课程目标

1、掌握 Kafka 消费者 Offset 维护原理

2、掌握 Kafka 消费者与分区的关系

3、了解 Kafka 高性能顺序读写

4、了解 Kafka 零拷贝机制

1. Kafka 消费者原理

这一篇我们学习消费者的原理，主要包括消费者和 Partition 的关系以及消费的一些问题。

1.1 Offset 的维护

我们先来看下消费者怎么消费消息。

1.1.1 Offset 的存储

我们知道在 Partition 中，消息是不会删除的，所以才可以追加写入，写入的消息连续有序的。

这种特性决定了 Kafka 可以消费历史消息，而且按照消息的顺序消费指定消息，而不是只能消费队头的消息。

正常情况下，我们希望消费没有被消费过的数据，而且是从最先发送（序号小的）的开始消费（这样才是有序和公平的）。

从 SimpleConsumer 和 SimpleProducer 中看到的默认结果也是这样的。

对于一个 Partition，消费者组怎么才能做到接着上次消费的位置（Offset）继续消费呢？肯定要把这个对应关系保存起来，下次消费的时候查找一下。

（还有一种方式是根据时间戳消费）

首先这个对应关系确实是可以查看的。比如消费者组 gp-assign-group-1 和 ass5part（5 个分区）的 Partition 的偏移量关系，可使用如下命令查看：

```
./kafka-consumer-groups.sh --bootstrap-server 192.168.8.144:9092,192.168.8.145:9092,192.168.8.146:9092
--describe --group gp-assign-group-1
```

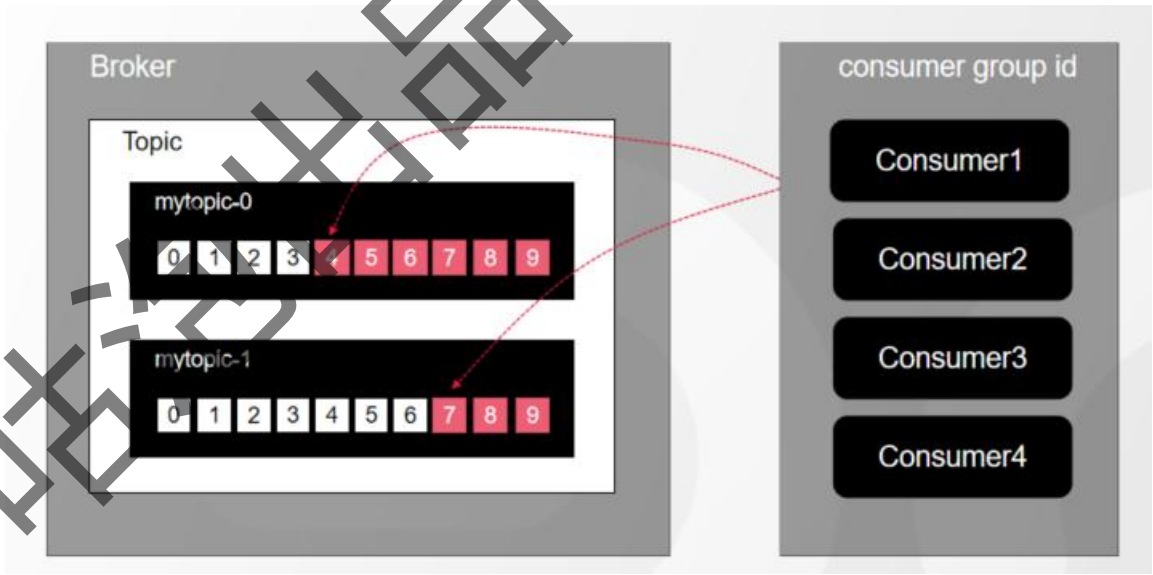
PARTITION	CURRENT-OFFSET	LOG-END-OFFSET	LAG	CONSUMER-ID
0	5	5	0	consumer-1
1	5	5	0	consumer-1
2	5	5	0	consumer-1
3	5	5	0	consumer-2
4	5	5	0	consumer-2

* CURRENT-OFFSET 指的是下一个未使用的 Offset。

* LEO ，Log End Offset：下一条等待写入的消息的 Offset（最新的 Offset + 1）

* LAG 是延迟量

注意：不是一个消费者和一个 Topic 的关系。是一个 consumer group 和 Topic 中的一个 Partition 的关系（Offset 在 Partition 中连续编号而不是全局连续编号）。



这个对应关系到底是保存在哪里的呢？首先肯定是不可能放在消费者本地的。为什么？因为所有的消费者都可以使用这个 Consumer Group id，放在本地是做不到统一维护的，肯定要放到服务端。

Kafka 早期的版本把消费者组和 Partition 的 Offset 直接维护在 ZK 中，但是读写

的性能消耗太大了。后来就放在一个特殊的 Topic 中，名字叫__consumer_offsets，默认有 50 个分区（offsets.topic.num.partitions 默认是 50），每个分区默认一个 Replication。

```
./kafka-topics.sh --topic __consumer_offsets --describe --bootstrap-server 192.168.8.146:9092
```

看起来这些分区副本在 3 个 Broker 上非常均匀和轮流地分布(123 123 123.....)。

这样一个特殊的 Topic 怎么存储消费者组 gp-assign-group-1 对于分区的偏移量的?

Topic 里面是可以存放对象类型的 value 的 (经过序列化和反序列化) 。这个 Topic 里面主要存储两种对象:

GroupMetadata：保存了消费者组中各个消费者的信息（每个消费者有编号）。

OffsetAndMetadata：保存了消费者组和各个 partition 的 offset 位移信息元数据。

```
./kafka-console-consumer.sh --topic __consumer_offsets --bootstrap-server
192.168.8.144:9092,192.168.8.145:9092,192.168.8.146:9092 --formatter
"kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter" --from-beginning
```

不加—from-beginning 消费不到消息。

[Group, Topic, Partition]::[OffsetMetadata[Offset, Metadata], CommitTime, ExpirationTime]

[gp-assign-group-1,ass5part,0]::OffsetAndMetadata(offset=6, leaderEpoch=Optional[2], metadata=, commitTimestamp=1596726098944, expireTimestamp=None)

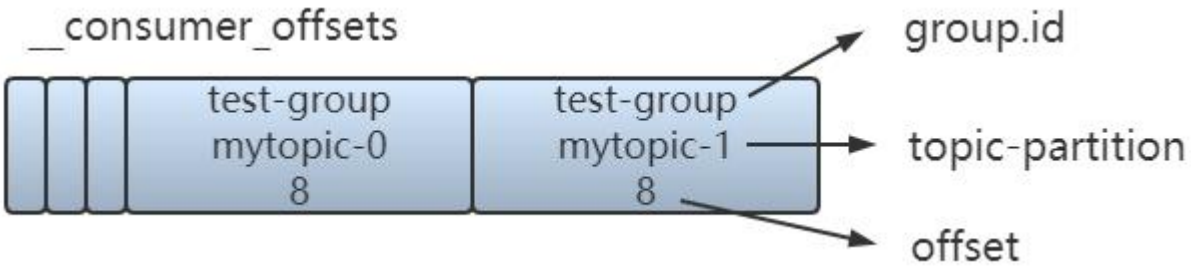
[gp-assign-group-1,ass5part,1]::OffsetAndMetadata(offset=6, leaderEpoch=Optional[0], metadata=, commitTimestamp=1596726098944, expireTimestamp=None)

[gp-assign-group-1,ass5part,2]::OffsetAndMetadata(offset=6, leaderEpoch=Optional[2], metadata=, commitTimestamp=1596726098944, expireTimestamp=None)

[gp-assign-group-1,ass5part,3]::OffsetAndMetadata(offset=6, leaderEpoch=Optional[2], metadata=, commitTimestamp=1596726099944, expireTimestamp=None)

[gp-assign-group-1,ass5part,4]::OffsetAndMetadata(offset=6, leaderEpoch=Optional[0], metadata=, commitTimestamp=1596726099944, expireTimestamp=None)

__consumer_offsets 大致的数据结构是这个样子的:



怎么知道一个 consumer group 的 Offset 会放在这个特殊的 Topic 的哪个分区呢?

com.gupaoedu.vip.mq.kafka.javaapi.simple.HashCalculate

```
System.out.println(Math.abs("gp-assign-group-1".hashCode()) % 50);
```

1.1.2 如果找不到 Offset

当然，这个是 Broker 有记录 Offset 的情况，如果说增加了一个新的消费者组去消费一个 Topic 的某个 Partion，没有 Offset 的记录，这个时候应该从哪里开始消费呢?

什么情况下找不到 Offset? 就是你没有消费过，没有把当前的 Offset 上报给 Broker。

消费者的代码中有一个参数，用来控制如果找不到偏移量的时候从哪里开始消费。

com.gupaoedu.vip.mq.kafka.javaapi.simple.SimpleConsumer

```
auto.offset.reset
```

What to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server (e.g. because that data has been deleted):

- **earliest: automatically reset the offset to the earliest offset**
- **latest: automatically reset the offset to the latest offset**
- **none: throw exception to the consumer if no previous offset is found for the consumer's group**
- **anything else: throw exception to the consumer.**

默认值是 latest，也就是从最新的消息（最后发送的）开始消费的。历史消费是不能消费的。

earliest 代表从最早的（最先发送的）消息开始消费。可以消费到历史消息。

none，如果 Consumer Group 在服务端找不到 offset 会报错。

1.1.3 Offset 的更新

前面我们讲了，消费者组的 Offset 是保存在 Broker 的，但是是由消费者上报给 Broker 的。并不是消费者组消费了消息，Offset 就会更新，消费者必须要有一个 Commit（提交）的动作。就跟 RabbitMQ 中消费者的 ACK 一样。

一样的，消费者可以自动提交或者手动提交。由消费端的这个参数控制：

com.gupaoedu.vip.mq.kafka.javaapi.simple.SimpleConsumer

```
enable.auto.commit
```

默认是 true。true 代表消费者消费消息以后自动提交此时 Broker 会更新消费者组的 Offset。

另外还可以使用一个参数来控制自动提交的频率：

```
auto.commit.interval.ms
```

默认是 5 秒钟。

如果我们要在消费完消息做完业务逻辑处理之后才 Commit，就要把这个值改成 false。如果是 false，消费者就必须要调用一个方法让 Broker 更新 Offset。

有两种方式：

consumer.commitSync()的手动同步提交。

consumer.commitAsync()手动异步提交。

代码：com.gupaoedu.vip.mq.kafka.javaapi.commit.CommitConsumer

```
while (true) {
  ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
  for (ConsumerRecord<String, String> record : records) {
```

```
        System.out.printf("offset = %d ,key =%s, value= %s,
partition= %s%n" ,record.offset(),record.key(),record.value(),record.partition())
;
        buffer.add(record);
}
  if (buffer.size() >= minBatchSize) {
        // 同步提交
        consumer.commitSync();
        buffer.clear();
}
}
```

如果不提交或者提交失败，Broker 的 Offset 不会更新，消费者组下次消费的时候会消费到重复的消息。

1.2 消费者消费策略（消费者与分区关系）

1.2.1 消费策略

第一个问题：多个 Consumer Group 和 Partition 的关系：重复消费。任何一个消费者组，都会把一个 Topic 的所有 Partition 瓜分干净。

前面我们讲过，一个消费者组里面的一个消费者，只能消费 Topic 的一个分区。

如果分区数量跟消费者的数量一样，那就一人消费一个就好了。如果是消费者比分区多，或者消费者比分区少，这个时候消费者跟分区的关系是怎么样的呢？

如果比消费者比分区多，肯定是有一些消费者消费不到的（站着上课）。

例如：2 个消费者消费 5 个分区，怎么分配呢？

先看默认的情况。

我们创建了一个 5 个分区的 Topic。

```
./kafka-topics.sh --create --bootstrap-server 192.168.8.146:9092 --partitions 5 --replication-factor 1 --topic
ass5part
```

启动两个消费者消费（同一个消费者组，目标是消费同一个 Partition，不同的 client id）：

消费者：

com.gupaoedu.vip.mq.kafka.javaapi.assign.ConsumerAutoAssignTest

```
// 两个消费者消费 5 个分区（同一个消费者组）
KafkaConsumer<String,String> consumer1=new KafkaConsumer<String, String>(props);
KafkaConsumer<String,String> consumer2=new KafkaConsumer<String, String>(props);

// 订阅队列
consumer1.subscribe(Arrays.asList("ass5part"));
consumer2.subscribe(Arrays.asList("ass5part"));
```

给 5 个分区分别发送 1 条消息：

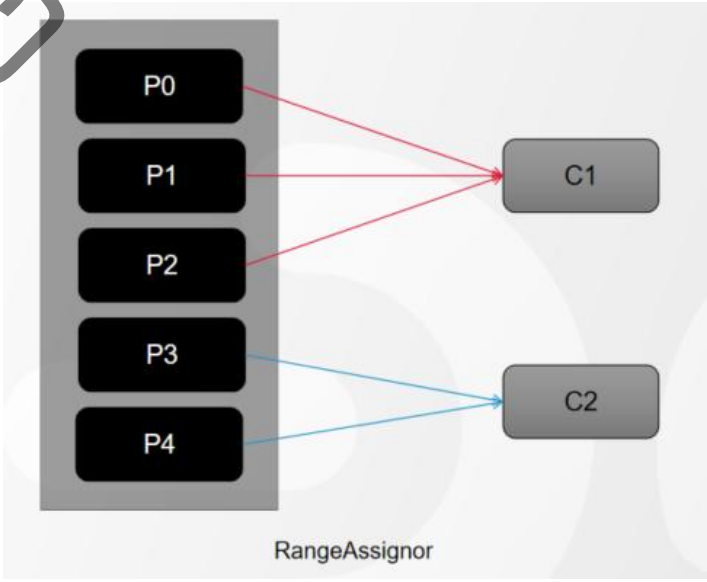
生产者：com.gupaoedu.vip.mq.kafka.javaapi.assign.ProducerSendTest

```
producer.send(new ProducerRecord<String,String>("ass5part",0,"0","0"));
producer.send(new ProducerRecord<String,String>("ass5part",1,"1","1"));
producer.send(new ProducerRecord<String,String>("ass5part",2,"2","2"));
producer.send(new ProducerRecord<String,String>("ass5part",3,"3","3"));
producer.send(new ProducerRecord<String,String>("ass5part",4,"4","4"));
```

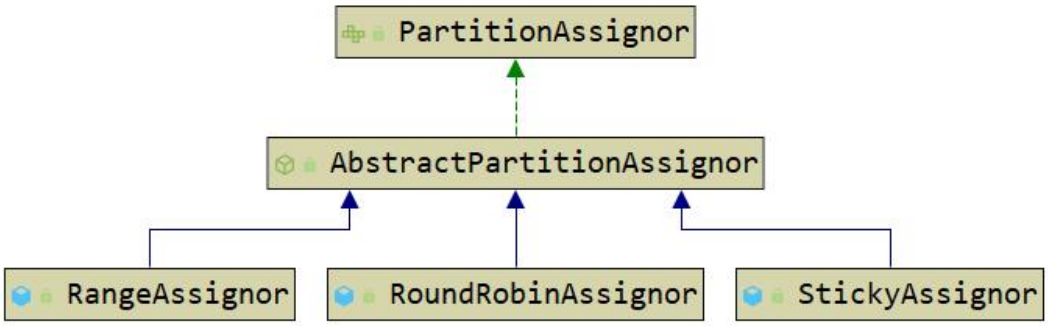
结果（打印顺序是不一定的）：

```
----consume1----offset = 4 ,key =0, value= 0, partition= 0
----consume1----offset = 4 ,key =1, value= 1, partition= 1
----consume1----offset = 4 ,key =2, value= 2, partition= 2
----consume2----offset = 4 ,key =3, value= 3, partition= 3
----consume2----offset = 4 ,key =4, value= 4, partition= 4
```

也就是，按照范围连续分配的，你一坨，我一坨。



实际上是采用了默认的策略：RangeAssignor。



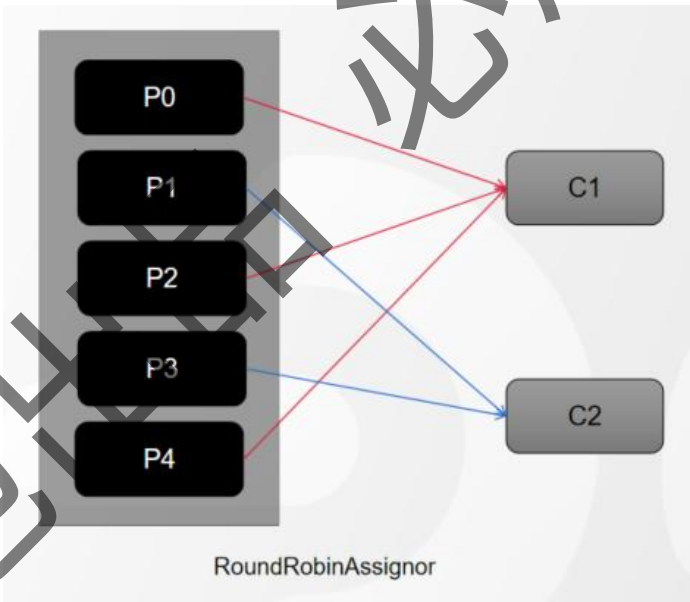
也通过 partition.assignment.strategy 修改为其他的消费策略：

代码：com.gupaoedu.vip.mq.kafka.javaapi.assign.ConsumerAutoAssignTest

```
props.put("partition.assignment.strategy","org.apache.kafka.clients.consumer.RoundRobinAssignor");
```

另外两种策略的消费结果：

RoundRobinAssignor（轮询）： c1 : p0 p2 p4　　c2 : p1 p3



StickyAssignor（粘滞）：这种策略复杂一点，但是相对来说均匀一点（每次的结果都可能不一样）。原则：

1）分区的分配尽可能的均匀

2）分区的分配尽可能和上次分配保持相同

Consumer 可以指定 topic 的某个分区消费吗？我就喜欢讲台旁边的这个黄金座位，我可以坐么？

这个时候我们要用到 assign 而不是 subscribe 的接口。subscribe 会自动分配消费者组的分区，而 assign 可以手动指定分区消费，相当于 comsumer group id 失效了。

com.gupaoedu.vip.mq.kafka.javaapi.assign.ConsumerHandAssignTest

```
TopicPartition tp=new TopicPartition("ass5part",0);
consumer.assign(Arrays.asList(tp));
```

一个新的问题来了：

站着上课的这个人一直是站着上课吗？有没有可能给他一点机会呢？

我们前面说，在第一次消费（消费者启动）的时候，一个组的消费者和分区的消费关系就已经确定了，如果分配策略没变，关系是不会变动的。

问题：什么时候站着上课的学生才可能坐别人的座位？

1.2.2 ReBalance 分区再均衡

1.2.2.1 什么是 ReBalance？

ReBalance 本质上是一种协议，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。比如某个 Group 下有 20 个 Consumer，它订阅了一个具有 100 个分区的 Topic。正常情况下，Kafka 平均会为每个 Consumer 分配 5 个分区。这个分配的过程就叫 ReBalance。

1.2.2.2 什么时候 ReBalance？

有两种情况需要重新分配分区和消费者的关系：

1、消费者组的消费者数量发生变化，比如新增了消费者，消费者关闭连接 —— 学生数量变多了；

2、Topic 的分区数发生变更，新增或者减少 —— 座位数量发生了变化。

为了让分区分配尽量地均衡，这个时候会触发 ReBalance 机制。

1.2.2.4 谁来执行 ReBalance 和 Consumer Group 管理？

Kafka 提供了一个角色：Coordinator 来执行对于 Consumer Group 的管理。Kafka 早起版本中的 Coordinator 是依赖 Zookeeper 来实现的。

最新的版本中，Kafka 对 Coordinator 进行了改进。每个 Consumer Group 都会被分配一个 Coordinator 用于组管理和 Offser 管理。这个 Group 内的 Coordinator 比原来承担了更多的责任，比如组成员管理、Offset 提交保护机制等。Consumer Group 中的第一个 Consumer 启动的时候，它会去和 Kafka 服务确定谁是它们组的 Coordinator。然后 Group 内的所有成员都会和这个 Coordinator 进行协调通信。很显然，有了 Coordinator 这个设计，就不再需要 Zookeeper 了，性能上可以得到很大的提升。

我们知道了为什么要设计 Coordinator 这个角色，那么 Consumer Group 又是如何确定自己的 Coordinator 是谁的呢？ 其实非常简单，就是找到分区的 Leader 所在的 Broker，就会被选定为 Coordinator。

我帮大家简单地总结了一下，分区重新分配分成这么几步：

1、 找一个话事人，它起到一个监督和保证公平的作用。每个 Broker 上都有一

个用来管理 offset、消费者组的实例，叫做 GroupCoordinator。第一步就是要从所有 GroupCoordinator 中找一个话事人出来。

2、 第二步，清点一下人数。所有的消费者连接到 GroupCoordinator 报数，这个叫 join group 请求。

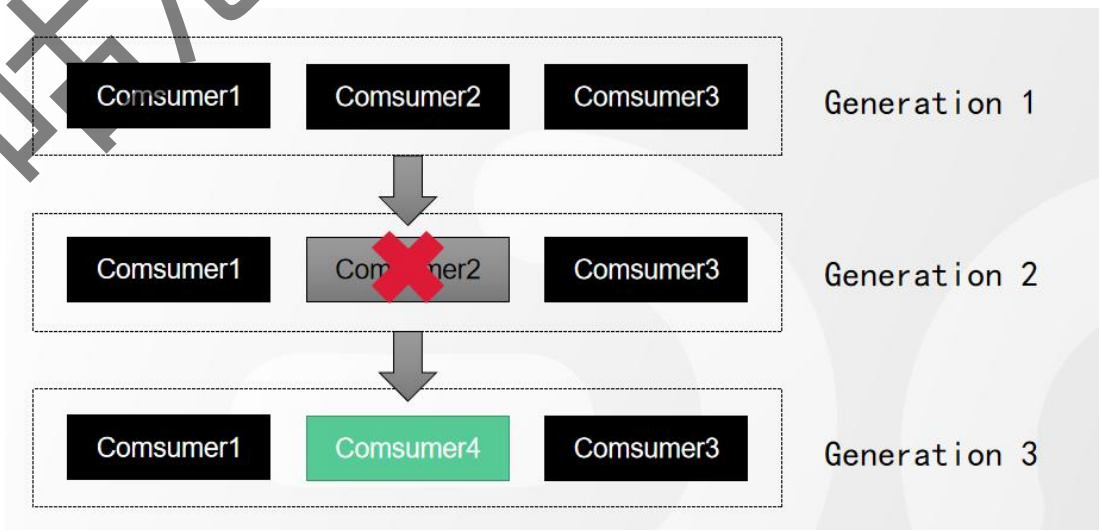
3、 第三步，选组长，GroupCoordinator 从所有消费者里面选一个 Leader。这个消费者会根据消费者的情况和设置的策略，确定一个方案。Leader 把方案上报给 GroupCoordinator，GroupCoordinator 会通知所有消费者。

其实他们不知道，已经内定好了，他们只是来陪跑的。

1.2.2.6 ReBalance Generation

JVM GC 的分代收集就是这个词，我这里把它翻译成“届”好了，它表示了 ReBalance 之后的一届成员，主要是用于保护 Consumer Group 隔离无效 Offset 提交。上一届的 Consumer 成员是无法提交 Offset 到新一届的 Consumer Group 中的。

每次 Group 进行 ReBalance 之后，Generation 号都会加 1，表示 Group 进入到了一个新的版本，如下图所示：Generation 1 时 Group 有 3 个成员，随后成员 2 退出组，Coordinator 触发 ReBalance，Consumer Group 进入 Generation 2，之后成员 4 加入，再次触发 ReBalance，Group 进入 Generation 3.



1.2.2.7 协议(Protocol)

前面说过，ReBalance 本质上是一组协议。Group 与 Coordinator 共同使用它来完成 Group 的 ReBalance。目前 Kafka 提供了 5 个协议来处理与 Consumer Group Coordination 相关的逻辑：

Heartbeat 请求：Consumer 需要定期给 Coordinator 发送心跳来表明自己还活着

LeaveGroup 请求：主动告诉 Coordinator 我要离开 Consumer Group

SyncGroup 请求：Group Leader 把分配方案告诉组内所有成员

JoinGroup 请求：成员请求加入组

DescribeGroup 请求：显示组的所有信息，包括成员信息，协议名称，分配方案，订阅信息等。通常该请求是给管理员使用

Coordinator 在 ReBalance 的时候主要用到了前面 4 种请求。

1.2.2.8 Liveness（不讲）

Consumer 如何向 Coordinator 证明自己还活着？ 通过定时向 Coordinator 发送 Heartbeat 请求。如果超过了设定的超时时间，那么 Coordinator 就认为这个 Consumer 已经挂了。一旦 Coordinator 认为某个 Consumer 挂了，那么它就会开启新一轮 ReBalance，并且在当前其他 Consumer 的心跳 Response 中添加“REBALANCE_IN_PROGRESS”，告诉其他 Consumer：不好意思各位，你们重新申请加入组吧！

1.2.2.9 Rebalance 过程（不讲）

终于说到 Consumer Group 执行 ReBalance 的具体流程了。大家可能估计对 Consumer 内部的工作机制也很感兴趣。下面就跟大家一起讨论一下。当然我必须要明确表示，ReBalance 的前提是 Coordinator 已经确定了。

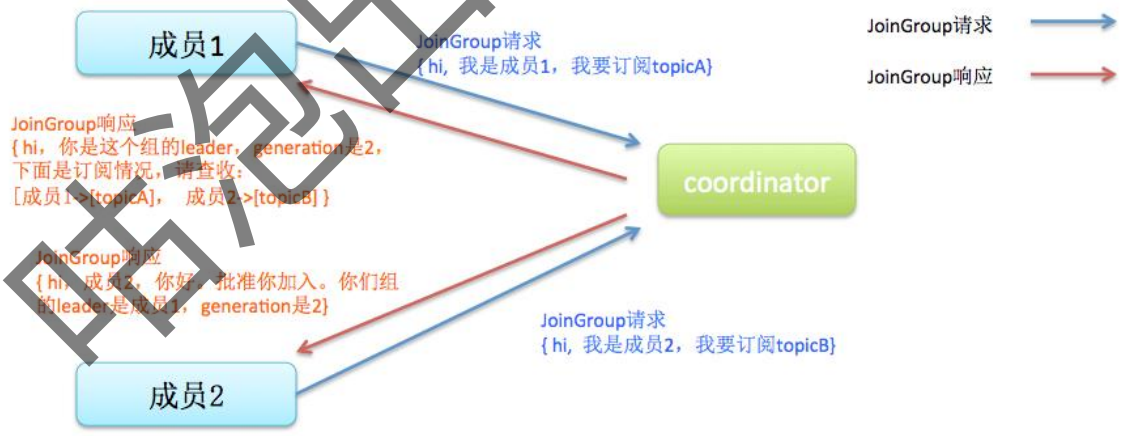
总体而言，Rebalance 分为 2 步：Join 和 Sync

1 Join，顾名思义就是加入组。这一步中，所有成员都向 Coordinator 发送 JoinGroup 请求，请求入组。一旦所有成员都发送了 JoinGroup 请求，Coordinator 会从中选择一个 Consumer 担任 Leader 的角色，并把组成员信息以及订阅信息发给 Leader。

注意：Leader 和 Coordinator 不是一个概念。Leader 负责消费分配方案的制定。

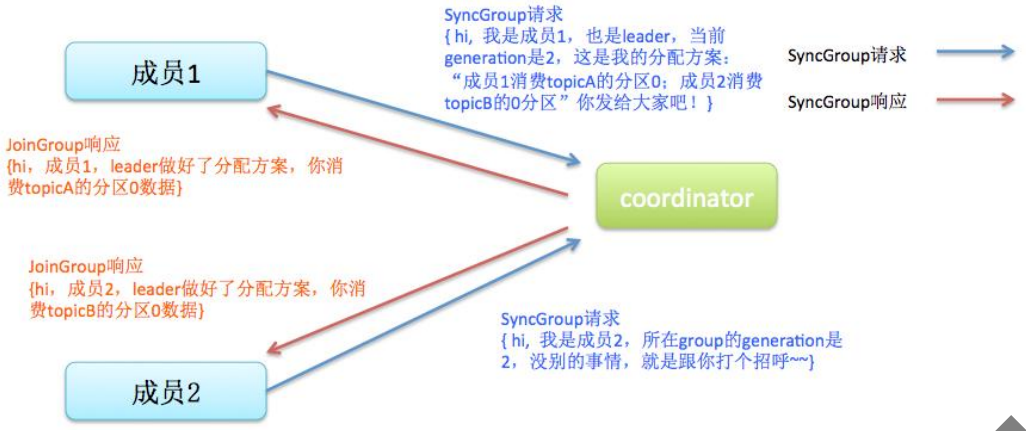
2 Sync，这一步 Leader 开始分配消费方案，即哪个 Consumer 负责消费哪些 Topic 的哪些 Partition。一旦完成分配，Leader 会将这个方案封装进 SyncGroup 请求中发给 Coordinator，非 Leader 也会发 SyncGroup 请求，只是内容为空。Coordinator 接收到分配方案之后会把方案塞进 SyncGroup 的 Response 中发给各个 Consumer。这样组内的所有成员就都知道自己应该消费哪些分区了。

还是拿几张图来说明吧，首先是加入组的过程:



值得注意的是， 在 Coordinator 收集到所有成员请求前，它会把已收到请求放入一个叫 Purgatory(炼狱)的地方。

然后是分发分配方案的过程，即 SyncGroup 请求:

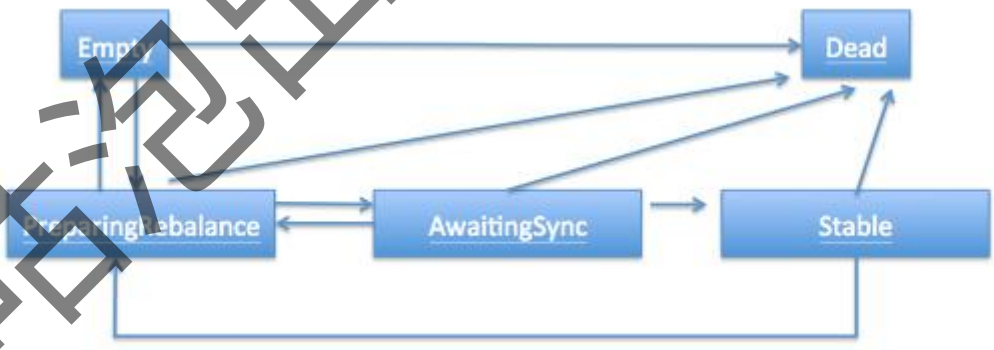


注意：Consumer Group 的分区分配方案是在客户端执行的

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。比如这种机制下我们可以实现类似于 Hadoop 那样的机架感知(rack-aware)分配方案，即为 Consumer 挑选同一个机架下的分区数据，减少网络传输的开销。Kafka 默认提供了两种分配策略：range 和 round-robin。

1.2.2.10 Consumer Group 状态机（不讲）

和很多 Kafka 中间组件一样，Group 也做了个状态机来表明组状态的流转。Coordinator 根据这个状态机会对 Consumer Group 做不同的处理，如下图所示



简单说明下图中的各个状态：

- Dead：组内已经没有任何成员的最终状态，组的元数据也已经被 Coordinator 移除了。这种状态响应各种请求都是一个 Response： UNKNOWN_MEMBER_ID

- Empty：组内无成员，但是位移信息还没有过期。这种状态只能响应 JoinGroup 请求
- PreparingRebalance：组准备开启新的 ReBalance，等待成员加入
- AwaitingSync：正在等待 Leader Consumer 将分配方案传给各个成员
- Stable：ReBalance 完成！可以开始消费了~

至于各个状态之间的流程条件以及 Action，这里就不具体展开了。

2.Kafka 总结

简单地总结一下：

我们已经把 Kafka 生产者发消息，Broker 分区分布，存储消息，清理消息，消费者分配分区理了一下。从生产者到 Broker 到消费者理了一遍，下面来简单地总结一下，为什么 Kafka 能做到这么高的吞吐。

MQ 的消息存储有几种选择，一种是内存，比如 ZeroMQ，速度很快但是不可靠。一种是第三方的数据库，会产生额外的网络消耗，而且数据库出问题会影响存储。所以最常见的是把数据放在磁盘上存储。

但是我们也都知道，磁盘的 I/O 是比较慢的，选择磁盘做为存储怎么实现高吞吐、低延迟、高性能呢？

（案例显示在普通服务器上可以达到百万级 TPS）

https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

总结起来，主要是 4 点：磁盘顺序 I/O、索引机制、批量操作和压紧、零拷贝。

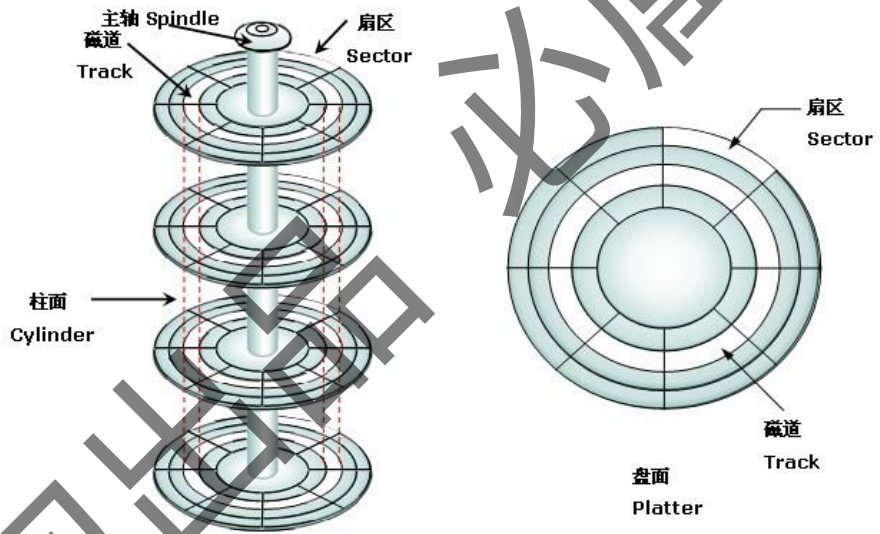
2.1Kafka 为什么这么快？

2.1.1 顺序读写

我们在讲 MySQL 的时候就提到了一种思路，写数据文件 data file，和用来做崩溃恢复的 redo log，区别在于哪里？

随机 I/O 和顺序 I/O。什么叫随机 I/O？

我们先说一下磁盘寻址的过程。这个是磁盘的构造。磁盘的盘片不停地旋转，磁头会在磁盘表面画出一个圆形轨迹，这个就叫磁道。从内到位半径不同有很多磁道。然后又用半径线，把磁道分割成了扇区（两根射线之内的扇区组成扇面）。如果要读写数据，必须找到数据对应的扇区，这个过程就叫寻址。



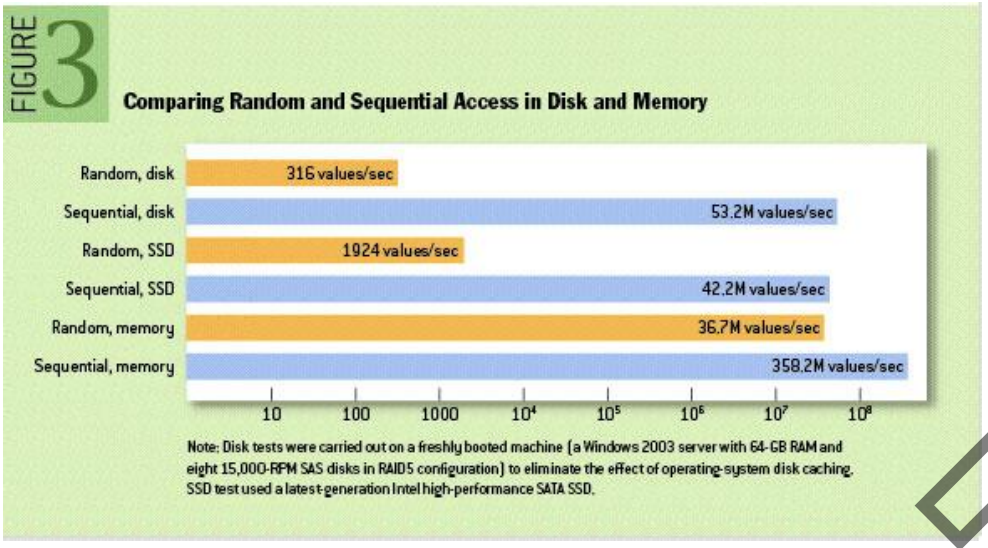
随机 I/O 就是：读写的多条数据在磁盘上是分散的，寻址会很耗时。

顺序 I/O 读写的数据在磁盘上是集中的，不需要重复寻址的过程。

Kafka 的 Message 是不断追加到本地磁盘文件末尾的，而不是随机的写入，这使得 Kafka 写入吞吐量得到了显著提升。

内存 I/O 是不是一定比磁盘 I/O 快呢？

https://queue.acm.org/detail.cfm?id=1563874



这张图片显示，在一定条件下测试，磁盘的顺序读写可以达到 53.2M 每秒，比内存的随机读写还要快。

2.1.2 索引

前面讲过了。

2.1.3 批量读写和文件压缩

它把所有的消息都变成一个批量的文件，并且进行合理的批量压缩，减少网络 IO 损耗。

http://kafka.apache.org/documentation/#recordbatch

2.1.4 零拷贝

首先有两个名词要给大家解释一下。

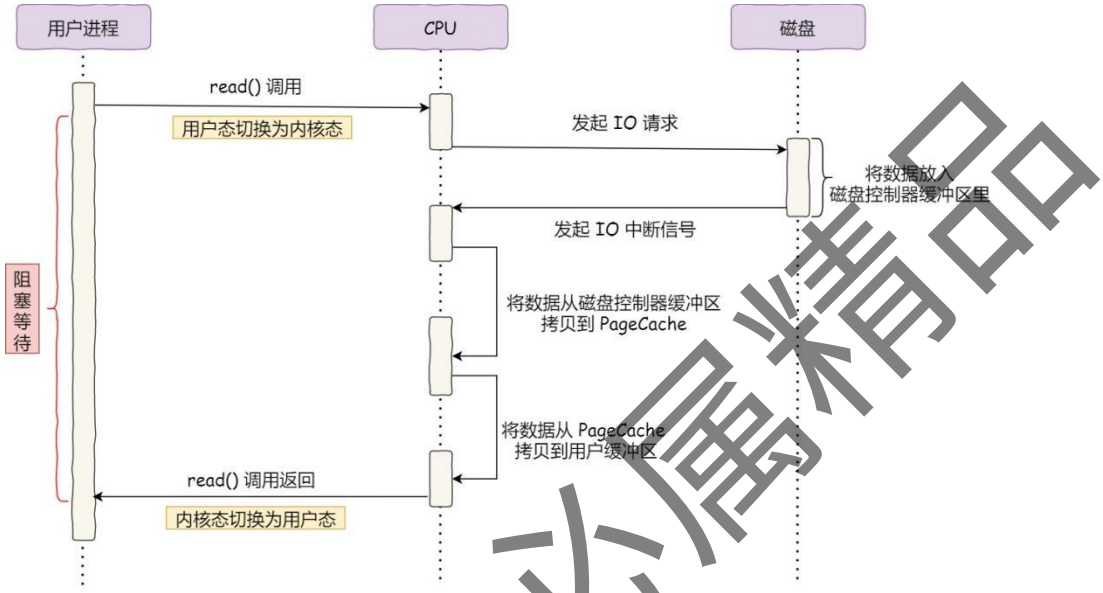
第一个是操作系统虚拟内存的内核空间和用户空间。

操作系统的虚拟内存分成了两块，一部分是内核空间，一部分是用户空间。这样就可以避免用户进程直接操作内核，保证内核安全。

进程在内核空间可以执行任意命令，调用系统的一切资源；在用户空间必须要通过一些系统接口才能向内核发出指令。

女生宿舍不允许男生进入，就算是修电脑也不行！只能让宿管大妈帮你把电脑拿出来。明白的刷 666。

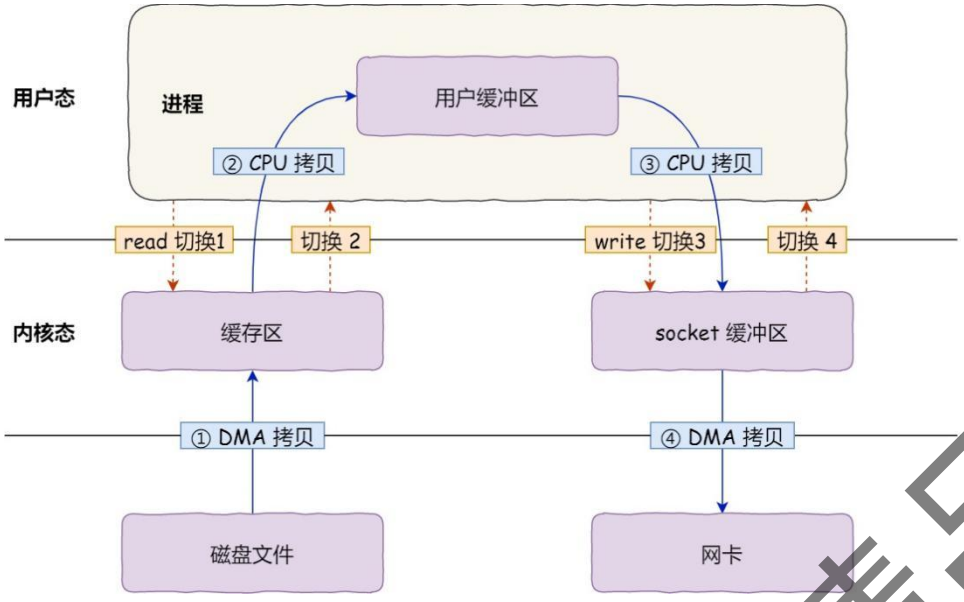
如果用户要从磁盘读取数据（比如 Kafka 消费消息），必须先把数据从磁盘拷贝到内核缓冲区，然后在从内核缓冲区到用户缓冲区，最后才能返回给用户。



第二个是 DMA 拷贝。没有 DMA 技术的时候，拷贝数据的事情需要 CPU 亲自去做，这个时候它没法干其他的事情，如果传输的数据量大那就有问题了。

DMA 技术叫做直接内存访问（Direct Memory Access），其实可以理解为 CPU 给自己找了一个小弟帮它做数据搬运的事情。在进行 I/O 设备和内存的数据传输的时候，数据搬运的工作全部交给 DMA 控制器，解放了 CPU 的双手（反正就是找了个小弟）。

理解了这两个东西之后，我们来看下传统的 I/O 模型：

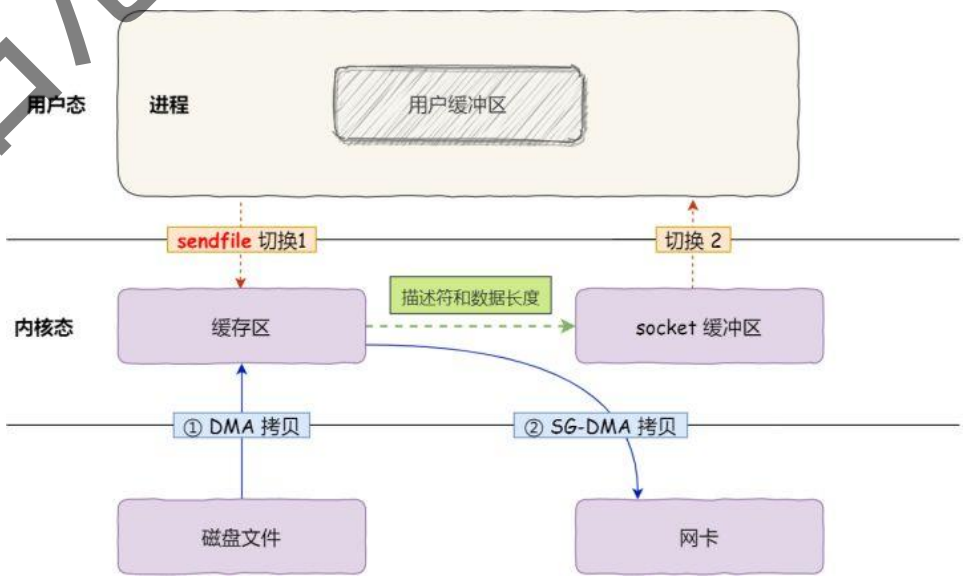


比如 Kafka 要消费消息，比如要先把数据从磁盘拷贝到内核缓冲区，然后拷贝到用户缓冲区，再拷贝到 Socket 缓冲区，再拷贝到网卡设备。这里面发生了 4 次用户态和内核态的切换和 4 次数据拷贝，2 次系统函数的调用（read、write），这个过程是非常耗费时间的。怎么优化呢?

我看着这张图，好像产生了一些联想，狗吃骨头都知道走直线……

在 Linux 操作系统里面提供了一个 sendfile 函数，可以实现“零拷贝”。这个时候就不需要经过用户缓冲区了，直接把数据拷贝到网卡（这里画的是支持 SG-DMA 拷贝的情况）。

因为这个只有 DMA 拷贝，没有 CPU 拷贝，所以叫做“零拷贝”。零拷贝至少可以提高一倍的性能。



可以直接从磁盘文件到网卡吗?

Kafka 文件传输最终调用的是 Java NIO 库里的 transferTo 方法（PlaintextTransportLayer）：

```
@Overridepublic
long transferFrom(FileChannel fileChannel, long position, long count) throws
IOException {
   return fileChannel.transferTo(position, count, socketChannel);
}
```

如果 Linux 系统支持 sendfile() 系统调用，那么 transferTo() 实际上最后就会使用到 sendfile() 系统调用函数。零拷贝技术可以大大地提升文件传输的性能。

2.2 Kafka 消息不丢失的配置

1、Producer 端使用 producer.send(msg, callback)带有回调的 send 方法，而不是 producer.send(msg)方法。根据回调，一旦出现消息提交失败的情况，就可以有针对性地进行处理。

2、设置 acks = all。acks 是 Producer 的一个参数，代表“已提交”消息的定义。如果设置成 all，则表明所有 Broker 都要接收到消息，该消息才算是“已提交”。

3、设置 retries 为一个较大的值。同样是 Producer 的参数。当出现网络抖动时，消息发送可能会失败，此时配置了 retries 的 Producer 能够自动重试发送消息，尽量避免消息丢失。

4、设置 unclean.leader.election.enable = false。这是 Broker 端的参数，在 Kafka 版本迭代中社区也多次反复修改过他的默认值，之前比较具有争议。它控制哪些 Broker 有资格竞选分区的 Leader。如果一个 Broker 落后原先的 Leader 太多，那么它一旦成为新的 Leader，将会导致消息丢失。故一般都要将该参数设置成 false。

5、设置 replication.factor >= 3。需要三个以上的副本。

6、设置 min.insync.replicas > 1。Broker 端参数，控制消息至少要被写入到多

少个副本才算是“已提交”。设置成大于 1 可以提升消息持久性。在生产环境中不要使用默认值 1。确保 replication.factor > min.insync.replicas。如果两者相等，那么只要有一个副本离线，整个分区就无法正常工作了。推荐设置成 replication.factor = min.insync.replicas + 1。

7、确保消息消费完成再提交。Consumer 端有个参数 enable.auto.commit，最好设置成 false，并自己来处理 Offset 的提交更新。