

# 1 认识Apache kafka

认识kafka及其生态圈

消息引擎系统相关知识（定义、特点、设计理念）

kafka概要设计及概念术语

kafka快速入门

## 1.1 kafka快速入门

kafka：是消息引擎系统中的佼佼者，提供了分布式消息引擎解决方案。

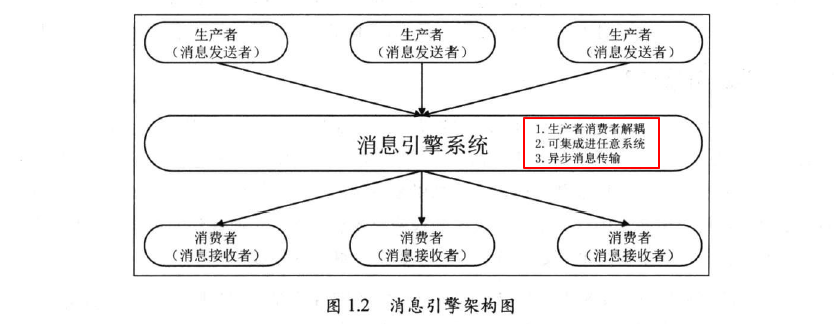
kafka的核心功能：高性能的消息发送与高性能的消息消费。

## 1.2 消息引擎系统（Messaging System）

也有人称消息队列、消息中间件。这里用引擎两字是为了突出消息系统天然具备的传递属性（即具有转换、传输的能力）。

**企业消息引擎系统（EMS）**是企业发布的一组规范。公司使用这组规范实现在不同系统之间传递语义准确的消息。在实际使用场景中，消息引擎系统通常以**软件接口**为主要形式，实现**松耦合**的**异步式**数据传递语义。

|  |
| --- |
| 同步方式的消息发送和异步方式的消息发送:  <https://blog.csdn.net/jisuanji198509/article/details/80331566> |



在设计一个消息引擎系统时需要考虑的两个重要因素：消息设计、传输协议设计。

### 1.2.1 消息设计

消息引擎系统在设计消息时，要考虑到**格式通用、语义清晰，**所以通常采用**结构化**的方式进行设计。比如SOAP协议中的消息就采用了XML格式，而WebService也支持JSON格式的消息。Kafka的消息是用二进制方式来保存的，但依然是结构化的消息。

可以发现，不论是使用XML、JSON、二进制表示，抑或是其他自定义的结构化类型，消息主体本身一般都是结构化的数据，这给后续消息引擎系统的处理带来了极大的便利。

### 1.2.2 传输协议设计

|  |  |
| --- | --- |
| **狭义** | 如AMQP、WebService+SOAP、微软的MSMQ等协议，指定了消息在不同系统之间传输的方式。 |
| **广义** | 这类协议可能包括任何能够在不同系统间传输消息或是执行语义操作的协议或框架。比如主流的rpc及序列化框架，像google protocol buffers、阿里dubbo等。 |

kafka自己设计了一套二进制的消息传输协议，而没有采用诸如Google PB这样的框架。

后面章节我们会详细阐述这其中的原因以及探讨Kafka消息传输协议及其背后的设计理念。

### 1.2.3 消息引擎范型

最常见的两种消息引擎范型是**消息队列模型**和**发布／订阅模型**。后续章节会介绍，Kafka是通过consumer group来同时支持这两种消息引擎模型的。

#### 消息队列模型

消息队列模型是基于队列提供消息传输服务的，多用于进程间通信以及线程间通信。

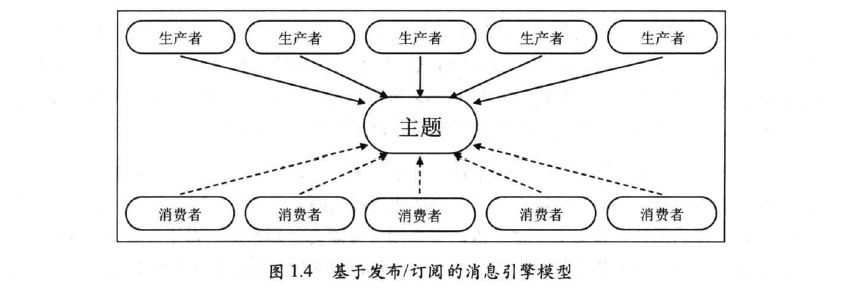
该模型定义了消息队列（queue）、发送者（sender）和接收者（receiver)，提供了一种点对点的消息传递方式，即发送者发送每条消息到队列的指定位置，接收者从指定位置获取消息。一旦消息被消费，就会从队列中移除该消息。每条消息由一个发送者生产出来，且只被一个消费者处理―发送者和消费者之间是一对一的关系。

生活中接线生的工作就是一个典型的基于队列的消息引擎模型。每个打进来的电话都进入一个排队队列，然后只由一个接线生进行处理。同一个客户不会被第二个接线生处理。

#### 发布／订阅模型

发布／订阅模型（publish/subscribe），与前一种模型不同，它有主题（topic）的概念：一个topic可以理解为逻辑语义相近的消息的容器。

这种模型也定义了类似于生产者/消费者这样的角色，即发布者（publisher）和订阅者（subscriber）。发布者将消息生产出来发送到指定的topic中，所有订阅了该topic的订阅者都可以接收到该topic下的所有消息。通常具有相同订阅topic的所有订阅者将接收到同样的消息，如下图所示。



生活中报纸的订阅就是一种典型的发布／订阅模型：很多读者都会订阅同一个报社（类比于同一个topic）出版的报纸，这样每当报纸更新（生产新的消息）时，这些读者都可以收到最新的报纸（接收最新的消息）。

### 1.2.4 JMS(java消息服务)

JMS只是一套API规范，提供了很多接口用于实现分布式系统间的消息传递。JMS同时支持上面两种消息引擎模型。

实际上，当前很多主流的消息引擎系统都完全支持JMS规范，比如ActiveMQ、RabbitMQ（通过RabbitMQ JMS Client）、IBM WebSphere MQ和Kafka等。当然Kafka并没有完全遵照JMS规范，它另辟蹊径，探索出了一条独有的道路。

## 1.3 kafka概要设计

kafka设计初衷就是为了解决互联网公司超大量级数据的实时传输，为了实现这个目标，kafka得成为一个完备的分布式消息引擎解决方案，设计上就得考虑：吞吐量/延时、消息持久化、负载均衡和故障转移、伸缩性。

### 1.3.1 吞吐量/延时

通常来说，**吞吐量**是某种处理能力的最大值。而对于Kafka而言，它的吞吐量就是每秒能够处理的消息数或者每秒能够处理的字节数。

**延时**衡量的是一段时间间隔，可能是发出某个操作与接收到操作响应之间的时间，或者是在系统中导致某些物理变更的起始时刻与变更正式生效时刻之间的间隔。对于Kafka而言，延时可以表示客户端发起请求与服务器处理请求并发送响应给客户端之间的这一段时间。

在实际使用场景中，这两个性能指标通常是一对矛盾体，即调优其中一个指标通常会使另一个指标变差。当然它们之间的关系也不是等比例地此消彼长的关系，比如牺牲20％的延时就能换来20％的吞吐量的提升。

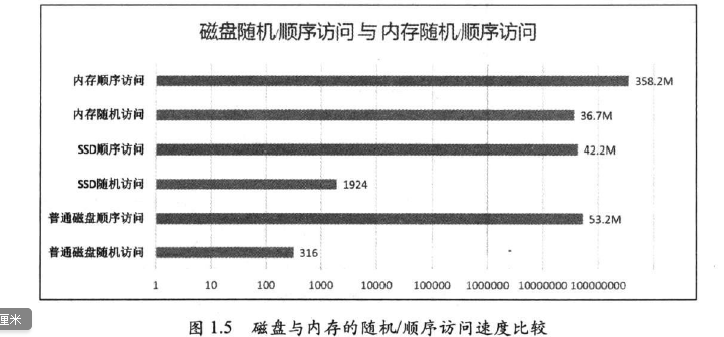
其实关于这一点，很多读者可能会感到困惑：为什么需要同时研究这两个指标呢？如果确定了其中的一个指标，另一个指标也应该是确定的才对啊？好吧，按照这种假设，若Kafka处理一条消息需要花费2毫秒，那么计算得到的吞吐量不会超过500条消息／秒（1000/2=500）。但是若我们采用批处理（batching）的思想，假设在发送前先等待一段时间（假设是8毫秒），那么此时消息发送的延时变成了10毫秒(2+8)，即延时增加了4倍，但假设在这8毫秒中我们总共累积了1000条消息，那么系统整体的吞吐量就变成了100000条／秒（1000/0.01==100000)，吞吐量提升了近200倍！各位读者，看到micro-batch的威力了吧？这也就是目前诸如Spark Streaming等消息处理平台所采用的处理语义思路。

#### 1.3.1.1 kafka 服务端如何做到高吞吐量、低延时

首先，Kafka的写入操作是很快的，这主要得益于它对磁盘的使用方法的不同。虽然Kafka会持久化所有数据到磁盘，但本质上每次写入操作其实都只是把数据**写入到操作系统的页缓存**（page cache）中，然后由操作系统自行决定什么时候把页缓存中的数据写回磁盘上。这样的设计有3个主要优势。

|  |
| --- |
| 1、操作系统页缓存是在内存中分配的，所以消息写入的速度非常快。  2、Kafka不必直接与底层的文件系统打交道，所有烦琐的IO操作都交由操作系统来处理。  3、Kafka写入操作采用**追加写入**（append）的方式，避免了磁盘随机写操作。 |

特别留意上面的第3点，对于普通的物理磁盘（非固态硬盘）而言，随机读/写操作是很慢的，但是顺序读／写操作其实是非常快的（它的速度甚至可以匹敌内存的随机IO速度，如下图所示）。鉴于此，Kafka在设计时采用了追加写入消息的方式，即只能在日志文件末尾追加写入新的消息，且不允许修改己写入的消息，因此它属于典型的磁盘顺序访问型操作，所以Kafka消息发送的吞吐量是很高的，在实际使用过程中可以很轻松地做到每秒写入几万甚至几十万条消息。



#### 1.3.1.2 kafka消费端如何做到高吞吐量、低延时

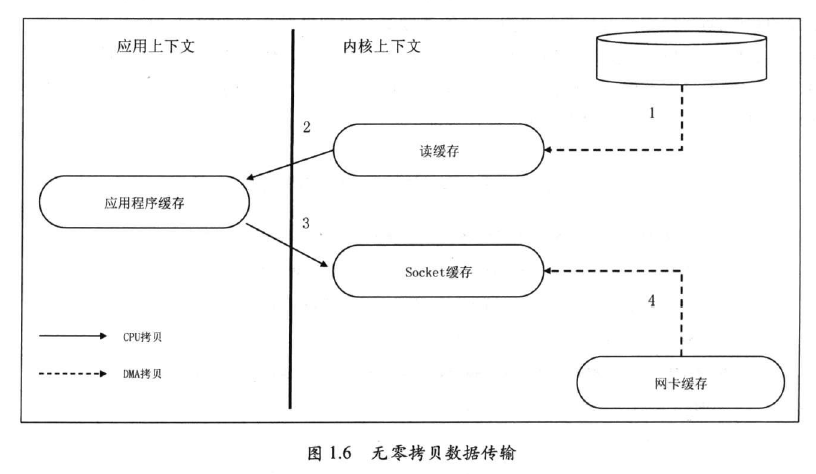
首先，之前提到了Kafka是把消息写入操作系统的页缓存中的，同样地，Kafka在读取消息时会首先尝试从操作系统的页缓存中读取，由于kafka大量使用页缓存，故读取消息时大部分消息很有可能依然保存在页缓存中，因此可以直接命中缓存，不用“穿透”到底层的物理磁盘上获取消息，从而极大地提升了消息读取的吞吐量。事实上，如果我们监控一个经过良好调优的Kafka生产集群便可以发现，即使是那些有负载的Kafka服务器，其磁盘的读操作也很少，这是因为大部分的消息读取操作会直接命中页缓存。

然后，如果成功命中了页缓存，kafka会把消息经页缓存直接发送到网络的Socket上。这个过程就是利用Linux平台的sendfile系统调用做到的，而这种技术就是大名鼎鼎的**零拷贝**（Zero Copy）。

##### 零拷贝

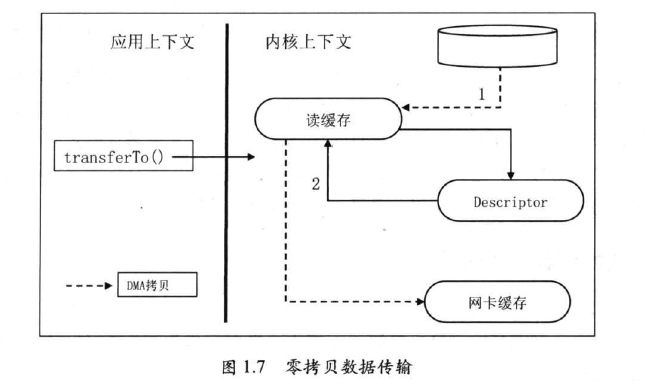
为了方便不太了解sendfile和零拷贝的读者理解，笔者在这里简单介绍一下它的特性。

传统的Linux操作系统中的I/O接口是依托于数据拷贝来实现的，但在零拷贝技术出现之前，一个I/O操作会将同一份数据进行多次拷贝，如下图所示。数据传输过程中还涉及内核态与用户态的上下文切换，CPU的开销非常大，因此极大地限制了OS高效进行数据传输的能力。

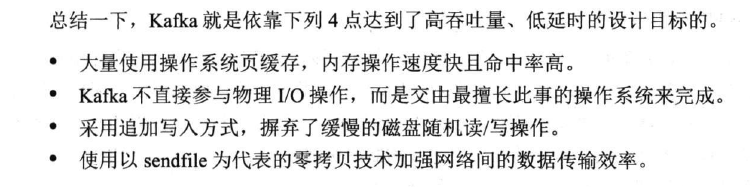


零拷贝技术很好地改善了这个问题：首先在内核驱动程序处理I/O数据的时候，它不再需要进行上下文的切换，节省了内核缓冲区与用户态应用程序缓冲区之间的数据拷贝，同时它利用**直接存储器访问技术（Direct Memory Access,DMA）**执行I/O操作，因此也避免了OS内核缓冲区之间的数据拷贝，故而得名零拷贝，如下图所示。

Linux提供的sendfile系统调用实现了这种零拷贝技术，而Kafka的消息消费机制使用的就是sendfile(严格来说是通过Java的FileChannet.transferTo方法实现的)。



#### 1.3.1.3总结



### 1.3.2 消息持久化 ：

|  |
| --- |
| **kafka将消息持久化到磁盘的好处** |
| 1、**解耦消息发送与消息消费**：本质上来说，Kafka最核心的功能就是提供了生产者一消费者模式的完整解决方案。通过将消息持久化使得生产者方不再需要直接和消费者方耦合，它只是简单地把消息生产出来并交由Kafka服务器保存即可，因此提升了整体的吞吐量。  2、**实现灵活的消息处理**：很多Kafka的下游子系统（接收Kafka消息的系统）都有这样的需求―对于己经处理过的消息可能在未来的某个时间点重新处理一次，即所谓的消息重演（message replay）。消息持久化便可以很方便地实现这样的需求。 |

另外，Kafka实现持久化的设计也有新颖之处。普通的系统在实现持久化时可能会先尽量使用内存，当内存资源耗尽时，再一次性地把数据“刷盘”；而Kafka则反其道而行之，所有数据都会立即被写入文件系统的持久化日志中，之后Kafka服务器才会返回结果给客户端通知它们消息已被成功写入。这样做既实时保存了数据，又减少了Kafka程序对于内存的消耗，从而将节省出的内存留给页缓存使用，更进一步地提升了整体性能。

### 1.3.3 负载均衡和故障转移

#### 1.3.3.1 负载均衡

**负载均衡**就是让系统的负载根据一定的规则均衡地分配在所有参与工作的服务器上，从而最大限度地提升系统整体的运行效率。具体到Kafka来说，默认情况下Kafka的每台服务器都有均等的机会为Kafka的客户提供服务，可以把负载分散到所有集群中的机器上，避免出现“耗尽某台服务器”的情况发生。

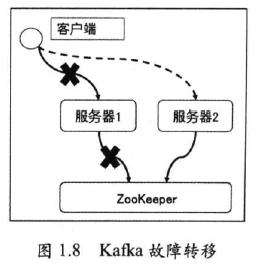
Kafka实现负载均衡实际上是通过智能化的**分区领导者选举**来实现的。Kafka默认提供了很智能的leader选举算法可以在集群的所有机器上以均等机会分散各个partition的leader，从而整体上实现了负载均衡。

#### 1.3.3.2故障转移

**故障转移**是指当服务器意外中止时，整个集群可以快速地检测到该失效（failure)，并立即将该服务器上的应用或服务自动转移到其他服务器上。

故障转移通常是以“心跳”或“会话”的机制来实现的，即只要主服务器与备份服务器之间的心跳无法维持或主服务器注册到服务中心的会话超时过期了，那么就认为主服务器己无法正常运行，集群会自动启动某个备份服务器来替代主服务器的工作。

Kafka务器支持故障转移的方式就是使用会话机制。每台Kafka服务器启动后会以会话的形式把自己注册到zooKeeper服务器上，一旦该服务器运转出现问题，与ZooKeeper的会话便不能维持从而超时失效，此时Kafka集群会选举出另一台服务器来完全代替这台服务器继续提供服务，如下图所示。



### 1.3.4 伸缩性(scalability)

有了消息的持久化，Kafka实现了高可靠性；有了负载均衡和使用文件系统的独特设计，Kafka实现了高吞吐量：有了故障转移，Kafka实现了高可用性。那么作为分布式系统中的高伸缩性，Kafka又是如何做到的呢？

伸缩性表示向分布式系统中增加额外的计算资源（比如CPU、内存、存储或带宽）时吞吐量提升的能力。举一个例子来说，对于计算密集型的业务而言，CPU的消耗一定是最大的，这类系统上的操作我们称之为CPU-bound。那么如果一个CPU的运算能力是U，我们自然希望两个CPU的运算能力是2U，即可以线性地扩容计算能力，这种线性伸缩性是最理想的状态，但在实际中几乎不可能达到，毕竟分布式系统中有很多隐藏的“单点”瓶颈制约了这种线性的计算能力扩容。阻碍线性扩容的一个很常见的因素就是状态的保存。我们知道，不论是哪类分布式系统，集群中的每台服务器一定会维护很多内部状态。如果由服务器自己来保存这些状态信息，则必须要处理一致性的问题。相反，如果服务器是无状态的，状态的保存和管理交于专门的协调服务来做（比如zooKeeper)，那么整个集群的服务器之间就无须繁重的状态共享，这极大地降低了维护复杂度。倘若要扩容集群节点，只需简单地启动新的节点机器进行自动负载均衡就可以了。

Kafka正是采用了这样的思想——每台Kafka服务器上的状态统一交由zooKecper保管。扩展Kafka集群也只需要一步：启动新的Kafka服务器即可。当然这里需要言明的是，在Kafka服务器上并不是所有状态都不保存，它只保存了很轻量级的内部状态，因此在整个集群间维护状态一致性的代价是很低的。

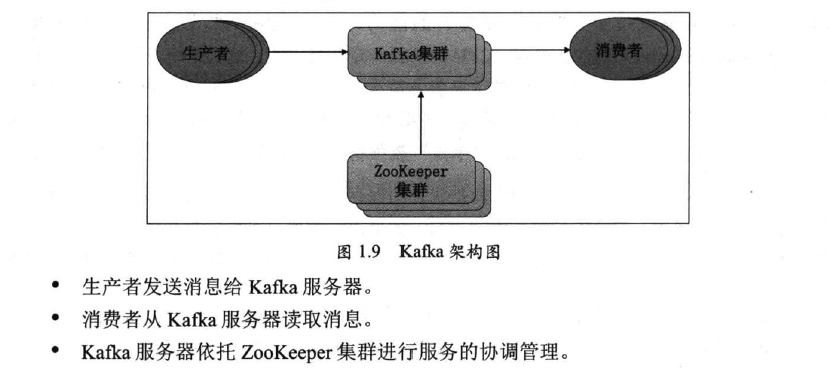
……

## 1.4 kafka基本概念与术语

最开始，kafka是作为消息引擎出现的，其强大的消息传输效率和完备的分布式解决方案，使它很快成为业界翘楚。

后来，kafka开发团队发现，经由kafka交由下游数据处理平台做的事情，kafka自己也能做到，因此推出了kafka streams流式处理组件，这样kafka就成为了一个流式处理框架了。现在kafka的标准定位是分布式流式处理平台。

不管是消息引擎还是流式处理平台，kafka的处理流程没变，如下图。



### 1.4.1 消息

#### 消息格式的设计与保存

……

### 1.4.2 topic（主题）和partition（分区）

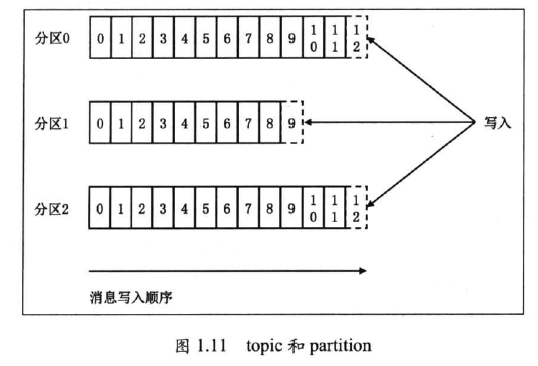
**topic（主题）**只是一个逻辑概念，可理解为消息被发送到的地方，通常用于区分实际的业务，不同的业务使用不同的topic。

Partition（分区）并没有太多的业务含义，它的引入就是单纯地为了提升系统的吞吐量，Kafka中的topic通常都会被多个消费者订阅，因此出于性能的考量，Kafka并不是topic-message的两级结构，而是采用了topic-partition-message的三级结构来分散负载。

从本质上说，每个topic都由若干个partition组成，一个topic的partition数在创建该topic时指定（可参考集群配置来指定合适的值）。

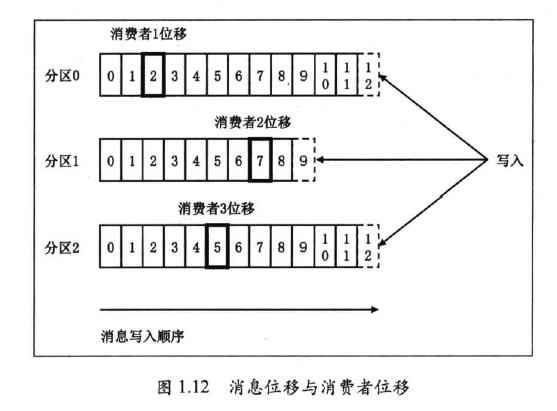
**partition（分区）**是不可修改的有序消息序列，用户对partition唯一能做的操作就是往消息序列的尾部追加写入消息。每个partition都有自己的partition号，partition号从0开始顺序递增的整数。

partition上的每条消息都会被分配一个唯一的序列号，称为**消息的offset（位移），**位移值从0开始顺序递增的整数。通过一个<topic, partition, offset>三元组可以在kafka集群中找到唯一对应的那条消息。



### 1.4.3 消费者位移

每条消息在其所属的partition上的位移是固定的；但是消费某partition的**消费者的位移**会随着消费进度不断前移，但不可能超过该partition中最新一条消息的位移。

****

### 1.4.4|5 replica、leader与follower

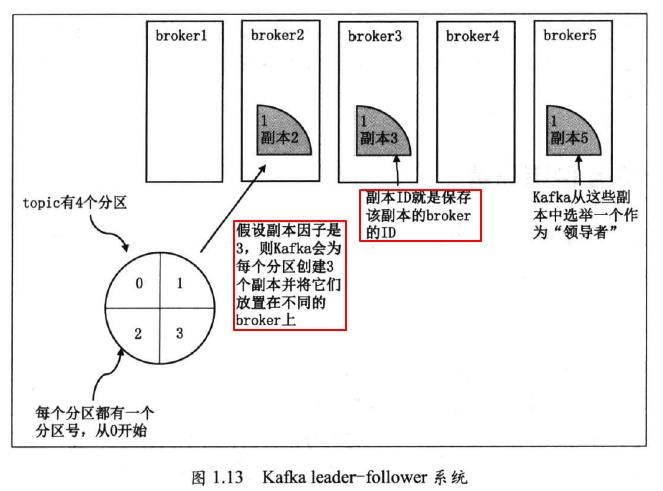
既然partition是有序消息日志，那么一定不能只保存这一份口志，否则一旦保存partition的Kafka服务器挂掉了，其上保存的消息也就都丢失了。

分布式系统必然要实现高可靠性，而目前实现的主要途径还是依靠**冗余机制**——简单地说，就是备份多份日志。这些备份日志在Kafka中被称为**副本（replica）**，它们存在的唯一目的就是防止数据丢失！

和传统的主备系统（Master-Slave，如mysql）不同的是，在**leader-follower系统**中，通常只有**leader（领导者）**对外提供服务，**follower（追随者）**只是被动地追随leader的状态，保持与leader的同步。follower存在的唯一价值就是充当leader的候补，一旦leader挂掉立即就会有一个follower被选举成为新的leader接替它的工作。

Kafka就是这样的设计，如下图所示。kafka中每个partition都由若干个replica组成，其中的一个replica充当leader角色，而剩下的replica充当follower角色。follower replica是不能给客户端提供服务的，也就是说不负责响应客户端发来的消息写入和消息消费请求。follower副本只是被动地向leader副本获取数据，而一旦leader 副本所在的broker宕机，Kafka会从剩余的follower副本中选举出新的leader继续提供服务。

Kafka保证同一个partition的多个replica一定不会分配在同一台broker上。



#### 1.4.6 ISR(in-sync replica)

**所谓ISR**，是指Kafka为partition动态维护的一个replica集合，该集合中的所有replica保存的消息日志都与leader replica保持同步状态。

只有ISR中的replica才能被选举为Ieader，也只有ISR中所有replica都接收到了同一条消息，Kafka才会将该消息置于“己提交”状态，即认为这条消息发送成功。

|  |  |
| --- | --- |
| 问 | 一个partition可以配置N个replica，是否意味着该partition可以容忍（N-1）个replica失效而不丢失数据呢？ |
| 答 | 答案为“否”！  **Kafka承诺只要ISR中至少存在一个“存活的”repica，那些“己提交”状态的消息就不会丢失**——记住这句话的两个关键点：①ISR中至少存在一个“活着的”replica；②“己提交”消息。可能出现情况是：N个replica中有几个replica不在ISR中，突发情况导致只剩下一个不在ISR中的replica存活，ISR中一个存活的replica都没有，这就没办法做到上述保证。  有些Kafka用户经常抱怨：我向Kafka发送消息失败，然后造成数据丢失。其实这是**混淆了Kafka的消息交付语义**：Kafka对于没有提交成功的消息不做任何交付保证，它只保证在ISR存活的情况下“己提交”的消息不会丢失。 |

## 1.5 kafka使用场景

……

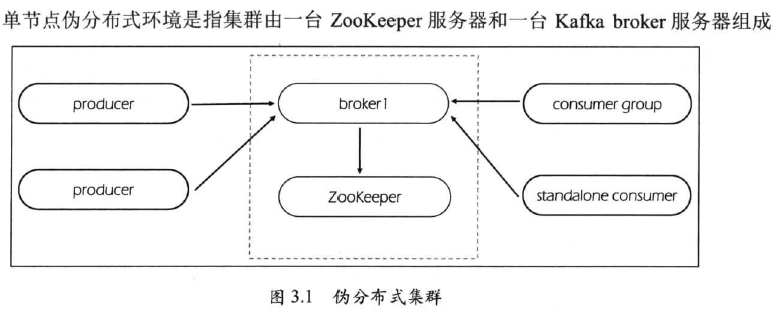
# 2 kafka发展历史

啊

# 3 kafka线上环境部署

## 3.1 集群环境规划

## 3.2 单节点kafka环境安装



当前的/etc/hosts配置如下表所示，主机名为tkafka。

|  |
| --- |
| **/etc/hosts:** |
| 127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4 tkafka |

### 3.2.1 安装java

不论是ZooKeeper还是Kafka都需要提前安装好Java并且正确配置好Java环境。推荐安装jdk8，并且使用比较成熟的oracle公司的Hotspot虚拟机，而非openJDK版本的虚拟机。

Java安装可以使用“java -version”命令来进行验证。默认情况下，该命令输出已安装java的版本，若没有安装java，该命令会提示“无法找到Java虚拟机”。

首先，使用wget命令下载rpm格式的jdk安装文件。

然后，使用命令“sudo rpm -ivh jdk-8u144-linux-x64.rpm”进行安装。（默认的安装路径是“/usr/java/jdk1.8.0\_144”）。

之后，找到相应配置文件（如“/etc/proifle”），设置如下环境变量，然后source该文件。

|  |
| --- |
| export JAVA\_HOME＝/usr/java/jdk1.8.0\_144  export JRE\_HOME=$JAVA\_HOME/jre  export PATH=$PATH:$JRE\_HOME/bin:$JAVA\_HQME/bin |

### 3.2.2 安装单节点zookeeper

zookeeper是安装Kafka集群必要的组件， 用于为kafka提供协调服务，Kafka大量地使用ZooKeeper来保存集群的元数据信息以及老版consumer的位移信息。

#### 使用kafka内置的zookeeper及相关的管理脚本

|  |
| --- |
| cd /usr/lib/lws/kafka\_2.11-1.0.0 |
| **启动zookeeper**  bin/zookeeper-server-start.sh **-daemon** config/zookeeper.properties  （或nohup …… &） |
| **验证zk启动状况：**  jps >> ps –ef|grep zookeeper >> netstat -tunlp|grep 2181 |
| **关闭zk：**  kill -9 jps结果 |

#### 单独安装外部的zookeeper

|  |
| --- |
| **zookeeper安装过程如下：**  1、用wget命令下载zookeeper二进制代码压缩包（如zookeeper-3.4.13.tar.gz）。  2、用“tar -xzvf /download/zookeeper-3.4.13.tar.gz -C /usr/lib/lws/”进行解压。  3、用“cd /usr/lib/lws/zookeeper-3.4.13/”进入解压后目录。  4、用“mv conf/zoo\_sample.cfg conf/zoo.cfg”将配置文件重命名为zoo.cfg。  配置文件内容不用做修改。  5、用“bin/zkServer.sh start”启动zookeeper。 |
| 安装完成后，可以使用**telnet**连入2181端口所在的zookeeper服务，并执行“**srvr**”命令获取zookeeper服务器的状态信息，如下图所示。  安装telnet见“<https://www.cnblogs.com/rusking/p/5413209.html>”。 |

### 3.2.3 安装单节点kafka

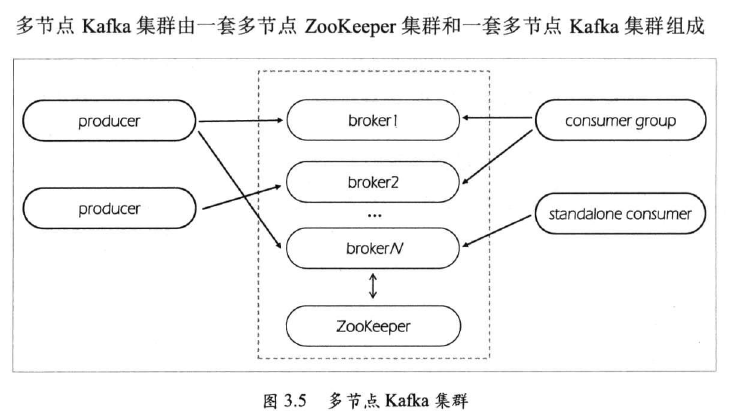
|  |
| --- |
| **kafka安装过程如下：**  1、用wget命令下载kafka的二进制压缩包（如kafka\_2.11-1.0.0.tgz，其中2.11指的是编译kafka的Scala语言版本，1.0.0才是kafka版本）。  2、用“tar -xzvf /download/kafka\_2.11-1.0.0.tgz -C /usr/lib/lws/”进行解压。  3、用“cd /usr/lib/lws/kafka\_2.11-1.0.0”进入解压后目录。  配置文件config/server.properties内容不用做修改。  4、用“bin/kafka-server-start.sh -daemon config/server.properties”启动kafka。（如果启动时报找不到主机名的UnknownHostException，就把主机名配置进/etc/hosts文件中并source一下该文件,再ping主机名能ping通就没问题了） |
| **验证kafka启动状况：**  jps >> ps –ef|grep kafka >> netstat -tunlp|grep 9092 |
| **关闭kafka：**  kill -9 jps结果 |

## 3.3 多节点kafka分布式环境安装

单节点环境应付日常的应用开发及功能验证绰绰有余。但在生产环境中我们应充分利用Kafka提供的分布式特性，比如负载均衡和故障转移等。

此外，Kafka还具有优秀的准线性扩容的能力，因此用户可以很容易地扩展Kafka节点数量以应对不断增长的消息处理需求。

同时由于Kafka提供了完备的备份机制，多节点集群天然地为用户提供高可用保障，极大地降低了人工维护的成本。



这里演示在**一台机器上**安装zookeeper和kafka的集群环境，先如下表配置/etc/hosts。

|  |
| --- |
| **/etc/hosts:** |
| 127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4 tkafka zk1 zk2 zk3 kafka1 kafka2 kafka3 |

### 3.3.1 配置多节点zookeeper集群

zookeeper集群通常被称为一个ensemble，只要这个ensemble中的**大多数**节点存活，那么zookecper集群就能正常提供服务。显然，既然是大多数，那么最好使用奇数个服务器，即(2n+1)个服务器，这样整个zookeeper集群最多可以容忍n台服务器宕机而保证依然提供服务。如果使用偶数个服务器则通常会浪费一台服务器的资源。

举例：假设我们使用5台zookeeper服务器构建集群，倘若2台服务器宕机，剩下的3台服务器占了半数以上，故而zookeeper服务正常工作；但假如我们使用了4台服务器，若2台服务器宕机，剩下的2台服务器不满足“半数以上服务器存活”的条件，因此此时zookeeper集群将停止服务——由此可见，使用4台或5台服务器，都只能容忍1台服务器崩溃，也就是说使用偶数个服务器通常会浪费一台服务器的资源。

基于上面的规则，生产环境一个zookeeper集群中最少节点数量是3，这样1个节点“挂掉”了不会影响整个集群的运作。在实际使用场景中，5台服务器构成的zookeeper集群也是十分常见的，而再多数量的集群则不常见。当然具体数量的确定要依据用户对高可靠性的需求，通常我们都需要在高可靠性与成本间取得适当的平衡。

#### zoo.cfg说明

|  |  |
| --- | --- |
| tickTime=2000 | zookeeper最小的时间单位，用于丈量心跳时间和超时时间等。通常设置成默认值2秒即可。 |
| initLimit=10 | 指定follower节点初始时连接leader节点的最大tick次数。假设是5，表示follower必须要在5\*tickTime时间内（默认是10秒）连接上leader，否则将被视为超时。 |
| syncLimit=5 | 设定了follower节点与leader节点进行同步的最大时间。与initLimit类似，它也是以tickTime为单位进行指定的。 |
| **dataDir**=/tmp/zks/zk1 | zookeeper会在内存中保存系统快照，并定期写入该路径指定的文件夹中。生产环境中需要注意该文件夹的磁盘占用情况。 |
| **clientPort**=2181 | zookeeper监听客户端连接的端口，一般设置成默认值2181。 |
| **server.1**= zk1:2888:3888  **server.2**= zk2:2888:3888  **server.3**= zk3:2888:3888 | **配置文件中的最后几行都是这种形式的**。这里的server.**x**的x必须是一个全局唯一的数字，且需要与myid文件中的数字相对应（关于myid文件的设置稍后会做详细讨论）。一般设置x值为1~255之间的整数。  这行的后面还配置了两个端口，通常是2888和3888。**第一个端口**用于使follower节点连接leader节点，而**第二个端口**则用于leader选举。 |

#### zookeeper服务器唯一ID

每个zookeeper服务器都有一个唯一的ID，这个ID主要用在两个地方：一个是在zoo.cfg配置文件的server.**x**处，另一个则是myid文件。

**myid文件**位于zoo.cfg中dataDir配置的目录下，其内容仅是一个数字，即ID。

#### 配置样例

首先用“cd /usr/lib/lws/zookeeper-3.4.13/”进入zookeeper安装目录。

然后，使用“cp conf/zoo.cfg conf/zooX.cfg”将zoo.cfg复制出3份，分别为zoo1.cfg、zoo2.cfg、zoo3.cfg，如下表修改这3个配置文件（主要修改**dataDir、clientPort、server.x**）。

|  |  |  |
| --- | --- | --- |
| **zoo1.cfg** | **zoo2.cfg** | **zoo3.cfg** |
| clientPort=2182  dataDir=/tmp/zks/zk1 | clientPort=2183  dataDir=/tmp/zks/zk2 | clientPort=2184  dataDir=/tmp/zks/zk3 |
| server.1=zk1:2888:3888  server.2=zk2:2889:3889  server.3=zk3:2890:3890 | | |

然后，创建相应的dataDir文件夹，以及myid文件。

|  |  |  |
| --- | --- | --- |
| **/tmp/zks/zk1/myid** | **/tmp/zks/zk2/myid** | **/tmp/zks/zk3/myid** |
| 1 | 2 | 3 |

最后，启动zookeeper集群，并查看启动状态。

|  |
| --- |
| cd /usr/lib/lws/zookeeper-3.4.13/  bin/zkServer.sh start conf/zoo1.cfg  bin/zkServer.sh start conf/zoo2.cfg  bin/zkServer.sh start conf/zoo3.cfg |
| bin/zkServer.sh status conf/zoo1.cfg  bin/zkServer.sh status conf/zoo2.cfg  bin/zkServer.sh status conf/zoo3.cfg  另外也可以用jdk的“jps”命令来查看是否有zookeeper进程被启动，zookeeper的主进程名为QuorumPeerMain。  最好确保在其它机器上用telnet能成功连上该zookeeper集群（如果连不上可能是虚拟机防火墙未关闭）。如“telnet hostname或ip 2183”。 |

### 3.3.2 配置多节点kafka集群

首先用“cd /usr/lib/lws/kafka\_2.11-1.0.0/”进入kafka安装目录。

然后，使用“cp config/server.properties conf/serverX.properties”将server.properties复制出3份，分别为server1.properties、server2.properties、server3.properties，如下表修改这3个配置文件（主要修改**broker.id、listeners、log.dirs、zookeeper.connect**）。

|  |  |  |
| --- | --- | --- |
| **server1.properties** | **server2.properties** | **server3.properties** |
| broker.id=1  listeners=PLAINTEXT:// 10.6.215.45:9093  log.dirs=/tmp/kafkas/kafka1 | broker.id=2  listeners=PLAINTEXT:// 10.6.215.45:9094  log.dirs=/tmp/kafkas/kafka2 | broker.id=3  listeners=PLAINTEXT:// 10.6.215.45:9095  log.dirs=/tmp/kafkas/kafka3 |
| zookeeper.connect=zk1:2182,zk2:2183,zk3:2184 | | |
| 1、每台Kafka服务器需要指定不同的**broker.id**，该ID在整个集群中必须是唯一的。  2、配置**listeners**时最好使用节点的全称域名(FQDN:Fully Qualified Domain Name)，尽量不要使用IP地址。 搭的简单测试环境，最后发现这里只能如PLAINTEXT:// 10.6.215.45:9093这样配置，不能配置成默认的PLAINTEXT://:9093，也不能配置成PLAINTEXT://127.0.0.1:9093。具体见另一份文档“kafka集群中ip配置问题记录.docx”。  3、**zookeeper.connect**必须同时指定所有的zookeeper节点。 | | |

然后，建好与配置文件中log.dirs属性对应的文件夹。

然后，启动kafka集群，并查看启动状态。

|  |
| --- |
| cd /usr/lib/lws/kafka\_2.11-1.0.0/  bin/kafka-server-start.sh -daemon config/server1.properties  bin/kafka-server-start.sh -daemon config/server2.properties  bin/kafka-server-start.sh -daemon config/server3.properties |
| 可用“tail –f logs/server.log”查看**kafka安装目录下的logs/server.log**确认启动过程无误。  另外，也可以用“jps”或“jps |grep Kafka”命令查看kafka进程（进程名为Kafka）启动情况。  最好确保在其它机器上用telnet能成功连上该kafka集群（如果连不上可能是虚拟机防火墙未关闭）。telnet作为客户端连接kafka集群也得受listeners配置限制，如果listeners处配置的是ip,这里telnet就只能用ip来连，如“telnet 10.6.215.45 9094”。 |

## 3.4 验证部署

### 3.4.1 测试topic相关操作

topic能够正确地创建与删除才能说明Kafka集群在正常工作，因为通常它都表明了Kafka的控制器（controller）己经被成功地选举出来并开始履行自身的职责。

建议在开始使用Kafka集群前最好提前把所需的topic创建出来，并执行对应的命令做验证，不能依赖于kafka的topic自动创建功能，不要在没验证集群的情况下直接执行producer和consumer。这样既可以测试整个集群的运行状况，也保证了producer和consumer运行时不会因为topic分区leader的各种问题导致短暂停顿现象。

创建一个topic用于消息的发送与接收。

|  |
| --- |
| **1、查看主题列表** |
| cd /usr/lib/lws/kafka\_2.11-1.0.0/  bin/**kafka-topics.sh**  **--zookeeper** zk1:2182,zk2:2183,zk3:2184  **-list** |
| **2、查看指定主题状态** |
| cd /usr/lib/lws/kafka\_2.11-1.0.0/  bin/**kafka-topics.sh**  **--zookeeper** zk1:2182,zk2:2183,zk3:2184  **--describe**  **--topic** topicname |
| **3、创建主题** |
| cd /usr/lib/lws/kafka\_2.11-1.0.0/  bin/**kafka-topics.sh**  **--zookeeper** zk1:2182,zk2:2183,zk3:2184  **--create**  **--topic** topicname  **--partitions** 3  **--replication-factor** 3 |
| 这里创建一个名称为“topicname”的***主题（topic）***，该topic有3个***分区（partition）***，每个partition有3个***副本（replica）***。 |
| **4、删除主题** |
| cd /usr/lib/lws/kafka\_2.11-1.0.0/  bin/**kafka-topics.sh**  **--zookeeper** zk1:2182,zk2:2183,zk3:2184  **--delete**  **--topic** topic名称 |
| 此命令只会将相关topic标记为待删除，至于topic是否会被真正删除取决于broker端参数“delete.topic.enable”（默认为true）。删除操作是一个异步过程，具体的删除操作对命令执行者来说是完全透明的！  用户可以通过在删除主题后查看主题列表来验证是否删除主题成功。有时会发现topic没有被成功删除，这可能是topic分区数过多或数据过多的原因。Kafka当前只能一个分区一个分区地删除数据，无法做到同时删除，因此用户可以**查询底层的文件系统**来判断删除操作是否在正常执行。如果发现删除操作停滞了，这通常表明该Kafka集群有问题，此时便需要进一步地详查才能确定无法删除的真正原因，比如是否有分区正处于被分配过程中等。 |

### 3.4.2 测试消息发送与消费

Kafka提供了一个脚本工具（**kafka-console-producer.sh**）来支持通过标准输入向Kafka的指定topic发送消息。用户在控制台终端下启动该脚本工具后，每当用户键入文本并按下回车键后，该行文本即会被封装成一条kafka消息发送给指定的topic。若一段时间内（默认是30秒）没有任何错误出现，则通常表明消息被成功发送了。按下Ctrl+C组合键则退出这个控制台。

kafka还提供了一个脚本工具（**kafka-console-consumer.sh**）用于消费指定topic下的消息并打印到标准输出。

注意，kafka-console-producer.sh和kafka-console-consumer.sh还有其他很多有用的参数选项，如果不加任何参数直接运行它们，则会打印它们各自的使用摘助文档。另外，如果使用的是Windows系统，那么改用bin/windows/\*.bat脚本即可。

|  |
| --- |
| **测试发送消息** |
| cd /usr/lib/lws/kafka\_2.11-1.0.0  bin/**kafka-console-producer.sh**  **--broker-list** 10.6.215.45:9093,10.6.215.45:9094,10.6.215.45:9095  **--topic** topicname |
| **测试新版consumer消费消息** |
| cd /usr/lib/lws/kafka\_2.11-1.0.0  bin/**kafka-console-consumer.sh**  **--bootstrap-server** 10.6.215.45:9093,10.6.215.45:9094,10.6.215.45:9095  **--topic** topicname  **--from-beginning** |
| **~~测试旧版consumer消费消息~~** |
| cd /usr/lib/lws/kafka\_2.11-1.0.0  ~~bin/~~**~~kafka-console-consumer.sh~~**  **~~--zookeeper~~** ~~zk1:2182,zk2:2183,zk3:2184~~  **~~--topic~~** ~~topicname~~  **~~--from-beginning~~** |

### 3.4.3 生产者吞吐量测试

### 3.4.3 消费者吞吐量测试

## 3.5 参数设置

### 3.5.1 broker端参数

### 3.5.2 topic级别参数

### 3.5.3 GC参数

### 3.5.4 JVM参数

### 3.5.5 OS参数

# 4 producer开发

## 4.1 producer概览

producer就是负责向kafka写入数据的应用程序，kafka支持多种语言的producer，这里我们只关注java版本的producer。

Kafka producer在设计上要比consumer简单一些，因为它不涉及复杂的组管理操作，即每个Producer都是独立进行工作的，与其他producer实例之间没有关联，因此它受到的牵绊自然也要少得多，实现起来也要简单得多。

### 4.1.1 Kafka封装了一套二进制通信协议

Kafka封装了**一套二进制通信协议**，用于对外提供各种各样的服务。producer底层都是由它们实现的，对于Producer而言，用户几乎可以直接使用任意编程语言按照该协议的格式进行编程，从而实现向Kafka发送消息。实际上内置的Java版本producer和其它语言的第三方producer库在底层都是相同的实现原理，只是在易用性和性能方面有所差别而己。

这组协议本质上为不同的协议类型分别定义了专属的紧凑二进制字节数组格式，然后通过Socket发送给合适的broker，之后等待broker处理完成后返还响应给producer。这样设计的好处在于具有良好的统一性——即所有的协议类型都是统一格式的，并且由于是自定义的二进制格式，这套协议并不依赖任何外部序列化框架，从而显得非常轻量级，而且也有很好的扩展性。第6章将详细讨论这套二进制通信协议的设计与使用。

### 4.1.2确认目标分区

producer的首要功能是向某个topic的某个分区发送一条消息，所以它首先需要确认到底要向topic的哪个分区写入消息——这就是**分区器（Partitioner）**要做的事情。

Kafka producer提供了一个**默认的分区器**，对于每条待发送的消息而言，如果该消息指定了key，那么该Partitioner会先计算key的哈希值，然后对总分区数进行求模来选择消息要被发送到的目标分区号；若该消息没有指定key，则partitioner会使用轮询的方式确认目标分区——这样可以最大限度地确保消息在所有分区上的均匀性。

producer也允许用户实现**自定义的分区策略**而非使用默认的partitioner，这样用户可以很灵活地根据自身的业务需求确定不同的分区策略。

另外，producer的API还赋予了用户自行指定目标分区的权力，即用户可以在消息发送时跳过partitioner直接指定要发送到的分区。

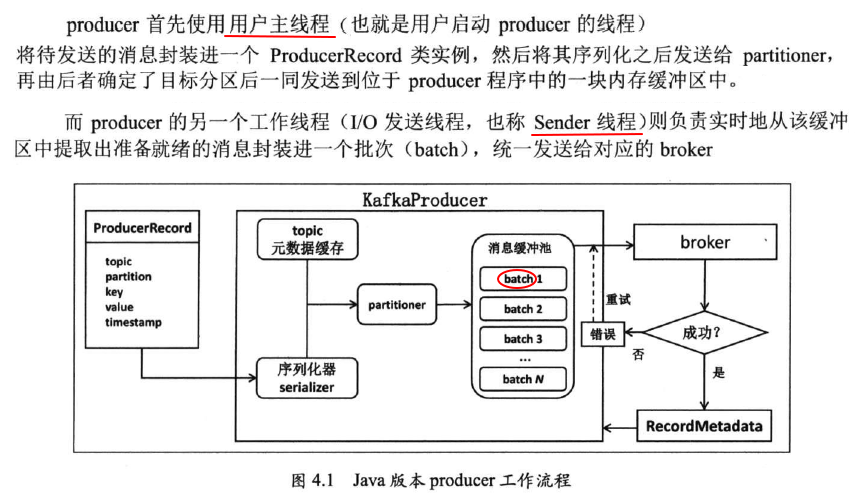
有了默认partitioner的帮助，我们就可以确信具有相同key的所有消息都会被路由到相同的分区中，这有助于实现一些特定的业务需求，比如可以利用局部性原理，将某些producer发送的消息固定地发送到相同机架上的分区从而减少网络传输的开销等。当然了，如前所述，如果没有指定key，那么所有消息会被均匀地发送到所有分区，而这通常也是最合理的分区策略。

### 4.1.3 找目标分区的leader

在确认了目标分区后，producer要做的是要寻找这个分区对应的leader，也就是该分区leader副本所在的Kafka broker。

前面章节中提到了partition的所有副本中，只有leader副本才能够响应clients发送过来的请求，剩下的副本中有一部分副本会与leader副本保持同步，即所谓的ISR。因此**在发送消息时，producer也就有了多种选择来实现消息发送**。比如不等待任何副本的响应便返回成功，或者只是等待leader副本响应写入操作之后再返回成功等。不同的选择也有着不同的优缺点，我们会在后续章节中讨论如何选择不同的策略。

### 4.1.4 java版本producer原理



## 4.2 构造producer

### 4.2.1 producer程序示例

*示例代码：org.liws.kafka.clients.producer. T1\_Producer*

#### 4.2.1.1 ProducerRecord<K, V>

一个ProducerRecord封装了一条待发送的消息，字段如下：

|  |  |
| --- | --- |
| String topic | 消息要发送至的topic |
| Integer partition | **用户可以在创建消息对象的时候直接指定要发送的分区**，这样producer后续发送该消息时可以直接发送到指定分区，而不用先通过Partitioner计算目标分区。 |
| K key |  |
| V value |  |
| ~~Long timestamp~~ | **我们可以直接指定消息的时间戮。**  要注意，一定要慎重使用这个功能，因为在目前的Kafka设计中，时间戳索引文件中的索引项都是严格按照时间戳顺序排列的，所以如果在producer端随意指定时间戳，会导致该消息的时间序列混乱，这样在使用根据时间戳查询位移的功能时则不会找到这条消息。同时Kafka的消息留存策略也会受到影响，因此最好还是让Kafka自行来指定时间戳比较稳妥。  它有**可能会令时间戳索引机制失效**（笔者曾经直接指定时间戳故意打乱发送顺序进行测试，比如先发送消息的时间戳大于后发送消息的时间戳，最后发现通过时间戳定位消息时会发生混乱。为此笔者还特意开了一个bug报告，不过被认为是“当前不被支持的用法”）。 |
| Headers headers |  |

#### 4.2.1.2 异常

不管是同步发送还是异步发送，都有可能失败，导致返回异常错误．当前Kafka的错误类型包含了两类：**可重试异常**和**不可重试异常**。

##### 可重试异常

|  |  |
| --- | --- |
| LeaderNotAvailableException | 分区的leader副本不可用，这通常出现在leader换届选举期间，因此通常是瞬时的异常，重试之后可以自行恢复。 |
| NotControllerException | controller当前不可用（controller是Kafka集群中非常重要的角色，我们会在第6章中详细讨论controller的设计与实现）。这通常表明controller在经历新一轮的选举，这也是可以通过重试机制自行恢复的。 |
| NetworkException | 网络瞬时故障导致的异常，可重试。 |

所有可重试异常都继承自org.apache.kafka.common.errors.RetriabIeException抽象类，对于可重试的异常，如果在producer程序中配置了重试次数，那么只要在规定的重试次数内自行恢复了，便不会出现在onCompletion的exception中。不过若超过了重试次数仍没有成功，则仍然会被封装进nCompletion的exception中，此时就需要Producer程序自行处理这种异常了。

##### 不可重试异常

理论上所有未继承自RetriabIeException类的其他异常都属于不可重试异常，这类异常通常都表明了一些非常严重或Kafka无法处理的问题，与producer相关的如下。

|  |  |
| --- | --- |
| RecordTooLargeException | 发送的消息尺寸过大，超过了规定的大小上限。显然这种异常无论如何重试都是无法成功的。 |
| SerializationException | 序列化失败异常，这也是无法恢复的。 |
| KafkaException | 其他类型的异常。 |

### 4.2.2 producer主要参数

|  |  |
| --- | --- |
| **bootstrap.servers** | 指定一组host:port对，用于创建向Kafka服务器的连接，比如”ip1:9092,ip2:9092,ip3:9092”。  如果Kafka集群中机器数很多，那么只需要指定部分broker即可，不需要列出所有的机器。因为不管指定几台机器，producer都会通过该参数找到并发现集群中所有的broker。为该参数指定多台机器只是为了故障转移使用，这样即使某一台broker挂掉了，producer重启后依然可以通过该参数指定的其他broker连入Kafka集群。  另外，如果broker端没有显式配置listeners使用IP地址， 那么最好将该参数也**配置成主机名，而不是IP**。因为Kafka内部使用的就是FQDN(Fully Qualified Domain Name)。 |
| **key.serializer** | 被发送到broker端的任何消息的格式都必须是字节数组，因此消息的各个组件必须首先做序列化，然后才能发送到broker。  该参数就是用于为消息的key做序列化的。这个参数指定的是实现了org.apache.kafka.common.serialization.Serializer接口的类的全限定名称。Kafka为大部分的初始类型（primitive type）默认提供了现成的序列化器。用户可以通过实现Serializer接口来自定义序列化器。  需要注意的是，即使producer程序在发送消息时不指定key，这个参数也是必须要设置的，否则程序会抛出ConfigException异常，提示“key.serializer”参数无默认值，必须要配置。 |
| **value.serializer** | 该参数用于为消息体做序列化，将消息的value部分转换为字节数组。 |
|  | |
| **acks** |  |
| **buffer.memory** |  |
| **retries** |  |
| **compression.type** | 此参数设置producer端是否压缩消息，默认值是none，即不压缩消息。Kafka的producer端引入压缩后可以显著地降低网络I/O传输开销从而提升整体吞吐量，但也会增加producer端机器的CPU开销。参考4.7  另外，如果broker端的压缩参数设置得与producer不同，broker端在写入消息时也会额外使用CPU资源对消息进行对应的“解压缩-重压缩”操作。 |
| **batch.size** | batch.size是producer最重要的参数之一！它对于调优producer吞吐量和延时性能指标都有着非常重要的作用。  前面提到过，producer会将发往同一分区的多条消息封装进一个batch中，当batch满了的时候，producer会发送batch中的所有消息。不过，producer并不总是等待batch满了才发送消息，很有可能当batch还有很多空闲空间时producer就发送该batch（参考linger.ms说明）。  显然，batch的大小就显得非常重要。通常来说，一个小的batch中包含的消息数很少，因而一次发送请求能够写入的消息数也很少，所以producer的吞吐量会很低；但若一个batch非常之巨大，那么会给内存使用带来极大的压力，因为不管是否能够填满，producer都会为该batch分配固定大小的内存。因此batch.size参数的设置其实是一种时间与空间权衡的体现。  batch.size参数默认值是16384，即16KB。这其实是一个非常保守的数字，在实际使用过程中合理地增加该参数值，通常都会发现producer的吞吐量得到了相应的增加。 |
| **linger.ms** | 上面说到batch.size时，我们提到了消息没有填满batch也可以被发送的情况。这是为什么呢？难道不是等batch满了再发送比较好吗？实际上这也是一种权衡，即吞吐量与延时之间的权衡。  linger.ms参数就是控制消息发送延时行为的。该参数默认值是0，表示消息需要被立即发送，无须关心batch是否己被填满，大多数情况下这是合理的，毕竟我们总是希望消息被尽可能快地发送。不过这样做会拉低producer吞吐量，毕竟producer发送的每次请求中包含的消息数越多，producer就越能将发送请求的开销摊薄到更多的消息上，从而提升吞吐量。 |
| **max.request.size** | 该参数控制的是producer端能够发送的最大消息大小。  由于请求有一些头部数据结构，因此包含一条消息的请求的大小要比消息本身大。不过姑且把它当作请求的最大尺寸是安全的。  如果Producer要发送尺寸很大的消息，那么这个参数就是要被设置的，默认的1048576字节太小了，通常无法满足企业级消息的大小要求。 |
| **request.timeout.ms** | 当Producer发送请求给broker后，broker需要在规定的时间范围内将处理结果返还给Producer，这段时间便是由该参数控制的，默认是30秒。这就是说，如果broker在30秒内都没有给Preducer发送响应，那么preducer就会认为该请求超时了，并在回调函数中显式地抛出TimeoutException异常交由用户处理。  默认的30秒对于一般的情况而言是足够的，但如果Producer发送的负载很大，超时的情况就很容易碰到，此时就应该适当调整该参数值。 |

## 4.3 消息分区机制

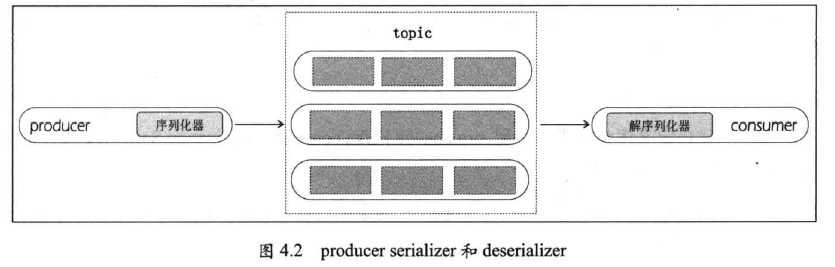
*示例代码：org.liws.kafka.clients.producer. T2\_CustomPartitioner*

|  |
| --- |
| **kafka提供的GetOffsetShell工具类可帮助我们查询指定topic每个分区的消息数** |
| cd /usr/lib/lws/kafka\_2.11-1.0.0/  bin/**kafka-run-class.sh** **kafka.tools.GetOffsetShell** -–broker-list localhost:9092 –-topic test-topic |

## 4.4 消息序列化

在网络中发送数据都是以字节的方式，Kafka也不例外。

序列化器（serializer）与解序列化器（deserializer）执行的是互逆的操作，序列化器负责在Producer发送前将消息转换成字节数组；而与之相反，解序列化器则用于将Consumer接收到的字节数组转换成相应的对象。



kafka支持用户给broker发送各种类型的消息，它可以是一个字符串、一个整数、一个数组或是其他任意的对象类型。Kafka1.0.0版本默认提供了十几种的serializer来支持常用类型的序列化，常见的serializer如下表所示，但若涉及复杂的类型（比如Avro或其他序列化框架），那么就需要用户自行定义serializer。

producer的序列化机制使用起来非常简单，只需要在构造producer时同时指定参数key.serializer和value.serializer的值即可。

|  |  |
| --- | --- |
| 常用serializer | |
| ByteArraySerializer | 本质上什么都不做，因为己然是字节数组。 |
| ByteBufferSerializer | 序列化ByteBuffer |
| BytesDeserializer | 序列化Kafka自定义的Bytes类 |
| DoubleDeserializer | Double类型 |
| IntegerDeserializer | Integer类型 |
| LongDeserializer | Long类型 |
| StringDeserializer | string类型 |

## 4.5 producer拦截器（interceptor）

producer拦截器和consumer拦截器是比较新的功能，它们是在Kafka0.10.0.0版本中被引入的，主要用于实现**clients端的定制化控制逻辑**。

**producer拦截器**使得用户在消息发送前以及producer回调逻辑前有机会对消息做一些定制化需求，比如修改消息等。同时，producer允许用户指定多个interceptor按序作用于同一条消息从而形成一个拦截链（interceptor chain）。intereetpor的实现接口是org.apache.kafka.clients.producer.ProducerInterceptor，其定义的方法如下。

|  |
| --- |
| ProducerRecord<K, V> onSend(ProducerRecord<K, V> record); |
| 该方法封装进KafkaProducer.send方法中，即它运行在用户主线程中。producer确保在消息被序列化以计算分区前调用该方法。用户可以在该方法中对消息做任何操作，但最好保证不要修改消息所属的topic和分区，否则会影响目标分区的计算。 |
| **void** onAcknowledgement(RecordMetadata metadata, Exception exception); |
| 该方法会在消息被应答之前或消息发送失败时调用，并且通常都是在producer回调逻辑触发之前。onAcknowledgement运行在producer的I/O线程中，因此不要在该方法中放入很“重”的逻辑，否则会拖慢producer的消息发送效率。 |
| **void** close(); |
| 关闭interceptor，主要用于执行一些资源清理工作。 |

如上所述，interceptor可能运行在多个线程中，因此在具体实现时用户需要自行确保线程安全。另外，若指定了多个inierceptor，则producer将按照指定顺序调用它们，同时把每个interceptor中捕获的异常记录到错误日志中而不是向上传递。这在使用过程中要特别留意。

## 4.6 无消息丢失配置

## 4.7 消息压缩

数据压缩能显著降低磁盘占用或带宽占用，从而有效提升I/O密集型应用的性能，不过引入压缩的同时会额外占用CPU资源，因此压缩是I/O性能和CPU资源的平衡。

Kafka支持压缩特性——producer端能够将一批消息压缩成一条消息发送，而broker端将这条压缩消息写入本地日志文件，当consumer获取到这条压缩消息时，它会自动地对消息进行解压缩，还原成初始的消息集合返还给用户。如果使用一句话来总结Kafka压缩特性的话，那么就是——producer端压缩，broker端保持，consumer端解压缩。所谓的broker端保持是指broker端在通常情况下不会进行解压缩操作，它只是原样保存消息而已。这里的“在通常情况下”表示要满足一定的条件，如果有些前置条件不满足（比如需要进行消息格式的转换等），那么broker端就需要对消息进行解压缩然后再重新压缩。

### 4.7.1 kafka支持的压缩算法

当前Kafka支持3种压缩算法：GZIP、snappy和LZ4。

虽然默认情况下Kafka是不压缩消息的，但用户可以通过设定producer端参数” compression.type”来开启消息压缩，比如：

|  |
| --- |
| props.put(ProducerConfig.***COMPRESSION\_TYPE\_CONFIG***, "snappy"); |

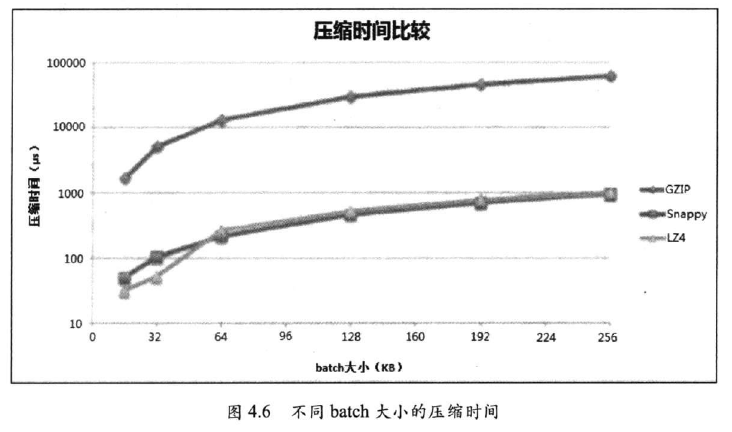
### 4.7.2 算法性能比较与调优

KafkaProducer.send方法逻辑的主要耗时都在消息压缩操作上，因此妥善地调优压缩算法至关重要。

首先比较一下目前Kafka支持的压缩算法的性能。对Kafka而言， LZ4 >> Snappy > GZIP，目前Kafka对LZ4压缩算法的支持是最好的，启用LZ4进行消息压缩的Producer的吞吐量是最高的。本来Snappy和LZ4无论在压缩速率还是压缩比率上都差不太多，但由于Kafka源代码中对Snappy的某个关键参数进行了硬编码，使得Snappy与Kafka的结合表现并不优秀，至少比LZ4要差很多。

下面说说如何调优producer的压缩性能。首先判断是否启用压缩的依据是I/O资源消耗与CPU资源消耗的对比。如果生产环境中的I/O资源非常紧张，比如producer程序消耗了大量的网络带宽或broker端的磁盘占用率非常高，而producer端的CPU资源非常富裕，那么就可以考虑为Producer开启消息压缩。反之则不需要设置消息压缩以节省宝贵的CPU时钟周期。

其次，压缩的性能与producer端的batch大小息息相关。通常情况下我们可以认为batch越大需要压缩的时间就越长，不过压缩时间的随batch大小的增长不是线性的，而是越来越平缓的，如下图所示。



如果发现压缩很慢，说明系统的瓶颈在用户主线程而不是I/O发送线程，因此可以考虑增加多个用户线程同时发送消息，这样通常能显著地提升producer吞吐量。

## 4.8 多线程处理

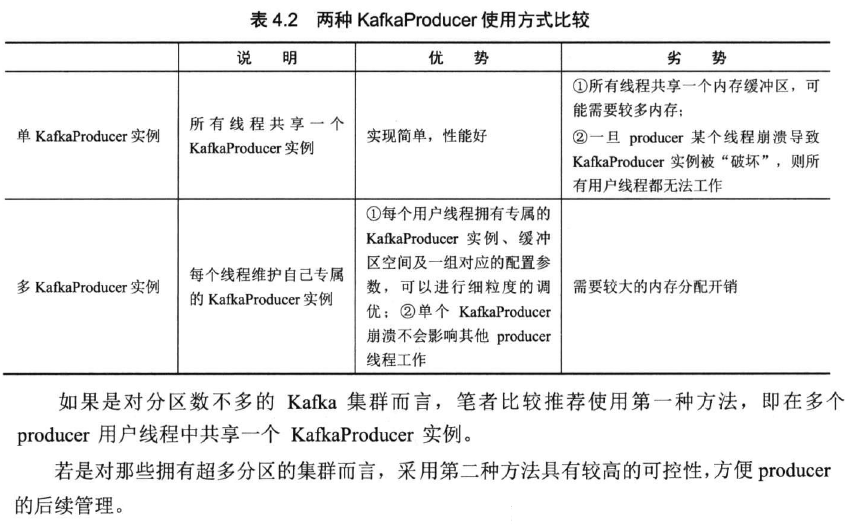
实际环境中只使用一个用户主线程通常无法满足所需的吞吐量目标，因此需要构造多个线程或多个进程来同时给Kafka集群发送消息，这样在使用过程中就存在以下两种使用方法。

***1、多线程单KafkaProducer实例***

这种方法是在全局构造一个KafkaProducer实例，然后在多个线程中共享使用。由于KafkaProducer是线程安全的，所以这种使用方式也是线程安全的。

***2、多线程多KafkaProducer实例***

这种方法是在每个producer主线程中都构造一个KafkaProducer实例，并且保证此实例在该线程中封闭（thread confinement，线程封闭是实现线程安全的重要手段之一）。



## ~~4.9 旧版本producer~~

# 5 consumer开发

## 5.1 consumer概览

kafka 消费者是从kafka集群中读取某些topic消息的应用。

若干consumer订阅kafka集群中的若干topic，并从kafka接收属于这些topic的消息。

### consumer版本

新旧两个版本的consumer设计上差异极大，新版consumer颠覆了旧版consumer**管理和保存位移的机制**。

### consumer分类

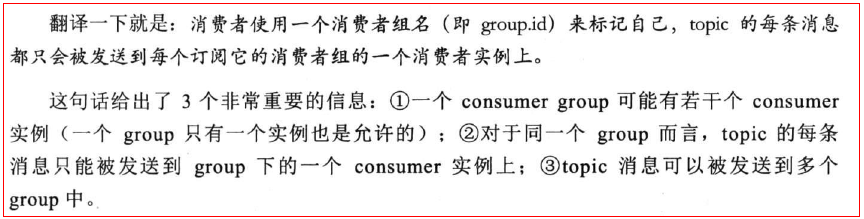
分类两类：消费者组（consumer group）、独立消费者（standalone consumer）

消费者组是由多个消费者实例（consumer instance）构成一个整体进行消费的，独立消费者则是单独进行消费的。

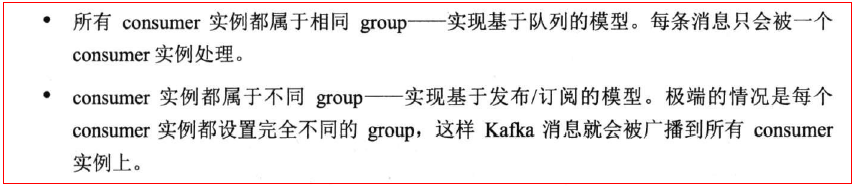
### 消费者组相关

#### 5.1.2 消费者组

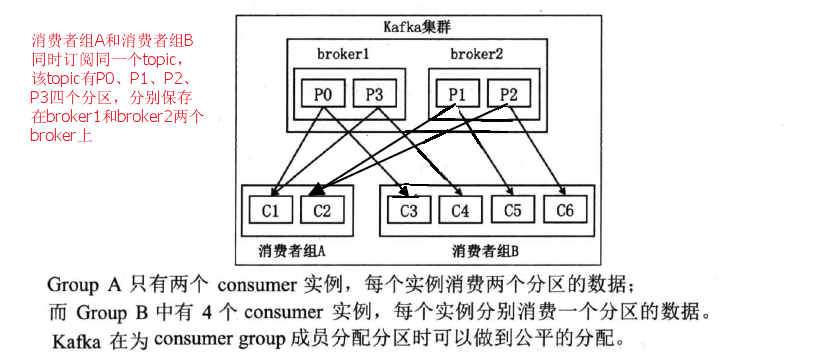
*消费者组定义：*



*kafka是通过消费者组来同时支持基于队列和基于发布/订阅这两种消息引擎模型的：*



*两个consumer group订阅相同topic的场景：*



*为什么需要consumer group？*

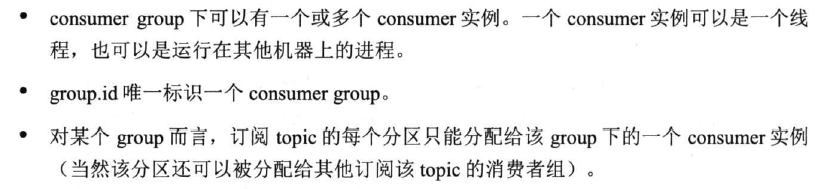
consumer group是用于实现高伸缩性、高容错性的consumer机制。

组内多个consumer实例可以同时读取kafka消息，一旦其中有某个consumer挂了，consumer group会立即将已崩溃consumer负责的分区转交给组内其它consumer来负责，从而保证真个consumer group可以继续工作，不会丢失数据——这个过程被称为**重平衡（rebalance）**。

*如何实现topic全局的消息读取顺序？*

由于kafka目前只提供单个partition内的消息顺序，而不会维护topic全局的消息顺序，因此如果用户要实现topic全局的消息读取顺序，就只能通过让每个consumer group只包含一个consumer实例的方式来间接实现。

*consumer group总结*



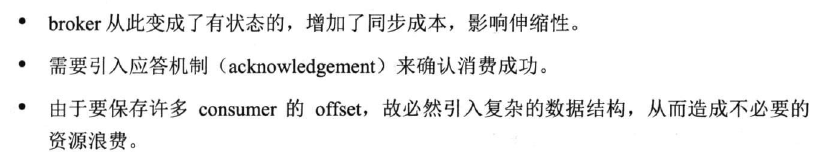
### 位移相关

#### 5.1.3 位移（offset）

**位移（offset）**：每个consumer实例都会为它消费的分区维护属于自己的位置信息来记录当前消费了多少条消息。

需要注意，这里的offset是指consumer端的offset，区别于分区日志中的offset。

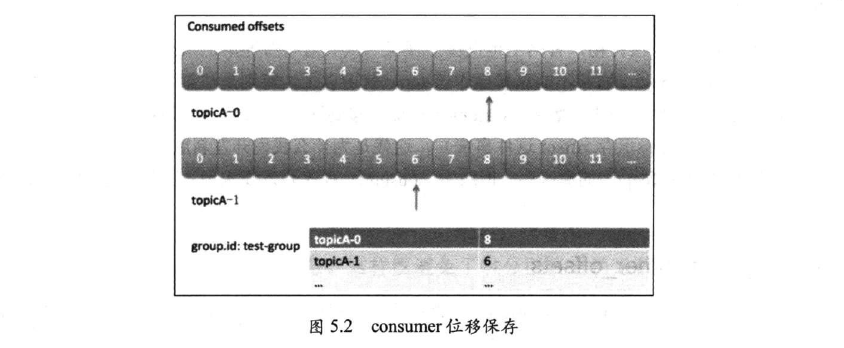
*很多消息引擎都把消费端的offset保存在broker端，这样虽然实现起来简单，但有以下3方面的问题：*



*因此，kafka选择了不同的方式，即让consumer来保存offset*。

如下图所示，consumer在内部使用一个map<topic-partition, long>来保存其订阅后所负责的topic-partition的offset。

同时， consumer还引入了检查点机制（checkpointing）来定期对offset进行持久化，从而简化了应答机制的实现。

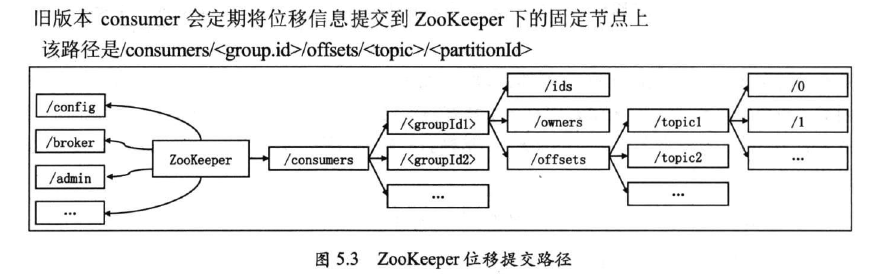
**

#### 5.1.4 位移提交

**位移提交（offset commit）**：consumer客户端需要定期向kafka集群汇报自己消费数据的进度。

位移提交不仅表征了consumer端的消费进度，同时也直接决定了**consumer端的消费语义保证**。

*旧版本位移提交方式：*

**

*新版本位移提交方式：*

随着Kafka consumer在实际场景的不断应用，社区发现旧版本consumer把位移提交到zooKeeper的做法并不合适。zooKeeper本质上只是一个协调服务组件，它并不适合作为位移信息的存储组件，毕竟频繁高并发的读／写操作并不是zooKeeper擅长的事情。

因此在新版本consumer中，位移提交的方式被彻底颠覆——新版本consumer把位移提交到Kafka的一个内部topic上（**\_\_consumer\_offsets**）。对于普通用户来说，只需要了解\_\_consumer\_offsets是一个内部topic，通常不能直接操作该topic就可以了，特别是注意不要擅自删除或搬移该topic的日志文件。

由于新版本consumer提交位移到这个topic，因此consumer不再依赖zooKeeper（主要是指保存offset这件事情），这就是为什么我们开发新版本consumer应用时不再需要连接zooKeeper的原因——旧版本consumer程序开发要求用户必须指定zooKeeper地址。

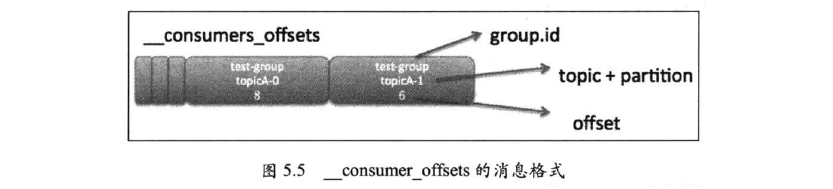
#### 5.1.5 \_\_consumer\_offsets

首先，要明确的是，这个topic通常是给新版本consumer使用的。为什么说是通常？因为旧版本consumer确实也提供了一个特定的参数让用户在使用旧版本consumer时把位移提交到这个topic上，那就是设置旧版本consumer的参数offsets.storage=kafka。不过笔者很少看到有这样的用法，因此我们可以安全地认为这个内部topic就是为新版本consumer保存位移的。

其次，\_\_consumer\_offsets是Kafka自行创建的一个内部topic，用户应尽量避免执行该topic的任何操作，特别是不可擅自删除或搬移该topic的日志文件。

当我们搭建起Kafka集群并执行了一些消费操作后，会发现在Kafka的日志目录下会出现50个与\_\_consumer\_offsets相关的目录（从\_\_consumer\_offsets\_0到\_\_consumer\_offsets\_49）。其中每个目录都是一个正常的kafka topic 日志文件目录，里面至少有一个日志文件（.log）和两个索引文件（.index和.timeindex)。

只不过该日志中保存的消息都是Kafka集群上consumer（特别是consumergroup)的位移信息罢了。\_\_consumer\_offsets的每条消息格式大致如下图所示，可以把它想象成一个KV格式的消息，key就是一个三元组（group.id + topic + 分区号），value就是offset值。



每当更新同一个key的最新offset值时，该topic就会写入一条含有最新offset的消息，同时Kafka会定期地对该topic执行压实操作（compact)，即为每个消息key只保存含有最新offset的消息。这样既避免了对分区日志消息的修改，也控制住了\_\_consumer\_offsets这个topic总体的日志容量，同时还能实时反映最新的消费进度。

考虑到一个Kafka生产环境中可能有很多consumer或consumer group，如果这些consumer同时提交位移，则必将加重**\_\_consumer\_offsets的写入负载**，因此社区特意为该topic创建了50个分区，并且对每个group.id做哈希求模运算，从而将负载分散到\_\_consumer\_offsets的不同分区上。这就是说，每个consumer group保存的offset都有极大的概率分别出现在该topic的不同分区上。

我们将会在监控相关章节中介绍如何利用＿consumer\_offsets定位并读取consumer group的offset信息。

## 5.2 构建consumer

### 5.2.3 consumer端主要参数

|  |  |
| --- | --- |
| bootstrap.servers  必须指定 | 该参数指定了一组host:Port对，用于创建与Kafka broker服务器的Socket连接，如”kafka1:9092, kafka2:9092, kafka3:9092”。  若Kafka集群中broker机器数很多，我们只需要指定部分broker即可，不需要列出完整的broker列表，这是因为不管指定了几台机器，consumer启动后都能通过这些机器找到完整的broker列表，因此为该参数指定多台机器通常只是为了常规的failover使用。这样即使某一台broker挂掉了，consumer重启后依然能够通过该参数指定的其他broker连接Kafka集群。  **注意**，如果broker端没有显式配置listeners（或advertised.listeners）使用IP地址的话，那么最好将bootstrap.servers配置成主机名而不要使用IP地址，因为Kafka内部使用的是全称域名（FQDN,FullyQualifiedDomainName）。倘若不统一，会出现无法获取元数据的异常。 |
| group.id  必须指定 | 该参数指定的是consumer group的名字，它能够唯一标识一个consumer group，通常为group.id设置一个有业务意义的名字就可以了。细心的读者可能会发现官网上该参数是有默认值的，即一个空字符串。但在开发consumer程序时我们依然要显式指定group.id，否则consumer端会抛出InvalidGroupldException异常。 |
| key.deserializer  必须指定 | consumer代码从broker端获取的任何消息都是字节数组的格式，因此消息的每个组件都要执行相应的解序列化操作才能“还原”成原来的对象格式。这个参数就是为消息的key做解序列化的。该参数值必须是实现org.apache.kafka.common.serialization.Deserializer接口的Java类的全限定名称。  Kafka默认为绝大部分的初始类型（primitive type）提供了现成的解序列化器。org.apache.kafka.common.serialization.StringDeserializer类会将接收到的字节数组转换成UTF-8编码的字符串。  consumer支持用户自定义Deserializer，这通常都与producer端自定义serializer“遥相呼应”。毕竟“原汤化原食”，使用了什么类型的serializer就必须使用对应类型的deserializer将其还原。  注意，不论consumer消费的消息是否指定了key，consumer都必须要设置这个参数，否则程序会抛出configException，提示“key.deserializer”没有默认值。 |
| value.deserializer  必须指定 | 与value.deserializer类似，该参数被用来对消息体（即消息的value）进行解序列化，从而把消息“还原”回原来的对象类型。 |
|  |  |
| session.timeout.ms |  |
| max.poll.interval.ms |  |
| auto.offset.reset |  |
| enable.auto.commit |  |
| fetch.max.bytes |  |
| max.poll.records |  |
| heartbeat.interval.ms |  |
| connections.max.idle.ms |  |

### 5.2.1 consumer程序实例

#### KafkaConsumer的poll方法

consumer使用poll方法订阅topic中并行地获取多个分区的消息。为了实现这一点，新版本consumer的poll方法使用了类似于Linux的select I/O机制——所有相关的事件（包括rebalance、获取消息等）都发生在一个事件循环(event Ioop）中。这样consumer端只使用一个线程就能够完成所有类型的I/O操作。一个常见的event loop写法如下：

|  |
| --- |
| try {  while (true) {  ConsumerRecords<String, String> records = consumer.poll(1000);  for (ConsumerRecord<String, String> record : records) {  // 执行具体的消费逻辑  }  }  } finally {  consumer.close();  } |
| 上面代码中最关键的调用方法当属consumer.poll(1000）。这里的1000是一个超时设定(timeout）。通常情况下如果consumer拿到了足够多的可用数据，那么它可以立即从该方法返回；但若当前没有足够多的数据可供返回，consumcr会处于阻塞状态，这个超时参数即控制阻塞的最大时间。  这个超时设定给予了用户能够在consumer消费间隔之余做一些其他事情的能力。若用户有定时方面的需求，那么根据需求设定timeout是一个不错的选择。否则，设定一个比较大的值甚至Integer.MAX\_VALUE，是不错的建议。 |

#### 处理ConsumerRecord对象

poll调用会返回以ConsumerRecord类封装的Kafka消息，拿到这些消息后我们通常都需要处理这些消息的逻辑。当consumer程序慢时，可能是调用poll慢，也可能是处理逻辑慢。如果你发现poll返回消息的速度过慢，那么可以调节相应的参数来提升poll方法的效率；若消息的业务级处理逻辑过慢，则应该考虑简化处理逻辑或者把处理逻辑放入单独的线程执行。

### 5.2.2 consumer脚本命令

除了自己写程序构建consumer之外，Kafka还自带了方便使用的控制台consumer脚本用于日常的调试验证。该脚本的名字叫kafka-console-consumer，在Linux平台下它位于＜kafka目录＞/bin下，在Windows平台下它位于＜kafka目录＞/bin/windows下。

kafka-console-consumer脚本常见的使用参数如下:

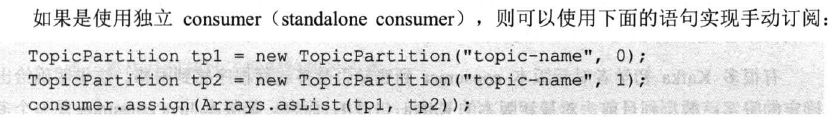
|  |  |
| --- | --- |
| --bootstrap-server | 指定Kafka broker列表 |
| --topic | 指定要消费的topic名 |
| --from- beginning | 指定从头消费 |
| 其它参数 | 我们可以不加参数地运行kafka-console-consumer脚本以获取完整的参数列表及其使用方法。 |
| bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning | |

## 5.3订阅topic

### 5.3.1 订阅topic列表

consumer group使用KafkaConsumer的subscribe方法来订阅要消费的topic列表。

|  |
| --- |
| // 订阅一个topic  consumer.subscribe(Arrays.asList("test-java-topic")); |
| // 订阅多个topic  consumer.subscribe(Arrays.asList("topic1", "topic2", "topic3"));  // **注意，subscribe方法不是增量式的**，这意味着后续的subscribe调用会完全覆盖之前的订阅语句。执行下面两行代码后，订阅的topic列表是topic4,topic5,topic6。  consumer.subscribe(Arrays.asList("topic1", "topic2", "topic3"));  consumer.subscribe(Arrays.asList("topic4", "topic5", "topic6")); |



不管是哪种方法，consumer订阅是延迟生效的，即订阅信息只有在下次poll调用时才会正式生效。如果在poll之前打印订阅信息，用户会发现它的订阅信息是空的，表明尚未生效。

### 5.3.2 基于正则表达式订阅topic

除了topic列表订阅方式，新版本consumer还支持基于正则表达式的订阅方式。利用正则表达式可以达到某种程度上的动态订阅效果。一个实际的例子如下：

|  |
| --- |
| consumer.subscribe(Pattern.*compile*("kafka-.\*"), **new** ConsumerRebalanceListener() {  @Override  **public** **void** onPartitionsRevoked(Collection<TopicPartition> partitions) {  // **TODO** Auto-generated method stub  }  @Override  **public** **void** onPartitionsAssigned(Collection<TopicPartition> partitions) {  // **TODO** Auto-generated method stub  }  }); |

使用基于正则表达式的订阅就必须指定ConsumerRebalanceListener实现类（），ConsumerRebalanceListener是一个回调接口，主要用于重平衡，用户需要通过实现这个接口来实现consumer分区分配方案发生变更时的逻辑。

如果用户使用的是自动提交（即设置enable.auto.conunit=true)，则通常不用理会这个类，使用空实现的new NoOpConsumerRebalanceListener()可以了。但如果用户是手动提交位移的，则至少要在ConsumerRebalanceListener实现类的onPartitionsRevoked方法中处理分区分配方案变更时的位移提交。

## 5.4 消息轮询

### 5.4.1 poll内部原理

Kafka的consumer是用来读取消息的，而且要能够并行读取多个topic的多个分区的消息。若要实现并行的消息读取，一种方法是使用多线程的方式，为每个要读取的分区都创建一个专有的线程去消费（这其实就是旧版本consumer采用的方式，本章后面会专门讨论旧版本consumer的设计与使用）；另一种方法是采用类似于Linux I/O模型的poll或se!ect等，使用一个线程来同时管理多个Socket连接，即同时与多个broker通信实现消息的并行读取——这就是新版本consumer最重要的设计改变。

一旦consumer订阅了topic，所有的消费逻辑包括coordinator的协调、消费者组的rebalance以及数据的获取都会在主逻辑poll方法的一次调用中被执行。这样用户很容易使用一个线程来管理所有的consumer I/O操作。

有很多Kaflca初学者对新版本consumer到底是否是多线程程序感到困惑。这里明确给出确定的答案：截止到目前，对最新版本的Kafka(1.0.0）而言，新版本Java consumer是一个多线程或者说是一个双线程的Java进程——创建Kafka Consumer的线程被称为用户主线程，同时consumer在后台会创建一个心跳线程，该线程被称为后台心跳线程。Kafka consumer的poll方法在用户主线程中运行。这也同时表明：消费者组执行rebalance、消息获取、coordinator管理、异步任务结果的处理甚至位移提交等操作都是运行在用户主线程中的。因此仔细调优poll方法相关的各种处理超时时间参数至关重要。

### 5.4.2 poll使用方法

consumer订阅topic之后通常以**事件循环**的方式来获取订阅方案并开启消息读取。听上去似乎有些复杂，但其实用户要做的仅仅是写一个循环，然后重复性地调用poll方法，剩下所有的工作都交给poll方法帮用户完成。

每次poll方法返回的都是订阅分区上的一组消息。当然如果某些分区没有准备好，某次poll返回的就是空的消息集合。下面的一段代码展示了常见的poll调用方式：

|  |
| --- |
| **try** {  **while** (isRunning) {  ConsumerRecords<String, String> records = consumer.poll(1000);  **for** (ConsumerRecord<String, String> record : records) {  // 执行具体的消费逻辑  }  }  } **finally** {  consumer.close();  } |
| 在上面的代码中，我们在while的条件语句中指定了一个布尔变量值isRunning来标识是否要退出consumer消费循环并结束consumer应用。具体的做法是，将isRunning标识为volatile型，然后在其他线程中设置isRunning=false来**控制consumer的结束**。 |
| 最后千万不要忘记关闭consumer。这不仅会清除consumer创建的各种Socket资源，还会通知消费者组coordinator主动离组从而更快地开启新一轮rebalance。比较推荐的做法是，在finally代码块中显式调用consumer.close()，从而保证consumer总是能够被关闭的。 |

poll方法根据当前consumer的消费位移返回消息集合。当poll首次被调用时，新的消费者组会被创建并根据对应的位移重设策略（auto.offset.reset）来设定消费者组的位移。一旦consumer开始提交位移，每个后续的rebalance完成后都会将位置设置为上次己提交的位移。

传递给poll方法的超时设定参数用于控制consumer等待消息的最大阻塞时间。由于某些原因，broker端有时候无法立即满足consumer端的获取请求（比如consumer要求至少一次获取1MB的数据，但broker端无法立即全部给出），那么此时consumer端将会阻塞以等待数据不断累积并最终满足consumer需求。如果用户不想让consumer一直处于阻塞状态，则需要给定一个超时时间。因此poll方法返回满足以下任意一个条件即可返回。

1、要么获取了足够多的可用数据。

2、要么等待时间超过了指定的超时设置。

前面我们谈到了consumer是单线程的设计理念（这里暂不考虑后台心跳线程，因为它只是一个辅助线程，并没有承担过重的消费逻辑），因此consumer就应该运行在它专属的线程中。新版本Java consumer不是线程安全的！如果没有显式地同步锁保护机制，Kafka会抛出”KafkaConsumer is not safe for multi-threaded access”异常。如果用户在调用poll方法时看到了这样的报错，通常说明用户将同一个KafkaConsumer实例用在了多个线程中。至少对于目前的Kafka设计而言，这是不被允许的，用户最好不要这样使用。

KafkaConsumer的poll方法为什么会有一个超时的参数？实际使用中应该根据什么规则来设定此值？要回答这些问题，我们需要思考一下引入这个参数的初衷。诚然，它是超时的设定，但Kafka社区引入这个参数的目的其实是想让consumer程序有机会定期“醒来”去做一些其他的事情。假设用户的consumer程序除了消费之外还需要定期地执行其他的常规任务（比如每隔10秒需要将消费情况记录到日志中），那么用户就可以使用consumer.poll(10000)来让consumer有机会在等待Kafka消息的同时还能够定期执行其他任务。这就是使用超时设定的最大意义。

另一方面，若用consumer程序除了消费消息之外没有其他的定时任务需要执行，即consumer程序唯一的目的就是从Kafka获取消息然后进行处理，那么用户可采用与上面代码完全不同的poll调用方法，如下面的代码所示：

|  |
| --- |
| **try** {  **while** (**true**) {  ConsumerRecords<String, String> records =  consumer.poll(Long.***MAX\_VALUE***);  **for** (ConsumerRecord<String, String> record : records) {  // 执行具体的消费逻辑  }  }  } **catch**(WakeupException e) {  // 异常处理  } **finally** {  // 关闭KafkaConsumer  consumer.close();  } |

上面代码中，我们让consumer程序在未获取到足够多数据时无限等待，然后通过捕获WakeupException异常来判断consumer是否结束。

如果上面这种方式调用poll，那么需要在另一个线程中调用consumer.wakeup()方法来触发consumer的关闭。前面我们说过，KafkaConsumer不是线程安全的，但是有一个例外：用户可以安全地在另一个线程中调用consumer.wakeup()。注意，只有wakeup方法是特例，其他KafkaConsumer方法都不能同时在多线程中使用。

WakeupException异常是在poll方法中被抛出的，因此如果当前事件循环代码正在执行poll之后的消息处理逻辑，则它并不会马上响应wakeup，只会等待下次poll调用时才进行响应。举一个实际的例子，假设每次poll返回消息后，用户consumer程序都需要为这些消息执行很繁重的计算工作，那么在计算过程中该consumer是不会响应另一个线程调用的wakeup的，它只能在下次poll时才响应。所以程序表现为不能立即退出，会有一段延迟时间，这也是为什么不推荐用户将很繁重的消息处理逻辑放入poll主线程执行的原因。

*poll的使用方法总结：*

|  |  |
| --- | --- |
| consumer需要定期执行其他子任务 | poll(较小超时时间) + 运行标识布尔变量 |
| consumer不需要定期执行子任务 | poll(MAX\_VALUE) + 捕获WakeupException |

## 5.5 位移管理

### 5.5.1 consumer位移

consumer端需要为它负责的每个分区保存消费进度，即保存分区中下一条待消费的消息的位置，该位置就被称为**位移（offset）**。consumer需要定期地向Kafka提交自己的位移信息。假设consumer己经读取了某个分区中的第N条消息，那么它应该提交位移值为N，因为位移是从0开始的，位移为N的消息是第(N+1)条消息。这样下次consumer重启时会从第(N+1)条消息开始消费。

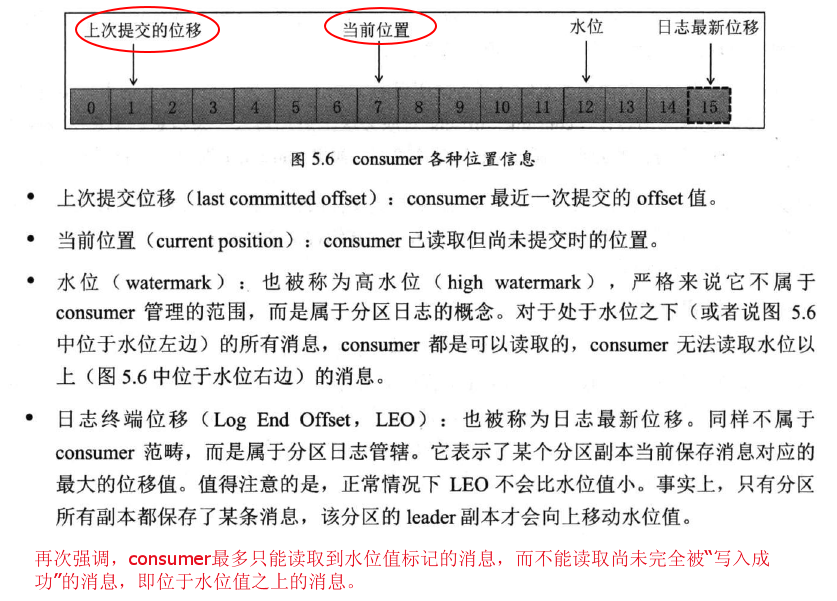
总而言之，offset就是consumer端维护的位置信息。

offset对于consumer非常重要，因为它是实现消息交付语义保证（message delivery semantic）的基石。常见的3种消息交付语义保证如下。

|  |  |
| --- | --- |
| 最多一次（at-most-once）处理语义 | 消息可能丢失，但不会被重复处理。 |
| 最少一次（at-least-once）处理语义 | 消息不会丢失，但可能被处理多次。 |
| 精确一次（exactly-once）处理语义 | 消息一定会被处理且只会被处理一次。 |

显然，若consumer在消息消费之前就提交位移，那么便可以实现at-most-once——因为若consumer在提交位移与消息消费之间崩溃，则consumer重启后会从新的offset位置开始消费，前面的那条消息就丢失了。相反地，若提交位移在消息消费之后，则可实现at-least-once语义。由于Kafka没有办法保证这两步操作可以在同一个事务中完成，因此Kafka默认提供的就是at-least-once的处理语义。好消息是Kafka社区已于0.11,0.0版本正式支持事务以及精确一次处理语义。

既然offset本质上就是一个位置信息，那么就需要和其他一些位置信息区别开来。图5.6给出了与consumer相关的多个位置信息。



### 5.5.2 新版consumer位移管理

consumer会在Kafka集群的所有broker中选择一个broker作为consumer group的coordinator，用于实现组成员管理、消费分配方案制定以及提交位移等。为每个组选择对应coordinator的依据就是前面介绍的\_\_consumer\_offsets主题。和普通的Kafka topic相同，该topic配置有多个分区，每个分区有多个副本。它存在的唯一目的就是保存consumer提交的位移。

当消费者组首次启动时，由于没有初始的位移信息，coordinator必须为其确定初始位移值，这就是consumer参数auto.offset.reset的作用。通常情况下，consumer要么从最早的位移开始读取，要么从最新的位移开始读取。

当consumer运行了一段时间之后，它必须要提交自己的位移值。如果consumer崩溃或被关闭，它负责的分区就会被分配给其他consumer，因此一定要在其他consumer读取这些分区前就做好位移提交工作，否则会出现消息的重复消费。

consumer提交位移的主要机制是通过向所属的coordinator发送位移提交请求来实现的。每个位移提交请求都会往\_\_consumer\_offsets对应分区上追加写入一条消息。消息的key是group.id、topic和分区的元组，而value就是位移值。如果consumer为同一个group的同一个topic分区提交了多次位移，那\_\_consumer\_offsets对应的分区上就会有若干条key相同但value不同的消息，但显然我们只关心最新一次提交的那条消息。从某种程度来说，只有最新提交的位移值是有效的，其他消息包含的位移值其实都己经过期了。Kafka通过压实（compact)策略来处理这种消息使用模式。我们会在第6章中详细讨论这种策略以及如何应用这种策略到内部topic和位移管理上。

### 5.5.3 自动提交与手动提交

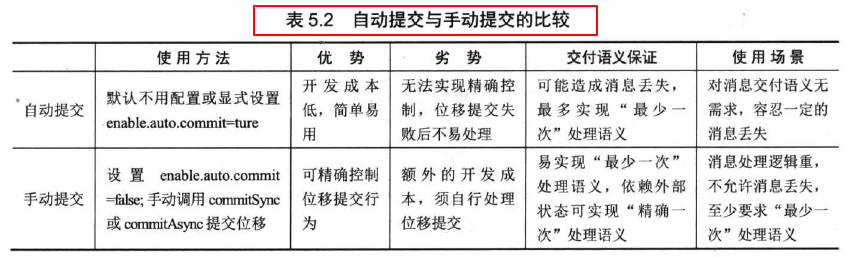
如前所述，位移提交策略对于提供消息交付语义至关重要。默认情况下，consumer是自动提交位移的，自动提交间隔是5秒。这就是说若不做特定的设置，consumer程序在后台自动提交位移。通过设置auto.commit.interval.ms参数可以控制自动提交的间隔。

自动位移提交的优势是降低了用户的开发成本使得用户不必亲自处理位移提交；劣势是用户不能细粒度地处理位移的提交，特别是在有较强的精确一次处理语义时，在这种情况下，用户可以使用手动位移提交。

所谓的手动位移提交就是用户自行确定消息何时被真正处理完并可以提交位移。在一个典型的consumer应用场景中，用户需要对poll方法返回的消息集合中的消息执行业务级的处理。用户想要确保只有消息被真正处理完成后再提交位移。如果使用自动位移提交则无法保证这种时序性，因此在这种情况下必须使用手动提交位移。

设置使用手动提交位移非常简单，仅仅需要在构建KafkaConsumer时设置enablc.auto.commit=false，然后调用commitSync或commitAsync方法即可。一段典型的手动提交代码如下：

|  |
| --- |
| Properties consumerProps = **new** Properties();  consumerProps.put("bootstrap.servers", "localhost:9092");  consumerProps.put("group.id", "test-group");  consumerProps.put("enable.auto.commit", **false**);  KafkaConsumer<String, String> consumer = **new** KafkaConsumer<>(consumerProps,  **new** StringDeserializer(), **new** StringDeserializer());  consumer.subscribe(Arrays.*asList*("test-java-topic"));// 订阅topic列表  // 定义一个缓冲区，然后按批获取消息并将它们加入缓冲区中。  // 当积累了足够多的消息时便统一入库，然后手动提交位移，并清空缓冲区以备缓存下一批消息。  **final** **int** minBatchSize = 500;  List<ConsumerRecord<String, String>> buffer = **new** ArrayList<>();  **while**(**true**) {  ConsumerRecords<String, String> records = consumer.poll(1000);  **for** (ConsumerRecord<String, String> record : records) {  buffer.add(record);  }  **if**(buffer.size() >= minBatchSize) {  *inertIntoDB*(buffer); // 入库  // 注意：若在成功入库后但在提交位移语句执行之前consumer程序崩溃了，  // 由于未成功提交位移，consumer重启后会重新处理之前的一批消息并将它们再次入库，从而造成消息被重复消费。  consumer.commitSync();  buffer.clear();  }  } |



#### commitAsync和commitSync

手动提交位移API进一步细分为同步手动提交(commitSync方法)和异步手动提交(commitAsync方法)。

如果调用的是commitsync，用户程序会等待位移提交结束才执行下一条语句命令。

相反地，若是调用commitAsync方法，则是一个异步非阻塞调用。特别注意的是，这里的异步提交位移不是指consumer使用单独的线程进行位移提交，只是该提交发起时此方法是不会阻塞的，因而被称为异步提交。实际上consumer会在用户主线程的下一次poll调用时不断轮询上次异步提交的结果。

目前为止我们谈到的只是commitSync或commitAsync的无参版本。当用户调用consumer. commitSync ()或consumer.commitAsync ()时，consumer会为所有它订阅的分区提交位移。commitSync或commitAsync还有另外带参数的重载方法，用户调用这个版本的方法时需要指定一个Map显式地告诉Kafka为哪些分区提交位移。实际使用过程中笔者更加推荐这个版本，因为consumer只对它所拥有的分区做提交是更合理的行为，而且consumer通常都有更加细粒度化的位移提交策略。一段典型的手动提交部分分区位移的代码如下：

|  |
| --- |
| **try** {  **while** (running) {  ConsumerRecords<String, String> records = consumer.poll(1000);  // 对poll方法返回的消息集合按分区进行分组，然后按分区进行位移提交  **for** (TopicPartition partition : records.partitions()) {  List<ConsumerRecord<String, String>> partitionRecords =  records.records(partition);  **for** (ConsumerRecord<String, String> record : partitionRecords) {  System.***out***.println(record.offset() + ": " + record.value());  }  **long** lastOffset = partitionRecords  .get(partitionRecords.size() - 1).offset();  // 注意：提交的位移应是下一条待读取消息的位移  consumer.commitSync(Collections.*singletonMap*(partition,  **new** OffsetAndMetadata(lastOffset + 1)));  }  }  } **finally** {  consumer.close();  } |

### 5.5.4 旧版consumer位移管理

旧版本consumer的位移默认保存在zooKeeper节点中，\_\_consumer\_offsets完全没有关系。读者可以复习5.1.4节的内容了解具体保存位移的ZooKecper节点位置。

旧版本consumer也区分自动提交位移和手动提交位移，只不过区分它们的参数名叫auto.commit.enable，而不是新版本的enable.auto.commit。另外旧版本consumer默认的提交间隔是60秒，而不是新版本的5秒。该间隔由参数auto.commit.interval.ms控制。

当设置成手动提交位移时，用户需要在consumer程序中显式调用ConsumerConnector.commitOffsets方法来提交位移。和新版本consumer类似，如果直接调用commitOffsets()，则会为该consumer订阅的所有分区都提交位移；若是调用commitOffsets (Map）版本，则可以实现细粒度化的位移提交。本章5.10节“旧版本consumer”中将给出该方法调用的具体示例。

## 5.6重平衡(rebanance)

### 5.6.1 rebanance概览

重平衡（rebalance）只对消费者组有效，独立消费者没有rebanance的概念。

消费者组的rebalance本质上是一种协议，它规定了一个consumer group下所有consumer如何达成一致来分配订阅topic的所有分区。举个例子，假设一个consumer group下有20个consumer实例，该group订阅了一个具有100个分区的topic；正常情况下，kafka会为group下的每个consumer平均分配5个分区；这个分配过程就被称作rebalance。

旧版consumer依托于zookeeper进行rebanance。

新版本consumer则使用了一个Kafka内置的组协调协议（group coordination protocol）。对于每个组而言，Kafka的某个broker会被选举为组协调者（group coordinator）。coordinator负责对组的状态进行管理，它的主要职责就是当新成员到达时促成组内所有成员达成新的分区分配方案，即coordinator负责对组执行rebalance操作。

### 5.6.2 rebanance触发条件

|  |
| --- |
| 组rebalancc触发的条件有以下3个。 |
| 1、组成员发生变更，比如新consumer加入组，或己有consumer主动离开组，再或是已  有consumer崩溃时则会触发rebalance。  2、组订阅topic数发生变更，比如使用基于正则表达式的订阅，当匹配正则表达式的新topic被创建时则会触发rebalance。  3、组订阅topic的分区数发生变更，比如使用命令行脚本增加了订阅topic的分区数。 |

真实应用场景中引发rebalance最常见的原因就是违背了第一个条件，特别是consumer崩溃的情况。这里的崩溃不一定就是指consumer进程“挂掉”或consumer进程所在的机器宕机。当consumer无法在指定的时间内完成消息的处理，那么coordinator就认为该consumer已经崩溃，从而引发新一轮rebalance。

举一个真实的案例，笔者曾经碰到过一个Kafka线上环境，发现该环境中的consumer group频繁地进行rebalance，但组内所有consumer程序都未出现崩溃的情况，另外消费者组的订阅情况也从未发生过变更。最终定位到原因：该group下的consumer处理消息的逻辑过重，而且事件处理时间波动很大，非常不稳定，从而导致coordinator会经常性地认为某个consumer已经挂掉，引发rebalance。而consumer程序中包含了错误重试的代码，使得落后过多的consumer会不断地中请重新加入组，最后表现为coordinator不停地对group执行rebalance，极大地降低了consumer端的吞吐量。

鉴于目前一次rebalance操作的开销很大，生产环境中用户一定要结合自身业务特点仔细调优consumer参数request.timeout.ms、max.poll.records和max.poll.interval.ms，以避免不必要的rebalance出现。

### 5.6.3 rebanance分区分配

之前提到过在rebalance时group下所有的consumer都会协调在一起共同参与分区分配，这是如何完成的呢？Kafka新版本consumer默认提供了3种分配策略，分别是range策略、

round-robin策略和sticky策略。

所谓的**分配策略**决定了订阅topic的每个分区会被分配给哪个consumer。

range策略主要是基于范围的思想。它将单个topic的所有分区按照顺序排列，然后把这些分区划分成固定大小的分区段并依次分配给每个consumer;

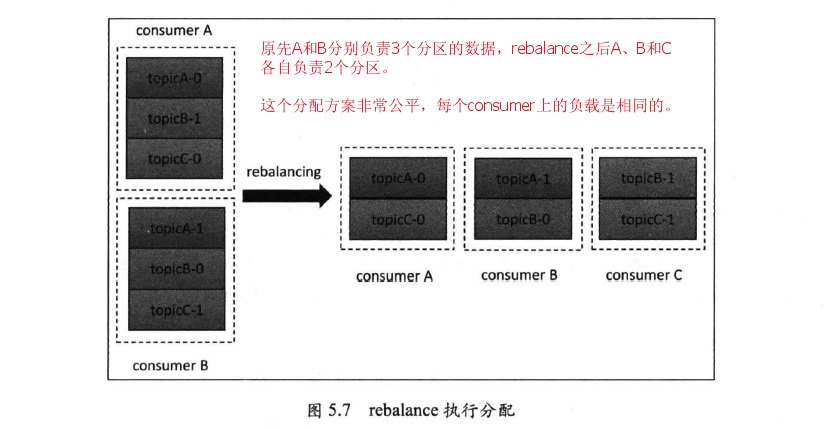
round-rohin策略则会把所有topic的所有分区顺序摆开，然后轮询式地分配给各个consumer。

最新发布的sticky策略有效地避免了上述两种策略完全无视历史分配方案的缺陷，采用了“有钻性”的策略对所有consumer实例进行分配，可以规避极端情况下的数据倾斜并且在两次rebalance间最大限度地维持了之前的分配方案。

通常意义上认为，如果group下所有consumer实例的订阅是相同，那么使用round-robin会带来更公平的分配方案，否则使用range策略的效果更好。此外，sticky策略在0.11.0.0版本才被引入，故目前使用的用户并不多。

用户根据consumer参数partition.assignment.strategy来设置分配策略，新版本consumer默认的分配策略是range。另外Kafka支持自定义的分配策略，用户可以创建自己的consumer分配器（assignor）。

下面举例说明rebalance过程中的分区分配，假设目前某个consumer group下有两个consumer（即A和B），该group订阅了3个topic（即topicA、topicB、topicC）,其中每个topic有两个分区。当第3个成员C加入时，满足了前面谈到的第一个触发条件，因此coordinator会执行rebalance，并根据range分配策略重新为A、B和C分配分区，如下图所示。

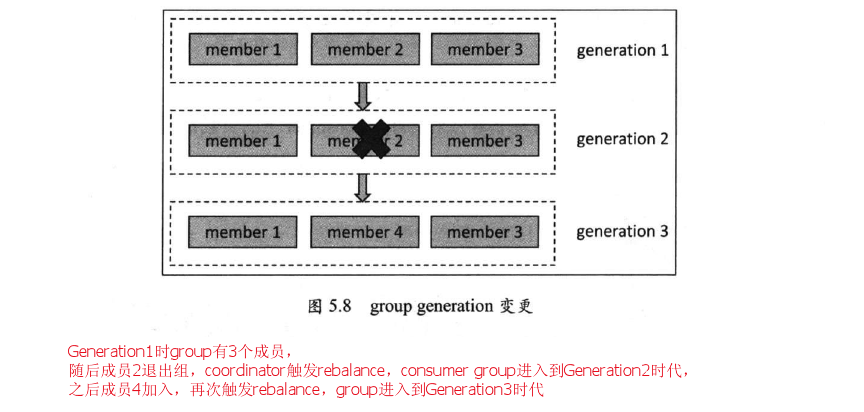


### 5.6.4 rebanance generation

某个consumer group可以执行任意次rebalance。为了更好地隔离每次rebalance上的数据，新版本consumer设计了rebalance generation用于标识某次rebalance。generation这个词类似于JVM分代垃圾收集器中“分代”（严格来说，JVM GC使用的是generational）的概念。笔者把它翻译成“届”，表示rebalance之后的一届成员，在consumer中它是一个整数，通常从0开始。

Kafka引入consumer generation主要是为了保护consumer group的，特别是防止无效offset提交。比如上一届的consumer成员由于某些原因延迟提交了offset，但rebalance之后该group产生了新一届的group成员，而这次延迟的offset提交携带的是旧的generation信息，因此这次提交会被consumer group拒绝。

很多Kafka用户在使用consumer时经常碰到的ILLEGAL\_GENERATION异常就是这个原因导致的。事实上，每个group进行rebalance之后，generation号都会加1，表示group进入了一个新的版本，如下图所示。



### 5.6.5 rebanance协议

前面提到过rebalance本质上是一组协议。group与coordinator共同使用这组协议完成group的rebalance。最新版本Kafka中提供了下面5个协议来处理rebalance相关事宜。

|  |  |
| --- | --- |
| JoinGroup请求 | consumer请求加入组 |
| SyncGroup请求 | group leader把分配方案同步更新到组内所有成员中。 |
| Heartbeat请求 | consumer定期向coordinator汇报心跳表明自己依然存活。 |
| LeaveGroup请求 | consumer主动通知coordinator该consumer即将离组。 |
| DescribeGroup请求 | 查看组的所有信息，包括成员信息、协议信息、分配方案以及订阅信息等。该请求类型主要供管理员使用。coordinator不使用该请求执行rebalance。 |

在rebalance过程中，coordinator主要处理consumer发过来的JoinGroup和SyncGroup请求。当consumer主动离组时会发送LeaveGroup请求给coordinator。

在成功rebalance之后，组内所有consumer都需要定期地向coordinator发送Heartbeat请求。而每个consumer也是根据Heartbeat请求的响应中是否包含REBALANCE\_IN\_PROGRESS来判断当前group是否开启了新一轮rebalance。

### 5.6.6 rebanance流程

consumer group在执行rebalance之前必须首先确定coordinator所在的broker，并创建与该broker相互通信的Socket连接。

确定coordinator的算法与确定offset被提交到\_\_consumer\_offsets目标分区的算法是相同的，算法如下：

|  |
| --- |
| 1、计算Math.abs(groupID.hashCode) % offsets.topic.num.partitions参数值（默认是50)，假设是10。  2、寻找\_\_consumer\_offsets分区10的leader副本所在的broker，该broker即为这个goup的coordinator。 |

成功连接coordinator之后便可以执行rebalance操作。目前rebalance主要分为两步：加入组和同步更新分配方案。

|  |  |
| --- | --- |
| 加入组 | 这一步中组内所有consumer（即group.id相同的所有consumer实例）向coordinator发送JoinGroup请求。当收集全JoinGroup请求后，coordinator从中选择一个consumer担任group的leader，并把所有成员信息以及它们的订阅信息发送给leader。  特别需要注意的是，group的leader和coordinator不是一个概念。leader是某个consumer实例，coordinator通常是Kafka集群中的一个broker。另外leader而非coordinator负责为整个group的所有成员制定分配方案。 |
|  | |
| 同步更新分配方案 | 这一步中leader开始制定分配方案，即根据前面提到的分配策略决定每个consumer都负责哪些topic的哪些分区。一旦分配完成，leader会把这个分配方案封装进SyncGroup请求并发送给coordinator。比较有意思的是，组内所有成员都会发送syncGroup请求，不过只有leader发送的syncGroup请求中包含了分配方案。coordinator接收到分配方案后把属于每个consumer的方案单独抽取出来作为SyncGroup请求的response返还给各自的consumer。 |
|  | |

consumer group分配方案是在consumer端执行的，Kafka将这个权力下放给客户端主要是因为这样做可以有更好的灵活性。比如在这种机制下用户可以自行实现类似于Hadoop那样的机架感知（rack-aware）分配方案。同一个机架上的分区数据被分配给相同机架上的consumer以减少网络传输的开销。而且，即使以后分区策略发生了变更，也只需要重启consumer应用即可，不必重启Kafka服务器。

### 5.6.7 rebanance监听器

## 5.7 解序列化

### 5.7.1 默认解序列化器

Kafka consumer从broker端获取消息的格式是字节数组，consumer需要把它还原回指定的对象类型。Kafka1.0.0版本默认提供了十几种的deserializer来支持解序列化为常用类型，常见的deserializer如下表所示。但若涉及复杂的类型（比如Avro或其他序列化框架），那么就需要用户自行定义deserializer。

consumer的序列化机制使用起来非常简单，只需要在构造consumer时同时指定参数key.deserializer和value.deserializer的值即可。

|  |  |
| --- | --- |
| 常见deserializer | |
| ByteArrayDeserializer | 本质上什么都不做，因为己然是字节数组。 |
| ByteBufferDeserializer | 解序列化成ByteBuffer |
| BytesDeserializer | Kafka自定义的Bytes类 |
| DoubleDeserializer | Double类型 |
| IntegerDeserializer | Integer类型 |
| LongDeserializer | Long类型 |
| StringDeserializer | string类型 |

### 5.7.2 自定义解序列化器

实现org.apache.kafka.common.serialization.Deserializer<T>接口即可。

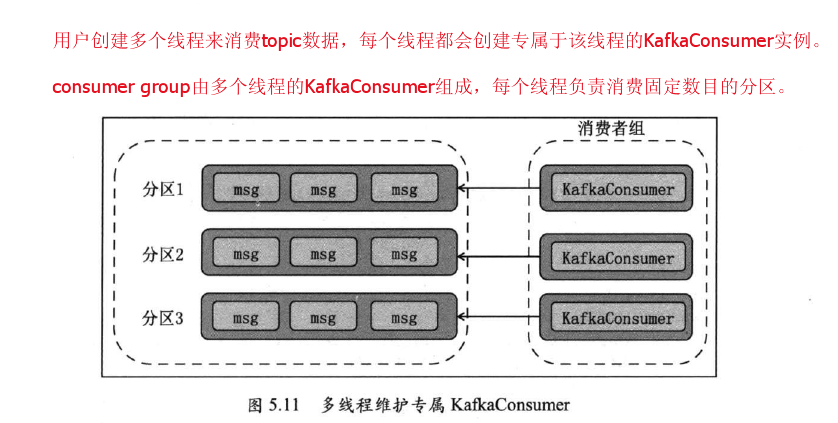
|  |
| --- |
| **public** **class** UserDeserilizer **implements** Deserializer<User> {  **private** ObjectMapper objMapper;  @Override **public** **void** configure(Map<String, ?> configs, **boolean** isKey) {  objMapper = **new** ObjectMapper();  }  @Override **public** User deserialize(String topic, **byte**[] data) {  User user = **null**;  **try** {  user = objMapper.readValue(data, User.**class**);  } **catch** (Exception e) {  e.printStackTrace();  }  **return** user;  }  @Override **public** **void** close() {}  } |

## 5.8 多线程消费实例

如前所述，KafkaConsumer是非线程安全的。它和Kafka Producer不同，后者是线程安全的，因此用户可以在多个线程中放心地使用同一个Kafka Producer实例，事实上这也是社区推荐的producer使用方法，因为通常它比每个线程维护一个KafkaProducer实例效率要高。

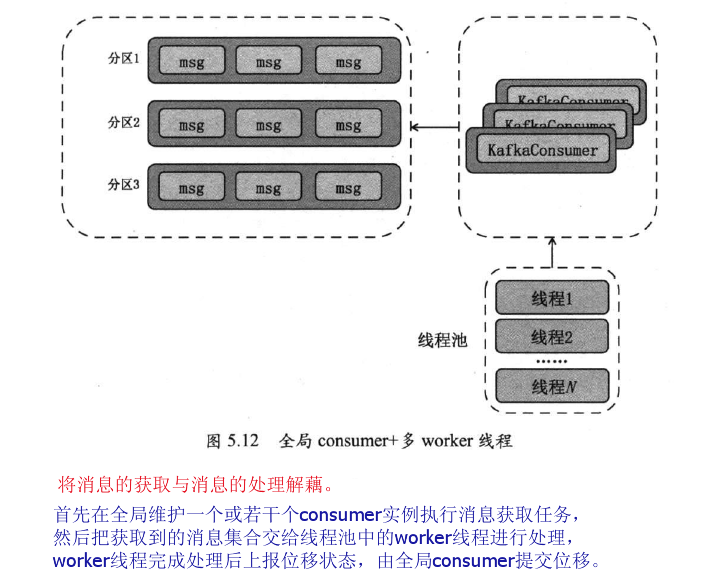
但是对于consumer而言，用户无法直接在多个线程中共享一个KafkaConsumer实例。那么应该如何实现多线程consumer消费呢？本节给出两种多线程消费的方法以及各自的实例。

### 5.8.1 每个线程维护一个KafkaConsumer



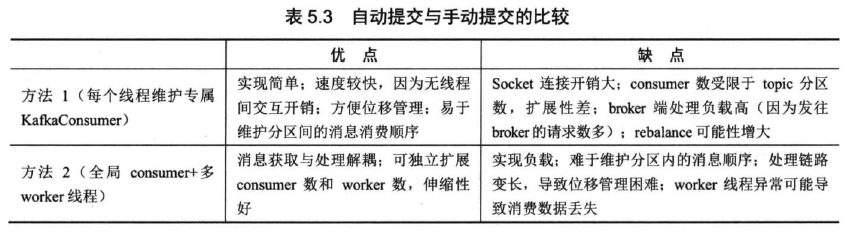
*程序代码见：org.liws.consumer.multithread.t1*

### 5.8.2 全局KafkaConsumer实例加多Worker线程



*程序代码见：org.liws.consumer.multithread.t2*

### 5.8.3 两种方法对比



## 5.9 独立消费者

目前为止我们讨论的consumer都是以consumer group的形式出现的。group自动帮用户执行分区分配和rebalance。对于需要有多个consumer共同读取某个topic的需求来说，使用group是非常方便的。

但有的时候用户依然有精确控制消费的需求，比如严格控制某个consumer固定地消费哪些分区。比如：

|  |
| --- |
| 1、如果进程自己维护分区的状态，那么它就可以固定消费某些分区而不用担心消费状态丢失的问题。  2、如果进程本身己经是高可用且能够自动重启恢复错误（比如使用YARN和Mesos等容器调度框架），那么它就不需要让Kafka来帮它完成错误检测和状态恢复。 |

以上两种情况中consumer group都是无用武之地的，取而代之的是被称为独立consumer(standaloneconsumer）的角色。standalone consumer间彼此独立工作互不干扰。任何一个consumer崩溃都不影响其他standaloneconsumer的工作。

使用standalone consumer的方法就是调用KafkaConsumer.assign方法。还记得吧，之前订阅topic我们使用的是KafkaConsumer.subscribe方法，而assign方法则接收一个分区列表，直接赋予该consumer访问这些分区的权力。

*程序代码见：org.liws.consumer.alone.T\_AloneConsumer*

如果发生多次assign调用，最后一次assign调用的分配生效，之前的都会被覆盖掉。

还有一个值得注意的是，assign和subscribe一定不要混用，即不能在一个consumer应用中同时使用consumer group和独立consumer。

## ~~5.10 旧版consumer~~

# 第六章 kafka设计原理

## 6.1 broker端设计

大多数消息队列框架都有broker或与之类似的角色。一个broker通常是以服务器的形式出现的，对用户而言，broker主要用于持久化消息以及将消息队列中的消息从发送端传输到消费端。

kafka broker本质上是一个服务载体，承载了绝大多数的kafka服务。 broker负责持久化producer端发送的消息，同时还为consumer端提供消息。

### 6.1.1 消息设计

#### 1 消息格式——使用什么数据结构来保存消息和消息队列？

*使用java类定义kafka消息的弊端？*

#### 2 版本变迁

### 6.1.2 集群管理

### 6.1.3 副本与ISR设计

#### 6.1.4 水印（watermark）和leader epoch

### 6.1.5 日志存储系统

### 6.1.6 通信协议

### 6.1.7 controller设计

### 6.1.8 broker请求处理

## 6.2 producer端设计

### 6.2.1 producer端基本数据结构

#### RecordMetadata

RecordMetadata表示broker端返回给客户端的消息的元数据信息。

|  |  |
| --- | --- |
| TopicPartition topicPartition | 消息所属topic及分区 |
| long offset | 消息在分区日志中的位移 |
| long timestamp | 消息时间戳 |
| int serializedKeySize | 消息序列化后的key字节数 |
| int serializedValueSize | 消息序列化后的value字节数 |
| Long checksum | 消息CRC32码 |

上面的元数据信息的前3项信息是比较重要的，producer端可以使用这些信息做一些消息发送成功之后的处理，比如写入日志等。

### 6.2.2 producer端工作流程【复杂】

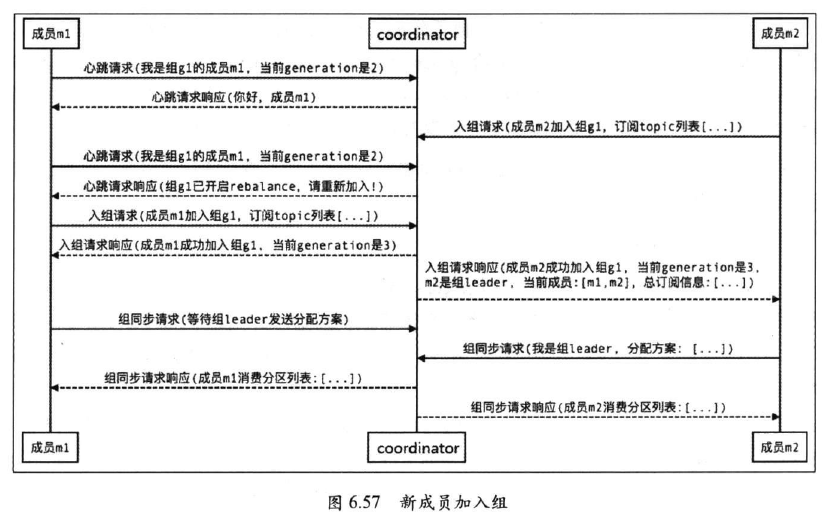
## 6.3 consumer端设计

### 6.3.1 consumer group状态机

### 6.3.2 group管理协议

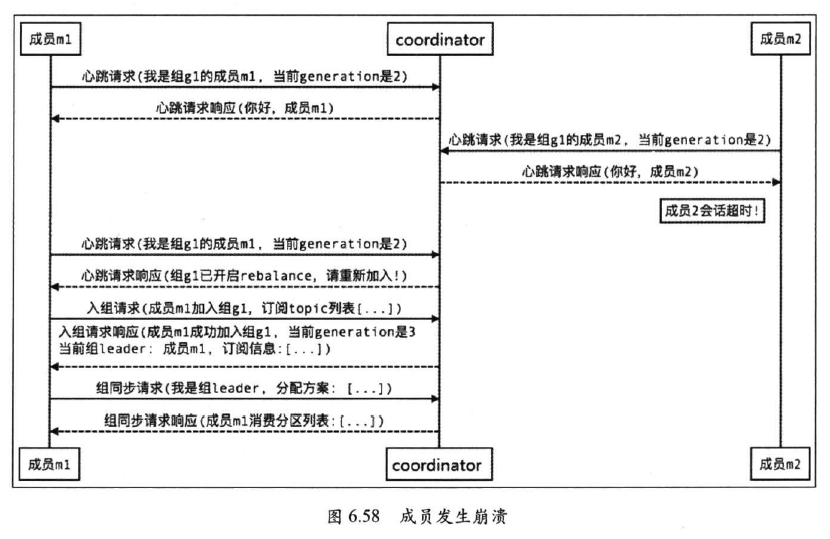
### 6.3.3 rebanance场景剖析

#### 场景1：新成员加入组

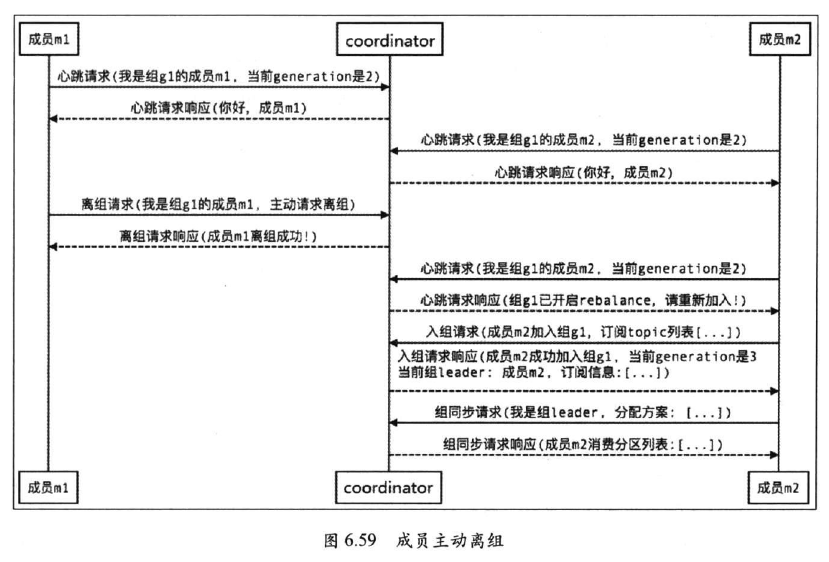


#### 场景2：成员发生崩溃

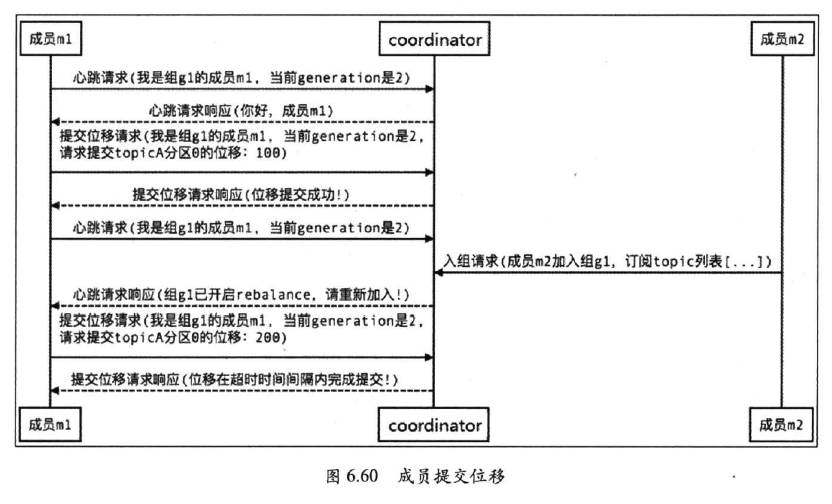
组成员崩溃和组成员主动离开是两种不同的场景。因为在崩溃时成员并不会主动告知coordinator此事，coordinator有可能需要一个完整的session.timeout周期才检测到这一崩溃，这必然会造成consumer的滞后。可以说离开组是主动地发起rebalance，而崩溃则是被动地发起rebalance。



#### 场景3：成员主动离组



#### 场景4：成员提交位移



## 6.4 实现精确一次处理语义EOS（exactly-once semantics）

### 6.4.1消息交付语义（message delivery semantic）

*clients端常见的3种消息交付语义：*

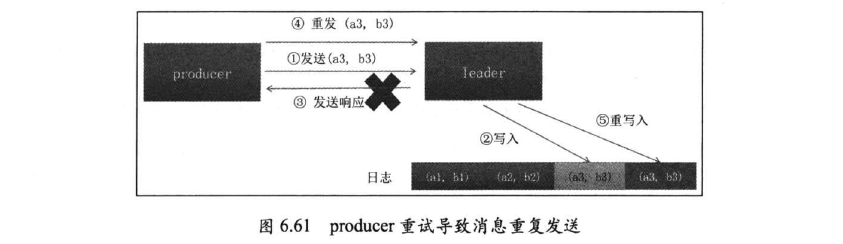
|  |  |
| --- | --- |
| 最多一次（at-most-once）处理语义 | 消息可能丢失，但不会被重复处理。 |
| 最少一次（at-least-once）处理语义 | 消息不会丢失，但可能被处理多次。 |
| 精确一次（exactly-once）处理语义 | 消息一定会被处理且只会被处理一次。 |

先明确一个观点：这3种消息交付语义没有好坏之分。不是说at-most-once就是不好的，而exactly-once总是对的。具体使用哪种语义要结合用户实际的业务需求来定，没有必要过分追求精确一次语义。一个典型的例子就是统计网页的PV(page view）和UV(user view）。在这种场景下，结果的不完全准确通常并不会影响我们利用它们所做的决策，所以没有必要引入很复杂的数据结构或功能来实现EOS。

#### kafka producer端消息交付语义

对producer而言，Kafka引入己提交消息（comitted message）的概念。一旦消息被成功地提交到日志文件，只要至少存在一个可用的包含该消息的副本，那么这条消息就永远不会丢失。由此可见Kafka producer提供的不是at-most-once语义，但它是at-least-once还是exactly-once呢？

在0.11.0.0版本之前，Kafka producer默认提供的是at-Ieast-once语义。设想这样的一个场景，当producer向broker发送新消息后，分区Ieader副本所在的broker成功地将该消息写入本地磁盘，然后发送响应给Producer。此时假设网络出现故障导致该响应没有发送成功，那么未接到响应的producer会认为该消息请求失败从而开启重试操作。若重试后网络恢复正常，那么显然同一条消息被写入到日志两次。在比较极端的情况下，同一条消息可能会被发送多次。具体流程如下图所示。



好消息是Kafka0.11.0.0版本推出了幕等性producer和对事务的支持，从而完美地解决了这种消息重复发送的问题。我们会在后面详细探讨具体的设计细节。

#### kafka consumer端消息交付语义

对consumer端而言，我们知道，相同日志下所有的副本都应该有相同的内容以及相同的当前位移值。consumer通过consumer位移自行控制和标记日志读取的进度。如果consumer程序崩溃，那么替代它的新程序实例就必须要接管这个consumer位移，即从崩溃时读取位置继续开始消费。若要判断consumer到底支持什么交付语义，位移提交的时机就显得至关重要。

一种方式是consumer首先获取若干消息，然后提交位移，之后再开始处理消息。这种方法下若consumer在提交位移后处理消息前崩溃，那么它实现的就是at-most-once语义，因为消息有可能不被处理，就算处理了最多也只会是一次。

另一种方式是consumer获取了若干消息，处理到最后提交位移。显然，consumer保证只有在消息被处理完成后才提交位移，因此它实现的就是at-least-once语义，因为消息处理过程中如果出现错误从而引发重试，那么某些消息就可能被处理多次。

那么如何实现consumer端的EOS呢？主要是依赖0.11.0.0版本引入的事务。后面我们会详细展开。

### 6.4.2 幂等性producer

幂等性producer是Apache Kafka0.11.0.0版本用于实现EOS的第一个利器。

若一个操作执行多次的结果与只运行一次的结果是相同的，那么我们称该操作为***幂等操作***。比如将变量a的值设置成1这个操作就是幂等操作，无论该操作执行多少次，a的值始终是1，与只运行一次该操作的效果是相同的。反之，将当前金额加100这样的操作就不是幕等的，因为该操作依赖于之前的金额，即操作有前置状态。

0.11.0.0版本引入的幂等性producer表示它的发送操作是幕等的。瞬时的发送错误可能导致producer端出现重试，同一条消息被producer发送多次，但在broker端这条消息只会被写入日志一次．对于单个topic分区而言，这种Producer提供的幂等性消除了各种错误导致的重复消息。如果要启用幂等性producer以及获取其提供的EOS语义，用户需要显式地设置producer端的新参数enable.idempotence为true。

幂等性producer的设计思路类似于TCP的工作方式。发送到broker端的每批消息都会被赋予一个序列号（sequence number）用于消息去重。但是和TCP不同的是，这个序列号不会被丢弃，相反Kafka会把它们保存在底层日志中，这样即使分区的leader副本挂掉，新选出来的leader broker也能执行消息去重工作。保存序列号只需要额外几字节，因此整体上对Kafka消息保存开销的影响并不大。

除了序列号，Kafka还会为每个producer实例分配一个producer id（下称PID）。producer在初始化时必须分配一个PID。PID分配的过程对用户来说是完全透明的，因此不会为用户所见。消息要被发送到的每个分区都有对应的序列号值，它们总是从0开始并且严格单调增加。对于PID、分区和序列号的关系，用户可以设想一个Map,key就是（PID,分区号），value就是序列号，即每对（PID,分区号）都有对应的序列号值。若发送消息的序列号小于或等于broker端保存的序列号，那么broker会拒绝这条消息的写入操作。

这种设计确保了即使出现重试操作，每条消息也只会被保存在日志中一次。不过，由于每个新的producer实例都会被分配不同的PID，当前设计只能保证单个producer实例的EOS语义，而无法实现多个producer实例一起提供EOS语义。这一点要特别注意。

引入了序列号和PID之后，producer应付重复发送的流程如下图所示。



当producer端重发消息时，由于broker端己经成功写入了该消息，因此消息所在的seq（序列号）值不大于broker端当前保存的最大seq值，因此broker将拒绝该PRODUCE请求，从而成功实现了消息去重。

### 6.4.3 事务

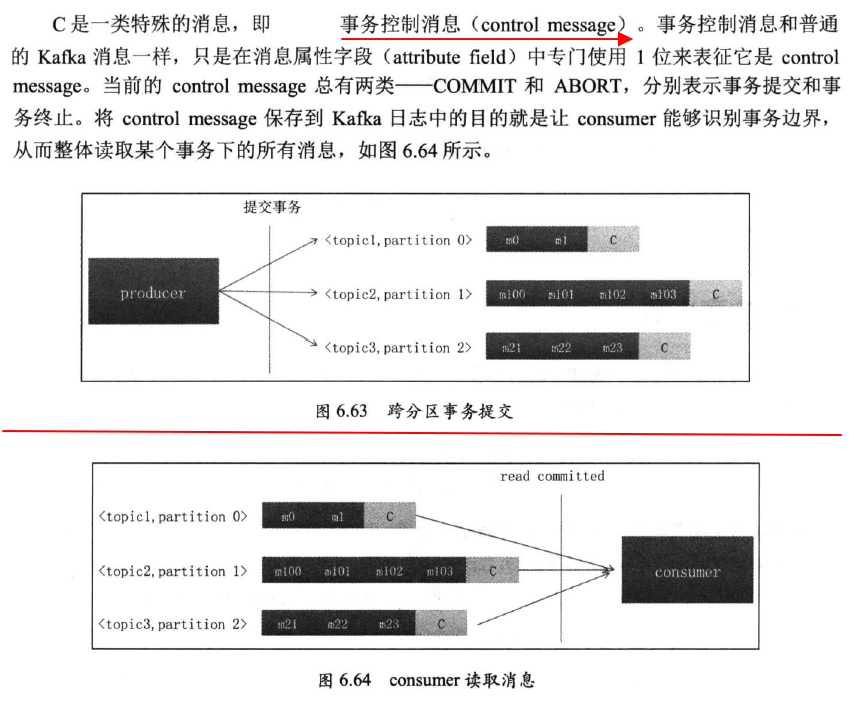
对事务的支持是Kafka实现EOS的第二个利器。

引入事务使得clients端程序（无论是Producer还是consumer）能够将一组消息放入一个原子性单元中统一处理。处于事务中的这组消息能够从多个分区中消费，也可以发送到多个分区中去。重要的是不论是发送还是消费，Kafka都能保证它们是原子性的，即所有的写入操作要么全部成功，要么全部失败。当然对于consumer而言，EOS语义的支持要弱一些，这是由consumer本身的特性决定的。也就是说，consumer有可能以原子性的方式消费这批消息，也有可能是非原子性的。设想consumer总是需要replay某些消息，如果是这样的使用场景，那么对于EOS的支持就要弱很多。

Kafka为实现事务要求应用程序必须提供一个唯一的id来表征事务。这个id被称为事务id,或TransactionalId，它必须在应用程序所有的会话上是唯一的。值得注意的是，TransactionalId与上面所说的PID是不同的，前者是由用户显式提供的，而后者是prodcuer自行分配的。

|  |
| --- |
| *当提供了TransactionalId后，Kafka就能确保：* |
| 1、跨应用程序会话间的幂等发送语义。具体的做法与新版本consumer的generation概念类似，使用具有版本含义的generation来隔离旧事务的操作。  2、支持跨会话间的事务恢复。如果某个producer实例挂掉了，Kafka能够保证下一个实例首先完成之前未完成的事务，从而总是保证状态的一致性。 |
| *如果以consumer的角度而言，如前所述，事务的支持要弱一些，原因如下:* |
| 1、对于compacted的topic而言，事务中的消息可能己经被删除了。  2、事务可能跨多个日志段（log segment)，因此若老的日志段被删除，用户将丢失事务中的部分消息。  3、consumer程序可能使用seek方法定位事务中的任意位置，也可能造成部分消息的丢失。  4、consumer可能选择不消费事务中的所有消息，即无法保证读取事务的全部消息。 |

下面分别讨论一下事务是如何在producer和consumer端实现的。图6.63给出了原子性写入多个分区的流程。



*典型的事务API使用范例：*

