**Kafka**

# 消息队列

## 1.1.消息队列的优点

1. 解耦

传统模式中系统间的耦合性太强，消息队列允许独立修改或扩展两边的处理过程，只要确保它们遵守同样的接口约束

1. 异步

传统模式中，一些非必要的业务逻辑以同步方式运行，耗时，消息队列提供了异步处理机制，允许用户把一个消息放入队列，异步处理非必要的业务逻辑

1. 削峰

并发量大的时候，所有请求直接涌向数据库，造成数据库连接异常，消息队列能够按照数据库处理并发量，从消息队列中慢慢拉取消息。

1. 冗余

消息队列把数据持久化直到他们已经被完全处理

1. 顺序保证

消息队列本身就是排序的，并且能保证数据按照特定顺序处理

1. 缓冲

有助于控制和优化数据流经系统的速度，解决生产者和消费者处理速度不一致的情况

## 1.2.消息队列的缺点

1. 系统可用性降低

当消息队列宕机，系统不可用

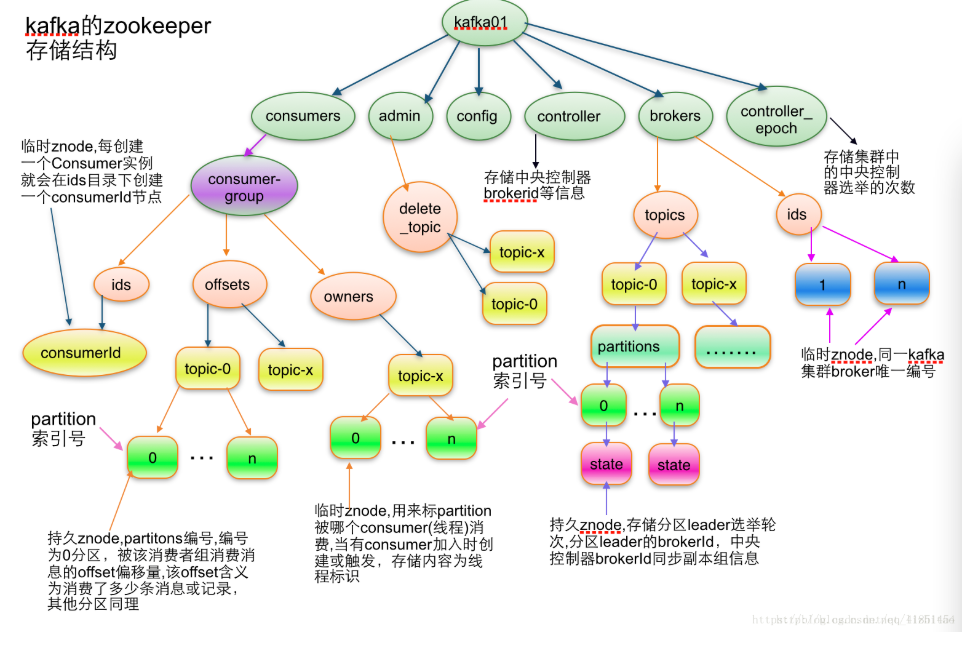
1. 系统复杂性增加

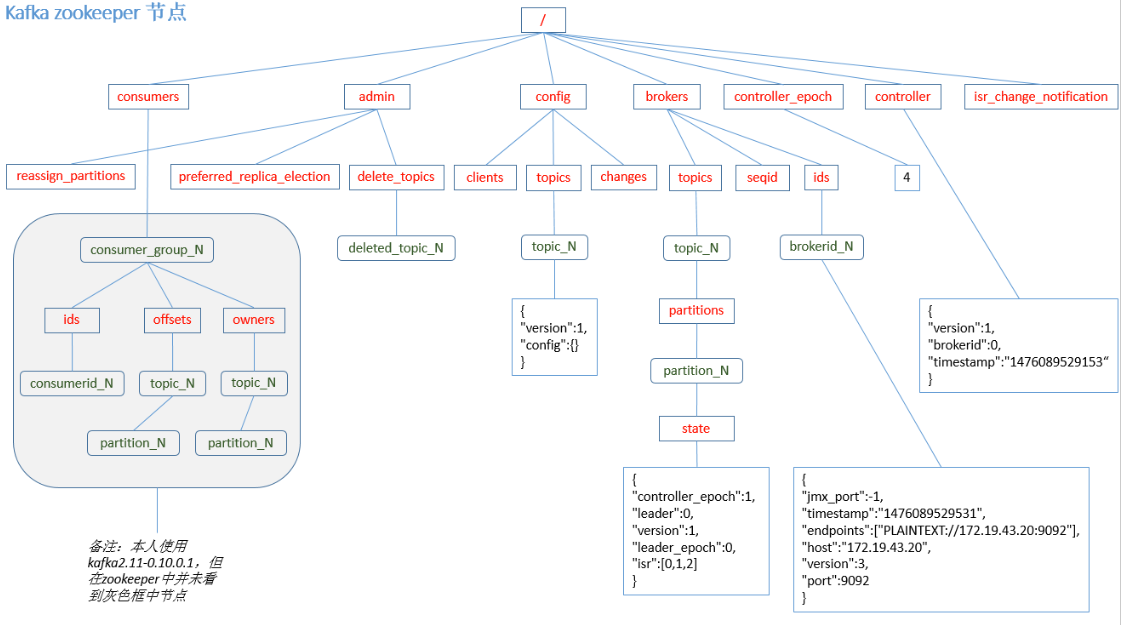
需要考虑一致性问题、如何保证消息不被重复消费、如何保证消息可靠传输

## 1.3.现有消息队列

* Kafka
* Redis
* activeMQ
* RocketMQ
* ZeroMQ

# Kafka





## 2.1.基本概念

* broker：消息中间件处理节点，一个kafka节点就是一个broker，多个broker可以组成一个kafka集群
* controller：集群中的某个broker被选举为controller，负责管理集群所有分区和副本状态
* topic：一类消息，例如page view日志、click日志等都可以以topic形式存在，kafka集群能够同时负责多个topic的分发
* partition：topic物理上的分组，一个topic分为多个partition，每个partition都是一个有序的队列
* segment：partition物理上由多个segment组成
* offset：每个partition都由一系列有序、不可变的消息组成，这些消息被追加到partition中，每个partition中每个消息都有一个连续的序列号---offset，用于partition唯一标识一条消息。另外，zookeeper中/consumers/consumergroup\_id/offset存储了每个topic的partition消费进度

## 2.2.topic结构

* 每个topic对应一个或多个partition，partition会被均匀分配到不同的broker上
* 每个partition对应多个副本（一般为3），分布在不同的broker上，由zookeeper选举出leader，读写操作都是由leader负责，其他follower负责备份
* 每个partition在在磁盘上是一个文件夹（topic+partition+编号）
* 一个partition由多个segment组成
* 每个segment由两个文件组成（.index文件和.log文件）
* index文件并没有为每一条message建立索引，而是采用了稀疏索引，避免了索引文件占用过多空间，从而将索引文件留在内存中
* 为什么要将一个topic分为多个partition？

若没有分区，当topic A的消息集很多，存储topic A的broker节点可能读写压力会比较大，分区可达到并发访问，增大吞吐量的效果。

* 为什么要将一个partition分为多个segment

若一个partition文件很大，读取消息时很耗时，通过分段，并且每个segment都有对应的稀疏索引文件，可以通过二分查找快速定位到指定的segment，然后在segment的index文件中二分查找定位要找的消息大体位置，继续顺序查找得到结果，提高读写操作

## 2.3.消息发送机制

### 2.3.1.发送模式

Producer端通过配置参数producer.type设置，这个参数指定了后台线程的消息发送方式是同步还是异步，producer.type=sync为默认的同步模式，如果设置为async，producer以batch形式push数据，可以极大提高broker的性能，但会增加丢失数据的风险。

### 2.3.2.消息路由

Producer采用push模式将消息发布到broker，每条消息被append到partition中，当producer往topic中发送数据时，数据会被hash到不同的partition。

1. producer指定了partition
2. 未指定partition，指定了key，可以通过key的值进行hash，选出一个partition
3. Partition和key都未指定，使用轮询选出一个partition

### 2.3.3.写入流程

1. producer先从zookeeper的/brokers/topics/topic-i/partitions/partition-j/state节点找到该partition的leader，
2. producer将消息发送给该leader
3. leader将消息写入本地log
4. follower从leader pull消息，写入本地log后发送Ack（leader有新消息会通知follower取新消息）
5. leader收到ISR中所有replicas的ACK后，增加HW并向producer发送ACk

### 2.3.4.复制原理和同步方式

* AR：assigned replicas，每个partition有多个副本replicas（分为leader和follower），放置这些副本的brokers集合
* ISR：In-sync replicas，AR中与leader同步延迟不超过指定时间的brokers集合，由leader节点维护
* OSR：Outof-sync replicas，当AR中被剔除ISR中的节点将被放入OSR，新加入的follower也会存放在OSR，AR=ISR+OSR
* Replica.lag.time.max.ms：作为ISR副本管理的参数，当某个follower与leader同步数据延时超过这个时间，就会被剔除ISR而放入OSR
* LEO：logEndOffset，表示partition的log最后一条message的位置
* HW：HighWatermark，去partition对应ISR中最小的LEO作为HW，consumer最多能消费到HW所在位置。即当一个消息被写入leader中，不能被立即消费，直到ISR中replicas同步更新HW，此时消息才能被consumer消费，保证了leader所在broker失效，该消息仍然可以从新选举的leader中获取。

Kafka的ISR管理最终都会反馈到zookeeper节点上，具体位置为/brokers/topics/topic-i/partitions/partition-j/state，有两个地方会对zookeeper这个节点进行维护：

* Controller维护：kafka集群中一个broker会被选举为controller，负责partition管理和副本状态管理，在某些条件下，controller会选举新的leader、ISR和新的leader\_epoch及controller\_epoch写入zookeeper节点
* Leader维护：leader有单独的线程定期检测ISR中follower是否脱离ISR，则会将新的ISR信息返回到Zookeeper的节点中

## 2.4.数据可靠性和持久性保证

当producer向leader发送数据，可以通过request.required.acks参数设置可靠性级别：

* 1：表示producer在ISR中的leader成功收到数据并得到确认，若leader宕机，丢失数据
* 0：表示producer无需等待来自broker的确认继续发送下一批消息，传输效率最高，但数据的可靠性是最低
* -1：producer需要等待ISR中所有follower都确认接收到数据后才算一次发送完成，可靠性最高

Min.insync.replicas这个参数设置ISR中最少副本数，默认值为1，当且仅当request.required.acks=-1时此参数才有效，当ISR中副本数少于min.insync.replicas的值时，将返回客户端异常。

## 2.5.leader选举

每个partition都有多个副本，client通过leader将消息写入partition中，leader将通知follower有新数据写入，follower从leader中pull新消息，当设置request.required.acks=-1时，只有ISR中全部follower复制成功才会返回给client写入成功的ack，leader在其中起到了至关重要的作用。

当leader宕机后，必须在follower中选出新的leader，因为follower可能落后很多或是crash，所以必须选择有最新消息的follower作为leader，即新的leader拥有原来leader commit的所有消息。

Kafka在zookeeper中为每个partition动态维护了一个ISR，这个ISR中所有replicas都跟上了leader，只有ISR中的成员才有被选为leader的可能。

ISR中至少有一个follower时，Kafka可以确保commit的数据不会丢失，如果一个partition的所有replicas都挂了，就无法保证数据不丢失，这时候有两种选择方案：

* 等待ISR中任意一个replicas活过来，选择他作为leader，可能等待时间较长
* 选择第一个活过来的replicas作为leader，可能活过来的replicas不是ISR中follower即可能丢失commit的数据

默认情况下，Kafka选择第二种策略，用unclean.leader.election.enable=true参数控制。

## 2.6.高可靠性分析

Kafka如何确保消息在producer和consumer之间传输，有三种可能：

* 1. At most once：消息可能丢失，但不会重复传输
  2. At least once：消息不会丢失，可能会重复传输
  3. Exactly once：每条消息肯定会被传输一次且仅仅一次

当producer发送给broker后网络中断，producer无法判断是否提交，但producer可以retry多次，确保消息正确传输到broker中，所以目前Kafka实现的是at least once。

### 2.6.1.消息去重

Kafka在producer和consumer端都会出现消息重复，这就需要去重处理。

Kafka文档中提及GUID，通过client生成算法得到每个消息的unique id，同时映射至broker上存储地址，通过GUID便可查询提取消息内容，便于发送方的幂等性保证，需要在broker上提供去重模块。

### 2.6.2.高可靠性配置

1. topic配置：replications.factor>=3，副本数最少为3，2<=min.insync.replicas<=relications.factor
2. broke配置：leader的选举条件unclean.leader.eletcion.enable=false，表示只从ISR中选举leader
3. producer配置：request.required.acks=-1，表示所有ISR回复leader才会想producer回复ack，producer.type=sync表示同步消息

## 2.7.消息消费机制

### 2.7.1.consumer API

Kafka提供了两套Consumer API：1.the high-level consumer API， 2. The simple consumer API。

#### 2.7.1.1.the high-level consumer API

提供了consumer group语义，一个消息只能被一个group内一个consumer消费，且consumer消费时不关注offset，最后一个offset由zookeeper保存。

* 如果consumer大于partition数量，则有的consumer将收不到消息
* 如果partition数量大于consumer，有的consumer将收到多个partition消息
* 如果一个consumer消费多个partition，无法保证收到的消息顺序，但一个partition内的消息是有序的

#### 2.7.1.2.the simple consumer API

如果想要对partition有更多的控制权，则使用simple consumer API

* 多次读取一个消息
* 只消费一个partition中的部分消息
* 使用事务保证一个消息仅被消费一次
* 需要在应用程序中跟踪offset，从而确定下一条应该消费哪条消息
* 需要获取每个partition的leader是谁
* 需要处理leader的变更

### 2.7.2.consumer group

每个consumer属于一个consumer group，一个partition中的消息只会被group中的一个consumer消费，但多个group可以同时消费这个partition。

Kafka设计目标之一就是实现离线处理和在线处理，可以使用spark/storm这类实时处理系统对消息在线处理，同时使用hadoop批处理系统进行离线处理，还可以将数据备份到另一个数据中心。

### 2.7.3.consumer delivery guarantee

如果将consumer设置为autocommit，consumer一旦读取数据自动commit，在这一读取消息过程中，Kafka保证了exactly once。

* 读完消息先commit，然后处理消息

如果consumer在commit后没有来得及处理就crash，下次重新开始后无法读到刚刚提交而未处理的消息，对应于at most once

* 读完消息后先处理再commit

如果处理完消息后crash而未commit，下次重新工作后会处理刚刚未commit的消息，对应at least once

* 如果要做到exactly once，需要协调offset和实际操作的输出

可以利用两阶段提交，可以让offset和实际操作输出存在同一个地方，但很多系统不支持两阶段提交

### 2.7.4.consumer rebalance

当有consumer加入或退出，以及partition变化（broker加入或退出）时会触发rebalance，算法如下：

* 将目标topic下所有partition排序，存于PT
* 对consumer group中所有consumer排序，存于CG，第i个consumer记为Ci
* N=size(PT)/size(CG)，向上取整
* 解决Ci对原来分配partition的消费权
* 将i\*N到（i+1）\*N – 1的partition分配给Ci

## 2.8.controller

Kafka集群中会有一个或多个broker，其中有一个broker被选举为Kafka controller，它负责管理整个集群中所有分区和副本状态。

* 当某个partition分区的leader发生故障，由controller负责为该分区选举新的leader
* 当检测到某个分区的ISR集合发生变化，由controller负责通知所有broker更新其元数据metadata
* 当为某个topic增加分区数量时，由controller负责分区的重新分配

在任意时刻，集群中只有一个controller，每个broker启动都会尝试去读取/controller临时节点下的brokerID值

1. 如果不存在/controller这个节点，会尝试创建这个节点，多个broker同时尝试创建/controller，只有成功的broker才会成为controller
2. brokerID != -1，表示其他节点竞选成功，当前broker放弃竞选
3. brokerID = -1，表示某个节点放弃了controller身份，需要重新选举

其中/controller\_epoch持久节点表示当前controller所在任期，每次变更，其值加一。每个和controller交互的请求都会携带上controller\_epoch这个字段

1. 请求中的controller\_epoch < 内存中的controller\_epoch，请求过期，无效请求
2. 请求中的controller\_epoch > 内存中的controller\_epoch,说明已经有新的controller当选

### 2.8.1.controller作用

具备控制器身份的broker比普通broker多一份职责：

1. 监听partition变化

* 为zookeeper中的/admin/reassign\_partition节点注册partitionReassignmentListener，用来处理分区重分配动作。
* 为zookeeper中的/isr\_change\_notification节点注册ISRChangeNotificationListener，用来处理ISR集合变更动作
* 为zookeeper中的/admin/preferred-replica-election节点添加preferredReplicaElectionListener，用来处理优先副本的选举动作

1. 监听topic变化

* 为zookeeper的/brokers/topics节点添加topicChangeListener，用来处理topic增减变化
* 为zookeeper中的/admin/delete\_topics节点添加TopicDeletionListener，用来处理删除topic动作

1. 监听broker变化

为zookeeper中/brokers/id/节点添加BrokerChangeListener，用来处理broker增减变化

1. 从zookeeper中读取获取当前所有与topic、partition以及broker有关的信息进行管理，对于所有topic对应的zookeeper中的/brokers/topics节点添加PartitionModificationsListener，用来监听topic中的分区分配变化
2. 更新集群元数据信息
3. 如果参数auto.leader.rebalance.enable设置为true，还会开启一个名为“auto-leader-rebalance-task”的定时任务来负责维护分区的优先副本均衡

### 2.8.2.controller

1. updateMetadataRequest：更新元数据请求，topic分区状态经常发生变化（leader重新选举或副本集合变化），当前client（producer和consumer）只能和分区的leader broker进行交互，一旦发生变更，controller会将最新的元数据广播给所有存活的broker，具体的方式就是向所有broker发送updateMetadataRequest请求
2. createTopic：创建topic请求，无论是通过API方式、脚本方式抑或是CreateTopic请求方式创建topic都是现在zookeeper的/brokers/topics下创建临时节点触发创建逻辑，而controller会监听该节点下的变更来执行真正的创建topic逻辑
3. deleteTopics：删除topic请求，也是通过创建zookeeper下/admin/delete\_topic/topic\_i节点来触发删除topic，controller执行真正删除逻辑
4. preferred leader分配：preferred leader选举有两种触发机制

* 自动触发：auto.leader.rebalance.enable=true
* Kafka-preferred-replica-election脚本触发

向zookeeper的/admin/preferred\_replica\_election写数据，controller读取数据执行

1. 分区重分配：kafka-reassign-partitions脚本，脚本写入/admin/reassign\_partitions节点来触发，controller负责按照方案执行
2. 分区扩展：即增加topic的分区数，

* 可以通过kafka-reassign-partitions脚本完成
* Client可以通过直接往zookeeper中写数据实现，比如直接把新增分区集合写入/brokers/topics/<topic>下，然后controller自动选择leader并增加分区

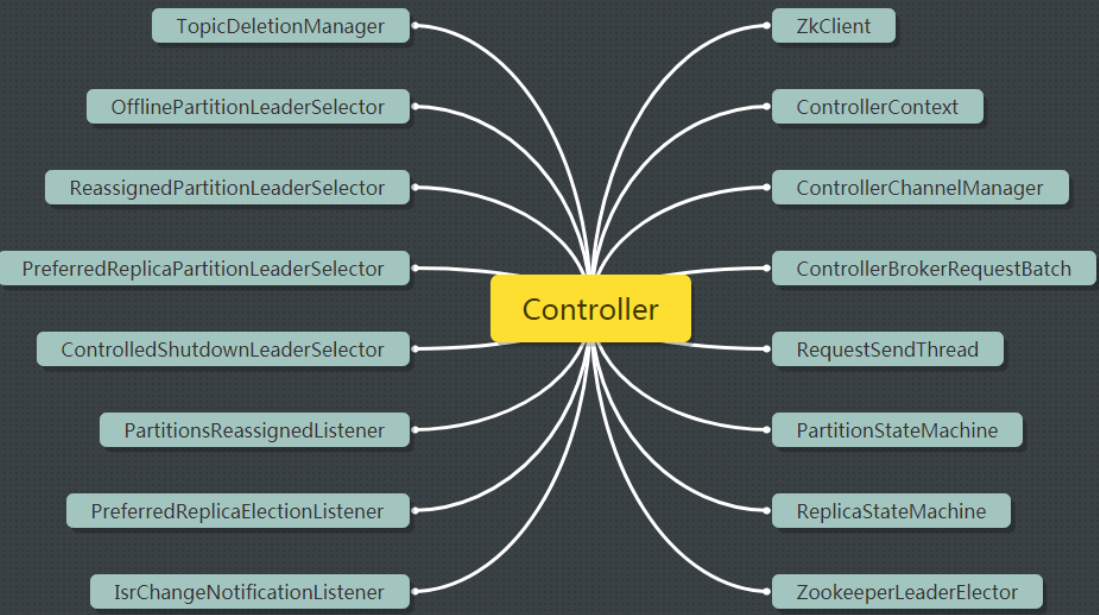
1. 集群扩展：新增broker时在zookeeper中/brokers/ids下新增临时节点，controller自动完成服务发现
2. Broker崩溃：controller通过zookeeper实时监测broker状态，一旦broker挂掉，controller感知后为受影响的partition选择新的leader
3. controlledShutdown：broker除了崩溃，还会优雅的退出，broker一旦自行终止，controller会接收到一个ControlledShutdownRequest请求，然后controller会妥善处理该请求并执行各种收尾工作

### 2.8.3.controller当前设计

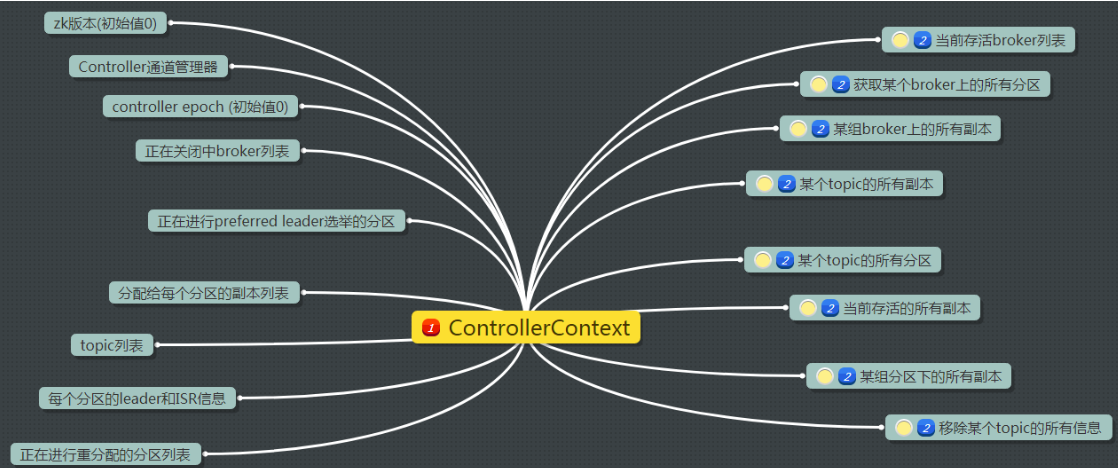
Controller启动时会为集群中所有broker各自创建一个连接，Kafka统一使用了NetworkClient类建模底层网络连接（主要依赖于java NIO的Selector），controller会为每个连接都创建一个对应的请求发送线程，专门负责给对应的broker发送请求，有三类请求：

1. updateMetaRequest：更新元数据
2. leaderAndISRRequest：创建分区、副本以及完成必要的leader或follower角色工作
3. stopReplicaRequest：停止副本请求，或删除分区副本

### 2.8.4.controller组成



ControllerContext：可以说是controller的缓存，当前controller为人诟病的原因之一就是使用大量同步机制保护ControllerContext，其结构如下：



## 2.9.负载均衡

### 2.9.1.producer负载均衡

由于同一个topic消息被分区到多个不同的broker上，所以producer需要将消息合理发送到这些分布式的broker上，Kafka支持传统的四层负载均衡，也支持zookeeper方式实现负载均衡。

* 四层负载均衡：主要是通过生产者的IP地址和端口，选择一个broker。这样一个producer对应一个broker，每个producer只要维护单个TCP连接，但无法真正做到负载均衡，因为每个producer产生的消息量和每个broker的存储能力都是不同的，会导致不同broker收到的消息总量差异巨大。
* 使用zookeeper进行负载均衡：每个broker启动时都会在zookeeper上完成注册过程，producer会通过监听/broker/ids动态感知broker列表的变化，实现动态的负载均衡机制。

### 2.9.2.consumer负载均衡

* Consumer注册到zookeeper的/consumers/consumer-group\_id/ids/consumer\_id，完成节点创建后，consumer将自己订阅的topic写入该临时节点。
* 对consumer group中consumer的变化注册监听，每个consumer都需要关注所属consumer group中其他consumer的变化，一旦发现consumer增加或减少就会触发consumer负载均衡
* 对broker注册监听，consumer监听broker列表变化，根据具体情况来决定是否需要进行consumer负载均衡
* 进行consumer负载均衡，为了让同一个topic下不同分区的消息尽可能均衡被多个consumer消费，而进行consumer和partition分配过程，通常对于一个consumer group，如果组内consumer变更或broker变更，均会进行负载均衡

# 3.kafka不提供读写分离

* Kafka的分区已经让读分散到多个broker上，而mysql提供主从复制是因为压力都集中一个主节点上
* Kafka的数据具有被消费的概念，是流数据，消费需要记录位移，如果从kafka的follower节点读，消费端offset控制更复杂
* 对于读多写少的场景（Redis，mysql）可以使用主从复制提升读性能，而kafka主要场景是消息引擎，通常涉及频繁 的生产消息和消费消息，因此读写分离方案在这个场景下不太适合
* Kafka是异步消息拉取，leader和follower之间存在不一致，如果实现读写分离，必然要解决副本落后引起的一致性问题

# 4.消息丢失和重复消费

## 4.1.避免消息丢失

消息可能丢失的情形：

* 生产者同步模式，request.required.acks=1，只保证leader成功，而leader宕机数据丢失
* 生产者异步模式，缓冲区满了，request.required.acks=0，还没有收到确认的情况下，缓冲区满后，会清空缓冲区的消息，数据会丢失
* 消费者丢失消息的情形有：如果在消息处理完成前就提交了offset，那么就有可能造成数据的丢失（kafka默认自动提交offset）

避免消息丢失有两个方面：

* 生产者数据不丢失

1. 如果是同步模式：request.required.acks=-1，保证所有ISR中的副本确认接收到消息才进行下一个消息的发送
2. 如果是异步模式：通过buffer控制数据的发送，设置block.on.buffer.full=true，在buffer满后阻塞等待

* 消费者端没有被消费的消息不会被清除，消费者没有遗漏消息

协调设置log.retention.bytes最大保留大小和log.retention.hours最大保留时间，配合消费者读取消息，可以通过读取和监控消费者的offset，保证消息不会被意外清除，设置自动提交offset为手动提交（enable.auto.commit=false）

## 4.2.避免重复消费

可能出现重复消费的根本原因是：消息已经被消费，但是offset没有提交，导致下次会重复消费offset的消息。

间接造成这样的原因有：consumer宕机前消费了消息，但没有来得及提交offset宕机，因为设置offset为自动提交，下次重启会重复消费。

可以通过设置offset手动提交。

## 4.3.消息乱序

Client一次发送两条消息到相同的分区：

Producer.send(record1);

Producer.send(record2);

如果此时由于某种原因导致record1没有成功发送，同时kafka又配置了重试机制和max.in.flight.requests.per.connection>1（该参数指定了生产者在收到服务器响应之前可以发送多少个消息，值越高，就会占用越多的内存，吞吐量也会提升，把它设置为1表示消息是按照发送的顺序写入服务器，即使发生重试），那么重试record1成功后，record1在分区中就在record2之后，从而造成消息的乱序。

* Block.on.buffer.full=true，已被标记为deprecated不推荐使用，它将使得producer一直等待缓冲区可用为止，阻塞等待
* acks=all，所有follower都响应了才认为消息提交成功
* retries=max，无限重试
* max.in.flight.requests.per.connection=1，表示kafka的broker在响应请求之前client不能向同一个broker发送请求，可以用于避免消息的乱序，使用kafkaProducer.send(record,callback)，自定义回调逻辑处理消息发送失败，比如记录在日志中，用定时脚本扫描重处理
* unclean.leader.election.enable=false，不允许非ISR中的副本被选举为leader，避免数据丢失
* replication.factor=3，3副本
* min.insync.replicas>1，消息至少要被写入这么多副本才算成功，也是提升数据持久性的一个参数
* replication.factor=min.insync.replicas+1，如果两者相等，一个副本宕机后无法正常工作

# 5.kafka高性能体现

* 顺序写磁盘，性能是随机写入的6000倍提升
* 内存零拷贝实现数据的高效传输，通过java类库的transferTo()方法，将字节从内核直接拷贝到socket中，而不需要经过用户态缓冲区，避免了数据拷贝在用户态和内核态之间的切换带来的开销
* 充分利用page cache
* IO调度器会将连续的小块写组装成大块的物理写从而提高性能
* IO调度器会尝试将一些写操作重新按顺序排好，从而减少磁盘头的移动时间
* 充分利用所有空闲的内存（非JVM内存），如果使用应用层的cache（JVM的堆内存），会增加GC负担
* 读操作可以直接在page cache中进行，如果消费和生产的速度相当，甚至不需要通过物理磁盘交换数据
* 如果进程重启，JVM的cache会失效，但page cache依然可用
* 数据压缩降低网络负载，kafka出支持单条消息的压缩传输外，还支持批量发送，将整个batch的消息一起压缩传输，数据压缩的基本原理是，重复数据越多压缩效果越好，broker收到消息后，并不直接解压，而是将消息以压缩的形式持久化到磁盘
* 批处理是一种常用的用于提高IO性能的方式，既减少了网络传输的overhead（网络协议信息），又提高了写磁盘的效率，kafka的producer API将同步与异步结合，虽然从send接口来看，一次只能发送一个producerRecord，而不能像之前的send一样接受消息列表，但是send方法并非立即将消息发送出去，而是通过batch.size和linger.ms控制实际发送频率，从而实现批量发送
* Kafka消息的key和value可以自定义，只需提供相应的序列化和反序列化器即可，因此用户可以通过使用快速且紧凑的序列化方式（Avro，protocal buffer）来较少实际网络传输和磁盘存储的数据规模，从而提高吞吐量