# 概述

## 定义

Kafka 是一个**分布式**的基于**发布/订阅**模式的消息队列（Message Queue），主要应用与大数据**实时**处理领域。主要设计目标如下：

1. 以时间复杂度为O(1)的方式提供消息持久化能力，即使对TB级以上数据也能保证常数时间的访问性能
2. 高吞吐率。即使在非常廉价的商用机器上也能做到单机支持每秒100K条消息的传输
3. 支持Kafka Server间的消息分区，及分布式消费，同时保证每个partition内的消息顺序传输
4. 同时支持离线数据处理和实时数据处理

Zookeeper是一种在分布式系统中被广泛用来作为：分布式状态管理、分布式协调管理、分布式配置管理、和分布式锁服务的集群。Kafka增加和减少服务器都会在Zookeeper节点上触发相应的事件kafka系统会捕获这些事件，进行新一轮的负载均衡，客户端也会捕获这些事件来进行新一轮的处理。

## 基本概念

### 消息/记录

消息：Kafka中的数据单元被称为消息，也被称为记录，可以把它看作数据库表中某一行的记录。

### 批次

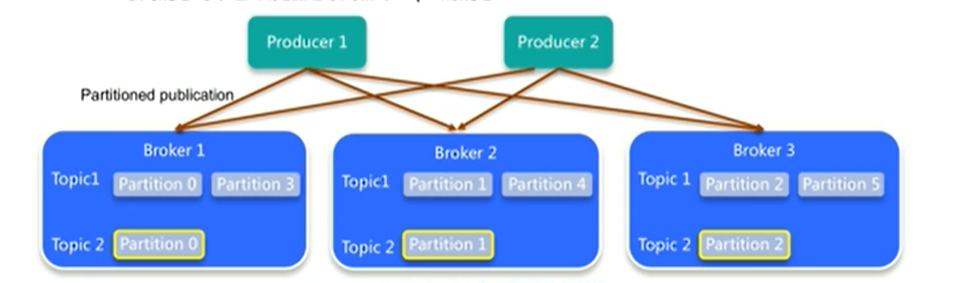
批次：为了提高效率，消息会分批次写入Kafka，批次就代指的是一组消息。

### Topic/主题

消息的种类称为主题（Topic）,可以说一个主题代表了一类消息。相当于是对消息进行分类。主题就像是数据库中的表。

另一种描述；

Topic在逻辑上可以被认为是一个queue。每条消费都必须指定它的topic，可以简单理解为必须指明把这条消息放进哪个queue里。



说明：

1. 逻辑概念，同一个tipic的消息可以分布在一个或多个节点（broker）上
2. 一个topic包含一个或多个partition
3. 每条消息都属于且仅属于一个topic
4. Producer发布数据时，必须指定将该消息发布到哪一个topic
5. Consumer订阅消息时，也必须指定订阅哪个topic的消息

### Partition/分区

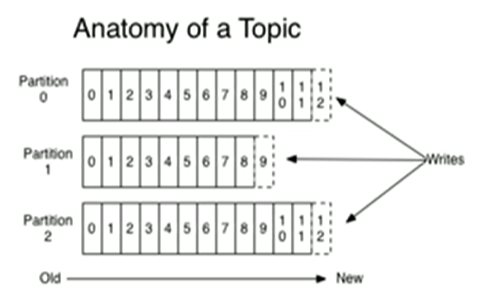
为了使得Kafka的吞吐率可以水平扩展，物理上把topic分成一个或多个partition，每个partition在物理上对应一个文件夹，分区是kafka消息队列组织的最小单位，一个分区可以看作是一个FIFO（ First Input First Output的缩写，先入先出队列）的队列。

说明：

kafka分区是提高kafka性能的关键所在，当你发现你的集群性能不高时，常用手段就是增加Topic的分区，分区里面的消息是按照从新到老的顺序进行组织，消费者从队列头订阅消息，生产者从队列尾添加消息（即append方式）。

主题可以被分为若干个分区（partition），同一个主题中的分区可以不在一个机器上，有可能会部署在多个机器上，由此来实现kafka的伸缩性，单一主题中的分区有序，但是无法保证主题中所有的分区有序。

分区具体在服务器上面表现起初就是一个目录，一个主题下面有多个分区，这些分区会存储到不同的服务器上面，或者说，其实就是在不同的主机上建了不同的目录。



说明：

1. 物理概念，一个partition只分布于一个broker上（不考虑备份）
2. 一个partition物理上对应一个**文件夹**
3. 一个partition包含多个segment（segment对用户透明）
4. 一个segment对应一个**文件**
5. segment是由一个个不可改变的记录组成（不可被修改和删除 ）
6. 记录只会被append到segment中，不会被单独删除或修改
7. 清除过期日志时，直接删除一个或多个segment

注：kafka中的记录无法修改和删除，所以kafka删除数据的时候直接删除segment

**这里kafka采用磁盘顺序写的原因，就是因为segment的这种append机制。**

跟数据库里面的分区差不多，是为了提高性能。至于为什么提高了性能，很简单，多个分区多个线程，多个线程并行处理肯定会比单线程好得多。Topic和partition像是HBASE里的table和region的概念，table只是一个逻辑上的概念，真正存储数据的是region，这些region会分布式地存储在各个服务器上面，对应于kafka，也是一样，Topic也是逻辑概念 ，而partition就是分布式存储单元。这个设计是保证了海量数据处理的基础。我们可以对比一下，如果HDFS没有block的设计，一个100T的文件也只能单独放在一个服务器上面，那就直接占满整个服务器了，引入block后，大文件可以分散存储在不同的服务器上。

注意：

1. 分区会有单点故障问题，所以我们会为每个分区设置副本数
2. 分区的编号是从0开始的

### Producer/生产者

向主题（也可以认为是broker）发布消息的客户端应用程序称为生产者（Producer），生产者用于持续不断的向某个主题发送消息。

### Consumer/消费者

订阅主题消息的客户端程序称为消费者（Consumer），消费者用于处理生产者产生的消息。

### Consumer Group/消费者群组

生产者与消费者的关系就如同餐厅中的厨师和顾客之间的关系一样，一个厨师对应多个顾客，也就是一个生产者对应多个消费者，消费者群组（Consumer Group）指的就是由一个或多个消费者组成的群体。

Consumer Group：消费者组（CG），消费者组内每个消费者负责消费不同分区的数据，提高消费能力。**一个分区只能由组内一个消费者消费，消费者组之间互不影响**。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。

我们在消费数据时会在代码里面指定一个group.id，这个id代表的是消费组的名字，而且这个group.id就算不设置，系统也会默认设置

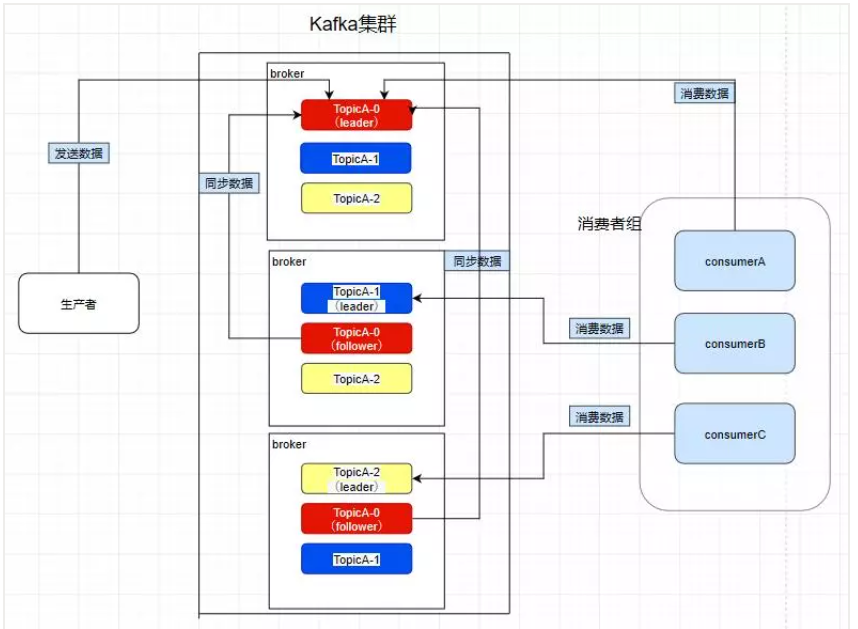
conf.setProperty("group.id","tellYourDream")

我们所熟知的一些消息系统一般来说会这样设计，就是只要有一个消费者去消费了消息系统里面的数据，那么其余所有的消费者都不能再去消费这个数据。可是kafka并不是这样，比如现在consumerA去消费了一个topicA里面的数据。

consumerA:  
 group.id = a  
 consumerB:  
 group.id = a  
  
 consumerC:  
 group.id = b  
 consumerD:  
 group.id = b

再让consumerB也去消费TopicA的数据，它是消费不到了，但是我们在consumerC中重新指定一个另外的group.id，consumerC是可以消费到topicA的数据的。而consumerD也是消费不到的，所以在kafka中，不同组可有唯一的一个消费者去消费同一主题的数据 。所以消费者组就是让多个消费者并行消费信息而存在的，而且它们不会消费到同一个消息，如下，consumerA，B，C是不会互相干扰的。

consumer group:a  
 consumerA  
 consumerB  
 consumerC



如图，因为前面提到过了消费者会直接和leader建立联系，所以它们分别消费了三个leader，所以一个分区不会让消费者组里面的多个消费者去消费 ，但是在消费者不饱和的情况下，一个消费者是可以去消费多个分区的数据的 。

### Consumer Offset/偏移量

偏移量（Consumer Offset）是一种元数据，它是一个不断递增的整数值，监控数据消费到什么位置，用来记录消费者发生重平衡时的位置，以便用来恢复数据。

### Broker

一个独立的Kafka服务器就被称为 broker，**broker接收来自生产者的消息，为消息设置偏移量，并提交消息到磁盘保存**。

**另一种描述：**

AMQP服务端（broker）：用来接收生产者发送的消息并将这些消息路由给服务器中的队列，便于kafka将生产者发送的消息，动态的添加到磁盘并给每一条消息一个偏移量，所以对于kafka一个broker就是一个应用程序的实例。

一个集群由多个Broker组成。一个Broker可以容纳多个Topic。

### broker集群

broker是集群 的组成部分，broker集群由一个或多个broker组成，每个集群都有一个broker同时充当了集群控制器的角色（自动从集群的活跃成员中选举出来）。

### Replica/副本

Kafka中消息的备份又叫做副本（Replica），副本的数量是可以配置的，Kafka 定义了两类副本：领导者副本（Leader Replica）和追随者副本（Follower Replica），前者对外提供服务，后者只是被动跟随。

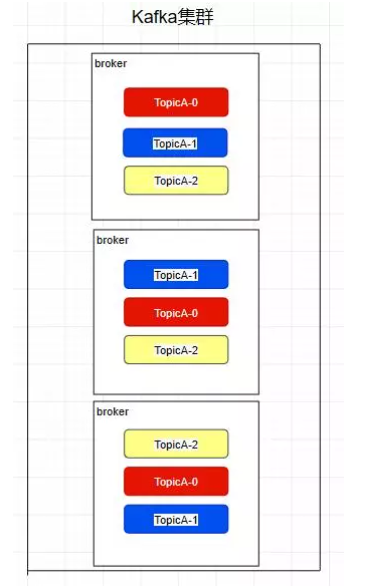
副本为实现备份的功能，保证集群中的某个节点发生故障时，该节点上的Partition数据不丢失，且Kafka仍然能够继续工作，Kafka提供了副本机制，一个Topic的每个分区都有若干个副本，一个Leader和若干个Follower。kafka中的partition为了保证数据安全，所以每个partition可以设置多个副本。

注意：kafka在0.8版本以前是没有副本机制的，所以在面对服务器宕机的突发情况时会丢失数据，所以尽量避免使用这个版本之前的kafka。

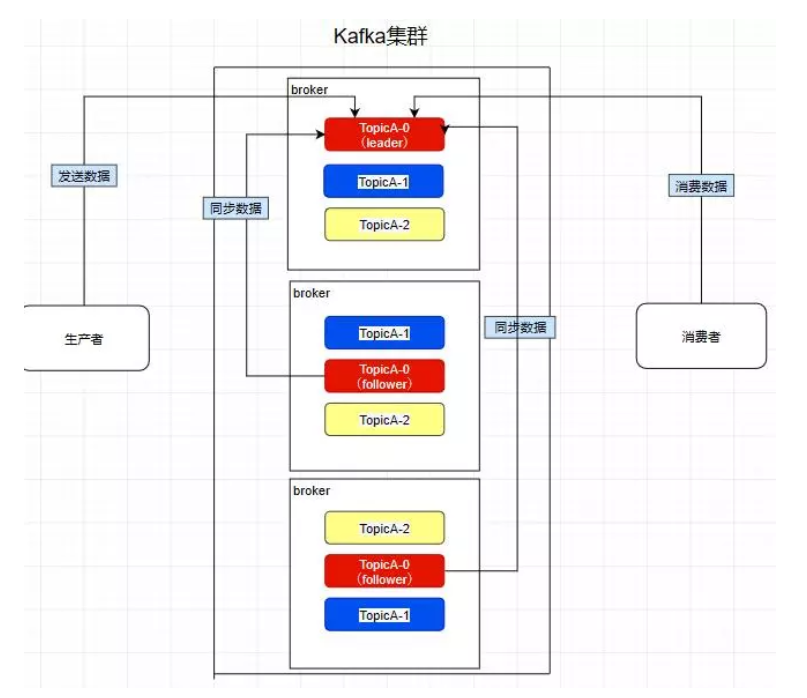
#### Leader

每个分区多个副本的“主”副本，生产者发送数据的对象，以及消费者消费数据的对象，都是Leader。

我们对分区0,1,2分别设置3个副本（其实设置两个副本是比较合适的）：



其实每个副本都是有角色之分的，它们会选取一个副本作为leader，而其余的作为follower，我们的生产者在发送数据的时候，是直接发送到leader partition里面 ，然后follower partition会去leader那里自行同步数据，消费者消费数据的时候，也是从leader那去消费数据的 。



#### Follower

每个分区多个副本的“从”副本，实时从Leader中同步数据，保持和Leader数据的同步。Leader发生故障时，某个Follower还会成为新的Leader。

#### Controller

熟知一个规律：在大数据分布式文件系统里面，95%的都是主从式的架构，个别是对等式的架构，比如ElasticSearch。kafka也是主从式的架构，主节点就叫controller，其余的为从节点，controller是需要和zookeeper进行配合管理整个kafka集群。

### Rebalance/重平衡

消费者组内某个消费者实例挂掉后，其他消费者实例自动重新分配订阅主题分区的过程。Rebalance是Kafka消费者端实现高可用的重要手段。

### ZooKeeper

Kafka集群能够正常工作，需要依赖于ZooKeeper，ZooKeeper帮助Kafka存储和管理集群信息。

kafka严重依赖于zookeeper集群。所有的broker在启动的时候都会往zookeeper进行注册，目的就是选举出一个controller，这个选举过程非常简单粗暴，就是一个谁先谁当的过程，不涉及什么算法问题。那成为controller之后，它会监听zookeeper里面的多个目录。例如有一个目录/brokers/，其他从节点往这个目录上注册（就是往这个目录上创建属于自己的子目录而已） 自己，这时命名规则一般是它们的id编号，比如/brokers/0,1,2注册时各个节点必定会暴露自己的主机名，端口号等等的信息，此时controller就要去读取注册上来的从节点的数据（通过监听机制），生成集群的元数据信息，之后把这些信息都分发给其他的服务器，让其他服务器能感知到集群中其它成员的存在 。此时模拟一个场景，我们创建一个主题（其实就是在zookeeper上/topics/topicA这样创建一个目录而已），kafka会把分区方案生成在这个目录中，此时controller就监听到了这一改变，它会去同步这个目录的元信息，然后同样下放给它的从节点，通过这个方法让整个集群都得知这个分区方案，此时从节点就各自创建好目录等待创建分区副本即可。这也是整个集群的管理机制。

注：集群的高可用，选主通过Zookeeper的ZAB协议完成（改进的Paxos）。

## 设计准则/特点

Kafka 本质上是一个 MQ（Message Queue），使用消息队列的好处？

解耦：允许我们独立的扩展或修改队列两边的处理过程。

可恢复性：即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

缓冲：有助于解决生产消息和消费消息的处理速度不一致的情况。

灵活性&峰值处理能力：不会因为突发的超负荷的请求而完全崩溃，消息队列能够使关键组件顶住突发的访问压力。

异步通信：消息队列允许用户把消息放入队列但不立即处理它。

**高吞吐、低延迟：**kakfa最大的特点就是收发消息非常快，kafka每秒可以处理几十万条消息，它的最低延迟只有几毫秒。  
 **高伸缩性：**每个主题(topic)包含多个分区(partition)，主题中的分区可以分布在不同的主机(broker)中。  
 **持久性、可靠性：**Kafka能够允许数据的持久化存储，消息被持久化到磁盘，并支持数据备份防止数据丢失，Kafka底层的数据存储是基于Zookeeper存储的，Zookeeper我们知道它的数据能够持久存储。  
 **容错性：**允许集群中的节点失败，某个节点宕机，Kafka集群能够正常工作。  
 **高并发：**支持数千个客户端同时读写。

# 消息队列

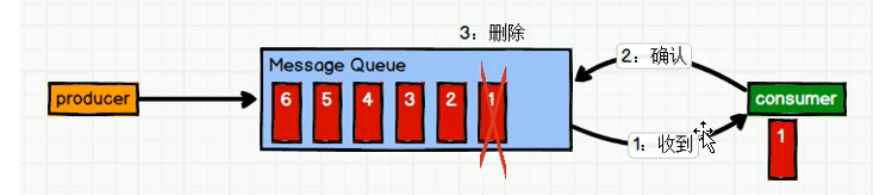
## 点对点模式

消息队列有两种模式：

1. 点对点模式（一对一，消费者主动拉取数据，消息收到后消息清除）

消息生产者生产消息发送到Queue中，然后消息消费者从Queue中取出并且消费消息。消息被消费以后，queue中不再有存储，所以消息消费者不可能消费到已经被消费的消息。Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。

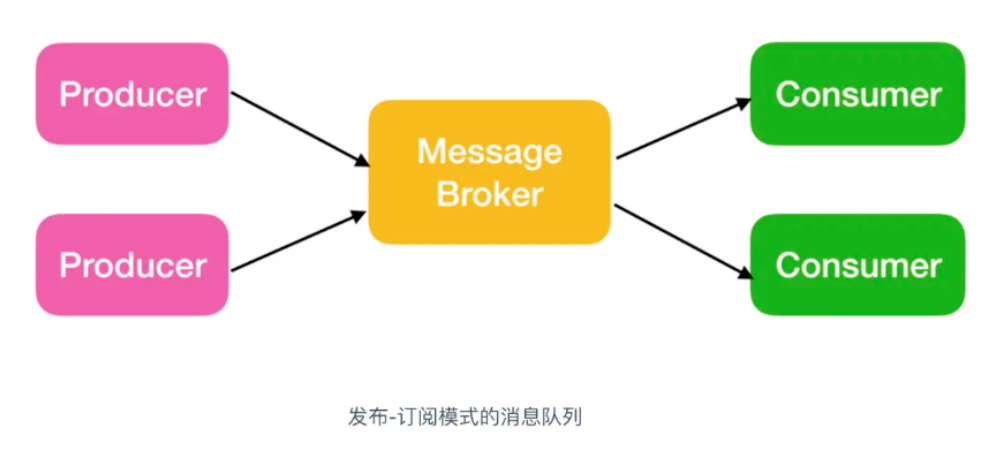


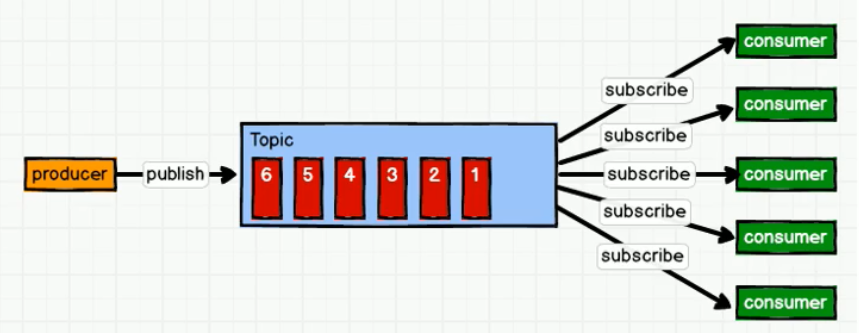


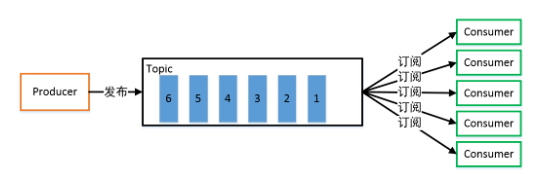
## 发布-订阅模式

1. 发布/订阅模式（一对多，消费者消费数据之后不会清除消息）

消息生产者（发布）将消息发布到topic中，同时有多个消费者（订阅）消费该消息。和点对点方式不同，**发布到topic的消息会被所有订阅者消费**。







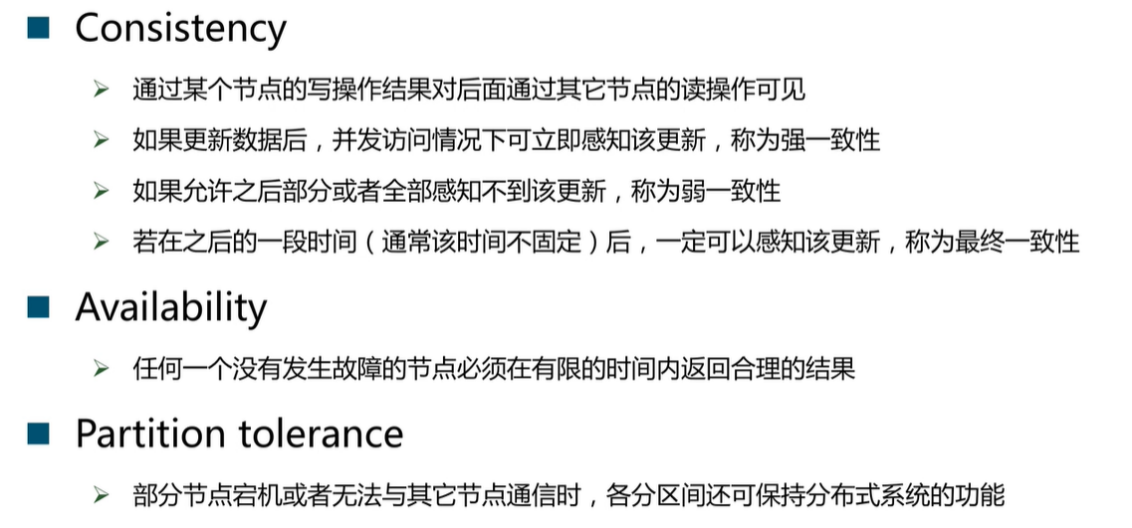
一对多，生产者将消息发布到Topic中，有多个消费者订阅该主题，发布到Topic的消息会被所有订阅者消费，被消费的数据不会立即从Topic清除。

## CAP

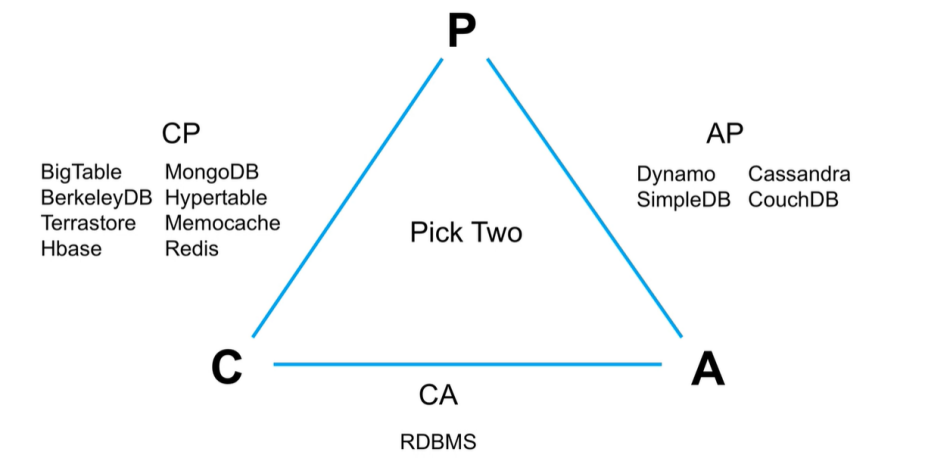
参考：

<https://www.bilibili.com/video/BV1bt411y72u?p=5&spm_id_from=pageDriver>

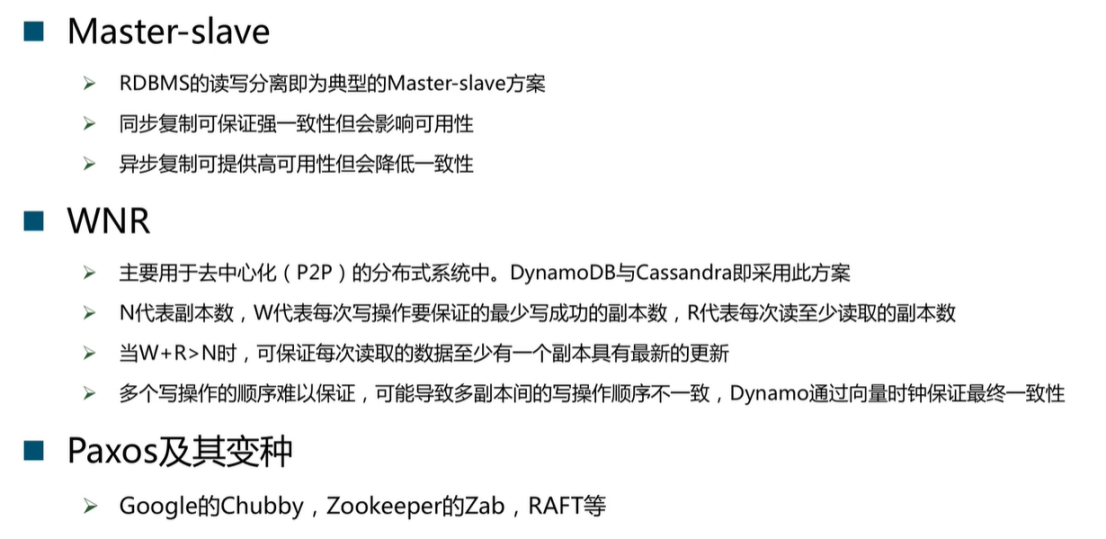
CAL理论：



一般分区容忍性都有保证，很多时候都是在可用性和一致性之间权衡。

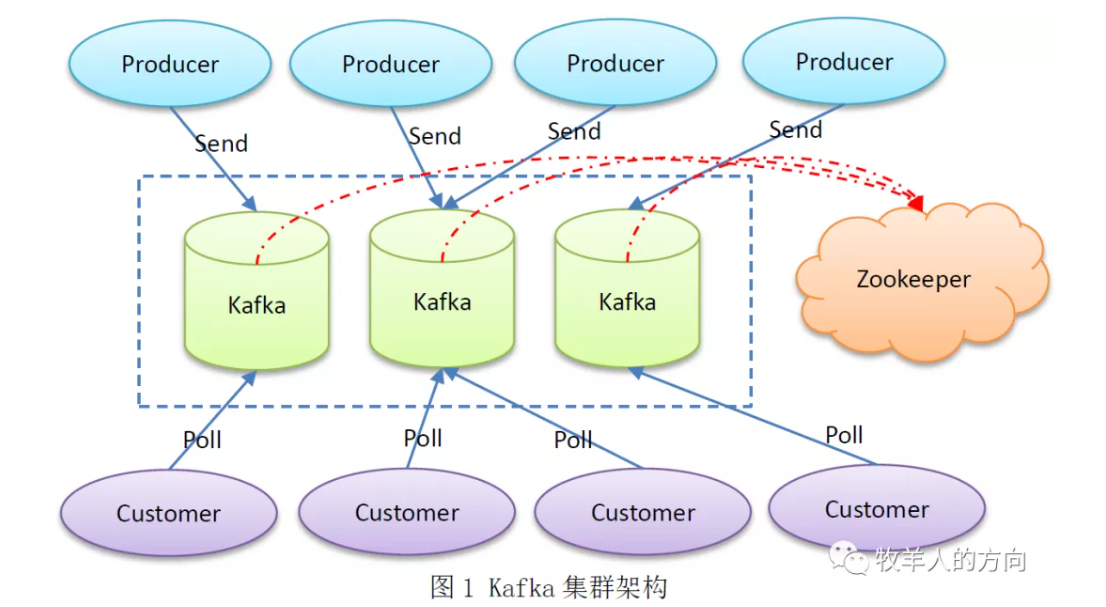


方案：



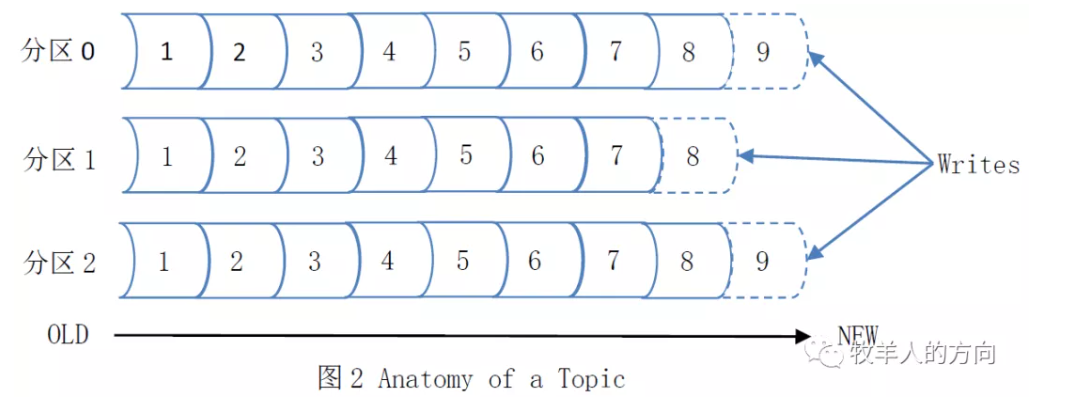
# 架构

Kafka集群架构：



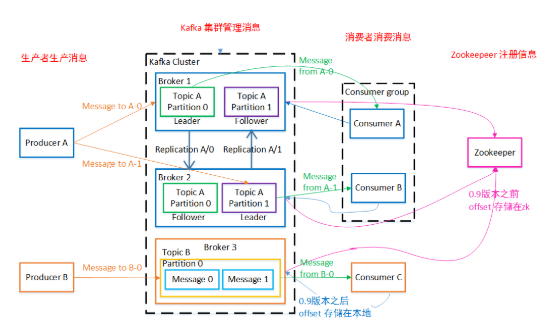
如上图所示，一个典型的kafka集群中包含若干producer（可以是web前端产生的page view，或者是服务器日志，系统CPU、memory等），若干broker（Kafka支持水平扩展，一般broker数量越多，集群吞吐率越高），若干consumer group，以及一个Zookeeper集群。Kafka通过Zookeeper管理集群配置，选举leader，以及在consumer group发生变化时进行rebalance。producer使用push模式将消息发布到broker，consumer使用pull模式从broker订阅并消费消息。

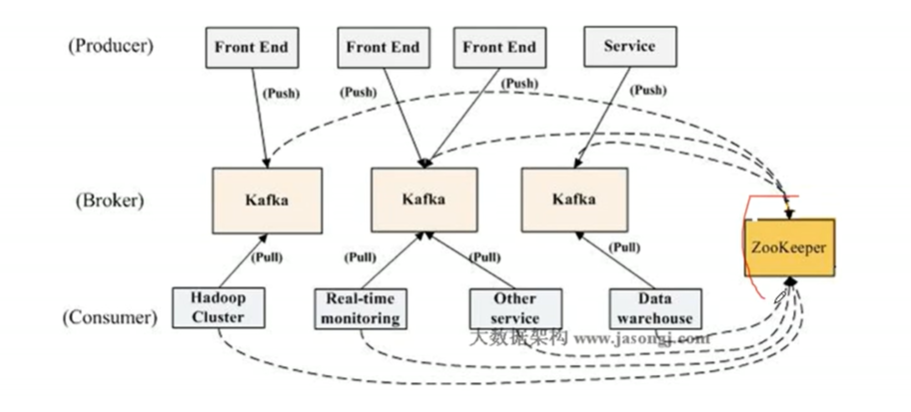
Kafka Topic结构 Kafka集群中的消息，是通过Topic（主题）来进行组织的，如下图所示：

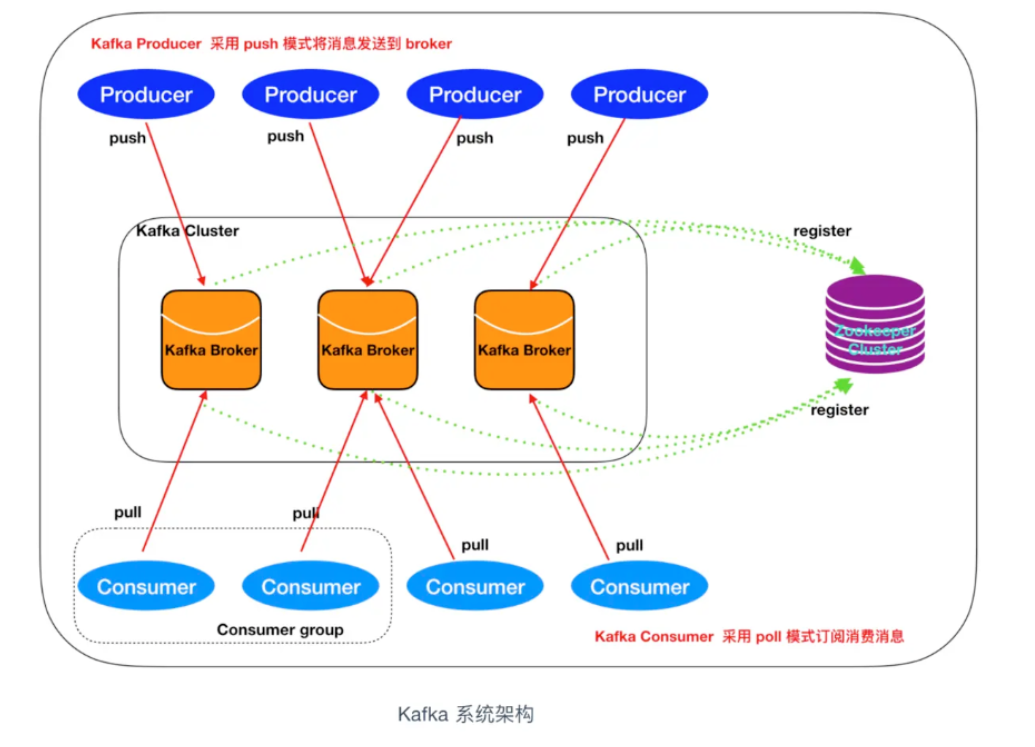


因为每条消息都被append到该partition中，是顺序写磁盘，因此效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是Kafka高吞吐率的一个很重要的保证）。每一条消息被发送到broker时，会根据paritition规则选择被存储到哪一个partition。如果partition规则设置的合理，所有消息可以均匀分布到不同的partition里，这样就实现了水平扩展。（如果一个topic对应一个文件，那这个文件所在的机器I/O将会成为这个topic的性能瓶颈，而partition解决了这个问题）

Kafka流程架构：







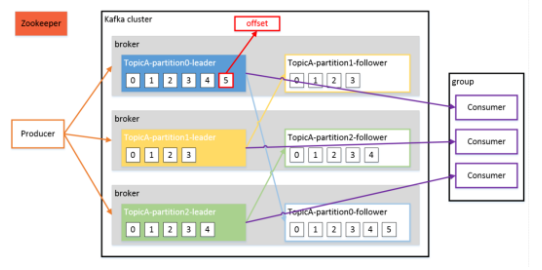
Kafka存储的消息来自任意多个被称为Producer生产者的进程。数据从而可以被发布到不同的Topic主题下的不同Partition分区。

在一个分区内，这些消息被索引并连同时间戳存储在一起。其它被称为Consumer消费者的进程可以从分区订阅消息。

Kafka运行在一个由一台或多台服务器组成的集群上，并且分区可以跨集群结点分布。

## 工作流程

Kafka集群将Record流存储在称为Topic的类别中，每个记录由一个键、一个值和一个时间戳组成。



Kafka是一个分布式流平台，这到底是什么意思？

发布和订阅记录流，类似于消息队列或企业消息传递系统。

以容错的持久方式存储记录流。

处理记录流。

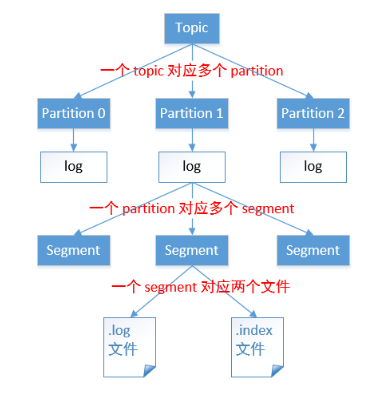
Kafka中消息是以Topic进行分类的，生产者生产消息，消费者消费消息，面向的都是同一个Topic。

Topic是逻辑上的概念，而Partition是物理上的概念，每个Partition对应于一个log文件，该log文件中存储的就是Producer生产的数据。

Producer生产的数据会不断追加到该log文件末端，且每条数据都有自己的Offset。

消费者组中的每个消费者，都会实时记录自己消费到了哪个Offset，以便出错恢复时，从上次的位置继续消费。

## 存储机制



由于生产者生产的消息会不断追加到log文件末尾，**为防止log文件过大导致数据定位效率低下，Kafka采取了分片和索引机制**。

**它将每个Partition分为多个Segment，每个Segment对应两个文件：“.index” 索引文件和“.log”数据文件**。

这些文件位于同一文件下，该文件夹的命名规则为：topic名-分区号。例如first这个topic有三分分区，则其对应的文件夹为first-0，first-1，first-2。

# ls /root/data/kafka/first-0

00000000000000009014.index

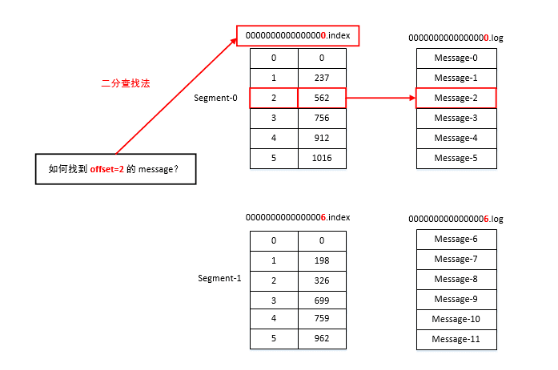
00000000000000009014.log

00000000000000009014.timeindex

00000000000000009014.snapshot

leader-epoch-checkpoint

index和log文件以当前Segment的第一条消息的Offset 命名。下图为index 文件和log文件的结构示意图：



“.index”文件存储大量的索引信息，“.log”文件存储大量的数据，索引文件中的元数据指向对应数据文件中Message的物理偏移量。

## 生产者

### 分区策略

**分区原因：**

方便在集群中扩展，每个Partition可以通过调整以适应它所在的机器，而一个Topic又可以有多个Partition组成，因此可以以Partition为单位读写了。

可以提高并发，因此可以以Partition为单位读写了。

**分区原则：**

我们需要将Producer发送的数据封装成一个ProducerRecord对象。

该对象需要指定一些参数：

topic：string类型，NotNull。

partition：int类型，可选。

timestamp：long类型，可选。

key：string类型，可选。

value：string类型，可选。

headers：array类型，Nullable。

指明Partition的情况下，直接将给定的Value作为Partition的值。

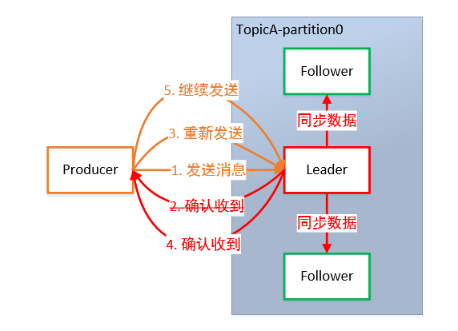
没有指明Partition但有Key的情况下，将Key的Hash值与分区数取余得到Partition值。

既没有Partition有没有Key的情况下，第一次调用时随机生成一个整数（后面每次调用都在这个整数上自增），将这个值与可用的分区数取余，得到 Partition 值，也就是常说的Round-Robin轮询算法。

### 数据可靠性保证

为保证Producer发送的数据，能可靠地发送到指定的Topic，Topic的每个Partition收到Producer发送的数据后，都需要向Producer发送ACK（ACKnowledge 确认收到）。

如果Producer收到ACK，就会进行下一轮的发送，否则重新发送数据。



#### 副本数据同步策略

何时发送ACK？确保有Follower与Leader同步完成，Leader再发送ACK，这样才能保证Leader挂掉之后，能在Follower中选举出新的Leader而不丢数据。

多少个Follower同步完成后发送ACK？全部Follower同步完成，再发送ACK。



#### ISR

采用第二种方案，所有Follower完成同步，Producer才能继续发送数据，设想有一个Follower因为某种原因出现故障，那Leader就要一直等到它完成同步。

这个问题怎么解决？Leader维护了一个动态的 in-sync replica set（ISR）：和Leader保持同步的Follower集合。

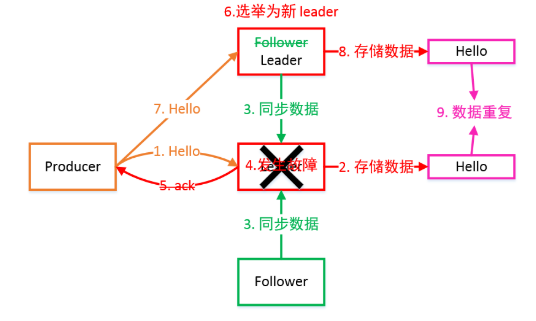
当ISR集合中的 Follower完成数据的同步之后，Leader就会给Follower发送 ACK。

如果Follower长时间未向Leader同步数据，则该Follower将被踢出ISR集合，该时间阈值由replica.lag.time.max.ms参数设定。Leader发生故障后，就会从ISR中选举出新的Leader。

#### ACK应答机制

对于某些不太重要的数据，对数据的可靠性要求不是很高，能够容忍数据的少量丢失，所以没必要等ISR中的Follower全部接受成功。

所以Kafka为用户提供了三种可靠性级别，用户根据可靠性和延迟的要求进行权衡，选择以下的配置。



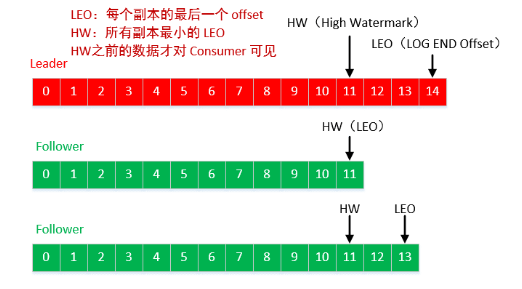
ACK参数配置：

0：Producer不等待Broker的ACK，这提供了最低延迟，Broker一收到数据还没有写入磁盘就已经返回，当Broker故障时有可能丢失数据。

1：Producer等待Broker的ACK，Partition的Leader落盘成功后返回ACK，如果在Follower同步成功之前Leader故障，那么将会丢失数据。

-1（all）：Producer等待Broker的ACK，Partition的Leader和Follower全部落盘成功后才返回ACK。但是在Broker发送ACK时，Leader发生故障，则会造成数据重复。

#### 故障处理细节



LEO：每个副本最大的Offset。HW：消费者能见到的最大的Offset，ISR队列中最小的 LEO。

Follower故障：Follower发生故障后会被临时踢出ISR集合，待该Follower恢复后，Follower会读取本地磁盘记录的上次的HW，并将log文件高于HW的部分截取掉，从HW开始向 Leader 进行同步数据操作。

等该Follower的LEO大于等于该 Partition的HW，即Follower追上Leader后，就可以重新加入ISR了。

Leader故障：Leader发生故障后，会从ISR中选出一个新的Leader，之后，为保证多个副本之间的数据一致性，其余的Follower会先将各自的log文件高于HW的部分截掉，然后从新的Leader同步数据。

注意：这只能保证副本之间的数据一致性，并不能保证数据不丢失或者不重复。

#### Exactly Once语义

将服务器的ACK级别设置为-1，可以保证Producer到Server之间不会丢失数据，即 At Least Once语义。

相对的，将服务器ACK级别设置为0，可以保证生产者每条消息只会被发送一次，即At Most Once语义。

At Least Once可以保证数据不丢失，但是不能保证数据不重复；相对的，At Most Once可以保证数据不重复，但是不能保证数据不丢失。

但是，对于一些非常重要的信息，比如交易数据，下游数据消费者要求数据既不重复也不丢失，即Exactly Once语义。

0.11版本的Kafka，引入了幂等性：Producer不论向Server发送多少重复数据，Server端都只会持久化一条。

即：At Least Once+幂等性=Exactly Once

要启用幂等性，只需要将Producer的参数中enable.idompotence设置为true即可。

开启幂等性的Producer在初始化时会被分配一个PID，发往同一Partition的消息会附带Sequence Number。

而Borker端会对<PID,Partition,SeqNumber>做缓存，当具有相同主键的消息提交时，Broker只会持久化一条。

但是PID重启后就会变化，同时不同的Partition也具有不同主键，所以幂等性无法保证跨分区会话的Exactly Once。

## 消费者

### 消费方式

Consumer采用Pull（拉取）模式从 Broker中读取数据。

Consumer采用Push（推送）模式，Broker给Consumer推送消息的速率是由Broker决定的，很难适应消费速率不同的消费者。

它的目标是尽可能以最快速度传递消息，但是这样很容易造成Consumer来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。

而Pull模式则可以根据Consumer的消费能力以适当的速率消费消息。Pull模式不足之处是，如果Kafka没有数据，消费者可能会陷入循环中，一直返回空数据。

因为消费者从Broker主动拉取数据，需要维护一个长轮询，针对这一点， Kafka的消费者在消费数据时会传入一个时长参数timeout。

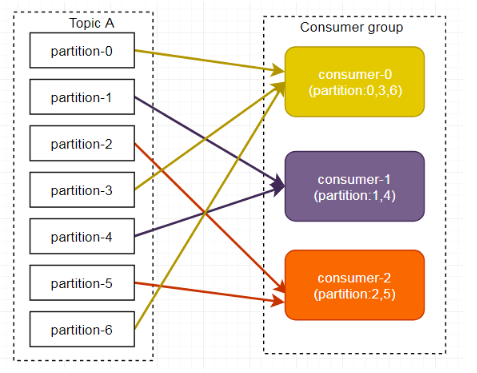
如果当前没有数据可供消费，Consumer会等待一段时间之后再返回，这段时长即为timeout。

### 分区分配策略

一个Consumer Group中有多个Consumer，一个Topic有多个Partition，所以必然会涉及到Partition的分配问题，即确定哪个Partition由哪个Consumer来消费。

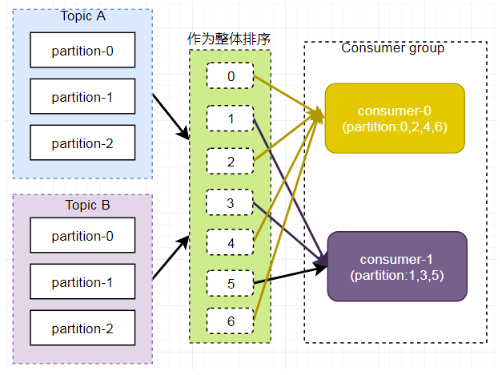
Kafka有两种分配策略，一个是RoundRobin，一个是Range，默认为Range，当消费者组内消费者发生变化时，会触发分区分配策略（方法重新分配）。

#### RoundRobin



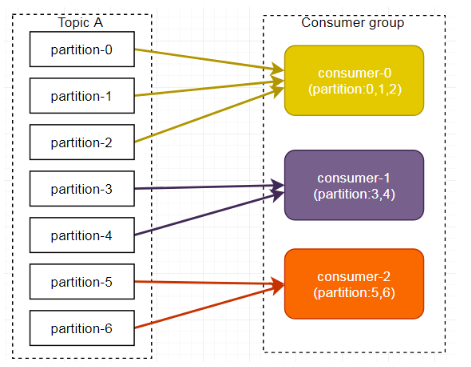
RoundRobin轮询方式将分区所有作为一个整体进行Hash排序，消费者组内分配分区个数最大差别为1，是按照组来分的，可以解决多个消费者消费数据不均衡的问题。

但是，当消费者组内订阅不同主题时，可能造成消费混乱，如下图所示，Consumer0订阅主题A，Consumer1订阅主题B。



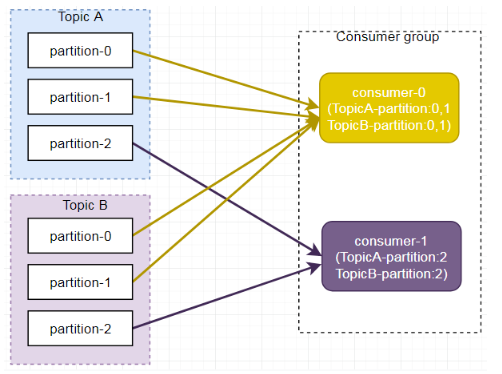
将A、B主题的分区排序后分配给消费者组，TopicB分区中的数据可能分配到Consumer0中。

#### Range



Range方式是按照主题来分的，不会产生轮询方式的消费混乱问题。

但是，如下图所示，Consumer0、Consumer1 同时订阅了主题A和B，可能造成消息分配不对等问题，当消费者组内订阅的主题越多，分区分配可能越不均衡。



### Offset的维护

由于Consumer在消费过程中可能会出现断电宕机等故障，Consumer恢复后，需要从故障前的位置继续消费。

所以Consumer需要实时记录自己消费到了哪个Offset，以便故障恢复后继续消费。

Kafka 0.9版本之前，Consumer默认将Offset保存在ZooKeeper中，从0.9版本开始，Consumer默认将Offset保存在Kafka一个内置的Topic中，该Topic为\_\_consumer\_offsets。

# 环境搭建

# 常用命令

## 启动

后台常驻方式，带上参数 -daemon，如：

/usr/local/kafka/bin/kafka-server-start.sh -daemon /usr/local/kafka/config/server.properties

指定JMX port端口启动，指定jmx，可以方便监控Kafka集群：

JMX\_PORT=9991 /usr/local/kafka/bin/kafka-server-start.sh -daemon /usr/local/kafka/config/server.properties

## 停止

/usr/local/kafka/bin/kafka-server-stop.sh

## Topic

### 创建Topic

参数 --topic 指定Topic名，--partitions 指定分区数，--replication-factor 指定备份数：

/usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test

注意，如果配置文件server.properties指定了Kafka在zookeeper上的目录，则参数也要指定，否则会报无可用的brokers（下面部分命令也有同样的情况），如：

/usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181/kafka --replication-factor 1 --partitions 1 --topic test

### 列出所有Topic

/usr/local/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181

### 查看Topic

/usr/local/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test

### 增加Topic的partition数

/usr/local/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --alter --topic test --partitions 5

### 查看topic指定分区offset的最大值或最小值

time为-1时表示最大值，为-2时表示最小值：

/usr/local/kafka/bin/kafka-run-class.sh kafka.tools.GetOffsetShell --topic test --time -1 --broker-list 127.0.0.1:9092 --partitions 0

### 删除Topic

/usr/local/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --topic test --delete

## 生产消息

/usr/local/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test

## 消费消息

### 从头开始

/usr/local/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning

### 从尾部开始

从尾部开始取数据，必需要指定分区：

/usr/local/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --offset latest --partition 0

### 指定分区

/usr/local/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --offset latest --partition 0

### 取指定个数

/usr/local/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --offset latest --partition 0 --max-messages 1

## 消费者Group

### 指定Group

/usr/local/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test -group test\_group --from-beginning

### 消费者Group列表

/usr/local/kafka/bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list

### 查看Group详情

/usr/local/kafka/bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group test\_group --describe

### 删除Group中Topic

/usr/local/kafka/bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group test\_group --topic test --delete

### 删除Group

/usr/local/kafka/bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group test\_group --delete

## 平衡leader

/usr/local/kafka/bin/kafka-preferred-replica-election.sh --bootstrap-server localhost:9092

## 自带压测工具

/usr/local/kafka/bin/kafka-producer-perf-test.sh --topic test --num-records 100 --record-size 1 --throughput 100 --producer-props bootstrap.servers=localhost:9092

# 备份

消息以partition为单位分配到多个server，并以partition为单位进行备份。 备份策略为：1个leader和N个followers，leader接受读写请求，followers被动复制leader。leader和followers会在集群中打散，保证partition高可用。  
 kafka将每个partition数据复制到多个server上,任何一个partition有一个leader和多个follower(可以没有)；备份的个数可以通过broker配置文件来设定。leader处理所有的read-write请求，follower需要和leader保持同步。Follower和consumer一样，消费消息并保存在本地日志中；leader负责跟踪所有的follower状态，如果follower”落后”太多或者失效，leader将会把它从replicas同步列表中删除。当所有的 follower 都将一条消息保存成功,此消息才被认为是”committed”,那么此时consumer才能消费它。即使只有一个replicas实例存活,仍然可以保证消息的正常发送和接收,只要zookeeper集群存活即可。(不同于其他分布式存储,比如 hbase需要”多数派”存活才行)当leader失效时,需在followers中选取出新的leader,可能此时 follower 落后于 leader,因此需要选择一个”up-to-date”的follower。选择follower时需要兼顾一个问题,就是新leader server上所已经承载的 partition leader 的个数,如果一个 server 上有过多的 partition leader,意味着此 server 将承受着更多的IO 压力。在选举新 leader,需要考虑到”负载均衡”。

# 可靠性

MQ要实现从producer到consumer之间的可靠的消息传送和分发。传统的MQ系统通常都是通过broker和consumer间的确认（ack）机制实现的，并在broker保存消息分发的状态。即使这样一致性也是很难保证的。**kafka的做法是由consumer自己保存状态，也不要任何确认（不需要确认就可以提高性能，但是会加重consumer的负担）。这样虽然consumer负担更重，但其实更灵活了**。因为不管consumer上任何原因导致需要重新处理消息，都可以再次从broker获得。  
 kafka的producer有一种**异步发送**的操作。这是为提高性能提供的。**producer先将消息放在内存中，就返回。这样调用者（应用程序）就不需要等网络传输结束就可以继续了。内存中的消息会在后台批量的发送到broker。由于消息会在内存呆一段时间，这段时间是有消息丢失的风险的。**所以使用该操作时需要仔细评估这一点。因此Kafka不像传统的MQ难以实现EIP，并且只有partition内的消息才能保证传递顺序。  
 另外，在最新的版本中，还实现了broker间的消息复制机制，去除了broker的单点故障（SPOF）。

# 持久性

kafka使用文件存储消息，这就直接决定kafka在性能上严重依赖文件系统的本身特性。且无论任何OS下，对文件系统本身的优化几乎没有可能。文件缓存/直接内存映射等是常用的手段。

因为kafka是对日志文件进行append操作,因此磁盘检索的开支是较小的；同时为了减少磁盘写入的次数，broker会将消息暂时buffer起来，当消息的个数(或尺寸)达到一定阀值时，再flush到磁盘，这样减少了磁盘IO调用的次数。

# 性能

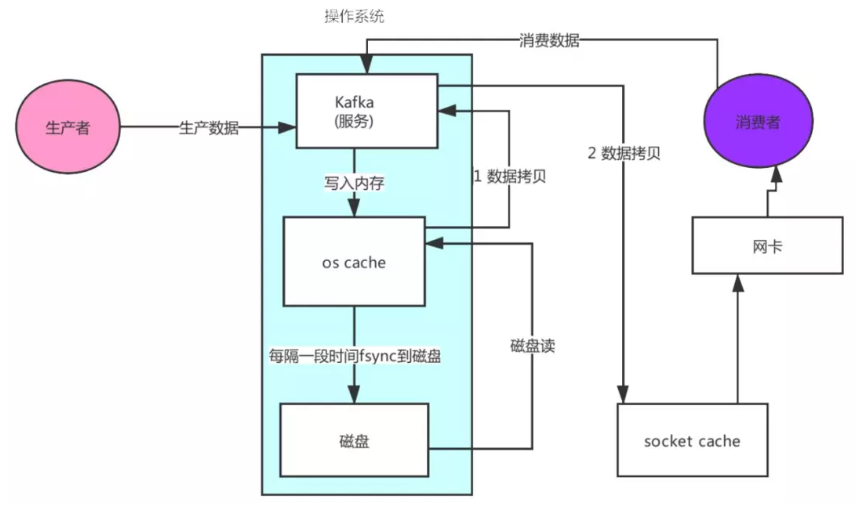
## 顺序写

操作系统每次从磁盘读写数据的时候，需要先寻址，也就是先要找到数据在磁盘上的物理位置，然后再进行数据读写，如果是机械硬盘，寻址就需要较长的时间。kafka的设计中，数据其实是存储在磁盘上面，一般来说，会把数据存储在内存上面性能才会好。但是kafka用的是顺序写，追加数据是追加到**末尾**，磁盘顺序写的性能极高，在磁盘个数一定，转数达到一定的情况下，基本和内存速度一致随机写的话是在文件的某个位置修改数据，性能会较低。

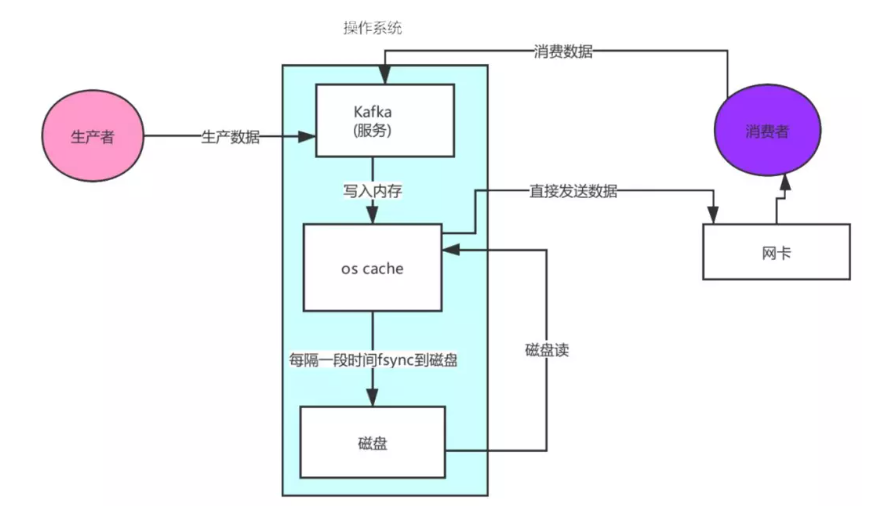
注：参考partition中segment的特性。

## 零拷贝

先来看看非零拷贝的情况



可以看到数据的拷贝从内存拷贝到kafka服务进程那块，又拷贝到socket缓存那块，整个过程耗费的时间比较高，kafka利用了Linux的sendFile技术（NIO），省去了进程切换和一次数据拷贝，让性能变得更好。



## 日志分段存储

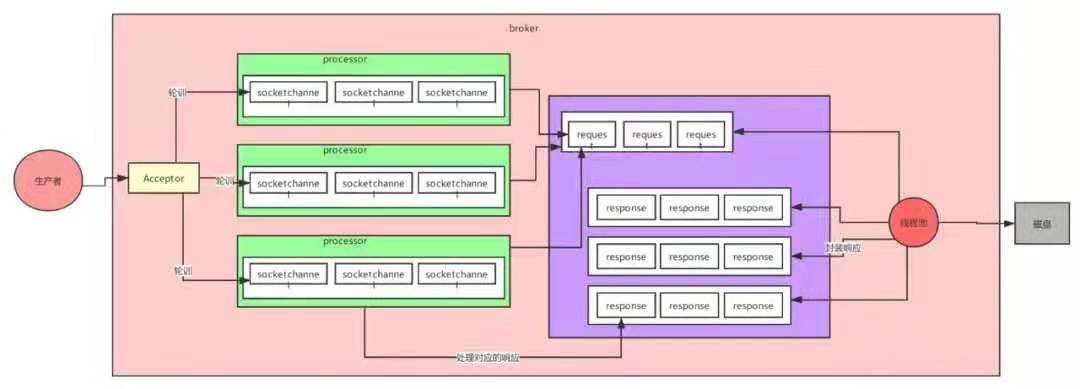
Kafka规定了一个分区内的.log文件最大为1G，做这个限制目的是为了方便把.log加载到内存去操作

00000000000000000000.index  
 00000000000000000000.log  
 00000000000000000000.timeindex  
  
 00000000000005367851.index  
 00000000000005367851.log  
 00000000000005367851.timeindex  
  
 00000000000009936472.index  
 00000000000009936472.log  
 00000000000009936472.timeindex

这个9936472之类的数字，就是代表了这个日志段文件里包含的起始offset，也就说明这个分区里至少都写入了接近1000万条数据了。Kafka broker有一个参数，log.segment.bytes，限定了每个日志段文件的大小，最大就是1GB，一个日志段文件满了，就自动开一个新的日志段文件来写入，避免单个文件过大，影响文件的读写性能，这个过程叫做log rolling，正在被写入的那个日志段文件，叫做active log segment。分析HDFS的框架时，就会发现NameNode的edits log也会做出限制，所以这些框架都是会考虑到这些问题。

## Kafka的网络设计

kafka的网络设计和Kafka的调优有关，这也是为什么它能支持高并发的原因。



首先客户端发送请求全部会先发送给一个Acceptor，broker里面会存在3个线程（默认是3个），这3个线程都是叫做processor，Acceptor不会对客户端的请求做任何的处理，直接封装成一个个socketChannel发送给这些processor形成一个队列，发送的方式是轮询，就是先给第一个processor发送，然后再给第二个，第三个，然后又回到第一个。

消费者线程去消费这些socketChannel时，会获取一个个request请求，这些request请求中就会伴随着数据。线程池里面默认有8个线程，这些线程是用来处理request的，解析请求，如果request是写请求，就写到磁盘里。读的话返回结果。processor会从response中读取响应数据，然后再返回给客户端。这就是Kafka的网络三层架构。所以如果我们需要对kafka进行增强调优，增加processor并增加线程池里面的处理线程，就可以达到效果。request和response那一块部分其实就是起到了一个缓存的效果，是考虑到processor生成请求太快，线程数不够不能及时处理的问题。所以这就是一个加强版的reactor网络线程模型。

Kafka实现了**零拷贝**原理来快速移动数据，避免了内核之间的切换。

Kafka可以将数据记录**分批发送**，从生产者到文件系统（Kafka主题日志）到消费者，可以端到端的查看这些批次的数据。  
 批处理能够进行更有效的**数据压缩**并减少I/O延迟，Kafka采取顺序写入磁盘的方式，避免了随机磁盘寻址的浪费。  
 总结一下其实就是四个要点：**顺序读写、 零拷贝、 消息压缩和分批发送**。

# 监控

# ZK高可用

Kafka的集群高可用依赖于ZK，但是新版本已经移除该限制。

3月30日，Kafka背后的企业Confluent发布博客表示，在即将发布的2.8版本里，用户可在完全不需要ZooKeeper的情况下运行Kafka，该版本将依赖于ZooKeeper的控制器改造成了基于Kafka Raft的Quorm控制器。

在之前的版本中，如果没有ZooKeeper，Kafka将无法运行。但管理部署两个不同的系统不仅让运维复杂度翻倍，还让Kafka变得沉重，进而限制了 Kafka在轻量环境下的应用，同时ZooKeeper的分区特性也限制了Kafka的承载能力。

这是一次架构上的重大升级，让一向“重量级”的Kafka从此变得简单了起来。轻量级的单进程部署可以作为ActiveMQ或RabbitMQ等的替代方案，同时也适合于边缘场景和使用轻量级硬件的场景。

## 背景

**为什么要抛弃使用了十年的 ZooKeeper**

ZooKeeper是Hadoop的一个子项目，一般用来管理较大规模、结构复杂的服务器集群，具有自己的配置文件语法、管理工具和部署模式。Kafka最初由LinkedIn开发，随后于2011年初开源，2014年由主创人员组建企业Confluent。

Broker是Kafka集群的骨干，负责从生产者（producer）到消费者（consumer）的接收、存储和发送消息。在当前架构下，Kafka进程在启动的时候需要往ZooKeeper集群中注册一些信息，比如BrokerId，并组建集群。ZooKeeper为Kafka提供了可靠的元数据存储，比如Topic/分区的元数据、Broker数据、ACL信息等等。

同时 ZooKeeper充当Kafka的领导者，以更新集群中的拓扑更改；根据ZooKeeper提供的通知，生产者和消费者发现整个Kafka集群中是否存在任何新Broker或Broker失败。大多数的运维操作，比如说扩容、分区迁移等等，都需要和ZooKeeper交互。

也就是说，Kafka代码库中有很大一部分是负责实现在集群中多个Broker之间分配分区（即日志）、分配领导权、处理故障等分布式系统的功能。而早已经过业界广泛使用和验证过的ZooKeeper是分布式代码工作的关键部分。

假设没有ZooKeeper的话，Kafka甚至无法启动进程。“在以前的版本中，ZooKeeper可以说是Kafka集群的灵魂。”

但严重依赖ZooKeeper，也给Kafka带来了掣肘。Kafka一路发展过来，绕不开的两个话题就是集群运维的复杂度以及单集群可承载的分区规模，比如腾讯云Kafka维护了上万节点的Kafka集群，主要遇到的问题也还是这两个。

首先从集群运维的角度来看，Kafka本身就是一个分布式系统。但它又依赖另一个开源的分布式系统，而这个系统又是Kafka系统本身的核心。这就要求集群的研发和维护人员需要同时了解这两个开源系统，需要对其运行原理以及日常的运维（比如参数配置、扩缩容、监控告警等）都有足够的了解和运营经验。否则在集群出现问题的时候无法恢复，是不可接受的。所以，ZooKeeper的存在增加了运维的成本。

其次从集群规模的角度来看，限制Kafka集群规模的一个核心指标就是集群可承载的分区数。集群的分区数对集群的影响主要有两点：ZooKeeper上存储的元数据量和控制器变动效率。

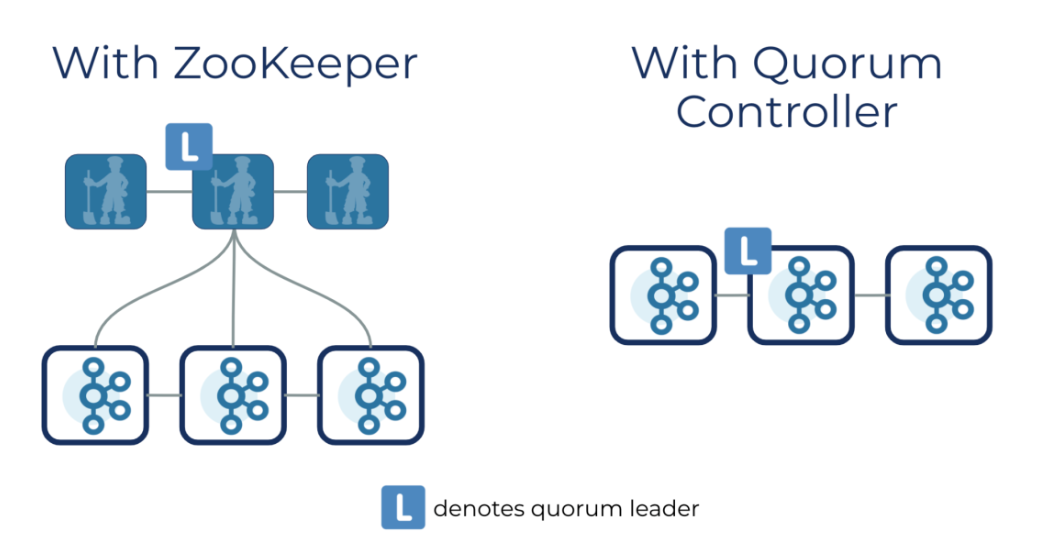
Kafka集群依赖于一个单一的Controller节点来处理绝大多数的ZooKeeper读写和运维操作，并在本地缓存所有ZooKeeper上的元数据。分区数增加，ZooKeeper上需要存储的元数据就会增加，从而加大ZooKeeper的负载，给ZooKeeper集群带来压力，可能导致Watch的延时或丢失。

当Controller节点出现变动时，需要进行Leader切换、Controller节点重新选举等行为，分区数越多需要进行越多的ZooKeeper操作：比如当一个Kafka节点关闭的时候，Controller需要通过写ZooKeeper将这个节点的所有Leader分区迁移到其他节点；新的Controller节点启动时，首先需要将所有ZooKeeper上的元数据读进本地缓存，分区越多，数据量越多，故障恢复耗时也就越长。

Kafka单集群可承载的分区数量对于一些业务来说，又特别重要。“腾讯云 Kafka主要为公有云用户以及公司内部业务提供服务。我们遇到了很多需要支持百万分区的用户，比如腾讯云Serverless、腾讯云的CLS日志服务、云上的一些客户等，他们面临的场景是一个客户需要一个topic来进行业务逻辑处理，当用户量达到百万千万量级的情况下，topic带来的膨胀是非常恐怖的。在当前架构下，Kafka单集群无法稳定承载百万分区稳定运行。”

## 架构

去除ZooKeeper后的Kafka



为了改善Kafka，去年起Confluent就开始重写ZooKeeper功能，将这部分代码集成到了Kafka内部。他们将新版本称为“ Kafka on Kafka”，意思是将元数据存储在Kafka本身，而不是存储ZooKeeper这样的外部系统中。Quorum控制器使用新的KRaft协议来确保元数据在仲裁中被精确地复制。这个协议在很多方面与ZooKeeper的ZAB协议和Raft相似。这意味着，仲裁控制器在成为活动状态之前不需要从ZooKeeper加载状态。当领导权发生变化时，新的活动控制器已经在内存中拥有所有提交的元数据记录。

去除ZooKeeper后，Kafka集群的运维复杂性直接减半。

在架构改进之前，一个最小的分布式Kafka集群也需要六个异构的节点：三个ZooKeeper节点，三个Kafka节点。而一个最简单的Quickstart演示也需要先启动一个ZooKeeper进程，然后再启动一个Kafka进程。在新的KIP-500版本中，一个分布式Kafka集群只需要三个节点，而Quickstart演示只需要一个Kafka进程就可以。

改进后同时提高了集群的延展性（scalability），大大增加了Kafka单集群可承载的分区数量。

在此之前，元数据管理一直是集群范围限制的主要瓶颈。特别是在集群规模比较大的时候，如果出现Controller节点失败涉及到的选举、Leader分区迁移，以及将所有ZooKeeper的元数据读进本地缓存的操作，所有这些操作都会受限于单个Controller的读写带宽。因此一个Kafka集群可以管理的分区总数也会受限于这单个Controller的效率。

Confluent流数据部门首席工程师王国璋解释道：“在KIP-500 中，我们用一个Quorum Controller来代替和ZooKeeper交互的单个Controller，这个Quorum里面的每个Controller节点都会通过Raft机制来备份所有元数据，而其中的Leader在写入新的元数据时，也可以借由批量写入（batch writes）Raft日志来提高效率。我们的实验表明，在一个可以管理两百万个分区的集群中，Quorum Controller的迁移过程可以从几分钟缩小至三十秒。”

## 升级

**升级是否需要停机？**

减少依赖、扩大单集群承载能力，这肯定是一个很积极的改变方向。虽然目前的版本还未经过大流量检验，可能存在稳定性问题，这也是让广大开发者担心的一个方面。但从长期意义上来讲，KIP-500对社区和用户都是一个很大的福音。

当版本稳定之后，最终大家就会涉及到“升级”的工作。但如何升级，却成了一个新的问题，在很多Kafka的使用场景中，是不允许业务停机的。拿腾讯云的业务举例说，“微信安全业务的消息管道就使用了腾讯云的Kafka，假设发生停机，那么一些自动化的安全业务就会受到影响，从而严重影响客户体验。”

“就我们的经验而言，停机升级，在腾讯云上是一个非常敏感的词。腾讯云Kafka至今为止已经运营了六七年，服务了内外部几百家大客户，还未发生过一次停机升级的情况。如果需要停机才能升级，那么对客户的业务肯定会有影响的，影响的范围取决于客户业务的重要性。从云服务的角度来看，任何客户的业务的可持续性都是非常重要的、不可以被影响的。”

对于Confluent来讲，提供不需要停机的平滑升级方案是一件非常有必要的事情。

据王国璋介绍，“目前的设计方案是，在2.8版本之后的3.0版本会是一个特殊的搭桥版本（bridge release），在这个版本中，Quorum Controller会和老版本的基于ZooKeeper的Controller，而在之后的版本我们才会去掉旧的Controller模块。”

## 技术展望

**消息系统掀起二次革命？**

Apache Kafka出现之后，很快击败其它的消息系统，成为最主流的应用。从2011年启动，经过十年发展，得到大规模应用之后，为什么现在又决定用“Raft协议”替换ZooKeeper呢？对此，王国璋说，Raft是近年来很火的共识算法，但在Kafka设计之初（2011年），不仅仅是Raft方案，就连一个成熟通用的共识机制（consensus）代码库也不存在。当时最直接的设计方案就是基于ZooKeeper这样一个高可用的同步服务项目。

在这十年里，Hadoop生态中的不少软件都在被逐渐抛弃。如今，作为Hadoop生态中的一员，ZooKeeper也开始过时了？韩欣给予了否认意见，“每一种架构或者软件都有其适合的应用场景，我不认可过时这个词。”

从技术的角度来看，历史的车轮在不断向前滚动，学术界和工业界的理论基础一直在不断进化，技术也要适应不断革新的业务不停去演进。不否认有一些软件会被一些新的软件所替代，或者说一些新的软件会更适合某些场景。比如流计算领域，Storm、Spark、Filnk的演进。但是合适的组件总会出现在合适的地方，这就是架构师和研发人员的工作和责任。

Kafka发展至今，虽然其体系结构不断被改进，比如引入自动缩放、多租户等功能，来满足用户发展的需求，但针对这次大的改进，且还存在需要验证的现状，网友在 HackerNews上提出了一个灵魂发问：“如果现在还要设计一个新系统，那么是什么理由选择Kafka而不是Pulsar？”

Confluent的科林·麦凯布（Colin McCabe）回应这个争议说，起码去掉ZooKeeper对Pulsar来说是一个艰巨的挑战。Kafka去除ZooKeeper依赖是个很大的卖点，意味着Kafka只有一个组件Broker，而Pulsar则需要ZooKeeper、Bookie、Broker（或者proxy）等多个组件。但也正因为抽离出一层存储层（Bookie），使得后起之秀的Pulsar在架构上天然具有了“计算存储分离”的优势。

总的来说，在企业加速上云的背景下，无论是Kafka还是Pulsar，消息系统必须是要适应云原生的大趋势的，实现计算和存储分离的功能也是Kafka下一步的策略。Confluent在另一个KIP-405版本中，实现了一个分层式的存储模式，利用在云架构下多种存储介质的实际情况，将计算层和存储层分离，将“冷数据”和“热数据”分离，使得Kafka的分区扩容、缩容、迁移等等操作更加高效和低耗，同时也使Kafka可以在理论上长时间保留数据流。

在发展趋势上，云原生的出现对消息系统的影响是比较大的，比如容器化和大规模云盘，为原本在单集群性能和堆积限制方面存在上限问题的Kafka，在突破资源瓶颈这里带来了新的思路。Broker的容器化，堆积的消息用大规模的云盘，再加上KRaft去掉了ZooKeeper给Kafka带来的元数据管理方面的限制，是否是给Kafka带来了二次的消息系统革命？容器化的消息系统是否会带来运营运维方面更多的自动化的能力？Serverless是否是消息系统的未来趋势？云和云原生的发展，给传统的消息系统安上了新的翅膀，带来了新的想象空间。

对于用户而言，这意味着如果想要从2.8版本以下升级到3.0以后的某一个版本，比如说3.1，则需要借由3.0版本实现两次“跳跃”，也就是说先在线平滑升级到3.0，然后再一次在线平滑升级到3.1。并且在整个过程中，Kafka服务器端都可以和各种低版本的客户端进行交互，而不需要强制客户端的升级。

而像腾讯这样的企业，也会持续采用灰度的方式来进行业务的升级验证，韩欣说，“一般不会在存量集群上做大规模升级操作，而是会采用新建集群的方式，让一些有迫切诉求的业务先切量进行灰度验证，在保证线上业务稳定运行的情况下，逐步扩展新集群的规模，这样逐步将业务升级和迁移到新的架构上去。”

# Flume对接Kafka

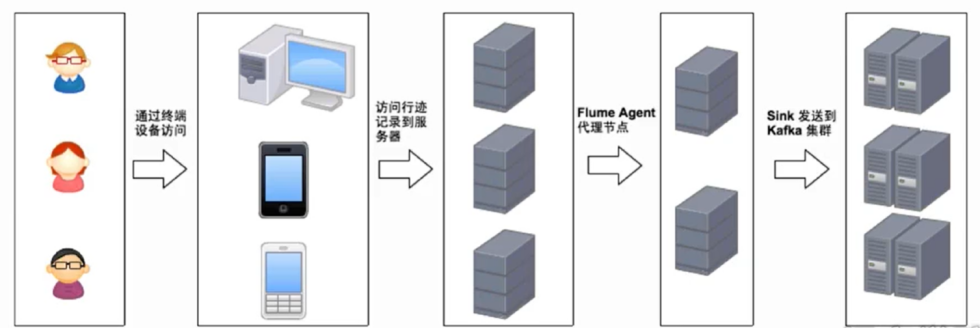
# 应用场景

## 常用场景

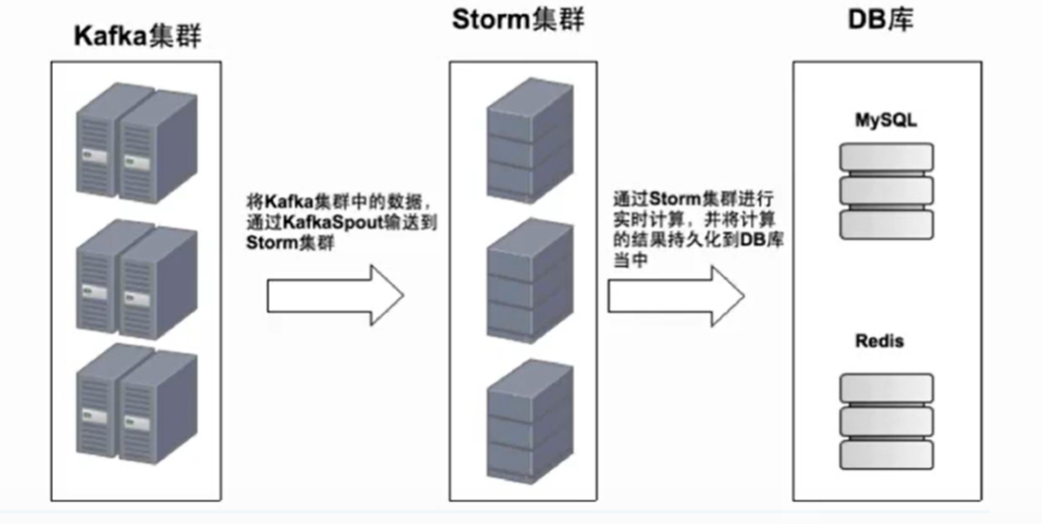
### 活动跟踪

**活动跟踪：**Kafka可以用来跟踪用户行为，比如我们经常回去淘宝购物，你打开淘宝的那一刻，你的登陆信息，登陆次数都会作为消息传输到Kafka ，当你浏览购物的时候，你的浏览信息，你的搜索指数，你的购物爱好都会作为一个个消息传递给Kafka，这样就可以生成报告，可以做智能推荐，购买喜好等。

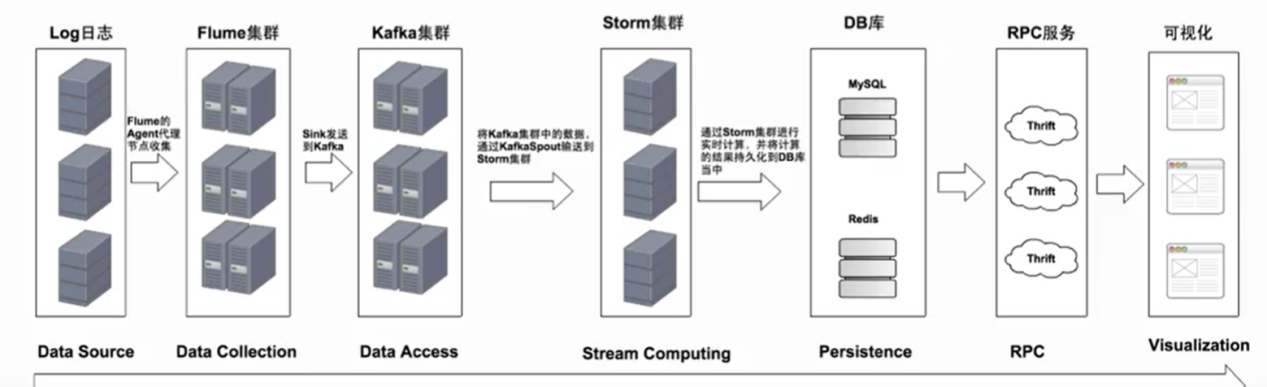
数据源生产流程（Kafka->Flume）：



数据源消费流程（kafka->strom）：



整体流程：



### 传递消息

**传递消息：**Kafka另外一个基本用途是传递消息，应用程序向用户发送通知就是通过传递消息来实现的，这些应用组件可以生成消息，而不需要关心消息的格式，也不需要关心消息是如何发送的。

### 度量指标

**度量指标：**Kafka也经常用来记录运营监控数据。包括收集各种分布式应用的数据，生产各种操作的集中反馈，比如报警和报告。

### 日志记录

**日志记录：**Kafka的基本概念来源于提交日志，比如我们可以把数据库的更新发送到Kafka上，用来记录数据库的更新时间，通过kafka以统一接口服务的方式开放给各种consumer，例如hadoop、Hbase、Solr等。

### 流式处理

**流式处理：**流式处理是有一个能够提供多种应用程序的领域。

### 限流削峰

**限流削峰：**Kafka多用于互联网领域某一时刻请求特别多的情况下，可以把请求写入Kafka中，避免直接请求后端程序导致服务崩溃。

## 分布式数据库实践

在分布式数据库中，高可用架构采用ZK方式实现，Kafka本身依赖于Zookeeper组件运行，保存采集端采集的所有类型的数据，每个类型对应一个队列。

大概步骤：

1、在每个服务器上有一个agent代理进程，获取该服务器上的数据（agent需要创建ramfs，需要占用一定的内存，采用ramfs是为了不影响磁盘IO）。

2、Kafka用来获取分布式的各种采集数据信息，包括告警和报告以及日志信息。

3、Kafka采集的数据经过logstash和strom分析后，存入ES。

4、在分布式数据库中，采集若干个服务器的数据量可能达到1T，考虑用ES存储。Kafka中所有的队列数据最终入库Elasticsearch进行持久化存储，ES中每个索引对应一个Kafka队列。

5、Web端获取ES和MySQL存储的信息，并显示相关信息。