*\*

         \* kafka配置列表，可参考（版本0.10.1.0）http://kafka.apache.org/0101/documentation.html#producerconfigs

         \*/

        authProps.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.StringSerializer");//序列化方式是string

        authProps.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.StringSerializer");//序列化方式是string

        authProps.put(ProducerConfig.COMPRESSION\_TYPE\_CONFIG, "snappy");

        return authProps;

    }

}

## 消费者代码

/\*\*

 \*

 \* 测试环境使用:

 \*

 \* 1. 需要host配置/etc/hosts

 \*      192.168.144.118 TEST-JDQ-144118.bdp.jd.local

 \*      192.168.144.109 TEST-JDQ-144109.bdp.jd.local

 \*

 \* 2. 另外需要设置测试环境的KDC地址,见getProperties内部设置

 \*

 \*/

public static void main(String[] args) {

    JDQ\_ENV.authClinetNV("NULL", 80);//这句话只有在线下测试环境认证鉴权不通过的时候才需要．线上正式环境不需要

    String username = args[0];//即申请的应用域名

    String password = args[1];//即accesskey

    String brokerlist = args[2];

    String clientId = args[3];

    String group = args[4];

    String topic = args[5];

    /\*\*

     \* 注意:

     \* 1. clientId

     \*      线上环境会验证clientId是否属于当前用户,所有这里的clientId一定要用用户申请下来的clientId

     \*      另外这里的clientId也会授权提速(默认5MB/s),非授权的clientId速度为1KB/s

     \* 2. username,password,brokerlist,group

     \*      数据来源为JDQ3.0申请审批后的客户端信息

     \*      brokerlist 对应页面中TOPIC信息标签下的 Brokerlist 数据

     \*      group 对应页面中TOPIC信息标签下的 GROUPID 数据

     \*      username和password 需要从客户端详情页面中的所属应用下查看(点击页面中对应的所属应用连接)

     \*          username 对应 应用域名

     \*          password 对应 accesskey

     \*

     \*/

    Properties props = getProperties(username, password, brokerlist, clientId, group);

    KafkaConsumer<byte[], byte[]> consumer = new KafkaConsumer<>(props);

    Collection<String> topics = Arrays.asList(topic);

    /\*\*

     \* 参考方法:

     \*

     \* consumer.partitionsFor(topic)

     \*      查询topic的分区信息,当本地没有这个topic的元数据信息的时候会往服务端发送的远程请求

     \*      注意: 没有权限的topic的时候会抛出异常(org.apache.kafka.common.errors.TopicAuthorizationException)

     \*

     \* consumer.position(new TopicPartition(topic, 0))

     \*      获取下次拉取的数据的offset, 如果没有offset记录则会抛出异常

     \*

     \* consumer.committed(new TopicPartition(topic, 0))

     \*      获取已提交的位点信息，如果没有查询到则返回null

     \*

     \* consumer.beginningOffsets(Arrays.asList(new TopicPartition(topic, 0)));

     \* consumer.endOffsets(Arrays.asList(new TopicPartition(topic, 0)));

     \* consumer.offsetsForTimes(timestampsToSearch);

     \*      查询最小,最大,或者任意时间的位点信息

     \*

     \* consumer.seek(new TopicPartition(topic, 0), 10972);

     \* consumer.seekToBeginning(Arrays.asList(new TopicPartition(topic, 0)));

     \* consumer.seekToEnd(Arrays.asList(new TopicPartition(topic, 0)));

     \*      设置offset给消费者拉取消息

     \*

     \* consumer.assign(Arrays.asList(new TopicPartition(topic, 0)));

     \*      手动分配消费的topic和分区进行消费，这里不会出发group management操作,指定分区消费数据

     \*      和subscribe方法互斥,如果assign 之后或者之后调用subscribe 则会报错，不允许再进行分配，2方法不能一起使用

     \*

     \* consumer.subscribe(topics);

     \*      自动发布消费的topic进行消费,这里触发group management操作

     \*      和assign方法互斥,如果subscribe 之后或者之后调用assign 则会报错，不允许再进行分配，2方法不能一起使用

     \*

     \* 注: group management

     \*      根据group 进行topic分区内部的消费rebanlance

     \*      例如消费的topic包含3个分区，启动了4个相同鉴权的客户端消费

     \*              分区0 -- consumer1   分区1 -- consumer2   分区2 --- consumer3    consumer4则会空跑不消费数据

     \*         当分区consumer1挂掉的时候则会出现rebalance，之后变为

     \*              分区0 -- consumer2   分区1 -- consumer3   分区2 --- consumer4

     \*

     \*/

    consumer.subscribe(topics);

    try{

        while(!Thread.currentThread().isInterrupted()){

            ConsumerRecords<byte[], byte[]> records = consumer.poll(10000);

            for(ConsumerRecord<byte[], byte[]> record : records){

                System.out.println("[offset]:" + record.offset() +", [key]:" + new String(record.key()) +", [value]:" + new String(record.value()));

            }

            /\*\*

             \* 提交位点:支持自动提交和手动提交

             \*

             \* 自动提交:

             \*      需要设置ConsumerConfig中  enable.auto.commit 为true(默认值就是true)

             \*      同时可设置提交时间间隔   auto.commit.interval.ms 默认为5000

             \*

             \* 手动提交(当采用手动提交位点的时候需要设置ConsumerConfig中  enable.auto.commit 为false):

             \* 1. 同步提交位点:

             \*      consumer.commitSync();

             \*      consumer.commitSync(offsets); //如果使用的是subcribe()方式订阅的topic，那么在使用传参方式提交位点的时候注意先获取一下该consumer的assignment().

             \*      Set<TopicPartition> assignment = consumer.assignment();//这样是为了防止由于reblance或者其他原因导致该consumer提交不属于自己分配的topic分区的位点的时候报错。

             \* 2. 异步提交位点:

             \*      consumer.commitAsync();

             \*      consumer.commitAsync(offsetCommitCallback);//如果使用的是subcribe()方式订阅的topic，那么在使用传参方式提交位点的时候注意先获取一下该consumer的assignment().

             \*      consumer.commitAsync(offsets,offsetCommitCallback);//如果使用的是subcribe()方式订阅的topic，那么在使用传参方式提交位点的时候注意先获取一下该consumer的assignment().

             \* Set<TopicPartition> assignment = consumer.assignment();//这样是为了防止由于reblance或者其他原因导致该consumer提交不属于自己分配的topic分区的位点的时候报错。

            \*/

        }

    }finally {

        /\*\*

         \* 注意在不用的时候释放资源

         \*/

        consumer.close();

    }

}

private static Properties getProperties(String username, String password, String brokerlist, String clientId, String group) {

    Properties props = new Properties();

    /\*\*

     \* 注意:

     \* 测试环境需要指定测试环境的KDC地址,线上环境不需要指定,默认会找到线上KDC

     \* 另外需要按照使用手册配置host

     \*/

    System.setProperty("java.security.krb5.kdc", "TEST-JDQ-144109.bdp.jd.local");

    /\*\*SASL\_PLAINTEXT

     \* JDQ 安全相关配置：1.JDQ服务端认证方式为SASL\_PLAINTEXT，需要在此设置,默认SASL\_PLAINTEXT

     \*/

    props.setProperty(CommonClientConfigs.SECURITY\_PROTOCOL\_CONFIG, "SASL\_PLAINTEXT");

    /\*\*

     \* JDQ 安全相关配置：

     \*

     \* 2.需要走其他方式如keytab访问服务的可设置为true，需要在系统参数中设置krb5和jaas

     \* client配置，服务采用的kerberos认证 如果使用SDK，可以设置用户名和密码进行访问，设置方式如下

     \*/

    props.setProperty(ConsumerConfig.JDQ\_USE\_KEYTAB\_CONFIG, "false");

    props.setProperty(ConsumerConfig.JDQ\_PRINCIPAL\_NAME\_CONFIG, username);

    props.setProperty(ConsumerConfig.JDQ\_PASSWORD\_CONFIG, password);

    /\*\*

     \* JDQ 安全相关配置：3.需要查看DEBUG日志的可以设置为true，默认为false

     \*/

    props.setProperty(ConsumerConfig.JDQ\_ENABLE\_DEBUG\_CONFIG, "false");

    /\*\*

     \* kafka配置列表，可参考（版本0.10.0.0）http://kafka.apache.org/0101/documentation.html#newconsumerconfigs

     \*/

    props.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, brokerlist);

    props.setProperty(ConsumerConfig.GROUP\_ID\_CONFIG, group);

    props.setProperty(ConsumerConfig.CLIENT\_ID\_CONFIG, clientId);

    props.setProperty(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.ByteArrayDeserializer");

    props.setProperty(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.ByteArrayDeserializer");

    props.setProperty(ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG, "earliest");

    props.setProperty(ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG, "false");//是否自动提交位点,默认是true.默认的自动提交位点的时间间隔是5000ms，false的情况下是需要用户自己调用commit方法自己手动提交位点信息的

 return props;

}

## 点位信息查询重置：

public class ConsumerSiteSeekBytimeAuth {

     private static final Logger logger = LoggerFactory.getLogger(Logger.class);

     private final static SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm");

     private static String NOW = "now";

     private static String BEFORE = "before";

     public static void main(String args[]) {

    // JDQ\_ENV.authClinetNV("null", 80);//线下环境测试的时候如果鉴权的时候爆出：找不到CLIRNT\_ID的时候加上这个试试

     String topic = args[0];

     String clientId = args[1];//clientId

     String username = args[2];//用户名

     String password = args[3];//密码

     String brokerlist = args[4];//集群ip列表

     String group = args[5];//group

     String timeStamp = args[6];//查询offset的时间(timeStamp取值类型为：{now,before,"yyyy-MM-dd HH:mm"格式的时间})

     System.out.println("Reset Offsets timeStamp: " + timeStamp);

     Properties props = getProperties(username, password, brokerlist, clientId, group);

     KafkaConsumer<byte[], byte[]> consumer = new KafkaConsumer<>(props);

     List<PartitionInfo> partitionInfos = consumer.partitionsFor(topic);

     List<TopicPartition> topicPartitions = new ArrayList<>();

     //打印出来该topic的分区列表

     System.out.println("topic: " + topic + " 分区信息:");

     for (PartitionInfo part : partitionInfos) {

     topicPartitions.add(new TopicPartition(part.topic(), part.partition()));

     System.out.println(part.topic() + " :　" + part.partition());

     }

     Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();

     //查询最新位点信息

     if (timeStamp.equals(NOW)) {

     Map<TopicPartition, Long> endOffsets = consumer.endOffsets(topicPartitions);

     for (Map.Entry<TopicPartition, Long> entry : endOffsets.entrySet()) {

     offsets.put(entry.getKey(), new OffsetAndMetadata(entry.getValue()));

     System.out.println("EndOffsets offset: " + entry.getKey() + ":" + entry.getValue());

     }

     //查询最小位点信息

     } else if (timeStamp.equals(BEFORE)) {

     Map<TopicPartition, Long> beginningOffsets = consumer.beginningOffsets(topicPartitions);

     for (Map.Entry<TopicPartition, Long> entry : beginningOffsets.entrySet()) {

     offsets.put(entry.getKey(), new OffsetAndMetadata(entry.getValue()));

     System.out.println("Beginning offset: " + entry.getKey() + ":" + entry.getValue());

     }

     } else {

     long resetOffsetTime = 0;

     try {

     resetOffsetTime = sdf.parse(timeStamp).getTime();

     } catch (ParseException e) {

     e.printStackTrace();

     }

     System.out.println("resetOffsetTime: " + resetOffsetTime);

     Map<TopicPartition, Long> timestampsToSearch = new HashMap<>();

     for (PartitionInfo part : partitionInfos) {

     timestampsToSearch.put(new TopicPartition(topic, part.partition()), resetOffsetTime);

     }

     //根据时间查询位点信息

     Map<TopicPartition, OffsetAndTimestamp> offsetsForTimes = consumer.offsetsForTimes(timestampsToSearch);

     if (offsetsForTimes != null && offsetsForTimes.size() > 0) {

     System.out.println("offsetsForTimes: " + offsetsForTimes);

     for (Map.Entry<TopicPartition, OffsetAndTimestamp> entry : offsetsForTimes.entrySet()) {

     if (entry.getValue() != null) {

     offsets.put(entry.getKey(), new OffsetAndMetadata(entry.getValue().offset()));

     System.out.println("OffsetAndTimestamp: " + entry.getKey() + "find offset: " + entry.getValue().offset());

     } else {

     System.out.println("Find Offsets which time is " + timeStamp + " and partition is " + entry.getKey() + "from kafka is null.");

     }

     }

     } else {

     System.out.println("Find Offsets which time is " + timeStamp + "from kafka is null.");

     }

     }

     /\*\*

     \* 提交位点注意：此时提交的位点

     \* 分为同步提交位点和异步提交位点

     \* commitAsync(Map<TopicPartition, OffsetAndMetadata> offset,OffsetCommitCallback callback)

     \*commitSync(Map<TopicPartition, OffsetAndMetadata> offset)

     \*

     \*/

     consumer.commitSync(offsets);

     logger.info("commit offset is ok ~");

     System.out.println("After commited the offsets check :");

     for (PartitionInfo part : partitionInfos) {

     System.out.println(topic + "-" + part.partition() + ":" + consumer.committed(new TopicPartition(part.topic(), part.partition())).offset());

     }

     }

     private static Properties getProperties(String username, String password, String brokerlist, String clientId, String group) {

     Properties props = new Properties();

     /\*\*

     \* 测试环境需要指定测试环境的KDC地址,线上环境不需要指定,默认会找到线上KDC,注意这里

     \*/

    // System.setProperty("java.security.krb5.kdc", "TEST-JDQ-144109.bdp.jd.local");

     /\*\*SASL\_PLAINTEXT

     \* JDQ 安全相关配置：1.JDQ服务端认证方式为SASL\_PLAINTEXT，需要在此设置,默认PLAINTEXT

     \*/

     props.setProperty(CommonClientConfigs.SECURITY\_PROTOCOL\_CONFIG, "SASL\_PLAINTEXT");

     /\*\*

     \* JDQ 安全相关配置：

     \*

     \* 2.需要走其他方式如keytab访问服务的可设置为true，需要在系统参数中设置krb5和jaas

     \* client配置，服务采用的kerberos认证 如果使用SDK，可以设置用户名和密码进行访问，设置方式如下

     \*/

     props.setProperty(ConsumerConfig.JDQ\_USE\_KEYTAB\_CONFIG, "false");

     props.setProperty(ConsumerConfig.JDQ\_PRINCIPAL\_NAME\_CONFIG, username);

     props.setProperty(ConsumerConfig.JDQ\_PASSWORD\_CONFIG, password);

     props.setProperty(CommonClientConfigs.ENABLE\_FORCE\_REFRESH\_CONFIG, "false");//是否强制刷新．当在线下测试的时候就设置为false

     /\*\*

     \* JDQ 安全相关配置：3.需要查看DEBUG日志的可以设置为true，默认为false

     \*/

     props.setProperty(ConsumerConfig.JDQ\_ENABLE\_DEBUG\_CONFIG, "false");

     /\*\*

     \* kafka配置列表，可参考（版本0.10.0.0）http://kafka.apache.org/documentation.html#newconsumerconfigs

     \*/

     props.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, brokerlist);

     props.setProperty(ConsumerConfig.GROUP\_ID\_CONFIG, group);

     props.setProperty(ConsumerConfig.CLIENT\_ID\_CONFIG, clientId);

     props.setProperty(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.ByteArrayDeserializer");

     props.setProperty(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.ByteArrayDeserializer");

     props.setProperty(ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG, "false");//不自动提交位点信息

     props.setProperty(ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG, "earliest");

     return props;

     }

    }

## 天眼中使用

配置：

auto.commit.interval.ms = 5000

auto.offset.reset = earliest

bootstrap.servers = [11.7.64.3:9092, 11.7.64.9:9092, 11.7.65.5:9092, 11.7.65.6:9092, 11.7.65.7:9092, 11.7.65.44:9092, 11.7.65.66:9092, 11.7.65.69:9092, 11.7.65.70:9092, 11.7.65.72:9092, 11.7.65.74:9092, 11.7.65.76:9092, 11.7.65.99:9092, 11.7.64.168:9092, 11.7.64.169:9092, 11.7.64.170:9092, 11.7.64.172:9092, 11.7.64.194:9092, 11.7.64.195:9092, 11.7.64.196:9092, 11.7.64.197:9092, 11.7.64.198:9092, 11.7.64.199:9092, 11.7.64.203:9092, 11.7.64.204:9092, 11.7.64.226:9092, 11.7.64.229:9092, 11.7.64.230:9092, 11.7.64.231:9092, 11.7.64.232:9092, 11.7.64.233:9092, 11.7.64.234:9092, 11.7.64.235:9092, 11.7.64.236:9092, 11.7.64.36:9092, 11.7.64.37:9092, 11.7.64.38:9092, 11.7.64.40:9092, 11.7.64.41:9092, 11.7.64.5:9092, 11.7.64.66:9092, 11.7.64.67:9092, 11.7.64.68:9092, 11.7.64.70:9092, 11.7.64.72:9092, 11.7.64.73:9092, 11.7.64.8:9092, 11.7.66.226:9092, 11.7.67.2:9092, 11.7.68.102:9092]

check.crcs = true

client.id = 2426f5e2

connections.max.idle.ms = 540000

enable.auto.commit = true

exclude.internal.topics = true

fetch.max.bytes = 52428800

fetch.max.wait.ms = 500

fetch.min.bytes = 1

group.id = flight\_user\_behavior\_consumer1533887371383

heartbeat.interval.ms = 3000

interceptor.classes = null

kafka.client.kerberos.debug = false

kafka.client.kerberos.force.refresh.config = true

kafka.client.kerberos.password = null

kafka.client.kerberos.principal = null

kafka.client.kerberos.useKeyTab = true

key.deserializer = class org.apache.kafka.common.serialization.ByteArrayDeserializer

krb5.kdc.bad.policy = tryLast

max.partition.fetch.bytes = 1048576

max.poll.interval.ms = 300000

max.poll.records = 500

metadata.max.age.ms = 300000

metric.reporters = []

metrics.num.samples = 2

metrics.sample.window.ms = 30000

partition.assignment.strategy = [class org.apache.kafka.clients.consumer.RangeAssignor]

receive.buffer.bytes = 65536

reconnect.backoff.ms = 50

request.timeout.ms = 305000

retry.backoff.ms = 100

sasl.kerberos.kinit.cmd = /usr/bin/kinit

sasl.kerberos.min.time.before.relogin = 600

ssl.secure.random.implementation = null

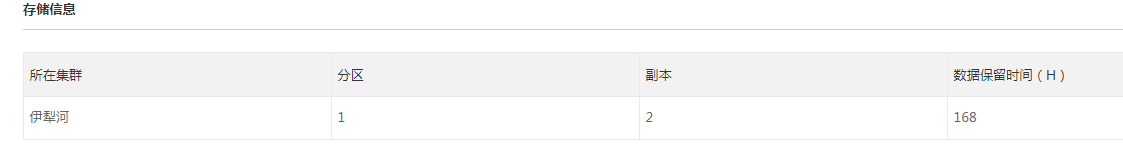
ssl.trustmanager.algorithm = PKIX

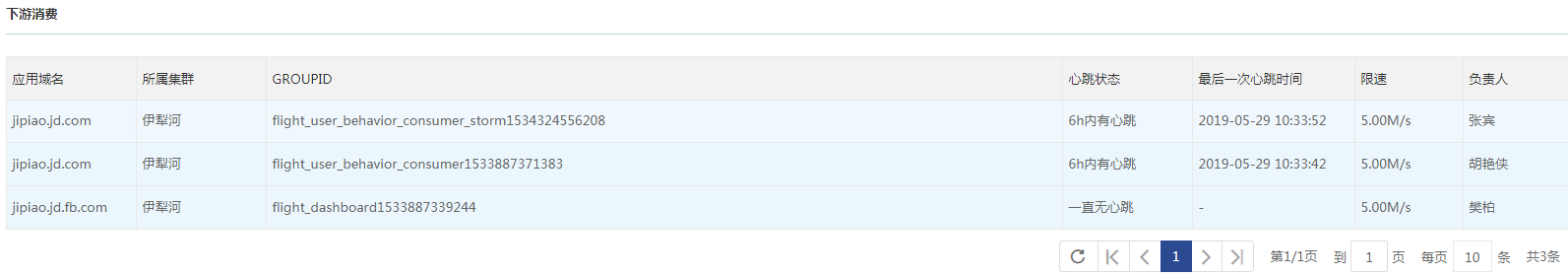
ssl.truststore.location = null

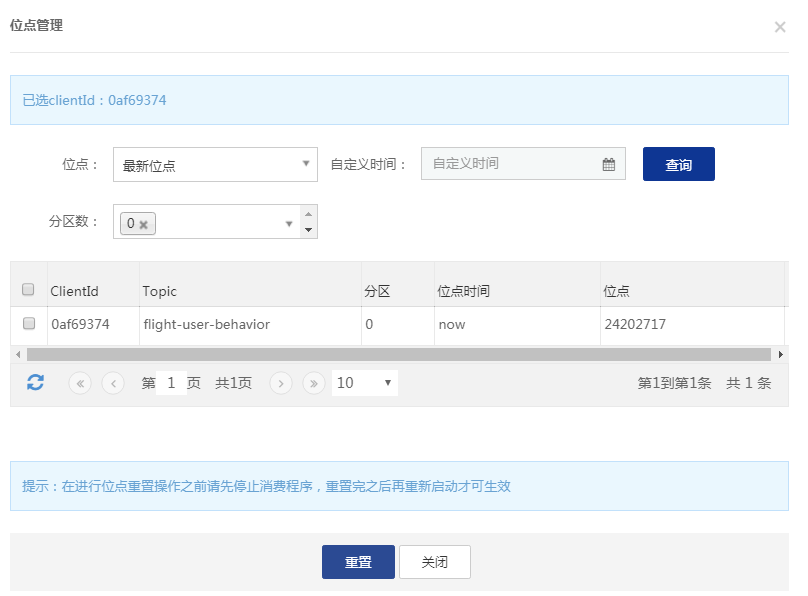
ssl.truststore.password = null

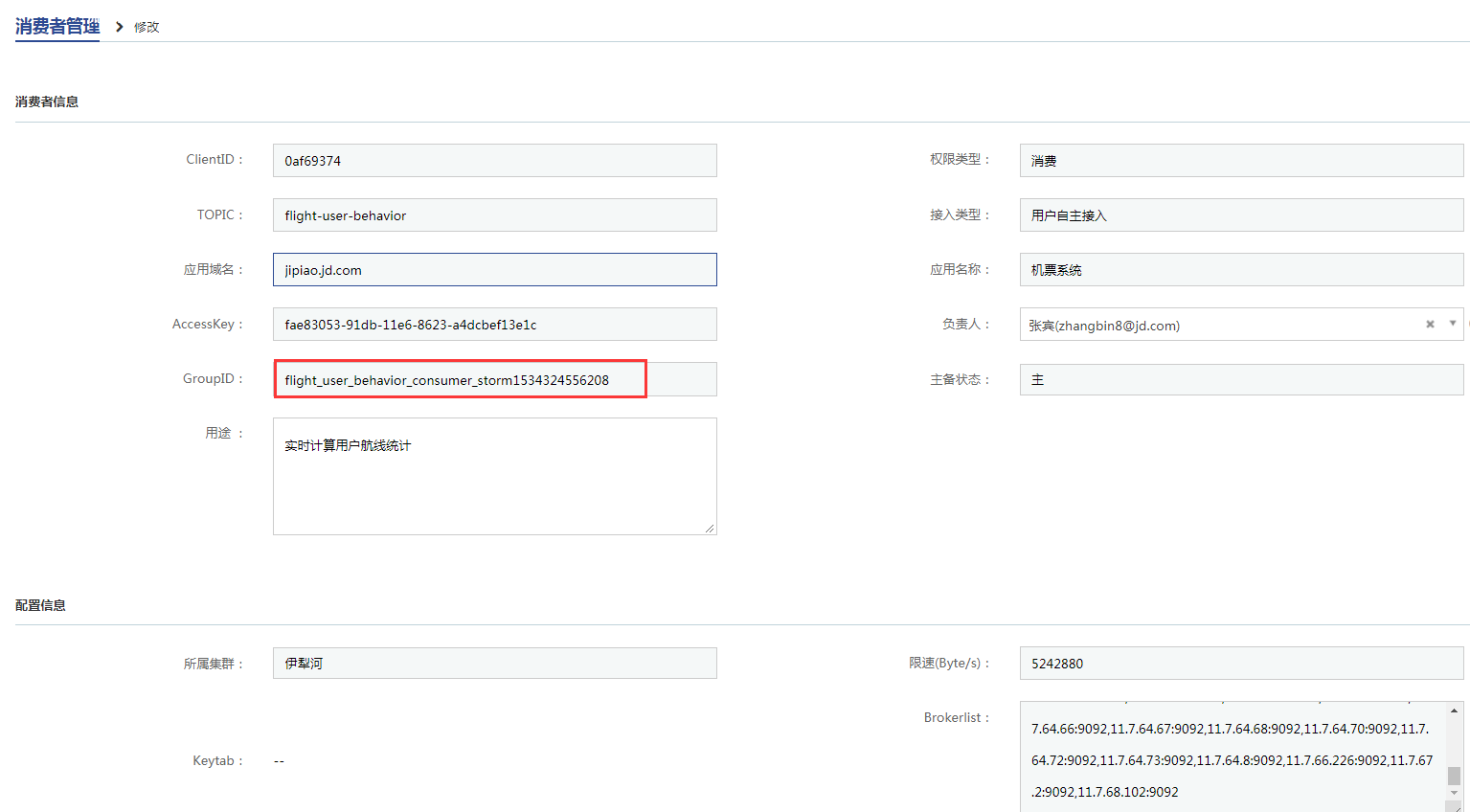
ssl.truststore.type = JKS

value.deserializer = class org.apache.kafka.common.serialization.ByteArrayDeserializer









只有两台机器的信息：



53机器的指标（预发）





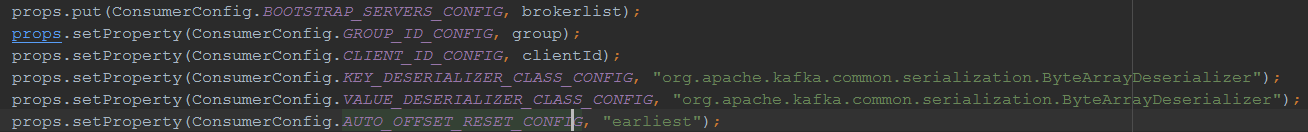
82机器的指标（线上）





### 多线程遇到问题分析：

消费者配置



问题描述

情况1：在main线程之外的另一个异步线程newSingleThreadExecutor中poll并进行所有的消费，若消费的数据为null则return，那么不会再poll消息。

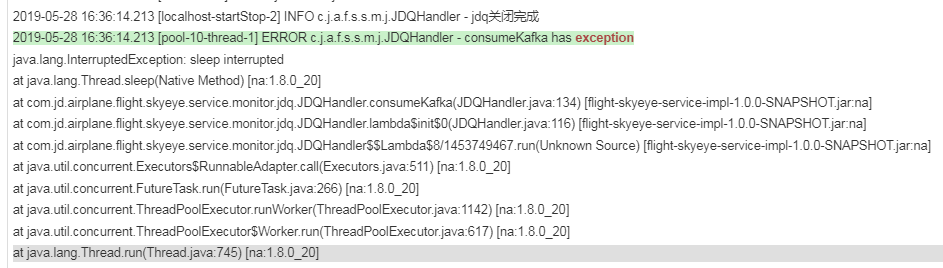
因为rerun出了while方法，和手动提交没有关系。

情况2：在main线程之外的另一个异步线程newSingleThreadExecutor中poll并进行所有的消费，若消费的数据为null则continue，正常执行

情况3：在main线程之外的另一个异步线程newSingleThreadExecutor中while (!Thread.currentThread().isInterrupted()) 循环poll，然后offer（value，timeout）放在ArrayBlockingQueue阻塞队列中，再开启异步线程池从队列中pool（），进行所有的消费流程。在newSingleThreadExecutor中sleep上10s后。调用consumer.commitSync()。

会抛出InterruptedException。这时候没有配置enable.auto.commit=false，默认是true，即5000ms提交一次点位

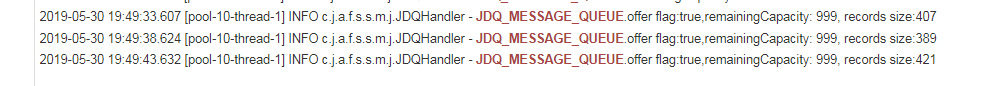
java.lang.InterruptedException: sleep interrupted



情况4：在3的基础上去掉改成while（true）+ sleep5秒，这次没有异常，但是拉取一会消息，就不消费了

还是不行

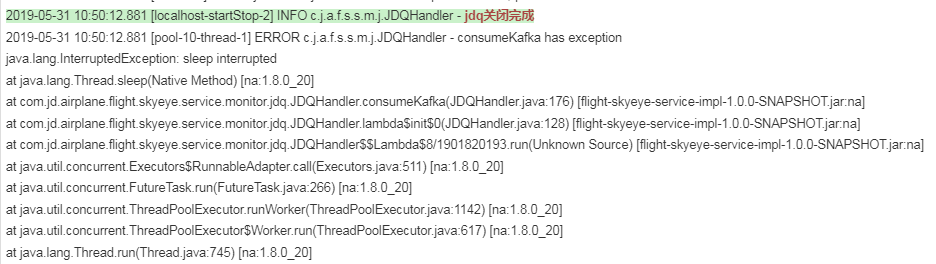
情况5：在3的基础上去掉newSingleThreadExecutor的sleep，还是不行







java.lang.InterruptedException: sleep interrupted



1. 加while循环内throwable
2. flag = JDQ\_MESSAGE\_QUEUE.offer(records, 5 , TimeUnit.SECONDS)由两分钟改成5秒
3. 在while循环外catch异常throwable

#### 真正的原因：



