# Kafka

## 一、Kafka 基础

### 1、Kafka 高吞吐量实现机制

为了增加存储能力，Kafka 将所有消息都写入到低速大容量硬盘。Kafka 主要通过以下措施实现高吞吐量：

（1）顺序读写：Kafka 将消息顺序追加到 Partition 中，顺序读写要快于随机读写。

（2）Zero Copy：Kafka 生产者、消费者 API 对于 Kafka 消息采用零拷贝实现，内核直接将数据从磁盘文件拷贝到 Socket 套接字，而无需通过应用程序。

（3）批量发送：Kafka 通过消息集合批量发送消息。

（4）消息压缩：Kafka 允许对消息集合进行压缩。

（5）操作系统页缓存：不直接写 IO，直接写入页缓存；消费时大多命中缓存。

### 2、Kafka 优点

（1）解耦：Kafka 消息引擎系统将业务处理过程进行解耦，消息引擎两端的业务处理过程只需要实现接口即可。

（2）冗余：消息队列把消息进行持久化，直到消息被完全处理，进而规避数据丢失风险。

（3）扩展性：Kafka 消息队列对业务处理过程进行解耦，很容易增大消息入队和处理的频率，只要另外增加处理过程即可，不需要改变代码、调节参数。

（4）削峰填谷：削峰填谷是流量整形的形象表达，是为应对上游瞬时大流量冲击，避免出现流量毛刺，保护下游应用和数据库不被瞬时流量洪峰打垮。

（5）可恢复性：消息队列降低了进程间的耦合度，即使一个消息处理进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

（6）顺序保证：Kafka 保证一个 Partition 内消息的有序性。

（7）缓冲：消息队列通过一个缓冲层来帮助任务最高效率的执行，写入队列会尽可能快速，有助于控制和优化数据流经过系统的速度。

（8）异步通信：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，但并不立即处理，在需要时再处理。

## 3、Apache Kafka 主要版本

（1）0.9

Kafka 0.9.0 在 2014 年 11 月发布，0.9 版本中 Producer API 已经比较稳定，但 Consumer API 的 Bug 较多，主要特性如下：

A、增加基础安全认证/权限功能。

B、使用 Java 重写 Consumer API。

C、引入 Kafka Connect 用于实现高性能的数据抽取。

（2）0.10

Kafka 0.10 在 2016 年 5 月发布，引入 Kafka Streams，正式升级成分布式流处理平台。0.10.2.2 版本修复了一个可能导致 Producer 性能降低的 Bug，并且 Consumer API 已经比较稳定，其主要特性如下：

A、Kafka Streams

B、机架感知(Rack Awareness)

C、消息时间戳

D、SASL 改进

E、显示所有支持的 Connectors 和连接状态/控制的 REST API。

F、Kafka Consumer Max Records

G、协议版本改进。Kafka Broke 支持返回所有支持协议版本的请求 API，允许一个客户端支持多个 Broker 版本。

（3）0.11

Kafka 0.11.0 在 2017 年 6 月发布，支持 EOS，其主要特性如下：

A、修改 unclean.leader.election.enabled 默认值为 false，保证正确性。

B、确保 offsets.topic.replication.factor 参数被正确应用。

C、优化对 Snappy 压缩的支持。

D、消息增加头部信息(Header)。

E、空消费者组延时 Rebalance。

F、消息格式变更。

G、增加 StickyAssignor 粘性分区分配算法。

H、Controller 重构，采用单线程+基于事件队列的方式重构 Controller。

I、支持 EOS。Kafka 0.11.0 通过三大特性：幂等 Producer、支持事务、支持 EOS 流式处理(保证读-处理-写全链路的 EOS)实现对 EOS 的支持。

（4）1.0

Kafka 1.0 在 2017 年 11 月发布，主要优化 Kafka Streams API 以及完善各种监控指标，主要如下：

A、新增用于查看运行时活跃任务的 API。

B、增强 print()和 writeAsText()方法让调试变得更容易。

C、改进 Connect 度量指标，新增大量用于健康监测的度量指标。

D、支持 Java 9，实现更快的 TLS 和 CRC32C，加快加密速度，降低计算开销。

E、调整 SASL 认证模块的错误处理逻辑。

F、更好地支持磁盘容错，更优雅地处理磁盘错误，单个 JBOD 上的磁盘错误不会导致整个集群崩溃。

G、提升吞吐量。max.in.flight.requests.per.connection 参数最大可以被设置为 5，极大提升吞吐量范围。

（5）2.0

Kafka 2.0 在 2018 年 7 月发布，其主要特性如下：

A、增加前缀通配符访问控制(ACL)支持，实现细粒度访问控制；

B、更全面数据安全支持。

C、SSL 连接默认启用主机名验证，确保默认 SSL 配置不受中间人攻击影响。

D、可以在不重启 Broker 情况下动态更新 SSL 信任库。

E、改进复制协议，避免在 failover 期间 Leader 和 Follower 之间日志分歧。

F、保证在线升级方便性，简化 Kafka Streams 升级过程。

G、加强 Kafka 可监控性，增加很多系统静态属性以及动态健康指标。

H、放弃对 Java 7 支持，并移除 Scala 编写的 Producer API 和 Consumer API 代码。

## 4、Kafka 适用场景

（1）Kafka 作为存储系统

（2）Kafka 作为消息传递系统

（3）Kafka 用作流处理

## 5、其它消息队列

（1）RabbitMQ：RabbitMQ 是基于 Erlang 的开源企业级消息队列，支持 AMQP、XMPP、SMTP、STOMP。RabbitMQ 实现了 Broker 构架，消息在发送给客户端时先在中心队列排队，对路由、负载均衡或者数据持久化支持很好。

（2）Redis：Redis 是基于 Key-Value 对的 NoSQL 数据库，支持 MQ 功能，可以作为轻量级 MQ 使用。入队时，当数据比较小时 Re

dis 性能要高于 RabbitMQ，而如果数据大小超过 10K，Redis 较慢；出队时，无论数据大小，Redis 性能都很好，而 RabbitMQ 出队性能则远低于 Redis。

（3）ZeroMQ：ZeroMQ 能够实现 RabbitMQ 不擅长的复杂队列，但开发人员需要组合多种技术框架，技术复杂度较大并且只提供非持久性队列。ZeroMQ 非中间件模式不需要安装和运行消息服务器或中间件，应用程序会扮作服务器角色。

（4）ActiveMQ：ActiveMQ 能够以代理人和点对点的技术实现队列，少量代码就可以高效地实现高级应用场景。

（5）Kafka/Jafka：Kafka 是高性能跨语言分布式发布订阅消息队列系统，Jafka 是基于 Kafka 升级版。特性如下：

A、快速持久化，可以在 O(1)的系统开销下进行消息持久化；

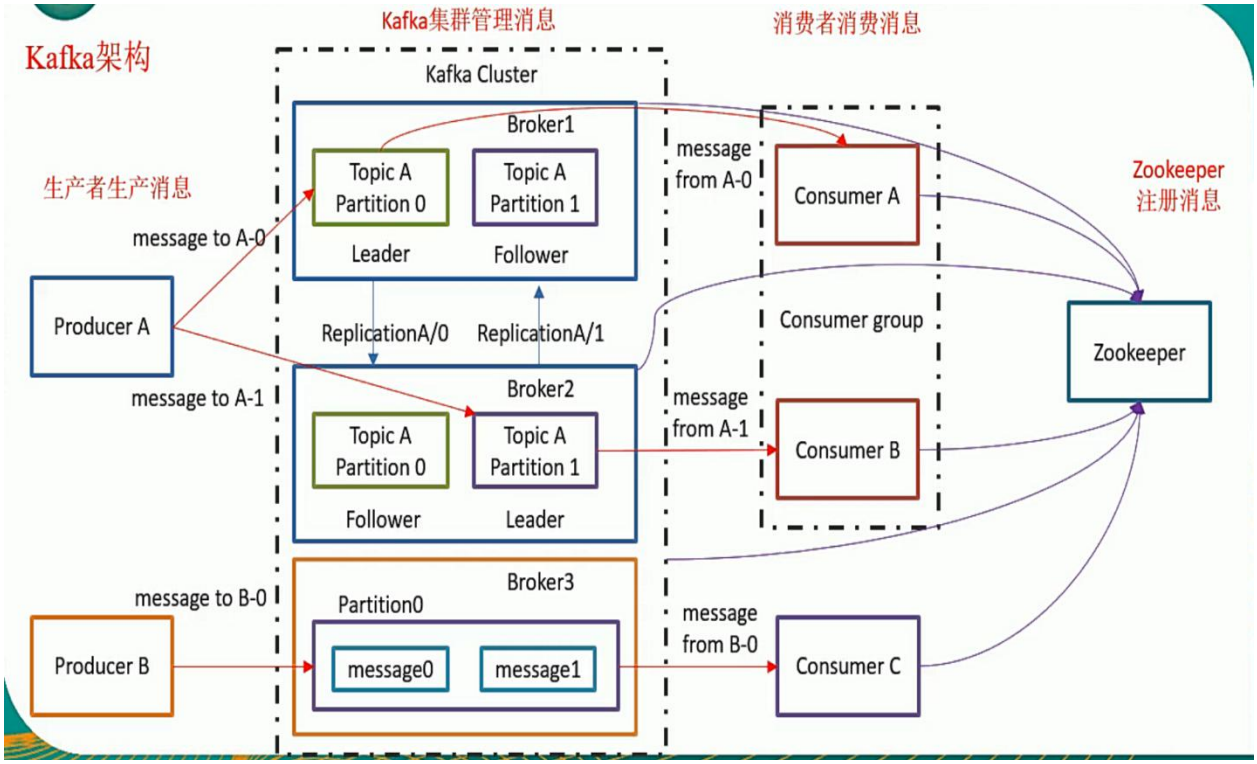
B、高吞吐量，在普通服务器可以达到 10W/s 吞吐速率；

C、完全分布式系统，Broker、Producer、Consumer 都原生自动支持分布式，自动实现负载均衡；

D、支持 Hadoop 数据并行加载，Kafka 通过 Hadoop 并行加载机制统一在线和离线消息处理。

（6）RocketMQ：Apache RocketMQ 是阿里开源的纯 Java 的分布式消息中间件，支持事务消息、顺序消息、批量消息、定时消息、消息回溯等。

## 二、Kafka 架构




（1）Record

Record 即 Kafka 消息，是 Kafka 处理的主要对象。

（2）Topic

Topic 是承载 Kafka 消息数据的逻辑容器，用于区分具体业务，但不同 Topic 的消息物理上分开存储，逻辑上一个 Topic 的消息虽然保存在一个或多个 Broker 上，但用户只需指定消息的 Topic 即可生产或消费数据而不必关心数据存储在何处。

（3）Partition

Partition 是一个物理概念，Topic 被分割为一个或多个 Partition，Partition 内部的消息是有序的，Partition 间的消息是无序的。

（4）Broker

Broker 是 Kafka 集群中的一个服务器节点，用于存储 Topic 数据。如果 Topic 有 N 个 Partition，集群有 N 个 Broker，那么每个 Broker 存储 Topic 的一个 Partition；如果 Topic 有 N 个 Partition，集群有(N+M)个 Broker，那么其中有 N 个 Broker 各存储 Topic 的一个 Partition，剩下的 M 个 Broker 不存储 Topic 的 Partition 数据；如果 Topic 有 N 个 Partition，集群中 Broker 数目少于 N 个，那么每个 Broker 存储 Topic 的一个或多个 Partition。生产环境中应尽量避免 Kafka 集群中 Broker 数量小于 Topic 分区数量，否则容易导致 Kafka 集群数据不均衡。

（5）Producer

Producer 是消息的发布者，负责选择将消息数据分配给 Topic 中的某个 Partition，即生产者生产的每一条消息会被写入到 Partition。

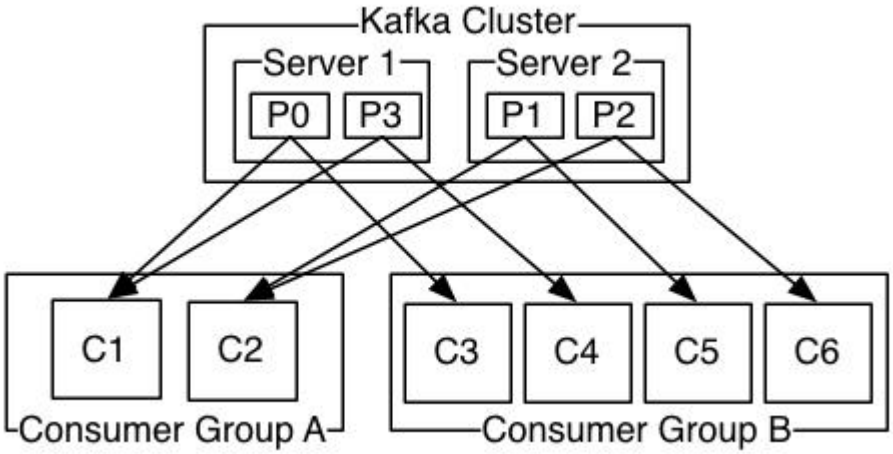
（6）Consumer

Consumer 从 Broker 中消费消息，一个 Consumer 可以消费多个 Topic 的消息，也可以消费同一个 Topic 中多个 Partition 中消息，Partition 允许多个 Consumer 同时消费，提高 Broker 的吞吐量。

（7）Consumer Group

Consumer Group 是 Kafka 提供的可扩展且具有容错性的消费者机制，多个 Consumer 组成一个 Consumer Group，同时消费多个 Partition 以实现高吞吐。Consumer Group 可以有多个 Consumer，共享一个 Group ID，Consumer Group 内的所有 Consumer 协调在一起来消费订阅 Topic 的所有分区。Kafka 保证同一个 C

onsumer Group 中只有一个 Consumer 会消费某条消息，Kafka 保证稳定状态下每一个 Consumer 只会消费某一个或多个特定的 Partition，而某个 Partition 的数据只会被某一个 Consumer 所消费。




如果所有 Consumer 都划分到同一个 Consumer Group 中，那么消息将会轮流被 Consumer Group 中的 Consumer 消费，即单播；如果所有 Consumer 都在不同 Consumer Group 中，那么每一条消息都会被所有的 Consumer 消费，即广播。多个 Consumer 可以组成一个 Consumer Group，Partition 中的每一条消息只能被一个 Consumer Group 中的一个 Consumer 进行消费，其它 Consumer 不能消费同一个 Topic 中同一个 Partition 的数据，不同 Consumer Group 的 Consumer 可以消费同一个 Topic 的同一个 Partition 的数据。

Replica 是一个 Partition 的备份，是为防止消息丢失而创建的 Partition 备份。Kafka 中同一条消息能够被拷贝到多个地方以提供数据冗余。副本分为 Leader 和 Follower，各自有不同的角色划分。副本是在 Partition 层级下的，即每个分区可配置多个副本实现高可用。

（8）Leader

每个 Partition 有多个副本，有且仅有一个作为 Leader，Leader 是当前负责消息读写的副本，所有读写操作只能发生于 Leader 上。

（9）Follower

Follower 需要从 Leader 同步消息，Follower 与 Leader 始终保持消息同步。Leader 与 Follower 是主备关系，而非主从关系。

（10）Zookeeper

Zookeeper 是一个分布式配置和同步服务，负责维护和协调 Kafka Broker 以及 Broker Controller 的选举。Kafka Broker 基本元数据例如主题、代理、消费者偏移(队列读取器)等信息存储在 Zookeeper 集群中，并通过 Zookeeper 集群在 Kafka Broker 间共享。

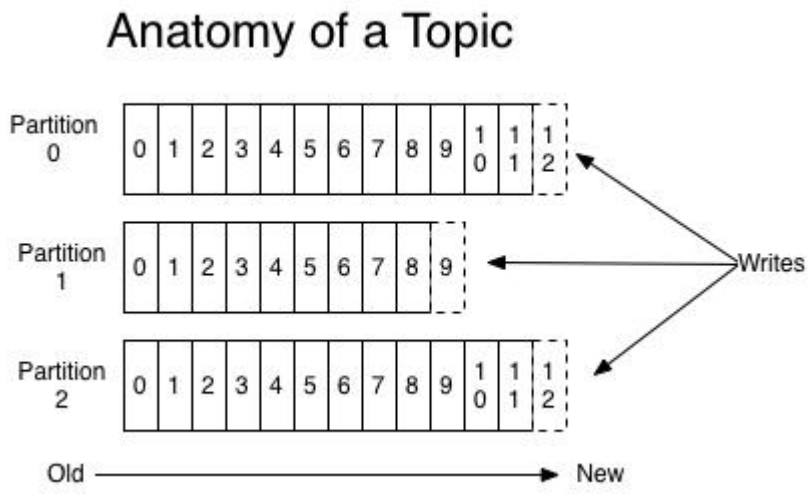
## 三、Kafka 分区机制

### 1、分区机制简介

分区机制是将每个 Topic 划分成多个 Partition，每个 Partition 是一组有序消息日志，生产者生产的消息只会被发送到一个 Partition 中。Partition 是一个有序的、不可变的数据序列，消息被不断追加到序列尾部。Partition 的每一条消息数据都被赋予一个连续的数字 ID，即偏移量（offset），用于唯一标识 Partition 中每条消息。

Partition 用于提供负载均衡能力，实现系统高伸缩性。不同 Partition 能够被放置到不同 Broker 上，而数据读写操作都是针对 Partition 进行，每个 Broker 都能独立地执行各自分区的读写请求处理，

同时可以通过添加新 Broker 来增加整体系统的吞吐量。




Offset（消息位移）是消息在 Partition 内的偏移量。每条消息都有一个当前 Partition 下唯一的 64 字节的消息位移，是相对于当前分区第一条消息的偏移量。

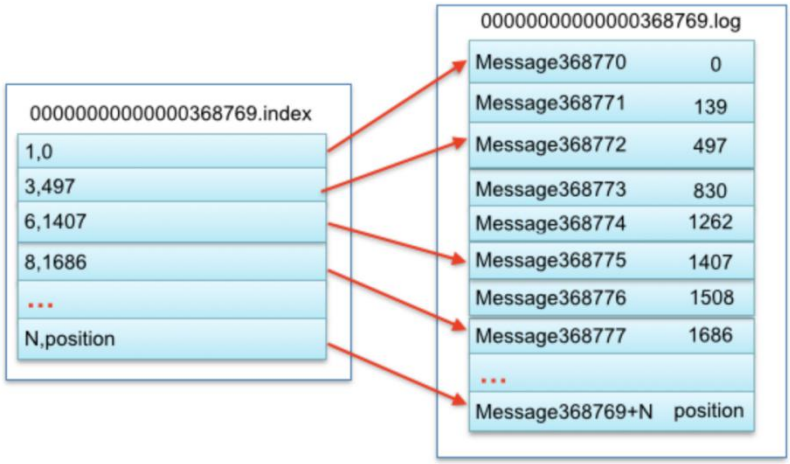
消息重试只是简单将消息重新发送到原来的 Partition，不会重新选择 Partition。

## 2、Log Segment 机制

Kafka 使用消息 Log 来保存数据，Log 是磁盘上一个只能追加写消息的物理文件。因为只能追加写入，因此避免了缓慢的随机 IO 操作，改为性能较好的顺序 IO 写操作。Kafka 通过 Log Segment 机制定期地删除消息以回收磁盘。Kafka 日志文件分为多个日志段，消息被追加写到当前最新日志段中，当写满一个日志段后，Kafka 会自动切分出一个新日志段，并将旧日志段封存。Kafka 在后台会定期地检查旧日志段是否能够被删除，从而实现回收磁盘空间的目的。

Kafka 将消息数据根据 Partition 进行存储，Partition 分为若干 Se

gment，每个 Segment 的大小相等。




index

segment的索引文件中存储着大量的元数据，数据文件中存储着大量消息，索引文件中的元数据指向对应数据文件中的message的物理偏移地址。以索引文件中的 `3，497` 为例，在数据文件中表示第3个message（在全局partition表示第368772个message），以及该消息的物理偏移地址为497。

Segment 由 index file 和 data file 组成，后缀为".index"和".log"，分别表示为 Segment 索引文件、数据文件，每一个 Segment 存储着多条信息。

Segment 文件的生命周期由 Broker 配置参数决定，默认 24x7 小时后删除。

依据 Kafka 消息在 Partition 的全局 offset，可以使用二分查找算法查找到相应 Segment 的.index 索引文件和.log 数据文件；根据索引文件找到消息在 Partition 的逻辑偏移量和物理地址偏移量，并取出消息数据。

## 3、分区策略

分区策略是决定生产者将消息发送到哪个 Partition 的算法。Kafka

提供默认分区策略，同时支持自定义分区策略。

Kafka 默认分区策略同时实现了两种策略：如果指定 Key，默认实现按消息键保序策略；如果没有指定 Key，则使用轮询策略。

（1）轮询策略


轮询策略（Round-robin），即顺序分配策略，是 Kafka 默认分区策略。如果一个 Topic 有 3 个分区，则第 1 条消息被发送到分区 0，第 2 条被发送到分区 1，第 3 条被发送到分区 2，以此类推。当生产第 4 条消息时会重新轮询将其分配到分区 0。

轮询策略是 Java 生产者 API 默认提供的分区策略。如果未指定 partitioner.class 参数，生产者程序会按照轮询方式在 Topic 的所有 Partition 间均匀地存储消息。轮询策略有非常优秀的负载均衡表现，能保证消息最大限度地被平均分配到所有 Partition 上。

（2）随机策略

随机策略是将消息随机地放置到任意一个 Partition 上。如果要实现随机策略版的 partition 方法，Java 版如下：

List<PartitionInfo> partitions = cluster.partitionsForTopic

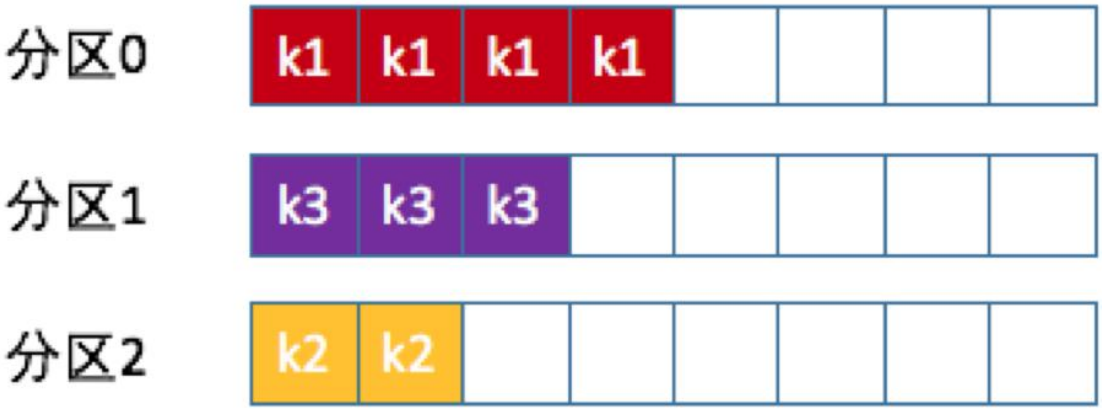
(topic);

return ThreadLocalRandom.current().nextInt(partitions.size());

先计算出 Topic 的总分区数，然后随机地返回一个小于分区数的正整数。随机策略本质上力求将数据均匀地分散到各个 Partition，但实际表现要逊于轮询策略，如果追求数据均匀分布，推荐使用轮询策略。

（3）按消息键保序策略

Kafka 允许为每条消息定义消息键，简称为 Key。Key 是一个有明确业务含义的字符串，如客户代码、部门编号或业务 ID 等，可以用来表征消息元数据。一旦消息被指定 Key，就可以保证相同 Key 的所有消息都进入到相同 Partition 中。




实现分区策略的 partition 方法只需要两行代码即可：

List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);

return Math.abs(key.hashCode()) % partitions.size();

（4）自定义分区策略

如果要自定义分区策略，需要显式地配置生产者端参数 partitioner.

class。编写生产者程序时，可以编写一个具体的类实现 org.apache.kafka.clients.producer.Partitioner 接口（partition()和 close()），通常只需要实现最重要的 partition 方法。

int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster);

设置 partitioner.class 参数为自己实现类的 Full Qualified Name，生产者程序就会按照自定义分区策略的代码逻辑对消息进行分区。

## 4、消息保序方案

Kafka 中 Partition 内消息是有序的，Partition 间的消息是无序的，保序实现方案如下：

（1）单分区 Topic 方案

为了实现消息保序，可以将 Topic 设置成单分区，单分区 Topic 的所有消息都只在一个分区内读写，可以保证全局消息的顺序性，但会丧失 Kafka 多 Partition 的高吞吐量和负载均衡的性能优势。

（2）按消息键保序策略

按消息键保序策略可以实现 Kafka 消息保序。通过对具体业务进行分析，提取出需要保序的消息逻辑主体并建立消息标志位 ID，对标志位设定专门分区策略（将相同业务类消息分配到一个分区），保证同一标志位的所有消息都发送到同一分区，既可以保证分区内的消息顺序，也可以享受到多分区带来的高吞吐量。

基于地理位置的分区策略通常只针对大规模 Kafka 集群，特别是跨

城市、跨国家甚至跨大洲的集群。假设天猫计划为每个新注册用户提供一份注册礼品，比如欧美用户注册天猫时可以免费得到一台 iphone SE 手机，而中国新注册用户可以得到一台华为 P40 Pro。为了实现相应注册业务逻辑，只需要创建一个双分区的 Topic，然后再创建两个消费者程序分别处理欧美和中国用户的注册用户逻辑即可，同时必须把不同地理位置的用户注册的消息发送到不同机房中，因为处理注册消息的消费者程序只可能在某一个机房中启动着。基于地理位置的分区策略可以根据 Broker 的 IP 地址实现定制化的分区策略。

```
List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
return partitions.stream().filter(p -> isChina(p.leader().host())).map(PartitionInfo::partition).findAny().get();
```

可以从所有分区中找出 Leader 在中国的分区，然后随机挑选一个进行消息发送。

## 5、消息生产过程

生产者生产消息，将消息发送给 Broker 并形成可供 Consumer 消费的消息，过程如下：

（1）Producer 先从 Zookeeper 中找到 Partition 的 Leader。

（2）Producer 将消息发送给 Partition 的 Leader。

（3）Leader 将消息接入本地的 Log，并通知 ISR 的 Followers。

（4）ISR 中的 Followers 从 Leader 中 pull 消息，写入本地 Log 后向 Leader 发送 ACK。

（5）Leader 收到所有 ISR 中的 Followers 的 ACK 后，增加 HW 并向 Producer 发送 ACK，表示消息写入成功。

## 6、消息路由策略

在通过 API 方式发布消息时，生产者以 Record 为消息进行发布。Record 中包含 key 与 value，value 是消息体，而 key 用于路由消息要存放到哪个 Partition。消息要写入到哪个 Partition 并非随机，而由路由策略决定。

（1）如果指定 Partition，则直接写入到具体指定 Partition。

（2）如果没有指定 Partition 但指定 key，则通过对 key 的 hash 值与 Partition 数量取模，结果即 Partition 索引。

（3）如果 Partition 和 key 都未指定，则使用轮询算法选出一个 Partition。

（4）增加分区时，Partition 内消息不会重新进行分配，随着数据继续写入，新分区才会参与再平衡。

## 7、消息消费过程

生产者将消息发送到 Topic 中，消费者即可对消息进行消费，其消费过程如下:

（1）Consumer 向 Broker 提交连接请求，Consumer 连接上的 Broker 会向 Consumer 发送 Broker  Controller 的通信 URL，即配置文件中的 listeners 地址；

（2）当 Consumer 指定要消费 Topic 后，会向 Broker Controller 发送消费请求；

（3）Broker Controller 会为 Consumer 分配一个或几个 Partition Leader，并将 Partition 的当前 offset 发送给 Consumer；

（4）Consumer 会按照 Broker Controller 分配的 Partition 对其中的消息进行消费；

（5）当 Consumer 消费完消息后，Consumer 会向 Broker 发送一个消息已经被消费的反馈，即消息的 offset；

（6）Broker 接收到 Consumer 的 offset 后，会更新相应的__consumer_offset 中；

Consumer 可以重置 offset，从而可以灵活消费存储在 Broker 上的消息。

## 四、Kafka Consumer Group

### 1、Consumer Group 简介

传统消息队列模型的消息一旦被消费，就会从队列中被删除，而且只能被一个消费者消费，因此伸缩性很差；发布订阅模型允许消息被多个消费者消费，但其伸缩性不高，因为每个订阅者都必须要订阅 Topic 的所有分区。

Consumer Group 是 Kafka 提供的可扩展且具有容错性的消费者机制，可以避免传统消息队列模型、发布订阅模型的缺陷。

当 Consumer Group 订阅多个 Topic 后，组内 Consumer 不要求

一定要订阅 Topic 的所有分区，只会消费部分分区的消息；而 Consumer Group 间彼此独立，互不影响，能够订阅相同的一组 Topic 而互不干涉。再加上 Broker 端的消息留存机制，Consumer Group 避免了伸缩性差的问题，同时实现了传统消息引擎系统的两大模型：如果所有 Consumer 实例都属于同一个 Group，是消息队列模型；如果所有 Consumer 实例分别属于不同 Group，是发布订阅模型。理想情况下，Consumer 实例数量应该等于 Consumer Group 订阅 Topic 的分区总数。

Consumer Group 可以有多个 Consumer，共享一个 ID（Group ID）。Consumer Group 的所有 Consumer 协调在一起消费所订阅 Topic 的所有 Partition，每个分区只能由同一 Consumer Group 的一个 Consumer 实例消费。

## 2、Consumer Group 特性

（1）Consumer Group 可以有一个或多个 Consumer，Consumer 可以是一个单独进程，也可以是同一进程下的线程。

（2）Group ID 是一个字符串，在一个 Kafka 集群中标识唯一的一个 Consumer Group。

（3）Consumer Group 的所有 Consumer 订阅的 Topic 的一个分区只能分配给 Consumer Group 内的某个 Consumer 消费，但可以被其它 Consumer Group 消费。

## 3、消费者组位移

Consumer Group 的位移是一组 KV 对，Key 是分区，V 是 Consumer 消费 Key 分区的最新位移。

旧版本中 Consumer Group 把位移保存在 ZooKeeper 中，Kafka 重度依赖 ZooKeeper 实现各种各样协调管理。将位移保存在 ZooKeeper 的优点是减少 Broker 端的状态保存开销，但 ZooKeeper 并不适合进行频繁写更新，而 Consumer Group 的位移更新是一个非常频繁的操作，大吞吐量写操作会极大地拖慢 ZooKeeper 集群的性能，因此 Kafka 0.9 版本的 Consumer Group 中重新设计了 Consumer Group 的位移管理方式，将位移保存在 Kafka 内置 Topic（__consumer_offsets）中。

## 4、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。如某个 Consumer Group 下有 20 个 Consumer 实例，订阅一个具有 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，分区的分配过程即 Rebalance。Consumer Group 触发 Rebalance 的条件有 3 个：

（1）Consumer Group 的 Consumer 数量发生变更。如新 Consumer 加入组或者离开组，或有 Consumer 崩溃从组中删除。

（2）订阅 Topic 数发生变更。Consumer Group 支持使用正则表

达式订阅 Topic，在 Consumer Group 运行过程中，如果新创建一个满足正则匹配条件的 Topic，那么 Consumer Group 就会触发 Rebalance。

（3）所订阅 Topic 的分区数发生变更。Kafka 只允许增加 Topic 的分区数，当分区数增加时，就会触发订阅 Topic 的所有 Consumer Group 开启 Rebalance。

Rebalance 发生时，Consumer Group 下所有 Consumer 都会协调在一起共同参与。Kafka 默认提供 3 种 Topic 分区的分配策略，每个 Consumer 实例都能够得到较为平均的分区数。

Rebalance 的缺点如下：

（1）Rebalance 过程中 Consumer Group 的所有 Consumer 停止消费。在 Rebalance 过程中，所有 Consumer 实例都会停止消费，等待 Rebalance 完成。

（2）Rebalance 过程中 Consumer Group 所有 Consumer 共同参与，重新分配所有分区，更高效做法是尽量减少分配方案的变动，如粘性分区分配策略。

（3）Rebalance 过程太慢。

## 5、Coordinator

Coordinator 是运行在每个 Broker 上的 Group Coordinator 进程，用于管理 Consumer Group 中各个 Consumer，主要用于 offset 位移管理和 Rebalance。一个 Coordinator 可以同时管理多个

Consumer Group。

kafka 引入 Coordinator 有其历史背景，原来 Consumer 信息依赖于 Zookeeper 存储，当代理或 Consumer 发生变化时，引发消费者平衡，此时 Consumer 间是互不透明的，每个 Consumer 和 Zookeeper 单独通信，容易造成羊群效应和脑裂问题。

为了解决这些问题，kafka 引入了 Coordinator。服务端引入 Group Coordinator，消费者端引入 Consumer Coordinator。每个 Broker 启动时，都会创建 Group Coordinator，管理部分 Consumer Group（集群负载均衡）和组内每个消费者消费的偏移量（offset）；每个 Consumer 实例化时，同时实例化一个 Consumer Coordinator 对象，负责 Consumer Group 内各个 Consumer 和服务端 Group Coordinator 之间的通信。

## 五、Consumer Offset

### 1、消费位移简介

Consumer Offset 是消费位移，记录 Consumer 要消费的下一条消息的位移，用于表征 Consumer 消费进度，每个 Consumer 都有自己的 Consumer Offset。

Consumer 需要向 Kafka 汇报自己的位移数据，即提交位移（Committing Offsets）。Consumer 能够同时消费多个分区的数据，因此位移的提交是在分区粒度上进行的，即 Consumer 需要为消费的每个分区提交各自的位移数据。提交位移主要是为了表征 Consume

r 的消费进度，当 Consumer 发生故障重启后，就能够从 Kafka 中读取原来提交的位移值，然后从相应的位移处继续消费，从而避免整个消费过程重来一遍。

对位移提交的管理直接影响 Consumer 所能提供的消息语义保障。Kafka Consumer API 提供了多种提交位移的方法。基于用户角度，位移提交分为自动提交和手动提交；基于 Consumer 角度，位移提交分为同步提交和异步提交。

## 2、offset commit

Consumer 从 Broker 中取一批消息写入 buffer 进行消费，在规定时间内消费完消息后，会自动将其消费消息的 offset 提交给 Broker，以记录下哪些消息是消费过的。若在时限内没有消费完毕，不会提交 offset。

## 3、自动提交位移

自动提交是指 Consumer 在后台定期自动提交位移；手动提交是指要自己提交位移。Consumer 端参数 enable.auto.commit 用于控制提交位移的方式，默认值为 true，表示自动提交，即 Java Consumer 默认自动提交位移。Consumer 端参数 auto.commit.interval.ms 用于控制自动提交位移的间隔，默认值是 5 秒，表明 Consumer 每 5 秒会自动提交一次位移。

enable.auto.commit 参数设置为 false 表示手动提交位移，Cons

umer 不会自动提交位移，需要调用相应的 API 手动提交位移。KafkaConsumer#commitSync()方法会提交 KafkaConsumer#poll()返回的最新位移，是一个同步操作，会一直等待直到位移被成功提交才会返回；如果提交过程中出现异常，会将异常信息抛出。commitSync()使用示例如下：

```
while (true) {
            ConsumerRecords<String, String> records =
                        consumer.poll(Duration.ofSeconds(1));
            process(records); // 处理消息
            try {
                    consumer.commitSync();
            } catch (CommitFailedException e) {
                    handle(e); // 处理提交失败异常
            }
}
```

需要在处理完 poll()方法返回的所有消息后调用 consumer.commitSync()方法。如果过早提交位移，可能会出现消费数据丢失的情况。对于自动提交位移，Consumer 会保证在开始调用 poll 方法时，提交上次 poll 返回的所有消息。因此，poll 方法的逻辑是先提交上一批消息的位移，再处理下一批消息，因此能保证不出现消费丢失的情况，但自动提交位移存在可能会出现重复消费的缺陷。在默认情况下，Consumer 每 5 秒自动提交一次位移，如果提交位移后 3 秒发生 Rebalance 操作，在 Rebalance 后，所有 Consumer 从上一次提交的位移处继续消费，但位移已经是 3 秒前的位移数据，因此在 Rebalance 发生前 3 秒消费的所有数据都要重新再消费一次。虽然能够通过减少 auto.commit.interval.ms 的值来提高提交频率，但只能缩小重复消费的时间窗口，不可能完全消除。

## 4、手动提交位移

手动提交位移则更加灵活，完全能够把控位移提交的时机和频率，但在调用 commitSync()时，Consumer 程序会处于阻塞状态，直到远端 Broker 返回提交结果。在任何系统中，因为程序而非资源限制而导致的阻塞都可能是系统瓶颈，会影响整个应用程序的 TPS。虽然可以选择拉长提交间隔，但会导致 Consumer 提交频率下降，在下次 Consumer 重启回来后，会有更多消息被重新消费。

因此，Kafka 为手动提交位移提供了一个异步 API 方法：KafkaConsumer#commitAsync()。调用 commitAsync()后，会立即返回，不会阻塞，因此不会影响 Consumer 应用的 TPS。同时，Consumer 针对异步提交位移接口提供了回调函数（callback），供实现提交后的逻辑，比如记录日志或处理异常等。commitAsync()使用示例如下：

```
while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(1));
            process(records); // 处理消息
            consumer.commitAsync((offsets, exception) -> {
  if (exception != null)
            handle(exception);
  });
}
```

commitAsync 在提交失败时不会自动重试，不能替代 commitSync，因为 commitAsync 是异步操作，倘若提交失败后自动重试，那么重试时提交的位移值可能早已经过期。因此，针对手动提交位移，需要将 commitSync 和 commitAsync 组合使用才能到达最理想的

效果，可以利用 commitSync 的自动重试来规避瞬时错误（如网络瞬时抖动，Broker 端 GC 等），使 Consumer 程序不会一直处于阻塞状态，不影响 TPS。commitSync 和 commitAsync 组合使用示例如下：

```
try {
    while(true) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.of
    Seconds(1));
        process(records); // 处理消息
        commitAysnc(); // 使用异步提交规避阻塞
    }
} catch(Exception e) {
    handle(e); // 处理异常
} finally {
    try {
         consumer.commitSync(); // 最后一次提交使用同步阻塞式提交
    } finally {
        consumer.close();
    }
}
```

上述代码对于常规性、阶段性的手动提交，调用 commitAsync()避免程序阻塞；而在 Consumer 要关闭前，调用 commitSync()方法执行同步阻塞式的位移提交，以确保 Consumer 关闭前能够保存正确的位移数据。因此，既实现了异步无阻塞式的位移管理，也确保了 Consumer 位移的正确性。

commitAsync()和 commitSync()提交的位移是 poll 方法返回的所有消息的位移，如果想更加细粒度化地提交位移，如提交 poll 返回的部分消息的位移，或是在消费的中间进行位移提交，需要新的 Consumer API 接口。为了帮助实现更精细化的位移管理功能，Consumer API 还针对同步/异步提交提供了一组更为方便的方法，com

mitSync(Map<TopicPartition，OffsetAndMetadata>)和 commitAsync(Map<TopicPartition，OffsetAndMetadata>)，参数是 Map 对象，键就是 TopicPartition，即消费的分区，而值是一个 OffsetAndMetadata 对象，保存的主要是位移数据。示例代码如下：

```
private Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();
int count = 0;
……
while (true) {
            ConsumerRecords<String, String> records =
  consumer.poll(Duration.ofSeconds(1));
            for (ConsumerRecord<String, String> record: records) {
                        process(record);  // 处理消息
                        offsets.put(new TopicPartition(record.topic(), record.partit
ion()),
                                    new OffsetAndMetadata(record.offset() + 1);
                       if ( count % 100 == 0 )
                                    consumer.commitAsync(offsets, null); // 回调
处理逻辑是 null
                        count++;
  }
}
```

创建 Map 对象，用于保存 Consumer 消费处理过程中要提交的分区位移，开始逐条处理消息，并构造要提交的位移值。要提交的位移是下一条消息的位移，因此构造 OffsetAndMetadata 对象时，使用当前消息位移加 1。与调用无参的 commitAsync 不同，带 Map 对象参数的 commitAsync 进行细粒度的位移提交。

## 5、CommitFailedException

CommitFailedException 表示 Consumer 客户端在提交位移时出现不可恢复错误或异常。如果异常是可恢复的瞬时错误，提交位移的 API 可以避免，很多提交位移的 API 方法支持自动错误重试。

CommitFailedException 异常通常发生在手动提交位移时，即用户显式调用 KafkaConsumer.commitSync()方法时。

当消息处理的总时间超过预设的 max.poll.interval.ms 参数值时，Kafka Consumer 端会抛出 CommitFailedException 异常。CommitFailedException 异常复现代码如下：

```
...
Properties props = new Properties();
...
props.put("max.poll.interval.ms", 5000);
consumer.subscribe(Arrays.asList("test-topic"));

while (true) {
    ConsumerRecords<String, String> records =
    consumer.poll(Duration.ofSeconds(1));
    // 使用 Thread.sleep 模拟真实的消息处理逻辑
    Thread.sleep(6000L);
    consumer.commitSync();
}
```

使用 KafkaConsumer.subscribe 方法随意订阅一个 Topic，设置 Consumer 端参数 max.poll.interval.ms=5 秒，最后在循环调用 KafkaConsumer.poll 方法间，插入 Thread.sleep(6000)和手动提交位移，成功复现异常。

避免 CommitFailedException 异常：

（1）缩短单条消息处理的时间。

（2）增加 Consumer 端允许下游系统消费一批消息的最大时长。如果消费逻辑不能简化，那么应该提高 Consumer 端参数 max.poll.interval.ms（Kafka 0.10.1.0 版本引入）值。

（3）减少下游系统一次性消费的消息总数。减少 Consumer 端参数 max.poll.records 的值，默认值是 500 条，因此调用一次 Kaf

ka Consumer.poll 方法，最多返回 500 条消息。

（4）下游系统使用多线程来加速消费。使用 Kafka Consumer 消费数据是单线程，当消费速度无法匹配 Kafka Consumer 消息返回的速度时，会抛出 CommitFailedException 异常。如果多线程进行消费，可以灵活地控制线程数量，随时调整消费承载能力，但实现较为复杂。主流的大数据流处理框架使用多线程消费方案，如 Apache Flink 在集成 Kafka 时就创建了多个 Kafka Consumer Thread 线程，自行处理多线程间的数据消费。

Kafka Java Consumer 端还提供了一个名为 Standalone Consumer 的独立消费者，每个 Standalone Consumer 实例都独立工作，彼此之间毫无联系。独立消费者的位移提交机制和 Consumer Group 一样，比如独立消费者也需要指定 group.id 参数才能提交位移。如果应用中同时出现具有相同 group.id 值的 Consumer Group 程序和 Standalone Consumer 程序，那么当独立消费者程序手动提交位移时，Kafka 就会立即抛出 CommitFailedException 异常，因为 Kafka 无法识别具有相同 group.id 的消费者实例，返回一个错误，表明不是消费者组内合法的成员。

## 六、Kafka Rebalance 机制

### 1、Rebalance 简介

当 Consumer Group 中的 Consumer 数量发生变化，或者 Topic 中 Partition 数量发生变化时，Partition 所有权会在消费者间转移，

即 Partition 会重新分配，分配过程称为 Rebalance。

Rebalance 能够给 Consumer Group 及 Broker 带来高性能、高可用性和高伸缩性，但在 Rebalance 期间 Consumer 无法消费消息，即整个 Broker 集群有小一段时间是不可用的。因此要避免不必要的 Rebalance。

Rebalance 是 Consumer Group 的所有 Consumer 就如何消费订阅 Topic 的所有分区达成共识的过程。Rebalance 过程中，所有 Consumer 共同参与，在 Consumer Coordinator 帮助下，完成订阅 Topic 分区的分配。Rebalance 过程中，所有 Consumer 都不能消费任何消息，因此对 Consumer 的 TPS 影响很大。

在 Kafka 中，Coordinator 负责为 Consumer Group 执行 Rebalance 以及提供位移管理和组成员管理等。Consumer 端应用程序在提交位移时，向 Coordinator 所在的 Broker 提交位移。当 Consumer 应用启动时，向 Coordinator 所在的 Broker 发送各种请求，然后由 Coordinator 负责执行消费者组的注册、成员管理记录等元数据管理操作。

所有 Broker 在启动时，都会创建和开启相应的 Coordinator 组件，Consumer Group 确定为其服务的 Coordinator 在哪台 Broker 上的算法有 2 个步骤：

（1）确定由位移 Topic 的哪个分区来保存 Consumer Group 数据：partitionId=Math.abs(groupId.hashCode() % offsetsTopicPartitionCount)。

（2）找出分区 Leader 所在 Broker，即为对应 Coordinator。

## 2、StickyAssignor 粘性分区分配策略

Kafka 每次 Rebalance 时，所有 Consumer 会共同参与重新分配所有分区，效率不高，因此 Kafka 0.11.0.0 版本推出 StickyAssignor 粘性分区分配策略。粘性分区分配是指每次 Rebalance 时会尽可能地保留原来的分区分配方案，尽量实现分区分配的最小变动。

## 3、Rebalance 触发条件

触发 Rebalance 的情况有三种：Consumer Group 内 Consumer 数量发生变化；订阅 Topic 数量发生变化；订阅 Topic 的分区数发生变化。

如果 Consumer Group 的 Consumer 数量发生变化，就一定会引发 Rebalance，是触发 Rebalance 常见原因。

当启动一个配置有相同 group.id 值的 Consumer 程序时，就向 Consumer Group 添加一个新的 Consumer。此时，Coordinator 会接纳新 Consumer 实例，将其加入到 Consumer Group 中，并重新分配分区。增加 Consumer 的操作都是计划内的，可能出于增加 TPS 或提高伸缩性的需要。

当 Consumer Group 完成 Rebalance 后，每个 Consumer 都会定期地向 Coordinator 发送心跳请求。如果某个 Consumer 不能及时地发送心跳请求，Coordinator 会认为 Consumer 已经死，从而

将其从 Group 中移除，然后开启新一轮 Rebalance。

Consumer 端 session.timeout.ms 参数用于指定 Consumer 的心跳超时，默认值 10 秒，如果 Coordinator 在 10 秒内没有收到 Consumer Group 内某 Consumer 的心跳，会认为 Consumer 已经挂掉。

Consumer 的 heartbeat.interval.ms 参数允许控制发送心跳请求频率，值越小，Consumer 发送心跳请求的频率就越高，频繁地发送心跳请求会额外消耗带宽资源，但能够更加快速地知晓当前是否开启 Rebalance。Coordinator 通知各个 Consumer 开启 Rebalance 的方法就是将 REBALANCE_NEEDED 标志封装进心跳请求的响应体中。

Consumer 端 max.poll.interval.ms 参数用于控制 Consumer 实际消费能力对 Rebalance 的影响，限定 Consumer 端应用程序两次调用 poll 方法的最大时间间隔，默认值是 5 分钟，表示 Consumer 程序如果在 5 分钟内无法消费完 poll 方法返回的消息，Consumer 会主动发起离开 Consumer Group 的请求，Coordinator 会开启新一轮 Rebalance。

有多种原因会导致产生非必要的 Rebalance：

（1）未能及时发送心跳，导致 Consumer 被踢出 Consumer Group 而引发。因此，需要仔细地设置 session.timeout.ms 和 heartbeat.interval.ms 的值。生产环境中推荐设置 session.timeout.ms = 6s，heartbeat.interval.ms = 2s。将 session.timeout.

ms 设置成 6s 主要是为了让 Coordinator 能够更快地定位已经挂掉的 Consumer。

（2）Consumer 消费时间过长导致的。max.poll.interval.ms 参数值的设置需要为业务处理逻辑留下充足的时间。

（3）生产环境中，Consumer 端如果出现频繁的 Full GC 导致的长时间停顿，也会引发 Rebalance。

## 七、Kafka 副本机制

### 1、副本机制简介

Replication（副本机制）是指分布式系统在多台网络互联的机器上保存有相同的数据拷贝。副本机制的优点如下：

（1）提供数据冗余。即使系统部分组件失效，系统依然能够继续运转，因而增加整体可用性以及数据持久性。

（2）提供高伸缩性。支持横向扩展，能够通过增加机器的方式来提升读性能，进而提高读操作吞吐量。

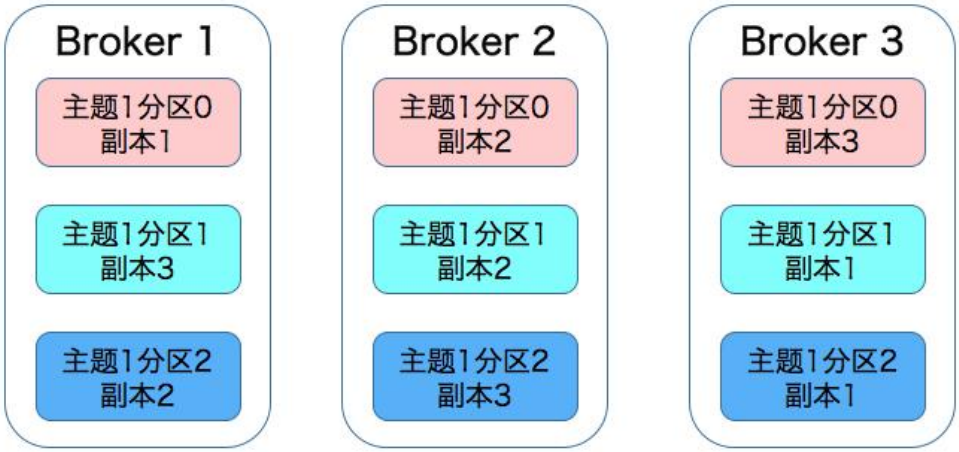
（3）改善数据局部性。允许将数据放入与用户地理位置相近的地方，从而降低系统延时。

### 2、Kafka 副本简介

Replica（Kafka 副本）本质是一个只能追加写消息的提交日志。根据 Kafka 副本机制定义，同一个 Partition 下的所有副本保存有相同的消息序列，并分散保存在不同 Broker 上，从而能够对抗部分 Bro

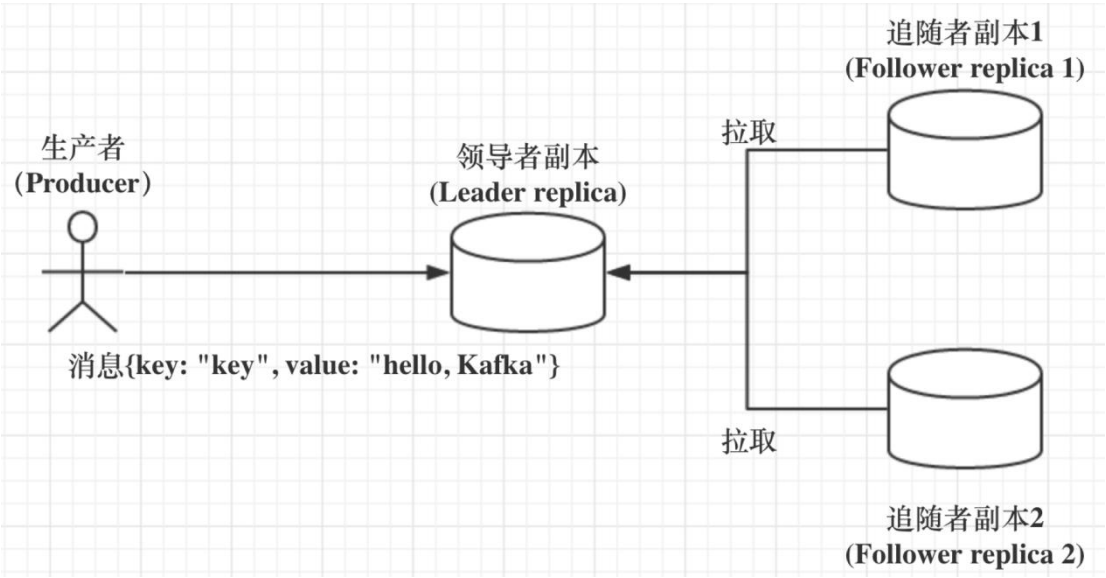
ker 宕机带来的数据不可用。生产环境中，每台 Broker 都可能保存有各个 Topic 下不同 Partition 的不同副本，因此，单个 Broker 上存有成百上千个副本是正常的。

3 台 Broker 的 Kafka 集群上，Topic 1 分区 0 的 3 个副本分散在 3 台 Broker 上，其它 Topic 分区的副本分散在不同的 Broker 上，从而实现数据冗余。




## 3、Kafka 副本机制简介

Kafka 采用基于领导者（Leader-based）的副本机制。



（1）在 Kafka 中，副本分为领导者副本（Leader Replica）和追随者副本（Follower Replica）。每个分区在创建时都要选举一个副本，称为 Leader 副本，其余副本自动称为 Follower 副本。

（2）Kafka 副本机制比其它分布式系统要更严格。Kafka 中 Follower 不对外提供服务，任何一个 Follower 都不能响应消费者和生产者的读写请求，所有的请求都必须由 Leader 来处理，即所有读写请求都必须发往 Leader 所在的 Broker 进行处理。Follower 不处理客户端请求，唯一任务是从 Leader 异步拉取消息，并写入到自己的提交日志中，从而实现与 Leader 的同步。

（3）当 Leader 副本挂掉时，Kafka 依托于 ZooKeeper 提供的监控功能能够实时感知到，并立即开启新一轮 Leader 选举，从 Follower 副本中选举出新 Leader。原 Leader 重启后，只能作为 Follower 加入到集群中。

## 4、Kafka 副本机制优点

Kafka Follower 不对外提供服务，因此 Kafka 不能提供读操作横向扩展以及改善局部性，但 Kafka 副本机制的优点如下：

（1）方便实现读写一致性（Read-your-writes）。读写一致性是当使用生产者 API 向 Kafka 成功写入消息后，马上使用消费者 API 去读取刚才生产的消息。如果允许 Follower 对外提供服务，由于副本同步是异步的，因此有可能出现 Follower 还没有从 Leader 拉取到最新的消息，从而使得客户端看不到最新写入的消息。

（2）方便实现单调读一致性。对于一个消费者用户，在多次消费消息时，不会看到某条消息一会儿存在一会儿不存在。如果允许 Follower 提供读服务，那么假设当前有 2 个 Follower 副本 F1 和 F2，异步地拉取 Leader 副本数据。倘若 F1 拉取了 Leader 的最新消息而 F2 还未及时拉取，那么，此时如果有一个消费者先从 F1 读取消息后又从 F2 拉取消息，可能会出现第一次消费时看到的最新消息在第二次消费时不见。

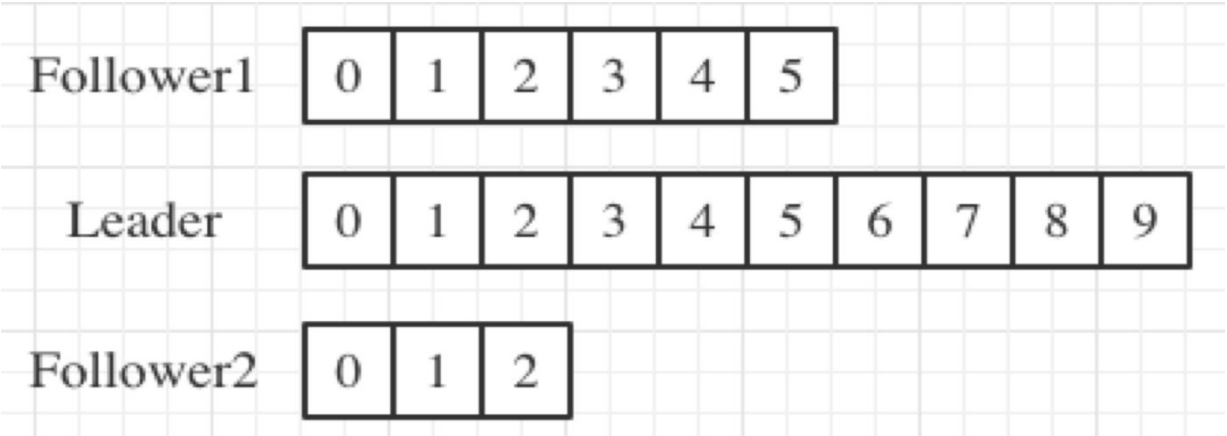
## 5、ISR 简介

Kafka 引入了 In-sync Replicas（副本同步列表）。ISR 中的副本都是与 Leader 同步的副本，不在 ISR 中的 Follower 副本与 Leader 是不同步的。Leader 天然就在 ISR 中，但 ISR 不只是 Follower 副本集合，是一个动态调整的集合，由 Leader 负责维护。

AR（Assigned Replicas）即已分配的副本列表，是指某个 Partition 的所有副本。

OSR（Out of-Sync Replicas），即非同步的副本列表。

AR = ISR + OSR



图中 Leader 副本当前写入 10 条消息，Follower1 副本同步了其中的 6 条消息，而 Follower2 副本只同步了其中的 3 条消息。但 Kafka 判断 Follower 是否与 Leader 同步的标准，并不是看 Follower 副本与 Leader 副本相差的消息数，而是 Broker 端 replica.lag.time.max.ms 参数值。replica.lag.time.max.ms 参数表示 Follower 副本能够落后 Leader 副本的最长时间间隔，默认值是 10 秒。只要一个 Follower 副本落后 Leader 副本的时间不连续超过 10 秒，Kafka 认为 Follower 副本与 Leader 是同步的，即使此时 Follower 副本中保存的消息明显少于 Leader 副本中的消息。Follower 副本唯一的工作就是不断地从 Leader 副本拉取消息，然后写入到自己的提交日志中。如果同步过程的速度持续慢于 Leader 副本的消息写入速度，在 replica.lag.time.max.ms 时间后，Follower 副本会被认为是与 Leader 副本不同步的，因此不能再放入 ISR 中。Kafka 会自动收缩 ISR 集合，将 Follower 副本踢出 ISR。

## 6、Leader 选举

ISR 可以动态调整，如果 ISR 为空，则表明 Leader 副本已经挂掉，因此 Kafka 需要重新选举一个新的 Leader。

Kafka 把所有不在 ISR 中的存活副本都称为非同步副本。通常，非同步副本落后 Leader 副本太多，因此，如果选择非同步副本作为新 Leader，可能会出现数据丢失。Kafka 中，如果 ISR 为空，从非同步副本选举 Leader 的过程称为 Unclean 领导者选举（Unclean Lea

der Election）。

Broker 端参数 unclean.leader.election.enable 控制是否允许 Unclean 领导者选举。开启 Unclean 领导者选举可能会造成数据丢失，但会使得分区 Leader 副本一直存在，不会停止对外提供服务，提高可用性。禁止 Unclean 领导者选举可以维护数据的一致性，避免消息丢失，但牺牲了高可用性。

当 Leader 宕机后，Broker Controller 会从 ISR 中挑选一个 Follower 成为新的 Leader。如果 ISR 中没有其它副本，可以通过 unclean.leader.election.enable 的值来设置 Leader 选举范围。

false：必须等到 ISR 列表中所有的副本都活过来才进行新的选举，可靠性有保证，但可用性低。

true：在 ISR 列表中没有副本的情况下，可以选择任意一个没有宕机的 Broker 作为新的 Leader，可用性高，但可靠性没有保证。

## 八、Kafka Controller 选举机制

### 1、Kafka Controller 简介

Controller 是 Kafka 核心组件，用于在 ZooKeeper 帮助下管理和协调 Kafka 集群，集群中任意一台 Broker 都能充当 Controller 角色，但在运行过程中，只能有一个 Broker 成为 Controller，行使管理和协调的职责。每个正常运转的 Kafka 集群，在任意时刻都有且只有一个 Controller。JMX 指标 activeController 可以帮助实时监控 Controller 的存活状态，在实际运维操作过程中需要实时查看。

Kafka 集群的多个 Broker 中，会选举出一个 Controller，负责管理整个集群中 Partition 和 Replicas 的状态，负责 Leader 的选举。只有 Broker Controller 会向 Zookeeper 中注册 Watcher，其他 Broker 及分区无需注册，即 Zookeeper 仅需监听 Broker Controller 的状态变化即可。

## 2、ZooKeeper 简介

ZooKeeper 是提供高可靠性的分布式协调服务框架，所使用数据模型是树形结构，树形结构上的每个 znode 节点用来保存一些元数据协调信息。如果以 znode 持久性来划分，znode 可分为持久性 znode 和临时 znode。持久性 znode 不会因为 ZooKeeper 集群重启而消失，而临时 znode 则与创建其的 ZooKeeper 会话绑定，一旦会话结束，节点会被自动删除。

ZooKeeper 赋予客户端监控 znode 变更的能力，即 Watch 通知功能。一旦 znode 节点被创建、删除，子节点数量发生变化，抑或是 znode 所存的数据本身变更，ZooKeeper 会通过节点变更监听器(ChangeHandler) 的方式显式通知客户端。ZooKeeper 常被用来实现集群成员管理、分布式锁、领导者选举等功能。Kafka Controoler 大量使用 Watch 功能实现对集群的协调管理。

## 3、Kafka Controller 选举

Broker 在启动时，会尝试去 ZooKeeper 中创建/controller 节点。

Kafka 选举 Controller 的规则是：第一个成功创建/controller 节点的 Broker 会被指定为 Controller。

## 4、Kafka Controller 职责

（1）Topic 管理（创建、删除、增加分区）。Topic 管理是指 Controller 帮助完成对 Kafka Topic 的创建、删除以及分区增加的操作。当执行 kafka-topics 脚本时，大部分后台工作由 Controller 完成。

（2）分区再分配。分区再分配是指 kafka-reassign-partitions 脚本提供的对已有主题分区进行细粒度的分配功能。

（3）Preferred 领导者选举。Preferred 领导者选举主要是 Kafka 为避免部分 Broker 负载过重而提供的一种更换 Leader 的方案。

（4）集群成员管理（新增 Broker、Broker 主动关闭、Broker 宕机），包括自动检测新增 Broker、Broker 主动关闭及被动宕机。自动检测是依赖于 Watch 功能和 ZooKeeper 临时节点组合实现的。Controller 会利用 Watch 机制检查 ZooKeeper 的/brokers/ids 节点下的子节点数量变更。当有新 Broker 启动后，会在/brokers 下创建专属的 znode 节点。一旦创建完毕，ZooKeeper 会通过 Watch 机制将消息通知推送给 Controller，Controller 就能自动地感知到变化，进而开启后续的新增 Broker 作业。侦测 Broker 存活性则是依赖于临时节点。每个 Broker 启动后，会在/brokers/ids 下创建一个临时 znode。当 Broker 宕机或主动关闭后，Broker 与 ZooKeeper 的会话结束，znode 会被自动删除。ZooKeeper 的 Watch 机制将变

更推送给 Controller，Controller 就能知道有 Broker 关闭或宕机，从而进行善后处理。

（5）数据服务。Controller 保存有最全的集群元数据信息，其它所有 Broker 会定期接收 Controller 发来的元数据更新请求，从而更新其内存中的缓存数据。

## 5、Kafka Controller 保存数据

所有 Topic 信息，包括具体的分区信息，比如领导者副本是谁，ISR 集合中有哪些副本等。

所有 Broker 信息，包括当前都有哪些运行中的 Broker，哪些正在关闭中的 Broker 等。

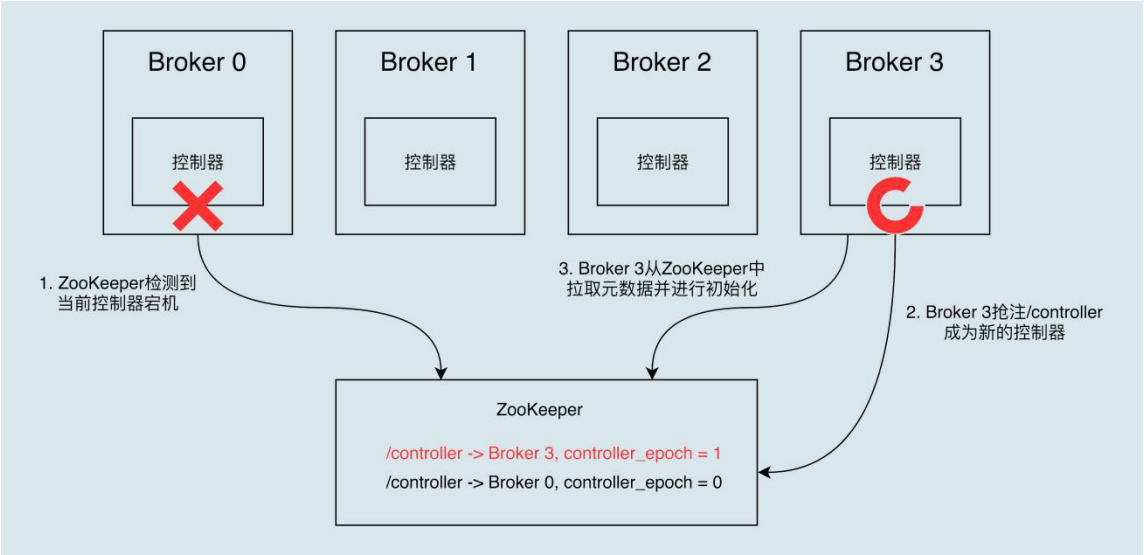
所有涉及运维任务的分区，包括当前正在进行 Preferred 领导者选举以及分区再分配的分区列表。

Controller 保存的数据在 ZooKeeper 中也保存一份，每当 Controller 初始化时，会从 ZooKeeper 上读取对应的元数据并填充到自己的缓存中。

## 6、Kafka 控制器故障转移

在 Kafka 集群运行过程中，只能有一台 Broker 充当 Controller，存在单点故障的风险，因此 Kafka 为 Controller 提供故障转移功能。故障转移是指当运行中的 Controller 突然宕机或意外终止时，Kafka 能够快速地感知到，并立即启用备用 Controller 来代替故障 Cont

roller。故障转移过程是自动完成。



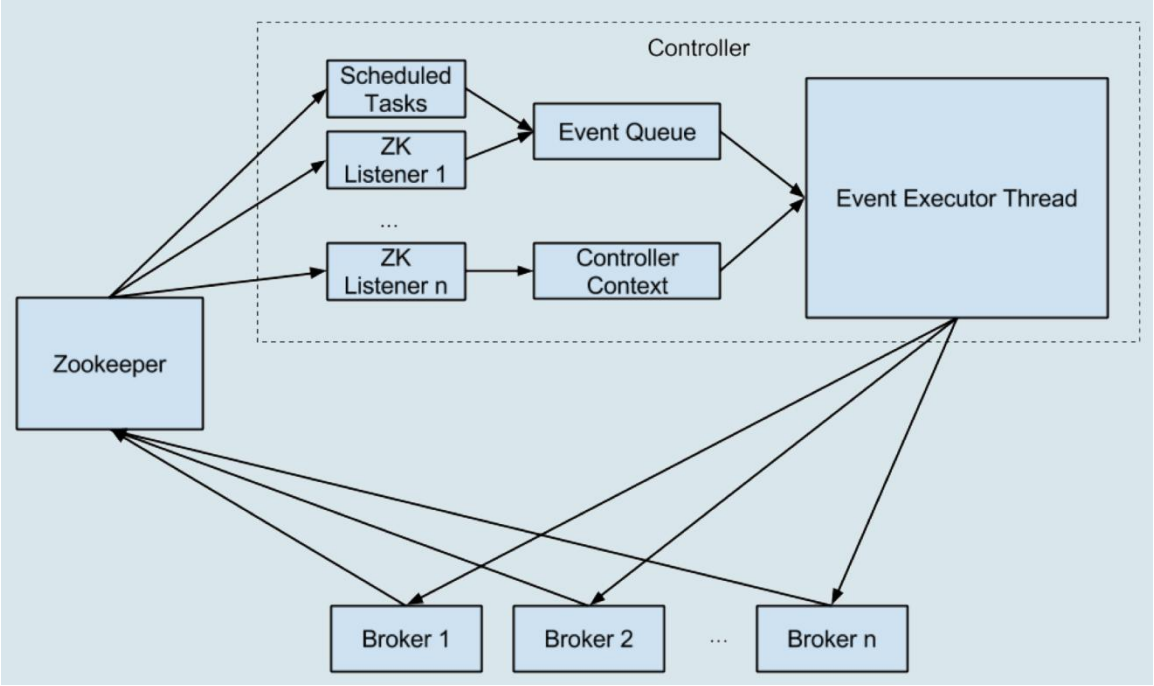

开始时，Broker 0 是 Controller。当 Broker 0 宕机后，ZooKeeper 通过 Watch 机制感知到并删除/controller 临时节点。然后，所有存活的 Broker 开始竞选新的 Controller，Broker 3 最终赢得选举，成功地在 ZooKeeper 上重建 /controller 节点。Broker 3 会从 ZooKeeper 中读取集群元数据信息，并初始化到自己的缓存中。

## 7、Kafka Controller 设计

Kafka 0.11 版本前，Controller 是多线程设计，会在内部创建很多个线程。Controller 需要为每个 Broker 都创建一个对应 Socket 连接，然后再创建一个专属线程，用于向 Broker 发送特定请求。Controller 连接 ZooKeeper 的会话，也会创建单独线程来处理 Watch 机制的通知回调。Controller 还会为主题删除创建额外的 IO 线程。Controller 创建的多线程会访问共享的 Controller 缓存数据，为了保护数据安全性，Controller 不得不在代码中大量使用 Reentrant

Lock 同步机制，进一步拖慢了整个 Controller 的处理速度。

Kafka 0.11 版本重构了 Kafka 0.11 版本的底层设计，把多线程的方案改成了单线程加事件队列的方案。




事件处理线程统一处理各种 Controller 事件，然后 Controller 将原来执行的操作全部建模成一个个独立事件，发送到专属事件队列中，供线程消费。Controller 缓存中保存的状态只被一个线程处理，因此不再需要重量级的线程同步机制来维护线程安全，Kafka 不用再担心多线程并发访问的问题，非常利于定位和诊断控制器的各种问题。

Kafka 0.11 版本将 Controller 对 ZooKeeper 的同步操作全部改为异步操作，大幅提高性能（ZooKeeper 写入性能提升 10 倍）。ZooKeeper 本身的 API 提供了同步写和异步写两种方式，Kafka 0.11 版本前 Controller 操作 ZooKeeper 使用同步 API，性能很差，当有大量 Topic 分区发生变更时，ZooKeeper 容易成为系统瓶颈。

# 九、Kafka 核心技术

## 1、生产者压缩算法

（1）Kafka 消息版本

压缩是用时间换空间的经典 trade-off 思想，使用 CPU 时间换磁盘空间或网络 IO 传输量，以较小 CPU 开销带来更少磁盘占用或更少网络 IO 传输。

Kafka 有 V1 版本和 V2 版本两类消息格式，V2 版本在 Kafka 0.11.0.0 中正式引入。

Kafka 消息层次都分为两层：消息集合以及消息。消息集合中包含若干条日志项（record item），而日志项才是真正封装消息数据的实体。Kafka 底层的消息日志由一系列消息集合日志项组成。Kafka 通常不会直接操作具体的消息，而是在消息集合层面进行写入操作。V2 版本主要针对 V1 版本的缺陷进行修正，将消息的公共部分抽取出来放到外层的消息集合里，不用每条消息都保存。比如，在 V1 版本中，每条消息都需要执行 CRC 校验，但在 Broker 端可能会对消息时间戳字段进行更新，CRC 值也会相应更新；比如 Broker 端在执行消息格式转换时（兼容老版本客户端程序），CRC 值也会变化。逐条消息执行 CRC 校验不仅浪费空间还浪费 CPU 资源，因此在 V2 版本中，消息 CRC 校验被移到消息集合。

（2）Kafka 消息压缩

在 Kafka 中，消息压缩可能发生在生产者端和 Broker 端。生产者程

序中配置compression.type参数即表示启用指定类型的压缩算法。Producer 启动后生产的每个消息集合都经 GZIP 压缩过，可以很好地节省网络传输带宽以及 Kafka Broker 端的磁盘占用。

通常 Broker 从 Producer 端接收到消息后会原封不动地保存而不会对其进行任何修改，但有两种情况会进行解压缩操作：

A、Broker 端和 Producer 端指定了不同的压缩算法。如 Producer 指定要使用 GZIP 进行压缩，但 Broker 指定必须使用 Snappy 算法进行压缩。因此，Broker 接收到 GZIP 压缩消息后，只能解压缩后使用 Snappy 重新压缩一遍。Broker 端 compression.type 参数用于指定压缩算法，默认值是 producer，表示 Broker 端会使用 Producer 端使用的压缩算法。如果在 Broker 端设置了不同的 compression.type 值，可能会发生预料外的压缩和解压缩操作，通常表现为 Broker 端 CPU 使用率飙升。

B、Broker 端发生消息格式转换。消息格式转换主要是为了兼容老版本的消费者程序。生产环境中，Kafka 集群中同时保存多种版本的消息格式非常常见。为了兼容老版本消息格式，Broker 端会对新版本消息执行向老版本格式的转换。转换过程中会涉及消息解压缩和重新压缩。消息格式转换对性能有很大影响的，除了压缩导致的性能损失，Kafka 也会丧失 Zero Copy 特性。

（3）Kafka 消息解压

解压缩通常发生在消费者程序中，Producer 发送压缩消息到 Broker 后，Broker 会原封不动保存。当 Consumer 程序请求消息时，Br

oker 会原样发出，当消息到达 Consumer 端后，Consumer 自行解压缩消息。Kafka 会将使用的压缩算法封装进消息集合中，当 Consumer 读取到消息集合时，会知道消息使用的压缩算法。除在 Consumer 端解压缩，Broker 端也会进行解压缩，每个压缩过的消息集合在 Broker 端写入时都要发生解压缩操作，对消息执行各种验证。解压缩对 Broker 端性能是有一定影响的。

（4）Kafka 压缩算法对比

Kafka 2.1.0 版本前，Kafka 支持 GZIP、Snappy、LZ4 三种压缩算法。Kafka 2.1.0 开始，Kafka 正式支持 Zstandard 算法，是 Facebook 开源的一个压缩算法，能够提供超高压缩比。压缩算法可以使用压缩比和压缩/解压缩吞吐量两个指标进行衡量。不同压缩算法的性能比较如下：

| Compressor name | Ratio | Compression | Decompress. |
|---|---|---|---|
| **zstd 1.3.4 -1** | 2.877 | 470 MB/s | 1380 MB/s |
| zlib 1.2.11 -1 | 2.743 | 110 MB/s | 400 MB/s |
| brotli 1.0.2 -0 | 2.701 | 410 MB/s | 430 MB/s |
| quicklz 1.5.0 -1 | 2.238 | 550 MB/s | 710 MB/s |
| lzo1x 2.09 -1 | 2.108 | 650 MB/s | 830 MB/s |
| lz4 1.8.1 | 2.101 | 750 MB/s | 3700 MB/s |
| snappy 1.1.4 | 2.091 | 530 MB/s | 1800 MB/s |
| lzf 3.6 -1 | 2.077 | 400 MB/s | 860 MB/s |

生产环境中，GZIP、Snappy、LZ4、zstd 性能表现各有千秋，在吞吐量方面：LZ4 > Snappy > zstd > GZIP；在压缩比方面，z

std > LZ4 > GZIP > Snappy。

如果要启用 Producer 端压缩，Producer 程序运行机器上的 CPU 资源必须充足。除了 CPU 资源充足，如果生产环境中带宽资源有限，也建议 Producer 端开启压缩。通常，带宽比 CPU 和内存要昂贵，因此千兆网络中 Kafka 集群带宽资源耗尽很容易出现。如果客户端机器 CPU 资源富余，建议 Producer 端开启 zstd 压缩，可以极大地节省网络资源消耗。对于解压缩，需要避免非正常的解压缩，如消息格式转换的解压缩操作、Broker 与 Producer 解压缩算法不一致。

## 2、EOS 精确一次语义

（1）消息交付语义

消息交付可靠性保障是指 Kafka 对 Producer 和 Consumer 要处理的消息提供什么样的承诺。常见消息交付语义有以下三种：

最多一次（at most once）：消息可能会丢失，但绝不会被重复发送。

至少一次（at least once）：消息不会丢失，但有可能被重复发送。

精确一次（exactly once）：消息不会丢失，也不会被重复发送。

Kafka 默认提供的交付可靠性保障是至少一次。消息已提交的含义，即只有 Broker 成功提交消息且 Producer 接到 Broker 的 ACK 应答才会认为消息成功提交。如果消息成功提交，但 Broker 应答没有成功发送回 Producer 端（如网络出现瞬时抖动），那么 Producer 就无法确定消息是否真的提交成功。因此，Producer 只能选择重试，

再次发送相同的消息。

Kafka 可以提供最多一次交付保障，只需要让 Producer 禁止重试即可，此时消息要么写入成功，要么写入失败，但绝不会重复发送。

Kafka 的精确一次处理语义是通过幂等性和事务两种机制实现。幂等性 Producer 和事务型 Producer 是 Kafka 为实现精确一次处理语义所提供的工具，但作用范围是不同的。幂等性 Producer 只能保证单分区、单会话上的消息幂等性；而事务能够保证跨分区、跨会话间的幂等性。

（2）幂等性

幂等性在数学领域中指某些操作或函数能够被执行多次，但每次得到的结果都是不变的。幂等性最大的优势在于可以安全地重试任何幂等性操作，不会破坏系统状态。

在 Kafka 中，Producer 默认不是幂等性的，但可以创建幂等性 Producer。Kafka 0.11.0.0 版本引入幂等性，指定 Producer 幂等性的方法只需要设置一个参数即可，enable.idempotence 被设置成 true 后，Producer 自动升级成幂等性 Producer，其它所有代码逻辑都不需要改变。Kafka 自动进行消息去重。

幂等性 Producer 的作用范围：只能保证单分区上的幂等性，即一个幂等性 Producer 能够保证某个主题的一个分区上不出现重复消息，无法实现多个分区的幂等性；只能实现单会话上的幂等性，不能实现跨会话的幂等性。

（3）事务

在数据库领域，事务提供的安全性保障是经典的 ACID，即原子性（Atomicity）、一致性（Consistency)、隔离性（Isolation）和持久性（Durability)。很多数据库厂商对于隔离级别有不同的实现，如有的数据库提供 Snapshot 隔离级别，但在其它数据库称为可重复读（repeatable read）。但对于已提交读（read committed）隔离级别，主流数据库厂商都比较统一。read committed 指当读取数据库时，只能看到已提交的数据，即无脏读；当写入数据库时，只能覆盖掉已提交数据，即无脏写。

Kafka 0.11 版本开始提供对事务的支持，目前主要在 read committed 隔离级别上做事情，能保证多条消息原子性地写入到目标分区，同时也能保证 Consumer 只能看到事务成功提交的消息。

事务型 Producer 能够保证将消息原子性地写入到多个分区中。消息集合的消息要么全部写入成功，要么全部失败。

事务型 Producer 可以进行进程重启，Producer 重启后，Kafka Broker 依然保证发送消息的精确一次处理。

设置事务型 Producer 需要开启 enable.idempotence = true，并设置 Producer 端参数 transactional.id。

事务型 Producer 会调用一些事务 API，如 initTransaction、beginTransaction、commitTransaction 和 abortTransaction，分别对应事务的初始化、事务开始、事务提交以及事务终止。Java 实例代码如下：

```
producer.initTransactions();
try {
```

```
            producer.beginTransaction();
            producer.send(record1);
            producer.send(record2);
            producer.commitTransaction();
} catch (KafkaException e) {
            producer.abortTransaction();
}
```

上述代码能够保证 record1 和 record2 被当作一个事务统一提交到 Kafka，要么全部提交成功，要么全部写入失败。但即使写入失败，Kafka 也会把消息写入到底层的日志中，即 Consumer 还是会看到消息。因此在 Consumer 端，读取事务型 Producer 发送的消息需要做一些修改，需要设置 isolation.level 参数的值。isolation.level 参数值可选如下：

read_uncommitted：默认值，表明 Consumer 能够读取到 Kafka 写入的任何消息，不论事务型 Producer 提交事务还是终止事务，其写入的消息都可以读取。

read_committed：Consumer 只会读取事务型 Producer 成功提交事务写入的消息，但 Consumer 也能看到非事务型 Producer 写入的所有消息。

## 3、Kafka  HW 机制

（1）水位简介

水位是一个单调增加且表征最早未完成工作的时间戳。水位通常用于流式处理领域。

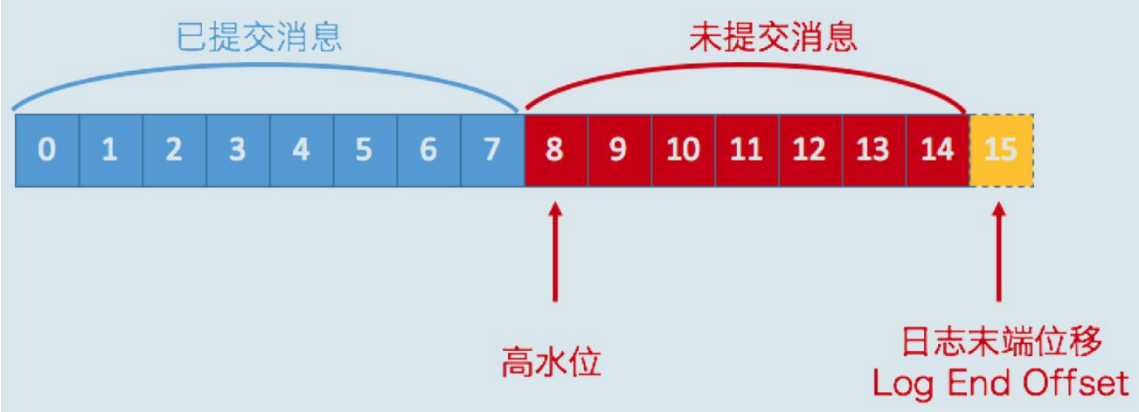
（2）Kafka 水位简介

Kafka 水位用消息位移来表征，不是时间戳，与时间无关。Kafka 水

位的作用如下：

A、定义消息可见性，即用来标识分区下的哪些消息是可以被消费者消费的。

B、帮助 Kafka 完成副本同步。

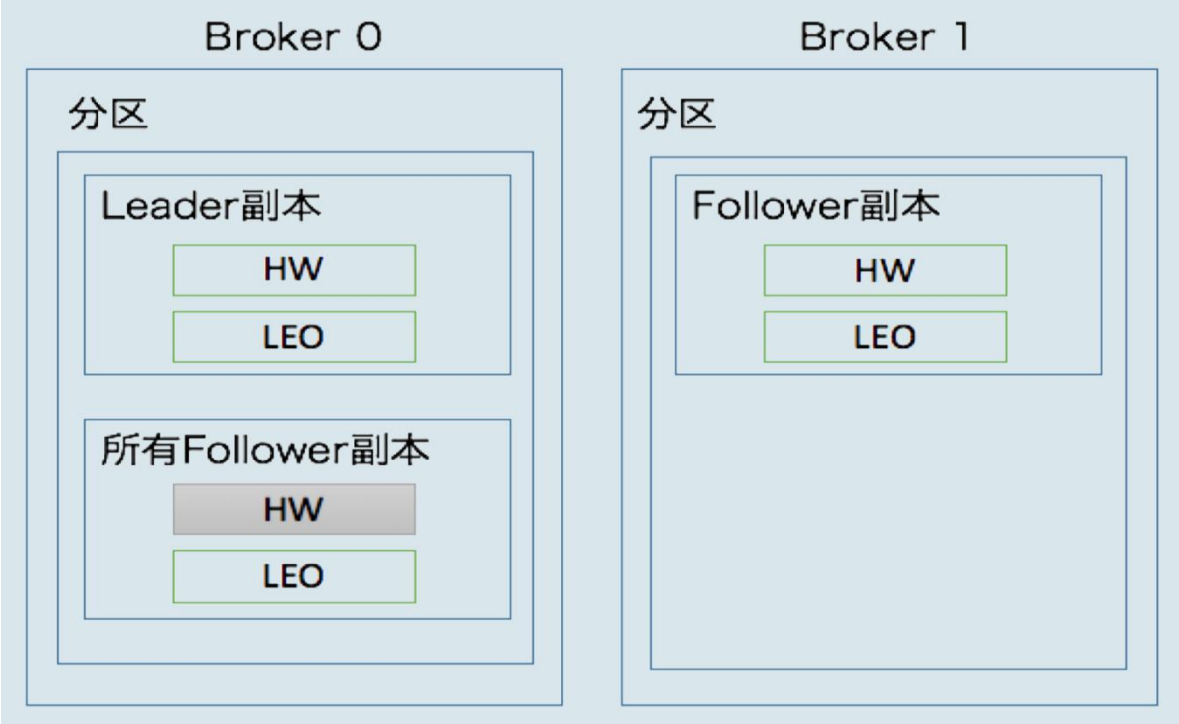



在分区高水位以下的消息被认为是已提交消息，在分区高水位以上的消息是未提交消息，消费者只能消费已提交消息，高水位上的消息是不能被消费者消费的。

日志末端位移（LEO，Log End Offset）表示副本写入下一条消息的位移值。当前分区副本当前只有 15 条消息，位移值是从 0 到 14，下一条新消息的位移是 15。介于高水位和 LEO 之间的消息就属于未提交消息。同一个副本对象，其高水位值不会大于 LEO 值。Kafka 所有副本都有对应的高水位和 LEO 值，而不仅仅是 Leader 副本。只不过 Leader 副本比较特殊，Kafka 使用 Leader 副本的高水位来定义所在分区的高水位，分区的高水位就是 Leader 副本的高水位。

（3）Kafka 高水位更新机制

每个副本对象都保存了一组高水位值和 LEO 值，在 Leader 副本所在的 Broker 上还保存了其它 Follower 副本的 LEO 值。




Broker 0 保存了某分区的 Leader 副本和所有 Follower 副本的 LEO 值，而 Broker1 上仅仅保存了分区的某个 Follower 副本。Kafka 把 Broker 0 上保存的 Follower 副本称为远程副本。Kafka 副本机制在运行过程中，会更新 Broker 1 上 Follower 副本的高水位和 LEO 值，同时也会更新 Broker 0 上 Leader 副本的高水位和 LEO 以及所有远程副本的 LEO，但不会更新远程副本的高水位值。在 Broker 0 上保存远程副本的主要作用是，帮助 Leader 副本确定其分区高水位。

（4）HW 截断机制

如果 Partition Leader 接收到新的消息，ISR 中其它 Follower 正在同步过程中，还未同步完毕时 Leader 宕机，此时需要选举出新的 Leader。若没有 HW 截断机制，将会导致 Partition 中 Leader 与 Follower 数据的不一致。

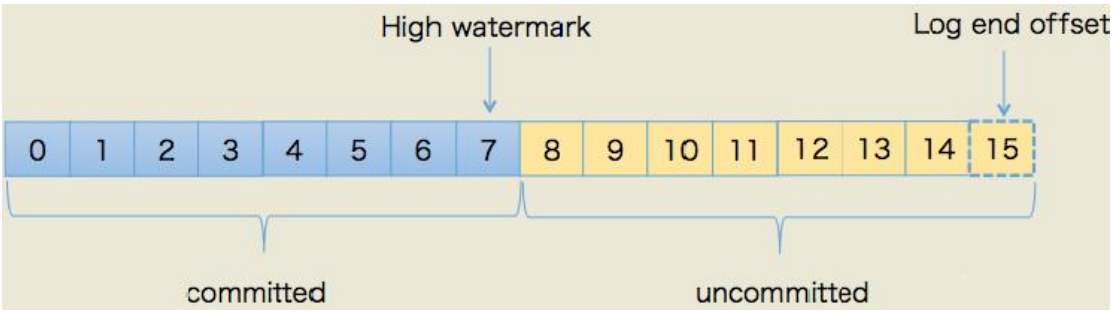
当原 Partition Leader 宕机后又恢复时，将其 LEO 回退到其宕机时的 HW，然后再与新的 Leader 进行数据同步，保证旧 Partition Leader 与新 Partition Leader 中数据一致，称为 HW 截断机制。

（5）HW 和 LEO

HW（High Watermark，高水位）表示 Consumer 可以消费到的最高 Partition 偏移量。HW 保证了 Partition 的 Follower 与 Leader 间数据的一致性，即保证了 Kafka 集群中消息的一致性。

LEO（Log End Offset，日志最后消息的偏移量）是当前最后一个写入的消息在 Partition 中的偏移量。

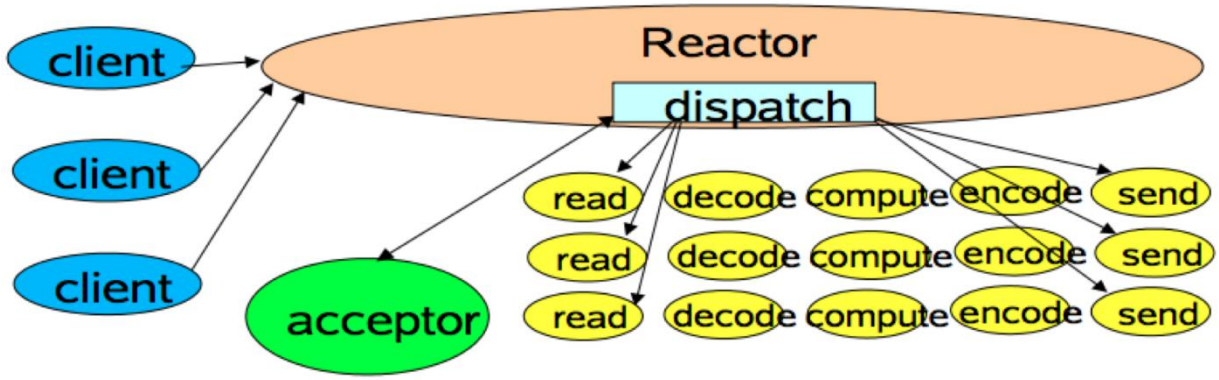
对于 Leader 新写入的消息，Consumer 是不能立刻消费的。Leader 会等待消息被所有 ISR 中的 Partition Follower 同步后才会更新 HW，此时消息才能被 Consumer 消费。




## 4、Kafka 并发请求处理

（1）Reactor 架构

Reactor 模式是事件驱动架构的一种实现方式，特别适合应用于处理多个客户端并发向服务器端发送请求的场景。

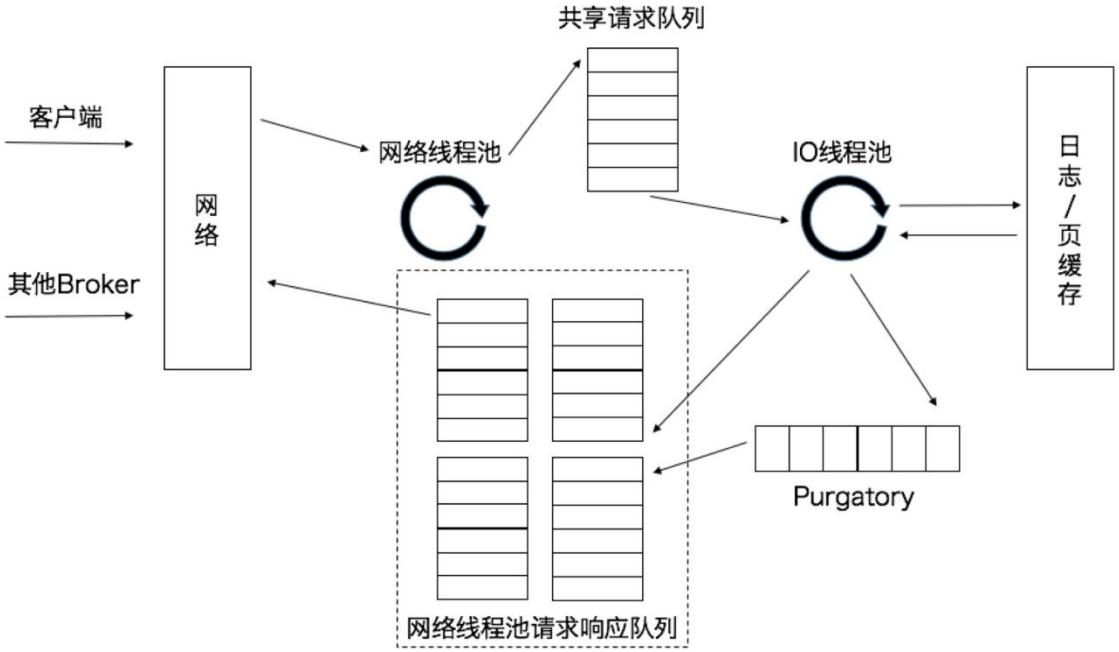



多个客户端会发送请求到 Reactor，Reactor 的请求分发线程 Dispatcher（Acceptor）会将不同请求分发到多个工作线程中处理。Acceptor 线程只用于请求分发，不涉及具体处理逻辑，非常轻量级，因此有很高吞吐量；工作线程可以根据实际业务处理需要任意增减，从而动态调节系统负载能力。

（2）Kafka 请求处理

Kafka 定义了一组请求协议，用于实现各种各样的交互操作。比如 PRODUCE 请求用于生产消息，FETCH 请求用于消费消息，METADATA 请求用于请求 Kafka 集群元数据信息。Kafka 2.3 版本共定义了多达 45 种请求格式。

Kafka 的 Broker 端的 SocketServer 组件是 Reactor 模式中的 Dispatcher 角色，有对应 Acceptor 线程和一个网络线程池。Kafka 的 Broker 端参数 num.network.threads 用于调整网络线程池的线程数，默认值 3，表示每台 Broker 启动时会创建 3 个网络线程，专门处理客户端发送的请求。Acceptor 线程采用轮询方式将入站请求公平地发到所有网络线程中。当网络线程接收到请求后，进入异步线程池的处理。




当网络线程收到请求后，会将请求放入到一个共享请求队列中。Broker 端的 IO 线程池，负责从共享请求队列中取出请求，执行真正处理。对于 PRODUCE 生产请求，则将消息写入到底层磁盘日志中；对于 FETCH 请求，则从磁盘或页缓存中读取消息。IO 线程池中线程是执行请求逻辑的线程。Broker 端参数 num.io.threads 用于控制线程池中线程数，参数默认值 8，表示每台 Broker 启动后自动创建 8 个 IO 线程处理请求，可以根据实际硬件条件设置 IO 线程池的个数。当 IO 线程处理完请求后，会将生成的响应发送到网络线程池的响应队列中，然后由对应的网络线程负责将 Response 返还给客户端。请求队列是所有网络线程共享，而响应队列则是每个网络线程专属，因为 Dispatcher 只是用于请求分发而不负责响应回传，因此只能让每个网络线程自己发送 Response 给客户端。

Purgatory 组件是用来缓存延时请求的。延时请求是一时未满足条件不能立刻处理的请求，如设置了 acks=all 的 PRODUCE 请求，一

旦设置 acks=all，那么请求就必须等待 ISR 中所有副本都接收消息后才能返回，此时处理请求的 IO 线程就必须等待其他 Broker 的写入结果。当请求不能立刻处理时，就会暂存在 Purgatory 中。一旦满足完成条件，IO 线程会继续处理请求，并将 Response 放入对应网络线程的响应队列中。

Kafka Broker 对所有请求是一视同仁的。但 Kafka 内部除了客户端发送的 PRODUCE 请求和 FETCH 请求外，还有很多执行其他操作的请求类型，如负责更新 Leader 副本、Follower 副本以及 ISR 集合的 LeaderAndIsr 请求，负责勒令副本下线的 StopReplica 请求等。Kafka 把 PRODUCE 和 FETCH 请求称为数据类请求，把 LeaderAndIsr、StopReplica 请求称为控制类请求。

## 十、Kafka Consumer 线程设计

### 1、Kafka Consumer 线程设计

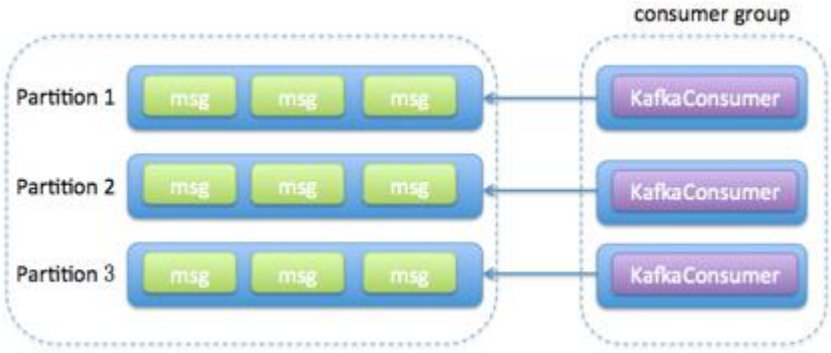
Kafka 0.10.1.0 版本开始，Kafka Consumer 采用双线程设计，即用户主线程和心跳线程。用户主线程是启动 Consumer 应用程序 main 方法的线程，而新引入的心跳线程只负责定期给对应的 Broker 机器发送心跳请求，以标识消费者应用的存活性。心跳线程的引入将心跳频率与主线程调用 KafkaConsumer.poll 方法的频率分开，从而解耦真实的消息处理逻辑与消费者组成员存活性管理。但由于消息获取逻辑依然是在用户主线程中完成的，因此，依然可以安全地认为 Kafka Consumer 是单线程设计。

## 2、Kafka Consumer 多线程设计

KafkaConsumer 类不是线程安全的，所有的网络 IO 处理都在用户主线程中，在使用过程中必须要确保线程安全。不能在多个线程中共享同一个 KafkaConsumer 实例，否则程序会抛出 ConcurrentModificationException 异常。KafkaConsumer 的 wakeup()方法是线程安全的，可以在其他线程中安全地调用 KafkaConsumer.wakeup()来唤醒 Consumer。

## 3、多线程方案一

消费者程序启动多个线程，每个线程维护专属的 Kafka Consumer 实例，负责完整的消息获取、消息处理流程。




优点：

（1）实现简单，使用多个线程并在每个线程中创建专属的 KafkaConsumer 实例即可。

（2）多个线程间没有任何交互，不用考虑线程安全开销。

（3）每个线程使用专属的 Kafka Consumer 实例来执行消息获取和消息处理逻辑，因此，Kafka 主题中的每个分区都能保证只被一个

线程处理，很容易实现分区内的消息消费顺序。

缺点：

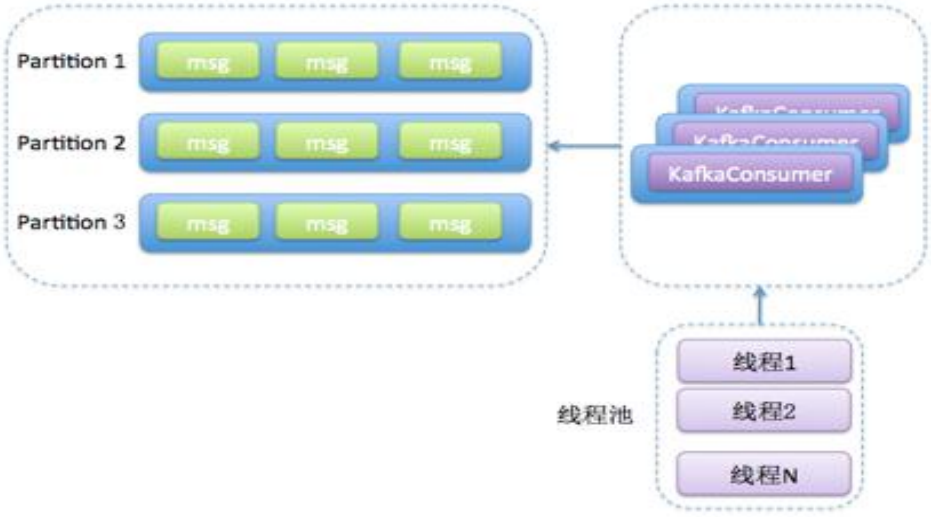
（1）每个线程都需要维护自己的 KafkaConsumer 实例，必然会占用更多的系统资源，比如内存、TCP 连接等。

（2）可以使用的线程数受限于 Consumer 订阅主题的总分区数，在一个消费者组中，每个订阅分区都只能被组内的一个消费者实例所消费。

（3）每个线程需要完整地执行消息获取和消息处理逻辑，如果消息处理逻辑很重，造成消息处理速度慢，容易产生不必要 Rebalance，从而引发整个消费者组的消费停滞。

## 4、多线程方案二

消费者程序使用单或多线程获取消息，同时创建多个消费线程执行消息处理逻辑。获取消息的线程可以是单线程，也可以是多线程，每个线程维护专属的 Kafka  Consumer 实例，处理消息则交由特定的线程池处理，从而实现消息获取与消息处理的真正解耦。



优点：

提高系统伸缩性。将任务切分成消息获取和消息处理两个部分，分别由不同的线程处理，可以独立地调节消息获取的线程数以及消息处理的线程数，而不必考虑两者之间是否相互影响。如果消费获取速度慢，那么增加消费获取的线程数即可；如果消息的处理速度慢，那么增加 Worker 线程池线程数即可。

缺点：

（1）实现难度要大，需要分别管理两组线程。

（2）无法保证分区内的消费顺序。由于使用两组线程，消息获取可以保证分区内的消费顺序，但消息处理时 Worker 线程池将无法保证分区内的消费顺序。

（3）使用两组线程，使得整个消息消费链路被拉长，最终导致正确位移提交会变得异常困难，可能会出现消息的重复消费。

## 5、消息重复消费问题

（1）同一个 Consumer 重复消费

当 Consumer 由于消费能力低而引发消费超时，则可能会形成重复消费。

在某数据刚好消费完毕，但正准备提交 offset 时，消费时间超时，则 Broker 认为消息未消费成功，产生重复消费问题。

解决方案：延长 offset 提交时间。

（2）不同的 Consumer 重复消费

当 Consumer 消费了消息，但还没有提交 offset 时宕机，则已经被消费过的消息会被重复消费。

解决方案：将自动提交改为手动提交。

## 6、Kafka 重复消费问题架构设计

（1）保存并查询

给每个消息都设置一个唯一的 UUID，在消费消息时，首先在持久化系统中查询消息是否被消费过，如果没有消费过，再进行消费；如果已经消费过，直接丢弃。

（2）利用幂等性

幂等性操作的特点是任意多次执行所产生的影响均与一次执行的影响相同。

如果将系统消费消息的业务逻辑设计为幂等性操作，就不用担心 Kafka 消息的重复消费问题，因此可以将消费的业务逻辑设计成具备幂等性的操作。利用数据库的唯一约束可以实现幂等性，如在数据库中建一张表，将表的两个或多个字段联合起来创建一个唯一约束，因此只能存在一条记录。

（3）设置前提条件

实现幂等性的另一种方式是给数据变更设置一个前置条件。如果满足条件就更新数据，否则拒绝更新数据，在更新数据的时候，同时变更前置条件中需要判断的数据。