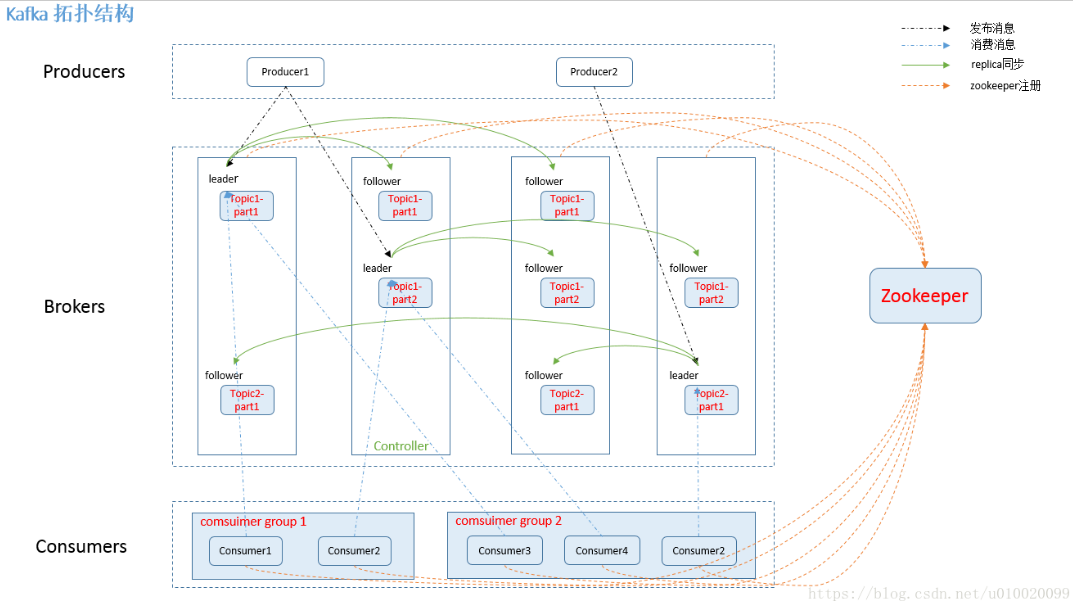
Kafka——流平台：在这个平台上可以发布和订阅数据流，并将它们保存起来进行处理。

类似于消息系统，允许发布和订阅消息流，区别：1）分布式系统，以集群方式运行，可以自由伸缩。 2）可存储数据，作为数据连接层 提供数据传递保证——可复制、持久化。 3）流式处理能力——动态处理派生流和大型数据流。



一、初识

1、

消息——kafka数据单元， 消息可以有一个可选的元数据——键。

批次—— 一组消息，为提高效率 消息分批写入kafka

消息模式——定义消息格式，通常Avro模式 模式和消息体分开。

主题(topic)——消息通过主题分类 类似于数据库的表、文件系统的文件夹

分区(partition)——主题分为若干分区，一个分区就是一个提交日志。消息以追加的方式写入分区，先入先出

流—— 一个主题的数据，从生产者移动到消费者。

生产者——（发布者、写入者）创建消息，

消费者—— （订阅者、读者）读取消息，

偏移量——另一种元数据，类似于id 递增整数，给定分区中每个消息的偏移量都是唯一的

broker—— 缓存代理，Kafka集群中的一台或多台服务器统称broker，从生产者接受消息 设置偏移量。不保存订阅者的状态，由订阅者自己保存,没有副本机制，一旦broker宕机，该broker的消息将都不可用

集群—— 多个broker组成集群，其中一个broker担任集群控制器“首领”

保留消息（在一定期限内）——kafka一个重要特性，要么保留一段时间，要么保留到消息达到一定大小的字节数

1. 多集群

需求：数据类型分离、安全需求隔离、 多数据中心（灾难恢复）

（Kafka 的消息复制机制只能在单个集群里进行，不能在多个集群之间进行。）

MirrorMaker——用它来实现集群间的消息复制。包含了一个生产者和一个消费者，两者之间通过一个队列相连。

Kafka 不仅支持多个消费者，还允许消费者非实时地读取消息，因为Kafka 的数据保留特性。消息被提交到磁盘，根据设置的保留规则进行保存。

伸缩性——为了能够轻松处理大量数据， Kafka可以从一个broker不断拓展 其不影响整体系统的可用性，即使个别 broker失效，仍然可以持续地为客户提供服务。要提高集群的容错能力

3、如何选定分区数：

如果你估算出主题的吐量和悄费者吞吐量，可以用主题吞吐量除以消费者吞吐量算出分区的个数。也就是说，如果每秒钟要从主题上写入和读取 1GB 的数据，并且每个消费者每秒钟可以处理 50MB的数据，那么至少需要 20 个分区

信息较少的话，分区的大小限制在 25G 以内可以得到比较理想的效果。

1. 生产者

生产者——向Kafka 写入数据

1. 创建kafka生产者

Kafka 生产者配置文件中有3个必选的属性：

1）bootstrap.servers

该属性指定 broker 的地址清单，地址的格式为 host:port。清单里不需要包含所有的broker 地址，生产者会从给定的broker里查找到其他broker 的信息。不过建议至少要提供两个 broker信息，一单其中一个若机，生产者仍然能够连接到集群上。

1. key.serializer

broker 希望接收到的消息的键和值都是字节数组。生产者接口允许使用参数化类型，因此可以把 Java 对象作为键和值发送给 broker。可通过该属性将键定义为字符串

1. value.serializer 可通过该属性将值定义为字符串

JAVA实例：创建生产者：

private Properties kafkaProps = new Properties();

kafkaProps.put(“bootstrap.servers”,”broker1:9092,broker2:9092”);

kafkaProps.put(“key.serializer”,

“org.apache.kafka.common.serialization.StringSerializer”);

kafkaProps.put(“value.serializer”,

“org.apache.kafka.common.serialization.StringSerializer”);

producer = new KafkaProducer<String,String>(kafkaProps);

注：①新建Properties对象。

②因为打算把键和值定义成字符串类型，所以使用内置的StringSerializer

③在这里我们建了个新生产者对象，井为键和值设置了恰当的类型，然后把Properties对象传给它。

其他属性：

1. acks

acks参数指定了须要有多少分区副本收到消息，生产者才会认为消息写入是成功的。影响消息写入的速度和消息丢失的可能性。

①acks=0,生产者在成功写入悄息之前不会等待服务器的响应。埋头写，速度最快，消息丢失的话生产者无从得知。

②acks=1,只要集群的首领节点收到消息，生产者就会收到一个来自服务器的成功响应。如果消息无撞到达首领节点（比如首领节点崩愤，新的首领还没有被选举出来），生产者会收到一个错误响应，为了避免数据丢失，生产者会重发消息。不过，如果一个没有收到消息的节点成为新首领，消息还是会丢失。

③acks=all,所有参与复制的节点全部收到消息时，生产者才会收到一个来自服务器的成功响应。这种模式是最安全的

1. retries

retries参数的值决定了生产者可以重发消息的次数，如果达到这个次数，生产者会放弃重试并返回错误。消息是否成功写入是很关键的，retries不建议设置为0

3）对消息的顺序有严格要求的情况下

可把max.in.flight.requests.per.connection设为1 ，这样在生产者尝试发送第一批悄息时，就不会有其他的消息发送给 broker 。不过这样会严重影响生产者的吞吐量 ，所以只有在对消息的顺序有严格要求的情况下才能这么做。

使用 Avro序列化消息

Apache Avro 是种与编程语言无关的序列化格式，目的是提供一种共享数据文件的方式。

数据通过与语言无关schema来定义schema通过 JSON 来描述。在读写文件时需要用到schema, schema一般会被内嵌在数据文件里。

Avro 有一个很有意思的特性是，当 负责写消息的应用程序使用了新的 schema ，负责读消息的应用程序可以继续处理消息而无需做任何改动，这个特性使得它特别适合用在像Kafka 这样的消息系统上。

Kafka 里使用 Avro

遵循通用的结构模式并使”schema 注册表”来达到目的。schema注册表并不属于Kafka

分区

通过键实现，键有两个用途 ：可以作为消息的附加信息，也可以用来决定消息该被写到主题的哪个分区。拥有相同键的悄息将被写到同一个分区。

如果键值为 null 井且使用了默认的分区器，那么 录将被随机地发送到主题内各个可用的分区上（轮询算法）

如果键不为空，并且使用了默认的分区器，那么Kafka 会对键进行散列然后根据散列值把消息映射到特定的分区上。关键之处在于，同一个键总是被映射到同一个分区上，所以在进行映射时，我们会使用主题所有的分区，而不仅仅是可用的分区。这也意味着，如果写人数据的分区是不可用的，那么就会发生错误。

四、消费者

1、

1）消费者对象——要从一个Kafka主题读取消息井验证这些消息，然后再把它们保存起来。生产者需要用序列化器把对象转换成字节数组再发送给Kafka，类似地，消费者需要用反序列化器把从Kafka收到的字节转换成Java对象

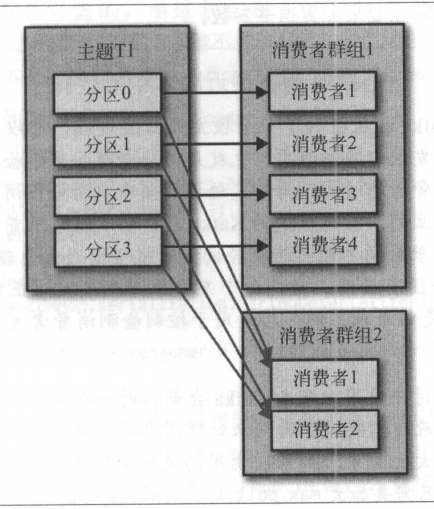
2）消息分流——写入速度超过验证速度，要对消费者进行横向伸缩。就像多个生产者可以向相同的主题写入消息一样，用多个消费者从同一个主题读取消息，

3）消费者群组—— 一个群组里的消费者订阅的是同一个主题，每个消费者接收主题一部分分区的消息。如果消费者数量>分区数量，一部分消费者会被闲置。 群组之间接受数据互不影响。

往群组里增加消费者是横向伸缩消费能力的主要方式。Kafka 消费者经常会做一些高延迟的操作，比如把数据写到数据库或 HDFS ，或者使用数据进行比较耗时的计算。在这些情况下，单个消费者无法跟上数据生成的速度，所以可以增加更多的消费者，让它们分担负载，每个消费者只处理部分分区的消息，这就是横向伸缩的主要手段。我们有必要为主题创建大量的分区，在负载增长时可以加入更多的消费者（但还是不要让消费者数量>分区数量 会被闲置）。

除了通过增加消费者来横向伸缩单个应用程序外，还经常出现多个应用程序从同个主题读取数据的情况。只要保证每个应用程序有自己的消费者群组，就能让它们获取到主题的所有消息。不同于传统的消息系统，横向伸缩Kafka消费者和消费者群组并不对性能造成负面影响。

图例：如果新增1个只包含1个消费者的群组 G2 ，那么这个消费者将从主题Tl 上接收所有的消息，与群组G1之间互不影响。群组G2可以增加更多的消费者，每个消费者可以消费若干个分区，就像群组G1那样。



2、

再均衡——当消费者或者分区数量发生变化时，部分分区的所有权从一个消费者转移到另一个消费者。为消费者群组带来了高可用性和伸缩性（添加或移除消费者）。在再均衡期间，消费者无法读取消息，造成整个群组一小段时间的不可用。另外，当分区被重新分配给另一个消费者时，消费者当前的读取状态会丢失，它有可能还需要去刷新缓存，拖慢应用。

消费者通过向被指派为群组协调器的 broker（不同的群组可以有不同的协调器）发送心跳来维持它们和群组的从属关系以及它们对分区的所有权关系。消费者会在轮询消息（为了获取消息）或提交偏移量时发送心跳。如果消费者停止发送心跳的时间足够长，会话就会过期，群组协调器认为它已经死亡，就会触发一次再均衡。

以避免出现活锁 (live lock) ，比如有时候应用程序并没有崩溃，只是由于某些原因导致无法正常运行。较新版本的Kafka，并且需要处理耗费较长时间的消息，只需要加大max.poll.interval.ms的值来增加轮均间隔的时长。

3、

分配分区的过程——消费者要加入群组时向群组协调器发送一个JoinGroup请求。第一个加入群组的消费者将成为“群主”。群主从协调器那里获得群组的成员列表（列表中包含了所有最近发送过心跳的消费者，它们被认为是活跃的），并负责给每一个悄费者分配分区。分配完毕之后，群主把分配情况列表发送给群组协调器，协调器再把这些信息发送给所有消费者。每个消费者只能看到自己的分配信息，只有群主知道群组里所有消费者的分配信息。这个过程会在每次再均衡时重复发生。

线程安全—— 一个消费者使用一个线程，同个群组里的多个消费者也是，最好把消费者的逻辑封装在自己的对象里。

退出消息轮询——通过另一个线程调用consumer.wakeup()方法。如果轮询的循环运行在主线程里。可在ShutdownHook里调用该方法。

五、深入kafka

1、集群成员关系

Kafka用zookeeper维护成员信息，每个broker都有个唯一标识符，这个标识符可以在配置文件里指定或自动生成。broker启动时，通过创建临时节点把自己的ID注册到Zookeeper。

1. 控制器

集群里第一个启动的broker，除了具有一般 broker 的功能之外，还负责分区首领的选举。

确保集群一次只有一个控制器存在：第一个启动的 broker 通过Zookeeper创建一个临时节点/controller 让自己成为控制器。其他broker启动时也会尝试创建，但会收到“节点已存在”的异常。其他 broker 在控制器节点上创建Zookeeper watch 对象，这样它们就可以收到这个节点的变更通知。

控制器被关闭或者与 Zookeeper 断开连接时，Zookeeper 上的临时节点就会消失。集群里的其他 broker 通过watch 对象得到控制器节点消失的通知，它们会尝试让自己成为新的控制器。第一个在Zookeeper里成功创建控制器节点的 broker就会成为新的控制器，其他broker收到“节点已存在”的异常，然后在新控制器节点上再次创建watch对象。新选出的控制器通过 Zookeeper 的条件递增操作获得一个全新的、数值更大的controller epoch，其他broker收到新controller epoch之后，会忽略由旧epoch发出的控制器消息。

当控制器发现 broker 已经离开集群（通过观察相关的 Zookeeper 路径），它就知道，那些失去首领的分区需要一个新首领（这些分区的首领刚好是在这个broker上）。控制器遍历这些分区，并确定谁应该成为新首领（简单来说就是分区副本列表里的下个副本），然后向所有包含新首领或现有跟随者的 broker发送请求。该请求消息包含了谁是新首领以及谁是分区跟随者的信息。随后，新首领开始处理来自生产者和消费者的请求，而跟随者开始从新首领那里复制消息。

当控制器发现 broker加入集群时，它会使用broker来检查新加入的broker是否包含现有分区的副本。如果有，控制器就把变更通知发送给新加入的broker和其他broker，新broker 上的副本开始从首领那里复制消息。

1. 复制

复制功能是 Kafka 架构的核心，因为它可以在个别节点失效时仍能保证 Kafka 的可用性和持久性。

Kafka 主题来组织数据，每个主题被分为若干个分区，每个分区有多个副本。那些副本被保存在broker上，每个broker可以保存成百上千个属于不同主题和分区的副本。副本分为首领副本和跟随者副本。

(2)跟随者副本：首领以外的副本都是跟随者副本。跟随者副本不处理来自客户端的请求，它们唯一的任务就是从首领那里复制消息，保持与首领一致的状态。如果首领发生崩渍，其中的跟随者会被提升为新首领。

(1)首领副本：每个分区都有一个首领副本，为了保证一致性，所有生产者请求和消费者请求都会经过这个副本。首领的另一个任务是搞清楚哪个跟随者的状态与自己是一致的。跟随者为了保持与首领的状态一致，在有新消息到达时尝试从首领那里复制消息，不过有各种原因会导致同步失败（网络阻塞导致复制变慢，broker奔溃 等重启之后复制才会继续）

为了与首领保持同步，跟随者向首领发送获取数据的请求，这种请求与悄费者为了读取悄息而发送的请求是一样的。首领将响应消息发给跟随者。请求消息里包含了跟随者想要获取消息的偏移量，而且这些偏移量总是有序的。

一个跟随者副本先请求淌息1 ，接着请求消息2 ，然后请求消息3 ，在收到这3个请求的响应之前，它是不会发送第4个请求消息的。如果跟随者发送了请求消息4 ，那么首领就知道它已经收到了前面3个请求的响应 通过查看每个跟随者请求的最新偏移 ，首领就会知道每个跟随者复制的进度。如果跟随者在 10s内没有请求最新的数据（设置replica.lag.time.max.ms），它将被视为**不同步**的，若首领失效 该跟随者不可能成为新首领——因为它没有包含所有的消息。相反，持续请求得到的最新悄息副本被称为**同步**的副本。在首领发生失效时，只有同步副本才有可能被选为新首领。

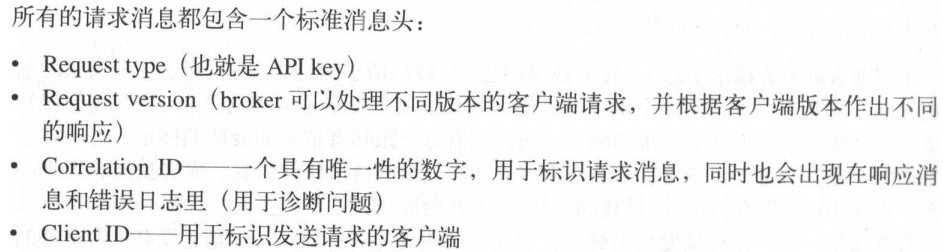
除了当前首领之外，每个分区都有一个首选首领——创建主题时选定的首领就是分区的首选首领。之所以把它叫作首选首领，是因为在创建分区时，需要在broker之间均衡首领。从分区的副本清单里第一个副本一般就是首选首领。

1. 处理请求

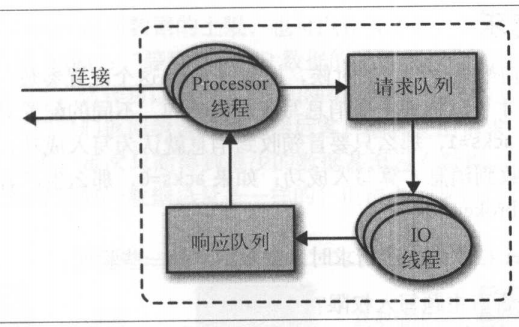
broker 的大部分工作是处理客户端、分区副本和控制器发送给分区首领的请求。

Kafka提供了一个二进制协议（基于TCP），指定了请求消息的格式以及broker如何对请求作出响应——包括成功处理请求或在处理请求过程中遇到错误。客户端发起连接并发送请求，broker 处理请求井作出响应。

broker按照请求到达的顺序处理它们——这种顺序保证让Kafka 具有了消息队列的特性，同时保证保存的消息也是有序的。



broker 会在它所监听的每个端口上运行Acceptor线程，这个线程会创建一个连接，并把它交给Processor线程去处理。Processor线程（也被叫作“网络线程”）的数量是可以配置的。网络线程负责从客户端获取请求消息，把它们放进请求队列，请求消息被放到请求队列后，IO线程会负责处理它们。然后从晌应队列取响应消息，把它们发送给客户端。过程如下图

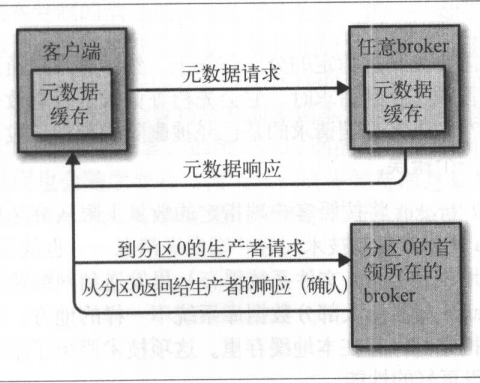


常见请求类型：①生产者请求：生产者发送的请求，它包含客户端要写入broker的消息。 ②获取请求：在消费者和跟随者副本需要从 roker 读取消息时发送的请求。

生产请求和获取请求都必须发送给分区的首领副本。如果 broker 收到一个针对特定分区的请求，而该分区的首领在另一个broker上，那么发送请求的客户端会收到一个“非分区首领”的错误响应。当针对特定分区的获取请求被发送到一个不含有该分区首领的 broker上，也会出现同样的错误。

客户端如何知道往哪儿发送请求：客户端使用了另一种请求类型，也就是元数据请求。这种请求包含了客户端感兴趣的主题列表。服务器端的响应消息里指明了这些主题所包含的分区、每个分区都有哪些副本，以及哪个副本是首领。元数据请求可以发送给任意一个 broker ，因为所有 broker 都缓存了这些信息。

一般情况下，客户端会把这些信息缓存起来，并直接往目标broker上发送生产请求和获取请求。它们需要时不时地通过发送元数据请求来刷新这些信息（刷新的时间间隔通过参数metadata.max.age.ms来配置），从而知道元数据是否发生了变更：比如新broker加入集群时，部分副本会移动到新的broker上（如下图）；，如果客户端收到“非首领”错误，它会在尝试重发请求之前先刷新元数据，因为这个错误说明了客户端正在使用过期的元数据信息，之前的请求被发到了错误的 broker上。



4.1 生产请求

配置生产者的时候，acks参数指定了需要多少个 broker 确认才可以认为一个消息写入是成功的：acks=1，只要首领收到消息就成；acks=all，所有同步副本收到消息才算成；acks=0，生产者消息发出后不需要等待broker的响应。

①包含首领副本的broker在收到生产请求时，会对请求做一些验证：发送数据的用户是否有主题写入权限？请求里包含的acks值是否有效（只允许出现0、1、all) ?如果 acks=all，是否有足够多的同步副本保证消息已经被安全写入？

②之后，消息被写入本地磁盘。Linux系统上消息会被写入文件系统缓存，不保证何时会被刷新到磁盘上。Kafka不会一直等待数据被写到磁盘上——它依赖复制功能来保证消息的持久性。

③消息被写入分区首领之后，broker开始检acks配置参数一一如果acks被设为0或1,那么broker立即返回响应；如果acks被设为all那么请求会被保存在一个叫作炼狱的缓冲区里，直到首领发现所有跟随者副本都复制了消息，晌应才会被返回给客户端。

4.2 获取请求

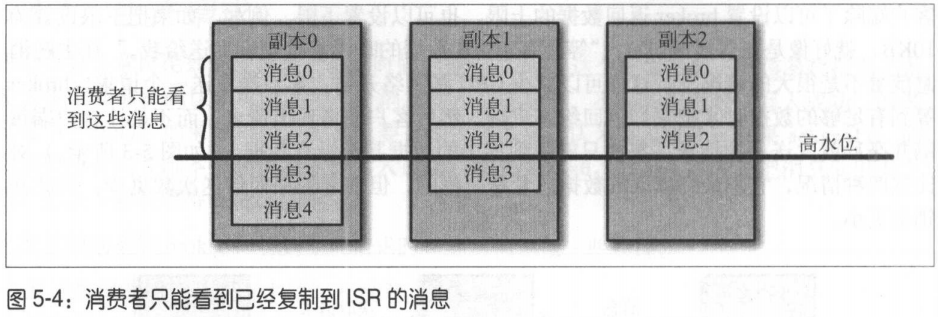
broker 处理获取请求的方式与处理生产请求的方式很相似。客户端发送请求，向 broker求主题分区里具有特定偏移量的消息，例“请把主题Test分区0偏移量从53开始的消息以及主题Test分区5偏移量从64开始的消息发给我。”客户端还可以指定broker多可以从一个分区里返回多少数据，保护内存。

请求需要先到达指定的分区首领上，然后客户端通过查询元数据来确保请求的路由是正确的。首领在收到请求时，它会先检查请求是否有效，如果客户端请求的是已经被删除的数据，或者请求的偏移量不在，那么broker将返回1个错误。

如果请求的偏移量存在，broker将按照客户端指定的数量上限从分区里读取消息，再把消息返回给客户端。Kafka使用零复制技术向客户端发送消息——也就是说，Kafka直接把消息从文件（或者更确切地说是Linux文件系统缓存）里发送到网络通道，而不需要经过任何中间缓冲区。（这是Kafka与其他大部分数据库系统不一样的地方，其他数据库在将数据发送给客户端之前会先把它们保存在本地缓存里。这项技术避免了字节复制，也不需要管理内存缓冲区，从而获得更好的性能。）

客户端除了可以设置broker返回数据的上限，也可以设置下限。例如，如果把下限设置为10KB，就好像是在告诉broker：“等到有10KB数据的时候再把它们发送给我。”在主题消息流量不是很大的情况下，这样可以减少CPU和网络开销（避免不停发送没什么数据的请求）。当然 ，我们不会让客户端一直等待 broker 累积数据。在等待了一段时间之后，就可以把可用的数据拿回处理，而不是一直等待下去。所以，客户端可以定义一个超时时间。

大部分客户端只能读取已经被写入所有同步副本的消息（跟随者副本也不行，尽管它们也是消费者——否则复制功能就无法工作）。分区首领知道每个消息会被复制到哪个副本上，在消息还没有被写入所有同步副本之前，是不会发送给消费者的——尝试获取这些消息的请求会得到空的响应而不是错误。因为还没有被足够多副本复制的消息被认为是“不安全”的一一如果首领发生崩愤，另个副本成为新首领，那么这些消息就丢失了。



4.3其他请求

目前Kafka协议可以处理20种不同类型的请求，而且会有更多的类型加入进来。协议在持续演化——随着客户端功能的不断增加，我们需要改进协议来满足需求。

例如，之前的Kafka消费者使用zookeeper来跟踪偏移量，消费者启动时通过检查保存在Zookeeper上的偏移量来找到该从哪儿开始处理消息。

现在不再用Zookeeper保存，而是是把偏移量保存在特定的Kafka主题上。为了达到这个目的，我们不得不往协议里增加几种请求类型：OffsetCommitRequst、OffsetFetchRequst和ListOffsetsRequst。现在应用程序调用CommitOffset()方法时，客户端不再把偏移量写入Zookeeper，而是往Kafka发送OffsetCommitRequst。

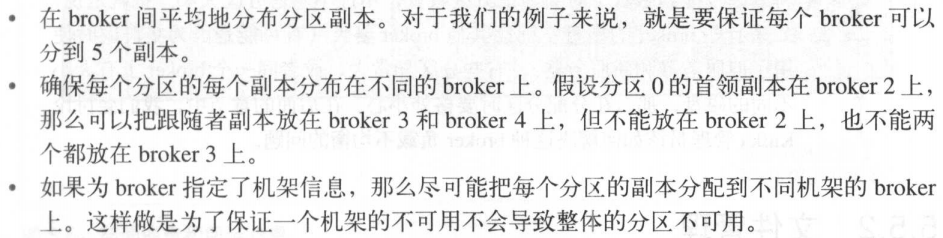
5 物理存储

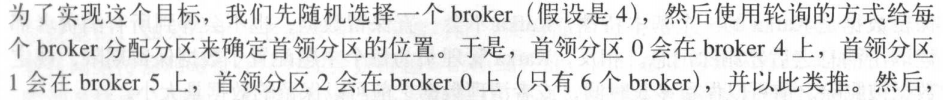
Kafka 的基本存储单元是分区。分区无法在多个broker间进行再细分，也无法在同一个broker的多个磁盘上进行再细分。所以，分区的大小受到单个挂载点可用空间的限制

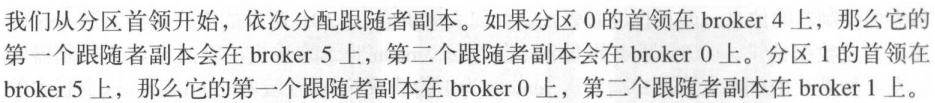
在配置 Kafka 的时候，管理员指定了用于存储分区的目录清单——也就是log.dirs参数的值（不要把它与存放错误日志的目录混淆了，日志目录是配置在log4j.properties文件里的）。该参数一般会包含每个挂载点的目录。

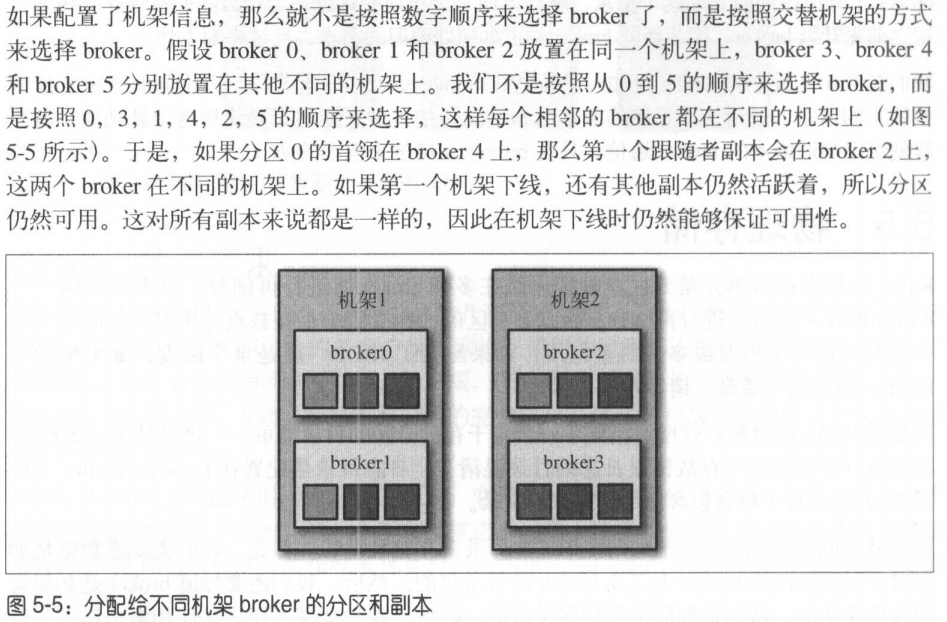
5.1 分区分配

在创建主题时，Kafka首先会决定如何在broker间分配分区。假设你有6个broker ，打创建一个包含10个分区的主题，并且复制系数为3。那么Kafka就会有30个分区副本，它们可以被分配给6个broker。在进行分区分配时，我们要达到如下的目标。









为分区和 副本选好合适的 broker 之后，接下来要决定这些分区应该使用哪个目录。我们单独为每个分区分配目录，规则很简单：计算每个目录里的分区数量，新的分区总是被添加到数量最小的那个目录里。也就是说，如果添加了一个新磁盘，所有新的分区都会被创建到这个磁盘上。因为在完成分配工作之前，新磁盘的分区数量总是最少的。

5.2 文件管理

Kafka 管理员为每个主题配置了数据保留期限，规数据被删除之前可以保留多长时间，或者清理数据之前可以保留的数据量大小。

因为在一个大文件里查找和删除消息是很费时的，也很容易出错，所以我们把分区分成若干个片段，默认情况下，每个片段包含1GB或一周的数据，以较小的那个为准。在broker往分区写入数据时，如果达到片段上限，就关闭当前文件，井打开一个新文件。

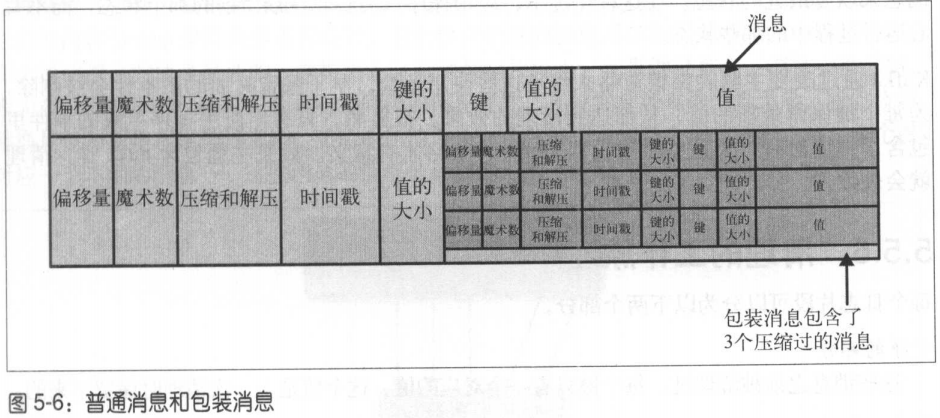
当前正在写入数据的片段叫作活跃片段。活动片段永远不会被删除，所以如果你要保留数据1天，但片段里包含了5天的数据，那么这些数据会被保留5天，因为在片段被关闭之前这些数据无法被删除。

5.3 文件格式

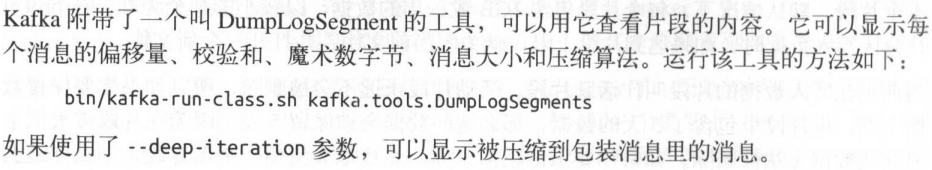
我们 Kafka 的消息和偏移 保存在文件里。保存在磁盘上的数据格式与从生产者发送过来或者发送给消费者的悄息格式是一样的。因为使用了相同的消息格式进行磁盘存储和网络传输 Kafka 可以使用零复制技术给消费者发送消息，同时避免了对生产者已经压缩过的消息进行解压和再压缩。

除了键、值和偏移量外， 消息里还包含了消息大小、校验和、消息格式版本号、压缩算法(Snappy、GZip或LZ4）和时间戳（可配置：生产者发送消息的时间 或是消息到达broker的时间）

如果生产者发送的是压缩过的消息，那么同一个批次的消息会被压缩在一起，被当作“包装消息”进行发送。broker 就会收到一个这样的消息，然后再把它发送给消费者。消费者 解压这个消息之后，会看到整个批次的消息，它们都有自己的时间戳和偏移量。



也就是说，如果在生产者端使用了压缩功能（极力推荐），那么发送的批次越大，就意味着在网络传输和磁盘存储方面会获得越好的压缩性能，同时意味着如果修改了消费者使用的消息格式（例如，在消息里增加了时间戳），那么网络传输和磁盘存储的格式要随之修改，而且broker要知道如何处理包含了两种消息格式的文件。



5.4 索引

消费者可以从Kafka的任意可用偏移量位置开始读取消息。为了帮助broker更快地定位到指定的偏移量，Kafka为每个分区维护了一个索引。索引把偏移量映射到片段文件和偏移量在文件里的位置。

索引也被分成片段，所以在删除消息时，也可以删除相应的索引 Kafka 不维护索引的校验和。如果索引出现损坏， Kafka 会通过重新读取消息并录制偏移量和位置来重新成索引。

5.5 清理

一般情况下，Kafka会根据设置的时间保留数据，把超过时效的旧数据删除掉。早于保留时间的旧事件会被删除为每个键保留最新的值，从而达到清理的效果。很显然，只有当应用程序生成的事件包含了键值对时，为这些主题设置compact策略才有意义。如果主题包含null键，清理就会失败。

5.6 清理的工作原理

5.7 被删除的事件

5.8 何时会清理主题

六

6.3 broker配置

broker 个配置参数会影响 Kafka 消息存储的可靠性。

6.3.1 复制系数

主题级别的配置参数是replication.factor，而在broker级别则可以通过default.replication.factor来配置自动创建的主题。

假设主题的复制系数N，也就是说每个分区总共会被N个不同的broker复制N次。如果复制系数为N，那么在N-1个broker失效的情况下，仍然能够从主题读取数据或向主题写入数据。所以，更高的复制系数会带来更高的可用性、可靠性和更少的故障。但是N个副本相应的会占用N倍的磁盘空间。要在可用性和存储硬件之间做出权衡，这要看主题的重要程度，以及你愿意付出多少成本来换取可用性。通常设为3已经足够安全。

6.3.2 不完全的首领选举

unclean.leader.election 只能在broker级别（实际上是在集群范围内）进行配置，它的默认值是true。

“完全”的选举：当分区首领不可用时，一个同步副本会被选为新首领，

如果在首领不可用时 其他副本都是不同步的，如果我们允许不同步的副本成为首领，那么就要承担丢失数据和出现数据不一致的风险。如果不允许它们成为首领，那么就要接受较低的可用性，因为我们必须等待原先的首领恢复到可用状态。

6.3.3 最少同步副本

在主题级别和broker级别上，这个参数都叫min.insync.replicas

尽管为主题配置了3个副本，还是会出现只有1个同步副本的情况。如果这个同步副本变为不可用，我们必须在可用性和一致性之间作出选择。若同步副本的数量低于配置值，broker就会停止接受生产者的请求。尝试发送数据的生产者会收到NotEnoughReplicasException异常。此时消费者仍可以读取已有的数据，但数据是只读的。

6.4 在可靠的系统中使用生产者

核心：1）根据可靠性需求配置恰当的acks值 2）在参数配置和代码里正确处理错误。

6.4.1 发送确认

3种确认模式：

1）acks=0，生产者只管发出消息，认为消息都被成功写入kafka。最快，吞吐量和带宽利用率最高，但是一定会丢失一些消息。

2）acks=1，若首领在收到消息并把它写入到分区数据文件（不 定同步到磁盘上）时会返回确认或错误响应。如果发生正常的首领选举，生产者会在选举时收到LeaderNotAvailableException异常，如果生产者能恰当地处理这个错误，它会重试发送悄息，最终消息会安全到达新的首领那里。不过在这个模式下仍然有可能丢失数据，比如消息已经成功写入首领，但在消息被复制到跟随者副本之前首领发生崩溃。

3）acks=all，等待所有同步副本都收到消息，如果和min.insync.replicas参数结合起来，就可以决定在返回确认前至少有多少个副本能够收到消息。这是最保险的做法。

6.4.2 配置生产者的重试参数

生产者需要处理的错误包括两部分，一部分是生产者可以自动处理的错误，还有一部分是要开发者手动处理的错误。生产者向broker发送消息时，broker可以返回一个成功晌应码或者一个错误响应码，错误响应码分为两种：自动处理（可重试错误）、手动处理（不可重试错误），如果broker 返回的错误可以通过重试来解决，那么生产者会自动处理这些错误，可以通过配置设置保持重试的次数。

注意，重试发送一个已经失败的消息会带来 些风险，如果两个消息都写入成功，会导致消息重复。例如，生产者因为网络问题没有收到 broker的确认，但实际上消息已经写入成功，生产者会认为网络出现了临时故障，就重试发送该消息。现实中的很多应用程序在消息里加入唯一标识符，用于检测重复消息，消费者在读取消息时可以对它们进行清理。还有一些应用程序可以做到消息的“幕等”，也就是说，即使出现了重复消息，也不会对处理结果的正确性造成负面影响（改变数据的描述方法）。

6.4.3 额外的错误处理

用生产者内置的重试机制可以在不造成消息丢失的情况下轻松地处理大部分错误，不过，开发人员仍然需要处理其他类型的错误，包括：1）不可重试的broker错误，例如消息大小错误、认证错误等 2）在消息发送之前发生的错误，例如序列化错误 3）在生产者达到重试次数上限时或者在消息占用的内存达到上限时发生的错误。

6.5 在可靠的系统里使用消费者

只有已提交消息对消费者是可用的，它是指已经被写入所有同步副本并且对消费者可见的消息 具有一致性，消费者唯一要做的是跟踪哪些消息是已经读取过的，哪些是还没有读取过的。这是在读取消息时不丢失悄息的关键。而己提交偏移量是指消费者发送给 Kafka 的偏移量，用于确认它已经收到并处理好的消息位置。

在从分区读取数据时，消费者会获取 批事件，检查这批事件里最大的偏移量，然后从这个偏移量开始读取另外 批事件。这样可以保证消费者总能以正确的顺序获取新数据，不会错过任何事件。

6.5.1 消费者的可靠性配置

1）group.id：如果两个消费者具有相同的group.id井且订阅了同一个主题，那么每个消费者会分到主题分区的一个子集，就是说它们只能读到所有消息的一个子集（不过群组会读取主题所有的消息）如果你希望消费者可以看到主题的所有消息，那么需要为它们设置唯一的group.id。

2）auto.offset.reset： 指定了在没有偏移量可提交时（比如消费者第启动时）或者请求的偏移量在broker上不存在时消费者怎么处理。这个参数有两种配置：①一种是earliest，消费者从分区开始位置读取数据，不管偏移量是否有效，这样会导致消费者读取大量的重复数据，但可以保证最少的数据丢失。 ②一种是latest，消费者会从分区的末尾开始读取数据，这样可以减少重复处理消息，但很有可能会错过一些消息。

3）enable.auto.commit： 让消费者基于任务调度自动提交偏移量 ，也可以在代码里手动提交偏移量。自动提交的优点：在实现消费者逻辑的时候可以少考虑一些，轮询处理数据过程中可以保证只提交已经处理过的偏移量，缺点是：无法控制重复处理消息，且如果由后台进程来处理消息，自动提交机制可能会在消息还没有处理完毕时就提交偏移量。

4）auto.commit.interval.ms： 如果选择了自动提交偏移量，可以通过该参数配置提交的频度，默认5ms一次，通常频繁提交会增加额外的开销，但也会降低重复处理消息的概率。

6.5.2 显式提交偏移量

几个在开发具有可靠性的消费者应用程序 需要注意的事项

1. 总是在处理完事件后再提交偏移量：如果所有的处理都是在轮询里完成，并且不需要在轮询之间维护状态，可以使用自动提交，或者在轮询结束时进行手动提交。
2. 提交频度是性能和重复消息数量之间的权衡：
3. 确保对提交的偏移量心里有数：在轮询过程中提交偏移量有一个不好的地方，就是提交的偏移量有可能是读取到的最新偏移量，而不是处理过的最新偏移量
4. 再均衡： 设计应用程序时要注意处理消费者的再均衡问题。一般在分区被撤销之前提交偏移量，井在分配到新分区时清理之前的状态。
5. 消费者可能需要重试： 有时候在进行轮询后，有些消息没被完全处理，你想稍后再来处理。假设要把Kafka的数据写到数据库里，不过那时数据库不可用，于是想稍后重试。注意，你提交的是偏移量，而不是对消息的“确认”，这个与传统的发布和订阅消息系统不同。例如：若记录#30处理失败 而#31处理成功，此时便不应该提交#31，那样会导致#31以内的偏移量都被提交 （包括#30）那就不对了。不过可以通过以下两种模式来解决。
6. 在遇到可重试错误时，提交最后一个处理成功的偏移量，然后把还没有处理好的消息保存到缓冲区里（这样下个轮询就不会把它们覆盖掉），调用消费者的pause()方法来确保其他的轮询不会返回数据（不需要担心在重试时缓冲区隘出），在保持轮询的同时尝试重新处理。。如果重试成功，或者重试次数达到上限井决定放弃，那么把错误记录下来井丢弃消息，然后调用resume()方能让消费者继续从轮询里获取新数据。
7. 遇到可重试错误时，把错误写入一个独立的主题，然后继续。一个独立的消费者群组负责从该主题上读取错误消息，井进行重试，或者使用其中的一个消费者同时从该主题上读取错误消息并进行重试，不过在重试时需要暂停该主题。
8. 消费者可能需要维护状态： 有时候你希望在多个轮询之间维护状态，例如，你想计算消息的移动平均数，希望在首次轮询之后计算平均数，然后在后续的轮询中更新这个结果。建议尝试一下KafkaStreams这个类库，它为聚合、连接、时间窗和其他复杂的分析提供了高级的DSL API。
9. 长时间处理：　有时候处理数据需要很长时间：你可能会从发生阻塞的外部系统获取信息，或者把数据写到外部系统，或者进行个非常复杂的计算。但是，暂停轮询的时间不能超过几秒钟，即使不想获取更多的数据也要保持轮询，客户端才能往broker发送心跳。

常见的做法是使用 个线程地来处理数据，因为使用多个线程可以进行并行处理，从而加快处理速度。在把数据移交给线程地去处理之后，你就可以暂停消费者，然后保持轮询，但不获取新数据，直到工作线程处理完成。在工作线程处理完成之后 ，可以让消费者继续获取新数据。

1. 仅一次传递： 有些应用程序不仅仅需要“至少一次”，还需要“仅一次”语义。，消费者还是有一些办越可以保证Kafka里的每个消息只被写到外部系统一次（但不会处理向Kafka 入数据时可能出现的重复数据）。

最简单且最常用的办能是把结果写到一个支持唯一键的系统里，比如键值存储引擎、关系型数据库、ElasticSearch或其他数据存储引擎。在这种情况下，要么消息本身包含一个唯一键（通常都是这样），要么使用主题、分区和偏移量的组合来创建唯一键——它们的组合可以唯一标识一个Kafka记录。相同消息覆盖前一条，这个模式被叫作幂等性写入。

若写入消息的系统支持事务， 那么就可以使用关系型数据库，不过 HDFS 里有一些被重新定义过的原子操作也经常用来达到相同的目的。把消息和偏移量放在同一个事务里，这样它们就能保持同步。在消费者启动时，它会获取最近处理过的消息偏移，然后调用seek （）方也从该偏移量位置继续读取数据。

6.6 验证系统可靠性

3个层面的可靠性验证：配置验证、应用程序验证、生产环境的应用程序监控。

6.6.1 验证配置

从应用程序里可以很容易对broker客户端配置进行验证，建议这么做的原因：①验证配置是否满足需求 ②帮助你理解系统的行为，了解系统的真正行为是什么，了解你对Kafka 基本准则的理解是否存在偏差，然后加以改进，同时了解这些准则是如何被应用到各种场景里的。

Kafka 提供了两个重要的工具用于验证配置：org.apache.kafka.tools里的VerifiableProducer和VerifiableConsumer这两个类。我们可以从命令行运行这两个类，或者把它们嵌入到自动化测试框架里。

核心思想：VerifiableProducer生成一系列消息，包含从1到你指定的某个数字。你可以使用与生产者相同的方式来配置VerifiableProducer，比如配置相同的acks重试次数和消息生成速度。运行VerifiableProducer时会把每个消息是否成功发送到broker的结果打印出来。VerifiableConsumer负责读取事件（VerifiableProducer生成）并按顺序打印这些事件。也打印已提交的偏移量和再均衡的相关信息。

选择一个场景打开VP、VC开始测试，例如，停掉正在接收消息的分区首领。如果期望在一个短暂的暂停之后状态恢复正常并且没有任何数据丢失，那么只要确保生产者生成的数据个数与消费者读取的数据个数是匹配的就可以了。

6.6.2 应用程序验证

应用程序的验证包括检查自定义的错误处理代码、偏移量提交的方式再均衡监听器及其他使用Kafka客户端的地方。

6.6.3 在生产环境监控可靠性

这些监控是为了确保数据按照期望的方式流动。除了监控集群的健康状况之外，监控客户端和数据流也是很重要的。

1）Kafka的Java客户端包含了JMX度量指标，这些指标可以用于监控客户端的状态和事件；

2）对于生产者，最重要的两个可靠性指标是消息的error-rate和retry-rate（聚合过的）如果这两个指标上升 说明系统出了问题；

3）对于消费者来说最重要的是consumer-lag，它表明7了消费者的处理速度与最近提交到分区里的偏移量之间还有多少差距。理想状态下此指标应该为0，即消费者总能读到最新的消息，但在实际当中，因为poll（）方法会返回很多消息，消费者在获取更多数据之前需要花一些时间来处理它们，所以该指标会有些波动。关键是要确保消费者最终会赶上去 ，而不是越落越远。

为了做到更好的监控，我们可以增加个“监控消费者”，这个消费者订阅一个特别的主题，它只进行消息的计数操作，井把数值与生成的消息、数量进行对比，这样我们就可以在没有消费者的情况下仍然能够准确地监控生产者。为确保消息被及时读取，应用程序需要记录生成消息的数量（用每秒多少个消息来表示），而消费者需要记录已读取消息的数量（n个消息/秒）以及消息生成时间到当前时间（读取消息的时间）之间的时间差。然后用工具比较生产者和消费者记录的消息数量（确保没丢消息）确保时间差不超出允许的范围。

七 构建数据管道