咕泡学院 VIP 课 :分布式消息通信 kafka 原理分析

课程目标

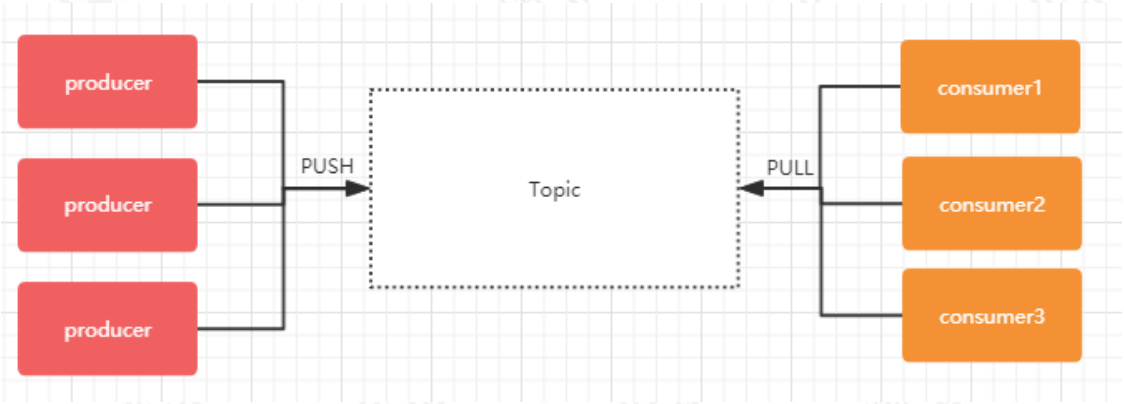
1. Topic&Partition
2. 消息分发策略
3. 消息消费原理
4. 消息的存储策略
5. Partition 副本机制

关于 Topic 和 Partition

Topic

在 kafka 中，topic 是一个存储消息的逻辑概念，可以认为是一个消息集合。每条消息发送到 kafka 集群的消息都有一个类别。物理上来说，不同的 topic 的消息是分开存储的，

每个 topic 可以有多个生产者向它发送消息，也可以有多个消费者去消费其中的消息。

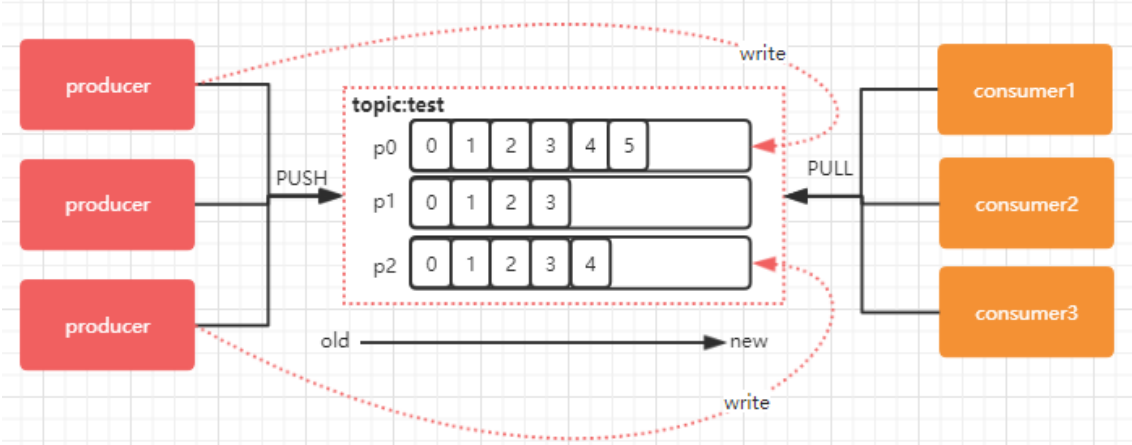


Partition

每个 topic 可以划分多个分区（每个 Topic 至少有一个分区），同一 topic 下的不同分区包含的消息是不同的。每个消息在被添加到分区时，都会被分配一个 offset（称之为偏移量），它是消息在此分区中的唯一编号，kafka 通过 offset 保证消息在分区内的顺序，offset 的顺序不跨分区，即 kafka 只保证在同一个分区内的消息是有序的。

下图中，对于名字为 test 的 topic，做了 3 个分区，分别是 p0、p1、p2.

- 每一条消息发送到 broker 时，会根据 partition 的规则选择存储到哪一个 partition。如果 partition 规则设置合理，那么所有的消息会均匀的分布在不同的 partition 中，这样就有点类似数据库的分库分表的概念，把数据做了分片处理。



Topic&Partition 的存储

Partition 是以文件的形式存储在文件系统中，比如创建一个名为 firstTopic 的 topic，其中有 3 个 partition，那么在 kafka 的数据目录（/tmp/kafka-log）中就有 3 个目录，firstTopic-0~3，命名规则是<topic_name>-<partition_id>

./kafka-topics.sh --create --zookeeper 192.168.11.156:2181 --replication-factor 1 --partitions 3 --topic firstTopic

关于消息分发

kafka 消息分发策略

消息是 kafka 中最基本的数据单元，在 kafka 中，一条消息由 key、value 两部分构成，在发送一条消息时，我们可以指定这个 key，那么 producer 会根据 key 和 partition 机制来判断当前这条消息应该发送并存储到哪个 partition 中。我们可以根据需要进行扩展 producer 的 partition 机制。

代码演示

参考课堂代码->gitlab

消息默认的分发机制

默认情况下，kafka 采用的是 hash 取模的分区算法。如果 Key 为 null，则会随机分配一个分区。这个随机是在这个参数"metadata.max.age.ms"的时间范围内随机选择一个。对于这个时间段内，如果 key 为 null，则只会发送到唯一的分区。这个值值哦默认情况下是 10 分钟更新一次。

关于 Metadata，这个之前没讲过，简单理解就是 Topic/Partition 和 broker 的映射关系，每一个 topic 的每一个 partition，需要知道对应的 broker 列表是什么，leader 是谁、follower 是谁。这些信息都是存储在 Metadata 这个

类里面。

消费端如何消费指定的分区

通过下面的代码，就可以消费指定该 topic 下的 0 号分区。其他分区的数据就无法接收

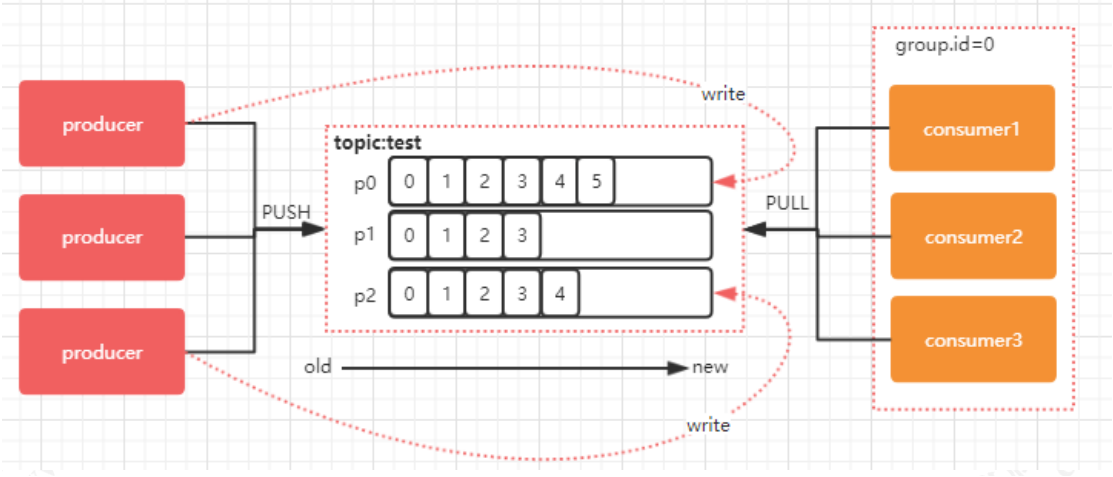
```
//消费指定分区的时候，不需要再订阅
//kafkaConsumer.subscribe(Collections.singletonList(topic));
//消费指定的分区
TopicPartition topicPartition=new TopicPartition(topic,0);
kafkaConsumer.assign(Arrays.asList(topicPartition));
```

消息的消费原理

kafka 消息消费原理演示

在实际生产过程中，每个 topic 都会有多个 partitions，多个 partitions 的好处在于，一方面能够对 broker 上的数据进行分片有效减少了消息的容量从而提升 io 性能。另外一方面，为了提高消费端的消费能力，一般会通过多个 consumer 去消费同一个 topic ，也就是消费端的负载均

衡机制，也就是我们接下来要了解的，在多个 partition 以及多个 consumer 的情况下，消费者是如何消费消息的同时，在上一节课，我们讲了，kafka 存在 consumer group 的概念，也就是 group.id 一样的 consumer，这些 consumer 属于一个 consumer group，组内的所有消费者协调在一起来消费订阅主题的所有分区。当然每一个分区只能由同一个消费组内的 consumer 来消费，那么同一个 consumer group 里面的 consumer 是怎么去分配该消费哪个分区里的数据的呢？如下图所示，3 个分区，3 个消费者，那么哪个消费者消分哪个分区？



对于上面这个图来说，这 3 个消费者会分别消费 test 这个 topic 的 3 个分区，也就是每个 consumer 消费一个 partition。

什么是分区分配策略

通过前面的案例演示，我们应该能猜到，同一个 group 中

的消费者对于一个 topic 中的多个 partition，存在一定的分区分配策略。

在 kafka 中，存在两种分区分配策略，一种是 Range(默认)、另一种另一种还是 RoundRobin（轮询）。 通过 partition.assignment.strategy 这个参数来设置。

Range strategy（范围分区）

Range 策略是对每个主题而言的，首先对同一个主题里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序。假设我们有 10 个分区，3 个消费者，排完序的分区将会是 0, 1, 2, 3, 4, 5, 6, 7, 8, 9；消费者线程排完序将会是 C1-0, C2-0, C3-0。然后将 partitions 的个数除于消费者线程的总数来决定每个消费者线程消费几个分区。如果除不尽，那么前面几个消费者线程将会多消费一个分区。在我们的例子里面，我们有 10 个分区，3 个消费者线程， 10 / 3 = 3，而且除不尽，那么消费者线程 C1-0 将会多消费一个分区，所以最后分区分配

的结果看起来是这样的：

C1-0 将消费 0, 1, 2, 3 分区

C2-0 将消费 4, 5, 6 分区

C3-0 将消费 7, 8, 9 分区

假如我们有 11 个分区，那么最后分区分配的结果看起来是这

样的：

C1-0 将消费 0, 1, 2, 3 分区

C2-0 将消费 4, 5, 6, 7 分区

C3-0 将消费 8, 9, 10 分区

假如我们有 2 个主题(T1 和 T2)，分别有 10 个分区，那么最后分区分配的结果看起来是这样的：

C1-0 将消费 T1 主题的 0, 1, 2, 3 分区以及 T2 主题的 0, 1, 2, 3 分区

C2-0 将消费 T1 主题的 4, 5, 6 分区以及 T2 主题的 4, 5, 6 分区

C3-0 将消费 T1 主题的 7, 8, 9 分区以及 T2 主题的 7, 8, 9 分区

可以看出，C1-0 消费者线程比其他消费者线程多消费了 2 个分区，这就是 Range strategy 的一个很明显的弊端

RoundRobin strategy（轮询分区）

轮询分区策略是把所有 partition 和所有 consumer 线程都列出来，然后按照 hashcode 进行排序。最后通过轮询算法分配 partition 给消费线程。如果所有 consumer 实例的订阅是相同的，那么 partition 会均匀分布。

在我们的例子里面，假如按照 hashCode 排序完的 topic-partitions 组依次为 T1-5, T1-3, T1-0, T1-8, T1-2, T1-1, T1-4,

T1-7, T1-6, T1-9，我们的消费者线程排序为 C1-0, C1-1, C2-0, C2-1，最后分区分配的结果为：

C1-0 将消费 T1-5, T1-2, T1-6 分区；

C1-1 将消费 T1-3, T1-1, T1-9 分区；

C2-0 将消费 T1-0, T1-4 分区；

C2-1 将消费 T1-8, T1-7 分区；

使用轮询分区策略必须满足两个条件

1. 每个主题的消费者实例具有相同数量的流
2. 每个消费者订阅的主题必须是相同的

什么时候会触发这个策略呢？

当出现以下几种情况时，kafka 会进行一次分区分配操作，也就是 kafka consumer 的 rebalance

1. 同一个 consumer group 内新增了消费者
2. 消费者离开当前所属的 consumer group，比如主动停机或者宕机
3. topic 新增了分区（也就是分区数量发生了变化）

kafka consuemr 的 rebalance 机制规定了一个 consumer group 下的所有 consumer 如何达成一致来分配订阅 topic 的每个分区。而具体如何执行分区策略，就是前面提到过的两种内置的分区策略。而 kafka 对于分配策略这块，提供了可插拔的实现方式， 也就是说，除了这两种之外，我

们还可以创建自己的分配机制。

谁来执行 Rebalance 以及管理 consumer 的 group 呢？

Kafka 提供了一个角色：coordinator 来执行对于 consumer group 的管理，Kafka 提供了一个角色：coordinator 来执行对于 consumer group 的管理，当 consumer group 的第一个 consumer 启动的时候，它会去和 kafka server 确定谁是它们组的 coordinator。之后该 group 内的所有成员都会和该 coordinator 进行协调通信

如何确定 coordinator

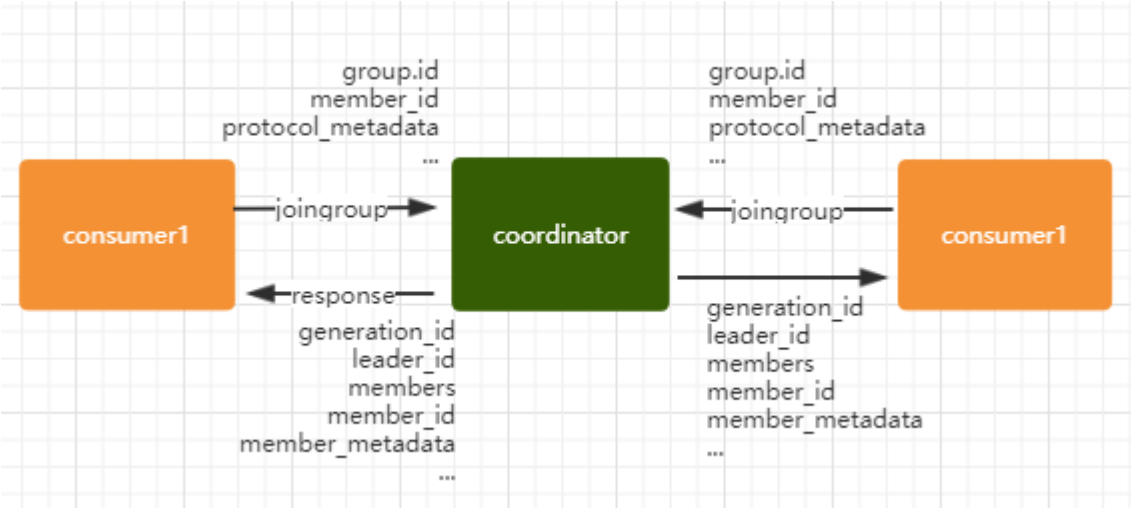
consumer group 如何确定自己的 coordinator 是谁呢，消费者向 kafka 集群中的任意一个 broker 发送一个 GroupCoordinatorRequest 请求，服务端会返回一个负载最小的 broker 节点的 id，并将该 broker 设置为 coordinator

JoinGroup 的过程

在 rebalance 之前，需要保证 coordinator 是已经确定好了的，整个 rebalance 的过程分为两个步骤，Join 和 Sync

join：表示加入到 consumer group 中，在这一步中，所有的成员都会向 coordinator 发送 joinGroup 的请求。一旦

所有成员都发送了 joinGroup 请求，那么 coordinator 会选择一个 consumer 担任 leader 角色，并把组成员信息和订阅信息发送消费者



protocol_metadata：序列化后的消费者的订阅信息

leader_id： 消费组中的消费者，coordinator 会选择一个座位 leader，对应的就是 member_id

member_metadata 对应消费者的订阅信息

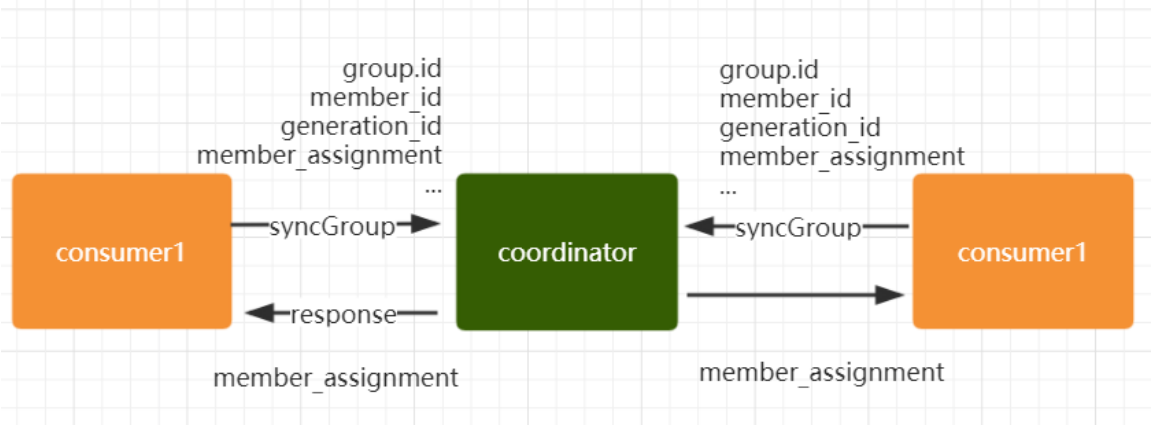
members：consumer group 中全部的消费者的订阅信息

generation_id：年代信息，类似于之前讲解 zookeeper 的时候的 epoch 是一样的，对于每一轮 rebalance，generation_id 都会递增。主要用来保护 consumer group。隔离无效的 offset 提交。也就是上一轮的 consumer 成员无法提交 offset 到新的 consumer group 中。

Synchronizing Group State 阶段

完成分区分配之后，就进入了 Synchronizing Group State

阶段，主要逻辑是向 GroupCoordinator 发送 SyncGroupRequest 请求，并且处理 SyncGroupResponse 响应，简单来说，就是 leader 将消费者对应的 partition 分配方案同步给 consumer group 中的所有 consumer



每个消费者都会向 coordinator 发送 syncgroup 请求，不过只有 leader 节点会发送分配方案，其他消费者只是打打酱油而已。当 leader 把方案发给 coordinator 以后，coordinator 会把结果设置到 SyncGroupResponse 中。这样所有成员都知道自己应该消费哪个分区。

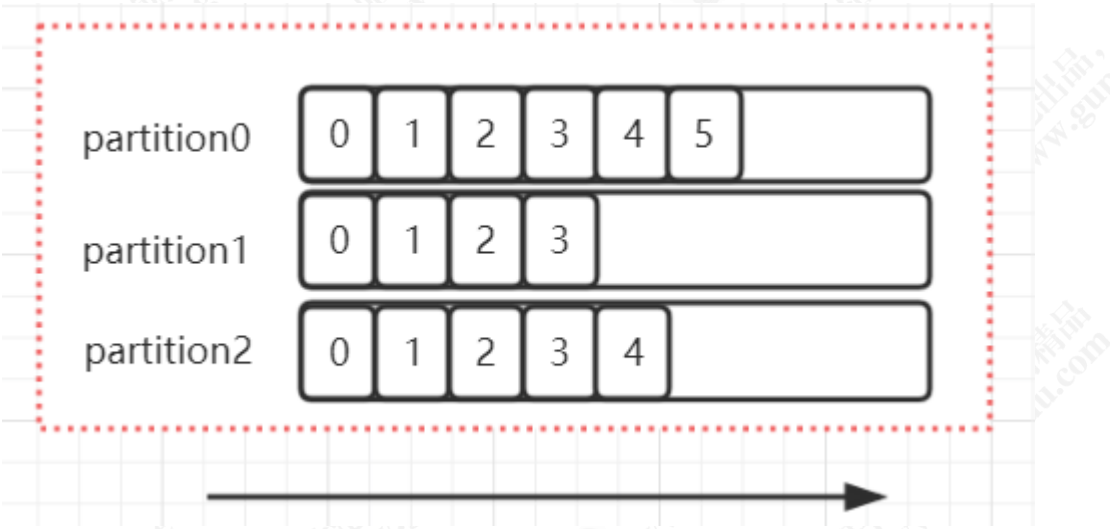
- consumer group 的分区分配方案是在客户端执行的！Kafka 将这个权利下放给客户端主要是因为这样做可以有更好的灵活性

如何保存消费端的消费位置

什么是 offset

前面在讲解 partition 的时候，提到过 offset， 每个 topic

可以划分多个分区（每个 Topic 至少有一个分区），同一 topic 下的不同分区包含的消息是不同的。每个消息在被添加到分区时，都会被分配一个 offset（称之为偏移量），它是消息在此分区中的唯一编号，kafka 通过 offset 保证消息在分区内的顺序，offset 的顺序不跨分区，即 kafka 只保证在同一个分区内的消息是有序的；对于应用层的消费来说，每次消费一个消息并且提交以后，会保存当前消费到的最近的一个 offset。那么 offset 保存在哪里？



offset 在哪里维护？

在 kafka 中，提供了一个__consumer_offsets_* 的一个 topic，把 offset 信息写入到这个 topic 中。__consumer_offsets——按保存了每个 consumer group 某一时刻提交的 offset 信息。__consumer_offsets 默认有 50 个分区。

根据前面我们演示的案例，我们设置了一个

KafkaConsumerDemo 的 groupid。首先我们需要找到这个 consumer_group 保存在哪个分区中

```
properties.put(ConsumerConfig.GROUP_ID_CONFIG,
"KafkaConsumerDemo");
```

计算公式

- Math.abs("groupid".hashCode())%groupMetadataTopicPartitionCount ；由于默认情况下 groupMetadataTopicPartitionCount 有 50 个分区，计算得到的结果为:35，意味着当前的 consumer_group 的位移信息保存在__consumer_offsets 的第 35 个分区
- 执行如下命令，可以查看当前 consumer_goup 中的 offset 位移信息

  sh kafka-simple-consumer-shell.sh --topic __consumer_offsets --partition 35 --broker-list 192.168.11.153:9092,192.168.11.154:9092,192.168.11.157:9092 --formatter

  "kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter"

从输出结果中，我们就可以看到 test 这个 topic 的 offset 的位移日志

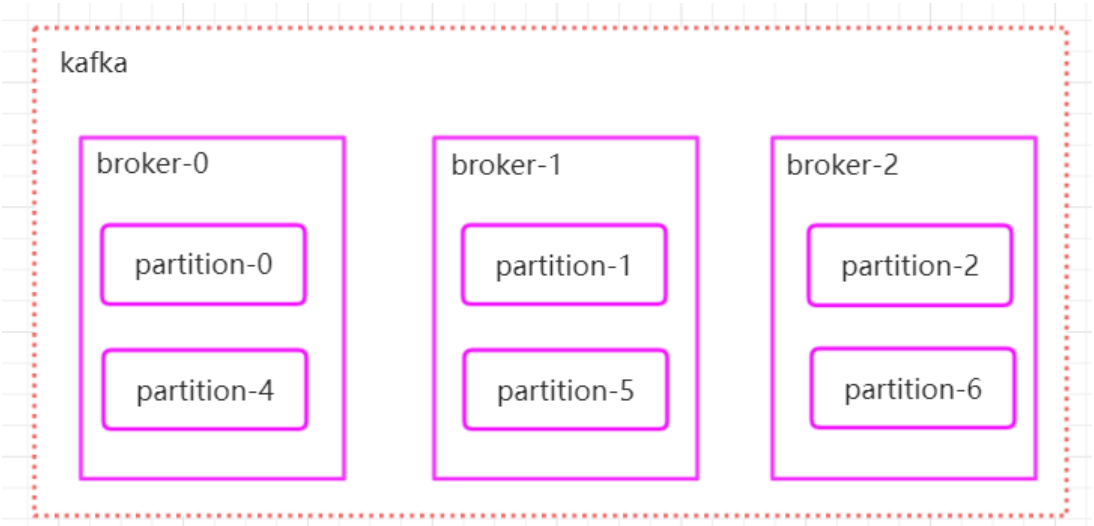
消息的存储

消息的保存路径

消息发送端发送消息到 broker 上以后，消息是如何持久化的呢？那么接下来去分析下消息的存储
首先我们需要了解的是，kafka 是使用日志文件的方式来保存生产者和发送者的消息，每条消息都有一个 offset 值来表示它在分区中的偏移量。Kafka 中存储的一般都是海量的消息数据，为了避免日志文件过大，Log 并不是直接对应在一个磁盘上的日志文件，而是对应磁盘上的一个目录，这个目录的明明规则是<topic_name>_<partition_id>
比如创建一个名为 firstTopic 的 topic，其中有 3 个 partition，那么在 kafka 的数据目录（/tmp/kafka-log）中就有 3 个目录，firstTopic-0~3

多个分区在集群中的分配

如果我们对于一个 topic，在集群中创建多个 partition，那么 partition 是如何分布的呢？
1.将所有 N Broker 和待分配的 i 个 Partition 排序
2.将第 i 个 Partition 分配到第(i mod n)个 Broker 上



了解到这里的时候，大家再结合前面讲的消息分发策略，就应该能明白消息发送到 broker 上，消息会保存到哪个分区中，并且消费端应该消费哪些分区的数据了。

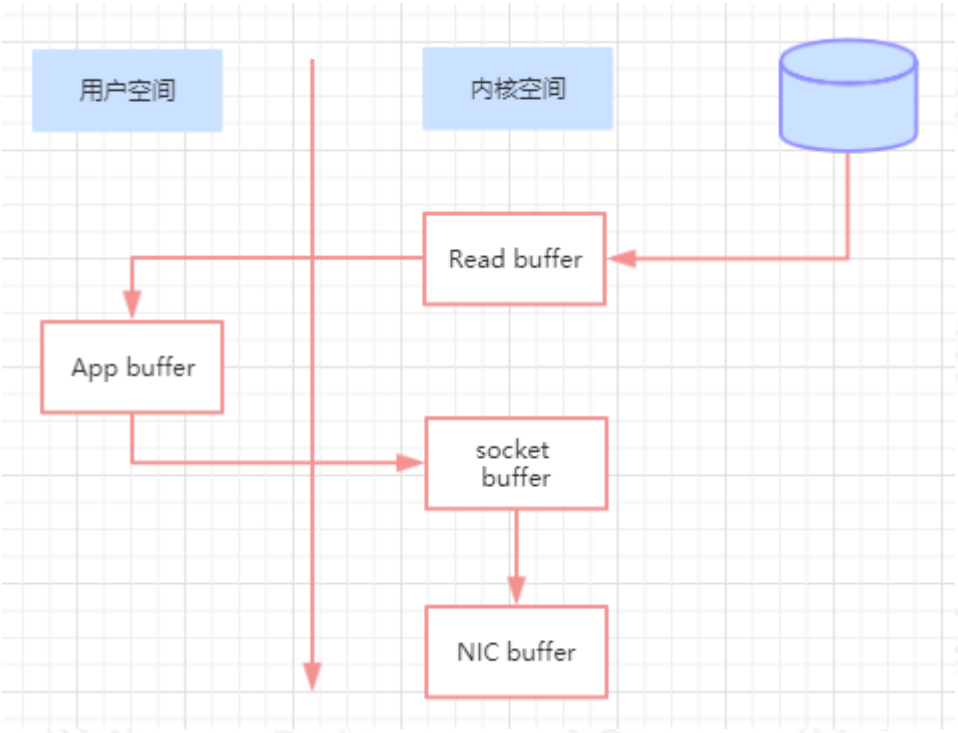
消息写入的性能

我们现在大部分企业仍然用的是机械结构的磁盘，如果把消息以随机的方式写入到磁盘，那么磁盘首先要做的就是寻址，也就是定位到数据所在的物理地址，在磁盘上就要找到对应的柱面、磁头以及对应的扇区；这个过程相对内存来说会消耗大量时间，为了规避随机读写带来的时间消耗，kafka 采用顺序写的方式存储数据。即使是这样，但是频繁的 I/O 操作仍然会造成磁盘的性能瓶颈，所以 kafka 还有一个性能策略

零拷贝

消息从发送到落地保存，broker 维护的消息日志本身就是

文件目录，每个文件都是二进制保存，生产者和消费者使用相同的格式来处理。在消费者获取消息时，服务器先从硬盘读取数据到内存，然后把内存中的数据原封不动的通过 socket 发送给消费者。虽然这个操作描述起来很简单，但实际上经历了很多步骤。

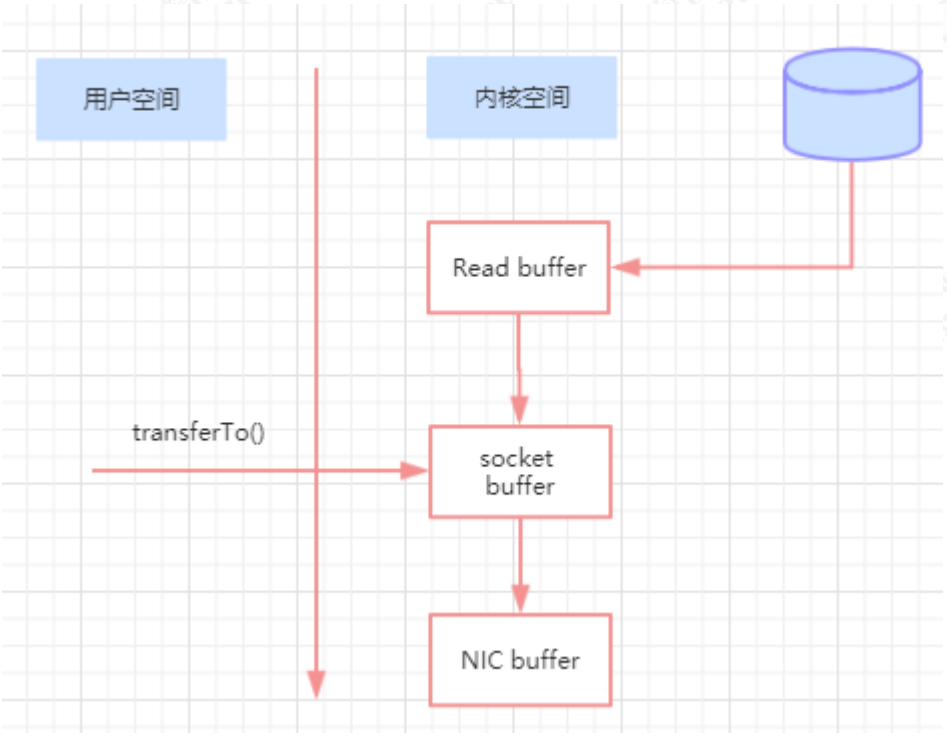


- 操作系统将数据从磁盘读入到内核空间的页缓存
- 应用程序将数据从内核空间读入到用户空间缓存中
- 应用程序将数据写回到内核空间到 socket 缓存中
- 操作系统将数据从 socket 缓冲区复制到网卡缓冲区，以便将数据经网络发出

这个过程涉及到 4 次上下文切换以及 4 次数据复制，并且有两

次复制操作是由 CPU 完成。但是这个过程中，数据完全没有进行变化，仅仅是从磁盘复制到网卡缓冲区。

通过“零拷贝”技术，可以去掉这些没必要的数据复制操作，同时也会减少上下文切换次数。现代的 unix 操作系统提供一个优化的代码路径，用于将数据从页缓存传输到 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



使用 sendfile，只需要一次拷贝就行，允许操作系统将数据直接从页缓存发送到网络上。所以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