**Kafka**

#### Kafka简介

1. Kafka是分布式发布-订阅消息系统，它实现了生产者与消费者之间的无缝连接。
2. 设计目标：

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

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

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

同时支持离线数据处理和实时数据处理

1. 基本架构
2. Broker：

Kafka集群包含一个或多个服务器，这种服务器被称为broker

2）Topic

每条发布到Kafka集群的消息都有一个类别，这个类别被称为topic。（物理上不同topic的消息分开存储，逻辑上一个topic的消息虽然保存于一个或多个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于何处）

3）Partition

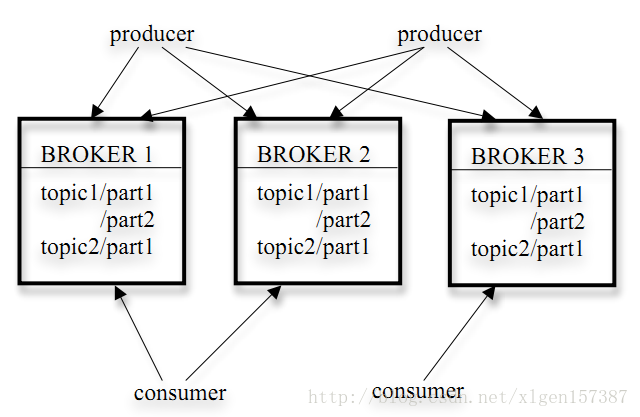
parition是物理上的概念，每个topic包含一个或多个partition，创建topic时可指定parition数量。每个partition对应于一个文件夹，该文件夹下存储该partition的数据和索引文件

4）Producer

负责发布消息到Kafka broker

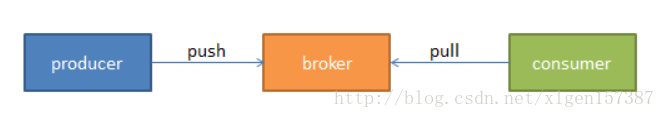
5）Consumer

消费消息。每个consumer属于一个特定的consumer group（可为每个consumer指定group name，若不指定group name则属于默认的group）。使用consumer high level API时，同一topic的一条消息只能被同一个consumer group内的一个consumer消费，但多个consumer group可同时消费这一消息。

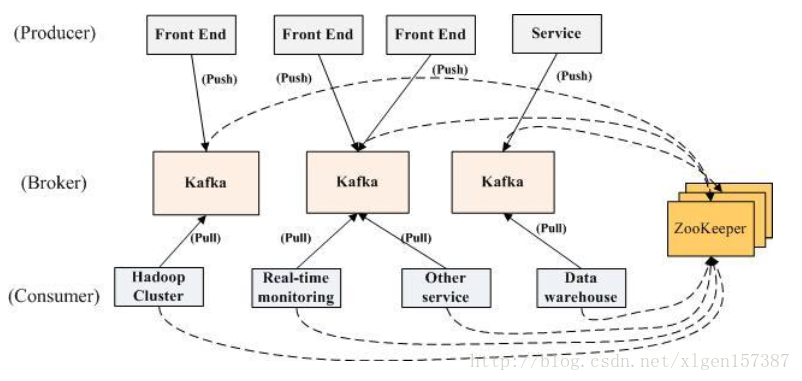


#### Kafka原理

1. 基本原理



生产者将数据生产出来，交给 broker 进行存储，消费者需要消费数据了，就从broker中去拿出数据来，然后完成一系列对数据的处理操作。



多个 broker 协同合作，producer 和 consumer 部署在各个业务逻辑中被频繁的调用，三者通过 zookeeper管理协调请求和转发。这样一个高性能的分布式消息发布订阅系统就完成了。

图上有个细节需要注意，producer 到 broker 的过程是 push，也就是有数据就推送到 broker，而 consumer 到 broker 的过程是 pull，是通过 consumer 主动去拉数据的，而不是 broker 把数据主动发送到 consumer 端的。

Push vs. Pull

push模式很难适应消费速率不同的消费者，因为消息发送速率是由broker决定的。push模式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。

1. Topic & Partition

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。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，该文件夹下存储这个partition的所有消息和索引文件。

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）。在创建topic时可以在$KAFKA\_HOME/config/server.properties中指定这个partition的数量，当然也可以在topic创建之后去修改parition数量。

在发送一条消息时，可以指定这条消息的key，producer根据这个key和partition机制来判断将这条消息发送到哪个parition。则key相同的消息会被发送并存储到同一个partition里。

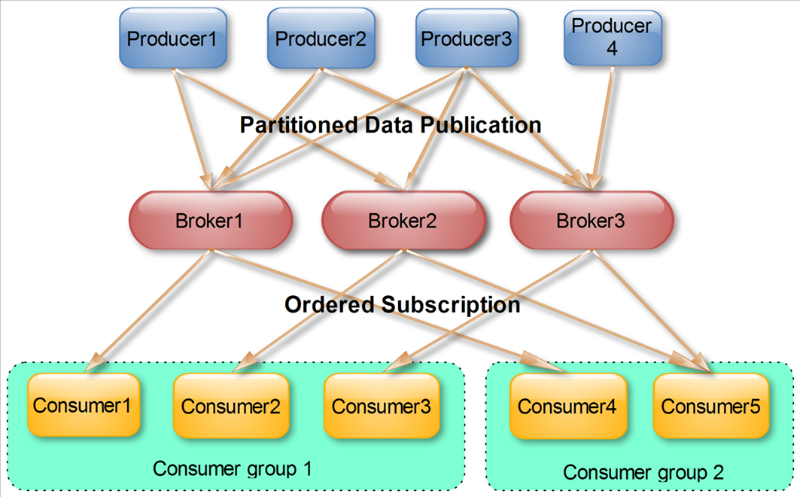
1. offset（偏移量）

Kafka只维护在Partition中的offset值，因为这个offsite标识着这个partition的message消费到哪条了。Consumer每消费一个消息，offset就会加1。其实消息的状态完全是由Consumer控制的，Consumer可以跟踪和重设这个offset值，这样的话Consumer就可以读取任意位置的消息。

如果一个topic的名称为"my\_topic"，它有2个partitions,那么日志将会保存在my\_topic\_0和my\_topic\_1两个目录中；日志文件中保存了一序列"log entries"(日志条目)，每个log entry格式为"4个字节的数字N表示消息的长度" + "N个字节的消息内容"；每个日志都有一个offset来唯一的标记一条消息，offset的值为8个字节的数字，表示此消息在此partition中所处的起始位置。每个partition在物理存储层面，有多个log file组成(称为segment)。segment file的命名为"最小offset"。kafka.例如"00000000000.kafka"；其中"最小offset"表示此segment中起始消息的offset。

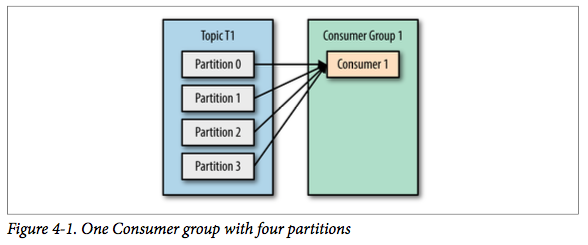
1. Consumer group & consumer

每一个consumer实例都属于一个consumer group，每一条消息只会被同一个consumer group里的一个consumer实例消费。（不同consumer group可以同时消费同一条消息）

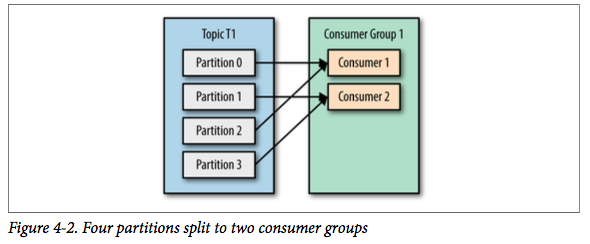


Kafka集群会保存所有的消息，不管消息有没有被消费；我们可以设定消息的过期时间，只有过期的数据才会被自动清除以释放磁盘空间，一个partition只能被一个消费者消费（一个消费者可以同时消费多个partition）。

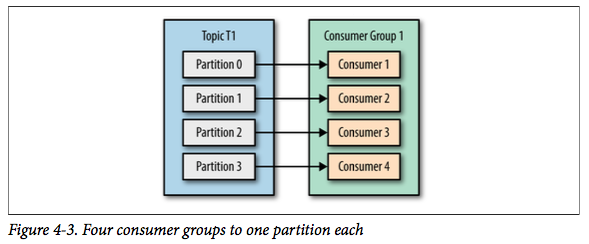
Kafka消费者是消费组的一部分，当多个消费者形成一个消费组来消费主题时，每个消费者会收到不同分区的消息。假设有一个T1主题，该主题有4个分区；同时我们有一个消费组G1，这个消费组只有一个消费者C1。那么消费者C1将会收到这4个分区的消息，如下所示：



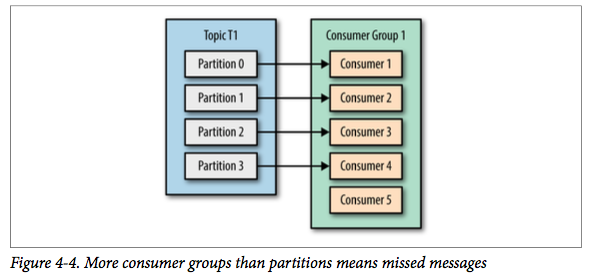
如果我们增加新的消费者C2到消费组G1，那么每个消费者将会分别收到两个分区的消息，如下所示：



如果增加到4个消费者，那么每个消费者将会分别收到一个分区的消息，如下所示：



但如果我们继续增加消费者到这个消费组，剩余的消费者将会空闲，不会收到任何消息：



1. 消费组与分区重平衡

当新的消费者加入消费组，它会消费一个或多个分区，而这些分区之前是由其他消费者负责的；另外，当消费者离开消费组（比如重启、宕机等）时，它所消费的分区会分配给其他分区。这种现象称为重平衡（rebalance）。重平衡是Kafka一个很重要的性质，这个性质保证了高可用和水平扩展。不过也需要注意到，在重平衡期间，所有消费者都不能消费消息，因此会造成整个消费组短暂的不可用。而且，将分区进行重平衡也会导致原来的消费者状态过期，从而导致消费者需要重新更新状态，这段期间也会降低消费性能。后面我们会讨论如何安全的进行重平衡以及如何尽可能避免。

消费者通过定期发送心跳（hearbeat）到一个作为组协调者（group coordinator）的broker来保持在消费组内存活。这个broker不是固定的，每个消费组都可能不同。当消费者拉取消息或者提交时，便会发送心跳。

如果消费者超过一定时间没有发送心跳，那么它的会话（session）就会过期，组协调者会认为该消费者已经宕机，然后触发重平衡。可以看到，从消费者宕机到会话过期是有一定时间的，这段时间内该消费者的分区都不能进行消息消费；通常情况下，我们可以进行优雅关闭，这样消费者会发送离开的消息到组协调者，这样组协调者可以立即进行重平衡而不需要等待会话过期。

在0.10.1版本，Kafka对心跳机制进行了修改，将发送心跳与拉取消息进行分离，这样使得发送心跳的频率不受拉取的频率影响。另外更高版本的Kafka支持配置一个消费者多长时间不拉取消息但仍然保持存活，这个配置可以避免活锁（livelock）。活锁，是指应用没有故障但是由于某些原因不能进一步消费。

#### Kafka集群管理

1. Leader选举

Kafka的核心是日志文件，日志文件在集群中的同步是分布式数据系统最基础的要素。

如果leaders永远不会down的话我们就不需要followers了！一旦leader down掉了，需要在followers中选择一个新的leader.但是followers本身有可能延时太久或者crash，所以必须选择高质量的follower作为leader.必须保证，一旦一个消息被提交了，但是leader down掉了，新选出的leader必须可以提供这条消息。大部分的分布式系统采用了多数投票法则选择新的leader,对于多数投票法则，就是根据所有副本节点的状况动态的选择最适合的作为leader.Kafka并不是使用这种方法。

Kafka动态维护了一个同步状态的副本的集合（a set of in-sync replicas），简称ISR，在这个集合中的节点都是和leader保持高度一致的，任何一条消息必须被这个集合中的每个节点读取并追加到日志中了，才回通知外部这个消息已经被提交了。因此这个集合中的任何一个节点随时都可以被选为leader.ISR在ZooKeeper中维护。ISR中有f+1个节点，就可以允许在f个节点down掉的情况下不会丢失消息并正常提供服。ISR的成员是动态的，如果一个节点被淘汰了，当它重新达到“同步中”的状态时，他可以重新加入ISR.这种leader的选择方式是非常快速的，适合kafka的应用场景。

1. 同步管理

对于某个分区来说，保存正分区的"broker"为该分区的"leader"，保存备份分区的"broker"为该分区的"follower"。备份分区会完全复制正分区的消息，包括消息的编号等附加属性值。为了保持正分区和备份分区的内容一致，Kafka采取的方案是在保存备份分区的"broker"上开启一个消费者进程进行消费，从而使得正分区的内容与备份分区的内容保持一致。一般情况下，一个分区有一个“正分区”和零到多个“备份分区”。可以配置“正分区+备份分区”的总数量，关于这个配置，不同主题可以有不同的配置值。注意，生产者，消费者只与保存正分区的"leader"进行通信。

#### Kafka的特性

1. 高吞吐量、低延迟：kafka每秒可以处理几十万条消息，它的延迟最低只有几毫秒，每个topic可以分多个partition, consumer group 对partition进行consume操作；
2. 可扩展性：kafka集群支持热扩展；
3. 持久性、可靠性：消息被持久化到本地磁盘，并且支持数据备份防止数据丢失；
4. 容错性：允许集群中节点失败（若副本数量为n,则允许n-1个节点失败）；
5. 高并发：支持数千个客户端同时读写；
6. 支持实时在线处理和离线处理：可以使用Storm这种实时流处理系统对消息进行实时进行处理，同时还可以使用Hadoop这种批处理系统进行离线处理；
7. 顺序性：kafka可以保证在同一个partition中消息的消费是有序的；

#### Kafka使用场景

1. 日志收集：一个公司可以用Kafka可以收集各种服务的log，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。

2. 消息系统：解耦和生产者和消费者、缓存消息等。

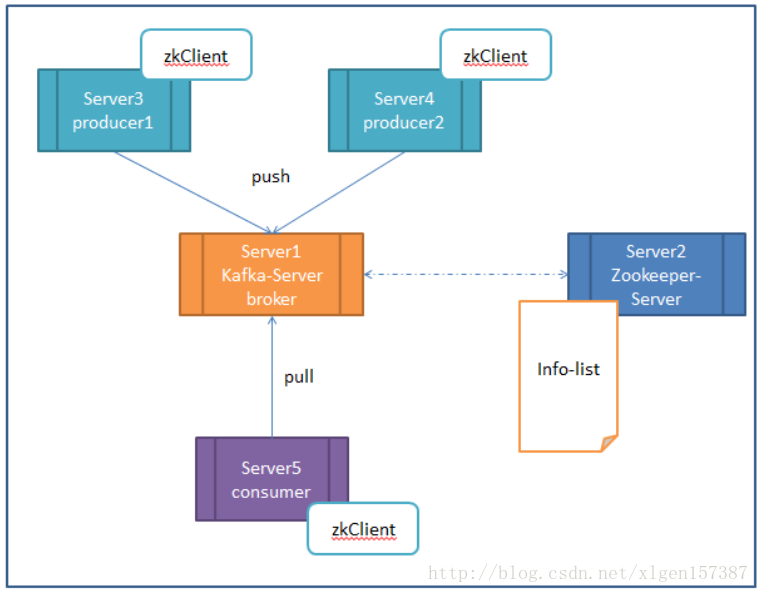
3. 用户活动跟踪：Kafka经常被用来记录web用户或者app用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到kafka的topic中，然后订阅者通过订阅这些topic来做实时的监控分析，或者装载到hadoop、数据仓库中做离线分析和挖掘。

4. 运营指标：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。

5. 流式处理：比如spark streaming和storm

6. 事件源

#### Kafka与zookeeper的工作流程



（1） 启动zookeeper 的 server

（2） 启动kafka 的 server

（3） Producer 如果生产了数据，会先通过 zookeeper 找到 broker，然后将数据存放到 broker

（4） Consumer 如果要消费数据，会先通过 zookeeper 找对应的 broker，然后消费。