Kafka学习笔记

# 概述

Kafka是一种快速，可扩展，容错的消息传递系统

Apache™Kafka是一种快速，可扩展，耐用且容错的发布 - 订阅消息传递系统。由于它具有更高的吞吐量，可靠性和复制性，Kafka通常用于代替传统的消息代理，如JMS和AMQP。

主要设计目标：

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

在Kafka中，客户端和服务器之间的通信是通过简单，高性能，语言无关的[TCP协议完成的](https://kafka.apache.org/protocol.html)。

## 基本概念

1. Kafka作为一个集群运行在一个或多个可跨多个数据中心的服务器上。
2. Kafka集群以称为主题的类别存储记录流。
3. 每条记录由一个键，一个值和一个时间戳组成。
4. **Broker：**Kafka集群包含一个或多个服务器，这种服务器被称为broker。
5. **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）。
6. **Partition：**partition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partition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。
7. **Producer：**负责发布消息到Kafka broker。
8. **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。

## 重要特性

| **特性** | **描述** |
| --- | --- |
| 可扩展性 | 分布式系统可轻松扩展，无需停机 |
| 持久性 | 在磁盘上保留消息，并提供群集内复制 |
| 可靠性 | 复制数据，支持多个订阅者，并在发生故障时自动平衡消费者 |
| 性能 | 发布和订阅的高吞吐量，磁盘结构即使有大量TB的存储消息也能提供持续的性能 |

# Kafka的架构设计

## Kafka的组件

1. Topic
2. Producer
3. Consumer
4. Brokers

在高级别Kafka提供以下保证：

1. 生产者发送到特定主题分区的消息将按其发送顺序附加。也就是说，如果记录M1由与记录M2相同的生成者发送，并且首先发送M1，则M1将具有比M2更低的偏移并且在日志中更早出现。
2. 消费者实例按照它们存储在日志中的顺序查看记录。
3. 对于具有复制因子N的主题，我们将容忍最多N-1个服务器故障，而不会丢失任何提交到日志的记录。

## Topic

Kafka集群以称为主题的类别存储记录流，每条记录由一个键，一个值和一个时间戳组成。

主题是发布记录的类别或订阅源名称。

对于每个主题，Kafka群集都维护一个分区日志，如下所示：



每个分区都是一个有序的，不可变的记录序列，不断附加到结构化的提交日志中。分区中的记录每个都被分配一个称为*偏移*的顺序ID号，它唯一地标识分区中的每个记录。

Kafka集群持久地保留所有已发布的记录 - 无论它们是否已被消耗 - 使用可配置的保留期。

Kafka的性能在数据大小方面实际上是恒定的，因此长时间存储数据不是问题。

消费组



维护组中成员资格的过程由Kafka协议动态处理。如果新实例加入该组，他们将从该组的其他成员接管一些分区; 如果实例死亡，其分区将分发给其余实例。

### 分区

日志的分区分布在Kafka集群中的服务器上，每个服务器处理数据并请求分区的共享。每个分区都在可配置数量的服务器上进行复制，以实现容错。

分区的设计：

领导者、“追随者”；

### Producer

### Consumer



### Brokers

# Kafka MirrorMaker

# 消息传统

消息传统上有两种模型：[排队](http://en.wikipedia.org/wiki/Message_queue)和[发布 - 订阅](http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern)。

## **Push vs. Pull**

作为一个messaging system，Kafka遵循了传统的方式，选择由producer向broker push消息并由consumer从broker pull消息。一些logging-centric system，比如Facebook的Scribe和Cloudera的Flume,采用非常不同的push模式。事实上，push模式和pull模式各有优劣。

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。

# Kafka的安全

安全包含的内容：

1. Kafka安全性包含多种需求 - 需要加密流经Kafka的数据，防止恶意代理向Kafka发布数据，以及管理对个人或组级别特定主题的访问的能力。
2. 因此，Kafka的最新更新通过SSL支持线加密，基于Kerberos的身份验证以及通过Apache Ranger或其他可插拔授权系统的粒度授权选项。

# Metrics

# 配额-多租户

# Kafka 技术点

1. 机架感知提高了弹性和可用性，使得副本被隔离，因此可以保证它们跨越多个机架或可用区域。
2. 自动副本leader选举，通过检测不均匀分布，自动，均匀地分配集群功能中的领导者，一些经纪人提供与其他人相比更多的数据并进行调整。
3. 消息时间戳，因此Kafka中的每条消息现在都有一个时间戳字段，指示消息的生成时间。
4. SASL改进包括外部认证服务器以及在一台服务器上支持多种类型的SASL认证

## **消息投递语义**

kafka支持3种消息投递语义：

1. At most once：最多一次，消息可能会丢失，但不会重复
2. At least once：最少一次，消息不会丢失，可能会重复
3. Exactly once：只且一次，消息不丢失不重复，只且消费一次（0.11中实现，仅限于下游也是kafka）

### At least once（业务中使用比较多）

先获取数据，再进行业务处理，业务处理成功后commit offset。

1. 生产者生产消息异常，消息是否成功写入不确定，重做，可能写入重复的消息。
2. 消费者处理消息，业务处理成功后，更新offset失败，消费者重启的话，会重复消费。

### **At most once**

先获取数据，再commit offset，最后进行业务处理。

1. 生产者生产消息异常，不管，生产下一个消息，消息就丢了。
2. 消费者处理消息，先更新offset，再做业务处理，做业务处理失败，消费者重启，消息就丢了。

### Exactly once

首先要保证消息不丢，再去保证不重复。所以盯着At least once的原因来搞。

1. 生产者重做导致重复写入消息----生产保证幂等性。
2. 消费者重复消费---消灭重复消费，或者业务接口保证幂等性重复消费也没问题。

业务处理的幂等性非常重要。Kafka控制不了，需要业务来实现。比如所判断消息是否已经处理。

解决重复消费有两个方法：

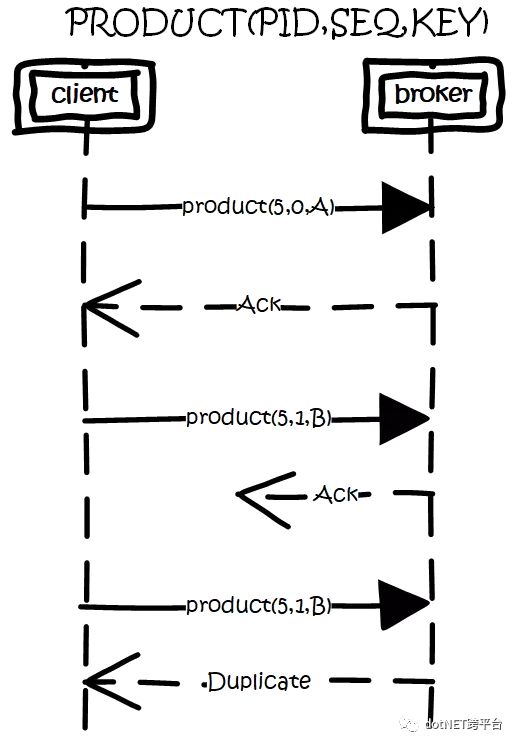
1. 下游系统保证幂等性，重复消费也不会导致多条记录。
2. 把commit offset和业务处理绑定成一个事务。

### Kafka幂等producer

幂等producer指producer.send的逻辑是幂等的，即发送相同的Kafka消息，broker端不会重复写入消息。

为每个producer分配一个pid，作为该producer的唯一标识。producer会为每一个<topic,partition>维护一个单调递增的seq。类似的，broker也会为每个<pid,topic,partition>记录最新的seq。当req\_seq == broker\_seq+1时，broker才会接受该消息。因为：

1. 消息的seq比broker的seq大超过时，说明中间有数据还没写入，即乱序了。
2. 消息的seq不比broker的seq小，那么说明该消息已被保存。



#### exactly once semantics，EOS

Kafka 0.11.0.0版本正式支持精确一次处理语义(exactly once semantics，下称EOS)。

Kafka的EOS主要体现在3个方面：

1. 幂等producer：保证发送单个分区的消息只会发送一次，不会出现重复消息
2. 事务(transaction)：保证原子性地写入到多个分区，即写入到多个分区的消息要么全部成功，要么全部回滚
3. 流处理EOS：流处理本质上可看成是“读取-处理-写入”的管道。此EOS保证整个过程的操作是原子性。即流处理EOS保证的是端到端(E2E)消息处理的EOS。

注意，流处理EOS只适用于Kafka Streams

启用方法：

1. 启用幂等producer：在producer程序中设置属性enable.idempotence=true，但不要设置transactional.id。注意是不要设置，而不是设置成空字符串或"null"
2. 启用事务支持：在producer程序中设置属性transcational.id为一个指定字符串(你可以认为这是你的事务名称，故最好起个有意义的名字)，同时设置enable.idempotence=true
3. 启用流处理EOS：在Kafka Streams程序中设置processing.guarantee=exactly\_once

幂等producer提供的语义保证是有条件的：

1. 单分区幂等性：幂等producer无法实现多分区上的幂等性。若要实现多分区上的原子性，需要引入事务。
2. 单会话幂等性：幂等producer无法跨会话实现幂等性。即使同一个producer宕机并重启也无法保证消息的EOS语义

#### 幂等producer的设计与实现

#### 参考资料

1. [**关于Kafka幂等producer的讨论**](http://www.cnblogs.com/huxi2b/p/7717775.html)**：**<http://www.cnblogs.com/huxi2b/p/7717775.html>
2. Kafka基本知识整理：<https://mp.weixin.qq.com/s?src=11&timestamp=1569170566&ver=1868&signature=8d3P0jiiVbW-5OBveQxccmBopY5WHFXZIu1FK4JE6GHBS939WM4rw7mMjNp89Db05ucK2OsNlcG2TAhYFnm12sX1zrOgBce3cWvhj8kvmL7DbSlLgFAvNOBaXao7xBJG&new=1>

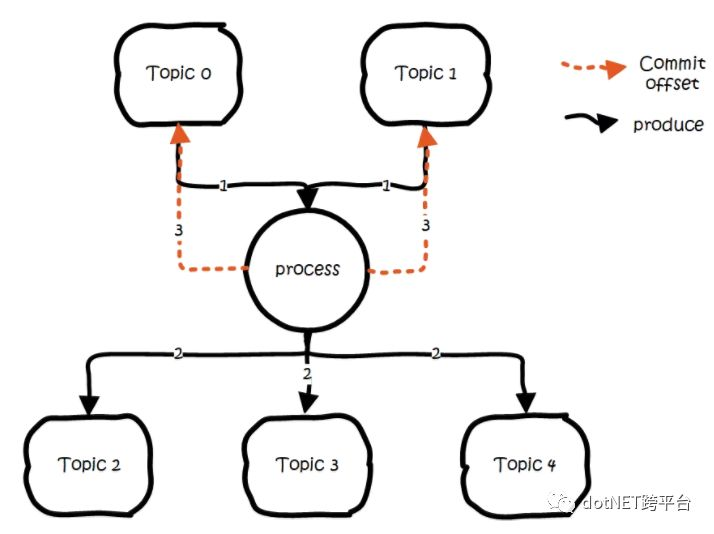
### **事务性和原子性**

事务(transaction)：保证原子性地写入到多个分区，即写入到多个分区的消息要么全部成功，要么全部回滚。

场景是这样的：

1. 先从多个源topic中获取数据。
2. 做业务处理，写到下游的多个目的topic。
3. 更新多个源topic的offset。

其中第2、3点作为一个事务，要么全成功，要么全失败。这里得益于offset实际上是用特殊的topic去保存（在0.10版本后，Kafka把这个offset的保存，从Zookeeper中剥离，保存在一个名叫**\_\_consumeroffsets topic**的Topic中），这两点都归一为写多个topic的事务性处理。



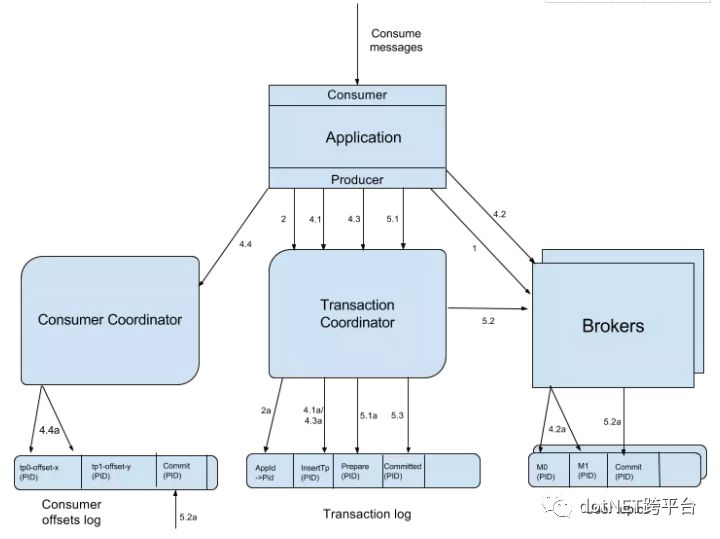
引入tid（transaction id），和pid不同，这个id是应用程序提供的，用于标识事务，和producer是谁并没关系。就是任何producer都可以使用这个tid去做事务，这样进行到一半就死掉的事务，可以由另一个producer去恢复。

同时为了记录事务的状态，类似对offset的处理，引入transaction coordinator用于记录transaction log。在集群中会有多个transaction coordinator，每个tid对应唯一一个transaction coordinator。

注：transaction log删除策略是compact，已完成的事务会标记成null，compact后不保留。

启动事务时，先标记开启事务，写入数据，全部成功就在transaction log中记录为prepare commit状态，否则写入prepare abort的状态。之后再去给每个相关的partition写入一条marker（commit或者abort）消息，标记这个事务的message可以被读取或已经废弃。成功后在transaction log记录下commit/abort状态，至此事务结束。

整体的数据流是这样的：



1. 首先使用tid请求任意一个broker（代码中写的是负载最小的broker），找到对应的transaction coordinator。
2. 请求transaction coordinator获取到对应的pid，和pid对应的epoch，这个epoch用于防止僵死进程复活导致消息错乱，当消息的epoch比当前维护的epoch小时，拒绝掉。tid和pid有一一对应的关系，这样对于同一个tid会返回相同的pid。
3. client先请求transaction coordinator记录<topic,partition>的事务状态，初始状态是BEGIN，如果是该事务中第一个到达的<topic,partition>，同时会对事务进行计时；client输出数据到相关的partition中；client再请求transaction coordinator记录offset的<topic,partition>事务状态；client发送offset commit到对应offset partition。
4. client发送commit请求，transaction coordinator记录prepare commit/abort，然后发送marker给相关的partition。全部成功后，记录commit/abort的状态，最后这个记录不需要等待其他replica的ack，因为prepare不丢就能保证最终的正确性了。

     这里prepare的状态主要是用于事务恢复，例如给相关的partition发送控制消息，没发完就宕机了，备机起来后，producer发送请求获取pid时，会把未完成的事务接着完成。

     当partition中写入commit的marker后，相关的消息就可被读取。所以kafka事务在prepare commit到commit这个时间段内，消息是逐渐可见的，而不是同一时刻可见。

### 消息消费事务

消费时，partition中会存在一些消息处于未commit状态，即业务方应该看不到的消息，需要过滤这些消息不让业务看到，kafka选择在消费者进程中进行过滤，而不是在broker中过滤，主要考虑的还是性能。kafka高性能的一个关键点是zero copy，如果需要在broker中过滤，那么势必需要读取消息内容到内存，就会失去zero copy的特性。

## Kafka日志留存策略

### 参考资料

1. 《关于Kafka日志留存策略的讨论.docx》

## [**Kafka Java consumer动态修改topic订阅**](http://www.cnblogs.com/huxi2b/p/7040617.html)

### 参考资料

1. Kafka Java consumer动态修改topic订阅：<http://www.cnblogs.com/huxi2b/p/7040617.html>

# Kafka管理、监控

## 管理、监控的内容

1. Topic管理：创建/删除主题
2. 安全管理：管理用户权限，更容易、更快速地分发安全令牌
3. 支持通过Kafka REST服务器/ API发布/消费数据

## 管理与监控的框架、贡献者

框架：

贡献者：

1. Ambari可视化Kafka运营指标的视图
2. Hortonworks的Kafka管理面板

# 常见的应用场景

1. 流处理
2. 网站活动跟踪
3. 度量标准收集和监控
4. 日志聚合

# 参考资料

<http://www.jasongj.com/2015/01/02/Kafka深度解析/>